SISTEMI E TRASMISSIONE DELLE INFORMAZIONI

RICCARDO CEREGHINO

Appunti

Settembre 2019 – 0.0.1classicthesis v4.6

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

LISTINGS

ACRONYMS

Parte I

RETI DI CALCOLATORI

1 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*

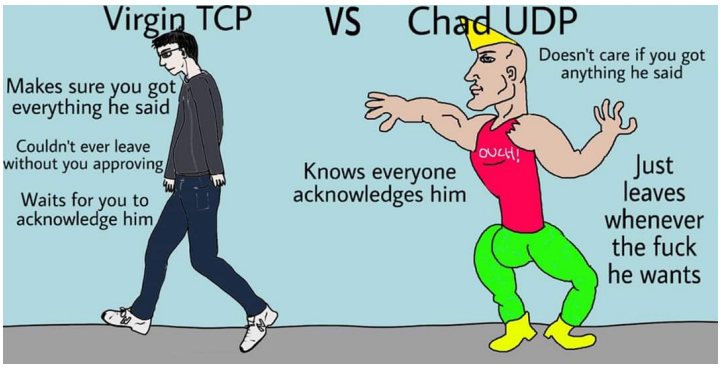
E' un modulo che gestisce le risore fisiche dei vari moduli, tra cui NIC, gestiscono le interruzioni del dispositivo.

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 verà 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, cosicchè NIC possa utilizzarli una volta ricevuto il pacchetto.

Il datagramma ricevuto sarà inserito dal DMA in un buffer, un interrupt nel frattempo creeà 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 perchè in attesa del processore verrano 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 processori 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'hos 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 sopravissuto 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 6*byte*.

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 modilità 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 4*byte* 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 16*bit*, 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 $IPv6$)

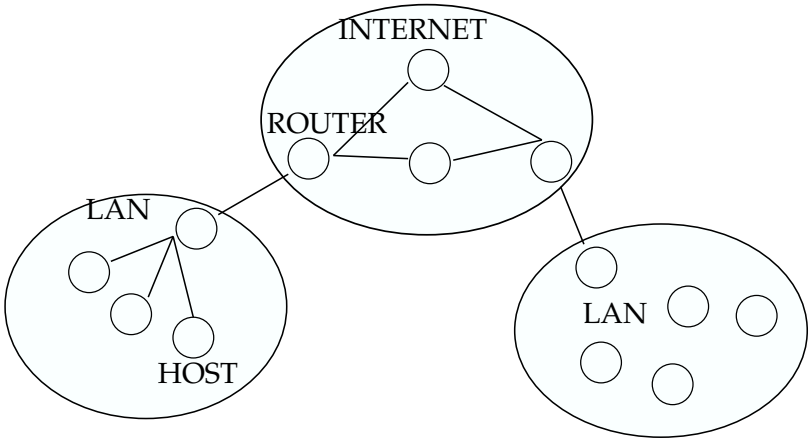


Figura 1.1: indirizzi

Un indirizzo *IPv*4 in default è composto da:

- 29*bit* dedicati al network;
- 3*bit* 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 messaggio alla macchina corretta sostituendo il proprio indirizzo alla richiesta assegnando una porta libera.

1.3.3 *Trasporto*

Vengono utilizzato 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 uteriore messaggio chiamato **ACK**, oltre che i dati.

