I protocolli di rete vengono identificati secondo la notazione RFC (request for comments), gli RFC sono documenti testuali che raccolgono tutte le specifiche di un certo protocollo. (possiamo considerarli quindi come una sorta di “libretto delle istruzioni” per i protocolli internet)

Livelli della rete (approccio pratico in 5 livelli):

1. Fisico (gestione della codifica sul canale)
2. Data Link (Ethernet IEEE 802.3, viene implementata la comunicazione a livello logico e fisico tra mittente e destinatario)
3. Rete (IPV4/V6, instradamento: sulla base dell’IP address del destinatario si calcola un percorso che poi verrà usato per il trasporto al passo successivo)
4. Trasporto (UDP, TCP, connessione tra applicazioni, numero di porta associato ad un processo in fase di esecuzione, tra tutte le applicazioni attive si può scegliere quella che ci interessa)
5. Applicativo (socket, http, gestione delle applicazioni)

Invio di un messaggio nella rete:

* Sender scrive messaggio nella RAM, con la DMA lo trasferisce nella NIC che lo invia in rete
* Receiver riceve il messaggio attraverso la NIC, lo memorizza, da lì la DMA sposta il mess. in RAM e l’OS legge la RAM, così il messaggio viene letto.

NIC: Network Interface Card.

Il passaggio da un host all’altro del messaggio avviene tramite la tecnica dello store&forward (il messaggio deve essere totalmente ricevuto, prima di poter essere ritrasmesso nel collegamento in uscita, questo implica che, se tra due host c’è un ritardo, esso verrà moltiplicato per tutti gli host che il messaggio deve attraversare prima di arrivare a destinazione, quindi ciò risulterà in una latenza non indifferente). Nel livello 3 “rete” viene realizzato l’instradamento dei messaggi multi-hop.

Protocollo IP: attraverso di esso si definisce la comunicazione tra due host mediante i loro indirizzi IP.

Praticamente si crea una “rete di reti” mettendo in comunicazione delle sottoreti locali.

Ad ogni PC nella Rete è assegnato un indirizzo IP che ne permette l’identificazione.

* IP v4 riserva **32 bit** per l’indirizzamento (che permettono 4 mld di indirizzi diversi)
* IP v6 riserva **128** bit per l’indirizzamento (che permettono migliaia di miliardi di indirizzi diversi)

Su una macchina di solito possono “girare” più applicazioni contemporaneamente: nel livello di trasporto si stabilisce il modo per indirizzare la singola applicazione all’interno di una macchina, è chiaro che l’indirizzo IP non basterà; servirà un ulteriore indirizzo (la porta di comunicazione): esso è composto di 16 bit e vi si collegano le applicazioni nel seguente modo:

attraverso meccanismi di multiplexing e demultiplexing dopo aver raccolto i messaggi, essi vengono mandati in rete nel primo caso, o indirizzati verso l’operazione corretta nel secondo.

Ogni applicazione ha la sua porta di comunicazione.

Per gestire la rete a livello applicativo sono necessarie delle interfacce/strutture software dette socket, a questo livello vengono utilizzate svariati protocolli che offrono diverse funzionalità, ad esempio l’ HTTP (web) o l’SMTP (mail). Attraverso i socket è possibile accedere a tutte le primitive (system call) di comunicazione disponibili sulla macchina.

Focus su **livello 4** e differenza UDP/TCP

UDP: Modalità Datagram.

Abbiamo un host mittente, un host destinatario dei dispositivi chiamati router attraverso i quali passeranno i messaggi (datagrammi) per arrivare a destinazione.

UDP: User Datagram Protocol. Immaginiamo i datagrammi come delle lettere da scrivere con una

* intestazione (“header”) che contiene l’id della porta del destinatario oltre ad altre info utili al trasferimento, seguita dal
* contenuto vero e proprio del messaggio (“payload”)

esempio comunicazione UDP:

* A: Host mittente
* B: Host destinatario

A e B si devono mettere d’accordo per eseguire correttamente le system call send() e recv(), la comunicazione si divide in 3 passi principali:

1. A esegue una send(), la quale termina quando è stato scritto il messaggio nei buffer di trasmissione. Nella send verrà specificato l’indirizzo dell’area di memoria che contiene il datagramma da inviare, attraverso due parametri: puntatore al primo byte del datagramma e indicazione della lunghezza in byte del datagramma (possiamo avere datagrammi di lunghezza compresa tra i 0 e i -1 byte
2. L’instradamento è stato affidato al livello 3, quindi UDP si occuperà solo dell’effettivo passaggio dei messaggi da A a B attraverso i router intermedi:

il datagramma passa dal buffer TX della NIC al buffer RX del primo router: il datagramma “salta” di router in router attraverso gli hop, passaggi delicati dove si rischia di perdere il messaggio. L’UDP è considerato poco affidabile proprio perché non viene garantito il recapito di tutti i messaggi

1. B esegue una recv(), che termina quando viene ricevuto il messaggio che aspettava.

Nella recv() viene specificata un’area di memoria (puntatore e dimensione) dove ricevere il messaggio da A e bisogna sapere anche quanto lungo è il messaggio da ricevere, affinché sia possibile destinare alla ricezione una zona di memoria di dimensione adeguata.

Breve focus su TX e RX buffer:

la capacità di ogni buffer è 2048 byte.

TX buffer: transmit buffer: transazioni di scrittura.

RX buffer: receive buffer: transazioni di lettura.

Quando uno dei due device in comunicazione, dopo aver mandato un messaggio, non riceve una risposta, esso deve rinviare il messaggio.

Buffer che misura il tempo trascorso dall’invio di un messaggio: Timeout TX buffer.

Il device che manda il messaggio lo salva sul Timeout TX buffer, e al raggiungimento di una certa soglia di timeout, se non è stata ricevuta alcuna risposta, lo manda di nuovo.

Come si calcola questa soglia di time-out?

Ogni volta che viene inviato un messaggio e ricevuta risposta (quindi passo della comunicazione va a buon fine) si calcola il tempo trascorso tra l’invio del messaggio e l’arrivo della risposta da parte dell’altro host, detto round-trip time.

Per calcolare la soglia di time-out si fa una media dei RTT precedenti e la soglia di time-out sarà leggermente superiore a questa media.

UDP non è la scelta giusta se non si possono tollerare perdite di messaggi o se si vogliono inviare messaggi più grandi di 2?16 byte. Occorrerà passare ad un altro protocollo:

TCP: Transmission Control Protocol.

Il TCP rappresenta una scelta preferibile all’UDP, risolvendo entrambi i maggiori “problemi” di quest’ultimo:

sia il problema della dimensione del messaggio che dell’affidabilità (garanzia di non “perdere il messaggio per strada”) vengono risolti attraverso l’implementazione di un canale virtuale bidirezionale di comunicazione:

le informazioni possono passare da entrambe le direzioni, una volta stabilita la connessione A può mandare a B una quantità arbitraria di byte e viceversa, poiché, se A manda dati a B, che li riceve nell’ordine in cui sono stati mandati, nel caso in cui A voglia mandare più byte di quelli ricevibili da B, B, una volta ricevuti tutti i dati che può ricevere, inizierà a svuotare il suo buffer (nel frattempo la parte “in eccesso” non ancora memorizzata rimane sul canale) e una volta svuotato il buffer potrà ricevere anche la parte rimasta sul canale. Se invece B cerca di acquisire un numero di byte maggiore di quello effettivamente mandato da A entrerà in gioco il meccanismo di blocco della recv che vedremo più in dettaglio più avanti con le flag.

Abbiamo spiegato come funziona la comunicazione una volta aperta la connessione tra A e B, ma ciò deve essere preceduto da una fase preliminare di apertura della connessione:

*3-way handshake*

Uno tra A e B si comporterà da Client (entità attiva), l’altro da Server (entità “passiva”).

Il client è colui che prende l’iniziativa di stabilire la connessione:

* manda un messaggio al server in cui richiede di aprire la connessione
* il server, arrivata la richiesta di connessione, risponde confermando la sua volontà di aprire la connessione
* una volta ricevuta la risposta da parte del server, il client manda un ulteriore messaggio comunicando che la connessione è stata aperta e pronta ad essere utilizzata.

Questo meccanismo di apertura della connessione funziona soprattutto grazie all’utilizzo dei header del protocollo TCP.

Anche nel protocollo TCP le unità di dati che vengono trasmesse sono divise in header e payload.

Lo header standard (può variare a seconda delle implementazioni) nel TCP è lungo circa 20 byte e contiene vari campi:

* numero di porta sorgente (codificato su 16 bit)
* numero di destinazione (codificato su 16 bit)
* numero di sequenza (codificato su 32 bit)
* numero di acknowledgement (codificato su 32 bit)

nel sesto byte dell’header 6 bit sono riservati alle cosiddette flag, che servono per gestire varie situazioni che si presentano durante la comunicazione

1. SYN: synchronization
2. ACK: acknowledgement
3. URG: urgent
4. PSH: push
5. RST: reset
6. FIN: finish

I numeri di sequenza ed acknowledgement servono per garantire sicurezza ed affidabilità alla trasmissione di informazioni. C’è la certezza che la quantità arbitraria di byte che si vuole mandare verrà interamente ricevuta dal destinatario, per questo la modalità stream è preferibile e considerata più affidabile della UDP.

3-WAY HANDSHAKE NEL DETTAGLIO:

Consideriamo due host H1 (client: C) ed H2 (server: S)

1. C->S

H1 manda un messaggio ad H2 attivando il flag syn (si richiede quindi l’apertura della connessione)

Nel Sequence number si inserisce un numero a caso: x.

Numero di acknowledge rimane vuoto, vengono riempiti i campi del numero di porta.

1. S->C

Il server ha due opzioni:

* se non vuole aprire la connessione attiva il flag RST e la comunicazione si interrompe. L’hand-shake non va a buon fine.
* Se vuole aprire la connessione risponde attivando:
  + - * Il flag syn (che significa che anche H2 vuole aprire la connessione)
      * Il flag ack (comunica che il messaggio del client è stato ricevuto con successo)

Nel sequence number si inserisce un numero a caso: y

Nel numero di acknowledge si inserisce x+1 (il numero di sequenza di H1 incrementato di 1, che è il numero di byte ricevuti).

Numero di sequenza e di acknowledge allora hanno un ruolo di contatori che ci dicono quanti byte sono stati rispettivamente inviati e ricevuti.

1. C->S

H1 riceve il messaggio da H2 e attiva il flag ack (così facendo manda un feedback positivo a H2), il flag sync viene spento perché non più utile.

Si inserisce x+1 nel sequence number e y+1 nell’acknowledge.

Numeri di sequenze e acknowledge sono riservati ai due host, ciò contribuisce a mantenere la connessione privata e sicura.

Fino ad adesso sono stati scambiati dunque due datagrammi/unità di dati che sono serviti ad aprire effettivamente la comunicazione.

Dal terzo “datagramma” in poi i due host possono iniziare a scambiarsi messaggi:

ad ogni byte inviato corrisponderà l’incremento di uno del numero di acknowledge del ricevente

nel passaggio 3 si può già mandare un messaggio, ad esempio, oltre a fare ciò che abbiamo già spiegato nel passo 3, H1 inserisce nel payload il messaggio “ciao” che, se codificato con 8 bit a carattere, corrisponde a 5 byte (8 bit=1 byte, ogni byte serve per codificare un carattere, in più c’è anche il null character terminatore di stringa), quindi vengono mandati 5 byte a H2.

1. S->C

H2 riceve il messaggio e attiva il flag ack per comunicare che ha ricevuto con successo il messaggio.

Inoltre inserisce:

* + Nel numero di sequenza y+1
  + Nel numero di acknowledge x+6, ovvero x+5+1.

Piggybacking

È una tecnica di ottimizzazione effettuata dal S.O. dell’host destinatario.

Abbiamo spiegato che, dopo la ricezione di un messaggio, nel TCP, l’host ricevente il messaggio attiva l’ACK per “comunicare” l’avvenuta ricezione di tale messaggio, tuttavia se il protocollo prevede successivi scambi di messaggi fra i due host, il destinatario può rimandare l’invio dell’ACK e includerlo nel successivo messaggio inviato al mittente. Il piggybacking potrebbe varie volte risultare in un rallentamento della comunicazione ed è conveniente disattivarlo se si intende inviare solo messaggi brevi.

In generale è utile per reti LAN (Local Area Network), ma poco efficiente per la rete globale (round trip time troppo elevato).

Banda e Latenza

Banda: dimensione delle informazioni/tempo di trasferimento, quindi

Latenza: tempo impiegato per inviare 1 byte di informazione da un device all’altro.

Pro e contro della modalità stream (TCP):

pro:

* + Siccome ho un “tubo bidirezionale”, dopo il 3-WH non devo più inserire la porta di destinazione
  + È una modalità sicura (non c’è rischio di perdere informazioni)

contro:

* + Prima di iniziare la vera e propria comunicazione bisogna aprire la connessione col 3-WH, questo richiede un bel po’ di tempo, tempo che non dipende dalla qualità della connessione ma dalla distanza tra i due device
  + Bisogna assegnare i ruoli di client e server ai dispositivi
  + Bisogna chiudere la connessione

Chiusura della connessione:

vi sono 3 possibili modi per chiudere il canale di comunicazione:

* Con il flag FIN: se un dispositivo vuole chiudere la connessione attiva questo flag, l’altro risponderà con un FIN-ACK, una volta che uno dei device ha chiuso la connessione, questa diventa unidirezionale: il dispositivo che ha chiuso non può più inviare messaggi ma può riceverli
* Con la flag RST: utilizzandola si va a chiudere “brutalmente” la comunicazione, chiudendo il canale in ambo le direzioni. (è la stessa flag che si usa quando, richiesta l’apertura di una connessione da H1, H2 non vuole aprirla nel 3-WH)
* Il canale di comunicazione si chiude automaticamente (non confermata sta cosa, è un po’ sus)

Pro e contro della modalità di tipo datagram (UDP):

* + È più veloce rispetto a TCP (non utilizza meccanismi di gestione degli errori, né di controllo del flusso)

Contro:

* + Proprio per il motivo sopra spiegato è però meno affidabile e meno sicuro, il che lo rende anche meno versatile, riducendo i suoi campi di applicazione a situazioni in cui non ci si preoccupa dell’ordine dei pacchetti di informazione né di garantire che un pacchetto effettivamente arrivi a destinazione (es. trasmissione video in streaming o audio in tempo reale: la perdita di alcuni pacchetti di dati in questa situazione è tollerabile e non compromette l’esperienza dell’utente)
  + Devo sempre inserire IP e numero di porta degli host.

Table

Description automatically generated

Controlli nel TCP:

* **Controllo di flusso**: serve per ridurre la perdita dei messaggi, agendo in base al riempimento del RX buffer dell’ host ricevente, con questo controllo si evita che il messaggio che viene inviato sia troppo grande rispetto al buffer di ricezione del destinatario: si rallenta l’invio del messaggio per lasciare più tempo al destinatario di svuotare il buffer. Ciò è reso possibile come abbiamo già visto dall’ header TCP: una volta aperta la connessione 3-WH nella finestra di ricezione dell’header viene scritta la dimensione del buffer di ricezione, quando il server manda la risposta mette nella receive window lo spazio disponibile nel buffer, quindi a ogni scambio di messaggi si può controllare che ci sia abbastanza spazio nel buffer del ricevente per ricevere effettivamente il messaggio, se così non fosse, l’ host mittente può mandare solo la parte del messaggio che può essere salvata e che “ci sta” nel buffer del destinatario, la restante parte verrà mandata quando l’host mittente riceve la comunicazione che si è liberato spazio nel buffer.

In casi in cui la comunicazione è unidirezionale, quindi uno dei due host, ad esempio B, ha già “chiuso” la connessione, e può dunque solo ricevere ma non più inviare messaggi e ad esempio A voglia mandare un messaggio e riesce a mandare una quantità di dati tale da riempire tutto il buffer di B, B non può più rispondere ad A una volta che ha svuotato il buffer per comunicarglielo in modo che A possa mandare la restante parte di messaggio. Questa situazione è definita **deadlock.**

Per risolvere questo problema e non far “aspettare A all’infinito” è stata introdotta una soglia di time-out oltre la quale A prova a mandare il resto del messaggio.

* **Controllo di congestione**: si concentra sulla gestione del traffico in una rete affollata. Avendo una connessione multi-hop il datagramma potrebbe passare attraverso diversi router prima di arrivare a destinazione; e se tali router attraverso cui potrebbe passare avessero il buffer pieno? Si introduce il concetto di congestion window, un valore nell’header che indica lo spazio libero minimo del buffer di ricezione dei router della rete (quelli che potenzialmente il datagramma attraverserà per arrivare a destinazione). Ci sono vari algoritmi per stimare questo valore; in quello usato da TCP si parte inviando messaggi di piccole dimensioni e andando avanti si aumenta la dimensione, se ricevo un ack vuol dire che la dimensione del messaggio è minore dello spazio disponibile sul buffer di ricezione del router con meno spazio disponibile. Si procede aumentando esponenzialmente la dimensione del messaggio, fino a quando non si ha mandato tutto il datagramma oppure non si ha ricevuto l’ack da parte dell’ host destinatario. Una volta che sappiamo il peso massimo sostenibile sapremo anche la congestion window.

Per determinare la dimensione massima del messaggio si prende il valore minimo tra la finestra di congestione e quella di ricezione.

MTU: dimensione massima dei datagrammi. (maximum transfer unit). Essa varia in base al protocollo.

In caso volessimo mandare 15000 byte di dati usando un protocollo in cui la MTU è 1500 non possiamo mandare tutto il datagramma assieme, quindi verrà diviso in 10 parti (frammentazione), che verranno mandate singolarmente. L’ host ricevente manderà un solo ack, al completamento della ricezione di tutto il datagramma. (quindi se anche una sola delle 10 parti non viene recapitata non ci sarà l’ ack)

**Livello 5: Socket**

Il socket è una sorta di porta di comunicazione che permette il passaggio di informazioni da una applicazione alla rete e viceversa.

L’informazione che arriva dalla rete passa attraverso il socket dell’applicazione di destinazione e giunge alla NIC, quindi si riescono a far passare informazioni dal livello applicativo a quello del sistema operativo e viceversa.

2 tipi di socket:

* Stream socket: connection-oriented, basati su protocolli come il TCP. Possono essere visti come una coppia di file per supportare la comunicazione bidirezionale: un file in lettura, uno in scrittura.

L’ host A può effettuare una serie di operazioni in scrittura sul socket; le informazioni viaggeranno sullo stream tramite TCP/IP e arriveranno all’ host B, che potrà leggerle attraverso operazioni di lettura sul socket.

Essendo basati su TCP garantiscono una comunicazione sicura, affidabile, full-duplex (cioè bidirezionale in modo che sia host A che host B possano sia inviare che ricevere dati allo stesso tempo), con un flusso di byte di lunghezza variabile

* Datagram socket: connectionless, basati su UDP.

Organizzazione dell’interfaccia di programmazione

Stream socket

si realizza attraverso varie chiamate di sistema.

Se il client chiama la system-call socket() viene creato un socket sul client che può essere utilizzato per stabilire una connessione con un server sulla rete. La chiamata restituisce un intero tipicamente positivo (file descriptor associato al socket appena creato), se non va a buon fine, come abbiamo già visto con le altre system call ad AdC, restituisce -1.

Se il client poi esegue la funzione connect() essa permette di mandare un messaggio SYN verso il server, flag che abbiamo visto servire a richiedere l’apertura della connessione.

Lato server:

il server crea anch’esso un socket stream, attraverso la socket(); il socket agisce in modalità server, aspettando una richiesta di apertura della connessione dal parte del client

* Dopo aver creato il socket il server chiama bind() che definisce il numero di porta sulla quale avverrà la comunicazione e alla quale il client si connetterà con connect()

(attraverso la bind() il server comunica al S.O. il suo numero di porta, così avviene il demultiplexing delle informazioni come spiegato in precedenza

* Il server esegue la listen() che informa il S.O. che il server è pronto ad accettare connessioni sulla porta indicata dalla bind()
* Dopo che il socket riceve la SYN mandata dal client, il server, se vuole aprire la connessione deve utilizzare la funzione accept() per inviare SYN/ACK e avviare la connessione

(accept() restituisce un intero corrispondente al file descriptor che identifica un altro socket, accept() inoltre è bloccante: ferma l’applicazione finché non viene aperta la connessione

Il server utilizza quindi n+1 socket:

* Il primo socket, su cui sono state effettuate bind() e listen(), che è servito a creare la prima connessione e crea gli altri socket
* Gli altri n socket sono per le n connessioni con gli n client che si collegano al server, essi mantengono comunque il numero di porta del socket. Esse rispondono alle primitive send, receive, write e read.

Datagram socket

Qui è possibile scrivere una “lettera” con degli header in cui viene specificato l’indirizzo del destinatario.

Verranno utilizzate delle primitive di comunicazioni diverse dalle receive, send e write degli stream socket: sendto() e recvfrom().

sendto() richiede che venga specificato l’indirizzo del destinatario e in recvfrom() viene specificato l’indirizzo del mittente. Qui per avviare la comunicazione non serve attribuire dei ruoli C/S ma è necessario che all’inizio uno dei due host conosca l’indirizzo dell’altro, dopo il primo mess. Attraverso la recvfrom() anche il secondo host conoscerà l’indirizzo del primo, a cui adesso sarà capace di inviare messaggi (conoscendone l’indirizzo).

Per creare la comunicazione di tipo datagram devo chiamare socket() per creare il socket, almeno uno dei due dispositivi che vogliono connettersi deve chiamare la bind() per informare il S.O. della connessione che si vuole aprire, attraverso questa syscall si assegna il numero di porta. Uno dei due host deve aver comunicato all’altro in precedenza il numero di porta che si vuole utilizzare, così l’altro può già inviare un messaggio usando la sendto().

DNS

Il DNS (domain name system) è un complesso di server molto grande in tutto il mondo che lavorano per “tradurre” i nomi dei domini dei siti web in indirizzi IP. In breve i client si connettono a dei server locali in modalità datagram UDP.

Struttura gerarchica ad albero:

* Root Name Server: primo server della gerarchia: contiene informazioni sui server TLD (quelli del livello successivo a questo). Funge da punto di partenza per la risoluzione di nomi dei domini.

Esso indirizza quindi le richieste verso i server TLD (se un client invia una richiesta per "www.example.com", il Root Name Server indirizzerà la richiesta verso il server TLD "com" per trovare l'indirizzo IP del server di nome corrispondente a "www.example.com".)

