

Architetture degli Elaboratori - Sommario

Introduzione intensiva da 4h sulle architetture degli elaboratori

1. Rappresentazione delle Informazioni

Rappresentazione dell'Informazione nei Calcolatori

Rappresentazione dell'informazione nei calcolatori. Principio zero: l'esistenza di solo due simboli. Rappresentazione dei numeri, del testo e delle immagini in binario.

1. Principio zero (uno)

IL BINARIO. La *rappresentazione dell'informazione* nei *calcolatori* si basa su un *principio cardine*, ovvero che abbiamo *solo* due simboli: vengono (solitamente) denominati come 0, 1.

Ci sono state alcune eccezioni, come con i *computer sovietici* che avevano tre simboli, ma questo è un dettaglio storico minore.

Quindi consegue che *ogni tipologia di informazione*, tra cui numeri, foto, testo, documenti, istruzioni, eccetera... vanno *rappresentati* con questi due soli simboli. Ora vedremo come sarà possibile superare questo apparente limite del binario.

2. La rappresentazione dei numeri

I BIT E I BYTE. Definiamo un *bit* come una *singola cifra binaria* che può solo assumere *uno dei valori* 0, 1. Si definisce un *byte* come *otto bit*. Segue che ogni *byte* può rappresentare 2^8 numeri.

Dopodiché si può prendere i *multipli* del *byte*, tra cui il *kilobyte*, *megabyte*, eccetera...

Tuttavia è possibile ci sono due *tipi di multipli del byte*: uno è di base 10^n , l'altro è di base 2^{10n} . Dato che gli ordini di grandezza sono più o meno uguali, è possibile fare uno fraintendimento tra questi multipli.

Dunque indicheremo quelli con base 10^n con i prefissi *kilo*, *mega*, *giga*, *tera*, *peta*, ...; invece per quelli di base 2^{10n} vengono indicati con *kibi*, *mebi*, *gibi*, *tebi*, *pebi*.

(Notare che queste sono semplici convenzioni, e non sono necessariamente rispettate da tutti!)

I numeri Naturali

I NUMERI NATURALI. Ora vediamo come "tradurre" un byte (o più) in un numero naturale. Generalmente, supponendo che un numero binario è della forma

$$\langle x_1 x_2 \dots x_n \rangle$$

Allora possiamo ottenere il suo numero *in base decimale* facendo il seguente calcolo:

$$\sum_{i=1}^n x_i 2^{n-i}$$

Ovvero "*sommiamo le potenze di 2^n , a seconda della posizione delle cifre*".

Si può usare altre basi, tra cui l'*ottale* e l'*esadecimale*.

OPERAZIONI TRA NUMERI IN BINARIO. Per sommare *due numeri in binario*, lo si fa come facciamo di solito in *decimale*, solo che abbiamo solo *due cifre* (quindi abbiamo più riporti del solito).

Ad esempio,

$$1101\ 1001 + 0010\ 0011 = 1111\ 1100$$

Similmente si può definire la *moltiplicazione di numeri in binario*.

OVERFLOW. Notiamo che con queste nozioni, abbiamo una limitazioni: queste operazioni (in somma) hanno un range possibile tra $[0, 2^{n-1}]$. Ma cosa succede se tentiamo di sommare "*due numeri troppo grandi*"? Ovvero se tentiamo di sommare ad esempio

$$1111\ 1111 + 1111\ 1111 = ?$$

Qui abbiamo infatti il cosiddetto "*overflow*": tentando di sommare due numeri tali che la somma venga maggiore di 2^{n-1} , non abbiamo abbastanza bit per rappresentare il nuovo numero.

Dunque si fa la somma come di consueto, troncando via le parti "*in eccesso*". In questo caso ho

$$\begin{array}{r} 1111\ 1111+ \\ 1111\ 1111 = \\ 1111\ 1110 \end{array}$$

Quindi paradossalmente stiamo "*sottraendo*" il primo numero con 1!

I numeri Interi

I NUMERI INTERI. Adesso, vogliamo rappresentare anche i *numeri negativi*, dal momento che potrebbero risultare utili per certi contesti, tra cui i debiti, conti negativi, eccetera...

Storicamente ci sono stati più approcci per formalizzare i *numeri negativi* nel *binario*, tra cui quello del *segno e modulo*, del *complemento-due* e dell'*eccesso N*.

SEGO E MODULO. Per sostituire il segno – del numero negativo, basta usare *uno dei bit* del numero in binario per rappresentare il segno. Ovvero, se ad esempio ho

1000 1011

questo non si legge più come $2^7 + 2^3 + 2^1 + 2^0$, bensì come $-\cdot(2^3 + 2^1 + 2^0)$.

Questo sembra un buon approccio, dal momento che ampliamo il *range dei numeri rappresentabili* in $[-2^{n-1} - 1, 2^{n-1} - 1]$. Tuttavia ci sono due problematiche:

- In questo modo abbiamo due rappresentazioni del numero zero: ovvero 1000 000 e 0000 000, così in un modo "*sprechiamo*" uno slot.
- La somma tra *numeri positivi e negativi* non è più definibile in una maniera intuitiva: bisogna vedere dei *casi specifici* per poter sommarli.
Tutto sommato, questo approccio è quello "*naïve*", dal momento che l'idea sottostante è molto semplice.

IL COMPLEMENTO 2. In questo caso il *primo bit* diventa -2^{n-1} , ovvero adesso il numero lo leggiamo come

1000 0000 $\rightarrow -2^7$

E il restante delle caselle vengono lette come di consueto. In questo modo abbiamo molti vantaggi:

- Il range dei numeri diventa $[-2^n, 2^{n-1} - 1]$
- Il primo bit indica comunque il *segno* (o la "*negatività*") del numero
- Possiamo sommare numeri positivi e negativi normalmente

ECCESSO N. Qui si tratta di "*sostituire il zero con un numero negativo*", ovvero di "*contare da un numero più basso*". Ad esempio, impostando $N = 128$, leggiamo il numero 0 come -128 . Allora si ha

1000 000 = 0
1111 111 = 127

I numeri razionali

I NUMERI RAZIONALI (O REALI). Adesso voglio rappresentare i numeri *razionali*, ovvero quelli con la *virgola*; oppure vogliamo rappresentare pure i *numeri reali*, anche se

sarebbe realisticamente impossibile per la *densità dei razionali nei reali*: ci servirebbero una quantità infinita di bit!

Quindi è necessaria una *buona approssimazione* per questi numeri, dal momento che una *vera e propria fedele rappresentazione* sarebbe fisicamente impossibile.

Storicamente ci sono state molte convenzioni per rappresentare questi numeri, oggi si usa lo *standard IEEE 754*, che ha come "*idea di base*" quella di "*simulare*" la *notazione scientifica*: si ha un *bit per il segno*, dei *bit* per l'*esponente delle cifre significative* in 2^n che viene moltiplicata per la *mantissa*.

Questo standard ci offre più "*tipologie di numeri*" con certi *livelli di precisione*, tra cui i numeri a *precisione singola e doppia*.

Inoltre, questa convenzione ci offre una possibilità per definire l'*infinito* $\pm\infty$ e il risultato di un'operazione *indefinita* NaN.

Per una visualizzazione di questa convenzione, vedere il sito

<https://bartaz.github.io/ieee754-visualization/>.

3. La rappresentazione del testo

IL TESTO. Per quanto riguarda invece il *testo*, storicamente ci sono state più convenzioni per "*trasformare*" i *bit* in caratteri.

Inizialmente c'è stata la convenzione *ASCII*, dove si usava un *byte* per rappresentare un singolo carattere. Si nota che in realtà *sette bit* erano già sufficienti per rappresentare tutti i caratteri nell'alfabeto anglo-sassone (ovvero latino escludendo gli accenti), quindi c'era un "*bit in eccesso*".

Da qui si hanno più "*varianti*" di *ASCII*, dove si usa il *bit in eccesso* per rappresentare altri caratteri, tra cui *lettere accentate, nuovi caratteri caratteristici della lingua*, eccetera...; il variante dipende dal *linguaggio di riferimento*.

Tuttavia questa *molteplicità* rappresenta un limite dell'*ASCII*, dal momento che in questo modo diventa *impossibile* scrivere documenti in *più linguaggi*. Oppure con l'*ASCII* rimane comunque impossibile *rappresentare* la scrittura di certi linguaggi, come ad esempio quello del *giapponese, cinese*, ...

Quindi oggi si usa la convenzione *Unicode*, che non solo comprende *una buona parte dei linguaggi*, ma anche gli *emoticon*.

4. La rappresentazione delle immagini

LE IMMAGINI. Qui basta considerare la convenzione *RGB*, dove si usa una *terna* di *byte* per rappresentare l'*intensità della luce* per il colore *rosso, giallo* e *blu*.

2. Architetture Base dei Calcolatori

Architettura Base del Calcolatore

Architettura base (generalizzata) di un calcolatore. Esecuzione di un programma: linguaggio di programmazione ad alto livello, linguaggio assembly e codice macchina.

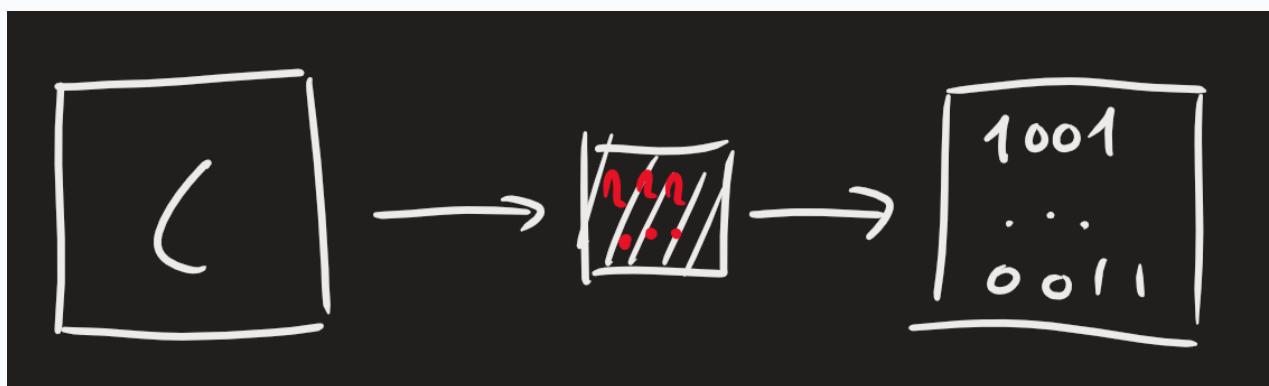
1. Il black box

IL BLACK BOX. Sapendo che *tutto dev'essere codificato nel linguaggio binario* nel calcolatore, vogliamo capire come il nostro *computer* è in grado di *eseguire le istruzioni*, che sono chiaramente scritte in un linguaggio *non binario* (come ad esempio **C** oppure **Python**).

Infatti quando vogliamo eseguire un programma, dobbiamo *convertire* il *codice sorgente* del programma in un linguaggio che sia leggibile dalla macchina: tuttavia questo processo non è esplicitamente visibile, dal momento che possiamo vedere solo l'*input* e l'*output*. Quindi c'è questa specie di "*black box*" che converte il codice sorgente in codice macchina.

Il nostro obiettivo è di chiarire questo "*black box*".

FIGURA 1.1. (*Il black box*)

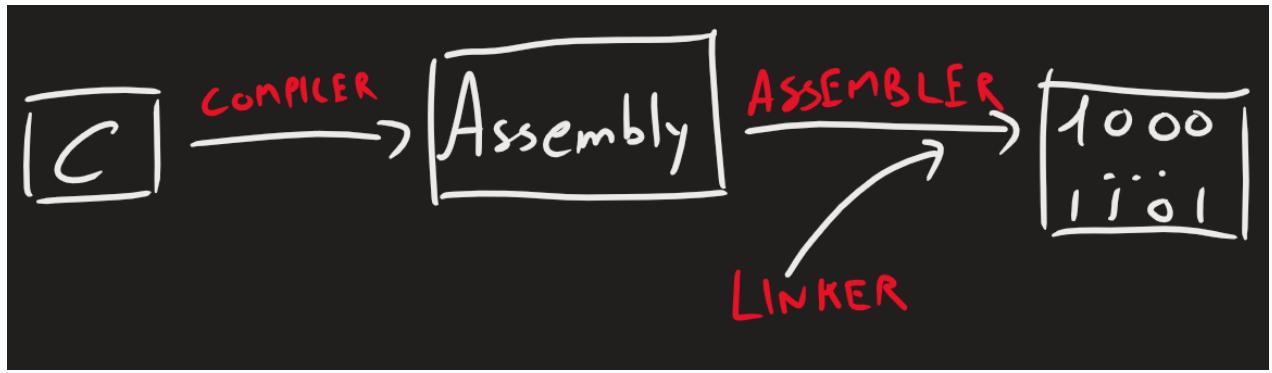


2. Compilatore e Assembler

IL COMPILATORE. Ci ricordiamo che il linguaggio C è un linguaggio di programmazione che va *compilato*, ovvero trasformato in un linguaggio di programmazione che è più "*leggibile*" dal computer. Stiamo parlando del linguaggio *Assembly*, che è *poco astratto*. Tuttavia, le istruzioni che caratterizzano questo linguaggio dipende dalla *singola architettura dell'elaboratore* (ovvero del *processore*). Infatti ci sono *molti* approcci diversi per il linguaggio *Assembly*. Quindi questo linguaggio si dice *non portatile*.

L'ASSEMBLER. Tuttavia il linguaggio *Assembly* non è sufficiente per essere compreso dalla macchina, dal momento che rimane ancora testo. Vogliamo quindi trasformare un'istruzione in una singola sequenza di cifre 0, 1: su questo processo interviene l'*assembler* per trasformare le istruzioni in valori numerici (ovvero in *forma statica*) e interviene anche il *linker* per le *librerie*. Per vedere più dettagli su questo processo, vedere il sito <https://godbolt.org/>.

FIGURA 2.1. (*Schema generale della conversione di un programma*)



3. Architettura di Von-Neumann

Architettura di Von-Neumann

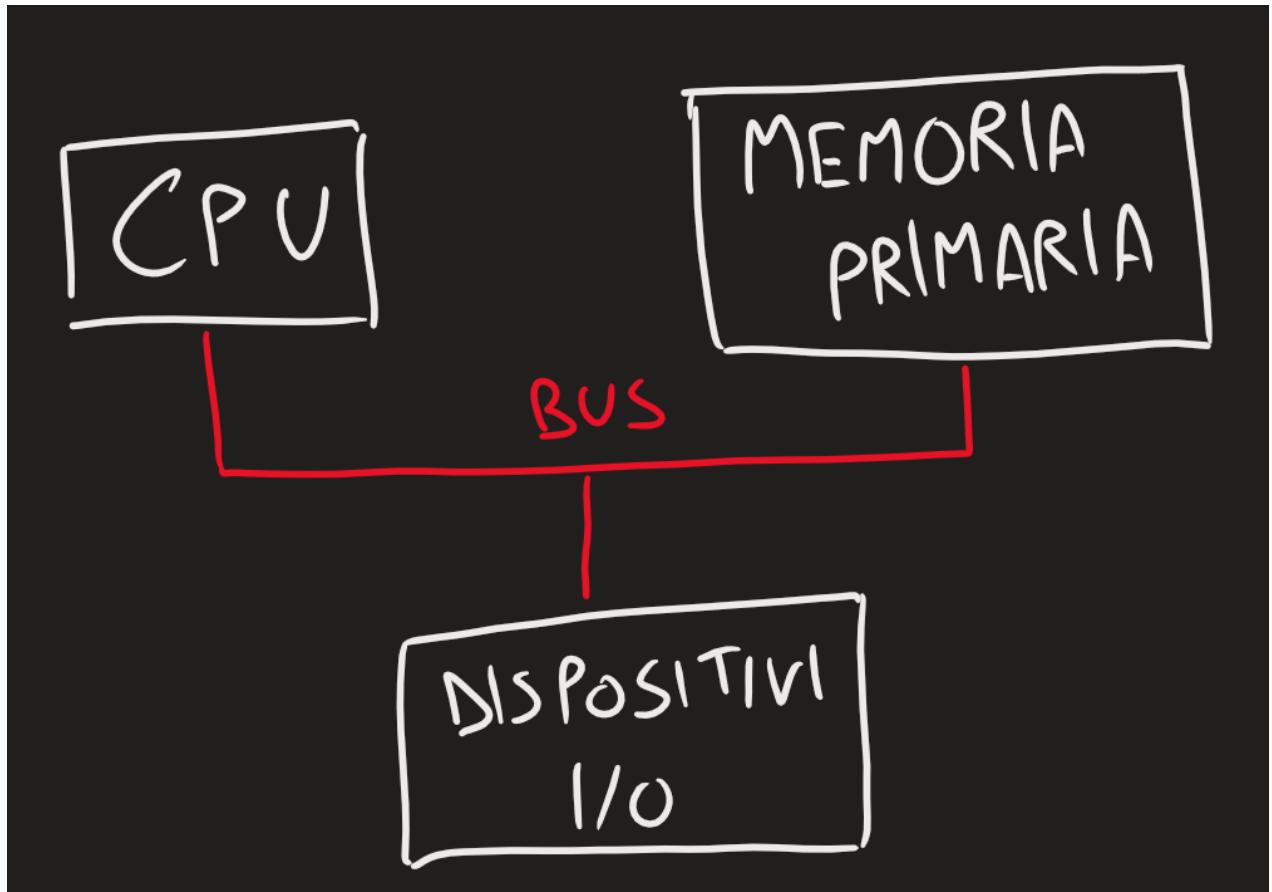
Caso specifico di un calcolatore generico: l'architettura di Von-Neumann. Struttura base di un calcolatore: CPU, Memoria primaria, periferiche e bus. Strutture specifiche di ogni singolo componente. Ciclo fetch-decode-execute della CPU.

1. Layout di un Architettura di Von-Neumann

IL LAYOUT GENERALE. L'architettura di Von-Neumann è una *disposizione specifica* di un *calcolatore*, ed è formata principalmente da quattro componenti:

- La *CPU*, ovvero la *Central Processing Unit* che *esegue* le istruzioni
- La *memoria primaria*, della quale l'unicità è caratteristica dell'*architettura di Von-Neumann*; contiene sia *dati* che *istruzioni*
- Le *periferiche*, ovvero i *dispositivi I/O* che permettono l'interazione del calcolatore con l'*esterno*: comprendono la *memoria di massa*, *dispositivi di input e output* come la *tastiera* o il *monitor*
- Il *bus* che mette in *comunicazioni* le diverse componenti appena viste.

FIGURA 1.1. (*Architettura di Von-Neumann*)



2. Strutturazione specifica dei componenti

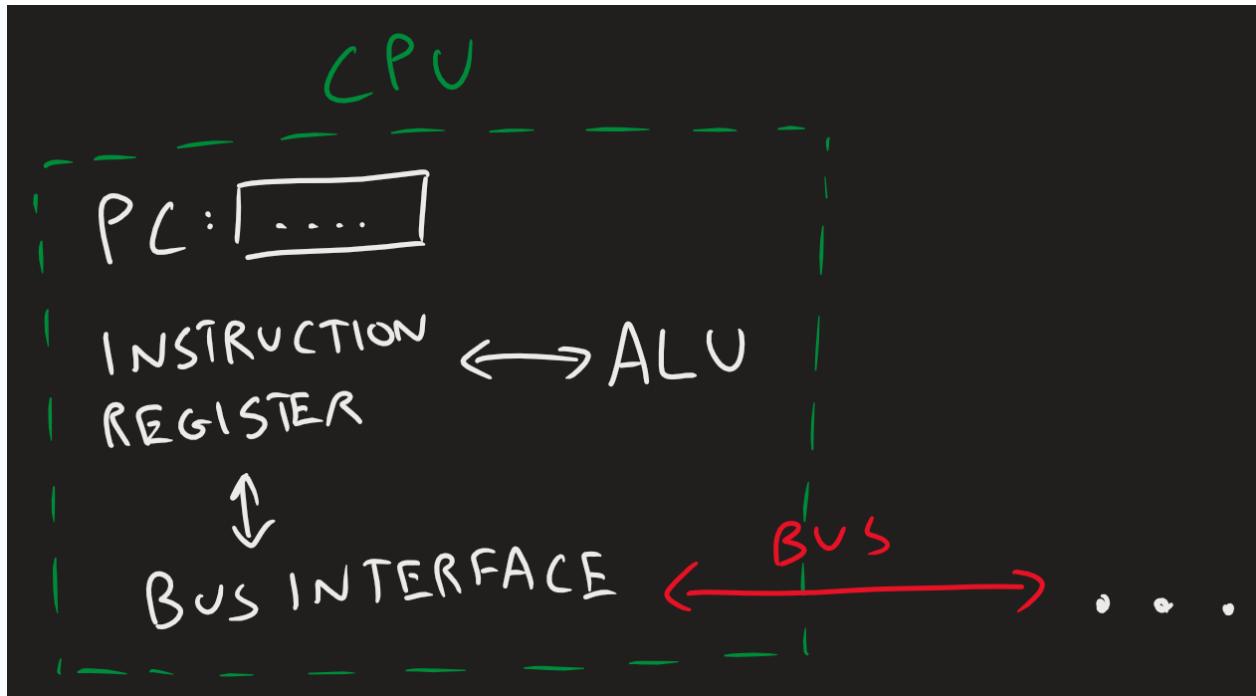
CPU. Specificamente la **CPU** è formata da altri componenti interni.

- Il *Program Counter*, abbreviato come **PC**: serve a *tenere conto dell'istruzione da eseguire* in un programma. In parole brevi, controlla il *flusso di esecuzione* di un programma.
- Il *Register file*, che tiene conto delle istruzioni.
- L'*ALU*, che esegue i singoli *calcoli aritmetici*.
- Il *Bus Interface*, che interfaccia il *register* con il *bus*.

MEMORIA PRIMARIA. La memoria primaria è organizzata in *modo lineare*, come uno *stack* di *locazioni di memoria* associati a degli *indirizzi di memoria*. Ogni *locazione di memoria* contiene un singolo *valore* numerico.

DISPOSITIVI I/O. I dispositivi **I/O** permettono di consentire la *comunicazione* del *calcolatore* col *mondo esterno*. Si può trattare ad esempio di *dati in arrivo*, che possono venire dalla *tastiera* o dal *mouse*; similmente si può parlare anche di *dati in uscita*, che sono visualizzabili col *monitor* o periferiche simili. Oppure si può trattare anche di un'*archiviazione larga di dati*, ovvero di "*mass storage*", che può essere fatto con componenti come l'*HDD*, l'*SSD*, ...

FIGURA 2.1. (*Struttura particolare della CPU*)



3. Ciclo fetch-decode-execute della CPU

IL CICLO. La **CPU** segue delle istruzioni, e lo fa effettuando un ciclo di **tre operazioni**: fetch, decode e infinite execute.

FETCH. In questo stato il processore *incrementa* il **PC** (*program counter*), ricava l'*istruzione* dalla *memoria primaria* e salva questa istruzione nell'*instruction register*.

DECODE. In questo processo la **CPU** interpreta il valore numerico in un'istruzione, il cui significato cambia al *variare dell'architettura*.

EXECUTE. In questo ultimo stato il processore esegue le operazioni contenute nell'*istruzione*, avvalendosi dell'*ALU*.

4. Architettura ARM

Architettura ARM

Esempio specifico-pratico di architettura Von-Neumann: ARM. Notizie storiche, introduzione generale, struttura di ARM-v7a, concetto di registri.

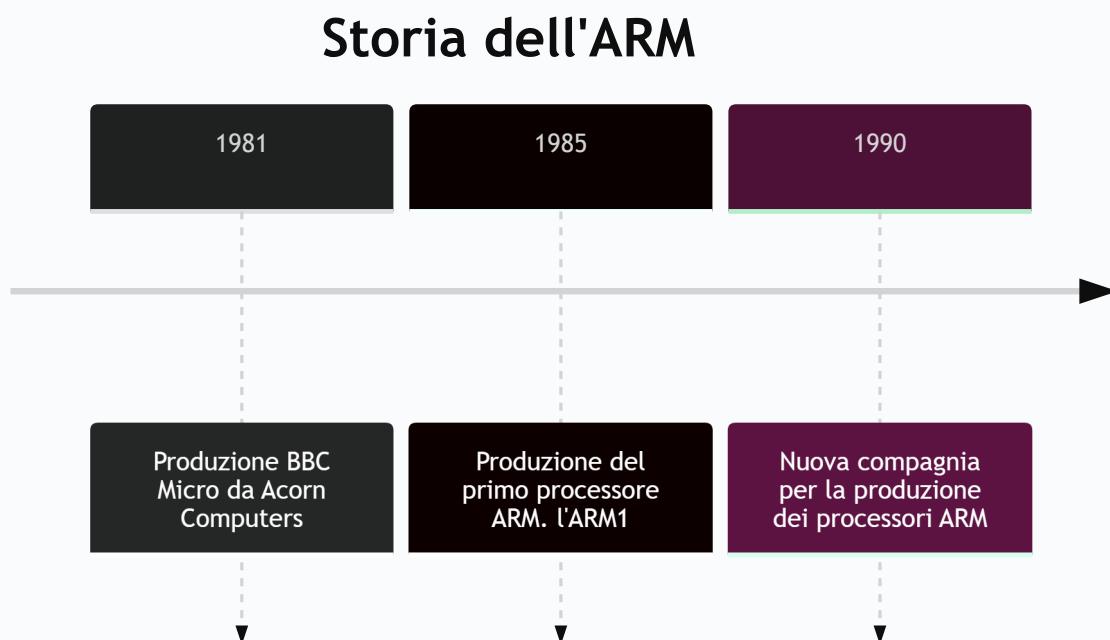
1. Introduzione Generale

UN ESEMPIO PRATICO. Finora abbiamo solo visto *strutture generali di architetture*: abbiamo capito come un *calcolatore* legge un *programma*, e una struttura *particolare* di un tale calcolatore. Adesso vediamo un esempio *pratico* di *un'architettura*: l'ARM,

presente ancora oggi nei nostri dispositivi, tra cui cellulari, tablet, portatili, ...

NOTIZIE STORICHE. Il primo *processore* dell'*architettura ARM* venne prodotto nel **1985**, dalla *Acorn Computers*, una società britannica. Dopodiché la progettazione dei *processori* venne delegata ad una diversa compagnia, nel **1990**. Attualmente l'*ARM* non produce più i processori, ma li progetta e si limita a *distribuire i diritti di proprietà intellettuale*. Nel corso della storia, il significato di *ARM* è passato a "*Advanced Risc Machine*". Ma che vuol dire "*Risc*"? Lo vedremo con i *paradigmi CISC-RISC* delle istruzioni (*Paradigmi CISC e RISC*).

FIGURA 1.1. (*Linea temporale*)

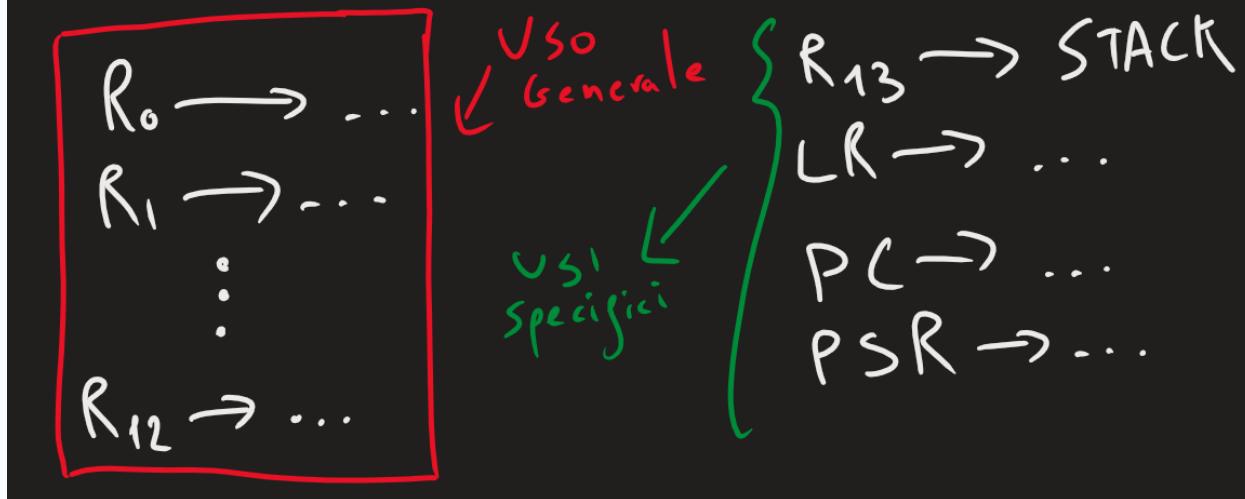


2. Struttura di Arm-v7a

Arm-v7a. Ora vediamo una *versione specifica* dell'ARM. In questa versione dell'*ARM* abbiamo *tredici registri* per l'*uso generale*, ognuno dotato di *32 bit* (ovvero *4 byte*). Dopodiché ci sono altri *quattro registri specifici*, tra cui *due* sono usati per *lo stack delle chiamate*, uno per il *program counter* e uno per il *program status register*.

FIGURA 2.1. (*Arm-v7a*)

Arm - V7a



5. Paradigmi CISC e RISC

Paradigmi CISC e RISC

Paradigmi CISC e RISC per le istruzioni del linguaggio Assembly. Filosofia generale degli approcci, svantaggi e vantaggi e spiegazione della necessità storica di due paradigmi diversi.

1. Introduzione Storica

INTRODUZIONE STORICA. Nel corso della storia sono stati sviluppati *due approcci* per codificare le *istruzioni Assembly*: una è il **CISC** (*Complex Instruction Set Computers*) e l'altra è **RISC** (*Reduced Instruction Set Computers*). La più tradizionale è la **CISC**, con le architetture Intel negli anni '70, e la più nuova è la **RISC**. Nei paragrafi successivi si vedrà il motivo.

2. RISC e CISC

CISC. Il *primo paradigma* è il **CISC**, da cui il nome ci dice che si vuole svolgere *tante operazioni complesse*: questo semplifica tantissimo la vita dei *programmatori Assembly*, dato che devono scrivere meno istruzioni! Tuttavia, con questo approccio si ha un *tempo di esecuzione* più lento (in particolare le istruzioni richiedono più di un *ciclo di clock*) dato che le istruzioni diventano *complesse* da *decodificare*. Questo è stato l'approccio preferito dall'inizio, dato che il *compilatore* non esisteva, oppure non era ancora abbastanza *"affidabile"*.

RISC. Con l'avvento dei compilatori, si vuole iniziare a semplificare tutto: tanto con il linguaggio di programmazione C, non bisognava trattare più con registri! Con questo approccio si vuole dare un "set" di istruzioni poche ma semplici, e di solito ogni istruzione ha un suo solo compito preciso. Nell'avvenire del tempo, l'approccio RISC è diventato più popolare dal momento che è più veloce e che non serviva più programmare in Assembly.

6. Linguaggio Assembly

Linguaggio Assembly

Funzionamento base del linguaggio Assembly: codifica delle istruzioni, program status register. Alcune istruzioni ARM del linguaggio Assembly. Istruzione MOV, ADD, SUB, B, BL, CMP, BEQ, LOAD, STORE. Chiamate di funzione con Assembly.

1. Codifica delle Istruzioni

LA CODIFICA DELLE ISTRUZIONI IN ASSEMBLY. Come detto prima, ogni informazione dev'essere rappresentata come un numero (in particolare binario). Questo vale anche per le istruzioni del linguaggio Assembly (in particolare v7a): ogni istruzione occupa esattamente 4 byte. Ora vediamo i "ruoli" di questi bit individuali

BIT 1-6. Sono dei bit di cui possiamo non occuparcene, dato che si trattano di dettagli.

BIT 7. (Segno) Questo bit è una specie di "flag" che ci dice se questa "stringa" binaria di numeri tratta il secondo operando come un registro o un immediato.

BIT 8-11. (Opcode) Indica l'operazione da eseguire

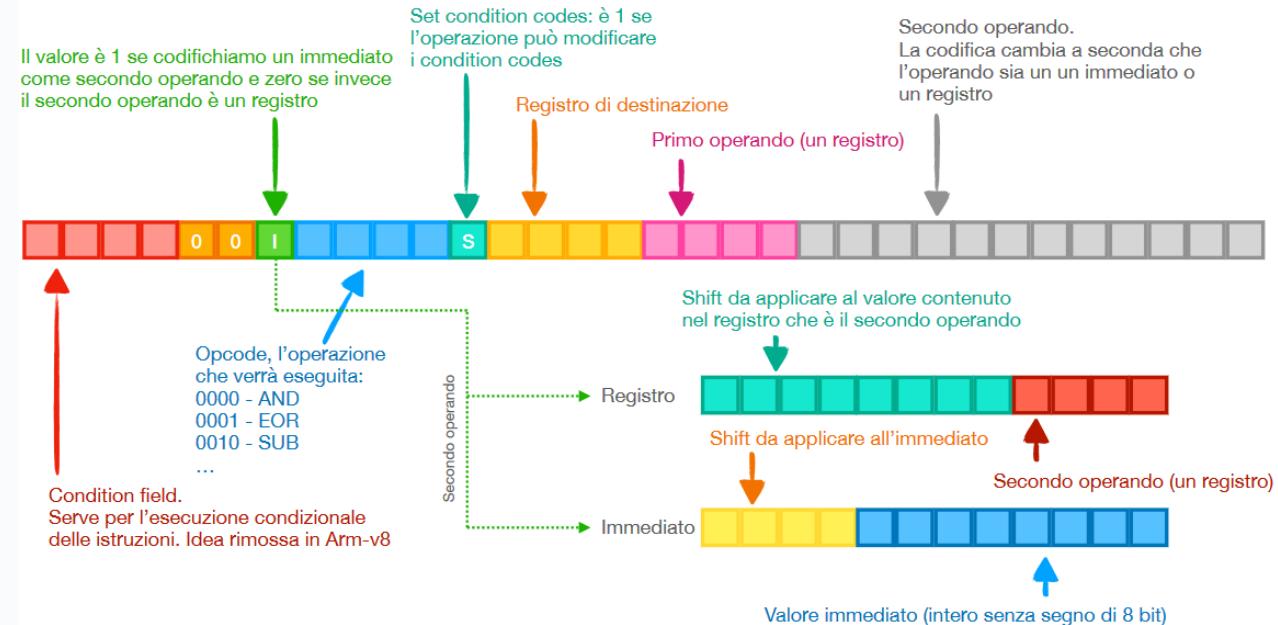
BIT 12. (Set condition codes) Dettagli.

BIT 13-16. (Registro di destinazione) Indirizzo del registro in cui salvare il risultato del calcolo.

BIT 17-20. (Primo operando) Sempre un registro

IL RESTO DEI BIT. (Secondo operando)

FIGURA 1.1. (Schema riassuntivo)



2. Program Status Register

L'IDEA. Il program status register è utile per *tenere conto* di operazioni di *comparazione*. I bit di questa componente verrà controllata dai comandi *branch condizionali*. In particolare tengono conto di quattro risultati: *negative result*, *zero result*, *carry set* e *overflow*.

3. Istruzioni Assembly

MOV. L'istruzione **MOV** permette di copiare un *valore* da un *registro* a un *altro registro* (oppure inserire un valore numerico in un registro). Viene scritta come **MOV dest, source**.

ADD. L'istruzione **ADD** permette di *sommare due valori* contenuti in due registri, o di *sommare un valore di un registro* con un *valore numerico*. Viene scritta come **ADD dest, op1, op2**.

SUB. L'istruzione **SUB** è analoga all'istruzione **ADD**, solo che al posto di *aggiungere numeri* si *sottrae* dei numeri.

BRANCH. L'istruzione branch **B** ci permette di saltare ad un'altra istruzione nel codice, che viene contrassegnata con un *label*. Viene scritta come **B label**. Notare che con *sola* questa istruzione, si può avere solo *cicli infiniti*.

CMP. L'istruzione **CMP** ci permette di comparare *due numeri* contenuti in *due registri* (o un numero di un registro con un valore numerico). Dal momento che *non serve* di nessuna destinazione di scrittura (infatti l'esito viene automaticamente salvato nel *PSR*), viene scritta solamente come **CMP op1 op2**.

BEQ. L'istruzione **BEQ** è *una delle istruzioni* di *branch condizionale*, ovvero del *branch* effettuata solo in certi casi. In questo caso, **BEQ** effettua un branch solo nel caso

dell'uguaglianza di due valori precedentemente comparati con l'istruzione **CMP**.

Di altri tipi ce ne sono **BNE** che verifica la *disuguaglianza* di due valori.

LOAD. L'istruzione **LDR** permette al calcolatore di *accedere* alla *memoria del lavoro*.

Dato che gli *indirizzi della memoria del lavoro* sono diversi da quelli dei *registri* (infatti se fossi così, avremmo solo 2^{24} locazioni di memoria che equivalgono a circa *16 MB*), l'indirizzo della memoria viene indicata mediante un *registro che contiene l'indirizzo da cui prendere il valore*. Ovvero si ha **LDR dest, [addr]**.

STORE. Analogamente al comando **LDR**, il comando **STR** ci permette di *salvare* un valore numerico di un registro in una locazione di memoria. Si ha quindi **STR src, [addr]**.

4. Chiamate di funzioni

LO STACK. Per effettuare una "*chiamata di funzione*" in Assembly, bisogna prima decidere le convenzioni per "*tradurre*" le seguenti:

- Come passare gli argomenti
- Come salvare lo stato prima della chiamata della funzione
- Come salvare lo stato della funzione
- Come ritornare il valore

Per far ciò usiamo una struttura particolare chiamata "*stack*". Questa struttura di indirizzi inizia con un indirizzo di memoria fissato, noto come "*top of the stack*"; dopodiché il registro *R13* indica la *prima posizione libera sullo stack*.

5. Esempio

ESERCIZIO. Effettuare una specie di "*conversione*" dal C in Assembly:

```
int x = 5;
int y = 10;
while (y>0)
{
    x = x + y + 3;
    y = y - 1;
}
```

SOLUZIONE.

```
MOV R0, #5
```

```
MOV R1, #10
```

loop:

```
CMP R1, 1
```

```
BEQ end
```

```
ADD R0, R0, R1
```

```
ADD R0, R0, #3
```

```
SUB R1, R1, #1
```

```
B loop
```

end:

7. Polling e Interrupt

Comunicazione Architettura e l'Esterno

Approcci PMIO, MMIO per la comunicazione tra architettura e strutture I/O. Metodi polling e interrupt per gestire la comunicazione tra CPU e I/O. Direct Memory Access per trasferimento di grandi quantità di dati.

1. Le questioni fondamentali

LE DOMANDE. Per poter aver un'idea di come poter far **comunicare** il nostro **calcolatore** con le strutture **I/O esterne**, dobbiamo porci le seguenti domande.

- Cosa succede se vogliamo comunicare con altri dispositivi? Come facciamo ad inviare e ricevere informazioni? Dove li salviamo? Come gestiamo la comunicazione asincrona?
- Come possiamo accedere ai dispositivi di memorizzazione di massa?

2. Collegamento delle informazioni

PMIO. Una prima idea per poter approcciarsi alla prima domanda è di usare l'approccio "**Port Mapped I/O**", dove semplicemente creiamo delle **istruzioni speciali** per comunicare

con dei dispositivi input-output. In particolare, sono delle istruzioni simili a dei *load* e *store*. Da notare che gli *indirizzi dei dispositivi I/O* sono diversi da quelli del *calcolatore*. Quindi, in questo caso *separiamo la memoria di lavoro* dai *dispositivi I/O*.

MMIO. Un'idea più aderente ad un approccio *RISC* è quello di usare *istruzioni già esistenti*: quindi dedichiamo alcuni *istruzioni già esistenti* della memoria per *operazioni di I/O*. In particolare servirà un *componente* che sarà in grado di *decodificare* se un *indirizzo* è dedicato al *calcolatore* o al *dispositivo I/O*.

3. Esecuzione delle Operazioni I/O

LA SECONDA DOMANDA. Ora, per risolvere la seconda domanda, ovvero *come gestire la comunicazione effettiva* tra il calcolatore e l'I/O, abbiamo due approcci per gestire casi in cui *i dispositivi* lavorano con *tempi diversi*, ovvero quando un dispositivo è più veloce dell'altro.

I due metodi principali sono il *polling* e l'*interrupt*.

POLLING. L'idea del *polling* è quella di controllare ad *intervalli regolari* se il dispositivo I/O sia pronto o meno. Il svantaggio di questo approccio è il *dispendio del tempo*, dedicato per fare il *polling*; oppure in certi casi, è stato effettuato in un tempo troppo tardivo.

INTERRUPT. Una seconda idea è quello di "avvisare" il *CPU* quando un dispositivo è pronto. Questo avviene tramite *supporto hardware*, che "*interrompe*" il processore forzandolo ad un pezzo di codice per gestire il dispositivo I/O.

4. Gestione di Grande Quantità di Dati

L'ULTIMA DOMANDA. Per gestire invece un *trasferimento di grandi quantità di dati*, gli approcci di *polling* e di *interrupt* sono inefficaci, dal momento che da un lato spostiamo sempre dati e dall'altro il processore viene continuamente interrotto. Allora l'idea è quello di *istituire un specifico* fatto apposta per questo caso, ovvero la *direct memory access* (DMA). Questo permette di copiare i dati dal dispositivo I/O alla memoria di lavoro, senza dover interrompere la *CPU*.

u1-s1-introduzione-so

Sistemi Operativi

Unità 1: Introduzione

Introduzione ai Sistemi Operativi

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Componenti di un sistema di elaborazione
 2. Architettura di un sistema di elaborazione
 3. Definizione di sistema operativo
 4. Componenti di un sistema operativo
 5. Definizioni relative ai sistemi operativi
 6. Tipologie di sistemi operativi
-

Le quattro componenti di un sistema di elaborazione

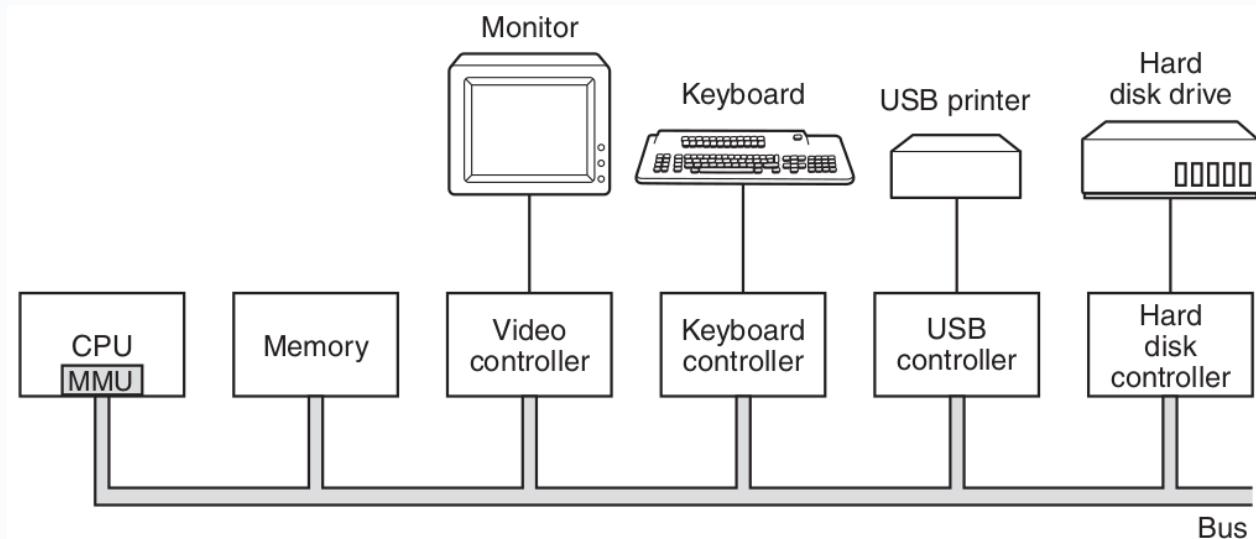
Un sistema di elaborazione si può suddividere in quattro componenti:

- *Hardware*
 - Fornisce le risorse fisiche: CPU, memoria, disco
- *Sistema Operativo*
 - Gestisce l'accesso all'hardware da parte dei programmi
- *Programmi Applicativi*
 - Eseguono i compiti desiderati dagli utenti o dal sistema
- *Utenti*
 - Lanciano e usano programmi

I *programmi applicativi* usano il *sistema operativo* per interagire con l'*hardware*.

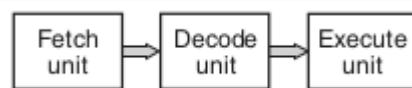
Architettura di un sistema di elaborazione (ripasso)

- Composto da diverse unità
- Il **bus** mette in comunicazione le componenti del sistema



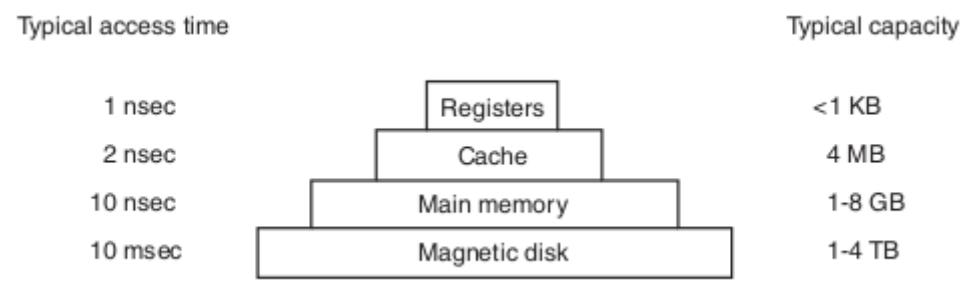
LA CPU.

- La **CPU** esegue le istruzioni che preleva dalla memoria.
- **Moduli fondamentali:**
 - Arithmetic Logic Unit
 - Control Unit
 - Registri
- Tre fasi per eseguire ogni istruzione: fetch, decode e execute. In particolare con i cicli **executre** e **fetch** si ha l'**interazione col mondo esterno**



LA GERARCHIA DELLE MEMORIA.

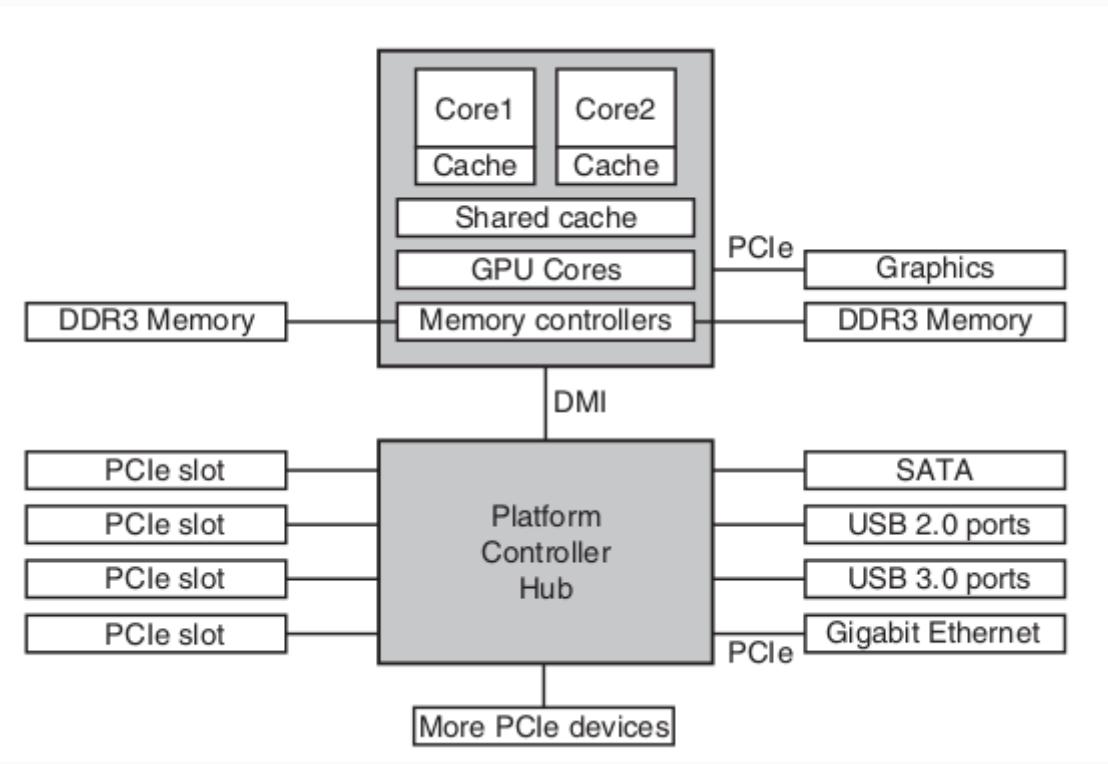
- La **Memoria** fornisce un modo per salvare i dati.
- Un sistema ha vari tipi di memoria, con caratteristiche diverse
 - Organizzazione in una gerarchia di memoria



Come esempio di "**magnetic disk**" si ha la **HDD** o la **SSD**, come esempio di "**main memory**" si ha la **RAM** e i "**registers**" si trovano nella **CPU**.

DISPOSITIVI I/O

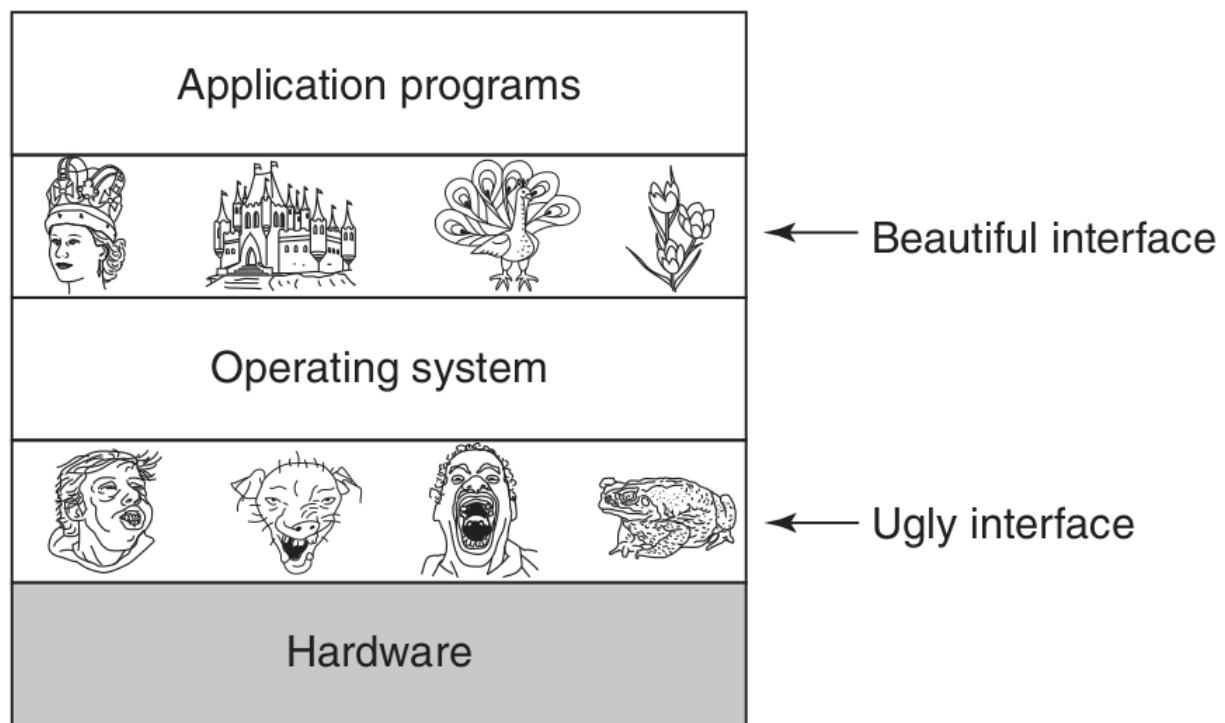
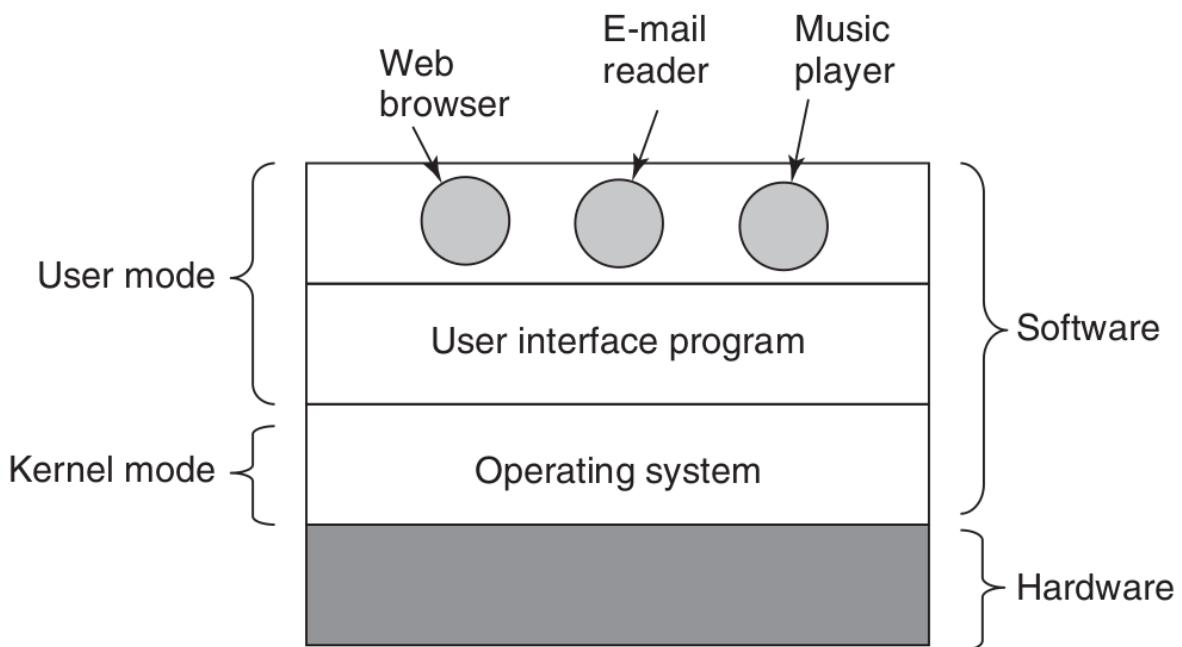
- Un sistema di elaborazione interagisce con l'esterno tramite dispositivi di **Input/Output**
 - Schermo
 - Tastiera
 - Mouse
 - Rete
 - **Sensori** (questi sono particolarmente importanti! sono presenti soprattutto nei smartphone)
- Comunicano con la CPU tramite il bus



Definizione di sistema operativo

IL COMPITO DEL SISTEMA OPERATIVO.

- Il compito dei sistemi operativi è:
 - Interagire con l'**hardware**
 - Fornire all'utente un **modello** di computer più semplice (ovvero con ciò che si chiama **UI**)
 - In un certo senso, i sistemi operativi rendono gradevole ciò che ha un'interfaccia sgradevole. Ovvero di **mediare** la comunicazione tra le **applicazioni** e **hardware**. Infatti, ogni **dispositivo I/O** ha una sua **logica**, quindi il compito di un **sistema operativo** è anche quello di **permettere la portabilità delle applicazioni**.



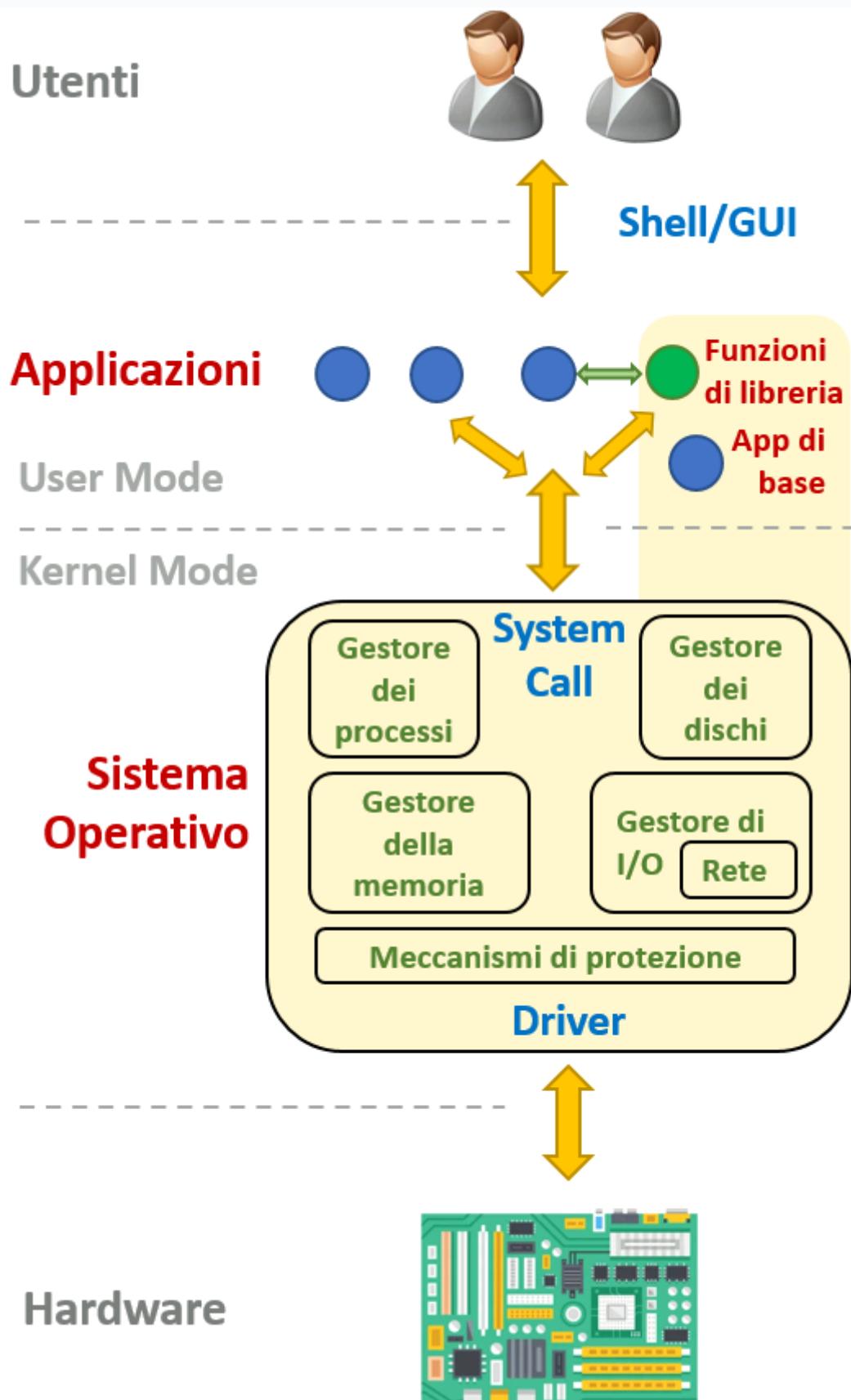
COME L'OS FA DA MEDIATORE?

- Il sistema operativo si interfaccia con i dispositivi hardware tramite dei *moduli software* chiamati *Driver*. Quindi si ha una situazione del tipo *OS ↔↔ Driver ↔↔ Hardware*.
- Offre servizi alle applicazioni tramite *API* chiamate *System Call*, che permettono alle applicazioni di *usare l'hardware facilmente*.
- Organizzato in moduli

Componenti di un sistema operativo

VISIONE GENERALE.

- Un sistema di elaborazione è composto di diversi moduli
 - Che offrono servizi a utente
 - Ma interagiscono anche tra loro



OSSERVAZIONI.

- I *system call* sono molto *fondamentali*, dal momento che per le *applicazioni* è *strettamente proibito* interagire direttamente con gli hardware. Infatti, si ha una situazione del tipo
 - *Software* \leftrightarrow *System call* \leftrightarrow *OS* \leftrightarrow *Driver* \leftrightarrow *Hardware*
- Qui il ruolo del *sistema operativo* diventa fondamentale

LE COMPONENTI SOFTWARE DEL SISTEMA OPERATIVO.

Principalmente un sistema operativo ha *tre gestori*, per gestire la mediazione *software-hardware*.

- **Gestore dei processi.** Crea e gestisce i *processi* (un processo è un programma in esecuzione) e *trova le risorse* di CPU e memoria *necessarie* all'esecuzione
- **Gestore della memoria.** La memoria di un elaboratore è un unico vettore (dato che stiamo implicitamente usando le architetture *Von-Neumann*) e il SO gestisce l'*allocazione di memoria ai processi* e la *condivisione della memoria* tra programmi.
- **Gestore dei dischi.** La memorizzazione su disco (come un HDD) è persistente e il SO organizza la memoria su disco in una struttura ad albero di directory e file
- **Gestore dei dispositivi di I/O (e di rete).** I dispositivi di I/O *non* sono gestiti *direttamente* dalle applicazioni (sarebbe infatti complicato e creerebbe problemi di funzionamento); i SO utilizza invece i *driver* per pilotare i dispositivi. La connessione di rete rappresenta un caso particolare di I/O: ha un trattamento speciale nei moderni SO.
- **Gestore dei meccanismi di protezione.** Nei SO ci sono tanti *utenti* con *diversi permessi di accedere alle risorse* (per accedere a file, dispositivi, configurazione, ecc...). I SO implementa le policy di accesso; vedremo in seguito il caso particolare di *Linux*.

Dopodiché ci sono altre componenti che agevola ai processi di fare ulteriori cose.

- **Sincronizzazione.** Permette ai processi di *comunicare* e di *sincronizzarsi* tra loro, dal momento che *ogni processo* va per *conto suo*, ma vanno comunque *coordinati*.
- **Virtualizzazione.** Per supportare la creazione di macchine virtuali o simili (e.g., container). Se il SO offre funzionalità per la virtualizzazione, *le VM saranno molto più veloci* perché possono avere *accesso diretto* ad (alcune) risorse. Ultimamente la *virtualizzazione* divenne un tema fondamentale per i *sistemi operativi*.

Definizioni relative ai sistemi operativi

Adesso diamo delle *definizioni* relativi ai *sistemi operativi*.

1. Kernel

1. Kernel

- Si dice il "*cuore del SO*"
- Include *tutti i moduli* visti in precedenza
 - Gestisce le operazioni fondamentali e a più basso livello
- Modulo software *sempre in esecuzione*
 - Con *privilegi speciali*, ovvero *massimi*
 - Tutte le altre applicazioni (di utente o di sistema) hanno *privilegi minori* (ad esempio a loro è *proibito* interagire con i dispositivi I/O)
 - Si *appoggiano al kernel*
- Diverse tipologie di kernel
 - *Monolitici*: il kernel è un unico programma che esegue tutto il codice necessario.
 - *Micro-Kernel*: cercano di delegare alle applicazioni più funzionalità possibile (per rendere più modulare un sistema operativo).
 - *A livelli stratificati*: il kernel (e i programmi) sono organizzati in una gerarchia di processi a privilegi crescenti (questo per creare meno distinzioni tra app e kernel).

2. Processo

2. **Processo**

- Programma in esecuzione
 - Con certi *privilegi e risorse* della CPU o RAM
- Accede alla CPU *a turno* con gli altri programmi
 - Il SO deve permettere che il suo stato sia conservato quando viene messo in pausa
 - Richiede salvataggio di registri di CPU
 - Ogni processo identificato da un numero detto *PID* (*Process Identifier*)
- Un processo può creare altri processi detti *figli*
 - Si dice che i processi sono organizzati in un *Albero dei Processi*

3. Thread

3. **Thread** (da inglese *"filo")

- Un processo *raggruppa le risorse* di un programma in esecuzione
- Un *thread individual* raggruppa a sua volta *un flusso di esecuzione*
- Un processo può avere invece al suo interno *uno o più flussi* in esecuzione (come ad esempio *Chrome* con le sue tab)
- Identificato da un identificatore detto *TID*

4. System call

4. **System call**

- Sono delle *funzioni* messe a disposizione dal SO alle applicazioni
- Offrono i servizi del SO per *creare processi, accedere ai dischi*, ecc...

- Il kernel **"serve"** (nel senso di compiere) le richieste di System Call dei programmi
- **ATTENZIONE!** Da non confondere con le **Funzioni di Libreria!!!**
 - Che sono **moduli software** (**pezzi di codice**) che svolgono compiti comuni a più applicazioni
 - Sono software comuni con stessi **privilegi di applicazione** (ovvero a "**user-level**")
 - Possono (ma non sempre) chiamare delle System Call
 - Non possono **interagire direttamente** con l'**hardware**
- Sintassi delle system call.:**
- Le system call sono funzioni C. Hanno aspetti diversi

POSIX su Linux

```
C
int read (int fd, void *buffer, size_t nbytes);
```

Win32/Win64 API su Windows

```
C
BOOL ReadFile (
    HANDLE fileHandle,
    LPVOID dataBuffer,
    DWORD numberOfBytesToRead,
    LPDWORD numberOfBytesRead,
    LPOVERLAPPED overlappedDataStructure
);
```

Esempi di System Call comuni.

- Gestione dei **processi**: **fork exec wait kill**
- Gestione dei **file su disco**: **open close read write**

5. I sistemi di libreria

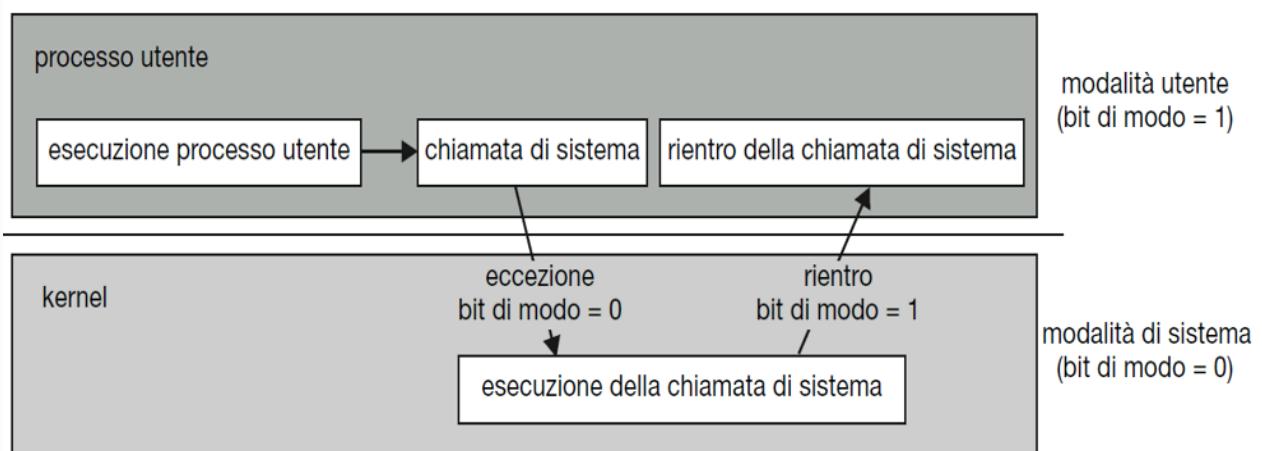
I SISTEMI DI LIBRERIA.

- **Nota:** i SO operativi offrono anche molte **funzioni di libreria**
 - Alcune funzioni di libreria non necessitano di System Call
 - Calcolare la radice quadrata: **double sqrt(double arg)**. Non necessita di una System Call
 - Scrivere su schermo: **int printf(char *format, arg list ...)**. Formatta la stringa da scrivere e poi chiama la System Call **write**
- **ATTENZIONE!** Quindi da questo discende che la funzione **printf(...);** **NON** può essere una **system call!** Questa **invoca** una system call, ma non lo è!

6. Kernel/User Mode

5. **Kernel/User mode** nella CPU: Ci sono principalmente *due livelli di privilegio*.

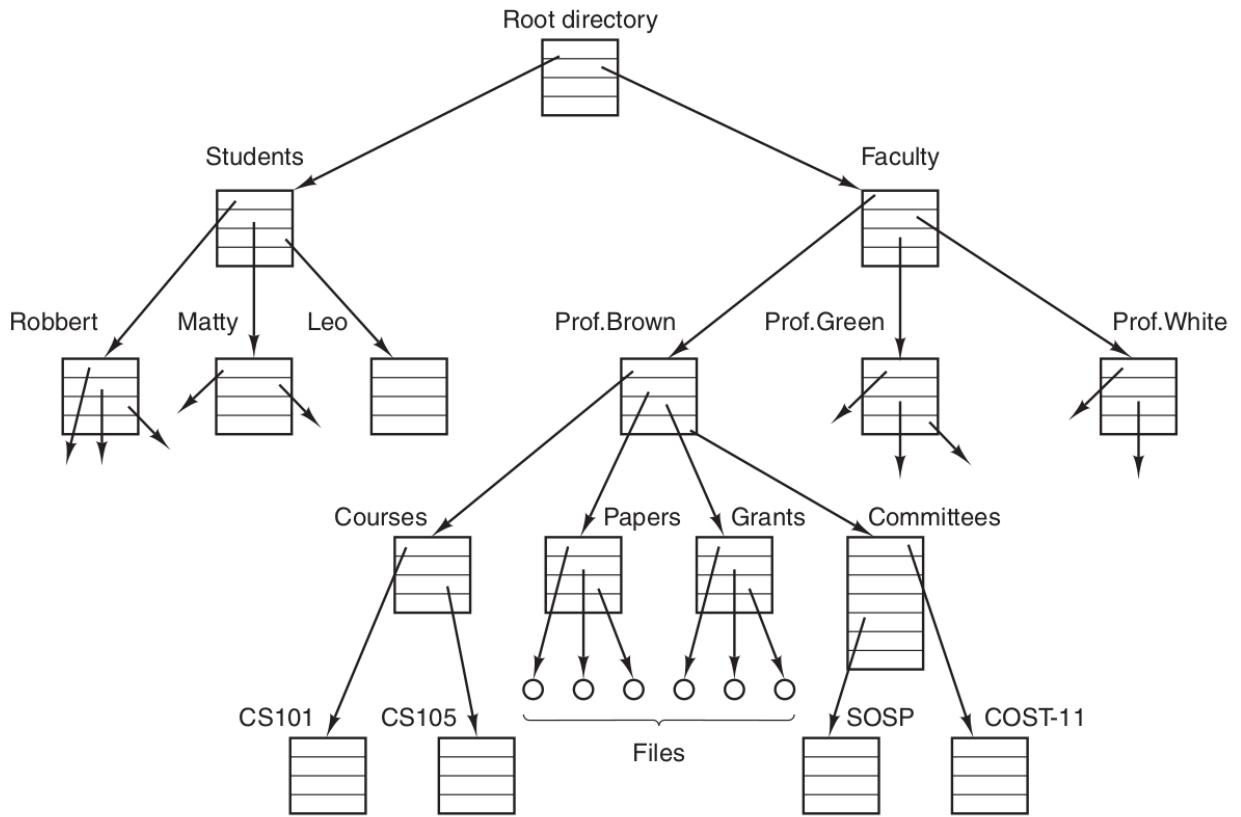
- **Kernel Mode (alto livello)**: Il kernel ha accesso *completo* all'hardware
 - Può accedere a ogni locazione di memoria e registro di dispositivo di I/O; può accedere a **TUTTO** per quanto concerne i dispositivi I/O
 - **Solo** codice di ottima qualità. Gli errori sono potenzialmente *distruttivi!!!*
- **User Mode**: le applicazioni hanno possibilità limitate
 - Accedono a uno spazio di indirizzamento limitato (*memoria virtuale*); quindi **non** possono accedere a "*memorie virtuali di altro processi*".
 - Non possono accedere direttamente ai dispositivi di I/O; quindi **niente Assembly**!
 - Necessaria cooperazione della CPU che implementa specifiche funzionalità
 - Le *funzioni di libreria* si eseguono in *User Mode*



7. File system

7. **File System**: ci sono due *significati* di *file system*

- Struttura che include un *insieme di file e cartelle*
- Organizzato secondo un *albero* (ovvero un *file system* è l'*albero*)



8. I path

8. Organizzazione dei path.

- **Root Directory:** la *radice di tutti le cartelle*. Si *identifica* con **/** in Linux
- **Working Directory:** directory dove un processo viene lanciato
Ci sono *due modi* per *esprimere un path*.
- **Path Assoluto:** *inizia* con **/** e identifica un percorso a partire dalla *Root Directory*, dalla *radice*
- **Path Relativo:** *non inizia* con **/** ma con un nome. Identifica un path relativo alla *Working Directory* del processo

9. Bootloader, login e shell

9. Bootloader:

- Il codice che carica in memoria il kernel al momento dell'accensione del sistema
- Contenuto in *ROM/EEPROM immunificabile*
- Cosa succede quando *accendiamo un computer*? La *CPU* accede un *indirizzo noto della memoria*, che a sua volta contiene una *ROM* con il *bootloader*.

9. Login:

- *Autenticazione* di un utente nel sistema, solitamente tramite *username e password*

9. Shell:

- *Programma* che legge *comandi* da tastiera, li *esegue* e ne stampa l'*output*
 - **NOTA.** Come input viene *accettato solo la tastiera*

- Metodo di accesso tradizionale
 - *Non fa parte del kernel*
 - Quando un sistema viene avviato, il kernel avvia sempre una shell o l'interfaccia grafica
 - Ne faremo un pesante utilizzo nel corso
-

Domande

Cosa é un processo?

- a) • Un programma
- b) • Un algoritmo
- c) • Un programma in esecuzione

Le System Call sono usate da:

- a) • SO per interagire con l'hardWare
- b) • Dai processi per interagire col SO

Le Funzioni di Libreria vengono eseguite:

- a) • In User Mode
- b) • In Kernel Mode

Il File System é organizzato come:

- a) • Un Grafo contentente cicli
- b) • Un Grafo NON contentente cicli

Risposte

C, B, A, B

u1-s2-tipologie-storia

Sistemi Operativi

Unità 1: Introduzione

Storia e Tipologie di Sistemi Operativi

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Storia dei sistemi operativi
 2. Tipologie di sistemi operativi
 3. Linux
-

Storia dei sistemi operativi e degli elaboratori

I Primi passi ai Sistemi Operativi

Condizione necessaria per un SO: Avere un sistema di elaborazione

LA PRIMA IDEA: Primo elaboratore *progettato* da Charles Babbage nella *prima metà dell'800*

- Puramente meccanico.
- Non fu mai costruito

I PRIMI ELABORATORI: I primi elaboratori vennero *costruiti* negli anni '40 del '900

- Basati su *valvole*, oggi si usano i *transistori* che sono più *veloci ed efficaci* (e meno grandi)
- Programmati direttamente in linguaggio macchina
- Nessun sistema operativo. L'elaboratore eseguiva un programma per volta

FIGURA: Elaboratore a valvole

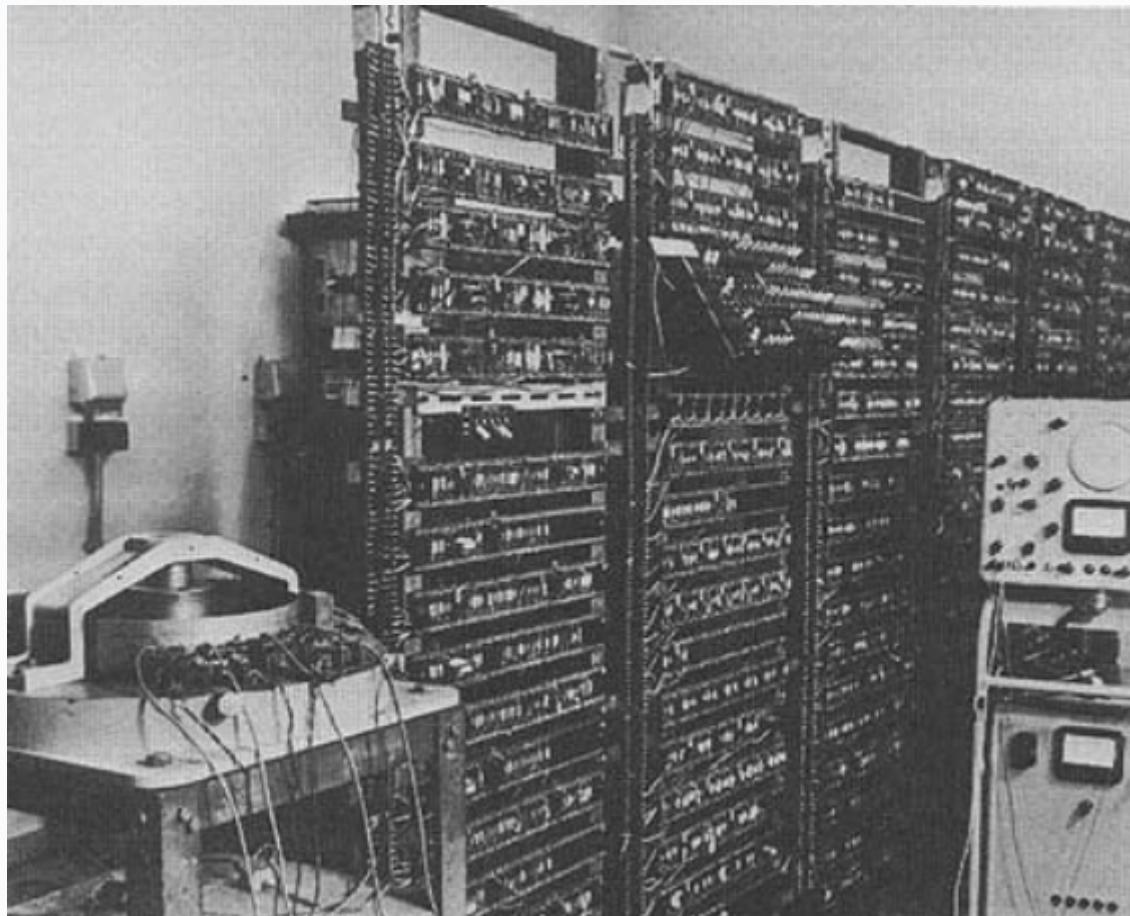
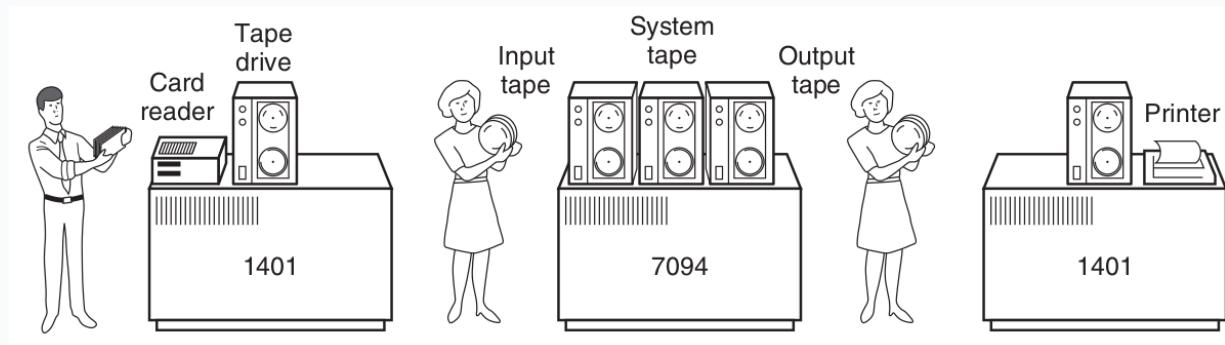


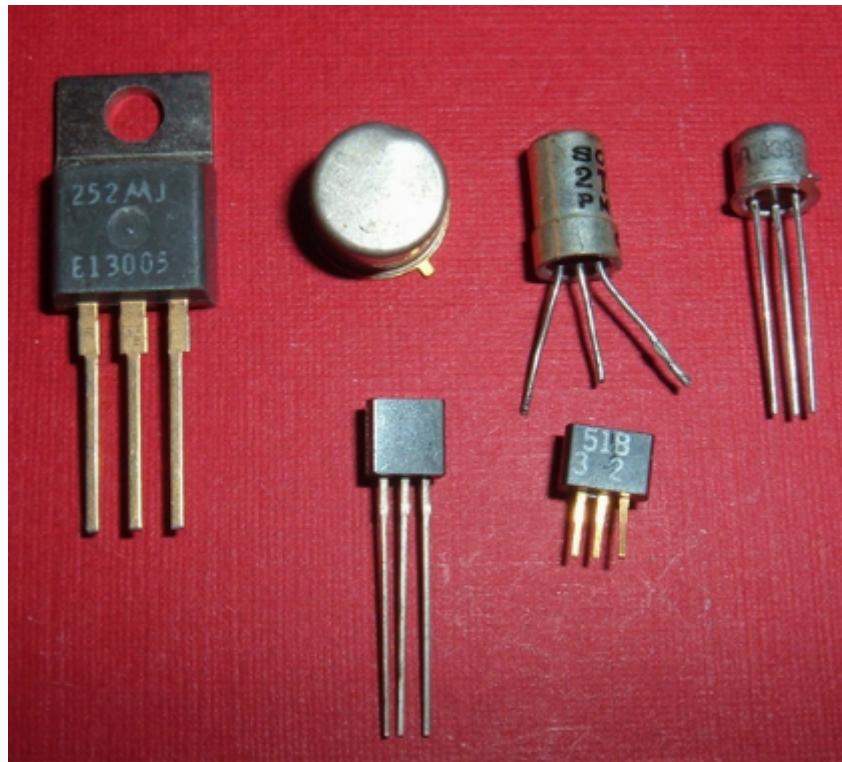
Il passaggio ai transistor

Elaboratori a transistor (1955-1965) (nel *dopoguerra*)

- Mainframe custoditi in locali e da tecnici specializzati
- Programmi scritti su schede perforate o nastri magnetici
- Primi linguaggi di programmazione (e.g., *FORTRAN*)
- I programmi venivano *eseguiti in sequenza*.
- Il *sistema operativo* aveva il *solo* compito di *eseguire programmi* in sequenza: il codice veniva rappresentato dalle *schede perforate*.

FIGURA: MACCHINA A TRANSISTOR





I circuiti integrati e i Primi Sistemi Operativi

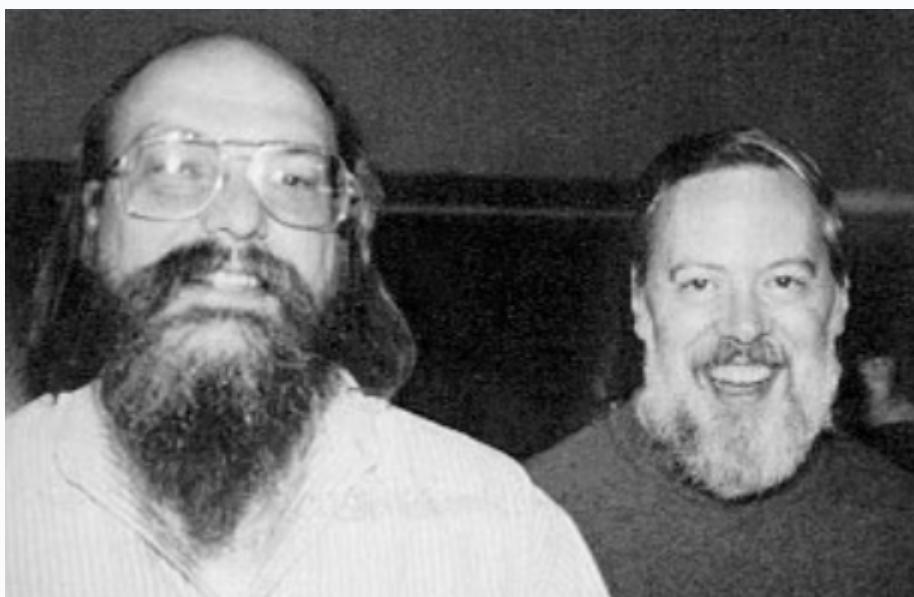
Circuiti Integrati (1965-1980)

- Prestazioni maggiori. Prezzi e dimensioni minori: infatti i *circuiti integrati* sono delle *singole schede* con molti *transistor*
- IBM crea la linea di computer IBM 360, col suo SO detto **OS/360**
 - Introduce multiprogrammazione: più processi in esecuzione contemporaneamente
 - Usato ancora oggi in alcuni campi
- Il MIT assieme a partner industriali (3) sviluppa **MULTICS**
 - Progettato per *main frame molto potenti*
 - Permette l'utilizzo a *centinaia di utenti*
 - Poco successo: *MULTICS* rimane comunque importante, dato che ha posto le *fondamenta* per la costruzione di altri *sistemi operativi*. Infatti, da MULTICS nasce *Unix* nei Bell Labs (New Jersey), come progetto personale di Ken Thompson, Dennis Ritchie e altri: poi da UNIX nascerà la famiglia Linux

FIGURA: Elaboratore a circuito integrato



FIGURA: Ken Thompson e Dennis Ritchie



Apple e Microsoft

I personal Computer (1980-oggi)

- Grazie a sviluppi nei Circuiti Integrati fu possibile produrre computer a prezzi bassi.
- Nei primi anni '80 nasce **Microsoft/DOS**, inizialmente pensato per computer IBM con CPU Intel (famiglia 8086)
- La **Apple** inventa un sistema operativo con **GUI** che ottiene molto successo, da cui **Windows** si ispirerà

FIGURA: Bill Gates e Steve Jobs



Microsoft: Fondata nel 1975 da Bill Gates e Paul Allen

- Nel 1981 commercializza **MS-DOS**
- Nel 1985 commercializza **Windows**
 - Sistema operativo con *interfaccia grafica a finestre* (da cui il nome)
 - Orientato a *processori Intel*
- Nel tempo ha commercializzato versioni a
 - 16 bit (Windows 1.0, 1985 – Windows 3.1, 1992)
 - 16/32 bit (Windows 9x, 1993-2000)
 - 32/64 bit (da Windows NT in poi)
 - Oggi abbiamo Windows 11

Apple: fondata nel 1976 da Steve Jobs, Steve Wozniak e Ronald Wayne

- Dal 1984 al 2001 commercializza Mac OS
 - SO completamente grafico
 - Raggiunge *limiti strutturali* di sviluppo alla fine del '90, non permettendo
 - *Multitasking preemptivo*: questo si rivelò come la "*falla fondamentale*" del sistema operativo **MAC OS**, in quanto con questo sistema *era il singolo processo a dire quando interrompere la CPU*. Se si ha un "*processo buggato*", allora l'intero processore sarebbe bloccato!
 - Memoria protetta
- Ricostruendo tutto *da capo*, nel 2001 commercializza Mac OS X
 - Nato per computer Macintosh
 - Inizialmente retro-compatibile con Mac OS
 - Basato su architettura UNIX

I SISTEMI OPERATIVI DI OGGI.

- Windows e Apple Mac OS continuano lo sviluppo fino ad ora.
 - Windows 11 e Mac OS 13 Ventura
- Dagli anno '90 in boom dei **telefoni cellulari**, porta alla nascita di sistemi operativi dedicati. Nascono:
 - (Symbian: morto nel 2011)
 - Android
 - Mac OS (*iOS*)

Tipologie di sistemi operativi

Diverse varietà di SO. Alcune ancora vive, altre morte e sepolte.

1. SO per mainframe

- Per elaboratori *enormi in grandi compagnie*
- Supportano tanti utenti e risorse
- In declino in favore di SO general purpose (Linux)
- Esiste ancora OS/390, discendente di OS/360 di IBM. Ad esempio ancora oggi i *server delle banche* usano questi OS.

2. SO per PC (ad uso personale)

- Sono i più diffusi.
- *Basati su interfaccia grafica*
- Pensati per un solo utente, *non esperto*
- Esempi: Windows, MacOS, Ubuntu (?), ...

3. SO per server

- Per *professionisti*
- Spesso dotati di *sola shell*
- Sono varianti di quelli per PC
- Esempi: Linux, Windows Server

4. SO per Smartphone o tablet

- Basati su GUI e input touch
- Esempi: Android, MacOS

5. SO integrati

- Per router, elettrodomestici, veicoli
- Non accettano programmi esterni

6. SO per sensori

- Su dispositivi con risorse *molto limitate* (come ad esempio calcolatrici)

- Molto leggeri e semplici

Usi comuni per le tipologie 5,6: Frighi, elettrodomestici, ...

7. SO real time

- Per applicazioni particolari dove il **tempo è fondamentale**
 - Processi industriali, *aerei*, autoveicoli
- Alcuni compiti devono essere svolti **tassativamente** entro *una deadline*: il più *velocemente possibile*
 - Design del sistema notevolmente più complicato

SO per smartcard

- Le *smartcard* (e.g., Bancomat) hanno un sistema di elaborazione e un SO (*spesso*)
 - Requisiti di **basso consumo** e **sicurezza**
-

Storia del Linux

UNIX

Abbiamo detto che **Unix** nasce negli anni '70 da MULTICS: da sottolineare che è stata nata negli anni '70 (o fine anni '60)! ancora oggi ci stiamo portando la *filosofia* dietro Unix, dal momento che è stato *progettato così bene*.

Nascono *numerose varianti* negli anni '80 che vengono *standardizzate* (in particolare sono standarizzati i *system calls*)

- Standard **ISO C** - 1972
- Standard **Posix** - 1988

Tutte le versioni erano a pagamento, in capo ad *AT&T*

- Il codice era *closed-source* (a *pagamento*), molto lungo e complesso

MINIX

Creato da Andrew *Stuart Tanenbaum*

- Uno degli autori di uno dei libri consigliati in questo corso

E' un *clone di UNIX*:

- *Open-Source*
- A *micro-kernel*
- Pensato per la *didattica*, in particolare per un corso universitario sui *sistemi operativi*
- Non adatto a essere un vero SO

FIGURA: Stuart Tanenbaum



LINUX

Nel 14.03.1991 il giovane finlandese universitario Linus Torvalds crea il kernel **Linux**:

- Sviluppato *a partire da Minix*
- Per esser un *vero SO* (non solo per scopi didattici): quindi anche un uso *professionale*
- Tante distribuzioni: *Ubuntu, Debian, Fedora* e infinite...
- Ormai diffuso globalmente

FIGURA: Linus Torvalds che se la ride



FIGURA: Linus Torvalds che alza un dito medio nel mezzo di una conferenza, gesto per insultare la compagnia NVIDIA



Definizione di Unix, Linux, GNU

Definizioni Generali

- **Unix**: sistema operativo sviluppato negli anni '80 in AT&T (*closed source!*)
- **Linux**: è un kernel Unix-like sviluppato da Linus Torvald dal 1991 (*open source!*)
- **GNU**: sistema operativo open source (*kernel escluso!*) Unix-Like. Può funzionare con diversi kernel; la maggior parte funziona col *linux*. GNU è un acronimo ricorsivo, che sta per *GNU's Not Unix!*⁽¹⁾. La GNU è quella parte che fornisce le *applicazioni di default* e roba del genere.

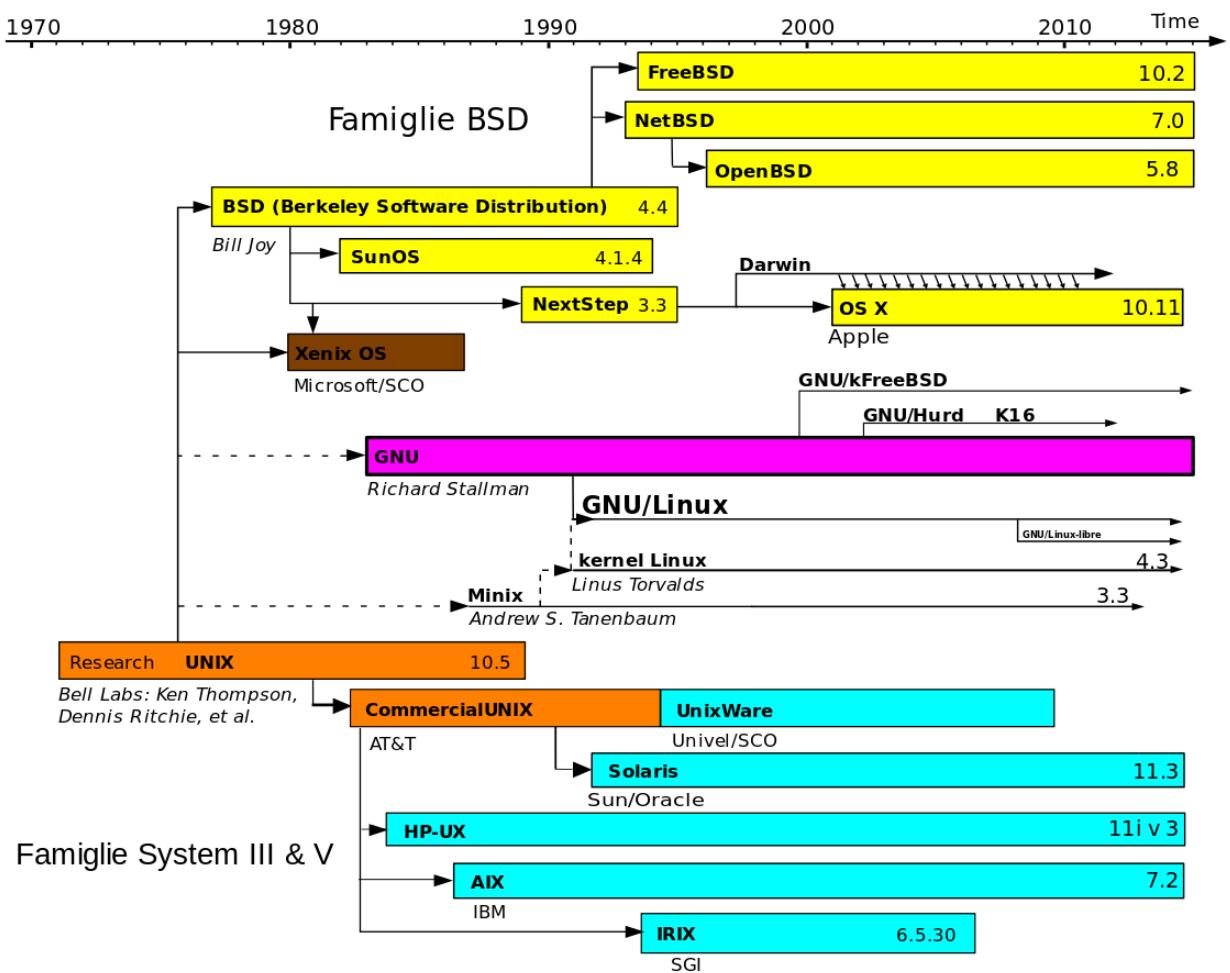
Ora lo standard è **GNU/Linux**: sistema operativo GNU con kernel Linux

Nota: L'evoluzione diretta di Unix é il SO *Berkeley Software Distribution* (BSD), da cui derivano *FreeBSD* e *Mac OS X*

⁽¹⁾. *GNU is a recursive acronym for "GNU's Not Unix!", chosen because GNU's design is Unix-like, but differs from Unix by being free software and containing no Unix code. Stallman chose the name by using various plays on words, including the song The Gnu.*

Diramazioni da UNIX (riassunto generale)

FIGURA: Linea temporale di diramazioni dall'UNIX



NOTA. Quindi da questo schema si sa che l'**OS X** (ovvero il sistema operativo per i dispositivi Apple) **deriva** dall'**UNIX**, che ha la stessa convenzione di **Linux**; è comunque sbagliato dire che **OS X** è **Linux**! Sono due cose comunque diversissime

Linux Oggi

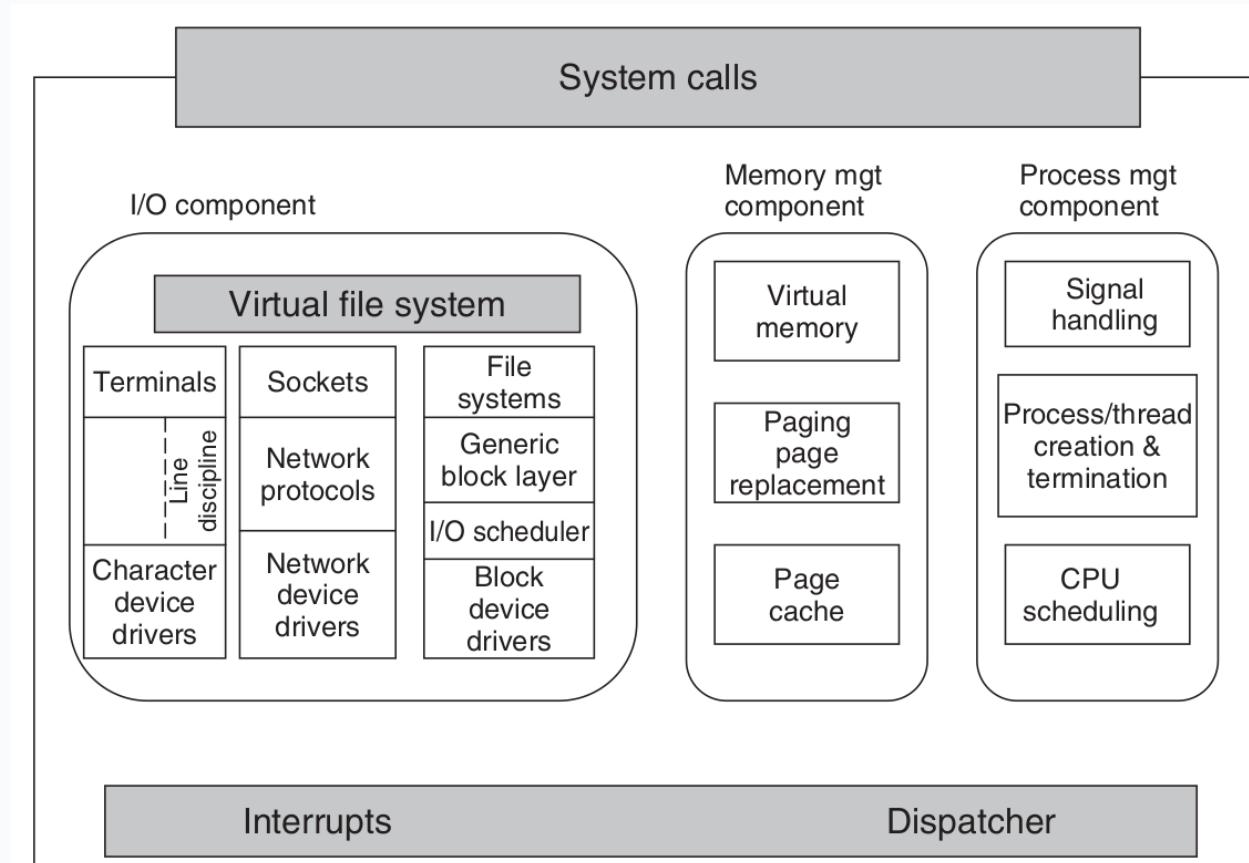
- Stabile, maturo e free (*a livello professionale*)
- Più complesso da usare di Windows
- Alla base di quasi tutte le tecnologie per:
 - Servizi web: **hosting** (e.g. di siti, ...)
 - Archiviazione dei dati: **database**, data warehouse
 - **Sistemi embedded**
 - Piattaforme per **Intelligenza Artificiale**; algoritmi addestrati su server con OS basati sul kernel Linux

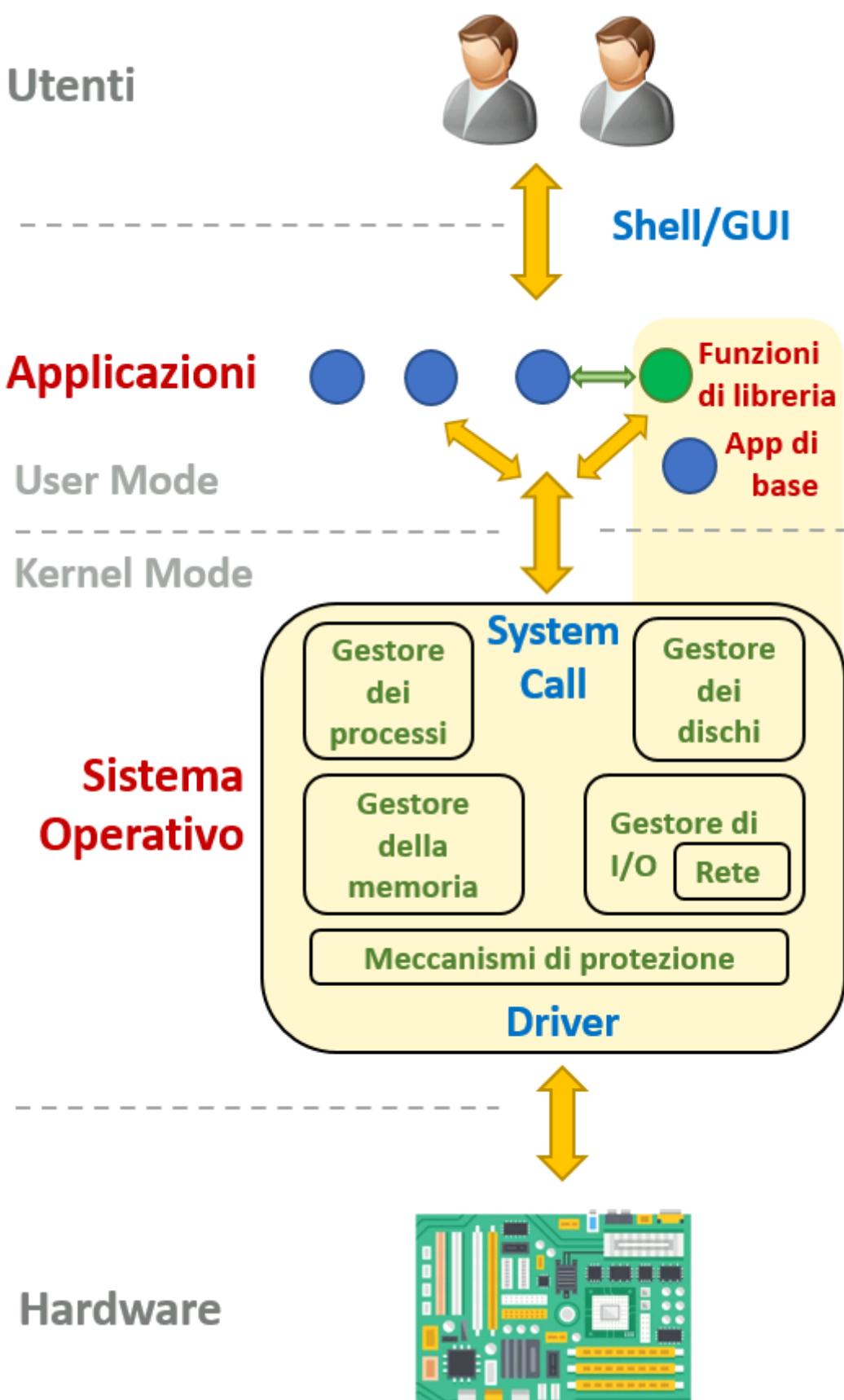
Questo corso si concentrerà sull'uso di Linux, dato che per noi diventerà importantissimo!!!

