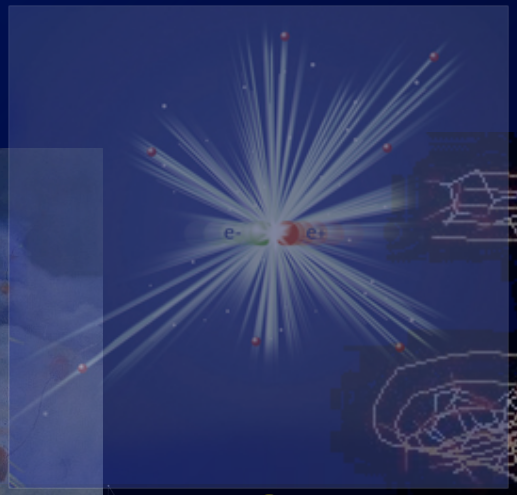
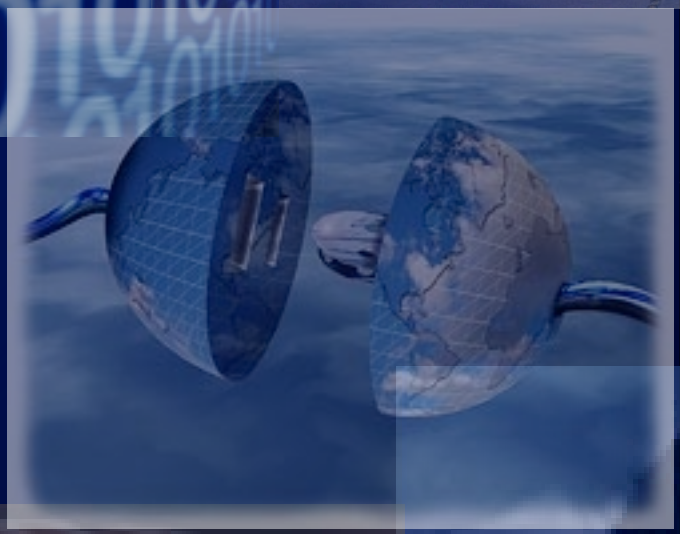
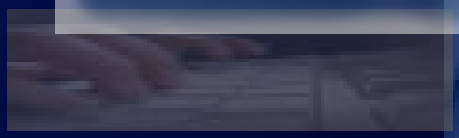
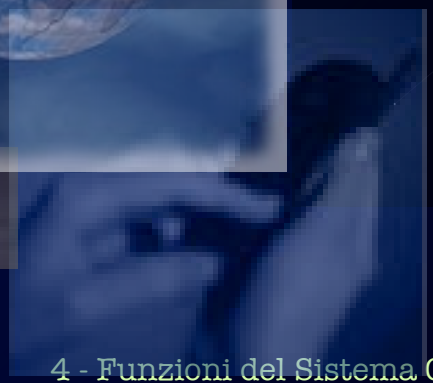
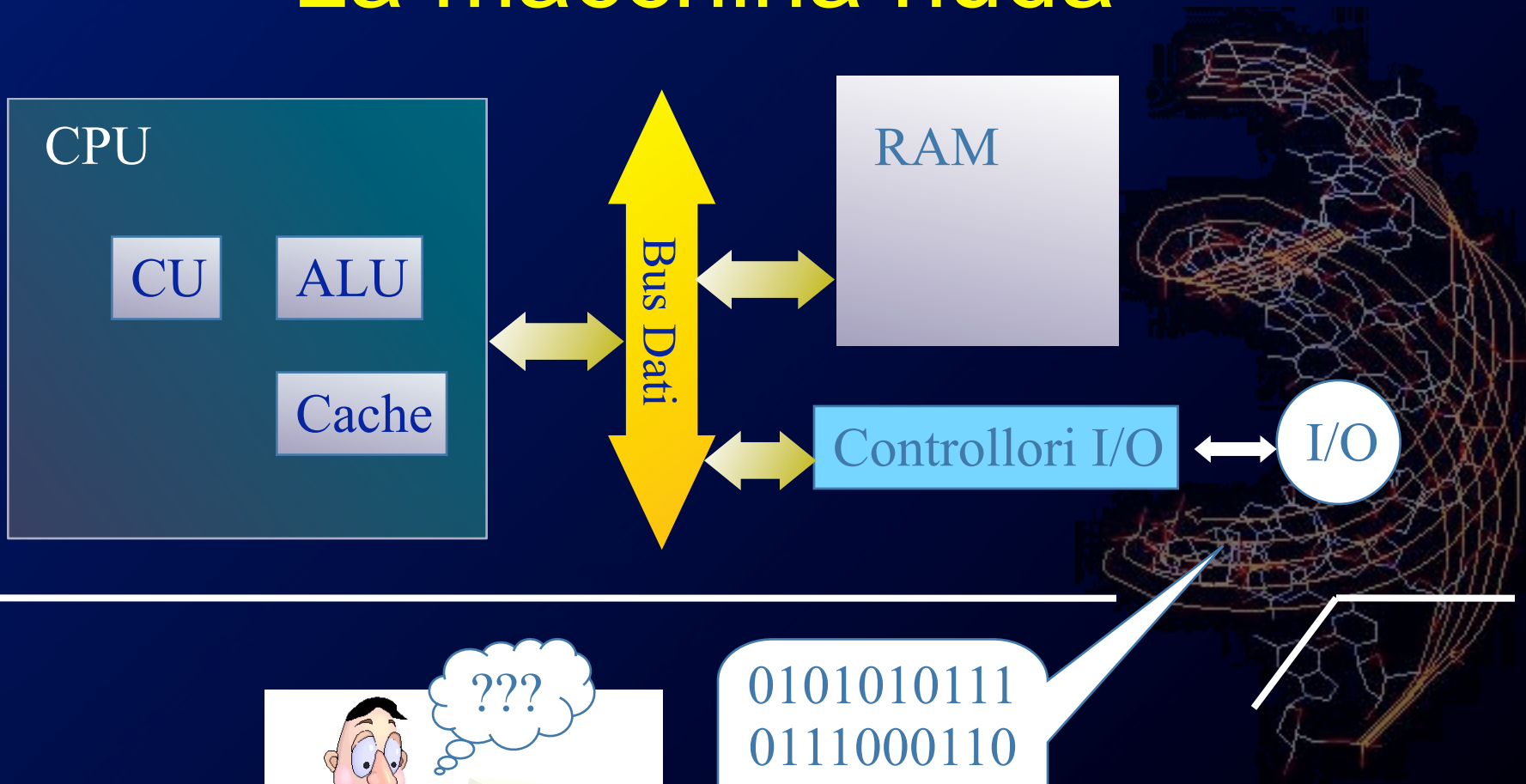


Funzioni del Sistema Operativo

- Concetto di OS
  - Gestione dei processi
  - Gestione della memoria
  - Gestione dell'I/O
  - Gestione del file system
- Compilatori/interpreti
- Programmi di utilità

La macchina nuda



Interfaccia

Come funziona quel televisore ?

Beh, c'è un sottile pennello di elettroni che viene sparato contro uno schermo fluorescente con elevata frequenza. Un circuito di sintonia permette di visualizzare le diverse stazioni che

Non fa nulla, ho trovato sul telecomando il tasto per accenderlo e quelli per cambiare canale...

La macchina virtuale

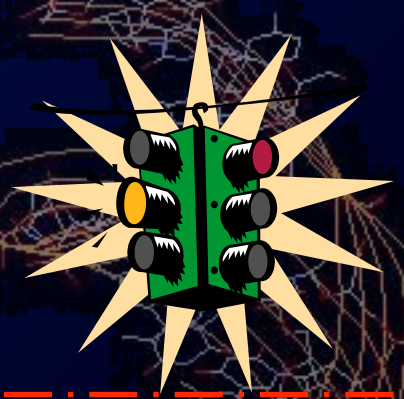
Tra la macchina nuda e l'utente si inserisce, con funzione di interfaccia, la macchina virtuale.

Si tratta di un insieme di programmi (software), che da una parte comunicano con l'hardware per gestire la macchina reale e dall'altra forniscono all'utente un ambiente virtuale i cui servizi fondamentali sono:

- Accesso semplificato alla realizzazione ed esecuzione di programmi
- Accesso e gestione delle risorse di I/O "ad alto livello"
- Accesso a dati e informazioni e protezione degli stessi

Il sistema operativo

L'OS gestisce le risorse hardware e controlla l'esecuzione di tutti gli altri programmi.



