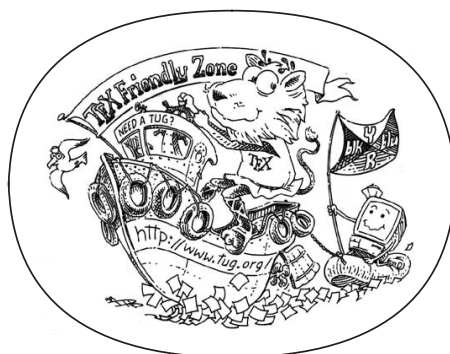


SISTEMI E TRASMISSIONE DELLE INFORMAZIONI

RICCARDO CEREGHINO



Appunti

Settembre 2019 – o.o.1classicthesis v4.6

Riccardo Cereghino : *Sistemi e trasmissione delle informazioni*, Appunti,
© Settembre 2019

INDICE

I RETI DI CALCOLATORI

1	ACCENNI AL FUNZIONAMENTO DI UN CALCOLATORE	3
1.1	Networking	3
1.1.1	TCP/UDP	3
1.1.2	Device driver	3
1.2	Trasmissione di dati	3
1.2.1	Invio di dati	3
1.2.2	Ricezione di dati	4
1.2.3	Trasmissione di dati su bus ring based	4
1.2.4	Commutazione	5
1.2.5	Dataflow	5
1.2.6	Routing	5
1.3	Internet protocol stack	5
1.3.1	Protocollo fisico	6
1.3.2	Network protocol	6
1.3.3	Trasporto	7
1.4	Application layer	10
1.4.1	System calls sui socket di tipo datagram	10
1.4.2	Esempio di scambio di datagrammi	12
1.4.3	Esempio di scambio di stream	12
1.5	Modello banda latenza	12
1.6	Protocollo a livello applicativo: DNS	13
1.6.1	Tipi di indirizzi	13
1.6.2	Risoluzione dei nomi	13
1.6.3	Algoritmo ricorsivo	14
1.6.4	Algoritmo iterativo	14
1.6.5	Algoritmo ibrido	14
1.7	Protocollo NTP	14
2	POSTA ELETTRONICA	17
2.1	SMTP	17
2.1.1	Protocollo Best Effort	17
2.1.2	Funzionamento	17
2.2	FTP	19
2.2.1	Funzionamento	19
2.2.2	FTP anonimo	19
3	HTTP	21
3.1	Richiesta	21
3.2	Risposta	21
3.2.1	Header	21
3.2.2	Body	22
3.3	Proxy	22

II SISTEMI OPERATIVI

4	INTRODUZIONE AI SISTEMI OPERATIVI	25
4.1	Virtualizzazione della CPU	25
4.2	Memoria virtualizzata	26
4.3	Periferiche	27
4.4	Memoria	28
4.4.1	Spazi di indirizzamento	28
4.4.2	Virtualizzazione	28
4.4.3	Frammentazione	29
4.4.4	Paginazione	31
4.5	System calls	33
4.5.1	fork	33
4.5.2	exit	33
4.5.3	wait	33
4.5.4	exec	33
4.5.5	open	33
4.5.6	dup	33
4.5.7	dup2	34
4.5.8	pipe	34
4.6	Shell	34

III UTILITÀ

A	APPENDIX TEST	37
----------	----------------------	-----------

ELENCO DELLE FIGURE

Figura 1.1	indirizzi	7
Figura 1.2	Three way handshake	8
Figura 4.1	Struttura base di un calcolatore	25

ELENCO DELLE TABELLE

LISTINGS

Listing 1.1	Esempio di syscall a socket	10
Listing 1.2	Esempio di syscall a bind	11
Listing 1.3	Esempio di syscall sendto	11
Listing 1.4	Esempio di syscall receivefrom	11
Listing 1.5	Esempio di syscall connect	12
Listing 1.6	Esempio di syscall listen	12
Listing 1.7	Esempio di syscall accept	12
Listing 4.1	Interazione con la CPU usando la funzione spin()	25
Listing 4.2	Esecuzione in concorrenza	26
Listing 4.3	Dimostrazione dell'utilizzo di indirizzi virtuali	26
Listing 4.4	Esecuzione in concorrenza	27
Listing 4.5	Esempio di pipe ()	34
Listing a.1	Esempio di MAKEFILE	37

ACRONYMS

Parte I

RETI DI CALCOLATORI

ACCENNI AL FUNZIONAMENTO DI UN CALCOLATORE

Un calcolatore è generalmente composto da:

- la *CPU*, esegue le istruzioni, tiene in memoria (*InstructionRegister*) l'istruzione corrente e la posizione dell'istruzione nella RAM (*P.C.*);
- la *RAM*, tiene in memoria le istruzioni da eseguire, certe sezioni sono riservate per specifiche funzioni (*stack*, *SP*, *BP*);
- dei moduli, come un disco fisso o altro;
- la *NIC*, che permette di inviare e ricevere pacchetti in rete.

1.1 NETWORKING

Vengono effettuate delle *syscall*, per eseguire operazioni necessario all'invio di dati. Per esempio la *syscall send*, copia dalla ram una certa sezione di memoria nel *NIC*, che verrà inviata in rete.

1.1.1 TCP/UDP

Il protocollo TCP garantisce l'invio del messaggio, dato che attende una risposta da parte del ricevente. Il protocollo UDP invia datagrammi, ma non si è certi se la ricezione è avvenuta.

1.1.2 Device driver

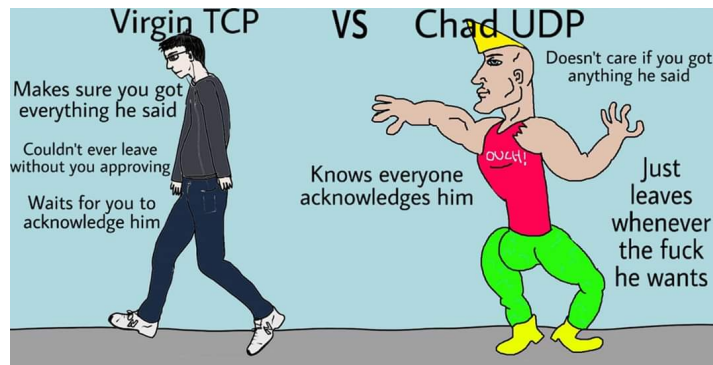
E' un modulo che gestisce le risorse fisiche dei vari moduli, tra cui *NIC*, gestiscono le interruzioni del dispositivo.

1.2 TRASMISSIONE DI DATI

1.2.1 Invio di dati

Il *DMA* presente nel *NIC* è il componente che accede alla memoria.

La maggior parte dei calcolatori include l'unità di gestione della memoria *MMU*, che permette al processore di gestire indirizzi virtuali per ottenere una maggiore efficienza, ma questo dispositivo è esclusivo del processore, motivo per cui il *DMA* del *NIC* utilizzerà indirizzi fisici.