Kernel Linux

Simile rispetto alla nostra *definizione generica di SO*

FIGURA: Kernel Linux





GNU/Linux

La **GNU/Linux** include *librerie e utility* di default, come i seguenti.

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
ps	List running processes
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

Domande

POSIX é:

- Uno standard
- Un SO
- Una famiglia di SO

Risposta: *Uno standard*

UNIX é:

- Closed-Source
- Open-Source

Risposta: *Closed-Source*

Linux é:

- Closed-Source
- Open-Source

Risposta: *Open-Source*

tmp

u2-s1-linux

Sistemi Operativi

Unità 2: Utilizzo di Linux

Ambienti Linux

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Distribuzioni di Linux
 2. Alternative per usare Linux
-

Distribuzioni di Linux

Distinzione tra Linux e GNU/Linux

LINUX.

Con **Linux** si intende *il Kernel UNIX-like creato da Linus Torvald*.

GNU/LINUX.

Con **GNU/Linux** si intende *una famiglia di sistemi operativi basati su Kernel Linux*

Ci sono **più di 100 SO** che sono della famiglia **GNU/Linux**. Condividono:

- Kernel Linux
 - Programmi e utility di base di GNU per gestione di file, processi, rete
-

Distribuzioni di Linux: Ubuntu

Le famiglie principali OS Linux sono:

Ubuntu: attualmente il *più diffuso*.

- Basato su un'altra distribuzione chiamata **Debian**
 - Debian è il *punto di partenza* per tanti altri OS Linux (o *distribuzioni Linux*)
 - Debian contiene *solo software libero*, Ubuntu no; ad esempio in Debian *non c'è la decodificazione di alcuni formati audio codecs*.
 - Ha lo scopo di offrire un SO completo e facile da usare per PC (e per server)
 - Ne derivano *altri distribuzioni* che di differenziano per il software che gestisce l'ambiente grafico (desktop, finestre); ad esempio abbiamo *XUbuntu*, *Kali* (storicamente) eccetera...
-

Distribuzioni di Linux: Red Hat

Red Hat Enterprise Linux e CentOS: *versioni professionali* di Linux, per il mercato aziendale

- *Red Hat* è la *ditta che crea queste distribuzioni*
 - Particolare attenzione a *stabilità e sicurezza*
 - Mantenute dall'azienda Red Hat, che offre *supporto (commerciale) a pagamento*
 - *RHEL* è la versione con supporto commerciale. CentOS è la versione *consumer*
 - L'OS **Fedora** è della stessa famiglia, è adotta funzionalità più innovative, sebbene meno stabili (*versione più aggiornata*)
-

Distribuzioni di Linux: Arch, openSUSE e Mint

Arch Linux: distribuzione leggera, adatta a sistemi minimali e con poche risorse

- Non prevede un ambiente Desktop di default
- Utilizza la filosofia KISS (Keep It Simple, Stupid)

openSUSE Linux: sviluppata da volontari. Nei primi anni 2000 era molto diffusa

Linux Mint: basata su Ubuntu. Ha avuto momenti di celebrità nei primi anni 2010

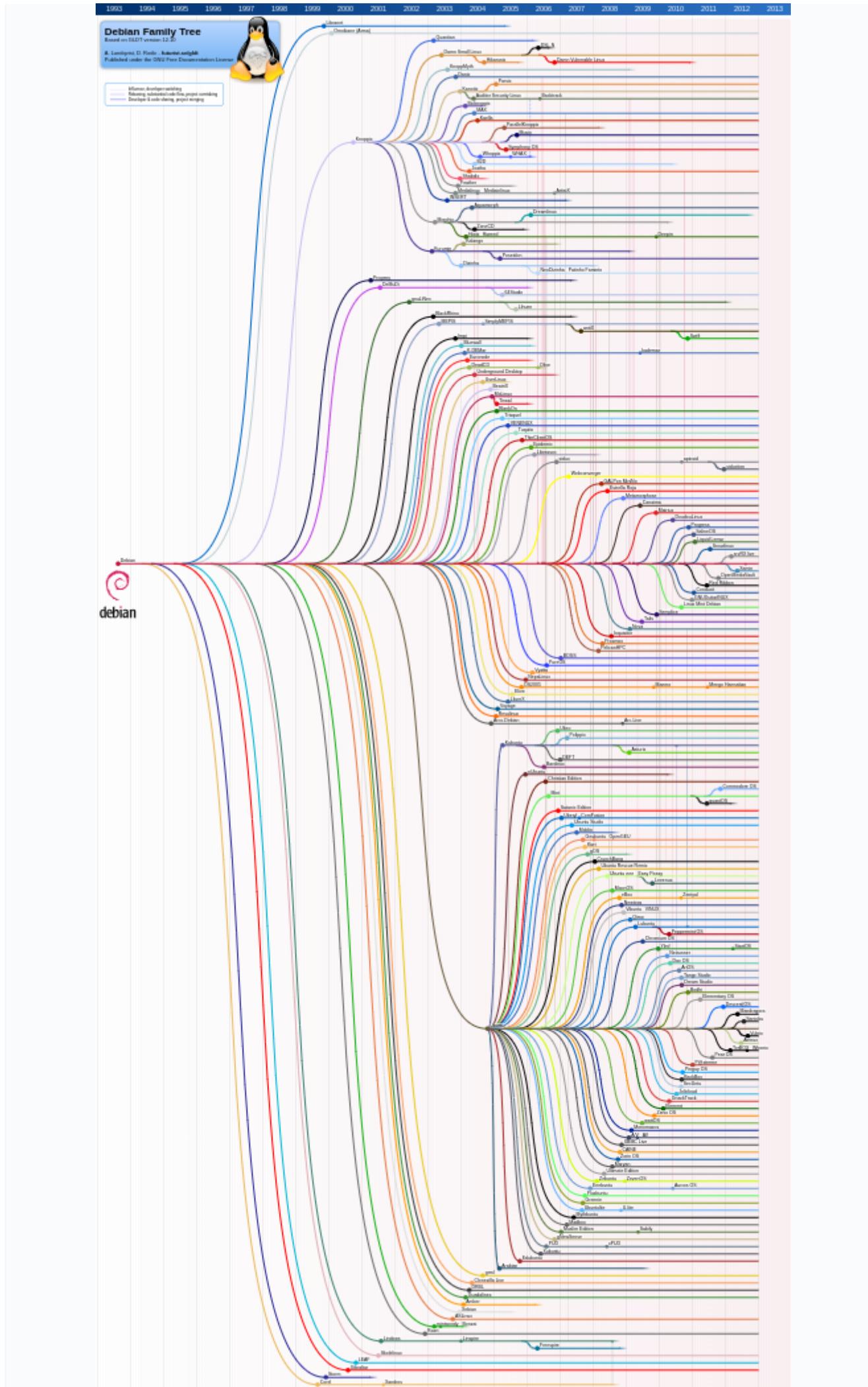
L'infinità delle distribuzioni di Linux

Impossibile enumerare tutte le distribuzioni (ce ne sono quasi *infinite!*).

Molte nascono e muoiono nel giro di pochi anni (si dicono "*abandonware*")

In questo corso utilizzeremo *Ubuntu*

- Diffuso
- Semplice
- Generico
 - Ha versione per PC e per server



Alternative per usare Linux

Alternative per usare Linux

Per utilizzare un sistema Linux, ci sono varie alternative a seconda che:

- Si abbia in PC o un MAC
- Si abbia tanto o poco spazio su disco
- Si sia più o meno esperti nell'utilizzo del computer

Alternative per usare Linux

Installazione Nativa: si installa un SO Linux su un PC.

- Necessario scaricare l'immagine dal sito di un SO Linux (e.g., Ubuntu)
- Il PC viene formattato e il SO è installato nativamente
- Si può mantenere Windows (o Mac OS) usando il *Dual Boot*
 - L'hard disk è partizionato in due drive logici, uno con Linux, uno con Windows
- Operazione non facilissima, e potenzialmente distruttiva

Alternative per usare Linux

Linux da USB "Live":

Ogni distribuzione di Linux può essere usata Live:

- Si crea una chiavetta USB *bootable*
- Si inserisce nel PC e lo si forza a fare *boot* da chiavetta
- Linux gira nativamente come se fosse installato
- Su Windows si può usare il software Rufus <https://rufus.ie/en/>
 - **Nota:** a meno che non lo si configuri esplicitamente, la chiavetta non è **persistente**. A ogni riavvio si perdono tutti i file modificati

Alternative per usare Linux

Macchina Virtuale: utilizzando un software chiamato **virtualizzatore** è possibile creare un PC virtuale.

- E' a tutti gli effetti un PC completo di tutte le funzionalità
 - Ha una CPU, memoria e disco virtuali
 - Che gira all'interno di un'applicazione
 - Non danneggia né impatta il SO nativo del PC
 - Tanti software per virtualizzazione
 - **VirtualBox** (consigliato)
 - **VMWare**
 - **QEMU**
-

Alternative per usare Linux

- Passi necessari:
 - Installare il virtualizzatore
 - Creare una nuova macchina virtuale
 - Specificare la quantità di risorse (CPU, memoria, disco) da allocare alla macchina virtuale
 - Installare il SO Linux preferito
 - Configurarla con i software desiderati, se necessario
 - Questa è l'**opzione consigliata**:
 - Facile, stesse potenzialità di avere Linux installato nativamente
 - Il PC deve essere abbastanza potente:
 - Almeno 8 core, 8GB di RAM e 20GB (di spazio libero) su Hard Disk
-

Alternative per usare Linux

Cygwin: è un software da installare su Windows

E' un layer di compatibilità POSIX che permette di usare programmi POSIX su sistemi Windows

- Mappa le system call POSIX su quelle di Windows.
- Include i tool GNU base per gestione di file, compilazione
- Necessario compilare i programmi usando Cygwin

Facile da installare:

- Si installa come un normale programma

- Sito Web: www.cygwin.com
-

Alternative per usare Linux

Windows Subsystem for Linux (WSL): è anche esso un layer di compatibilità per programmi Linux su Windows.

- Sviluppato direttamente da Microsoft
 - A partire da Windows 10
 - Permette di eseguire eseguibili Linux senza ricompilare
 - Si può installare tramite command line di Windows
 - Dopodichè è possibile installare pacchetti di software Linux
 - Ad esempio si può installare l'applicazione "Ubuntu" tramite software center.
 - Nota: l'applicazione "Ubuntu" non è un vero SO. E' solo un pacchetto che contiene i software di base di Ubuntu e una shell
-

Alternative per usare Linux

Terminale via browser:

- Tante opzioni online
 - Cercare su Google: *linux box online*
 - Una è: <https://linuxcontainers.org/incus/try-it/>
 - Non persistente
 - Limitata a 30 minuti
 - Va bene per provare i comandi Linux *al volo*
-

Domande

Ubuntu è un SO che utilizza il Kernel:

- **Linux**
- **UNIX**
- **POSIX**

Red Hat è un:

- **Kernel**
- **SO**
- **Uno standard**

u2-s2-concetti-linux

Comandi di Linux

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Sessione Shell
 2. Comandi di base
 3. File System
 4. Utenti e permessi
 5. Processi e programmi
 6. Altri comandi
 7. Esercizi
-

Sessione Shell

Login

Per fare login, è necessario inserire le proprie *credenziali sul terminale* (in realtà di solito viene fatta *automaticamente*):

```
login as: <username>
password: <password>
```

Per fare logout: **CTRL+D**, **exit** o **shutdown** (solo *superuser*, detto *root*)

Terminali Remoti attraverso SSH

E' possibile usare un terminale *remoto* utilizzando SSH (*Secure Shell*).

Su un'altra macchina collegata *in rete*, digitare su terminale:

SHELL

```
ssh <username>@<indirizzo IP della macchina>
```

Si utilizza il protocollo *Secure Shell*, che *trasmette* in maniera cifrata i comandi e il loro output tramite la rete.

Comandi di base

Argomenti e Opzioni dei Comandi

Vengono digitati sul terminale. Avviano il corrispondente eseguibile

Alcuni ammettono *argomenti*, ovvero gli *oggetti* su cui il comando deve agire.

- Specificati dopo il nome del comando

Alcuni ammettono *opzioni* che specificano comportamenti particolari

- Iniziano per `-` seguite da una singola lettera
 - Oppure per `--` seguite da una stringa
-

Formato dei comandi

Formato:

SHELL

```
comando [opzioni] [argomenti]
```

Esempio: stampa il contenuto di `file.txt`

SHELL

```
cat file.txt
```

Esempio: lista il contenuto della cartella `dir`, includendo anche i file nascosti (che iniziano per `.`):

```
SHELL  
ls -a dir
```

```
SHELL  
ls --all dir
```

Concatenazione dei comandi

E' possibile avere più comandi con una sola riga, separandoli con `;`:

```
SHELL  
comando1 ; comando2; ...
```

Altri comportamenti:

- I comandi possono essere concatenati tramite il carattere `|` (questo concetto sarà ben noto come *pipe*).
- Si può redirezionare l'output di un comando su file tramite il carattere `>` (questo concetto sarà ben noto quando vedremo i *flussi stdin, stdout e stderr*).
- Analizzato in dettaglio più avanti

Comando manuale

Manuale in linea: i comandi sono documentati

```
man <comando>
```

Restituisce la pagina di manuale del `<comando>`. Particolarmente utile per capire gli *argomenti* e le *opzioni* del comando.

Comandi simili:

- **apropos**: ricerca in tutti i manuali dei comandi
- **whereis**: trova il binario, il sorgente e il manuale di un comando

Altri comandi di base utili

Altri comandi di base:

- **date**: visualizza la *data*
 - **who**: mostra gli *utenti attualmente collegati*.
 - **uptime**: tempo di vita di un sistema, numero di utenti collegati, carico del sistema negli ultimi 1, 5, 15 minuti; utile nell'ambito in cui si usano i *server*, dato che di solito vanno rimasti accesi *per sempre*. Con questo comando si vede se un *server* sia stato *riavviato o meno*, che potrebbe essere sorgente di problemi.
 - **hostname**: nome della macchina
-

File System

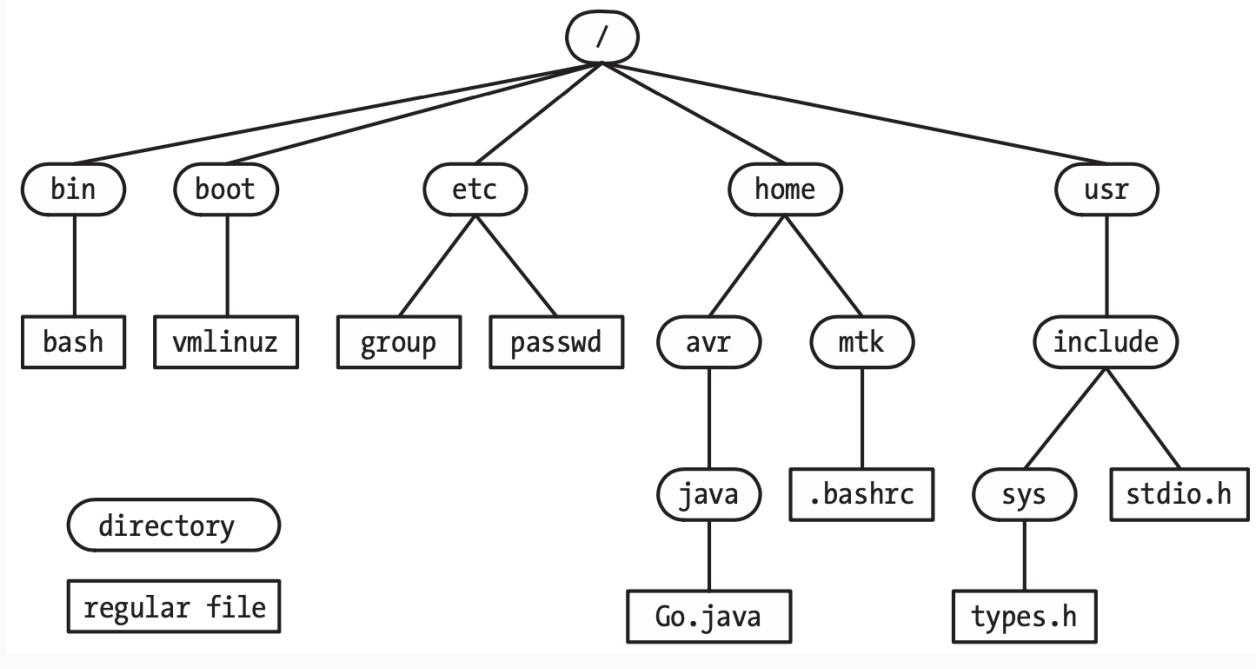
Organizzazione del File System

Il file system su Linux è *gerarchico*, ovvero *ad albero*.

- Organizzato in directory annidate l'una dentro l'altra
 - La directory radice è **/** (detta "*root*")
 - Tutte le cartelle del sistema sono contenute nella directory radice.
Esempio: la home degli utenti si trova in
/home/nomeutente
-

Esempio di File System

Esempio di albero (*parziale*) delle cartelle di un sistema Linux



Path

DEFINIZIONE DI PATH.

Un **path** identifica un **file o una cartella**.

Un terminale (e i comandi che vi vengono lanciati) sono **sempre posizionati** su una cartella, la **Working Directory**; ovvero "*da dove li eseguo*"

- Ci muove da una cartella all'altra col comando **cd <path>**
- Un path può essere:
 - Assoluto: inizia con **/** e indica un path completo a partire dalla radice
 - Relativo: **non** inizia con **/** e indica un path *a partire dalla Working Directory*
- Ci sono dei "**path speciali**":
 - Ci si può riferire directory corrente con **.**
 - La directory padre di quella corrente è indicata con **..**

Esempio di Path

Esempio: si consideri il seguente albero di cartelle

```

/
└── tmp
    └── directory
        └── file.txt

```

Col comando `cd /tmp` si posiziona il terminale in `/tmp`.

A questo punto:

- **directory** identifica la directory `/tmp/directory`
 - **.** identifica la directory `/tmp`
 - **..** identifica la directory `/`
 - **directory/..../** rappresenta la directory `/`
-

Strutturazione generale delle cartelle Linux

Su tutti i sistemi Linux, il file system è organizzato con *le seguenti cartelle di sistema*, necessari al *funzionamento del sistema operativo*. Di queste ne elenchiamo 16.

- `/`: radice
 - `/bin`: file *eseguibili* (e *preinstallati*) del sistema – ls, pwd, cp, mv
 - `/boot`: (*alcuni*) file necessari per l'avvio del sistema, *boot loader* ...
 - `/dev`: file speciali che descrivono i *dispositivi* (I/O) – dischi, scheda audio, porte seriali ...
 - `/etc`: file eseguibili, script, inizializzazione, *configurazione sistema* (ad esempio indirizzo di rete), file password, ...
 - `/home`: directory delle *home directory degli utenti*
 - `/lib`: *library di sistema* (ricordiamoci che queste *NON* sono i *system calls!*)
 - `/lost+found`: contiene i file *danneggiati*
 - `/mnt` : punto di *montaggio file system* (mount point) (per i *dispositivi rimuovibili*)
 - `/proc`: file system virtuale che contiene informazioni sui *programmi in esecuzione* (processi)
 - `/sys`: programmi di sistema (su *informazioni del sistema*)
 - `/tmp`: directory *temporanea*; il sistema operativo ha diritto di *svuotare* questa cartella per *certe situazioni* (quando non si ha abbastanza spazio) o per un certo *intervallo di tempo*.
 - `/usr`: file relativi alle *applicazioni installate*
 - `/usr/include`: header file libreria *standard C* (come `#include stdio.h`)
 - `/usr/bin` e `/usr/lib`: file binari disponibili agli utenti
 - `/var`: (*molto importante!*) file di sistema che *variano con frequenza elevata*; ad esempio è utile per le *storage delle web-app installate di sistema*.
-

Comando "list directory"

Comando `ls [opzioni] [dir]`: lista il contenuto della directory. Opzioni principali:

- `-1`: stampa su una colonna

- **-l**: formato lungo
- **-n**: come **-l** ma visualizza gli ID al posto del nome del proprietario e del gruppo
- **-t**: ordina per data
- **-s**: mostra la dimensione dei file in blocchi
- **-a**: mostra tutti i file compresi **.** e **..**
- **-R**: elenca il contenuto in modo ricorsivo

Esempio: **ls -ahl** è un utilizzo comune per utilizzare questo comando.

Esempio di utilizzo del comando **ls**

Esempio: differenti forme di **ls**

```
$ ls  
compile.txt style.css u1-introduzione u2-linux
```

```
$ ls -l  
total 16  
-rw-rw-r-- 1 martino martino 102 set 30 14:16 compile.txt  
-rw-rw-r-- 1 martino martino 199 set 30 15:27 style.css  
drwxrwxr-x 3 martino martino 4096 ott 118:33 u1-introduzione  
drwxrwxr-x 3 martino martino 4096 ott 4 10:20 u2-linux
```

Comando "remove"

Comando rm [-rf] [filename]: *rimuove* il/i file selezionati. Opzioni principali:

- **-r**: rimozione *ricorsiva* del contenuto delle directories.
- **-f**: rimozione di *tutti i file* (anche *protetti in scrittura*) senza avvisare.
- **-i**: con questa opzione **rm** chiede conferma

Esempio: cancella tutti i file in **cartella**

```
rm cartella/*
```

Nota: con ***** si intendono tutti i file dentro una cartella

NOTA BENISSIMO! Questo comando è *pericolosissimo*, quindi quando si scrive un comando che usa **rm** bisogna stare non attenti, ma di più! Ad esempio il comando

```
sudo rm -rf /*
```

è in grado di cancellare l'*intero computer* e tocca ri-installare un qualsiasi sistema operativo.

Comandi per modificare il File System

Comando cd <dir>: *cambia directory*. ("*change directory*")

Comando mkdir <dir>: crea sub-directory. ("*make directory*")

Comando rmdir <dir>: rimuove sub-directory, solo se vuota. Altrimenti fallisce. Questa è l'opzione più sicura, rispetto a **rm**. ("*remove directory*")

Comando cp <file1> <file2> e mv <file1> <file2>: *copia/sposta file o cartelle*. Opzioni principali

- **-f**: effettua le operazioni senza chiederne conferma
 - **-i**: chiede conferma nel caso che la copia sovrascriva il file di destinazione
 - **-r**: ricorsivo. Copia/sposta la directory e tutti i suoi file, incluso le sottodirectory ed i loro file
-

Collegamenti su Linux

Comando ln [-s] <sorgente> <destinazione>: crea un *link*. In Linux esistono due tipi di link:

- **HARD LINK**: associa un *secondo path* al contenuto del file. Se il primo file viene spostato, il link rimane valido e funzionante. E' l'*opzione di default* (!!)
 - *Robusto*: non può mai essere invalido. Non si può usare tra dischi diversi, né per linkare cartelle
 - **SOFT LINK**: è un *semplice rimando* a un altro path. Se il path destinazione non esiste o viene spostato, il link semplicemente non funziona. Si usa l'opzione **-s**.
 - *Flessibile*: può linkare a un altro file system o a una cartella
-

Ricerca di file nelle directories

Comando `find [path] [-n nome] [-print]`: ricerca ricorsiva di directories

Esempio: cerca i file che finiscono per `.txt` nella directory `/tmp`:

SHELL

```
find /tmp -name *.txt
```

E' possibile filtrare su varie **proprietà** dei file o cartelle:

- Tempo di creazione/modifica
- Utente o gruppo proprietario
- Grandezza

Nota: Non effettua ricerca nel *contenuto* del file, bensì solo *attributi* del file.

Stampare e creare (o toccare) file

Comando `cat <file>`: stampa il *contenuto* di un file

Comando `touch <file>`: *crea* il file se non esiste; altrimenti *modifica la data dell'ultimo accesso al file*

Esempio: creare un file `a.txt`, aprirlo con un editor e scrivervi dentro `ciao`, poi stampare il file

SHELL

```
$ touch a.txt  
... modificare con editor  
$ cat a.txt  
ciao
```

Visualizzare e scrivere su file

Comando `less <file>`: apre il file in un visualizzatore interno alla shell dove si può scorrere in entrambe le direzioni, utile per i *file lunghi*

Esistono svariati *altri comandi per visualizzare il contenuto* di un file.

- Comandi per stampare file binari (`hexdump`)

- Comandi per stampare le prime (**head**) o le ultime righe (**tail**) di un file
 - *Editor avanzati* utilizzabili dentro la shell.
 - **nano** il più semplice
 - Ne esistono molti. Sono in competizione **emacs** e **vi** (o **vim**), detta *Guerra degli editor*. Qui gli editor diventano una specie di "*religione*" per i programmati.
-

Utenti e permessi

UNIX è un sistema multiutente

Un dispositivo con OS Linux può avere più *utenti* (infatti si dice che è un *sistema multiutente*).

- Essi possono fare login su una *shell* o un *terminale remoto* (SSH)
 - Ogni utente ha la sua *Home Directory* in **/home/<utente>**
Serve per permettere all'utente di immagazzinare file personali come documenti, immagini, programmi.
-

Utenti e permessi

Un utente può essere assegnato a uno o più *gruppi*.

- Ogni utente deve essere associato ad *uno ed uno solo gruppo primario**
- Eventualmente un utente può essere assegnato a più *gruppi secondari*
Meccanismo utente-gruppi serve per gestire l'accesso a file e risorse.

L'utente **root** esiste sempre ed ha massimi privilegi

ATTENZIONE! Non bisogna *assolutamente* confondere l'*utente root* con il *kernel-mode!* ([Definizioni Relative ai Sistemi Operativi > ^33592a](#)) In ogni caso si avviano le applicazioni *SEMPRE* in *user-mode!*

Comandi relativi agli Utenti e ai Permessi

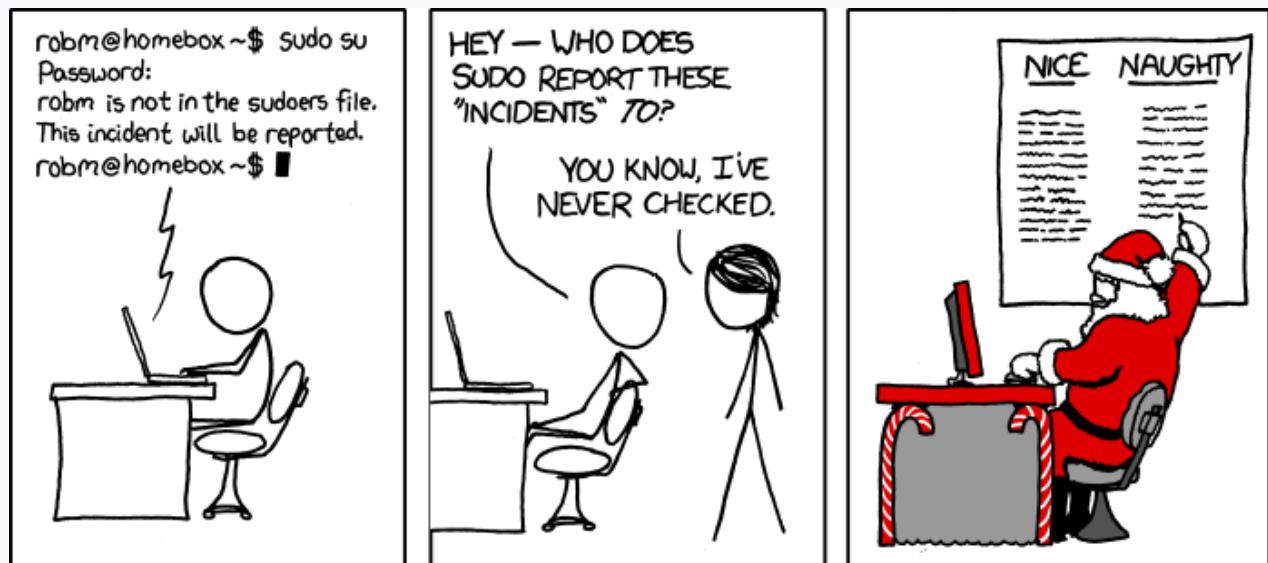
Gestione: Comandi per creare o rimuovere utenti e gruppi: **useradd**, **groupadd**, **userdel**, **groupdel** (*di solito sono complicati da usare*)

- Su molti OS Linux, esistono dei **comandi più facili da usare**: **adduser**, **addgroup**, **deluser**, **delgroup**

Altri comandi:

- **groups**: stampa i gruppi ai quali appartiene l'utente corrente
- **whoami**: stampa l'**utente corrente** (per dubbi esistenziali?)
- **su <utente>**: **cambia utente** (chiede password)
- **sudo <comando>**: esegue il comando **come utente root**, dopo aver chiesto la password
 - Nota: **sudo** sta per "*super user does*"
 - Nota 2: Se si tenta di usare questo comando con un utente che non *ha il privilegio di usare sudo*, si vede questo spaventoso messaggio: **<user> is not in the sudoers file. This incident will be reported.**; ovvero si dice che l'*incidente* verrà "*segnalato*". Nel passato (fino ad un anno fa) veniva effettivamente *segnalato* tramite una *mail* all'**amministratore effettivo**. Adesso questo tentativo viene semplicemente registrato sul file **/var/log/auth.log**, se non specificato (per ulteriori dettagli vedere il commit <https://github.com/sudo-project/sudo/commit/6aa320c96a37613663e8de4c275bd6c490466b01>)

FIGURA: Babbo natale che controlla la lista dei non-sudoers cattivi



Tipologie di Permessi

I file e le cartelle hanno tre tipi di **permessi**:

- 1. Permesso di **Lettura**: Per i file, *accedere a contenuto*. Per cartelle, *listare i file*.
- 2. Permesso di **Scrittura**: Per i file, *modificare il contenuto*. Per le cartelle, *creare file o cartelle in essa* (alterare la lista).
- 3. Permesso di **Esecuzione/Attraversamento**:

- Per i **file**, esiste il permesso di **esecuzione**. Necessario per *eseguire programmi*.
- Per le **cartelle**, esiste il permesso di **attraversamento**. Necessario per *accedere a sotto cartelle*.

NOTA BENE. Con i *permessi* non c'è *nessuna eredità*; ad esempio nel caso in cui un utente ha il permesso di *scrivere* su un file, questo non vuol dire che questo utente ha necessariamente anche il permesso di *leggere* su questo file; se l'utente ha il *solo permesso di scrivere sul file*, allora questa è l'unica cosa che può fare (anche se potrebbe risultare strana come cosa).

Utente e Gruppo proprietario

DEFINIZIONE DI UTENTE/GRUPPO PROPRIETARIO.

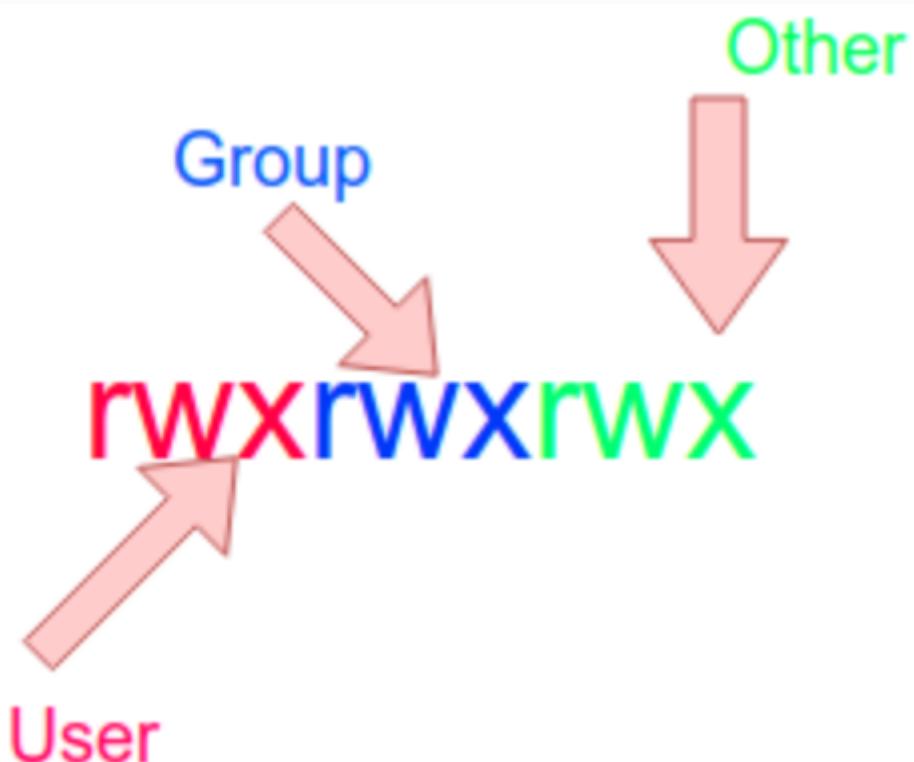
I file e le cartelle hanno *uno ed uno solo utente proprietario* e un *gruppo proprietario*.

I permessi su file sono gestibili separatamente per:

- Utente proprietario del file
- Utenti del gruppo proprietario del file
- Tutti gli altri utenti

Per riassumere, ad ogni file abbiamo *nove permessi*, separati per tipologia del permesso e per tipologia dell'utente: in totale ogni file o cartella ha 3×3 permessi, in formato ***rwxrwxrwx***, i *primi tre dedicati* per l'*utente proprietario*, i *secondi tre* per il *gruppo proprietario* e gli *ultimi tre* per *tutti* gli altri (quindi in ordine "*U_rU_wU_xG_rG_wG_xO_rO_wO_x*").

- Ogni permesso può essere attivo o no (quindi 1 o 0)



Esempio:

```
$ ls -l  
-rw-rw-r-- 1 martino docenti 102 set 30 14:16 compile.txt  
-rw-rw-r-- 1 martino docenti 199 set 30 15:27 style.css
```

Esempio generale sui file:

si considerino le seguenti informazioni sul file **my-program**:

```
-rwxr-xr-- 1 martino docenti 102 set 30 14:16 my-program
```

- L'utente **luca** del gruppo **docenti** può eseguire **my-program**?
SI: il gruppo ha permessi **r-x**, quindi **luca** può eseguire **my-program**
- L'utente **marco** del gruppo **studenti** può eseguire **my-program**?
NO: gli altri hanno permessi **r--**, quindi **marco** non può eseguire **my-program**

Esempio generale sulle cartelle:

si considerino le seguenti informazioni sulla cartella **data**:

```
dr-xr--r-- 3 martino docenti 4096 ott 118:33 data/
```

- L'utente **luca** del gruppo **docenti** può listare i file?
SI: il gruppo ha permessi **r--**
- L'utente **luca** del gruppo **docenti** può accedere alla cartelle dentro **data**?
NO: il gruppo ha permessi **r--**. Servirebbe **r-x**
- L'utente **martino** del gruppo **docenti** può creare file in **data**?
NO: l'utente **martino** ha permessi **r-x**. Servirebbe **rwx**

Comando per modificare i permessi di un file

1. Modifica dei permessi di un file:

si usa il comando **chmod [-r] <permessi> <file>**

I permessi possono essere indicati con (principalmente) **due sintassi: assoluto e mirato**.

- **Assoluto**, con **tre cifre ottali**, che rappresentano rispettivamente i **permessi a utente, gruppo e altri**. Ogni cifra ha **3 bit** e rappresenta **permessi di lettura, scrittura ed esecuzione/attraversamento**.
Esempio: **chmod 750 file.txt** dà permessi totali a utente ($7_8 = 111_2$), lettura/esecuzione al gruppo ($5_8 = 101_2$) e niente agli altri ($0_8 = 000_2$)
- **Mirato**: Modifica permessi esistenti tramite una stringa composta di 3 parti:
 - Quali utenti: **u** (user), **g** (gruppo), **o** (other) (**chi?**)

- Che operazione: **+ (aggiungi)**, **- (rimuovi)** (**cosa?**)
- Quale permesso: **r (lettura)**, **w (scrittura)**, **x (esecuzione/attraversamento)** (**quale?**)
Esempio: **chmod g+w file.txt** dà permessi in scrittura agli utenti del gruppo proprietario del file
- **-r** applica il comando ricorsivamente a file e cartelle contenute
- **Chi può modificare i permessi:** Utente proprietario e utente **root**

Esempio: usi di **chmod**

- **chmod 600 file.txt**: l'utente può leggere e scrivere. Il gruppo e gli altri niente.
- **chmod 640 file.txt**: l'utente può leggere e scrivere. Il gruppo può leggere. Gli altri niente
- **chmod u+x file.txt**: Aggiungi i permessi di esecuzione all'utente
- **chmod go+w file.txt**: Aggiungi i permessi di scrittura al gruppo e a gli altri

2. Modifica di proprietario e gruppo di file o cartella

- **chown utente file**: modifica utente proprietario ("change owner")
- **chgrp gruppo file**: ("change group")
- **chown utente:gruppo file**: modifica contemporaneamente entrambi
- **Note:**
 - Posso assegnare un file *solo a un gruppo che posseggo*
 - Sulla maggior parte degli OS, solo **root** può cambiare utente proprietario; non esiste il "give-away" dei file.
 - Opzione **-r**: applica il comando *ricorsivamente* a cartelle e file contenuti

Processi e programmi

Processi e programmi

DEFINIZIONE DI PROCESSO (RICHIAMO).

Un processo è un programma in esecuzione.

In Linux, ogni processo è identificato da un'identificatore detto **PID**.

Il **PID** si usa per *effettuare operazioni sul processo*.

Comandi Relativi ai Processi

Abbiamo i seguenti comandi per gestire i processi.

- **kill <PID>**: termina il processo (*se ho i privilegi opportuni!*)

- **top**: mostra in maniera interattiva i *processi in esecuzione*. Simile a un Task Manager via Shell
- **ps [opzioni]**: mostra informazioni sui processi attivi.
 - **a**: informazioni su tutti i processi (non solo generati dalla sessione shell corrente)
 - **x**: mostra anche i processi n background
 - **f**: stampa i processi in modo che se ne veda il rapporto padre-figli (ovvero l'*albero dei processi*)
 - **u**: stampa più informazioni
Esempio: **ps fax** è un utilizzo molto comune di questo comando
 - **Nota:** **ps** è tra i pochi programmi in cui le opzioni non vanno iniziate con **-**. Ciò è un *relitto* delle primissime versioni di Unix in cui le opzioni non avevano il **-**.
- **lsusb**: lista i dispositivi usb
- **lspci**: lista i dispositivi su bus pci
- **lsblk**: lista i dischi
- **ifconfig**: lista le interfacce di rete (*lo stato della rete*)
- **pwd**: stampa la *directory corrente*
- **free**: mostra quanta memoria *RAM libera* ed occupata ha il sistema
- **df [-htv]**: visualizza *informazioni sui file system del sistema* (in particolare lo *stato di occupazione dei dischi*).
 - **-t** : nr totale di blocchi e i-node liberi
 - **-v** : percentuale di blocchi e i-node
 - **-h** : stampa in GB/MB anzichè in numero di byte

SHELL

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
udev	3923948	4	3923944	1%	/dev
tmpfs	787220	1552	785668	1%	/run
/dev/sda6	425085288	259786508	143682652	65%	/
none	4	0	4	0%	/sys/fs/cgroup
none	5120	0	5120	0%	/run/lock
none	3936080	436	3935644	1%	/run/shm
none	102400	44	102356	1%	/run/user

Esercizi

Esercizi

1. Stampare il contenuto del file **/etc/hosts**
2. Posizionarsi nella cartella **/tmp** e listare il contenuto della cartella **/home** usando un path relativo e uno assoluto
3. Usare l'editor nano per creare un file **file.txt** in una qualsivoglia cartella. Creare una link simbolico del file e cancellare l'originale. Cosa succede se si prova a stampare il contenuto del link? Ripetere con Hard Link
4. Creare una cartella e due file in essa. Cancellare la cartella con un unico comando.
5. Creare un nuovo gruppo **studenti** e un utente **studente** assegnato a tale gruppo.
Nota: usare le opzioni **-m -g <group>** del comando **useradd**
E' necessario usare **sudo**?
Un utente normale può listare i file nella home della home directory di **studente**?
Modificare i permessi della home di **utente** affinchè tutti possano leggere, scrivere ed eseguire

Soluzioni

1. Basta scrivere **cat ~/etc/hosts**
2. Indipendentemente da dove si trova, prima si scrive **cd ~/tmp**. Per il path relativo si scrive **cd ..~/home**; per il path assoluto si scrive **~/home**.
3. **touch file.txt**. Con un *soft link*, diventa impossibile leggere il file. Con un *hard link*, si puo' comunque leggere il file.
4. **rm -r(i) cartello** (la parte -i sarebbe opzionale, anche se è saggio usarlo)
5. Sì, è necessario usare sudo per creare il gruppo e l'utente. No, l'utente normale non può listare i file nella home della home directory di studente. Basta scrivere **chmod 777 /home**

u2-s3-programmazione-bash

Sistemi Operativi

Unità 2: Utilizzo di Linux

Programmi in Bash

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Script Bash
2. Variabili

3. Strutture di controllo

4. Esercizi

Script Bash

Definizione di Bash

Con **Bash** si intende il **software shell di default** in GNU/Linux

Permette di:

- Eseguire comandi (già visto)
- Definire variabili e utilizzarle
- Controllare il flusso (**if**, **while**, ecc...) con i vari **costrutti**

E' un **linguaggio completo** di tutti i costrutti.

Ha una sintassi particolare e **problematica**; nel senso che è molto "**particolare**" e "**pedantica**" nella sua **sintassi**.

Una lista di comandi può essere racchiusa in un file detto **script**.

Esempio di Script Bash

Uno script di esempio nel file **script.sh**:

SHELL

```
#!/bin/bash
# primo esempio di script
echo $RANDOM
```

#!/bin/bash Indica che il file è uno **script bash**; questa è **obbligatoria!**

primo esempio di script E' un commento

echo \$RANDOM Stampa il **contenuto** della variabile **\$RANDOM**

Per eseguire lo script.

```
./script.sh
```

Il file **deve** avere permessi di lettura ed esecuzione; quindi bisogna anche usare **chmod u+x script.sh** per aggiungere i permessi opportuni.

Il valore di Ritorno in Bash

Uno script è una lista di comandi che vengono eseguiti più delle strutture di controllo.

SHELL

```
ls # Lista i file  
./myprog # Avvia il programma myprog
```

Ogni processo in Linux/POSIX deve fornire un **Valore di Ritorno** al chiamante, ovvero:

- La **shell**
- Uno **script bash**
- Un **qualsiasi altro processo**

Il valore di ritorno è un **valore numerico intero**

- Usato dal chiamante per vedere se c'è stato errore
- Per convenzione 0 se **successo**, ≠ 0 in **caso di errore**

In uno script bash, si accede al Valore di Ritorno dell'ultimo comando tramite la variabile speciale **\$?**

Accedere ai parametri di riga del comando

Ricordare che i **script Bash** sono dei **programmi**, ergo possono usare i **parametri** forniti dalla riga del comando.

Si **ottengono con**:

- **\$1, \$2, ...**: **contenuti dei parametri**
- **\$0**: **nome dello script**
- **\$#**: **numero di argomenti**

Esempio: il nome dello script è **script.sh** e viene eseguito come **./script.sh ciao**.

```
#!/bin/bash
echo $0 # Stampa "./script.sh"
echo $1 # Stampa "ciao"
echo $# # Stampa "1"
```

Variabili

Concetti preliminari per le Variabili

REGOLE PER LA DENOMINAZIONE DELLE VARIABILI.

1. Sono una combinazione illimitata di lettere, numeri e underscore.
2. *Non* possono *cominciare con numeri* e sono *CASE sensitive*.

COMANDO READ PER LEGGERE LE VARIABILI DA STDIN.

Istruzione **read nomevar** (stesso compito di **scanf**)

```
read nome
```

COMANDO ECHO PER STAMPARE LE VARIABILI SU STDOUT.

Si utilizza **echo** (stesso compito di **printf**)

```
echo "Testo su schermo"
```

ACCESSO ALLE VARIABILI.

Si accede con **\$nomevar**.

```
read x # Legge X da terminale
y=$x # Assegna a y il valore di x
echo $y # Stampa quanto letto
```

I DUE TIPI DI VARIABILI.

I tipi principali sono solo i seguenti:

- **Stringhe:** `a="testo"`
- **Interi:** `a=47`

Nota: non inserire spazi prima e dopo `=` durante assegnazione

Quoting per le stringhe

Le stringhe vanno racchiuse tra `'` o tra `"`.

Il carattere `\` indica il quoting. Permette di usare nella stringa *il carattere di quoting* (un'escape se vuoi)

Esempio:

`a="ciao"` indica la stringa `ciao`

`a='ciao a tutti'` indica la stringa `ciao a tutti`

`a=="ha detto: \"ciao\""` indica la stringa `ha detto: " ciao"`

DIFFERENZE TRA QUOTING IN `'` E IN `"`.

Le stringhe definite con `'` possono contenere delle variabili che vengono valutate.

SHELL

```
a="test"  
b="this is a $a"  
c='this is a $a'
```

La variabile `b` contiene `this is a test`

La variabile `c` contiene `this is a $a`

Esempio generale: leggere due stringhe da tastiera e stamparle.

SHELL

```
read a  
read b  
echo $a $b
```

Operazioni Matematiche con le Variabili

Solo numeri interi con segno (ovvero non esistono i float)

- Se si usano valori floating non segnala errore ma fa i calcoli con numeri interi (ovvero il risultato verrà troncato)

Operazioni ammesse: + - * / % << >> & ^ (or esc.) |

SINTASSI.

L'espressione `$((var1 + var2))` restituisce la somma di due variabili

Alternativamente scrivere `$(($var1 + $var2))` è equivalente (quindi il dollaro qui non è importante)

ESEMPIO: Si scriva un programma che legge due interi da tastiera e stampa il prodotto

SHELL

```
#!/bin/bash
read a
read b
c=$(( a * b ))
echo "Il prodotto è $c"
```

Note:

Osservare la forma `c=$((a * b))`

Osservare la concatenazione naturale in `echo "Il prodotto è $c"`

Strutture di controllo

Struttura if-then-elif-else-fi

Le condizioni hanno forme

```

if condizione then
    ramo 1
elif condizione2 then
    ramo 2
else
    ramo alternativo
fi

```

Esistono molte *sintassi alternative* per esprimere le *condizioni*. Ne vedremo una parte.

Condizioni tra Numeri

Si utilizza la sintassi **((espressione))** (*attenzione che qui NON c'è il dollaro!*)

Gli operatori di confronto sono i classici: `==` `!=` `<` `>` `<=` `>=`

Esempio:

```

read n1 n2
if (( n1<n2 ))
then
    echo "$n1 minore di $n2"
elif (( n1==n2 )) then
    echo "$n1 uguale a $n2"
else
    echo "$n1 maggiore di $n2"
fi

```

Condizioni tra Stringhe

Si utilizza la *sintassi* **[[espressione]]**

Gli operatori di *confronto* sono:

- `=` `!=`: uguaglianza o differenza (*ATTENZIONE! QUI SI USA UNA SOLA =*)
- `>` `<`: ordinamento alfabetico
- `-z`: vero se la stringa è vuota (*o la variabile non è definita*; questa è unica per Bash);
`! -z` è vero se la variabile non è vuota (*o se è definita*)
- E' *necessario* usare l'operatore `$` e mettere *spazi tra operandi*
Esempio: **if [[\$a != \$b]]**

Esempio: **if [[! -z \$var]]**: vero se **var** esiste e non è vuota (questo è molto comune in Bash, dal momento che **non esistono** le eccezioni)

Esempio:

SHELL

```
#!/bin/bash
read s1
read s2
if [[ $s1 = $s2 ]]
then
    echo "Le stringhe sono uguali"
else
    echo "Le stringhe sono diverse"
fi
```

Condizioni su File

E' molto semplice testare **se un file (ovvero sui path) esiste, è vuoto o è una cartella**

- (**esiste?**): **-a path**: vero se **path** esiste
- (**è un file o una cartella?**):
 - **-f path**: vero se **path** è un file
 - **-c path**: vero se **path** è una cartella
- (**non è vuoto?**): **-s path**: vero se **path** non è vuoto
- (**ho i permessi?**)
 - **-r path**: vero se posso leggere **path**
 - **-w path**: vero se posso scrivere **path**
 - **-x path**: vero se eseguire/attraversare leggere **path**

N.B. Per le condizione si usano le parentesi quadre **[[...]]**, dato che si trattano di stringhe.

Esempio: si scriva un programma che legge due path da tastiera. Se sono uguali, controlla che il path corrisponda a una cartella. Se affermativo, stampa il path.

```
#!/bin/bash
echo "Inserisci il primo path:"
read s1
echo "Inserisci il secondo path:"
read s2
if [[ $s1 = $s2 ]]
then
    if [[ -d $s1 ]]
    then
        echo "$s1 è una cartella"
    else
        echo "$s1 non è una cartella"
    fi
else
    echo "Le due stringhe non sono uguali"
fi
```

Operatori logici

Si possono creare **condizioni composte** con gli **operatori booleani**

- **&&**: and
- **||**: or
- **!**: not

Sintassi: **if condizione1 && condizione2**

Esempio: **if ((a>b)) && [[\$c="hello"]]**

OSS. Il Bash è un **linguaggio "lazy"**, ovvero nel senso che se abbiamo condizioni composte, verranno eseguite in una maniera più "**ottimale**"; nel senso che se, ad esempio abbiamo **if condizione1 && condizione2** e abbiamo che la **prima condizione** è **falsa**, allora Bash non verrà **mai controllato** (dunque eseguito). Vedremo a seguito come sarà utile questa caratteristica.

OSS 2. Ricordandoci che ogni programma deve **fornire un valore di ritorno**, posso utilizzare la precedenza osservazione per eseguire comandi **secondo una logica voluta**

- Ogni comando/programma avviato in bash fornisce alla shell/script chiamante un valore di ritorno
- Per convenzione un comando ritorna: 0 se successo, $\neq 0$ in caso di errore
 - In bash, il valore 0 è interpretato come **true**; un valore $\neq 0$ come **false**

- NOTA: diverso da altri linguaggio come C o Java!

CONSEGUENZA: Utilizzo in espressioni di comandi

Ora, combinando il fatto che Bash è un *linguaggio lazy* e che i programmi *devono fornire un valore di ritorno*, abbiamo il seguente utilizzo di comandi.

Esegue **comando2** se **comando1** non dà errore

SHELL

```
comando1 && comando2
```

Esegue **comando2** se **comando1** dà errore

SHELL

```
comando1 || comando2
```

Esempio: eseguo **myprog** solo se la compilazione è andata a buon fine

SHELL

```
gcc myprog.c -o myprog && ./myprog
```

Esempio: eseguo un gestore dell'errore **gestione_err** se un'istruzione **istruzione** non è andata a buon fine

SHELL

```
istruzione || gestione_err
```

Cicli **for**

Abbiamo *due modi* per esprimere un *ciclo for* in Bash.

1) Versione semplice

Scelgo un "*iterabile*" (una successione di numeri oppure una stringa) e ci itero, come se fossimo su Python

SHELL

```
for n in 1 2 3 4
do
    echo "valore di n = $n"
done
```

SHELL

```
for nome in mario giuseppe vittorio
do
    echo "$nome"
done
```

2) Versione completa

Questa sintassi si dice "*stile-C*".

Sintassi:

SHELL

```
for ((INITIALIZATION; TEST; STEP))
do
    [COMMANDS]
done
```

Esempio:

SHELL

```
for ((i = 0 ; i <= 1000 ; i++))
do
    echo "Counter: $i"
done
```

Cicli **while**

SINTASSI:

SHELL

```
while condizione  
do  
...  
done
```

ESEMPI:

SHELL

```
n=0  
while ((n<4))  
do  
((n=n+1))  
echo $n  
done
```

SHELL

```
n=0  
until((n>4))  
do  
((n=n+1))  
done
```

Esercizi Bash

1. Si scriva un programma che riceve due argomenti. Se entrambi sono dei file, stampa il contenuto di entrambi
2. Si scriva un programma che per ogni file/cartella nella cartella corrente dice se esso è un file o una cartella.
3. Si scriva un programma che riceve un intero come argomento. Se esso è minore di 10, crea i file **0.txt**, ..., **9.txt**

Soluzioni agli Esercizi

1. Esercizio 1

```
#!/bin/bash
if [[ "$#" != "2" ]]
then
    echo "Servono due argomenti"
else
    if [[ -f $1 ]] && [[ -f $2 ]]
    then
        cat $1
        cat $2
    else
        echo "Non sono due file"
    fi
fi
```

2. Esercizio 2

```
#!/bin/bash
for file in *
do
    if [[ -f $file ]]
    then
        echo "$file è un file"
    elif [[ -d $file ]]
    then
        echo "$file è una cartella"
    fi
done
```

3. Esercizio 3

```
#!/bin/bash
if [ "$#" != "1" ]
then
    echo "Serve un argomento"
else
    if (( $1 < 10 ))
then
        for (( i=0; i<$1 ; i++ ))
        do
            touch $i.txt
        done
    else
        echo "L'argomento non è minore di 10"
    fi
fi
```

```
Dataview (inline field '='): Error:
-- PARSING FAILED -----
-----
> 1 | =
| ^

Expected one of the following:

', 'null', boolean, date, duration, file link, list ('[1,
2, 3]'), negated field, number, object ('{ a: 1, b: 2 }'),
string, variable
```

u2-s4-comandi-bash

Unità 2: Utilizzo di Linux

Comandi in Bash

Argomenti

1. Pipe e redirect
 2. Filtri e simili
 3. Esercizi
-

Pipe e Redirect

I standard di comunicazione in Linux

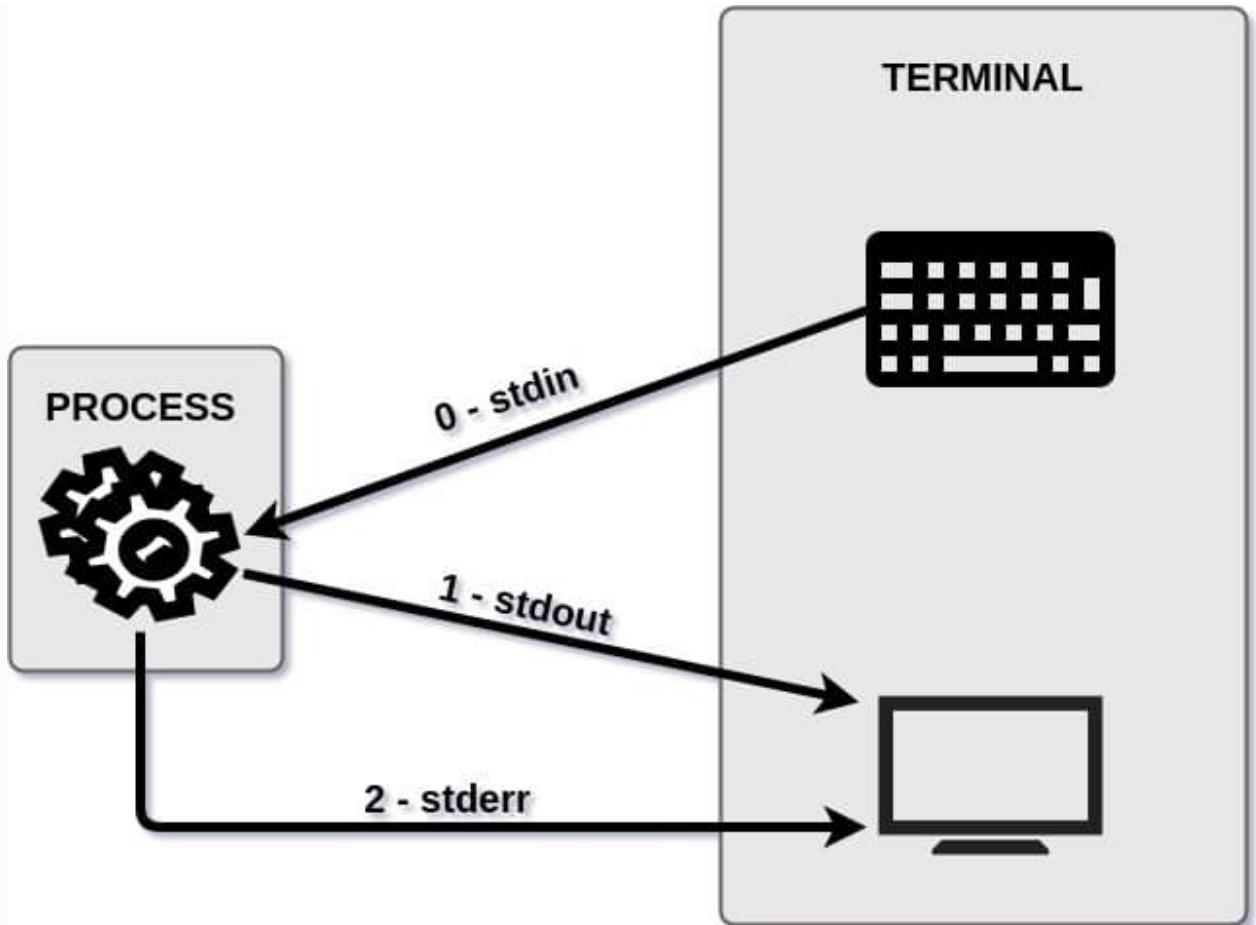
NOZIONE PRELIMINARE. In Linux, ogni processo, ha a disposizione *3 canali standard di comunicazione* con l'*esterno* (ovvero i dispositivi I/O).

- **Standard Input (`stdin`)**: per ricevere dati in ingresso.
- **Standard Output (`stdout`)**: per stampare l'output
- **Standard Error (`stderr`)**: per stampare eventuali errori

Questa convenzione è *caratteristica* per i sistemi *POSIX-like* (Storia e Definizione di Linux > ^1ad28a).

Di default, un programma riceve lo Standard Input da tastiera, e stampa Standard Output e Standard Error su console.

FIGURA: Illustrazione grafica dei canali di comunicazione



CONSEGUENZE. Questo implica quello che già abbiamo visto:

- **read** legge da **stdin**, quindi *di default da tastiera* (quindi non *direttamente* dalla *tastiera!*)
- **echo** stampa su **stdout** che *di default è console* (quindi non *direttamente* sullo *schermo!*)
- Per stampare su **stderr**, si può usare: **echo "An error!" >&2**. Di default, lo **stderr** è visualizzato a schermo
Tutti i programmi ben scritti, devono attenersi a usare questi *canali standard*.
- Ciò permette una *grande flessibilità* (flessibilità)
- *Tutti* i programmi di default di Linux *lo fanno* (universalità)

Redirezionare i canali (redirect)

1. Redirezione **stdout su file:** è possibile eseguire un programma e *redirezionare* lo **stdout** su file anziché stamparlo sul terminale (come di default)

- **Formato:** **comando > file** oppure **comando 1> file**
- Questo perché 1 indica **stdout** mentre 2 indica **stderr**

Esempio: **date > data.txt** La data corrente viene salvata in **data.txt** e non stampata ad output

Nota: se **file** esiste, il contenuto viene sovrascritto, a meno che si usa la *modalità append*.

2. Append **stdout su file:** simile alla *redirezione*. Il file non viene cancellato, ma lo **stdout** del programma *viene aggiunto in coda*.

- **Formato:** **comando >> file** oppure **comando 1>> file**

Esempio: Scrivere su un file la data due volte, con un intervallo di 5 secondi

SHELL

```
date > file.txt
sleep 5 # Pausa di 5 secondi
date >> file.txt # "Appende" a file.txt
```

Esempio: Si scriva un programma che riceve due argomenti. Ricerca nella folder corrente tutti i file che hanno il nome del primo argomento e salva la lista nel file il cui nome è il secondo argomento

SHELL

```
#!/bin/bash
if (( "$#" != "2" ))
then
    echo "Servono due argomenti"
else
    find . -name $1 > $2
fi
```

3. **stderr su file:** permette di redirigere lo **stderr** su un file.

- **Formato:** **comando 2> file**
- Questo perchè 1 indica **stdout** mentre 2 indica **stderr**

4. **stdin da file:** permette di prelevare da file anziché da tastiera lo **stdin** un programma; quindi si *"dirotta"* il flusso di **stdin**, rimpiazzando lo *default* (ovvero la tastiera) con il **file**.

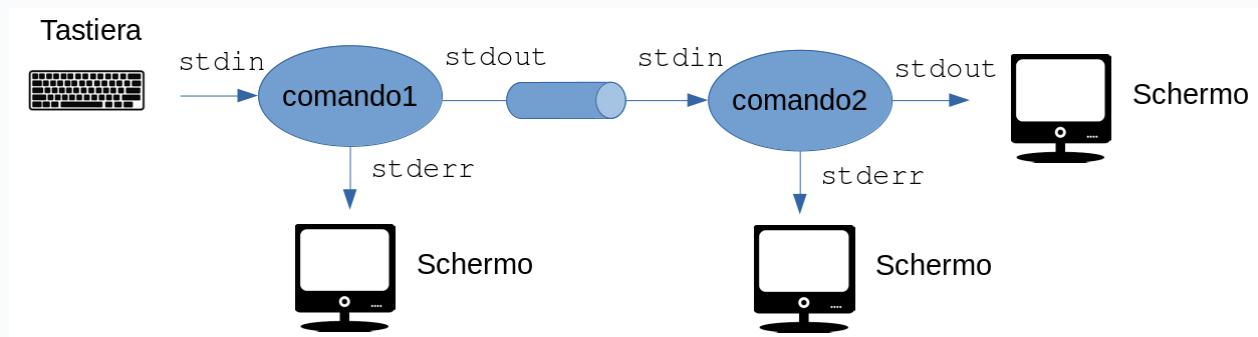
- **Formato:** **comando < file**

5. Pipe: è possibile redirezionare lo **stdout** di un primo comando nello **stdin** di un secondo. Ovvero avrà una situazione del tipo ...→*comando1*→**stdout**→**stdin**→*comando2*→.... In essenza, sto *concatenando* due comandi, creando così una *catena di processi*.

- **Formato:** **comando1 | comando2**

- **NOTA.** E' uno dei costrutti più potenti del bash, siccome permette di svolgere *compiti complessi con poco codice*: ne vedremo le potenzialità assieme ai comuni comandi bash

FIGURA: Illustrazione del concetto di pipe



6. Sostituzione in variabile: è possibile usare lo **stdout** di un comando come una variabile.

- **Formato:** `$!(comando)` oppure `'comando'`

Esempi:

- `a=$(ls /tmp)`: la stringa `a` contiene l'elenco dei file in `/tmp`
- `rm $(find / -name "*.tmp")`: rimuove tutti i file nel sistema che terminano per `.tmp`

Filtri e simili

L'idea di filtrare testi

L'IDEA PRELIMINARE. Negli OS Linux esistono una serie di comandi per manipolare testo.

- Filtrare, ordinare, comporre in una *maniera arbitraria*

Essi si aspettano di lavorare *su dati testuali organizzati in righe*, come normali file di testo (o di configurazione) (o file `.csv`)

Permettono di fare operazioni complesse con *poco codice*

- Spesso si usano assieme alle **pipe** al fine di creare *pipeline di processamento*

Comando **grep**

Il comando **grep** è uno dei comandi *utili per filtrare dei contenuti*.

grep [options] pattern [file...]: stampa le linee del file che contengono il *pattern*. Se non metto il file usa lo standard input: posso usare grep in pipe. Alcune opzioni:

- **-n**: stampa il numero di riga
- **-i**: case insensitive
- **-c**: stampa il numero di match
- **-v**: stampa solo le linee che *non* contengono il pattern
- **-e**: interpreta i pattern come delle *espressioni regolari*

Esempio dell'utilizzo di **grep**:

- **grep main *.c**: stampa le linee che contengono **main** in tutti i file che finiscono in **.c**
- **ps -ef | grep bash**: stampa tutti i processi che sono istanze del programma **bash**
Osserviamo che ci sono dei *caratteri speciali* che vengono processati in qualche modo da Bash. Questo è *Bash Expansion*.

Bash Expansion

OSS. Osservare il comando **grep main *.c**:

- Esso ricerca il pattern **main** in *tutti* i file che terminano con ***.c**

La **bash** (quindi NON il comando/programma grep!) *espande* il termine ***.c** in tutti i file che matchano l'espressione **prima** di eseguire il comando

- **grep** non riceve la stringa ***.c** come argomento
- **grep** riceve già la lista di file

Esempio: la cartella corrente contiene **prog.c** e **module.c**

- Il comando **grep main *.c**
- Viene trasformato dalla bash nel comando **grep main module.c prog.c**

Questo è un *meccanismo* flessibile per operare su file:

- Agisce *prima* di avviare il comando
- Il testo non deve essere quotato né con ***** né con **?**
- ***** matcha qualsiasi numero di ogni caratteri (*wildcard*)
- **?** matcha un solo carattere (*single wildcard*)
- **~** rappresenta la home directory:
 - **~/file.txt** equivale a **/home/martino/file.txt** se l'utente è **martino**
 - **NOTA!** **.** e **..** non vengono espansi, *sono propriamente parte di un path*
- Liste racchiuse tra **{ ... }** vengono espanso
 - **mkdir /tmp/{dir1,dir2}** viene "espanso" in **mkdir /tmp/dir1 /tmp/dir2**
(attenzione che in realtà *non sono equivalenti*, tuttavia hanno lo stesso effetto).

Esempio: si scriva un programma che riceve due argomenti: il primo argomento è una cartella, il secondo argomento un pattern. Il programma trova tutte le linee dei file **.c** o **.h** nella cartella, che contengono il pattern. Le linee vengono salvate nel file **/tmp/output.txt**.

SHELL

```
#!/bin/bash
if (( "$#" != "2" ))
then
    echo "Servono due argomenti"
else
    cat $1/*.c $1/*.h | grep $2 > /tmp/output.txt
fi
```

Comando **cut**

cut: estrarre colonne (o campi) dall'input. Ha *diverse modalità*, di cui ne vedremo *due*; modalità *byte* e modalità *campi*.

- **Modalità byte**: estraie i byte specificati da ogni riga. Si utilizza l'opzione **-b** **byterange**
- **Modalità campi**: estraie i campi specificati, delimitati da un separatore specifico. Si utilizza l'opzione **-d delimitatore -c campi**. Questo è particolarmente utile per i file **.csv**.

Esempio: il file **file.txt** contiene:

```
luca 1985 milano
martino 1990 torino
```

cat file.txt | cut -b1-2 estraie i primi *2 byte* (caratteri) da ogni riga, e stampa su **stdout**:

```
lu
ma
```

cat file.txt | cut -d " " -f 2 estraie il *secondo campo del file*, delimitato da uno spazio. Stampa su **stdout**:

1985

1990

Comando **tr** (translate)

tr [-cds] [set1] [set2]: legge dei dati e sostituisce i caratteri specificati con altri caratteri (quindi fa una specie di "translate" automatica). Opzioni comuni:

- **-d**: (del) cancella tutti i caratteri specificati. E' necessario un solo set come argomento
- **-s**: sostituisce le ripetizioni del carattere specificato con un solo carattere

Esempio: **tr a A < file1 > file2**: sostituisce le a minuscole con A maiuscole. Notare lo **stdin** di **tr** è letto da file con l'operatore <. Qui si ha una situazione del tipo (*file1*→*tr*)->*file2*

Comando **sort** (ordinamento)

sort [-dfnru] [-o outfile] [file...]: Ordina i dati del file o dello **stdin**. Di default si usa l'ordinamento alfabetico. Opzioni principali:

- **-f**: tratta maiuscole come minuscole (*case insensitivity*).
- **-n**: riconosce i numeri e li ordina in modo *numerico* (*numerical*).
- **-r**: ordina i dati in modo inverso (*reverse*).
- **-k**: ordina secondo il numero di colonna dato dopo il k (*su file a campi*)
- **-t SEP**: usa un separatore di campo diverso da quello di default (una *non-blank to blank transition*)
- **-u**: ordina e rimuove linee duplicate

Esempio: il file **file.txt** contiene:

```
luca 1985 milano
martino 1990 torino
giovanni 1971 trieste
```

sort < file.txt > sorted.txt ordina le righe e stampa nel file **sorted.txt**, che conterrà:

giovanni 1971 trieste

luca 1985 milano

martino 1990 torino

cat file.txt | sort -k 2 -n ordina le righe *per anno* (secondo campo) e stampa su **stdout**:

giovanni 1971 trieste

luca 1985 milano

martino 1990 torino

Comando **uniq** (duplicati)

uniq [-cdw]: esamina i dati linea per linea *cercando linee duplicate* e può:

- Di default *elimina duplicati*
- **-c** per ogni riga prepende il *numero di occorrenze* (*dice quanti ce n'erano*)
- **-d** stampa solo le linee duplicate (*solo duplicate*)
- **-u** stampa solo le linee uniche (*solo uniche*)

NOTA. Il comando **uniq** non ordina le righe. E' necessario fornirle già ordinate!

Comando **wc** (conteggio parole)

wc [-lwc] [file]: conta linee (**l**), parole(**w**) e caratteri(**c**) dello **stdin** o del file

Altri comandi utili (non per esame):

- **sed**: ricerca e sostituzione di *espressioni regolari*
- **awk**: esecuzione di script (stile C) sulle righe di un file
- **comm**: trova le linee in comune (uguali) tra due file
- **paste**: concatena le linee di più file
- **rev**: scrive l'input in ordine inverso di caratteri, linea per linea

Esercizi

Esercizi

Dato il file **vini.txt** contentente il nome, l'anno, la città e il prezzo di alcune bottiglie di vino:

```
ribolla 2012 udine 21
prosecco 2018 trieste 15
barbera 2009 torino 20
freisa 2010 torino 18
barbera 2013 torino 14
barolo 1984 alba 45
```

Si trovino il nome e l'anno del vino più caro:

SHELL

```
$ sort -k4 -r < vini.txt | head -1 | cut -d " " -f 1-2
barolo 1984
```

Si trovino i nomi dei vini prodotti a Torino:

SHELL

```
$ cat vini.txt | grep torino | cut -d " " -f 1 | sort | uniq
barbera
freisa
```

Esercizi

Utilizzando lo stesso file dell'esercizio precedente:

1. Si calcoli quanti vini sono presenti per ogni città:

SHELL

```
$ cat vini.txt | cut -d " " -f 3 | sort | uniq -c
1 alba
3 torino
1 trieste
1 udine
```

2. Si calcoli quanti anni passano tra il vino più vecchio e più nuovo:

SHELL

```
$ min=$(sort -k2 < vini.txt | cut -d " " -f 2 | head -n 1)  
$ max=$(sort -k2 < vini.txt | cut -d " " -f 2 | tail -n 1)  
$ echo "Intercorrono $((max-min)) anni"  
Intercorrono 34 anni
```

Esercizi

Dato il file **file.txt** contentente:

```
luca 1985 milano  
martino 1990 torino  
giovanni 1971 trieste  
andrea 1984 milano
```

Si calcoli il numero di righe nel file:

SHELL

```
wc -l < file.txt # Output 4
```

Si calcoli quante città sono incluse nel file:

SHELL

```
cat file.txt | cut -d " " -f 3 | sort | uniq | wc -l # Output 3
```

Si trovi la città che appare il maggior numero di volte e il numero di occorrenze:

SHELL

```
cat file.txt | cut -d " " -f 3 | sort | uniq -c | sort | tail -n 1 # Output 2 milano
```

Ricorda: il comando **tail -n N** stampa le ultime **N** righe di un file o dello **stdin**

Esercizi

Si crei un programma che cerca ricorsivamente tutti i file presenti in una cartella passata come primo argomento.

Collochi quei file in una cartella ricevuta come secondo argomento, suddividendoli in sottocartelle separate per estensione.

Nota: si assuma che i nomi di cartelle non contengano `.` e i file ne contengano uno solo, nella forma **nome.estensione**

SHELL

```
#!/bin/bash

if (( $# != 2 ))
then
    echo "Usage: $0 indir outdir"
    echo "    The program assumes directories do not contain '!'"
    echo "    and files contain one"
    exit 1 # Signal error to the caller
fi

for f in $( find $1 -type f )
do
    folder=${2%/$( echo $f | cut -d . -f 2 )}
    mkdir -p $folder
    cp $f $folder
done
```

u3-s1-intro-c

Sistemi Operativi

Unità 3: Programmazione in C

Introduzione al linguaggio C

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Storia del C
 2. Compilazione in C
-

Caratteristiche Generali del C

Introduzione al C

Definizione del C.

Il **C** è un linguaggio di programmazione:

- *Ad alto livello*: non si scrive in istruzioni macchina
- *Imperativo*: il programma è una sequenza di istruzioni
- *Procedurale*: le istruzioni che svolgono una compito vengono raggruppate in *funzioni*, per permettere pulizia del codice e riuso

Livello C.

Tra i linguaggi di programmazione ad alto livello, il C è quello *più vicino al linguaggio macchina*.

- Libertà di utilizzo degli *indirizzi di memoria*, mediante i *puntatori*
- Utilizzato dentro *Linux* per scrivere il *kernel* e i *driver*

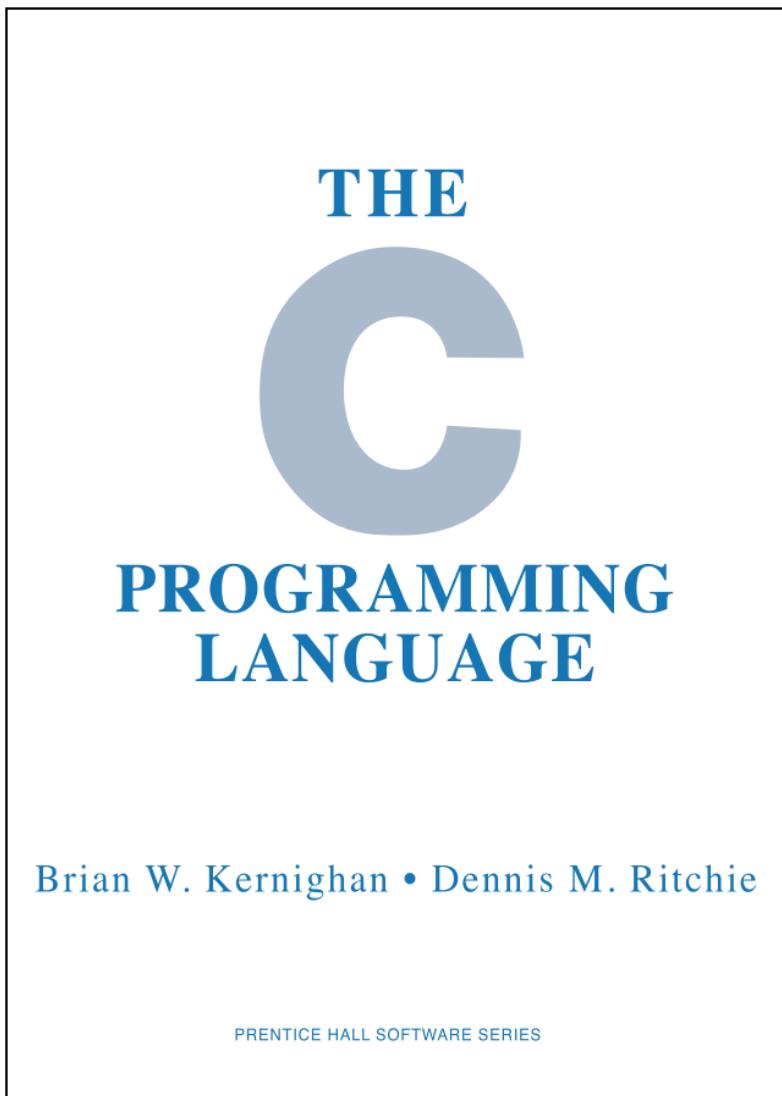
Caratteristiche del C:

- **Linguaggio minimalista**: pochi concetti semplici, vicini a quelli del linguaggio macchina
 - Molte istruzioni mappabili direttamente con una istruzione Assebly
 - Solo 32 parole riservate
 - **Ruolo centrale dei puntatori**: i puntatori sono variabili che contengono un indirizzo di memoria.
 - Permette perciò l'*indirizzamento indiretto*. Accedo a una variabile non tramite il suo nome, ma tramite il suo indirizzo.
 - Il programmatore ha un controllo molto elevato sulla memoria della macchina, consentendo di ottimizzare il codice
 - Tuttavia, questo potrebbe rappresentare una **vulnerabilità** nella sicurezza; infatti, le autorità statunitensi consigliano fortemente di **non** utilizzare il linguaggio C ([report ufficiale dettagliato](#))
 - **Tipizzazione statica**: ogni variabile ha un tipo di dato che deve essere esplicitamente dichiarato dal programmatore
-

Storia del C

Storia del C.

- Creato da Dennis Ritchie nel 1972 presso gli AT&T labs, col fine di scrivere il sistema operativo Unix
- Pubblicato nel 1978 col famoso libro *Il linguaggio C*
- Standardizzato a partire dal 1989. Standard ANSI X3.159-1989



THE C PROGRAMMING LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

PRENTICE HALL SOFTWARE SERIES

C oggi.

- Il C è in *continua evoluzione*. Si sono susseguiti vari standard negli anni.
 - Dalla prima versione **C89** (*anni '90*) ora siamo alla versione **C17**.
 - Nei prossimi anni ci sarà una nuova versione, per ora chiamata **C2x** (verosimilmente sarà **C23**)
- La standardizzazione garantisce la *portabilità* del *codice sorgente*. Uno stesso programma in C può essere compilato su *diversi SO* (Linux, Windows, MacOS).

Utilizzo del C

Applicazioni del C.

Attualmente il C è utilizzato per:

- Scrivere componenti di base di *Linux*, in particolare per i *kernel* e i *driver*
- Scrivere programmi che necessitano di *grande efficienza*
- Scrivere programmi in *domini critici*: telecomunicazioni, processi industriali, software real-time (ovvero dove c'è un *interazione a basso livello*)

Confronto del C con altri linguaggi

C vs Java:

C	Java
Compilato in codice macchina	Compilato in bytecode
Eseguito direttamente	Eseguito nella JVM
Gestione manuale di memoria	JVM gestisce la memoria
Generalmente Veloce	Lento

Compilazione in C

Linguaggi compilati e interpretati.

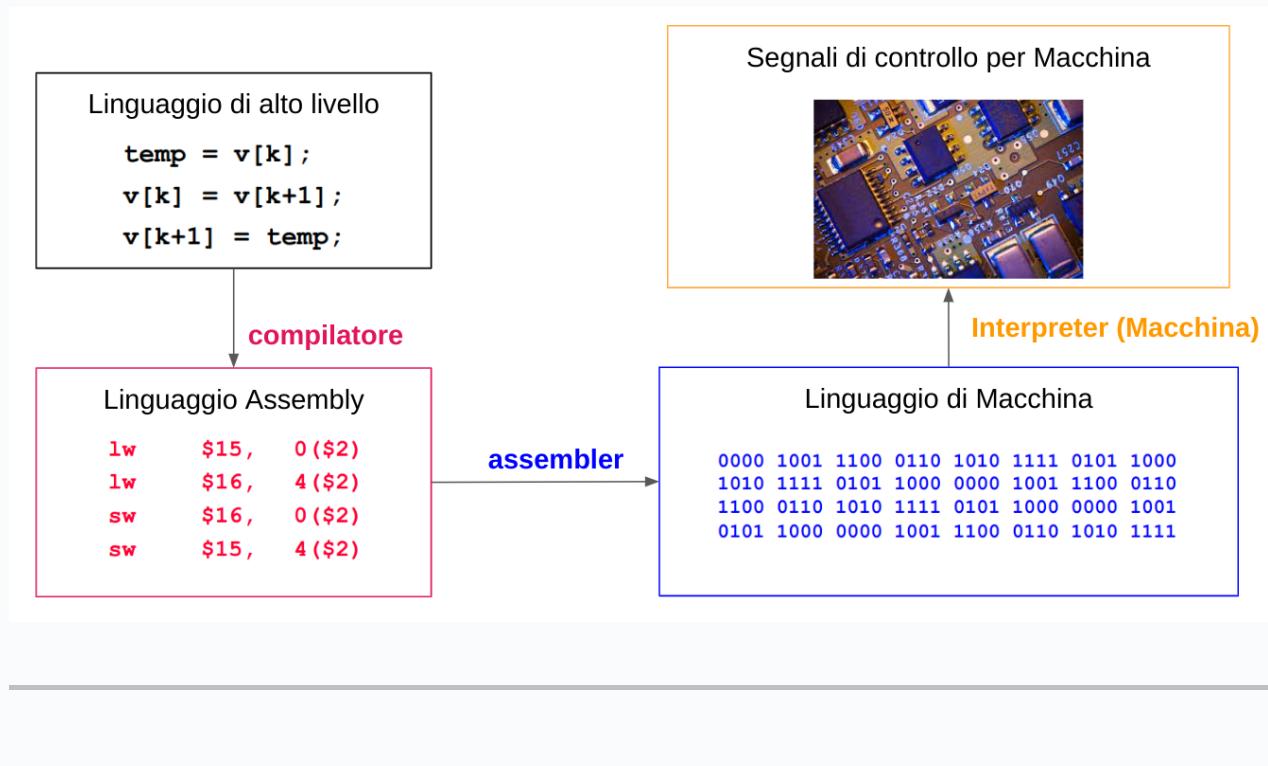
Il C è un *linguaggio compilato*.

- Un software chiamato *compilatore* traduce il codice sorgente in un eseguibile in *linguaggio macchina*
- Altri linguaggi compilati: C++, Go, Rust, ...

Il C non è un *linguaggio interpretato*

- Un linguaggio interpretato viene eseguito da un *interprete*, che legge ed esegue le istruzioni.
- Esempi di linguaggi interpretati: Python, R.

Figura: schema generale della compilazione.



Fasi della compilazione

Fasi della compilazione:

1. Il **Preprocessore** esegue eventuali sostituzioni testuali nel codice sorgente.
Necessario per costanti e macro. Osservare che qui non *entra in nessun modo* la **CPU**.
2. Il **Compilatore** crea il codice eseguibile per ogni file sorgente in C.
3. Il **Linker** assembla i codici eseguibili nel programma finale, collegando il programma alle funzioni di libreria.
 - Ogni linguaggio C fornisce varie funzioni di libreria per calcoli matematici, interazione col SO, realizzazione di interfacce grafiche.

Comandi per la Compilazione in C

Compilazione in Linux: si usa il compilatore standard **gcc**

Sintassi:

SHELL

```
gcc [<opzioni>] file1.c file2.c file3.c ... [-l librerie]
```

(Nota: "gcc" sta per "GNU C Compiler")

Normalmente vengono utilizzati come:

SHELL

```
gcc file.c      # compila e linka mettendo il codice eseguibile in a.out  
gcc file.c -c  # compila e non linka mettendo il codice oggetto in file.o  
gcc file.c -o outfile # compila e linka. codice exe in outfile  
gcc file.c -o outputfile -l libreria # compila e linka con libreria
```

Editor grafici: esistono molteplici IDE per il C. Uno semplice, snello e ben adatto a Ubuntu: **CodeBlocks**

Esempio di Compilazione in C

Primo programma in C: il seguente programma stampa a schermo la scritta **Hello World!**

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

Per compilare ed eseguire, inserire il codice sorgente nel file **hello.c** ed eseguire i seguenti comandi:

SHELL

```
$ gcc hello.c -o hello
$ ./hello
Hello World!
```

Descrizione delle istruzioni dell'esempio.

- **#include <stdio.h>** Indica che usiamo la libreria standard di I/O, nella quale sono definite le principali funzioni per la gestione dell'input/output
- **int main() {** Definisce la funzione **main**, che costituisce il corpo principale di ogni programma. Deve esserci in ogni programma. Deve restituire un intero.
- **printf("Hello World!\n");** La funzione di libreria **printf** stampa a video
- **return 0;** Istruzione di ritorno dalla funzione **main**. Termina il programma. Il *valore di ritorno* del programma verso il chiamante è 0 (*no errore*)
- Le parentesi graffe **{ ... }** delimitano i *blocchi funzionali*, come ad esempio

```
int main()
{
    ... istruzioni...
}
```

- **Osservazione:** lo stesso approccio è usato per delimitare blocchi funzionali in tutti i costrutti

```
C  
if (condizione)  
{  
    ... istruzioni...  
}
```

Struttura di Programmazione in C

Struttura minima di un programma

```
C  
#include librerie  
int main(void) {  
    definizione variabili  
    istruzioni eseguibili  
}
```

Commenti:

I commenti sono testo che non viene analizzato dal compilatore
Servono per aumentare la leggibilità del codice.

Sintassi:

```
C  
/* commento  
multiriga */
```

```
C  
// commento su singola riga
```

Spaziatura: gli spazi e i ritorni a capo non hanno funzione in C

Le istruzioni che non iniziano un blocco sono terminate da ;. Si dice che la spaziatura è "zucchero sintattico".

```
C  
int main(){  
    printf("hello\\n");  
    return 0;  
}
```

equivale a

```
C  
int main(){ printf("hello\\n"); return 0;}
```

Utilizzo di librerie:

Le librerie si possono usare dopo averle menzionate con la direttiva:

```
C  
#include <libreria.h>
```

Soltamente per le *librerie di sistema* usiamo le *parentesi angolari* <>, altrimenti usiamo le apici ''.

Nota: Le istruzioni di include non vanno terminate con ; e non si possono inserire spazi a inizio riga

Librerie C

Librerie principali: ci sono in *tutti* i sistemi operativi e sono le seguenti.

- <stdio.h>: Funzioni di lettura/scrittura su terminale e su file
- <stdlib.h>: Funzioni base per gestione di memoria, processi, conversione tra tipi di dato
- <math.h>: Funzioni matematiche
- <string.h>: Funzioni di manipolazione delle stringhe
- <ctype.h>: Manipolazione di caratteri

Altre librerie:

- <complex.h>: Manipolazione di numeri complessi
- <errno.h>: Gestione dei codici di errore di funzioni di libreria
- <time.h>: Per ottenere e manipolare date e orari
- <limits.h> e <float.h>: Costanti utili per lavorare su interi e numeri reali

Libreria solo per sistemi POSIX ([Linux](#), [UNIX](#), [Mac OS](#)), quindi non standard di C

- **<unistd.h>**: API standard di POSIX. Contiene le [System Call](#), qualora volessi usarle in una maniera diretta (1)

Librerie e System Call:

- Queste librerie (*ad esclusione di <unistd.h>*) sono raccolte nella [C standard library \(clib\)](#)
- Le funzioni di libreria [NON](#) sono delle System Call
 - Utilizzano al loro interno le System Call
- La **libc** è implementata su diversi SO
 - Utilizzando System Call diverse
- Permette di compilare lo stesso codice su SO diversi

Esempio: Aprire file in C

Per aprire un file si usa la funzione della **libc** chiamata **fopen**; a seconda del [sistema operativo](#) abbiamo una logica diversa per l'apertura del file. Infatti, si dice che questa funzione di libreria rappresenta un "*layer di compatibilità*".

- Su [Linux](#) utilizza la System Call **open**
- Su [Windows](#) utilizza la System Call **CreateFileA**

u3-s2-variabili-console

Sistemi Operativi

Unità 3: Programmazione in C

Variabili e utilizzo della console

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Variabili
2. Il tipo **int**
3. Il tipo **float**
4. Gli altri tipi
5. La funzione **printf**
6. La funzione **scanf**
7. Operazioni di base

Definizione di Variabile in C

Definizione di linguaggio tipizzato.

Ricordiamo che il C è un linguaggio *tipizzato* ([link da aggiungere!!!](#))

- Ogni variabile o costante ha un *tipo*
- Il tipo è *specificato esplicitamente* dal programmatore

Tipi di Variabili.

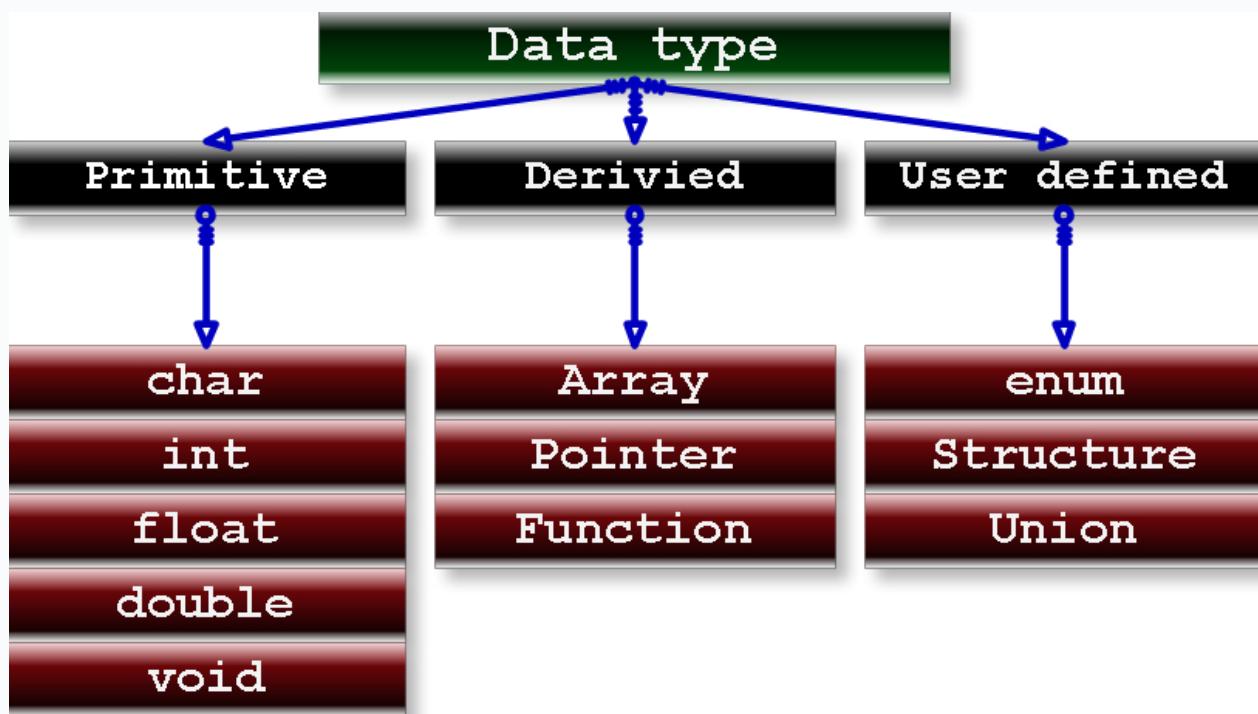
I tipi *principal*i sono:

- *Semplici*: `int`, `float`, `char`
- *Derivati*: insiemi di tipi semplici
 - Vettori (insiemi *omogenei*)
 - Struct (insiemi *non-omogenei*)
- Puntatori: contengono *indirizzi di memoria* a variabili di un certo tipo (come interi)

Nota. In C, i tipi di dato non hanno un'ampiezza standard, ma *varia da sistema a sistema*

- Ad esempio, un `int` può essere di 16, 32 o 64 bit; il linguaggio usa la *dimensione naturale* del *processore*
- Permette a ogni calcolatore di operare secondo la sua *dimensione naturale*
- Necessario fare *attenzione* in fase di scrittura del codice

FIGURA: Schema dei tipi di variabili



I Tipi Semplici

Il tipo **int**

- Rappresenta un *numero intero*.
- Rappresentato in *complemento a due su 16, 32 o 64 bit* (1).

SINTASSI

Dichiarazione:

```
int a;
```

Assegnazione:

```
a = 19;
```

Dichiarazione e Assegnazione:

```
int a = 19;
```

Il tipo **float** o **double**

- Rappresenta un *numero con la virgola*.
- Rappresentato con numero a *virgola mobile*, solitamente su 32bit (1).
- Esiste il tipo **double** che ha *precisione doppia*, solitamente su 64bit.

```
float a;
```

Assegnazione:

```
a = 3.14;
```

Altri Tipi di Base

I *tipi di base* in C sono:

- **int**: sono i numeri interi.
 - **float**: sono i numeri a virgola mobile
 - **double**: sono i numeri a virgola mobile a *precisione doppia*
 - **char**: sono le variabili che contengono *un carattere*.
 - **void**: *nessun tipo*, usato in situazioni particolari
-

Modificatori su Tipi

Possono essere usati dei *modificatori* sui tipi.

Esempio: **long int** indica un intero su più bit (ad es. 64 anziché 32).

CATEGORIA 1: LUNGHEZZA VARIABILI

- **long**: forza l'uso di un numero maggiore di bit
- **short**: forza l'uso di un numero minore di bit
 - **short int a;** indica un intero su 16 bit se di default è 32 bit.

CATEGORIA 2: SEGNO VARIABILI

- **signed**: indica che il tipo ha segno. Applicato di default
- **unsigned**: indica variabile che assume solo valori positivi

CATEGORIA 3: COSTANTI

- **const**: dichiara una costante.
 - **const float pi = 3.14**

Tipi **int** Speciali

Se si vuole avere il controllo sul *numero di bit di una variabile*, si possono usare i tipi:

- **int8_t**
- **int16_t**
- **int32_t**
- **int64_t**
- **uint8_t**
- **uint16_t**
- **uint32_t**
- **uint64_t**

A seconda dell'*architettura*, sarò *sicuro* sul numero dei bit. Questo è particolarmente

utile per scrivere *programmi molto efficienti*, o per *casi specifici* (come i *bitmask*)

Nota. Per usare questi tipi è necessario includere la libreria **#include**

<stdint.h>

Tipi Alias

Tipi di dato di sistema.

La libreria standard del C definisce dei tipi di dato *alias* (ovvero dei "soprannomi"), definiti nella Man Page **system_data_types**

- Aiutano la portabilità del codice.
- L'*alias* indica l'*obiettivo del tipo*, mentre su architetture diverse è *implementato con tipi diversi*; sono sostanzialmente dei *tipi di dati* con *scopi precisi*.

Esempi

- **size_t**: indica una lunghezza. E' solitamente **unsigned int**
- **off_t**: indica una offset di memoria. E' solitamente **int**

Ne esistono tanti: **pid_t uid_t gid_t time_t**

Operatori sui Tipi Dati

Operatore **sizeof**

L'operatore **sizeof** fornisce la *dimensione in Byte di un tipo di dato*.

- Ritorna un **size_t**

Importante! Perché la dimensione di un tipo dipende dalla macchina. Sarà utile quando useremo il comando **malloc** per allocare *dati* sullo *heap*.

Esempio: su PC 64bit

```
printf("%lu\n", sizeof(char)); // Stampa 1
printf("%lu\n", sizeof(int)); // Stampa 4
printf("%lu\n", sizeof(float)); // Stampa 4
printf("%lu", sizeof(double)); // Stampa 8
```

La funzione `printf`

Serve per stampare su *Standard Output* (1) (in genere *console*) del testo *arbitrario*.

- Per *interagire con utente*
- Per stampare il *risultato dell'elaborazione*
- Per stampare *informazioni* che sono *processate da altri programmi* tramite *pipe*

Contenuta nella libreria `stdio`.

Necessaria la direttiva:

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

Formato:

```
printf("formato", args...);
```

Il *formato* definisce il *testo da stampare*:

- *Tutti* i caratteri possono essere stampati
- Con `\n` si inserisce un ritorno a capo
- Per stampare il carattere `"` è necessario usare la sequenza di escape `\\"`
- Per stampare *valori numerici*:
 - Inserire le sequenze `%d` (per `int`) e `%f` (per `float`) nella *posizione desiderata*
 - Specificare negli `args` le variabili desiderate

Esempi:

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

```
Hello World
```

C

```
int a = 14;
printf("Intero: %d\n", a);
```

Intero: 14

C

```
printf("Il numero %f ", 3.14);
printf("e' pi greco\n");
```

Il numero 3.14 e' pi greco

C

```
int a = 12;
float b = 1.1;
printf("a=%d\nb=%f\n", a, b);
```

a=12

b=1.1

La funzione **scanf**

Definizione.

La funzione **scanf** permette di *leggere* dallo **stdin** (1), tipicamente per *richiedere un input* all'utente da terminale.

Si possono leggere un **int** un **float** (o altri tipi)

Formato:

C

```
scanf("tipo", &variabile) ;
```

Tipo:

- Per leggere un **int**: **%d**.
- Per leggere un **float**: **%f**

Variabile:

Inserire una variabile di tipo **int** o **float** già dichiarate

- Preceduta dal simbolo **&**
 - Vedremo che il motivo è che la funzione **scanf** richiede un *puntatore*
 - Con **&variabile** si passa alla **scanf** l'indirizzo di **variabile**

Esempi:

Lettura di un **int**

```
C
int a;
scanf("%d", &a);
```

Lettura di un **float**

```
C
float b;
scanf("%f", &b);
```

Esempio Misto:

```
C
int a;
printf("Inserisci un numero: ");
scanf("%d", &a);
printf("Il quadrato del numero immesso è: %d\n", a*a);
```

Assegnazione

Assegnazione: si utilizza l'operatore **=**.

Esempi:

C

```
int a;
a = 12; // Assegnazione da costante
int b;
b = a; // Assegnazione da variabile
```

C

```
float f = 12; // Assegnazione assieme a dichiarazione
f = f + 12; // Assegnazione che incrementa
```

Operazioni Aritmetiche sui Numeri

Operazioni aritmetiche

- Somma: **a + b**
- Sottrazione: **a - b**
- Somma: **a * b**
- Divisione: **a / b**
 - Nota: se entrambi gli operandi sono **int** lo è anche il risultato.
- Resto della divisione: **a % b**
- Incremento: **i++**
- Decremento: **i--**

Casting

Conversione tra tipi: si chiama operazione di **casting**.

Sintassi. Il formato è: **(tipo) variabile**. Ad esempio: **(float) a**

Esempio:

C

```
int a = 5;
int b = 2;
float c;
c = a/b; // contiene 2
c = ( (float) a ) / ( (float) b ); // contiene 2.5
```

Parentesi: si possono utilizzare per annidare operazioni nella maniera desiderata.

Osservazione. Posso usare il *"casting implicito"*, ovvero la *conversione* dei tipi dati in mediante operazioni aritmetiche. Ad esempio **2+0.0** risulta un **float**.

Operazioni sui Bit

Operatori sui bit: eseguono operazioni logiche bit a bit

- **a & b**: *AND* bit a bit
- **a | b**: *OR* bit a bit
- **a ^ b**: *XOR* bit a bit
- **~a**: *NOT* bit a bit (operatore unario)

NOTA! Da non confondere con operatori logici (**&&**, **||**, **!**, che vedremo più avanti)

FIGURA: Schema degli operatori sui bit

X	Y	X&Y	X Y	X^Y	~(X)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Esercizi

Si scriva un programma che legge due interi da tastiera e stampa la loro somma.

```
#include <stdio.h>

int main(void)
{
    int a, b /* addendi */
    int c /* somma */

    /* LEGGI GLI ADDENDI A E B */
    printf("Somma due numeri\n\n");

    printf("Immetti il primo numero: ");
    scanf("%d", &a);

    printf("Immetti il secondo numero: ");
    scanf("%d", &b);

    /* CALCOLA LA SOMMA */
    c = a + b;

    /* STAMPA IL RISULTATO C */
    printf("La somma di %d + %d vale: %d\n", a, b, c);

    return 0; /* Valore di ritorno, significante no errore*/
}
```

Si scriva un programma che dato un numero di minuti, calcola a quante ore (e minuti rimanenti) equivale.

```
#include <stdio.h>

int main(void)
{
    int a; /* minuti input*/
    int b, c ; /* ore e minuti in output */
    /* LEGGI I MINUTI */
    printf("Calcolo delle ore\n\n");

    printf("Immetti il numero di minuti: ");
    scanf("%d", &a);

    /* CALCOLA LA SOMMA */
    b = a/60;
    c = a%60;

    /* STAMPA IL RISULTATO C */
    printf("Una quantità di %d minuti equivale a %d ore e %d minuti\n", a, b, c);

    return 0; /* Valore di ritorno, significante no errore*/
}
```

u3-s3-controllo-flusso

Sistemi Operativi

Unità 3: Programmazione in C

Controllo del flusso e cicli

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Controllo del flusso
 2. Cicli
-

Controllo del flusso

Definizione di Controllo del flusso

DEFINIZIONE. (*controllo di flusso*)

Un "controllo di flusso" è la possibilità di eseguire *set alternativi di istruzioni* a seconda del verificarsi di una *condizione* (ovvero una *espressione booleana* che assume solamente due valori, tradizionalmente scritti come 0 e 1).