Servizi richiesti dagli utenti
(programmi, operazioni di I/O etc.)

Livello utente

Livello kernel (privilegiato)

Interfaccia verso l'esterno (system call)

Kernel

Macchina Virtuale

Gestione processi	Gestione memoria	Gestione I/O	Gestione file system	Traduttori

Hardware

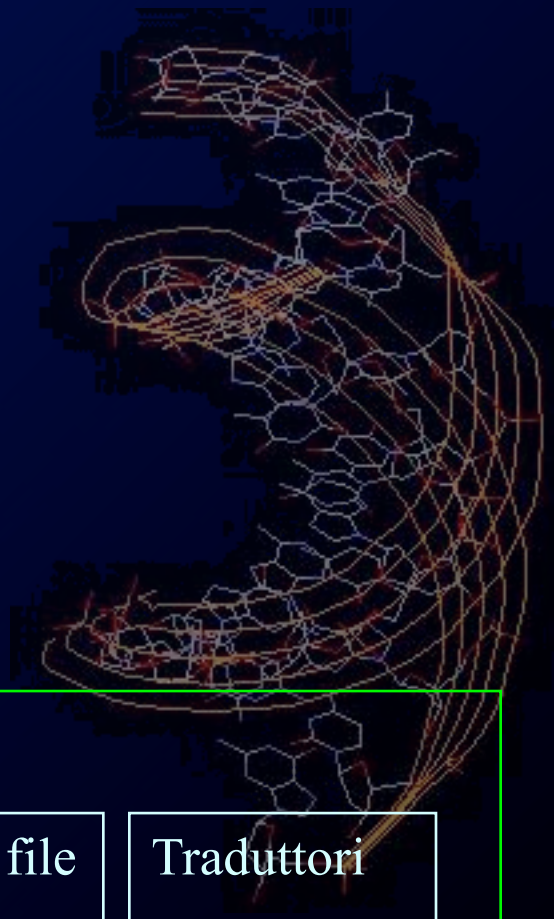
Protezione e condivisione

Oltre a costituire una macchina virtuale un sistema operativo (OS) deve essere in grado di condividere le risorse fra le applicazioni, nel contempo proteggendo ciascuna dai danni potenzialmente arrecati dalle altre.

Esempi:

Memoria non protetta: un'applicazione può invadere lo spazio usato da un'altra o addirittura dallo stesso OS

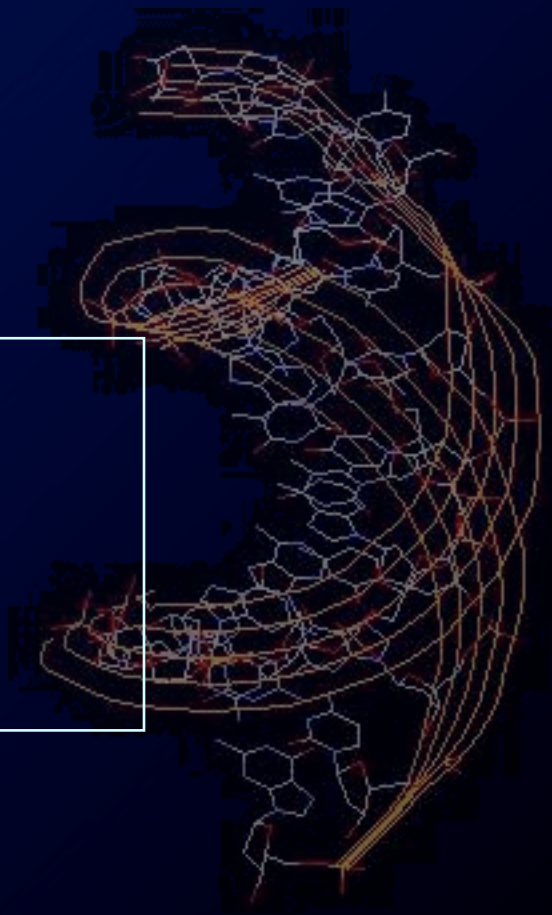
I/O non protetto: un'applicazione può "appropriarsi" di un dispositivo di I/O a tempo indeterminato



Kernel

Gestione processi	Gestione memoria	Gestione I/O	Gestione file system	Traduttori

Gestione processi



Programmi e Processi

Un programma eseguibile è un insieme di istruzioni ben ordinato in linguaggio macchina che può essere caricato in memoria : è un oggetto statico

Un processo può essere definito, grosso modo, come un programma in esecuzione, ed è quindi per natura un oggetto dinamico.

Un processo è descritto non solo dal programma, ma anche da tutte le informazioni (dinamiche) sullo stato di esecuzione (program counter, registri, dati etc.) dello stesso.

Monotasking e Multitasking

I primi OS erano Monotasking: un solo processo poteva essere in esecuzione in un dato momento.

Questo sistema è molto inefficiente nell'utilizzo delle risorse.

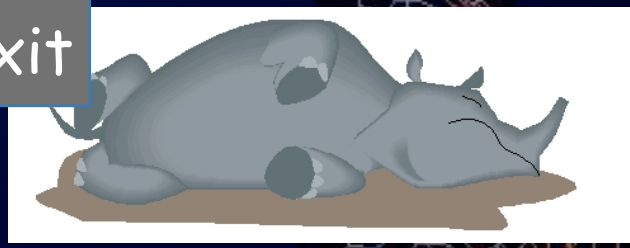
Per esempio un processo *I/O bound,* che spende la maggior parte del tempo in attesa di risposta dall'I/O blocca inutilmente la CPU, mentre un processo di tipo *CPU bound* può bloccare per lungo tempo l'interazione con l'utente "congelando" la macchina.

Multitasking: il controllo della CPU viene passato a turno da un processo all'altro. L'operazione di passaggio prende il nome di context switching.

Stati di un processo

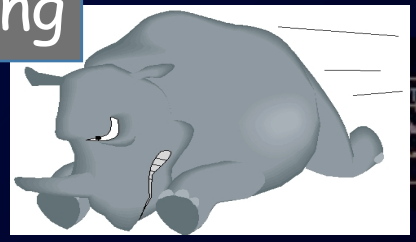
New

Exit

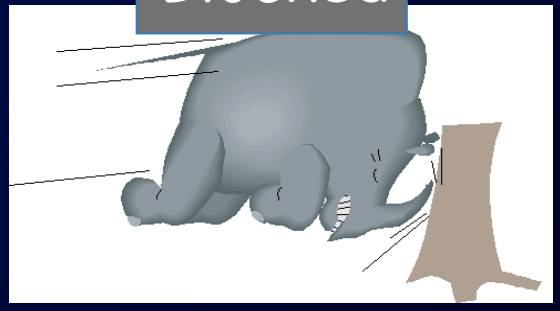


Ready

Running



Blocked

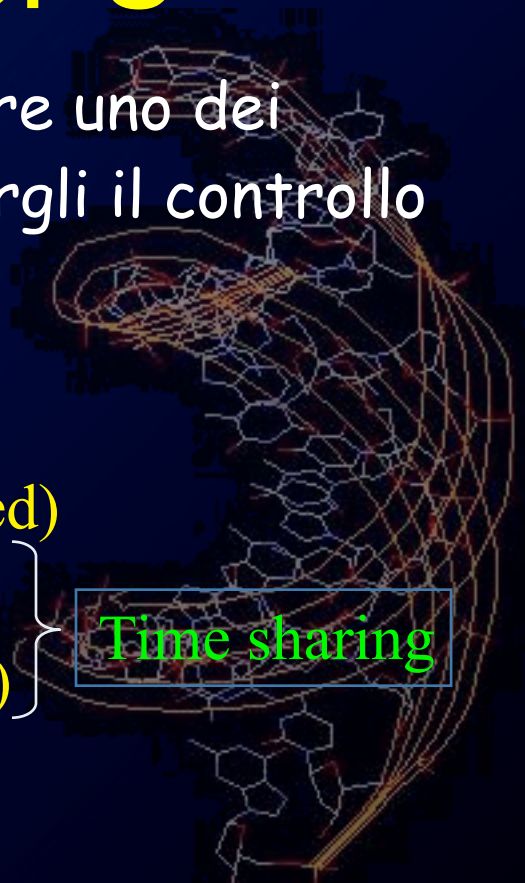


Scheduler

Interrupt

Lo scheduler della CPU

La sua funzione, come visto, è di selezionare uno dei processi che sono nello stato "Ready" e dargli il controllo della CPU.