* TLD(Top-Level Domain): segmento a destra del punto nei nomi dei domini (com come nell’esempio precedente, web, org, gov, eccetera)
* DNS autoritativo: contiene informazioni complete e aggiornate su un dominio o sottodominio, risponde alle richieste dei client per tale dominio ed è designato come fonte ufficiale e attendibile per le informazioni su un determinato dominio



Algoritmi di ricerca:

* Ricorsivo: client contatta server locale, il quale contatta root name server, il quale contatta il giusto tld, il quale contatta il server autoritativo
* Iterativo: client contatta server locale, il quale risponde con l’IP del root, il client contatta il Root, il quale risponde con l’IP del TLD, il client contatta il TLD, il quale risponde con l’IP del server autoritativo, il client contatta il server autoritativo

Nella pratica la ricerca avviene in due fasi: quella ricorsiva e poi quella iterativa:

esempio: scrivo sulla barra di ricerca del mio browser [www.google.com](http://www.google.com)

dopo che il client fa la richiesta al server DNS locale, il server locale contatta il Root Name Server, che, avendo l’elenco di tutti i TLD, risponde al server locale dandogli l’IP di com, ma non sono ancora arrivato alla risoluzione, il server locale quindi fa la richiesta al TLD (avendone l’indirizzo IP) e il TLD risponde con l’IP del DNS autoritativo, il server locale allora andrà da lui a fare un’altra richiesta, per trovare finalmente la risoluzione di [www.google.com](http://www.google.com).

Questo approccio “combinato” in cui il server locale agisce da intermediario crea meno carico sulla gerarchia dei server.

I server locali devono “mantenere a mente” le richieste mandate che sono rimaste senza risposta, dopo una certa quantità di tempo (soglia di time-out) se non ricevono risposta rinviano la richiesta.

Alla fine di una richiesta di risoluzione il server locale conosce la risposta alla richiesta fatta, la salva nella sua cache e quindi sarà pronto a soddisfare un’altra richiesta identica a questa, una seconda volta potrà dare una risposta immediata senza fare tutta la “trafila” descritta sopra. Gli elementi in cache hanno un TTL (time to leave), dopo il quale vengono cancellati.

Struttura dei messaggi che i server DNS si scambiano:

* Identificatore: identifica univocamente una richiesta
* Richieste: stringa alfanumerica (cioè nome del dominio che si vuole risolvere, ex www.servizionline.unige.it)
* Risposte: risposte contenenti l’eventuale indirizzo IP cercato e trovato nella cache
* Risposte autoritative: risposte contenenti l’indirizzo IP cercato e trovato in seguito a una ricerca ricorsiva/iterativa (senza utilizzo della cache quindi)
* Risposte addizionali: indirizzo IP del server autoritativo (necessario per verificare la validità di una risposta autoritativa ricevuta)

Esempio di funzionamento:

si vuole risolvere un nome di dominio in indirizzo IP:

* Per prima cosa il client cerca nella sua cache (in /etc/hosts) se è già presente quello che si sta cercando, se così non fosse, il client va a chiedere al server locale.
* Il server locale, ricevuta la richiesta dal client, si “mette all’opera”, cerca innanzitutto nella sua cache,

se trova quello che si sta cercando fornirà una risposta non autoritativa e avremo finito, altrimenti interroga il Root Name Server e inizia tutto il processo spiegato in precedenza.

Ci sono 13 copie di Root Name Server nel mondo, ognuna con un diverso IP, così che non ci sia troppo carico su un server solo, inoltre, nel caso uno si guastasse ci sono sempre gli altri che possono rispondere alle richieste e garantire il funzionamento del sistema dei nomi dei domini.

Gli internet service provider (ISP) sono organizzazioni che forniscono accesso a internet a utenti (privati o aziende), essi hanno dei server chiamati nameserver, i quali funzionano come intermediari tra i clienti dell’ISP e la struttura globale DNS, quindi i name server fanno riferimento ai Root Name Server per risolvere i nomi di dominio in indirizzi IP.

I client possono operare in modalità stateless: non ricordarsi della richiesta appena fatta e non conservarla in memoria, ciò renderà ogni richiesta indipendente dalle precedenti.

DNS Cache Poisoning:

Un utente malevolo potrebbe intercettare la richiesta di un client al server locale e di conseguenza modificare la risposta che esso darà.

Quante possibilità ci sono che questo accada? Il malintenzionato, per fare ciò, dovrebbe conoscere sia l’ID della richiesta del client che il suo numero di porta, entrambi numeri generati casualmente ad ogni richiesta, entrambi codificati su 16 bit, quindi ha una chance su di “azzeccare” la combinazione giusta.

Se ci riuscisse, nella cache del server locale verrebbe memorizzata la risposta “sbagliata” voluta dal malintenzionato e ogni client che da allora in poi farà quella richiesta riceverà tale risposta corrotta.

I browser possiedono una memoria in cui sono salvati gli indirizzi IP più famosi così da non mandare una richiesta al server locale, ma avere direttamente una risposta non autoritativa nel caso un utente cerchi di accedervi.

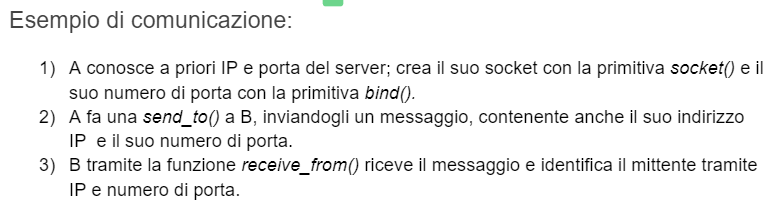
DNS utilizza principalmente UDP come “metodo” di trasporto. (Abbiamo visto prima cosa vuol dire stateless: ci accorgiamo che UDP può essere considerato stateless poiché a ogni nuovo scambio di messaggi si richiede di inserire indirizzo IP degli host coinvolti e numero di porta di comunicazione, quindi ogni connessione è indipendente dalle precedenti)

Comunicazione di tipo datagram

Come abbiamo già visto useremo la sendto() per l’invio dei messaggi e la receive\_from() per la ricezione di messaggi.

Per avviare la connessione serve che un host (ad esempio il client) conosca l’indirizzo dell’altro (ad esempio il server locale) e verranno usate socket() per creare il socket e bind() per assegnare il numero di porta.

Il **numero di porta** stabilito a priori dalla IANA per l’uso con il protocollo DNS è **53**, quindi i client si connettono in modalità Datagram ai server locali sulla porta 53 e mandano una stringa (URL del dominio) da risolvere in indirizzo IP.



Come abbiamo già visto UDP ha un flusso di byte di lunghezza limitata e per gestire un flusso più grande di dimensioni arbitrarie bisogna usare TCP.

Principalmente in DNS si usa UDP ma è possibile usare anche TCP per superare il limite appena descritto, per fare ciò però bisognerà prima aprire la connessione con il 3-WH.

Se avessimo un server al quale si devono connettere uno o più client, ognuno dei client dovrà chiamare la **socket()**; anche il server deve chiamare la socket() per consentire la comunicazione con il S.O. e quindi l’apertura della connessione. Come abbiamo già visto il server chiamerà la **bind()** per avere il numero di porta sulla quale avverrà la comunicazione e alla quale il client si dovrà connettere con la **connect()**, il server esegue una **listen()**, quindi si mette “in ascolto”, aspettando la connessione del client, che avverrà grazie a **connect()**, con la quale il client manda un messaggio SYN verso il server, si richiede quindi di aprire la connessione con il 3-WH. Il server, se vuole aprire la connessione, utilizza la accept(), con la quale manda SYN/ACK al client. Come abbiamo già visto, se ho n client in comunicazione con un server, il server arriverà a creare n+1 socket (il primo socket che apre la connessione e “resta in ascolto” per le connessioni in ingresso e gestisce le connessioni con i nuovi host + un nuovo socket per ogni nuovo client che si connette), quindi ad esempio: 3 client🡪 il server apre 4 socket.

Prestazioni di una rete

Nell’invio di un messaggio da A e B, esso può passare attraverso diverse strade nella rete per arrivare a destinazione, host A chiama la funzione write(buf, n) con:

* Buf il buffer di partenza del messaggio
* n il numero di byte da inviare

e host B chiama la funzione read(buf, n) con:

* buf il buffer di arrivo del messaggio
* n il numero di byte da ricevere

per calcolare il tempo che ci mettiamo a inviare effettivamente un messaggio usiamo l’applicazione Ping Pong, che ci permette di inviare più volte lo stesso messaggio, per poi fare una media tra le differenze di write e read (tempo di andata= tempo che B ci mette per completare la read - tempo che A ci mette per completare la write; tempo di ritorno= tempo che A ci mette per completare la read – tempo che B ci mette per completare la write) si sommano tempo di andata e di ritorno per calcolare il Round-Trip Time.

NTP

Network Time Protocol è un protocollo Client-Server appartenente al livello applicativo, vi è destinata la porta **123** e utilizza il protocollo di trasporto **UDP**. È fondamentale per la sincronizzazione degli orologi di una rete (utile per il caching e altri processi). Gli orologi sono contatori che si incrementano ad ogni secondo.

Come abbiamo già visto ad AdC ci sono due standard per la rappresentazione del tempo (Unix e UTC)

Il valore 0 del contatore (chiamato epoca) è:

* il 1° gennaio 1970 per UNIX
* il 1° gennaio 1900 per UTC

contatore diviso in due sezioni: una conta i secondi, l’altra i nanosecondi. Il tempo è codificato a 32 bit, ciò genera il problema dell’anno 2038, infatti avendo 32 bit, per il caso UNIX 1° gennaio 1970 + è la massima data rappresentabile, quindi si raggiungerà l’overflow.

È per questo che si sta passando quindi alla rappresentazione a 64 bit, grazie alla quale non si avranno più problemi di questo tipo, potendo rappresentare date come 1970+trilioni e trilioni di secondi (circa 500 miliardi di anni, quindi non ci dovremmo preoccupare di questo problema per un bel po’), facendo così si aumenterà anche la precisione, perché avendo così tanti bit a disposizione, conteremo i picosecondi e non i nanosecondi con il secondo contatore (perché comunque è abbastanza improbabile arrivare all’overflow di miliardi di anni siccome il sistema solare probabilmente non esisterà più).

I server sono classificabili in base al loro strato cioè la loro “vicinanza” ad un orologio atomico, un server che al suo interno contiene un orologio atomico è detto di strato 0, un server collegato ad un server di strato 0 è detto di strato 1, un server collegato ad un server di strato n sarà un server di strato n+1.

Per convenzione si identifica un server di strato n=15 come server non sincronizzato.

Quando si sincronizza un client con un server di strato n, il client è di strato n+1.

Abbiamo detto che lo scopo di NTP è sincronizzare l’ora di un client rispetto ad un server, siccome NTP usa UDP, la poca affidabilità caratteristica di UDP viene risolta interrogando più server (si va ad aumentare quindi la ridondanza), una volta interrogato un certo numero di server vado a stabilire quale sia il server più affidabile, in base a quante risposte ricevute e non andate perse ho avuto.

Scelgo inoltre un server anche in base al suo strato (precisione) e alla variazione dei round-trip-time.

Problemi di sicurezza

Un certificato digitale è un’attestazione di proprietà di una chiave pubblica crittografata, tali certificati hanno una data e un’ora di scadenza, viene facile capire che, se ci fossero problemi con la sincronizzazione del tempo, ad esempio se fosse più indietro di quello che dovrebbe essere, si potrebbero accettare certificati non validi perché ormai scaduti. Essendo basato su UDP, NTP può essere soggetto a spoofing (falsificazione di indirizzo IP sorgente di un pacchetto UDP, così da falsificare l’identità del mittente o del destinatario ad esempio) e di conseguenza anche ad attacchi DDos:

un host malevolo, una volta individuato il suo “host vittima”, potrebbe inviare con quell’indirizzo IP moltissime richieste NTP a svariati server, l’host vittima quindi riceverà tantissime risposte dai vari server, e ciò porterà al malfunzionamento della macchina, un’altra possibilità è che si crei attraverso spoofing un falso indirizzo IP (non associabile ad una fonte reale) e si inondi di richieste uno o più server NTP, così da impedirne il corretto funzionamento. A questi problemi ci sono diverse soluzioni o tentativi di soluzione, una di queste è il protocollo NTPsec, variante più sicura di NTP.

Un esempio di uso di NTP è il sistema GPS (tramite l’interrogazione di più orologi atomici presenti nei satelliti si può risalire ad una determinata posizione nel globo).

SNTP (Simple Network Time Protocol) è una versione semplificata di NTP, usata ad esempio nei dispositivi mobili. (comporta un minore dispendio di energia e memoria, ma anche una minore precisione)

FTP

È un protocollo client - server appartenente al livello applicativo per la trasmissione di dati tra host (File Transfer Protocol) basato su TCP.

È uno dei primi protocolli definiti della Rete Internet.

La prima specifica risale al 1971 (RFC-114) e quella attuale fa riferimento all’RFC-959.

Il protocollo è utilizzabile in due modalità:

* Passiva: qui client e server entrano in comunicazione sulla porta **20** (la loro connessione è detta connessione di controllo), il client manda comandi sotto forma di stringhe e il server esegue tali comandi. I due comandi principali sono GET (download) e PUT (upload), i comandi FTP sono codificati in ASCII (caratteri maiuscoli, non più di 4 generalmente). Per effettuare un trasferimento bisogna aprire una connessione (connessione dati) sulla porta **21**, i dati verranno mandati come interi a 8 bit.

Per utilizzare FTP un dispositivo deve essere autenticato sia sul server che sul client, per poter identificare un file devo indicare un pathname per il file, uno per la directory sorgente e un altro per quella di destinazione.

* Attiva(default): il server apre la connessione dati, mentre il client apre la connessione controllo. Per aprire la connessione, il server deve conoscere l’indirizzo IP e il nr di porta del client, che troverà nella richiesta di connessione controllo da parte del client sulla porta **20**. Questa modalità è ormai obsoleta, aveva problemi di sicurezza, similmente a NTP anche qui un malintenzionato può falsificare l'IP potendo compier attacchi DDoS. Un’altra problematica sarebbe potuta incombere quando un server contattasse un client senza autorizzazione, ma ciò è stato risolto introducendo i firewall.

Per passare da attiva a passiva si usa il comando passv.

Svantaggi di FTP:

* Deve gestire due connessioni
* Obbliga l’utente ad autenticarsi
* L’autenticazione utilizza una password non cifrata🡪rischio SNIFFING

(per questo ultimo problema la soluzione è una variante di FTP chiamate SFTP).

Alternativa migliore a FTP è l’HTTP che vedremo avanti.

Anonymous FTP:

consente di definire un account anonimo al quale si accede anche senza password. In modalità anonima non si può sia scaricare che caricare contemporaneamente. Le due attività sono mutualmente esclusive, cioè o fai download nel momento x ma non puoi fare upload, o viceversa. (per motivi di sicurezza).

Richieste e risposte FTP

Abbiamo già visto che le richieste (i comandi) sono codificate in ASCII a max 4 caratteri, le risposte in vece sono codificate in numeri seguiti da una breve frase in “linguaggio umano”, ad esempio 200 OK.

OK è la parte destinata all’utente (l’utente vede che il server accetta la richiesta), 200 è la parte destinata alla macchina (la prima cifra indica una categoria generale di risultati: con 2xx si indica successo, con 4xx errore lato client, con 5xx errore lato server)

Attualmente non si usa più la versione originale di FTP, ma quelle aggiornate, che cifrano le password.

SMTP

È un protocollo a livello applicativo per l’invio di messaggi di posta elettronica (Simple Mail Transfer Protocol). Si stabilisce una connessione in modalità di trasferimento asincrono (l’idea è, proprio come nella posta fisica, che l’utente possa leggere il contenuto del messaggio anche se quando lo riceve “non è a casa”, infatti, se il postino inserisce una lettera nella nostra casella postale, essa rimarrà lì fino a quando noi non decideremo di controllare e aprirla).

La comunicazione asincrona tra due dispositivi è resa possibile da server nei quali è presente una mailbox, in modo che il destinatario possa accedervi e controllare la propria posta.

SMTP è basato sul protocollo di trasporto TCP e su quello di rete IP, il servizio viene fornito sulla porta **25**.

Ci possono essere dei server “intermedi” che memorizzano il messaggio prima di recapitarlo nella mailbox, così da rendere ininfluente il fatto che il server con la mailbox del destinatario non sia effettivamente connesso, quest’ultimo, quando tornerà online andrà a prendere il messaggio memorizzato nel server intermedio. I dati trasferiti sono codificati in ASCII a 7 bit.

Messaggi composti da:

* Un headeer, con diversi campi: subject, from, to, date (di cui to è l’unico obbligatorio)
* Un body

Le righe di testo sono di massimo 72 caratteri, per andare a capo si usa <CR> (carraige return) (sequenza di controllo che fa tornare all’inizio della riga corrente) in combinazione con <LF> (Line Feed) (sequenza di controllo che fa avanzare di una riga).

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 MIME, estensione del protocollo che si usa anche per ammettere l’invio di foto e video come allegati.

Si può riassumere la comunicazione come un modello Client-Server-Client, perché il dispositivo di un utente che vuole mandare una mail a un altro utente è identificato come Client, perché fa la richiesta di inviare i dati, il Server la accetta, riceve i dati e (in molti casi anche attraverso l’intermezzo di altri server) li consegna al destinatario, ma anche il dispositivo del destinatario stesso è un Client, perché accedendo all’applicazione di posta elettronica sta facendo una richiesta, andando ad aprire la mailbox e controllare i messaggi.

Esempio di comunicazione:

abbiamo un client: una applicazione (Mail User Agent), ovvero un software che gestisce l’interfaccia utente per la ricezione/invio dei messaggi di posta elettronica, che dialoga con un server MTA (Mail Transfer Agent). L’idea è che il server si occupi di recapitare al destinatario il messaggio inviato dall’applicazione MUA.

* L’ host A con il suo MUA si connette a un server SMTP, aprendo una connessione sulla porta 25.
* Una volta stabilita la connessione, il server manda un messaggio di tipo 220 OK per comunicare che è pronto a ricevere messaggi
* Il client A, ricevuta la risposta positiva del server, manda al server il comando HELO, con il quale si identifica al server e fornisce informazioni sul proprio indirizzo IP
* Il server risponde con 250 OK, comunicando che il comando HELO è stato eseguito correttamente e che il server è pronto a ricevere altri comandi, quindi che il client è stato identificato correttamente.

a questo punto il client può mandare una serie di comandi per spedire un messaggio, ovvero

RCPT TO (con cui specifichiamo l’indirizzo mail del destinatario a cui recapitare il messaggio)

MAIL FROM (con cui specifichiamo l’indirizzo mail del mittente)

Il server, ricevuti i comandi, se tutto va a buon fine, risponde ogni volta con 250 OK.

* Con il comando DATA il client richiede al server il permesso di poter mandare il messaggio.
* Il server, ricevuto anche il comando DATA, risponderà con 354, risposta che indica che il client può iniziare a trasmettere i dati del messaggio. Oltre a 354 il server scrive anche la stringa “ end with <CR> <LF> . <CR> <LF> ” poiché, per quanto spiegato prima si aspetta che il testo sia formattato correttamente, e quindi finisca con il punto e la nuova riga: terminatore del body.
* Adesso inizia il vero e proprio invio del messaggio, il client comincia a mandare sequenze di byte in ASCII 7 bit, iniziando con l’ header e poi concludendo con il body.

Chi riceve i messaggi non ha modo di controllare o agire in qualche modo sul “from” dei messaggi che gli arrivano una volta arrivati, questo porta a problematiche come spoofing e spam, ma un modo per risolvere questo problema è evitare che certe mail arrivino proprio in primis, si può filtrare il from di tutti i messaggi che arrivano dal server e non salvare direttamente nella mailbox quelli dal “from” che ci “sta antipatico”, quindi se un messaggio è riconosciuto da un certo algoritmo come “messaggio spam” esso non sarà inserito nella mailbox del destinatario, che ne gioverà perché, più messaggi si ricevono via mail, più risorse si consumano, quindi ricevere migliaia di messaggi inutili sarebbe un grande spreco di risorse.

Un server senza filtri è detto open relay. Tale tipo di server permette quindi a chiunque sulla rete di inviare e-mail tramite esso, per “aggirare” il problema di chi voglia usare server open relay si implementa un meccanismo di autenticazione attraverso il comando EHLO, così da identificare possibili spammer.

Altre informazioni su SMTP:

Essendo uno dei primi protocolli internet sviluppati, SMTP riceve un “trattamento speciale” da parte del DNS. Il domain name system contiene delle “raccolte” chiamate record MX nel quale viene specificato, per ciascun dominio, quale è il server SMTP da contattare.

La qualità del servizio offerta dal protocollo SMTP viene definita best effort (miglior rapporto tra “sforzo” e “resa”), è una via di mezzo in termini di affidabilità, utilizza una quantità limitata di risorse per produrre la minima quantità possibile di errori, non garantisce quindi la massima affidabilità ma “fa del suo meglio” per raggiungerla. L’idea è quella di mantenere in memoria il messaggio da inviare finché non abbiamo la conferma di ricezione da parte del destinatario, una volta che si viene a conoscenza di questo, il server mittente può procedere a eliminare il messaggio (store&forward).

Per come l’abbiamo visto, quindi, SMTP permette l’invio di messaggi e-mail, ma quali sono i protocolli che si occupano della ricezione di tali messaggi?

POP e IMAP:

l’idea è quella di poter consentire ad un utente che si può autenticare, di consultare la sua mailbox anche da altre macchine, diverse da quella con cui ha “creato l’account” di posta elettronica, ad esempio.