- Alla *base* di quasi ogni programma.
 - E' necessario definire una condizione, ovvero una *espressione booleana* che può essere vera (**true**) o falsa (**false**)
 - Vedremo:
 - Operatori *di confronto*: `=`, `≠`, `<`, `>`, `<`, `>`, `<`
 - Operatori *di booleani* (logici): `&&`, `||`, `!`
-

Istruzione **if-else**

SINTASSI.

```
if ( condizione )
{
    A; // Ramo Vero
}
else
{
    B; // Ramo Falso
}
```

Concettualmente identico ai costrutti **if** in Python o Java.

Possibile avere più possibili rami.

```

if ( condizione1 )
{
    A; // Eseguito se condizione1
}
else if (condizione2)
{
    B; // Eseguito se condizione2
}
else
{
    C; // Eseguito altrimenti
}

```

Esercizio: si scriva un programma che legge da tastiera un intero e scrive se esso è positivo o negativo

```

#include <stdio.h>

int main(void)
{
    int a;
    printf("inserisci un numero: ");
    scanf("%d", &a);

    if (a>0){
        printf("%d e' positivo\n", a);
    }
    else
    {
        printf("%d e' negativo\n", a);
    }
}

```

Osservazione sull'Istruzione **if-else**

Osservazioni:

- E' possibile *omettere* il ramo **else**:

```
if ( condizione )
{
    A; // Ramo Vero
}
```

- Se un ramo è composto da una *sola istruzione*, è possibile *omettere* le {}

```
if ( condizione )
    A;
else
{
    B;
}
```

oppure

```
if ( condizione )
    A;
```

- ...
 - Questo vale per *tutti i costrutti* in C: cicli **for** e **while**, ...
 - In ogni caso è una buona idea *non* ometterli, per sicurezza

Esempio:

```
int a;
printf("inserisci un numero: ");
scanf("%d", &a);

if (a>0)
    printf("%d e' positivo\n", a);
else
    printf("%d e' negativo\n", a);
```

Errore comune: omettere le {} quando il blocco ha più di una istruzione!

Operatori di confronto

Sono equivalenti a quelli di *Java* o *Python*

- Uguaglianza: `a == b`
- Differenza: `a != b`
- Maggiore: `a > b`
- Maggiore o uguale: `a >= b`
- Minore: `a < b`
- Minore o uguale: `a <= b`

Nota. Questi valgono *solo* per i *tipi di dati esistenti*, quindi *non* stringhe!

Errore comune: confondere operatore di assegnazione `=` con quello di uguaglianza `==`

Operatori booleani

Gli operatori di confronto permettono di definire *condizioni semplici*.

Spesso è necessario *combinare condizioni semplici*.

Esempio: un valore è compreso in un intervallo?

`a>10 e a<20`

Per combinare condizioni semplici si usano gli *operatori booleani*.

- AND: `(cond1) && (cond2)`
- OR: `(cond1) || (cond2)`
- NOT: `!(cond1)`

Albero Sintattico degli Operatori Booleani.

Precedenza degli operatori nelle condizioni. *Ordine di priorità*:

- Operatori di confronto
- Operatore `!`
- Operatore `&&`
- Operatore `||`

Esempi:

```
if ( (a>10) && (b>10) )
/* Equivale a */
if ( a>10 && b>10 )
```

```
if ( !(a>10) || (b>10) )
/* Equivale a */
if ( !a>10 || b>10 )
```

Errore comune: Confondere operatore di AND booleano **&&** con l'operatore di **bitwise** AND **&**.

Annidamento di istruzioni **if else**

E' possibile annidare istruzioni **if else** per creare ramificazioni complesse.

```
if ( condizione1 )
    if ( condizione2 ){
        A;
    }
    else {
        B;
    }
else{
    C;
}
```

Esercizio

Si scriva un programma che risolve un'equazione di primo grado.

```

#include <stdio.h>
int main()
{
    float a, b ; /* coefficienti a e b */
    float x ; /* valore di x che risolve l'equazione */
    printf("Risoluzione di un'equazione di primo grado\n");
    printf("Equazione nella forma: ax + b = 0\n");
    /* LEGGI a e b */
    printf("Immetti coefficiente a: ");
    scanf("%f", &a);
    printf("Immetti coefficiente b: ");
    scanf("%f", &b);

    /* x VIENE CALCOLATO COME x=-b/a. SI DEVONO VERIFICARE I VALORI DI
    a E b */
    if( a != 0 ) {
        x = - b / a;
        printf("La soluzione e' x = %f\n", x);
    } else { /* CASO a==0 */
        if( b==0 ) {
            printf("Equazione indeterminata (ammette infinite soluzioni)\n");
        } else {
            printf("Equazione impossibile (non ammette soluzioni)\n");
        }
    }
}

```

Istruzione **switch**

Semplifica il codice in caso di *scelta in base al valore di una espressione*.

Sintassi:

```

switch (espressione)
{
    case v1:
        A;
        break;
    case v2:
        B;
        break;
    default:
        C;
}

```

Osservazioni:

- **espressione** può essere una **variabile** o un'**espressione**
- **v1**, **v2** devono essere una **costante**. Non possono essere una variabile.
- Necessario sempre delimitare ogni caso con **case** e **break**
 - *Errore comune*: dimenticare il **break**
- **default** si comporta come **else** nel costrutto **if**. E' opzionale.
- La sintassi è *molto arcarica*: questa istruzione è un relitto delle versioni vecchie del C
 - Infatti non utilizza **{}** ma **case** e **break** per *delimitare blocchi*
 - Relitto di istruzioni con **goto**.
- *Non* utilizzare se non serve strettamente.
 - Usata tipicamente per migliore performance rispetto a **if** quando ho tanti casi.

Operatore ternario

Permette di scrivere in *maniera concisa* un'espressione **if then else**.

Sintassi:

```
var = condizione ? espressione1 : espressione2;
```

Equivale a:

```
if(condizione)
    var = espressione1;
else
    var = espressione2;
```

Limitazione. **espressione1** e **espressione2** possono essere solo **espressioni**, non istruzioni! L'operatore ternario può essere usato per fornire un'**espressione** in una istruzione

Esempi:

Assegnazione di valore minimo

```
c = (a < b) ? a : b; // Assegna a 'c' il minore tra 'a' e 'b'
```

Invocazione di una funzione (che è un'espressione):

```
a > b ? printf("%d\n", a) : printf("%d\n", b);
```

Cicli

Definizione di Ciclo

Definizione. (*ciclo*)

Hanno lo scopo di *ripetere in maniera controllata un blocco di istruzioni*.

Permettono di svolgere compiti ripetitivi senza duplicare il codice.

Solitamente organizzati in:

- Un blocco di istruzioni da eseguire ripetutamente.
- Una condizione che regola la fine del ciclo.

In C esistono *due tipi* di ciclo: **while** (e variante **do-while**) e **for**.

Ciclo **while**

Esegue finché *una condizione è vera*.

C

```
while ( C )
{
    A;
}
```

Comportamento:

1. Viene valutata C.
2. Se C è falsa, **salta** il blocco di istruzioni
3. Se C è vera, **esegue** il blocco di istruzioni A e **torna** al punto 1

Esempio:

C

```
int i = 1;
while ( i < 10 )
{
    printf("Numero = %d\n", i) ;
    i = i++; // Operatore di incremento
}
```

Risultato: stampa i numeri da 1 a 10 compresi.

Errore comune: sbagliare la condizione di terminazione e generare un ciclo infinito (anche se in realtà si potrebbe creare cicli infiniti di proposito).

Esercizio.

Si scriva un programma che calcoli la media di un numero di **float** specificato dall'utente.

```
#include <stdio.h>
int main()
{
    int i=0, n;
    float dato;
    float somma = 0.0;

    printf("Introduci n: "/* Leggi n */
    scanf("%d", &n) ;
    if( n>0 )
    {
        while(i < n) /* Esegui n volte */
        {
            printf("Valore %d: ", i+1);
            scanf("%f", &dato) ;
            somma = somma + dato ; /* Accumula la somma */
            i = i + 1; /* Contatore */
        }
        printf("Risultato: %f\n", somma/n) ;
    }
    else
        printf("Non ci sono dati da inserire\n");
}
```

Ciclo **for**

Definizione

Rende più facile eseguire un blocco N volte, determinato *a priori*.

- E' possibile anche col ciclo **while** ma è più prone a errori, siccome è necessario:
 1. inizializzare un *contatore*
 2. impostare la condizione del **while**
 3. *incrementare il contatore* a ogni operazione

Il ciclo **for** *sistematizza* queste operazioni

Sintassi

C

```
for ( I; C; A )
{
    B;
}
```

- **I** è l'**istruzione** di inizializzazione
- **C** è la **condizione** di terminazione
- **A** è l'**istruzione** di aggiornamento

Esempio:

C

```
int i;
for ( i=0; i<10; i=i++ )
{
    printf("Numero = %d\n", i);
}
```

equivale a:

C

```
int i;
i=0;
while ( i<10; )
{
    printf("Numero = %d\n", i);
    i++;
}
```

Osservazione.

E' semplice annidare cicli **for**, ad esempio per iterare su una tabella o matrice.

```

for( i=0; i<N; i++ )
{
    for( j=0; j<N; j++ )
    {
        printf("i=%d - j=%d\n", i, j);
    }
}

```

Esercizio: si crei un programma che determina su un numero inserito dall'utente è primo.

```

#include <stdio.h>
int main()
{
    int n, i;
    int primo=1;

    printf("inserisci il numero: ");
    scanf("%d", &n);

    for (i=2; i<n; i++)
        if(n%i==0)
            primo=0;

    if ( primo == 0)
        printf("Il numero %d non è primo\n", n);
    else
        printf("Il numero %d è primo\n", n);

}

```

Domanda: il numero 2 147 483 647 è primo? Quanto ci mette a calcolare?

Osservazioni

- Come nell'esempio precedente, se il blocco ha una sola istruzione, si possono omettere le {}
- Possibile annidare blocchi di una istruzione in più costrutti di flusso.
- Esempio:

```
for (i=2; i<n; i++)
    if(n%i==0)
        primo=0;
```

Istruzioni speciali sui Cicli

All'interno dei cicli **for** e **while** è possibile usare le seguenti istruzioni speciali.

- **break**: *termina il ciclo immediatamente*, passando alle istruzioni seguenti al ciclo.
- **continue**: *passa immediatamente alla iterazione successiva* senza eseguire le rimanenti istruzioni del blocco.

Nota. Da usare con parsimonia! Possono spesso, se inattenti, causare *cicli infiniti*.

Esercizio: quante volte viene invocata la **printf**?

```
for (i=0; i<10;i++){
    printf("Iterazione\n");
    if (i==3)
        break;
}
```

Esercizio: quante volte viene invocata la **printf**?

```
for (i=0; i<10;i++){
    if (i<3)
        continue;
    printf("Iterazione\n");
    if (i>5)
        break;
}
```

```
Dataview (inline field '='): Error:  
-- PARSING FAILED -----  
-----  
  
> 1 | =  
| ^  
  
Expected one of the following:  
  
'()', 'null', boolean, date, duration, file link, list ('[1,  
2, 3]'), negated field, number, object ('{ a: 1, b: 2 }'),  
string, variable
```

```
Dataview (inline field '='): Error:  
-- PARSING FAILED -----  
-----  
  
> 1 | =  
| ^  
  
Expected one of the following:  
  
'()', 'null', boolean, date, duration, file link, list ('[1,  
2, 3]'), negated field, number, object ('{ a: 1, b: 2 }'),  
string, variable
```

u3-s4-variabili-derivate

Sistemi Operativi

Unità 3: Programmazione in C

Tipi complessi

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Vettori
 2. Le **struct**
 3. Le **union**
-

Osservazione Preliminare

Osservazione.

I tipo di dato semplici (**int** o **float**) possono contenere un solo dato alla volta.

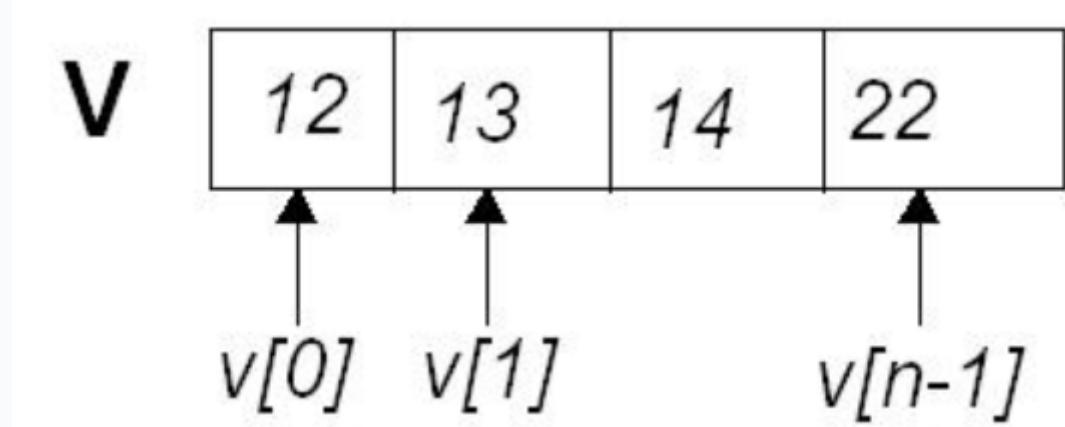
In C, si possono creare tipi di dato *complessi*, che contengono *più valori*. Noi vedremo:

- I vettori o **array**
 - Le strutture o **struct**
 - Le unioni o **union**
-

Vettori

Vettori: Definizione

Definizione: Un vettore o **array** è un insieme di variabili dello *stesso tipo*. E' composto di **N** celle, ognuna identificata da un *indice*.



Utilizzi: *vastissimi*. Permettono di trattare liste di oggetti *senza ripetere il codice*

- Effettuare operazioni matematiche: media, varianza
- Gestire flussi di dati
- Ripetere il codice è una pratica *sbagliata*; come ad esempio ho

```
int dato1, dato2, dato3, dato4, dato5 ;  
int dato6, dato7, dato8, dato9, dato10 ;  
  
scanf("%d", &dato1);  
scanf("%d", &dato2);  
scanf("%d", &dato3);  
...
```

- Questo è inutile e prone a errori!
- Una versione corretta sarebbe

```
int dato[10]; // Definizione di array  
  
for(i=0; i<10; i++)  
    scanf("%d", &dato[i]);  
  
for(i=9; i>0; i--)  
    printf("%d\n", dato[i]);
```

- Vedremo la sintassi esatta nelle prossime pagine.

Sintassi di Vettori

Definizione di un vettore:

```
tipo nome [N];
```

Esempio:

```
int vettore [10];
```

Definisce un array chiamato **vettore** composto da 10 interi (**int**).

- **Nota:** La lunghezza del vettore deve essere nota in *fase di compilazione*. Deve essere una *costante*!
- Il seguente codice è errato:

```
C
int N;
scanf("%d",&N);
float data[N];
```

- Questa è una grande *differenza* rispetto ad altri linguaggi di programmazione come Java o Python.
 - Esistono metodi per creare *array di lunghezza arbitraria* (i.c.d. "*array dinamici*") in C (la funzione **malloc**), che vedremo più avanti nel corso.

Costanti: esistono due modi in C per dichiarare delle costanti.

1. Tramite una direttiva **define**:

```
C
#define N 10
```

2. Tramite il modificatore **const** applicato a una variabile.

```
C
const int N = 10; // N non è modificabile
int dato[N];
```

Sintassi Alternativa: si può *definire ed inizializzare* allo stesso tempo un vettore.

```
C
int v[4] = {2, 7, 9, 10};
// equivalentemente
int v4[]; v[0]=2; v[1]=7; v[2]=9; v[3]=10;
```

In questo caso, si può *omettere la lunghezza*, che viene inserita *automaticamente* dal compilatore.

C

```
int v[] = {2, 7, 9, 10};
```

Operazioni sui Vettori

Indicizzazione

Accesso agli elementi: Si deve specificare l'*indice*. La sintassi è la seguente.

C

```
nomevettore[valoreindice]
```

Esempio:

C

```
int v [] = {4,5,6};
printf("%d\n", v[1]); // stampa 5
```

Indici:

- Partono da 0 e arrivano a $N - 1$
- Devono essere **int**
- Possono essere delle variabili o i risultati di una espressione

Importante! In C, *non viene controllato* che l'indice sia minore di $N - 1$. Se si accede con indici oltre i limiti, si va a leggere locazioni di *memoria arbitrarie*, che contengono *dati arbitrari*. Quindi attenti!

Esercizio: si leggano 5 interi e si stampino in ordine inverso.

```
#include <stdio.h>
#define N 5
int main (){
{
    int v[N];
    int i;

    for (i=0; i<N; i++){
        printf ("Inserisci l'elemento %d: ", i);
        scanf("%d", &v[i]);
    }
    for (i=N-1; i>0; i--)
        printf ("Elemento %d: %d\n", i, v[i]);

    return 0;
}
```

Copia di un Vettore

Copia di un vettore:

- Bisogna copiare il contenuto *elemento per elemento*
- Tra vettori della *stessa lunghezza e stesso tipo*.
- *Sbagliato* tentare di copiare usando una sola istruzione.

Corretto:

```
for (i=0; i<N; i++)
    v2[i] = v1[i];
```

Sbagliato:

```
v1 = v2;
v1[] = v2[];
```

Spiegazione.

- La variabile vettore è un *contenitore di elementi*
 - Di per se è *immutable*
 - Si possono solo *modificare* gli *elementi contenuti*
 - Approfondiremo quando vedremo i *puntatori* (infatti i vettori *sono* dei *puntatori* particolari)
-

Altri Utilizzi dei Vettori

- **Matrici:** un array di array è una *matrice*.

```
C  
int matrice [3][2];  
matrice[1][0]=12;  
float m[2][2]={{1,2},{3,4}};
```

Volendo, posso fare *array* di *array* di *array*, che è un *tensore*. Utile per le *reti neurali*.

- **Stringhe:** un array di **char** è una *stringa*.

Per definizione, in C le stringhe sono array di char, il cui ultimo elemento è 0 (o **'\0'**) , per convenzione; perché così se ne può *derivare la lunghezza*.

Molti usi e funzioni sulle stringhe. Vedremo più avanti.

```
C  
char s[4] = {'a', 'p', 'e', '\0'};
```

Esercizio sui Vettori

Esercizio: si un numero N da tastiera. Si leggano poi N interi e si stampi se essi includono duplicati.

```

#include <stdio.h>
#define MAXN 50 // Limite massimo del vettore
int main (){
{
    int v[MAXN]; // Vettore sovradimensionato
    int N, i, j;

    printf ("Si inserisca N: ");
    scanf("%d", &N); // Lunghezza effettiva del vettore

    if (N>MAXN){
        printf("N deve essere minore o uguale a %d\n", MAXN);
        return 1; // Ritorna un errore
    }

    for (i=0; i<N; i++){
        printf ("Inserisci l'elemento %d: ", i);
        scanf("%d", &v[i]);
    }

    for (i=0; i<N; i++)
        for (j=0; j<i; j++)
            if (v[i]==v[j])
                printf("L'elemento %d è duplicato dell'elemento %d\n", i,j);
    return 0;
}

```

Osservazione: si è scelto di sovradimensionare il vettore **v** rendendolo lungo **MAXN**, ma utilizzandolo fino all'elemento **N-1**.

Con la memoria dinamica che vedremo più avanti, questo work-around non sarà più necessario.

Le **struct**

Strutture: Definizione

Le **strutture** o **struct** sono collezioni che contengono variabili *non necessariamente dello stesso tipo* (ovvero possono essere di tipi diversi).

Funzionamento:

1. Si definisce la **struct**, un *nuovo tipo di dato* complesso formato da più campi
2. Si *creano* e si *usano* variabili del tipo appena creato.

Sintassi per le Strutture

1. Definizione di una **struct**:

```
struct nome {  
    campi  
};
```

Esempio:

```
struct punto{  
    float x;  
    float y;  
};  
  
struct lista  
{  
    int INFO;  
    lista* NEXT;  
}
```

2. Creazione di variabili **struct**:

Per creare nuove variabili di un tipo **struct** definito in precedenza.

```
struct nome variabile;
```

Esempio:

```
struct punto p1, p2;
```

3. Acesso ai campi **struct**:

```
variabile.campo
```

Esempio:

```
p1.x = 2.5;  
p1.y = 3.0;
```

Zucchero Sintattico per le **struct**

Utilizzo di `typedef`: per evitare di dover *premettere struct* ogniqualvolta si crea una variabile, si può usare la keyword **typedef**, con la seguente *sintassi*.

```
typedef struct {  
    campi  
} nome;
```

Esempio:

```
typedef struct{  
    float x;  
    float y;  
} punto;  
punto p1, p2; // Si può omettere struct
```

Esercizio sulle Struct

Esercizio: si crei un programma che effettua la somma vettoriale tra due vettori bidimensionali.

```
#include <stdio.h>

// La dichiarazione di una struct è solitamente fuori da ogni funzione
typedef struct{
    float x;
    float y;
} punto;

int main ()
{
    punto p1, p2, p3; // Tre variabili di tipo 'punto'

    // Lettura
    printf ("P1 → x: ");
    scanf("%f", &p1.x);
    printf ("P1 → y: ");
    scanf("%f", &p1.y);
    printf ("P2 → x: ");
    scanf("%f", &p2.x);
    printf ("P2 → y: ");
    scanf("%f", &p2.y);

    // Somma
    p3.x = p1.x + p2.x;
    p3.y = p1.y + p2.y;

    printf("La somma vettoriale è il punto: (%f, %f)\n", p3.x, p3.y);

}
```

Le **struct**: Osservazioni

Osservazioni

1. Non è possibile confrontare due **struct** con gli operatori **==** o **≠**. Infatti è necessario confrontare tutti i campi

```
C  
punto p1, p2;  
if (p1==p2) // Sbagliato! (Anche se in certi casi potrebbe funzionare, ma non è garantito)  
...  
if (p1.x==p2.x && p1.y == p2.y) // Corretto  
...
```

Inizializzazione: si può fare come coi vettori; ovvero *manualmente*

```
C  
punto p1 = {1.1, 2.4} // x=1.1 e y = 2.4
```

Usi delle Strutture

Le **struct** sono molto usate per creare *nuovi tipo di dato complesso*:

- Sono come record di un database
- *Esempi.* Numero complesso, indirizzo stradale, ecc...

Sono molto usate nelle *librerie del C*.

- Permettono di creare tipi di dato arbitrari
- Per rappresentare strutture del sistema operativo.
- *Esempi.* Variabili di sincronizzazione, pacchetti di rete, ecc...

Le **union**

Le Unioni: Definizione

Definizione.

Una **union** o *unione* è una variabile che può contenere *in momenti diversi* oggetti di tipo (e dimensione) *diversi*, con, *in comune*, il ruolo all'interno del programma.

- Le **union** servono per *risparmiare memoria*; si usa un'unica *cella di memoria*.
- Usate particolarmente in sistemi *embedded* con stretti vincoli di risorse
- Le **union** servono anche per avere un *tipo di dato generico* che ha tipo diverso a seconda della circostanza, quindi per *interpretare* zone di memorie *diversamente*.

Implementazione.

Occupava tanta memoria quanto il **campo più grande**, visto che esse non possono mai essere utilizzate contemporaneamente (la scelta di una esclude automaticamente le altre)

- I campi **condividono** il medesimo spazio di memoria.
- Se si cambia il valore a un campo, il valore di tutti gli altri campi viene sovrascritto

Sintassi delle Unioni

In C una unione viene definita tramite la **parola chiave union**

- La **definizione** di un'unione è molto **simile** a quella di una **struct**, ed è **medesimo** il **modo di accedervi**
- Si può usare **typedef** per evitare di dover premettere **union** ogniqualvolta si crea una variabile

Esempio.

```
C

union student
{
    uint64_t tessera_sanitaria;
    uint32_t matricola;
}; // Nota: Qui si prende automaticamente una cella di memoria da 64bit (8 byte)
```

- Si dichiarano e utilizzano come le **struct**

C

```
#include <stdio.h>
#include <stdint.h> /* Per uint32_t e uint64_t */
union student
{
    uint64_t tessera_sanitaria;
    uint32_t matricola;
};

int main( int argc, char *argv[] ){
    union student luca;
    luca.matricola = 1234;
    printf("Matricola: %d\n", luca.matricola);

    luca.tessera_sanitaria = 2897189786;
    /* Usare il formato %ld essendo in intero lungo */
    printf("Tessera Sanitaria: %ld\n", luca.tessera_sanitaria);
    return 0;
}
```

Nota. L'assegnazione di un campo **sovrascrive** il valore gli altri campi, che non potranno essere più letti.

C

```
union student luca;
luca.matricola = 1234;

// Sovrascrivo luca.matricola
luca.tessera_sanitaria = 2897189786;

// Leggo correttamente tessera_sanitaria
printf("Tessera Sanitaria: %d\n", luca.tessera_sanitaria);

// Leggo matricola, che è stata sovrascritta!
printf("Matricola di Luca: %d\n", luca.matricola); // Errore!
```

Casi d'uso delle Unioni

Si usano quando serve *dichiarare variabili* che possono assumere *tipo diverso a seconda delle circostanze*.

Esempio: Uno studente può essere identificato col *numero di tessera sanitaria* o con la *matricola* a seconda della situazione

```
C  
union student  
{  
    uint64_t tessera_sanitaria;  
    uint32_t matricola;  
};
```

La **union student** *non* può contenere *contemporaneamente* tessera sanitaria e matricola

- Il programma deve essere scritto di conseguenza

Rappresentazione delle Unioni nella Memoria

Come detto, una **union** occupa lo *spazio necessario al campo più grande*.

FIGURA: Schema

```
Dataview (inline field '='): Error:  
-- PARSING FAILED -----  
-----  
  
> 1 | =  
| ^  
  
Expected one of the following:  
  
'()', 'null', boolean, date, duration, file link, list ('[1,  
2, 3]'), negated field, number, object ('{ a: 1, b: 2 }'),  
string, variable
```

u3-s5-stringhe

Sistemi Operativi

Unità 3: Programmazione in C

Stringhe

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Il tipo **char**
 2. Stringhe
 3. Funzioni sulle stringhe
 4. Conversione tra stringhe e altri tipi
-

Il tipo **char**

Carattere e Stringa: Definizione

Recap.

Per ora abbiamo visto i tipi di dato:

- Intero: **int**
- Reale: **float**
- Vettori: **[]**
- Strutture: **struct**

Definizione di Carattere e Stringa.

Esiste il tipo **char** che rappresenta un *singolo carattere*.

Un *vettore di caratteri* è una *stringa*.

```
char s [10]; // Stringa di lunghezza 10
```

La convenzione ASCII

Il tipo **char** rappresenta un singolo carattere come un *numero*, codificato mediante la convenzione *ASCII*.

- Come numero tra **0 e 127**
- Ogni numero rappresenta un possibile carattere
- **Non** ci sono caratteri *speciali, accentati o simili*
 - Lo standard per trattarli si chiama *Unicode*, non lo vedremo

FIGURA: Tabella ASCII

| Dec Chr |
|---------|---------|---------|---------|---------|
| 0 NUL | 26 SUB | 52 4 | 78 N | 104 h |
| 1 SOH | 27 ESC | 53 5 | 79 O | 105 i |
| 2 STX | 28 FS | 54 6 | 80 P | 106 j |
| 3 ETX | 29 GS | 55 7 | 81 Q | 107 k |
| 4 EOT | 30 RS | 56 8 | 82 R | 108 l |
| 5 ENQ | 31 US | 57 9 | 83 S | 109 m |
| 6 ACK | 32 : | 58 : | 84 T | 110 n |
| 7 BEL | 33 ! | 59 ; | 85 U | 111 o |
| 8 BS | 34 " | 60 < | 86 V | 112 p |
| 9 HT | 35 # | 61 = | 87 W | 113 q |
| 10 LF | 36 \$ | 62 > | 88 X | 114 r |
| 11 VT | 37 % | 63 ? | 89 Y | 115 s |
| 12 FF | 38 & | 64 @ | 90 Z | 116 t |
| 13 CR | 39 ' | 65 A | 91 [| 117 u |
| 14 SO | 40 (| 66 B | 92 \ | 118 v |
| 15 SI | 41) | 67 C | 93] | 119 w |
| 16 DLE | 42 * | 68 D | 94 ^ | 120 x |
| 17 DC1 | 43 + | 69 E | 95 _ | 121 y |
| 18 DC2 | 44 , | 70 F | 96 ` | 122 z |
| 19 DC3 | 45 - | 71 G | 97 a | 123 { |
| 20 DC4 | 46 . | 72 H | 98 b | 124 |
| 21 NAK | 47 / | 73 I | 99 c | 125 } |
| 22 SYN | 48 0 | 74 J | 100 d | 126 ~ |
| 23 ETB | 49 1 | 75 K | 101 e | 127 DEL |
| 24 CAN | 50 2 | 76 L | 102 f | |
| 25 EM | 51 3 | 77 M | 103 g | |

Sintassi per i Caratteri

Implementazione dell'ASCII.

Sono *sufficienti* 7bit per rappresentare *un carattere ASCII*; in C, ogni carattere occupa comunque 1B=8bit

Esempio:

Stringa: **ciao** è rappresentata come 4Byte:

```
99 105 97 111  
c i a o
```

Nota: non confondere numeri e caratteri

int a = 5; La variabile **a** contiene **5**

char c = '5'; La variabile **c** contiene **53**

Sintassi.

In C, un carattere si rappresenta con una *variabile* di tipo **char**.

Un **char** è molto simile a un **int** che occupa solo *1B di memoria*.

Rappresenta allo stesso tempo un carattere oppure un numero da 0 a 255.

```
char c;  
c = '5';  
c = 53; Equivalente!
```

Nota: necessario usare (solo) gli apici singoli **'**; gli apici doppi **"** racchiudono invece le *stringhe*, ovvero *vettori* di *caratteri*.

Sequenze di escape.

- Il carattere **** serve per introdurre un carattere speciale-
 - Ad esempio **\n** rappresenta il carattere di ritorno a capo
- **\n** è un singolo carattere
- Per rappresentare il carattere **** si usa la sequenza ****
- *ASCII* contiene alcuni caratteri non stampabili, detti di *speciali*. Per rappresentarli su C utilizziamo le seguenti sequenze.
 - 7 - BEL - **\a**: emetti un bip dall'altoparlante
 - 8 - BS - **\b**: cancella l'ultimo carattere
 - 9 - TAB - **\t**: tabulazione (spazio lungo)
 - 10 - LF - **\n**: avanza di una riga (*nota: questa funziona per Linux*)
 - 13 - CR - **\r**: torna alla prima colonna (*nota: funziona solo nel terminale*)

Esempio:

```
C  
char c = '\n'; //Contiene un ritorno a capo
```

Operazioni sui Caratteri

1. Stampare un carattere: 2 funzioni possibili

- Tramite **printf**:

```
C  
char c = 'a';  
printf("%c", c); // stampa: a
```

- Tramite **putchar**:

```
C  
char c = 'a';  
putchar(c); // stampa: a
```

2. Lettura di un carattere

- Tramite **scanf**:

```
C  
char c;  
scanf("%c", &c); // Legge da tastiera e mette in c
```

- Tramite **getchar**:

```
C  
char c;  
ch = getchar(); // Stesso comportamento
```

Nota: è complicato leggere un **solo** carattere. Bisogna gestire il carattere di **Invio**, che anch'esso è letto dalla **getchar**

Esempio di Operazioni sui Caratteri

Esempio: Stampare tutte le lettere maiuscole e minuscole

```
char ch;

// Maiuscole
for( ch = 'A' ; ch <= 'Z' ; ch++)
    putchar(ch);

// Minuscole
for( ch = 'a' ; ch <= 'z' ; ch++)
    putchar(ch);

putchar("\n");
```

Stringhe

Stringhe: Definizione

Definizione di Stringa.

Una *stringa* è una sequenza di caratteri.

Implementazione.

In C, si rappresenta tramite un *vettore di caratteri*.

Esistono una serie di *funzioni di libreria* per processare facilmente le *stringhe*

- Si possono anche manipolare *a mano* dei vettori di caratteri.
- Ma è *più veloce e sicuro usare le funzioni di libreria*.

Lunghezza di Stringhe

Lunghezza di una stringa: ogni funzione che processa stringhe deve conoscere il vettore su cui opera e la sua lunghezza. Questo rappresenta un *limite* per le eventuali funzioni che devono *manipolare stringhe*, dato che devono sapere tale lunghezza *a priori*.

Convenzione: Null-terminated string.

In C, per facilitare le operazioni si usano le *Null-terminated string*

- A una stringa si aggiunge sempre un **carattere terminatore**, di solito è il carattere **\0** (dopo vedremo perché).
- Quando la stringa viene processata, il carattere terminatore **indica che la stringa è finita**
- Non è necessario indicare anche la **lunghezza**
- Ogni stringa è lunga **un carattere in più**

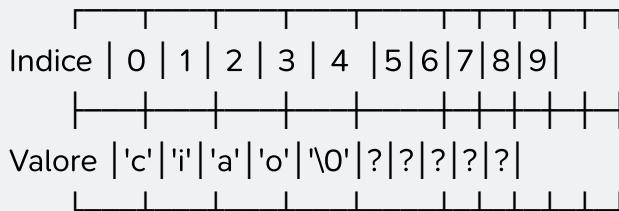
Il terminatore.

Il terminatore deve essere

- Un carattere **speciale non stampabile**, per non generare ambiguità
- Essere **ASCII** e rientrare in un **char** quindi compreso tra 0 e 255.
- Si utilizza per **convenzione** il carattere **\0** che corrisponde al numero 0

Errore comune: se si crea una stringa senza il terminatore, le funzioni di libreria hanno **comportamenti inaspettati** (di solito pericolosi!)

Esempio: si rappresenti in un vettore di lunghezza 10 la stringa **ciao**.



Non è importante il valore delle ultime 5 posizioni, **non verrà mai usato**.

Implementazione delle Stringhe

Definizione di stringhe: per definire in C una stringa ho vari modi.

1. Definendo un vettore di caratteri (**manuale**):

```
C
char s[] = {'c', 'i', 'a', 'o', '\0'}; // Il terminatore è messo dal programmatore
```

2. Usando le virgolette doppie **" "** per indicare una stringa (**automatica**):

```
C
char s[] = "ciao"; // Il terminatore è messo in automatico dal compilatore
```

Lettura di Stringhe

Principalmente ci sono *tre funzioni* per leggere stringhe: **scanf**, **gets**, **gets_s** o **fgets**

1. **Lettura di stringhe da `stdin`**: si usa la **scanf** con lo specificatore di formato **%s**.

- L'argomento deve essere un vettore di caratteri
- Non si usa l'operatore **&** (dato che ho già un *puntatore*)
 - L'operatore **&** si utilizza per passare come argomento l'indirizzo di una variabile
 - In C, passare come argomento un vettore già significa passarne l'indirizzo
- Legge fino al *primo spazio* o *ritorno a capo*.
- Termina la stringa letta col terminatore **'\0'**

Esempio: leggi una stringa

```
C  
char s[20];  
printf("Inserisci il tuo nome: ");  
scanf("%s", s); // Senza &
```

Importantissimo: se la stringa letta è più lunga di 19 caratteri, la **scanf** va a scrivere in zone di memoria arbitrarie.

Fonte di molte *vulnerabilità* software!

2. **Lettura di stringhe da `stdin`**: esiste anche la funzione **gets** che legge una stringa *fino al ritorno a capo*.

```
C  
char s[20] ;  
printf("Inserisci il tuo nome: ");  
gets(s) ;
```

Nota: ha lo stesso problema della **scanf**. Può andare a scrivere fuori dal vettore. Rimossa a partire da C11.

3. **Lettura di stringhe da `stdin`**: per scrivere un programma sicuro, utilizzare la funzione **gets_s(vettore, N)** che *non scrive più di N* caratteri su **vettore** (*compreso terminatore*)

```
char s[20];
printf("Inserisci il tuo nome: ");
gets_s(s, 20);
```

Nota: in Ubuntu, `gets_s` non è ancora implementata.

Tuttavia `gets_s(s,N)` equivale a `fgets(s,N,stdin)`.

Scrittura di Stringhe

Scrittura di stringhe su `stdout`: si usa la `printf` con lo specificatore di formato `%s`.

- L'argomento deve essere un *vettore di caratteri*
- Deve essere terminato da `'\0'`, altrimenti vengono stampati *caratteri casuali* finché non si incontra un `'\0'`

Esempio:

```
char nome [] = "Martino";
printf("Il mio nome: %s\n", nome);
```

Manipolazione delle Stringhe

Libreria Standard per le Stringhe

Le funzioni comuni su stringhe sono implementate nella *libreria standard del C*.

Necessario includere:

`#include <string.h>`

Permette di *non re-implementare* funzioni come *calcolo della lunghezza, copia, duplicazione, concatenazione*, eccetera... (quindi non scoprire l'acqua calda).

Lunghezza

Lunghezza: Si usa la funzione **strlen(s)**. Conta i caratteri finché trova il terminatore, *terminatore escluso*.

Esempio:

```
C  
char s [50];  
int l;  
printf("Inserisci una stringa: ");  
gets_s(s, 50);  
l = strlen(s);  
printf("La stringa e' lunga: %d\n", l);
```

Viene stampata la *lunghezza effettiva* della stringa immessa.

Copia di Stringhe

Copia di stringhe: si usa la funzione **strcpy(dst,src)**

Esempio:

```
C  
char s1[]="ciao";  
char s2[10];  
strcpy(s2,s1)
```

La stringa **s2** conterrà **ciao**, terminata da **'\0'**.

La **strcpy** copia *carattere per carattere*. Infatti, come abbiamo visto *non si può assegnare un vettore a un altro vettore* (**LINK DA FARE**).

```
C  
s2 = s1; // Sbagliato!
```

Concatenazione (o somma) delle Stringhe

Concatenazione: si usa la funzione **strcat(dst,src)**

Concatena **dst** e **src** e scrive tutto in **dst**

Nota! Il vettore **dst** deve essere sufficientemente lungo per contenere *la stringa risultante!*

Esempio sbagliato:

```
char s1[]="ciao";
char s2[]=" mondo";
strcat(s1, s2); // Errore! s1 è lunga 5
```

Esempio corretto:

```
char s1[15]="ciao";
char s2[]=" mondo";
strcat(s1, s2); // Corretto! s1 è lunga 15 > 4+6+1
```

Confronto tra le Stringhe

Confronto: si usa la **strcmp(a,b)** che *confronta le due stringhe carattere per carattere* e fornisce l'*ordinamento alfabetico*.

Essa ritorna un *valore numerico*, definito come:

- 0 se le stringhe sono uguali
- < 0 se **a** precede **b** in ordine alfabetico
- > 0 se **b** precede **a** in ordine alfabetico

Esempio:

```

char s1="ciao";
char s2={'c','i','a','o','\0'};
char s3="mondo";

strcmp(s1, s2); // ritorna 0
strcmp(s1, s3); // ritorna un numero <0
strcmp(s3, s1); // ritorna un numero >0

if (strcmp (s1, "ciao") ){ // Comparazione con costante
    ...
}

```

Funzioni Miste

Altre funzioni:

- *Ricerca di sotto stringhe*: **strchr** e **strstr**, **strspn**, **strcspn**
- Operazioni *su caratteri*: in **<ctype.h>** e non in **<string.h>**!
 - *Classificazione* di caratteri: **isalpha**, **isdigit**, **isupper**, **islower**
 - *Conversione* tra caratteri: **toupper**, **tolower**

Versioni sicure: le funzioni viste finora, hanno comportamenti imprevedibili se le stringhe fornite non sono terminate da **'\0'**.

- Ne esistono versioni *sicure*, in cui si forniscono la lunghezza del vettori coinvolti, per evitare di andare a *leggere o scrivere* oltre.
 - **strncpy(dst, src, n)**: come **strcpy**
 - **strncat(dst, src, n)**: come **strcat**
 - **strncmp(s1, s2, n)**: come **strcmp**
- Una buona norma è *utilizzare* queste versioni per pararsi da *stringhe malformate* di proposito.

Esercizio sulle Stringhe

Esercizio: si acquisisca una stringa da tastiera e si verifichi se è palindroma

```
#include <stdio.h>
#include <string.h>
#define MAXN 100

int main ()
{
    char s[MAXN];
    int len, i;
    printf("Inserisci una parola: ");
    scanf("%s", s);

    len=strlen(s);
    for (i=0; i<len; i++)
        if (s[i] ≠ s[len-1-i]){
            printf("Parola '%s' NON palindroma\n", s);
            return 0;
        }

    printf("Parola '%s' palindroma\n", s);
    return 0;
}
```

Conversione tra stringhe e altri tipi

Obiettivo

Esistono *funzioni* per *convertire una stringa* in un *numero intero o con virgola*.

Esempio:

- Stringa: "123" convertibile in **int** 123
- Stringa: "3.14" convertibile in **float** 3.14

Conversione Stringa → Numeri

Prime Funzioni.

- **n = atoi(s)**: converte stringa in **int**
- **f = atof(s)**: converte stringa in **float**

Nota! la stringa deve avere il **terminatore**. Non c'è controllo di errori: `atoi("ciao")` ritorna 0.

Funzione `sscanf`.

Si può usare la funzione **sscanf**. *Equivalente* alla funzione **scanf** ma ottiene i caratteri da **una stringa** e non da **stdin** (quindi si ha una situazione del tipo **stringa**→**sscanf**→**scanf**→**indirizzo variabile**). Sintassi: **sscanf(stringa, formato, argomenti)**

Esempio:

```
C  
char s[] = "314";  
int i;  
sscanf(s, "%d", &i);
```

Conversione Numeri → Stringa

Funzione `sprintf`.

Per convertire da **float** o **int** a stringa, si usa la funzione **sprintf(buffer, formato, argomenti)**, concettualmente *identica* alla **printf**, con la differenza che il risultato è salvato in **buffer** (e non stampato su **stdout**); similmente a **sscanf** si ha una situazione del tipo **numero**→**sprintf**→**printf**→**buffer**

Esempio:

```
C  
char s[100];  
int n = 425;  
sprintf(s, "%d", n); // s conterrà la stringa "425", terminata da '\0'
```

Esercizio

Esercizio: si acquisisca una riga da tastiera e si trasformi in **title case**.

Una stringa in **title case** ha le iniziali (e solo le iniziali) di ogni parola maiuscole.

Esempio: **Nel Mezzo Del Cammin Di Nostra Vita**

```

#include <stdio.h>
#include <string.h>
#include <ctype.h> // Necessario per isalpha e toupper

#define MAXN 100

int main ()
{
    char s[MAXN];
    int len, i;
    printf("Inserisci una frase: ");
    /* Notare che istruiamo fgets per leggere da standard input */
    fgets(s, MAXN, stdin);

    len=strlen(s);
    for (i=0; i<len; i++)
        /* Osservare la condizione seguente. Il secondo non genera mai
           errore perché eseguito solo se il primo è falso */
        if (i==0 || !isalpha(s[i-1])) {
            /* toupper semplicemente non ha
               effetto su numeri */
            s[i] = toupper(s[i]);
        }

    /* Non è necessario stampare '\n'. Con la fgets è incluso nella stringa */
    printf("Title Case: %s", s);
    return 0;
}

```

u3-s6-funzioni

Sistemi Operativi

Unità 3: Programmazione in C

Le funzioni

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Le funzioni
 2. La funzione **main**
-

Le funzioni

Le funzioni: Definizione

Definizione. (*funzione*)

Una *funzione* è un *insieme di istruzioni* che svolge un *completo compito*. Principalmente ha i seguenti scopi.

- Per rendere il *codice ordinato*
- Permettere il riuso del codice
- Avere codice *generico*

Quindi una funzione delimita un frammento di *codice riutilizzabile*.

- Può ricevere dei argomenti in ingresso
- Può fornire un valore di ritorno

Dopo essere definita la funzione viene *invocata*, ovvero utilizzata.

Il *copia e incolla* è da evitare!

- Disordinato
- Diventa difficile correggere errori

Esempio. (*funzione principale main*)

Il **main** è una funzione. Viene invocata dal SO quando viene *avviato il programma*.

- Riceve degli *argomenti* (non sempre, vedremo)
 - Ritorna un **int**
-

Le funzioni: Sintassi

Definizione di una funzione in C:

C

```
tipoDiRitorno nome (argomenti){
    ...
    istruzioni
    ...
    return valoreDiRitorno; // Opzionale
}
```

Esempio:

C

```
int somma (int a, int b){
    return a+b;
}
```

Argomenti di una Funzione

1. Argomenti di una funzione: Caso generale

Specificano i dati sui quali la funzione deve lavorare

- Rendono la funzione *generica*
 - Ma una funzione può non ricevere argomenti
- La funzione non deve operare *unicamente* sugli argomenti

Sintassi:

C

```
tipoDiRitorno nome (tipo nome, tipo nome, ...) {...}
```

Esempio:

C

```
float radice ( float numero ){...}
```

2. Caso Particolare: Assenza di Argomenti

Se la funzione non riceve argomenti, si indica **void**.

Esempio:

```
C  
int pigreco(void){  
    return 3.14;  
}
```

Altre funzioni che non richiedono parametri:

- Dimmi l'ora corrente: **int time(void)**

3. Caso Particolare: Argomenti Arbitrari:

Se non si indica niente come argomento **()**, quindi *non* l'opzione **void**, la funzione può ricevere un *numero arbitrario* di argomenti.

- Sistema utilizzato per funzioni come **printf** o **scanf**.
- Difficile creare funzioni con *numero variabile di argomenti*.
 - *Non ce ne occuperemo*

Esempio.

```
C  
void stampa(){  
    printf("ciao!\n");  
}  
  
stampa();  
stampa(2, 3); // Corretto, parametri ignorati
```

Valore di Ritorno di una Funzione

Valore di ritorno: Definizione

Specifica il tipo di dato ritornato dalla funzione come risultato

Se non deve ritornare un risultato, si indica **void**

Esempio:

```
C  
void stampaCiao(void){  
    printf("ciao\n");  
}
```

C

```
int somma(int a, int b){ return a+b;} // Ritorna un intero
```

Valore di ritorno:

L'istruzione **return** termina istantaneamente la funzione.

- Specifica il valore di ritorno (*se previsto*)
- *Non è necessaria* se la funzione non ha valore di ritorno (ritorna **void**)

C

```
int somma(int a, int b){
    return a+b; // Necessario ritornare un intero
}
```

C

```
void stampaCiao(void){
    printf("ciao\n");
    return; // Può essere omesso
}
```

Valore di ritorno: Uso Particolare

L'istruzione **return** può sempre essere usata per far terminare la funzione prima della fine delle istruzioni.

C

```
void stampaCiao(void){
    printf("ciao\n");      // Sempre eseguito
    return;
    printf("Mai Eseguito!\n"); // Non viene eseguito
}
```

```

int radice(int n){
    if (n<0)      // In caso di errore
        return -1; // Termina e ritorna -1
    ... calcolo...
    return r;     // Ritorna la radice se tutto ok
}

```

Le funzioni

Importanza della definizione:

La prima riga di una funzione definisce *il valore di ritorno e gli argomenti*.

Fondamentale per capire *input* e *output* della stessa.

Essa è utilizzata *per documentare* il codice

- Solo le istruzioni non sono incluse nella definizione
- Non interessa nella documentazione

Nota: in un codice C, le funzioni devono *prima* essere definite nel codice. Più avanti nel codice altre funzioni possono invocarle

Esempio:

int strlen(const char *s) calculate the length of a string

La funzione main

La funzione main: Definizione

La funzione **main** viene eseguita dal SO quando il *processo* viene avviato.

- Ovvero quando il programma *viene messo in esecuzione*

Riceve come argomenti i *parametri delle linea di comando*.

- Ovvero il testo scritto in coda al nome del programma quando lanciato

```
./myprog arg1 arg2 ...
```

Fornisce un **int** come valore di ritorno, detto *exit code*.

- Canale di comunicazione programma-SO per comunicare *errori di esecuzione*; se non ho errori ritorno 0, altrimenti ritorno un altro numero.

La funzione **main**: Sintassi

Definizione:

```
C  
int main(int argc, char *argv[]);
```

Argomenti:

- **int argc**: *numero di parametri* sulla riga di comando.
 - In assenza di parametri vale 1.
 - *Incrementato per ogni parametro*.
- **char* argv[]**: *vettore dei parametri*.
 - E' un *vettore di puntatori a carattere* (ovvero *vettore di stringhe*)
 - Ogni puntatore a carattere del vettore è un *argomento in forma di una stringa*
 - **argv[0]** è sempre il *nome del programma*. I parametri effettivi iniziano da **argv[1]**

Esempio:

```
C  
.myprog ciao mondo
```

argc vale 3

argv vale ".myprog", "ciao", "mondo"

```
C  
.myprog
```

argc vale 1

argv vale ".myprog"

Esempio: Stampa di **argc** e **argv**

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int i;
    printf("argc = %d\n", argc);
    for(i=0; i<argc; i++)
        printf("argv[%d] = \"%s\"\n", i, argv[i]);
    return 0;
}
```

Esecuzione:

SHELL

```
./sample
argc = 1
argv[0] = "./sample"
```

SHELL

```
./sample ciao mondo
argc = 3
argv[0] = "./sample"
argv[1] = "ciao"
argv[2] = "mondo"
```

La funzione **main**: Osservazioni

Osservazioni:

argv contiene un vettore di stringhe. Se essi devono essere interpretati *come numeri*, vanno convertiti tramite funzioni come **atoi**, **atof** o **sscanf**.

Se un programma *non ha necessità* di ricevere dei parametri, può definire il **main** senza argomenti.

```
int main(){}
int main(void){}
```

La funzione **main**: Valore di Ritorno

Valore di ritorno:

il **main** può restituire un **int** che viene esaminato dal SO.

Esso indica se c'è stato un errore nell'esecuzione. Per convenzione.

- 0 indica che non c'è stato errore
- Un numero diverso da 0 indica un errore.
 - Il significato del numero, è *specifico del programma*

Questo sistema viene molto utilizzato:

- In *script Bash* che necessitano di sapere se i programmi eseguiti hanno avuto errore
- Per i *moduli del SO*, che devono avviare servizi, demoni e programmi in background.

Esempio: si scriva un programma che accetta un solo parametro e lo stampa.

```
#include <stdio.h>
int main(int argc, char *argv[]){
    /* Con un parametro, argc=2,
       siccome il primo elemento è il nome del programma */
    if (argc!=2){
        printf("Numero di parametri errato\n");
        return 1; /* Il programma termina in questo punto */
    }

    printf("%s\n", argv[1]);
    return 0;
}
```

Lettura del valore di ritorno in Bash:

Per ottenere il valore di ritorno di un programma all'interno di uno script Bash si usa la variabile **\$?**.

Contiene il *valore di ritorno dell'ultimo programma lanciato*

```
./myprog ciao mondo
code=$? //Necessario salvarlo, altrimenti sovrascritto dopo echo
echo "MyProg ha fornito come codice di ritorno $code"
if (( $code == 0 )) ; then
    echo "Nessun errore"
else
    echo "Errore"
fi
```

Nota: il valore di `$?` viene scritto dopo ogni comando, anche dopo `echo`, dato che è un *programma!*

Osservazione: si può dichiarare la funzione `main` perché non ritorni nulla. Il programma funziona ma non è corretto. Il SO riceve un *valore di ritorno casuale*.

```
void main (int argc, char * argv[]){...}
```

Sarebbe da evitare.

Esercizio

Esercizio: si scriva un programma che riceve due interi come parametri e ne stampa la somma.

```

#include <stdio.h>
#include <stdlib.h> /* Necessario per atoi */
int main(int argc, char *argv[]){
    int a, b;

    if (argc!=3){ /* Considerare che argv[0] è il nome del programma */
        printf("Usage: somma a b\n");
        return 1; /* Codice di errore */
    }

    a = atoi(argv[1]); /* Conversione a int*/
    b = atoi(argv[2]);
    printf("%d\n", a+b);

    return 0; /* Nessun errore */
}

```

Nota: cosa succede se viene lanciato come **./somma 3 4** e come **./somma ciao mondo**?

u3-s7-puntatori

Sistemi Operativi

Unità 3: Programmazione in C

I puntatori

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Puntatori in C
2. Puntatori e vettori
3. Puntatori e stringhe
4. Puntatori e funzioni
5. Puntatori a **struct**

Puntatori in C

Definizione di Puntatore

Definizione. (*puntatore*)

Un *puntatore* è una variabile che contiene un indirizzo di memoria.

Non ci interessa come è strutturato l'indirizzo (di solito è di 8byte)

- Esso è comunque un *indirizzo virtuale*, che viene tradotto in un *indirizzo fisico dalla Memory Management Unit*.

Implementazione in C.

In C, una *variabile puntatore* contiene un *indirizzo di memoria* dove è contenuta una variabile *di un certo tipo* (quindi sono tipizzati!).

- Tutte le volte che dichiaro un puntatore, devo dichiarare anche che tipo di dato contiene l'indirizzo di memoria che contiene. Questo perché il compilatore dev'essere in grado di *interpretare l'indirizzo* dato.
 - *Fondamentale per l'utilizzo pratico*
-

Puntatori in C: Sintassi

1. Dichiarazione di un puntatore: **tipo * nome;**

Esempio:

```
int * pi; // Puntatore a int
float * pf; // Puntatore a float
```

Il tipo **int *** indica una variabile puntatore a intero: contiene l'indirizzo di memoria alla quale troviamo una variable intera.

2. Assegnazione un puntatore: si può usare l'operatore **&** per ottenere l'indirizzo di una variabile esistente.

Esempio:

```
int a = 5;  
int * pi;  
pi = &a; // pi contiene l'indirizzo di a
```

Nota: gli indirizzi sono dei numeri interi. Non è dato sapere quanto lunghi, dipende dall'architettura.

3. **Accesso alla variabile puntata** (oppure *dereferenziazione*): l'operatore ***** applicato a un puntatore serve per accedere alla variabile puntata. Detto *operatore di dereferenziazione*.

Esempio:

```
int a = 5;  
int * pi;  
pi = &a; // pi contiene l'indirizzo di a  
int b = *pi; // b contiene il valore 5
```

L'operatore ***** è l'inverso di **&**:

Pertanto: ***(&a) == a** e **&(*pi) == pi**

4. **Scrittura nella variabile puntata:**

Usando l'operatore ***** si può sia leggere che *scrivere nella zona di memoria*.

Esempio:

```
int a = 5;  
int * pi = &a; // pi contiene l'indirizzo di a  
*pi = 10; // Modifico il contenuto di a  
printf("%d", a); // Stampa 10
```

Esempio Generale di Puntatori

Esempio:

```
#include <stdio.h>

int main ()
{
    int a=5, *p;

    p = &a;
    printf("a=%d\n", a); // Stampa: 5
    printf("p=%p\n\n", p); // Stampa un indirizzo, e.g., p=0x7ffc6de0703c

    printf("&a=%p\n", &a); // Stampa lo stesso indirizzo, e.g., p=0x7ffc6de0703c
    printf("*p=%d\n", *p); // Stampa: 5

    return 0;
}
```

Puntatori in C: Osservazione sulla funzione scanf

Osservazione: ora appare più chiaro perché nella funzione **scanf** bisogna usare l'operatore **&** per passare gli argomenti in lettura.

```
float a;
scanf("%f", &a);
```

Significa che la funzione **scanf** riceve come argomento l'indirizzo di una variabile **float**.

La funzione scriverà in quell'indirizzo il valore letto da tastiera.

Internamente effettuerà un'operazione del tipo:

```
*pf = valore;
```

Puntatori e vettori: Similitudini

Osservazione. (relazione puntatori-vettori)

In C, puntatori e vettori hanno una *stretta relazione*. Infatti, sono praticamente le *stesse*.

Il nome di un vettore senza indice, ritorna l'indirizzo del primo elemento del vettore.

Esempio:

```
C  
int v[5] = {5,6,7,8,9};  
int * pi;  
pi = v; // Operazione consentita  
printf("%d\n", *pi); // Stampa: 5
```

Dato il vettore: **int v[5];**, sono equivalenti

- **v** \Leftrightarrow **&(v[0])**
- **v[0]** \Leftrightarrow ***v**

Corollario. (Aritmetica dei puntatori)

E' possibile *sommare interi a un puntatore*, per accedere a *locazioni contigue*.

Ogni incremento di 1 di un puntatore, fa accedere al blocco successivo *di lunghezza del tipo del puntatore*.

Esempio

- Il puntatore **int * pi** contiene l'indirizzo 1000
- Il tipo **int** è su $32bit = 4B$
- Allora: **pi+1** indica l'indirizzo 1004, **p+2** indica 1008
- In generale: **pi+N** indica l'indirizzo $1000 + N \times 4$ (ovvero vado alla "*prossima cella*")

Osservazione.

Sono quindi equivalenti:

- **&(v[2])** e **v+2**
- **v[2]** e ***(v+2)**
- In generale ***(v+i)** \Leftrightarrow **v[i]**, per tali valori *i* consentiti.

Si può iterare su un vettore facendo:

```
for (i=0; i<N; i++)
    v[i] = ...
    ... equivale a ...
    *(v+i) = ...
```

Puntatori e vettori: Differenze

Differenze tra puntatori e vettori

- Notiamo che i *puntatori e vettori* sono comunque *diverse*.
- Un *puntatore* può essere *riassegnato* per puntare a un altro indirizzo
- Un *vettore* è un contenitore *immutabile*.
 - Tecnicamente è un *puntatore costante*. Non gli può essere assegnato un altro valore.

```
char v[10], *pv;
pv = v; // Consentito
pv = v + 3; // Consentito
v = pv; // Errore!
v = pv + 2; // Errore!
```

Corollario. (*Rappresentazioni delle stringhe*)

Sappiamo che una *stringa* è un vettore di **char** terminato dal terminatore '\0'. Abbiamo due modi per rappresentare tale stringa:

```
char stringa [] = "ciao";
char * ps = stringa;
```

- Un puntatore a **char** può *riferirsi a una* stringa.
 - Il puntatore **ps** contiene l'*indirizzo del primo elemento* di **stringa**.
- Tutte le *funzioni di manipolazione delle stringhe* prendono come argomento *un puntatore* a **char**

```
strlen(stringa);
```

La funzione **strlen** prende come argomento un **char ***

Puntatori e stringhe: Errori gravi

Errori gravi:

1. Dereferenziare un puntatore non inizializzato: *comportamenti imprevedibili*, al meglio un *segmentation fault*

```
int * pi;
int a = *pi; // Errore! pi contiene un indirizzo a caso!
```

2. Dereferenziare un intero (oppure qualsiasi cosa che *non sia* un puntatore): *comportamenti ancora più imprevedibili*

```
int i = 12;
int j = *i; // Errore: i contiene 12, non un indirizzo
```

In questo caso, il compilatore *solleva un errore (segmentation fault)*.

Puntatori e funzioni: Passaggio dei Parametri per Riferimento

Definizione. (*passaggio di parametri per riferimento*)

I puntatori si usano per passare parametri *per riferimento*.

In questo modo, la funzione *può modificare gli argomenti* che riceve e il chiamante vederne gli effetti.

- Come la **scanf** che *modifica il valore* di una variabile argomento.
- Tecnica usata quando una funzione deve ritornare *più di un valore* (ricordiamo che una funzione può tornare al massimo *un solo* valore)
 - I valori di ritorno aggiuntivi *sono puntatori forniti dal chiamante*
 - In cui la funzione *colloca il risultato*

Esempio: si scriva una funzione che calcola la lunghezza di una stringa terminata da '\0'.

```
C  
int len ( char * s ){  
    int i = 0;  
    while ( *(s+i) != '\0' ) // Aritmetica dei puntatori  
        i++;  
    return i;  
}
```

Utilizzo:

```
C  
char stringa [] = "ciao!";  
int a;  
a = len(stringa); /* Non è necessario l'operatore &.  
Il nome di una variabile vettore  
già indica l'indirizzo del primo elemento */
```

Esempio: si scriva una funzione che prende due interi per riferimento e ne scambia il valore.

```
C  
void swap ( int * a, int * b ){  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

Utilizzo:

```
C  
int i = 5;  
int j = 7;  
swap (&i, &j);
```

Dopo l'esecuzione: **i=7** e **j=5**

Puntatori a **struct**

Puntatori a struct: un puntatore può tranquillamente puntare una **struct**.

```
C  
struct punto {float x; float y;}  
struct punto p1 = {1, 4};  
struct punto * pp;  
pp = &p1; // Assegno a pp l'indirizzo di p1
```

Se può accedere agli elementi di una struct *tramite puntatore*. Alternativamente è possibile usare la sintassi →, spiegata come segue.

```
C  
(*pp).x; // Contiene il valore 1  
(*pp).y; // Contiene il valore 4
```

Operatore →: si utilizza su puntatori a **struct**.

permette di accedere direttamente a un campo della **struct** puntata.

Sintassi: **puntatore→campo** ($\equiv (*\text{puntatore}).\text{campo}$)

Esempio:

```
C  
struct punto {float x; float y;}  
struct punto p1 = {1, 4};  
struct punto * pp = &p1;
```

Le seguenti istruzioni si equivalgono e ritornano l'**int** 1.

```
C  
(*pp).x;  
pp→x;
```

Nota. Con la prima sintassi, le parentesi sono *fondamentali*, dato che la precedenza viene data prima all'operatore **.**

Puntatori a funzione: Definizione

Osservazione preliminare.

E' possibile passare delle **funzioni** come **argomento a una funzione**.

- Si usa per rendere il **codice generico e modulare**, per il **multithreading**, ecc...
- Useremo quando ci occuperemo di **thread**.

Esempio di applicazione: Voglio creare una funzione che riceve come argomento un vettore e una **funzione**.

- Essa applica la **funzione** fornita su ogni elemento del vettore (funzione nota come "*mapping*" o "*mappatura*")

Definizione. (**puntatore a funzione**)

- Un **puntatore a funzione** è una variabile che rappresenta la posizione di una funzione.
- **Dereferenziarlo** significa **invocare** la funzione.

Un puntatore a funzione è tipizzato:

- Può puntare funzioni che ritornano un tipo ben definito
- E accettano un certo tipo di argomenti

Puntatori a funzione: Sintassi in C

Dichiarazione:

La sintassi è:

tipoDiRitorno (* nome) (tipoArg1, tipoArg2, ...)

Esempio:

```
C  
int (*pf) (int, int);  
// Il puntatore accetta solo funzioni del tipo f: int x int → int
```

Dichiara il puntatore a funzione di nome **pf** che può **puntare a funzioni** che:

- accettano due **int** come argomento
- ritornano un **int**
- ovvero una funzione del tipo $f : \text{int} \times \text{int} \rightarrow \text{int}$

Puntatori a funzione: Operazioni

1. **Assegnazione e utilizzo:** si usano gli operatori `=` per assegnazione (*senza &!*) e `*` per dereferenziazione, come di consueto.

Esempio:

```
C  
int somma (int a, int b){return a+b;}  
...  
int (*pf) (int, int); // Dichiarazione  
  
pf = somma; // Assegnazione. Non serve &  
res = (*pf)(3,5); // Invoca la funzione  
// res contiene 8
```

Caso d'Uso Frequenti

Funzioni che accettano come argomento puntatori a funzione:

E' uno degli *utilizzi più frequenti*. Rendono *generiche* le funzioni il più possibile.

Esempi:

- Funzione che *ordina* secondo un criterio fornito dall'utilizzatore
- Funzione del SO che avvia un *thread* che esegue una funzione fornita dall'utente

Puntatori a funzione: Esempi

Esempio: si crei e si usi una funzione che applica a ogni elemento di un vettore di interi una funzione fornita dal chiamante.

```
#include <stdio.h>
void apply(int * v, int n, int (*f)(int) ){
    for (int i=0; i<n; i++) // Dichiaro i nel loop
        v[i] = (*f)(v[i]);
}

int square(int a) {return a*a;}

int main(){
    int vec [] = {1,2,3,4,5};
    apply(vec, 5, square);
    for (int j=0; j<5; j++)
        printf("vec[%d]=%d\n", j, vec[j]);
    return 0;
}
```

Esempio: si crei e si usi una funzione che combini due interi usando una funzione fornita dal chiamante e stampi il risultato.

```
#include <stdio.h>

void combineAndPrint(int a, int b, int (*comb)(int,int) ){
    int p = (*comb)(a, b); // Dereferenziazione di comb
    printf("Combinazione: %d\n", p);
}

int add(int a, int b) {return a+b;}
int mult(int a, int b){return a*b;}

int main(){
    combineAndPrint(3, 4, mult); // Stampa 12
    combineAndPrint(3, 4, add); // Stampa 7
    return 0;
}
```

Nota:

combineAndPrint(3, 4, mult); e **combineAndPrint(3, 4, &mult);** sono equivalenti

Operazioni sui file

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. File
 2. Apertura e chiusura file
 3. Lettura/scrittura su file
 4. Gestione degli errori in C
-

File in C

Premessa

In C è possibile interagire coi File.

In C, come in tutti i linguaggi, è possibile *leggere e scrivere su file*.

Esistono varie funzioni per farlo.

1. **Funzioni di libreria:** portabili tra SO. In queste pagine vedremo queste, contenute in `<stdio.h>`; sono le *stesse* in tutti i *sistemi operativi*.
2. **System Call** per *Linux*

Tipi di File.

I file possono essere:

- Binari: contenere *sequenze di bit arbitrarie*.
 - |◦ Immagini, file compressi
- Testuali: contengono *solo caratteri stampabili*
 - |◦ File `.txt`, `.html`, sorgenti di programmi `.c`

Noi vedremo principalmente file testuali.

Operazioni di base sui file.

Le operazioni di base sono:

- Apertura di un file; informo l'**S.O.** che sto per farci qualcosa
- Lettura o scrittura nel file; *effettuo le operazioni necessarie*
- Chiusura del file; *concludo e salvo le modifiche*

Vedremo le funzioni principali per queste operazioni.

- Esistono *anche altre operazioni*, che vedremo nel corso quando parleremo di file e **file system**
-

Apertura e chiusura file

Apertura di un File

Apertura di un File in generale.

Un programma può accedere a file su disco, tramite il loro **path** (ovvero quella stringa che indica la posizione del file sul disco).

Prima di leggere o scrivere un file, il programma deve *aprirllo*.

- Indicare al sistema operativo che accederà a tale file e in che **modalità**

La **modalità** può essere:

- Lettura (read)
- Scrittura (write)
- Aggiunta (o *append*)

L'accesso (scrittura/lettura) ai file è **sequenziale**.

Si inizia a leggere o scrivere dall'inizio e si prosegue

- Simile all'idea di **cursore** degli editor grafici; infatti c'è un "**cursore virtuale**", che rappresenta la **posizione del file aperto**.
- Esistono funzioni per riposizionare il **cursore**, vedremo più avanti

Quando si apre un file, bisogna *indicare* se esso è **binario o testuale**.

- Se testuale, le funzioni si aspettano **\n** per delimitare le righe

Implementazione in C.

Per indicare un **file aperto**, su cui è possibile effettuare operazioni, in C si usa il tipo **FILE**

- Tecnicamente esso è *un puntatore* a una variabile di tipo **FILE**.
- Non ci interessa che tipo è **FILE**; precisamente sarebbe una *struct*, ma è in ogni caso una questione che non ci riguarda.

FILE * è un cosiddetto *handle opaco*: è un puntatore. *Ma non ci interessa a cosa punta*.

Solo le funzioni di libreria hanno interesse ad accedere al dato, al *programmatore* non ne interessa niente.

Pertanto il contenuto di **FILE** può cambiare o non seguire uno standard.

Apertura di File in C

Per aprire un file si usa la funzione **fopen(path, modo)**. Essa ritorna un **FILE ***.

- **path** può essere assoluto o relativo.
- **modo** indica se apriamo in lettura (**r**), scrittura (**w**) o aggiunta (**a**). Di default la modalità è *testuale*. Per indicare *modalità binaria*, aggiungere **b** (es. **rb** o **wb**).
- Nota benissimo: si deve inserire una *stringa* in entrambi i casi!

Esempio:

```
FILE * f;  
f = fopen("file.txt", "r");  
if (f==NULL){ /*Errore*/ }
```

In caso di errore, la **fopen** ritorna il puntatore nullo **NULL**.

Note. (*Comportamenti in casi di non-esistenza del file*)

- In modalità *scrittura e aggiunta*, se il file non esiste, *viene creato*.
- In modalità *lettura*, se il file non esiste, la **fopen** ritorna **NULL**.
- In modalità *scrittura*, se il file esiste, il suo contenuto *viene cancellato all'apertura*

Chiusura di File

Chiusura file: quando non si accede più a un file, bisogna *chiuderlo* e dismettere il corrispondente **FILE ***

Si chiama la funzione **fclose(file)** che accetta come argomento un **FILE ***.

- Mai chiamare la **fclose** con un **FILE *** invalido settato a **NULL**!
- Si può chiudere un file aperto *solo una volta*

Se un file non viene chiuso, la chiusura è effettuata *automaticamente dal SO* quando il programma termina.

Lettura/scrittura su file

Lettura su File

Le funzioni che si usano simili a quelle che si usano per leggere da tastiera e scrivere su console.

1. Lettura:

- **fgetc(file)**: legge *un carattere* e lo fornisce come *valore di ritorno*
 - Ritorna la costante **EOF** se il file è finito ("*End of File*")
- **fgets(buffer, N, file)**: legge *una stringa* di massimo **N** caratteri.
 - Legge fino a quando ha letto **N** caratteri o trova **\n**, che è *incluso* nella stringa ritornata e terminata da **\0**
 - Ritorna **NULL** (= 0) se il file è finito, altrimenti ritorna **buffer**

Scrittura su file

2. Scrittura:

- **fputc(carattere, file)**: scrive un *carattere* su file
- **fputs(stringa, file)**: scrive una *stringa* su file

Nota: queste funzioni possono leggere e scrivere *anche su terminale*. E' sufficiente dire loro di leggere dal file **stdin** e scrivere su file **stdout**.

- **fputc('a', stdout) ≡ putc('a')**
- Abbiamo già visto che la **fgets** è una valida alternativa alla insicura **gets**.
- Infatti queste funzioni sono le "*versioni generiche*" delle funzioni appena menzionate.

Esempio: si legga un path da tastiera e se ne stampi il contenuto come file di testo

```
#include <stdio.h>

int main ()
{
    char s[100], buffer [100];
    FILE * f;

    printf("Inserisci un path: ");
    scanf("%s", s);

    f = fopen(s, "r");
    if (f==NULL){
        printf("Impossibile aprire %s\n", s);
        return 1; /* Errore */
    }

    /* Non è importante la lunghezza di buffer */
    while ( fgets(buffer, 100, f) )
        fputs(buffer, stdout); // Equivale a printf("%s", s);

    fclose(f);
    return 0;
}
```

Comandi Generali di Lettura/Scrittura su File

Comandi Alternativi.

- Alternativamente si possono usare le funzioni **fprintf** e **fscanf** che sono equivalenti a **printf** e **scanf** con la differenza che scrivono *su file* e *non da console*.

Sintassi:

fprintf(file, formato, argomenti)
fscanf (file, formato, &argomenti)

Queste funzioni sono particolarmente utili per leggere e scrivere *numeri interi e reali*

Esempio:

```
fprintf(f, "%d\n", 123); // scrive il numero 123 e un ritorno a capo
```

Esempio: si legga un numero N da tastiera e si stampi sul file **numeri.txt** i numeri da 1 a N

```
C

#include <stdio.h>

int main ()
{
    int n, i;
    FILE * f;

    printf("Inserisci un numero: ");
    scanf("%d", &n);

    f = fopen("numeri.txt", "w");
    if (f==NULL){
        printf("Impossibile aprire numeri.txt\n");
        return 1;
    }

    for (i=1; i<=n; i++)
        fprintf(f, "%d\n", i);

    fclose(f);
    return 0;
}
```

Nota su **fgetc** e **fgets**

Nota.

In caso di **lettura**, abbiamo visto che **fgetc** e **fgets** hanno *comportamenti diversi*. In pratica, in caso di file completamente letto:

- **fgetc** ritorna la costante **EOF**
- **fgets** ritorna la costante **NULL**
- **fscanf** ritorna la costante **EOF**
- Non abbiamo **numeri** 1 o 0
Questo è un retaggio storico delle versioni vecchie.

E' possibile usare la funzione **eof(FILE *)** per verificare se il file è stato letto completamente.

- Ritorna **TRUE** / **FALSE**

Formati per comandi `printf` e `scanf`

Abbiamo visto che `(f/s)printf` e `(f/s)scanf` permettono di *specificare il formato* come `%d %f %s %c`.

Riassumiamo:

- Carattere `char`: `%c`
- Stringa `char []`: `%s`
- Intero `int`: `%d`
- Reale `float`: `%f`

Lista completa: <https://man7.org/linux/man-pages/man3/printf.3.html>

Esistono modificatori per definire numero di cifre decimali, padding per interi, ecc..

Esercizio su Lettura/Scrittura di File

Esercizio: si legga `persone.txt` che contiene su ogni riga nome ed età di una persona, separati da `' '`. Si calcoli l'età media. Esempio di file:

```
martino 31
```

```
andrea 37
```

```
#include <stdio.h>

int main (){
{
    int n=0, s, e;
    FILE * f;
    char nome[100];

    f = fopen("persone.txt", "r");
    if (f==NULL){
        printf("Impossibile aprire persone.txt\n");
        return 1;
    }

    /* La fscanf si aspetta di trovare su ogni riga una parola e un intero */
    while (fscanf(f, "%s %d\n", nome, &e) != EOF){
        n++; s+=e; /* Accumula i contatori */
    }

    printf("La media è %f\n", (float)s/n ); /* Notare il casting */
    return 0;
}
```

Gestione degli errori in C

Problematica della Gestione degli Errori

Osservazione.

Abbiamo visto che le funzioni di libreria segnalano un eventuale errore tramite il valore di ritorno

Esempio:

```
f = fopen("file.txt", "r");
if (f==NULL){
    /*Errore*/
}
```

Problema.

Con questo meccanismo, non è possibile sapere niente su *quale sia stato l'errore*.

- File non esistente?
- No permessi di lettura?
- Non si sa

Metodo Errno

Soluzione: Metodo Errno

La *libreria standard del C* utilizza il seguente meccanismo per *specificare la causa* di errore

- Ogni programma in C ha la *variabile globale int errno*
- Una funzione di libreria che fallisce, setta **errno** con un *codice di errore esplicativo*
- Il chiamante invoca una funzione di libreria.
 - Tramite il valore di ritorno, rileva se c'è stato un errore
- Se c'è stato un errore, il chiamante legge in **errno** il codice di errore

Necessaria includere la libreria

```
#include <errno.h>
```

Definizione: Errno

La variabile globale **errno** è *intera* e contiene un *codice di errore*.

- il manuale di Linux (**man errno**) contiene la descrizione dei codici di errore
- *Esempi:* **fopen** può fallire con **ENOENT**= 2 (file inesistente), **EACCES**= 13 (permessi insufficienti) e molti altri

Stampa dell'errore

Implementazione di Errno.

Tutti i codici di errore sono costanti definite nella libreria standard. Adesso basta trovare un modo per *associare* il codice all'*errore* effettivo.

- Il programmatore può confrontare **errno** con le costanti definite in **<errno.h>** per identificare l'errore

Esempio:

```
C  
FILE * f = fopen("file.txt", "r");  
if (f==NULL){  
    if (errno == ENOENT)  
        printf("File inesistente\n");  
    else if (errno == EACCES)  
        printf("Permessi insufficienti\n");  
    else  
        printf("Errore generico\n");  
    return 1;  
}
```

- Esistono delle funzioni di libreria per semplificare la gestione dell'errore, tra cui la funzione **errno**.

```
C  
#include <stdio.h>  
void perror(const char *s);
```

Stampa il messaggio di errore relativo al valore corrente di **errno**, premettendo la stringa **s**

Esempio:

```
C  
FILE * f = fopen("file.txt", "r");  
if (f==NULL){  
    perror("Error opening file.txt");  
    return 1;  
}
```

Stampa: **Error opening file.txt: No such file or directory**

- Funzione **strerror**

```
#include <string.h>
char *strerror(int errnum);
```

Ritorna *una stringa* che spiega l'errore dal codice **errnum**

Esempio:

```
FILE * f = fopen("file.txt", "r");
if (f==NULL){
    printf("Impossibile aprire il file. Errore: %s\n", strerror(errno));
    return 1;
}
```

Stampa: **Impossibile aprire il file. Errore: No such file or directory**

Gestione degli errori in C: i limiti

Problemi.

La gestione degli errori tramite **errno** è considerata obsoleta.

- I linguaggi più moderni usano i costrutti **try catch** (oppure **try**, **except** in Python)

Casi d'uso problematici.

La gestione degli errori tramite la variabile globale **errno** è una *tecnica problematica*, in caso di:

- In caso di segnali (*vedremo*)
- Fortunatamente **errno** è thread safe (*vedremo*)

u3-s9-esercizi

Sistemi Operativi

Unità 3: Programmazione in C

Esercizi

[Martino Trevisan](#)

[Università di Trieste](#)

Argomenti

1. Stampa di file
 2. Area di un triangolo
 3. Calcolo del minimo
 4. Calcolo della media
 5. Somma di vettori bidimensionali
-

Stampa di file

Si scriva un programma che riceve il nome di un file da riga di comando e ne stampa il contenuto.

```
C  
#include <stdio.h>  
int main(int argc, char *argv[]){  
    FILE * f;  
    char buffer[100];  
  
    if (argc!=2){ /* Controllo degli argomenti */  
        printf("Uso: ./stampa path\n");  
        return 1;  
    }  
  
    f = fopen(argv[1], "r");  
    if (f == NULL){ /* Controllo sul file */  
        printf("Impossibile aprire il file\n");  
        return 1;  
    }  
  
    /* Stampa finchè non termina il file */  
    while (fgets(buffer, 100, f) !=NULL)  
        printf("%s", buffer);  
  
    fclose(f);  
    return 0;  
}
```

Area di un triangolo

Si scriva un programma che riceve base e altezza di un triangolo da riga di comando e stampa la sua area. Base e altezza sono numeri con virgola.

```
C

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    float base, altezza;

    if (argc!=3){ /* Controllo degli argomenti */
        printf("Uso: ./area base altezza\n");
        return 1;
    }

    /* Conversione */
    base = atof(argv[1]);
    altezza = atof(argv[2]);

    /* Controllo base e altezza maggiori di 0 */
    if (base<0 || altezza <0){
        printf("Parametri non validi. Devono essere maggiori di 0.\n");
        return 1;
    }

    printf("Area: %f\n", base*altezza/2);

    return 0;
}
```

Calcolo del minimo

Si scriva un programma che riceve come parametro il nome di due file:

- Il primo file è di input e contiene un intero *positivo* per riga
- Il secondo file è di output e vi viene scritto il numero minimo del file di input

Si crei una riga di comando in bash che svolge lo stesso compito, ipotizzando che il file di input sia **in.txt** e quello di output **out.txt**

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    FILE * fin, *fout;
    int n, min;

    if (argc!=3){ /* Controllo degli argomenti */
        printf("Uso: ./minimo filein fileout\n");
        return 1;
    }

    fin = fopen(argv[1], "r");
    if (fin == NULL){ /* Controllo sul file */
        printf("Impossibile aprire il file in input\n");
        return 1;
    }

    fout = fopen(argv[2], "w");
    if (fout == NULL){ /* Controllo sul file */
        printf("Impossibile aprire il file di output\n");
        return 2;
    }

    fscanf(fin, "%d\n", &min); /* Valore iniziale per il minimo */
    while (fscanf(fin, "%d\n", &n) != EOF ) /* Cerca il minimo */
        if (n<min)
            min = n;

    fprintf(fout, "%d\n", min);
    fclose(fin);
    fclose(fout);
    return 0;
}
```

La versione in bash è molto più compatta

SHELL

```
cat in.txt | sort | head -1 > out.txt
```

Calcolo della media

Si scriva e si testi una funzione che calcola la media di un vettore

```
#include <stdio.h>

float media(int n, float * v){
    float s = 0;
    int i;
    for (i=0; i<n; i++)
        s+=v[i];
    return s/n;
}

int main(int argc, char *argv[]){
    float lista [] = {1.5, 2.5, 4};
    printf("Media: %f\n", media(3, lista) );
    return 0;
}
```

Somma di vettori bidimensionali

Si scriva e si testi una funzione che calcola la somma di due vettori bidimensionali.
Se ne fornisca una versione con e senza l'uso delle **struct**.

Senza **struct**

E' necessario l'uso dei puntatori, siccome la funzione deve ritornare due valori.

```
#include <stdio.h>

void sommaV(float x1, float y1, float x2, float y2,
            float * pxres, float * pyres) {
    *pxres = x1+x2;
    *pyres = y1+y2;
}

int main(int argc, char *argv[]){
    float punto1x = 1.1, punto1y = 2.0, punto2x = 3.6, punto2y = 2.7;
    float puntoSx, puntoSy;

    sommaV(punto1x, punto1y, punto2x, punto2y, &puntoSx, &puntoSy);
    printf("La somma vettoriale e': (%f, %f)\n", puntoSx, puntoSy);

    return 0;
}
```

Con **struct**

Utilizziamo una **typedef** per evitare di ripetere molte volte la keyword **struct**.

```
#include <stdio.h>

typedef struct {
    float x;
    float y;
} punto;

punto sommaP(punto p1, punto p2){
    punto risultato;
    risultato.x = p1.x + p2.x;
    risultato.y = p1.y + p2.y;
    return risultato;
}

int main(int argc, char *argv[]){
    punto p1 = {1.4, 4.2};
    punto p2 = {3.2, 5.9};

    punto s = sommaP(p1, p2);
    printf("La somma vettoriale e': (%f, %f)\n", s.x, s.y);

    return 0;
}
```

u4-s1-file-system

Sistemi Operativi

Unità 4: Il File System

I dischi e i file system

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Il disco nei sistemi di elaborazione
 2. Dati e disco
 3. I file
 4. I Direttori
 5. Allocazione dei blocchi
 6. File System Comuni
 7. Tabella delle partizioni
-

Definizione di Disco negli Elaboratori

Definizione (*disco*)

Dal modulo sulle [*architetture degli elaboratori*](#) (1), ricordiamoci che il *disco* è un *componente fondamentale dei sistemi di elaborazione*

- Permette di avere una memoria persistente
 - Sopravvive al riavvio dell'elaboratore
- È una memoria riscrivibile
 - Diversamente da ROM, PROM e EPROM

Tipologie di Disco

Tipologie di Disco

Ci ricordiamo che ci sono diverse tecnologie per costruire i dischi

- *Nastri magnetici*: obsoleti/storici
- *Dischi magnetici*: (o note come "HDD") i più usati
- *Stato solido (memorie flash)*: (o note come "SSD/formalmente EPROM") in ascesa ancora oggi

Ogni tipologia si differisce per *prestazioni, costi, affidabilità, meccanismo di accesso*.

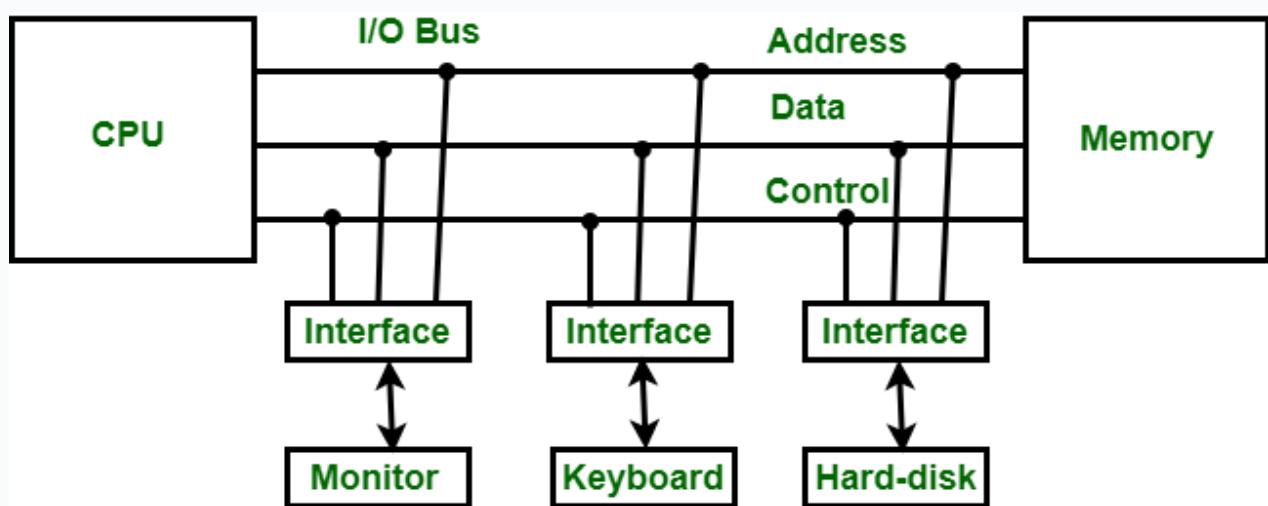
- Nei dischi magnetici la testina si sposta (HDD)
 - Accesso al disco non ha un tempo costante
 - Letture sequenziali preferite
 - Costo minore
- Memorie flash: tempo di accesso quasi costante (HDD)
 - Scrittura più lenta di lettura
 - Costo maggiore

Accesso al Disco

Definizione (interfaccia)

Il disco è un dispositivo *di I/O*, quindi la CPU lo utilizza attraverso un'*interfaccia*

FIGURA (Schema di architettura)



modo approssimato dell'accesso al disco

In prima approssimazione (ad alto livello), la CPU accede al disco nel seguente modo:

1. La CPU scrive nell'*interfaccia del disco* la *locazione di memoria* che vuole leggere o scrivere
 - Assieme a *informazioni di controllo* (e.g., se Read o Write)
 - L'accesso all'interfaccia avviene come a una qualunque locazione di memoria
2. Il disco *esegue* l'operazione
3. Il disco *setta dei flag* nell'interfaccia che segnalano che l'*operazione è conclusa*
4. La CPU, leggendo i flag, realizza che l'*operazione è terminata*
 - Eventualmente legge i dati dall'interfaccia (in caso di Read)
 - Nota come la tecnica del "*polling*" (1)

Ottimizzazione dell'Accesso ai Dischi

Esistono altre tecniche per rendere **più efficiente** l'accesso al disco. Le accenniamo, e sono le seguenti.

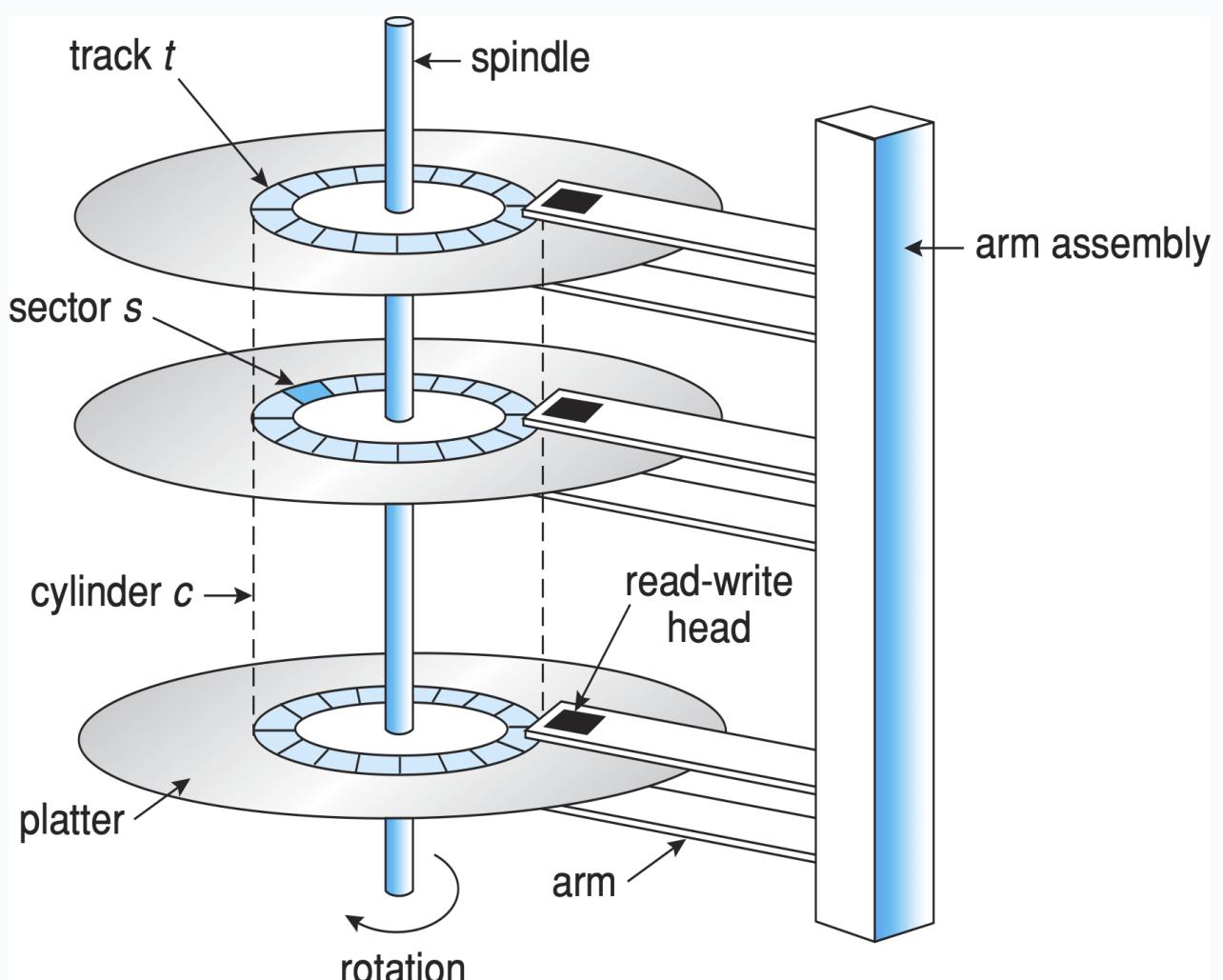
- **DMA**: Direct Memory Access
 - La CPU istruisce il disco sul compito da effettuare
 - Il DMA controller legge/scrive autonomamente i dati in memoria
- **Caching**: Il sistema operativo tiene in RAM **le porzioni di disco più lette** (quindi abbiamo delle zone già preventivamente copiate)

Organizzazione dei Dati nei Dischi

Dischi Magnetici (HDD)

In disco magnetico, i dati sono organizzati in **tracce concentriche e settori**. Abbiamo dei **dischi in parallelo**, con **due facciate** ognuna.

FIGURA (*Schema di un disco magnetico*)



Dischi a stato solido (SSD)

I dischi a stato solido invece sono **simili alle celle di memorie RAM**.

- Matrice di celle
-

Dati e Dischi

Definizione di Blocco

Definizione (*blocco*)

I dischi sono utilizzabili come un *vettore di blocchi*

- Blocchi di $512B - 8KB$

FIGURA (*L'idea dei blocchi*)

Blocco 0	Blocco 1	Blocco 2	Blocco 3	Blocco 4
Blocco 5	Blocco 6	Blocco 7	Blocco 8	Blocco 9
			

Definizione (*File System, cenno*)

Il *File System* permette di organizzare questi blocchi per avere

- *File* di grandezza variabile (quindi la sostanza)
 - *Organizzati* in un albero di cartelle (quindi la struttura)
Quindi possiamo definire il *file system* come un *algoritmo per rappresentare le cartelle in un disco*.
-

I File

Definizione. (*File*)

I *file* sono una *sequenza ordinata di bit* che contengono delle *informazioni*.

Hanno un *nome* e degli *attributi* (ovvero metadati):

- Identificativo nel SO
- Permessi
- Tempo di creazione, di ultimo accesso

I file sono *organizzati* in *direttori* o *cartelle* o *folder* o *directory*

- Possono essere create, modificate o cancellate *come i file* (vedremo che in realtà le cartelle non sono altro che dei "*file speciali*")
- A differenza dei file non contengono byte *ma altri direttori o file*

Operazioni sui File.

Sui file, un programma (tramite System Call del SO) può effettuare le *operazioni* di:

- Creazione
- Lettura
- Scrittura
- Cancellazione
- *Seek* (movimento del cursore) (ricordiamoci che l'accesso ai file è *sequenziale*)

Osservazione. (*Accesso sequenziale dei file*)

Le operazioni di lettura e scrittura sono sempre *sequenziali*. Il file viene letto/scritto byte per byte, tramite un *cursore*. E' possibile riposizionare il cursore tramite l'operazione di *seek*

I Direttori

Definizione. (*Directory/Direttori*)

Una *Directory* è un contenitore di nodi (file o altre Directory)

- Organizzazione tipicamente ad albero.
- Tipicamente è un grafo *senza cicli* (a meno che consideriamo i *link*, come faremo dopo)

Definizione. (*Partizioni*)

Un disco è diviso in una o più *partizioni*

Ogni partizione contiene un *albero di direttori*, in particolare

- Vi è un direttorio *radice*
- *Tutti* i file e direttori vi son contenuti

Operazioni con le cartelle.

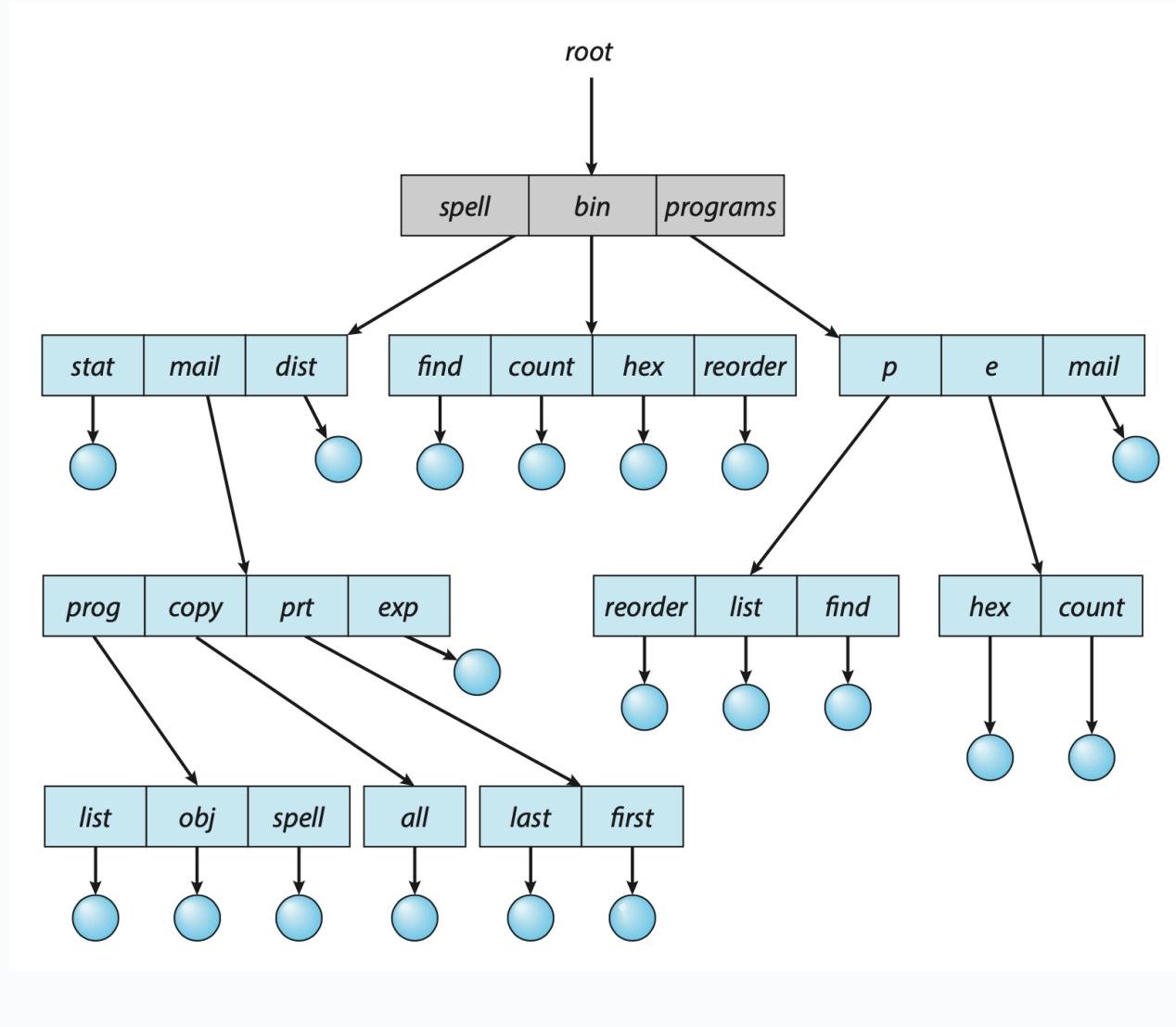
Operazioni sui direttori sono *simili a quelle su file*:

- Creazione
- Rimozione
- *Listing*
- *Renaming*

Ricordare! Il SO mette a disposizione delle *System Call* per queste operazioni.

- Esse sono a **basso livello**. Possono essere difficili da usare
- La libreria standard del C mette a disposizione delle **funzioni a più alto livello** (più facili da usare) che al loro interno **utilizzano le necessarie System Call** (quindi non c'è mai scampo da queste System Call!)
 - Ricordiamo che abbiamo **sempre** una situazione del tipo
 - **Hardware \leftrightarrow Sistema Operativo \leftrightarrow System Call \leftrightarrow Libreria \leftrightarrow Software**

FIGURA (*L'albero delle cartelle*)



Link e cicli

Problema (*i link*)

L'albero è l'organizzazione **più naturale**.

Tuttavia i **link** (1, 2) possono creare dei **cicli**

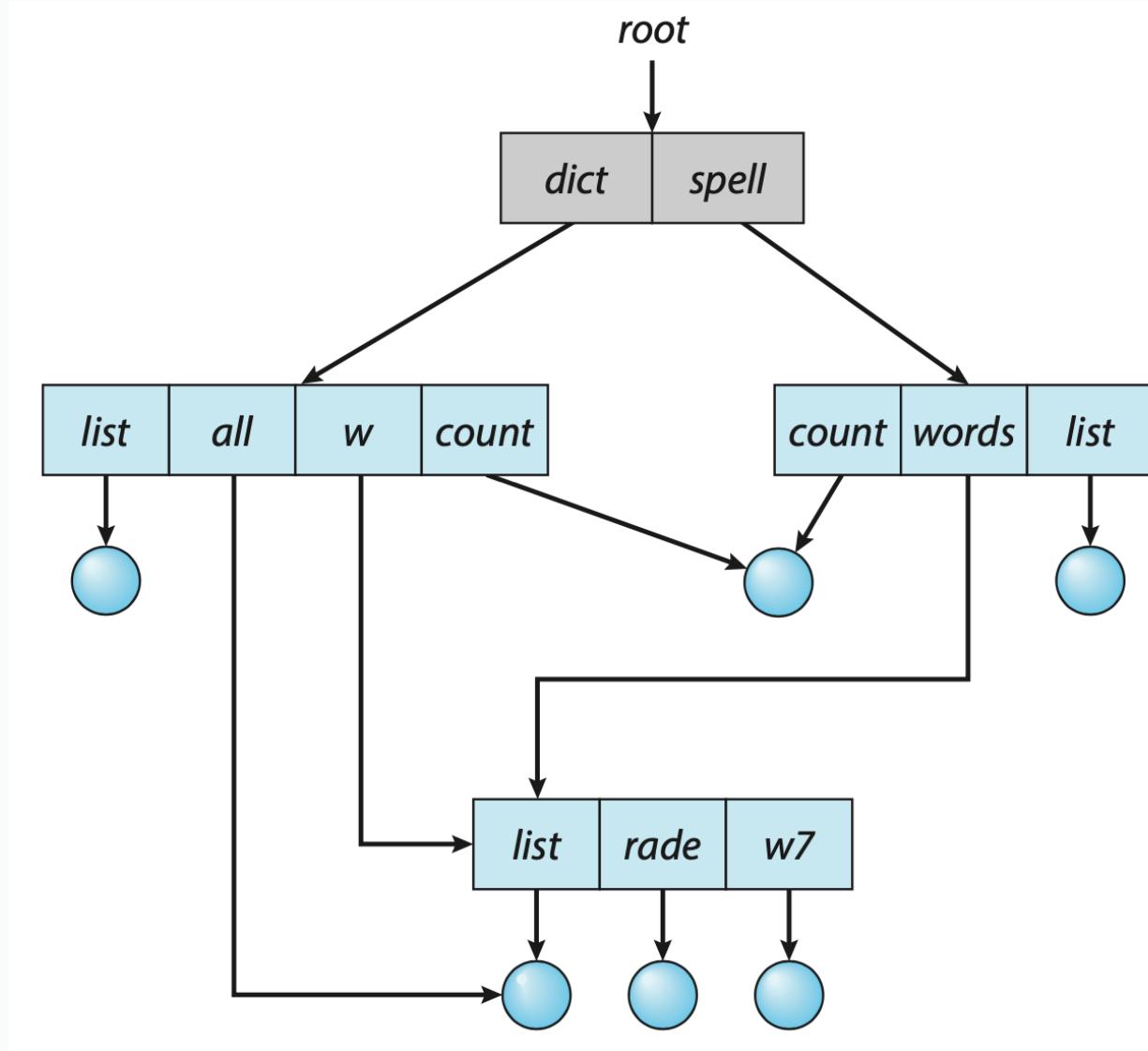
- Coi link, **cancellazione più complessa**
 - Sto cancellando il file originale o una copia?
 - Vedremo in fondo dopo

I cicli **complicano** molto la **gestione del File System**

- Immaginare un processo di **ricerca ricorsiva** in una cartella con cicli
 - Processo **potenzialmente infinito se non gestito correttamente!** (pericoloso)

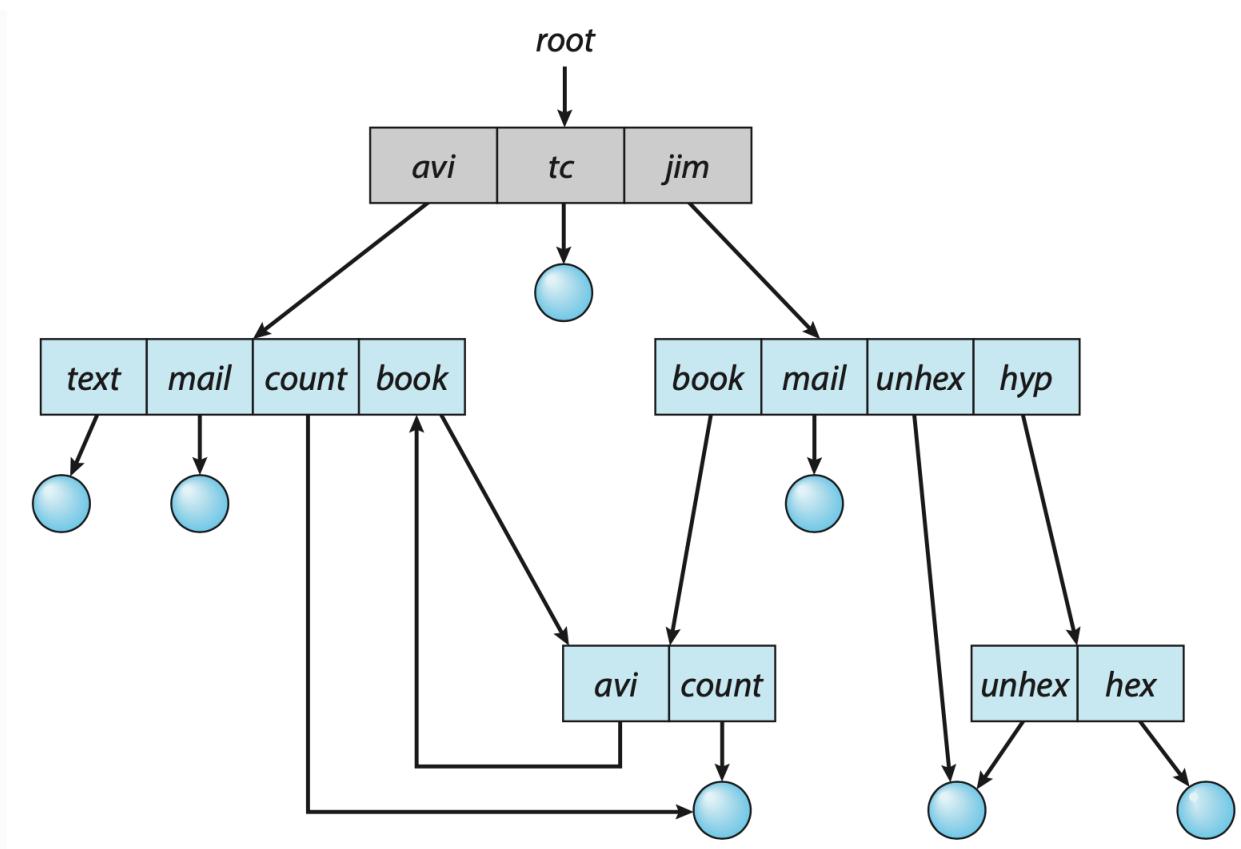
Osservazione 1. (*i link ai file vanno bene*)

I link a file non possono generare cicli. Infatti abbiamo una situazione del tipo



Osservazione 2. (*i link a direttori vanno gestiti*)

I link a direttori possono generare cicli:



Possibili soluzioni: mai visitare i link durante ricerca (questa è la soluzione comune); altrimenti avrei un *ciclo infinito*.

Allocazione dei blocchi

Richiamo. (*il concetto dei blocchi*)

I file system risiedono su disco

I dischi permettono letture e scritture a *blocchi*

- Tipicamente da *512B a 8KB*
- E' possibile *leggere/scrivere un blocco per volta, e interamente*

Blocco 0	Blocco 1	Blocco 2	Blocco 3	Blocco 4
Blocco 5	Blocco 6	Blocco 7	Blocco 8	Blocco 9
.....			

Accesso ai Blocchi

Il *driver* (1) del disco permettono di accedere a un blocco.

- Ricevono comandi del tipo:

```
read block 431 to memory address 0x5984
write block 126 from memory address 0x9163
```

Un *File System* mappa *accessi a file e direttori in comandi per il driver*

- Quindi abbiamo una situazione del tipo
- APP → System Call → (OS) → Kernel → File System → Driver → (I/O) Disco

Allocazione dei Blocchi: Definizione

Definizione (*allocazione dei blocchi*)

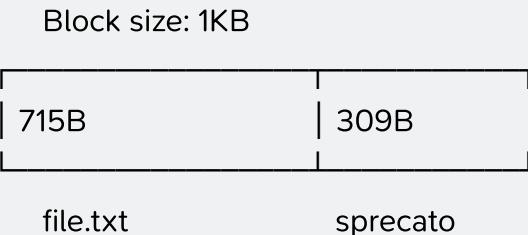
L'*allocazione* è il meccanismo con cui i blocchi sono allocati ai file.

- Ogni file *occupa 1 o più blocchi* (≥ 1)

Problema. (*frammentazione interna*)

Come conseguenza del meccanismo dell'allocazione dei blocchi abbiamo un noto problema, detto come "*frammentazione interna*": in sostanza è lo spreco intrinseco di capacità quando un file non ha dimensione multipla della grandezza dei blocchi. La parte sprecata del disco non può essere *mai* usato per altri file

FIGURA. (*Frammentazione interne*)



Metodi per l'Allocazione dei Blocchi

Adesso iniziamo a vedere dei metodi per l'*allocazione dei blocchi*, quindi degli *algoritmi specifici di File System*.

Allocazione Continua

Metodo. (*Allocazione Continua*)

Ogni file *occupa un insieme di blocchi contigui* (ovvero una dopo l'altra)

Vantaggi

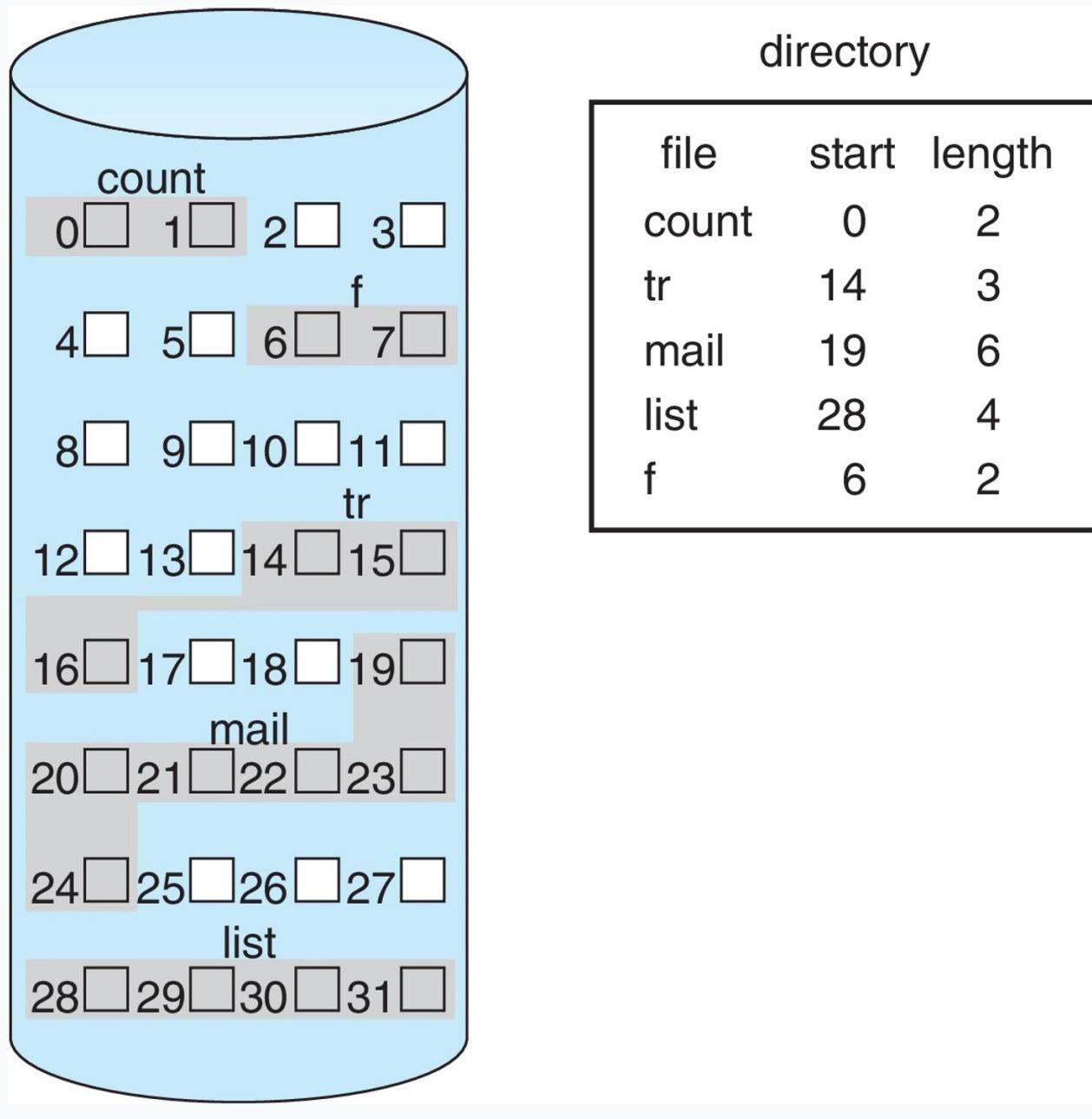
- *Semplice e veloce* (forse *troppo* semplice!)

- *Banale accedere* al byte N , visto che il file è memorizzato in *maniera contigua* sul disco
 - Ad esempio, per accedere ai byte *centrali* di un file basta andare al blocco *centrale*.
- *Pochi metadati* per file sono necessari
 - Solo *file*, *start* e *length*.

Svantaggi

- Crea la *Frammentazione Esterna*: rimangono *blocchi vuoti sparsi* per il disco, che *non possono essere utilizzati* che per file molto piccoli.
- Grave spreco! I buchi diventano *sempre* più grandi col *tempo*.
 - Una *soluzione storica* era quella di fare la c.d. "*deframmentazione del disco*" (defrag); comunque per fare un'operazione del genere ci si può mettere un sacco di tempo.

FIGURA. (*Allocazione continua*)



Allocazione Concatenata

Modello. (*allocazione concatenata*)

Ogni File è una *Linked List* di blocchi.

- Ogni blocco contiene il *numero di blocco successivo*; l'equivalente del puntatore **NEXT**
- L'ultimo blocco contiene un *numero speciale che indica la fine*; l'equivalente del puntatore nullo **NULL**

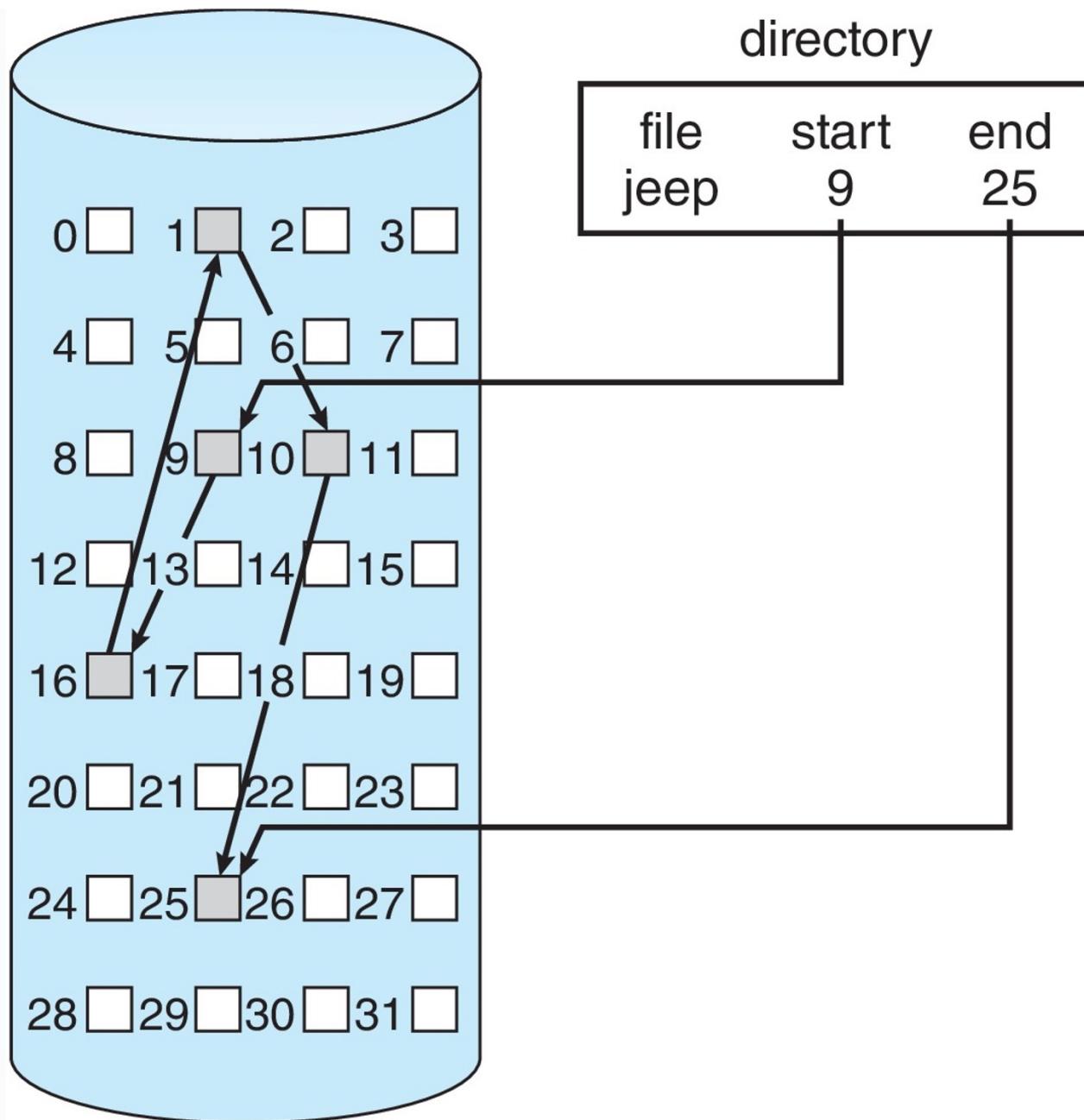
Vantaggi:

- No *Frammentazione Esterna*!
- *Tutti i blocchi sono usabili* per ogni file
- Ancora meno metadati necessarie: bastano solo "*file*" e "*start*" (volendo anche "*end*")

Svantaggi:

- Efficiente solo per *accesso sequenziale*
- Accedere ai byte *centrali* del file *richiede di scorrere tutta lista*, dalla *testa* del file.
 - Anche sono leggere il *puntatore* richiedere di leggere tutto il blocco
 - Ricorda: i dischi permettono di *leggere/scrivere un blocco per volta*
- *Poco affidabile!* Un blocco invalido invalida tutto il file
 - Problema per file grandi, che col tempo diventano sempre più soggetti a falle.
 - Sono come delle *candele* in serie: se si rompe una, il circuito non gira piùPer ovviare ad alcuni di questi problemi useremo una versione *ottimizzata* di questo modello. In particolare l'allocazione **FAT**.

FIGURA. (*Allocazione concatenata*)



Allocazione FAT (File Allocation Table)

Modello. (*Allocazione FAT*)

Come detto prima, questo è una *variante* dell'*allocazione concatenata*.

I *primi blocchi* del disco contengono una *tabella della FAT* ("File Allocation Table")

- E' una *Linked List* di blocchi
- Approccio *simile a Allocazione concatenata*
 - Ma la lista contenuta nei primi blocchi
 - Più veloce, la FAT può essere in cache
- Usato in Windows e MS-DOS coi File System *FAT* e *FAT32*

Vantaggi

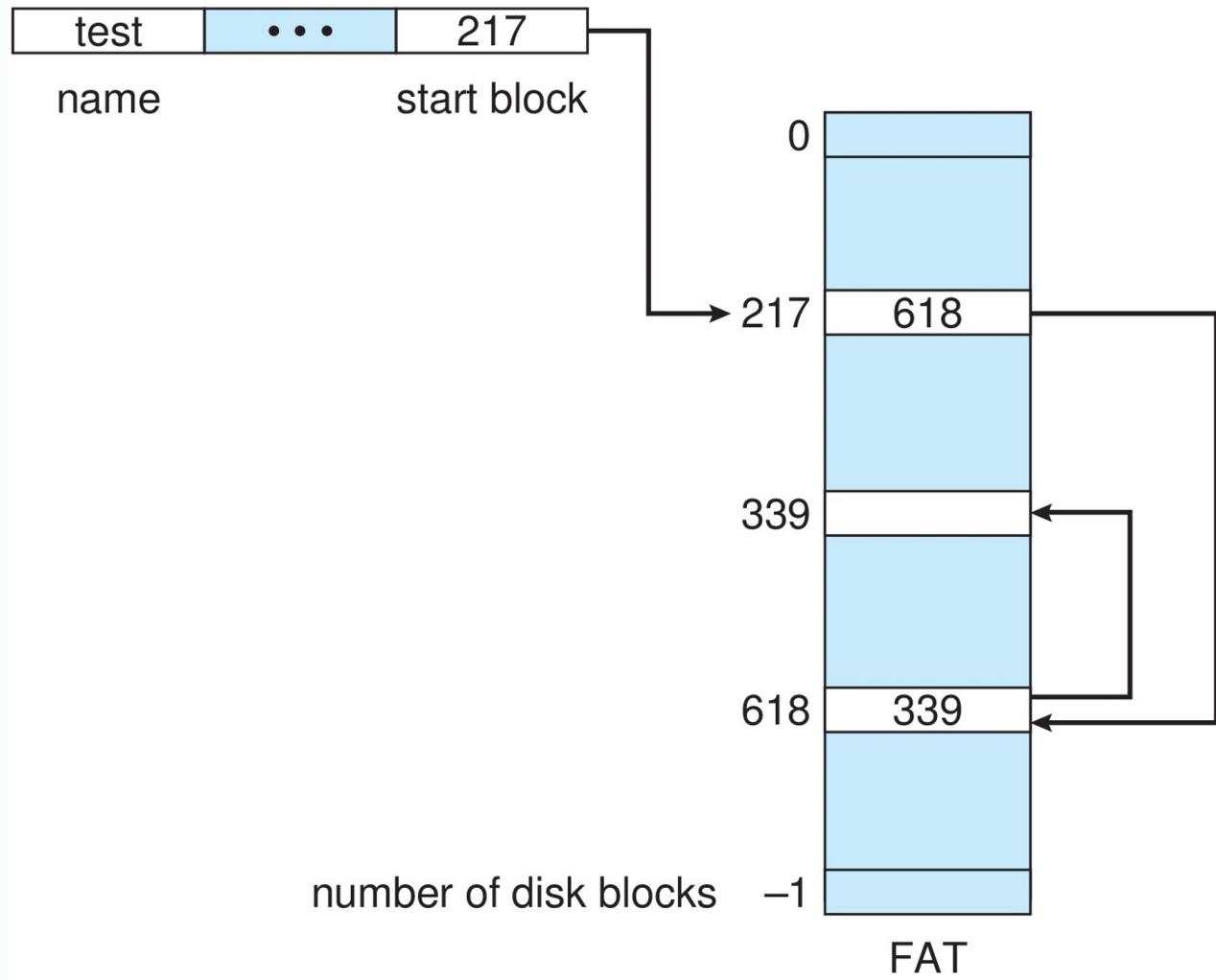
- La FAT è *cache-abile* (quindi ottimizzabile con pochi KB)
- Mi servono ancora meno metadati; basta aver salvato la *start block* della *FAT*.

Svantaggi

- Lento accedere a ultimi byte del file (come allocazione concatenata)
- Se perdo la FAT perdo tutto!
- La *FAT* diventa grossa per dischi grandi (difficilmente *scalabile*)

FIGURA. (*FAT*)

directory entry



Allocazione Indicizzata

Modello. (*Allocazione indicizzata*)

Ogni file ha un *blocco indice* che contiene i numeri di tutti i blocchi.

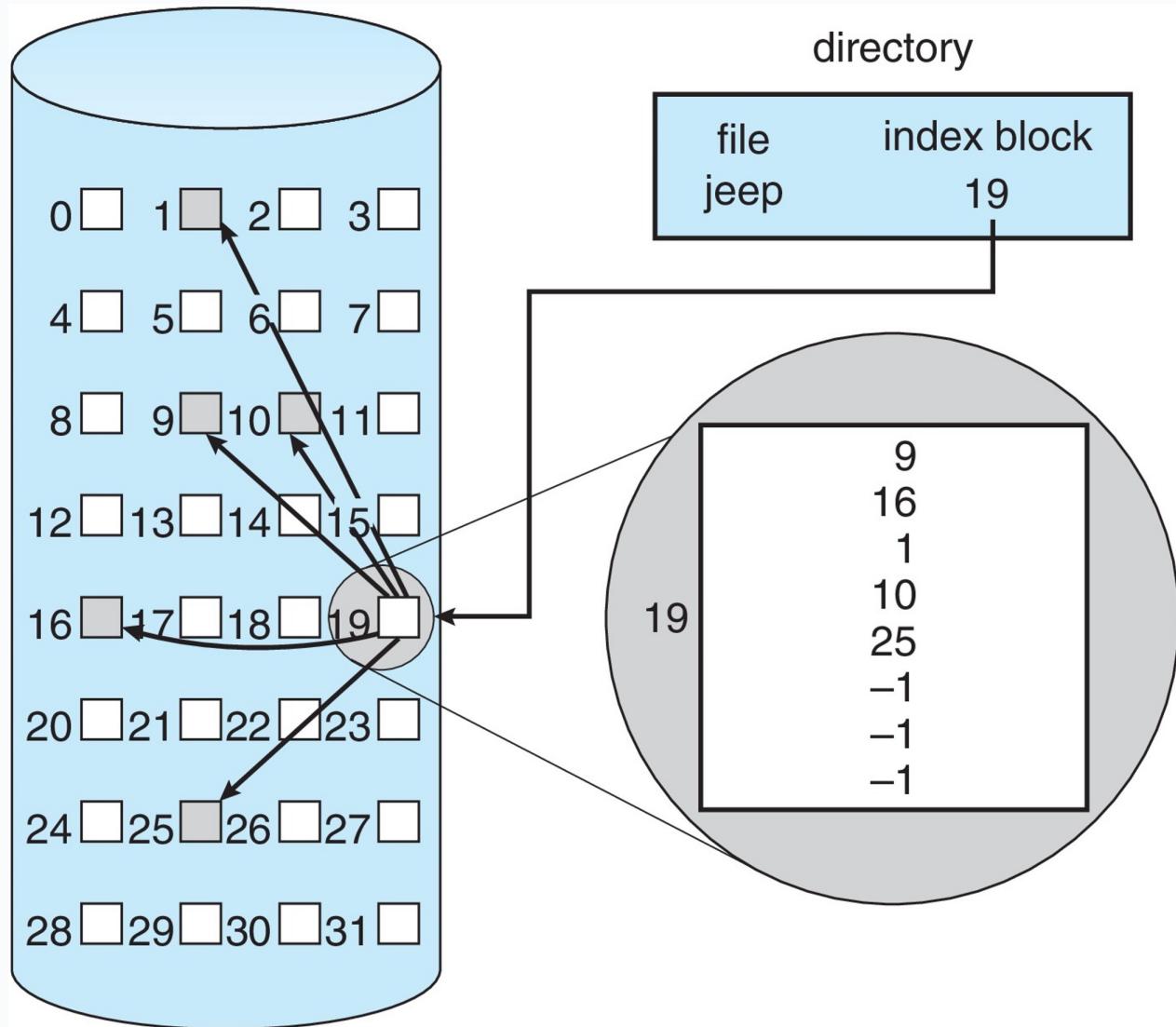
Vantaggi

- Accedere a un *byte arbitrario* è veloce
 - Basta leggere il *blocco indice* ed il *blocco desiderato*

Svantaggi

- Si sposta un blocco per file (oltre a quello del file)
- Se un file è troppo grande diventa impossibile

FIGURA. (Allocazione indicizzata)



Allocazione Combinata

Modello. (Allocazione combinata)

Utilizzata in *Linux*; considerata il *migliore compromesso*

Ogni *file o directory* ha una *struttura* detta "*inode*", che contiene le seguenti informazioni.

- Metadati e permessi del file/direttorio
- I numeri dei *primi 12 blocchi*
 - Alcuni inutilizzati, se file più piccolo; vengono salvati direttamente nelle inode
- *Puntatori indiretti*:
 - Numeri di blocchi i quali contengono a loro volta *una tabella*
 - Su *uno, due e tre livelli* (quindi abbiamo *indici ai dati, indici agli indici ai dati e indici agli indici agli indici ai dati*)

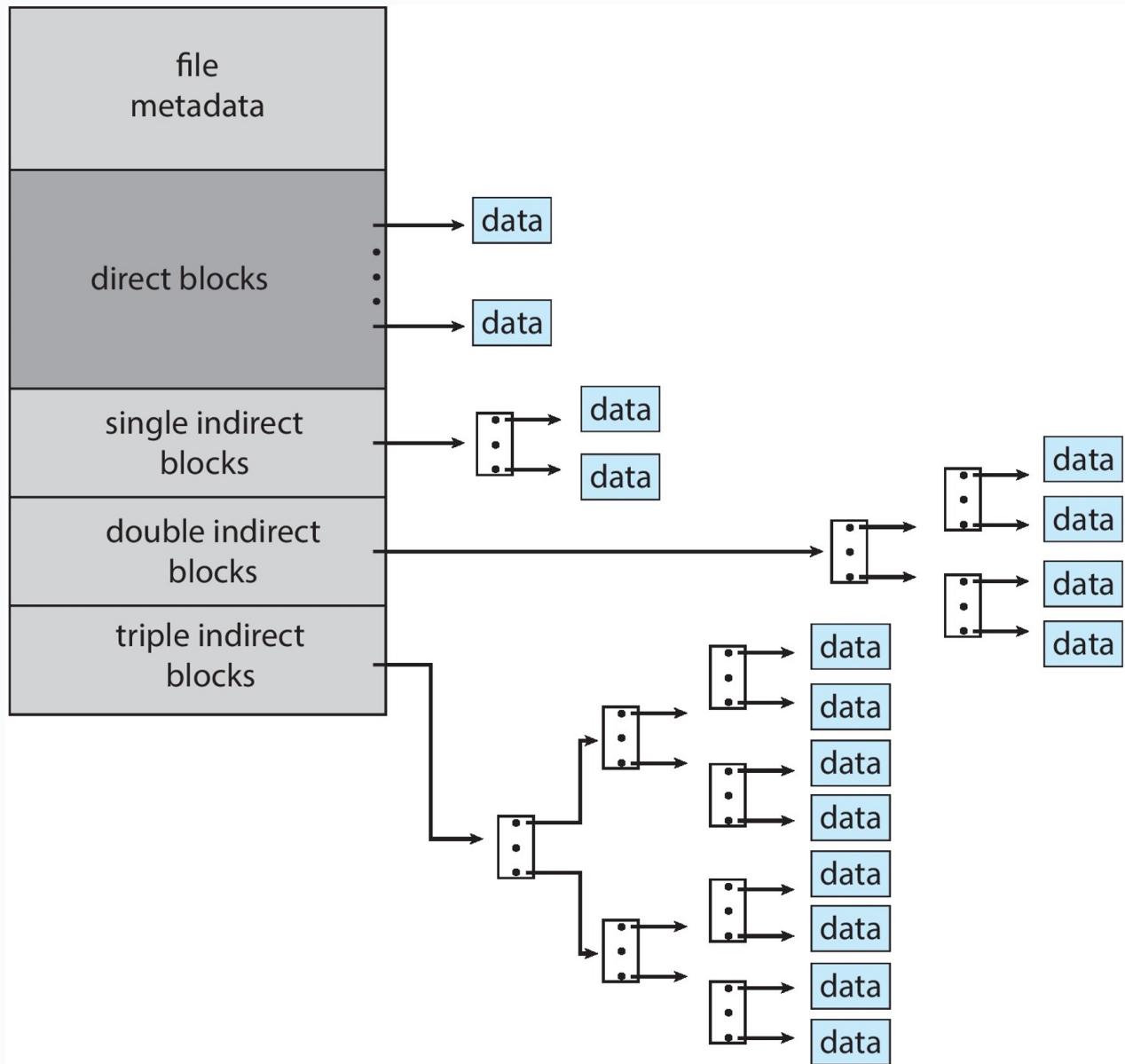
Vantaggi:

- Buon compromesso
- *No frammentazione esterna*
- Come *indexed* (indicizzata) per file piccoli
- Può *indicizzare file anche molto grandi*
- Per file di *medie dimensioni* ho l'*allocazione indicizzata* lo stesso

Svantaggi:

- Può richiedere di leggere più di un blocco per accedere a posizioni avanzate nel file (quindi abbiamo tempo di lettura *non costante*)
- C'è una *grandezza massima* per i file, di circa *4TB*.

FIGURA. (*Allocazione Combinata con gli inode*)



File system comuni

Adesso vediamo degli *algoritmi comuni* per i l'*allocazione dei blocchi*.

Esempi di File System

Esempi. (File System comuni)

Ogni OS si porta dietro i suoi File System

- **Unix:** UFS, FFS
- **Linux:** tantissimi.
 - **ext3** and **ext4** sono gli *standard di fatto*. Usano *allocazione Combinata*
- **Windows:**
 - FAT, FAT32 basati su FAT
 - NTFS: con indirizzamento ad albero
- **Apple:** HFS, HFS+, APFS (Hierarchical File System)
- **File System distribuiti per Big Data:** GoogleFS, HDFS, CEPH. Sono recenti e vengono utilizzati negli *ambienti professionali*. Li approfondiremo alla fine.

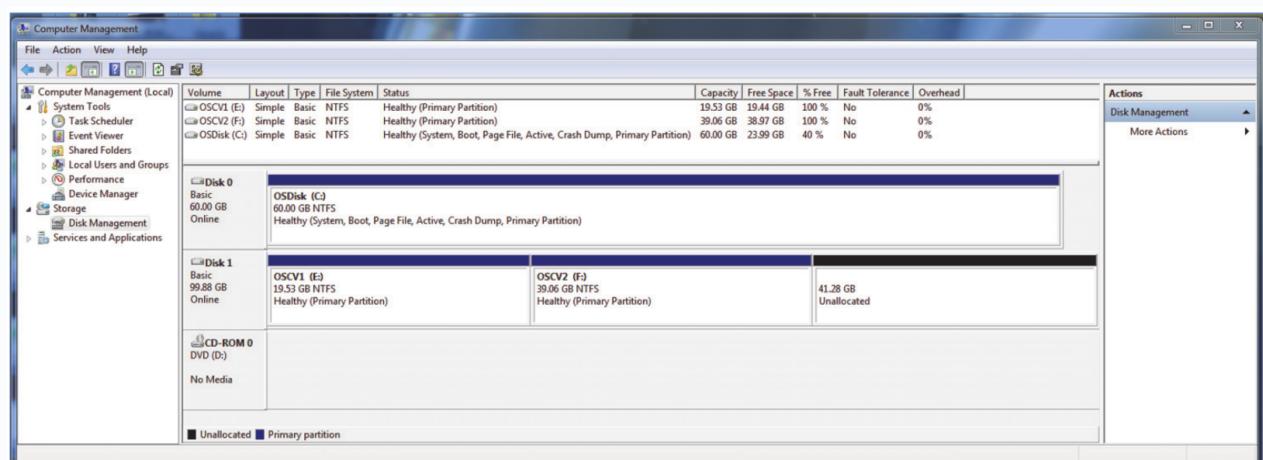
Tabelle delle partizioni

Definizione. (Partizione dei Blocchi)

Oltre ai FS, esistono degli standard per *partizionare i dischi in più partizioni* (quindi dare una *grandezza fissa* per i dischi).

- **Master boot record (MBR):** metodo classico. I primi blocchi del disco *indicano le partizioni*
 - Contiene anche il *codice iniziale* per avviare l'elaboratore
 - Massimo dischi da **2 TB e 4 partizioni** (limitata)
- **GUID Partition Table (GPT)** moderno, parte dello Unified Extensible Firmware Interface (**UEFI**) standard
 - Supera le limitazioni di MBR

FIGURA. (Management delle partizioni su Windows)



Esempio. (Linux)

Su Linux si opera da riga di comando con **fdisk** o **parted** o con l'utilità grafica **gparted**

Domande

La CPU accede al disco:

- Direttamente
- Attraverso un'interfaccia
- Attraverso la memoria

Attraverso la cache

Risposta: Attraverso un'interfaccia

I link a file possono generare cicli:

- Vero
- Falso

Risposta: Falso

I link a directory possono generare cicli:

- Vero
- Falso

Risposta: Vero

Un disco ha blocchi grandi 4KB (4096B). Un file di grandezza 510B. Quanto spazio viene sprecato a causa della frammentazione **interna**:

- 4606B
- 3586B
- Impossibile da stabilire

Risposta: 3586B

Un disco ha blocchi grandi 4KB (4096B). Un file di grandezza 510B. Quanto spazio viene sprecato a causa della frammentazione **esterna**:

- 4606B
- 3586B
- Impossibile da stabilire

Risposta: Impossibile da stabilire

Con l'allocazione concatenata si ovvia al problema della:

- Frammentazione interna
- Frammentazione esterna

Risposta: Frammentazione esterna

In Linux, il FS usa lo schema:

- FAT
- Allocazione concatenata
- Allocazione combinata
- Allocazione continua

Risposta: Allocazione combinata

Quali tra questi é un formato per le tabelle delle partizioni:

- Ext
- MBR
- FAT
- NTFS

Risposta: MBR

u4-s2-file-system-linux

Sistemi Operativi

Unità 4: Il File System

I file system in Linux

Argomenti

1. File System in Linux
 2. Permessi in Linux
 3. Comandi Bash per i dischi
-

File System in Linux

Richiami Storici

Richiamo. (*Storia del Linux*)

Nel mondo Unix/Linux, esistono molti file system.

- Il primo è stato lo Unix File System (UFS)
- Da esso si sono gli Extended File System (**ext**) per Linux
- Ora siamo alla versione **ext4**
Come visto, basato sul *concetto di inode* (1) che rappresenta *un file o un directory*

Virtual File System (VFS)

Definizione. (*Virtual File System*)

Si possono usare *diversi file system*.

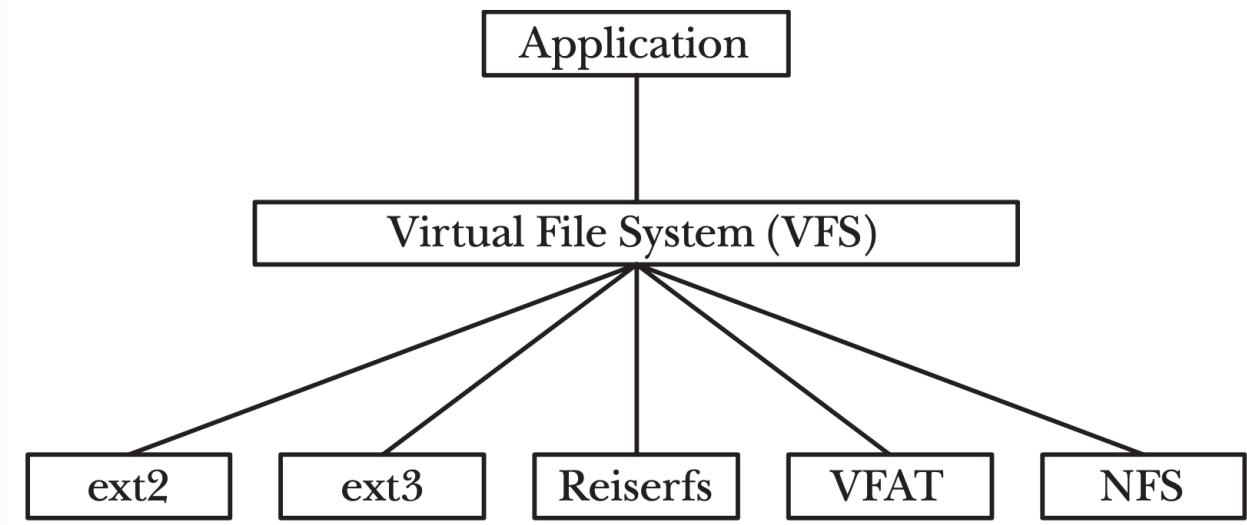
Devono implementare l'interfaccia standard *Virtual File System (VFS)*

- Ovvero permettano di effettuare alcune *funzioni fondamentali* e di rendere *generico* il kernel Linux:

C
`open(), read(), write(), lseek(), close(), truncate(), stat(),
mount(), umount(), mmap(), mkdir(), link(), unlink(), symlink(), rename()`

- Quindi si ha una situazione del tipo *Applicazione → VFS → File System → ... → Hardware*

FIGURA. (*L'idea del VFS*)



Montare File su Linux

Richiamo. (*Cartella root*) & **Definizione.** (*Operazione di montaggio dischi*)

Su Linux, tutti i file da ogni disco sono sotto un unico albero di cartelle, detto "*root*" (1)

- Che nasce da 1; tutto è figlio di questa cartella
- Dischi aggiuntivi vengono *montati* come sotto alberi di 1

Operazione. (*Montare dischi*)

Per *montare* un *FS* si usa il comando

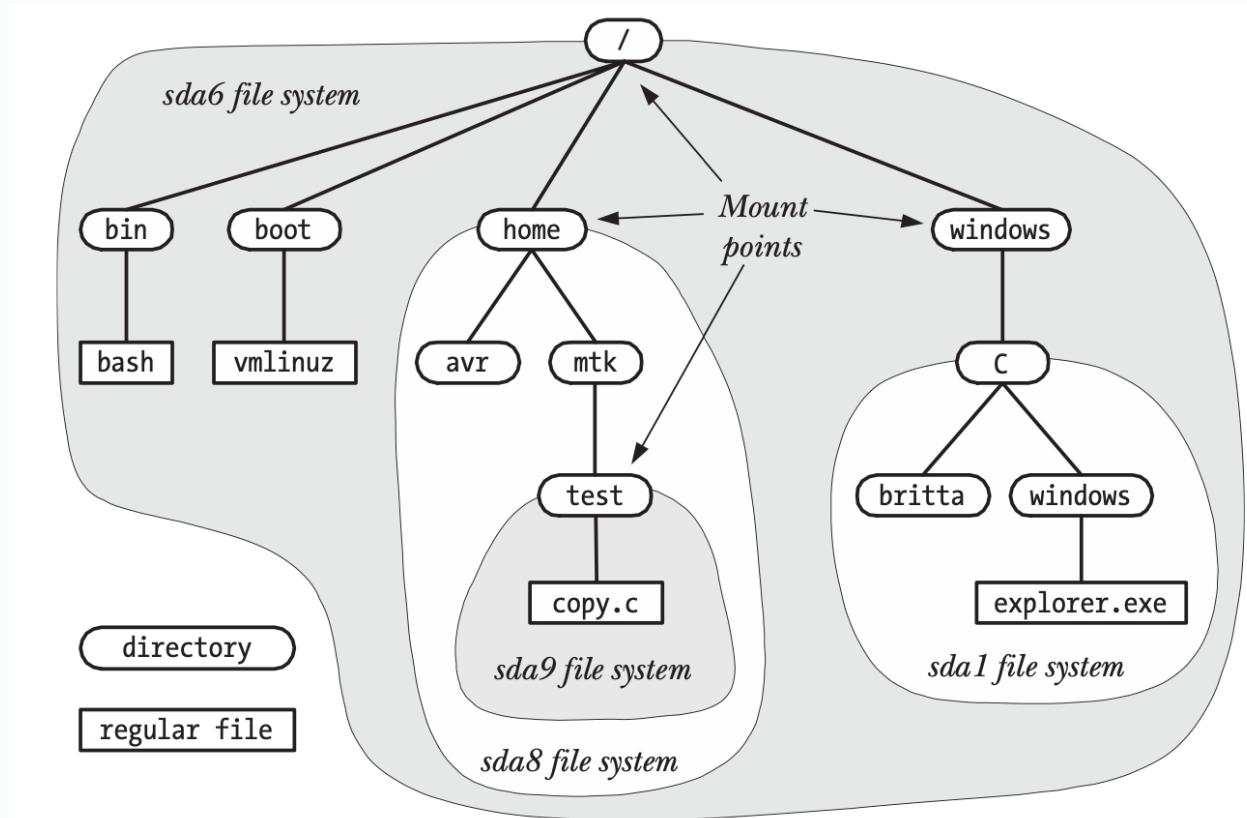
mount device directory

Operazione. (*Vedere i file montati*)

Per vedere il contenuto di un disco, esso va *montato*. Per vedere i dischi montati si usa lo stesso comando, omettendo tutti gli argomenti.

```
$ mount
/dev/sda6 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,mode=0620,gid=5)
/dev/sda8 on /home type ext3 (rw,acl,user_xattr)
/dev/sda1 on /windows/C type vfat (rw,noexec,nosuid,nodev)
/dev/sda9 on /home/mtk/test type reiserfs (rw)
```

FIGURA. (Esempio di una gerarchia di FS montati)



Definizione. (File di configurazione **fstab**)

In un sistema Linux, i *dischi che vengono montati automaticamente all'avvio* sono specificati nel file **/etc/fstab**

- Contiene una riga per ogni disco
- Formato **<disk> <mount point> <type> <options> <dump> <pass>**
 - Ovvero "disco X" sulla "cartella Y" del "tipo Z" con certe cose...
- Fondamentale saperlo: se questo file ha certi casini, l'elaboratore non sarà più in grado di partire!

Esempio:

SHELL

```
/dev/sda1 / ext4 errors=remount-ro 0 1
/dev/hda1 /media/hda1 vfat defaults,utf8,umask=007,gid=46 0 0
```

Inode, Partizione e Blocco Dato

Nota: Si trovano in sezione A - OK

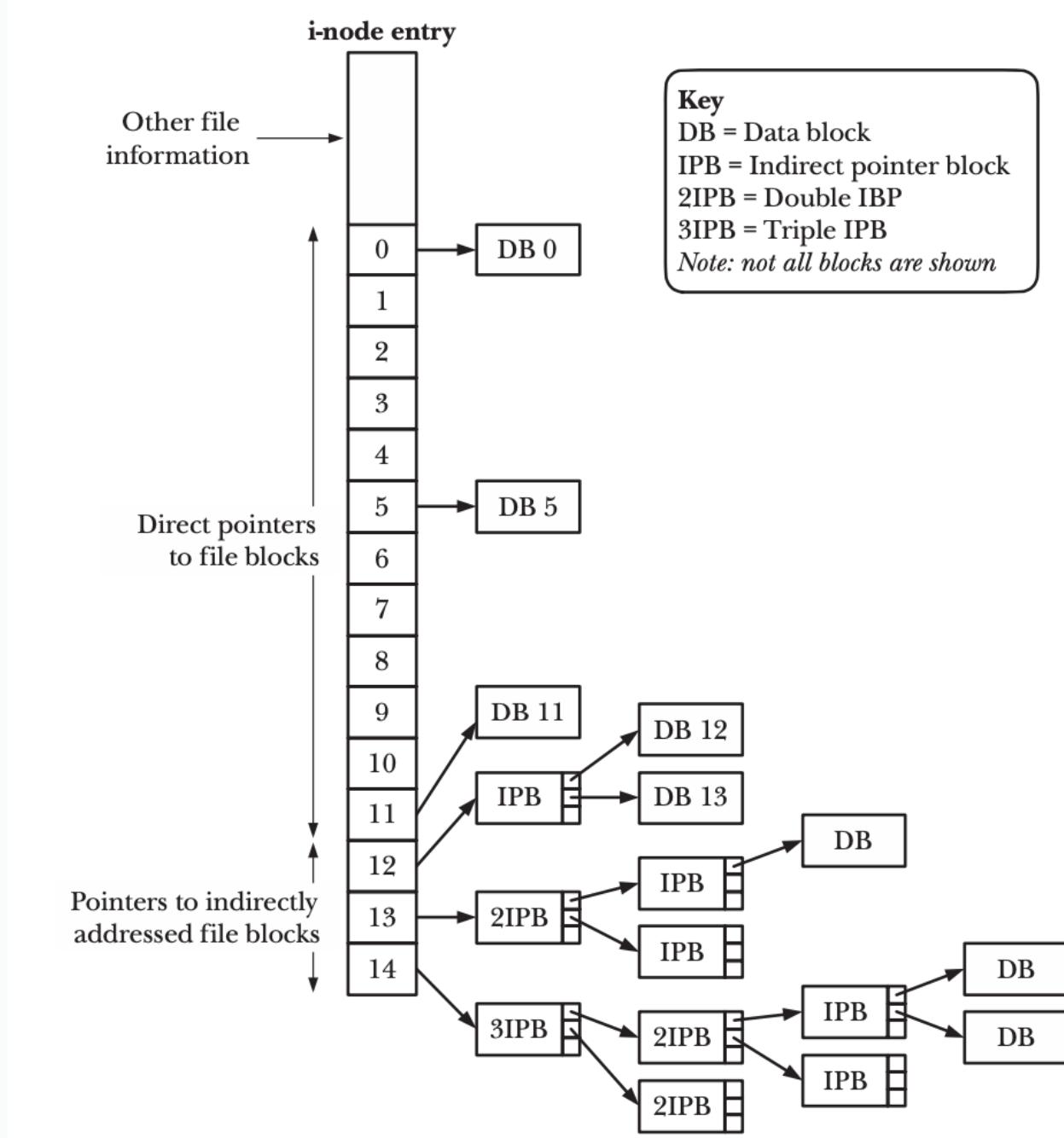
Inode

Definizione. (*Inode*)

Rappresentano un *file/cartella* (1). Memorizzati in una tabella nei primi blocchi

- Ogni inode è una struttura di pochi byte (ha una dimensione fissa)
- Identificati da *inode number*
 - **ATTENZIONE!** Ciò vuol dire che l'inode *non* contiene il *nome del file*! Per trovare ciò bisogna reperirlo con altre funzioni.
- Sono *in numero finito e immutabile*
 - Non si possono memorizzare infiniti file minuscoli

FIGURA. (*Struttura dell'i-node*)



Layout di un disco, definizione di partizione

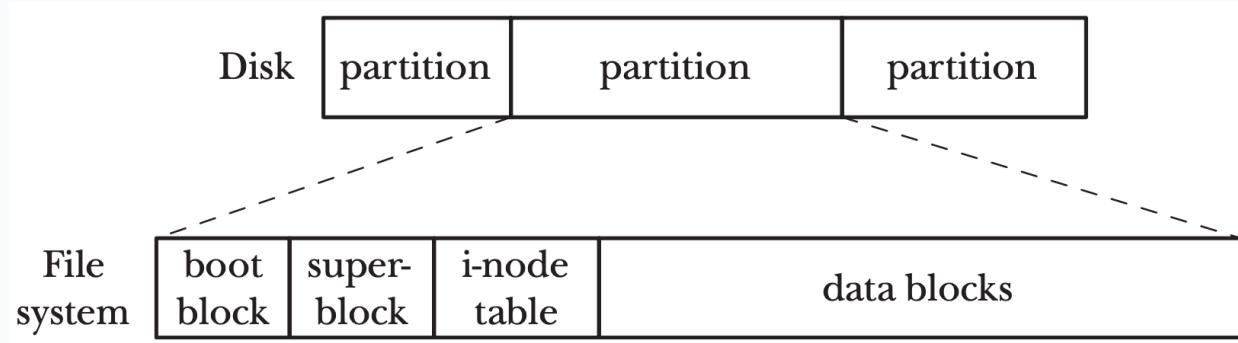
Definizione. (*Partizione di un disco*)

Un disco è diviso in *partizione* (1)

Ogni partizione contiene

- Informazioni di controllo; tra cui informazioni per far *partire* il *sistema operativo*, dei *metadati* e così via...
- *Tabella degli inode*
- Blocchi di dato (ovvero quelli in cui mettiamo i *dati*)

FIGURA. (*L'idea della partizione*)



Tipi di Blocco Dato

Definizione. (*Data block, Directory block*)

I blocchi di dato sono di due tipi:

- *Data Block*: hanno il contenuto di un file. *Dati binari*
- *Directory Block*: hanno il contenuto di una cartella. *Lista di coppie (nome, inode)*

FIGURA. (*Esempio generale di un disco*)



File System in Linux (ulteriori dettagli)

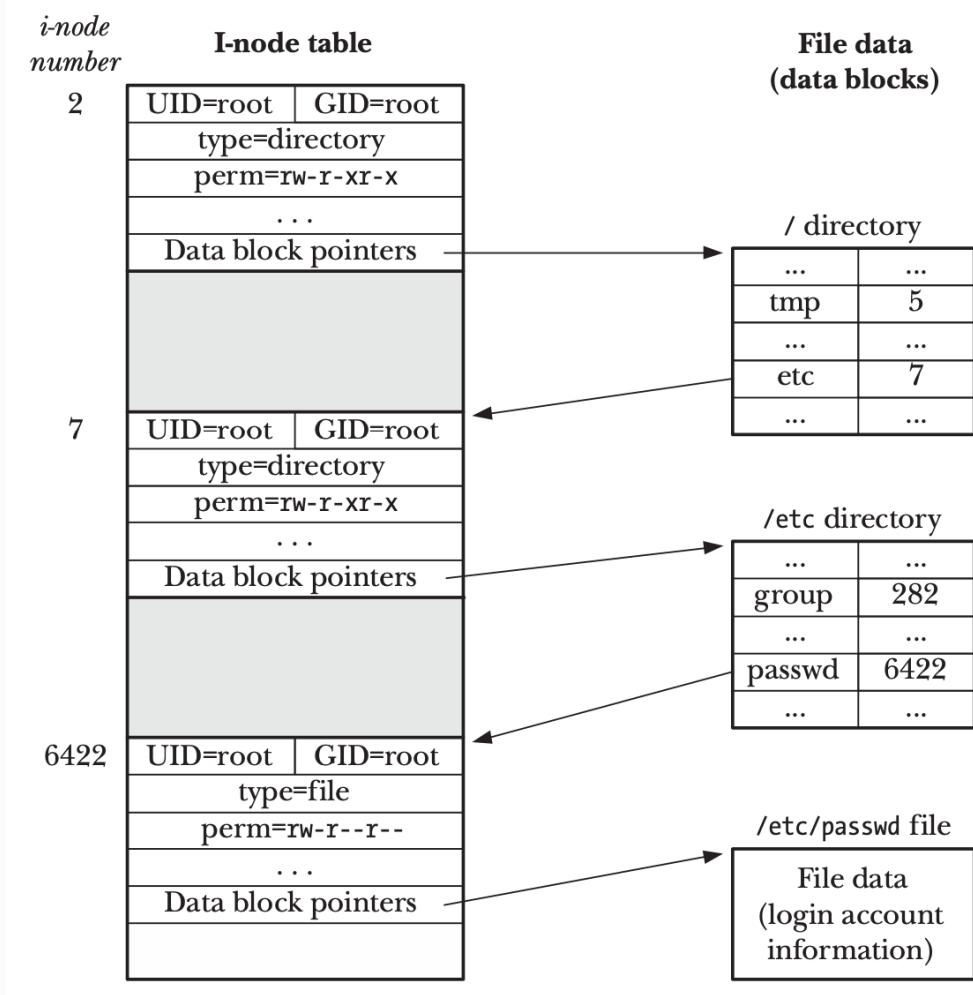
Definizione di Directory per Linux

Corollario. (*Definizione del directory su Linux*)

Ogni directory é un inode (1).

- Occupa almeno un blocco contenente la tabella dei nodi che contiene Pertanto possiamo considerarla *come un file "speciale"*:
- Il cui contenuto non é un insieme di byte
- Ma una *lista di coppie (nome, inode)*

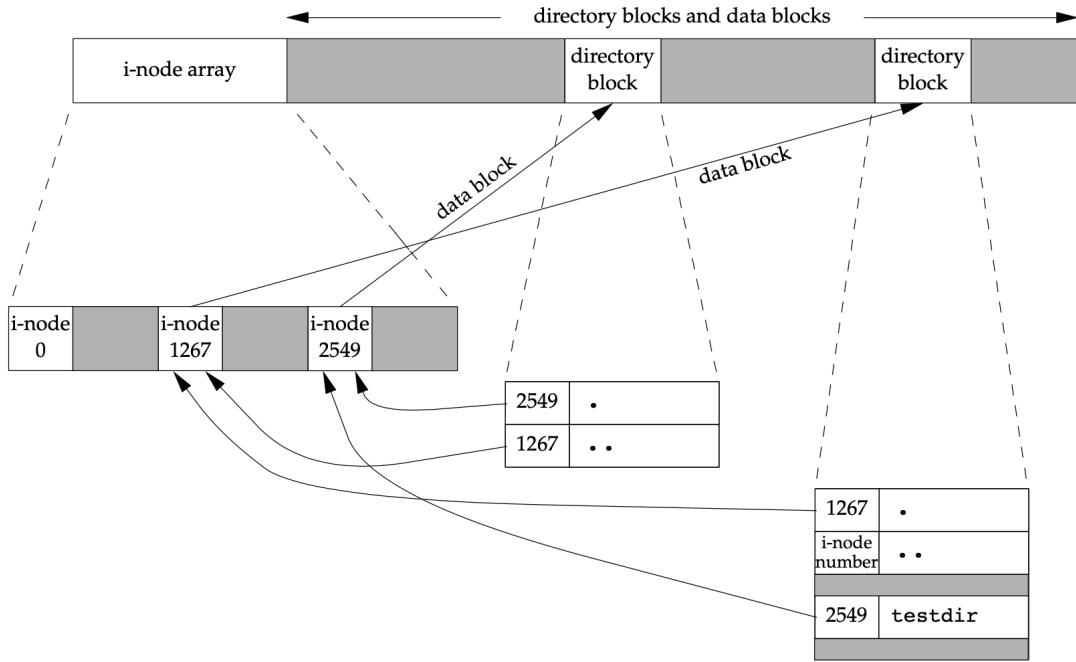
Esempio.



Dalla figura vediamo che

- inode 2 é la directory **/**
 - Contiene **etc**: inode 7
- inode 7 é la directory **/etc**
 - Contiene **passwd**: inode 6442
- inode 6442 é il file **/etc/passwd**
 - Il contenuto é in un blocco dati

Esempio.



- La cartella **d** (inode 1267) contiene la cartella **testdir** 2549 che é vuota

Permessi in Linux (ulteriori dettagli)

Utenti e Gruppi

In parte visto all'inizio del corso ([Utenti e Permessi Linux](#))

In Linux, esistono:

- **Utenti:** account che possono utilizzare il sistema, creare processi, accedere a file
- **Gruppi:** insiemi di utenti. Ogni utente ha un:
 - **Gruppo principale:** solo uno (= 1)
 - **Gruppi secondari:** senza limiti di numero (≥ 0)

Ogni utente e gruppo è identificato da un *nome* e da un *id* numerico

Esempio:

- **Utenti:** **martino**, **luca**, **paolo**
- **Gruppo principale:**
 - **martino** → **docenti**
 - **luca** → **studenti**
 - **paolo** → **studenti**
- **Gruppo secondario:**
 - **martino luca** → **sistemioperativi**
 - **martino paolo** → **reti**

Utente root

Definizione. (*Utente root*)

L'utente *root* esiste su tutti i sistemi (1)

- Ha *id* 0
- Bypassa tutti i controlli sui permessi (ovvero fa ciò che vuole, nei limiti del software)

Nota: *root* è un utente con privilegi illimitati. NON è né parte del kernel, né il suo codice esegue in modalità kernel.

Errore comune: dire che l'utente *root* esegue processi in kernel-mode! Non può mica accedere alla *memoria* o ai *dispositivi I/O*!

File per utenti e gruppi

File di configurazione per utenti e gruppi.

Le informazioni su utenti e gruppi attivi salvate in file di configurazione accessibili *solo a root*:

- **/etc/passwd**: lista di utenti e dettagli (*ID*, *home directory*)
 - Nota: Una volta c'erano le *password* dentro, adesso non più per ovvi motivi di sicurezza (nonostante il nome ci dia un'idea del genere).
- **/etc/shadow**: password cifrate con *hash*.
- **/etc/group**: lista di *gruppi* e dei rispettivi componenti

Permessi per i File o Cartelle

Definizione. (*Utente e gruppo proprietario*)

Ogni file/cartella ha uno e uno solo utente *proprietario*, e uno e uno solo *gruppo proprietario* (1).

E' possibile separare i permessi per classi di utenti.

Ovvero si specificano permessi separatamente per:

- **Utente proprietario**: si applica quando il proprietario tenta di fare accesso
- **Gruppo proprietario**: si applica quando un utente del gruppo proprietario accede
 - Nota: il gruppo proprietario non è necessariamente il **Gruppo principale** del proprietario, ma può essere un **Gruppo secondario**
- **Altri**: tutti gli altri utenti

Permessi di base

Definizione. (*Permessi base*)

Richiamiamo i *permessi base* (1, 2, 3), tenendo conto del fatto che le *cartelle* non sono

altro che dei *file contenenti* delle tabelle di i-node.

- **Lettura:** per i file, *leggere il contenuto*. Per le cartelle, elencare *nodi contenuti*
- **Scrittura:** per i file, *scrivere il contenuto*. Per le cartelle, *aggiungere/rimuovere nodi contenuti*
- **Esecuzione:** per i file, *eseguirli*. Per le cartelle, attraversarle, ovvero *accedere a file (o sottocartelle) contenuti*.
 - Nota: Per le cartelle, attraversare è diverso da listare (permesso lettura)!

Nota: non sono i permessi su un file a determinare se esso può essere *cancellato*, ma sono i *permessi sulla directory* che lo contiene a farlo.

Permessi speciali

Definizione. (*Permessi speciali*)

Oltre i 3 permessi di base, esistono altri tre permessi speciali (o flag) che si possono applicare a file/cartelle. Essi sono "*set user ID (suid)*", "*set group ID (guid)*" e lo "*sticky bit*".

- **set user ID (suid):** per i *file*, se eseguito, il processo è eseguito coi privilegi di *utente proprietario*, non di esecutore. Per le *cartelle, non ha effetto*.
 - **Utilizzo:** sui PC, i comandi di sistema (e.g., **reboot**) hanno il **suid**, per permettere riavvio senza chiedere password. Sui server, di solito no!
- **set group ID (guid):** per i *file*, se eseguito, il processo è eseguito coi privilegi di *gruppo proprietario*, non di esecutore; questo di solito è *inutile*. Per le *cartelle*, i *file creati hanno il gruppo della cartella* e non il gruppo principale del creatore (che è azione di default); questo è l'uso *principale* del guid.
 - **Utilizzo:** quando si creano *cartelle condivise* tra utenti che appartengono a un gruppo creato ad hoc.
- **sticky bit:** per i *file*, non ha *più effetto*. Per le *cartelle*, i file in essa contenuti possono essere *cancellati e spostati solamente dagli utenti che ne sono proprietari*, o dall'*utente proprietario della cartella*.
 - **Utilizzo:** nelle cartelle **/tmp** e **/var/tmp** tutti gli utenti devono poter creare e modificare dei file. Nessuno eccetto il *superuser* (o il proprietario) deve poter rimuovere o spostare file temporanei di altri utent

Esempio-Esercizio. Per un progetto si crea il gruppo **progettoSysOp**, che contiene 3 utenti. Si crea la cartella condivisa **/share/progetto** e la si assegna al gruppo **progettoSysOp**

Esempi dei Permessi Speciali

SUID. Abbiamo il comando **reboot** in **\bin\reboot** e il suo codice sorgente è qualcosa del tipo

```
main()
{
    ...
    reboot();
    ...
}
```

dove **reboot()**; è una *System Call*, che può essere eseguita *soltanto* da root. Vediamo il comportamento *default* e il comportamento col *suid*.

- *Default*

SHELL

```
DINO:$\bin\reboot # → rwx r-x r-x
> Fallisce!
```

- *Con suid*

SHELL

```
PAOLO:$\bin\reboot # → swx r-x r-x
> Ok! Il PC si riavvia
```

GUID. Il nostro computer contiene i seguenti utenti e gruppi:

- *Utenti*: MARTINO, LUCA
- *Gruppi*: DOCENTI, SISOP, STUDENTI

Denotiamo " \Rightarrow " per "X ha il gruppo primario Y" e " \rightarrow " per "X appartiene al gruppo secondario Y".

Abbiamo le appartenenze ai gruppi come:

- MARTINO \Rightarrow DOCENTI, MARTINO \rightarrow SISOP, LUCA \Rightarrow STUDENTI

Voglio creare una cartella comune chiamata **/SHARED**.

Dentro questa cartella il prof. MARTINO ci inserisce una sottocartella

/SHARED/PROGETTO, con appartenenza **MARTINO:SISOP**.

Adesso vediamo i comportamenti di questa sottocartella.

- *Default* (caso **rwx rwx ---**)

SHELL

MARTINO:\$ `mkdir /SHARED/PROGETTO/DATI`
> OK! File creato con permessi rwx rwx --- appartenente a MARTINO:DOCENTI
LUCA:\$... # ci fa qualcosa in nella cartella ./DATI
> No! Non ha i permessi sufficienti
> Per ovviare a questo bisogna usare il comando chgrp, che è molto scomodo da fare. Quindi modiflico il comportamento default come segue

- Con *GUID* (caso **rwx rws ---**)

SHELL

MARTINO:\$ `mkdir /SHARED/PROGETTO/DATI`
> OK! File creato con permessi rwx rwx --- appartenente a MARTINO:SISOP
LUCA:\$... # ci fa qualcosa in nella cartella ./DATI
> Ok! Va bene

STICKY-BIT. Prendiamo la cartella **/tmp**

- Default (**rwx rwx rwx**)

SHELL

MARTINO:\$ `touch /TMP/F1 # → MARTINO:DOCENTI; RWX --- ---`
LUCA:\$ `rm /TMP/F1`
> Comando eseguito con successo! Ma non dovrebbe essere così. Ognuno può fare ciò che vuole con i **file** degli altri.

- Con *sticky bit* (**rwT rwx rwx**)

SHELL

MARTINO:\$ `touch /TMP/F1 # → MARTINO:DOCENTI; RWX --- ---`
LUCA:\$ `rm /TMP/F1`
> Eh no! Non può, cicce.

Permessi dei Link

Caso Symlink (caso simbolico).

Non hanno permessi propri, ma *ereditano i permessi del file/cartella linkato*.

Nota: la loro creazione/distruzione resa possibile dai permessi della cartella in cui si trovano

Esempio:

SHELL

```
$ ls /lib  
...  
lrwxrwxrwx 1 root root 16 feb 24 2020 sendmail → ../sbin/sendmail*
```

...

Caso Hard-link.

La modifica dei permessi su un hard link, affligge anche il file originario; infatti con un *hard link* stiamo creando un *file* che punta allo *stesso* ed *identico* inode (1).

Motivazione:

- Un hard link non è altro che un riferimento allo stesso *inode* con un nome differente
 - e/o in una cartella differente
- File originario e suo hard link hanno la *stessa importanza*
 - Il file originale è *indistinguibile da un suo hardlink*
- I permessi di un file sono memorizzati *nel suo inode*

Rappresentazione dei Permessi in Linux

Esistono due notazioni per indicare i permessi di un file/cartella in Linux.

Rappresentazione simbolica.

Usata da **ls -l** e la più diffusa

Il primo carattere indica il tipo di file o directory elencata, e non rappresenta propriamente un permesso:

- **-**: file regolare
- **d**: directory
- **b**: dispositivo a blocchi
- **c**: dispositivo a caratteri
- **l**: collegamento simbolico
- **p**: named pipe
- **s**: socket in dominio Unix

Dopodiché abbiamo altri *9 caratteri*, che rappresentano i *permessi di lettura, scrittura ed esecuzione* per l'*utente proprietario*, il *gruppo proprietario* e *tutti gli altri*.

Inoltre permessi speciali vengono *aggiunti* a questa notazione sostituendo alcune lettere degli ultimi nove caratteri, come segue.

- **s** al posto del primo **x** per il SUID
- **s** al posto del secondo **x** per il GID
- **t** al posto del terzo **x** per lo sticky bit

FIGURA. (Schema generale della rappresentazione simbolica)

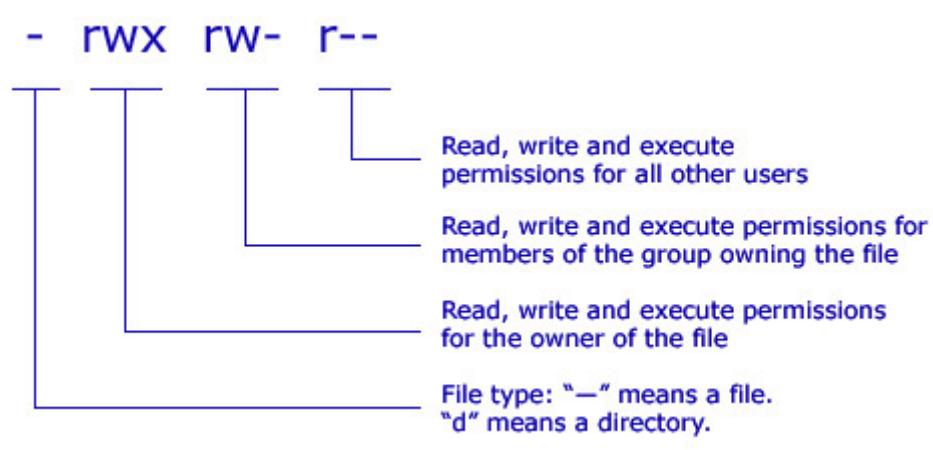
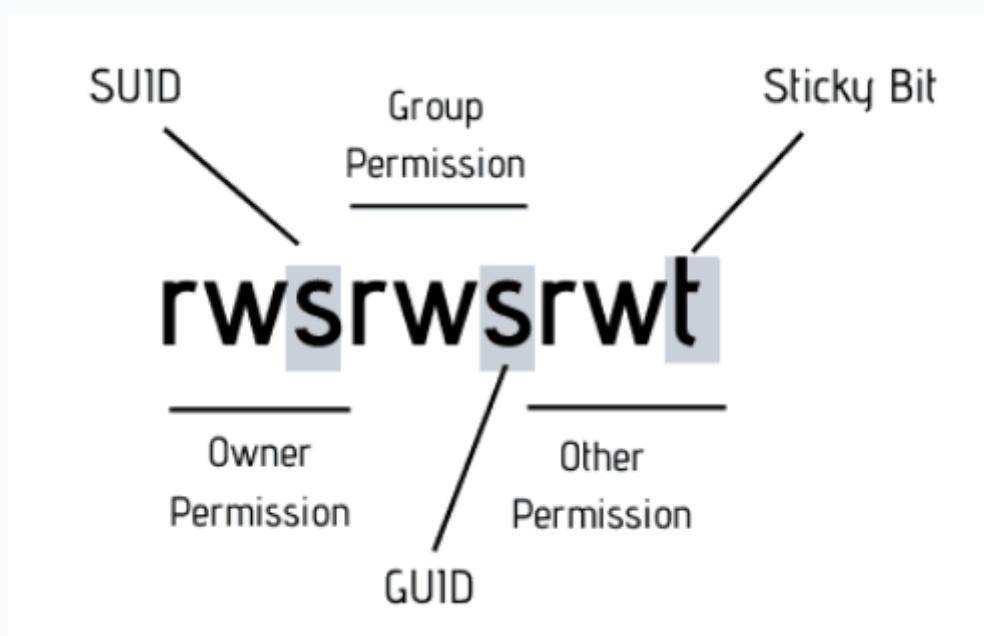


FIGURA. (Schema di rappresentazione simbolica per permessi speciali)



Rappresentazione Ottale.

La stessa informazione può essere rappresentata con 4 cifre in base 8. Dove:

- La prima cifra rappresenta i permessi speciali
 - Le altre tre cifre rappresentano i permessi per l'utente proprietario, il gruppo proprietario e tutti gli altri.
- La prima cifra è quasi sempre 0 ed è *omessa*

FIGURA. (Schema della rappresentazione ottale)

Octal:	0	6	4	0
Binary:	000	110	100	000
Symbolic:	s s t	r w x	r w x	r w x
	Special attributes	User (u)	Group (g)	Other (o)
All (a)				

Esempio

750 equivale a **rwx r-x ---**

- $7 = 4 + 2 + 1$ mentre $5 = 4 + 1$
644 equivale a **rw- r-- r--**
- $6 = 4 + 2$

Trucchetto.

Possiamo ricordarci la tabella di conversione dall'ottale al binario che è

- **0 \Leftrightarrow 000**
 - **1 \Leftrightarrow 001**
 - **2 \Leftrightarrow 010**
 - **3 \Leftrightarrow 011**
 - **4 \Leftrightarrow 100**
 - **5 \Leftrightarrow 101**
 - **6 \Leftrightarrow 110**
 - **7 \Leftrightarrow 111**
-

Comandi Bash per i dischi

Utility GNU.

I SO Linux/Posix hanno dei programmi *pre-installati* per gestire i file (1)

- Sono delle *utility*, parte della *GNU*, che permettono di svolgere compiti semplici e ripetitivi da riga di comando
- Senza dover scrivere un programma apposito che chiami le *System Call* o *Funzioni di Libreria necessarie*.
- Documentati nella sezione 1 di **man** (*User Commands*) e nella sezione 8 (*System Administration tools and Daemons*)

Comandi Bash per i dischi

Enchiammo alcuni comandi importanti.

- **df**: visualizza dischi e loro occupazione ("disk free")
- **mount**: permette di:
 - vedere quali dischi sono in uso
 - *montare* un disco, ovvero agganciarlo all'albero di file della macchina
 - usa la System Call **mount**
- **fdisk**: visualizza dischi e partizioni e crea partizioni
- **lsblk**: visualizza in maniera semplice le partizioni e i dischi ("list block devices")
- **mkfs**: formatta e inizializza un File System su un disco ("make file system"*)

- **lspci** e **lsusb**: lista dispositivi PCI e USB, tra cui dischi ("list PCI/USB")
-

Domande

Un inode può rappresentare:

- File
- Cartelle
- File o cartelle
- Link
- Partizioni

Risposta: "File o cartelle"

Il comando **mount** serve a:

- Montare dischi
- Formattare dischi
- Manipolare directory

Risposta: "Montare dischi"

Gli inode possono essere memorizzati:

- In qualsiasi posizione del disco
- All'inizio
- Alla fine

Risposta: "All'inizio"

Gli elementi di una cartella sono memorizzati:

- All'interno del suo inode
- In un blocco dati separato

Risposta: "In un blocco dati separato"

In Linux, ogni utente può appartenere a un solo gruppo:

- Vero
- Falso

Risposta: "Falso" (domanda tricky!)

L'utente root esegue i suoi processi in kernel-mode?

- Si
- No

Risposta: "No"

Il permesso di esecuzione sulle directory:

- Non ha effetto
- Permette di eseguire i programmi contenuti
- Permette di attraversare la cartella

Risposta: "Permette di attraversare la cartella"

La cartella **d** contiene un file **f**. L'utente **u** ha permessi sulla cartella **d r-x** e sul file **f rw-**. L'utente può rimuovere il file?

- Si
- No

Risposta: "No"

La cartella **d** contiene un file **f**. L'utente **u** ha permessi sulla cartella **d r-x** e sul file **f rw-**. L'utente può eseguire il file?

- Si
- No

Risposta: "No"

L'utente **u** appartiene ai gruppi **g1** e **g2**. Una cartella contiene i seguenti file.

```
-rw-r--r-- 1 u g3 2577901 Jul 28 2013 f1.txt  
-r--r--r-- 1 v g1 5634545 Jul 13 2013 f2.txt  
-rwxrwxrwx 1 z g4 8753244 Jul 29 2013 f3.txt
```

Su quali di questi file **u** ha permesso di scrittura?

- Tutti
- Solo f1
- Solo f3
- f1 e f3

Risposta: "f1 e f3"

u4-s3-file

Sistemi Operativi

Unità 4: Il File System

I file

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Funzioni di libreria per file
 2. System Call per file
 3. Funzioni di libreria vs System Call
 4. Comandi Bash per File
-

Funzioni di libreria per file

Funzioni di libreria per file: Recap

Abbiamo già visto le funzioni di libreria più comuni per leggere su file.
Esse sono parte della *libreria standard* **libc**.

- Sono utilizzabili su *qualsiasi SO* (portabile)
- E' sufficiente ricompilare il programma
- Utilizzano *System Call diverse a seconda del SO*

Funzioni principali:

- Puntatore a file: **FILE ***
- Apertura/chiusura: **fopen**, **fclose**
- Lettura/scrittura di caratteri: **fgetc**, **fputc**
- Lettura/scrittura di righe: **fgets**, **fputs**
- Lettura/scrittura con formato: **fscanf**, **fgets**
- Lettura/scrittura grezza **fread**, **fwrite**
- Riposizionamento: **fseek**, **rewind**, **ftell**

Esempio: si legga un path da tastiera e se ne stampi il contenuto come file di testo

```
#include <stdio.h>

int main ()
{
    char s[100], buffer [100];
    FILE * f;

    printf("Inserisci un path: ");
    scanf("%s", s);

    f = fopen(s, "r");
    if (f==NULL){
        printf("Impossibile aprire %s\n", s);
        return 1; /* Errore */
    }

    /* Non è importante la lunghezza di buffer */
    while ( fgets(buffer, 100, f) )
        fputs(buffer, stdout); // Equivale a printf("%s", s);

    fclose(f);
    return 0;
}
```

Funzioni per manipolare il Cursore

Nota. (*sequenzialità*)

La lettura/scrittura su file è *sequenziale*.

Un file aperto ha un *cursore* simile a quello di un editor testuale

- Determina la posizione di partenza per la prossima operazione
- Esso è posizionato a un *offset* preciso all'interno del file

- Una lettura sposta in avanti il cursore dei caratteri letti
- Una scrittura inserisce i caratteri nella posizione del cursore

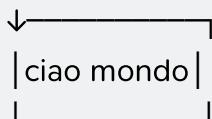
Esistono delle funzioni per spostare arbitrariamente il cursore, come viste prima.

Esempio.

Consideriamo un file 10 caratteri contenente la frase **ciao mondo**. Viene aperto con:

```
FILE * fp = fopen("ciao.txt", "r");
```

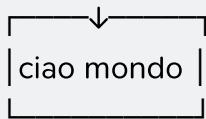
Il cursore a questo punto è all'inizio del file.



Effettuando una lettura con la **fscanf** si legge una parola (spazio finale incluso).

```
fscanf(fp, "%s", buffer);
```

Il cursore sarà quindi all'inizio della parola successiva. Una successiva **fscanf** leggerebbe **mondo**



Elenco.

1. E' possibile *spostare manualmente il cursore* con la funzione

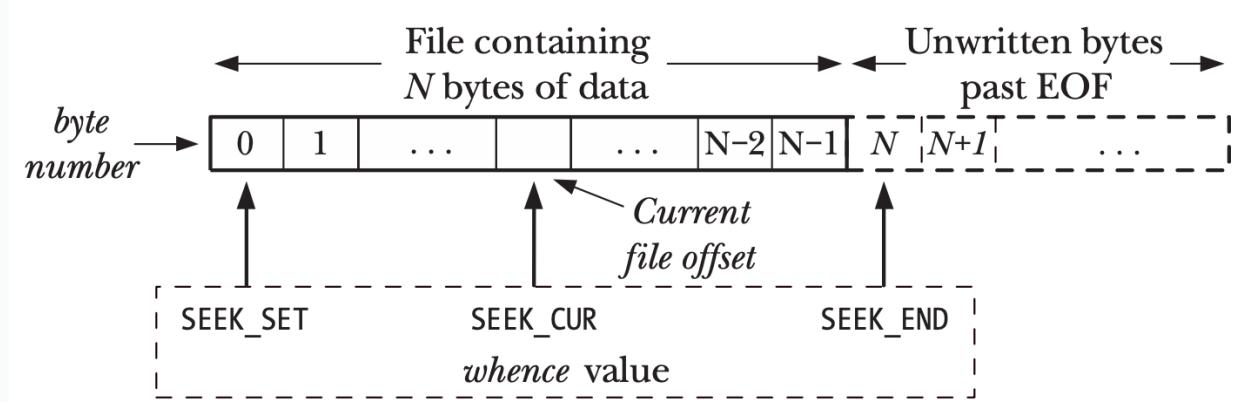
```
int fseek(FILE *fp, long distanza, int partenza)
```

Dove:

- **FILE *fp**: è il puntatore a file sul quale agire
- **long distanza**: è il nuovo offset.

- **int partenza** o **whence** indica da dove **distanza** viene calcolata. Può assumere
 - **SEEK_SET**: inizio del file
 - **SEEK_END**: fine del file
 - **SEEK_CUR**: posizione corrente del cursore
 Nota: tutti i tre e valori sono delle **costanti** definite nella libreria.

Figura. (funzionamento di **fseek**)



2. Per sapere *in che posizione è il cursore*:

```
long ftell(FILE *stream);
```

Esempio.

Consideriamo un file 10 caratteri contenente la frase
ciao mondo.

```
FILE * fp = fopen("ciao.txt", "r");
fscanf(fp, "%s", buffer); // Buffer contiene "ciao". Il cursore è prima di "mondo"

fseek (fp, 0, SEEK_SET); // Il cursore torna all'inizio del file
fscanf(fp, "%s", buffer); // Leggo di nuovo "ciao"
```

Similmente:

```
FILE * fp = fopen("ciao.txt", "r");
fgets(buffer, 2, fp); // Buffer contiene "ci". Il cursore è tra "ci" e "ao mondo"
fseek (fp, -3, SEEK_END); // Il cursore si posiziona tra "ciao mo" "ndo"
fgets(buffer, 2, fp); // Leggo "ndo"
```

Funzioni per la lettura e la scrittura su File Binari

Funzioni `fread` e `fwrite`

Simili a `fgets` e `fputs`, ma *per file binari*.

- Ignorano il ritorno a capo `\n`.
- Leggono/scrivono *una quantità fissa di byte*
- Ottime per file binari: *contengono caratteri non stampabili*, e anche tanti '`\0`' (*terminatori*).
 - Impossibile leggere correttamente i terminatori.
 - Possiamo scrivere qualsiasi cosa! Permettono di leggere/scrivere file binari (che contengono '`\0`').
 - Oppure `int`, `float`, `struct` e vettori

Funzionamento: leggi/scrivi nella `nmemb` oggetti, ognuno grande `size` byte dal puntatore a file `stream` e scrivilo/leggilo da `ptr`.

Valore di ritorno: il numero di elementi effettivamente letti/scritti.

```
C  
size_t fread(void *restrict ptr, size_t size, size_t nmemb,  
            FILE *restrict stream);  
size_t fwrite(const void *restrict ptr, size_t size, size_t nmemb,  
             FILE *restrict stream);
```

Esempio.

Lettura di un vettore di interi.

Si supponga un file contenente (in binario) i 2 interi che rappresentano i numeri 1990 e 2023. In esadecimale e considerando `int` su 4 byte, avremo nel file:

```
0x00 0x00 0x07 0xC6 0x00 0x00 0x07 0xE7
```

Suggerimento: crea il file con `echo -n -e '\x00\x00\x07\xC6\x00\x00\x07\xE7' > ciao.txt`

Per leggerli in C, si procede con la funzione `fread`

```
FILE * fp = fopen("ciao.txt", "rb"); // Notare modalità "rb"
int v [2];
fread(v, sizeof(int), 2, fp);
```

Nota: è errato usare **fscanf(... , "%d", ...)** che si aspetta dei numeri scritti come stringhe!

Osservazioni dall'esempio:

- **size_t** è un alias per il tipo di dato *intero senza segno* che viene usato per rappresentare grandezze di strutture dati.
- **sizeof** è un operatore che ritorna il numero di byte che un tipo di dato occupa
 - Durante la *compilazione*, il compilatore sostituisce l'espressione col suo risultato

Bufferizzazione

Adesso evidenziamo un *aspetto particolare* della lettura/scrittura di file su C

Definizione. (*Bufferizzazione*)

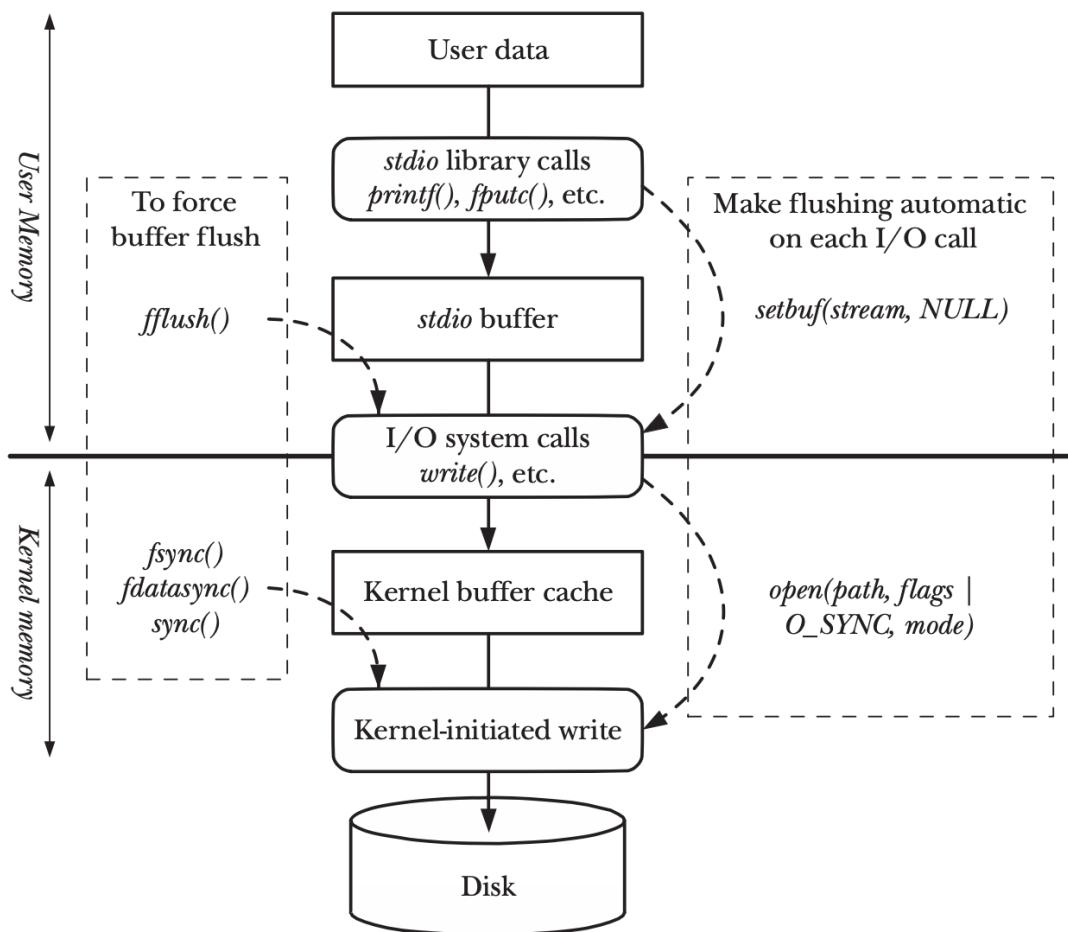
In C, l'input/output su file è *bufferizzato*

- Le funzioni di scrittura come **fprintf** *non invocano sempre la System Call* per la scrittura su file
- Esse scrivono *su un buffer in memoria*
- Quando esso è *pieno*, il contenuto è *effettivamente scritto* su file

Motivazione.

Questo comportamento *migliora significativamente le prestazioni* in caso di tante scritture di piccole dimensione: bisognerebbe accedere al disco ad ogni operazione! Quindi in un'unica andata scriviamo ciò che ci interessa.

Figura. (*Bufferizzazione*)



Manipolazione delle Bufferizzazioni

Definizione. (*Costante BUFSIZ*)

La dimensione di default è la costante **BUFSIZ**, che è di circa **65kB**.

E' possibile *settare la dimensione del buffer* per casi particolari:

```
void setbuf(FILE *stream, char *buf);
```

Funzione.

1. Si può *forzare una scrittura su file*:

```
int fflush(FILE *stream);
```

Nota: quando lo standard output è su console esso è *new line-buffered*. Un ritorno a capo `\n`, forza il *flush*.

Per rendere uno stream *new line-buffered* si usa la funzione **setvbuf**.

Funzione per la Rimozione dei File

Rimozione

```
C  
int remove(const char *pathname);
```

Non è necessario che il file sia aperto.

In Linux, **remove()** usa la *System Call*:

```
C  
int unlink(const char *pathname);
```

Vedremo come mai si chiama "**unlink**" in seguito, quando approfondiremo gli *hard/soft link*.

System Call per file

System Call per file: Recap

Le funzioni viste precedentemente *sono parte della libreria standard del C*.

- Esse *utilizzano delle System Call* per compiere le operazioni richieste
- Le System Call *variano* a seconda del SO

System Call usate, esempi.

- Linux/POSIX: **open**, **read**, **write**, **lseek**, **close**
- Windows: **CreateFile**, **WriteFile**, **ReadFile**, **CloseHandle**, **SetFilePointer**

System Call di Linux.

Noi ci soffermiamo sulle *System Call di Linux*, usabili includendo **<unistd.h>**.

- Esse sono *simili alle funzioni di libreria*, ma sono di più basso livello.
- Si possono usare su *sistemi Linux e Posix*
- *Non esistono su Windows*. Non sono parte di Libreria Standard del C

Figura. (*Primo confronto tra System call e Funzioni di Libreria*)

System calls	Library functions
file descriptor (<i>int</i>)	file stream (<i>FILE *</i>)
<i>open()</i> , <i>close()</i>	<i>fopen()</i> , <i>fclose()</i>
<i>lseek()</i>	<i>fseek()</i> , <i>ftell()</i>
<i>read()</i>	<i>fgets()</i> , <i>fscanf()</i> , <i>fread()</i> . . .
<i>write()</i>	<i>fputs()</i> , <i>fprintf()</i> , <i>fwrite()</i> , . . .
—	<i>feof()</i> , <i>ferror()</i>

Definizioni Relative all'Apertura di File

Definizione. (*file descriptor*)

In Linux, un *file aperto* è identificato da un *file descriptor*.

Esso è un *numero intero non negativo*, che per convenzione hanno tre significati:

- Standard Input: descrittore 0 (stdin)
- Standard Output: descrittore 1 (stdout)
- Standard Error: descrittore 2 (stderr)

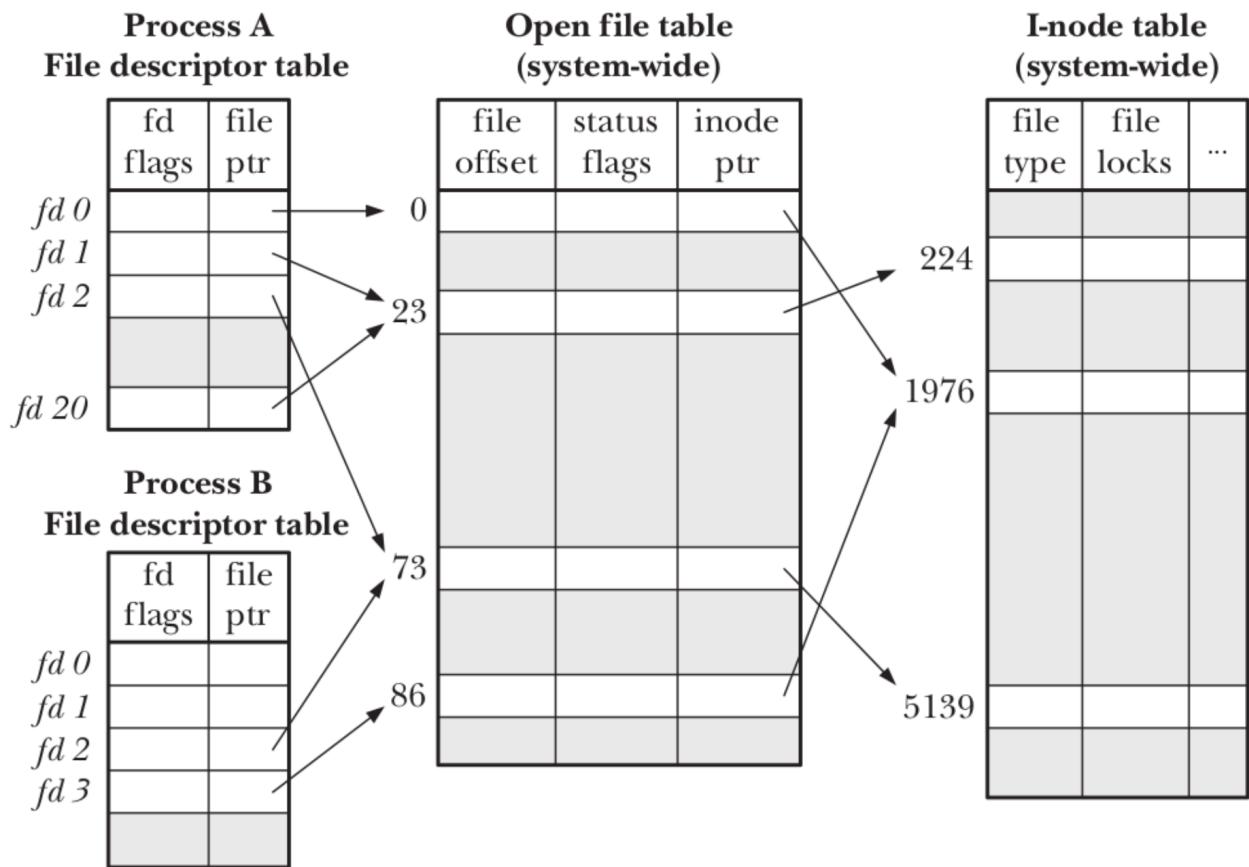
Nota. (*differenza*)

Nelle funzioni di libreria un file aperto è un **FILE ***. Nelle System Call Linux è un **int**.

Definizione. (*File table*)

Il sistema operativo mantiene delle *tabelle* (dette come "*file table*") che mappano i *file descriptor* ai file fisici su disco (*inode*), ovver una *serie di strutture dati*.

Figura. (*Schema*)



Nota. Le tabelle dei *processi A, processi B* rispondono alla domanda: "*quali file sono aperti nei processi individuali?*"; la tabella centrale, ovvero quella generale del S.O. agisce su *dei processi* e l'ultima tabella è la *tabella degli i-node corrispondente*, che serve le richieste.

Analisi Approfondita.

- Per ogni *processo*, vi è una tabella che contiene i *file descriptor*:
 - Contiene un *riferimento alla tabella generale*
 - E dei *flag* (come ad esempio le modalità di apertura del file)
- La *tabella generale* (una per tutto il SO), contiene:
 - Access Mode: *R, W, RW*
 - File Offset: posizione del *cursore*
- La *tabella degli inode* è semplicemente una *copia in memoria degli inode interessati* (che si trovano su disco)

Apertura di File con System Call

Apertura di un file

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- Apre il file identificato dal path **pathname**.

- **flags** determinano modalità di accesso al file.
 - Uno tra **O_RDONLY**, **O_WRONLY**, e **O_RDWR** deve essere *obbligatoriamente* presente.
 - Altri flag sono:
 - **O_CREAT** crea il file se non esiste
 - **O_APPEND** apri il file in modalità aggiunta
 - *Nota:* con le funzioni di libreria questo è il *comportamento default*.
 - I flag si sommano usando l'operatore OR bit a bit |

Esempi.