Può far partire un nuovo processo quando

- Un processo si blocca (running => blocked)
- Una time slice finisce (running => ready)
- Un processo si sblocca (blocked => ready)
- Un processo termina (running => exit)

Time sharing

Esistono varie tecniche di assegnazione della CPU:
First In First Out, Shortest Job First, Round Robin…

Assegnazione della priorità

Confrontiamo il tempo di attesa media per un processo in caso sia utilizzata una strategia tipo FIFO e SJF per una data sequenza di processi P_i ognuno dei quali utilizza la CPU per un tempo t_i.

FIFO (esecuzione nell'ordine in cui si entra nella lista dei ready):

Attesa: 0 24 27

Attesa media= (0+24+27)/3 = 17 ms

SJF(precedenza ai processi che usano meno la CPU):

Attesa: 0 3 6

Attesa media= (0+3+6)/3 = 3 ms

La sfera di cristallo

Ma come può lo scheduler prevedere quale processo impegnerà per un tempo più breve la CPU ?

L'idea è quella di mediare la "storia recente" del processo e la sua "storia passata": la tecnica è la *media esponenziale:*

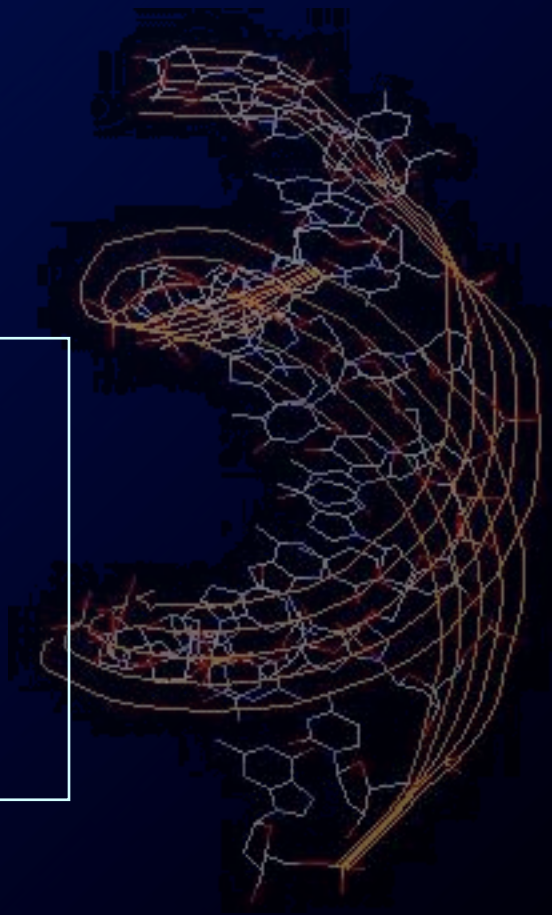
$$\langle t_{n+1} \rangle = a t_n + (1-a)\langle t_n \rangle \qquad 0<a<1$$

In t_n è contenuta la storia recente: la durata dell'ultimo "CPU burst".

In $\langle t_n \rangle$ è contenuta la storia passata: la media esponenziale dei tempi fino all'n-simo. Notare che la definizione è *ricorsiva* .

$\langle t_0 \rangle$ è definito come una costante o come la media complessiva dei tempi registrati dal sistema. Il parametro a regola il peso relativo di storia recente e storia passata del processo.

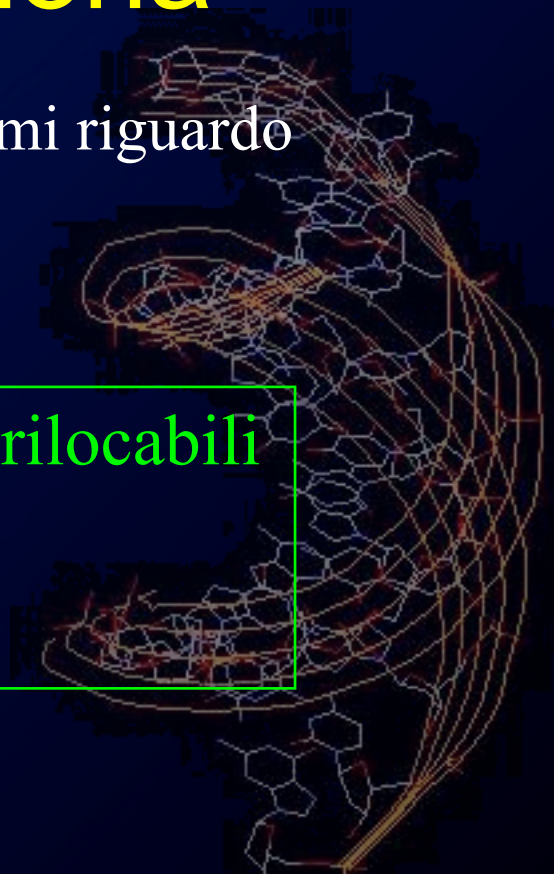
Gestione memoria



Gestione della memoria

Il sistema multitasking pone una serie di problemi riguardo la gestione della memoria:

1. Necessità di utilizzare programmi rilocabili
2. Protezione delle risorse
3. Condivisione delle risorse



Il sistema di gestione della memoria si occupa di tali questioni, nonché dell'organizzazione fisica e logica della memoria.

Gestione della memoria

La rilocabilità di un programma è necessaria. Nella RAM i programmi spesso hanno delle lunghezze tali da lasciare alcuni "buchi".

Questo spreco renderebbe spesso la memoria insufficiente. Inoltre un programma è tanto più efficiente quanto più vicini sono i dati (contigui).

Però al crescere dell'occupazione la contiguità diventa difficile. Inoltre spesso alcune risorse nella RAM vengono condivise da diversi programmi ma in modi diversi: alcuni leggono e scrivono altri posso solo leggere.

Per gestire tutte queste problematiche si usa una memoria virtuale

Memoria virtuale

I programmi utilizzano *indirizzi logici*, che vengono tradotti a run time in *indirizzi fisici* dall' hardware MMU.

La MMU può facilmente implementare protezioni e rilocazioni dei programmi. Inoltre si possono far corrispondere gli stessi indirizzi fisici a diversi indirizzi logici, condividendo così delle risorse: *dynamic link libraries (dll), shared objects (so).*

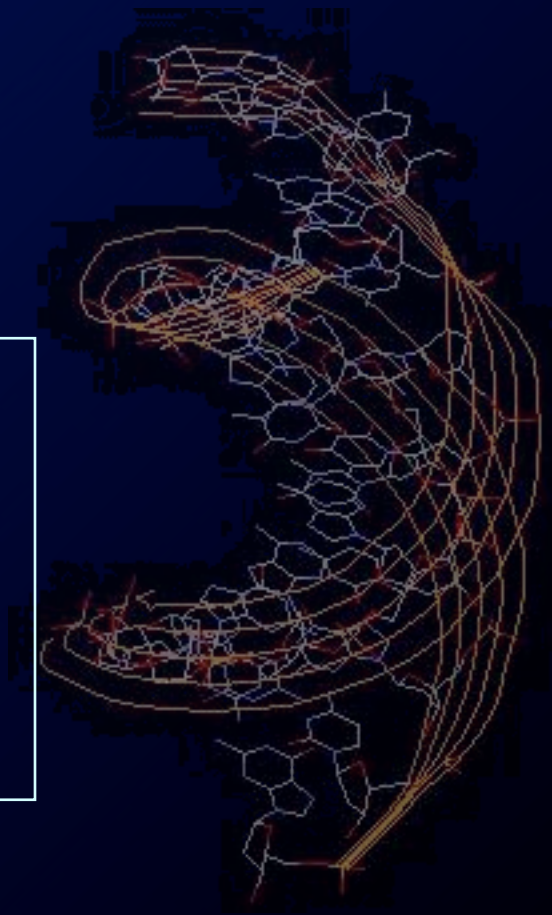
Swapping

Un processore a 32 bit può indirizzare 2^{32}=4 GB di RAM, superiore alla memoria fisica della massima parte dei computer.

La struttura a “memoria virtuale” suggerisce di utilizzare questa possibilità per utilizzare parte della memoria di massa (hard disk) per “simulare” memoria e contenere processi inattivi.