Delle porzioni di memoria della RAM saranno destinate ad essere utilizzate per i *buffer*, ovvero delle aree dove NIC può immagazzinare i dati da inviare o da ricevere.

Al momento dell'invio di un datagramma, esso verrà salvato in un buffer, quindi inviato in rete appena possibile, quindi cancellato dalla memoria.

1.2.2 Ricezione di dati

Per la ricezione di datagrammi dei buffer devono essere sempre disponibili, cosicché NIC possa utilizzarli una volta ricevuto il pacchetto.

Il datagramma ricevuto sarà inserito dal DMA in un buffer, un interrupt nel frattempo creerà nuovi buffer per possibili nuovi messaggi, quindi il sistema legge il buffer copiandolo nel buffer dell'applicazione, direzionando l'output verso l'utente corretto, quindi la memoria viene liberata.

Il comportamento standard di un'applicazione che implementa networking implementa un meccanismo bloccante per cui l'applicazione rimane in attesa fintanto che il buffer non è stato scritto al suo interno.

1.2.3 Trasmissione di dati su bus ring based

La problematica principale del DMA di tipo *bus mastering* è la necessità di dover programmare in anticipo i buffer allocati, limitandone la capacità.

Fintanto che il DMA è impegnato in un'operazione perché in attesa del processore verranno persi i dati nel frattempo ricevuti dalla rete.

Per risolvere questo problema si implementa una struttura dati più complicata, integrando una piccola CPU all'interno della NIC (ring based).

1.2.3.1 Ring

Il ring è composto da un anello contenente più di un buffer, e due puntatori, un puntatore di inizio ed uno di fine gestiti da due pro-

cessori diversi; la CPU si occupa di aggiungere elementi all'interno del ring quindi gestirà l'indice di fine, il puntatore al primo elemento viene gestito dal processore all'interno della NIC. Si ottiene una coda di buffer che si possono aggiungere o togliere a seconda se si sta gestendo un invio o ricezione di dati.

Quindi la NIC legge i buffer in memoria e usa un flag per impostare se il buffer è libero oppure se deve essere processato.

L'unico tipo di comunicazione tra processore e DMA è la trasmissione della quantità di buffer che dovranno essere utilizzati al DMA.

1.2.4 *Commutazione*

Si rende necessario, per una questione di efficienza, un circuito fisico *R* attraverso cui trasmettere i pacchetti.

I pacchetti sono inviati come datagrammi, e nella struttura includono il destinatario e le informazioni trasmesse.

1.2.5 *Dataflow*

L' *host* compila ed invia il buffer in locale al buffer di ricezione di *R1*(router), il cui compito è di leggere il datagramma, ricavarne il destinatario, quindi di inviarlo: l' *host* può quindi cancellare il buffer.

La parte ricevente (*R2*) copia il datagramma nel suo buffer di ricezione, quindi *R1* può rimuovere dal proprio buffer il datagramma. Quindi *R2* invia il datagramma all'effettivo destinatario con un altro *store and forward*.

Ogni passaggio di trasmissione del datagramma viene chiamato *hop*.

1.2.6 *Routing*

Identifichiamo per il processo di routing l' *host*, gli endpoint della trasmissione di pacchetti e i *router*, calcolatori. Più router possono essere collegati tra loro.

1.3 INTERNET PROTOCOL STACK

1. Physical
2. Datalink
3. Network
4. Transport
5. Application

1.3.1 Protocollo fisico

L'unico protocollo fisico sopravvissuto ed in uso è il protocollo ethernet.

Ogni richiesta sarà composta da: header, payload e trailer.

Il protocollo definisce che ad ogni host venga assegnato un indirizzo *MAC*, composto da *6byte*.

L'header conterrà come dati gli indirizzi IP e le porte dei nodi comunicanti.

Il trailer è necessario per il controllo di integrità nella rete ethernet, l'algoritmo utilizzato è **CRC32**.

Il datalink utilizza le stesse modalità di funzionamento del protocollo fisico.

1.3.2 Network protocol

Il protocollo network utilizza come gestore degli indirizzi i protocolli *IPv4* e *IPv6*. *IPv4* utilizza *4byte* per il funzionamento, significa che può gestire fino a circa 4 miliardi di indirizzi, mentre *ipv6* ne può utilizzare molti di più.

I primi sviluppatori dei protocolli non avevano la possibilità di interagire direttamente con i sistemi operativi, motivo per cui si è dovuto introdurre il concetto di **porta di comunicazione**, un numero intero in *16bit*, con cui si identifica univocamente un numero di una porta ad una tipologia di applicazione.

Le porte da 0 a 1023 sono state dedicate ai protocolli internet ed alle applicazioni più comuni, un'altra serie di porte sono sempre utilizzate per il networking mentre le restanti sono le così dette porte effimere che vengono utilizzate dalle applicazioni in locale.

Quindi se la porta è la parte che un calcolatore espone ad internet, il *socket* è il corrispettivo componente nel nodo in locale, così da associare alla richiesta una macchina ed applicazione in particolare.

Sono usati i protocolli *UDP* e *text*. Il protocollo *UDP* invia datagrammi mentre il protocollo *TCP* permette di implementare una modalità di comunicazione più avanzata, di tipo *stream*, che permette di interagire con l'interfaccia di comunicazione come se fossero file.

Gli indirizzi **IP** sono utilizzati per essere associati agli indirizzi **MAC** nel network.

IL *TTL* (Time To Live) definisce da quanto tempo un pacchetto è nel network.

RFC (Remote Function Call) cerca prima di chiamare le versioni dei protocolli più recenti prima di doverne eseguire uno più vecchio.

1.3.2.1 Indirizzi

Ad ogni macchina corrisponde un IP (*IPv4* o *IPv6*)

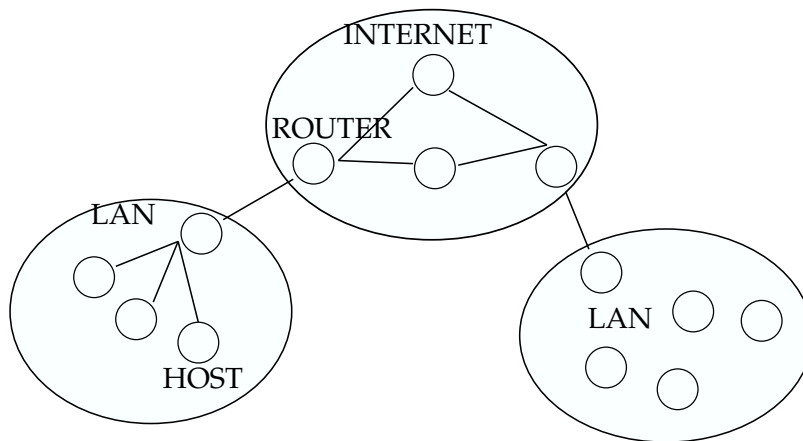


Figura 1.1: indirizzi

Un indirizzo *IPv4* in default è composto da:

- 29bit dedicati al network;
- 3bit dedicati all'host.