```
C
int fd = open("file.txt", O_RDONLY);
int fd = open("file.txt", O_WRONLY | O_APPEND);
```

Flag **O_CREAT**.

Nel caso si specifichi il flag **O_CREAT**, il file viene creato *coi permessi specificati* in **mode**.

Esistono 9 flag:

- **S_I[RWX]USR**, **S_I[RWX]GRP**, **S_I[RWX]OTH**
Ricordiamoci che in Linux i file hanno 3 tipi di permessi (Read, Write, Execute), gestibili separatamente per il *proprietario, il gruppo e gli altri utenti* (1).

Chiusura di File con System Call

Chiusura di un file:

```
C
int close(int fd);
```

Chiude il *file descriptor*. Il numero **fd** non si riferisce più a nessun file aperto e *può essere riutilizzato in successive* **open** da parte del SO.

Lettura e Scrittura su File con System Call

1. Lettura da file:

```
ssize_t read(int fd, void *buf, size_t count);
```

Leggi **count** byte da **fd** e mettili nella memoria all'indirizzo **buf**.

Valore di ritorno: il *numero di byte letti*. Può essere minore di **count** se il file finisce.

- In caso di errore -1
- Se si è giunti a EOF 0

2. Scrittura su file:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Scrivi **count** byte su **fd** e prendendoli nella memoria all'indirizzo **buf**.

Valore di ritorno: il *numero di byte scritti*. Può essere inferiore a **count** se il disco si riempie.

- In caso di errore -1

Nota. **ssize_t** sta per "signed size_t".

Manipolazione Cursore con System Call

Riposizionamento Cursore:

```
off_t lseek(int fd, off_t offset, int whence);
```

Molto simile alla funzione di libreria **fseek**.

Riposiziona il file descriptor **fd** all'offset **offset** secondo la direttiva **whence** come segue:

- **SEEK_SET**: **offset** è rispetto a inizio file
- **SEEK_CUR**: **offset** è rispetto a posizione corrente
- **SEEK_END**: **offset** è rispetto a fine file. **offset** dovrà essere negativo

Nota. C'è una corrispondenza uno a uno con la *funzione di libreria fseek*

Esempi Generali

Esempio: scrittura utilizzando System Call

```
const char *str = "Arbitrary string to be written to a file.\n";
const char* filename = "innn.txt";

int fd = open(filename, O_RDWR | O_CREAT);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}

write(fd, str, strlen(str));
printf("Done Writing!\n");

close(fd);
```

Esempio: equivalente usando funzioni di libreria

```
const char *str = "Arbitrary string to be written to a file.\n";
const char* filename = "innn.txt";

FILE* output_file = fopen(filename, "w+");
if (!output_file) {
    perror("fopen");
    exit(EXIT_FAILURE);
}

fwrite(str, 1, strlen(str), output_file); // Si può usare fputs o fprintf
printf("Done Writing!\n");

fclose(output_file);
```

Esempio: sono equivalenti le seguenti forme per stampare il messaggio **Hello World** su console.

Usando la **printf** si stampa su Standard Output.

C

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

Si può usare la **fprintf** dicendole di stampare sul file **stdout**, che è un **FILE *** pre-definito.

C

```
fprintf(stdout, "Hello World\n");
```

Si può usare la System Call **write** (solo su Linux/POSIX).

- Si stampa sul file descriptor 1, per convenzione lo Standard Output
- E' necessario specificare quanti byte scrivere. **write** stampa dei byte, non utilizza il terminatore '**\0**' (valore 0)

C

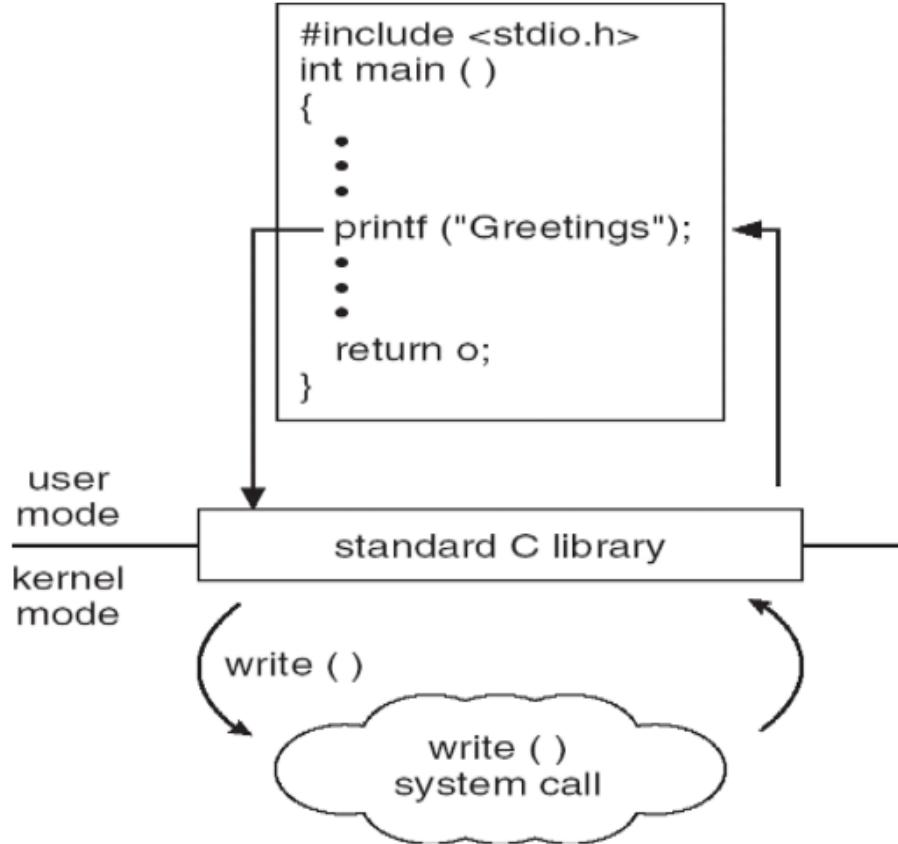
```
write(1, "Hello World\n", 13);
```

Funzioni di libreria e System Call

Adesso confrontiamo le *funzioni di libreria* con le *system call* per manipolare file.

Aspetti Diversi

1. Le funzioni di libreria utilizzando le System Call



2. Le **Funzioni di Libreria** sono eseguite in *modalità utente*. Non hanno nessun privilegio particolare.

- Sono semplicemente delle *funzioni che facilitano l'uso delle System Call*

3. Le **System Call** sono eseguite in *modalità kernel*.

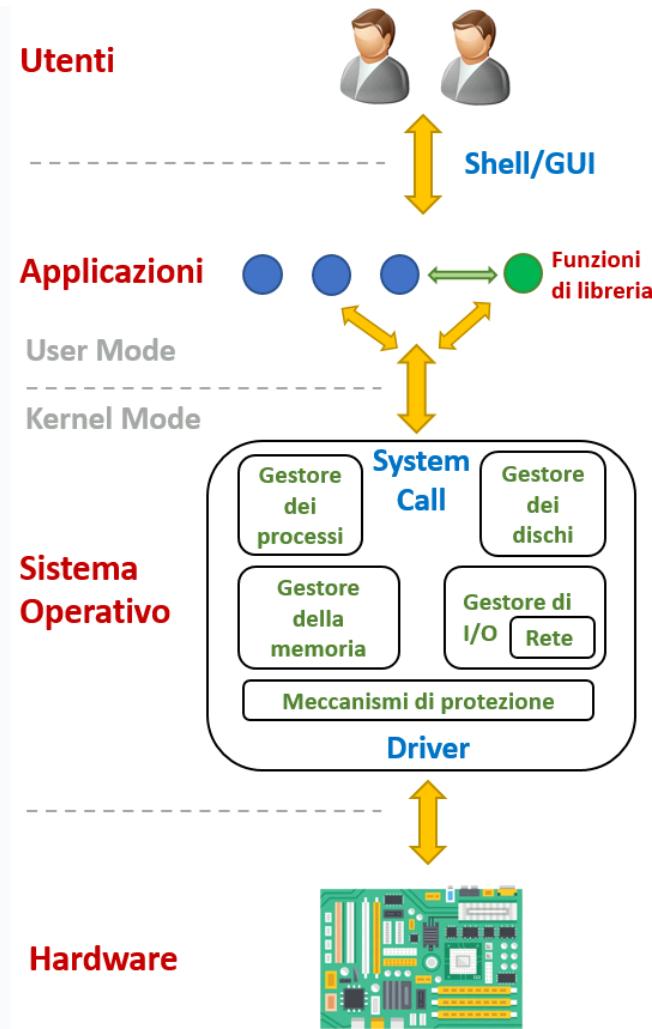
- Hanno accesso alla *memoria fisica*
- Possono accedere alle interfacce dei dispositivi di I/O
- Vengono fornite dal *Sistema Operativo*

Aspetti Comuni

Le **Applicazioni** possono invocare *sia funzioni di libreria che System Call*.

Se vogliono usufruire dei servizi del SO è *sempre necessario usare le system call*

- Lo fanno *le applicazioni direttamente*
- Oppure lo fanno *le funzioni di libreria* invocate dalle applicazioni

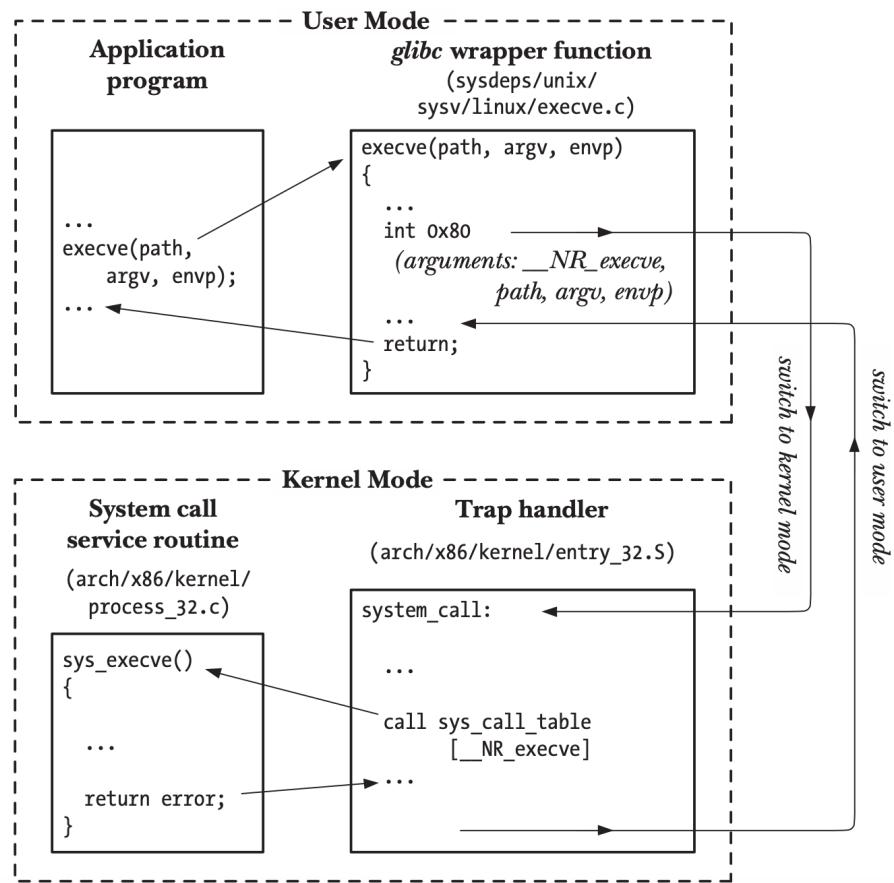


Passaggi per l'invocazione di una System Call

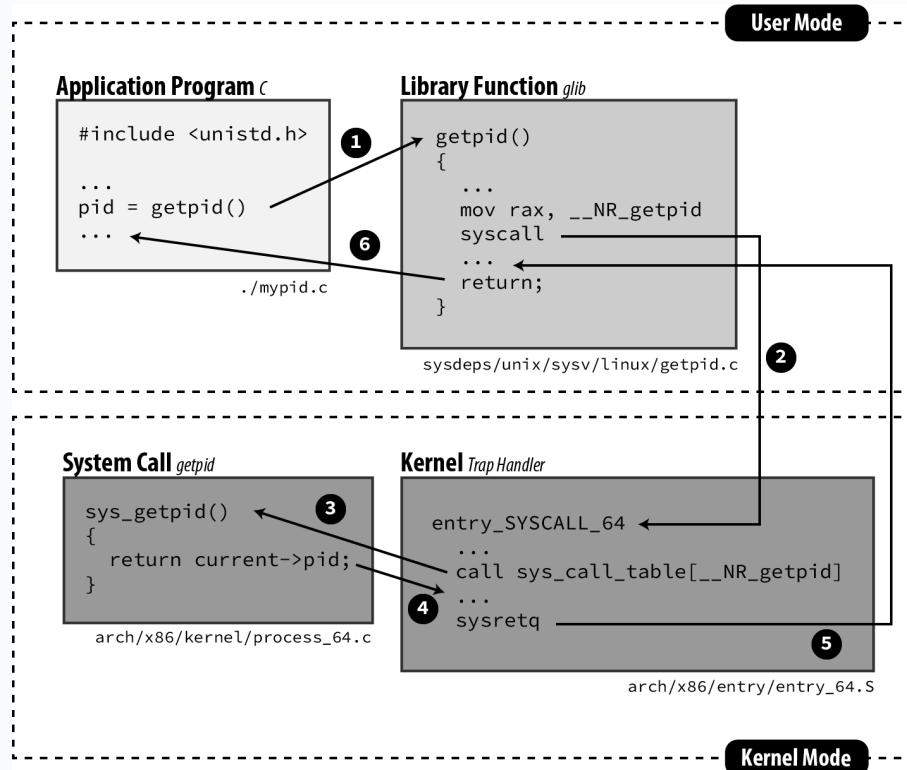
Ci sono *due metodi* per invocare una system call.

1. Tramite **int 0x80** (*metodo classico*)

- **Nota.** La funzione **int 0x80** non è altro che una *interrupt* (1), dal momento che viene chiamata la **CPU** viene interrotta ed esegue un pezzo di codice predefinito. Da osservare che c'è un pezzo di assembly nel codice!



2. Tramite **syscall** (metodo moderno)



Osservazione. Qui abbiamo sempre una **situazione delicata**! Se non siamo abbastanza attenti, questo potrebbe generare delle **vulnerabilità di sicurezza**, dal momento che stiamo passando da **user-mode** a **kernel-mode** in una maniera **quasi-arbitraria**.

Manuale di Linux

Manuale:

Il manuale di Linux è diviso in sezioni:

1. User Commands
2. System Calls
3. C Library Functions
4. Devices and Special Files
5. Eccetera...

La **fopen** è in sezione 3, la **open** in sezione 2.

Invece **printf** è sia un *comando bash* che una *funzione di libreria*

- Ha *due pagine di manuale*
 - **man 1 printf**: comando Bash
 - **man 3 printf**: funzione di libreria C

Strumenti per Debuggare Codici con System Call

Esistono degli strumenti di debug per vedere quali System Call vengono invocate da un processo.

- Si può fare tramite **profile** (e.g., **valgrind**: utile per vedere le prestazioni), **debugger** (e.g., **gdb**, utile per vedere tutto passo a passo) o tool nativi (e.g., **strace**, per stampare una lista di System Call invocati)
 - Ci focalizziamo in particolare su **strace** e su **gdb**.
1. **strace**
Non richiede di ricompilare i programmi. Funziona sempre, anche se non ho il sorgente del programma. E' un tool del SO.

Funzionamento: **strace comando**

Esempio: Di default **strace** lista le system call

SHELL

```
$ strace pwd
execve("/usr/bin/pwd", ["pwd"], 0x7ffd37cdfc80 /* 72 vars */      = 0
...
getcwd("/tmp", 4096)          = 5
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x5), ...}) = 0
write(1, "/tmp\n", 5/tmp)     = 5
```

Si può semplicemente contare le invocazioni a System Call

SHELL

```
$ strace -c date
mar 25 ott 2022, 10:16:48, CEST
% time    seconds  usecs/call   calls  errors syscall
-----
0,00  0,000000      0       6      read
0,00  0,000000      0       1      write
0,00  0,000000      0       9      close
...
...
```

2. **ltrace**

Simile a **strace**. Permette di visualizzare le *funzioni di libreria* usate da un processo

Funzionamento: **ltrace comando**

Nota: *non funziona su tutti gli eseguibili.*

Solo quelli compilati con *lazy binding* (opzione del linker), di default fino a Ubuntu 16.

Per essere sicuri, compilare con l'opzione: **-z lazy**

Esempio: **gcc sample.c -o myprog -z lazy**

Esempio: Di default **ltrace** lista le invocazioni a funzione

SHELL

```
$ ltrace pwd
...
getcwd(0, 0)           = ***
puts("/home/det_user/trevisan")/home/det_user/trevisan
)                      = 24
...
```

Per contare le funzioni invocate:

```
$ ltrace -c date
Tue Oct 25 10:35:00 CEST 2022
% time    seconds  usecs/call  calls  function
-----
15.59  0.000543      67      8 fwrite
13.90  0.000484      60      8 fputc
 8.76  0.000305     305      1 setlocale
...
...
```

Comandi Bash per File

Ripassiamo i comandi Bash per manipolare i file.

Le Utility di GNU/Linux.

I SO Linux/Posix hanno dei *programmi pre-installati per gestire i file*:

Sono delle *utility* che permettono di svolgere compiti semplici e ripetitivi da riga di comando

Senza dover scrivere un programma apposito che chiami le *System Call o Funzioni di Libreria necessarie*.

Sono documentati nella sezione 1 di **man**

1. Lettura di file:

- **cat filename**: stampa su Standard Output il contenuto di un file
- **less filename** e **more filename**: visualizzazione passo-passo
- **head filename** e **tail filename**: stampa prime/ultime righe di un file
- **grep pattern file**: stampa delle righe che contengono un **pattern**

2. Scrittura di file:

- Per scrivere su file si usa tipicamente la *redirezione su file* della Bash:

```
echo "Ciao" > filename
```

- Con **touch filename** creo un file se non esiste

3. Rimozione di file:

- **rm filename**: rimuove un file (se ho i permessi per farlo)

4. Altre operazioni:

- **cp origine destinazione**: copia un file
- **chmod, chgrp, chown**: modificano permessi, gruppo e proprietario di un file

Domande

La **fopen** é:

- Una funzione di libreria
- Una System Call
- Una struct

Risposta: *Funzione di Libreria*

La **read** é:

- Una funzione di libreria
- Una System Call
- Una struct

Risposta: *System Call*

Si consideri il file **f.txt** contente il testo **frase di prova**.

Dove si trova il cursore () del file dopo il seguente codice?

```
FILE * fp = fopen("f.txt","r");
fscanf(fp, "%s", buffer);
fseek (fp, 3, SEEK_CUR);
```

- **fra↓se di prova**
- **frase di ↓prova**
- **frase di pr↓ova**
- **frase ↓di prova**

Risposta: *frase di ↓prova*

Quale delle seguenti linee di codice é corretta?

- **int fd = open("file.txt", O_RDONLY);**
- **FILE * fd = open("file.txt", O_RDONLY);**
- **FILE * fd = fopen("file.txt", O_RDONLY);**
- **FILE fd = open("file.txt", "rw");**

Risposta: *int fd open("file.txt", O_RDONLY);*

Quale relazione c'è tra la **fprintf** e **write**?

- **fprintf** é una funzione di libreria e usa la SysCall **write**
- **write** é una funzione di libreria e usa la SysCall **fprintf**
- **fprintf** e **write** sono due funzioni di libreria
- **fprintf** e **write** sono due System Call

Risposta: **fprintf è una funzione di libreria e usa la SysCall write"*

Quale dei seguenti comandi stampa a schermo il contenuto del file **ciao**?

- **echo ciao**
- **print ciao**
- **cat ciao**

Risposta: *cat ciao*

Link e Directory

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Link
 2. Soft Link
 3. Hard Link ---
 4. System call per Directory
 5. Funzioni di libreria per Directory
 6. Comandi Bash per Link, Directory e Disco ---
-

Link

Definizione. (*Link generalizzato*)

Un link é un *nome aggiuntivo per un altro file*.

Utile per svariati compiti

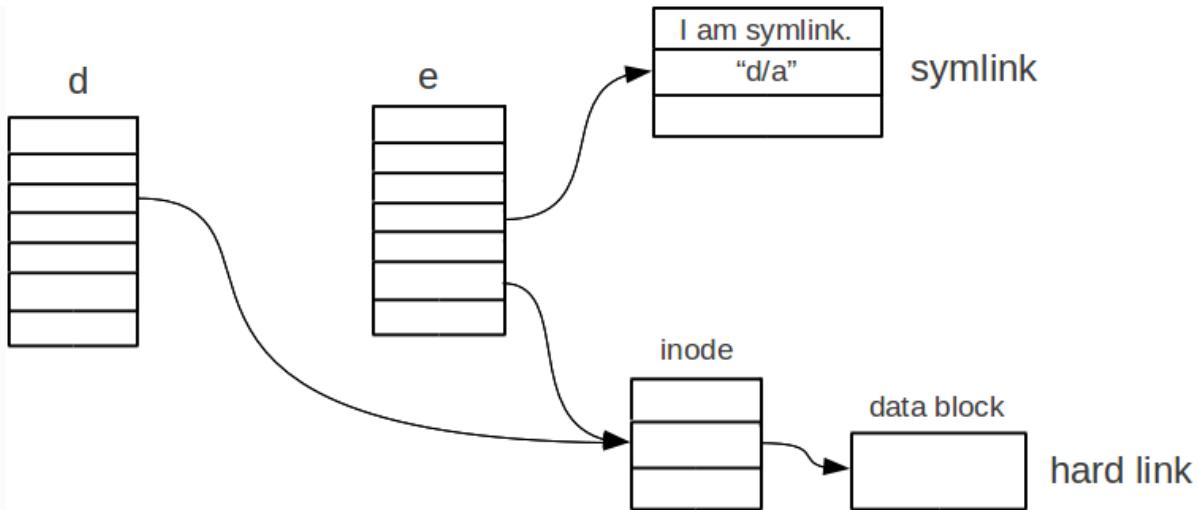
- Gestione file di configurazione
- Condivisione di informazioni tra utenti
- Mantenimento ordinato della struttura dei file
- E molto altro!

Definizione. (*Hard Link e Soft Link*)

Esistono due tipi di link:

- **Hard Link:** l'inode appare *in una seconda directory* che vi punta; abbiamo quindi un *riferimento aggiuntivo* allo stesso *i-node*.
- **Soft Link:** é un alias a un certo path; in questo caso è un "*file speciale*" che va a recuperare il contenuto del file puntato, se possibile.

Figura. (*Idea*)



Comportamenti del Soft Link

Come detto prima, i soft link sono degli *Shortcut* per un file o una directory

- Se ho un grande file con un *path lungo e complesso*, ne posso creare un soft link nella mia Home Directory
- Se cancello un Soft Link, *non succede niente al file originale*
- Se il file originale viene cancellato, il Soft Link *continua a esistere* ma diventa *invalido*
- Se creo un altro file con quel nome, il Soft Link *torna a essere valido*
- I Soft Link sono molto flessibili

System Call.

E' un concetto di Linux

Per creare un Soft Link si usa la *System Call*:

```
int symlink(const char *target, const char *linkpath);
```

Si rimuove un Soft Link come fosse un normale file.

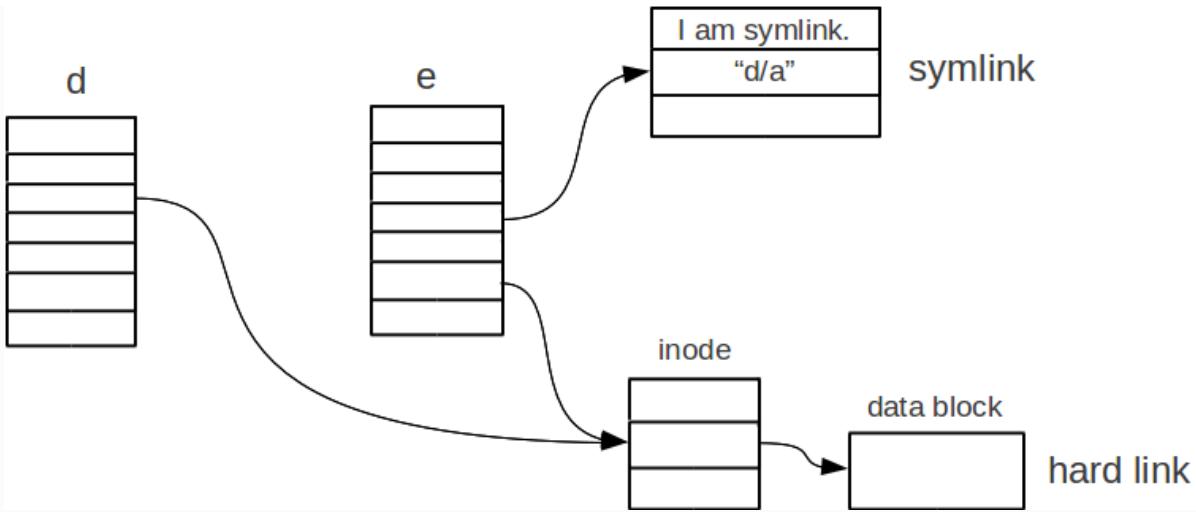
Nota: Si possono creare Soft Link a cartelle e verso altri dischi

Le funzioni di ricerca *non devono attraversare i Soft Link* per evitare cicli

Comportamenti del Hard Link

Un Hard Link é un *riferimento aggiuntivo a un inode*

La directory dove viene creato, contiene *una nuova entry* che ha lo stesso *inode number*



Implicazioni.

- Un Hard Link è *un link al contenuto del file* (quindi non un semplice rimando)
- Non può mai essere *invalido* (a meno che qualcosa di *veramente brutto* sia successo col File System)
- Hard Link e file originario hanno la *stessa importanza e la stessa natura*: in un certo senso diventano la stessa cosa
- Cancellare un Hard Link causa la cancellazione del file *solo se non vi sono altri Hard Link* (o il riferimento originale). Come si fa? Questo è il compito del *sistema operativo*.

Compiti del sistema operativo

- Mantenere un *reference count* per ogni inode
- Cancellare un inode e il suo contenuto *se esso va* a 0
- Questo è l'approccio accettato, dato che altri approcci sarebbero impensabili. Ad esempio, quello di cercare l'interno disco in ricerca di altri riferimenti non può essere fattibile.

System Call.

Si creano con la System Call:

```
int link(const char *oldpath, const char *newpath);
```

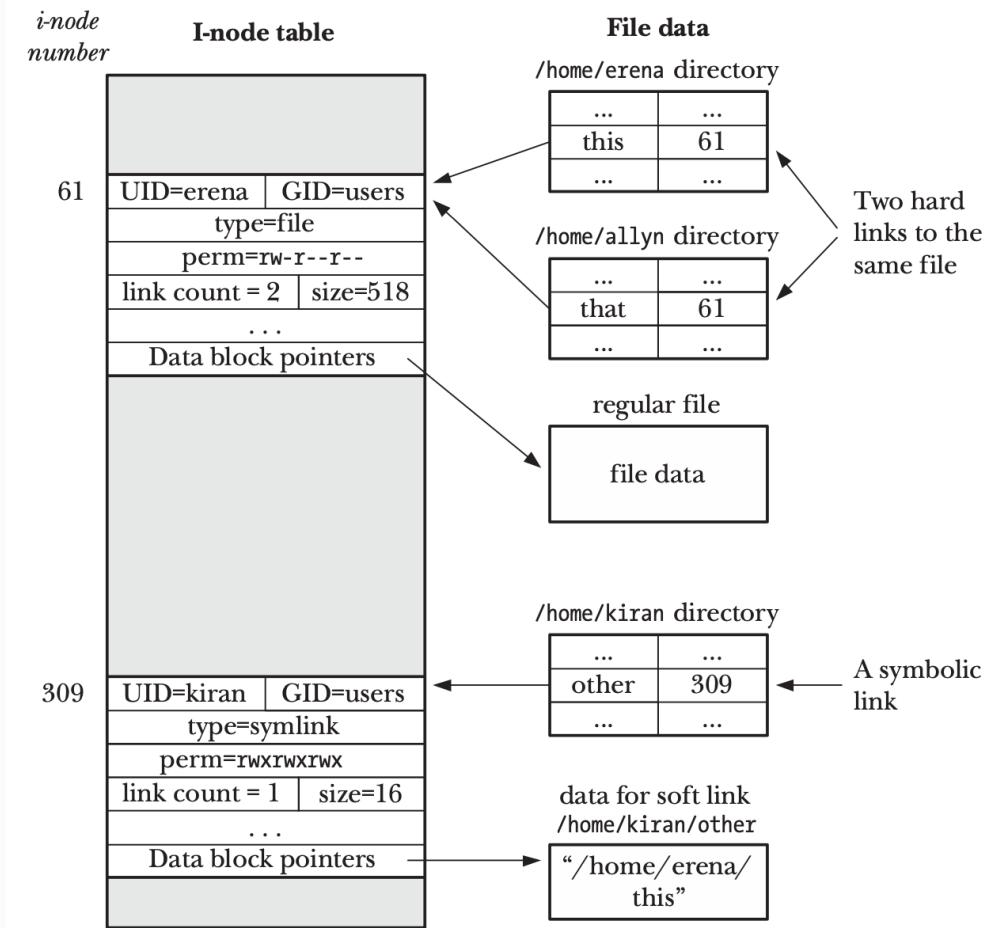
Nota: La System Call per rimuovere un file in Linux è **unlink**.

Di fatto rimuovere un file, significa decrementare il suo *reference count* (e *eventualmente rimuovere l'inode*)

Non si possono creare Hard Link a cartelle. Concettualmente sbagliato; ci sarebbe un pastroccio!

Differenza tra Hard Link e Soft Link

Figura. (*Differenza tra Hard Link e Soft Link*)



Osservazioni. Non posso creare degli *hard link* per file su *dischi diversi*, dato che avrei degli *inode diversi* per *dischi diversi*.

System Call per Interagire con Directory

System call per Directory: Leggere

1. Lettura di informazioni su una directory o un file:

```
#include <sys/stat.h>
int stat(const char *restrict pathname,
         struct stat * statbuf);
```

Ritorna informazioni su **pathname**, le colloca nella struttura puntata da **statbuf**

- Passaggio di variable per riferimento che é di fatto un valore di ritorno
- Ritorna 0 in caso di successo, -1 in caso di errore
- La struttura **statbuf** contiene i *metadati* dell'inode

Definizione. (*struct stat*)

La **struct stat** ritornata contiene i seguenti campi:

```
struct stat {  
    dev_t    st_dev;      /* ID of device containing file */  
    ino_t    st_ino;      /* Inode number */  
    mode_t   st_mode;     /* File type and mode */  
    nlink_t  st_nlink;    /* Number of hard links */  
    uid_t    st_uid;      /* User ID of owner */  
    gid_t    st_gid;      /* Group ID of owner */  
    dev_t    st_rdev;     /* Device ID (if special file) */  
    off_t    st_size;     /* Total size, in bytes */  
    blksize_t st_blksize; /* Block size for filesystem I/O */  
    blkcnt_t st_blocks;   /* Number of 512B blocks allocated */  
  
    struct timespec st_atim; /* Time of last access */  
    struct timespec st_mtim; /* Time of last modification */  
    struct timespec st_ctim; /* Time of last status change */  
};
```

Definizione. (*mode_t*)

Il campo **mode_t** indica *se si tratta di file* (o qualsiasi altra cosa) in forma di una **bit mask**.

Si possono usare le seguenti macro per testare facilmente **mode_t**

- **S_ISREG(m)**: True se file regolare
- **S_ISDIR(m)**: True se file directory
- **S_ISLNK(m)**: True se file un Symbolic Link

System call per Directory: Esempio

Si scriva un programma che riceve un path da riga di comando e stampa se esso é file, directory o link simbolico. *Importante per l'esame!*

```
#include <stdio.h>
#include <stdlib.h> // Necessario per exit
#include <sys/stat.h> // Necessario per stat
int main (int argc, char * argv[])
{
    struct stat buf;
    if (argc!=2){
        printf("Specifica un path\n");
        return 1;
    }

    if (stat(argv[1], &buf) < 0) {
        printf ("Impossibile leggere le informazioni sul file\n");
        exit (1); /* Termina subito il programma con codice 1 */
    }

    if (S_ISREG(buf.st_mode))
        printf("%s: file\n", argv[1]);
    else if (S_ISDIR(buf.st_mode))
        printf("%s: directory\n", argv[1]);
    else if (S_ISLNK(buf.st_mode))
        printf("%s: link simbolico\n", argv[1]);
    else
        printf("%s: altro tipo di path\n", argv[1]);

    return 0;
}
```

System call per Directory: Creazione e Rimozione

2. Creazione di directory

```
int mkdir (const char *path, mode_t mode);
```

3. Rimozione di directory

```
int rmdir (const char *path);
```

- Rimuove *solo se* la cartella è vuota.

mode ha stesso ruolo che nella **open**

Valore di ritorno 0 in caso di successo –1 in caso di errore

System call per Directory: Listare

Per *listare il contenuto di una directory* si possono usare le System Call **open** e **getdents** e la **struct linux_dirent**.

1. Si apre una directory come fosse un file

```
int fd = open("path", O_RDONLY | O_DIRECTORY);
```

2. Si leggono batch di **struct linux_dirent**

```
int nread = syscall(SYS_getdents, fd, buf, BUF_SIZE);
```

Tutto ciò è difficile, *poco pratico e non portabile*. Infatti, manca esiste un wrapper in C per invocarla. Per noi sarà inutile, e useremo direttamente le *funzioni di libreria*.

Si usano sempre le *funzioni di libreria POSIX* per leggere il contenuto di una cartella

Funzioni di Libreria per Directory

Listare File con Funzione di Libreria

Per listare il *contenuto di una cartella*, si usano le funzioni di libreria

```
#include <sys/types.h> // per definire tutti i tipi derivati
#include <dirent.h> // per dirent
DIR * opendir(const char *name);
struct dirent *readdir(DIR *dirp);
// faccio quello che voglio
int closedir(DIR *dirp);
```

Funzionano **solo** su sistemi POSIX.

Su Windows si usano **FindFirstFile()** e **FindNextFile()**.

1. **Apertura:** una cartella, prima di essere letta, va aperta con **opendir**.

Essa ritorna un puntatore a **DIR *** se l'apertura va a buon fine, altrimenti **NULL**

Un **DIR *** è l'equivalente di **FILE *** per le directory

```
DIR * d;  
d = opendir("/path/");  
if (d!=NULL)  
...  
...
```

Struttura "dirent".

Una **struct dirent** rappresenta un elemento di una directory.

Contiene i campi:

```
struct dirent  
{  
    ino_t      d_ino;    /* inode number */  
    char       d_name[256]; /* filename */  
    ... // Da qui in poi è irrilevante per noi  
};
```

2. **Listare il contenuto:** si usa la funzione **readdir** che ritorna **una struct dirent ***

- Opera in maniera sequenziale
 - A ogni invocazione ritorna l'elemento successivo
- Va invocata finche non ritorna **NULL** (ovvero ho finito tutti i file da elencare)

```
struct dirent * entry;  
while ((entry = readdir(d)) != NULL)  
    printf(" %s\n", entry->d_name);
```

- L'ordine degli elementi ritornati *dipende dal FS* e di solito *non ha alcun significato*

3. **Chiusura cartelle.** Infine bisogna chiudere una cartella per rilasciare le risorse associate.

```
int r = closedir(d);
```

Ritorna 0 se l'operazione va a buon fine, altrimenti –1. Ad esempio, se provo a chiudere una cartella *più di una volta*, allora finisce male.

Funzioni di libreria per Directory: Esempio

Si scriva un programma che riceve una directory da riga di comando e ne lista il contenuto. Importante per l'esercitazione e per l'*esame*!

```
C

#include <stdio.h>
#include <stdlib.h> // Necessario per exit
#include <sys/stat.h> // Necessario per stat
#include <dirent.h> // Necessario per struct dirent *
int main (int argc, char * argv[])
{
    struct stat buf;
    struct dirent *dirp;
    DIR *dp;

    if (argc!=2){
        printf("Specifica un path\n"); exit (1);
    if (stat(argv[1], &buf) < 0) {
        printf ("Impossibile leggere le informazioni sul file\n"); exit (1);
    if (!S_ISDIR(buf.st_mode)){
        printf("%s deve essere una directory\n", argv[1]); exit(1);
    if ( (dp = opendir(argv[1])) == NULL) {
        printf("%s impossibile da aprire\n", argv[1]); exit (1);

    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);

    return 0;
}
```

Comandi Bash per Link, Directory

Sono programmi pre-installati che facilitano l'uso delle System Call per compiti comuni.
Ne facciamo un ripasso

Comandi per link:

SHELL

```
ln target link_name
```

Crea un link al path **link_name** verso un path esistente **target**

- Di default crea un hard link
- L'opzione **-s**
- Si rimuovono i link con **rm**
- Si possono anche usare i comandi più grezzi **link** e **unlink**

Comandi per stat:

SHELL

```
stat path
```

Invoca la System Call **stat()** sul **path** specificato

- Stampa in formato human-readable tutto ciò che c'è dentro la **struct stat** risultante (e poco altro)
- Esempio:

```
$ stat file.tex
  File: file.tex
  Size: 0          Blocks: 0          IO Block: 4194304 regular empty file
Device: 31h/49d Inode: 1099555504104 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 5012/trevisan)  Gid: ( 5000/det_user)
Access: 2022-01-07 11:00:28.001143767 +0100
Modify: 2022-01-07 11:01:29.860547368 +0100
Change: 2022-01-07 11:01:29.860547368 +0100
 Birth: -
```

Comandi per directory:

- **ls directory**: lista il contenuto
- **ll directory**: alias per **ls -lh directory**
- **mkdir directory** e **rmdir directory**: crea o rimuove una directory
- **find ...**: cerca all'interno di una cartella

- **tree directory**: stampa l'albero di file e cartelle contenuti
 - **du directory**: ottiene la dimensione della cartella e di tutto ciò che vi è contenuto
-

Domande

E' possibile creare link a cartelle

- **Di tipo Hard Link**
- **Di tipo Soft Link**
- **Sempre**
- **Mai**

Risposta: "*Di tipo Soft Link*"

Un Hard Link può riferirsi a un file inesistente

- **Vero**
- **Falso**

Risposta: "*Falso*"

Un Soft Link può riferirsi a un file inesistente

- **Vero**
- **Falso**

Risposta: "*Vero*"

Quale System Call permette di conoscere lo user ID del proprietario di una directory?

- **open**
- **opendir**
- **readdir**
- **stat**

Risposta: "*stat*"

La directory **dir** contiene i file **f.txt** e **g.txt**. Il seguente codice:

```
dp = opendir("dir1");
while ( (dirp = readdir(dp)) != NULL){
    if (strcmp(dirp->d_name, "dir/f.txt") == 0 )
        printf("%s\n", "Found\n");
```

Stampa:

- **"Found" una volta**
- **"Found" due volte**
- **Niente**

Risposta: *Niente*

u4-s5-raid

Osservazione preliminare per RAID e FDS: le problematiche dei sistemi professionali.

Problematiche dei Sistemi professionali

I Sistemi Professionali.

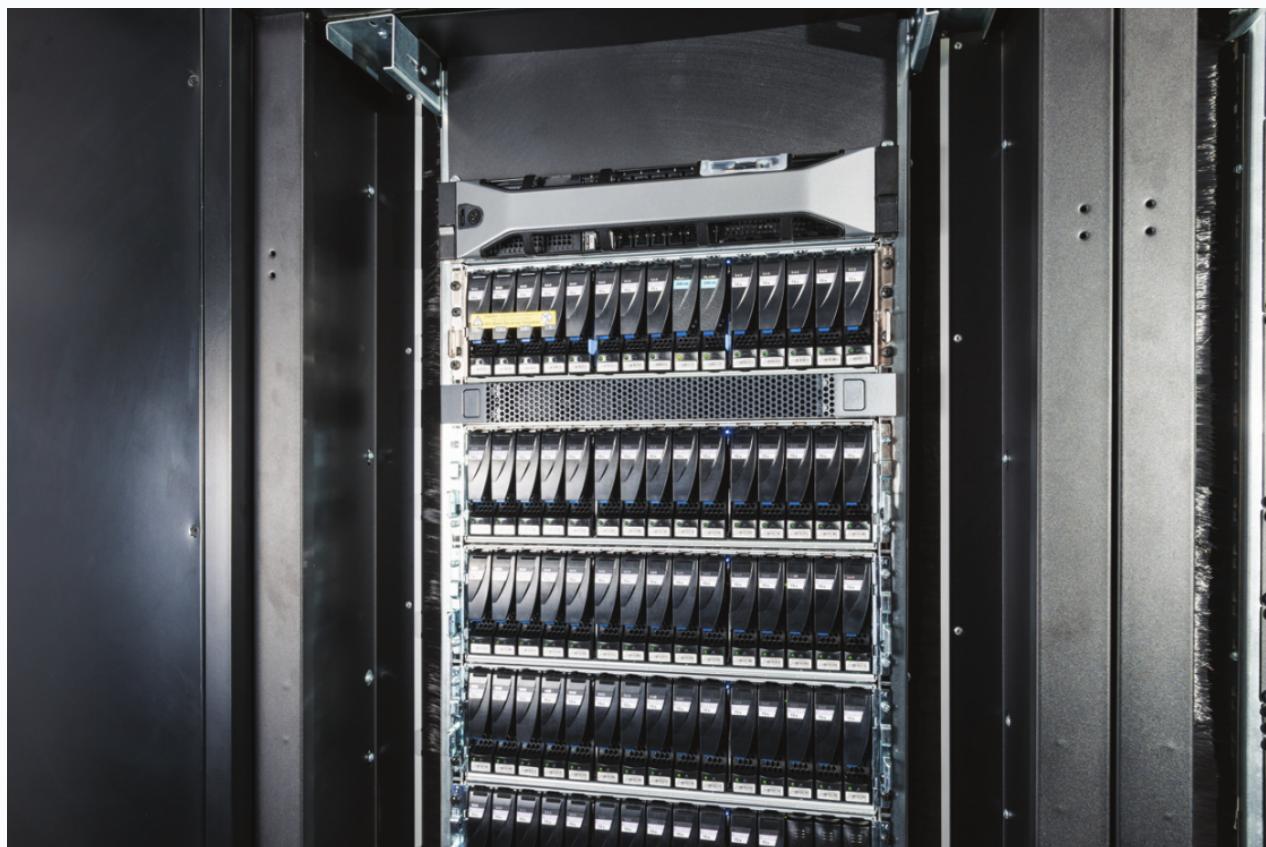
Nei sistemi di grandi dimensioni, una singola macchina ha tanti dischi (possiamo averne fino a 100!)

- Fino a 100 dischi su una stessa macchina
- Premettono di conservare *enormi quantità di dati*

Spesso ci sono server dedicati allo *storage*

- I calcolatori accedono *via rete* ai dati dello storage
- Tramite *protocolli di rete*

FIGURA. (*Esempio di rack storage*)



Allora seguono i seguenti *problem*i, da *gestire*.

Problemi.

1. Sono necessarie tecniche per gestire i *guasti* (*failure*)

- Un disco ha l'1% di probabilità di rompersi *ogni mese*
 - Dato reale per i dischi magnetici
- Se ho 100 dischi, ho in media *un guasto al mese*; nel senso probabilistico, dopo un certo periodo di tempo la *probabilità di avere almeno un guasto* tende a

salire verso 1 ([vedere la Scimmia di Borel](#)).

- Non è pensabile *perdere dati* in sistemi professionali!
- Voglio gestire gli errori in una *maniera automatica*

2. Sono auspicabili tecniche per *aumentare le prestazioni*

- Se 100 dischi vengono opportunamente usati in parallelo, possono moltiplicare $\times 100$ la *velocità* del sistema

-

Adesso vedremo *due soluzioni* per gestire le problematiche: da un lato avremo il *RAID*, e dall'altra i *FDS*.

Definizione di RAID, brevissimo excursus storico. Concetto essenziale di RAID: lo striping. Livelli di RAID: 0, 1, 4, 5, 6. Osservazioni conclusive su RAID.

Definizione di RAID

Le tecniche *RAID* ("*redundant array of independent disks*") hanno lo scopo di affrontare i problemi di *prestazioni e affidabilità*

- Proposto nel 1988 da David A. Patterson (e altri) nel paper *A Case for Redundant Arrays of Inexpensive Disks (RAID)* (questo era infatti il nome storico)
- Famiglia di metodi per organizzare dati su *batterie di dischi*
- Molto comunemente utilizzato, sia in ambienti professionali che personali.

Concetti Essenziali di Raid

Definizione. (*striping*)

Si basa su *striping*, ovvero distribuire i dati su N dischi. Possiamo farlo in due modi:

- A livello di *bit*: il disco i -esimo contiene i bit $n \mid n \bmod N = i$ (ad esempio i bit pari vanno su disco 1, i bit dispari sull'altro disco)
- A livello di *blocco*: il disco i -esimo contiene i blocchi $n \mid n \bmod N = i$ (la più *comune*)
Ciò migliora le prestazioni, permettendo *lettura parallela*

Definizione. (*codice di parità*)

Eventualmente con l'aggiunta di *codici di parità*

- Sono dei codici "*backup*" che servono per rilevare errori effettuando somme. In grado di ripristinare i dati, prendendo il complementare.
- Per essere *fault-tolerant*
- Non si perdono i dati in caso di guasti di un disco

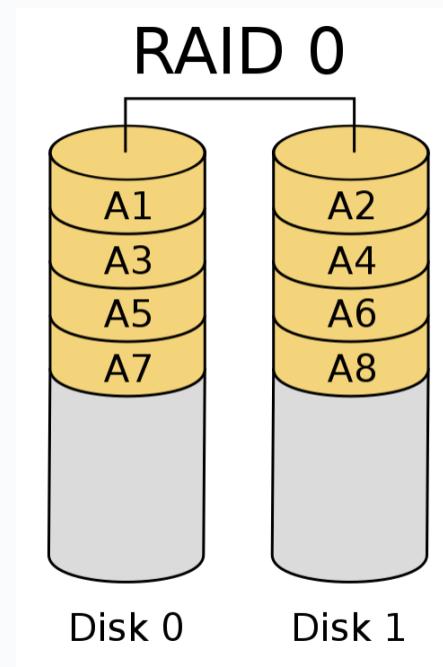
Livelli di RAID

Ci sono *diverse configurazioni o schemi di dischi possibili*, detti "*livelli*". Differiscono da:

- A seconda che offrano aumenti di *prestazione o affidabilità*
- *Numero minimo di dischi richiesto*
- *Robustezza a guasti* multipli
- Alcune tecniche sono intuitive, altre meno.

RAID 0. (*Selezionamento*)

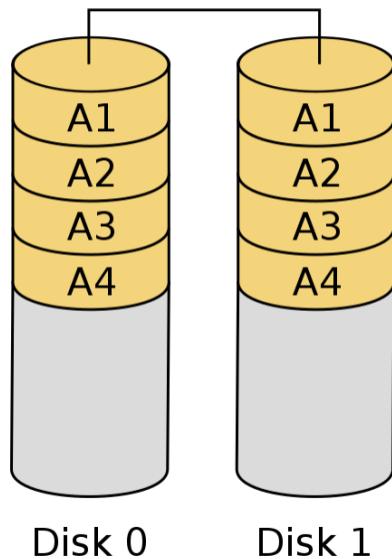
- **Idea:** I dati sono *divisi* tra i dischi tramite *striping* (a livello di blocco di solito)
- **Minimo numero di dischi:** ≥ 2
- **Vantaggi:** Alta velocità sia di lettura che di scrittura grazie ad accessi paralleli. Come vedremo, questo è l'*unico vantaggio* di RAID 0.
- **Svantaggi:** Decresce affidabilità del sistema; con un guasto, ho perso tutti i dati!
- **Casi d'uso:** Traffici di rete ad alta velocità (come 10Gb/s)



RAID 1. (*Mirroring*)

- **Idea:** I dati sono *replicati* su più dischi. L'opposto di *RAID 0*.
- **Minimo numero di dischi:** ≥ 2
- **Vantaggi:** Con N dischi, resiste a $N - 1$ guasti. Alte prestazioni di lettura.
- **Svantaggi:** Bassa *velocità di scrittura* limitata dal disco più lento. Prendi infatti il *minimo della velocità* tra i dischi.
- **Casi d'uso:** Server dove è necessaria una buona resistenza a guasti, oppure una alta velocità di lettura.

RAID 1

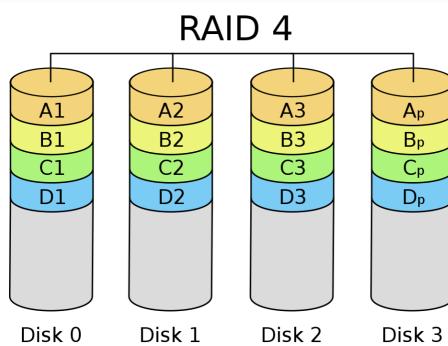


RAID 2, 3.

- Ignorate, dato che sono assolutamente obsolete. Per un approfondimento vedere la pagina Wikipedia ([Link](#))

RAID 4. (*Disco di parità*)

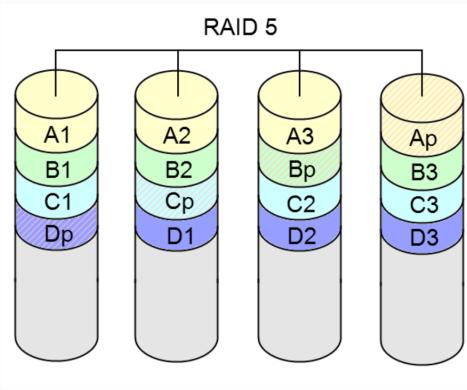
- **Idea:** Il disco N memorizza la parità dei dati sugli altri $N - 1$ dischi
- **Minimo numero di dischi:** ≥ 3 . Due (o $n + 1$) di dato più parità
- **Vantaggi:** Resiste a un guasto e permette letture parallele (quindi alte prestazioni di lettura).
- **Svantaggi:** Scrittura lenta. Necessario calcolare e scrivere parità. Inoltre, sollecitando sempre l'ultimo disco per la scrittura della parità, rischio sia di avere il "bottleneck" (letteralmente ingorghi) e di incrementare la probabilità del guasto.
- **Casi d'uso:** Non si usa tanto. Si userà una sua variante più furba, la RAID 5.



RAID 5. (*Parità distribuita*)

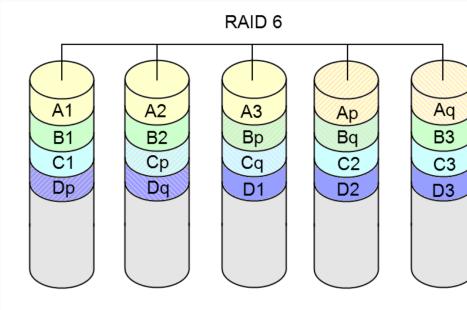
- **Idea:** Come RAID 4, ma codici di parità distribuiti su tutti i dischi equalmente
- **Minimo numero di dischi:** ≥ 3
- **Vantaggi:** Sostanzialmente ci sono solo vantaggi rispetto a RAID 3: resiste a un guasto. Scritture più veloci di RAID 4: non è necessario accedere sempre a disco di parità, rimuovendo eventuali bottleneck.

- **Svantaggi:** Scrittura comunque lenta (a causa di parità)
- **Casi d'uso:** Molto usato in sistemi reali



RAID 6 (*Doppia parità distribuita*)

- **Idea:** Codici di parità memorizzati due volte. Tra tutti i dischi. Ovvero, per ogni riga *dedico due dischi per la parità*.
- **Minimo numero di dischi:** ≥ 4
- **Vantaggi:** Resiste a *due guasti*
- **Svantaggi:** Scrittura molto lenta (a causa di doppia parità)
- **Casi d'uso:** Molto usato in sistemi reali, in particolare dove *abbiamo dati sensibili*, quindi abbiamo bisogno di una garanzia maggiore.



RAID: Conclusioni

Riassunto RAID.

Gli schemi RAID permettono di migliorare *prestazioni e affidabilità* quando si hanno molti dischi su una stessa macchina

Limiti della RAID.

1. Non proteggono da un *failure completo della macchina*: eventualmente alcuni livelli proteggono le *failure dei dischi*. Come ad esempio abbiamo
 - *Temporaneo*: manca la corrente
 - *Permanente*: si rompe la scheda madre
- Ciò non è accettabile per servizi *mission-critical* (esempio: le *ASL*)
2. Le tecniche RAID *non scalano*:

- C'è un *massimo numero di dischi collegabili* a una macchina
- Il *BUS PCI ha un limite* (di circa 36 Gb/s)

Per sistemi *molti grandi* (quindi ancora più grossi) si usano *File System distribuiti*

Soluzione FDS: Introduzione (collegamento via rete), approcci per la FDS (NFS, NBD). Definizione di File System Distribuito, software orchestratore. Tecnologie attuali di FDS: HDFS e CEPH.

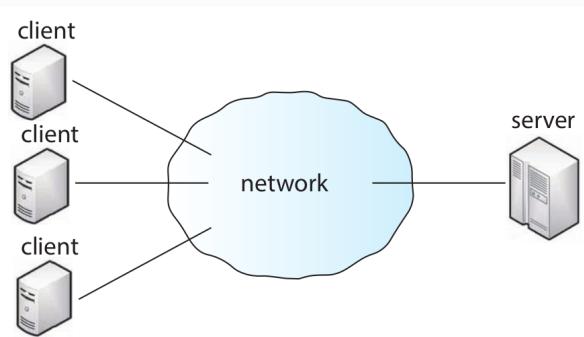
File System Distribuiti: Introduzione

Problema. (*File System Distribuiti*)

E' possibile usare un FS che *si trova su un'altra macchina*

- Tipicamente un *server dedicato allo storage*
- Si utilizzano *protocolli dedicati*
 - **Network File System (NFS)**: il più *usato e flessibile*
 - **Samba**: Microsoft (anche se di dubbia reputazione)
 - **File Transfer Protocol (FTP)**: obsoleto, purtroppo in certi casi ancora utilizzata
 - **Fiber Channel**: per grandi *Storage Area Network* (in declino, oggi si preferisce usare i file server normali)

Figura. (*L'idea del FSD*)

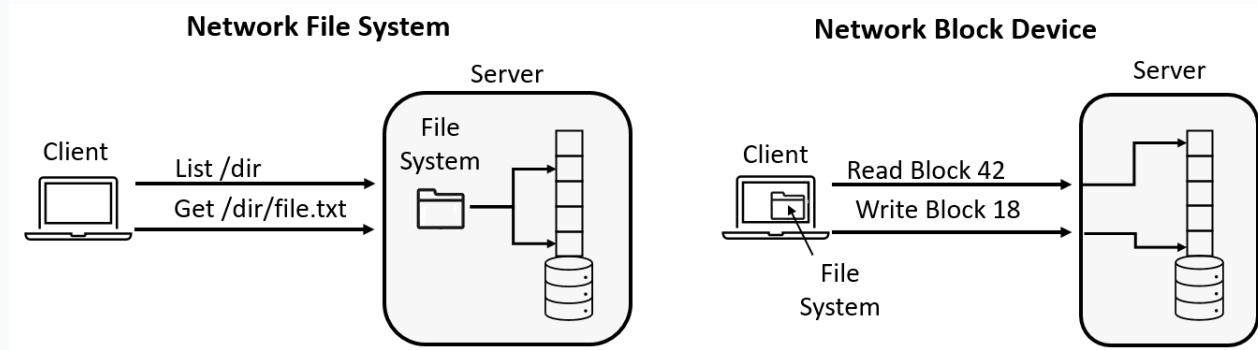


Approcci per i File System Distribuiti

Abbiamo *due approcci* per i *file system distribuiti*, che rispondono alla domanda: "*Dove poniamo il File System*"? Quindi questi approcci sono concettualmente diversi:

- *File System di Rete* (o Network File System): FS gira sul server
 - *Dispositivi a Blocchi di Rete* (o Network Block Device): FS gira sul client
- Vedremo che ognuna ha i suoi vantaggi e svantaggi.

Figura. (L'idea del NFS e NBD)



Network File System

- **Vantaggi:** Possono collegarsi più *client in parallelo*, possiamo svolgere *operazioni di alto livello*..
- **Svantaggi:** Carico troppo sul *file system del server*, soprattutto quando ho più *clienti in parallelo*. Problemi di scalabilità.

Network Block Device

- **Vantaggi:** Possiamo svolgere *operazioni di basso livello*, come ad esempio decidere quale file system usare. Necessario in certi casi, come ad esempio usare il database *MYSQL*.
- **Svantaggi:** Può girarci *solo ed esclusivamente solo* un client.

File System Distribuiti: Definizione

Definizione. (*File System Distribuito, software orchestratore*)

Un *File System Distribuito* è un file system che risiede *su più dischi su macchine diverse*

- E' necessario un software *orchestratore*
- Per far sì che l'utilizzatore ne fruisca come un unico FS
 - Quindi è *molto complesso*, dato che deve dare l'*illusione* di essere un *unico FS*.

Un FS distribuito:

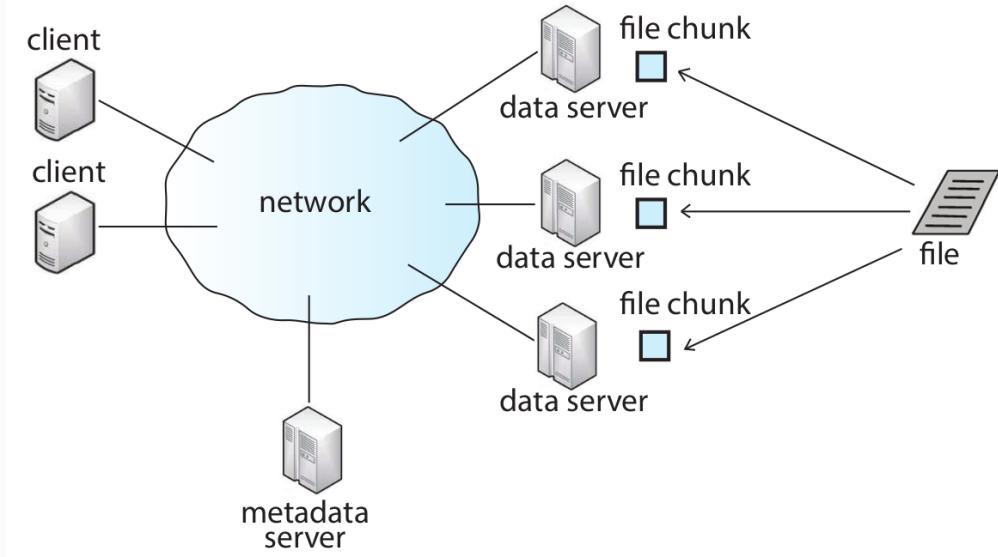
- E' visto da utilizzatori come un unico FS *grande e affidabile*
- Vi si accede tipicamente come disco di rete (è un File System di Rete)

Modello. (*Client-Server*)

Basati su modello *client-server*

- Client consulta il *metadata server* per listare directory e ottenere *informazioni sui file*
- Client accede al *contenuto da uno o più data server*

Figura. (L'idea)



Tecnologie per FS distribuiti

I FS distribuiti si installano con *software di orchestrazione* dedicati

- Organizzano i dati nei vari dischi e nodi
 - Replicano i dati per aumentare le prestazioni
 - Recuperano i dati quando un utilizzatore vi accede
- Adesso ne studiamo alcuni.

1. Hadoop Distributed File System (HDFS)

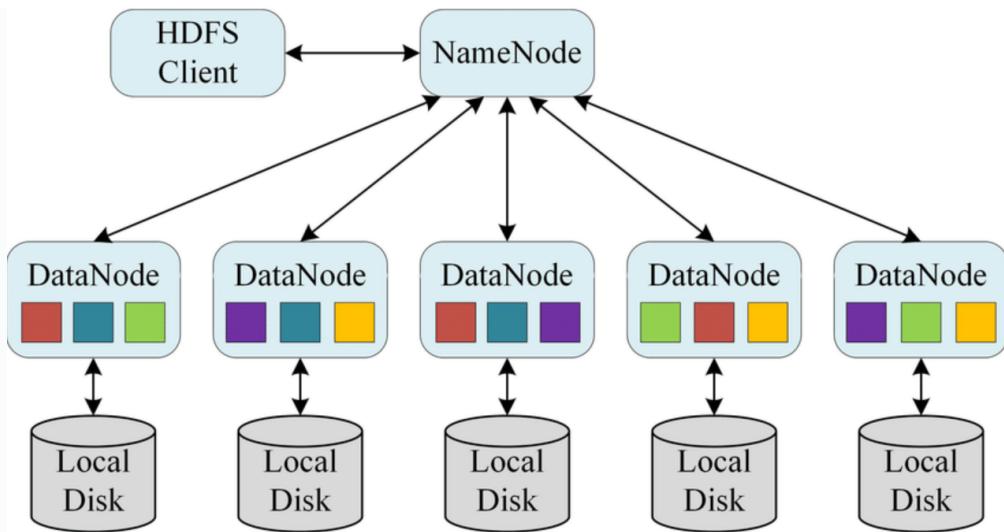
Parte della suite *Hadoop per Big Data*. E' un FS distribuito

- Si installa su un *cluster* (insieme) di server/nodi
- I *Name Node* hanno l'indice dei *file*
- I *Data Node* memorizzano il contenuto dei *file*
- Tutto viene replicato N volte

Vantaggi: Funziona benissimo, infatti viene utilizzato quasi da tutti (banche, eccetera...)

Svantaggi: Problemi di scalabilità: infatti in realtà HDFS è in declino e sta vivendo di "*rendita aziendale*", ovvero viene usata solo perché le aziende non possono o non vogliono cambiare la tecnologia per FSD.

Figura. (*Schema HDFS)

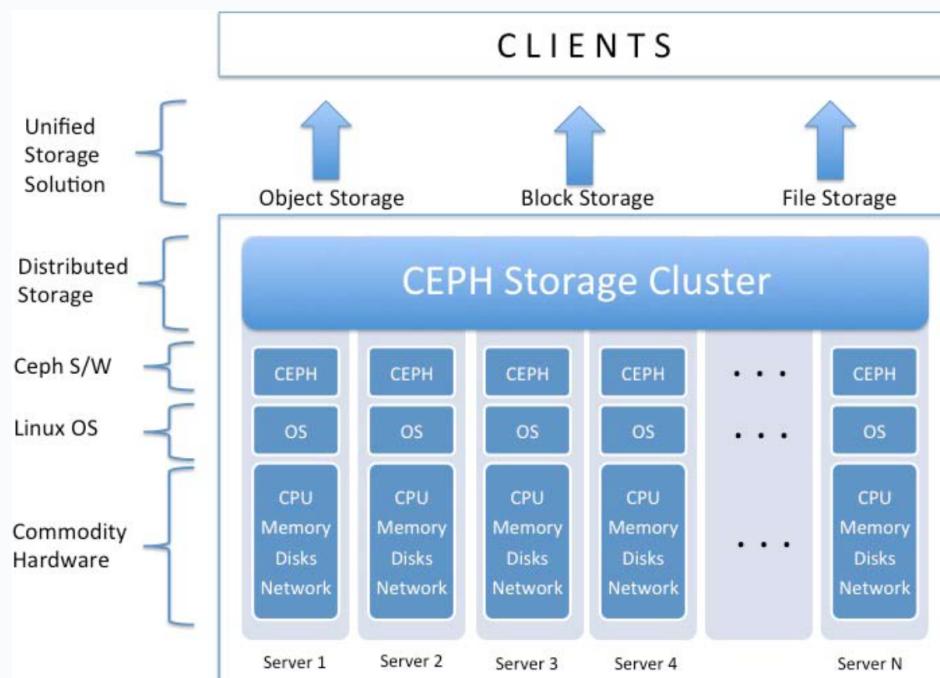


2. CEPH

Concettualmente simile a HDFS. Si usa su cluster di nodi. Implementa nuovi concetti più moderni, tra cui:

- **FS distribuito:** i client accedono a *file e cartelle*
- **Dispositivo a blocchi:** i client vedono *disco grezzo a blocchi*
- **Object storage:** i client accedono a *bucket generici* identificati da ID
- **Vantaggi:** Molto flessibile, infatti permette di avere sia *Newtork FS* che *Network BD*. Inoltre, abbiamo già implementato un *database molto efficiente*. La tecnologia **CEPH** è "in rising".

Figura. (CEPH)



Domande

Un sistema di dischi basato su RAID è sempre più affidabile di un disco singolo?

- Si
- No

In un sistema RAID 0, quali sono le conseguenze in caso di fallimento di un disco?

- I dati vengono persi
- E' possibile recuperare i dati

In un sistema RAID 1, quali sono le conseguenze in caso di fallimento di un disco?

- I dati vengono persi
- E' possibile recuperare i dati

E' possibile creare un sistema RAID 6 con 3 dischi?

- Si
- No

Qual è il sistema di accesso tipico a un FS Distribuito?

- Bus PCI
- Rete
- USB

u5-s1-processi

Sistemi Operativi

Unità 5: I processi

Aspetti Teorici

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Concetto di processo
 2. Stati di un processo
 3. Process Control Block
 4. Scheduling
 5. Algoritmi di Scheduling
-

Concetto di processo

Definizione. (*Processo*)

- Un elaboratore svolge uno o più *compiti*
 - Esempio: Controllare la temperatura di una stanza
- Un *compito* si svolge tramite un procedimento formale detto *algoritmo*
- Un programma implementa un algoritmo *tramite istruzioni* in linguaggio macchina (passaggio astratto→concreto)
 - Può essere scritto in un linguaggio di programmazione e *compilato*
- Un *processo* è un *programma in esecuzione*

Nota storica. (*Pre-S.O. e con S.O.*)

Inizialmente, ogni elaboratore *eseguiva un programma per volta*, senza *sistema operativo*.

- Caricato all'*avvio del sistema*

- Oppure eseguiti *sequenzialmente* (batch processing)
In ogni caso, *mai* venivano eseguiti in *contemporanea*.
Oggi i sistemi moderni hanno un SO che permette di *eseguire più processi in contemporanea*
- Le risorse del sistema sono gestite *dal SO* che le mette a disposizione tramite le System Call
- Il SO gestisce l'esecuzione dei processi: *scheduling*
- I processi vengono eseguiti a turno

Struttura del Processo

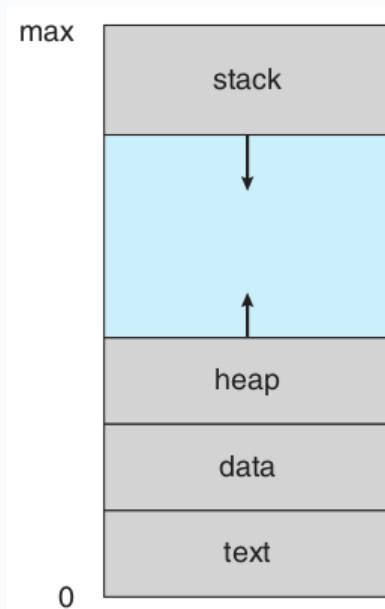
Struttura del processo in memoria.

Un processo risiede *in memoria*, da allocare.

La struttura di un processo in memoria è generalmente suddivisa in *più sezioni*.

- **Sezione di testo:** contiene il *codice eseguibile*
- **Sezione dati:** contiene le *variabili globali*
- **Stack:** memoria *temporaneamente utilizzata* durante le chiamate di funzioni
- **Heap:** memoria allocata *dinamicamente* durante l'esecuzione del programma (ξ)

FIGURA. (*Struttura di un processo da allocare*)



Tipologie di Processi

Principalmente abbiamo due tipi di processi eseguiti.

1. Definizione (*Processo INIT*)

Nei moderni SO, all'avvio del sistema viene avviato un processo fondamentale detto *init*

- Per *Linux* o sistemi *POSIX-like*
- Eseguito fino allo shutdown (+∞)
- Eseguito automaticamente dal *kernel*

- Ha PID 1 (vedremo dopo)
- Il processo `init` avvia altri processi (tramite svariati file di configurazione) in background:
- Per gestire *periferiche*: rete, antivirus
 - Per creare l'*interfaccia grafica* della GUI o del terminale

Definizione. (*Servizio*)

Un processo avviato in background da `init` è detto *Servizio*

- Formati e comandi diversi tra distribuzioni Linux per gestirli
- Comandi: `service` o `systemctl`

2. Definizione. (*Processo utente*)

L'utente può *creare dei processi* - mediante terminale o interfacce grafiche, che a loro volta usando delle System Call - per svolgere i propri compiti

- Browser
- Editor
- Programmi server: server Web, server DNS
- Eccetera...

Ruolo del Sistema Operativo nei Processi

Ruolo del S.O. per i processi.

Il SO mette a disposizione delle *System Call* per:

1. *Creare* nuovi processi
2. *Sincronizzazione*: Attendere il completamento di altri processi per coordinare un compito complesso
3. *Uccidere* processi

Definizione. (*PID*)

I processi sono identificati dal un *PID*

- Il processo `INIT` ha PID 1 per definizione
-

La Vita di un Processo

Stati di un Processo

Un processo si può trovare in diversi stati.

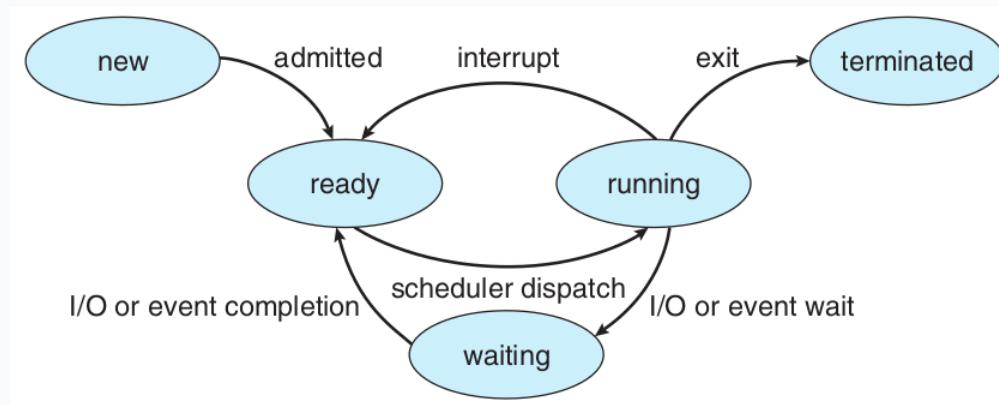
1. *Nascita* (new): è stata richiesta la creazione di un processo mediante un *system call*. Il sistema operativo vede se la richiesta sia *"fattibile o meno"* e ben posta: in tal caso, viene ammessa (admitted).
2. *Pronto* (ready): il processo è pronto per l'esecuzione, ma non è ancora in *esecuzione*.

Aspetta che lo *scheduler* del S.O. dia il turno al processo.

3. **In esecuzione** (running): Il processo fa le sue cose. Da questo punto possiamo avere tre esiti:

1. Il processo ha *finito il suo lavoro* (o qualcosa gli costringe di farlo) ed esce. In tal caso è in stato di "*morte*" (terminated). Ho messo le virgolette, perché il processo starebbe aspettando che la sua morte venga "*confermata*" dal suo padre; il processo non è completamente morto.
2. Lo scheduler del sistema operativo lo mette in pausa e lo *interrompe* (interrupt), facendolo ritornare allo stato di *ready*.
3. Ho una *system call* lenta e complessa (come quella di leggere un disco magnetico), e il processo viene bloccato. Allora il processo viene messo in *attesa* (waiting), finché ciò che lo blocca viene completato. Quando viene completato, viene rimesso in *ready*.

FIGURA. (*Schema della vita di un processo*)



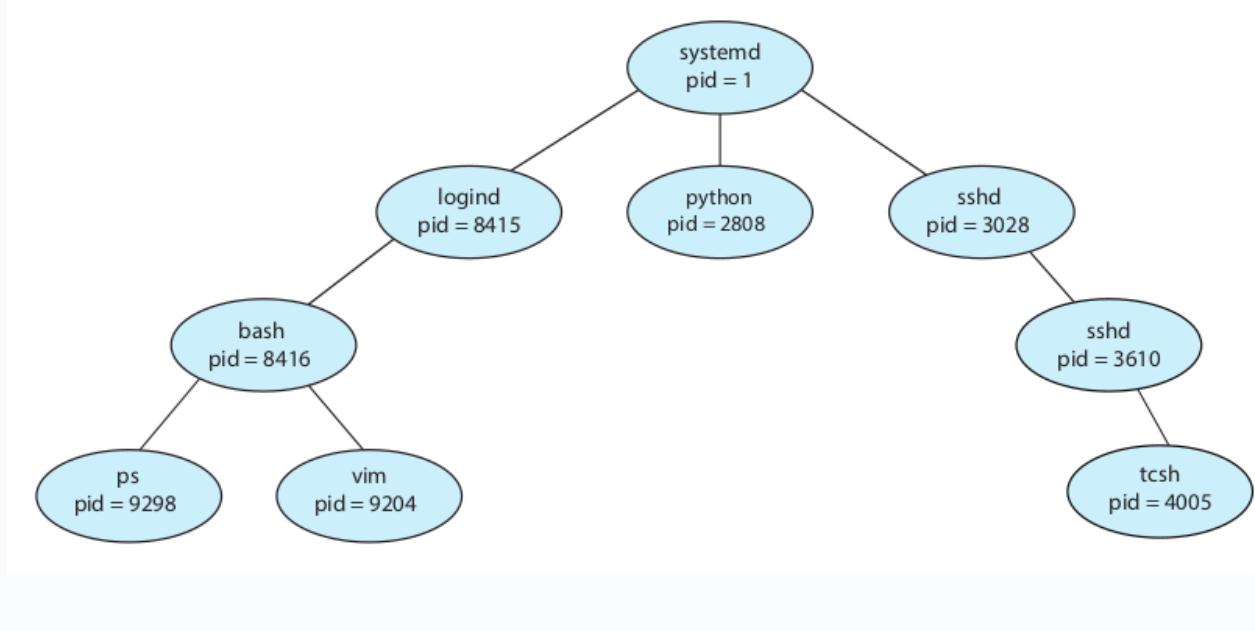
L'albero dei processi

Usando le System Call, un processo può creare un altro processo. Infatti, tutti i processi sono figli di qualcuno (tranne *INIT*)

- Il processo generato è *figlio* del processo generante
- Si crea un *albero dei processi*
- Se il processo padre termina, i figli *NON* vengono terminati
- I processi *senza padre* diventano figli del processo *init*

Esempio di albero dei processi:

Nota: su alcuni sistemi **init** è rinominato **systemd**



Process Control Block

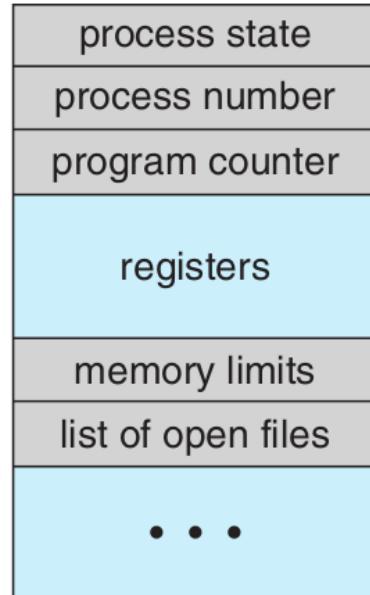
Ogni processo è rappresentato nel sistema operativo da un *blocco di controllo* (process control block, PCB) contenente le informazioni connesse. Diventa un'equivalente dell *inode* per i file (1).

Definizione. (*Process Control Block*)

Un *PCB* (Process Control Block) è una struttura dati, contenente dati rappresentativi per un *processo*. Le informazioni registrate sono le seguenti.

- **Stato del processo:** nuovo, pronto, esecuzione, attesa, arresto
- **Program counter:** indirizzo della successiva istruzione da eseguire
- **Registri della CPU:** permettono di interrompere il processo
 - Fondamentale per *interrompere processi*, ovvero *bloccare stato del processo*.
Stesso discorso per il PC
- **Informazioni di scheduling:** priorità, risorse consumate. Specificate dall'utente
- **Informazioni sulla gestione della memoria:** puntatori alle varie zone di memoria.
- **Informazioni di I/O:** file aperti, operazioni in attesa, ecc...
- Eccetera...

FIGURA. (*PCB*)



Scheduling

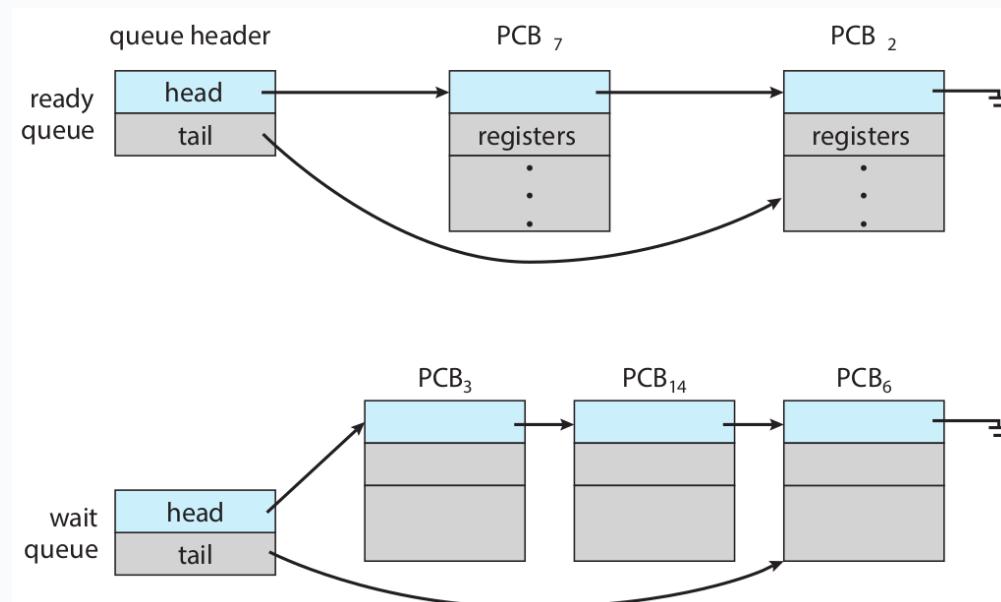
Scheduler dei Processi

Lo scheduler dei processi ha più ruoli.

Ruolo 1.

Lo *scheduler dei processi* seleziona un processo da eseguire dall'insieme di quelli disponibili

- Mantiene una coda dei *processi pronti*
- Mantiene una coda dei *processi in attesa di evento*. Esempio: completare un'azione di I/O



Idea. (*Classificazione dei processi*)

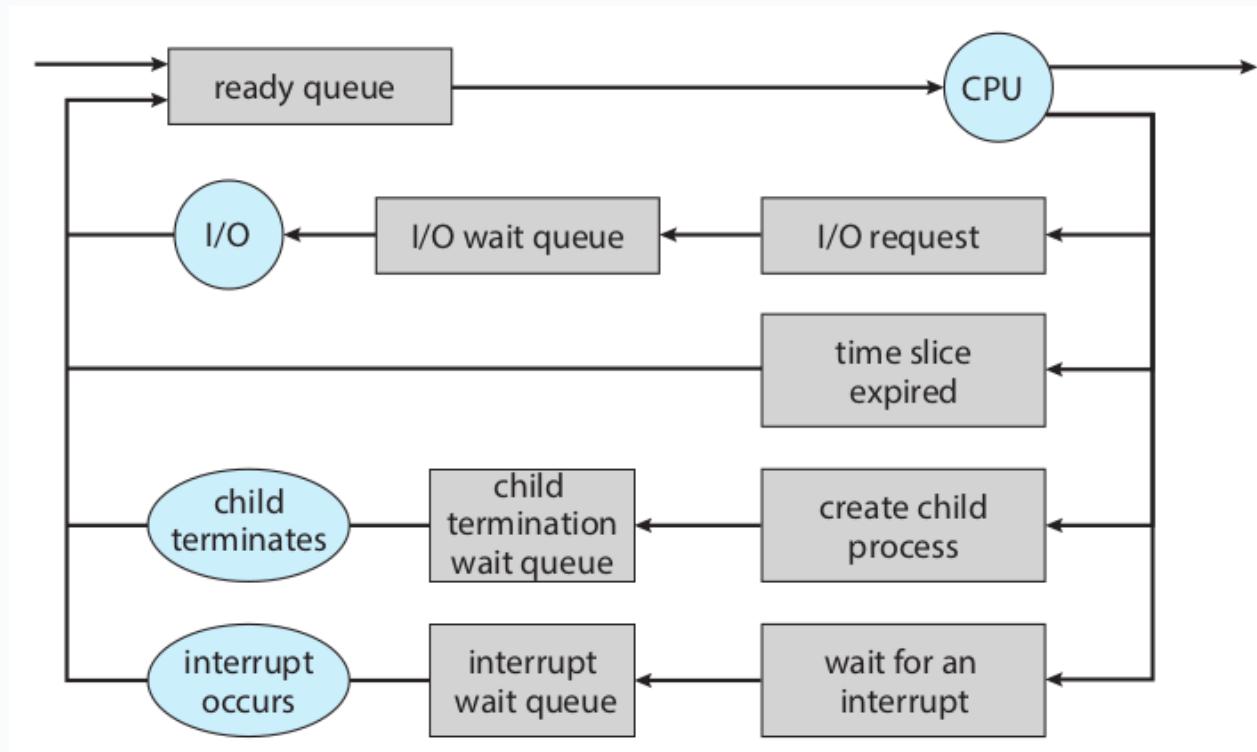
I processi possono essere classificati *in base al tipo di carico* che generano e il collo di bottiglia che li limita.

- **Processo I/O bound:** impiega la maggior parte del proprio tempo nell'*esecuzione di operazioni di I/O*
- **Processo CPU bound:** impiega la maggior parte del proprio tempo nelle *elaborazioni*

Il compito di uno *scheduler* è di *ottimizzare* l'esecuzione dei processi per farli eseguire nel minor tempo possibile

- Interviene più volte al secondo
- Gestisce l'esecuzione col meccanismo del *time sharing*

Ogni processo inizia dalla *Ready Queue* e segue il *diagramma di accodamento* finché non termina



Ruolo 2. + Definizione. (*Context Switching*)

Lo scheduler decide a quale processa assegnare la/le CPU

Quando decide che (una) CPU deve essere assegnata a un altro processo, esso deve:

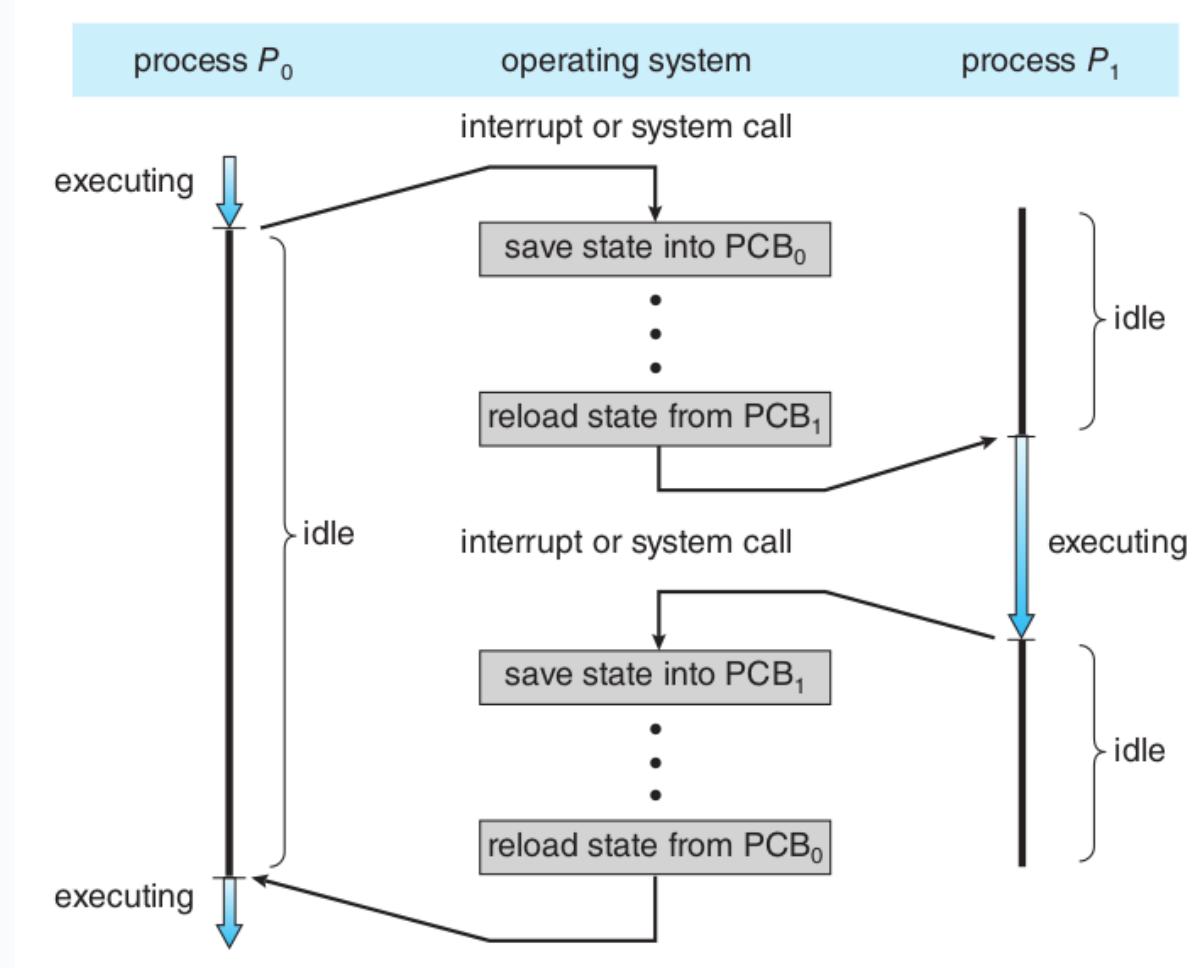
- *Salvare lo stato del processo corrente*
 - Per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione
- *Caricare un nuovo processo* ripristinandone lo stato salvato precedentemente
- Fa molte operazioni sui registri della **CPU**
Si esegue un salvataggio dello stato e, in seguito, un corrispondente ripristino dello stato, detto **Context Switching**

Nota. (Context Switching e l'efficienza)

Il Context Switching deve essere rapido, siccome è **tempo sprecato**, che non svolge nessun compito utile

- Il SO è ottimizzato per compiere questa azione **velocemente**
- Attualmente nell'ordine di pochi **micro secondi**
- Dipende da hardware e dalle caratteristiche del processo, specialmente la quantità di memoria usata
- Di solito facciamo **context switching** circa una migliaia di volte al secondo.

Idea Grafica.



- Gli scarti temporali "**idle**" a destra sono **tempo perso!**

Operazione Yield

Operazione di Yield (dall'inglese "dare precedenza ∇ ") : un processo dice al kernel, che per il momento **non ha operazioni da fare**.

Il kernel rimuove dalla CPU il processo e lo **riaccoda nella lista dei processi pronti** (stato **ready**)

- E' un modo per rilasciare la CPU prima che scada il quanto di tempo assegnato

- Specialmente usato per processi *real time*, per avere maggior *prestazioni*

In Linux:

```
#include <sched.h>
int sched_yield(void);
```

Algoritmi di Scheduling

Problema Preliminare

Richiamo. (*L'obiettivo dello scheduler*)

L'obiettivo di un SO è di ridurre il *tempo di esecuzione dei processi*. Tuttavia abbiamo dei problemi con questi obiettivi.

- Obiettivo *ambiguo*: conviene eseguire prima un lavoro lungo o uno corto?
- Obiettivo *complesso*: il SO non sa se un processo è lungo/corto, *CPU/I/O intensive*. Praticamente, neanche l'uomo potrebbe delineare un confine netto tra questi due tipi di processi.

Conseguenza.

Esistono diversi *Algoritmi di Scheduling* che si usano per determinare quale processo assegnare a una CPU

First-Come First Served (FCFS)

Idea Il primo processo che richiede la CPU, la ottiene *finché non termina*

Pro Semplice!

Contro: Inefficiente (!!). Sconveniente eseguire un *processo lungo* per primo

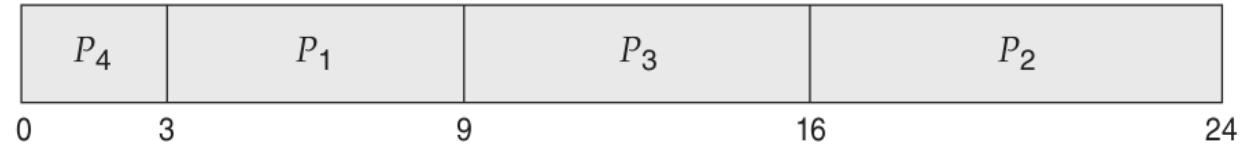


Shortest-Job First Served (SJFS)

Idea Il primo più breve, la ottiene la CPU per primo

Pro Efficiente (!) Il tempo medio di completamento si abbassa

Contro: Inattuabile (!!!): non si sa quanto dura un processo, impossibile determinarlo a priori.

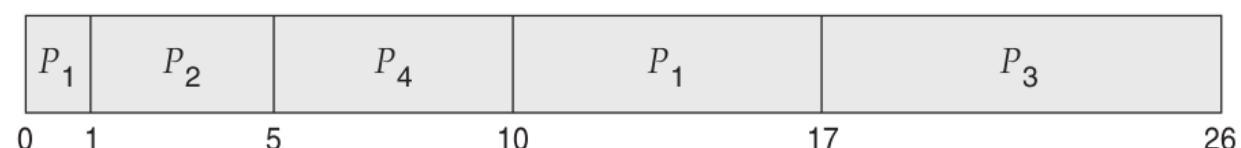


Round Robin (RR)

Idea A turno, ogni processo prende la CPU *per un tempo fissato*

Pro Semplice ed equo, il tempo fissato è lo stesso per tutti

Contro: Non si possono avere processi ad alta priorità. Forse *troppto* equo

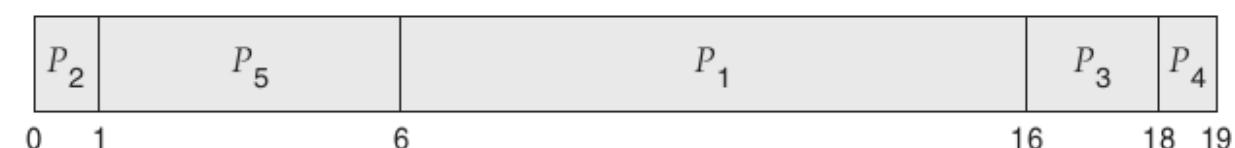


Priority Scheduling

Idea Ogni processo ha una *priorità data dall'utente*. Viene eseguito il processo a priorità più alta

Pro Gestisce la priorità

Contro: Un processo a bassa priorità *potrebbe non venire mai eseguito*: funziona bene solo se l'utente definisce bene le priorità



Multi-Level Queue Scheduling (MLQS)

Idea. Ci sono *code diverse* per ogni livello livello di priorità.

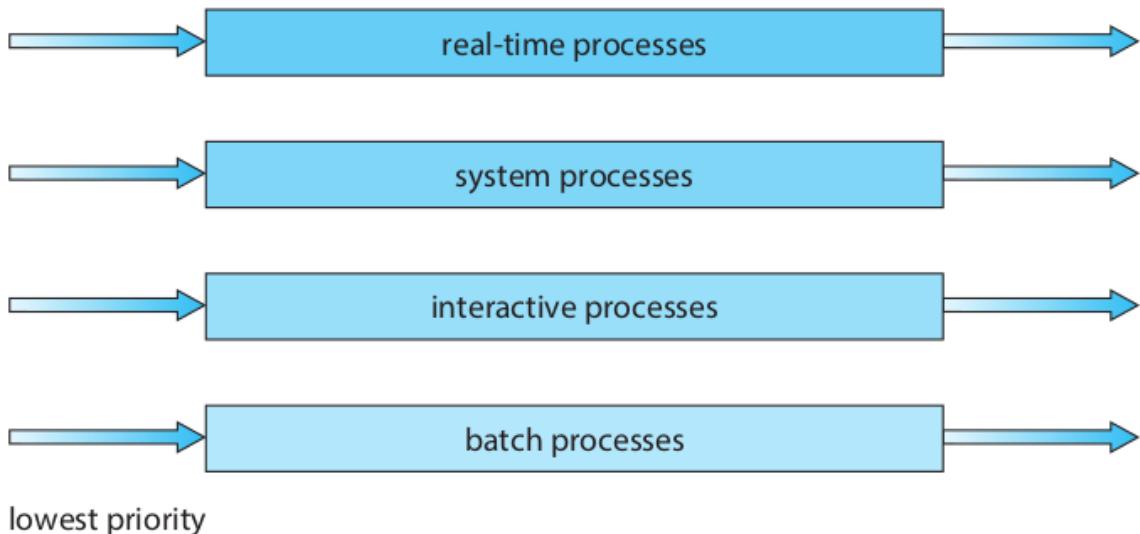
- Ogni coda ha un suo *algoritmo di scheduling*: RR, FCFS
- C'è un *algoritmo di scheduling tra code* (intra-coda)
In questo modo c'è *flessibilità*: si può avere priorità ma non c'è il rischio che un processo non venga mai eseguito

Pro Flessibile, furbo e idealmente semplice

Contro: Complesso, un casino da implementare con molti algoritmi e code

Usato in Linux, con varianti

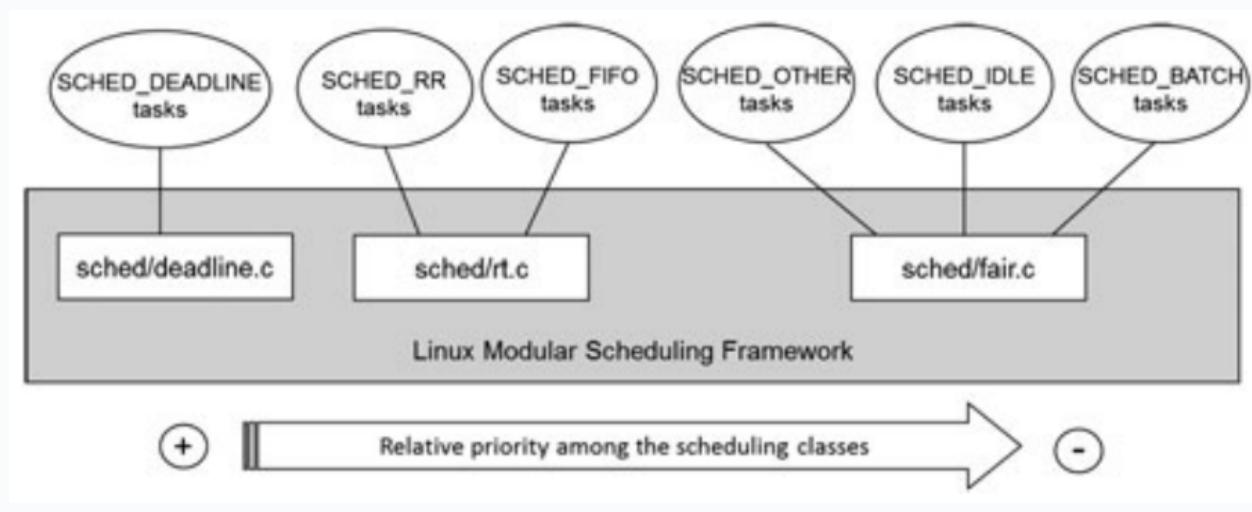
highest priority



Linux: Completely Fair Scheduler (CFS)

In Linux lo scheduler si chiama *Completely Fair Scheduler*

- I processi sono assegnati a una *Policy di Scheduling* dall'*utente*, ognuna con *meccanismi diversi*
- Il *sistema* provvede a eseguire processi in ogni *policy*, che hanno diversi requisiti
- Le *policy* al loro interno possono gestire *priorità, deadline, ecc...*
 - Ad esempio, **SCHED_DEADLINE** contiene i processi che hanno la garanzia di *essere eseguiti per un certo intervallo di tempo*.
 - Di default ho **SCHED_OTHER**, con *Round Robin* + priorità
 - Le system call per definire priorità sono
 - nice(2) getpriority(2) setpriority(2) sched_setscheduler(2) sched_getscheduler(2) sched_setparam(2) sched_getparam(2) sched_yield(2)**



Domande

Un processo è

- **Una componente del sistema operativo**
- **Il codice eseguibile di un programma**
- **Un programma in esecuzione**

Risposta: Un programma in esecuzione

I processi sono identificati da:

- **Un nome**
- **Un ID numerico**
- **Non hanno identificativi**

Risposta: Un ID numerico

I processi sono in relazione tra loro in una struttura:

- **Ciclica**
- **Ad albero**

Risposta: Ad albero

Il Context Switching è:

- **La momentanea sospensione di un processo**
- **La terminazione di un processo**

Risposta: La momentanea sospensione di un processo

Nello scheduler di Linux, processi:

- **Hanno tutti la stessa priorità di scheduling**
- **Hanno una priorità assegnabile dall'utente**
- **Hanno una priorità calcolata automaticamente dal SO**

Risposta: Hanno una priorità assegnabile dall'utente

u5-s2-operazioni

Sistemi Operativi

Unità 5: I processi

Operazioni sui processi

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

Principalmente vedremo le *System Call* per gestire i processi. Ce ne sono poche, ma atomiche.

1. Creazione di un processo

2. Funzione **fork**
 3. Funzione **wait**
 4. Funzione **exec**
 5. Funzione **system**
 6. Funzione **exit**
 7. Altre funzioni
 8. Comandi Bash per Processi
-

Manipolazione dei Processi

In un SO, la manipolazione dei processi è effettuata tramite System Call

In Windows:

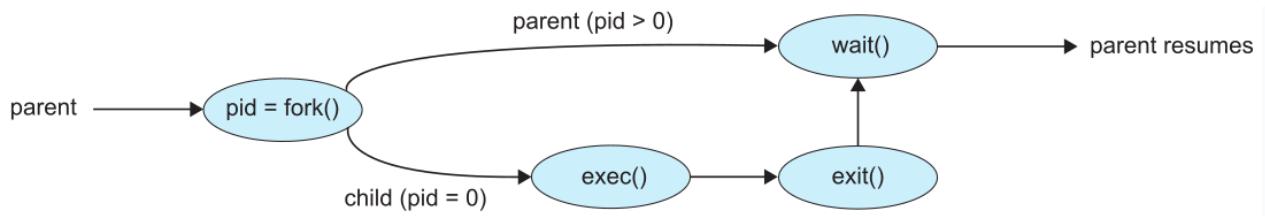
```
BOOL CreateProcessA(  
    LPCSTR      lpApplicationName,  
    LPSTR       lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL        bInheritHandles,  
    DWORD       dwCreationFlags,  
    LPVOID      lpEnvironment,  
    LPCSTR      lpCurrentDirectory,  
    LPSTARTUPINFOA   lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
) ;
```

Molto complessa!

In Linux: esistono 6 System Call principali

- **fork**: crea un *processo duplicato*
- **exec**: carica un *codice eseguibile* (esegue un altro programma)
- **wait**: aspetta la *terminazione di un figlio* (importante per la sincronizzazione!)
- **kill**: invia un segnale (uccide/termina processi)
- **signal**: cattura un segnale
- **exit**: *termina* il processo corrente

FIGURA. (*Esempio di chiamate di System Call*)



Confronto tra Windows e Linux

Windows vs Linux:

Windows ha una System Call complessa (**CreateProcessA**)

- Molto verboso
- Molti parametri
- Molto tipizzata

Linux preferisce System Call *semplici* e sintetiche:

- **fork** clona un processo
- **exec** permette di eseguire un file eseguibile nel processo corrente

Funzione **fork**

Definizione di Fork

Definizione.

```
#include <unistd.h>
pid_t fork (void);
```

C

Crea un nuovo processo figlio, copiando completamente l'immagine di memoria del processo padre (data, heap, stack)

- I due processi evolvono *indipendentemente*
- La memoria è *completamente indipendente* tra padre e figlio
- Il codice viene generalmente *condiviso tra padre e figlio*
 - Codice copy-on-write (copiato quando viene modificato)

Nota: **pid_t** è un alias per un **int**, come **size_t**

Note tecniche.

- Tutti i *descrittori dei file aperti* nel processo padre sono *duplicati nel processo figlio*

- Sia il processo child che il processo parent continuano ad eseguire *l'istruzione successiva alla fork*
- Valore di ritorno:
 - Processo *figlio*: 0
 - Processo *padre*: PID del processo figlio
 - Errore della fork: PID negativo (solo padre). Di solito non accade
- Si può sfruttare il valore di ritorno per *differenziare* il processo padre e figlio.

Osservazioni:

Il valore di ritorno della **fork** è fondamentale!

Un programma scritto in termini di **fork** non è immediatamente comprensibile

- Operazione atomica, ma *effetti complessi*
- E' possibile creare *alberi di processi complessi*, con codice complesso

Esempio di utilizzo: *Scrivere un programma che si duplica, ed esegue rami di codici diversi.*

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0){
        printf("Sono il figlio!\n");
    }
    else{
        printf("Sono il Padre!\n");
    }
    return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void Figlio(void);
void Padre(void);
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
        Figlio();
    else
        Padre();
}
void Figlio(void)
{
    int i=0;
    for(i=0;i<10;i++){
        usleep(200);
        printf("\tSono il figlio. i= %d\n",i);
    }
}
void Padre(void)
{
    int i=1;
    for(i=0;i<10;i++){
        usleep(250);
        printf("Sono il padre. i= %d\n",i);
    }
}
```

Esempi di Fork

Fork Bomb: un programma che chiama la **fork** in un ciclo infinito, blocca la macchina a causa dei troppi processi

```
#include <unistd.h>
int main(void)
{
    while(1)
        fork();
}
```

Fork Bomb in Bash:

SHELL

```
:(){ :|:& };:
```

che equivale a:

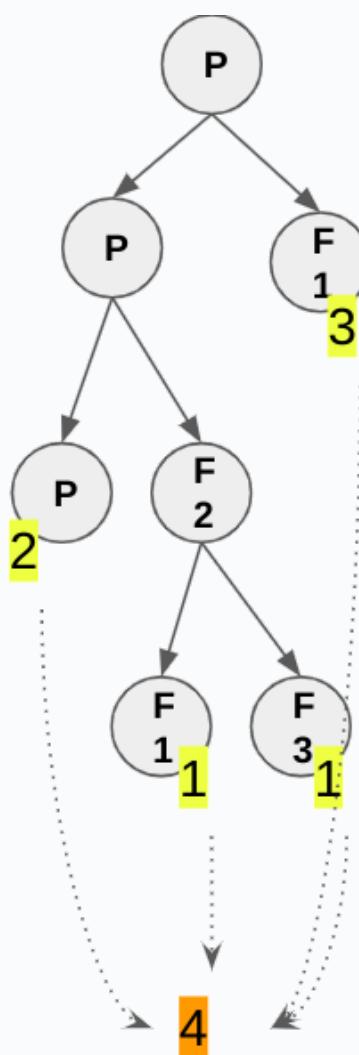
SHELL

```
myfork() {
    myfork | myfork &
}
myfork
```

Albero di Processi

Esercizio: si determini l'albero di processi generato dal seguente codice e l'output generato (*importante per l'esame!*)

```
#include <stdio.h>
#include <unistd.h>
int main(){
    if (fork()){
        if (!fork()){
            fork();
            printf("1 ");
        }
    }  
    else
        printf("2 ");
    else
        printf("3 ");
    printf("4 ");
    return 0;
}
```



Output:

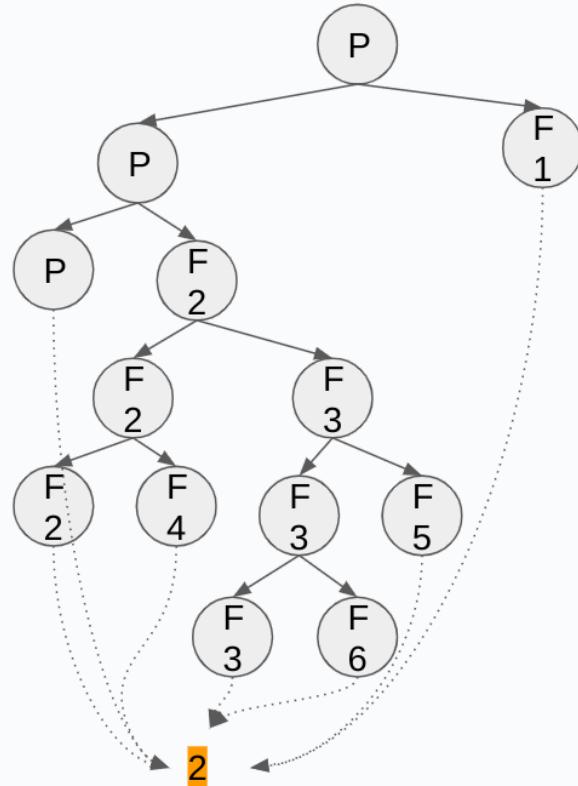
2 4 3 4 1 4 1 4

Esercizio: si determini l'albero di processi generato dal seguente codice e l'output generato

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

int main(){
    printf("\n");
    if (fork() && (!fork())) {
        if (fork() || fork()) {
            fork();
        }
    }

    printf("2 ");
    return 0;
}
```



Output:

2 2 2 2

Osservazione sulla Bufferizzazione

Esercizio: si determini l'output generato dal seguente programma

```
#include <stdio.h>
#include <unistd.h>
int main(int argc,char *argv[]){
    printf("A\n");
    fork();
    printf("B\n");
    fork();
    printf("C\n");
    return 0;
}
```

Output:

A
B
B
C
C
C
C

Esercizio: si determini l'output generato dal seguente programma

Nota: non ci sono i \n nelle printf

```
#include <stdio.h>
#include <unistd.h>
int main(int argc,char *argv[]){
    printf("A ");
    fork();
    printf("B ");
    fork();
    printf("C ");
    return 0;
}
```

Output:

```
A B C A B C A B C A B C
```

Perchè?

Dipende dalla duplicazione della memoria dopo la **fork** e dall'I/O bufferizzato della **printf**. Infatti, l'output è **bufferizzato**! Importante per l'esame

Funzione **wait**

Definizione di Wait

Vediamo la *seconda System Call* fondamentale per i processi.

```
#include <sys/wait.h>
pid_t wait (int *status);
```

Attende la *prima terminazione* di *un* figlio. Entra in stato di *attesa*.

Argomento **status**:

- Puntatore ad un intero;
- Se non è *NULL* specifica lo stato di uscita del processo figlio (*valore restituito dal figlio*) (di solito non ci interessa)

Valore di ritorno:

- Il *PID del figlio terminato*
- 0 in caso di *errore* (esempio: non ci sono figli)

Casistica:

- Se il processo non ha figli: *Errore*
- Se il processo ha dei figli che sono *già terminati*: ritorna *istantaneamente*
- Se il processo ha dei figli *non ancora terminati*: *blocca il chiamante* finché non termina un figlio

Note tecniche.

La funzione **wait** *consuma* un figlio per volta.

Dopo che un figlio è stato *ritornato* al padre tramite una **wait**:

- Il SO rilascia *le risorse del processo figlio* (rappresentate sul PCB)

- Il SO mantiene informazioni su processi terminati di cui non è ancora stata effettuata una **wait**
- Traccia che il processo è esistito
- Valore di ritorno e informazioni su esecuzione
- Non verrà ritornato in successive invocazioni

Da questo comportamento possiamo definire altre *classi di processi*.

Processi Zombie: processo terminato il cui padre non ha ancora effettuato una **wait**

- Dopo che viene effettuata, il *processo è morto definitivamente* e non ne rimane traccia
- In questo stato, il PCB è ancora presente in memoria

Processi Orfani: processi in cui *padre è morto*.

- Se il padre muore, i figli *continuano l'esecuzione*
- Diventano figli del processo *init* ($PID = 1$)
- *Periodicamente*, *init* esegue delle **wait** per consumare gli *orfani morti*

System Call Alternativo

Alternativamente possiamo usare un altro *system call*, più configurabile.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Attende la *prima terminazione* di:

- Un qualsiasi figlio se **pid == -1** (come **wait** classica)
- Un figlio con PID **pid** se **pid>0**
- Un qualsiasi figlio il cui *group ID* è uguale a quello del chiamante se **pid == 0**
- Il figlio il cui *group ID* è uguale a **abs(pid)** se **pid < -1**
 - Esempio: -3 equivale a GID 3.
 - **group ID**: intero positivo associato a un processo. Serve per definire gruppi di processi creati dall'utente. Utile per mantenere ordine.

Altri argomenti di **waitpid:**

- **status** come nella **wait**, un "ritorno al valore" secondario
- **options**: controlla *se la funzione è bloccante*. È una *bitmask*.
 - 0 *bloccante*
 - **WNOHANG**: *non blocca* in caso di assenza di figlio già morto. Si usa per "controllare" se un figlio sia morto o meno: utile!
 - Altri flag per intercettare solo figli morti in *condizioni particolari*

Grafi di Precedenze

Introduciamo una conseguenza di `fork`, `wait` molto importante: i grafi i precedenze.

Esercizio 1

Esercizio: si scriva un programma che implementa il seguente grafo di precedenze con `fork` e `wait`.

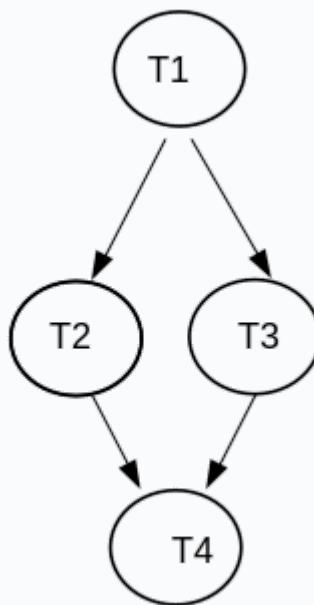
Nota:

Ogni biforcazione si implementa tramite una `fork` e ogni ricongiungimento tramite una `wait`

Per risolvere questi tipi di esercizi, è importante *prima fare un diagramma* che pianifica il codice, poi creare il programma effettivo!

Importante: questi esercizi permettono di scrivere codice che parallelizza diverse operazioni

Fondamentale per programmazione parallela



SOLUZIONE.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid;
    printf ("T1\n");
    pid = fork();
    if (pid == 0) {
        printf ("T3\n");
        return 0;
    } else {
        printf ("T2\n");
        wait ((int *) 0);
    }
    printf ("T4\n");
    return 0;
}
```

Esercizio 2

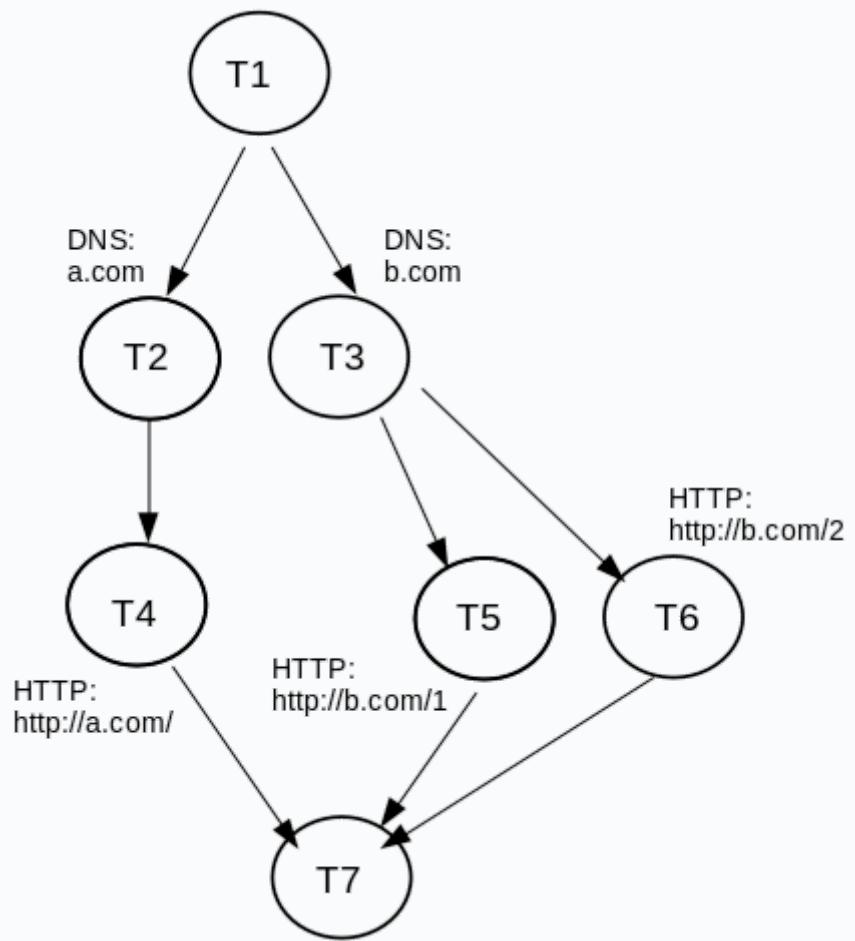
Esercizio: si scriva un programma che implementa il seguente grafo di precedenze con **fork** e **wait**.