Questa opportunità va però usata con moderazione perché la lettura e scrittura su/da disco (swapping) di pagine di memoria è molto lenta e rischia di paralizzare il computer. In UNIX lo swap viene abilitato solo se la RAM libera è al di sotto di una soglia prestabilita.

Gestione I/O



I dispositivi di I/O sono di tipo molto vario, e presentano ciascuno peculiari caratteristiche => la gestione dell'I/O è fra i compiti più complessi di un OS.

Gestione dell' I/O (II)

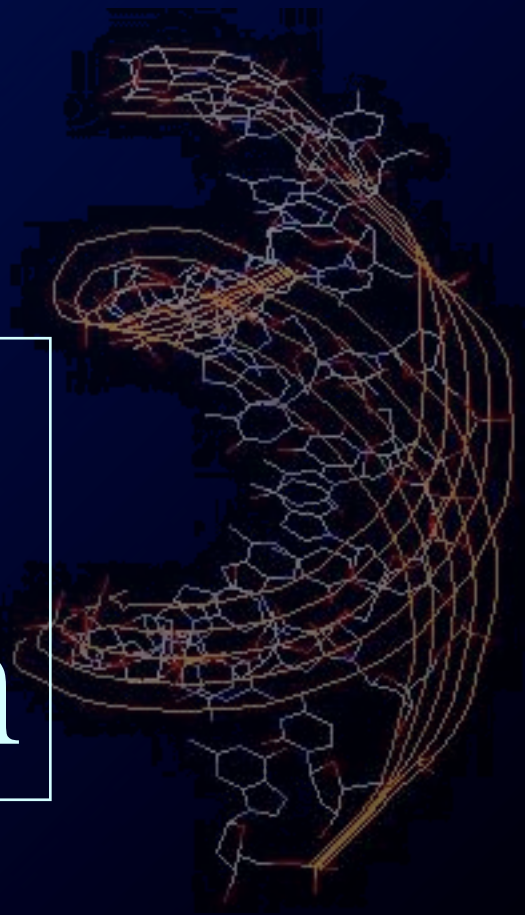
L'OS fornisce dei devices virtuali, che permettono ad esempio al programmatore di prevedere un'istruzione di tipo

```
printf("Hello world \n");
```

per stampare a video la frase Hello world, senza preoccuparsi se il monitor è un Philips 17" o un Sony e se è collegato su una scheda PCI o AGP…

A livello hardware l'OS poi gestisce fisicamente i devices tramite i device drivers. Uno dei compiti in assoluto più complessi e delicati è la gestione degli interrupts e l'assegnazione delle risorse, per evitare di incorrere in circoli viziosi (deadlocks).

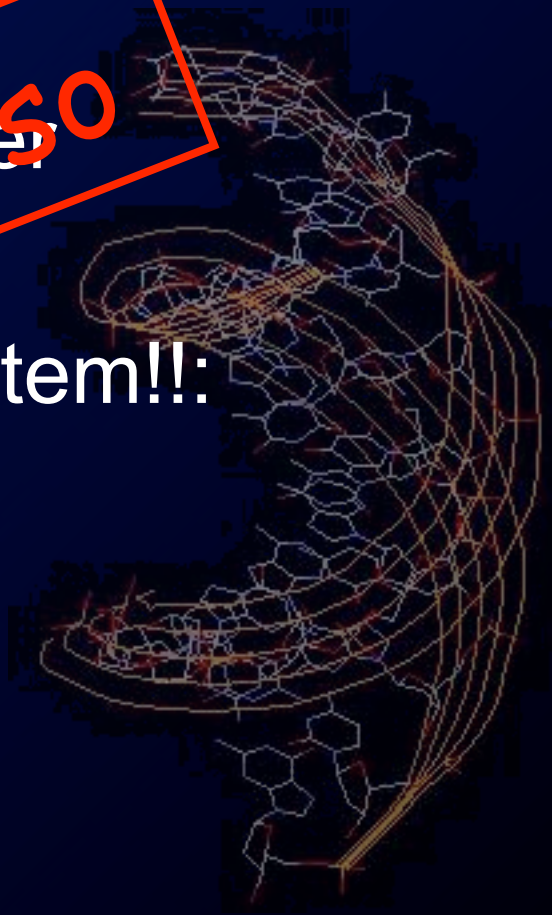
Gestione File System



Cos'è il File System

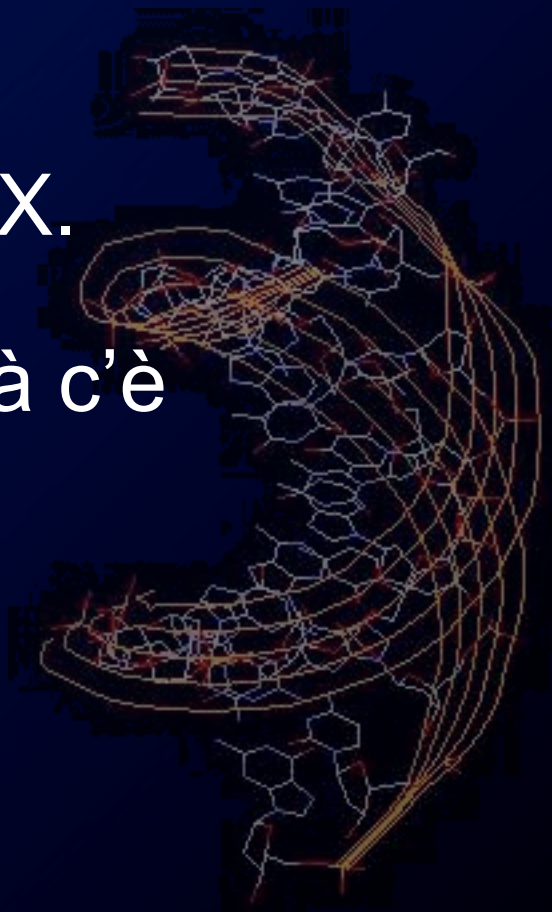
- E' l'insieme dei files nel computer

Falso



  - Nello UNIX tutto è file system!!:
    - files
    - drivers
    - periferiche
    - processi
    - etc..

Tutto è File System



- Questa la genialità dello UNIX.
- infatti per qualsiasi delle entità c'è bisogno delle stesse cose:
  - creare e distruggere
  - gestire l'accesso
  - scrivere o leggere dati
  - organizzazione gerarchica

Il file system

I servizi che un OS deve fornire per la gestione dei files sono, essenzialmente:

1. Fornire un metodo per associare un nome ad un file
2. Manipolare l'accesso, la creazione e la cancellazione dei files
3. Gestire la corrispondenza fra blocchi logici e blocchi fisici in modo trasparente all'utente
4. Realizzare meccanismi di protezione dei dati per determinare chi può avere accesso o controllo sui vari files.