* **IMAP**: dà la possibilità a più dispositivi hardware di accedere alla stessa mailbox; IMAP mantiene la mailbox sul server ed effettua copie parziali attraverso caching: quando il client richiede di vedere un determinato messaggio, con IMAP se ne fa una copia dalla mailbox del server e la si mostra al client, mantenendo quindi questa copia in locale, ma l’originale sempre nella mailbox nel server. Il caching permette inoltre che, quando si esaurisce lo spazio nella macchina, venga eliminato i/il messaggi/o letti/o con minor frequenza per lasciare spazio a un nuovo messaggio (sempre discorso del time to leave e least frequently used già affrontato a AdC). La versione ora in uso è la IMAP4
* **POP**: effettua una copia completa della mailbox dell’utente dal server alla macchina, e poi svuota la mailbox del server, quindi i messaggi rimangono solo sul client locale: ciò non è efficiente né rapido, ma è il metodo di utilizzo più comune di POP3, un altro metodo di utilizzo prevede che vengano copiati tutti i messaggi sul client e che vengano conservati anche sul server per un periodo di tempo limitato, ciò però comporta, oltre a una maggiore affidabilità, un grandissimo utilizzo di risorse e memoria.

viene facile comprendere perché IMAP sia considerato più flessibile ed efficiente di POP.

Attualmente per gestire la posta elettronica si usa Webmail (basato su http).

PROTOCOLLO IP NEL DETTAGLIO

Come abbiamo già visto, IP è un protocollo a livello di rete che serve per l’instradamento dei pacchetti sulla rete attraverso router intermedi.

Tra host A e host B ci possono essere diversi instradamenti possibili.

L’idea generale è quella di A che si collega a un primo router e il router B che si collega ad un router, che definiamo come “router di frontiera” perché è l’ultimo router attraverso il quale il datagramma passerà prima di arrivare a B. Bisogna trovare tutti i router intermedi.

A picture containing shape

Description automatically generated

Nell’ header dei datagrammi che vengono instradati con IP c’è l’indirizzo IP del destinatario.

Il primo router riceve il datagramma e lo immagazzina nel proprio RX buffer, poi controlla dentro all’intestazione quale sia l’IP Address del destinatario, e in base a questo, sceglie a quale prossimo router mandare il messaggio per farlo arrivare a B.

Supponiamo di avere un router che ha la possibilità di connettersi con altri 8 router diversi, quindi questo router, una volta letto l’IP di B può decidere tra 8 vie, quale sceglie? Per prima cosa sarebbe ottimale che l’operazione di routing venga effettuata nel minor tempo possibile.

A questo scopo viene definita una tabella, chiamata tabella di inoltro (forwarding table) che associa un IP con un numero di porta in uscita, bisognerà considerare solo i primi 8 bit di un indirizzo IPv4 (composto da 32 bit), in modo da non creare una tabella troppo grande (arriviamo a celle).

Gli indirizzi IP sono divisi in due parti:

* Network: indica la rete locale a cui il router fa riferimento
* Host: indica l’ host appartenente a questa rete locale

Esiste un numero delle stesse dimensioni dell’indirizzo IP che definisce le dimensioni della parte Network, si chiama netmask e funziona così:

ci sono tanti uni quanti numeri nella parte di identificazione di rete, tanti zeri quanti i numeri nella parte di identificazione degli host.

Es: 11111111000000000000000000000000

Più piccolo è il numero di bit dedicato al network, peggiore sarà la qualità dell’instradamento. All’aumentare di questo numero migliora la qualità dell’instradamento ma anche la tabella di forwarding sarà più grande.

Per costruire le forwarding table bisogna avere un supporto hardware adeguato che ce lo permetta: una memoria associativa che supporti celle di lunghezza variabile e vari bus che permettano la connessione della tabella con i router. Gli algoritmi di routing/instradamento funzionano “offline”, prima che il pacchetto arrivi l'algoritmo calcola dove esso verrà smistato. Tali algoritmi saranno eseguiti continuamente, così da trovare strade più vantaggiose, ecco perché se mandiamo una serie di datagrammi allo stesso destinatario non è detto che seguano tutti la stessa strada.

ICMP

Anche i router devono essere capaci di comunicare tra loro, per scambiarsi informazioni utili alla realizzazione di un instradamento ottimale, la comunicazione tra router avviene tramite Internet Control Message Protocol.

ICMP e IP sono stati sviluppati assieme e sono necessari l’uno all’altro, infatti ICMP utilizza IP per inviare messaggi tra router.

ICMP è quindi un algoritmo ausiliario che si occupa di trasmettere informazioni riguardanti malfunzionamenti o informazioni di controllo tra i vari componenti di una rete.

Header IMCP:

vi sono codificati due interi:

1. Type
2. Code

Table, calendar

Description automatically generated

Esistono varie applicazioni che possono essere usate per avere “informazioni” riguardo una rete o riguardo alla possibilità o meno di instaurare connessioni, tali applicazioni utilizzano proprio il protocollo ICMP.

Ad esempio ping, utilizzata proprio per verificare se sia possibile instaurare una connessione tra due host e calcolarne il RTT, essa sostanzialmente permette di inviare messaggi di tipo ICMP tra un host e l’altro:

funzionamento:

1. il primo messaggio inviato da H1 a H2 sarà di tipo 8 code 0 (echo request come vediamo nella tabella: richiesta di ping)
2. la risposta a tale messaggio sarà un messaggio ICMP tipo 0 code 0 (echo reply)
3. continua uno scambio di questi messaggi per diverse iterazioni, finché si può calcolare con abbastanza precisione il round-trip time, allora se si trova un RTT non ragionevole si dichiara che non è conveniente instaurare una connessione tra i due host.

Il vantaggio di questa applicazione è che si rimane al livello 3, non bisogna salire di livello e complicarsi con numeri di porta eccetera, ma si può verificare previamente che la connessione sia stabile/affidabile.

Algoritmo di routing:

come abbiamo già visto, se ad esempio ho 2 indirizzi, uno con 4 bit di network e un altro con 24 sceglierò il secondo.

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

1. Link state (approccio globale):

ogni router comunica con tutti gli altri router della rete e comunica le sue connessioni, viene creato un grafo, in cui i nodi sono tutti i dispositivi connessi alla rete, a questo punto si può usare l’algoritmo di Dijsktra per determinare il percorso minimo tra due nodi (host A e host B) di un grafo.

1. Distance vector (approccio locale):

ogni router comunica coi suoi vicini**,** il messaggio si propaga e dopo un breve lasso di tempo tutti i router hanno delle stime abbastanza buone sulla situazione della rete, approssimativamente quindi si decide su quale porta di uscita è meglio mandare i messaggi, dopo che ogni router ha misurato la distanza tra esso e i suoi vicini, si cercherà anche qui di raggiungere la destinazione in meno hop possibili.

Viene facile capire che il link state è molto più dispendioso a livello di risorse (siccome richiede una copia completa di tutta la mappa di rete) e di tempo, perché ci vorrà più tempo affinché tutti comunichino con tutti, d’altra parte però c’è molta precisione; il distance vector invece, sarà meno preciso ma sicuramente più veloce, in base alla dimensione ed al traffico della rete sarà opportuno scegliere l’uno o l’altro approccio.

Traceroute:

funzionalità software a cui si può accedere da terminale che serve per visualizzare il percorso che un pacchetto farà nella rete per passare dall’host sorgente a quello di destinazione.

Traceroute si basa su ICMP, si inserisce come destinazione l’indirizzo IP dell’ host di destinazione, e come porta una porta casuale:

una volta azionato il comando, l’ host di partenza inizia a inviare pacchetti: invia il primo pacchetto con TTL=1, il datagramma effettua quindi il primo hop e arriva al primo router, fatto ciò, il TTL viene decrementato, ma siccome il TTL era 1 adesso è diventato 0 (il pacchetto deve essere scartato, non può arrivare a destinazione), arriva una segnalazione di errore ICMP type 11 all’ host di partenza (in questo datagramma di risposta c’è anche l’indirizzo IP del router che l’ha inviato).

Così facendo abbiamo identificato il primo router per cui il pacchetto dovrà passare.

Viene inviato un secondo pacchetto, questa volta con TTL=2, , il datagramma effettua quindi il primo hop e arriva al primo router, fatto ciò, il TTL viene decrementato (TTL=1), il datagramma effettua quindi un altro hop e il TTL viene decrementato, siamo al secondo router e TTL=0, adesso quindi sarà il secondo router a rispondere con un messaggio ICMP type 11.

Così facendo abbiamo identificato il secondo router per cui il pacchetto dovrà passare. Si invia poi un pacchetto con TTL=3 e così via, fino a completare l’instradamento.

Es:

Diagram

Description automatically generated

Arrivati all’ultimo router per cui il pacchetto dovrà passare prima di arrivare a destinazione, riceviamo un’ultima volta il messaggio ICMP type 11, ora aumentando ancora di 1 il TTL possiamo raggiungere l’ host di destinazione, la risposta che possiamo avere è type 3 code 3 (porta non raggiungibile) oppure type 0 code 0 (echo reply), in entrambi i casi sappiamo che siamo arrivati a destinazione siccome non abbiamo più la risposta “time exceeded”.

**Header IPv4**

Table

Description automatically generated

L’header IPv4 ha dimensioni variabili, identificate dal campo lunghezza dell’header.

Il campo versione indica la versione del protocollo in uso (v6 o v4), il campo identificatore di servizio non è sempre usato ma specifica i requisiti di servizio per il pacchetto come priorità o qualità del servizio, in lunghezza del datagramma c’è scritto quanto è lungo tutto il datagramma (header + contenuto effettivo); la lunghezza minima è 20 byte, che è la minima lunghezza dell’header, quella massima byte, l’ identificatore è un numero che identifica in modo univoco il pacchetto, la flag di frammentazione è un valore booleano che indica se il pacchetto subisce o meno frammentazione (per il discorso di prima sappiamo che se un pacchetto di dati che dobbiamo mandare ha dimensioni maggiori della MTU: >1500 byte dovrà essere frammentato), l’ offset di frammentazione indica “che frammento” del messaggio originale è questo datagramma (il primo frammento ha offset 0, il secondo offset 1, eccetera).

Il TTL indica il numero massimo di salti che il datagramma può fare prima di essere cancellato dalla memoria del router, questo serve affinché un messaggio non stia sulla rete per troppo tempo, congestionandola e provocando errori, il campo protocol indica quale protocollo verrà usato al prossimo livello (livello di trasporto): UDP o TCP. Il checksum è un valore che viene creato nel momento in cui il mittente manda il datagramma, cambia ad ogni hop ma può essere sempre ricalcolato per verificare l’integrità dell’header e del messaggio stesso durante il trasporto. Indirizzo sorgente è l’indirizzo del “primo host della catena”, cioè quello che ha inviato il datagramma, l’indirizzo destinazione è l’indirizzo dell’ host a cui deve arrivare il datagramma.

**Header IPv6**

Table

Description automatically generated

La classe di traffico è simile all’identificatore di servizio di IPv4, permette di assegnare ad ogni pacchetto una classe di priorità rispetto ad altri pacchetti provenienti dalla stessa source.

Etichetta di flusso indica i datagrammi dello stesso flusso (i datagrammi con la stessa label saranno parte dello stesso flusso)

Per quanto riguarda la lunghezza di un datagramma IPv6 essa è variabile, ma la lunghezza dell’header è fissa, per questo c’è solo il campo lunghezza payload (perché la lunghezza dell’header è sempre 40 byte).

L’header di alto livello: indica, similmente al campo “protocol” di IPv4, quale protocollo verrà usato a livello di trasporto.

Simile al TTL in IPv6 abbiamo un valore detto hop limit, che viene decrementato ad ogni hop, una volta giunto a 0 bisogna cancellare il messaggio dalla memoria del router. Indirizzo sorgente e destinazione già spiegati nella versione 4, qua sono di 128 bit. Vediamo come l’header IPv6 ha decisamente meno campi; la frammentazione in questa versione è gestita dal mittente, non più dai router, quindi vengono eliminati i campi ad essa relativi, il checksum anche non c’è perché il controllo di integrità avviene a livello datalink.

Specifichiamo che un datagramma può contenere più header, definiamo la distribuzione di questi header “a livelli”: infatti un pacchetto avrà un header IP, che verrà elaborato a livello di rete, un header TCP/UDP che verrà elaborato a livello di trasporto e così via: una volta che è stato completato l’instradamento ci si occupa degli header sottostanti, ma i router, per attuare l’instradamento non controllano gli header interni, guardano solo quelli di alto livello.

**Compatibilità IPv6-IPv4:**

ci sono macchine che supportano una versione di IP e macchine che supportano l’altra.

Poniamo il caso che due host che vogliono comunicare supportino IPv6, ma nella rete, che permette loro di comunicare, ci sia un router intermedio IPv4:

se un router supporta esclusivamente IPv4 e riceve un datagramma IPv6, lo incapsula in un datagramma IPv4, mettendolo nel suo payload, se poi il ricevente supporta IPv6, rimuoverà l’header IPv4; questo processo è detto tunneling.

Se invece i due host fossero uno IPv4 e l’altro IPv6?

Bisogna che l’ host IPv6 si adatti: per passare da indirizzi IPv4 a IPv6 basta aggiungere una serie di 0 davanti all’indirizzo fino al completamento di esso. Vi è un processo di mapping che definisce quelli che sono chiamati IPv4 – mapped addresses che sono indirizzi pensati per favorire la transizione da v4 a v6, permettono infatti di definire indirizzi v6 per host che supportano solo v4:

i primi 80 bit sono posti a 0, i successivi 16 bit sono posti a 1 e negli ultimi 32 bit c’è l’indirizzo IPv4.

**Indirizzi privati e pubblici:**

esistono reti pubbliche e reti private, e di conseguenza indirizzi pubblici e privati, questi ultimi si dividono in 3 classi:

Table

Description automatically generated

Servono per identificare i dispositivi di una rete locale e permettere comunicazione ad essi senza che siano raggiunti da Internet.

Gli indirizzi privati non sono instradabili dai router, questo perché, in giro per il mondo ci possono essere device con lo stesso identico indirizzo IP privato ma appartenente a reti locali diverse.

Per mandare messaggi da una rete locale all’altra bisogna usare il NAT (Network Address Translation), una tecnologia che traduce indirizzi privati in pubblici. Il dispositivo che effettua la conversione è un router (di frontiera) con un indirizzo detto default gateway.

Se A, (avente indirizzo privato 10.10.10.10) all’interno di una rete locale, vuole comunicare con B al suo esterno dovrà mandare un datagramma che avrà come indirizzo sorgente l’indirizzo privato di A e come indirizzo di destinazione l’indirizzo pubblico di B, a questo punto con il NAT si andrà a tradurre l’indirizzo privato di A in pubblico. Se B vorrà rispondere userà come indirizzo di destinazione l’indirizzo pubblico, appena tradotto, di A e come sorgente il suo indirizzo privato, che verrà tradotto col NAT, quando il messaggio arriverà a destinazione (cioè ad A), il NAT ritradurrà l’indirizzo di destinazione da pubblico a privato, riottenendo 10.10.10.10.

Per spiegare brevemente a cosa serve la NAT possiamo dire che a più macchine sotto la stessa rete locale (privata) corrisponde sostanzialmente lo stesso indirizzo IP pubblico. Per attuare una comunicazione tra host A e host B appartenenti alla rete privata si useranno gli indirizzi IP privati, ma non sono questi quelli che vengono utilizzati per comunicare in Internet, all’esterno della rete privata, perché immaginiamoci di avere, in una rete privata, una miriade di dispositivi diversi che vogliono connettersi ad Internet, ad esempio 1000, e immaginiamoci che ci siano tante reti private del genere, se si desse ad ognuna di queste macchine un nuovo indirizzo IP pubblico univoco, questo vorrebbe dire esaurire tutti i possibili indirizzi IP in men che non si dica, soprattutto se siamo nel contesto di IPv4, che abbiamo visto essere più limitante a livello di, è proprio quando si usava solo IPv4 che nacque la NAT, per risolvere questo problema.

Brevemente la NAT funziona nel seguente modo:

un indirizzo privato associato ad una macchina X nella rete privata viene mappato in indirizzo pubblico seguendo questo criterio: si “parte” assegnando a tale rete privata un indirizzo IP pubblico con cui identificare tutti i dispositivi appartenenti, ad esempio:

128.243.20.20 e si userà un numero di porta diverso per identificare i diversi dispositivi della rete privata, in questo modo si “sprecheranno” molti meno indirizzi IP, e, siccome nessuna macchina vorrà mai usare tutte le possibili porte di connessione allo stesso momento non ci sarà “virtualmente” nessun problema.

Per esempio se una macchina A nella rete privata con indirizzo 10.10.10.10 fa una richiesta di connessione sulla porta 32621 (per una certa applicazione), si può mappare questa richiesta “per l’esterno” nel seguente modo:

128.243.20.20 porta 16 (esempio)

E se un’altra macchina B nella rete privata con indirizzo 10.10.10.11 fa la stessa richiesta sulla porta 32621 si potrà “mappare” come

128.243.20.20 porta 17 eccetera.

Per tenere traccia delle traduzioni viene creata una tabella indicizzata con i numeri di porta creati.

L’unico problema in tutto ciò è che esistono delle porte well-known che sono riservate per convenzione a determinati processi (dalla IANA): tra 0 e 1023, quindi le porte, chiamate porte effimere, che il NAT può effettivamente usare sono circa ( è il totale, al quale si sottraggono le porte well-known).

Port Forwarding

Un’altra tecnologia dei router che serve a permettere la comunicazione tra dispositivi appartenenti a reti locali diverse.

La sua funzione è quella di indirizzare il traffico in entrata verso una specifica porta e dispositivo nella rete locale: quindi in poche parole permette agli “estranei” la comunicazione con dispositivi interni alla rete privata. In un certo senso si può dire che il port forwarding sia la controparte lato server del NAT: il NAT traduce gli indirizzi privati dei singoli dispositivi della LAN in indirizzi pubblici per la comunicazione con l’esterno (client), mentre il port forwarding rende accessibili i dispositivi della LAN all’esterno di essa (server)

Diagram

Description automatically generated

Diagram

Description automatically generated

Nel NAT il numero di porta è associato all’IP dinamicamente, nel port forwarding è associato staticamente.

A (client) e B (server) vogliono comunicare:

Text

Description automatically generated

Livello 2 **Data Link**

In questo livello si mettono in comunicazione due o più host appartenenti alla stessa rete locale (**LAN**), ricordiamo che per rete locale si intende una rete di dimensioni contenute che possa occupare una piccola area geografica: un palazzo o una stanza.

Il protocollo principale di questo livello è Ethernet, alla fine degli anni ’70 si è affermato lo standard di questo protocollo: IEEE 802.3.

Oggi giorno una rete locale può anche essere completamente wireless, ossia la comunicazione tra i vari host è resa possibile senza l’utilizzo di cavi (questo tipo di implementazioni sono descritte nell’IEEE 802.11, una parte dell’IEEE 802, in cui è descritto anche ETHERNET come già detto). (Wi-Fi)

Le prime versioni di Ethernet utilizzavano un cavo coassiale: un tipo di cavo utilizzato per trasmettere segnali elettrici di radiofrequenza, composto da un conduttore centrale, circondato da un materiale isolante, il quale è avvolto da uno schermo di protezione (per evitare le interferenze elettromagnetiche).

L’idea era quella di “far correre” questo cavo lungo un corridoio e connettere i singoli host al cavo nei punti desiderati:

Diagram

Description automatically generated

I segnali venivano mandati sul filo e i singoli bit venivano codificati direttamente sotto forma di differenza di potenziale sul filo elettrico.

La comunicazione che Ethernet permetteva era quindi una di tipo **broadcast**: se host 1 voleva mandare un messaggio a host 2, tutti i dispositivi connessi al cavo erano in grado di “ascoltare”, il messaggio era mandato a tutti i dispositivi connessi al cavo, e poi ognuno di essi doveva decidere se tale messaggio fosse destinato a lui o meno:

Diagram

Description automatically generated

Ci accorgiamo che sia una implementazione di gran lunga più semplice a livello logico rispetto a ciò che abbiamo visto finora (niente routing, dns, menate varie) ma allo stesso tempo il fatto che ogni messaggio venga ricevuto da tutti i dispositivi della LAN comporta una congestione di rete nel momento in cui molti dispositivi vogliano inviare messaggi contemporaneamente.

Indirizzo MAC:

Nel pacchetto dati inviato c’è anche l’indirizzo univoco del destinatario, così che, nonostante tutti i dispositivi ricevano tutti i messaggi, solo il destinatario specifico possa rispondere alla richiesta. Tale indirizzo è chiamato indirizzo MAC (media access control).

Questo indirizzo è assegnato univocamente ad ogni scheda di rete al momento della produzione del dispositivo, quindi ogni dispositivo nel mondo ha un indirizzo MAC diverso dagli altri dispositivi.

Struttura di un indirizzo MAC:

configurazione binaria su 6 byte (quindi 48 bit), divisa in:

* 3 byte per i costruttori (identificano la ditta che ha prodotto il dispositivo)
* 3 byte per i dispositivi di quel determinato costruttore.

Il pacchetto di dati che viene trasmesso nelle comunicazioni Ethernet si chiama frame Ethernet, ogni frame ha la seguente struttura:

* Trama (preambolo): identifica l’inizio di un nuovo messaggio da parte di un host
* Indirizzo MAC di destinazione
* Indirizzo MAC di origine
* Payload (contenuto effettivo del messaggio)
* Valore di controllo (CRC-32): permette di rilevare se sono presenti errore di trasmissione, l’ host che riceve il messaggio calcola il suo valore di controllo mediante l’algoritmo CRC, e poi lo confronta con il valore ricevuto in questo campo, se i due valori sono uguali, si è certi che i dati sono stati trasmessi correttamente.

Quindi in breve:

Diagram

Description automatically generated

In una comunicazione di tipo broadcast, se un host sta usando il canale per inviare un messaggio, gli altri host non possono fare altrettanto, solo quando il messaggio verrà completamente trasmesso un nuovo host può iniziare a trasmettere un messaggio, solo un host alla volta può trasmettere dati sul cavo.

Come coordinare i vari host:

Carrier Sense Multiple Access with Collision Detection: CSMA/CD.

**Carrier Sense**: Ogni host monitora la situazione del canale e, solo quando c’è un “momento di silenzio” (il canale non sta essendo utilizzato da nessun altro host), inizia l’invio del suo messaggio.

Può succedere però che due o più host inizino contemporaneamente a mandare messaggi (per descrivere questa situazione si utilizza il termine di collisione), per via della lunghezza del canale nessuno dei due si accorge della collisione.