Per ovviare a problemi di carenza di indirizzi IP, si introduce il concetto di *NETMASK*, una parola composta da una serie di 1 ed una di 0, gli 1 rappresentano quanti bit sono dedicati al network, gli 0 per l'host.

INDIRIZZI PRIVATI Sono definiti nel protocollo *RFC 1918* dichiara che un indirizzo **IPv4** deve essere composto da 4 numeri decimali seguiti da punto, inoltre dichiara che gli indirizzi che cominciano con 10. appartengono alla *classe A*, questi indirizzi sono i cosiddetti indirizzi privati, utilizzabili solo in **LAN**.

Gli indirizzi di classe **B** cominciano da 192.168.0.0 per arrivare fino 192.168.255.255, sono sempre indirizzi privati.

Si può configurare il router per usare il **NAT** (Network Access translator), con lo scopo di assegnare due indirizzi IP allo stesso router, con lo scopo di esporne uno alla rete locale e l'altro a quella internet.

Con il NAT è possibile convertire indirizzi pubblici in indirizzi privati, direzionando i messaggi alla macchina corretta sostituendo il proprio indirizzo alla richiesta assegnando una porta libera.

1.3.3 Trasporto

Vengono utilizzati i protocolli *UDP* e *TCP* e la tecnica della *3 way handshake*.

Il trasporto comincia con l'invio di un messaggio **SYN** (accompagnato da **SEQ**), all'arrivo il server risponde con un messaggio che viene chiamato **SYN-ACK**, la **SEQ** e la **ACK**; quindi l'host invia un ulteriore messaggio chiamato **ACK**, oltre che i dati.

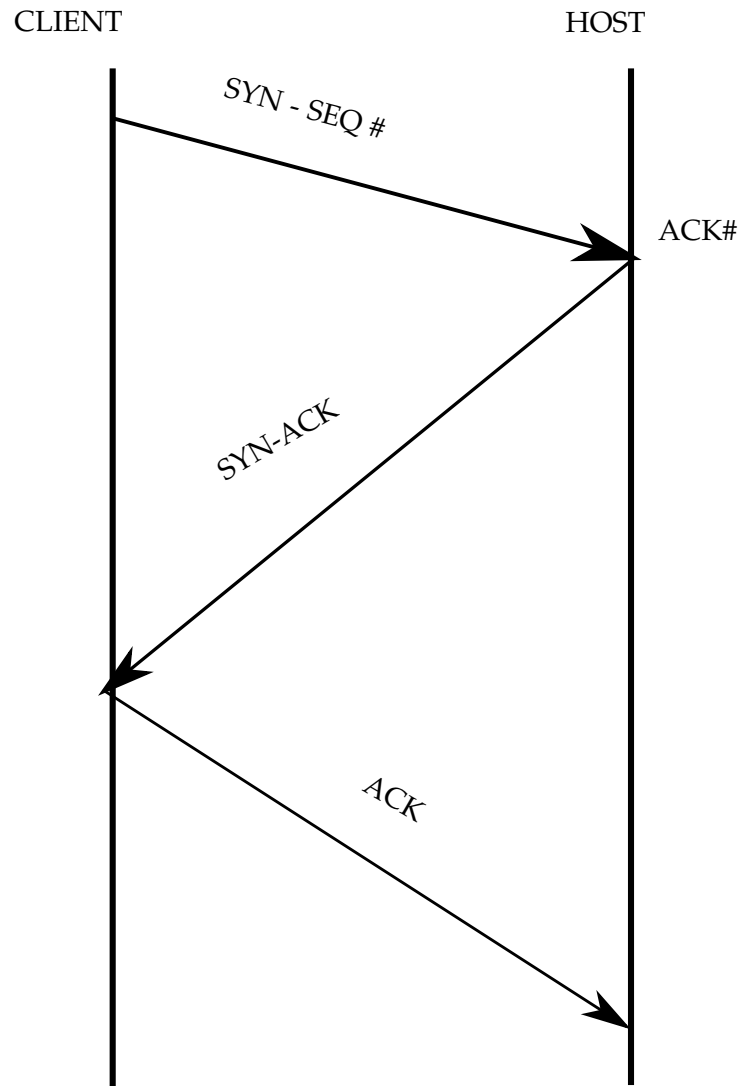


Figura 1.2: Three way handshake

L'header TCP contiene una serie di informazioni tra le quali i **FLAG** SYN e ACK oltre che la porta sorgente (**SPORT**) e di destinazione (**DPORT**).

Vengono utilizzate 2 parole, la **SEQ** e la **ACK**, permettono la di realizzare l'astrazione dello stream di byte; questi numeri per motivi di sicurezza non partono da 0, ma da un valore casuale; lo scopo della prima sequenza di passaggi è di scambiare queste informazioni.

Inoltre **SEQ** segnala anche l'inizio dello stream di byte da leggere.

Se il messaggio viene suddiviso in tanti datagrammi cumulativi, viene inviato un unico **ACK cumulativo**.

Il **timeout** è il tempo che il processo impiega ad inviare e ricevere i messaggi, dipende dalla lunghezza dei messaggi e dalla velocità dei calcolatori.

La **media mobile** dà più peso agli ultimi valori, si rende necessaria per calcolare il tempo di timeout.

$$\text{Estimated} = (1 - \alpha) \times \text{Estimated} + \alpha \text{ SampleRTT}$$

Inoltre si usa una stima della variabilità:

$$\text{Dev} = (1 - \beta) \times \text{Dev} + \beta \times |\text{SampleRTT} - \text{Estimated}|$$

I parametri α, β hanno un valore consigliato: $\alpha = \frac{1}{8}, \beta = \frac{1}{4}$.

Quindi inizialmente viene assegnato un valore fisso viene inizialmente assegnato ad *Estimated* e *Dev*, quindi il primo messaggio viene inviato, quindi nel primo scambio, in cui il client riceve il SYN-ACK, si ottiene il primo *SampleRTT*.

Quindi si possono comparare i valori *Estimated* e *Dev*, (Deviazione) ogni scambio di ACK e SIN-ACK.

Il **timeout** sarà:

$$\text{Time-out} = \text{Estimated} + 4 \times \text{Dev}$$

1.3.3.1 Flow control

Il motivo principale della perdita di messaggi è la mancanza di sincronizzazione, un tempo era molto probabile ricevere messaggi corrotti, ora la perdita di messaggi è legata alla mancanza di buffer liberi di ricezione.

Il **mittente** avrà un buffer di ricezione (TX buffer), così come il destinatario (RX buffer), nei quali i datagrammi vengono ricevuti.

Per garantire che vi siano sempre buffer di ricezione liberi sarà necessario comunicare all'interlocutore il numero di buffer allocati.

All'interno dell'header **TCP** vi sarà un campo **receive window** che contiene la grandezza dei buffer di ricezione del destinatario.