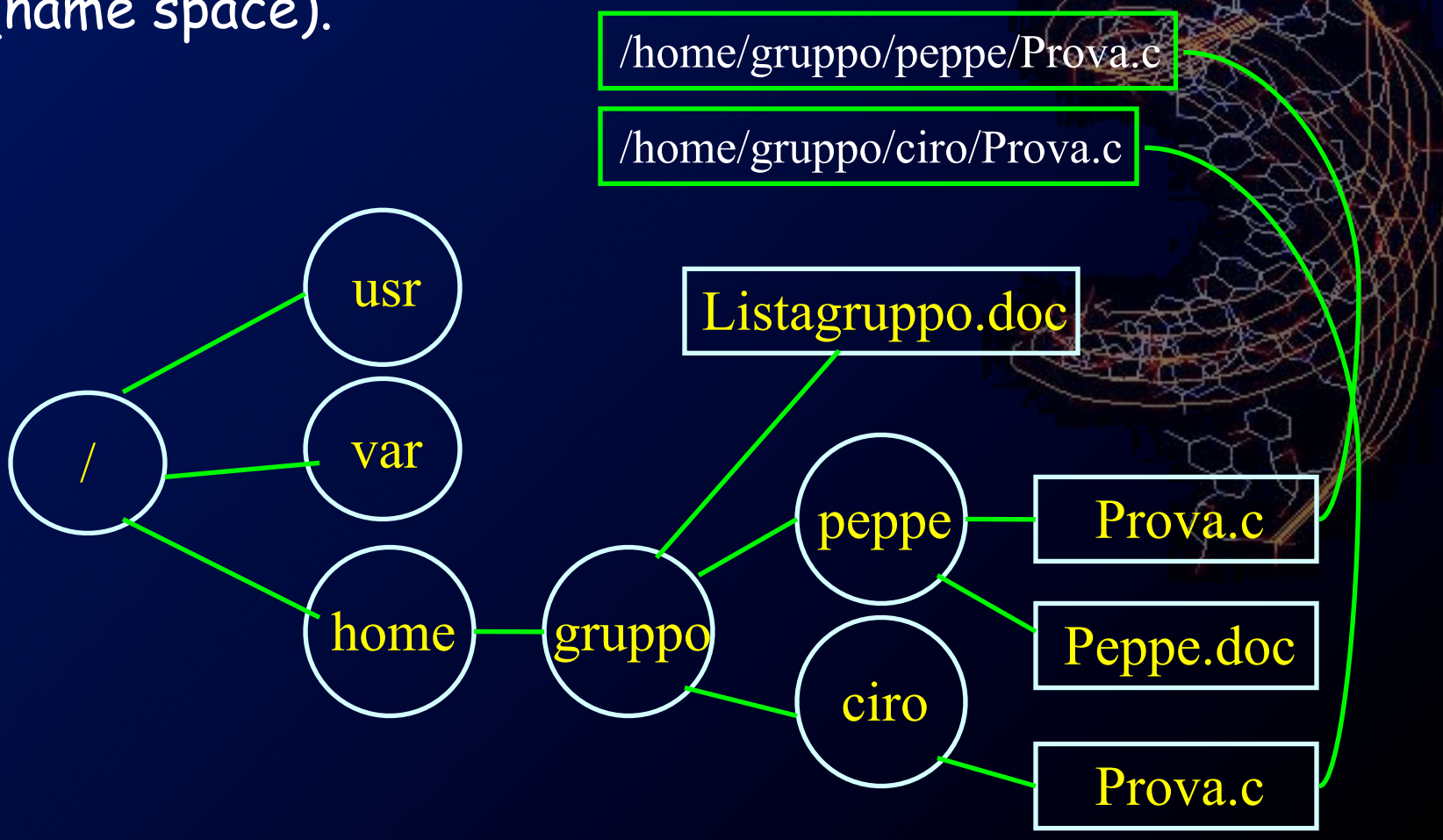
File & Directories

- File: contenitore logico di informazione immagazzinato nella memoria di massa. Può contenere programmi, testi, immagini, suoni, etc..

- Directory o cartella: area della memoria di massa che contiene files o altre cartelle (sottocartelle). Di fatto e' un file contenente l'elenco dei files appartenenti a quel gruppo e le infomazioni che permettono al s.o. di localizzare i files.

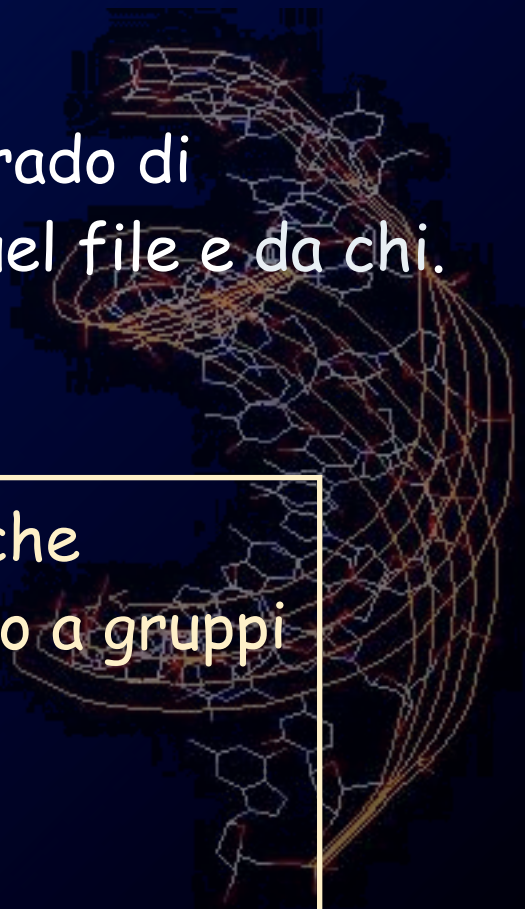
Directories e organizzazione gerarchica

Una directory è un particolare file, contenitore di files e directory. Essa definisce l'ambito di validità di un nome file (name space).



Controllo di Accesso

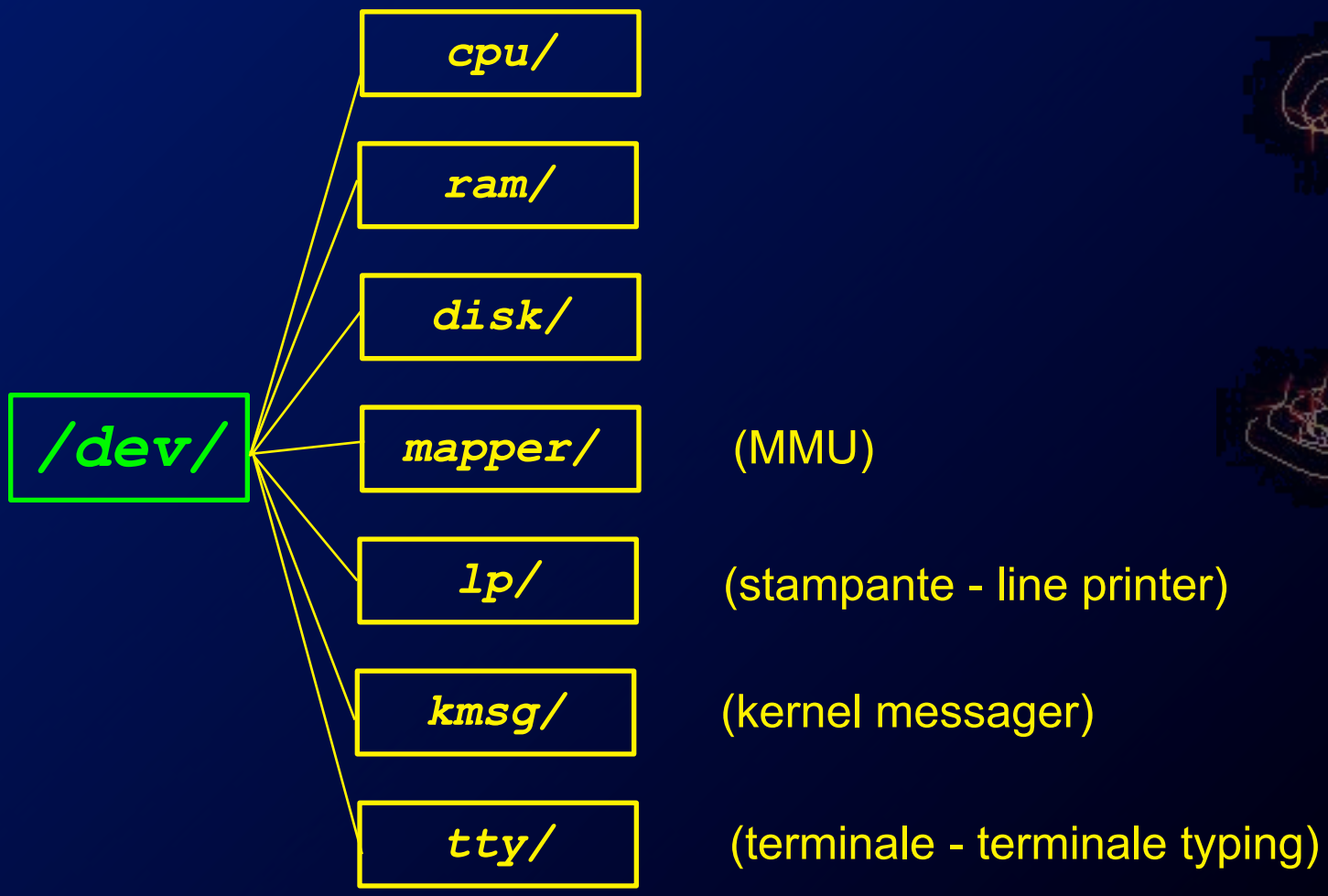
Il creatore di un file dovrebbe essere in grado di controllare ciò che può essere fatto con quel file e da chi.