**Collision Detection**: tramite questo protocollo viene rilevata una eventuale collisione in corso, la collisione viene risolta con il

**Multiple Access**:

diviso in tre fasi:

1. **Jamming**: Tutti i dispositivi che hanno causato la collisione mandano una sequenza di bit, chiamata appunto sequenza “di jamming”, che rende evidente la presenza della collisione a tutti gli host nella rete
2. **STOP**: i dispositivi che hanno causato la collisione interrompono la trasmissione.
3. **Ritrasmissione**: viene ritenta una trasmissione dei dati dopo un certo tempo di attesa casuale (in modo che ciascun dispositivo che ha causato la collisione abbia una probabilità diversa di trasmettere per primo), se si genera un’altra collisione si raddoppia il tempo di attesa precedentemente utilizzato, tale tempo di attesa quindi ad ogni iterazione va a crescere esponenzialmente, e quindi la probabilità di collisione diminuisce, fino a diventare quasi impossibile, i messaggi vengono mantenuti nel buffer di trasmissione finché non arrivano a destinazione. Il limite massimo a cui il tempo di attesa può crescere è .

Le prime versioni di Ethernet con cavo coassiale viaggiavano a 10Mb/s, nel tempo la velocità è aumentata grazie all’introduzione di nuove tecnologie:

**Hub** e **Switch**:

negli anni ’70 il cavo coassiale veniva proprio bucato per permettere la connessione, usando l’HUB, un ripetitore e amplificatore di segnale, questo non è più necessario, si usano spinotti di connessione, ogni host è collegato al HUB attraverso uno spinotto.

Non si usa più il cavo coassiale ma il doppino telefonico (due fili di rame schermati che fanno passare i segnali in direzioni opposte) che aumenta la velocità di connessione fino a 100Mb/s.

La versione di Ethernet attualmente in uso si basa su un dispositivo chiamato Switch, dotato di varie porte a cui sono collegati gli host della rete.

Attraverso di esso si riesce a stabilire una comunicazione di tipo full duplex, si abbandona quindi la comunicazione di tipo broadcast: il messaggio non viene inviato a tutti, per poi venir gestito da un solo host che risponde, il messaggio adesso viene inviato solo al destinatario specifico grazie allo switch, che ha proprio la funzione di indirizzare il messaggio verso il destinatario corretto, dopo aver controllato l’indirizzo MAC di destinazione scritto nel messaggio.

Funzionamento(di tipo store & forward) :

l’ host mittente manda la trama Ethernet allo Switch, il quale la memorizza all’interno del suo buffer di ricezione, controlla cosa c’è nel campo indirizzo MAC di destinazione, e sposta il messaggio nel buffer di trasmissione della porta connessa all’host di destinazione.

Vengono usati 2 doppini telefonici, così da ottenere una comunicazione full-duplex.

La velocità resta sempre a 100Mb/s ma in HUB veniva spartita tra tutti i dispositivi collegati (siccome ogni messaggio veniva mandato a tutti), mentre qua la velocità è raggiungibile da ogni singolo host.

L’ unico svantaggio di fare store & forward è che la latenza aumenta (prima che il messaggio mandato da un host possa essere ricevuto dall’host di destinazione bisogna che esso sia completamente ricevuto dallo switch, che in seguito deve “smistarlo”).

Per fare questo lo Switch deve essere a conoscenza di tutti gli indirizzi MAC degli host collegati alle sue porte, bisogna associare ad ogni porta un indirizzo MAC, tramite un algoritmo chiamato di autoapprendimento:

1. **Fase dinamica** (di apprendimento): l’algoritmo parte all’accensione del dispositivo Switch, cre una tabella di associazione vuota, un dispositivo non viene registrato nella tabella quando si collega allo switch, bensì quando manda il suo primo messaggio, lo switch legge il messaggio, guarda il MAC sorgente e associa questo MAC alla porta alla quale l’ host che ha mandato il messaggio è collegato.

In caso un host debba ricevere un messaggio ma non abbia ancora mandato nessun messaggio, lo switch non conosce il suo MAC, quindi tale messaggio verrà mandato in broadcast (a tutti gli host della rete) e una volta che l’ host destinatario manderà la risposta, verrà associato il suo MAC alla rispettiva porta.

1. **Fase statica** inizia una volta costruita tutta la tabella di associazioni, qui non c’è più bisogno di inviare messaggi broadcast.

Nella fase iniziale si noterà una velocità addirittura minore a quella dell’HUB (sarebbe uguale per il fatto di inviare messaggi in broadcast ma è minor per via della latenza causata dallo switch), ma una volta completate le associazioni si potrà apprezzare la vera velocità della connessione switch, a questo punto molto elevata.

Lo switch possiede una modalità promiscua: se un host è amministratore di rete può decidere di ricevere tutti i messaggi passanti per quello switch, anche quelli che non sarebbero indirizzati a lui. È chiaro che tale modalità rappresenta una minaccia a livello di sicurezza della rete (potrebbe essere sfruttata per sniffing o spoofing, DDoS)

Diagram

Description automatically generated

**ARP:**

come abbiamo già visto un indirizzo IP è diviso in parte network e parte host, la prima parte identifica la rete locale di cui l’ host fa parte

esempio: 192.168.1.33 e 192.168.1.13 appartengono alla stessa LAN. Se un messaggio uscente da un host contiene come indirizzo di destinazione un indirizzo con la parte network identica a quella dell’indirizzo di tale host significa che il messaggio è destinato ad un host nella stessa sottorete.

Address Resolution Protocol è un protocollo che permette di “tradurre indirizzi IP in MAC”.

L’idea è quella di utilizzare una tabella di corrispondenza tra indirizzi IP e indirizzi MAC, è previsto l’utilizzo di una cache dove vengono memorizzate le associazioni tra IP e MAC.

inizialmente tale tabella sarà vuota, essa verrà riempita e aggiornata costantemente, tramite richieste e risposte ARP. Le unità dati che vengono trasmesse con questo protocollo sono molto simili ai frame ethernet, ai quali viene aggiunto il campo ARP, diviso in 28 byte, esso è a sua volta suddiviso in diversi campi, tra cui

ARP operation:

* + valore 1: ARP request
  + valore 2: ARP response/reply
* Un host (A) invia una richiesta ARP (ARP request) a tutti gli host sotto quella stessa LAN (broadcast), nel file ci sarà scritto l’indirizzo IP di cui si vuole conoscere il MAC
* Il dispositivo che possiede l’indirizzo IP in questione manda una risposta all’host A fornendo il proprio indirizzo MAC
* L’ host A, che aveva inviato la richiesta, registra l’associazione tra MAC e IP nella sua tabella.

Permettendoci di associare indirizzi IP a indirizzi MAC in una LAN, ARP mette in comunicazione il secondo livello con il terzo. Ethernet🡨🡪IP

Cache poisoning

In ARP non vengono memorizzate le singole richieste o risposte effettuate (si dice che è un protocollo stateless), ciò lo rende vulnerabile a vari attacchi.

Se qualche utente malevolo volesse mandare delle ARP reply con informazioni sbagliate, tali informazioni verranno inserite nella cache (nella tabella di associazioni IP-MAC).

Man in the Middle:

se un utente malevolo E (con indirizzo MAC 11:22:33:44:55:60) volesse intromettersi nella comunicazione tra A e B, potrebbe mandare ad A una ARP reply, nella quale dice che l’indirizzo MAC che A cerca è 11:22:33:44:55:60, quando in realtà A sta cercando l’indirizzo MAC che corrisponde all’IP di B, allo stesso modo può mandare una reply a B (che sta cercando il MAC di A) dicendo che l’indirizzo MAC che sta cercando è 11:22:33:44:55:60. In questo modo tutti i messaggi destinati ad A e B vengono inoltrati ad E, che può:

* Semplicemente “spiare”/monitorare la comunicazione tra A e B, ovvero, leggere i messaggi senza modificarli e re inoltrarli inalterati ai rispettivi destinatari (attacco passivo), senza che A e B siano a conoscenza di ciò
* Modificare i messaggi che A e B si vogliono mandare (attacco attivo)

Diagram

Description automatically generated

Diagram

Description automatically generated

DHCP: dynamic host configuration protocol

Protocollo che serve ad assegnare un indirizzo IP ad un dispositivo che si è appena connesso alla rete. (ad esempio dispositivo che si può spegnere e riaccendere, quindi non rimane sempre connesso)

Funzionamento:

1. **Richiesta** broadcast: per fare tale richiesta si usa IP/UDP (porta 68 client e 67 server) ma questo presupporrebbe che il client abbia un indirizzo IP da usare come sorgente ed uno come destinazione, ma noi non abbiamo un indirizzo IP, altrimenti non lo staremmo richiedendo, ci “autoassegniamo” un IP fittizio (0.0.0.0) e come destinazione verrà usato un indirizzo IP broadcast che ha nella prima parte gli stessi numeri della network e il resto tutti 1. L’indirizzo MAC sorgente è l’indirizzo del dispositivo che fa la richiesta, l’indirizzo MAC destinazione è l’indirizzo broadcast: FF:FF:FF:FF:FF:FF
2. **Risposta** unicast: un server DHCP riceve la richiesta e, avendo una lista di indirizzi IP liberi, ne assegna uno al richiedente
3. **Accettazione** broadcast: l’ host manda un messaggio di accettazione, comunicando che ha accettato l’indirizzo IP che il server gli ha assegnato. (non viene mandata al server DHCP specifico ma viene mandata in broadcast perché in una rete ci potrebbero essere più di un server DHCP, così facendo li raggiungo tutti, in caso più server abbiano assegnato un indirizzo IP all’host in questione, l’ host comunica quale ha scelto, così facendo inoltre, tutti i server DHCP sono stati informati e quindi dovranno cancellare dalla loro lista di indirizzi l’indirizzo scelto)
4. **Conferma** unicast: il server che ha assegnato l’ indirizzo comunica all’ host in questione anche altre informazioni sulla rete ed indica un tempo di scadenza di tale indirizzo, passato questo tempo l’ host dovrà richiedere un nuovo indirizzo e questo ritornerà nella lista dei disponibili. (ecco perché si chiama assegnazione dinamica, una volta scaduto il tempo, quando tale host richiederà un altro indirizzo non è detto che gli venga assegnato lo stesso di prima)

**HTTP (**protocollo del livello applicativo)

HyperText Transfer Protocol è indispensabile per la infrastruttura Internet odierna ed è utilizzato per trasmettere varie informazioni, accedere a siti web, scaricare risorse, ecc. …

Esistono varie versioni di http: 1.0, 1.1, 2.0

È un protocollo di tipo Client-Server e stateless (non memorizza le richieste e risposte). Il client (il browser generalmente) apre la connessione sulla porta TCP 80 per mettersi in comunicazione con il server. Le richieste e risposte http, similmente a quanto succedeva in SMTP ed FTP, sono codificate in ASCII.

La prima riga della richiesta contiene l’indicazione di:

* Un **metodo** (tipo di operazione da effettuare: get, put, delete, connect, etc.)
* Una **risorsa** sulla quale vogliamo agire (ad esempio << GET index.html 1.0 >> richiede la risorsa index.html
* Un **protocollo** da utilizzare (TCP, UDP, eccetera)

Terminiamo la riga, come abbiamo già visto in SMTP con <CR><LF>.

A questa riga seguono spesso delle righe opzionali con numero di opzioni arbitrarie:

* Host: permette di impostare il nome dell’ host a cui ci si vuole riferire (utile quando si utilizzano virtual host per avere diversi domini sullo stesso server)
* User-Agent: specifica il tipo di client che sta effettuando la richiesta (che browser è, che versione, etc)
* Cookie
* If-Modified-Since
* connection\_close (nella versione 1.0)

Ogni opzione deve essere terminata con <CR><LF>. Per concludere il messaggio di richiesta bisogna inserire un’altra riga vuota.

La risposta server ha una prima riga in cui c’è lo stato, codificato in forma numerica, a cui è associata una stringa di caratteri in “linguaggio umano” (come in FTP)

Graphical user interface, text, application

Description automatically generated

Opzioni:

* Server: specifica il tipo di server che sta rispondendo
* Content-Type: indica il tipo di file allegato con la risposta

Dopo si inserisce una riga vuota, dopodiché c’è il contenuto del file richiesto (body).

Versione 1.0:

semplice (usa connessioni non permanenti: una volta ricevuta la risposta per una richiesta la connessione viene chiusa, evita quindi di intasare il server con tante connessioni aperte) ma poco efficiente (proprio perché ogni volta devo riaprire la connessione), perché per ricevere un file devo aspettare 2 round-trip time.

Versione 1.1:

usa connessioni permanenti, che rimangono aperte fino allo scadere di una time-out, che inizia ad essere conteggiato quando i due host hanno smesso di comunicare.

Richiesta in pipeline: una volta aperta la connessione con il server, fatta una richiesta e ricevuta una risposta, il client può mandare una richiesta dopo l’altra, senza aspettare ogni volta una risposta dal server, ciò permette di “risparmiare” molti round-trip time. Il server dovrà rispondere alle richieste nell’ordine in cui sono state mandate. Questo appena descritto è proprio un limite della pipe-line e della versione 1.0:

se la prima richiesta richiede il download di un file di grandi dimensioni, il client dovrà aspettare di scaricare tutto il primo file, prima di poter passare a “visionare” le risposte seguenti.

Versione 2.0:

prevede l’utilizzo di un sistema di demultiplexing attraverso identificatori che permettono di associare ad ogni richiesta una risposta avente stesso identificatore, togliendo quindi il vincolo di rispettare l’ordine richiesta-risposta.

Meccanismo dei cookie:

come abbiamo già detto HTTP è stateless (il server, una volta soddisfatta una richiesta e mandata la risposta, si dimentica della richiesta), ciò causa delle problematiche in ambito di autenticazione.

Esempio: immaginiamo di voler accedere a un sito per l’accesso al quale bisogna autenticarsi tramite username e password, per come l’abbiamo visto, http ci farebbe reinserire nome utente e password ogni volta che ricarichiamo la pagina, poi che una volta mandata e soddisfatta la richiesta, viene cancellata.

Soluzione introdotta: i cookie, nell’header della risposta ci può essere una stringa “set-cookie” che contiene nome, valore e data di scadenza del cookie (un piccolo blocco di dati contenente informazioni, come quella che ci serve in questo caso: l’informazione che l’utente ha effettuato il login e non deve rieffettuarlo), tale cookie viene assegnato al client, esso lo memorizza, e in ogni futura richiesta che manderà a quel server manderà anche il cookie, in modo da essere riconosciuto.

Due tipi principali di cookie:

di tracciamento(servono per ricostruire la sequenza delle richieste nel client), di autenticazione (ne abbiamo appena parlato qui sopra).

Proxy:

un tipo di server che si interpone nella comunicazione tra client e server (utilizzato per velocizzare l’accesso alle pagine web) e:

* Si comporta da server nei confronti del client
* Si comporta da client nei confronti del server

Il client contatta il proxy, questo contatta il server inoltrando la richiesta del client, il server risponde al proxy, il quale memorizza la risposta e la inoltra al client. In questo modo, se il client richiedesse di nuovo un file già chiesto precedentemente, il proxy, avendo già la risposta memorizzata, glie lo invierebbe immediatamente, senza bisogno di passare per il server.

La copia del file contenuta nel proxy potrebbe non essere aggiornata rispetto a quella del server, è stato introdotto, nell’header della risposta da parte del server il parametro last modified, che specifica in GMT la data e ora dell’ultima modifica di quel file sul server. Ecco che il client potrà fare una richiesta con l’opzione “if-modified-since”, se la versione del file è aggiornata riceverà una risposta 304 “not modified”.

Il proxy può essere:

* Trasparente: intercetta la comunicazione a livello di rete 3, senza richiedere configurazione da parte del client
* Esplicito: il client decide esplicitamente di utilizzare un proxy per aumentare efficienza della comunicazione.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ inizia il magico mondo di Lagorio ~~~~~~~~~~~~~~~~~~~~~~~~~~~

Studio dei sistemi operativi.

I S.O. si basano su principi di:

* Virtualizzazione
* Concorrenza
* Persistenza

Come già abbiamo visto nel corso di Architettura dei Calcolatori, il componente hardware che esegue le istruzioni in un **programma** è la CPU, attraverso le fasi di fetch, decode, execute.

La CPU contiene dei registri che altro non sono che delle piccole celle di memoria a cui è assegnato un nome, invece che un indirizzo; essi, essendo interni alla CPU sono più “immediati” da raggiungere e quindi più veloci della RAM stessa.

Durante la fase di fetch viene utilizzato un particolare registro della CPU detto EIP (Extended Instruction Pointer), controllando il contenuto di tale registro si acquisisce l’informazione da eseguire.

Nella fase di decode viene decodificata l’istruzione da eseguire appena trovata col fetch, nella fase di execute l’istruzione decodificata viene eseguita (come abbiamo già visto ad AdC).

Differenza tra RAM e ROM:

la ROM non è volatile, quindi ipoteticamente anche dopo assenza di corrente mantiene i dati contenuti al suo interno, è possibile accedervi solo in lettura.

La RAM invece è volatile, vi si accede sia in scrittura che in lettura.

Anche il sistema operativo stesso è un programma come un altro, semplicemente lo identifichiamo come il secondo programma ad essere eseguito una volta accesa la macchina. Quale è il primo? Una volta acceso il calcolatore e partita la fase di fetch (che parte dalla ROM, intuitivamente, siccome nella RAM, appena acceso il calcolatore non c’è nulla di rilevante, perché come abbiamo detto è volatile) viene avviato il BIOS, programma presente sulla scheda madre che serve appunto per far partire il sistema del computer, attraverso il BIOS viene avviato:

* Un programma salvato in memoria come il sistema operativo
* Oppure un bootloader, che ci permette di scegliere, tra i diversi sistemi operativi salvati in memoria quale avviare

Il KERNEL

La componente principale di un sistema operativo è il cosiddetto Kernel, o nucleo, altre componenti software necessarie al funzionamento di un S.O. sono l’ambiente grafico, la shell e altri “sotto-programmini” che vedremo più avanti.

Il kernel:

* funge da intermediario tra hardware e software, permettendo la comunicazioni tra i vari livelli di astrazione.
* Gestisce i processi (programmi in esecuzione) e ne garantisce efficienza e equità a livello di sfruttamento della CPU
* Gestisce e virtualizza le risorse: ad un processo x viene garantito l’accesso solo ad una area di memoria ad esso riservata, così da preservare confidenzialità e integrità dei dati

Svolge queste mansioni grazie ai:

* device driver, parti di software installabili dopo esser state scaricate dalla rete o direttamente da disco, che consentono ad un dispositivo hardware (come, ad esempio, una periferica di I/O) di comunicare con i livelli più astratti

e al

* file-system, una struttura dati che contribuisce alla gestione delle risorse stabilendo una gerarchia dei dati/file su disco e gestendone organizzazione ed accesso.

Capiamo da questa prima descrizione che la robustezza e l’affidabilità di un Kernel sono caratteristiche cruciali: un bug nel kernel potrebbe essere sfruttato da applicazioni malevole permettendo ad un certo processo di prendere il controllo del sistema, se il kernel non riesce a limitarne l’accesso alla memoria.

CPU e CORE:

più core una CPU ha, più elevate sono le prestazioni che essa riesce ad offrire, poiché un core permette di eseguire una singola istruzione e, una volta terminata si può passare alla prossima; quindi, se si avesse una CPU con un solo core si potrebbe eseguire solo un programma per volta. Attraverso il sistema operativo si riesce a creare una “illusione”, in modo da far sembrare che più programmi alla volta siano in esecuzione allo stesso tempo (virtualizzazione). (esempio se avessimo un solo processore da 8 core riusciremmo a far “girare” veramente in parallelo solo 8 processi, non di più, il resto è un’illusione che noi creiamo attraverso la virtualizzazione)

Storia dei SISTEMI nei primi COMPUTER

La scelta del processo da mandare in esecuzione era fatta da un operatore umano che, manualmente, inseriva le schede perforate nella macchina.

Uno dei primi, se non il primo, supporto di memorizzazione dei computer, erano fatte di cartone, i fori sulla loro superficie rappresentavano dati alfanumerici.

In questo contesto non c’era differenza tra il codice di un qualsiasi programma e quello del sistema operativo, per evitare i problemi che questo poteva causare fu necessario in seguito prevedere due modalità di esecuzione distinte: eseguire istruzioni del S.O oppure eseguire codice utente.

Viene introdotto così l’utilizzo delle system call, operazioni fatte dall’applicazione al sistema, che prevedono un cambio del livello di privilegio di esecuzione, ciò suppone anche un controllo, in modo da evitare operazioni illecite da parte dei processi.

Implementate attraverso il meccanismo di trap/interrupt studiato ad AdC. Vengono utilizzati interrupt handler differenti che corrispondono a codici differenti, così da assegnare diversi livelli di privilegio/priorità ai vari dispositivi. Se ci fossero due interrupt relative allo stesso oggetto e durante la gestione della prima arrivasse la seconda, bisognerebbe aspettare di terminare la gestione della prima per poi passare alla seconda.

Con il termine multiprogrammazione si intende una tecnica utilizzata nei sistemi operativi per consentire a più programmi di essere eseguiti “contemporaneamente” su un singolo processore; l’idea è quella di sfruttare i “tempi morti” della CPU, avendo più processi in memoria e alternando l’uso della CPU tra i vari processi.

**UNIX**

Nasce come sistema operativo negli anni ’60, la versione 4 è scritta in C (non completamente, ma buona parte, il resto in Assembler) essendo il suo codice scritto in linguaggio ad alto livello, Unix viene visto come primo sistema adatto ad essere portato su macchine diverse. Alla fine degli anni ’80 si iniziano a creare standard per interoperabilità tra sistemi diversi, i principali sono POSIX e Single UNIX.

Il sistema che utilizzeremo in questo corso è Linux, un “derivato” di Unix, considerato UNIX-Like. Non è POSIX-compliant al 100% ma quasi, viene detto POSIX-oriented.

Non aderisce del tutto allo standard POSIX

SHELL

Con il termine shell si indica una interfaccia utente a linea di comando, può essere utilizzata sia inserendo i comandi “uno per volta” usando la linea di comando (in modalità interattiva), sia attraverso script.

Il nome (che in inglese significa “guscio”) deriva dal fatto che la shell agisce proprio come una sorta di “protezione”/involucro per il kernel, rendendolo accessibile agli utenti attraverso un’interfaccia di utilizzo.

La shell a cui noi faremo riferimento è Bash (Bourne-Again Shell).

Per utilizzare la shell serve un terminale.

Il terminale (fisico) è una evoluzione della telescrivente (TeleTYpe), un dispositivo elettromeccanico che veniva usato per trasmettere messaggi di testo attraverso la rete telegrafica.