Nota: questo è il grafo per eseguire in maniera efficiente 3 richieste HTTP alle URL.

- **http://a.com/**
- **http://b.com/1**
- **http://b.com/2**

Prima di ogni richiesta, è necessario effettuare la risoluzione DNS. Due URL hanno lo stesso dominio.

Nota: nei casi reali, il programmatore deve risolvere il problema efficientemente. Deve costruire da solo il grafo di precedenze.



Soluzione.

```

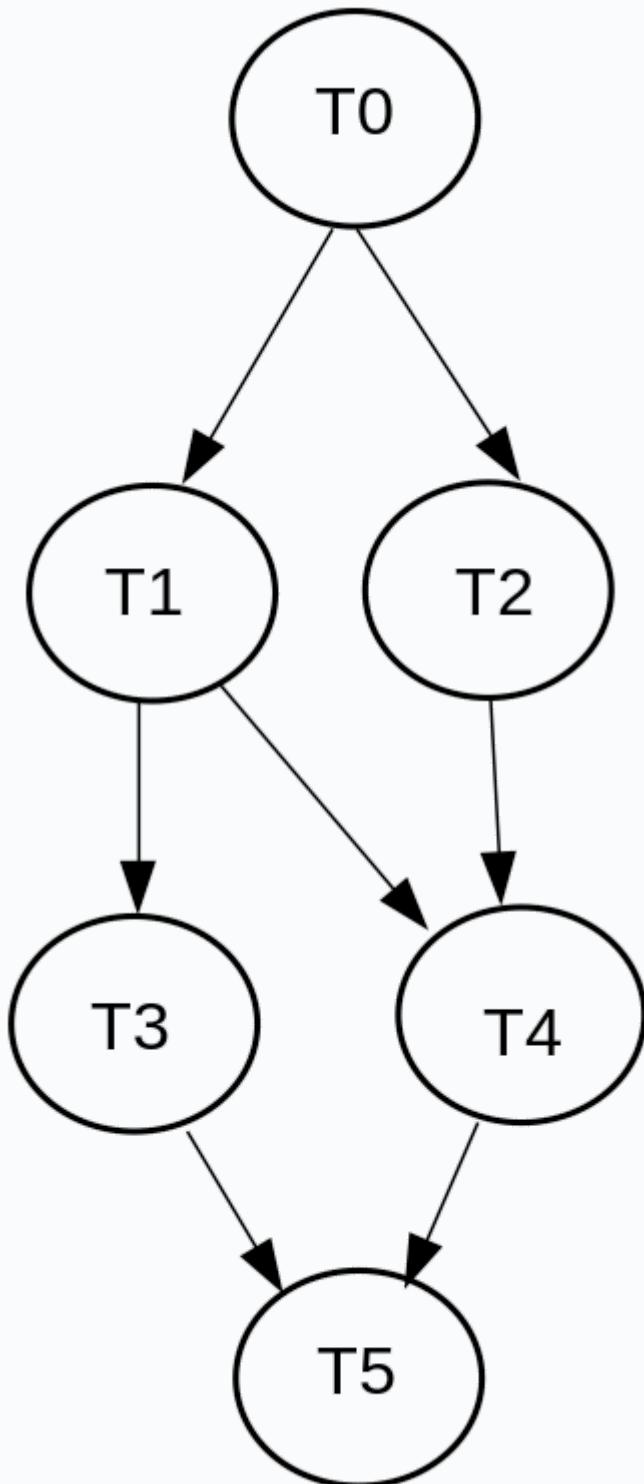
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    pid_t pid;
    printf ("T1 - Start\n");
    pid = fork();
    if (pid == 0) {
        printf ("T3 - DNS b.com\n");
        pid_t pid2;
        pid2=fork();
        if (pid2==0){
            printf ("T6 - HTTP http://b.com/1\n");
            return 0;
        }
        else{
            printf ("T5 - HTTP http://b.com/2\n");
            wait ((int *) 0); /* Attende T6 */
            return 0;
        }
    } else {
        printf ("T2 - DNS a.com\n");
        printf ("T4 - HTTP http://a.com\n");
        wait ((int *) 0); /* Attende T3 - T5 */
    }
    printf ("T7 - Utilizzo i risultati\n");
    return 0;
}

```

Nota: T5 aspetta T6 che è suo figlio. T7 non può **aspettare** T6, in quanto non è suo figlio
La **wait** **aspetta** solo sui figli, **NON** sui nipoti!

Esercizio 3

Esercizio: si scriva un programma che implementa il seguente grafo di precedenze con **fork** e **wait**.



Questo grafo è **molto difficile** da realizzare mediante sole **fork** e **wait**

T4 non può **attendere** T1. Non è suo figlio! Come faccio a gestire la freccia diagonale?
In generale:

- Si possono **attendere** solo i figli
- Ogni figlio può essere atteso una volta sola

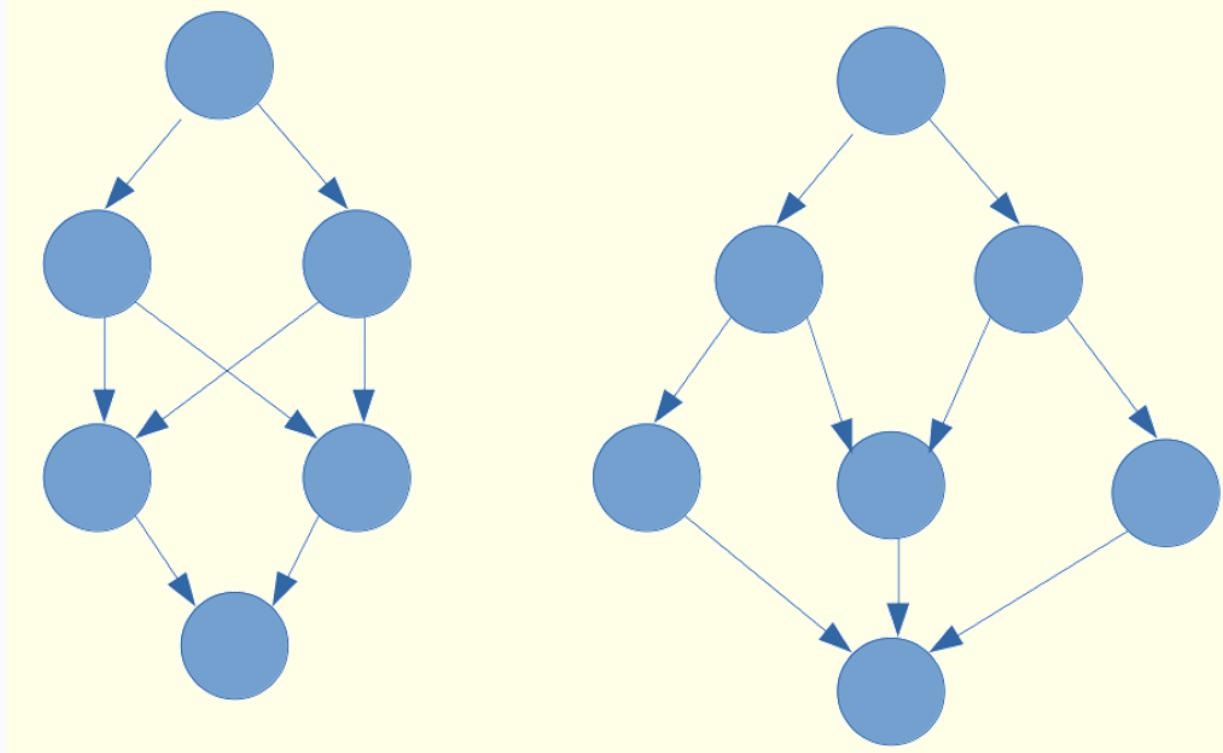
PROBLEMA.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    pid_t pid;
    printf ("T0\n");
    pid = fork();
    if (pid == 0) {
        printf ("T2\n");
        wait ( ??? ) /* ----- IMPOSSIBILE! */
        printf ("T4\n");
    } else {
        printf ("T1 -\n");
        printf ("T3 -\n");
        wait ((int *) 0); /* Attende T4 */
    }
    printf ("T7 - Utilizzo i risultati\n");
    return 0;
}
```

Come si può implementare questo grafo? (Da fare per esercizio!)

Grafi Impossibili

Grafi impossibili (matematicamente dimostrabili con la teoria dei grafi):



Funzione `exec`

Differenza tra Fork e Exec

FORK.

La `fork` permette di *duplicare un processo*

- Usata quando il figlio deve eseguire lo stesso programma del padre
- In programmi paralleli: web server, database

EXEC.

La `exec` permette di *cambiare la natura di un processo* corrente

- Caricando ed eseguendo un programma diverso
- Usata ognqualvolta bisogna avviare un nuovo programma

Quando un processo chiama una `exec`:

- Il processo viene rimpiazzato *completamente* dal codice contenuto nel file specificato (text, data, heap, stack vengono *sostituiti*)
- Il nuovo programma inizia a partire *dalla sua funzione main*, come se fosse un nuovo processo
- Il PID non cambia

Conseguenza.

Cosa *eredita* il processo dopo una `exec`:

- *Variabili d'ambiente*: le variabili definite nel terminale
 - Accessibili tramite la `char *getenv(const char *name)`

- PID e PPID (PID del padre)
- Privilegi, current working directory, root e home directory

Cosa *non viene ereditato*:

- File aperti se hanno il flag **close-on-exec**
- Altrimenti lasciati aperti

Funzioni Exec

Esistono 7 versioni della **exec**.

Hanno la *stessa funzione*, varia *il modo in cui ricevono gli argomenti*. Sono le seguenti

```
#include <unistd.h>

int execl(const char *pathname, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *pathname, const char *arg, ..., char *const envp[]);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(char *pathname, char *argv[], char* envp[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Path

Le funzioni con **p** ricevono il *nome dell'eseguibile* e non il path.

- Il SO rintraccia l'eseguibile nelle cartelle dei programmi installati nel sistema
- Che sono definite nella variabile d'ambiente **PATH**

Esempio: si equivalgono

```
execlp("cp", ...);
execl("/usr/bin/cp", ...);
```

Perchè, sul mio PC:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
```

Passaggio di Argomenti

- Le funzioni con **l** ("list") specificano gli *argomenti* del *nuovo programma* tramite una lista di argomenti. Simile a **printf**.
 - Esempio:** `execlp("cp", "cp", "file1", "file2");`
 - Nota:** Non dimenticare **argv[0]**!
- Le funzioni con **v** specificano gli *argomenti* del nuovo programma tramite un unico vettore di puntatori a **char**. Equivalente a **argv** nel **main**
 - Il primo argomento deve contenere il nome del file associato all'eseguibile che viene caricato (**argv[0]**)
 - L'array di puntatori deve essere terminato da un puntatore **NULL**
 - Mancando **argc**, questo serve a comunicare la lunghezza del vettore

Esempio:

```
// Semplice
execlp("cp", "cp", "file1", "file2");

// Generico
const char *args[4];
args[0] = "cp";
args[1] = "file1";
args[2] = "file2";
args[3] = NULL;
execvp("cp", args);
```

Nota: Non c'è un modo per specificare la lunghezza nel vettore, per convenzione si pone l'ultimo argomento come **NULL**.

Variabili d'Ambiente

Le funzioni con **e** ricevono un vettore di variabili d'ambiente. Quindi esse *non* vengono ereditate dal processo esistente.

- Le variabili d'ambiente sono specificate nell'ultimo argomento tramite un vettore di puntatori a **char** (come con **execv-**)
 - Terminato da puntatore **NULL**
 - Ogni elemento è una stringa nella forma **nome=valore**

```
char *const args[] = {"ls", "/tmp", NULL};  
execv("/usr/bin/ls", args);  
  
char *const envs[] = {"a=1", "b=2", NULL};  
execve("/usr/bin/ls", args, envs);
```

Funzione **execve**

Osservazione:

- La funzione **execve** è una System Call.
- Le altre funzioni sono di libreria, e invocano la **execve** dopo aver correttamente gestito e aggiustato i parametri

Esercizi su Exec

1. **Esercizio:** si scriva una *simple shell* usando le funzioni **fork**, **wait** e **exec**.
Importante per l'esame!

Soluzione.

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define MAXLINE 128
int main() {
    char buf[MAXLINE];
    pid_t pid;
    int status;
    printf("%% "); /* prompt */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0;
        if ((pid = fork()) < 0) {
            printf("errore di fork "); exit(1);
        } else if (pid == 0) { /* figlio */
            execlp(buf, buf, NULL);
            printf("non posso eseguire: %s\n", buf);
            exit(127);
        } else
            if ((pid = waitpid(pid, &status, 0)) < 0) /* padre */
                {printf("errore di waitpid"); exit(1);}
            printf("%% ");
    }
    exit(0);
}

```

Nota: per gestire gli *argomenti* dei comandi invocati, bisognerebbe manipolare le stringhe

2. **Esercizio:** si consideri il seguente programma.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char ** argv) {
    char str[10];
    int n;
    n = atoi(argv[1]) - 1;
    printf ("%d\n", n);
    if (n>0) {
        sprintf (str, "%d", n);
        execl (argv[0], argv[0], str, NULL);
    }
    printf ("End!\n");
    return 1;
}
```

Cosa viene stampato eseguendo **./prog 5** ?

Soluzione.

```
4
3
2
1
0
End!
```

Funzione **system**

Definizione di System

E' una funzione di libreria che invoca un comando **Bash** e ne attende la conclusione

- Utile per usare programmi esterni in un programma
- Internamente usa: **fork**, **exec** e **wait**

C

```
#include <stdlib.h>

int system(const char *command);
```

Equivale a una **fork** il cui figlio esegue:

C

```
exec("/bin/sh", "sh", "-c", command, (char *) NULL);
```

Valore di ritorno: il valore di ritorno *del comando eseguito*

Implementazione di System

Implementazione semplice: (*Importante!*)

C

```
int system(const char *cmd)
{
    int stat;
    pid_t pid;
    if (cmd == NULL)
        return(1);
    if ((pid = fork()) == 0) /* Son */
        exec("/bin/sh", "sh", "-c", cmd, (char *)0);
        _exit(127);
    if (pid == -1) {
        stat = -1; /* Error */
    } else { /* Father */
        while (waitpid(pid, &stat, 0) == -1) {
            if (errno != EINTR){
                stat = -1;
                break;
            }
        }
    }
    return(stat);
}
```

Esercizi su System

Esercizio: Si scriva un programma che fa il listing dettagliato di una cartella.

- La cartella è passata come argomento
 - Se non ci sono argomenti, lista la directory corrente
- Usando **ls -lh cartella**

Soluzione.

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

int main (int argc, char * argv[1]) {
    char command[50] = "ls -lh ";
    if (argc == 2)
        strcat(command, argv[1]);
    system(command);
    return(0);
}
```

Terminazione di un Processo

Ci sono diversi modi per terminare un processo:

Funzione Exit

1. Modo Standard (valore di ritorno, funzione exit)

- Dal **main** avviene una **return**

```
return status;
```

- Viene chiamata la funzione **exit**
c #include <stdlib.h> void exit(int status);

Tutti i buffer (di console e file) vengono *flushed*

L'argomento **status** è ritornato al SO

Per permettere queste operazioni di pulizia, vengono chiamate tutte le *funzioni di chiusura*:

1. Della *libreria standard*

2. *Definite dall'utente* tramite la funzione **int atexit(void (*function)(void));**

```
C

void fun(void) { printf("Exiting\n"); }

int main()
{
    atexit(fun);
    exit(10);
}
```

Viene stampato **Exiting**

System Call _exit

2. System Call **_exit**

```
C

#include <unistd.h>
void _exit(int status);
```

Termina immediatamente *senza controllare i buffer*.

Invocata nei processi *figli* che potrebbero leggere *buffer* in stato intermedio dei padri

Usata specialmente dopo **exec** fallite

- Il figlio *non dovrebbe eseguire nessuna istruzione* dopo la **exec**!
- I buffer possono contenere *dati del padre che non devono essere scritti dai figli*

Nota:

- La **_exit** è una System Call
- La **exit** è una funzione di libreria. Fa pulizia e poi invoca la **_exit**

Funzione Abort

3. Terminazione Anomala:

- Viene ricevuto un **segnale** non gestito (vedremo)
- Il programma chiama la **abort**.

```
#include <stdlib.h>
void abort(void);
```

Riassunto

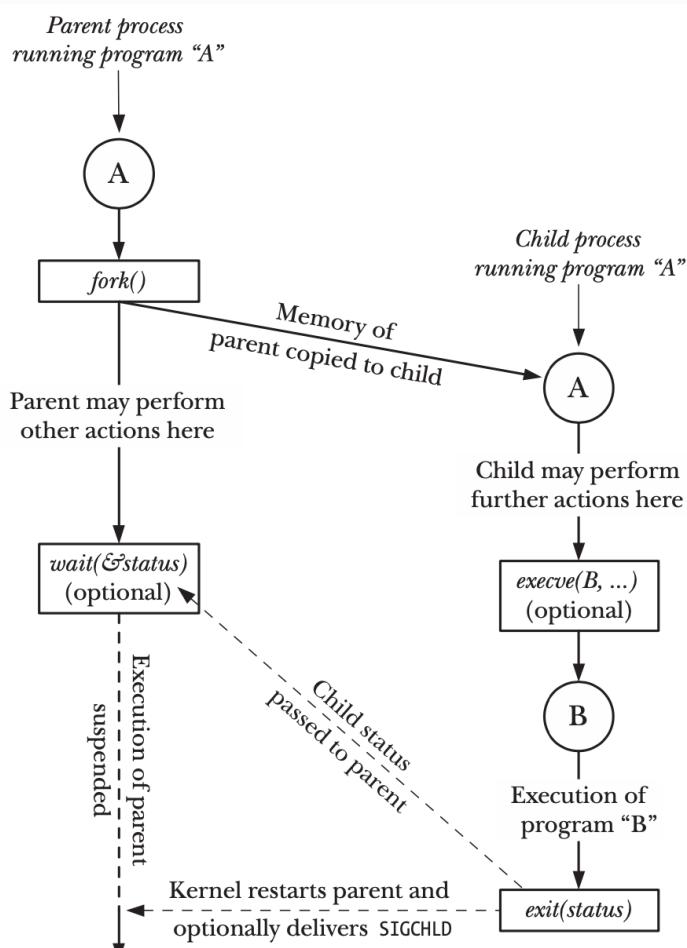
In qualunque modo termini il processo, il kernel compie le seguenti azioni:

- *Rimozione della memoria* utilizzata dal processo
- Chiusura dei *descrittori aperti*

Stato di un processo: raccolto con `wait()`, `waitpid()`. Diventa un *processo-zombie*.

Riassunto delle Operazioni con Processi

- **fork** duplica il processo corrente
- **execve** tramuta il processo corrente in un altro programma
- **exit** termina il processo corrente (uguale a **return** dal **main**)
- **wait** blocca finchè un processo figlio non termina



Ottenere PID

Vediamo altre due system call (funzioni) per ottenere *PID* di processi.

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

- La **getpid()** ritorna il PID del processo chiamante
- La **getppid()** ritorna il PID del *padre* del processo chiamante

Comandi Bash per Processi

Vediamo alcuni *comandi Bash* per i processi. Iniziamo riassumendo quelli già noti

- **ps**: lista i processi del sistema
 - Di default mostra solo processi figli del terminale corrente.
 - Con l'opzione **a** mostra tutto
 - Uso comune: **ps fax**
 - Di default, mostra solo processi che sono in foreground (hanno una shell)
 - Con opzione **x** mostra anche quelli in background
 - Opzioni utili: **u** mostra utente proprietario. **f** rende graficamente gerarchia padre-figlio
- **top**: mostra i processi in maniera interattiva
- **htop**: come **top** ma grafica migliorata
 - **top**, **htop** sono l'equivalente del *Task Manager* per il *Linux* sul terminale.
- **which**: fornisce il path assoluto di un programma di sistema. *Molto utile!*

SHELL

```
$ which ls
/usr/bin/ls
```

- I comandi di sistema vengono cercati nelle cartelle indicate nella variabile d'ambiente **\$PATH**
 - Solitamente: **/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin**
- **pgrep**: stampa il PID di tutti i processi di un programma. Letteralmente è **ps + grep**.

```
$ pgrep chrome
480492
480498
480505
```

Sintassi per Processi in Bash

Note tecniche. (*importanti!*)

Esecuzione di processi figli da *script bash*:

- Un comando che termina con `&` viene *eseguito in background* (si dice "detached")
 - Viene eseguita una `fork` e una `exec` per eseguire il comando
 - Non esegue la `wait`. Lo script e il programma eseguono *in parallelo*
- Il PID del processo appena creato può essere ottenuto con `$!`
 - Sovrascritto a ogni processo creato! Usare *attentamente*
- Si può usare il comando `wait [PID]` per aspettare
 - Attende il figlio `PID` se specificato, altrimenti un figlio qualsiasi

Esempio: Quanto ci mette a eseguire questo codice? (*classica domanda quiz per l'esame!*)

```
sleep 4 & # Sleep viene eseguito in background
PID=$! # Lo script recupera il PID
sleep 2
wait $PID # Lo script attende che sleep termini
```

Soluzione. Ci mette 4 secondi, non 6! I programmi eseguono in parallelo

Informazioni sui Processi

Il `/proc` file system.

I sistemi Linux/POSIX espongono informazioni sui processi correnti tramite uno *Pseudo File Virtuale* detto `procfs`

- File System Virtuale
- Montato in `/proc` in automatico dal `kernel`
- Permette a chiunque di conoscere lo stato dei processi in esecuzione
- *Tramite normali letture da file*
- Buona alternativa a System Call complicate

- Non *corrisponde* fisicamente sul disco, è un modo per *mappare* le info sul *file system*.

Sottocartelle del **/proc** file system.

1. Le informazioni su un processo *PID* si trovano nella *directory* **/proc/PID**
- Il *file* **/proc/PID/status** contiene varie informazioni:

SHELL

```
$ cat /proc/1566/status
```

Name: grep

State: R (running)

Tgid: 5452

Pid: 5452

PPid: 743

...

VmPeak: 5004 kB

VmSize: 5004 kB

VmLck: 0 kB

VmHWM: 476 kB

VmRSS: 476 kB

2. La **subdirectory** **/proc/PID/fd** contiene un link per ogni file aperto dal processo

- Il nome di questi link è il *numero del descrittore* usato nel processo
- Ricordare: ogni file aperto *identificato da un numero*

Esempio:

SHELL

```
/proc/1968/fd/1
```

Rappresenta lo **stdout** del processo 1968.

- Ricorda: 0 è **stdin**, 1 è **stdout**, 3 è **stderr**

Altri *file/subdirectory* del processo *PID* sotto **/proc/PID**

File	Description (process attribute)
cmdline	Command-line arguments delimited by \0
cwd	Symbolic link to current working directory
environ	Environment list <i>NAME=value</i> pairs, delimited by \0
exe	Symbolic link to file being executed
fd	Directory containing symbolic links to files opened by this process
maps	Memory mappings
mem	Process virtual memory (must <i>lseek()</i> to valid offset before I/O)
mounts	Mount points for this process
root	Symbolic link to root directory
status	Various information (e.g., process IDs, credentials, memory usage, signals)
task	Contains one subdirectory for each thread in process (Linux 2.6)

3. Il file system **/proc** fornisce anche molte informazioni sul sistema e possibilità di configurazione
- **/proc/cpuinfo**: informazioni su CPU
 - **/proc/meminfo**: informazioni su memoria

Directory	Information exposed by files in this directory
/proc	Various system information
/proc/net	Status information about networking and sockets
/proc/sys/fs	Settings related to file systems
/proc/sys/kernel	Various general kernel settings
/proc/sys/net	Networking and sockets settings
/proc/sys/vm	Memory-management settings
/proc/sysvipc	Information about System V IPC objects

4. **sysfs**: montato in **/sys**, contiene informazioni sullo *stato del kernel* e sulle periferiche
- Complementare a **/proc**
5. **/dev**: contiene i file speciali che rappresentano le *periferiche*
- Dispositivi a blocchi:
 - Dischi: **/dev/sda1**, **/dev/hda2**
 - CDRom: **/dev/cdrom**; Floppy: **/dev/fd0**
 - Dispositivi a carattere: tastiera, mouse
 - Quando si legge/scrive a questi file speciali, viene invocato il *driver* della periferica corrispondente
 - Volendo (non consigliato!) si potrebbe fare delle operazioni su questi file virtuali.

Domande

L'esecuzione del seguente codice quanti processi genera (incluso il processo che esegue il **main**) ?

```
#include <stdio.h>
#include <unistd.h>
int main(){
    int N = 2;
    for (i=0; i<N; i++)
        fork();
}
```

- 2
- 3
- 4
- 6

Risposta: 4 (2^2)

Un processo il cui processo padre muore:

- Viene terminato dal SO
- Riceve un segnale dal SO
- Viene ereditato (diventa figlio) dal processo init

Risposta: Viene ereditato (diventa figlio) dal processo init

Cosa stampa il seguente codice?

```
#include <stdio.h>
#include <unistd.h>
int main(){
    if ( fork() ){
        printf("A\n");
    }else{
        fork();
        printf("B\n");
    }
}
```

- | | | | |
|-----|-----|-----|-----|
| • A | • A | • A | • A |
| B | A | B | A |
| B | A | | B |

Risposta:

A

B

B

La System Call `execve` crea un nuovo processo?

- Sempre
- Mai
- Dipende da come viene invocata

Risposta: Mai (cambia la natura del processo)

La funzione `system` crea un nuovo processo?

- Sempre
- Mai
- Dipende da come viene invocata

Risposta: Sempre (è stata implementata mediante fork)

u5-s3-segnali

Sistemi Operativi

Unità 5: I processi

I Segnali

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Concetto di segnale
 2. Segnali in Linux
 3. System Call `sigaction`
 4. System Call `kill`
 5. System Call `raise`
 6. System Call `pause`
 7. System Call `alarm`
 8. Considerazioni
 9. Segnali nella shell
-

Concetto di segnale

Richiamo agli Interrupt

RICHIAMO. (*Interrupt*)

In quasi tutti i sistemi ad elaboratore, esistono gli *interrupt*:

Un *interrupt* informa la CPU che deve interrompere il compito corrente per eseguire un'azione impellente (1).

Un interrupt viene generato da:

- Un *dispositivo hardware* che vuole notificare al sistema un evento (di solito una linea elettrica, circa 5V)
- Particolari *istruzioni nel codice* (e.g., istruzione **INT**)
 - Quando un processo chiama una System Call genera un interrupt software
 - Sono delle "*finte interrupt*"

Definizione di Segnale

Un *segnale* permette la *gestione di eventi asincroni* che interrompono il normale funzionamento di un *processo*

- E' un interrupt software
 - In particolare mirate sui *processi*
- Notifica un evento a un processo specifico

Possono essere *generati* da

- *Kernel* per comunicare eventi eccezionali:
 - Condizioni di errore
 - Azioni dell'utente (e.g., **CTRL+C** su tastiera)
- Un *altro processo* (se ne ha i permessi):
 - Permettono una primitiva comunicazione tra processi
 - Usando la system call **kill**

Notizie Storiche

Esistono dalle *prime versioni di Unix* (siamo negli anni '90; 1)

- Formalizzati in Unix 4
- Rappresentava un *primo meccanismo di Inter-process communication*

In principio erano inaffidabile e gestiti in modo *best-effort*

- *Potevano andare perduti*
- La gestione era *complicata*
- *Poca configurazione* possibile

I segnali esistono anche in *Windows*, sebbene abbiano un funzionamento leggermente diverso (in realtà molto!). Tuttavia ci concentreremo sui *segnali in Linux*.

Segnali in Linux

Molteplicità dei Segnali

Esistono *diversi tipi di segnali* in Linux

- Dipende dalle versioni di Linux
- Comando **kill -l** lista i segnali
 - 64 in Ubuntu 20
 - Circa 10 saranno importanti per noi

Ogni segnale ha un *identificatore mnemonico e numerico*

- Identificatori di segnali iniziano con i tre caratteri SIG (che sta per "SI~~G~~nal")
 - Es. **SIGINT** è il segnale di interruzione e ha numero 2
- I nomi simbolici corrispondono ad un intero positivo (in C queste costanti vengono definite su **signal.h**)

Come detto prima, ogni segnale viene generato da un *evento specifico nel SO*, o *manualmente* da un processo.

Un segnale può avere i seguenti effetti su un processo:

Effetti dei Segnali

Importanti

1. Viene *ignorato*
2. *Termina il processo*
3. Interrompe momentaneamente il processo. Esegue una *funzione handler*.
Dopodiché il processo riprende

Secondari

4. Crea un *core dump*: un *file* che contiene lo stato del programma per poter essere debuggato
5. *Stoppa* il processo
6. *Fa ripartire* il processo

Adesso vediamo *in specifico* quali segnali hanno i comportamenti elencati sopra

Segnali ignorati di default:

- **SIGCHLD**: inviato al padre quando un figlio termina

Segnali che di default terminano un processo:

Sono tanti!

- **SIGINT**: viene inviato al processo in esecuzione quando si preme **CTRL+C**; vedremo che si può modificare il comportamento
- **SIGABRT**: inviato da system call **abort()** (ha un'utilità abbastanza discutibile)
- **SIGFPE**: inviato da eccezione aritmetica (ad esempio l'operazione $\frac{1}{0}$)
- **SIGHUP**: Inviato ad un processo se il terminale *viene disconnesso* (sta per "Signal HangUP")
- **SIGKILL**: Maniera sicura per uccidere un processo.
Nota: Non si può creare un handler per **SIGKILL**! Ovvero il suo comportamento non può essere modificato!
- **SIGSEGV**: Accesso di memoria non valido
- **SIGTERM**: Segnale di *terminazione* normalmente usato. Generato dal comando **kill** di default
- **SIGUSR1** e **SIGUSR2**: generati solo da processi utente, mai dal SO. Servono per comunicazione tra processi
 - Di conseguenza sono da *modificare!*

Lista completa degli segnali su Linux

Lista più completa.

Il comportamento di default può essere modificato:

- Per *ignorare* un segnale
- Per *gestirlo* tramite un *handler*
- *NON* per indurre Terminazione o Core Dump
- Tranne **SIGKILL** e **SIGSTOP** (non vanno mai modificati!)

Name	Description	Default
SIGABRT	Abort process	Core
SIGALRM	Real-time timer expiration	Term
SIGBUS	Memory access error	Core
SIGCHLD	Child stopped or terminated	Ignore
SIGCONT	Continue if stopped	Cont
SIGFPE	Arithmetic exception	Core
SIGHUP	Hangup	Term
SIGILL	Illegal Instruction	Core
SIGINT	Interrupt from keyboard	Term
SIGIO	I/O Possible	Term
SIGKILL	Sure kill	Term
SIGPIPE	Broken pipe	Term
SIGPROF	Profiling timer expired	Term
SIGPWR	Power about to fail	Term
SIGQUIT	Terminal quit	Core
SIGSEGV	Invalid memory reference	Core
SIGSTKFLT	Stack fault on coprocessor	Term
SIGSTOP	Sure stop	Stop
SIGSYS	Invalid system call	Core
SIGTERM	Terminate process	Term
SIGTRAP	Trace/breakpoint trap	Core
SIGTSTP	Terminal stop	Stop
SIGTTIN	Terminal input from background	Stop
SIGTTOU	Terminal output from background	Stop
SIGURG	Urgent data on socket	Ignore
SIGUSR1	User-defined signal 1	Term
SIGUSR2	User-defined signal 2	Term
SIGVTALRM	Virtual timer expired	Term
SIGWINCH	Terminal window size changed	Ignore
SIGXCPU	CPU time limit exceeded	Core
SIGXFSZ	File size limit exceeded	Core

Signal Handler

Un processo può definire un *signal handler*.

- Una *funzione* che viene eseguita quando il processo riceve il segnale
- Se non lo fa, c'è il *comportamento di default*

L'idea si esprime con la seguente frase: "*Se e quando ricevi un certo segnale, esegui questa funzione*"

Stato di un Segnale

Fasi di vita di un segnale:

1. **Generazione:** da parte del *kernel o di un processo*
2. **Consegna:** nel più breve tempo possibile consegna il segnale al processo.
 - Finché un segnale non è consegnato è *pending*

- Questa è una fase importante!
- Vengono gestiti mediante una **bitmask** (lo terremo in mente per l'osservazione seguente)

3. Gestione:

- Il kernel avvia la **funzione handler** del processo nel caso ce ne sia una
- Altrimenti compie l'**azione di default** per quel segnale (termina o ignora)

Osservazione:

I segnali non vengono accodati.

I segnali pendenti per un processo sono gestite da una **mask**.

- Se lo **stesso segnale** è generato **più volte** prime che sia consegnato, esso lo sarà **una sola volta**

Manipolazione dei Segnali

Adesso vediamo una serie di **system call** su C per manipolare le System Call. Si userà la libreria **<signal.h>**.

System Call **sigaction**

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Modifica il comportamento del processo corrente a un segnale particolare.

Argomenti:

- **signum**: segnale da trattare
- **act**: puntatore a struttura che definisce trattamento
- **oldact**: puntatore a comportamento precedente. Può servire per ristabilire il comportamento precedente. Ci permette di poter *sapere il vecchio comportamento*.

Ritorna -1 se c'è stato errore

Adesso vediamo la struct **sigaciton**.

```

struct sigaction {
    void (* sa_handler )( int );
    sigset_t sa_mask ;
    int sa_flags ;
    void (* sa_restorer )( void );
};

```

- **sa_handler** specifica il comportamento
 - Se **funzione**, specifica un *handler* (un puntatore a funzione che ritorna nulla)
 - Se **SIG_IGN** ignora
 - Se **SIG_DFL** ripristina comportamento di default
- **sa_mask**: *segnali da bloccare* mentre l'handler è in esecuzione (gestita con **bitmask**)
 - Si inizializza con la funzione **int sigemptyset(sigset_t *set);**
 - Nel corso non faremo altre operazioni su questo campo
- **sa_flags**: flag (non vediamo). Settiamo sempre a 0
- **sa_restorer**: per uso interno

Esempio: si crei una funzione per ignorare un segnale definito dal chiamante

```

int ignoreSignal ( int sig )
{
    struct sigaction sa ;
    sa.sa_handler = SIG_IGN ;
    sa.sa_flags = 0;
    sigemptyset ( &sa.sa_mask );
    return sigaction ( sig , &sa , NULL );
}

```

Funzione Handler.

La funzione handler deve prendere un argomento **int**

- Quando *invocata dal SO*, contiene il numero del segnale
- Deve ritornare **void**

```
void myHandler ( int sig )
{
    /* Actions to be performed when signal
       is delivered */
}
```

Nota: L'handler è una funzione del programma.

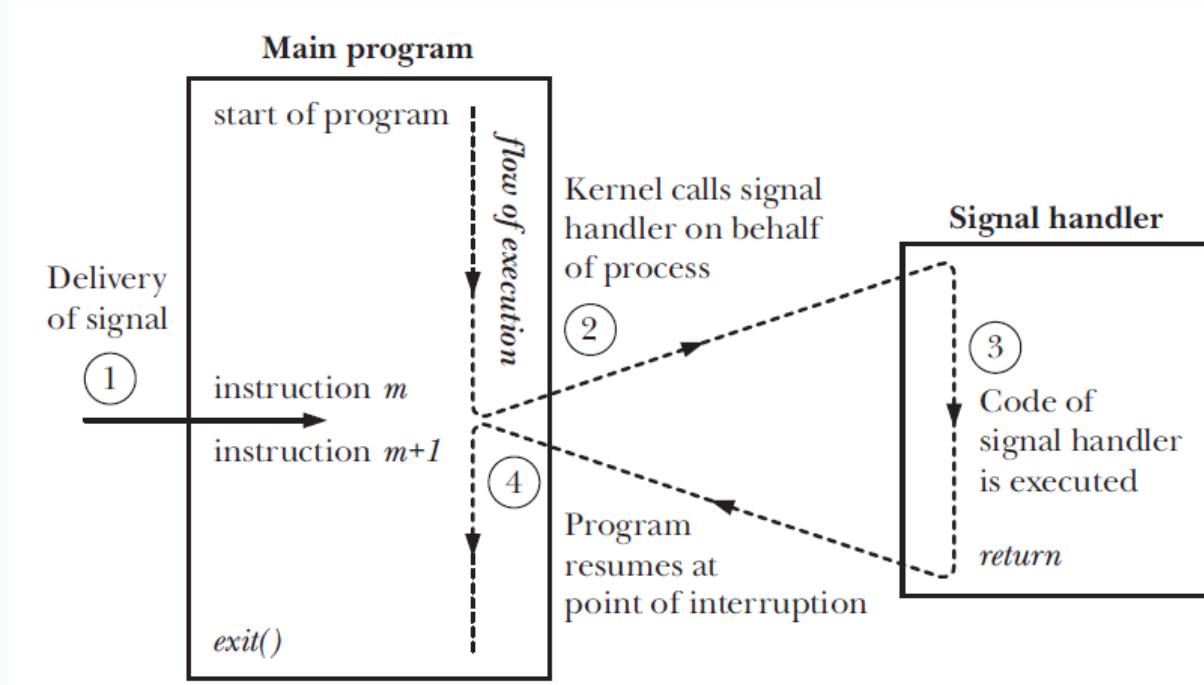
- Che viene invocata **automaticamente** dal SO alla ricezione del segnale
- E **non dal programmatore**. Potrebbe, ma non ha senso

Funzionamento Interno di **sigaction**

Viene invocata **automaticamente dal kernel** alla consegna del segnale

Il programma si **interrompe**, esegue l'handler

Infine, **continua l'esecuzione** dal punto di interruzione



Esercizio con **sigaction**

Esempio: si crei un programma che gestisce i segnali **SIGINT**, **SIGHUP** e **SIGTERM**

```
#include <signal.h>
#include <stdio.h>
void func(int signum){
    printf("ricevo %d\n", signum);
}

int main (){
    struct sigaction new_action, old_action;

    new_action.sa_handler = func;
    sigemptyset (&new_action.sa_mask); /* Si noti l'uso di sigemptyset */
    new_action.sa_flags = 0;

    sigaction (SIGINT, &new_action, NULL);
    sigaction (SIGHUP, &new_action, NULL);
    sigaction (SIGTERM, &new_action, NULL);

    while(1) ;
    return 0;
}
```

Per terminare il programma, bisogna mandargli un segnale **SIGKILL**. Lo si fa digitando
pkill -KILL <nome prog>

System Call **signal**

Esiste la System Call **signal**, più a basso livello.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Argomenti:

- **sig**: quale segnale gestire
- **handler** specifica il comportamento. E' *puntatore a funzione*.

Nota: consigliato usare **sigaction**

System Call **kill**

Manda un segnale *ad un processo* oppure a un *gruppo di processi*

```
C  
#include <sys/types.h>  
#include <signal.h>  
  
int kill(pid_t pid, int sig);
```

Argomenti:

- **sig**: segnale da mandare
- **pid**:
 - se **> 0**: spedito al processo identificato da **pid**
 - se **0**: spedito a tutti i processi appartenenti allo *stesso gruppo* del processo che invoca **kill()**
 - se **<0**: spedito al gruppo di processi identificati da **-pid** (ovvero $|pid|$)
 - se **-1**: a *tutti i processi*: da *non usare!*

Privilegi: un processo può mandare un segnale solo a processi dello *stesso utente*. Tranne *root*, che può mandare a tutti.

Esercizio con **kill**

Esercizio: si crei un programma che genera un processo figlio. Il padre manda al figlio un segnale **SIGUSR1** ogni secondo. Il figlio stampa l'avvenuta ricezione.

C

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

void handler(int signum){
    printf("Ricevuto\n");
}

int main (){
    pid_t pid;
    struct sigaction action;

    pid = fork();
    if (pid!=0){ /* Father */
        while(1){
            sleep(1);
            kill (pid, SIGUSR1);
            /* pause(); Would be equivalent */
        }
    } else{ /* Child */
        action.sa_handler = handler;
        sigemptyset (&action.sa_mask);
        action.sa_flags = 0;
        sigaction (SIGUSR1, &action, NULL);
        while (1);
    }
    return 0;
}
```

System Call **raise**

C

```
#include <signal.h>
int raise (int sig);
```

Permette a un processo di *inviare un segnale a se stesso*.

Di fatto:

```
C  
raise (sig);
```

equivale a:

```
C  
kill (getpid(), sig);
```

System Call **pause**

```
C  
#include <unistd.h>  
int pause (void);
```

Sospende il processo fino all'arrivo di un segnale

Serve a implementare l'*attesa passiva* di un segnale

Ritorna dopo che il segnale è stato catturato ed il gestore è stato eseguito, *restituisce sempre* (-1)

System Call **alarm**

```
C  
#include <unistd.h>  
unsigned int alarm (unsigned int seconds);
```

Implementa un *timeout*

Il SO manda un segnale **SIGALRM** al processo dopo **seconds** secondi

- Se non vi era già un timeout settato, restituisce subito 0
- Altrimenti, restituisce *i secondi che mancano* allo scadere dell'ultimo allarme settato.
Cancella il vecchio timeout e inserisce il nuovo
- Se **seconds** è 0, si *disattiva* il timeout

Osservazioni:

Il timeout è gestito dal kernel.

Il tempo effettivo può essere leggermente maggiore a causa del tempo di reazione del kernel

- Quindi **non** è affidabile fino a millisecondi *ms* o *μs*!
 - Ovvero non vanno implementati in sistemi dove i millisecondi contano
-

Esercizio con **alarm** e **pause**

Esempio: funzione **sleep** implementata con **alarm** e **pause**

```
C  
static void myAlarm (int signo) {  
    return;  
}  
void mySleep (unsigned int nsecs) {  
    signal(SIGALRM, myAlarm)  
    alarm (nsecs);  
    pause ();  
}
```

Considerazioni sulle System Call per Segnali

Un handler è un flusso di **esecuzione concorrente**

- Può iniziare in *qualsiasi istante*
- Mentre il flusso principale sta compiendo qualsiasi azione
- Operazione molto delicata!

Importante:

Per questo motivo l'handler **non deve modificare variabili** globali che sono usate anche dal flusso principale

- Potrebbe portare in stato inconsistente
- Potrebbe portare al **problema dell'incremento perso**

Problema dell'incremento perso

- Immaginiamo che il programma venga interrotto tra la riga 2 e la riga 3 del seguente codice:

```
1 local = global; /* Supponiamo global = 1 */  
2 local++; /* Local = 2 */  
--- Interruzione ---  
3 global = local;
```

- L'handler esegue l'operazione:

```
global++; /* Global = 2 */
```

- La variabile **global** viene *incrementata*
- Il programma riprende dall'istruzione 3.

```
...  
--- Interruzione ---  
3 global = local; /* Global = 2 */
```

- L'incremento dell'handler si è perso. **global** = 2 anzichè 3. Questo è un grande problema!

Problema che vedremo approfonditamente per i programmi *multi-thread*

Conseguenze

Da questo definiamo le nozioni di *funzione rientrante* e *funzione non rientrante*.

Definizioni:

1. **Funzione rientrante:** può essere usata con sicurezza in più flussi
2. **Funzione non rientrante:** *NON* può essere usata con sicurezza in più flussi (può portare a stati inconsistenti)

In generale, negli handler, bisogna:

- Chiamare *solo funzioni rientranti*
- Evitare di *manipolare variabili globali* che sono usate dal flusso principale.

Classificazione. Classifichiamo alcune funzioni già note

- Molte funzioni di libreria C *NON* sono rientranti
 - **fprintf**, **fscanf**: gestione del *buffer* problematica.

- Non vanno chiamate dentro un handler!
- Alcune funzioni *sono rientranti* e possono essere interrotte senza problemi: **sleep**, etc. Vedi **man**
- Le *System Call* sono rientranti:
 - Se un programma riceve un segnale mentre è eseguita una sua system call (e.g., **read**):
 - Le System Call *bloccanti* terminano e non riprendono (e.g., **read**, **write**, **wait**)
 - Le System Call *non bloccanti* riprendono (e.g., **fork**, **getpid**)

Segnali nella shell

Vediamo alcuni comandi per *operare* con i segnali.

1. Kill

SHELL

```
kill pid
```

Invia un segnale al processo **PID**.

Di default manda **SIGTERM**.

Possibile specificare con opzioni **-KILL** **-INT**

SHELL

```
pkill nome
```

```
killall nome
```

Stesso comportamento, ma manda il segnale a tutti i processi del programma **nome**

Esercizio Misto

Esercizio: si scriva un programma in C che memorizza quanti **SIGTERM** ha ricevuto. Alla pressione di **CTRL+C** stampa tale numero e termina. Si nomini il programma **sample**.

Si scriva anche uno script bash che manda 10 segnali **SIGTERM** al processo.

Programma Bash:

```
for i in $( seq 5) ; do
    pkill sample
done
```

Programma C:

```
C

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

int c;

void handler(int signum){
    if (signum==SIGTERM)
        c+=1;
    else if (signum==SIGINT){
        printf("Ricevuti %d SIGTERM\n", c);
        exit (0);
    }
}

int main (){
    struct sigaction action;
    c=0;
    action.sa_handler = handler;
    sigemptyset (&action.sa_mask);
    action.sa_flags = 0;
    sigaction (SIGTERM, &action, NULL);
    sigaction (SIGINT, &action, NULL);

    while (1);
}
```

Notare l'uso dell' handler! E' in grado di ricevere e maneggiare segnali diversi! (Volendo potevo pure creare due handler diversi)

Quando si preme **CTRL+C**, viene mandato un **SIGINT** al programma, che stampa **c** e termina

Extra: si faccia uno script bash che automatizza tutta la sequenza: avvio del programma in C, consegna segnali e chiusura.

SHELL

```
./sample &
PID=$!
for i in $( seq 5) ; do
    kill $PID
done
kill -INT $PID
```

Comandi per Mandare Segnali

Abbiamo dei *comandi* per mandare *segnali a processi*

1. Se **CTRL+C**, viene inviato **SIGINT**
 - Programma termina se non c'è un handler
2. Se **CTRL+Z** viene inviato **SIGTSTP**
 - Di default l'applicazione viene sospesa
 - E messa in background dalla shell
 - A questo punto possiamo fare due cose:
 - **fg** fa riprendere l'esecuzione in foreground
 - **bg** far riprendere l'esecuzione in background

Molto utile se ho lanciato un comando lungo e voglio usare la shell mentre esegue

SHELL

```
$ ./longjob
^Z
[1]+ Stopped      ./longjob
$ bg
[1]+ ./longjob &
$ terminale libero
```

3. Quando eseguo un programma in background (**./job &**) e chiudo il terminale, viene mandato il segnale di Hang Up **SIGHUP**
 - Di default il programma viene terminato
 - Si può modificare comportamento

Oppure uso il comando **nohup** che esegue un comando immune a `SIGHUP`

```
nohup ./job
```

Utile se lancio job su terminale remoto e devo andare a casa!

Alternativa più pulita: comando **screen** che genera *terminale virtuale*

Handler in Bash

Handler di segnali in script bash: si usa il comando **trap**. Ascoltate la trap?

SHELL

```
trap command SIGNAL
```

Esegue il comando o la funzione **command** se lo script riceve il segnale **SIGNAL**
Esiste lo pseudo-segnale aggiuntivo **EXIT**, chiamato quando lo script termina

SHELL

```
tempfile=/tmp/tmpdata  
trap "rm -f $tempfile" EXIT
```

Esercizio Bash

Esercizio: si crei un programma bash che conta quanti SIGUSR2 riceve, e li stampa quando viene premuto **CTRL+C** e lo si nomini **sample.sh**

```
#!/bin/bash

count=0
function husr(){
    let count++
}
function hint(){
    echo "Ricevuti $count SIGUSR2"
    exit 0
}

trap husr SIGUSR2
trap hint SIGINT

while true; do
    sleep 1
done
```

Si inviano i segnali col comando: **bash pkill -USR2 sample.sh**

Nota: dichiarazione di funzione in Bash

Domande

Quale System Call si usa per generare un segnale?

- **signal**
- **kill**
- **write**
- **send**

Risposta: kill

Una funzione handler riceve degli argomenti?

- **No**
- **Riceve una stringa**
- **Riceve un intero**

Risposta: Riceve un intero

Quale è il comportamento di default di un processo quando riceve un segnale?

- **Il segnale viene ignorato**
- **Il processo termina**
- **Dipende dal segnale**

Risposta: Dipende dal segnale

Un signal handler può modificare le variabili globali del processo?

- **Sì**
- **No**

Risposta: Tecnicamente sì, ma fortemente sconsigliato (sì)

Quale segnale viene inviato dal SO quando si preme **CTRL+C** sulla tastiera?

- **SIGKILL**
- **SIGINT**
- **SIGHUP**
- **SIGSTP**

u5-s4-inter-process-communication

Sistemi Operativi

Unità 5: I processi

Inter-Process Communication

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Obiettivi
 2. Le *pipe*
 3. Le *FIFO*
 4. Cenni di memoria virtuale
 5. Memoria condivisa con **shmget**
 6. Memoria condivisa con **mmap**
 7. Problematiche
-

Obiettivi

Definizione. (*Processi indipendenti e cooperanti*)

In un sistema dotato di SO, diversi processi sono in esecuzione contemporaneamente.

Essi possono essere classificati in:

- **Processi indipendenti:** non sono influenzati né influenzano altri processi
- **Processi cooperanti:** *interagiscono con altri processi*. Devono usare meccanismi opportuni per farlo
Ci concentreremo sui *processi cooperanti*, in particolare i loro meccanismi.

Tutti i SO mettono a disposizione strumenti per la *Inter-Process Communication*

Sono tipicamente basati su:

- *Scambio di messaggi* (esempio: segnali)
- *Scambio di dati* (vedremo le *pipe* o le *FIFO*)
- *Memoria condivisa* (vedremo *mmap* o *shmget*)

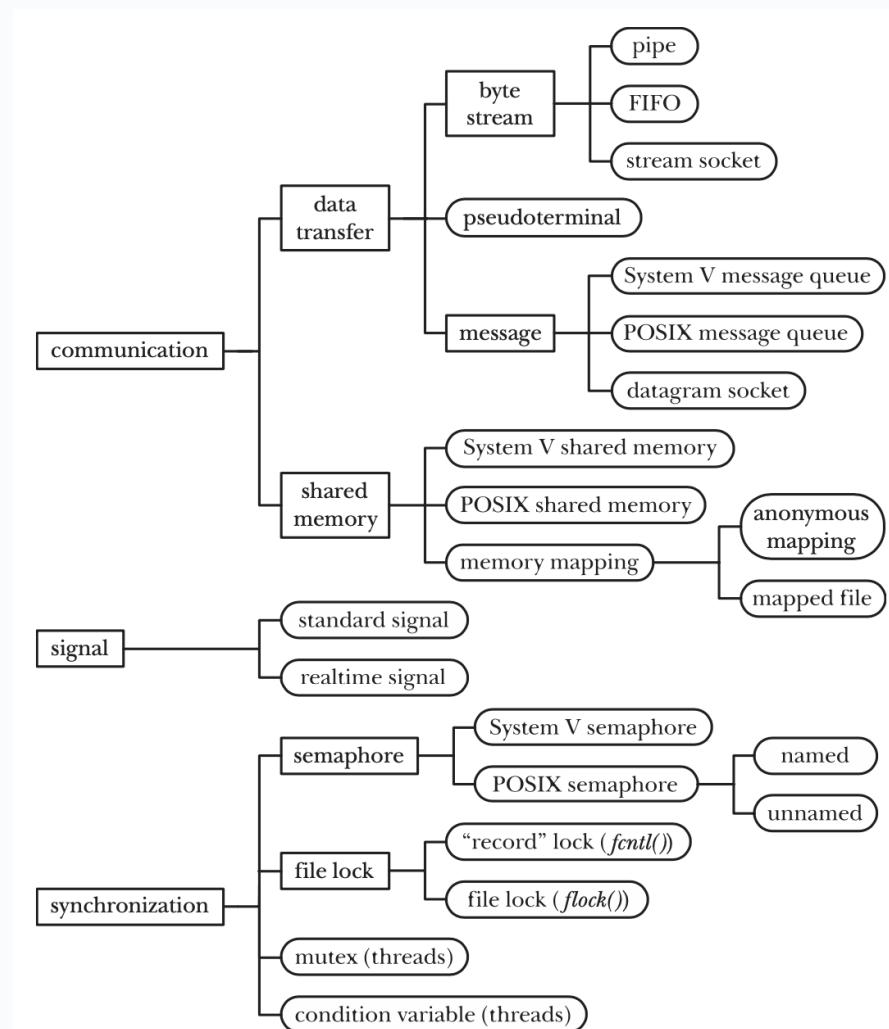
- Altri costrutti di sincronizzazione
Ogni SO utilizza meccanismi diversi.

LINUX.

In Linux, ci sono *tanti meccanismi*.

- Storicamente stratificati
 - Ereditati da System V
 - Parte di standard POSIX
- Ci concentreremo su:
- Pipe, FIFO (*Byte stream*)
 - Memory Mapping (*Shared memory*)
 - Sincronizzazione (semafori, eccetera...)

Ogni meccanismo ha una sua caratteristica diversa: alcuni presuppongono *legami di parentela*, altri senza, eccetera...



Le pipe

Concetto di Pipe

Idea. (Pipe)

Le **pipe** sono la più vecchia e la più usata forma di IPC introdotta in Unix

- Permettono di scambiare **dati** (sotto forma di bytestream) tra processi
- Modello **produttore-consumatore**; **produttore** → **consumatore**
- Si usano con le stesse System Call dei file: **read**, **write**
- Risiedono in memoria; molto veloce!
- **Non sono persistenti**: quando i processi terminano, tutto ciò che rimane viene distrutto

Limitazioni:

- Sono half-duplex (comunicazione *in un solo senso*)
- Utilizzabili solo tra processi *con un "antenato" in comune*
 - Si dice che le pipe sono **anonime**.

Come superare queste limitazioni?

- Le **FIFO** (o **named pipe**) possono essere utilizzati tra più programmi
 - Si identificano tramite un nome

Primo Esempio di Pipe

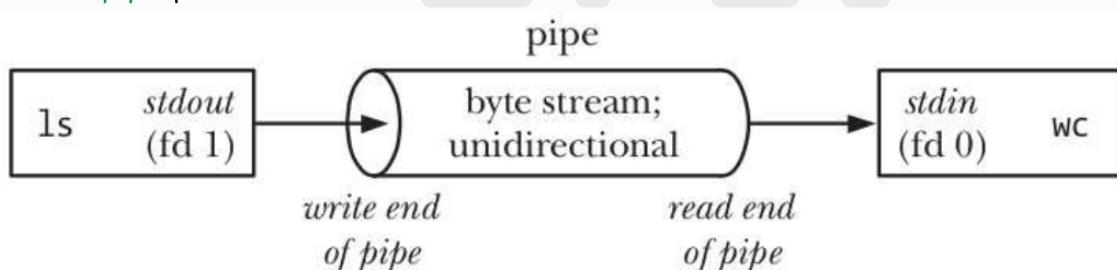
Le **pipe** sono comunemente usate nella shell, per redirezionare gli **stdout** e **stdin**, come visto con **Bash**.

Esempio:

```
ls | wc -l
```

Per fare questa operazione, la shell:

- Usa due **fork** e **exec** per creare i processi **ls** e **wc**
- Crea una **pipe** per connettere lo **stdout** di **ls** con lo **stdin** di **wc**



- Studieremo questo **a basso livello**

Definizione di Pipe

Definizione:

Le pipe sono un *byte stream*

- Vi si scrivono/leggono byte
- Non solo caratteri stampabili
Sono *unidirezionali*:
- Hanno un ingresso e una uscita
Hanno *capacità limitata*:
- I dati accodati (scritti ma non ancora letti) *non possono eccedere una soglia*
- Soglia configurabile: 65 KB di default
 - Si può variare con **fcntl(fd, F_SETPIPE_SZ, size)**

Sintassi in C per Pipe

Creazione:

```
#include <unistd.h>
int pipe (int filedes [2]);
```

Ritorna due descrittori di file attraverso l'argomento fd (*passato per riferimento*; ricordiamoci che un vettore è un puntatore!)

- **fd[0]** è aperto in lettura (*out*)
- **fd[1]** è aperto in scrittura (*in*)
- L'output di **fd[0]** è l'input di **fd[1]**

I/O su pipe.

Si usano le funzioni **read** e **write**

- Il valore di ritorno è il numero di byte scritti/letti

Lettura:

- La **read** è bloccante finché non è letto almeno un byte. Se la pipe è morta, ritorna 0.

Scrittura:

- Se la *pipe* è piena, la **write** è *bloccante*

Chiusura di una pipe:

- Se leggo un pipe e ottengo 0, significa che *sono stati chiusi tutti i file descriptor*. Così posso effettuare l'*operazione di gestione* per la chiusura di pipe.

Confronto Pipe-File.

- Sui *file* assumo che il contenuto sia *immutable*, letto in sequenza fino all'*EOF*.
- Sui *pipe* la *EOF* non esiste! Teoricamente potrei scriverci quanto voglio.

Nota: se scrivo su una *pipe* che non ha un *reader* (*fd[0]* è stato chiuso), il processo riceve *il segnale SIGPIPE* (*broken pipe*)

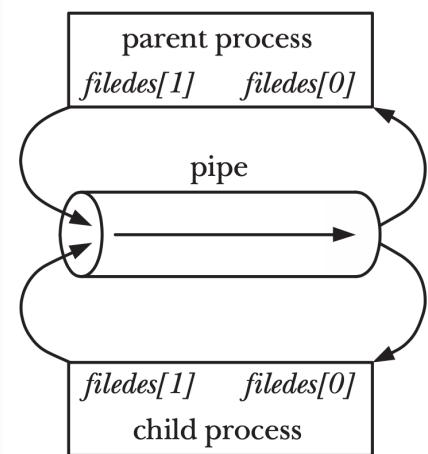
- Comporta la terminazione del processo, se non c'è un *Signal Handler* opportuno

Prassi delle pipe

Condivisione tra processi:

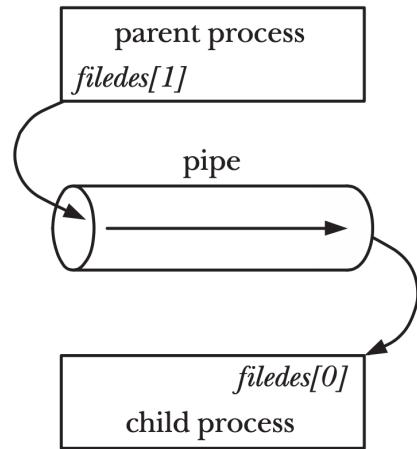
Per utilizzare una pipe tra più processi:

- Il processo padre *crea la pipe* e ottiene i due *fd*
- Esso fa una *fork*
- Entrambi i processi possono accedere alla *pipe* usando i due *fd*



Nota.

- Solitamente un processo (e.g., padre) scrive, e un altro (e.g., figlio) legge
- Tecnicamente possibile che un processo legga e scriva
 - Crea però *problemi di sincronizzazione*
- Ogni processo chiude i *fd* che *non usa* (buona pratica)



Esempio di Pipe

Esempio:

```

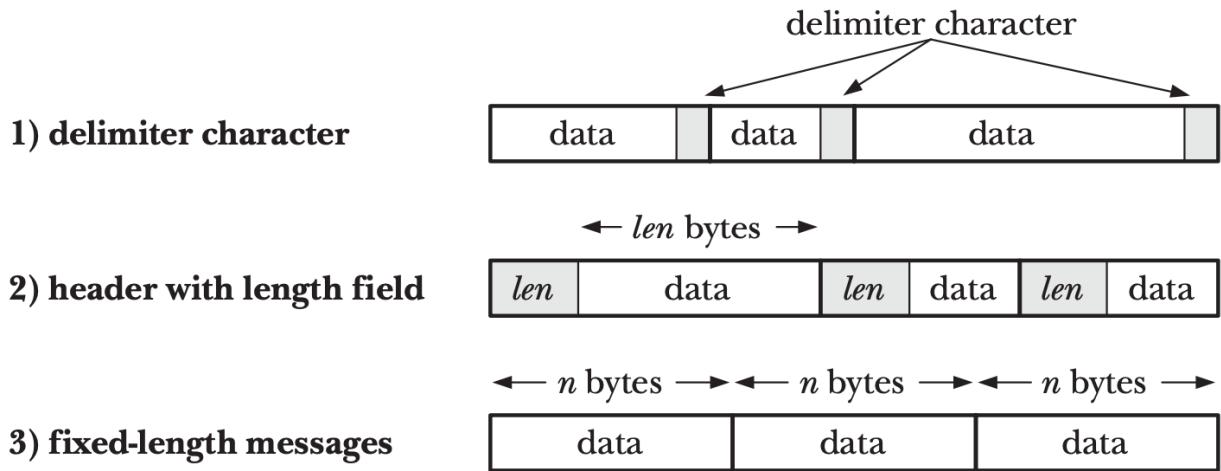
int pfd [2];
pipe ( pfd ); /* Crea la pipe */
switch ( fork () ) {
    case -1: exit(1);
    case 0:
        /* Child */
        close ( pfd [1]);
        /* Può ora leggere */
        break;
    default :
        /* Parent */
        close ( pfd [0]);
        /* Scrive nella pipe */
        break;
}

```

Messaggi su pipe

Messaggi su pipe

Ci sono diverse strategie per scambiare messaggi tramite *pipe*



1. Scelgo un byte per rappresentare la *fine* del messaggio (come **\0**)
2. Approccio complementare: prima scrivo il *la lunghezza del messaggio*, poi il messaggio effettivo
3. Ho n bytes fissati per messaggio. Il più semplice

Esercizio sulle pipe

Esercizio: si crei un programma che genera un figlio. Il processo padre riceve una stringa da riga di comando e la passa al figlio tramite una *pipe*. Il figlio riceve la stringa e la stampa.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#define MAXLINE 1024

int main(int argc, char *argv[])
{
    int pfd[2], status;
    char line[MAXLINE];

    pipe(pfd);
    if (fork() > 0) { /* Padre */
        close(pfd[0]);
        write(pfd[1], argv[1], strlen(argv[1]));
        wait(&status);
    } else { /* Figlio */
        close(pfd[1]);
        read(pfd[0], line, MAXLINE);
        printf("Ricevuto: %s\n", line);
    }
    exit(0);
}
```

Esercizio: si crei un programma con due processi. Il processo padre riceve il nome di un file da riga di comando e ne passa il contenuto al figlio tramite una *pipe*. Il figlio riceve il contenuto e lo stampa.

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAXSIZE 1000

int main(int argc, char * argv[]){
    int pfd[2], status;

    pipe(pfd);
    if(fork()>0){
        close(pfd[0]);
        FILE * fp;
        char line[MAXSIZE];

        fp=fopen(argv[1],"r");
        while(fgets(line,sizeof(line),fp)!=NULL) /* Itera riga per riga*/
            write(pfd[1],line,sizeof(line)); /* Scrive nella pipe */
        close(pfd[1]);
        wait(&status);
        exit(0);
    }
    else {
        char buffer [MAXSIZE];
        close(pfd[1]);
        while (read(pfd[0],buffer,sizeof(buffer)) > 0) /* Quando read ritorna 0, la
pipe è morta*/
            exit(0);
    }
}

```

Le FIFO

Motivazioni per le FIFO

Richiamo.

Facciamo un breve richiamo sulle *pipe*, per poter parlare delle *FIFO* e darne una motivazione.

Pipe "normali"

- Possono essere utilizzate solo da processi che hanno un *"antenato" in comune* (sono anonime)
- Motivo: unico modo per ereditare descrittori di file

Named pipe o FIFO

- Permettono a processi *non collegati* di comunicare
- Utilizzano il file system per *"dare un nome"* alla pipe
- Le *FIFO* sono un tipo di file
 - La macro **S_ISFIFO** dopo una **stat** restituirà **true**
- La procedura per creare un fifo è simile alla procedura *per creare file*
 - In certi versi le FIFO possono essere trattati come file, similmente alle pipe

Sintassi C per FIFO

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

Crea un *FIFO* dal **pathname** specificato

Argomento **mode** specifica i permessi (come nella **open**).

- E.g.: **S_IRWXU**, **S_IRGRP**, etc.

Valore di ritorno: 0 se successo, -1 se errore

Utilizzo:

Come file e *pipe*: tramite **read** e **write**

Ogni processo che ha i permessi per **pathname** può usarla

1. **Apertura:** dopo essere state create con **mkfifo**, le FIFO vanno aperte con una **open** o una **fopen**

Flag.

File aperto senza flag **O_NONBLOCK**:

- Se il file è aperto in *lettura*, la **open si blocca** fino a quando un altro processo non apre la FIFO in scrittura
- Se il file è aperto in *scrittura*, la **open si blocca** fino a quando un altro processo non apre la FIFO in lettura

File aperto con flag **O_NONBLOCK**

- Se il file è aperto in lettura, la **open** ritorna immediatamente
- Se il file è aperto in scrittura, e nessun altro processo è stato aperto in lettura, la **open** ritorna un messaggio di errore

Input/Output:

- Con **read** e **write**
- I dati nella **FIFO** sono *bufferizzati dal kernel*

Importante:

- Una **FIFO** ha un pathname ma è *solo un espediente* per permettere a diversi processi di accedervi
- Quando un **FIFO** viene chiusa (o i processi terminano) il nome del file persiste nel file system, ma esso *non contiene alcun dato*
- Non ho dei **file** presenti sul disco! Sono dei **file virtuali**

Sintassi Bash per FIFO

Si possono creare e usare le **FIFO** in Bash in maniera semplice:

SHELL

```
mkfifo myfifo
tr 'aeiou' 'AEIOU' < myfifo &
man 2 pipe > myfifo
```

Esercizi sulle FIFO

Esercizio: si crei un programma che:

1. se riceve **read** come argomento, stampa ciò che viene scritto su una pipe e
2. se riceve **write** come argomento, legge iterativamente una riga da tastiera e la scrive su una pipe.

Soluzione.

```
#include <stdio.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#define FIFO "my-fifo"
#define BUF_SIZE 512

int main(int argc, char * argv[]){
    FILE * f;
    char buffer[BUF_SIZE];

    if (argc != 2 || ( strcmp(argv [1], "read")==0 && strcmp(argv [1], "write")==0 ) ){
        printf("Usage: fifo read|write\n");
        return 1;
    }

    if (mkfifo(FIFO, S_IRWXU)<0)
        perror("Warning. FIFO not created");

    if (strcmp(argv [1], "read")==0){
        f = fopen(FIFO, "r");
        if (f==NULL){
            perror("Impossible to open the FIFO");
            return 1;
        }

        printf("Read mode:\n");
        while(fgets(buffer,BUF_SIZE,f)≠NULL)
            printf("%s", buffer);

    }else{
        f = fopen(FIFO, "w");
        if (f==NULL){
            perror("Impossible to open the FIFO");
            return 1;
        }

        printf("Write mode. Write lines of text:\n");
        while(fgets(buffer, BUF_SIZE, stdin)≠NULL){
            fputs(buffer, f);
            fflush(f);
        }
    }
}
```

```
    }
    return 0;
}
```

Esercizio: si crei un programma che legge da una FIFO e stampa il contenuto in maiuscolo.

Soluzione.

```
#include <stdio.h>
#include <ctype.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char * argv[]){
    int i, n, l;
    FILE * f;
    char buffer[512];

    i = mkfifo("myfifo", S_IRWXU);
    if (i<0){
        printf("Impossibile creare la FIFO\n"); /* Potrebbe già esistere */
    }

    f = fopen("myfifo", "r");
    if (f==NULL){
        printf("Impossibile aprire la FIFO\n");
        exit(1);
    }

    while(fgets(buffer,sizeof(buffer),f)!=NULL){
        l = strlen(buffer);
        for (i=0; i<l; i++)
            putc(toupper(buffer[i]), stdout);
    }
}
```

La si testi con: **echo "ciao mondo" > myfifo**

Cenni di memoria virtuale

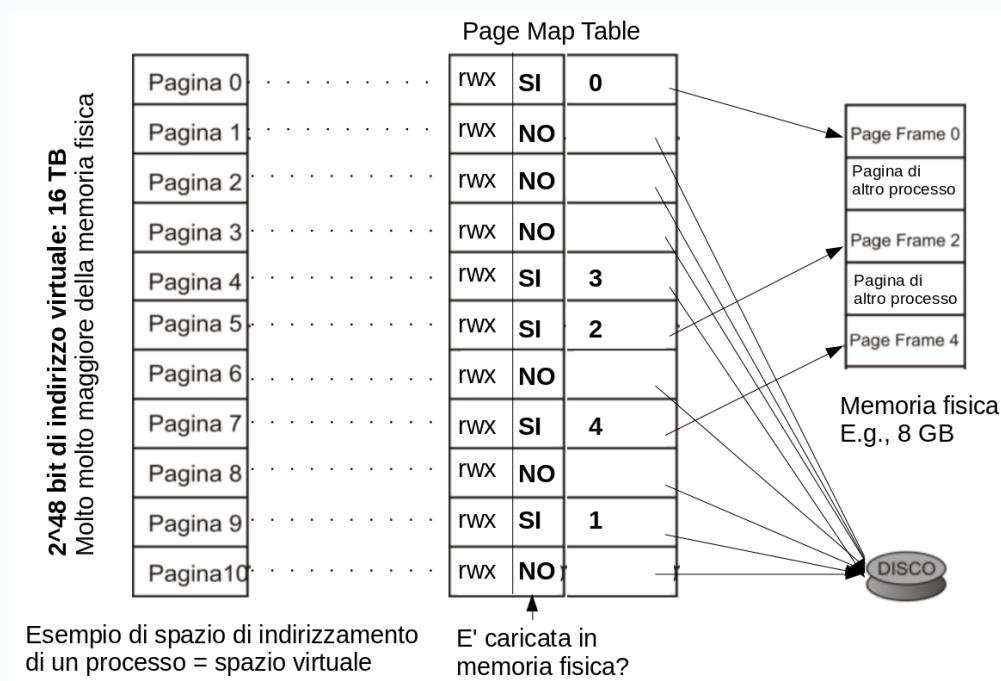
Diamo dei *cenni di memoria virtuale*, prima di poter parlare di *condivisione della memoria*.

I processi emettono *indirizzi virtuali*

- Permettono di indirizzare *più memoria di quella disponibile* (fisicamente)
 - Su architettura AMD64: 48bit; 256TB di memoria virtuale. La memoria fisica è di solito minore (e.g., 16 GB)
- Evitano che un processo acceda a memoria di altri

La *memoria virtuale* è divisa in *pagine* e una *tabella* mappa le pagine da *spazio di indirizzi virtuali a indirizzi fisici*

- Azione compiuta dalla *Memory Management Unit* in Hardware
 - Il sistema operativo interviene a *collocare pagine in memoria*

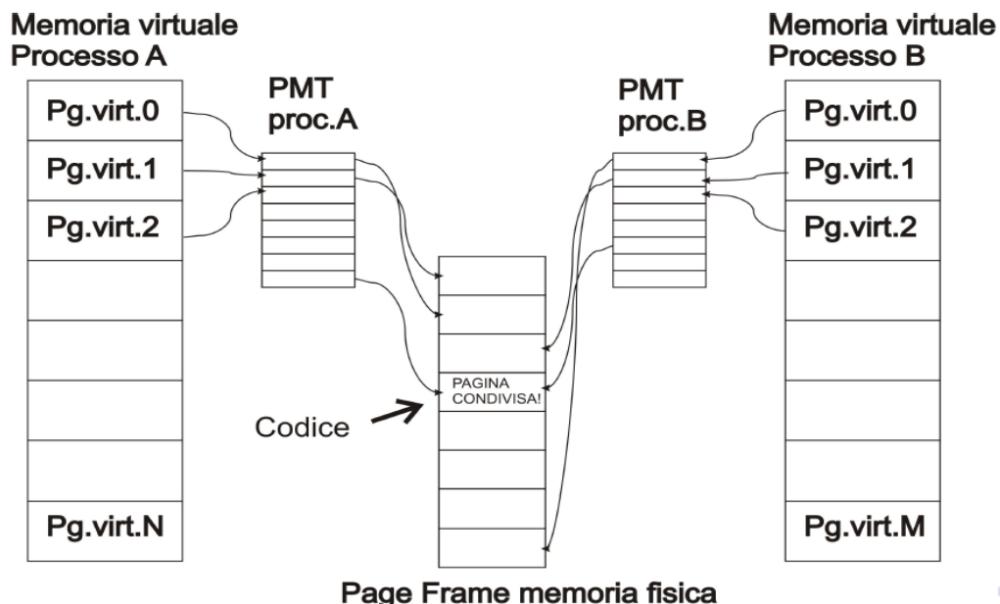


Ogni processo ha uno *spazio di indirizzi virtuali* dedicato

- C'è una tabella delle pagine per processo
- *Isolamento della memoria tra processi*
 - Essenziale per *sicurezza*
 - Non permette la condivisione di memoria
 - Dà al processo l'illusione di essere *l'unico processo*

Per condividere la memoria, è necessario *condividere una o più pagine*

- Il SO mette a disposizione delle System Call per questo scopo



Esistono due set di System Call per avere *memoria condivisa* tra processi in Linux:

- **shmget shmat shmdt ftok**
- **mmap munmap shm_open shm_unlink**: questa è l'opzione più moderna, di cui vedremo

L'approccio con **mmap** è più *moderno e flessibile*

In Windows si usa la System Call **CreateFileMapping**

Memoria condivisa con **shmget**

Nota. Opzionale

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

Crea un segmento di memoria condivisa.

Argomenti:

- **key**: identificativo definito dall'utente. Usare **IPC_PRIVATE** se anonimo (usato solo con **fork**)
- **size**: dimensione della memoria condivisa
- **shmflg**: flag e permessi. **IPC_CREAT** crea se non esistente.
 - Uso tipico **IPC_CREAT | 0666**

Valore di ritorno:

- Un identificativo della zona create
- -1 se insuccesso

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Mappa il segmento di memoria virtuale nello spazio degli indirizzi del processo.

Argomenti:

- **shmid**: identificativo ritornato da **shmget**
- **shmaddr**: se non nullo, la memoria viene mappata a **shmaddr** (arrodonato per difetto al **page size**)
- **shmflg**: flag. **SHM_RDONLY** mappa in Read Only

Valore di ritorno:

- L'indirizzo virtuale del segmento mappato
- -1 se insuccesso

```
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Rimuove in mapping del segmento all'indirizzo virtuale **shmaddr**

Valore di ritorno:

- 0 in caso di successo
- -1 se insuccesso

```
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

Crea una **key** a partire da un path, garantendo che:

- Due path daranno sempre chiavi diverse
- Stesso path e stesso **proj_id** darà sempre chiavi uguali
- Evita che programmi diversi che per sfortuna hanno scelto stessa chiave usino la stessa memoria

Argomenti:

- **pathname**: il path
- **proj_id**: usato per creare la **key**. Non deve essere nullo.

Valore di ritorno:

- *key* in caso di successo
- *-1* se insuccesso

Utilizzo: Memoria condivisa con figlio creato tramite **fork**

```
C

int shmid = shmget(IPC_PRIVATE, 1*sizeof(int), IPC_CREAT | 0666);
void * shm = shmat(shmid, NULL, 0);
if (fork()){ /* Padre */

    ...
} else { /* Figlio */

    ...
}
shmdt(shm);
```

Utilizzo: Memoria condivisa tra due processi indipendenti

- Proceso creatore

```
C

key_t key = ftok(path, proj );
int shmid = shmget(key, size, IPC_CREAT | 0666);
void * data = shmat(shmid, NULL, 0);

...
shmdt(data);
```

- Proceso utilizzatore

```
C

key_t key = ftok(path, proj );
int shmid = shmget(key, size, 0666);
void * data = shmat(shmid, NULL, 0);

...
shmdt(data);
```

Esercizio: Si creino due programmi che hanno una memoria condivisa con **shmget**. Il primo programma permette di scrivere una stringa nella memoria, mentre il secondo permette di leggerla.

Programma 1:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    key_t key = ftok("test",123);
    if (key < 0){printf("Errore. Il file per la chiave esiste?"); exit(1);}
    printf("Key: %d\n", key);

    int shmid = shmget(key,1024,0666|IPC_CREAT);
    char *str = (char*) shmat(shmid,(void*)0,0);
    while(1){
        printf("Input Data : ");
        scanf(" %s", str);
    }
    shmdt(str);
    return 0;
}
```

Programma 2:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    key_t key = ftok("test",123);
    if (key < 0){printf("Errore. Il file per la chiave esiste?"); exit(1);}
    printf("Key: %d\n", key);

    int shmid = shmget(key,1024,0666|IPC_CREAT);
    char *str = (char*) shmat(shmid,(void*)0,0);
    while(1){
        printf("Premi enter per leggere");
        getchar();
        printf("Data: %s\n\n", str);
    }

    shmdt(str);
    return 0;
}
```

Memoria condivisa con mmap

Creazione di Memoria Condivisa

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Crea una zona di memoria condivisa, mappandolo su un file. Comando **complesso** e **configurabile!**

Argomenti:

- **addr**: se non nullo, la memoria viene mappata a **addr** (arrotondato per difetto al **page size**)
 - Tipicamente lo si mette a **NULL**, l'indirizzo viene deciso dal **sistema operativo**.

- **length**: dimensione
- **prot**: può essere: **PROT_READ**, **PROT_WRITE**, **PROT_EXEC**, **PROT_NONE**
 - Cioè la pagina può essere letta, scritta, eseguita, non può essere acceduta
 - Normalmente **prot = PROT_READ|PROT_WRITE**
- **flags** determina come i cambiamenti sono visibili o meno ad altri processi
 - **MAP_ANONYMOUS**, **MAP_SHARED**, **MAP_PRIVATE**
 - Normalmente si impone **flags=MAP_ANONYMOUS | MAP_SHARED** o **MAP_SHARED**
- **fd** e **offset** sono usati per mappare la memoria su un **file** (diventerà l'identificativo globale)
 - Normalmente si impone **fd** e **0**, o **-1** e **0** in caso di anonimità.

Valore di ritorno:

- L'indirizzo virtuale del segmento mappato (andrà convertito col meccanismo di *casting*)
- **-1** se insuccesso

Rimozione di Memoria Condivisa

```
C
int munmap(void *addr, size_t length);
```

Rimuove la mappatura e rende disponibile la memoria all'indirizzo **addr**

Casi d'Uso della **mmap**

Utilizzo: ci sono tre modi per usare la **mmap**

1. Zona di memoria *anonima*: utilizzata con **fork**

```
C
mmap(NULL, size, PROT_READ|PROT_WRITE,
      MAP_SHARED|MAP_ANONYMOUS, -1, 0);
```

2. Zona di memoria *mappata su file*:

```
C
fd = open("/home/martino/file.txt", O_RDWR|O_CREATE);
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

3. Zona di memoria mappata *su file temporaneo*:

```
fd = shm_open("temporaneo.txt", "rw");
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

Adesso li vediamo in dettaglio

Zona di Memoria Anonima

1. Zona di memoria anonima:

- La zona di memoria non ha un nome
- *Solo* un figlio nato con una **fork** può accedervi
- *Semplicissimo da usare*
- Praticamente è come una *pipe* bidirezionale

```
void* shmem = mmap(NULL, size, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);

if (fork()) { /* Padre */
    ...
} else { /* Figlio */
    ...
}
munmap(shmem, size);
```

Nota: Dopo la **fork**, la memoria dei processi è indipendente. Solo con **mmap** è possibile creare una zona di memoria condivisa.

Zona di Memoria Condivisa

2. Zona di memoria mappata su file:

Idea.

- Si usa *un file* come contenitore
- Il contenuto della zona condivisa verrà salvato su file
- *Più programmi* possono accedere alla memoria condivisa
 - Il path va identificatore (diventa l'espeditore)
- Efficace ma lento a causa del *disco*
- E' necessario che il file sia grande a sufficienza per contenere la regione mappata
 - Per assicurarci di questo useremo le funzioni **truncate**, **ftruncate**

```
#include <unistd.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Assicura che il file aperto **fd** o il file **path** si lungo almeno **length**.

- Se necessario il file è troncato
- Se necessario esteso e riempito con caratteri **'\0'** (o **0x00** in hex).

Modello tipico.

```
fd = open(path, O_RDWR|O_CREAT); /* Non è una fopen()*/
ftruncate(fd, size);
void* shmem = mmap(NULL, size, PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0);
...
munmap(shmem, size);
```

3. Zona di memoria mappata su file temporaneo

- A volte è necessario avere un **file temporaneo identificabile** che risiede in memoria
 - Per utilizzo da parte di più programmi
- Una zona di memoria mappata **su file** è **inutilmente lenta**
 - **Se** non è necessario che i dati della zona di memoria sopravvivano

Si possono utilizzare le funzioni **shm_open** e **shm_unlink**, **fatte ad-hoc** per il **file system virtuale**.

- Creano e rimuovono un file temporaneo
- Che si trova nella cartella **/dev/shm/** che ha montato un FS temporaneo (**tmpfs**)
- I dati sono **in memoria**

```
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

Creano e rimuovono zone di memoria temporanee.

- Semantica analoga a **open** e **unlink**
- Operano su zone di memoria/file temporanee
- **name** è un nome di file, non un path completo
 - Tutte le zone di memoria sono file sotto **/dev/shm/**

Flusso tipico

```
C  
int fd = shm_open(nome, O_RDWR, 0);  
ftruncate(fd, size);  
void * mem = mmap(NULL, size, PROT_READ | PROT_WRITE,  
                  MAP_SHARED, fd, 0);  
...  
munmap(shmem, size);  
  
/* Non obbligatorio */  
shm_unlink(nome)
```

Esercizi con Memoria Condivisa

Esercizio: Si creino due programmi che hanno una memoria condivisa con **mmap** e **shm_open**.

Il primo programma permette di scrivere una stringa nella memoria, mentre il secondo permette di leggerla.

Programma 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h> /* Per O_RDWR */
int main(){
    int fd;
    char * mem;

    fd = shm_open("mymem", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    ftruncate(fd, 512);
    mem = (char *) mmap(NULL, 512, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    while(1){
        printf("Scrivere: ");
        scanf("%s", mem);
    }
    munmap(mem, 512); /* Inutile causa loop infinito*/
}
```

Programma 2:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h> /* Per O_RDWR */
int main(){
    int fd;
    char * mem;

    fd = shm_open("mymem", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    ftruncate(fd, 512);
    mem = (char *) mmap(NULL, 512, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    while(1){
        printf("Premi enter per leggere");
        getchar();
        printf("Data: %s\n\n", mem);
    }
    munmap(mem, 512);
}

```

Nota: Compilare con **gcc prog.c -lrt -o prog**

Include **librt, libposix4 - POSIX.1b Realtime Extensions library**

Problematiche della Memoria Condivisa

L'utilizzo della memoria condivisa è complesso

- Preferire le *pipe* o *FIFO* quando possibile

La memoria condivisa ha problemi di *sincronizzazione* e *race conditions*

- Processi concorrenti possono leggere dati in stato inconsistente
- Mentre un altro processo li stava modificando
- Come abbiamo già visto coi segnali

Esempio:

Time	τ_x	τ_y
t_1	READ (A)	—
t_2	$A = A - 50$	—
t_3	—	READ (A)
t_4	—	$A = A + 100$
t_5	—	—
t_6	WRITE (A)	—
t_7	—	WRITE (A)

LOST UPDATE PROBLEM

La variabile A viene incrementata di 100.

- Il decremento di 50 viene perso

Per ovviare a questi problemi esistono le *tecniche di sincronizzazione*

- Permettono di evitare che un processo sia interrotto mentre effettua un'operazione critica
 - Permettono a un processo di attendere il verificarsi di una condizione
-

Domande

Le *pipe* sono

- Monodirezionali • Bidirezionali • Dipende dai parametri di creazione

Risposta. Monodirezionali

Le *pipe* sono identificate da un nome?

- Si • No

Risposta. No

Le *FIFO* sono identificate da?

- Un ID numerico • Una stringa • Un Path

Risposta. Un Path

Si consideri il seguente codice C che opera sulla FIFO **myfifo**:

```
int n;
FILE * f = fopen("myfifo", "r");
fscanf (f, "%d", &n);
```

Che operazione compie?

- Crea la FIFO myfifo
- Scrive un intero in myfifo
- Legge un intero da myfifo

Risposta. Legge un intero da myfifo

Una zona di memoria condivisa creata tramite **shmget** e **shmat** può essere condivisa anche tra processi senza legami di parentela?

- Si
- No

Risposta. Non posso rispondere (parte saltata)

Le funzioni **shmat** e **mmap** hanno valore di ritorno:

- char *
- int
- int*
- void*
- void

Risposta. void* (puntatore a void)

Si consideri il seguente spezzone di codice:

```
int fd = open("/tmp/mymem", O_RDWR|O_CREAT);
ftruncate(fd, 64);
void* shmem = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED,
fd, 0);
sprintf( (char*)shmem, "Ciao Mondo!");
```

Dopo che il programma è terminato, il contenuto della zona di memoria presiste?

- Si, in memoria
- Si, nel file /tmp/mymem
- No

Risposta. No

tmp

u6-s1-memoria

Sistemi Operativi

Unità 6: La memoria

Organizzazione delle memoria

[Martino Trevisan](#)

[Università di Trieste](#)

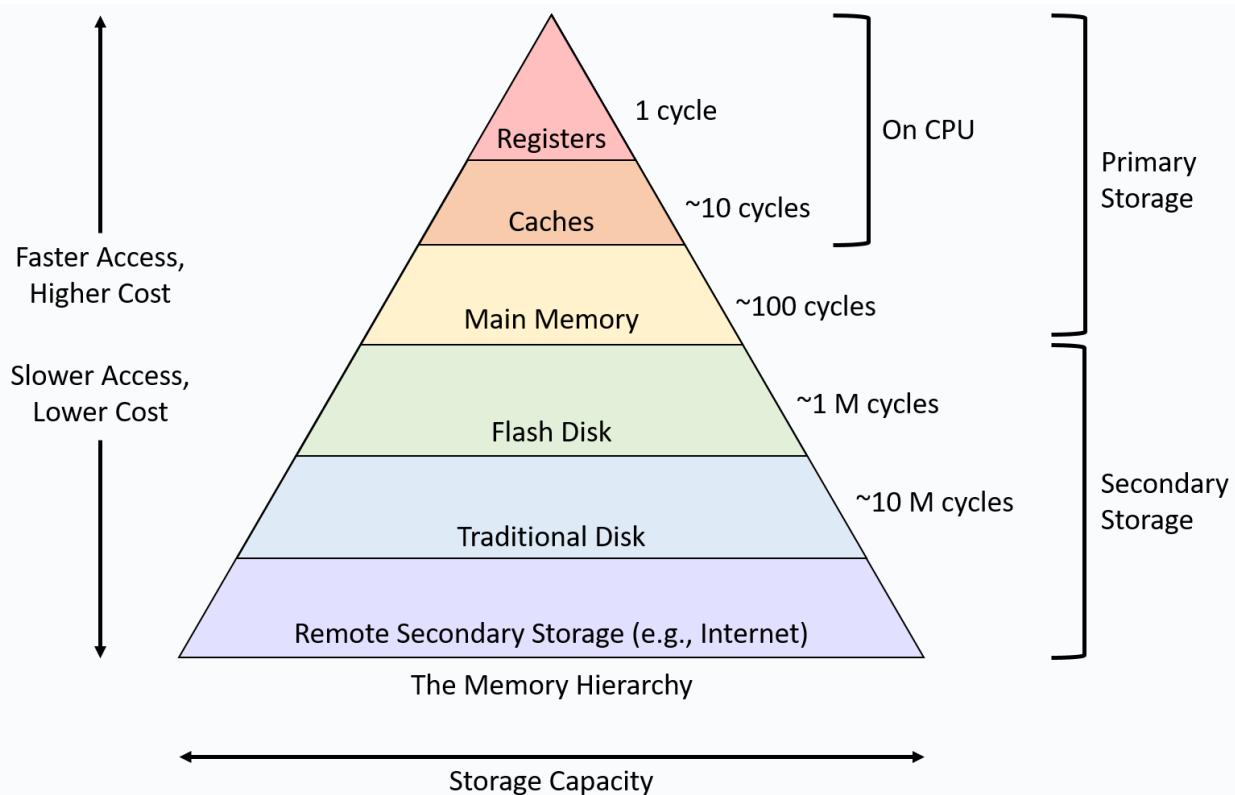
[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Memoria nei sistemi ad processore
 2. Approcci storici
 3. La memoria virtuale
 4. Rimpiazzamento delle pagine
 5. Layout della memoria
 6. Loader, librerie e pagine condivise
 7. Gestione della memoria in Bash
-

Richiamo alla Memoria dei Sistemi a Processore

I sistemi ad processore possiedono molte *memorie*



Il compito del SO è gestire l'utilizzo della memoria da parte dei processi, con gli obiettivi di:

- *Massimizzazione delle prestazioni*: usare la memoria più veloce possibile
- *Isolamento tra processi*: evitare problemi di sicurezza e stabilità
- *Facilità per il programmatore*: si vorrebbe che il SO fosse *trasparente* per chi programma

Approcci Storici per la Memoria

Pre-S.O.

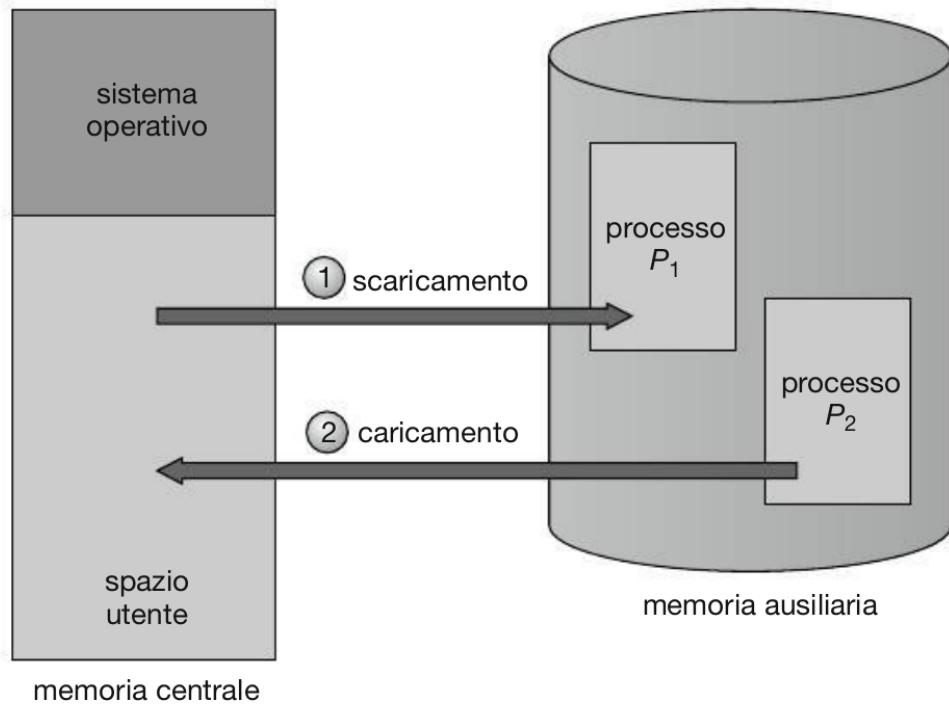
Inizialmente, non vi era SO: l'elaboratore eseguiva un programma per volta

- Un programma poteva accedere a qualsiasi locazione di memoria
Nota: ancora non esisteva la memoria virtuale e la MMU, ecc...

Sostituzione Totale

Un SO che sostituisce completamente la memoria principale del programma in esecuzione ad ogni *Context Switching*

- L'approccio più *fallimentare* e *lento*: ho troppo tempo sprecato!

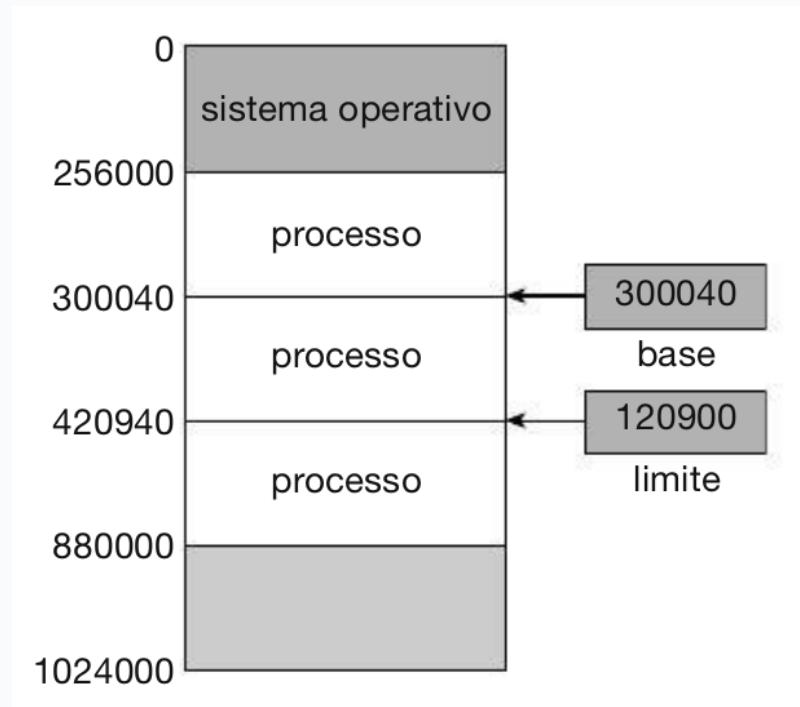


Approccio Base e Limit

Introdotta negli *anni '70-'80*

IDEA.

- Più processi condividono la memoria
- Hanno il permesso di accedere a una sola zona di memoria
- Le "*fette*" sono identificate da due numeri
 - Ogni processo ha una sua "*fetta*"



Molto importante, ebbe un grande successo

- Nasce il concetto di *Indirizzo Logico o Virtuale*

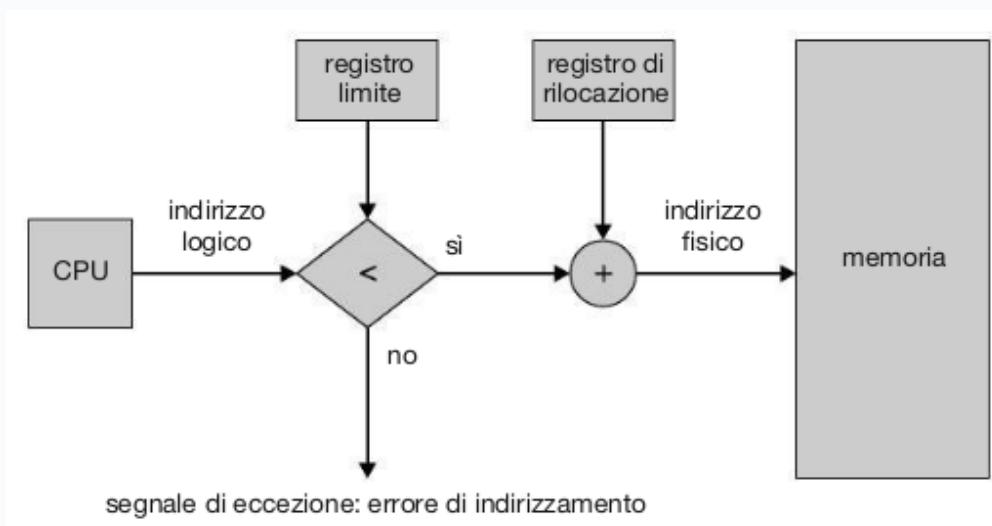
- Utile per la **sicurezza** della memoria
- La CPU ha i registri **Base** e **Limit**, settati dal SO
- Essa permette ai processi di emettere solo indirizzi consentiti

Pro:

- Permette di avere **più processi**
- **Sicuro**: un processo non può accedere a memoria di altri

Contro:

- Allocazione **contigua**: si rischia spreco di memoria
 - Rischio la **frammentazione esterna**
- Poca flessibilità



Approccio a memoria segmentata

Come Base + Limit, ma ogni processo ha a disposizione più **segmenti**
 Solitamente, ogni segmento ha scopi diversi:

- Segmento di **Codice**
 - Segmento di Dati (Variabili Globali e Costanti)
 - Segmento di Stack (Variabili di funzioni)
- Sostanzialmente questa è una **generalizzazione** dell'approccio a memoria segmentata, migliorandola (quasi!) in ogni aspetto

Pro:

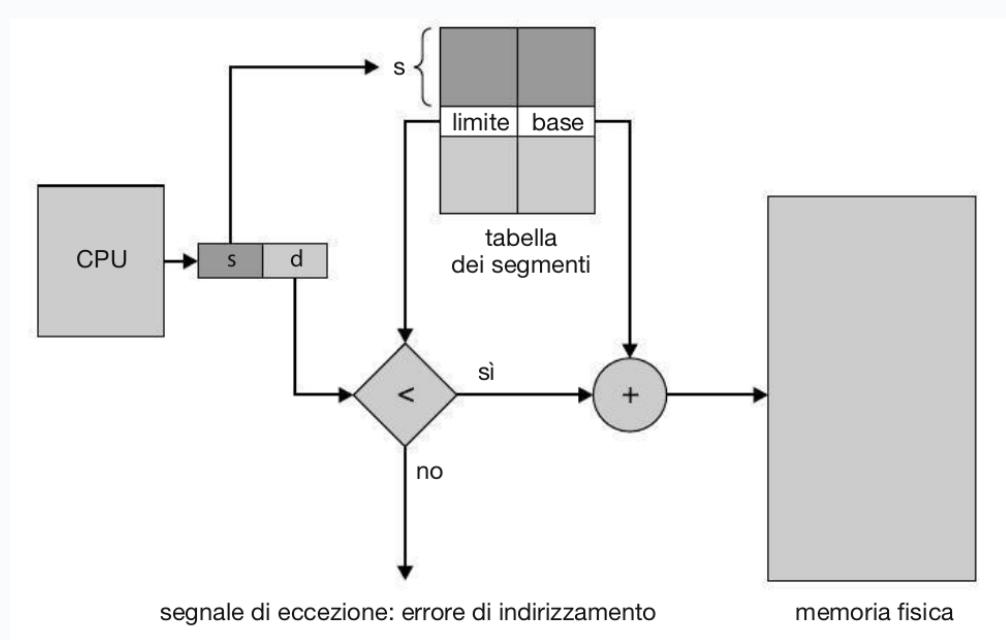
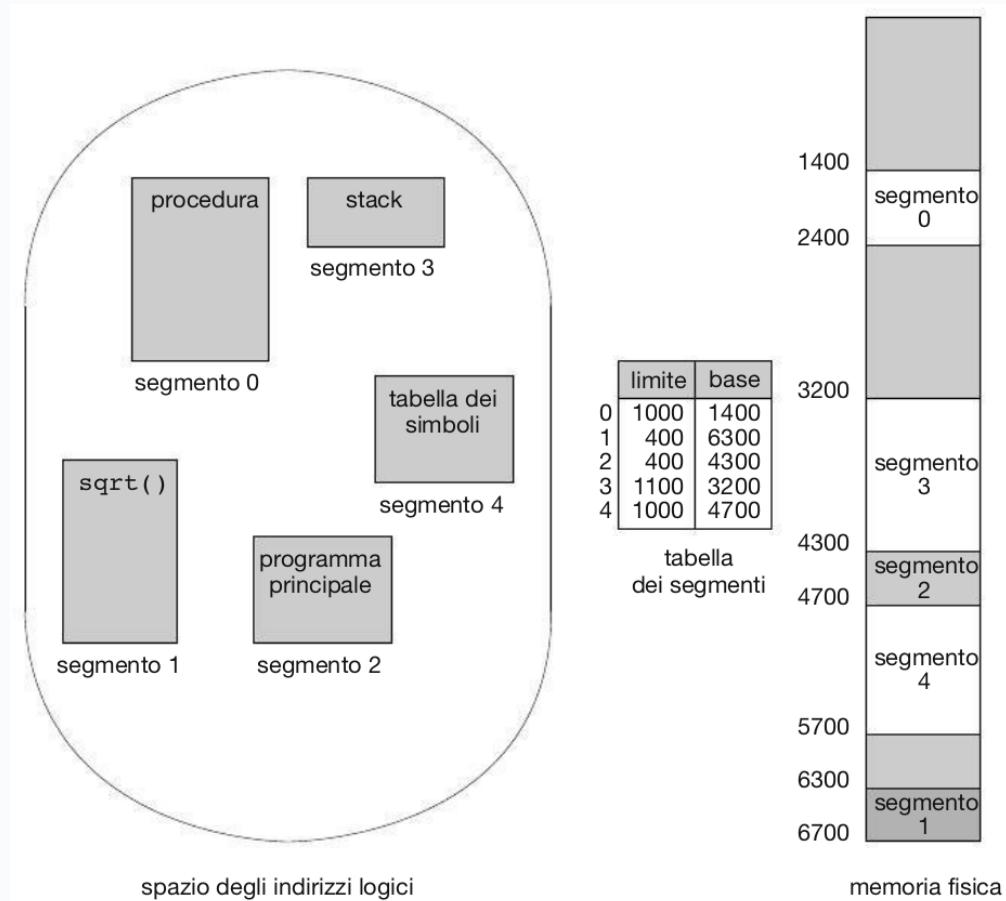
- Abbastanza **flessibile**
- **Semplice** da implementare
- Veloce

Contro:

- Segmenti di lunghezza diversa sono problematici da gestire

- Introducono **frammentazione** come in *Base + Limit*
 - Il limite principale non è stato superato

E' stato comunque molto usato negli anni '80 e '90



Approccio a paginazione

Vediamo l'approccio moderno.

IDEA.

Un processo emette *Indirizzi Virtuali*

- Un modulo hardware detto *Memory Management Unit* li traduce in *Indirizzi Fisici*

Lo spazio degli indirizzi virtuali è diviso in blocchi di lunghezza fissa dette *pagine*

- Una tabella mappa la posizione delle pagine dallo spazio virtuale a quello fisico

In generale abbiamo che:

- Il processo vede uno *spazio virtuale* che è diviso in *pagine di grandezza dimensione fissa*
- La *tavella delle pagine* indica come sono state allocate nella memoria fisica
- La *MMU* effettua la traduzione
- Il *SO* imposta/programma la MMU
- Tipicamente lo spazio degli indirizzi virtuali è più grande di quello degli indirizzi fisici
 - *Esempio.* Il programma può emettere indirizzi su 1024 pagine, ma in memoria ce ne stanno solo 80. Se non basta, vedremo cosa succede; per ora si "spera" che tutto basti
- Il programmatore non deve sapere quanta memoria ha il sistema; gestisce tutto il sistema

Pro:

- No frammentazione (ho *pagine* con dimensioni fisse)
- Flessibile
- E' lo standard *de-facto* *
- Utilizzato in tutti i moderni processori e SO

Contro:

- Richiede un Hardware veloce, che assiste tutto il processo (in particolare deve avere anche la *MMU*); tuttavia è l'*unico* difetto superabile.

Figura: (*Livello circuitale*)

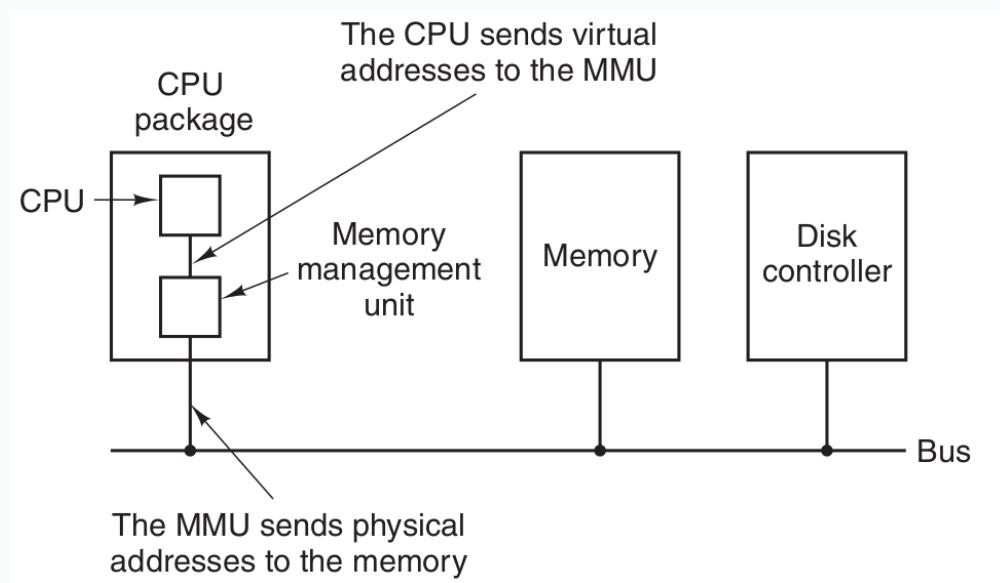
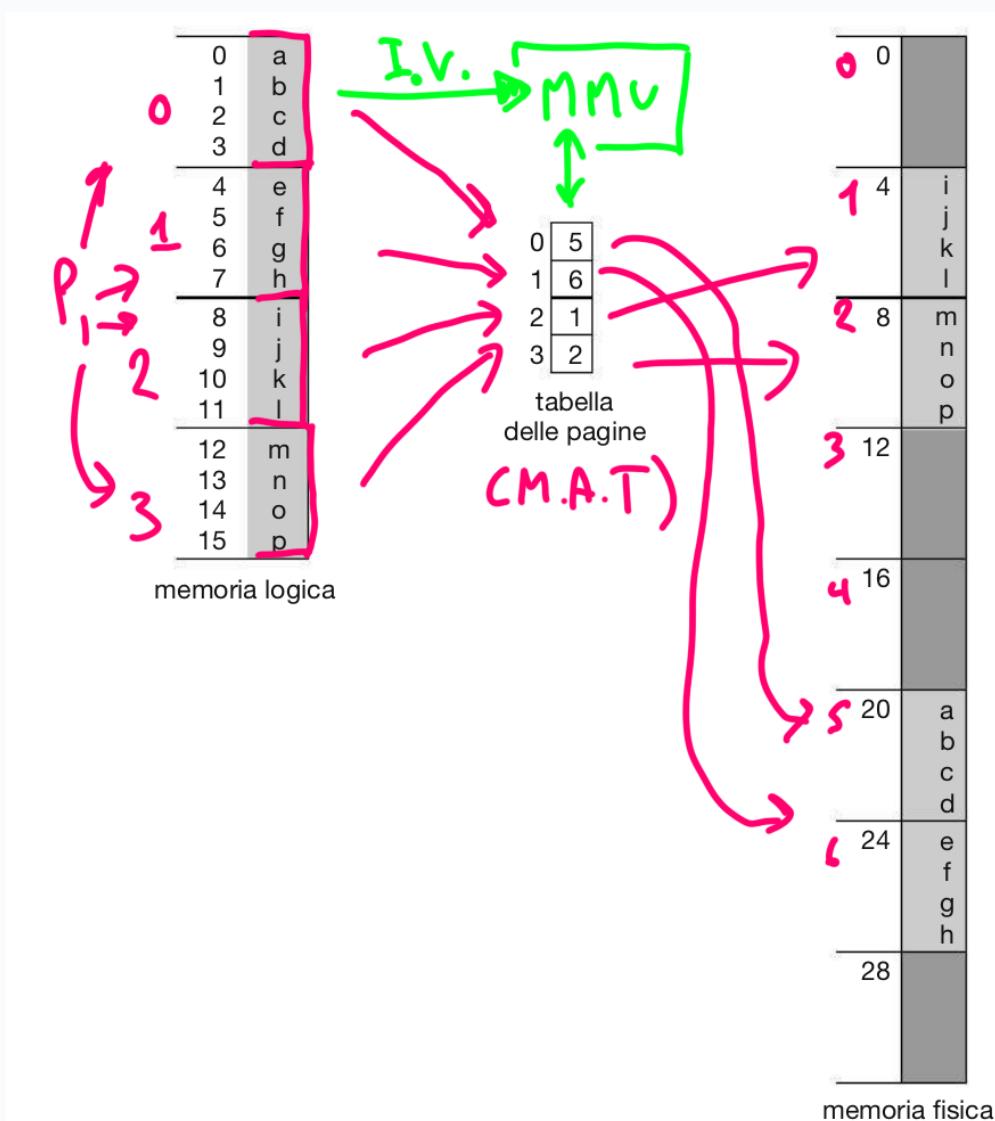
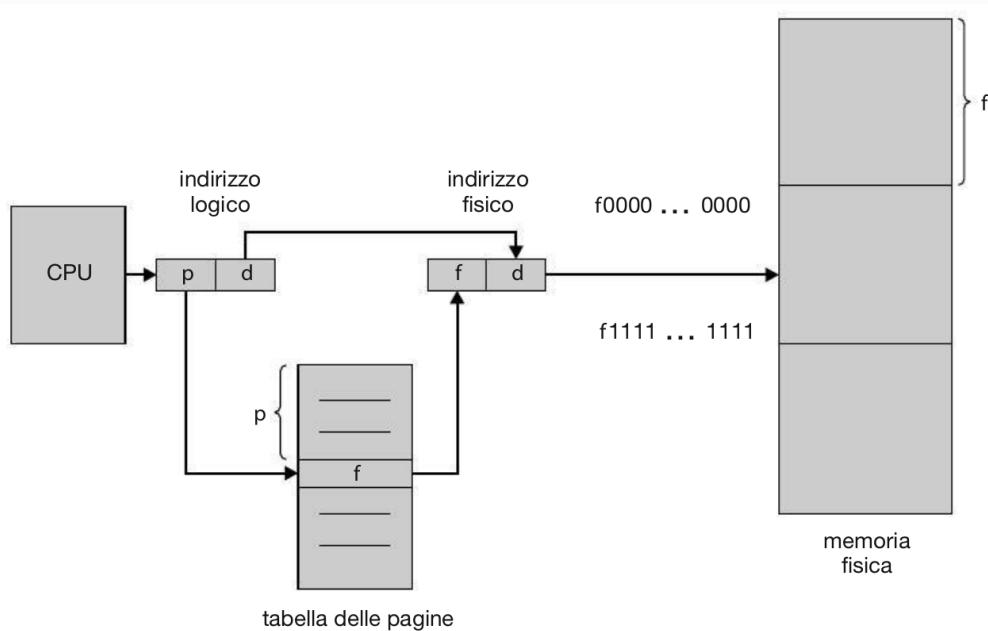
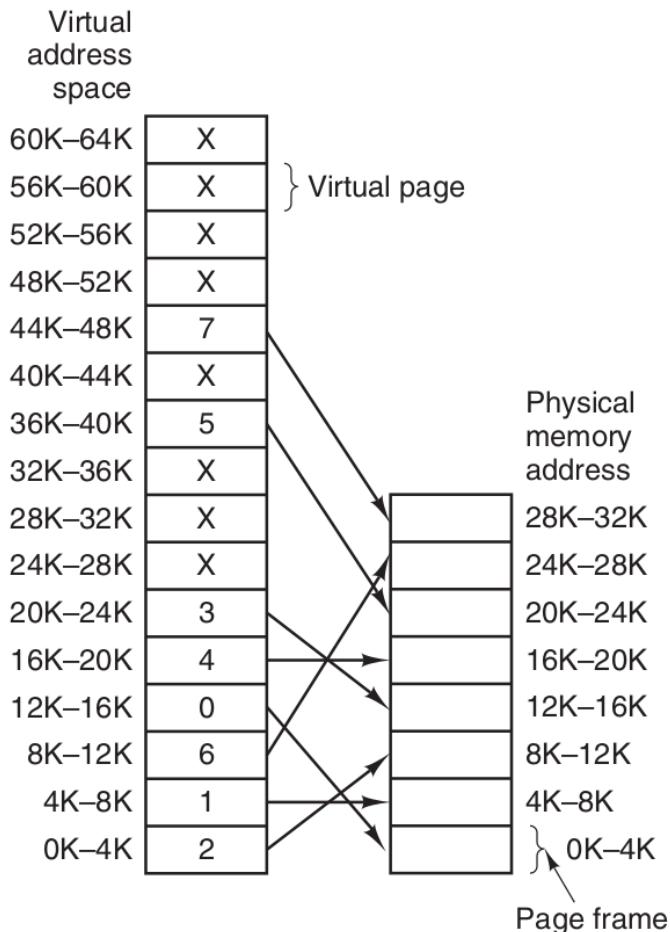


Figura: (Livello logico)





La memoria virtuale

E' il naturale effetto *della memoria paginata*

- Il processo vede uno spazio di *indirizzi virtuale*, *mappato sulla memoria fisica*

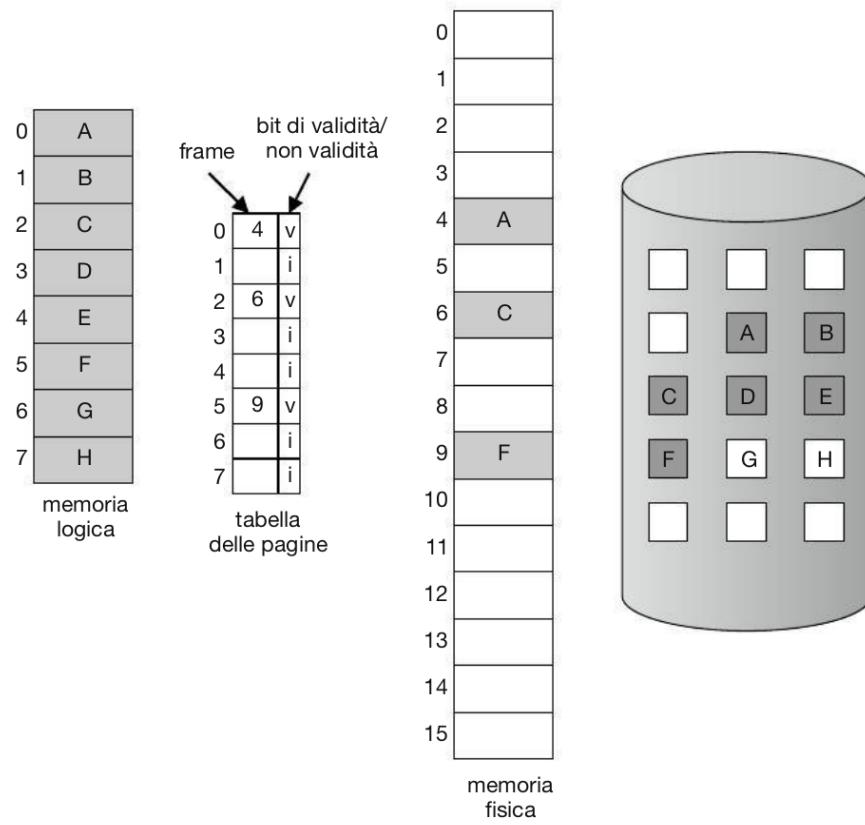
Architettura x86-64:

- Pagine da 4KB
 - Offset da 12 bit* ($\log_2(4000) \approx 12$)
- Indirizzi Virtuali su 64bit, ma solo 48 significativi utilizzati
 - Memoria virtuale di **256TB**
- La MMU traduce ad indirizzi fisici di 48bit
 - Ma la memoria fisica è sempre **molto** più piccola; di solito abbiamo fino a 32GB per un computer personale.

Swap

Se le pagine non stanno tutte in memoria, si mettono su disco.

Lo spazio su disco che contiene le pagine non in memoria si chiama **Swap**

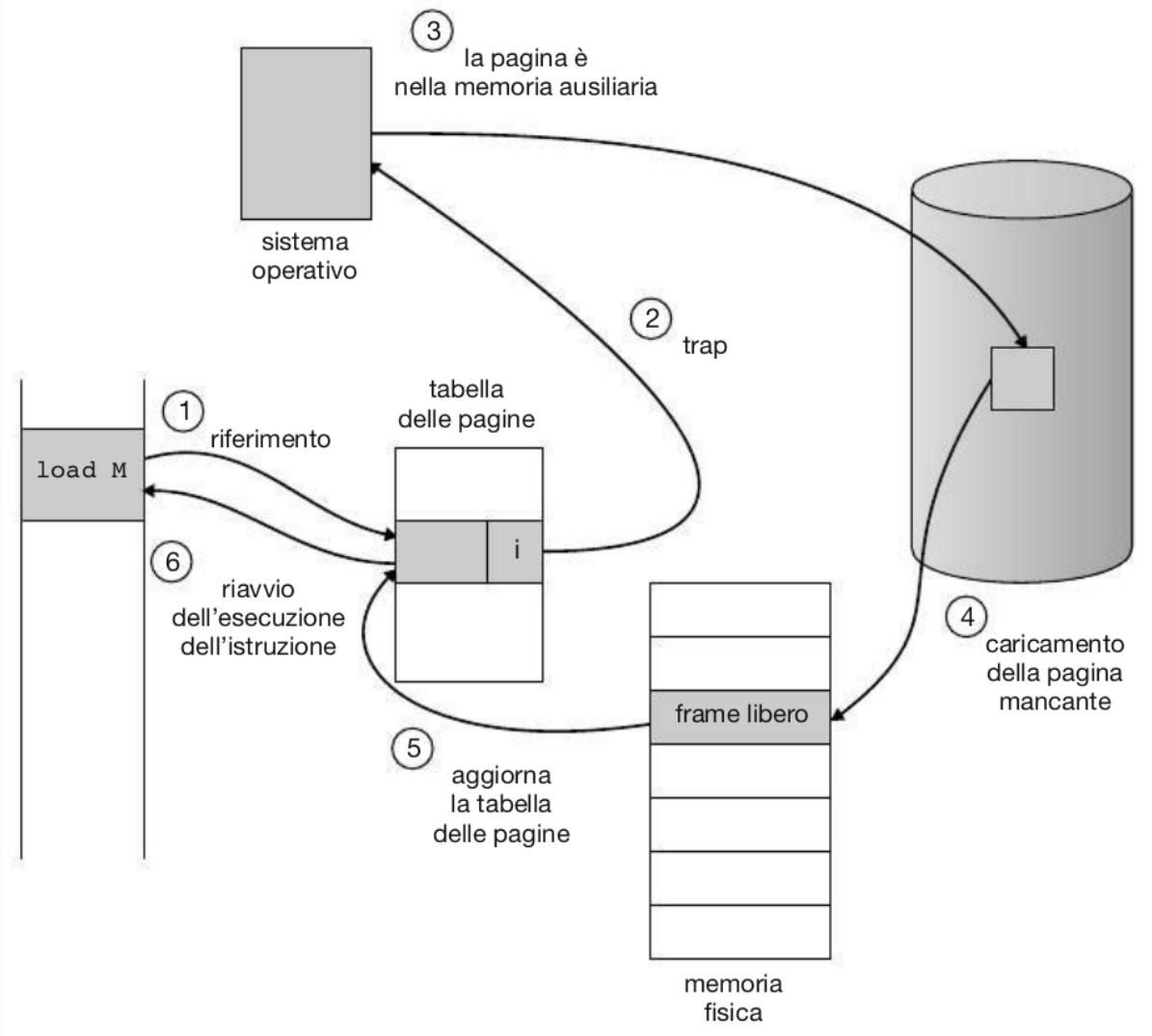


Page Fault

Se il processo *emette un indirizzo di una pagina che non è in memoria*, si verifica un *Page Fault*:

1. La MMU avverte il SO
2. Il SO *interrompe il processo*
3. Il SO *carica* la pagina (o la crea) da disco
4. Il SO *reimposta* la MMU
5. Il processo *riprende*

Questo è un processo molto *lento*, quindi da *minimizzare*!



Rimpiazzamento delle pagine

La memoria fisica è *sempre più piccola* di quella virtuale

Se è piena di pagine utilizzate da processi attivi, il SO deve *scegliere* quale pagine eliminare e salvare su disco

Esistono diversi *algoritmi di rimpiazzamento* per effettuare ciò in maniera furba

Algoritmo FIFO

Rimuovo dalla memoria *la pagina caricata da più tempo*

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	4	4	4	4	0			0	0		7	7	7	
	0	0	0		3	3	3	2	2	2	3		1	1		1	0	2	

- Semplice, ma inefficiente

- **Ragionamento analogico:** A casa non ho abbastanza spazio per mettere nuove cose; quindi prendo la cosa più **vecchia**, come il **divano** e la metto in cantinato.

Algoritmo Ottimo

Rimuovo la pagina che non mi servirà per più tempo ***nel futuro***

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



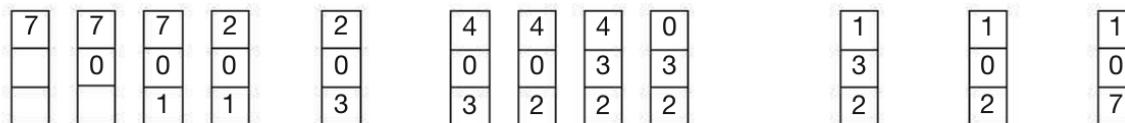
frame delle pagine

- Ottimo, ma **impossibile prevedere il futuro** (purtroppo)
 - **Ragionamento analogico:** A maggio tolgo gli scii, a dicembre tolgo la tavola da surf. Purtroppo impossibile da "**automatizzare**" ragionamenti del genere in una maniera rigorosa

Algoritmo Least Recently Used

Rimuovo la pagina che **non viene usata da più tempo**

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



frame delle pagine

- Semplice, efficace.
- Serve collaborazione della **MMU** per tenere traccia di accessi
- Usato quasi sempre (con varianti)
- Non si fa differenza tra processi: le pagine sono uguali

Layout della memoria

Un processo può accedere a **qualsiasi locazione di memoria nello spazio degli indirizzi virtuali**

- Lo spazio degli indirizzi virtuali è **diviso in pagine**
- Se la pagina è in **memoria**, la MMU traduce in **indirizzo fisico**
- Se la pagina **non è in memoria**, il SO la creerà/preleverà **da disco**

Organizzazione della Memoria

Abbiamo molti modi per *organizzare* l'accesso agli indirizzi virtuali.

Un programma che accede a indirizzi "casuali" non è efficiente

- Utilizzo di pagine e memoria sarebbe *molto penalizzato*

Storicamente si cominciavano a usare indirizzi a partire *da quelli "bassi"*:

- Si inizia a utilizzare indirizzo **00 00 00 00**, poi **00 00 00 01**
- Così si riempie una pagina completamente, poi se inizia a usare un'altra

Ci sono diverse convenzioni, che dipendono da architettura dell'elaboratore e OS. Noi vediamo quella per *Linux*

Convenzione Linux

Attualmente, si usano *sia indirizzi all'estremo alto che all'estremo basso*

- La memoria può crescere in due direzioni
- Posso avere due zone di memoria che crescono a seconda dell'esigenza del programmatore

Principalmente abbiamo *direzioni*:

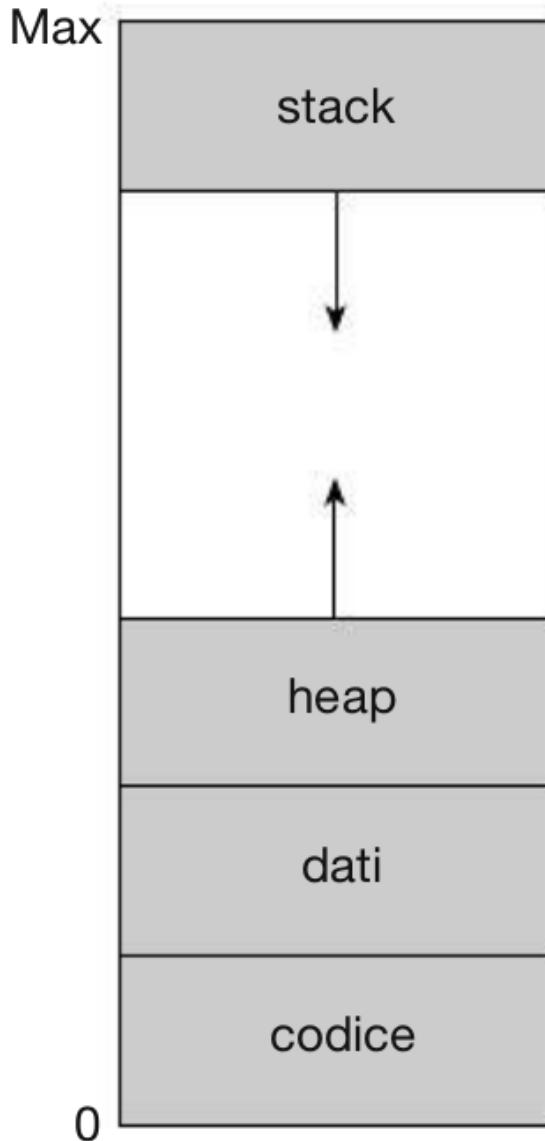
- **Heap** (ξ): cresce dal basso verso l'alto. Usato dal programmatore per allocare memoria quando gli serve
- **Stack** ($\sigma - \rho$): cresce dall'alto verso il basso. Usato dal compilatore per posizionare le variabili delle funzioni

Un processo può accedere a *qualsiasi locazione di memoria*.

Per convenzione e prestazioni si preferisce iniziare gli *estremi*

Ci sono 4 *zone di memoria*:

- Codice (ζ) - *Dimensione fissata*
- Dati (δ) - *Dimensione fissata*
- Stack ($\sigma - \rho$) - *Dimensione variabile dall'alto al basso*
- Heap (ξ) - *Dimensione variabile dal basso all'alto*



Adesso andiamo ad approfondire le parti varie.

Codice ζ

Il SO copia il codice del programma dal disco *verso gli indirizzi più bassi*

- Il codice deve obbligatoriamente trovarsi *in memoria*
- Il registro *Program Counter* della CPU punta a un indirizzo in questo range

Questa parte della memoria è *Read Only* (imposta dalla *CPU*): un programma non può modificare se stesso (senno ci sarebbero casini assurdi)

Dati δ

Gli indirizzi immediatamente *maggiori* del codice, sono usati per le *variabili globali*

- Il compilatore usa questi indirizzi per le variabili globali
- Le variabili globali *inizializzate* vengono riempite direttamente dal SO quando viene avviato il processo
 - Per questo quando inizializziamo gli *array*, devono avere *dimensione nota*.

- Le altre contengono tutti '0' (ovvero hanno valore 0)

Heap ξ

Usato per la *Memoria Dinamica*

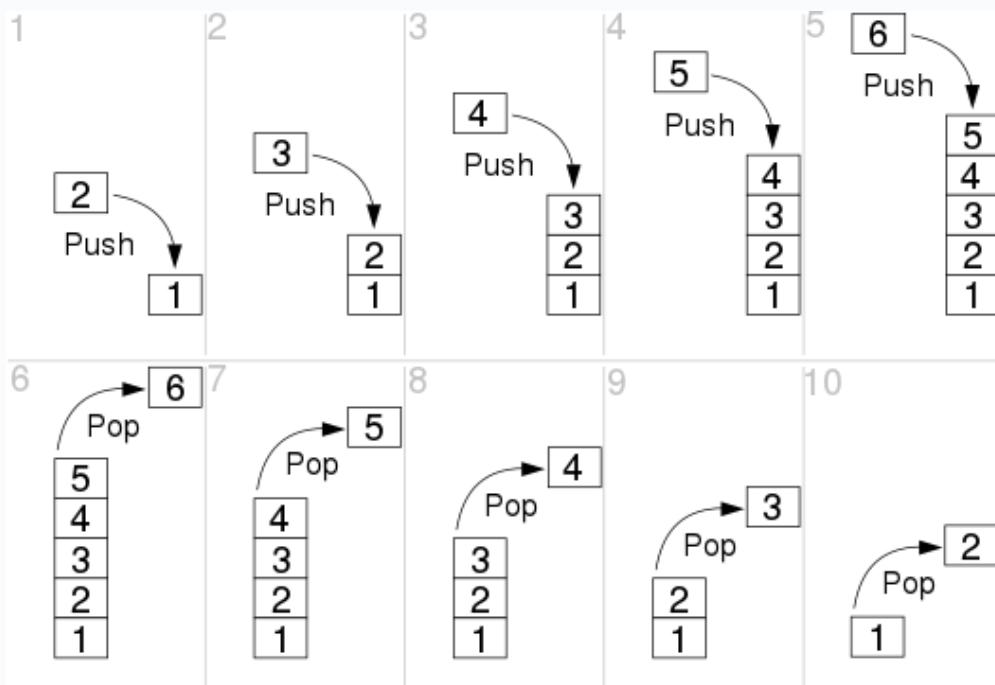
- Il programmatore può aver bisogno di memoria la cui dimensione non viene prevista in fase di programmazione
- Gestita tramite le funzioni di libreria *malloc* e *free*
- Vedremo *in seguito*

Stack $\sigma - \rho$

Usato per le variabili relative a funzioni: argomenti e variabili interne

- Come dice, questa zona è gestita come se fosse una *pila*
I dati vengono:
• Impilati per essere aggiunti: *Push* (aggiungo stack)
- Tolti dalla pila quando devono essere usati: *Pop* (rimuovo uno stack)
Si comportano come una *specie di molla*, che va su e giù

I dati vengono aggiunti e tolti dalla cima della pila come nella seguente figura:



- Abbiamo una struttura del tipo *last in, first out* (LIFO)

Operazioni di Push e Pop

Concetto pratico per gestire le *funzioni*!

Quando viene chiamata una funzione, si aggiunge un blocco allo stack contenente (*Push*):

- Indirizzo di *ritorno*
- Parametri* copiati (passaggio per valore)

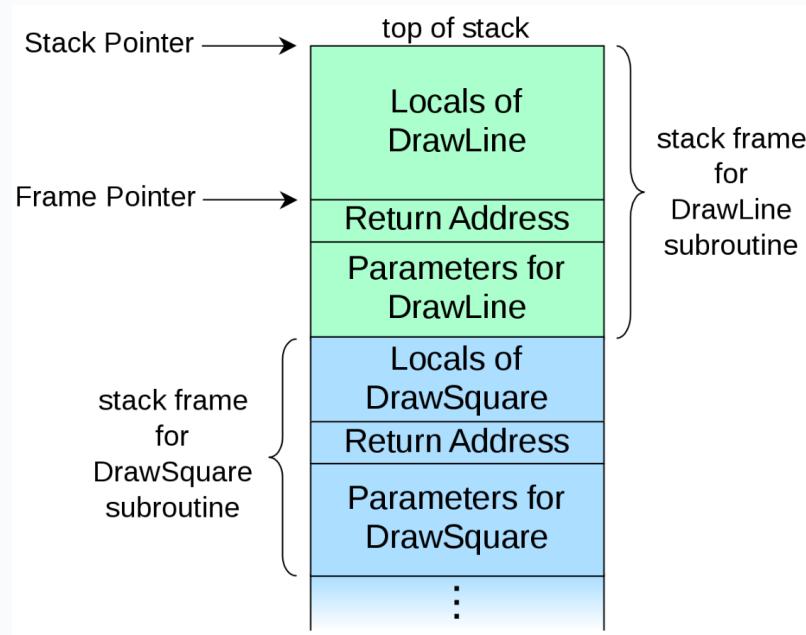
- *Variabili locali* (si cerca il più *vicino*, preferibilmente nello stack stesso)

Quando la funzione ritorna, il blocco si elimina (*Pop*)

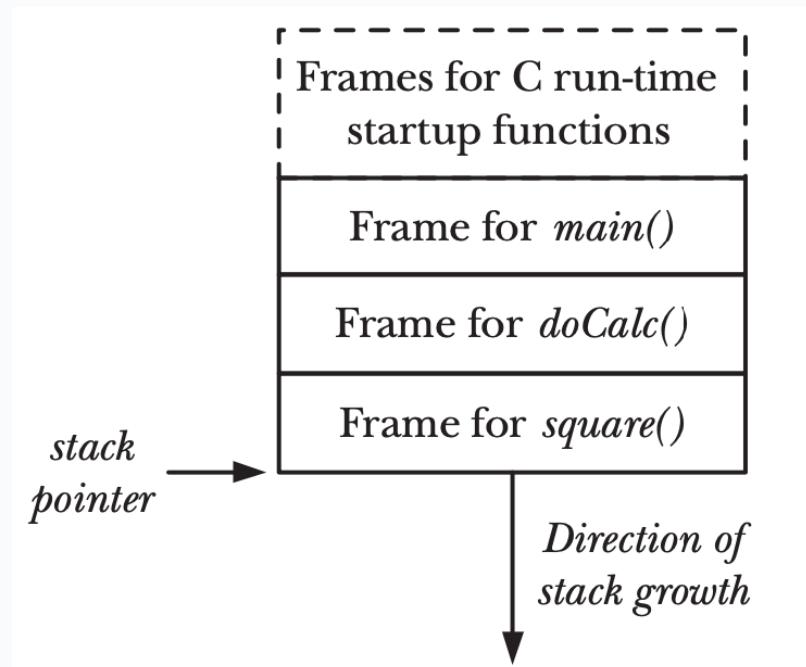
Ogni blocco si chiama *Stack Frame*

- Creato quando la funzione viene invocata
- Cancellato quando la funzione ritorna

Stack



Stack Frame



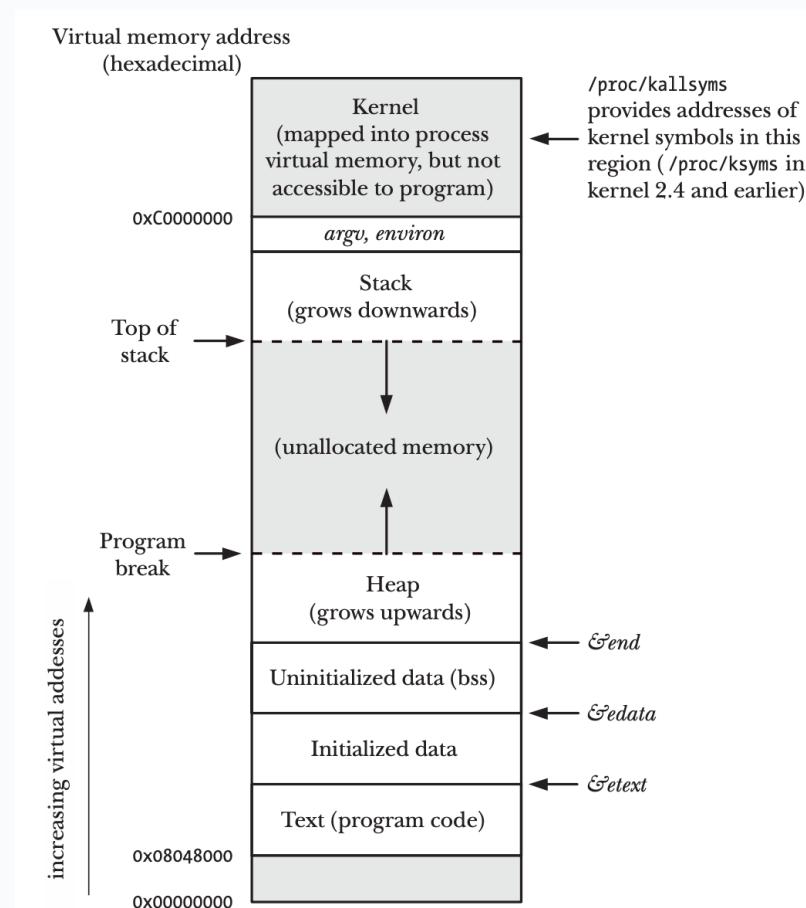
Stack Frame in Linux

Il layout visto prima è *generico*.

In Linux, precisiamo che ci sono altri costrutti.

Per *data* δ :

- Un segmento per *variabili globali inizializzate* e uno per *quelle non inizializzate*
Per *stack* $\rho - \sigma$:
 - **argc** e **argv** in indirizzi alti
 - **Top of Stack**: indirizzo minimo per parte alta
- Per *heap* ξ :
 - **Program Break**: indirizzo massimo per parte bassa



Loader, librerie e pagine condivise della Memoria

Loader

Il *Loader* è il componente del SO che *avvia i processi*.

I suoi compiti sono:

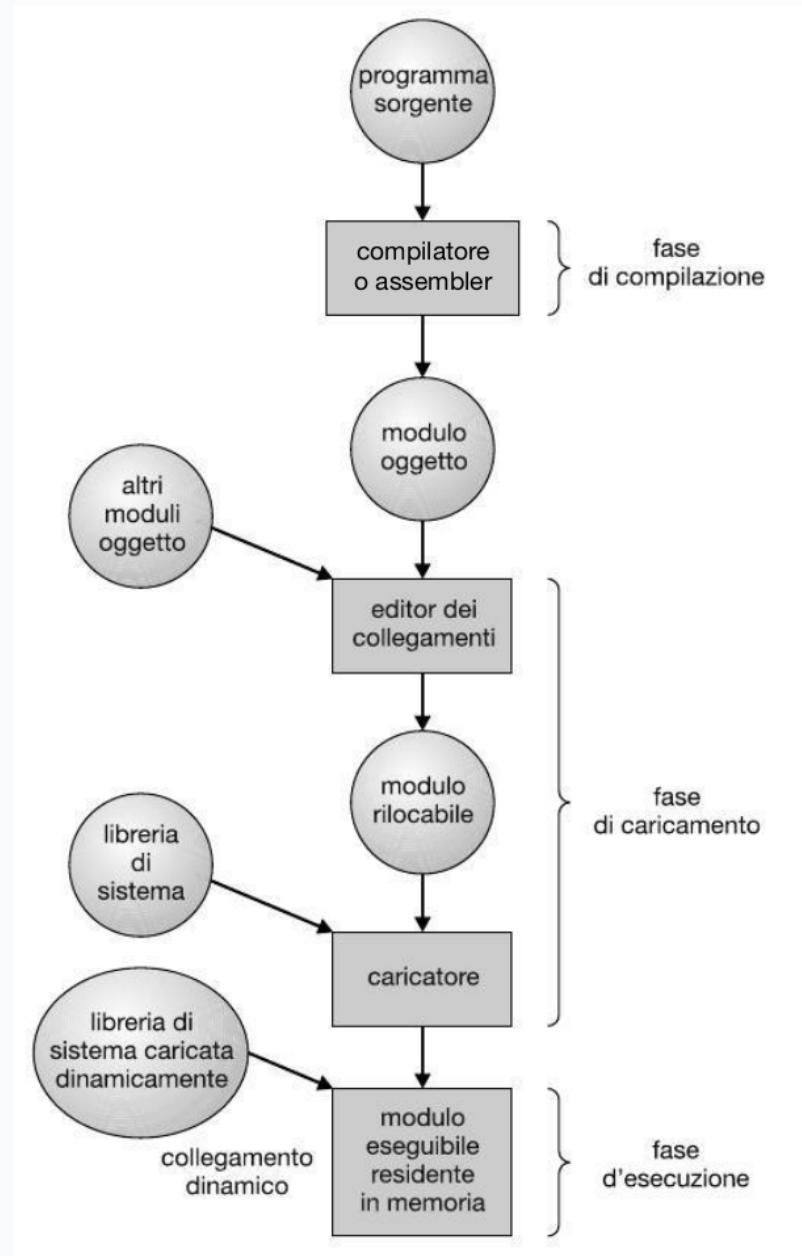
- Verificare che l'utente abbia i *permessi*
- Copia il codice del programma e le variabili globali inizializzate *in memoria* (δ)

- Caricare le *librerie condivise*
- Valorizzare **argc** e **argv** (copiate in alto)
- Avviare il processo dal **main** impostando lo *scheduler* del SO

Il compilatore crea il codice macchina

Il loader lo *carica* e ne *avvia* l'esecuzione

- Deve caricare *anche le librerie di sistema*, se sono usate



Librerie Condivise

I programmi possono usare *librerie condivise*:

- Offerte *dal SO* per facilitare la chiamata a System Call
- Installate da utente per scopi particolari (e.g., trigonometria)

Le librerie sono codice che gira in *User Mode*

- Non hanno alcun privilegio rispetto al codice utente
- **Non** sono parte del Kernel

Le **librerie condivise** sono **codice eseguibile** in cartelle predefinite del sistema

In **Linux**:

- **/lib**
- **/usr/lib**
- Directory elencate nel file **/etc/ld.so.conf**

In **Windows**:

- **C:\Windows\SYSTEM32**
- Cartella corrente

Formato ELF Linux

Gli eseguibili in Linux sono in formato ELF (*Executable and Linking Format*)

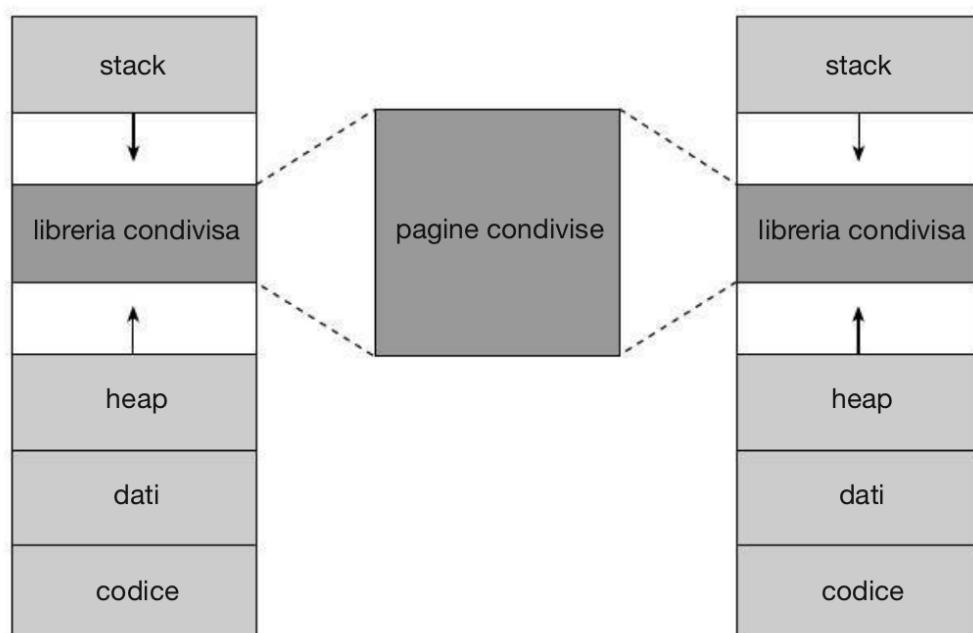
- Oltre il codice, contengono **la lista delle librerie di sistema** che useranno
- Contenute in una posizione predefinita

Le librerie e condivise sono identificate dal nome e dalla versione

Locazione delle Librerie Condivise

Le librerie sono caricate **dal Loader** in indirizzi intermedi

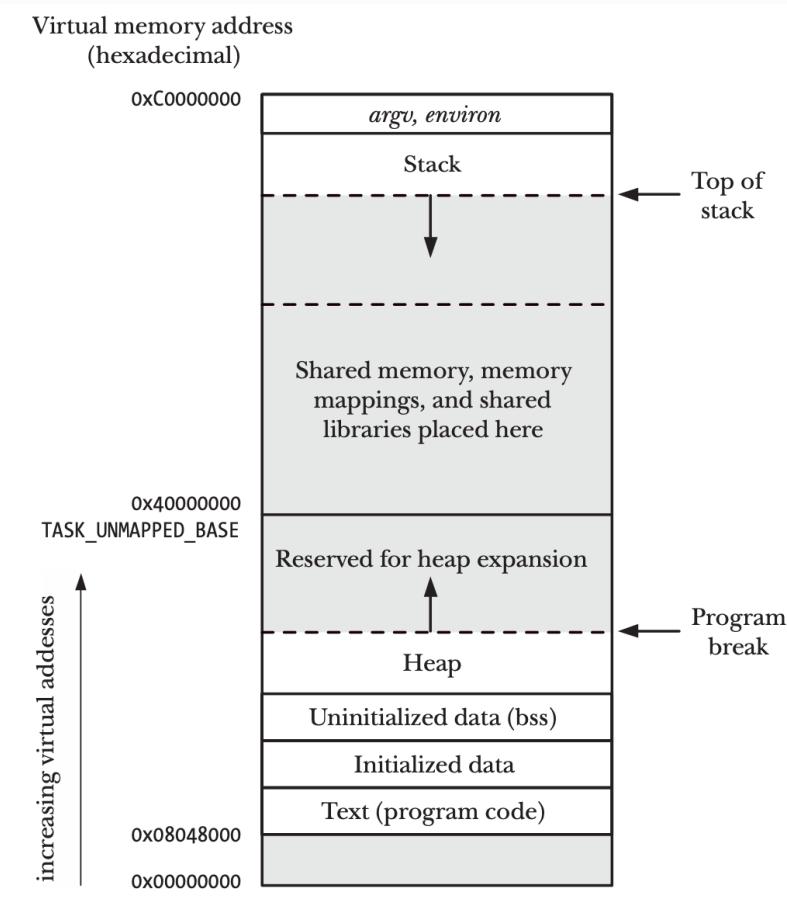
Se più processi usano la **stessa libreria**, la **pagina viene condivisa** (ovvero gli indirizzi virtuali vengono tradotti negli stessi indirizzi fisici)



Memoria Condivisa

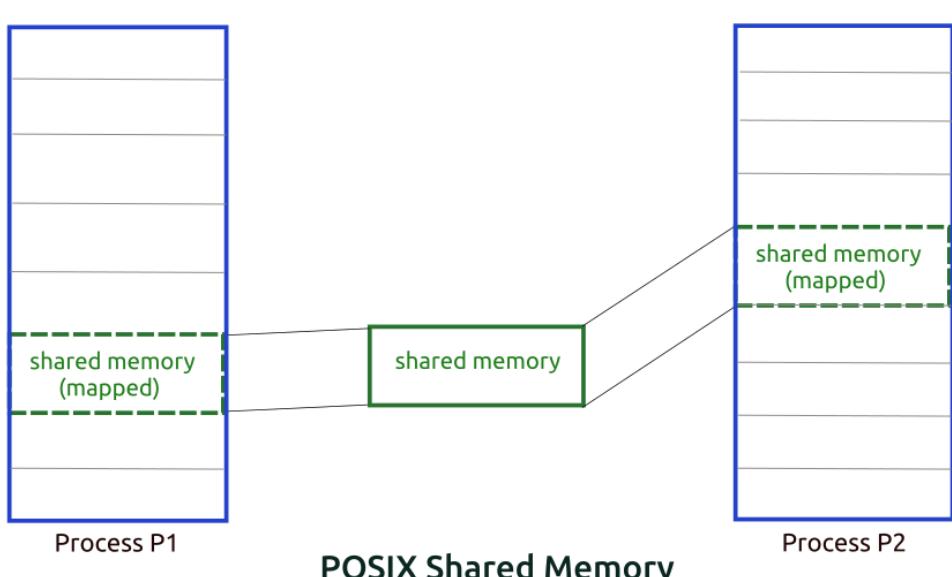
La memoria condivisa tra processi funziona *nello stesso modo*

- Gli *indirizzi intermedi*, tra heap e stack, sono usati per tutto ciò che è condiviso
- Esempi: *memory mapping*, eccetera...



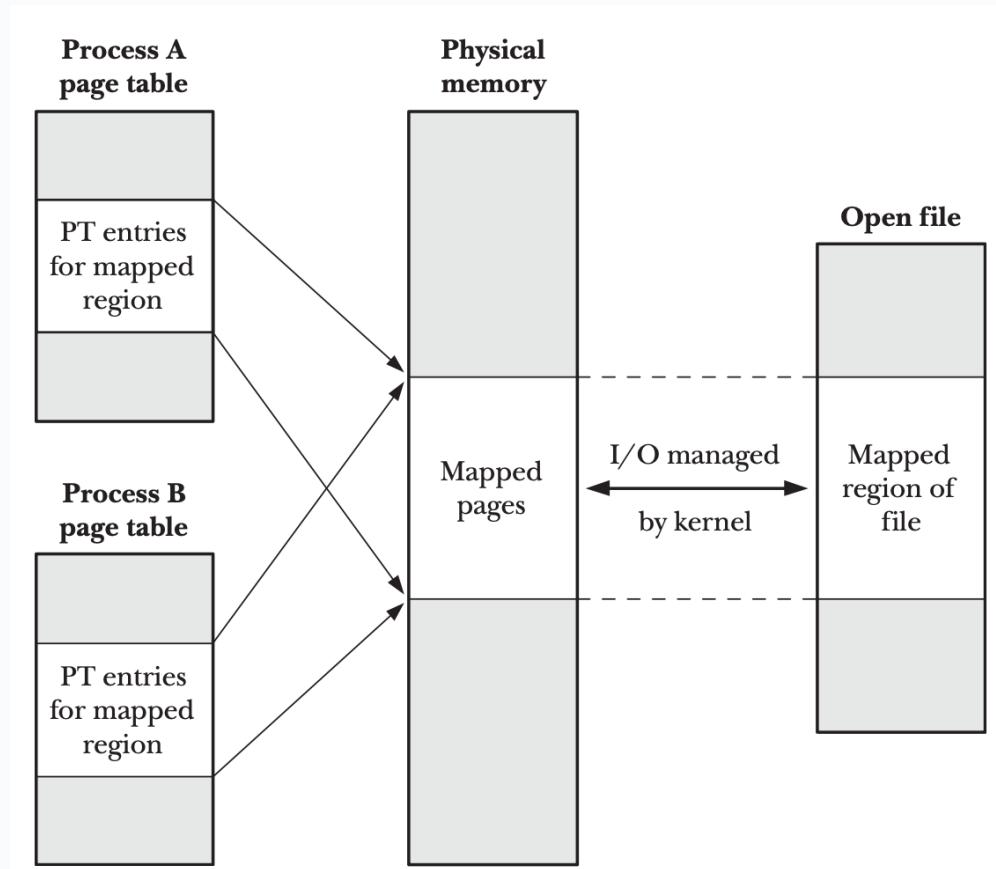
Tutto ciò è possibile *grazie alla MMU*

- Il SO imposta la MMU per *implementare questo schema*
 - Due indirizzi virtuali vengono *tradotti* per puntare alla medesima zona di memoria condivisa



Esempio di Memoria Condivisa

In caso di **mmap** con *persistenza su file*, il kernel si occupa di allineare la zona di memoria condivisa *su disco*



Gestione della memoria in Bash

Vediamo dei comandi per *gestire la memoria* in Bash.

Lista. (*Comandi Bash*)

- **free**: mostra quanta memoria è disponibile/utilizzata/libera nella macchina
- **top**: mostra varie informazioni sui processi
 - Colonna **RES**: *Resident Set Size* quante *pagine* dello spazio virtuale di un processo sono *caricate in memoria fisica*
 - Colonna **VIRT**: *Virtual Set Size* quante *pagine sono state usate* dal processo nella sua storia
 - Colonna **%MEM**: **RES/totale**, ovvero *percentuale di memoria fisica* della macchina contenente pagine del processo
- **cat /proc/meminfo**: mostra informazioni dettagliate su memoria della macchina (questa è la fonte del comando **free**)
- **ldd eseguibile**: mostra quali librerie condivise esso richiede
- **objdump -p eseguibile**: dissector del formato ELF (*mostra* i campi dell'ELF)

Domande

Il compito della Memory Management Unit è:

- Gestire il funzionamento della cache
- Allocare zone di memoria
- Tradurre gli indirizzi da virtuali a fisici

Risposta: Tradurre gli indirizzi da virtuali a fisici

Cosa fa la MMU quando una pagina non è in memoria?

- La carica
- Termina il processo che ha generato l'indirizzo
- Avverte il SO

Risposta: Avverte il SO (mediante una Page Fault)

La zona di memoria Stack viene utilizzata per:

- Memorizzare variabili globali
- Memorizzare il codice del programma
- Contenere le variabili relative alle funzioni
- Allocare la memoria dinamica

Risposta: Contenere le variabili relative alle funzioni

Un sistema ha pagine da 1KB, indirizzi virtuali da 32bit e fisici da 16bit. Quanti bit sono dedicati all'offset di pagina?

- 6
- 10
- 22

Risposta: 10 (1KB → $2^{10} B$; allora ho 10 bit per l'offset)

Un sistema ha pagine da 1KB, indirizzi virtuali da 32bit e fisici da 16bit. Di quante pagine dispone un processo nello spazio degli indirizzi virtuali?

- 64
- 1024
- circa 4 Milioni

Risposta: circa 4 Milioni ($2^{32-10} \approx 4 \cdot 10^6$)

In Linux, il Loader è:

- Un componente del SO
- La funzione principale di un programma
- La zona di memoria dove è memorizzato il codice del programma in esecuzione

Risposta: Un componente del SO

In Linux, le librerie condivise sono mappate in memoria:

- In una zona intermedia tra stack e heap
- Nella zona di dati
- Nella zona di codice
- Nello stack

Risposta: In una zona intermedia tra stack e heap

La memoria Dinamica

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Limiti della memoria statica
 2. La memoria dinamica
 3. La funzione **malloc**
 4. La funzione **calloc**
 5. La funzione **realloc**
 6. La funzione **free**
 7. Cenni di funzionamento interno
-

Limiti della memoria statica

Diamo un po' di *motivazioni* per la *memoria dinamica*.

Variabili Globali, Locali e Statici

Facciamo prima un ripasso sulle *variabili globali, locali e statici*

Variabili Globali

Le variabili globali sono allocate nella segmento di *dati* δ (Layout della Memoria Virtuale > ^6aeac3).

Il loader *inizializza* il valore

```
int a = 40; /* Inizializzata dal loader*/  
int main(){...}
```

Se non specificato, la variabile è inizializzata a 0.

```
C  
int a; /* Inizializzata a 0 */  
int main(){...}
```

Variabili Locali

Le variabili di funzione sono allocate nello **stack** $\sigma - \rho$ ([Layout della Memoria Virtuale > ^b73f4b](#))

NON viene inizializzato il valore! *Possono contenere dati arbitrari*

```
C  
int f(int a, int b){  
    int s = a + b;  
    return s;  
}
```

Gli argomenti **a** e **b**, la variabile **s** e il valore di ritorno si trovano nello stack

Variabili Statiche

le variabili in una funzione con la keyword **static** sono allocate *nel segmento dati* δ e non nello stack.

Inizializzate dal *loader*.

Conservano in valore dopo il termine della funzione.

```
C  
#include <stdio.h>  
int fun(){  
    static int count = 0; /* Inizializzata UNA volta sola dal loader all'avvio del  
    processo.  
        E NON ogni volta che la funzione viene invocata*/  
    count++;  
    return count;  
}  
  
int main(){  
    printf("%d ", fun());  
    printf("%d ", fun());  
    return 0;  
}
```

Stampa 12

Sono effettivamente strane, non li useremo tanto

Problema delle Variabili Globali

Problema. (*Variabili globali non determinate a priori*)

Ci sono casi in cui il programmatore *non sa quanti dati deve caricare in memoria*

- Lettura di una struttura dati da file
- Input utente di lunghezza variabile

Con quello che abbiamo visto, in C gli array hanno lunghezza fissa, nota a tempo di compilazione

```
#define N 50  
int v[N];
```

Come soluzione si può optare per la seguente

Sovradimensionamento: approccio con cui i programmatori creano vettori o matrici di dimensione molto grande

- Atti a contenere (quasi) ogni possibile input
- Approccio funzionante, ma non risolutivo
- La *Memoria Dinamica* risolve questo problema

Cosa non si può assolutamente fare è il seguente.

In C, **NON** si possono creare array di lunghezza non nota al compilatore

Il seguente codice è *sbagliatissimo!!!!!!*

```
scanf("%d", &n);  
int v[n];
```

Esempio di Sovradimensionamento

Esempio di sovradimensionamento: media di N numeri letti da tastiera

Nota: la media di N numeri si può calcolare anche senza tenerli in memoria. Basta tenersi la somma Σ

```
#include <stdio.h>
#define MAXN 50 /* Se n>50 il programma non funziona */
int main() {
    int n, i;
    float v[MAXN], s = 0;

    printf("Quanti numeri vuoi leggere? ");
    scanf("%d", &n); /* Se n>50 il programma non funziona */
    printf("Inserisci %d numeri:\n", n);

    for (i=0; i<n; i++)
        scanf("%f", &v[i]); // &v[i] == v+i
    for (i=0; i<n; i++)
        s += v[i];
    printf("Media: %f\n", s/n);

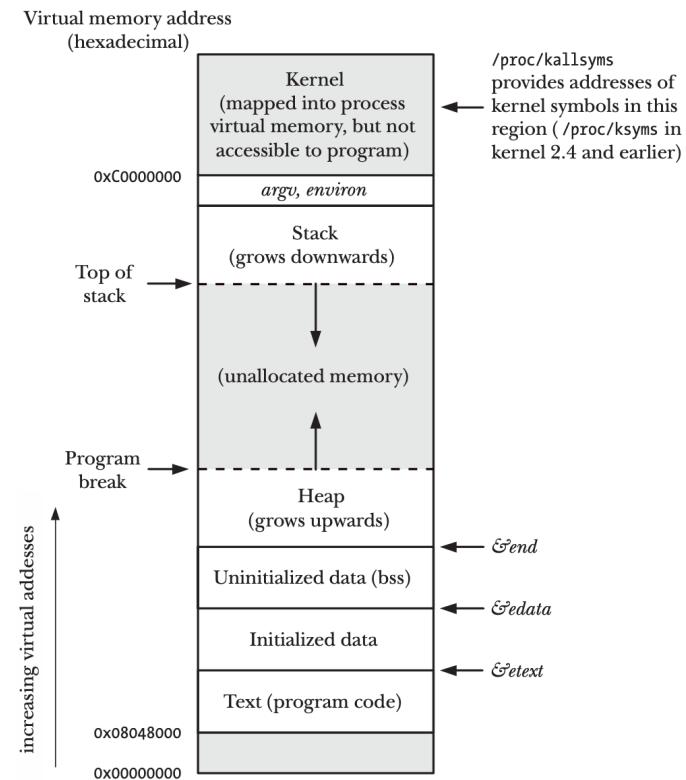
    return 0;
}
```

La memoria dinamica

In C è possibile utilizzare la *memoria dinamica* per creare *strutture dati* la cui dimensione non è nota in fase di compilazione

Uso tipico: creazione di vettori di lunghezza arbitraria e decisa a *run time*. Esempio: lista linkata

Funzionamento: si utilizzano indirizzi virtuali nel segmento *Heap* ξ . Esso può crescere durante l'esecuzione del programma. Il *program break* serve per "stabilire" il limite degli indirizzi virtuali nello heap ξ , e si può andare a modificarlo.



Manipolazione della Memoria Dinamica in Linux

In Linux

Per utilizzare la memoria dinamica si utilizzano delle *funzioni di libreria* (**malloc**, **free**, ecc...) per *allocare o liberare blocchi di memoria*.

Le funzioni di libreria utilizzano la System Call **sbrk** che informa il sistema operativo che il processo emetterà indirizzi virtuali *in zone precedentemente* non usate (aumenta o decrementa il *program break*).

- In pratica si informa il SO che l'Heap sta crescendo e il processo accederà a *pagine di memoria virtuale aggiuntive*

Thread-Safe

Tutte le funzioni di libreria per la memoria dinamica sono *Thread Safe*.

- Possono essere invocate in parallelo da molteplici *thread*
 - *Ma non all'interno di Signal Handler!*
- Internamente mantengono e usano *mutex* (che vedremo) per regolare l'accesso alle strutture dati

Funzione **malloc**

```
#include <stdlib.h>
void *malloc(size_t size);
```

Allocà **size** byte di memoria e ritorna il puntatore alla memoria allocata.

La memoria **NON** è inizializzata, può contenere qualsiasi valore

Se l'allocazione **fallisce** (e.g., manca memoria), ritorna **NULL**. Altrimenti ritorna l'indirizzo **void *** da convertire mediante un cast.

Note (*Utilizzo*)

1. La **malloc** richiede **size** in byte. Bisogna utilizzare l'operatore **sizeof** per conoscere la dimensione del tipo di variabile da allocare. Ricordiamoci che viene valutata in *fase di compilazione*!
2. Il valore di ritorno è **void ***, ovvero un puntatore senza tipo.
Per utilizzare la memoria allocata, conviene *assegnarla a un puntatore al tipo desiderato*

Esempio:

```
/* Vogliamo allocare un vettore di float*/  
float * v;  
/* La lunghezza è determinata a run time */  
scanf("%d", &n);  
/* I byte da allocare sono n blocchi ognuno lungo quanto un float */  
v = malloc(n * sizeof(float)); /* Un void* è assegnato a un float* */  
v[0] = 12.2; /* Aritmetica dei puntatori */
```

La funzione **calloc**

```
#include <stdlib.h>  
void *calloc(size_t nmemb, size_t size);
```

Simile alla **malloc**. Funzione gemella

Allocà memoria per un array di **nmemb** *elementi* ognuno di **size** byte e ne *ritorna il puntatore*.

La memoria è *inizializzata* a 0.

Osservazione: a differenza della **malloc**, la **calloc** riceve **size** e **nmemb** e fa la moltiplicazione internamente. Come mai? Boooh; comunque useremo più la **malloc**

La funzione **realloc**

Voglio *modificare* la dimensione della memoria dinamica appena creata con **malloc**.

Come famo?

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

C

Modifica la dimensione della zona di memoria puntata da **ptr** a **size** byte.

Il valore di ritorno è il puntatore *alla zone estesa*

- Se comporta un *restringimento* della zona di memoria, i dati in eccesso sono *persi*
- Se comporta un aumento, la zona aggiuntiva *NON* è inizializzata (come con malloc)

Nota: **ptr** deve essere stato ottenuto con **malloc** **calloc** o **realloc**. Altrimenti succedono cose gravi...

Osservazione: se possibile, la **realloc** estende la zona di memoria corrente, e il valore di ritorno è uguale a **ptr**

Se non è possibile, i dati vengono copiati in *una nuova regione*, il cui indirizzo viene ritornato

La funzione **free**

```
#include <stdlib.h>
void free(void *ptr);
```

C

Dealloca (o libera) la zona di memoria indicata da **ptr**.

- Ovviamente **ptr** deve essere stato ottenuto con **malloc** **calloc** o **realloc**
Se si tenta di liberare più volta una zona di memoria, il comportamento non è definito (ovvero è meglio evitare di far casini, probabilmente si ottengono dei *segmentation Fault*).

Esempio

Esercizio: si scriva un programma che memorizza un numero *N* di **float** letti da tastiera.

Il numero *N* è letto da tastiera all'inizio del programma.

Infine il programma ne stampa la media.

```

#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, i;
    float * v, s = 0;

    printf("Quanti numeri vuoi leggere? ");
    scanf("%d", &n);
    printf("Inserisci %d numeri:\n", n);

    v = malloc (n*sizeof(float)); /* Allocazione. Notare cast implicito da void* a
float* */
    for (i=0; i<n; i++)
        scanf("%f", &v[i]);

    for (i=0; i<n; i++) /* Calcola la somma */
        s += v[i];

    printf("Media: %f\n", s/n);
    free (v); /* Deallocazione */
    return 0;
}

```

Importanza della **free**

Tutte le zone di memoria vanno deallocate tramite la **free**

Se non viene fatto, la memoria è liberata al termine del processo

Importante!

- Non deallocare la memoria è sempre un errore!
- Nei programmi che devono essere eseguiti per lungo tempo, la memoria non deallocated causa il cosiddetto *Memory Leak*: a un certo punto, viene allocata tutta la memoria del sistema!
- Per questo in certi contesti *non* è consentito usare la memoria dinamica.

Errori comuni con la Memoria Dinamica

Errori comuni:

Valore di ritorno di **malloc** non assegnato a un puntatore

C

```
// Errato
float v = malloc(5*sizeof(float));
float v[10] = malloc(5*sizeof(float));
// Corretto
float * v = malloc(5*sizeof(float));
```

Creare un array la cui dimensione non è nota durante la compilazione

C

```
// Errato
float v[n];
// Corretto
float * v = malloc(n*sizeof(float));
```

Errori comuni:

Utilizzo errato dell'aritmetica dei puntatori

C

```
float * v = malloc(5*sizeof(float));
// Errato
v+2 = 43.5; // v+2 è un puntatore
&(v+2) = 43.5; // (v+2) è già un puntatore. Usare '!' non ha senso
// Corretto
*(v+2) = 43.5;
v[2] = 43.5;
```

Utilizzo errato nella **scanf**

C

```
// Errato
scanf("%f", v[2]);
scanf("%f", *(v+2));
// Corretto
scanf("%f", &v[2]);
scanf("%f", v+2);
```

Esercizi con la Memoria Dinamica

Esercizio: si scriva una funzione che ritorna un sequenza di N **float** equispaziati tra a e b

```
#include <stdlib.h>
#include <string.h>
float * seq(int N, float a, float b){
    int i;
    float * v;

    v = malloc(N*sizeof(float));
    for (i=0; i<N; i++)
        v[i] = a + (float)i/N*(b-a); /* Cast a float necessario per 'i' */

    return v;
}
```

Utilizzo:

```
int i ;
float * s = seq(10, 2, 5);
for (i=0; i<10; i++)
    printf("s[%d]==%f\n", i, s[i]);
free(s); /* Importante! */
```

Esercizio: si scriva una funzione che riceve come argomenti un intero N e un pattern p .

La funzione ritorna una stringa lunga N che contiene il pattern p ripetutamente.

Esempio: **repeat(5, "ah")** → **ahaha**

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * repeat(int N, char * pattern){
    int i, l;
    char * s;

    s = malloc((N+1)*sizeof(char));
    s[N] = '\0';
    l = strlen(pattern);

    for (i=0; i<N; i++)
        s[i] = pattern[i%l];

    return s;
}

```

Utilizzo: `printf("%s\n", repeat(15, "ciao! "));` stampa: `ciao! ciao! cia`

Cenni di funzionamento interno della Malloc

Le funzioni **malloc calloc realloc free** sono delle funzioni di libreria: mantengono lo heap ξ con delle *strutture dati*. Vediamo in dettaglio *come* fanno a mantenere questo heap

System Call sbrk

Esse usano la System Call **sbrk**.

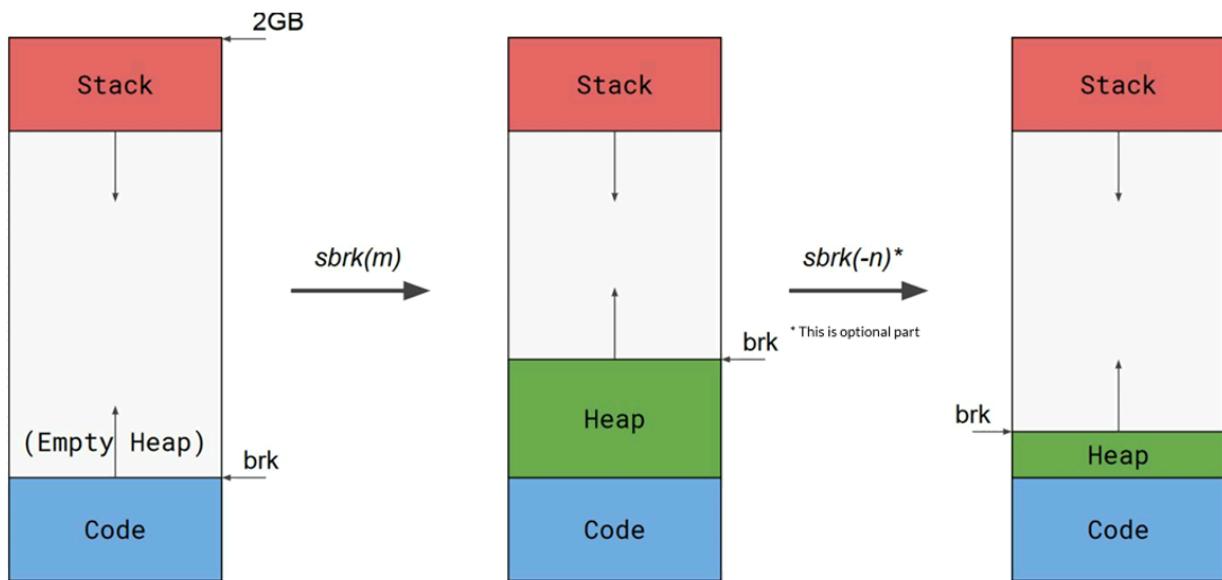
```
void *sbrk(intptr_t increment);
```

Incrementa di **increment** il *data segment*, inteso come unione di *segmento codice, dati e heap* ($\zeta \cup \delta \cup \xi$).

In pratica, informa il SO che l'*heap* si sta *espandendo*.

- Il SO, se necessario, *imposterà* la MMU per accogliere pagine aggiuntive

Chiamare la **sbrk** è di per sé sufficiente per poter usare indirizzi virtuali più alti



Tuttavia, per il programmatore sarebbe difficile gestire la memoria dinamica solo usando la **sbrk**

- Dovrebbe tenere traccia di *ogni allocazione e di ogni deallocazione*
- Dovrebbe avere una tecnica per riusare i *buchi* lasciati liberi da una deallocazione
 - Nel momento in cui si fa una nuova allocazione, così da evitare delle eventuali *frammentazioni*.
- Invocare la **sbrk** a ogni allocazione è *inefficiente*
 - Una System Call è lenta (implica un *Context Switch*)
Che incubo! Meno male che non bisogna reinventarsi la ruota...

Le funzioni di libreria **malloc**, etc., gestiscono tutto ciò per il programmatore

- Utilizzando opportune *strutture dati*

Storia della Malloc

La moderna funzione **malloc** deriva dalla proposta di Doug Lea, professore della *State University of New York at Oswego*

Internamente usa una *linked list* per tenere traccia delle zone occupate.

Nota Tecnica: *heap* ha due significati!

1. Una struttura dati che implementa una coda a priorità tramite un albero (*informatica teorica*)
 - Permette di trovare facilmente il massimo di un insieme di numeri
 - Veloce da aggiornare

2. La zona della memoria virtuale dove viene allocata la memoria dinamica (*informatica ingegneristica*) (ξ)

Queste due definizioni non c'entrano per niente a vicenda

Funzionamento della Malloc

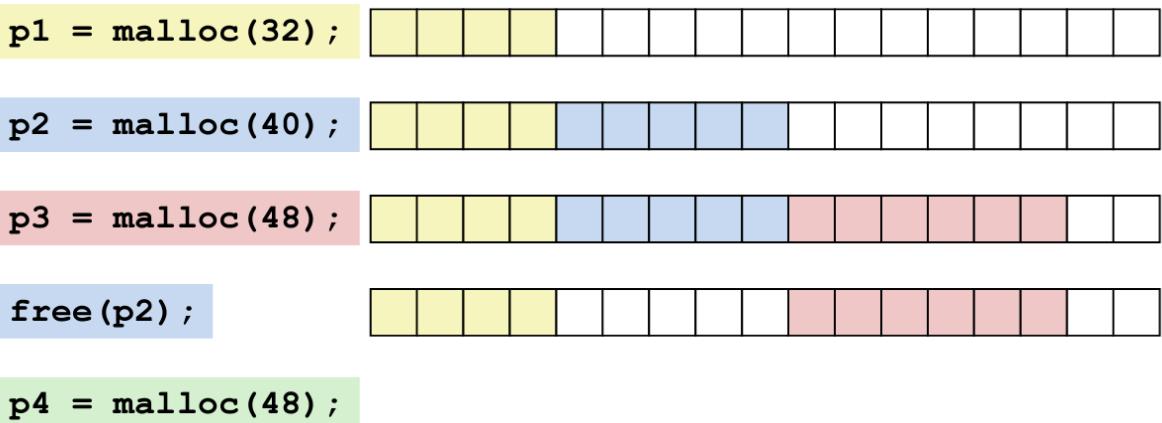
La **malloc** gestisce *blocchi di grandezza variabile*

- Non c'è nessuna discretizzazione o utilizzo di blocchi di grandezza fissa
- Porta ad avere *frammentazione esterna*: memoria sprecata perchè è una zona contigua troppo piccola per essere allocata

Esempio. (*Frammentazione esterna*)

E' possibile che si giunga a situazione come questa:

- Esempio con blocchi di dimensione fissa di 8B



malloc(48) potrebbe essere evasa, se la memoria libera fosse contigua. Cosa si fa per risolvere questo problema? Niente, la si accetta com'è.

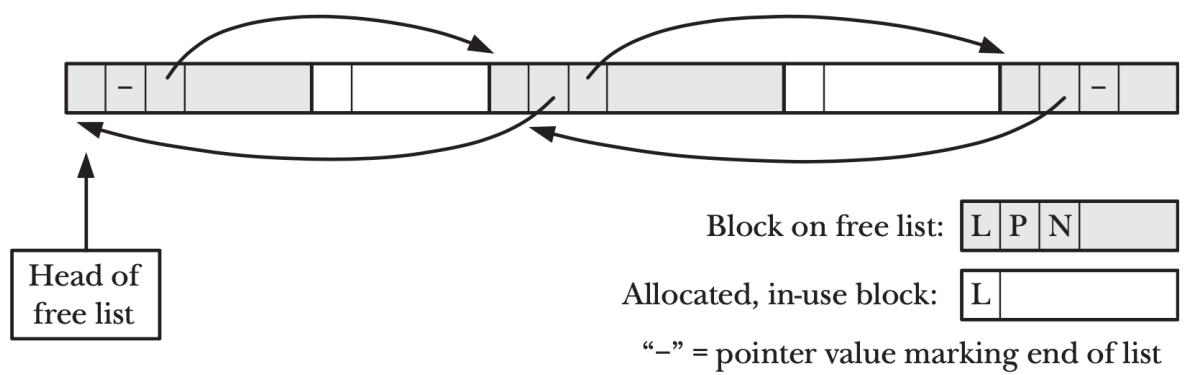
Struttura Dati: Arena

La **malloc** gestisce indipendentemente *più di una zona di memoria*, dette *Arenas*.

- Le strutture dati sono replicate
- Rende *più efficiente l'utilizzo in contesti multithread*
 - Le funzioni **malloc**, etc., sono Thread Safe
- Evita che diversi thread vengano rallentati aspettando il release di un *lock*
 - I lock sono necessari, ma l'utilizzo di più di un struttura ne diminuisce l'impatto
- Le *Arena* sono praticamente un espediente per la *malloc efficiente e sicura*.

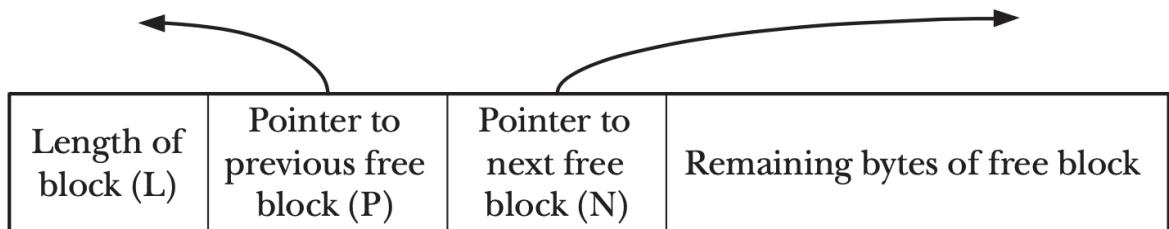
Struttura Dati: Linked List e Doubly Linked List

1. Una zona di memoria gestita dalla **malloc** è amministrata tramite una *linked list*
2. I segmenti ancora liberi sono una *Doubly linked list*
 - Le zone allocate sono momentaneamente rimosse dalla lista

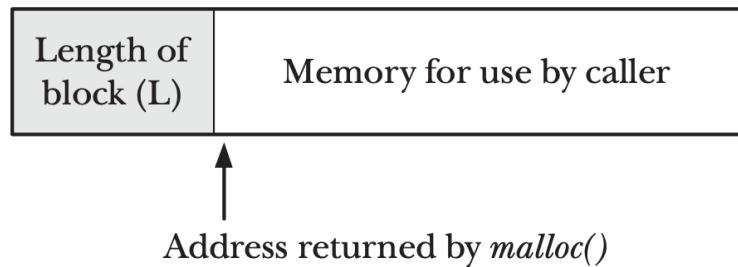


3. Ogni **zona libera o allocata** ha una **struct** nei **primi byte** che fornisce **informazioni su di essa e sui blocchi adiacenti** (struttura interna delle zone di memoria)

- **Zona Libera**



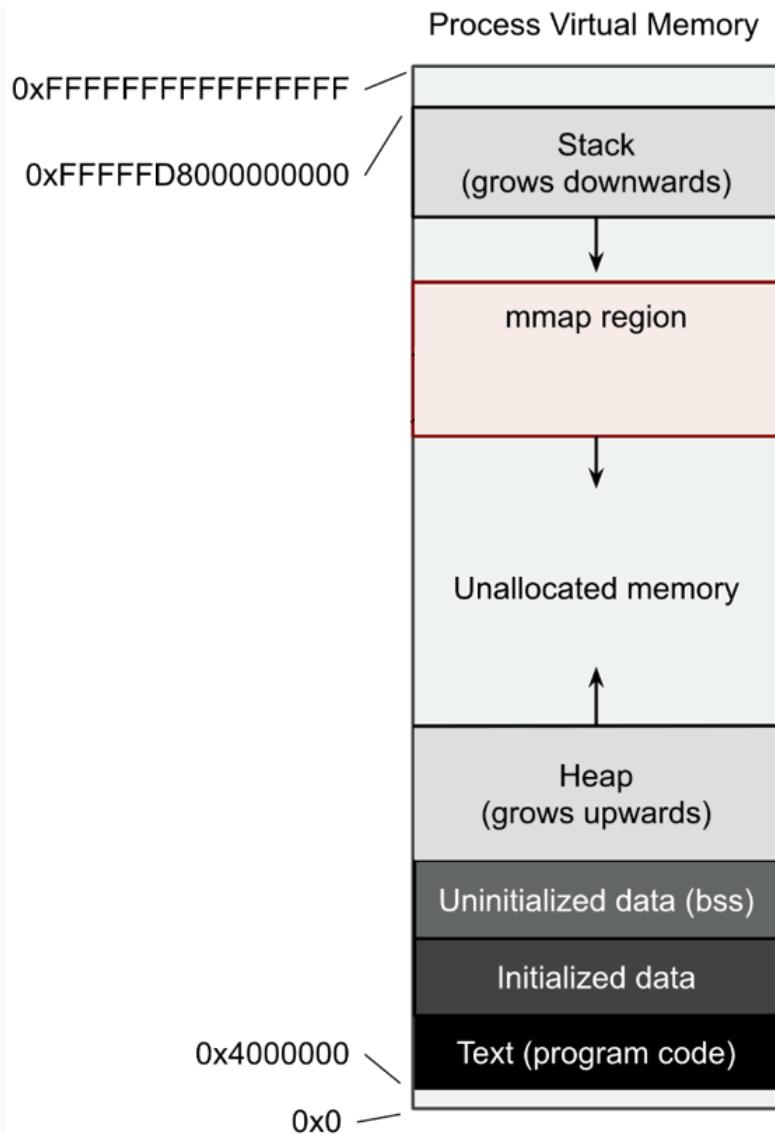
- **Zona Allocata**



Allocazione di Grandi Regioni di Memoria

In caso la **malloc** debba allocare **grandi regioni di memoria** (tipicamente $> 128 kB$) usa la System Call **mmap** per allocare una zona di memoria.

- **malloc** chiede una regione di tipo **MAP_ANONYMOUS|MAP_PRIVATE**. Non deve essere condivisa con nessuno (no flag **MAP_SHARED**!).
- Il SO crea **una o più pagine per il processo**
- Le colloca in una posizione **a sua scelta** nello **spazio degli indirizzi virtuali**
- Non sempre ottengo un **puntatore** in ξ



Domande

Si consideri il seguente codice C:

```

int c = 40;
int main(){
    int i;
    static int j;
    ...
}

```

Quali variabili risiedono nello stack?

- Tutte
- i e j
- i

Risposta: *i*

Il seguente codice è corretto in C?

C

```
#define size 1024  
int i [size];
```

- Si
- No

Risposta: *Si*

Si completa il seguente codice C

C

```
double * a, int i;  
scanf("%d", &i);  
a = ...
```

- float[i]
- malloc(i);
- malloc(i * sizeof(double));
- malloc(sizeof(double));

Risposta: *malloc(i*sizeof(double));*

La **malloc**:

- è una System Call
- è utilizzata dalla funzione sbrk
- utilizza la System Call sbrk

Risposta: *utilizza la System Call sbrk*

u7-s1-thread

Sistemi Operativi

Unità 7: I Thread

I Thread in Linux

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Concetto di Thread
 2. Thread in Linux
 3. Funzioni per i Pthread
 4. Esempi
 5. Thread in Bash
-

Concetto Teorico di Thread

Definizione di Thread

In Linux (e in quasi tutti i SO), un **processo** può avere molteplici flussi di esecuzione, detti **Thread**

- I thread possono essere visti come un *insieme di processi che condividono la memoria*
- Ma eseguono lo stesso programma

Nota: anche Windows permette di creare thread con la System Call **CreateThread()**

Ogni Thread esegue *lo stesso programma e condivide gli stessi dati*

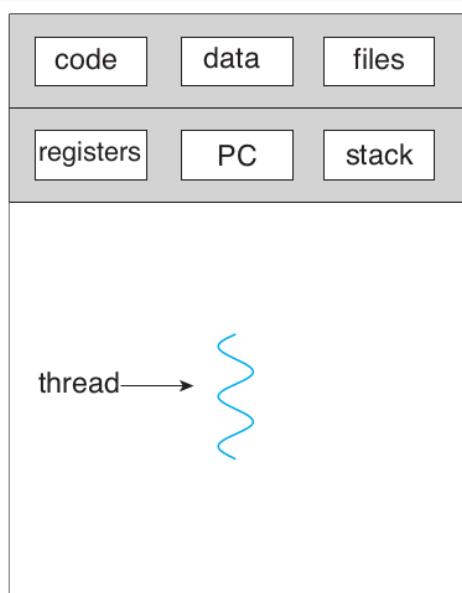
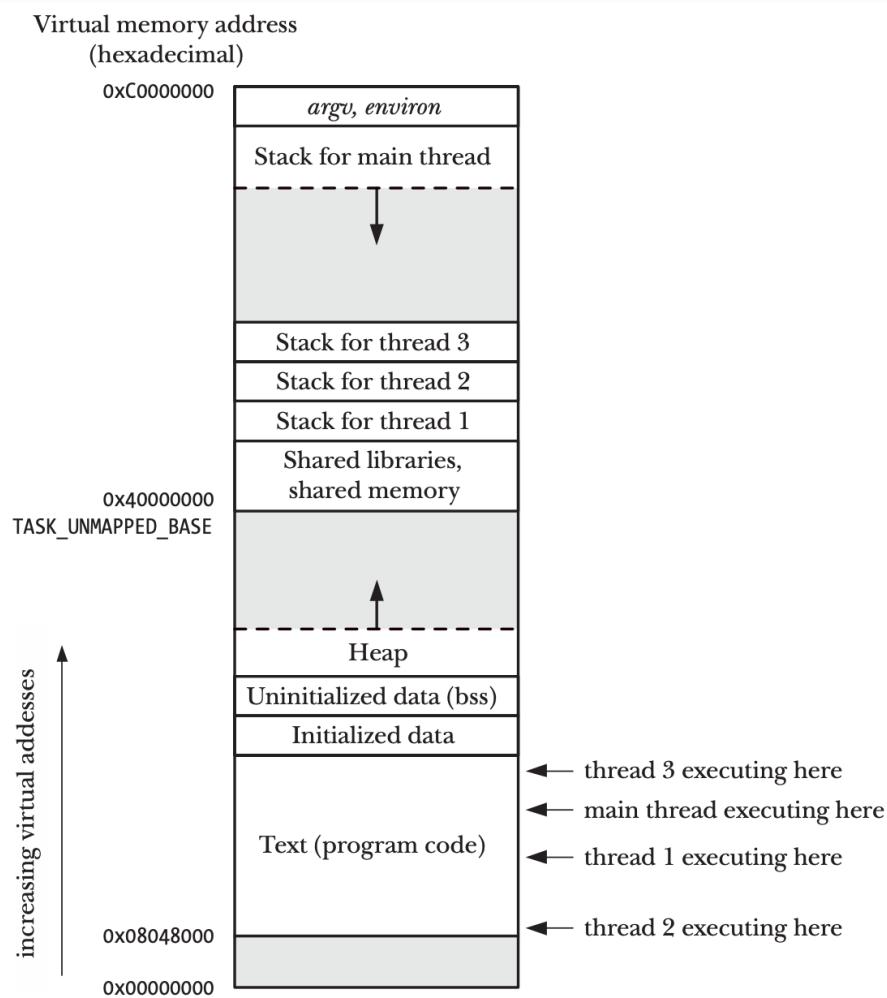
- I segmenti **data**, **heap** e **code** (ζ, δ, ξ) sono condivisi

Un Thread è un *flusso del codice in esecuzione*

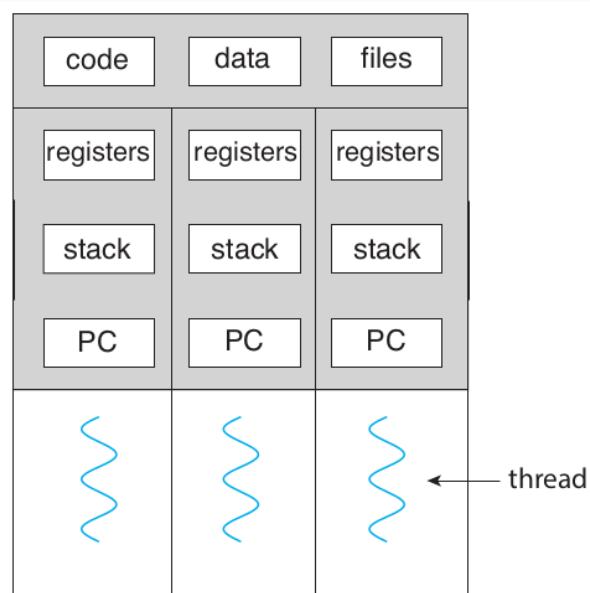
- Ha il *suo stack*
- Contiene lo *stato delle funzioni in esecuzione*

Ogni *thread ha uno stack*

- E chiaramente opera su *Registri* e ha un *Program Counter*



single-threaded process



multithreaded process

Comunicazione tra Thread

I Thread possono comunicare tra loro più facilmente che i processi, usando:

- *Variabili globali* in δ (che è *nativamente* condivisa)
- Costrutti di sincronizzazione
 - *Mutex*
 - *Condition Variable* (vedremo solo sommariamente)
 - *Semafori*

Oggiorno è più spesso usata un'architettura *multi-thread* che *multi-process*. In realtà dipende sono scelte: ad esempio *Chrome* ha un paradigma *multi-process*. Ognuno ha i suoi vantaggi e svantaggi;

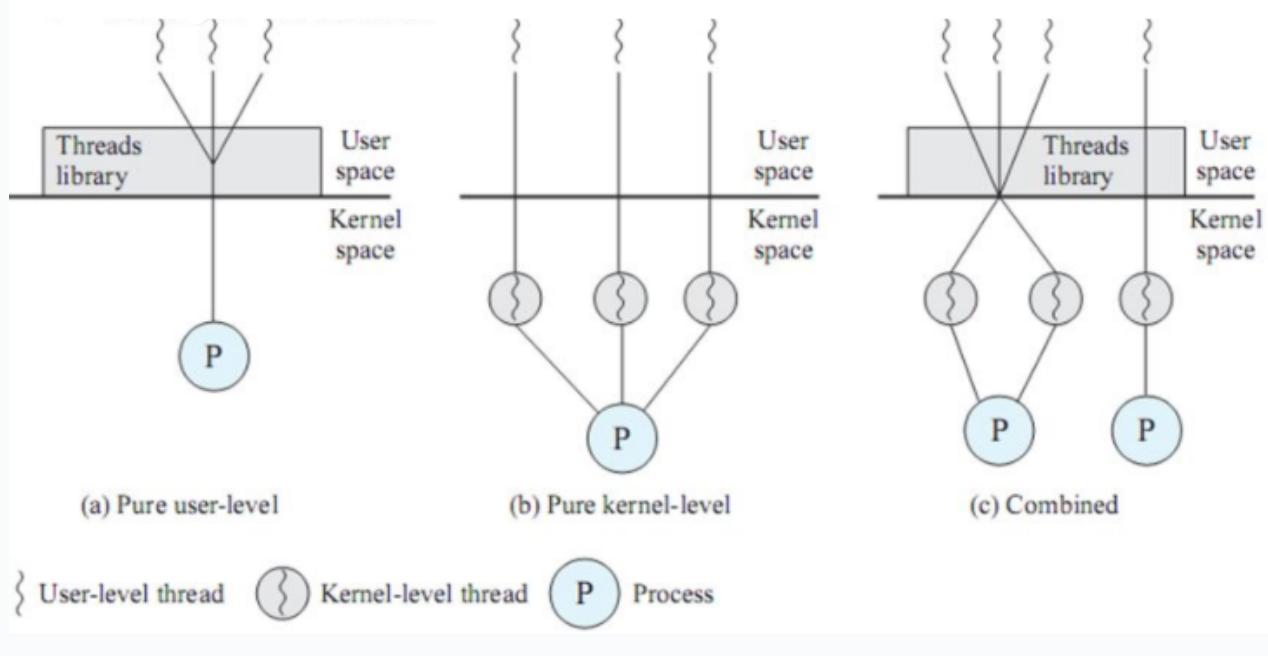
- *Multi-thread*: un po' più "*semplice*" da implementare e sincronizzare. Però se il processo principale cade, gli altri thread cadono assieme
- *Multi-process*: complicato da sincronizzare, tuttavia non ha lo svantaggio di programmi multi-thread

User e Kernel Thread

Esistono due modi per implementare i thread.

- **Kernel Thread**: il *kernel* permette di creare thread
 - Sono di fatto dei processi *light*
 - Vedremo questi, principalmente (in Linux)
- **User Thread**: creati dal *programmatore* o da una *libreria*
 - Il processo (in qualche modo) gestisce e orchestra più flussi di esecuzione
 - Il kernel ne è allo scuro
 - Molto complicato! Inoltre ho le limitazioni dell'esecuzione in *User Mode*

Poi si può fare anche robe strane, come *combinarle*, ma ne staremo allo scuro.



Thread in Linux

LinuxThreads

Inizialmente i Pthread erano implementati dalla libreria *LinuxThreads*

- I thread erano dei processi che condividevano la memoria, i file aperti, ecc.
- Ognuno aveva *diverso PID*
- *Implementazione problematica*: si mischiava concetto di thread e processo. In quell'epoca non c'era ancora il supporto *nativo* di Thread, infatti non avevo altro che delle *fork sofisticate*.

Ora (da 2002), Linux/POSIX usa la libreria *Native POSIX Threads Library (NPLT)*

- Coopera col kernel, che offre supporto ai thread
- Migliori prestazioni

Posix Thread

Nei sistemi POSIX (e Linux), le *funzioni di libreria* per gestire i thread sono chiamate *Pthread*

I thread permettono a un processo:

- Di svolgere più task in maniera concorrente
 - Mentre un thread attende l'I/O o la rete, un altro thread può svolgere un altro compito
- Di sfruttare un sistema *multi-core*
 - Più flussi davvero in esecuzione parallela

I thread in Linux sono *Kernel Thread*

Qui i Posix-Threads condividono:

- La *memoria globale*
- PID e PPID
- File aperti
- Privilegi
- Working directory

Ogni thread ha invece le seguenti caratteristiche distinte:

- Un *Thread ID*
 - Il Kernel mantiene la lista dei thread e li *schedula*, facendoli eseguire sulla CPU. Identificativo univoco per il sistema.
- Il suo *stack*
 - Per poter eseguire le funzioni

- Un thread *mal configurato* può comunque accedere/corrompere lo *stack* di un altro thread (comunque una cattiva idea!)
- Metadati: scheduling, etc...

Compilazione con Pthreads

Il codice deve includere la direttiva:

```
#include <pthread.h>
```

Per compilare, bisogna includere la libreria **pthread**

```
gcc MyProgram.c -o MyProgram -lpthread
```

Funzioni per i Pthread

Adesso vediamo come *lavorare* con i *Pthread*

Creazione di un thread

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start)(void *), void *arg);
```

Crea un nuovo thread che esegue la funzione **start** chiamata con l'argomento **arg**

- Come se si invocasse **start(arg)** su un *flusso di esecuzione separato*

Nota: Ogni programma, quando nasce, ha un solo thread, detto *main thread*

PARAMETRI E VALORI DI RITORNO.

- L'argomento **arg** è un **void***, ovvero un puntatore a un tipo di dato a piacere. Questo per avere la *massima flessibilità*.
- Similmente, il valore di ritorno di **start** è un **void***.
- Non ci interessa l'argomento **attr** che specifica attributi particolari

- L'argomento **thread** è un puntatore a una variabile **pthread_t** che andrà a contenere il Thread ID, per poterlo usare in successive funzioni di libreria (poi per lavorarci sopra)
- In caso di successo, ritorna 0, altrimenti un codice di errore

Nota Implementativa

La **pthread_create()** è una *funzione di libreria*

Essa usa la System Call **int clone(...)**

- La **clone()** è simile alla **fork()**
- Crea un *processo figlio*
- Più *flessibile* e precisa della **fork()**
 - Permette di *controllare cosa condividono* padre e figlio
- La **pthread_create()** crea un nuovo processo che *condivide la memoria* col padre
 - Che è la definizione di *Thread*

Terminazione di un thread

Un thread termina se:

- La funzione di lancio **start** esegue una **return** (quindi è *finita*)
- Il thread esegue una **pthread_exit()**
- Il thread viene cancellato tramite una **pthread_cancel(pthread_t thread);**, invocata da un altro thread
- Il processo termina se un qualsiasi thread invoca una **exit()** o il thread principale termina il **main**

```
C
include <pthread.h>
void pthread_exit(void *retval);
```

- Termina il *thread corrente* col valore **retval**.
- Equivalente a effettuare una **return** nella funzione di avvio del thread.

Thread ID

```
C
include <pthread.h>
pthread_t pthread_self(void);
```

Permette a un thread di ottenere il *proprio Thread ID*.

Il Thread ID va trattato come un *handle opaco*

- Su Linux é un **long int**
- Ma potrebbe essere un puntatore a una struttura dati arbitraria
- Non é affidabile decifrarne il valore

Join di un thread

```
include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

Attende che il thread **thread** **termini**.

- Se é già terminato, ritorna istantaneamente

Immagazzina il valore di ritorno all'indirizzo **retval**

- **retval** é specificato dal *thread morente* tramite **pthread_exit()** o **return**
- **retval** é un **void****, ovvero un puntatore a puntore a **void**
 - E' l'indirizzo di una variabile che contiene un puntatore. Infatti devo salvare su un *puntatore a void*, quindi devo avere l'*indirizzo del puntatore a void*, sarebbe il *puntatore al puntatore a void*.

I thread devono essere tutti attesi tramite una **pthread_join()**, altrimenti diventano zombie

- *Come avviene per i processi*

Usando la funzione **int pthread_detach(pthread_t thread)** è possibile indicare che il thread **thread** non necessita di una **join**

- Il valore di ritorno viene *scartato*
- Il sistema rimuove ogni informazione sul thread quando esso termina

Note:

I thread sono pari tra loro

- Qualunque thread può fare una **pthread_join** su un altro; anche se comunque non è (di solito) una buona idea fare *join* tra fratelli
Non esiste un modo per aspettare la terminazione di un *qualsiasi* thread
- Coi processi si può invece usare la **wait**.
Una **pthread_join** é sempre bloccante
- Diverso da **waitpid** con flag **WNOHANG**

Esempio di Creazione di un Thread

Creazione di un Thread

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void * threadFunc(void *arg){
    printf("From Thread: %s", (char *) arg);
    int * ret = malloc(sizeof(int));
    *ret = strlen(arg);
    return ret; // Valore di ritorno del thread
    // Equivale a pthread_exit(ret);
}

int main(int argc, char *argv[]){
    pthread_t t1;
    void *res; // Per valore di ritorno
    int s,

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n"); // Creazione
    if (s != 0){
        printf("Cannot create thread");
        exit(1);
    }

    printf("Message from main()\n");
    s = pthread_join(t1, &res); // Join. Richiede un void **, ovvero &res
    if (s != 0){
        printf("Cannot join thread");
        exit(1);
    }
    printf("Thread returned %d\n", *((int *)res)); // Utilizzo del valore di ritorno
    free(res); // Needed as that zone was allocated with malloc
    exit(0);
}

```

Esercizio

Esercizio. Si crei un programma che avvia 10 thread che attendono un tempo casuale tra 0 e 5 secondo prima di terminare

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MAXSLEEP 5
#define THREADNB 10

static void * sleepFunc(void *arg){
    char thread_number = *((char*)arg);
    int n=rand() % MAXSLEEP;
    sleep(n);
    printf("Thread %c terminated after %d seconds\n", thread_number, n);
    return NULL;
}

int main(int argc, char *argv[]){
    int i;
    pthread_t t [THREADNB];
    char names [THREADNB];

    for (i=0;i<THREADNB;i++){
        names[i] = 'A' + i;
        pthread_create(&t[i], NULL, sleepFunc, &names[i]);
    }
    for (i=0;i<THREADNB;i++)
        pthread_join(t[i], NULL);
    return 0;
}

```

Thread in Bash

Normalmente, i comandi **ps** e **top** mostrano solo i processi

Per visualizzare i thread:

- **ps -T opzioni**. Esempio: **ps -T ax**
- **top -H**

Ogni thread presente nel **/proc** file system

- Come se fosse un processo: **/proc/[tid]**
 - Per ottenere la lista di thread di un processo: **/proc/[pid]/task**
 - Contiene la lista dei thread di un processo
-

Domande

Due Thread dello stesso processo condividono le variabili globali?

- Si
- No

RISPOSTA: Si

La funzione **pthread_join** attende la terminazione:

- Di un qualsiasi thread del sistema
- Di un qualsiasi thread del processo corrente
- Di un thread specifico

RISPOSTA: Di un thread specifico

Quando un thread invoca la funzione **pthread_exit**:

- Il thread corrente termina
- Il processo corrente termina
- Il thread specificato come argomento della funzione termina

RISPOSTA: Il thread corrente termina

Si consideri il seguente codice:

```
C

void * func(void *arg){
    sleep(5);
    exit(0);
}

int main(){
    ...
    pthread_create(&t, NULL, func, NULL);
    sleep (10)
    pthread_join(t, NULL);
    exit(0);
}
```

Dopo quanti secondi termina il processo?

- 5
- 10
- 15

RISPOSTA: 10

Sincronizzazione

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Perché è necessaria
 2. I mutex
 3. I semafori
-

Motivazioni per la Sincronizzazione

Definizioni di Concorrenza e Parallelismo

Diamo delle *definizioni preliminari*.