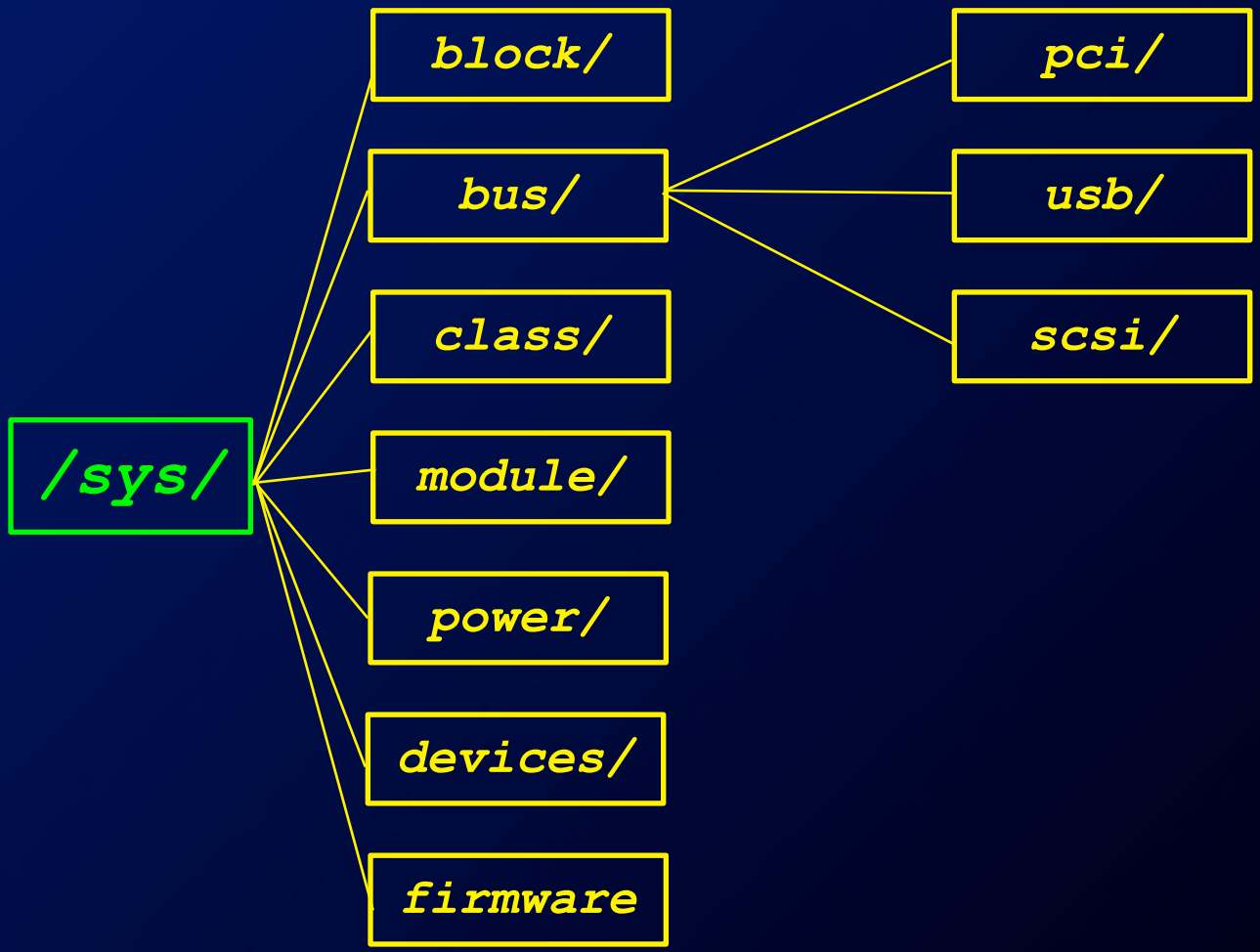
Sono possibili varie modalità di accesso, che possono essere permesse a singoli utenti o a gruppi di utenti:

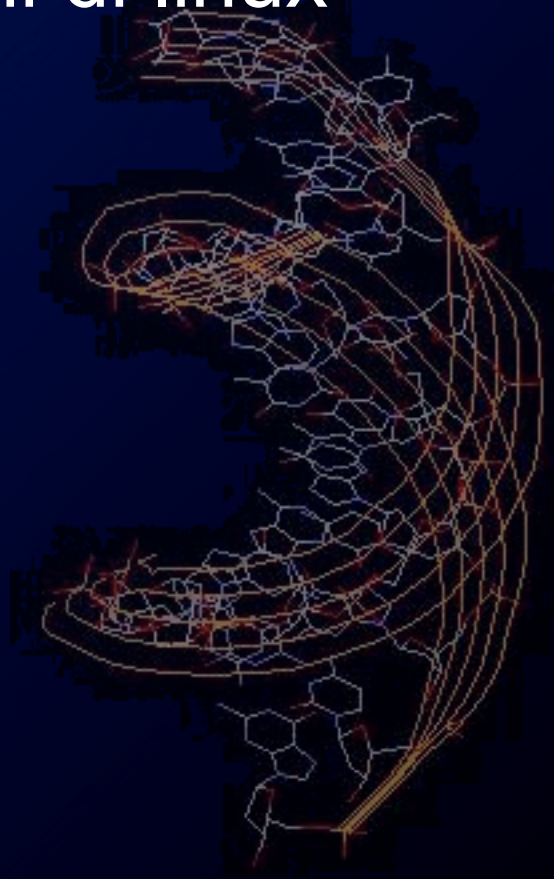
- Read Write Execute
- Append Delete List

- Ogni periferica di sistema (terminali, dischi, modem, etc..) compare come un file nell directory **/dev** (device)

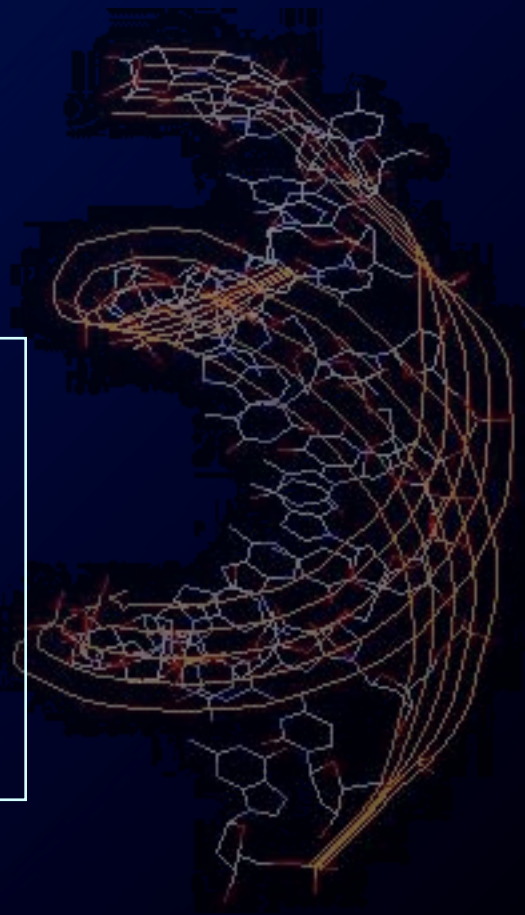


- Più in generale le ultime versioni di linux c'è addirittura **/sys**





Traduttori



Linguaggi ad alto e basso livello

Come visto la macchina di Von Neumann è capace di eseguire sequenze ben ordinate di istruzioni in linguaggio macchina del tipo:

```
0110100101000100-0010011000001101-0001…
```

```
Codice istruzione-indirizzo1…indirizzoN
```

L'istruzione deve essere presente nel set di istruzioni della ALU.

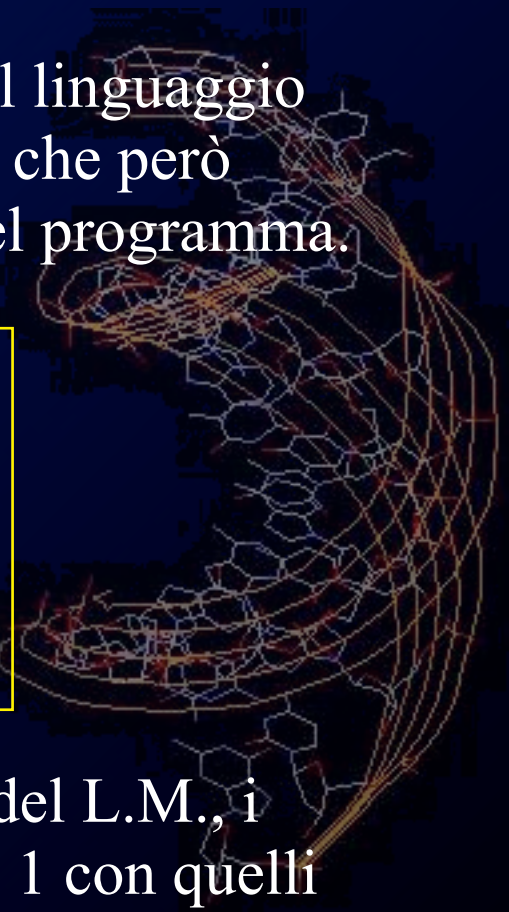
Un programma in L.M. è intrinsecamente legato all'hardware per cui è stato realizzato. Il L.M. è un linguaggio di *basso livello*.