Per questo ancora adesso anche i terminali virtuali vengono designati con l’acronimo TTY

Nel nostro contesto il terminale è una applicazione che fornisce l’interfaccia grafica per la shell (che eseguirà i comandi e fornirà gli output). Nelle versioni recenti di Ubuntu tty1 è il login manager, tty2 l’ambiente grafico, le altre consolle virtuali (fino a tty6) non hanno un utilizzo specifico.

(dev/tty è sinonimo del terminale associato a un processo)

Si può passare da una consolle virtuale all’altra utilizzando il comando chvt (che sta per Change Virtual Terminal) seguito dal numero del terminale che stiamo scegliendo.

Sugli ambienti desktop (come GNOME) si usa un emulatore di terminale, un programma che emula un terminale testuale, il collegamento tra terminale reale e programma avviene tramite pty (pseudo-terminale). Il metodo di funzionamento di questi pty è “master-slave”

* Quando apriamo un terminale grafico, per esempio attraverso Gnome lanciando uno Gnome-terminal, viene creato il lato master dello pseudo terminale
* Il lato slave fornisce una interfaccia che si comporta esattamente come un classico terminale

Quando un processo necessita di uno pseudo terminale, apre /dev/ptmx e riceve un file descriptor per il lato Master dello pseudo terminale (PTM) e un “dispositivo” pseudo terminale slave viene creato in /dev/pts. Il processo poi può aprire il corrispondente slave e interagire con esso come fosse un terminale vero e proprio.

Un file descriptor è un numero non negativo (già visto ad AdC) che corrisponde ad un file aperto da un processo. Ogni volta che un processo vuole utilizzare un file lo deve aprire attraverso la systemcall open(), come sappiamo, questa systemcall “crea” effettivamente un file descriptor e lo ritorna (se va a buon fine viene sempre restituito il più piccolo file descriptor disponibile, in caso di errore invece ritorna -1), questo numero potrà essere preso come parametro dalle altre funzioni di sistema di gestione dei file (read e write).

Ogni processo utilizza 3 fd:

* 0 standard input (stdin)
* 1 standard output (stdout)
* 2 standard error (stderr)

Differenziando input, output ed errori è possibile stampare e leggere da sorgenti diverse.

Ad esempio si può stampare l’output di un programma sul file, e i suoi errori sulla finestra del rispettivo pseudoterminale.

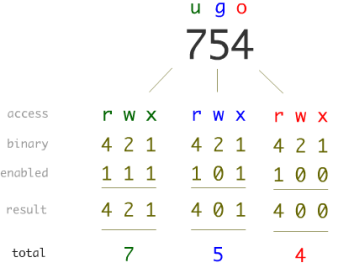
Permessi sui file: viene utilizzato il così detto concetto ugo.

User: permessi all’utente

Group: permessi al gruppo

Others: permessi a tutti gli altri.

Come già forse sappiamo ci sono i permessi di read, write ed execute, ognuno di essi è costituito da una flag rappresentata da 1 bit:



Quindi se volessimo dare solo il permesso di lettura all’utente e nessun altro permesso agli altri avremmo 400. L’uso più comune che se ne fa è cambiare i permessi di un file attraverso il comando chmod.

Tali valori vengono usati anche nella systemcall open() ad esempio quando si usa la open per creare un file (se si dà un pathname relativo ad un file non esistente come primo parametro), ma con due zeri davanti, perché in ottale, quindi volessimo creare un file e dare solo i permessi di lettura all’user, faremmo

open(“pathname”, O\_CREAT, 00400);

pathname flags mode

mode vale solo per le prossime aperture: quindi possiamo scrivere in un file a sola lettura, ma poi quando lo apriremo di nuovo non ci sarà più permesso di scrivere.

Un processo non può interagire direttamente con il Hardware, per fare ciò entra in gioco il nostro kernel come intermediario (abbiamo già visto che il ruolo del kernel è quello di mettersi “in mezzo” tra utente/programmi ed Hardware), a questo scopo vengono utilizzate quelle che si chiamano systemcall (chiamate di sistema), una syscall rappresenta una interfaccia tra un programma utente e il kernel, quando un programma utente invoca una syscall avviene una “transizione” da utente a kernel, il quale esegue la funzionalità e restituisce il controllo al programma utente. Per effettuare una vera e propria syscall si dovrebbe utilizzare assembly, linguaggio che potrebbe risultare di difficile comprensione in quanto di basso livello; per questo vengono introdotte le così dette funzioni wrapper: funzioni ad alto livello che consentono di effettuare systemcall in modo più conveniente e semplificato, la libreria C standard ne offre diverse. Come dice appunto il nome, una funzione wrapper “incapsula” una syscall, offrendo una sintassi e semantica più intuitive, con un livello di astrazione più alto. Dopo essere state chiamate le wrapper preparano gli argomenti per le syscall e dopo aver “fatto fare tutto il lavoro” alla chiamata di sistema sottostante, controllano il risultato, in caso di errore impostano errno (che indica il motivo per cui la chiamata non è andata a buon fine) e restituiscono un codice di errore (di solito -1), altrimenti restituiscono il risultato al chiamante.

Si può dire che la open() sia una wrapper per la systemcall assembly sottostante e che fopen() sia una wrapper per la open(), che semplifica l’apertura di un file (togliendo le “complicazioni” dei file descriptor).

A ogni file aperto è associato un file offset che indica in che posizione leggere/scrivere all’interno del file (ciò viene usato solo per i file regolari: non ha senso andare avanti e indietro su un socket, un device, eccetera).

Dopo la open l’offset è 0, viene spostato da read e write o da una systemcall che abbiamo già visto a AdC fatta apposta per questo: lseek. Attraverso questa funzione l’offset può essere spostato anche oltre la fine del file, ma questo non cambia la dimensione del file.

off\_t lseek(int fd, off\_t offset, int whence); dove whence può essere:

SEEK\_SET(0), SEEK\_CUR(1) o SEEK\_END(2). viene calcolato dalla fine del file

Offset viene calcolato dall’inizio del file viene calcolato dalla posizione corrente del puntatore

Su Linux esiste un file-system virtuale, chiamato /proc, esso viene utilizzato da sistema per fornire all’utente informazioni sul sistema stesso, come informazioni sulla CPU (tipo, utilizzo della memoria, ecc.), informazioni su hardware (come RAM, ROM), informazioni su processi in esecuzione (inclusi i relativi ID), ecc.

Per ogni processo del sistema esiste una sub-directory in /proc. Ogni processo è identificato da un numero chiamato PID (process identifier), il PID viene memorizzato in int/long a seconda del sistema.

Digitando echo $$ possiamo richiedere di vedere il PIRD della shell attualmente in uso:

A screenshot of a computer

Description automatically generated with medium confidence

tre finestre del terminale diverse, tre PID diversi

Usando il comando ls (per listare tutti i file presenti in una directory) trovandoci all’interno di /proc/$$/fd è possibile vedere i file descriptor del processo in esecuzione:

Graphical user interface, text, application

Description automatically generated

Ogni tipo di filesystem ha dei limiti, per cui, per esempio, la lunghezza massima di un nome file può variare a seconda della directory considerata, per conoscere questi limiti si può usare getconf e fpathconf/pathconf

Indirizzamento output/input:

usando > si può indirizzare l’output su un file/terminale di propria scelta, ad esempio scrivendo echo > /dev/pts/… è possibile scrivere del testo su un terminale e vederlo stampato su un altro.

Anche il comando cat, oltre a echo, può essere usato per indirizzare l’output su un pts diverso:

cat > /dev/pts…

attenzione:

usando >fname reindirizzo l’output sul file fname, sovrascrivendone il contenuto

utilizzando invece >>fname reindirizzo l’output sul file fname aggiungendolo al contenuto esistente del file, quindi non sovrascrivo.

allo stesso modo possiamo utilizzarlo per prendere l’input da un’altra shell, diversa da quella attualmente in uso:

cat < /dev/pts/…

Esempio: lanciare comando da finestra con pts/0 per leggere input da finestra con pts/1 e stamparlo su finestra con pts/2

Text

Description automatically generated

Graphical user interface, text

Description automatically generatedsto scrivendo su questa finestra, ma ciò che scrivo appare su quest’altra:

Text

Description automatically generated

Anche lo standard error, oltre agli input e output, può essere indirizzato:

Text

Description automatically generated

creiamo un file pippo

con questo comando chiediamo di listare il contenuto del file ‘questo\_file\_non\_esiste’ che appunto non esiste, e quindi ci viene mostrato a schermo l’errore corrispondente, mentre su pippo verrà stampato l’output (cioè nulla)

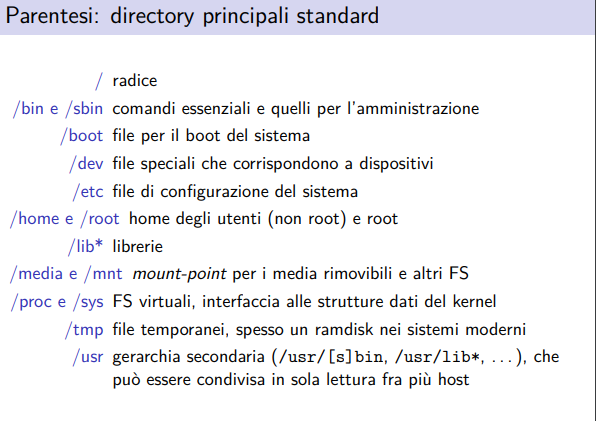
con questo comando invece chiediamo di listare il contenuto di quel file, chiedendo però di indirizzare l’errore sul file pippo (infatti stiamo usando il 2: file descriptor che si riferisce a stderr), aprendo il file vediamo lo stesso messaggio d’errore che prima era stato stampato a schermo:

a proposito di reindirizzamento, due file speciali che possono essere utili in diverse situazioni (per esempio per scartare output o generare degli stream di dati specifici):

/dev/null e /dev/zero

Text

Description automatically generated

directory principali utili: root può eseguire/leggere/scrivere qualsiasi file (quindi anche file che, vedendo da home ci dicono di non avere permessi di esecuzione/scrittura/lettura), proprio per questo però non è consigliato abusarne, potremmo modificare file che non dovremmo toccare e creare errori difficilmente riparabili.

Esempio in WSL:



Così facendo apro una nuova istanza da superuser root, in seguito mi posso spostare nella directory di mia preferenza, partendo dalla home:



Dove visualizzerò tutto allo stesso modo dell’utente normale, ma potrò eseguire/modificare qualsiasi file, senza cambiarne manualmente ogni volta i permessi.

Per detectare errori di uso della memoria di cui forse non ci accorgiamo sempre si può usare Valgrind, o in alternativa l’ address sanitizer: quest’ultimo va incluso al momento della compilazione:

esempio: gcc -fsanitize=address -g nomefile.c

(il vantaggio di Valgrind è che può essere usato su qualsiasi file binario, senza bisogno di ricompilare per “trovare gli errori” come con il sanitizer.)

Disciplina di linea

In alcune modalità il kernel può interporsi tra terminale e processo e gestire ciò che viene scritto:

scrivendo qualcosa su terminale non si ha nessun effetto, è solo dopo aver premuto invio che si vedono gli effetti del comando digitato: questa è la modalità canonica (standard, di default), tuttavia ci sono casi in cui un programma potrebbe richiedere una interazione più diretta senza buffering o la necessità di premere invio per ogni comando, in questo caso si può disattivare la modalità canonica, usando il comando:

stty -icanon

si è quindi passati alla modalità non canonica.

Per tornare alla canonica si può usare stty icanon o stty sane.

Sequenze di escape

Sono sequenze di caratteri che iniziano con Esc (ASCII 27) che servono per mandare comandi al terminale. Ad esempio scrivendo:

echo -e '\x1b[38;5;123mciao'

stampiamo la parola “ciao” sul terminale colorata di azzurro, come funziona:

* \x1b[ è il control sequence introducer (si usa per iniziare la sequenza di escape)
* 38;5;123m è una sequenza di escape che indica il colore del testo: 38 indica la selezione del colore del testo, 5 indica la modalità di colore a 256 bit e 123 è il codice del colore specifico (azzurro), si chiude con m perché si sta usando la funzione SGR (Select Graphics Rendition), che viene usato appunto per cambiare il modo in cui “si presenta” il contenuto da terminale, oltre il colore si può cambiare lo stile (grassetto, italico, sottolineato, ecc.) e diverse altre cose.

Comandi nella shell

* Un comando è una semplice parola o, in molti casi, una sequenza di parole separate da spazio vuoto (blank) (in caso il comando prenda uno o più argomenti):

Ad esempio, ls è un comando (una sola parola), ma anche echo ciao è un comando.

cat < /dev/pts/1 > /dev/pts/2

anche questo è un comando.

* Una sequenza di comandi, separati da **|** o da **|&** viene definita pipeline

cat numeri.txt | sort | uniq è una pipeline (i comandi di questa pipeline sono cat numeri.txt, attraverso il quale possiamo vedere sul display il contenuto del file numeri.txt, sort che ordina questo contenuto, secondo un certo criterio, ad esempio lessicografico, e uniq che elimina le righe duplicate)

* Una lista è una sequenza di pipeline, separate da **;** o da **&** oppure da **&&** o anche **||**, opzionalmente terminata da **;** o **&** o newline

Non è raro vederle racchiuse tra ( e ) o { e } per applicare una unica redirezione a tutti i comandi della lista

Ogni commando restituisce un exit status, che finisce in $?, per convenzione 0 OK, non-0 errore

Attenzione: nei contesti “booleani” 0🡪true, non-0🡪false (sì, è “al contrario” rispetto a quello che potremmo pensare basandoci sui linguaggi di programmazione, ma ha un senso

Job, Job Control e segnali

Con il termine “job” si definisce un gruppo di processi. Per ogni pipeline che inseriamo nella shell viene creato un job.

I job sono raggruppati in sessioni, una per terminale.

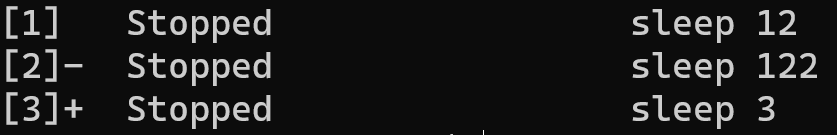
Con job control si indica il meccanismo che permette di gestire più lavori con uno stesso terminale.

Ogni terminale può avere:

* Un solo job in foreground che può sia leggere dal terminale che ricevere segnali
* Tanti job in background

Alcune combinazioni di tasti in Linux sono dette di controllo, esse servono a mandare segnali alla shell, la quale a sua volta lo manda a tutti i job che gestisce, ad esempio se l’utente usa la combinazione ctrl-C (mostrato a terminale come ^C) viene inviato un segnale SIGINT(2) al job in foreground, il comportamento di default in risposta a questo segnale è la terminazione (di tutti i processi del job).

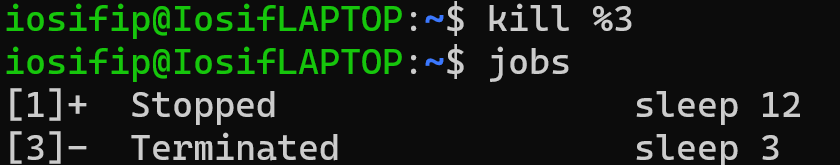
Usando ^Z possiamo sospendere un processo, il quale potrà essere poi ripreso in background con bg + ID del job e in foreground con fg + ID del job:

lanciando il comando jobs, ci verranno listati tutti i job attuali (che siano in esecuzione o stoppati), con il loro id 

per riprendere il primo in foreground basta fare ad esempio: fg 1, per riprendere il secondo in back-ground bg 2, eccetera.

Oltre alle combinazioni di controllo esistono anche comandi tramite cui si possono mandare segnali, ad esempio il comando kill, con il quale possiamo inviare SIGHUP, segnale che viene inviato a un processo quando il terminale ad esso associato è chiuso e quindi per qualche ragione si è interrotta la connessione, SIGTERM(5), lo stesso SIGINT(2) O SIGKILL(9), quest’ultimo, a differenza del SIGTERM(5) termina forzatamente il processo, senza dare la possibilità di eseguire azioni di pulizia o altro. L’exit-status di un processo terminato per un segnale s è (s+128), quindi ad esempio SIGSEGV(11)🡪exit-status sarà 139.

Per usare kill si mette % prima dell’ID di nostro interesse:

 il processo è stato terminato con kill.

(Quando apriamo l’emulatore di terminale, viene creato un pty per la parte master, viene creato un nuovo processo p che apre lo slave ed esegue la shell, quando chiudiamo la finestra del terminale il segnale SIGHUP viene mandato alla shell dal kernel, la shell lo manda a tutti i job che sta gestendo e i processi vengono terminati).

Scripting

Abbiamo già visto che la shell può essere usata sia in modalità interattiva sia attraverso script.

Nel nostro caso useremo file con l’estensione .sh, per provarli con Bash.

Dentro a uno script mettiamo in modo organizzato tutti i comandi che vogliamo far eseguire ad una shell. Oltre ai concetti appena visti (comandi, pipeline, ecc) possiamo usare anche funzioni, definendole come segue:

function name {cmd-list}

oltre a questo, due concetti importanti da sfruttare a nostro vantaggio dentro un file .sh possono essere i cicli e costrutti condizionali (già ampiamente visti in diversi linguaggi di programmazione) nonostante questi possano anche essere usati da linea di comando.

File eseguibili e processi.

Il codice di un programma è contenuto all’interno di uno o più file eseguibili.

Un file eseguibile contiene:

* Codice macchina
* Dati e dati di sola lettura (contenuto effettivo del file)
* Metadati (tutta una serie di informazioni sul file e sulle sue caratteristiche, come la versione del programma, l’autore, data di compilazione, architettura, eccetera, che possono quindi essere accedute senza aprire ed esaminare il contenuto effettivo del file, e per questo molto utili a utenti e sistemi operativi)

Per recuperare i metadati di un file esistono varie funzioni POSIX, come stat(), lstat() e fstat(), quest’ultima restituisce informazioni sui metadati di un file che viene identificato da un file descriptor (quindi viene passato come primo argomento il fd associato al file di cui vogliamo recuperare le informazioni ed esse vengono scritte dentro all’area di memoria puntata dal puntatore passato come secondo parametro: tale area di memoria è una struct stat, il cui contenuto viene aggiornato quando si chiama una di queste funzioni relative ad un file).

A screenshot of a computer code

Description automatically generated with low confidence

Queste informazioni riguardo ad un file si possono recuperare anche attraverso il comando stat da terminale come segue:

stat nomefile

A picture containing text, screenshot, font

Description automatically generated

Le altre due funzioni stat() ed lstat() hanno un funzionamento molto simile, il secondo parametro è lo stesso già descritto, mentre come primo parametro prendono un pathname (percorso relativo/assoluto del file di cui si vogliono i metadati), la differenza tra stat e lstat() sta nel diverso modo di utilizzare i link simbolici, spieghiamo prima brevemente cosa sono questi e poi la differenza:

un link simbolico è un tipo speciale di file che punta ad un altro file, ma è gestito in modo indipendente: cancellare il link simbolico non cancella il file di destinazione a cui esso punta. Per creare un link simbolico su Linux si usa questa sintassi:

ln -s percorsoDelLinkSimbolico percorsoAlQualeIlLinkPunta.

Gcc e clang sono programmi che gestiscono il processo di compilazione dei file, lanciando preprocessore, compilatore, assemblatore e linker:

A picture containing clipart, font, graphics, graphic design

Description automatically generated

* compilatori e assemblatori producono il file oggetto/rilocabile (quello con l’estensione .o per capirci)
* linker (statici o dinamici) mettono “insieme i pezzi”, cioè partendo da un file rilocabile lo collegano ad alte librerie e risorse esterne per creare un file eseguibile completo.

come abbiamo appena detto il linking può essere di due tipi:

* + linking statico:

vengono di fatto creati programmi autocontentuti copiando i “pezzi” di librerie che servono all’interno del programma stesso🡪ciò comporta che l’eseguibile funzionerà su una varietà di macchine diverse perché c’è già “tutto pronto”

* + linking dinamico: (di default sui sistemi moderni che utilizziamo anche noi)

viene preparato l’eseguibile “annotando” le dipendenze esterne, ma i “pezzi” mancanti vengono messi assieme solo a tempo di esecuzione🡪questo comporta un risparmio di spazio ma portare l’eseguibile da una macchina all’altra può diventare problematico.

Per spiegare l’esecuzione di programmi un concetto importante da introdurre è quello di spazio di indirizzamento (address space), esso è una astrazione della memoria fisica che il programma in esecuzione (il processo in questione) “vede”. Quindi un programma in esecuzione non ha accesso a tutta la memoria del sistema, ma solo al suo address space. Quello che succede in prima battuta quando si vuol eseguire un programma è la mappatura:

vengono mappati in memoria codice (sorgente) e dati (utilizzati e non) del programma, non viene caricato quindi tutto il programma in memoria per eseguirlo, nella page-table (rivedere paginazione e segmentazione nella sezione di virtualizzazione di AdC) vengono marcate certe pagine come appartenenti al file in questione. Dopo la mappatura può iniziare l’esecuzione: si arriva all’entry-point del programma (primo punto di esecuzione) e la pagina corrispondente non sarà in RAM (non è ancora stata inserita, è avvenuta solo la mappatura), quindi ci sarà un page-fault (si prova ad accedere ad una pagina di memoria virtuale non ancora caricata in memoria fisica RAM)🡪viene quindi caricata in RAM la pagina in questione, riavviata l’istruzione che aveva generato il page-fault e si proseguirà con l’esecuzione.

Quindi le varie pagine verranno caricate in memoria centrale quando servono (“on-demand”), per “far girare” dunque un programma che ad esempio nel suo complesso pesa anche 4 GB, si può usare una RAM da 1 GB, perché non va caricato tutto in RAM prima di eseguirlo: in un certo momento ci saranno caricate in memoria delle pagine relative alle istruzioni da svolgere in quel momento, in momenti successivi ci potranno essere altre pagine relative ad altre porzioni di codice e dati che il programma sta gestendo in quel momento.

Per far “girare” i programmi non ci basta però solo mappare codice e dati:

nello spazio di indirizzamento del processo ci saranno anche

* stack: usato per memorizzare variabili locali, informazioni su chiamate di funzione (ad esempio parametri, indirizzi di ritorno, eccetera): cresce a indirizzi decresecenti: cioè quando ci “aggiungiamo roba” lui va giù
* heap: regione usata per la gestione della memoria dinamica (allocazione tramite funzioni malloc, free, etc)
* kernel (serve per forza, come abbiamo visto prima, a fare da intermediario per l’esecuzione delle systemcall).