I messaggi saranno inviati o alla ricezione di un **ACK** o alla scadenza del **TIMEOUT**.

In generale, *UDP* utilizza un approccio ottimistico per cui in una comunicazione fluida è più efficiente rispetto a *TCP* che si accerta che ogni messaggio venga ricevuto correttamente.

1.3.3.2 Congestion control

Il **TCP** implementa anche un controllo di congestione oltre al *flow control*, è necessario per ridurre la perdita di messaggi sulla rete, ma non da parte del destinatario, ma dai *router* che implementano lo *store and forward*.

Quindi è il *router* che rileva la congestione di dati (mancanza di buffer) e rallenta la ricezione di dati.

Il congestion control avviene in tre fasi:

1. **slow start**: invio di pochi bytes;

2. dopo l' **ACK**, raddoppio la quantità di datagrammi inviati;
3. quindi se i dati inviati diventano troppo grandi, si torna ad un passaggio indietro e si aumentano a passi i dati inviati.

1.4 APPLICATION LAYER

L'ultimo livello di *networking* è l'implementazione a livello applicativo. Per cui si rende necessaria un interfaccia per applicazione, **API**, si userà lo standard **POSIX**; si utilizzano i **socket**.

FILE DESCRIPTOR: un processo che permette di interagire con il *file system*, permettono di stabilire un collegamento tra i buffer a livello applicativo e quelli di sistema.

Il file descriptor (**fd**) creato, per esempio, alla chiamata di *open*, sarà un **int** che rappresenta l'indice del file nell'array dei file in utilizzo.

Nel caso di errore, *fd* sarà un numero negativo.

1.4.1 System calls sui socket di tipo datagram

Ogni **system call** è bloccante, nel caso di un errore il programma si ferma. Si possono riprogrammare le *syscall* per renderle non bloccanti, con la gestione del messaggio di errore **E_WOULDBLOCK**.

1.4.1.1 Socket system call

Quindi alla *syscall* su **socket**, verrà creato un nuovo *fd*.

Listing 1.1: Esempio di syscall a socket

```
int fd = socket(int domain, int type, int protocol);
```

Dove:

- **domain**:
 - *AF_UNIX*: locale, nomi di file;
 - *AF_INET*: *IPv4*;
 - *AF_INET6*: *IPv6*;
- **type**:
 - *SOCK_DGRAM*: *IPv4*;
 - *SOCK_STREAM*: *IPv6*;
 - *SOCK_RAW*: utilizzato per saltare i passaggi di trasporto e di rete fino a *datalink*;
- **protocol**: esiste uno ed uno solo per type, è possibile lasciare il predefinito (0).

1.4.1.2 *Bind system call*

La *syscall* **bind** è necessaria per associare un indirizzo ad un **socket**.

Listing 1.2: Esempio di syscall a bind

```
int fd = bind(int fd, const struct sockaddr*addr, socklen_t
  addrlen);
```

Dove:

- **fd**: il file descriptor del *socket* riferimento;
- **addr**: alloca lo spazio necessario per salvare il numero di *bit* corretti;
- **addrlen**: la lunghezza dell'indirizzo da salvare (o del nome del file).

1.4.1.3 *Send to system call*

Listing 1.3: Esempio di syscall sendto

```
ssize_t sendto(int fdA, const void*buf, size_t len, int flags,
  const struct sockaddr*addr, socklen_t addrlen);
```

Dove:

- **fdA**: file descriptor del socket da cui inviare;
- **buf**: buffer da cui inviare;
- **len**: lunghezza del messaggio;
- **flags** ;
- **sockaddr**: indirizzo del socket (comprensivo di porta);
- **socklen**: lunghezza del socket;

1.4.1.4 *Receive from system call*

Listing 1.4: Esempio di syscall receivefrom

```
ssize_t recvfrom(int fdB, const void*buf, size_t len, int flags,
  struct sockaddr*addr, socklen_t addrlen);
```

Dove:

- **fdA**: file descriptor del socket in cui ricevere;
- **buf**: buffer nel quale scrivere il messaggio;
- **len**: lunghezza del messaggio;

- **flags:** ;
- **sockaddr:** indirizzo del socket (comprensivo di porta del sender);
- **socklen:** lunghezza del socket;

1.4.1.5 *Connect system call*

Listing 1.5: Esempio di syscall connect

```
int recvfrom(int fdA, const struct sockaddr*addr, socklen_t len);
```

1.4.1.6 *Listen system call*

Listing 1.6: Esempio di syscall listen

```
int recvfrom(int fdB, int backlog);
```

1.4.1.7 *Accept system call*

Listing 1.7: Esempio di syscall accept

```
int accept(int fdB, 2 parametri);
```

Il ritorno della funzione è un **fd**. Per ogni successo il server associa un nuovo socket.

1.4.2 *Esempio di scambio di datagrammi*

Il server ed il client creano un socket di tipo *datagram*, il server esegue un *bind*. Il client invece esegue una *syscall* di tipo **sendto**, il server utilizza **recvfrom**, etc..

1.4.3 *Esempio di scambio di stream*

Il server ed il client creano socket di tipo *stream*, il server esegue la *bind*, il client la *connect*, per ogni *send* e *receive* non è più necessario lo scambio di indirizzo. Il server deve attivare **listen** prima dell'arrivo della *connect*.

1.5 MODELLO BANDA LATENZA

n = numero di byte da inviare

$$D(n) = L_0 + \frac{n}{B}$$

1.6 PROTOCOLLO A LIVELLO APPLICATIVO: DNS

Lo scopo è di trasformare gli indirizzi numerici in stringhe. Lo scopo della funzione *addrinfo()* è di trasformare appunto l'indirizzo.

Il protocollo prevede che ci sia un *client* che invia una richiesta ed un *server* che risponda, basato sul protocollo di trasporto **UDP**, è possibile utilizzare il protocollo **TCP** per poter inviare datagrammi più estesi.

La porta standard è la 53.

I datagrammi (serie da 32bit, con dimensione massima di 512byte) inviati sono composti da:

ident (16bit)	FLAG(16bit)
#domande	#risposte
#record autoritativi	#record aggiuntivi

Il #domande indica quante righe precedono le risposte. I record autoritativi seguono le risposte, a loro volta sono seguiti dai record aggiuntivi.

1.6.1 Tipi di indirizzi

- **Tipo A:** indirizzo standard (es. www.dibris.unige.it);
- **Tipo MX:** indirizzo del server di posta elettronica;
- **Tipo NS:** informazione interna del protocollo *DNS*, simile a **MX**, indica il server **DNS** del dominio;
- **Tipo CNAME:** la **C** rappresenta **Canonical**, possono corrispondere più indirizzi alla stessa macchina, *CNAME* serve per identificare la macchina interpellata.

1.6.2 Risoluzione dei nomi

Per ottenere l'indirizzo di un qualunque server collegato alla rete si utilizza una struttura gerarchica.