I linguaggi che permettono di svincolarsi dalla rappresentazione hardware vengono chiamati linguaggi di *alto livello*.

Assembler

Il primissimo passo nella gerarchia dei linguaggi è il linguaggio assembler. Si tratta di un linguaggio di basso livello che però ammette una rappresentazione "human readable" del programma.

Linguaggio Macchina	Linguaggio Assembler
01000101010101001001	LOAD R1,X
00101010100010101001	LOAD R2,Y
11001010101010010010	ADD R3,R1,R2
11100101001010100100	STORE R3,Y



Il linguaggio assembler è una semplice traduzione del L.M., i comandi assembler ammettono una trascrizione 1 a 1 con quelli della ALU, e sono quindi specifici per ogni CPU.

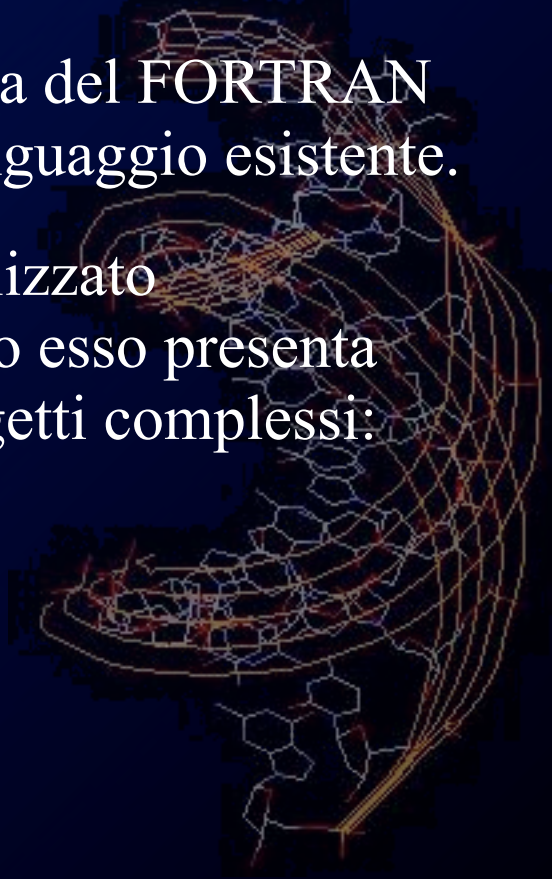
Un *assemblatore* è quella parte del sistema operativo che trascrive un programma in assembler nel suo corrispondente in L.M.

Linguaggi ad alto livello (I)

Agli albori della programmazione (fino alla nascita del FORTRAN alla fine degli anni ’50) l’assembler era l’unico linguaggio esistente.

Sebbene l’assembler possa in principio essere utilizzato direttamente per implementare qualsiasi algoritmo esso presenta numerose gravi limitazioni per un utilizzo in progetti complessi:

- Portabilità
- Leggibilità
- Riutilizzabilità
- Controllo degli errori



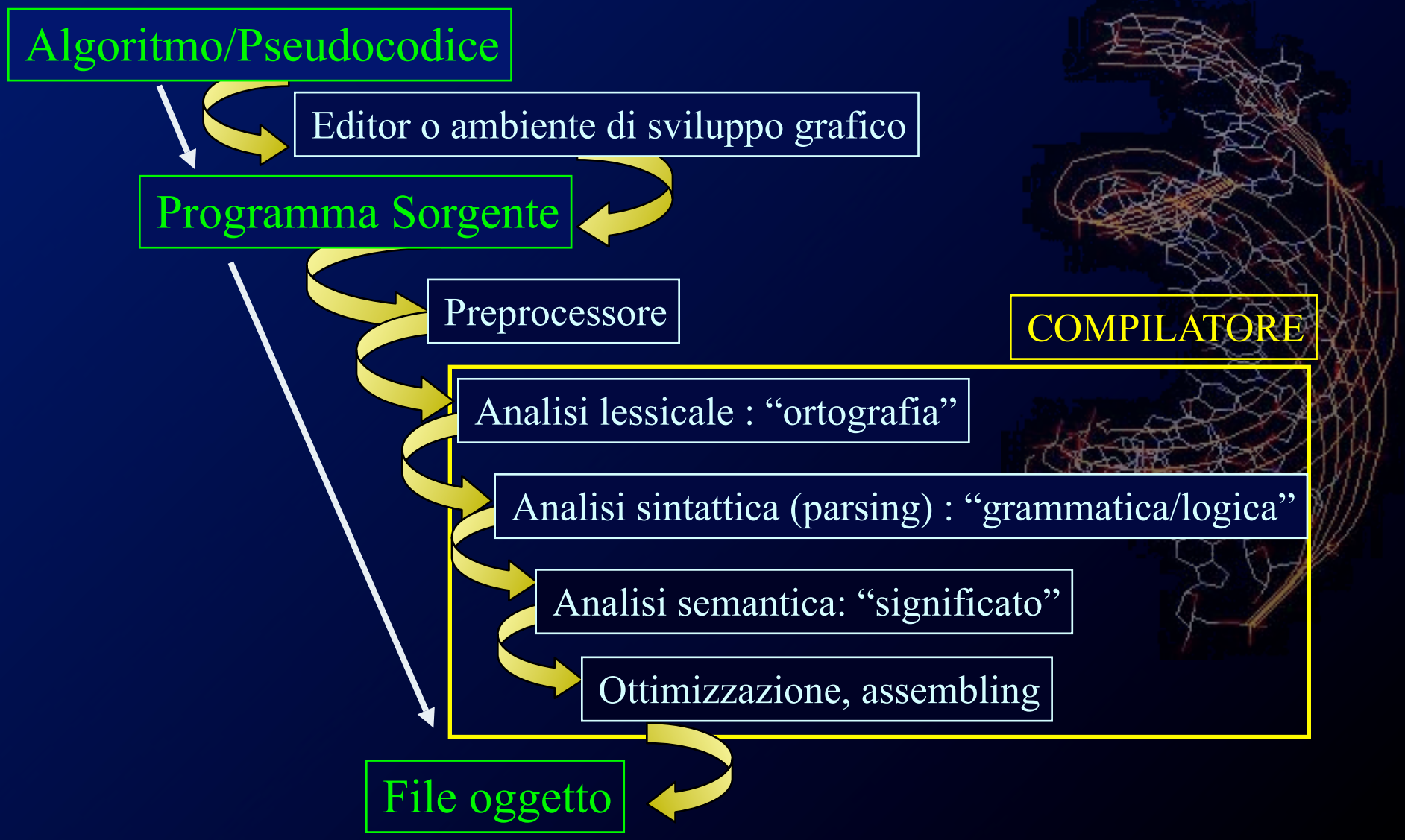
Questo ha portato alla creazione di linguaggi più vicini al linguaggio naturale e più lontani da quello della macchina….