Ritorniamo a parlare dei PID, già introdotti in precedenza, una informazione utile che aggiungiamo su di essi è che: vengono riutilizzati, quindi identificano i processi solo in uno specifico momento (ad esempio, il pid 223 non è detto che starà a identificare il processo x per sempre).

Per recuperare il PID di un certo processo si possono usare le funzioni getpid() che restituisce il PID del processo corrente e getppid() che restituisce il PID del processo genitore del processo corrente.

I processi formano un albero con radice PID=1, chiamata init su diversi sistemi e systemd su Ubuntu.

Il kernel espone le informazioni attraverso lo pseudo-filesystem /proc di cui abbiamo parlato in precedenza:

per ogni processo c’è una directory, usando /proc/[PID]/maps mostriamo lo spazio di indirizzamento di un certo processo identificato da [PID].

Ogni processo ha una directory root (radice) che viene usata per i percorsi assoluti (che iniziano con /) ed una directory di lavoro, che viene usata per i percorsi relativi, la funzione getcwd() restituisce la directory di lavoro corrente mentre attraverso chdir o fchdir posso modificare la directory di lavoro corrente.

Per gestire i processi esistono 4 syscall principali:

* fork, crea un nuovo processo (definita in unistd.h)
* \_exit, termina il processo chiamante (definita in stdlib.h)
* wait, aspetta la terminazione di un processo figlio (definita in sys/wait.h)
* execve, esegue un nuovo programma nel processo chiamante, lo spazio di indirizzamento del processo corrente viene quindi sostituito, di solito viene invocata dopo fork, questa che abbiamo preso in esame è una singola funzione facente parte di una famiglia di più funzioni (exec) tutte usate per eseguire programmi ma che differiscono per i tipi di argomenti che accettano. (definite in unistd.h)

Nel dettaglio:

la fork non prende alcun argomento e ritorna un PID, sostanzialmente chiamare la fork nel processo corrente crea un nuovo processo quasi identico al corrente, detto figlio del processo chiamante.

(ciò che cambierà tra processo figlio e processo chiamante sono sicuramente il PID e PPID. L’intero spazio di indirizzamento viene copiato. In caso di successo restituisce il PID del figlio al padre e 0 al figlio.

Il debugger gdb normalmente continua a seguire solo uno dei due processi, possiamo scegliere quale con:

set follow-fork-mode [child|parent]

per seguirli entrambi invece:

set detach-on-fork off.

exit ed \_exit servono entrambe a terminare un processo, solo che la prima permette di eseguire pulizia e terminazione del processo in modo ordinato e completo: svuota i buffer di I/O, chiama tutte le funzioni registrate tramite atexit() o on\_exit() per eseguire operazioni di pulizia personalizzate, elimina i file temporanei e chiude tutti i file aperti, mentre la seconda è una syscall (di più basso livello) che termina immediatamente il processo corrente. Sono entrambe funzioni void quindi non restituiscono alcun valore.

In entrambi i casi vengono chiuse/rilasciate le risorse del processo, il processo termina con un exit-status che possiamo recuperare dalla shell con $? ed eventuali figli orfani vengono “adottati” da init (PID=1).

Lo standard C definisce le costanti EXIT\_SUCCESS (=0) ed EXIT\_FAILURE(=1).

La wait attende un “cambio di stato” in un processo figlio, che può essere:

* terminazione
* stop/ripartenza, tramite segnali

per aspettare un figlio particolare (identificato dal suo PID) si può usare la funzione waitpid, alla quale viene passato come argomento il PID del processo figlio che ci interessa (immaginiamo di avere un processo corrente che è genitore di molteplici processi e di volere aspettare un cambio di stato relativo solo ad uno di essi). Per quanto riguarda i valori di ritorno di queste funzioni: quando vanno a buon fine ritornano il PID del processo figlio terminato/che ha cambiato stato, in condizione di errore ritornano -1.

wait(int \*wstatus);

puntatore a una variabile in cui verrà inserito lo stato del processo figlio in questione

waitpid(pid\_t pid, int \*wstatus, int options)

esistono varie opzioni per il comportamento di waitpid(ad esempio il comportamento non bloccante o l’opzione che fa aspettare fino a che il figlio non viene interrotto a seguito di un segnale, andarle a vedere tutte sul man)

PID del processo figlio che ci interessa

In aggiunta ai valori di ritorno già spiegati, la waitpid, se è stato settato il comportamento non bloccante attraverso WNOHANG, nel caso in cui uno o più figli specificati dal pid esistono ma non hanno ancora cambiato stato, restituisce 0.

Se wait va a buon fine wstatus!=0 e possiamo esaminare il contenuto di questo intero passandolo come argomento a una delle 8 macro descritte nel man (attenzione passiamo l’intero stesso come argomento non un puntatore ad esso, come nelle funzioni wait e waitpid):

WEXITEDSTATUS(wstatus) ci permette di recuperare l’exit-status del figlio, ad esempio.

Un processo terminato, ma non “aspettato” dal padre, quindi su cui non viene eseguita una wait/waitpid viene definito “zombie”. Eseguire una wait permette al sistema di rilasciare tutte le risorse associate al figlio, nello stato di “zombie” il kernel è obbligato invece a mantenere un po’ di informazioni sul processo (PID, exit-status, informazioni sull’utilizzo della memoria).

Attraverso le funzioni della famiglia exec viene avviato un programma all’interno del processo chiamante, sostituendone lo spazio di indirizzamento, ma file descriptor e PID, PPID rimangono gli stessi. Se una funzione di questa famiglia ritorna al chiamante significa che l’esecuzione del nuovo programma non è andata a buon fine: ritorna -1 (quando funziona con successo non ritorna nulla)

In generale consideriamo la execve (le altre, come abbiamo detto, sono “variazioni” di alto livello):

int execve(const char \*pathname, char \*const argv[], char \*const envp[]);

percorso dell’eseguibile array di argomenti alla linea di comando del programma array di variabili di ambiente

l’esecuzione del programma specificato nel pathname comporta che il programma in esecuzione in quel momento venga rimpiazzato dal nuovo programma, con i vari segmenti re-inizializzati: stack, heap e segmento dei dati (sia inizializzati che non).

Ancora su input/output e file-descriptor:

Su Linux esistono varie funzioni per duplicare un file descriptor:

int dup(int oldfd); duplica il file-descriptor: passato come argomento, a livello kernel entrambi i fd punteranno allo stesso file aperto, saranno usabili in modo intercambiabile.

La funzione dup restituisce il più piccolo fd disponibile

int dup2(int oldfd, int newfd); simile alla precedente ma permette anche di scegliere quale sarà il file descriptor nuovo: esempio voglio duplicare il fd 3, così che il nuovo fd che creo punterà allo stesso file a cui punta il fd 3, posso scegliere il valore da dare al mio nuovo fd, se oldfd è valido si andrà a chiudere newfd se necessario (ad esempio se come newfd scegliamo un file descriptor già esistente) e poi avverrà la duplicazione. Invece se oldfd==newfd la dup2 non fa nulla.

Una funzione importante per la comunicazione tra processi è la pipe

int pipe(int pipefd[2]); crea un canale bidirezionale anonimo, pipefd[0] è il lato per la lettura, pipefd[1] quello per la scrittura, ciò che viene scritto nella pipe finisce in un buffer del kernel:

come funziona: di solito un processo che deve comunicare con un suo processo figlio scrive dal lato [1] della pipe e il figlio leggerà dalla pipe dal lato [0].

Quando il buffer è pieno le write vengono messe in attesa o falliscono se il flag O\_NONBLOCK è abilitato, quando il buffer è vuoto le read vengono messe in attesa o falliscono se il flag O\_NONBLOCK è abilitato.

È importante chiudere i “lati” non utilizzati ad ogni processo, nel nostro esempio il processo padre, che scrive, dovrà chiudere pipefd[0] e il processo figlio, che legge, dovrà chiudere pipefd[1] perché non servono per le rispettive operazioni.

SCHEDULING

Abbiamo già introdotto (anche se non con questo nome) il concetto di time sharing: ogni processo riceve l’uso (esclusivo) della CPU per un po’, poi si passa a un altro. Questo è un concetto chiave per la virtualizzazione della CPU.

Il meccanismo che consente di cambiare processo è detto context switching, lo scheduler è quella parte di kernel che decide a quale processo dare l’uso della CPU (chi è il prossimo), secondo una certa policy. I primi problemi che sorgono sono quelli di garantire che un processo rilasci effettivamente l’uso della CPU e capire come implementare il context switch in modo efficiente, ci sono due approcci:

* cooperativo: ogni tanto i processi eseguono una syscall anche se non ne hanno bisogno (tipicamente syscall ad-hoc: yield)
* non cooperativo: il S.O. si riprende il controllo “forzatamente”, tramite un timer interrupt

l’idea è quella di salvare i registri del processo che si sta eseguendo nella memoria kernel e caricare nella CPU i registri del processo che vogliamo mandare in esecuzione (così che, siccome “a una certa sbattiamo fuori” il processo corrente, quando lo faremo riprendere si ritroverà gli stessi valori che aveva prima di essere sbattuto fuori) l’implementazione però è abbastanza complicata perché molti dettagli dipendono dal Hard Ware con cui stiamo lavorando.

Generalmente ogni processo ha un kernel stack ed un user stack.

Possiamo usare questa immagine per visualizzare meglio il concetto dell’interrupt introdotto prima:

A picture containing text, screenshot, font, receipt

Description automatically generated

Quindi c’è questo timer sulla scheda madre che viene programmato dal sistema operativo per mandare un segnale di interruzione una volta ogni tot tempo (esempio 1000 volte al secondo), quando arriva l’interrupt la CPU interrompe quello che stava facendo e passa il controllo all’interrupt handler registrato per quel particolare interrupt (il S.O. registra sé stesso come interrupt handler per tutti gli interrupt, così ogni volta che c’è un interrupt causato da questo timer il controllo passa di nuovo al S.O. e quindi ogni volta che succede questo il S.O. “rivaluta la situazione” cioè vede quale è l’ultimo processo che ha avuto l’utilizzo della CPU, quali sono gli altri processi che aspettano di ricevere l’uso e entra in gioco la politica di cui parlavamo antecedentemente per cui bisogna scegliere uno o l’altro).

Per distinguere tra i processi in esecuzione e tutti gli altri processi in un sistema viene usato il concetto di stato di un processo:

A diagram of a running schedule

Description automatically generated with low confidence

Come abbiamo già detto, avessimo una CPU da 4 core, i processi realmente in esecuzione (running) allo stesso tempo possono essere massimo 4, ma ciò non vuol dire che su quel sistema ci siano solo 4 processi, ce ne possono essere centinaia: tutti gli altri possono ritrovarsi nello stato “ready” (pronti ad essere eseguiti, lo stato di un processo cambierà da ready a running quando lo scheduler lo “butterà” su un core di esecuzione della CPU), un altro stato interessante è quello blocked, quando è che un processo è blocked? Quando per esempio ha fatto una richiesta di input/output e aspetta che venga soddisfatta. (esempio se in un programma chiedo all’utente di inserire un numero e lui non lo inserisce per diverso tempo non ha senso continuare a usare la CPU che è una risorsa preziosa per quel processo che non sta di fatto facendo niente, quando l’utente inserirà il numero allora si potrà ridare l’uso della CPU a quel processo: anche il meccanismo I/O è gestito attraverso interrupt; quando i dati sono pronti scatta un interrupt, che interrompe l’esecuzione del processo corrente sulla CPU e si va all’interrupt handler, quindi il sistema rimette in ready il processo che era in wait/blocked e poi lo scheduler, quando deciderà lo rimetterà a runnare sulla CPU)

Per decidere quale è il prossimo processo a cui dare la CPU si usano diversi algoritmi di scheduling.

Chiamiamo workload l’insieme di tutti i processi che girano in un sistema. Nel contesto dello scheduling i processi sono spesso chiamati job (quindi a differenza di ciò che abbiamo detto prima, quando indicavamo un insieme di processi come job, qua un singolo processo è chiamato job).

Facciamo delle assunzioni che poi verranno “smontate” gradualmente:

* ciascun job dura lo stesso tempo
* tutti i job arrivano allo stesso momento
* una volta iniziato un job lo si porta a termine senza interruzioni
* tutti i job usano solo la CPU, senza I/O
* il tempo di ciascun job è noto a priori

per “momento in cui arrivano” si intende momento in cui vengono messi in coda (ready).

Per stabilire quanto un algoritmo di scheduling sia “migliore” di un altro bisogna affidarci a delle metriche, la prima che andiamo ad analizzare è il turn-around time (definito come tempo di arrivo - tempo di completamento, quindi il tempo che passa da quando un job arriva a quando viene effettivamente terminato)

A picture containing text, screenshot, font, line

Description automatically generated

Per un attimo abbiamo preso per buona la seconda assunzione, ma analizziamo adesso la prima: può essere mai realistico che in un vero sistema tutti i job abbiano la stessa durata/ci mettano lo stesso tempo fino a che terminano? No, ma esaminiamo l’impatto che questo può avere sul tempo di turnaround medio:

A picture containing text, screenshot, line, font

Description automatically generated

Immaginiamo 3 job ognuno della durata di 10 e che “arrivano” tutti all’istante 0: per l’esecuzione del primo si impiegano 10 secondi, il secondo processo rimane in “ready” per tutta la durata del primo e poi viene eseguito fino alla sua terminazione per altri 10 secondi, quindi 10+10=20, similmente il terzo job 10+10+10=30 quindi avremo 10+20+30, che dobbiamo dividere per 30 per fare una media.

Ma se adesso invece avessimo un job che dura decisamente più degli altri?

A picture containing text, screenshot, display, font

Description automatically generated

Questa immagine descrive il così detto effetto “convoglio”, nonostante l’esecuzione di B e quella di C impieghino esattamente 10 il fatto che prima di loro venga eseguito A (che dura 100) viene “pagato” anche da loro: cioè non importa che la loro esecuzione sia 10 volte più ridotta, tanto B dovrà comunque aspettare finché A non sarà finito per poter iniziare e C aspettare che A e B abbiano finito.

(esempio pratico per spiegare l’effetto convoglio: cercare di fare un piccolo “task” in un supermercato mega affollato all’ora/giorno di punta; immaginiamo che il mio job sia chiedere e farmi dare 1 etto di prosciutto, peccato che davanti a me ci sia un cliente che da 10 minuti sta “svuotando” il banco alimentare e ne ha ancora per molto, anche se il mio task dura un minuto, devo comunque stare al supermercato per mezz’ora, quindi il fatto che il cliente che ci mette tanto ha un impatto su quelli che vengono dopo di lui)

Ma allora a rigor di logica se io so già da prima quanto dura ogni job, e li dispongo in quest’altro modo:

A picture containing screenshot, line, rectangle, design

Description automatically generated

(quindi prima B, poi C e poi A) andrò ad ottenere un turnaround medio minore, di fatti succede questo.

A picture containing text, font, screenshot, white

Description automatically generated

(questo algoritmo che dà precedenza ai job più corti si chiama appunto shortest-job first).

Con l’assunzione (la seconda) che tutti i job arrivano allo stesso momento SJF si rivela un ottimo algoritmo.

Se togliamo questa assunzione però non è detto che sia ugualmente buono:

A picture containing text, screenshot, font, number

Description automatically generated

Ci sarebbe poi un ulteriore caso problematico: se avessimo un numero n di processi di durata 10 e iniziassimo a eseguirli uno dopo l’altro e, mentre questa esecuzione va avanti arrivasse un processo di durata 100 e poi dopo di lui altri di durata 10, verrebbero logicamente mandati in esecuzione prima questi e poi il processo più lungo, e quindi, avendo adesso questo processo lungo nello stato “ready” e continuassero ad arrivare altri processi più corti, questo processo lungo non verrebbe mai eseguito, questo fenomeno si chiama “starvation”­­­ e bisogna cercare di evitarlo quando si pensano/scelgono gli algoritmi di scheduling perché va bene avere buone performance, ma bisogna anche prendere in considerazione la “fairness” (equità)🡪non è molto desiderabile che in un sistema un certo gruppo di processi vengano eseguiti ed un altro gruppo non venga mai eseguito:

A picture containing diagram, text, sketch, line

Description automatically generated

Una cosa che posso fare allora è “droppare” anche l’altra assunzione secondo cui non posso interrompere l’esecuzione di un job ma devo per forza portarlo a termine: infatti immaginiamo lo stesso scenario; A arriva per primo ed è molto lungo, lo iniziamo ad eseguire ma poi all’istante 10 arrivano B e C, molto più corti, allora potrei togliere l’uso della CPU ad A e darlo a B (abbiamo già imparato che si può fare), completare B e C e poi finire l’esecuzione di A:

A picture containing text, screenshot, font, number

Description automatically generated

Questo algoritmo è detto Shortest Time-To-Completion First.

Altra metrica che andiamo a considerare è il response time:



Consideriamo anche questa metrica perché nei sistemi interattivi non basta il turnaround (in un sistema batch, che quindi non ha interazioni con l’utente, invece può bastare).

Possiamo intendere il response time come un “quanto tempo passa dal momento in cui il job è ready/arrivato in coda fino a quando da effettivamente il primo segno di vita, quindi viene runnato per la prima volta”, questo è estremamente importante per i sistemi che prevedono una interazione con l’utente, proprio perché l’utente umano, dato un comando alla macchina si aspetta “un qualcosa in cambio”/un feedback in maniera abbastanza celere, non che si aspetti che finisca tutto il lavoro in poco tempo, ma almeno che faccia vedere che ha iniziato a processare (pensare ad esempio quando un device lagga perché sovraccarico e noi ci spazientiamo perché sullo schermo non succede niente anche se stiamo cliccando a raffica). Negli algoritmi visti fino ad adesso il response time non è tanto buono: avessimo n algoritmi, l’ultimo che viene mandato in esecuzione dovrebbe poter aspettare la terminazione di tutti gli altri prima di essere mandato in esecuzione.

Per avere un tempo di risposta più decente si potrebbe allocare ad ogni job un “quanto”/”fetta” di tempo (una unità di tempo, definita quanto in gergo informatico, prendendo il termine in prestito dalla fisica) per poi passare al prossimo job:

A picture containing text, screenshot, line, font

Description automatically generatedquesto algoritmo si chiama Round-Robin

Così facendo avremmo un tempo di risposta medio più basso rispetto che con gli algoritmi appena visti

A picture containing font, text, screenshot, line

Description automatically generated A black text on a white background

Description automatically generated with low confidence

Ovviamente facendo le “fette” più piccole renderemmo il tempo di risposta ancora più breve, ma questo andrebbe a creare un bel problema: se abbiamo fette più brevi per lo stesso numero di processi a parità di durata vorrà dire che dovremo fare un maggior numero di “cambi” da un processo all’altro (context switch), ma anche il context switch ha un costo (buttare fuori un processo dalla CPU e buttarne dentro un altro non è una cosa immediata, anche se veloce, e quindi fare questa cosa troppe volte non è efficiente). Quindi quanto piccole vanno fatte queste fette? Abbastanza grandi da avere comunque un tempo di esecuzione più grande di quello del context switch. (se il tempo impiegato per ogni fetta fosse paragonato al tempo che serve per un context switch staremmo sprecando metà di tutto il tempo a fare “dentro-fuori” coi processi dalla CPU).

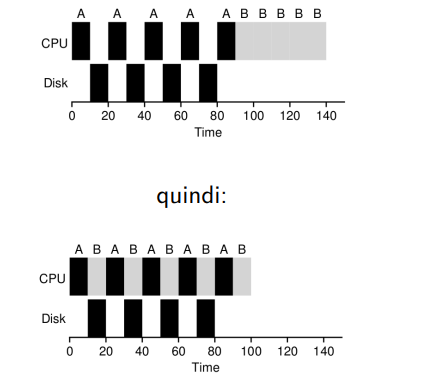
Rispetto alla metrica del turnaround time Round-Robin non si rivela un buon algoritmo (a rigor di logica cerchiamo di eseguire “ almeno per un po’ ” il prima possibile ogni job, questo però porta al fatto che il tempo di completamento si allungherà, e di conseguenza anche il turnaround.

Rimangono due assunzioni da smontare:

* tutti i job usano solo la CPU, senza I/O
* il tempo di ciascun job è noto a priori

In un reale sistema queste due cose non si verificano, almeno, non sono la norma per così dire; infatti un programma senza Input o Output non serve a molto (calcola qualcosa, ma come sappiamo cosa calcola se non ci fa vedere niente?), inoltre non sappiamo quasi mai quanto tempo servirà al programma x per completare la sua esecuzione. Il discorso dell’input/output fatto in precedenza vale anche qua: nel tempo in cui un programma sta aspettando input/output possiamo buttare nella CPU un programma che almeno “fa qualcosa”, non ha senso tenere allocata la CPU per l’attesa di I/O.

Lo schematizziamo così:



Parliamo adesso di algoritmi che si utilizzano realmente (quindi a livello più pratico rispetto alla spiegazione degli algoritmi teorici precedenti)

Il cosiddetto MLFQ (Multi-Level Feedback Queue) tenta di ottimizzare sia il turnaround che il response time.

Questo algoritmo si può implementare attraverso diverse code con priorità differenti, i processi sulla stessa coda verranno messi in schedulazione con il Round-Robin. Come si stabilisce però la priorità? Bisogna “studiare il passato per cercare di predire il futuro”, ovvero si guardano le caratteristiche storiche di un processo per predire come si comporterà in futuro: se un processo ha sempre fatto I/O ci si aspetta che lo faccia di nuovo, non è detto che succeda ma è probabile, ecco, se non succede ci si aggiusterà di conseguenza.

Priorità CPU vs I/O bound:

se un processo, storicamente, ci ha sempre “bruciato” la CPU gli diamo sempre meno priorità e quindi lo andremo ad eseguire solo quando non ci sono altri job da fare, viceversa ad un processo al quale abbiamo visto fare spesso richieste di I/O diamo grande priorità, perché si presume che abbia una interazione con l’utente e quindi dobbiamo avere un buon response time (inoltre dandogli priorità gli “facciamo fare quello che deve fare” prima di fare la richiesta e poi sappiamo che dopo che fa la richiesta possiamo mettere altro nella CPU, nel frattempo che lui aspetta i dati I/O).