1. **Root:** conosce tutti i domini *top level*;
2. **Top level:** i domini top level gerarchicamente sono i figli di root, esempio: *it,com,uk etc.*;

3. **Autoritativi:** figli di *top level*, esempio: *unige*, *google*, *amazon*, un livello *autoritativo* può avere come figlio un altro livello *autoritativo*, fino a circa 127 livelli;

1.6.3 Algoritmo ricorsivo

Il client invia una richiesta al suo server in locale, il quale va a vedere l'indirizzo, quindi contatta il server root perchè non ha abbastanza informazioni per risolvere la richiesta.

Quindi il server *root* inoltra la stessa richiesta al *top level* domain corretto, il quale a propria volta invia la medesima richiesta al server *autoritativo* e così via.

Una volta terminata la ricorsione l'indirizzo del server contenuto nell'ultimo dominio autoritativo viene mandato indietro fino al client seguendo la chiamata ricorsiva (risposta autoritativa).

1.6.4 Algoritmo iterativo

Il client invia la richiesta al suo server in locale, che se non riesce a rispondere, il client reindirizza la richiesta al server root, questo rimbalzo si ripete fintanto che il dominio corretto non risponde (risposta autoritativa).

1.6.5 Algoritmo ibrido

I server che contengono le informazioni sono contattati in maniera iterativa, i local servers invece supportano l'algoritmo ricorsivo.

Il server in locale può dare una risposta non autoritativa se il nome è salvato in cache.

1.7 PROTOCOLLO NTP

Il protocollo **NTP** (Network Time Protocol) si basa sulla gestione del tempo, necessario per la sincronizzazione di orologi nella rete.

La versione attuale è la 4 ed è definita nel protocollo *RFC5905*.

Il problema che affronta questo protocollo è relativo agli orologi locali dei calcolatori che si connettono tra di loro.

Il protocollo **NTP** utilizza 4 timestamp:

- t_1 : il tempo del momento in cui viene inviato il messaggio (client);
- t_2 : il tempo del momento in cui viene ricevuto il messaggio (host);
- t_3 : il tempo del momento in cui viene inviato il messaggio (host);

- t_4 : il tempo del momento in cui viene ricevuto il messaggio (client).

Quindi a differenza dei protocolli **UTP** e **TCP** individuo 2 timestamp in più.

Il *Round Trip Time* sarà dato da:

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

La **epoca** è lo standard cui si fa riferimento secondo cui dal punto di vista dei calcolatori comincia il tempo (1 gennaio 1970).

Considerando la precisione teorica degli orologi a disposizione dei calcolatori, i server **NTP** hanno una struttura gerarchica:

- **livello 0**: orologio atomico;
- **livello 1**: macchina collegata direttamente allo strato 0;
- **livello 2**: macchine collegato allo strato 1, i livelli successivi sono ricorsivi.

Ognuna di queste macchine è sia client, che server.

POSTA ELETTRONICA

Una delle caratteristiche principali della posta elettronica è l' *asincronicità*.

2.1 SMTP

Le email utilizzano diverse protocolli tra cui **SMTP**, definito per la prima volta nell' **RFC 821** (che risale al 1980) ed espanso nell' **RFC 5321**.

Utilizza una codifica **ASCII a 7bit**, la porta predefinita è la 25 ed utilizza il protocollo **TCP**.

2.1.1 Protocollo Best Effort

L'applicazione del protocollo *Best Effort* nell'ambito del protocollo *SMTP* implica un salvataggio da parte del mittente dei dati da inviare fintanto che non si è certi che il messaggio sia arrivato, oppure alla scadenza di un timeout.

2.1.2 Funzionamento

Il mittente (client) invia il messaggio ad un server, quindi il client alla fine dell'invio può chiudere la connessione. A questo punto il server che ha ricevuto il messaggio lo invia ad un altro server, se necessario, con l'obiettivo di spostare il messaggio al server del ricevente, in questo punto termina il compito del protocollo **SMTP**, successivamente il ricevente riceve il messaggio nel momento in cui apre una connessione con il server.

2.1.2.1 Three way connection

Usa la *three way connection*, una volta stabilita la connessione (dopo il segnale di ACK)

1. **SERVER:** invia la stringa `220 server@example.com;`
2. **CLIENT:** invia la stringa `HELO server@example.it`, oppure la stringa `EHLO server@example.it` per connessioni sicure;
3. **SERVER:** invia la stringa `250 client@example.it;`
4. **CLIENT:** invia la stringa `MAIL FROM: client@example.it;`

5. **SERVER:** invia la stringa *250 client@example.it server ok*;
6. **CLIENT:** invia la stringa *RECPT TO: recipient@example.it* ;
7. **SERVER:** invia la stringa *250 recipient@example.it recipient OK*;
8. **CLIENT:** invia la stringa contenente *DATA*;
9. **SERVER:** risponde con *354*;
10. **CLIENT:** invia il testo del **messaggio** che si conclude con un newline ed un punto;
11. **SERVER:** risponde con *250* segnalando la fine della comunicazione;
12. **CLIENT:** invia la stringa *quit*.

2.1.2.2 Formato del messaggio

Il messaggio è composto da 72 caratteri per riga, le sequenze di caratteri *<CR><LF>* costituiscono un a capo; lo *header* termina con due a capi di seguito.

Lo *header* è composto di chiavi, alcune sono ridondanti rispetto ai dati scambiati secondo il protocollo **SMTP**, difatti i dati esposti nel corpo del messaggio sono puramente a scopo informativo.

- subject;
- from;
- to;
- cc;
- bcc;
- date

I messaggi possono essere codificati in base 64 Radix, come caratteri comprende le cifre, le lettere maiuscole e minuscole, *_* ed *=*.

Quindi ad ogni *6bit* del messaggio corrisponde 1 carattere Radix.

La prima versione del protocollo **SMTP** utilizzava la codifica *uuencode*.

Un evoluzione del Radix 64 utilizza i MIME, per esempio un *MIME/TYPE text/ascii* indica che devo codificare un file di testo formato *ascii*.

MAIL USER AGENT

POP3

TCP/IP

IMAP

2.2 FTP

Definito ne **RFC 114** nel 1971, prima del **TCP/UDP**, successivamente ne **RFC 765** e **RFC 959**, la versione attualmente in uso.

Viene usata la porta **TCP/21** su cui viene aperto il *canale di controllo*, attraverso cui il cliente può mandare una serie di comandi al server.

La porta **TCP/20** viene usata per la connessione dati, ovvero per scambiare i file.

Solitamente viene eseguita una forma di autenticazione.

Esistono una serie di comandi:

- LIST;
- CD;
- PORT;
- LCD;
- RETR;
- STOR;
- DELE.

Nell' **RFC 1579** viene definito un nuovo parametro, il **PASV** che abilita la modalità passiva.