Linguaggi ad alto livello (II)

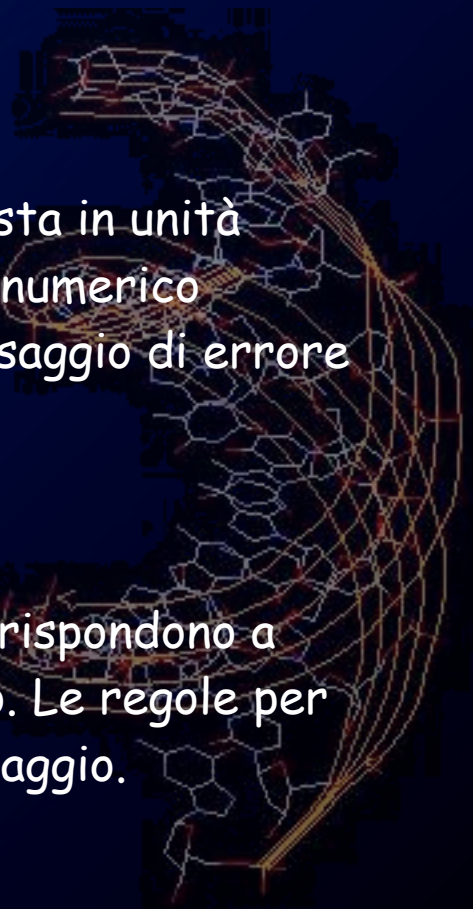
Un linguaggio di alto livello NON ammette una semplice traduzione 1 a 1 con le istruzioni della macchina, ma definisce costrutti e comandi specifici del linguaggio e indipendenti dalla macchina su cui devono essere eseguiti.

Il processo di traduzione in L.M. (l'unico veramente compreso dal calcolatore) è quindi di gran lunga più complesso che per un programma assembler.

Dall'algoritmo al processo (I)



Compilazione



Analisi lessicale (scanning):

la sequenza di caratteri del programma sorgente è scomposta in unità fondamentali (lessemi) caratterizzate da un identificativo numerico univoco (token). I simboli non riconosciuti generano un messaggio di errore e la fine della compilazione.

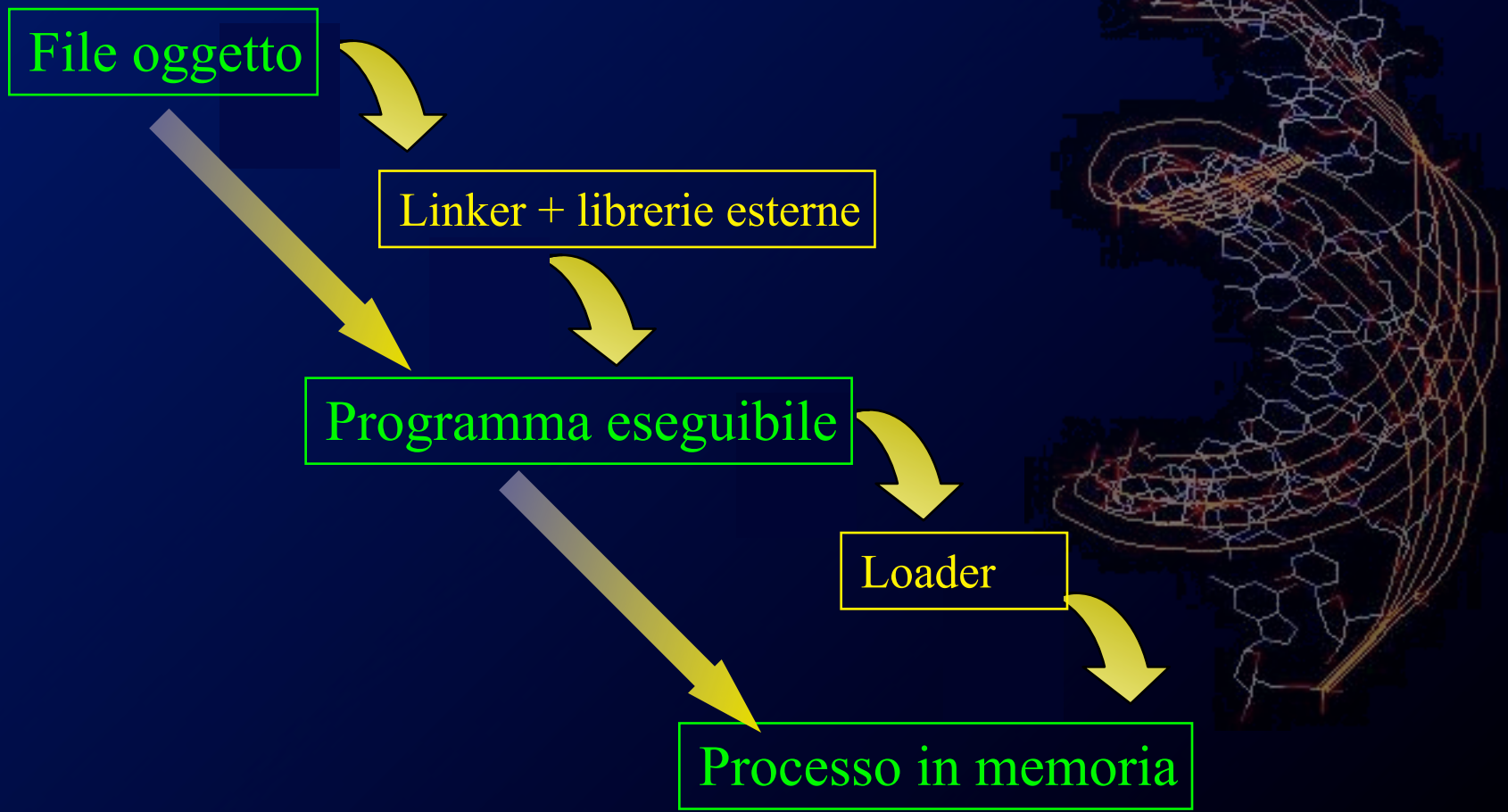
Analisi sintattica (parsing):

le sequenze di token vengono analizzate per vedere se corrispondono a costrutti sintatticamente definiti all'interno del linguaggio. Le regole per costruire questi costrutti formano la grammatica del linguaggio.

Analisi semantica:

le sequenze di istruzioni riconosciute dal parser vengono interpretate e si generano le sequenze di codice assembler necessarie per realizzarle.

Dall'algoritmo al processo (II)

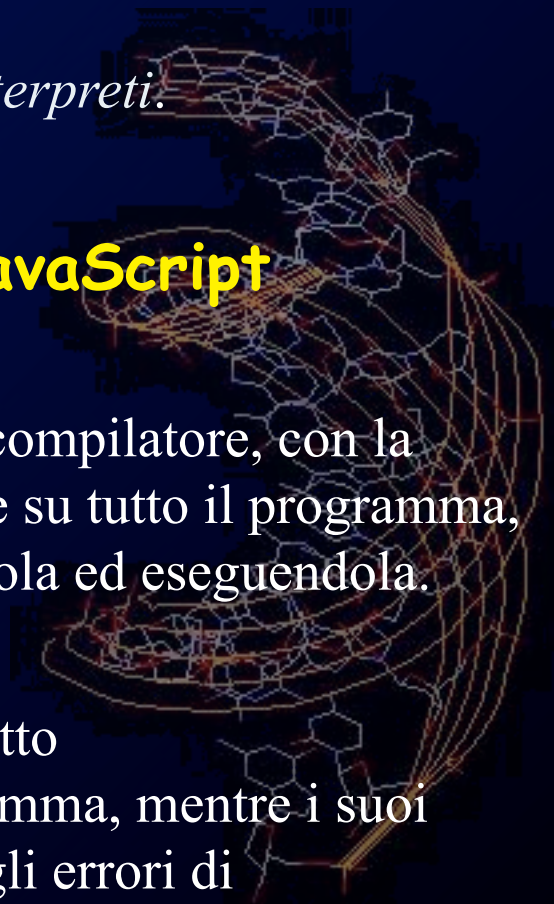


Interpreti

Oltre ai compilatori un altro tipo di traduttori sono gli *interpreti*.

Tra i linguaggi interpretati più utilizzati

HTML, perl, Python, PHP, Java, JavaScript



Per molti versi l'interprete svolge un ruolo simile ad un compilatore, con la differenza che mentre un compilatore agisce globalmente su tutto il programma, l'interprete lavora su una istruzione alla volta, traducendola ed eseguendola.

Lo svantaggio di utilizzare un interprete risiede soprattutto nell'impossibilità di ottimizzare l'esecuzione del programma, mentre i suoi vantaggi sono soprattutto la semplice localizzazione degli errori di programmazione e la possibilità di utilizzare direttamente il codice sorgente su macchine diverse senza dover effettuare nuovamente la compilazione del codice.

Linguaggi Compilati

- ottimizzabili
- velocità di esecuzione
- gestione dinamica della memoria
- più efficienti
- non portabili, legati all'architettura (hardware, librerie)

Linguaggi Interpretati

- non ottimizzabili
- lentezza di esecuzione
- mediamente efficienti
- alta portabilita', slegati dall'architettura