Iniziamo a definire delle regole.

(p: priorità)

* Se p(A)>p(B), gira A (e non gira B)
* Se p(A)=p(B), A e B vanno in Round-Robin questo ci garantisce un buon response-time
* Un nuovo job entra con la priorità massima
* Se un job usa tutto il suo quanto di tempo, riduciamo la sua priorità

Se un job usa tutto il suo quanto di tempo e quindi non fa mai una richiesta di I/O significa che non è interattivo, ma come già spiegato prima vogliamo dare priorità a quelli interattivi.

In questo modo però un processo può “fare il furbo” e usare, per esempio il 99% del suo quanto di tempo e poi fare una richiesta di I/O totalmente inutile così che venga rilevato come “interattivo” e acquisisca priorità, ecco perché non ha senso basarsi sul concetto di “usare tutto il quanto” ma più sul “quanto viene usato il quanto” (ovvero in che percentuale), un altro problema derivante da queste prime regole è che se ogni nuovo job entra con priorità massima, quelli che erano in wait prima scendono di priorità e se continuano ad arrivare job scendono ancora e ancora, quindi si incorre di nuovo nella starvation.

“rifacciamo le regole”:

* Se p(A)>p(B), gira A (e non gira B)
* Se p(A)=p(B), A e B vanno in Round-Robin
* Un nuovo job entra con la priorità massima
* Quando un job usa la CPU tutto per un tempo fissato t a una certa priorità x, riduciamo la sua priorità
* Ogni s secondi, spostiamo tutti i job alla priorità più alta

Sistema il problema della starvation

Sistema i processi che “fanno i furbi”, non si può più, perché una volta che il processo supererà una certa soglia verrà “buttato giù”.

Rimane da vedere quanto valgono s, t, quante code usare, ecc. tutte cose che possiamo lasciar scegliere ad un admin volendo.

(che poi in casi realistici questo amministratore di sistema potrebbe dover ritarare questi valori a seconda dell’uso che si fa del sistema in quel determinato periodo).

Ci sono alternative al MLFQ, ad esempio gli scheduler proportional share invece di cercare di ottimizzare turn-around e response time si basano sull’equità: ogni processo dovrebbe avere la sua percentuale di tempo di CPU (se ho 10 processi ha senso che io dia un 10% a ciascuno).

C’è poi il CFS (completely fair scheduler) di Linux. Esso si basa sul concetto di virtual runtime/vruntime.

(nel caso più semplice vruntime è proporzionale al tempo vero). Quando bisogna scegliere un processo da mandare in esecuzione CFS sceglie quello con vruntime più piccolo, per decidere per quanto tempo lasciare un processo in esecuzione bisogna considerare l’overhead del context-switch (ovvero il costo computazionale e temporale necessario a fare il cambio di contesto, come abbiamo detto bisogna salvare i registri del processo attuale, (ri)caricare i registri del processo che si vuole (ri)mandare in esecuzione, ecc), qui entra in gioco due parametri chiamati min\_granularity e sched\_latency: se ci sono n processi pronti si manda in esecuzione quello con il vruntime più basso e lo si esegue per un tempo sched\_latency/n, se la “fetta” di tempo che voglio dare al processo (ovvero sched\_latency/n) è più piccola di min\_granularity allora darò al processo una “fetta” di tempo grande quanto min\_granularity.

Inoltre bisogna anche tenere conto della priorità di ogni processo, quindi non farò veramente sched\_latency/n ma farò

sched\_latency \* un qualche valore che indica il peso (importanza) di ogni processo/n, quindi:



Una volta che un processo è “uscito” dalla CPU, incremento in questo modo il suo vruntime:

A picture containing font, text, white, line

Description automatically generated peso

Per trovare efficientemente il minimo/aggiornare il vruntime dei processi pronti, essi vengono tenuti in un albero binario bilanciato di tipo rosso/nero

A picture containing circle, diagram, sketch, clipart

Description automatically generated

(già visto ad ASD) quindi trovare il minimo mi costerà log(n) invece che n, che per un numero abbastanza grande di processi è una differenza enorme.

Rimane un ultimo dettaglio da curare:

abbiamo detto che il CFS manda in esecuzione il processo con il vruntime più basso, ma quindi se un processo rimanesse in blocked (attesa di I/O) per un bel po’ di tempo e poi tornasse in ready, questo comporterebbe successivamente un monopolio di questo processo rispetto agli altri sulla CPU (perché avrebbe per un bel po’ di tempo sempre il vruntime minore).

Soluzione: quando un processo diventa ready (dopo essere stato blocked) gli si assegna un vruntime uguale al minimo degli altri.

Deduciamo che in realtà non è proprio “completely fair”, verso i processi che fanno frequentemente input/output (perché, ad esempio, se dessimo ½ secondo di esecuzione come fetta di tempo ad un processo, che già dopo 10 ms fa I/O perde tutto il “resto della fettina”, quindi in generale c’è equità, tranne per questo aspetto).

THREADING

Un concetto importante per la virtualizzazione è quello dei thread. Per thread si intende una sequenza virtuale di istruzioni, che possiamo vedere come una sorta di “CPU virtuale”.

Attraverso i thread possiamo avere più flussi di esecuzione in un processo: quindi possiamo avere ad esempio due unità di esecuzione (“CPU virtuali”) per un solo processo, queste unità di esecuzione hanno ognuna i propri registri ma hanno lo stesso spazio di indirizzamento.

Quindi la differenza principale tra creare un thread e creare un processo è che creando un processo si crea un nuovo spazio di indirizzamento (anche usando la fork, per creare un processo figlio del chiamante, noi copiamo lo spazio di indirizzamento per avercelo identico a quello del processo padre, ma comunque sono due spazi di indirizzamento separati: le modifiche apportate ad uno non avranno effetto su quello dell’altro), mentre creando un thread si condivide lo stesso spazio di indirizzamento.

Dal punto di vista logico ogni thread ha il suo stack (ha il suo ma potrebbe accedere anche a quello di altri thread, siccome sono sullo stesso spazio di indirizzamento), ma condivide codice e dati con altri thread dello stesso processo. Ogni stack ha anche il suo errno (sappiamo già essere quella variabile che contiene il codice d’errore in caso di fallimento di una systemcall).

A livello di protezione non è molto conveniente lavorare sullo stesso spazio di indirizzamento.

In Linux a livello di kernel “non c’è differenza” tra thread e processi, sono “cose” schedulabili in CPU.

PCB vs TCB

process control block thread control block

sono strutture dati in cui il kernel tiene le informazioni, rispettivamente, per un determinato processo o un determinato thread.

Riprendendo il discorso del context switching, specifichiamo che in realtà i job sono proprio i thread (non i processi), ossia si andranno a schedulare i thread, quando io decido di lanciare un processo viene creato un unico thread (thread principale) che verrà schedulato/deschedulato, posso creare ulteriori thread per quel processo usando funzioni apposite (proprio come posso creare nuovi processi usando le funzioni studiate fork etc) però c’è una bella differenza tra “buttare fuori” dalla CPU un thread di un processo e “mettere dentro” un altro thread dello stesso processo, oppure buttare fuori un thread e mettere dentro il thread di un altro processo. Nel primo caso il context switch costa di meno: perché non dovendo cambiare lo spazio di indirizzamento non dovrò andare a salvare registri, ecc. come visto in precedenza.

Il TLS (Thread Local Storage) è uno spazio locale a ogni thread, può servire quando volessimo avere delle sorte di variabili “globali” che non valgano però per tutti quanti i thread di un processo, ma solo per uno specifico thread, ossia che ogni thread abbia le sue “variabili globali”.

Al TLS si accede attraverso una interfaccia tipo dizionario chiave/valore.

La cosa più facile però da fare è “lasciare il lavoro” al compilatore, dichiarando una variabile globale e metterci vicino \_\_thread, questo specifica che non è una variabile globale in senso assoluto ma è globale al thread.

Vantaggio di usare i thread è usare il parallelismo:

ciò significa ad esempio che, quando noi abbiamo un processo semplice come quelli visti fino ad ora con un solo thread e basta, e dovessimo per esempio fare molti calcoli, tutti quei calcoli verranno svolti da un solo core della CPU (nel senso che, anche quando il processo viene messo in wait e poi rischedulato, magari verrà buttato su un altro core della CPU, ma sarà sempre e comunque un solo core), invece usando i thread possiamo “dividere” tutti i calcoli che il processo deve fare in più thread, che quindi verranno eseguiti allo stesso tempo in parallelo da più core.

Avessimo 10 thread su un processo e 16 core di CPU potremmo effettivamente far “andare” tutti quanti i thread realmente in parallelo su 10 core diversi.

Altro vantaggio è quello di non doversi bloccare per fare I/O come visto fin’ora:

utilizzando più thread possiamo designarne uno solo all’interazione con l’utente e altri alla computazione.

Per creare un thread si usa la funzione

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void\*(\*start\_routine)(void\*), void \*arg)

puntatore all’area dove la funzione andrà a scrivere l’id del thread parametro per specificare opzioni particolari che il 99% delle volte non useremo quindi passeremo null

puntatore a funzione che prende un void\* e restituisce un void\*

tipo più generico possibile in C, non so che tipi verranno gestiti

la pthread\_create, in caso di successo restituisce 0, altrimenti restituisce direttamente il codice di errore (senza scriverlo dentro errno).

Per uscire con un certo valore da un thread possiamo:

* Fare un return dalla funzione iniziale, oppure
* Chiamare void pthread\_exit(void \*retval)

Per aspettare la terminazione di un thread e “raccattarne” il valore di uscita usiamo

pthread\_join(pthread\_t thread, void \*\*retval);

(una sorta di wait, ma per i thread) se non la usiamo il thread diventa “zombie”

puntatore a puntatore

in alternativa possiamo usare pthread\_detach, per dichiarare che un thread può liberare automaticamente le sue risorse dopo essere terminato, senza dover essere esplicitamente atteso da pthread\_join.

Terminologia:

chiamiamo sezione critica il frammento di codice che accede a risorse condivise.

(abbiamo già visto in precedenza che avere dati condivisi tra thread può essere un vantaggio ma anche un problema)

Nel caso di più flussi di esecuzione si parla di race condition quando il risultato finale dipende da come lo scheduler schedula i thread.

(si ha quindi una race condition quando più thread eseguono una sezione critica più o meno allo stesso tempo)

Per evitare questi problemi serve sincronizzare i thread, per fare ciò si utilizzano primitive di sincronizzazione, vediamo quelle per la mutua esclusione.

Una pratica comune è introdurre dei lock, implementati dai mutex (mutual exclusion) in pthread. Un lock si dichiara con pthread\_mutex\_t e si può inizializzare con l’assegnazione della costante PTHREAD\_MUTEX\_INITIALIZER o attraverso la funzione pthread\_mutex\_init. Un Lock può essere acquisito da un thread alla volta utilizzando prhtread\_mutex\_lock e va rilasciato il prima possibile tramite pthread\_mutex\_unlock.

Un esempio dell’utilità dei mutex può essere questo codice:



Questo programma utilizza due thread, entrambi svolgono lo stesso lavoro di incrementare la variabile counter, come notiamo nel codice tale variabile è condivisa tra i due thread, lanciando il programma con argomento 2 ci aspettiamo un final value del doppio di 2 (cioè 4) e così succede, lanciandolo con 200 abbiamo final value 400 e così via, ma se aumentiamo di tanto questo numero ci vediamo restituire risultati inaspettati, questo perché, come abbiamo appena detto i due thread stanno accedendo ad una risorsa condivisa ma la cosa non è gestita nel migliore dei modi:

la nostra sezione critica è la riga in cui facciamo counter++, perché questa istruzione non è una singola istruzione in linguaggio macchina, corrisponde bensì a queste 3 distinte istruzioni eseguite in sequenza

A close-up of a computer screen

Description automatically generated with low confidence

Ciò significa che, può essere che il thread inizi a fare il lavoro (ad esempio facendo la prima mov) ma poi venga “buttato fuori” dallo scheduler e che poi l’altro thread inizi a fare il lavoro e venga buttato fuori anche lui dopo un po’ di tempo e ciò crea una discrepanza tra risultato atteso ed effettivo, noi non sappiamo come lavora lo scheduler e dobbiamo fare in modo che il nostro codice non sia influenzato dal suo comportamento. Il problema si verifica con numeri “abbastanza grandi” perché con numeri piccoli possiamo dire che il thread fa tutte le iterazioni nella fetta di tempo che gli viene dedicata inizialmente dallo scheduler, quindi il problema non “ha tempo di accadere”, ma con un alto numero di interazioni ci accorgiamo che, ogni volta che runniamo, otteniamo un risultato diverso. Questo è risolvibile attraverso i mutex lock di cui abbiamo parlato precedentemente, infatti se andiamo a “sincronizzare” proprio in corrispondenza della sezione critica possiamo far sì che i thread lavorino in parallelo nelle altre sezioni del codice mentre, quando si arriva a quella istruzione, l’ “accesso” venga garantito soltanto ad uno e solo dopo che uno ha finito, venga garantito all’altro (mutua esclusione, per l’appunto):



Creiamo un mutex e poi facciamo lock ed unlock proprio in questa parte del codice, così da avere la sincronizzazione nella sezione critica: quando un thread lavora su questa sezione, l’altro non può farlo, e viceversa.

Fatto questo, otterremo risultati attesi anche con un numero di iterazioni più alto, adesso però ci mettiamo più tempo ad ottenere un risultato rispetto a prima (è anche abbastanza ovvio: prima funzionava tutto in parallelo; un thread “se ne fregava” dell’altro mentre adesso li abbiamo sincronizzati quindi quando va uno sulla risorsa condivisa non ci va anche l’altro).

Si può risolvere però anche questo problema, usando il thread local storage di cui abbiamo parlato antecedentemente. Usiamo la keyword thread\_local per la variabile counter, così che ogni thread abbia la sua copia della variabile, in questo modo non ci sarà bisogno di “sincronizzare”, ma ogni thread si “farà i fatti suoi” e quando ha finito restituisce il suo risultato, alla fine nel main sommeremo tutti i risultati ottenuti per arrivare al risultato finale.

In linea generale il numero di lock da usare può essere determinato dal numero di strutture dati/risorse condivise a cui i thread dovranno accedere: se ho 4 array a cui devo accedere con 20 thread diversi avrà senso usare 4 lock, così che l’accesso ad ognuno di quegli array venga “protetto” da un lock.

Per quanto riguarda il numero di thread da creare anche qui non c’è una regola precisa, ma, se ad esempio il mio programma dovrà girare su una macchina con una CPU a 4 core, so che al massimo 4 thread potranno girare realmente in parallelo, quindi che senso ha crearne di più? (magari per gestire I/O), però avendo 4 core già creare 8 thread dovrebbe essere più che sufficiente.

Oltre ai pthread\_mutex appena visti esistono anche altre implementazioni dei lock, che possono tornare utili a seconda della situazione.

Parliamo degli spin lock.

A picture containing text, screenshot, font

Description automatically generated

(già spiegato dai commenti: per acquisire il lock settiamo is\_locked a 1, per rilasciarlo lo settiamo a 0)

L’attesa che vediamo nel while è una attesa attiva, è molto costosa: occupiamo un core solo per aspettare “facendo niente”, però ci evita il context switch; appena il lock si libera, riusciamo ad entrare.

Il problema di questa prima implementazione è che non funziona, perché, in una situazione in cui sono appena uscito dal while, prima di settare a 1 il campo is\_locked di mutex può succedere che lo scheduler vada a schedulare un altro thread (che stava cercando, allo stesso modo del primo, di acquisire il mutex), ora questo secondo thread setta il valore a 1, verrà quindi rischedulato l’altro thread che entra anche lui quindi ci ritroviamo due thread nella sezione critica, cosa che dovremmo evitare a tutti i costi (non viene garantita la mutua esclusione). Per evitare questo problema dovremmo trovare un modo di TESTARE il valore di is\_locked e SETTARLO in modo atomico (così da non rischiare che lo scheduler butti fuori un thread nel frattempo). Su x86 abbiamo l’istruzione XCHG, che è una operazione di scambio di due valori svolta in maniera atomica. Usando la XCHG possiamo implementare dei lock senza i problemi appena descritti.

A picture containing text, screenshot, font

Description automatically generated

Il problema di questa implementazione rimane che non c’è alcuna garanzia che venga evitata la starvation: ossia ci possono essere dei thread che vogliono ottenere il lock ma potrebbero non ottenerlo mai. Inoltre se un thread che ha già il lock vuole riacquisire il lock non riesce a farlo, ciò significa che la nostra implementazione non è ricorsiva. (esistono varianti ricorsive un po’ più complicate però)

A livello di performance va abbastanza bene solo se si aspetta poco, quella attesa attiva ci “mangia” tutto un core della CPU, quindi se dura troppo non siamo efficienti. Negli anni sono anche state proposti vari approcci ibridi, ovvero una fase di attesa attiva (se il valore cambia durante questa fase bene, se no amen), seguita da una fase in cui si sospende il thread, per non far lavorare troppo la CPU con l’attesa.

Funzioni rientranti e thread-safe:

una funzione si dice rientrante se si comporta correttamente anche quando interrotta a metà di un’esecuzione per essere nuovamente chiamata.

Una funzione si dice thread-safe se si comporta correttamente anche se eseguita da più thread contemporaneamente. Abbiamo introdotto questi due concetti insieme perché sono collegati. Uno non implica l’altro però.

Prendendo in considerazione lo standard posix però le funzioni thread-safe vengono definite rientranti rispetto ai thread, quindi di fatto rientrante da ora in poi diventa sinonimo di thread-safe.

Non tutte le funzioni POSIX sono rientranti, quelle che lo sono hanno la variante \_r.

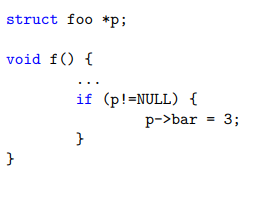
Tipicamente guardando la pagina di man riguardante una funzione possiamo vedere se è safe o meno nella sezione Attributes

A black screen with white text

Description automatically generated with low confidence

In alcuni casi le funzioni possono essere “conditionally safe” cioè se ci sono dei contesti in cui chiamarle risulta unsafe (esempio solo l’inizializzazione potrebbe essere unsafe…altri dettagli nella pagina man attributes). Del codice corretto in un mondo single-thread può avere molti problemi quando si passa a un mondo multi-thread.

Esempio:



Questo codice andrebbe più che bene se non fossimo in multi-thread, il problema è che p è una variabile globale e quindi, in un ambiente multi-thread, tra quando viene fatto il controllo e quando si accede a p, il valore di p cambia perché un altro thread può andare a mettere NULL nella variabile globale p.

Altro esempio:

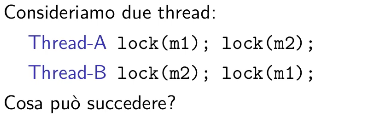
A picture containing text, screenshot, font

Description automatically generated

Questo codice in contesti multi-thread creerebbe race condition:

assumiamo che la funzione init definita dentro a cui viene inizializzato glob\_thread venga chiamata, il thread principale chiama create\_thread per creare un nuovo thread passando thread\_func come argomento. Il nuovo thread inizia ad eseguire thread\_func() mentre il thread principale prosegue con il resto del codice. Nel frattempo, thread\_func() accede a glob\_thread per leggere glob\_thread->bar, nel frattempo però il thread principale potrebbe assegnare un nuovo valore a glob\_thread (che, come vediamo, è una variabile globale). Questo errore è simile a quelli visti all’inizio, per cui bisognerà per forza implementare una sincronizzazione tra thread.

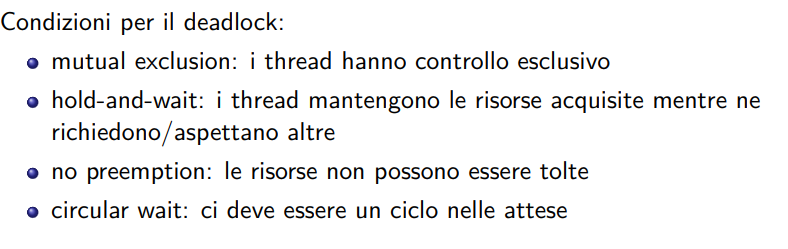
Consideriamo questa ulteriore situazione:



Ci possono essere scenari in cui “tutto va come previsto”, oppure…

Succede questo ed è tutto a posto per adesso, poi però il thread A potrebbe venire deschedulato e si passa al thread B, il quale chiede di acquisire il lock m2, e lo fa senza problemi, il problema sorge da adesso in poi:

il thread B adesso sta aspettando che si liberi il lock m1, ma il thread A sta aspettando che si liberi l’m2, lock che è tenuto da B, quindi il thread A non libererà mai m1, e quindi il thread B aspetterà all’infinito. Questa situazione si chiama deadlock.



Per prevenirlo dovrei pensare a come togliere una di queste condizioni, siccome tutte insieme causano deadlock. A “occhio” sappiamo che togliere la prima non avrebbe senso, se non viene garantita la mutua esclusione non viene garantita neanche la correttezza del programma quindi non ha neanche senso stare a parlare di thread. (come abbiamo già visto non avere la mutua esclusione causa race condition).

In sistemi in cui è possibile sapere a priori quali sono i mutex/lock che possono essere acquisiti dai thread si può fare in modo che i lock vengano sempre acquisiti tutti assieme. Un modo più semplice di risolvere il problema sarebbe imporre un ordine sui lock e acquisirli in ordine: andiamo alla radice del “problema ciclico” che abbiamo descritto all’inizio.

SICUREZZA

Il sistema operativo costituisce le fondamenta di ogni applicativo: siccome ogni altro applicativo/programma si basa sul sistema operativo, se il s.o. non è sicuro neanche gli applicativi lo saranno, e se l’hardware (su cui il s.o.) si basa non è sicuro non lo sarà neanche il s.o.

Quale è il concetto di “sicurezza” di cui stiamo parlando?

**C**onfidentiality (alcune informazioni devono poter essere visibili solo ad alcuni utenti, non a tutti)

**I**ntegrity (nessuno deve poter modificare dati di cui non ha i permessi)

**A**vailability (i dati devono essere presenti quando ci servono)

Per far sì che questi tre obiettivi vengano raggiunti, il kernel, ogni volta che viene chiamata una syscall dall’utente, deve valutare se la richiesta è sensata (quindi i parametri inseriti sono validi) e se rispetta la politica di sicurezza, espressa rispetto a subject (cioè l’entità che fa la richiesta), object (le risorse/file a cui la richiesta è relativa), accesso (lettura o scrittura o esecuzione).