2.2.1 Funzionamento

Prima viene aperta la connessione attraverso il canale di controllo, quindi vengono inviati i dati di autenticazione formattati come:

USER username

PASS password

Nel caso di successo il server risponde con 200OK.

Per scaricare un file è necessario avere informazioni sullo stesso, quindi il client crea un socket, ne invia le informazioni al server, quindi il server invia il file. Alla fine del trasferimento dati viene chiusa la connessione.

2.2.2 FTP anonimo

Se come utente passiamo **USER anonymous**, allora parliamo di FTP anonimo, il sistema non richiede password nonostante venga data la possibilità per identificare l'utente, attraverso l'email per esempio.

HTTP

Definito per la prima volta nel 1977, nell' **RFC 2068**, poi nel 1999 nell' **RFC 2616** e nel 2015 nell' **RFC 7230**.

Un aspetto fondamentale del protocollo HTTP sono gli **URL** (Uniform Resource Identifier) la cui sintassi è:

http_URL = "http:" "/" host [":" port] [abs_path]

3.1 RICHIESTA

I metodi con cui si possono accedere alle risorse da appendere ad un URL sono:

- GET: Transfer a current representation of the target resource;
- HEAD: Same as GET, but only transfer the status line and header section;
- POST: Perform resource-specific processing on the request payload;
- PUT: Replace all current representations of the target resource with the request payload;
- DELETE: move all current representations of the target resource;
- CONNECT: establish a tunnel to the server identified by the target resource;
- OPTIONS: describe the communication options for the target resource;
- TRACE: perform a message loop-back test along the path to the target resource.

Questi metodi sono inclusi nei parametri inviati nell'header, che oltre al metodo include: l'url a cui si vuole accedere e la versione del protocollo HTTP in utilizzo.

3.2 RISPOSTA

3.2.1 Header

La risposta di un server ad una richiesta http conterrà un header che include lo stato che può essere:

- 1xx: Informational - Request received, continuing process;
- 2xx: Success - The action was successfully received, understood, and accepted;
- 3xx: Redirection - Further action must be taken in order to complete the request;
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled;
- 5xx: Server Error - The server failed to fulfill an apparently valid request;

L'header del protocollo HTTP 1.0 inoltre conterrà *Connection Close*, ovvero la segnalazione che la connessione è stata chiusa, dal protocollo HTTP 1.1 questa risposta non è più inclusa, ma la gestione delle connessioni aperte sarà implementata nel server con un meccanismo di timeout.

3.2.2 Body

Ogni risposta conterrà un body, ovvero il contenuto della risposta.

3.3 PROXY

I server proxy sono responsabili della cache, nell'header di richiesta è presente il parametro *IfModifiedSince* che contiene una data nel formato GMT, nell'header di risposta viene mandato il parametro *LastModified*, è il server che confronta le date e valuta se generare una nuova richiesta.

Parte II

SISTEMI OPERATIVI

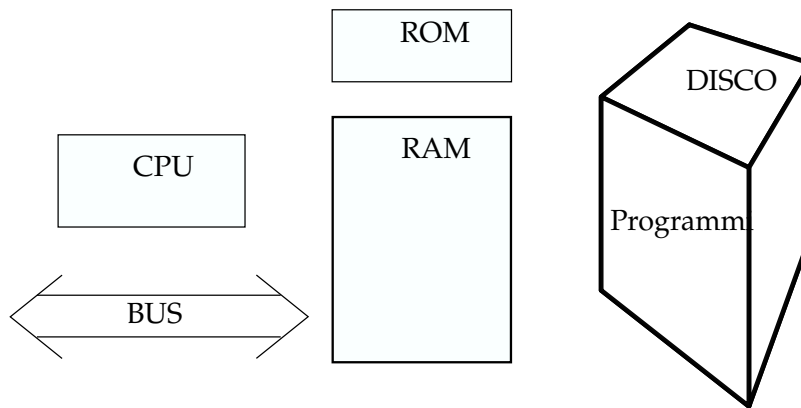


Figura 4.1: Struttura base di un calcolatore

Lo scopo di questa sezione è di virtualizzare le componenti fisiche di un calcolatore per renderne più facile l'utilizzo.

Sinonimi di *sistema operativo* sono **macchina virtuale** (virtual machine) e **gestore delle risorse** (resource manager).

4.1 VIRTUALIZZAZIONE DELLA CPU

L'unica nota in questo programma è l'utilizzo della funzione *spin()*, che controlla l'orologio di sistema fino a quanto 1 minuto non è passato, quindi riprende l'esecuzione.

Listing 4.1: Interazione con la CPU usando la funzione *spin()*

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];

    while (1) {
        printf("%s\n", str);
        Spin(1);
    }
}
```

```
    return 0;
}
```

Da notare che se il programma viene eseguito con un comando del tipo:

Listing 4.2: Esecuzione in concorrenza

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D
[7353]
[7354]
[7355]
[7356]
A
B
D
C
A
B
D
C
A
...
```

Possiamo osservare che sembra che vengano eseguiti simultaneamente quattro programmi diversi, ciò è dovuto alla *virtualizzazione della CPU*.

4.2 MEMORIA VIRTUALIZZATA

Listing 4.3: Dimostrazione dell'utilizzo di indirizzi virtuali

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    int value;
    int *p = &value;
    if(argc != 2) {
        fprintf(stderr, "usage: %s <value>\n", *argv);
        return EXIT_FAILURE;
    }
    printf("(pid:%d) addr of p: %llx\n", (int)getpid(), (unsigned
        long long)&p);

    printf("(pid:%d) addr stored in p: %llx\n", (int)getpid(), (
        unsigned long long)p);

    *p = atoi(argv[1]);
    while(1) {
```



```

    sleep(1);
    ++*p;
    printf("(pid:%d) value of p: %d\n", getpid(), *p);
}
}

```

La memoria **RAM** è generalmente virtualizzata, difatti se eseguiamo il programma precedente con un comando tipo:

Listing 4.4: Esecuzione in concorrenza