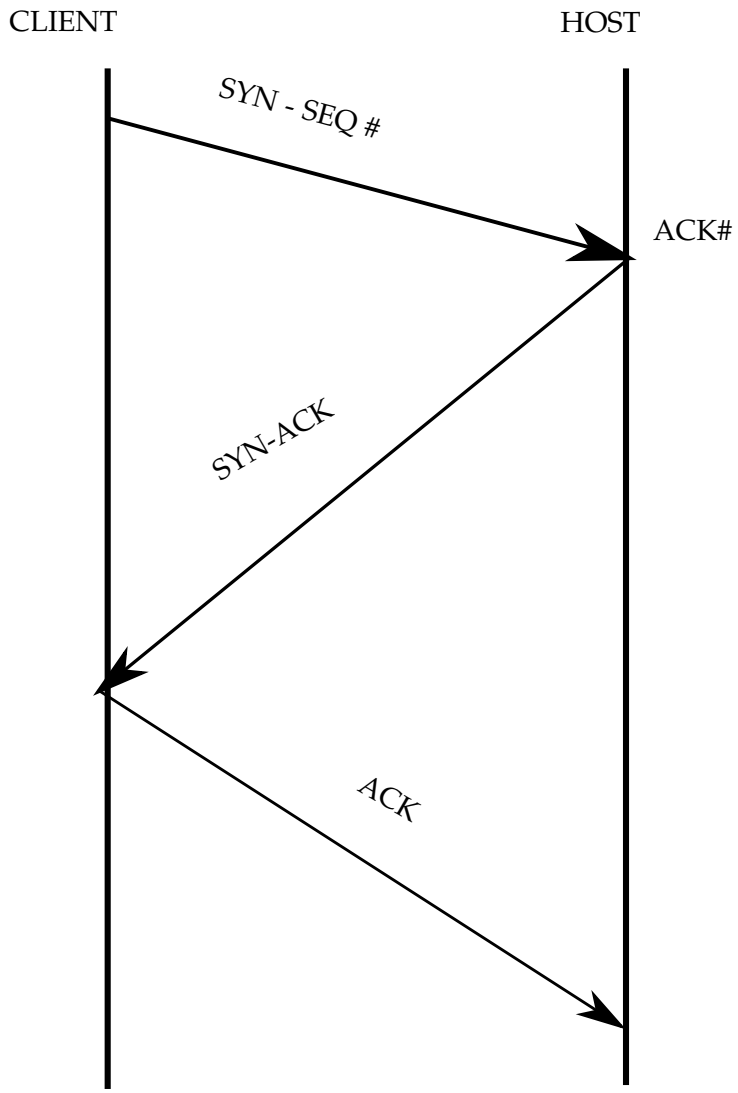


Figura 1.2: Three way handshake

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 squenza 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 stime della variabilità:

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

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 scambuio, 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 datagramm 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 reta, 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 aplicativo 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*: *IPV*4;
  - *AF_INET6*: *IPV*6;
- **type:**
  - *SOCK_DGRAM*: *IPv*4;
  - *SOCK_STREAM*: *IPv*6;
  - *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 possibie utilizzare il protocollo **TCP** per poter inviare datagrammi più estesi.

La porta standard è la 53.

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

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

Il *#domande* indica quante righe precedono le *risposte*. I record autoratitativi seguono le risposte, a loro volta sono seguiti dai 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 *autoratitativo*, 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 autoratitativo 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 piò dare una risposta non autoritativa se il nome è salvato in cache.

Parte II

SISTEMI OPERATIVI

2

INTRODUZIONE AI SISTEMI OPERATIVI

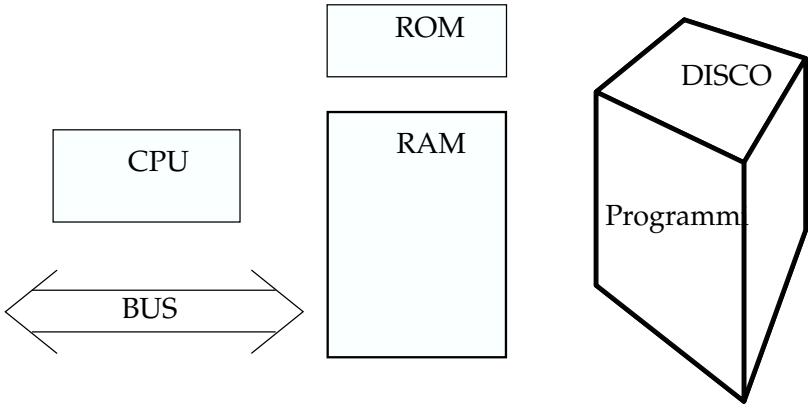


Figura 2.1: Struttura base di un calcolatore

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 dlle risorse** (resource manager).

2.1 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 2.1: 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 2.2: 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*.

2.2 MEMORIA VIRTUALIZZATA

Listing 2.3: 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 2.4: 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.

2.3 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 provilegiato, utilizzato da tutti gli utenti compreso l'utente di *root*.

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

2.4 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 frammentazionem 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).

2.4.1 *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.

2.4.2 *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.

2.4.2.1 *Segmentazione*

La segmentazione è utilizzata per poter utilizzare la memoria frammentata esternamente, per cui offre la possibilità di creare uno spazio di indirizzamento in memoria non contigua.

Nell' 8086 in 16*bit* la segmentazione era classificata in:

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

Nel 386, introduce le tabelle dei descrittori, che descrivono i segmenti che che includon, quindi contengono dati per: base, limite, bit di protezione e privilegi (DPL).

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

$$MAX(CPL \quad RPL) \leq DPL \quad BASE + OFFSET$$

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

Parte III

UTILITÀ

a

APPENDIX TEST

Listing a.1: Esempio di MAKEFILE

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