L’utente, come già accennato in altre sezioni, non parla direttamente con il kernel (bensì lancia un programma, che diventa un processo, il quale viene poi gestito dal kernel e ci parla), ci deve essere un modo per riconoscere l’identità degli utenti e una associazione tra utenti e processi.

Tripla A della sicurezza

Authentication (verifica dell’identità)

Nei sistemi Unix-like gli utenti si identificano con uno username e si autenticano con una password, i metadati sulle informazioni degli utenti (username, eccetera) vengono salvati su /etc/passwd, file leggibile da tutti, mentre su etc/shadow vengono salvati gli hash delle password, questo file è invece solo accessibile dal root e i membri del gruppo shadow. Ogni utente viene identificato con un UID, analogamente ogni gruppo ha un GID. Storicamente l’UID = 0 corrisponde a root (l’amministratore di sistema).

I processi si dividono in processi privilegiati (con UID=0) e non (con UID!=0)

Authorization (il sistema decide se accettare/rifiutare le richieste)

Ci sono due approcci generali:

* Access control list: per ogni object c’è una lista delle coppie subject/access, l’ACL in Unix è codificata su 9 bit
* Capabilities: per ogni coppia object/access ci sono delle chiavi che ne permettono l’uso

I sistemi si distinguono in MAC (mandatory access control: una qualche autorità impone regole su chi può/non può accedere ad una risorsa)/DAC (discretionary access control: il proprietario di una risorsa decide chi può/non può accedere alla risorsa). I sistemi più comuni seguono il DAC, ma non al 100%, ricordiamoci ad esempio che il root può leggere qualsiasi file, quindi il proprietario di un file non può, anche volendo, impedire al root di leggerlo.

Accounting (logging: richieste vengono registrate e gestione del consumo delle risorse, utile per “risolvere” casi riguardanti avvenuti attacchi)

Principio del minimo privilegio

Principio della sicurezza informatica secondo il quale nessun processo dovrebbe mai poter accedere a più risorse del minimo necessario per svolgere i suoi compiti. In generale è importante considerare il ruolo dell’utente nello specifico momento, ad esempio, se io sono il proprietario del computer ha senso eseguire processi in qualità di super-user (modalità amministratore) se voglio installare dei driver ma non ha senso continuare a girare come amministratore anche quando non è strettamente necessario, perché avendo più “poteri” ho anche più possibilità di mettere a rischio la sicurezza del sistema.

In Unix per aumentare i privilegi si è storicamente usato su (per passare a super-user, cambiando proprio utente, e quindi autenticandosi con la password di root), oggi si può utilizzare sudo (che richiede la password dell’utente che lancia il comando) per eseguire un singolo comando in modalità super-user, per poi, una volta eseguito, tornale a stato di utente normale. Esempio classico sudo apt install…

Ogni processo ha 3 UID:

* Real UID: viene usato nella syscall kill (l’unica systemcall che guarda il real e non l’effective per ragioni di sicurezza; immaginiamoci di lanciare un binario con sudo, facendolo diventare root, se qualcosa poi andasse storto non potremmo più terminarlo, non essendo riconosciuti dal sistema come root, ecco perché la kill non guarda l’effective ma il real, dandoci la possibilità di terminare il processo)
* Effective UID: viene usato per determinare permessi di accesso a risorse (ad esempio quando si fa una system call)
* Saved UID: usato appunto per salvare l’Effective UID precedente quando un programma con il setuid viene eseguito

Quando ci logghiamo, tutti e tre coincidono. Un nuovo processo “eredita” gli id del parent (quindi quando facciamo fork avremo tutti e tre gli id uguali a quelli del padre), di solito execve non cambia gli id del processo chiamante.

Ritornando a parlare di metadati, se si fa execve ed il file eseguibile ha il bit set-userid abilitato allora Effective UID e Saved UID diventano quelli del proprietario del file (questo è in generale usato per eseguire un programma con privilegi di root: se il bit set-userid non è abilitato il programma verrà eseguito con l’Effective UID dell’utente che l’ha lanciato, mentre se il bit è abilitato e il proprietario del file eseguibile è root, allora chi lo esegue avrà privilegi che di norma sono garantiti solo al proprietario del file, e quindi privilegi di root, praticamente l’utente che esegue “diventa root” per la durata d’esecuzione del programma, cioè avrà effective UID e saved UID=0).

Tramite la funzione setuid un processo può modificare l’Effective UID, facendolo diventare uguale al Saved UID o al Real UID.

chroot è una system call che permette di modificare il significato di “/” nella risoluzione dei percorsi assoluti

(cambia la directory radice). Si possono creare le cosiddette chroot jail limitando la visibilità del file-system a una certa directory (con relative sottodirectory), questa è ancora una applicazione del principio del minimo privilegio.

FILE SYSTEM

(avevamo detto che i S.O. si basano su tre principi: la virtualizzazione, la concorrenza e la persistenza, abbiamo visto la virtualizzazione nella parte iniziale con Kernel, systemcall, i terminali, ecc. poi abbiamo visto la concorrenza con lo scheduling, i thread, i core di esecuzione ecc.)

Siamo arrivati al capitolo persistenza: cioè vorremmo che dei dati che abbiamo salvati “rimangano lì” e possano essere acceduti anche dopo che il sistema è stato chiuso/è mancata la corrente. Per raggiungere questo obiettivo si utilizzerà, come abbiamo detto nell’introduzione, la memoria ROM.

Chiamiamo dischi i dispositivi a blocchi di memoria secondaria.

Ci sono vari supporti di memoria, ad esempio:

* Dischi magnetici (hard disk/floppy disk)
* Dischi ottici (DVD, Bluray)
* Pendrive USB
* SSD (solid-state storage Drive)

Come abbiamo già detto a BdD, su ogni piatto di un disco ci sono dei cerchi concentrici detti tracce. Le informazioni sono memorizzate proprio su queste tracce, ogni traccia contiene dai 100 ai 1000 settori, tipicamente da 512 byte l’uno.

Quando i cosiddetti “dischi” (dispositivi di memoria) erano veri dischi accedere a dati “vicini” costava meno che accedere a dati “lontani” a causa del ritardo rotazionale, ovvero il tempo che ci mettevano per spostare la testina da una traccia all’altra. Questo discorso ovviamente non vale per tecnologie nuove come gli SSD, che non sono dischi, ma vale per gli hard disk, i dischi ottici, eccetera.

Altra considerazione già vista a BdD è che l’I/O su disco è ordini di grandezza più costoso dell’accesso in RAM, quindi i sistemi, una volta fatta un’operazione di lettura su disco, cercheranno di salvarne il risultato in una cache e mantenerlo lì per più tempo possibile, per non dover di nuovo fare l’operazione costosa di I/O, stessa cosa vale anche per la scrittura: quando dovremmo scrivere su disco, il sistema non andrà a scrivere direttamente su disco ma scriverà dentro la cache, e prima o poi andrà a scrivere anche su disco.

Su Linux per forzare la scrittura dei dati/metadati in cache: sync(2).

Fun fact: da qui capiamo perché il “rimuovi dispositivo”/ “espelli drive” non è solo un meme ma un avviso importante, se noi vogliamo scrivere qualcosa su un dispositivo esterno di memoria (ad esempio vogliamo copiare dei file dal nostro PC ad una pendrive) potremmo vedere i file subito copiati e pensare che l’operazione sia già stata eseguita, in realtà sono solo stati copiati nella cache di cui abbiamo parlato prima, e quindi possiamo vedere le icone, ma non è detto che la copia sia stata effettivamente completata, se scolleghiamo la periferica dal PC potremmo corrompere/perdere i dati che volevamo avere sul nostro supporto di memoria, se invece clicchiamo “espelli in modo sicuro” il sistema aspetterà finché la copia sarà effettivamente completata e poi potremo rimuovere la pendrive.

Può essere utile creare diversi “dischi logici” all’interno di un unico disco fisico vero, ad esempio se si volessero installare diversi sistemi operativi. Ciò si può ottenere attraverso il partizionamento del disco.

Ogni partizione si comporterà come un dispositivo a blocchi a sé stante. Esistono due standard usualmente utilizzati per partizionare un disco:

* Master boot record (MBR): 4 partizioni primarie + altre estese
* GUID partition table (GPT): ogni partizione è identificata da un UUID (Unique Universal Identifier)

Come abbiamo già visto, in Unix, “ogni cosa è un file”, anche i dispositivi di I/O sono visti come file (ovviamente non file regolari) speciali a caratteri o blocchi, ognuno di questi file ha un major ed un minor number. Il major number identifica il tipo di dispositivo (ad esempio: disco IDE), mentre ogni minor number identifica un singolo dispositivo di quel tipo (ad esempio se avessimo sette dischi IDE potremo avere, per ognuno di essi, sempre il major number 4 che indica il tipo di supporto, e poi ognuno di essi sarà identificato da un diverso minor number, ad esempio 1 per il primo disco, 2 per il secondo ecc.).



Un file di questo tipo può essere creato con mknod (vedi man) (bisogna essere super-user root per usare il comando) per essere utilizzato va montato con mount.

In un sistema Unix c’è un solo file system, il comando mount aggancia l’albero di file e directory di un dispositivo all’albero del file system globale, su quello che è chiamato mount-point.

A picture containing diagram, sketch, technical drawing, drawing

Description automatically generated

Nella directory /dev di Linux sono presenti i file che rappresentano i dispositivi con cui il sistema interagisce.

/dev in realtà è memorizzata in RAM e vengono creati solo i file speciali che servono, ad esempio se colleghiamo un HDD verrà creato un file speciale che lo rappresenta, ma quando smontiamo e scolleghiamo, quel file scompare da /dev.

Il kernel solleva eventi quando vengono (s)collegati dispositivi.

Esistono processi utente, come udev, che possono monitorare questi eventi

* Creare/rimuovere file speciali
* (s)caricare moduli kernel
* notificare il file-manager.

Un file-system è una struttura dati che risiede su un dispositivo a blocchi e serve per gestire dati e metadati. Ma come viene implementato su disco? (a me utente che interagisco con i file non interessa dei blocchi di memoria o del disco, mi interessa che quando creo un file che chiamo pippo e ci scrivo dentro dei dati e salvo, poi in un secondo momento possa ritrovare il mio file pippo con gli stessi dati dentro).

Formattare rispetto ad un certo formato significa preparare questa struttura dati sul disco.

Quando un processo usa una open, fra le altre cose il kernel deve:

* recuperare l’ inode (struttura su disco) corrispondente al percorso specificato come stringa
* allocare una struttura che corrisponde al file aperto che “punta” all’inode
* allocare un File Descriptor (già visto) che altro non è che un indice nella tabella dei file aperti

A picture containing text, diagram, line, plan

Description automatically generated

I settori (da 512 byte) visti prima, spesso vengono raggruppati in cluster (blocchi logici) (4 kb è una dimensione ragionevole per un cluster)

A picture containing line, measuring stick

Description automatically generated

(immaginiamo di avere un piccolo disco e di avere questi 64 blocchi da 4 kb), non possiamo usare tutto il disco per i dati, bisogna anche memorizzare i metadati di ogni file

A picture containing text, font, line, number

Description automatically generated

Ogni file viene rappresentato da un inode, ognuno di questi è contenuto nella tabella degli inode, la dimensione di questa tabella ci dà anche il numero massimo di file che possiamo creare su quel determinato disco. (se ho una tabella da 10 inode, più di 10 file non potrò memorizzare).

Quali inode sono utilizzati e quali no? Per memorizzare questa informazione si usano i bitmap: ogni bit indica se l’inode corrispondente è utilizzato o meno, è più efficiente fare così piuttosto che scrivere direttamente dentro la struttura stessa della tabella degli inode (già la tabella degli inode può essere grande, appesantirla ancora di più con un booleano usato/non usato per ogni inode non è una buona idea, se poi si dovrà andare a cercare un inode disponibile bisognerà leggere un sacco di dati, quindi cala l’efficienza, mentre usando bitmap si va a controllare direttamente lì, leggendo meno blocchi, che abbiamo detto essere una operazione costosa), continuando questo discorso esiste anche un data bitmap, che ci indica quali blocchi dati sono utilizzati e quali no. Esiste inoltre un superblocco che contiene informazioni sull’intero file-system (numero di inode, numero di blocchi dati, etc.).

Ogni inode contiene metadati relativi ad un file, ma NON il nome.

A screenshot of a computer

Description automatically generated with low confidence

Logicamente un “contenitore per file”

Pipe con nome

Un altro importante metadato riguarda quali sono i blocchi dati corrispondenti a tale file.

Due file diversi non possono condividere lo stesso blocco, immaginiamo di avere blocchi da 4 kb come descritto in precedenza. Se avessi un file in cui voglio scrivere, il sistema prima o poi mi darebbe un blocco da utilizzare, immaginiamo di scrivere solo 1 byte su questo file, starei quindi sprecando 4095 byte. In questo caso si parla di frammentazione interna. Con una operazione di deframmentazione si può rendere più veloce il sistema, ad esempio se avessimo un file molto grande (che occupa più blocchi) si potrebbero mettere tali blocchi in posizioni vicine, così, quando si va a leggere quel file non bisognerà fare tanti movimenti della testina e quindi la lettura sarà più veloce.

A picture containing diagram, plan, technical drawing, line

Description automatically generated

Questa organizzazione ottimizza nel caso di file “piccoli” (in un sistema di solito ogni singolo file non è molto pesante). Sui file molto grossi ci saranno più indirezioni.

Per i file più piccoli di “3” blocchi in questo caso sapremo direttamente quali sono i blocchi di memoria relativi a quei file.

Una directory contiene associazioni tra nome e numero di inode (inode number).

Abbiamo detto che un inode contiene tutti i metadati di un file tranne il nome, il nome andrà scritto nella directory in cui il file viene creato; se nei blocchi allocati alla directory c’è lo spazio per scrivere questo nuovo nome ce lo aggiungiamo, altrimenti bisognerà “allungare” il file della directory, cercando un nuovo blocco dati.

Abbiamo visto in precedenza i link simbolici e la creazione di essi tramite il comando ln, con questo comando si possono creare anche hard link e non solo symbolic link (che abbiamo già visto si possono creare con ln -s), creare un hard link non significa altro che aggiungere un nome ad un file che esiste già, quindi adesso potrò riferirmi a quel file anche con il nuovo nome, ma se cancellerò l’hard link, il file a cui faceva riferimento non verrà anch’esso cancellato. È possibile creare hard-link solo all’interno dello stesso file-system (la codifica del link è fatta tramite il numero di inode, e il numero dell’inode ha senso in un specifico file-system, non tra più file-system diversi) e non si possono creare hard-link per directory.

Vediamo un esempio su Linux per visualizzare meglio il concetto:

dentro alla directory /tmp abbiamo un file x

A picture containing text, screenshot, font

Description automatically generated

Se mandiamo il comando stat x da dentro la directory tmp avremo restituiti un po’ di informazioni sul file x, come il suo inode number, i “links” relativi ad esso (cioè i nomi con cui ci si può riferire, eccetera)

A picture containing text, screenshot, font

Description automatically generated

Se adesso creiamo un hard link con il comando ln



E in seguito visualizziamo di nuovo le informazioni di x vedremo come i links sono aumentati a 2

A picture containing text, screenshot, font

Description automatically generated

Ma di fatto x e y indicano la “stessa cosa”, perché se scriviamo stat y vediamo che ha esattamente lo stesso inode number che abbiamo visto su x, quindi se l’inode è lo stesso, la risorsa in memoria è la stessa

A screenshot of a computer screen

Description automatically generated with medium confidence

Per quanto riguarda link e link simbolici esistono anche due systemcall per crearli: link e symlink.

Usando il comando rm (che serve a rimuovere file/directory) su un nome di un file non è detto che quel file sparisca, ad esempio facessimo rm y staremmo cancellando il nome y, ma se quello stesso file ha anche altri nomi non ce ne siamo ancora sbarazzati (in questo caso c’è ancora il nome x), ogni volta che cancelliamo un nome del file il parametro links che vediamo qui sopra viene decrementato, è quando questo arriva a 0 che si sta “buttando via il file”. Anche se il numero di hard-link è a 0, finché ci sono FD aperti il file corrispondente viene “tenuto in vita”.

Unica cosa da aggiungere rispetto a cosa abbiamo già detto nell’altra sezione riguardo ai link simbolici: un link simbolico non deve necessariamente “puntare” a un file esistente, il contenuto del file link simbolico è un percorso, non per forza un percorso esistente.

Abbiamo già visto i permessi rwx per i file regolari, ma che senso ha parlare di permessi di lettura/scrittura/esecuzione di una directory?

r: se posso “leggere” una directory posso listarne i contenuti

w: se posso “scrivere” in una directory significa che ne posso modificare il contenuto e quindi creare/cancellare/rinominare dei file

x: se posso “eseguire” una directory posso accedervi/entrarci (eseguire tra mille virgolette, non ha senso “eseguire” una directory, ma vuol dire che posso entrarci).

Risoluzione di percorsi

Risolvere un percorso significa passare da una stringa (il percorso specificato) a un inode.

Un percorso assoluto è uno che inizia con / (viene risolto a partire dalla root), un percorso relativo non inizia con / e viene risolto a partire dalla directory corrente.

Ogni componente del percorso è separata da un /

Per ogni componente si guarda se si hanno i permessi di ricerca nella tale directory, se questo non è vero si ritorna errore (EACCESS in perror), se invece è vero si cerca dentro a tale directory e si recupera l’inode corrispondente (sempre che esista, se non si trova si ritorna errore ENOENT: no such file or directory), se esaminando l’inode corrispondente si trova che siamo su una directory (e non un file regolare), si riparte da lì: si guarda se si hanno i permessi di ricerca in questa directory e così via, se invece non è una directory ma un link simbolico si risolve il link simbolico, se il risultato non è una directory si ritorna errore ENOTDIR, se lo è si continua da lì.

Per la componente finale non si pretende che sia una directory (può essere un file regolare), inoltre come già sappiamo . serve per indicare la directory corrente e .. la directory genitore di quella corrente.

Esempio

/home/iosifip/hello.cpp

prima componente: siccome inizia con / si parte dalla root: se abbiamo i permessi di ricerca cerchiamo qua dentro il nome home, se lo troviamo ed è una directory ci entriamo dentro

~~/home~~/iosifip/hello.cpp

Seconda componente, siamo nella home e cerchiamo iosifip, se essa è una directory ci entriamo dentro

~~/home/iosifip/~~hello.cpp

Ultima componente, siamo nella directory iosifip e cerchiamo hello.cpp, esso è un file e va bene, siccome è l’ultima componente del percorso non è richiesto sia una directory. Abbiamo risolto tutto il percorso, siamo sul file desiderato.

Abbiamo visto che le operazioni su disco/blocchi di memoria possono essere molto costose a livello di tempo, cosa succederebbe se, ad esempio io creassi un file?

Bisogna allocare un inode, andare sul bitmap degli inode e “marchiare” che quel certo inode è in utilizzo, creando il file e dovendogli dare un nome devo andare ad aggiungere questo nome in una directory, il che significa di nuovo scrivere su disco, eccetera, chi mi garantisce che tutte queste operazioni di scrittura/lettura su disco avverranno una dopo l’altra senza che manchi la corrente? Potrebbe mancare la corrente subito dopo l’allocazione dell’inode. Il file-system rimarrebbe in uno stato inconsistente.

C’è un ordine migliore di altri per eseguire queste modifiche su disco?

Non esiste un ordine “del tutto più sicuro di altri”, in qualsiasi ordine si eseguano queste operazioni può succedere che manchi la corrente proprio nel momento più inopportuno e lo stato del file-system diventi inconsistente.

Esiste un comando Unix: fsck, per controllare l’integrità di un file-system, esso controlla

* che il superblocco sia “ragionevole”,
* se c’è consistenza tra i blocchi visti come liberi nella bitmap e i puntatori ai file,
* se più inode puntano a uno stesso blocco (cosa che non dovrebbe succedere perché abbiamo detto in precedenza che file diversi non dovrebbero poter condividere lo stesso blocco),
* se ci sono blocchi puntati che risultano liberi o blocchi liberi che risultano usati (si va nella bitmap e si marca come usato/non usato a seconda del caso)
* lo stato degli inode (ad esempio il tipo di un inode, che ci dice se si riferisce a un file regolare, una directory, ecc)
* il numero di link (se un inode ha 3 nel campo link count ma “spazzolando” tutto il file-system si scopre che ha 4 nomi significa che c’è inconsistenza, bisogna aggiornare il link count a 4)
* se ci sono puntatori fuori dal range dei blocchi
* che ogni directory abbia il suo . e .. e che ognuna non sia collegata più volte all’albero

ovviamente fare questi controlli di integrità impiega molto tempo, sempre di più al crescere della dimensione dei dischi, non si può quindi ad esempio fare ad ogni avvio/bootstrap, rallenterebbe troppo.

L’idea è quella di eseguire questa operazione quando ragionevolmente c’è la possibilità che si sia verificata inconsistenza. Come si fa?

Un File-System smontato “in modo pulito” non viene controllato ogni volta che viene rimontato, all’avvio se trovo il FS “pulito” non faccio niente, se invece lo trovo “sporco” faccio il controllo di integrità, per sapere se è “pulito” o “sporco” si usa una flag. (tipicamente se la corrente manca o il sistema si chiude inavvertitamente non avverrà l’operazione che “pulisce il flag” quindi lo troveremo “sporco” al prossimo avvio). Un altro approccio (che può venire in mente studiando teoria dei database o banalmente ciò che abbiamo visto sui thread) moderno è quello del Journaling, che consiste nel rendere atomiche queste operazioni: ovvero, abbiamo visto che semplicemente per creare un file vengono eseguite diverse operazioni di scrittura/lettura su disco, se tutto ciò venisse eseguito atomicamente, non potrebbe “mancare la corrente in mezzo”🡪si tratta di transazioni, vediamo tutto l’insieme di operazioni come una unità indivisibile di esecuzione e quindi o viene eseguita tutta, o fallisce, quindi lo stato del file-system rimane consistente (o avviene tutta la transazione come si deve e quindi il FS è in un nuovo stato sempre consistente, oppure si fa rollback: si riporta il FS allo stato precedente al tentativo di esecuzione della transazione, e quindi sempre consistente)