```

prompt > ./mem & ./mem &
[1] 24113
[2] 24114
(24113) address pointed by p: 0x200000
(24114) address pointed by p: 0x200000
(24113) p: 1
(24114) p: 1
(24113) p: 2
(24114) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...

```

Possiamo osservare il funzionamento della virtualizzazione della memoria, infatti ogni programma è lanciato in un proprio **virtual address space**; semplifica la gestione della memoria per il programma ed è mappato ad una sezione della memoria fisica.

4.3 PERIFERICHE

Nell'interazione di un calcolatore con le periferiche, è il kernel che interagisce direttamente con l'hardware, mentre le applicazioni possono utilizzare le periferiche solo attraverso il kernel.

Vi sono 4 diversi livelli di privilegi per interagire con i dispositivi:

- 0, il livello più privilegiato, utilizzato dal kernel;
- 1 solitamente non utilizzato;
- 2 solitamente non utilizzato;
- 3 il livello meno privilegiato, utilizzato da tutti gli utenti compreso l'utente di *root*.

Se l'utente richiede un'interazione con una periferica, un comando di *TRAP* viene inviato al kernel, il quale restituisce un comando di *return from TRAP*.

4.4 MEMORIA

4.4.1 Spazi di indirizzamento

Ad ogni processo in esecuzione corrisponde un *address space*, una virtualizzazione della memoria che comporta diversi vantaggi, tra cui non permettere ai programmi di modificare memoria non nella propria *address space* ed in generale un'efficienza maggiore.

Lo spazio di indirizzamento non può essere più grande della memoria fisica ed occupa una sezione di memoria contigua.

Utilizzando questo metodo abbiamo due problemi di frammentazione interna ed esterna.

Un processo è composto da:

- **codice:** statico, regione fissa;
- **dati:** regione fissa;
- **stack:** regione dinamica, può crescere o diminuire;
- **heap:** regione dinamica.

Quindi l'*address space* è composto, in ordine, da:

- **codice del programma:** dove risiedono le istruzioni;
- **heap:** contiene i dati creati con i *malloc* e le strutture dinamiche, cresce verso il basso;
- **memoria libera:** che può essere occupata dallo heap e dallo stack;
- **stack:** dove risiedono le variabili locali, i valori di ritorno delle funzioni, etc. e cresce verso l'alto.

All'interno del sistema operativo è contenuta una struttura dati che contiene informazioni relative ai processi in memoria, **PCB** (Process Control Bloc).

4.4.2 Virtualizzazione

Gli obiettivi della virtualizzazione sono: trasparenza, efficienza e protezione ovvero un programma non deve accorgersi che sta utilizzando solo una porzione della memoria disponibile.

Per implementare queste funzioni si utilizza una **MMU**; al suo interno salviamo un registro chiamato base, il quale cambia a seconda del programma in esecuzione.

Questa implementazione non è corretta in quanto non abbiamo indicato un limite al valore che possiamo assegnare alla base, quindi i processi potrebbero accedere a memoria successiva alla loro, per cui

sono gestiti anche i limiti su quali valori posso assegnare al registro base.

Nel caso di un *segmentation fault* (errore relativo all'accesso di memoria non propria) vengono sollevati gli **interrupt handlers**, e gli **exception handlers**, i quali gestiscono le eccezioni. Altri errori simili che sollevano un interrupt sono la divisione per 0.

Gli interrupt possono anche essere utilizzati per interagire con il sistema operativo, il sistema operativo può disabilitare gli interrupt.

4.4.3 Frammentazione

La frammentazione interna è composta dalla memoria allocata per un programma, ma non utilizzata.

La frammentazione esterna invece è la memoria che non viene utilizzata perchè non abbastanza grande da ospitare un processo e tra due address spaces.

Il *compattamento* della memoria non è una soluzione attuabile in quanto sarebbe estremamente inefficiente.

Esistono diversi metodi per poter utilizzare le porzioni di memoria frammentata.

4.4.3.1 Segmentazione

La segmentazione è utilizzata per poter utilizzare la memoria frammentata esternamente. Si utilizza la *rilocalizzazione dinamica*, ovvero la traduzione di indirizzi logici in indirizzi fisici attraverso una funzione di rilocalizzazione implementata nell'hardware, **MMU**.

REGISTRI BASE LIMITE Nell' 8086 in 16bit la segmentazione era classificata in:

- **CS:** code segment;
- **DS:** data segment;
- **SS:** stack segment;
- **ES:** extra segment.

In questa versione, la **MMU** mantiene una lista di registri *base/limite*, ad ogni coppia corrisponde un segmento di memoria frammentata, la base contiene il primo indirizzo di memoria utile, mentre il limite quanta memoria contigua è disponibile.

Nel momento in cui il processore genera un indirizzo virtuale prende in considerazione questi registri; l'indirizzo viene generato in fase di *fetch* dell'istruzione. Se l'indirizzo generato supera il valore contenuto nel registro limite, un eccezione viene generata in quanto l'indirizzo richiesto non apparterebbe al segmento.

TABELLE DEI DESCRITTORI L'architettura 386, risolve due problemi relativi alla precedente soluzione: una più veloce risoluzione dei permessi ed il limite di registri salvabili in MMU.

Per ottenere l'indirizzo di memoria giusto viene ora utilizzata una parola che indica sia il *segmento* di riferimento che l'offset all'interno del segmento (indirizzo virtuale).

Inoltre viene introdotta una struttura dai tabellare, mantenuta all'interno della memoria riservata del sistema operativo, detta *tabella dei descrittori*, contenente i dati relativi alla coppia dei registri *base/ limite* di ogni segmento, il bit di protezione e la gestione dei privilegi DPL). La MMU conterrà soltanto due registri, **STLR** (Segment Table Limit Register) la dimensione della tabella dei segmenti e **STBR** (Segment Table Base Register) l'indirizzo base della tabella dei segmenti.

- **CS**: code segment, con una local descriptor table;
- **DS**: data segment, con una global descriptor table;
- **SS**: stack segment;
- **ES**: extra segment.

$$\text{MAX}(\text{CPL} - \text{RPL}) \leq \text{DPL} - \text{BASE} + \text{OFFSET}$$

I segmenti di codice (**CS**) possono essere utilizzati da più processi uguali.

Quindi le funzioni *Jmp*, *Call* e *Ret* per accedere ad un segmento utilizzano due dati: **offset** e **selettore offset**.

Definiamo **EIP** (Instruction Pointer) e **ESP** (Stack Pointer), rispettivamente appartengono ad un determinato **CS** (Code Segment) e **SS** (Stack Segment).

EIP e **CS** alla chiamata di *Jmp*, *Call* o *Ret*, verranno salvati nello stack (**ESP**).

Il **Call Gate** è uno strato che si aggiunge all'indirizzo virtuale per non permettere l'erroneo accesso a sezioni di memoria non utilizzabili dall'utente o dal programma.

Ogni livello di privilegio ha uno **stack separato**, per offrire una maggiore stabilità del sistema, quindi se una *Call* cambia il livello di privilegio dovrà anche cambiare lo *stack*: se il livello di privilegio si riduce, tutti i registri che contengono riferimenti a sezioni di memoria privilegiata vengono azzerati.

INTERRUPT Se l'utente è kernel, al momento dell'arrivo di un interrupt non c'è bisogno di cambiare lo stack, altrimenti sì.

All'arrivo di un interrupt **eflag** (flag dell'interrupt) viene salvato, quindi l'interrupt viene disattivato, **iret** esce dall'interrupt handler.

4.4.4 Paginazione

Per il processo la paginazione è trasparente. La memoria fisica è divisa equamente in **page frame**, la virtualizzazione corrispondente sono le **pagine**, ovvero le pagine sono ordinate, mentre la rappresentazione nel **page frame** non è necessariamente ordinata, ma i **page frame** sono numerati. Le pagine sono grandi 4096byte , in generale sempre potenze di 2.

Quindi si utilizza una tabella di appoggio, lunga tanto quanto il numero di pagine, per associare *page frames* a *pagine*, in questa tabella la colonna **present(flag)**, vero se la pagina è utilizzata nella memoria fisica. Una tabella simile è creata ed associata per ogni processo, la tabella è creabile e modificabile solo dal kernel.

Dato un indirizzo virtuale si divide per la grandezza della pagina per ottenere il numero della pagina, quindi dalla tabella di appoggio si seleziona il *page frame* corrispondente, quindi si utilizza il resto della divisione per determinare l'indirizzo di **memoria fisica** corrispondente.

Indirizzo fisico = $\text{Indirizzo virtuale} / \text{Grandezza delle pagine} + \text{Indirizzo virtuale} \% \text{Grandezza delle pagine}$

Stabilita la grandezza della pagina si interpretano i primi *bit* dell'indirizzo virtuale per trovare il numero di pagina, mentre il restante si usa come *offset*, dato l'indirizzo della base si appende l'offset per ottenere l'indirizzo fisico.

La paginazione usa appunto le **paging tables**, per ogni riga abbiamo:

- il **PPN** (Physical Page Number);
- un bit di validità;
- un bit di presenza (se la pagina è presente);
- un bit di accesso (se ha già acceduto);
- un bit di modifica;
- dei bit per il livello di privilegio (r,w,x).

Per migliorare l'efficienza si utilizza una cache, la **TLB** (Translation Lookaside Buffer), da un **VPN** (Virtual Page Number), si ottiene il **PPN** (Physical Page Number) ed il **PFN** (Physical Frame Number).

Per migliorare ulteriormente la paginazione, non si utilizzano i bit di accesso e modifica, sarà quindi il kernel a tenere traccia di questi valori, lo stesso ragionamento si può fare per i bit sui permessi.

OSSERVAZIONE Per indicare una ram di 4GB sono necessari 32bit , un indirizzo virtuale è composto da 20bit per selezionare la pagina e:

- $v = 1\text{bit}$;

- $p = 1bit$;
- $r/w/x = 3bit$;
- $d = 1bit$.

Quindi abbiamo un totale di $27bit$ ($32bit$ normalizzati), per una RAM da 4GB la *paging table* occuperebbe 400mb di memoria contigua nella RAM.

Per rendere più efficiente la tabella, i $20bit$ di indirizzamento di dividono logicamente in 2 blocchi da $10bit$, il primo per indirizzare il blocco di memoria, il secondo per ottenere l'effettivo indirizzo fisico.

Il primo blocco in realtà indirizza su di una nuova tabella di appoggio il cui contenuto sarà se la pagina è occupata o no e l'indirizzo della tabella corrispondente, quindi si usa la tabella di appoggio ed il secondo blocco di $10bit$ per ottenere l'indirizzo fisico.

4.4.4.1 Base limite, Segmentazione e paginazione

Compariamo i sistemi di suddivisione della memoria: **B+L**, **segmentazione** e **paginazione**.

B+L:

- ? spazio di indirizzamento contiguo in RAM;
- – frammentazione interna;
- – frammentazione esterna;
- – condivisione codice;
- + isolamento protezione;
- – efficiente;
- – spazio di indirizzamento contiguo.

Segmentazione:

- ? spazio di indirizzamento contiguo nel segmento, ma non nel processo ;
- – frammentazione interna;
- – frammentazione esterna;
- \mp condivisione codice;
- + isolamento protezione;
- \pm efficiente;

- – spazio di indirizzamento contiguo.

Paginazione:

- – spazio di indirizzamento non contiguo;
- \pm frammentazione interna ($4k$);
- – frammentazione esterna;
- \mp condivisione codice;
- + isolamento protezione;
- \pm efficiente;
- – spazio di indirizzamento contiguo.

4.5 SYSTEM CALLS

4.5.1 *fork*

La funzione *fork* crea un processo quasi identico al processo padre e restituisce un numero positivo chiamato *pid_t*. Ogni processo ha un *ppid* (*pid* del padre).

4.5.2 *exit*

La funzione *exit* ritorna 0 se il processo si chiude correttamente, altrimenti un numero.

4.5.3 *wait*

Ritorna un *pid_t* quando il processo passato come argomento termina.

4.5.4 *exec*

Sostituisce il processo corrente con uno nuovo, l'argomento è un PATH.

4.5.5 *open*

Apri un file e gli assegna il primo file descriptor disponibile.

4.5.6 *dup*

Duplica un fd creandone uno nuovo che punta allo stesso oggetto.

4.5.7 *dup2*

La *dup2* ha due parametri, il primo il vecchio file descriptor ed il secondo il nuovo, duplica il primo parametro nel secondo, fa una *close* del secondo parametro nel caso fosse già aperto.

4.5.8 *pipe*

Listing 4.5: Esempio di pipe (1)

```
# (1)
ls -l | grep cammello

# (2)
ls -l > camelus
grep cammello < camelus
```

L'operatore `|` permette di chiamare la syscall *pipe()* la quale permette di eseguire l'operazione esclusivamente in memoria, difatti nel secondo esempio *camelus* sarebbe scritto su disco e, su risultati molto grandi sarebbe molto più lento ed inefficiente.

Quindi *pipe(fds)* prende come argomento un array di file descriptors ed utilizza un buffer riservato al kernel per svolgere la lettura.

Il funzionamento per il comando (1) sarà:

- il processo padre chiama la *pipe()*, 2 *fork()*, *close(3)*, *close(4)* e 2 *wait()*;
- la prima figlia *ls -l* chiama:
 - *close(1)*;
 - *dup(3)*;
 - *close(2)*;
 - *close(4)* dove 4 è il fd della pipe;
 - *exec(...)*;
- la seconda figlia *grep cammello* chiama:
 - *close(0)*;
 - *dup(4)*, dove 3 è il file descriptor della fork;
 - *close(2)*;
 - *close(4)* dove 4 è il fd della pipe;
 - *exec(...)*;

4.6 SHELL

La shell è il programma che attende la scrittura e l'invio dei comandi, una delle funzionalità è di espandere le variabili passate (es. *.c).

Parte III

UTILITÀ

APPENDIX TEST

Listing a.1: Esempio di MAKEFILE

```
CFLAGS=-Wall -ansi -pedantic -Werror -std=c11
CC=clang
EXES=cpu mem threads.v0 threads.v1 threads_101

all: $(EXES)

cpu: cpu.c
    $(CC) $(CFLAGS) -o $@ $^

cpu: mem.c
    $(CC) $(CFLAGS) -o $@ $^

threads.v0: threads.v0.c
    $(CC) $(CFLAGS) -o $@ $^ -pthread

threads.v1: threads.v0.c
    $(CC) $(CFLAGS) -o $@ $^ -pthread

threads_101: threads.v0.c
    $(CC) $(CFLAGS) -o $@ $^ -pthread

clean:
    rm -f $(EXES)

.PHONY: clean
```