Concorrenza: un programma con più flussi di esecuzione

Parallelismo: un programma che esegue su più calcoli contemporaneamente

Notiamo subito che non ci dev'essere nessun legame tra di loro, soprattutto del tipo



1. Un programma può essere *concorrente senza essere parallelo*
 - Ha tanti thread che eseguono su un sistema con una sola CPU
2. Un programma può essere *parallelo senza essere concorrente*
 - Le moderne CPU hanno istruzioni che manipolano più dati
 - Paradigma *Single Instruction Multiple Data (SIMD)*
 - Una *singola istruzione per sommare due vettori*, componente per componente
 - La CPU ha una *ALU* che permette di effettuare più operazioni in parallelo
 - Usando un *singolo thread/processo*

Obiettivi della Programmazione Parallelia

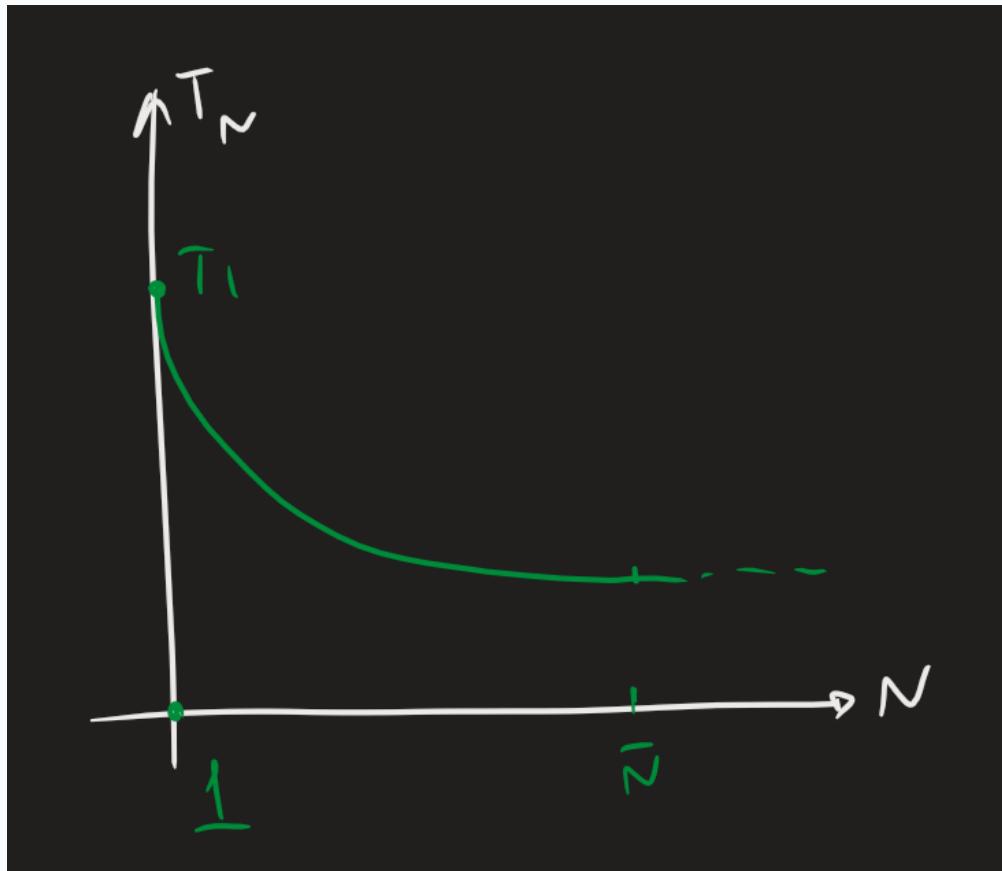
Teoricamente, parallelizzando e usando N core anziché 1, dovremmo avere:

$$T_N = \frac{T_1}{N}$$

Ovvero *minimizziamo* il tempo T_N con un comportamento del tipo $T_N \rightarrow 0$.

In realtà, vale solo per un *numero ridotto di processori e core*.

- Solitamente, con un numero ridotto di core, si ha davvero un *incremento*
- Poi c'è un *appiattimento*. Questo è dovuto alla *legge di Ahmdal*
Quindi si ha un andamento del tipo



LEGGE. (*Di Ahmdal, o del buonsenso*)

"Il miglioramento delle prestazioni di un sistema che si può ottenere ottimizzando una certa parte del sistema è limitato dalla frazione di tempo in cui tale parte è effettivamente utilizzata"

Ovvero: la parte di codice non parallelizzabile, penalizza tutto il programma. Abbiamo dei cosiddetti "*bottleneck*"

Problema: non tutti gli algoritmi sono parallelizzabili!

Definizione di Parallelizzabilità (e non)

Definizione: *Esecuzione di un algoritmo tramite più flussi simultanei.* Non tutti gli algoritmi sono parallelizzabili

Parallelizzabile: (esempi)

- Calcolare la somma di un vettore (array); posso spezzare l'array in *due*, farci le somme individuali poi sommare le ridotte.

Non Parallelizzabile: (esempi)

- Calcolare le cifre di $\sqrt{2}$; devo in un modo o l'altro usare i *metodi dell'analisi numerica*, che sono *iterativi* (o addirittura *ricorsive...*)

Attualità della Programmazione Parallelia

Ancora oggi questo tema è *attuale*.

C'è *molta ricerca* per tentare di *parallelizzare* gli algoritmi

- Trovando *espedienti matematici*
- Oggi abbiamo sistemi con *tanti core*, e vogliamo sfruttarli al massimo
- Calcolando *soluzioni approssimate* (tipo per $\sqrt{2}$ posso usare gli *sviluppi di Taylor*)

Problema sentito nel *machine learning*

- Addestrare una rete neurale usando molti core (e nodi)
 - Problema risolto
- Algoritmi di *clustering* (classificazione) paralleli
 - Problema in parte aperto

I mutex

Vediamo un *primo costrutto* di sincronizzazione: i *mutex*.

Problema delle Sezioni Critiche

I thread condividono la memoria

- Possono condividere informazioni usando *Variabili Condivise*

E' necessario sincronizzare l'*accesso alle variabili condivise*

- Due thread non devono scrivervi contemporaneamente
- Un thread non deve leggere una variabile condivisa mentre un'altro la scrive

- Altrimenti avrei *casini!*
- Tema accennato con i *segnali*, mediante il *problema dell'incremento perso* ([Segnali > ^45417b](#)).

PROBLEMA.

Immaginiamo due thread che eseguono il seguente codice:

```
C

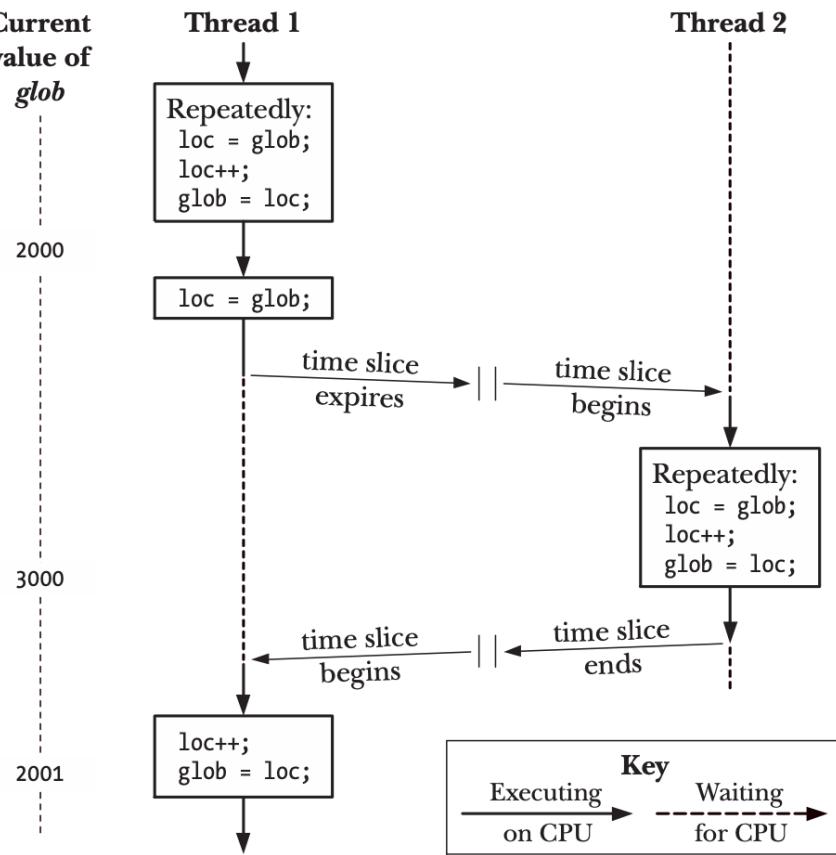
static int glob = 0;
static void * threadFunc(void *arg){
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }
    return NULL;
}
```

Il seguente codice produce risultati non predicibili.

Esempio:

- Thread 1 è interrotto durante l'incremento
- Thread 2 effettua l'incremento
- Thread 1 completa l'incremento

L'incremento effettuato dal Thread 2 *è perso!* (o potenzialmente); ho uno *stato inconsistente del programma*. Avrò l'incremento perso circa al ~ 50%.



Osservazioni

Sostituire:

```
loc = glob;
loc++;
glob = loc;
```

con **glob++;** non risolve il problema.

In molti processori (e.g., ARM) non hanno una istruzione di incremento

- Il compilatore traduce **glob++;** in istruzioni Assembly equivalenti alle 3 righe di codice di cui sopra

Definizione di Sezione Critica

Definizione. (*Sezione critica*)

Una *Sezione Critica* è una sezione di codice la cui esecuzione deve essere *atomica* (nel senso autonoma)

- Non può essere *interrotta* da un altro thread
- Nessun altro thread può eseguire quel codice *contemporaneamente*

Una sezione critica accede a *risorse condivise*

- Solo un thread per volta vi può fare accesso

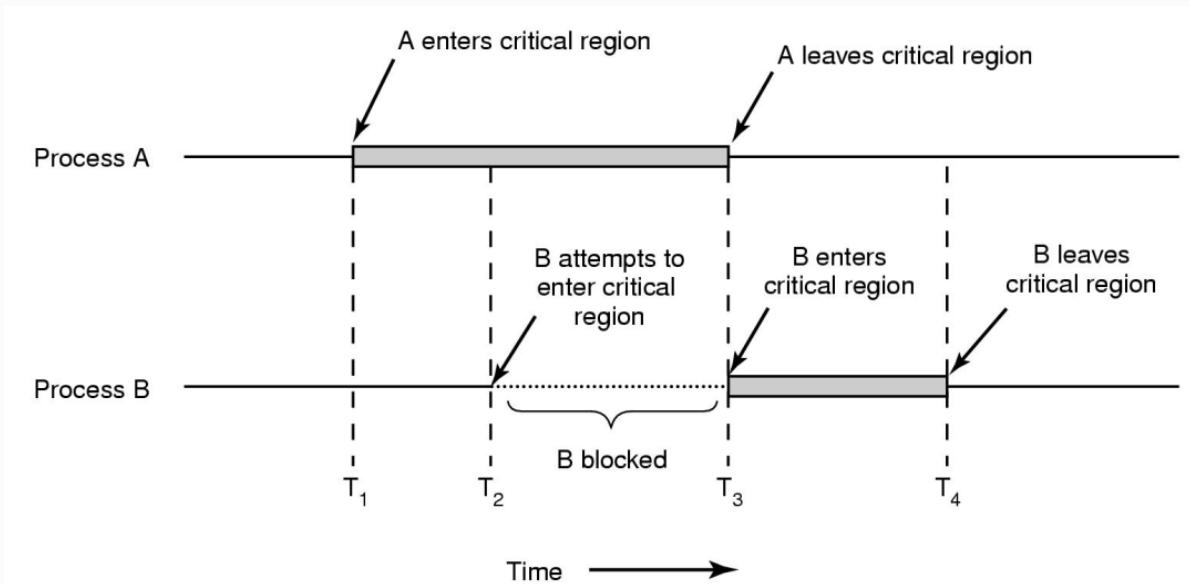
Le sezioni critiche sono anche dette *Regioni Critiche* (sinonimo).

Funzionamento di sezione critica

Prima di vedere il costrutto, vediamo come funzionerebbe (da un punto di vista teorico) una *sezione critica*

L'accesso a una sezione critica avviene in *Mutua Esclusione*

- Un thread si *prenota* per l'accesso
 - Se la sezione critica non è utilizzata, il thread vi accede (*lock*)
 - Altrimenti attende finché non si libera
- Al termine della sezione critica, il thread *rilascia* la sezione (*post*)



Adesso siamo pronti per vedere il *mutex*.

Definizione di Mutex

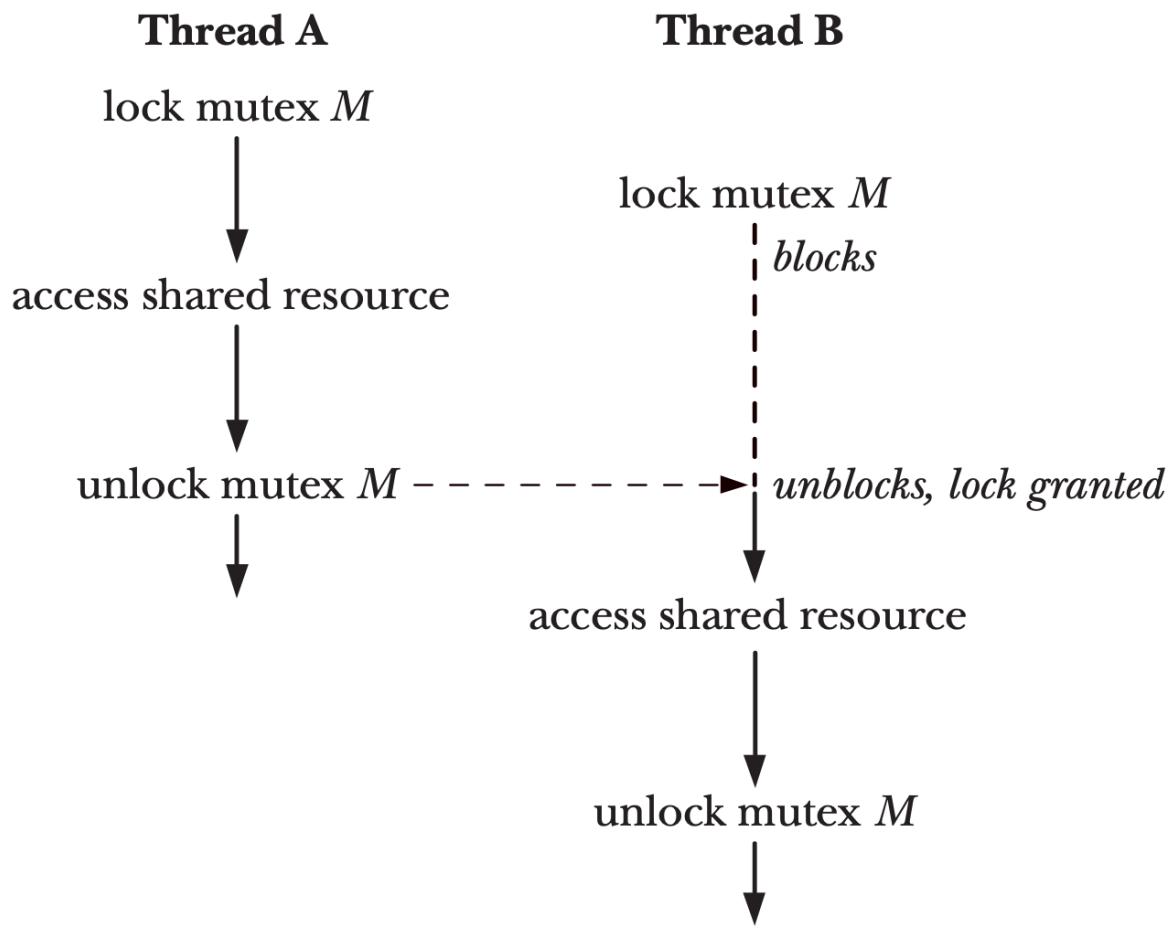
Un *Mutex* è un costrutto di sincronizzazione che gestisce l'accesso a una sezione critica

Un mutex ha due stati

- **Locked**: la sezione è occupata
- **Free**: la sezione è libera

Un thread può fare due azioni su un mutex:

- **Lock**: prenota l'accesso per l'occupazione della sezione critica
- **Release/Unlock**: rilascia la sezione critica



Implementazione in Pthread

I mutex sono variabili di tipo **pthread_mutex_t**

- Sono solitamente *variabili globali*
- Inizializzate dal **main**
- Usate da *qualsiasi thread*

Necessario includere:

```
#include <pthread.h>
```

Si utilizzano con le funzioni di libreria **pthread_mutex_***

Inizializzazione dei Mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t * mutex , const pthread_mutexattr_t *
attr );
```

Inizializza il mutex **mutex**, che viene *passato per riferimento* (tipo **pthread_mutex_t ***)

L'argomento **attr** specifica gli attributi, che non vedremo

- Può essere **NULL**

Valore di ritorno, come in tutte le funzioni di Pthread (omesso nelle successive slide):

- 0 in caso di successo
- Il codice di errore altrimenti

Lock di Mutex

```
C  
#include <pthread.h>  
int pthread_mutex_lock(pthread_mutex_t * mutex );
```

Acquisisce il *lock* del mutex

- Blocca il chiamante finchè il lock non diventa libero

Release di Mutex

```
C  
#include <pthread.h>  
int pthread_mutex_unlock(pthread_mutex_t * mutex );
```

Rilascia il lock

Nota: **mutex** è **sempre** passato per riferimento!

Altre Operazioni

```
C  
#include <pthread.h>  
int pthread_mutex_trylock ( pthread_mutex_t *mutex);
```

Acquisisce il lock

- Se il lock è già preso da qualcun'altro fallisce con errore (valore di ritorno) **EBUSY**

Distruzione di Mutex

C

```
#include <pthread.h>
int pthread_mutex_destroy ( pthread_mutex_t *mutex );
```

Rilascia la *memoria occupata* dal lock mutex

Tale lock non sarà più utilizzabile

Esempio

Realizzazione del precedente programma (incremento di una variabile da parte di due thread in parallelo) usando in mutex

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

static int glob = 0;
static pthread_mutex_t mtx;

static void * threadFunc(void *arg){
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mtx); /* LOCK */
        loc = glob; /* T */
        loc++; /* | Critical Section */
        glob = loc; /* L */
        pthread_mutex_unlock(&mtx); /* RELEASE */
    }
    return NULL;
}

int main(int argc, char *argv[]){
    pthread_t t1, t2;
    int loops = 10000000;

    pthread_mutex_init(&mtx, NULL);
    pthread_create(&t1, NULL, threadFunc, &loops);
    pthread_create(&t2, NULL, threadFunc, &loops);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&mtx);
    printf("glob = %d\n", glob);
    exit(0);
}
```

Il programma senza l'uso di mutex:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int glob = 0;

static void * threadFunc(void *arg){
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        loc = glob;          /* T */
        loc++;              /* | Critical Section */
        glob = loc;          /* L */
    }
    return NULL;
}

int main(int argc, char *argv[]){
    pthread_t t1, t2;
    int loops = 10000000;
    pthread_create(&t1, NULL, threadFunc, &loops);
    pthread_create(&t2, NULL, threadFunc, &loops);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("glob = %d\n", glob);
    exit(0);
}

```

La somma non è correttamente 20000000, ma un numero inferiore (e.g., 10493368)

Deadlock

Un **Deadlock** o stallo è una situazione in cui due o più thread risultano bloccati

- *Ognuno attende una condizione che non potrà mai verificarsi*
- Il programma cessa di eseguire

Quando si usano due o più mutex possono capitare situazioni di questo tipo

- Necessario che il programmatore le preveda e le eviti

Esempio Analogico: Vado in segreteria segreteria per chiedere qualcosa relativo al bando ERASMUS+; la segreteria mi manda all'ufficio internazionale per le informazioni. L'ufficio internazionale mi manda alla segreteria studenti (oppure Banana Joe).

Esempio:

Thread A:

```
C  
pthread_mutex_lock(mutex1); // ←- LOCK 1  
pthread_mutex_lock(mutex2); // ←- LOCK 2  
... Sezione Critica ...  
pthread_mutex_unlock(mutex2);  
pthread_mutex_unlock(mutex1);
```

Thread B:

```
C  
pthread_mutex_lock(mutex2); // ←- LOCK 2  
pthread_mutex_lock(mutex1); // ←- LOCK 1  
... Sezione Critica ...  
pthread_mutex_unlock(mutex1);  
pthread_mutex_unlock(mutex2);
```

Come evitare i deadlock:

- Usare altri *tipi di sincronizzazione* quando possibile:
 - Pipe, FIFO
- Usare un *basso numero di mutex*
- *Modellare l'uso di tanti mutex* con espedienti matematici
 - Tecniche basate sui grafi
 - Non vediamo in questo corso
- Usare il buonsenso!

I Semafori

Definizione di Semaforo

Definizione (*Semaforo*)

Un *Semaforo* è un numero **Intero Positivo** condiviso da più thread

- Inizializzato a un certo valore in fase di creazione

Thread concorrenti (in realtà anche processi) possono fare due operazioni:

- *Incremento di 1*
- *Decremento di 1*

Il semaforo non può *mai* assumere *valori negativi*.

Se il decremento comporta che il semaforo diventi negativo, allora

- Il thread si *blocca*, *attendendo* che un altro thread faccia un incremento

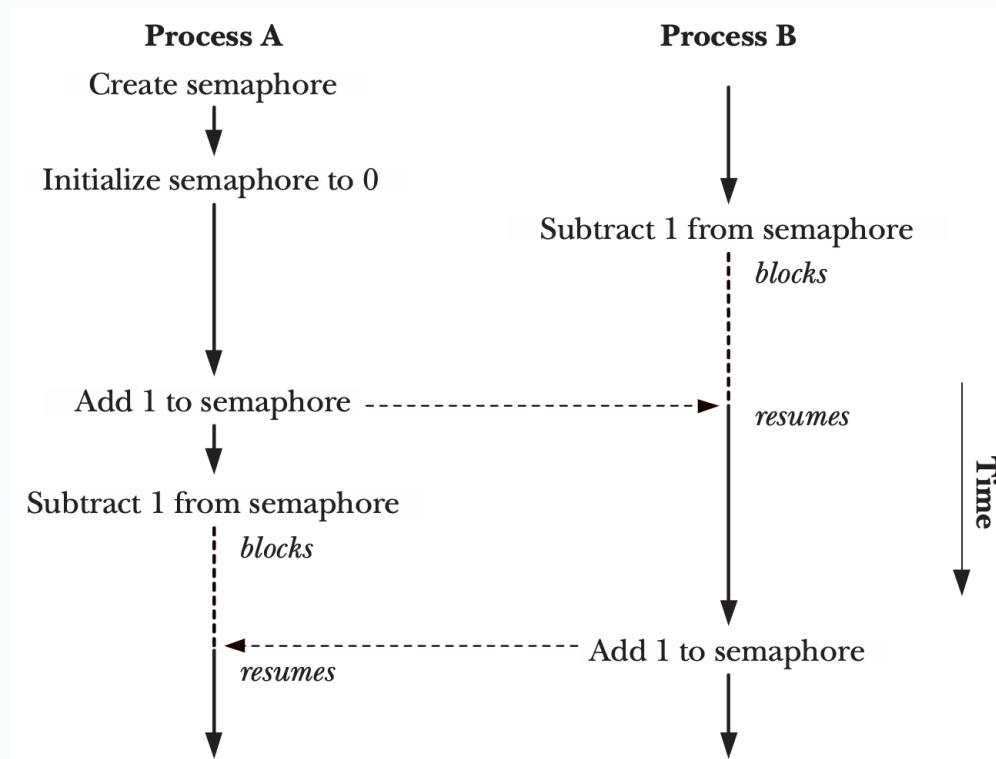
Un *semaforo* è come un *secchio* con dei *gettoni*: o *metto* dei gettoni, o provo a *toglierli*.

Se provo a togliere un secchio senza gettoni, aspetto che il prossimo ne prenda uno.

Esempio

Supponiamo di avere *due processi* e *un semaforo*

1. Il *semaforo* è inizializzato a 0
2. *B* decremente
 - Il semaforo non può assumere valori negativi
 - *B* entra in attesa
3. *A* incrementa
 - *B* si sblocca
 - Il semaforo ha valore 0
4. *A* decremente
 - *A* si blocca
5. *B* incrementa
 - *A* si sblocca
6. Il semaforo ha valore 0



Storia dei Semafori

Sono un costrutto di sincronizzazione semplice, potente e flessibile

- Inventato da *Dijkstra* nel 1965
- Usato per svariati scopi in tutti i linguaggi di programmazione e sistemi operativi

In Linux, due implementazioni

- *System V semaphores*: più vecchi, complessi. Non li vedremo
- *POSIX semaphores*: li vedremo

NOTA: possono essere usati anche tra processi diversi (e non solo tra thread di uno stesso processo)

Tipologie di Semafori

I *POSIX semaphores* possono essere:

- *Named*: hanno un nome univoco. Possono essere usati da più processi indipendenti (anche senza relazioni di parentela)
 - Il più pratico
- *Unnamed*: non hanno nome. Possono essere condivisi tra:
 - Thread, senza particolari accorgimenti (l'unico caso in cui diventa pratico)
 - Processi: se creati tramite **fork** e risiedono in una zona di memoria condivisa (con **shmget** o **mmap**) (il caso meno pratico, anche se possibile)

Il principio di funzionamento è lo stesso:

1. Il semaforo viene creato/inizializzato
2. I processi/thread possono effettuare delle:
 - *Post* per incrementare il semaforo
 - *Wait* per decrementare il semaforo (ed eventualmente attendere)
3. Il semaforo viene distrutto/chiuso

Named Semaphores

Si utilizzano le seguenti funzioni:

1. **sem_open()**
2. **sem_post(sem)**, **sem_wait(sem)** e **sem_getvalue()**
3. **sem_close()** e **sem_unlink()**

Necessario includere l'header:

```
#include <semaphore.h>
```

I semafori sono handle opachi di tipo:

```
sem_t
```

1. Creazione

```
#include <fcntl.h> /* Defines O_* constants */
#include <sys/stat.h> /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char * name , int oflag , ...
               /* mode_t mode , unsigned int value */ );
```

Argomenti obbligatori:

Crea un semaforo dal nome **name**

- Deve iniziare con **/**
- Può essere un qualsiasi identificativo
Esempio: **/mysem**
- L'argomento **oflag** specifica cosa fare *se il semaforo esiste* o no:
 - **O_CREAT**: crea e apre se non esiste. Apre se esiste
 - **O_CREAT | O_EXCL**: crea e apre. Fallisce se già esiste

Argomenti opzionali:

- **value** specifica il valore iniziale
- **mode** specifica i permessi, come per i file

Se si usa il flag **O_CREAT**, **value** vanno specificati!

Valore di ritorno: il semaforo in caso di successo, se no **SEM_FAILED**

2. Chiusura e distruzione

```
#include <semaphore.h>
int sem_close(sem_t * sem );
int sem_unlink(const char * name );
```

sem_close chiude il semaforo per il processo corrente

sem_unlink rimuove il semaforo per tutti i processi

Valore di ritorno: 0 in caso di successo, se no –1

3. *Incrementa/Decrementa*

```
#include <semaphore.h>
int sem_wait(sem_t * sem );
int sem_post(sem_t * sem );
```

sem_wait decrementa di 1 il semaforo

- Se il semaforo dovesse assumere valori negativi, blocca il chiamante

sem_post incrementa di 1 il semaforo

Valore di ritorno: 0 in caso di successo, se no –1

3. *Operazioni particolari*

```
#include <semaphore.h>
int sem_trywait(sem_t *sem);
int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

sem_trywait come la **sem_wait**

- Ma non blocca in caso il semaforo vada in negativo
- Ma fallisce

sem_getvalue colloca nell'intero puntato da **sval** il valore del semaforo

Esempio di Semafori Non Anonimi

Si creino due programmi che comunicano tramite un semaforo.

- Il primo effettua una **post** ogni volta che l'utente preme **Enter**
- Il secondo stampa una stringa ogni volta che il primo effettua una **post**

Programma 1

```

#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <semaphore.h>
#include <string.h>

int main(int argc, char *argv[]){
    sem_t * s;

    s = sem_open("/semaforo", O_CREAT , S_IRUSR | S_IWUSR, 0);
    if(s == SEM_FAILED) {
        printf("Error creating/opening the semaphore %s\n", strerror(errno));
        exit (1);
    }

    while(1){
        printf("Premi enter per una post: ");
        getchar();
        sem_post(s);
    }
    sem_close(s); /* Codice irraggiungibile*/
    return 0;
}

```

Programma 2

```

#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <semaphore.h>
#include <string.h>

int main(int argc, char *argv[]){
    sem_t * s;
    int i = 0;

    s = sem_open("/semaforo", O_CREAT , S_IRUSR | S_IWUSR, 0);
    if(s == SEM_FAILED) {
        printf("Error creating/opening the semaphore %s\n", strerror(errno));
        exit (1);
    }

    while(1){
        sem_wait(s);
        printf("Wait %d effettuata\n", i);
        i++;
    }
    sem_close(s); /* Codice irraggiungibile*/
    return 0;
}

```

Osservazioni:

- Il valore del semaforo è persistente. Se Programma 2 non viene eseguito, il semaforo può crescere di valore
- Si possono eseguire più istanze di entrambi i programmi
 - Più istanze di Programma 1 accumulano valore nel semaforo
 - Se ci sono più istanze di Programma 2, solo una può essere sbloccata per ogni incremento
 - Il sistema operativo tendenzialmente è *fair*. Fa load balancing tra più semafori in attesa

Unnamed semaphores

Si utilizzano in maniera simile, ma più semplice rispetto ai *Named Semaphores*
 Diversa procedure di apertura chiusura

1. Creazione

```
#include <semaphore.h>
int sem_init(sem_t * sem , int pshared , unsigned int value );
```

Crea il semaforo e lo colloca in **sem**, inizializzato a **value**

Importante:

sem_open ritorna un puntatore a semaforo (**sem_t ***), che viene allocato dalla libreria
sem_init colloca il puntatore a semaforo in **sem**

- Il programmatore deve decidere dove allocare il semaforo, di tipo **sem_t**
- Può essere una variabile globale, locale, allocata dinamicamente o su una regione di memoria condivisa

Argomenti obbligatori

Se **pshared** è 0, il semaforo non viene condiviso tra processi, ma solo tra thread

- **sem** può essere una comune variabile globale

Se **pshared** è $\neq 0$, il semaforo viene condiviso tra processi (tramite **fork**)

- **sem** deve essere in una zona di memoria condivisa

Conseguenza: meglio usare Named Semaphore con applicazioni multi-processo

2. Distruzione

```
#include <semaphore.h>
int sem_destroy(sem_t * sem );
```

Distrugge il semaforo **sem**.

Se esso è condiviso tra processi, tutti i processi devono invocare **sem_destroy**

Nota: **sem_close** e **sem_unlink** sono usati solo coi *Named Semaphores*

3. Incremento/Decremento

Si usano **sem_post()** e **sem_wait()** come per i *Named Semaphores*

Unnamed semaphores - Esempio

Si crei un programma con due thread. Il primo ogni secondo manda un messaggio al secondo, usando una variabile globale condivisa (di tipo **char[]**).

Il secondo lo stampa.

Struttura del programma:

```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <pthread.h>

sem_t s_scrittura, s_lettura; /* Due semafori */
char buffer [50]; /* Buffer condiviso tra Thread */

void * sender(void *arg){

    ...

}

void receiver(){

    ...

}

int main(int argc, char *argv[]){
    pthread_t t;
    sem_init(&s_scrittura, 0, 0);
    sem_init(&s_lettura, 0, 1);
    pthread_create(&t, NULL, sender, NULL); /* Thread creato per sender */
    receiver();                      /* Il Main fa da receiver */
}
```

Logica del programma:

Bisogna evitare che un thread legga mentre un altro scrive

- Si potrebbe leggere una stringa in stato inconsistente!
- Senza terminatore!

Servono due semafori:

- **s_scrittura** notifica che **sender** ha terminato una scrittura
 - **sender** mette un **gettone** quando finisce la scrittura, **receiver** attende il gettone per iniziare la lettura
- **s_lettura** notifica che **receiver** ha terminato la lettura
 - **receiver** mette un **gettone** quando finisce la lettura, **sender** attende il gettone per iniziare la nuova scrittura

s_scrittura deve essere inizializzato a 0 perché **receiver** aspetti la prima scrittura
s lettura deve essere inizializzato a 1 perché **sender** possa fare la prima scrittura

Sender:

1. **sem_wait(s_lettura)**: per essere sicuro che **receiver** abbia terminato la lettura
2. Scrive su **buffer**
3. **sem_wait(s_scrittura)**: per notificare termine scrittura

Receiver:

1. **sem_wait(s_scrittura)**: per essere sicuro che **sender** abbia terminato la scrittura
2. Legge su **buffer**
3. **sem_post(s_lettura)**: per notificare termine lettura

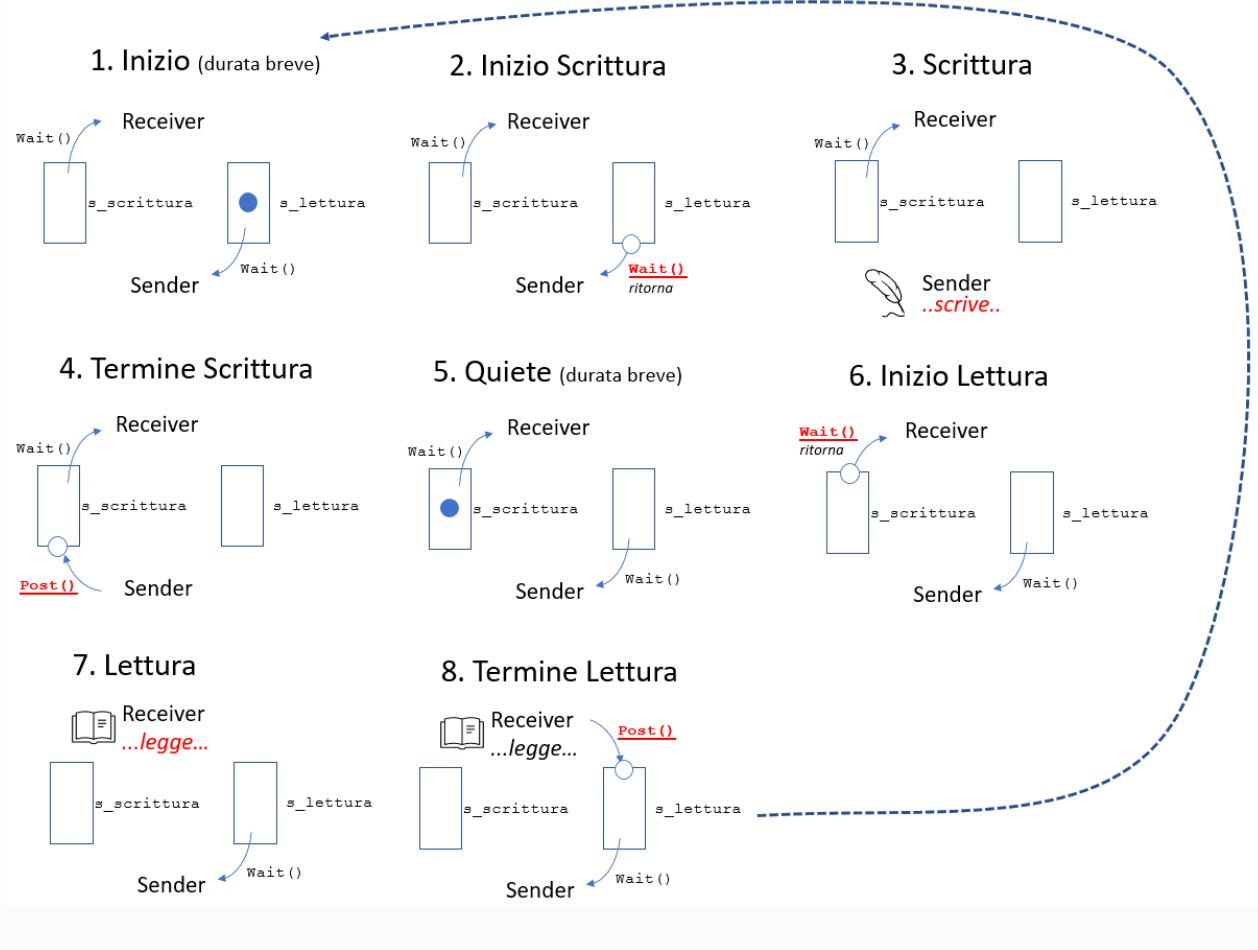
Sender e Receiver:

```
C

void * sender(void *arg){
    int i = 0;
    while (1){
        sem_wait(&s_lettura);
        sprintf(buffer, "Message %d\n", i);
        sem_post(&s_scrittura);
        i++;
        sleep(1);
    }
}

void receiver(){
    while (1){
        sem_wait(&s_scrittura);
        printf("Received: %s\n", buffer);
        sem_post(&s_lettura);
    }
}

...
sem_init(&s_scrittura, 0, 0);
sem_init(&s_lettura, 0, 1);
```



Domande

La parallelizzazione è una soluzione per migliorare le prestazioni:

- di qualsiasi algoritmo
- solo di algoritmi che accedono al disco
- solo di algoritmi che possono eseguiti per mezzo di più flussi contemporanei

Risposta: Solo di algoritmi che possono eseguiti per mezzo di più flussi contemporanei

Il seguente codice è corretto?

```
pthread_mutex_lock(&mtx);
var++;
pthread_mutex_lock(&mtx);
```

- Si, il lock viene rilasciato
- No, il thread entra in uno stato di attesa perpetuo

Risposta: No, il thread entra in uno stato di attesa perpetuo

Un semaforo può essere inizializzato:

- A qualsiasi valore intero

- A qualsiasi intero non negativo

- A qualsiasi intero positivo

Risposta: A qualsiasi intero non negativo

Un programma esegue il seguente codice:

```
C  
sem_init(&s, 0, 0);  
for (i = 0; i<10; i++){  
    sem_wait(&s);  
    sem_post(&s);  
}
```

Al termine del programma che valore assume il semaforo?

- 0
- 10

- Il programma non termina perché entra in uno stato di attesa perpetuo

Risposta: Il programma non termina perché entra in uno stato di attesa perpetuo

Si immaginino due thread di un processo che operano su semaforo **s** inizializzato a 1.

Il Thread 1 esegue:

```
C  
void * t1(void *arg){  
    sem_post(&s);  
    sem_post(&s);  
}
```

Il Thread 2 esegue:

```
C  
void * t2(void *arg){  
    sem_wait(&s);  
    sem_wait(&s);  
    sem_wait(&s);  
    sem_post(&s);  
}
```

Il programma:

- Termina

- Entra in uno stato di attesa indefinito

Risposta: Termina

u7-s3-sync-problems

Sistemi Operativi

Unità 7: I Thread

Problemi di Sincronizzazione

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Mutex e Semafori
 2. Grafi di precedenza
 3. Produttore e consumatore
-

Mutex e Semafori

I **Mutex** regolano l'accesso a una sezione critica:

- Solo un thread per volta può avere il lock
- Operazioni: **lock** **unlock**

I **Semafori** sono degli interi positivi condivisi:

- Simili a un contenitore di gettoni
- Operazioni: **post** **wait**

I **Semafori** sono un costrutto più generale

- Un **Semaforo** può facilmente essere usato come **mutex**. Non vale il contrario!

Costruzione di Mutex con Semaforo

Inizializzazione:

Mutex

C

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

Semaforo: deve essere inizializzato al valore 1

C

```
sem_t sem;
sem_init(&sem, 0, 1);
```

Lock:

Mutex

C

```
pthread_mutex_lock(&lock);
```

Semaforo

C

```
sem_wait(&sem);
```

Release:

Mutex

C

```
pthread_mutex_unlock(&lock);
```

Semaforo

C

```
sem_post(&sem);
```

Per implementazione completa, vedi implementazione in **esercizi/myMutex.c**
myMutex.c

Idea. (implementazione di lock, unlock)

```

typedef struct{
    sem_t s;
} myMutex;

myMutex myMutex_init(){
    myMutex m;
    sem_init(&(m.s), 0, 1);
    return m;
}

void myMutex_lock(myMutex * m){
    sem_wait( &(m->s) );
}

void myMutex_unlock(myMutex * m){
    sem_post( &(m->s) );
}

```

Costruzione di Semafori con Mutex

Si può costruire un semaforo con un **mutex**, ma è *inefficiente*

- Un semaforo è un intero condiviso *positivo*
- Un mutex protegge l'accesso a questo intero

Funzionamento:

- In caso venga effettuato un decremento (**wait**) quando il semaforo ha valore 0:
Il thread attende che un altro thread effettui un incremento (**post**)
- L'unico modo con cui si attendere, è *busy waiting*
 - Un ciclo **for** che verifica ripetutamente
 - Inefficiente

Implementazione (by ChatGPT; se neanche il prof. ha voluto fare...):

```

struct semaphore {
    pthread_mutex_t mutex;
    int count;
};

void semaphore_init(struct semaphore *sem, int count) {
    pthread_mutex_init(&sem->mutex, NULL);
    sem->count = count;
}

void semaphore_wait(struct semaphore *sem) {
    pthread_mutex_lock(&sem->mutex);
    while (sem->count == 0) {
        pthread_mutex_unlock(&sem->mutex);
        pthread_mutex_lock(&sem->mutex);
    }
    sem->count--;
    pthread_mutex_unlock(&sem->mutex);
}

void semaphore_post(struct semaphore *sem) {
    pthread_mutex_lock(&sem->mutex);
    sem->count++;
    pthread_mutex_unlock(&sem->mutex);
}

```

Grafi di precedenza

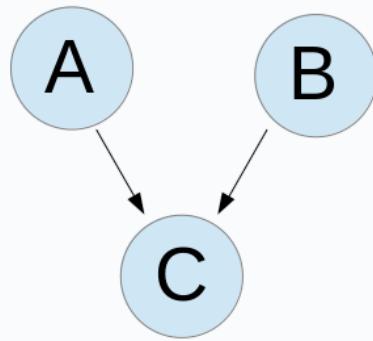
I semafori sono pratici da usare per costruire *grafi di precedenza*

- Un insieme di task che devono essere eseguite in un ordine particolare

I grafi di precedenza modellano molto bene *sistemi distribuiti e concorrenti*

- Le *Reti di Petri* sono un astrazione per trattare grafi di precedenza con l'utilizzo di semafori
- Non vedremo

Esempio 1

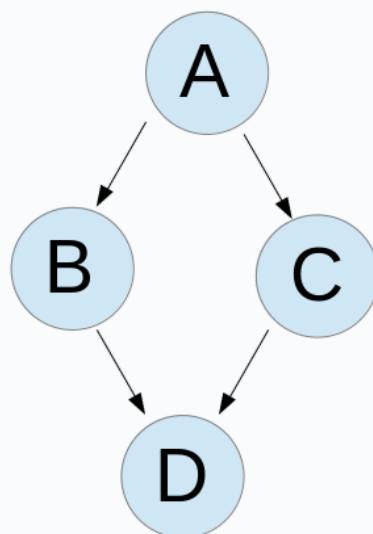


C

```

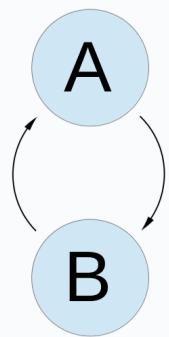
sem_t s1;
void* t_A(void* arg){
    A();
    sem_post(&s1);
}
void* t_B(void* arg){
    B();
    sem_post(&s1);
}
void* t_C(void* arg){
    sem_wait(&s1);
    sem_wait(&s1);
    C();
}
  
```

Esempio 2



```
sem_t s1, s2;
void* t_A(void* arg){
    A();
    sem_post(&s1);
    sem_post(&s1);
}
void* t_B(void* arg){
    sem_wait(&s1);
    B();
    sem_post(&s2);
}
void* t_C(void* arg){
    sem_wait(&s1);
    C();
    sem_post(&s2);
}
void* t_D(void* arg){
    sem_wait(&s2);
    sem_wait(&s2);
    D();
}
```

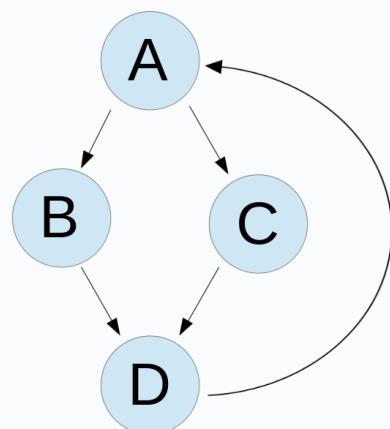
Esempio 3: Grafo ciclico



```
sem_t s1, s2;
sem_init(&s1, 0, 1); // Inizializzato a 1
sem_init(&s2, 0, 0); // Inizializzato a 0
void* t_A(void* arg){
    while (1){
        sem_wait(&s1);
        A();
        sem_post(&s2);
    }
}
void* t_B(void* arg){
    while (1){
        sem_wait(&s2);
        B();
        sem_post(&s1);
    }
}
```

NOTA: esercizio uguale a lettore/scrittore visto in precedenza

Esempio 4



```

sem_t s1, s2, s3; // s1 inizializzata a 1, gli altri a 0
void* t_A(void* arg){
    while (1){
        sem_wait(&s1);
        A();
        sem_post(&s2);
        sem_post(&s2);
    }
}
void* t_B(void* arg){
    while (1){
        sem_wait(&s2);
        B();
        sem_post(&s3);
    }
}
void* t_C(void* arg){
    while (1){
        sem_wait(&s2);
        C();
        sem_post(&s3);
    }
}
void* t_D(void* arg){
    while (1){
        sem_wait(&s3);
        sem_wait(&s3);
        D();
        sem_post(&s1);
    }
}

```

Produttore e consumatore

Vediamo un *problema classico* dell'informatica.

Problema. (*Produttore e consumatore*)

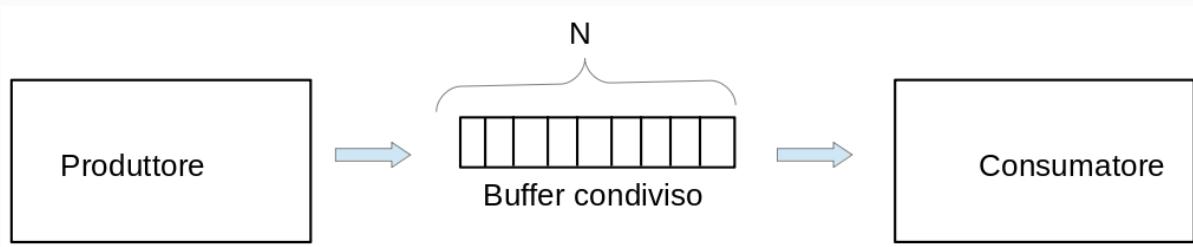
Problema classico dell'informatica, applicabile in molti contesti

- Pacchetti di rete

- Calcolo parallelo

Definizione:

- Due thread comunicano tramite *un buffer di grandezza limitata*, che contiene massimo N oggetti
 - Il thread *producer* inserisce gli oggetti nel buffer
 - Il thread *consumer* estraie gli oggetti dal buffer, nell'ordine in cui sono stati inseriti



Soluzione non-concorrente

Variabili Condivise tra Produttore e Consumatore:

```
C
<tipo> buffer [N]; // Il buffer
int contatore = 0; // Indicazione di elementi usati nel buffer
```

Variabili NON Condivise:

```
C
int in; // Indice dove il produttore inserisce in buffer
       // Gestito in aritmetica Modulo N
int out; // Indice dove il consumatore estrarre
```

Produttore:

```
C
while (1) {
    while (contatore == BUFFER_SIZE); /* non fa niente se il buffer è pieno */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    contatore++;
}
```

Consumatore:

```
C  
while (1) {  
    while (contatore == 0); /* non fa niente se il buffer è vuoto */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    contatore--;  
}
```

Questa

Il codice della slide precedente non funziona.

- C'è accesso *concorrente a variabili condivise*
Le istruzioni **contatore++**; e **contatore--**; non possono essere eseguite simultaneamente
- Alcuni incrementi o decrementi potrebbero essere persi
- Il programma ha un *baco!*
- Il programma non è *thread safe*.

Prima Soluzione Concorrente

1. Accesso concorrente a **contatore**:

è possibile usare un *mutex*

Nota: non c'è mai accesso concorrente a stesso elemento di **buffer**

- Tuttavia le istruzioni **while (contatore == BUFFER_SIZE);** e **while (contatore == 0);** effettuano *Busy Waiting*
 - Controlla continuamente la variabile **contatore**
 - Spreco enorme di CPU!

Soluzione Classica

Si usano due semafori

- Semaforo **empty**: conta quanti posti *liberi* ci sono nel buffer
- Semaforo **full**: conta quanti posti *occupati* ci sono nel buffer

La variabile **contatore** diventa *inutile*. I semafori già contano quanti posti liberi e occupati ci sono

Soluzione completa nel *materiale* in **esercizi/myProdCons.c** (*myProdCons.c*)

Inizializzazione

C

```

<tipo> buffer [N];
sem_t empty, full;

int main(){
    ...
    sem_init(&empty, 0, N); /* Inizialmente N posti liberi */
    sem_init(&full, 0, 0); /* e 0 occupati */
    ...
}

```

Produttore

C

```

int in = 0;
while (1) {
    sem_wait(&empty); /* Attende che ci posto libero nel buffer */
    buffer[in] = next_produced;
    in = (in + 1) % N;
    sem_post(&full); /* Un dato un più nel buffer */
}

```

Consumatore

C

```

int out = 0;
while (1) {
    sem_wait(&full); /* Attende che ci siano dati da consumare */
    <type> next_consumed = buffer[out];
    out = (out + 1) % N;
    sem_post(&empty); /* Un posto libero in più nel buffer */
}

```

tmp

u8-s1-vm-container

Sistemi Operativi

Unità 8: Altri Argomenti

Macchine Virtuali e Container

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Necessità di isolamento
 2. Macchine Virtuali
 3. Container
 4. Cloud
 5. Layer di compatibilità
-

Necessità di isolamento (Motivazioni per le VM)

Diamo una serie di *motivazioni* per parlare delle *macchine virtuali*

Premessa 1: Le organizzazioni *comprano macchine molto potenti*

- Una server potente costa meno di tanti server piccoli

Premessa 2: Ogni utente/dipartimento *ha bisogno un macchina dedicata*

- Un crash in una macchina non compromette l'altra
 - Esempi: Database, Web Server, eccetera...

Conseguenza: Si vuole *dividere* una macchine potente in più macchine *meno potenti*, dandone ognuna un suo servizio

Esempio.

Un organizzazione compra una *macchina potente*, e necessita di un *server Web, FTP e mail*

- Non vuole far girare i **3 software sulla stessa macchine**
- Un problema in uno solo, **può compromettere tutto** e bloccare l'intera azienda
 - Esempio: memory leak, disco pieno, ecc...
- Una vulnerabilità di sicurezza compromette tutti e 3 i sistemi

Soluzione: il server viene diviso in tre **Macchine Virtuali**, ad ognuna un suo compito
 Questa è la tecnica usata **quasi sempre** nelle aziende IT moderne:

- I servizi sono sempre in VM dedicate
 - Vengono eseguiti su server potenti dotati di **Hypervisor** (vedremo che cos'è)
 - I **servizi Cloud** offrono la possibilità di usare VM (vedremo)
-

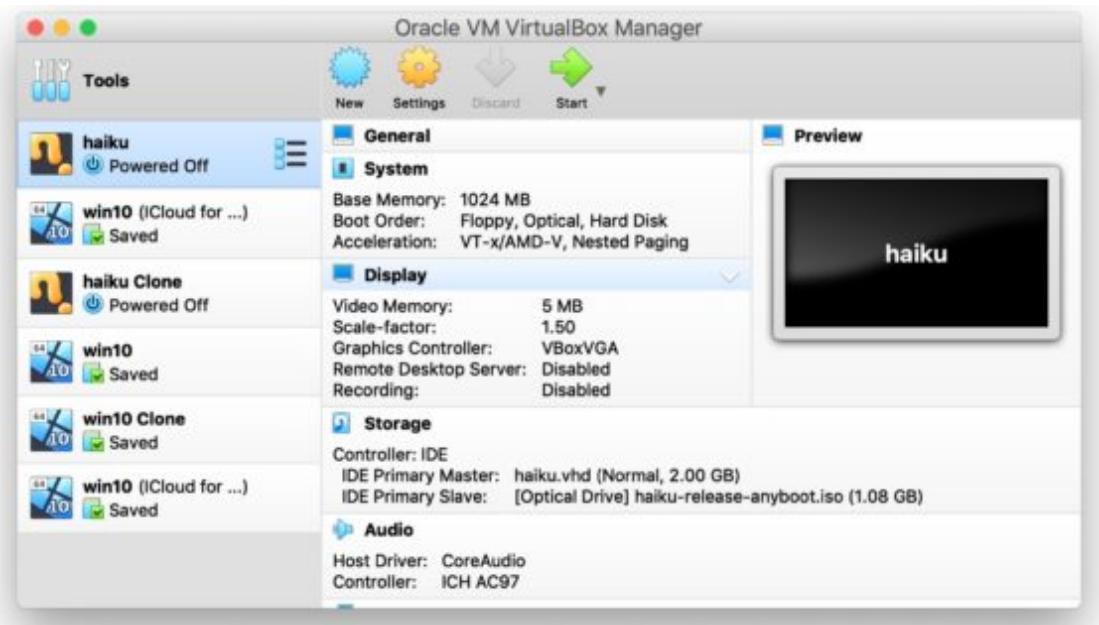
Macchine Virtuali

Definizione di VM e Hypervisor

Una **Macchina Virtuale (VM)** è un **ambiente virtuale che emula un sistema ad elaboratore** (quindi software)

Un **Hypervisor** è il software che rende possibile ciò, usando tecniche di **virtualizzazione**
 Questi devono avere i seguenti requisiti teorico-tecnici:

- **Sicuri:** una VM non deve compromettere il sistema o accedere ad altre VM (ovvero non ha nessun **leak**, l'isolamento deve essere proprio sicuro)
 - **Affidabili:** una VM non deve essere **meno affidabile** di una macchina fisica (in certi casi si può raggiungere il caso in cui le **VM** sono ancora più affidabili!)
 - **Efficienti:** una VM non deve essere **significativamente** meno veloce di una macchina fisica
 - Tante tecniche per arrivare a ciò (in particolare sia dal lato software che dal lato hardware)
 - La "**penalità**" può essere al più 5% (**VM < HW**).
- Per riassumere, un **Hypervisor permette di creare un sistema ad elaboratore virtuale**, con CPU, memoria e disco virtuali
- Eventualmente con accesso rete e dispositivi di I/O fisici o virtuali



Storia delle Macchine Virtuali

1. Il **concepto** nasce negli anni '60, nell'epoca dei mainframe
Poco utilizzati fino ai primi anni 2000
 - Gli **hypervisor erano lenti**, e non vi era grande necessità
 - Si comprava **una macchina fisica per ogni servizio**
 - Riassunto: esisteva, ma era rimasto sulla carta per limiti tecnologico-economici.
2. Tornano alla ribalta negli **anni 2000**
 - Gli Hypervisor hanno fatto un **salto tecnologico**, diventando **efficientissimi**
 - Sono in grado di emulare l'**Hardware** come se fosse nativo: il lavoro degli Hypervisor viene facilitato proprio dall'architettura stessa
 - I server sono diventati **molto potenti**, rendendo conveniente **dividerli** in più macchine di potenza intermedia
 - Abbiamo più core, memoria, in generale i computer sono molto più potenti
3. Oggi le **macchine virtuali** e gli **Hypervisor** sono una tecnologia pienamente matura:
 - **Sicura**
 - **Efficiente:** meno del 5% di penalizzazione rispetto a macchina fisica
 - Tutte le aziende hanno **cluster** dedicati a **ospitare VM**
 - Un **team specializzato** gestisce il cluster e il software di virtualizzazione
 - I **team di sviluppo** (anche questo specializzato) installano i servizi su VM dedicate

Tipologie di Hypervisor

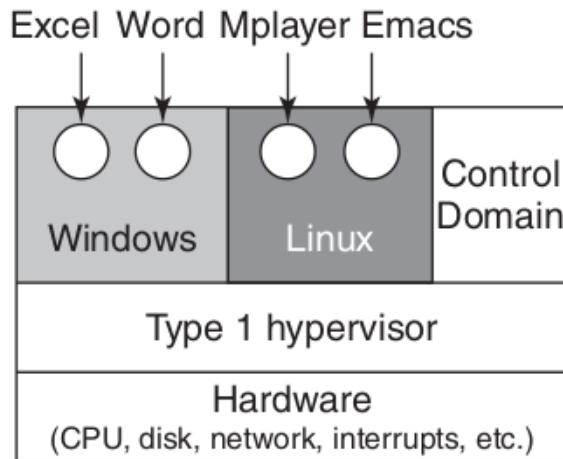
Ci sono due tipologie di Hypervisor.

Hypervisor di Tipo 1

E' un **SO dedicato che serve solo a creare VM**

Efficienti perché hanno il controllo completo della macchina

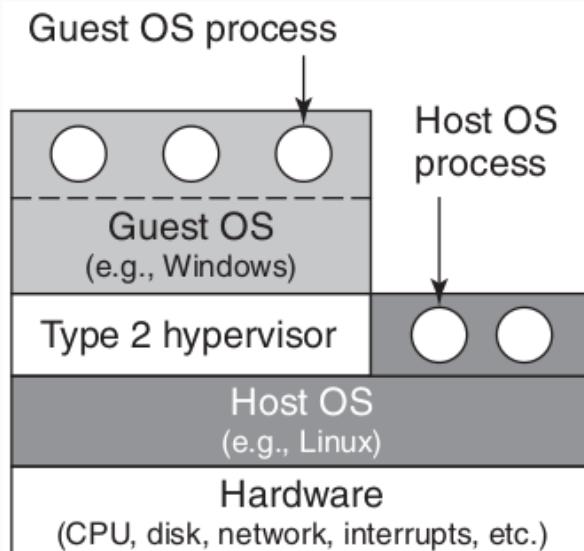
Esempi: Xen, Microsoft Hyper-V, VMware ESXi



Hypervisor di Tipo 2

E' un *software eseguito in un normale SO* che permette di creare *Macchine Virtuale Meno efficienti* (dato che, come si vede nella figura, ho uno "strato" di software in più); ma ormai i SO offrono assistenza a Hypervisor di Tipo 2

Esempi: VMWare Player, Virtual Box, QEMU, Parallels



Problemi degli Hypervisor

Come visto prima, gli *Hypervisor* hanno il compito di emulare *sistemi ad elaboratori* dal lato software, mantenendo la stessa *affidabilità* ed *efficienza* dell'hardware. In particolare avremo i seguenti problemi:

- Ottimizzazione della CPU
 - Ottimizzazione della Memoria
- Vedremo come verranno risolti questi problemi, sia dal lato *software* che dal lato *hardware*.

Ottimizzazione della CPU

Un Hypervisor permette di *emulare in software una CPU virtuale*, potenzialmente di *architettura diversa* rispetto alla macchina fisica

- **Esempio:** emulare ARM su CPU x86
- **Problema:** *moltissimo lento!* Si deve implementare in software una CPU. In questo caso si ha VM << HW.

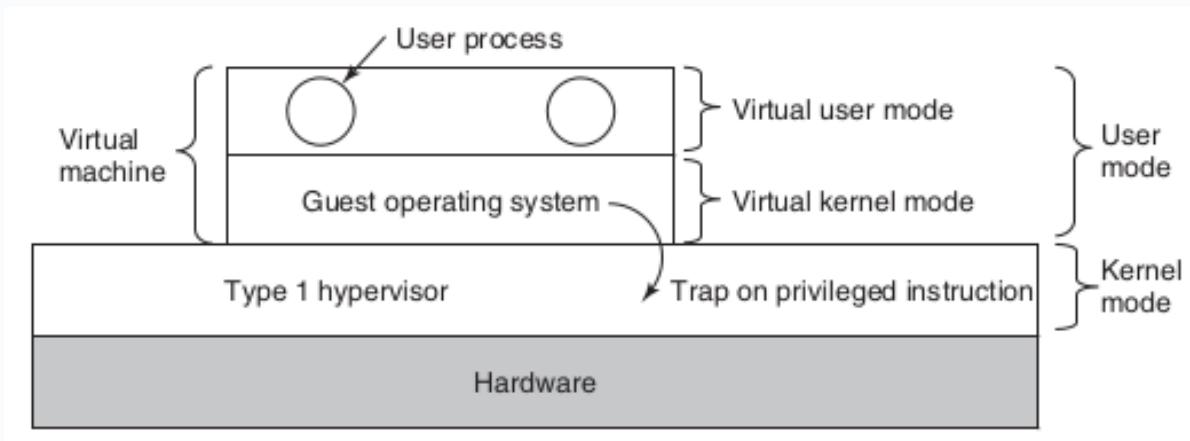
Solitamente ciò non avviene e si *ottimizza* l'uso della CPU

- La VM esegue le istruzioni direttamente sulla CPU fisica
 - Ovvero sono *eseguite davvero* sulla *CPU fisica*. Da questo ci sono ulteriori problematiche, tra cui la gestione delle *risorse globali*: su questo aiuterà il lato *Hardware*.
- *Necessaria cautela*, per evitare i *leak*.

Soluzione. (*Virtual Kernel Mode*)

Nei moderni Hypervisor, la VM esegue le istruzioni sulla *CPU fisica*

- Le CPU moderne permettono il *virtual kernel mode*
 - Permette di eseguire istruzioni in *kernel-mode*
 - Limitando i *privilegi*
 - Ovvero il *VM* crede di essere in *modalità kernel*, anche in realtà è limitata.
- Il kernel della VM esegue il suo codice in *virtual kernel mode*
 - Altrimenti potrebbe leggere tutta la memoria della macchina fisica! Che sarebbe pericolosissimo (esempi: leak, corruzione, accesso potenziale ai malintenzionati, eccetera...)



Ottimizzazione della Memoria

Abbiamo il problema analogo, solo che al posto della *CPU* occupiamo della *memoria*.

Un Hypervisor, se emula la CPU, *emula anche memoria in software*

- Ogni volta che una VM accede a una *locazione di memoria*, l'*Hypervisor* esegue del codice per fornigli il risultato
- *Lentissimo!*

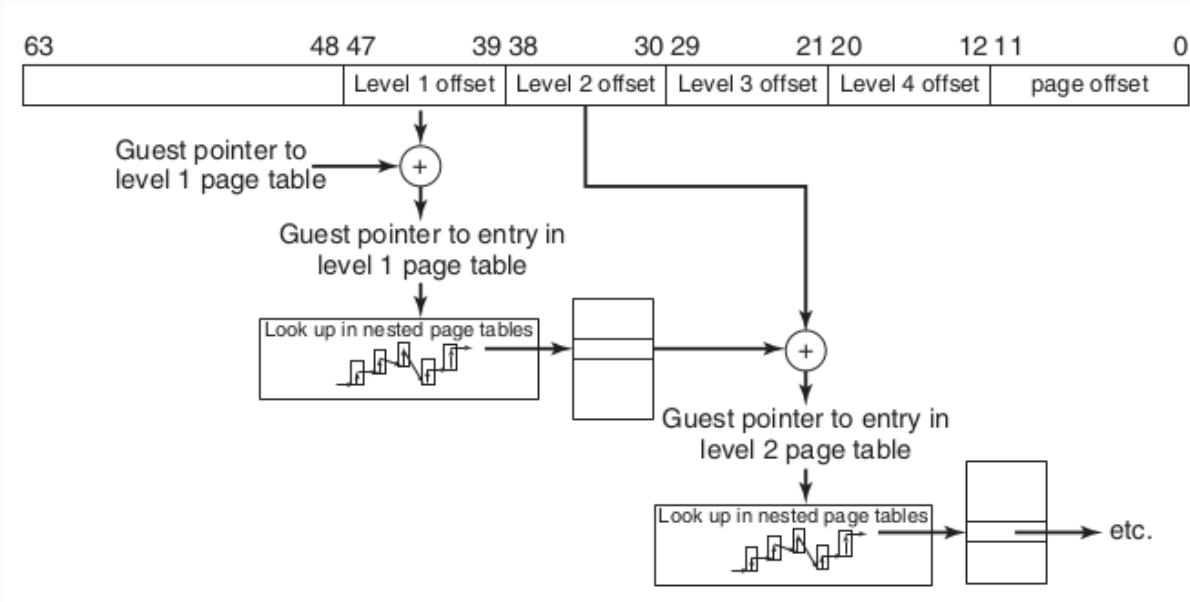
Gli Hypervisor moderni permettono alle VM di accedere *direttamente a porzioni di memoria fisica* (quindi come prima, usiamo veramente la *memoria fisica*)

- Necessari due livelli di *paginazione*
 - All'*interno della VM*: da *memoria virtuale di processo* a *memoria della VM*
 - Da *memoria della VM* a *memoria fisica*
- Abbiamo una composizione del tipo $(P)_{VM} : IV_{VM} \mapsto IF_{VM} \mapsto IF_{HW}$
Serve cooperazione del sistema fisico e della CPU!

Soluzione. (*Page Table Annidate*)

Le CPU moderne supportano *Page Table annidate*

- *Primo livello*: mappa tra processo nella VM a memoria della VM ($IV_{VM} \mapsto IF_{VM}$)
- *Secondo livello*: mappa tra memoria della VM e memoria fisica ($IF_{VM} \mapsto IF_{HW}$)



Problema Attuale

In realtà ancora oggi c'è un altro problema ovvero la gestione dell'*Input/Output*: come faccio ad esporre direttamente l'*I/O* alla *VM*?

- Questo problema rappresenta la "*ultima frontiera*" degli Hypervisor
- Situazioni applicabili: Reti neurali, per le *GPU*

Tecnologie Principali per le VM

- Per giocare: **VirtualBox** o **QEMU**
- Per installare VM su un server: **Kernel-based Virtual Machine (KVM)** + **Libvirt**
- Per un cluster di VM: **OpenStack**

Altre alternative possibili, non tutte *open source* e *free* (tipo **VMWARE**)

Container

Motivazioni per Container: Limiti delle VM

1. Allocazione statica delle risorse

Una VM ha allocate **staticamente** una certa quantità di risorse della macchina fisica

Esempio: un server con 16 core e 64GB di RAM

- Posso fare 3 VM con 5 core e 20GB di RAM ognuna
- Alcune risorse vanno mantenute per il funzionamento della macchina fisica: 1 core 4GB di RAM

Questo può essere **inefficiente**

- Non sempre tutte le VM hanno necessità di 5 core! Si potrebbe allocare **dinamicamente** le risorse, compiendo una specie di **"overbooking"** in certi casi.

2. Eccesso Relativo di Software

Con l'approccio **"una VM - un'applicazione"**, su tutte le VM gira un SO, che di fatto **esegue pochi processi**

- Quelle per cui è dedicata la VM
- **Inefficiente!** proliferazione di **SO che non fanno quasi niente!**
- Esempio: 10 VM \Rightarrow 10 Ubuntu

Quando voglio avviare una **nuova applicazione**, devo:

- Creare una VM
- Installare il **SO**
- Avviare la mia applicazione
Ci metto tanto tempo (di lavoro umano)

VM vs SO

Facciamo un parallelismo filosofico tra le **VM** e i **SO** (processi).

Ricordiamoci a cosa servono le **VM**:

- **Isolare** sistemi indipendenti
- **Controllare** che essi non si danneggino a vicenda
Ma è simile allo scopo di un **processo** in un SO. Il **SO** serve a:
 - **Isolare** processi diversi (con le tecniche della **memoria virtuale**)
 - **Controllare** l'accesso alle risorse tramite utenti e privilegi
Quindi l'idea per superare il problema iniziale è quello di usare i processi, al posto delle VM.

Problema.

Purtroppo, in un **SO, un'applicazione problematica** può **bloccare il sistema**, se:

- Usa al 100% la CPU
- Riempie il disco o la RAM
Un'**applicazione** potrebbe **provocare problemi** a un'altra applicazione
- Satura le risorse di I/O rete, ecc...

- Se *modifica i suoi file di configurazione* (solo se eseguita come **root**)

Soluzione: Potrei avviare un processo che ha *risorse limitate*

- Il SO si occupa di limitare l'accesso a CPU/memoria/disco (quindi di creare delle specie di gabbie)
Sarebbe *quasi* come una VM
- Un'applicazione che gira *senza poter influenzare le altre applicazioni!*
I sistemi Linux forniscono queste funzionalità:
- Ovvero far girare processi con *privilegi limitati*. Vediamo di dare una definizione formale ad un costrutto del genere

Definizione di Container

Un *container* è un albero di processi che gira con privilegi limitati

- Non ha accesso completo alle risorse (disco, CPU, memoria, file, etc.)
- Pensa di essere l'*unico* (insieme di) *processo(i) in esecuzione*
I container sono un'illusione: illudono un processo di avere poche risorse.
Vedere: [Containers as an illusion](#) per approfondire il discorso

I processi di un container quindi non possono:

- Vedere gli altri processi della macchina
- Vedere le risorse che *non* gli sono state assegnate
Ovviamente un container *non deve poter compromettere l'intera macchina*
- Si utilizzano varie funzionalità di Linux per raggiungere questi scopi

Vedremo in particolare le funzionalità Linux per:

- Isolare File System
- Isolare risorse della CPU e della Memoria
- Isolare i Namespace

Isolamento del File System

Linux permette di avviare un processo che vede solo un sotto albero del FS

Funzionalità chroot: cambia *radice del FS*

Permette di evitare che un processo (e i suoi figli) legga/modifichi file fuori dall'albero

Sintassi:

SHELL

```
chroot /path/to/new/root command
```

Ovvero il processo **command** ha la radice in **/path/to/new/root** e vede solo questo sottoalbero.

Isolamento delle Risorse CPU-Memoria

E' possibile limitare quanta CPU e memoria un processo usa.

Funzionalità cgroup: offerta dalle System Call Linux

- Permettono di limitare:
 - Uso della CPU
 - Uso della memoria
 - Velocità di I/O
 - Traffico di rete

Ovvero, permettono di evitare che un processo sovraccarichi il sistema

I **cgroup** sono relativamente nuovi. *Stabili dal 2018* con una modifica al *Kernel Linux*.

Vengono usati attraverso *uno pseudo file system*

- In **/sys/fs/cgroup**

Operazioni:

1. Creazione di un gruppo di processi:

mkdir /sys/fs/cgroup/my-group

2. Limitazione delle risorse:

echo 50000 100000 > /sys/fs/cgroup/cpu/my-group/cpu.max (ovvero scrivo su **cpu.max** la frazione)

Significa che i processi del gruppo, in totale, non possono usare più del 50% del tempo CPU della macchina

3. Collocazione di un processo nel gruppo:

echo 8764 > /sys/fs/cgroup/cpu/my-group/cgroup.procs (ovvero scrivo su **cgroup.procs** il processo a cui appartiene **cgroup**)

Isolamento dei Namespace

E' possibile creare processi che *non vedono le risorse globali della macchina fisica*, ovvero:

- Quali sono gli *altri processi in esecuzione*
- Le *interfacce di rete*
- I *dispositivi di I/O*
- Gli *utenti* e *gruppi* sulla macchina

Funzionalità Namespace: offerta dalle System Call Linux

- Vedi comandi **unshare** e **nsenter**

Container Engine

Queste funzioni del SO appena elencate sono *potenti*, ma *poco usabili*:

- Per usarle, *necessario conoscerle a fondo*
 - *Errori nell'utilizzo* possono compromettere il sistema
 - Non c'è sicurezza by default:
 - Necessario *togliere* privilegi ai processi
- Quindi devo fare tutto a mano... qual è la soluzione?

Esistono dei software che si chiamano *Container Engine* (ovvero dei tool) che permettono di usare *in maniera semplice queste funzionalità*

- *Avviare container*: gruppi di processi isolati
- *Monitorarne* il funzionamento

Offrono comandi/API semplici per creare container. Popolari:

- **Linux Containers (LXC)**: tra i primi a nascere nel 2008
- **Docker**: Nato nel 2013. Standard *de facto*

Principio di funzionamento: eseguono processi con risorse limitate, che vivono in un file system limitato

- Di *default*, i container hanno *privilegi minimi*
- Possibile configurarli per avere maggiori privilegi: e.g., accedere a porzioni del FS

Docker

Docker: Definizione

Si può installare *su ogni macchina Linux*

- Disponibile anche su MacOS e Windows (ma implementato tramite una VM)

Permette di avviare container a partire da una *Immagine*:

- E' un File System che *contiene il programma da eseguire*
- Ed *eventuali dipendenze*: librerie condivise, altri programmi, file di configurazione

La componente interna di Docker che permette di eseguire i container si chiama **containerd**

Docker: container e immagine e hub

Un *Container* è una *Immagine* in esecuzione:

- Un insieme di processi che può operare solo sui file presenti nell'immagine
- I file dell'immagine vengono copiati
- I processi possono creare nuovi file o modificare quelli esistenti
- Non può accedere ai file della macchina fisica
- I figli condividono la stessa "gabbia"

Esiste una libreria di immagini pre-costruite su **Docker Hub** (<https://hub.docker.com/>)

- Ognuna contiene un software installato con le sue dipendenze
- Può essere scaricata ed eseguita, creando un container
- Ogni immagine ha una versione identificata da un **tag**
 - **latest** identifica l'ultima versione

E' anche possibile *creare la propria immagine* col proprio software

Comandi di Docker

- **docker pull <immagine>**: scarica un **immagine** da Docker Hub
 - **docker ps**: mostra i container in esecuzione
 - **docker run --name <nome> <immagine>**: esegue un container da **immagine** e gli assegna il **<nome>**
 - Argomento **-v pathLocale:pathContainer**: permette al container di accedere **pathLocale** che viene montato in **pathContainer**
 - Argomenti **--cpus <n>**, **--memory=<n><s>**: limita le risorse dell'immagine
 - Argomento **-d**: processo in background (diventa un **demone**)
 - Argomento **-e VAR=VAL**: specifica delle variabili d'ambiente (come password, eccetera...)
 - **docker stop <nome>**: termina il container identificato da **nome**
 - **docker logs <nome>**: vedere l'output dell'immagine in esecuzione
 - **docker inspect <nome>**: per vedere informazioni sull'immagine
- Molti altri comandi...

Necessari permessi di **superuser**, da fornire con **sudo**

Docker e file montati

Di default, un container **non** può accedere ai file della macchina fisica, ma solo a una copia di quelli dell'immagine

L'opzione **-v pathLocale:pathContainer** permette al container di accedere a **pathLocale**, che viene **montato** in **pathContainer** all'interno del FS del container

Esempio:

```
docker run --name nome \
-v /home/martino:/opt/home-di-martino \
immagine
```

Il container **nome** può accedere al path fisico **/home/martino** tramite il path **/opt/home-di-martino**

Docker e Rete

Ogni container ha un **indirizzo IP** in una **rete virtuale** che collega tutti i container

- Possibile comunicazione tra **container**
- Possibile comunicazione tra **macchina fisica e container** (esempio: Server Web)
- Possibile comunicazione tra **container e Internet** tramite Default Gateway virtuale

Docker e Risorse

Limitazione di CPU: **docker run --cpus 2 <immagine>**

Limitazione di memoria: **docker run --memory=512m <immagine>**

Docker: Esempio

Creazione di container per eseguire il DBMS **PostgreSQL**

- Un DBMS relazionale
- Un processo del database deve essere in esecuzione
- Vi si accede tramite rete e un protocollo dedicato

Scaricamento dell'immagine:

docker pull postgres

Avviamento del container:

```
docker run -d \
--name some-postgres \
-e POSTGRES_PASSWORD=mysecretpassword \
-e PGDATA=/var/lib/postgresql/data/pgdata \
-v /home/martino/db:/var/lib/postgresql/data \
postgres
```

Opzioni usate:

- **-d**: fai partire il processo in background
- **-e VAR=VAL**: specifica variabili d'ambiente visibili nel container
 - Usato per password del DB e per specificare dove esso salva i dati
- **-v /home/martino/db:/var/lib/postgresql/data**: i dati sono salvati sulla macchina fisica in **/home/martino/db** ma nel container al path **/var/lib/postgresql/data**

Privilegi del container:

Il container **some-postgres** esegue l'immagine **postgres**.

Ha accesso alle risorse fisiche di:

- CPU e memoria senza limiti
- File System: solo **/home/martino/db**
- Rete: ha un indirizzo IP. Il server si mette in ascolto sulla porta di default 5432

Monitoraggio:

Se tutto è andato a buon fine, il container è in esecuzione. Si osserva con:

SHELL

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
c6320fa9eb9b postgres "docker-entrypoint.s..." 50 seconds ago Up 49
seconds 5432/tcp some-postgres
```

Si può ottenere il suo IP con

SHELL

```
$ docker inspect some-postgres
...
"IPAddress": "172.17.0.2",
...
```

Sulla macchina fisica:

I processi di **postgres** sono normali processi in esecuzione, ma con privilegi **molto limitati**

SHELL

```
$ ps fax
443964 ? Sl 0:05 /usr/bin/containerd-shim-runc-v2 -namespace moby ....
443985 ? Ss 0:01 \_ postgres
444080 ? Ss 0:00 \_ postgres: checkpointer
444081 ? Ss 0:00 \_ postgres: background writer
444083 ? Ss 0:00 \_ postgres: walwriter
444085 ? Ss 0:00 \_ postgres: logical replication launcher
...
...
```

I file dove il DB salva i suoi dati sono in

SHELL

```
$ sudo tree /home/martino/db/ -L 2
/home/martino/db/
└── pgdata
    ├── base
    ├── global
    └── pg_commit_ts
...
...
```

Utilizzo:

Il DB si può usare installando il client **psql**, col comando:

SHELL

```
$ PGPASSWORD=mysecretpassword psql -U postgres -h 172.17.0.2
psql (12.12 (Ubuntu 12.12-0ubuntu0.20.04.1), server 15.1 (Debian 15.1-
1.pgdg110+1))
WARNING: psql major version 12, server major version 15.
Some psql features might not work.
Type "help" for help.
```

```
postgres=#
```

Il DB salva i dati nella cartella fisica: **/home/martino/db**

Con 3 semplici comandi si è installato PostgreSQL!

Utilizzo odierno

L'utilizzo di **container** sta *prendendo il posto* dell'utilizzo delle **VM**.

- Più scalabile
- Costringe a separare codice da dati
- Ho lo stesso *livello di isolamento*, con *meno fatica!*

Nelle grandi aziende, si utilizzano **cluster di nodi** che eseguono container.

Esistono **software di orchestrazione di container** basati su Docker:

- **Kubernetes**: il più usato. Open-Source
- **OpenShift** e **OKD**: proprietari di **Red Hat**

Tecnologie Cloud

Scenario

Le tecnologie di VM e container permettono a un'azienda di *collocare i propri servizi in qualsiasi luogo del mondo* (ovvero è tutto *virtualizzato*)

Per molte aziende è conveniente *affittare una VM* da un'azienda specializzata, anziché comprare server fisici

- Avere server farm è *costoso*: necessario raffreddamento e sorveglianza
 - Economia di scala con data center grandi
 - Vedi: requisiti per *costruire un Data Center di livello 4, standard EN50600*
- Il personale specializzato è *poco e costa molto!*
 - Vedi: per costruire un data center, bisogna avere *più team specializzati* per progettare tutto, poi bisogna costruire il data center, poi un team per mantenere tutto, eccetera...
- Malfunzionamenti possono provocare *gravi danni economici!*

Cloud Provider

Conseguenza: sempre più spesso le aziende comprano servizi da **Cloud Provider**

Tra i più popolari cloud provider:

- **Amazon Web Services**
- **Google Cloud**
- **Microsoft Azure**
- **Aruba** (in Italia)
- Poi molti altri! (tipo dei provider ad-hoc per la *Pubblica Amministrazione*)

Pro: Il costo immediato è molto basso

Contro: Il costo a lungo termine è significativamente alto (in alcuni anni copri il costo immediato)

Vedremo i *pro/contro* in dettaglio ulteriormente.

Servizi offerti dai Cloud Provider

Diverse tipologie di servizi offerti dai cloud provider

- **IAAS (Infrastructure As A Service)**: possibilità di creare e utilizzare *VM* o *container*
- **PAAS (Platform As A Service)**: il cloud provider offre una *piattaforma di sviluppo*. L'utente scrive solo l'applicazione
 - **Esempio 1:** Database SQL remoto in Cloud
 - **Esempio 2:** servizio di hosting per siti web dinamici: supporto a hosting HTML, esecuzione server-side di PHP e SQL
- **SAAS (Software As A Service)**: l'utente/azienda compra una *subscription* a un *servizio completo*
 - **Esempio:** un'azienda compra un abbonamento a Microsoft Teams



Prospettive Odierne delle Tecnologie Cloud

Sempre più spesso aziende ed enti pubblici fanno ricorso a Cloud Provider per IAAS/PAAS/SAAS; tuttavia ancora oggi queste tecnologie sono ancorai in *fase di discussione*.

Vantaggi:

- Minore costo iniziale
- Maggiore affidabilità (infatti, *Google*, *Amazon* sono affidabili)

Svantaggi:

- *Vendor Lock-in* (ovvero praticamente sono "legato" all'ente fornitore, ho bisogno di tutele legali)
- Perdita di *Know How* (in particolare della *sovranità dei dati*: ripercussione particolare sui *problemi geopolitici*, come conflitti internazionali, leggi che tutelano dati (GDPR), eccetera...)
- Costo elevato nel lungo termine (in *2 anni* avrei coperto il costo immediato per *un server*)

Layer di compatibilità

Torniamo indietro con queste tecnologie di virtualizzazione. In particolare siamo sul *livello utente*.

VM e Software

Le VM permettono di avere un sistema ad elaboratore *virtuale*

- Su cui installare *un SO a piacere*
- Esempio: VM con Linux su PC Windows

Spesso per l'utente, la VM serve solo a usare un *software* scritto per un SO diverso

- Esso può girare solo su (*Architettura, SO*) *per cui è stato compilato*
- Non è possibile usare su altro *SO*, anche se stessa *Architettura*. *Le System Call sono diverse!*

Tuttavia le *macchine virtuali* sono un po' "*esagerate*" per questa casistica. Vediamo un'altra tecnica di virtualizzazione, ovvero i *Layer di Compatibilità*.

Definizione di Layer di Compatibilità

Un *Layer di compatibilità* è un *software* che permette di eseguire un programma scritto per un *SO* diverso: sostanzialmente implementa delle *System Call*

- Ma compilato su stessa *Architettura*
Implementa le **System Call** di un altro SO, tramite quelle del SO corrente.
- **Esempio:** Win32 **ReadFile** ⇒ POSIX **read**

Funzionamento *complesso* e problematico

- Esistono meccanismi *non-mappabili*
- Gestione di I/O complessa: dipende da SO e da driver
- Le System Call diverse hanno una semantica diversa
- Molto difficile, comunque possibile

Types of System Calls	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

Layer di Compatibilità a Livello API e ABI

Layer di compatibilità a livello Application Programming Interface (API):

Richiede ricompilazione del software

- Basato su una libreria software implementa le System Call di un SO tramite *quelle di un altro*
- Si fa una specie di "*massaggio degli argomenti*" delle System Call

Layer di compatibilità a livello Application Binary Interface (ABI): *NON* richiede ricompilazione del software

- Il programma usa le System Call del proprio SO. Il Layer *le intercetta e invoca quelle del SO corrente*

Tecnologie Principali

1. CYGWIN

Permette di usare programmi che usano System Call **POSIX** su Windows

- A livello **API**
- Richiede ricompilazione

Nota: POSIX \neq Linux

Cygwin è semplicemente un'altro ambiente per compilare ed eseguire programmi POSIX che usano le System Call e Librerie POSIX

2. WINE

Permette di usare programmi per **Windows** su Linux e MacOS

- A livello **ABI**
- Non richiede ricompilazione
 - Non sarebbe possibile con software closed-source Windows

Molto matura e usata:

- Funzionano anche programmi con interfaccia grafica
- Alcuni programmi complessi invece non si possono usare

3. WSL 1

Permette di usare programmi per **Linux** su Windows

- A livello **ABI**
- Non richiede ricompilazione

Nota. (WSL 2)

Invece la **WSL 2** è una VM minimale con un vero kernel

- **NON** è un Layer di compatibilità
- Più flessibile, ma più lenta

Domande

Quale tra questi non è una motivazione per l'uso di VM?

- **Maggiore sicurezza**
- **Maggiore affidabilità**
- **Maggiore velocità della memoria**

Risposta: Maggiore velocità della memoria

Una macchina fisica sta eseguendo una VM. Quanti kernel sono in esecuzione?

- **Nessuno**
- **1**
- **2**
- **3**

Risposta: 2

Una VM può usare direttamente la memoria fisica della macchina?

- **Mai**
- **Sempre**
- **Se la CPU lo permette**

Risposta: Mai

Una macchina fisica sta eseguendo una container. Quanti kernel sono in esecuzione?

- **Nessuno**
- **1**
- **2**
- **3**

Risposta: 1

Cosa è un container?

- **Un FS isolato**
- **Un namespace**
- **Un gruppo di processi con privilegi limitati**

Risposta: Un gruppo di processi con privilegi limitati

Un container può accedere al File System della macchina ospitante?

- **Sempre**
- **Mai**
- **Dipende da come è stato creato**

Risposta: Dipende da come è stato creato

Quale tra questi non è un servizio offerto dai Cloud Provider?

- **Esecuzione di VM**
- **Abbonamento a database remoto**
- **Licenze di software da eseguire su PC**

Risposta: Licenze di software da eseguire su PC

Quali delle seguenti affermazioni é vera? Un layer di compatibilità:

- **é una VM**
- **é un insieme di container**
- **permette di eseguire programmi compilati su un'architettura diversa**
- **permette di eseguire programmi compilati su un SO diverso**

Risposta: permette di eseguire programmi compilati su un SO diverso

u8-s2-socket

Sistemi Operativi

Unità 8: Altri Argomenti

Rete e socket in Linux

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

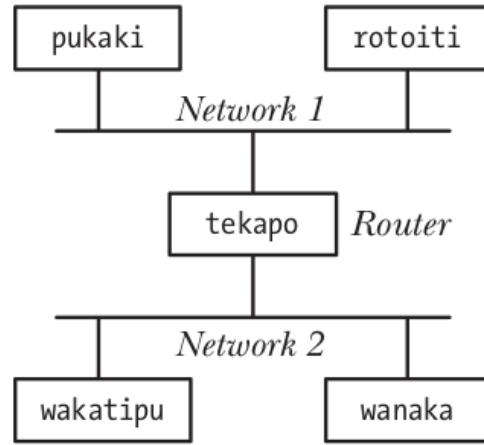
1. Lo stack di rete TCP/IP in Linux
 2. I Socket
 3. Funzioni e System Call per i Socket
 4. Comandi per Networking in Linux
-

Lo stack di rete TCP/IP in Linux

Definizione di Internet

Internet è un l'*insieme di nodi e apparati di rete* che permettono una *comunicazione mondiale*

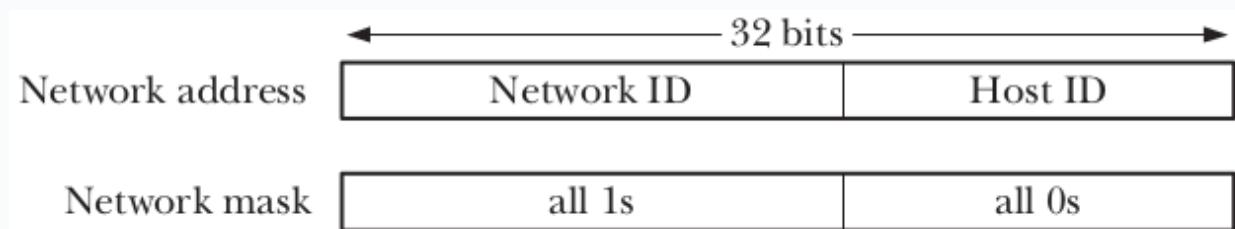
- Internet è l'unione di tante *Network*
- Collegate tramite *Router* (serve per *distinguere traffici di rete*)
- Ogni nodo è identificato da un *Indirizzo IP*



Indirizzi IP

Un indirizzo IP identifica univocamente un nodo in Internet

- Numero su 32 bit. Sono *pochi!* Con dei calcoli si trova che $2^{32} \ll 7 \cdot 10^9$
- Composto da una *parte di network* e *una di host*
 - La *netmask* delimita le *due parti*
 - Necessario per la trasmissione di pacchetti tramite Ethernet
 - L'indirizzo IP 127.0.0.1 identifica per convenzione il *Local Host*
 - Ovvero serve per mandare un pacchetto a se stesso



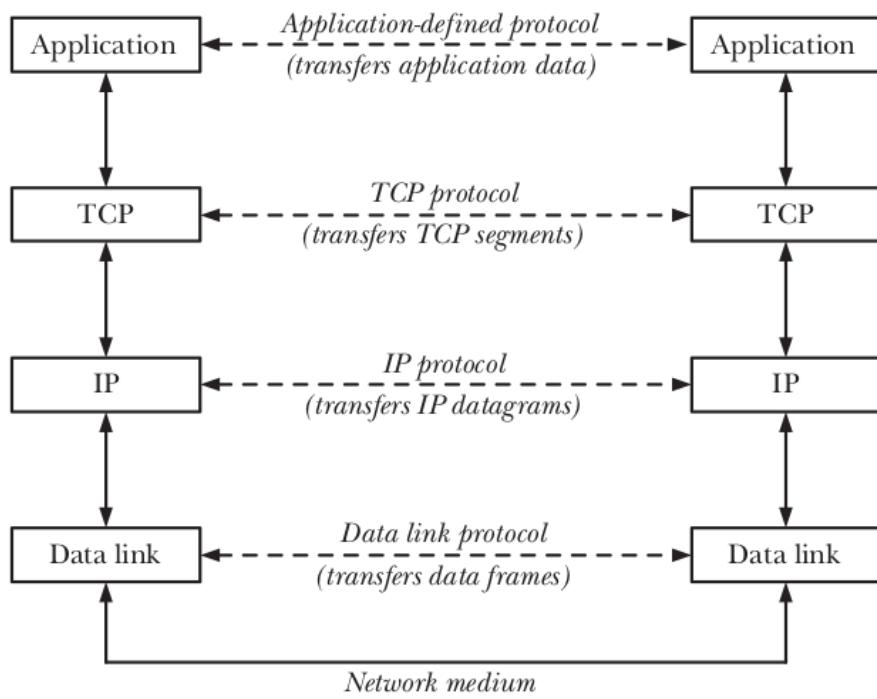
Protocolli Rete

I protocolli formano una *Pila*:

- Il livello N usa i servizi del livello $N - 1$
- Li migliora e li offre al livello $N + 1$
- Il livello N parla col suo omologo su un altro nodo

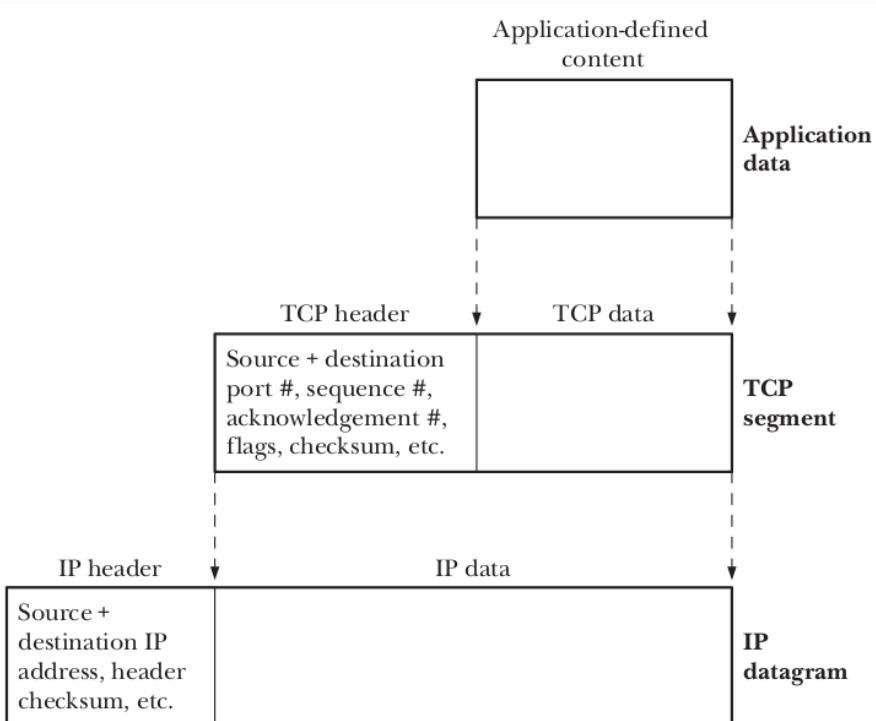
In particolare i livelli sono i seguenti:

- *L1* - Dati binari (funzionalità semplice)
- *L2* - Messaggi (ethernet, wifi, ...)
- *L3* - Identificatori (IP)
- *L4* - Identificativi dei processi, porte
- ...
- *L7* - Applicazioni (Esempio: Server Web)



I protocolli vengono **inscatolati** uno dentro l'altro:

- Un frame **Ethernet** trasporta un pacchetto **IP**
- Un pacchetto **IP** trasporta un segmento **TCP**
- Un segmento **TCP** contiene i dati dell'**applicazione**



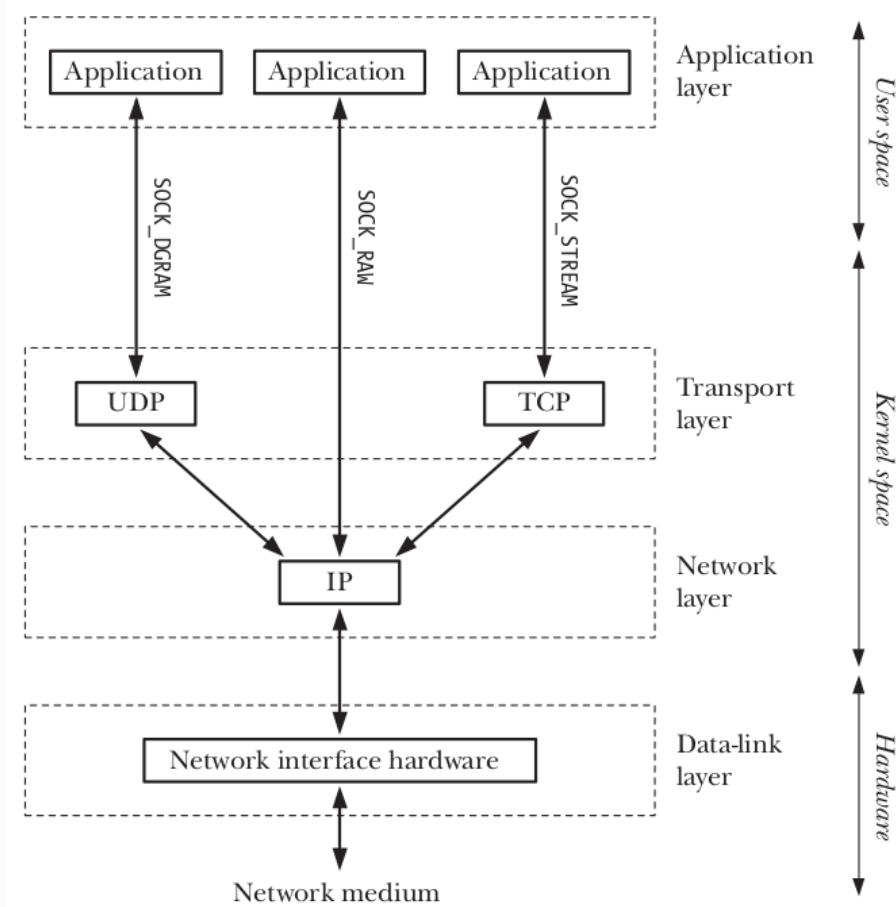
Le applicazioni in Linux possono usare i servizi di:

- **TCP** per avviare una comunicazione orientata al flusso (**SOCK_DGRAM**)
- **UDP** per mandare datagrammi (**SOCK_STREAM**)
- **Pacchetti IP generici** (**SOCK_RAW**)

Non li implementeremo, impareremo ad usare uno dei servizi

Il *kernel* implementa i moduli TCP, UDP, IP

Offre delle *System Call* per poterne utilizzare i servizi

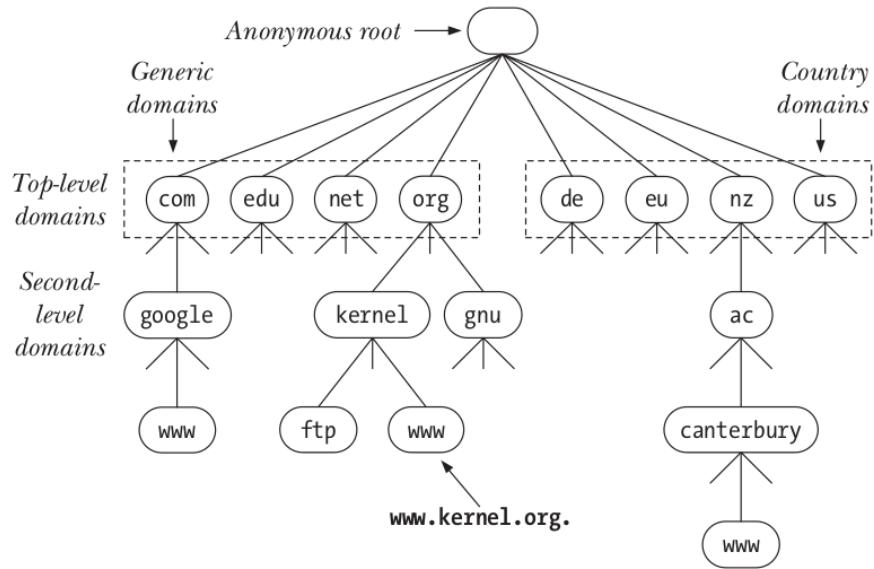


Domain Name System (DNS)

Il *Domain Name System (DNS)* è un sistema di directory distribuito e gerarchico

- Serve per identificare nodi di Internet tramite un *nome di dominio* anziché un indirizzo IP
- Permette la *conversione tra indirizzi IP e nomi di dominio*
 - Una specie di "*elenco telefonico dell'internet*"

Linux offre funzioni per usare il DNS in maniera semplice



I Socket

Definizione di Socket

I **Socket** uno strumento di *Inter-Process Communication* per scambiare dati tra diversi *nodi di rete*

Utilizzo simile alle *pipe* e alle *FIFO*

- Identificati da un *file descriptor*
- Vi si accede con le System Call **read** e **write**

A differenza di *pipe* e alle *FIFO*

- Conneggono *nodi diversi*
- Vengono *creati in maniera diversa* con System Call dedicate
- Questo è l'*unico modo* per comunicare tra *sistemi operativi!*

Tipologie di Socket

Esistono quattro tipologie di socket:

- **Stream Socket**: permettono comunicazione tramite *TCP (String byte)*
- **Datagram Socket**: permettono comunicazione tramite *UDP (Messaggi)*
- **Raw Socket**: permettono comunicazione tramite *pacchetti grezzi IP*
- **UNIX**: permettono comunicazione tra processi *di uno stesso nodo*

Sempre basati su modello **client/server**

Modello Client/Server

Per implementare il *modello client/server* abbiamo i *socket passivi* e *attivi*.

Un **Passive Socket** *aspetta connessioni* in arrivo

- Implementa un *server*

Un **Active Socket** è effettivamente connesso a un altro nodo

- Permette lo *scambio di dati*
- Usato da un *client* per comunicare col server
- Usato anche dal *server*, *dopo* aver accettato una nuova connessione

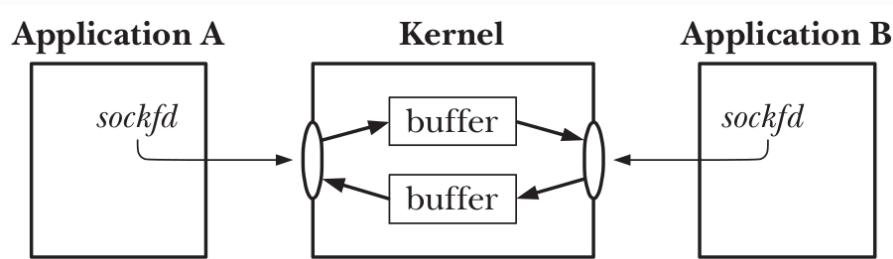
UNIX Socket

Comunicazione tra processi di uno stesso nodo

- Concettualmente *molto simili* a una *pipe* o *FIFO*

Differenza

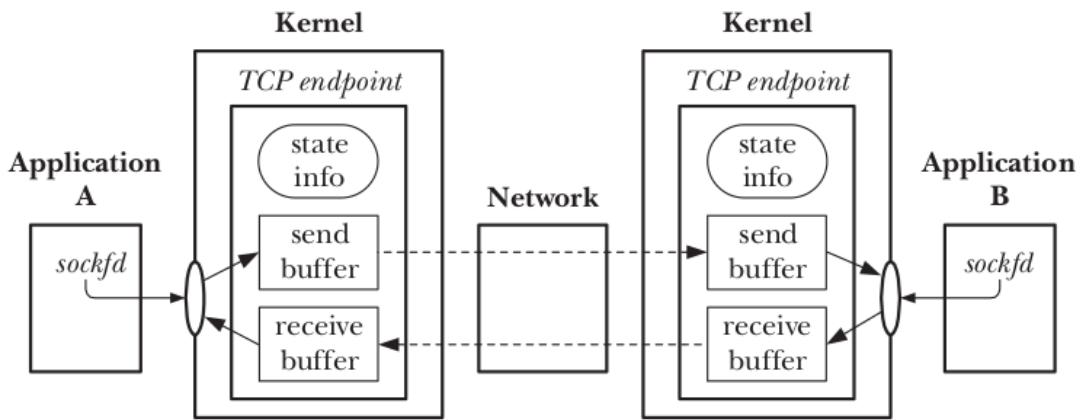
- Usano modello *client/server*
- Un *server* si mette in ascolto
- Un *client* contatta il server e inizia la comunicazione
- Sono *peer-to-peer*



Stream Socket

Comunicazione tramite *TCP*

- Servizio orientato alla *connessione*
 - Client e server comunicano tramite un *flusso di byte*
 - Molto affidabile! Circa 1/1000 dei pacchetti vengono persi, che vengono comunque ritrasmessi
- Simile a una *pipe* o *FIFO* tra *nodi diversi*



Datagram Socket

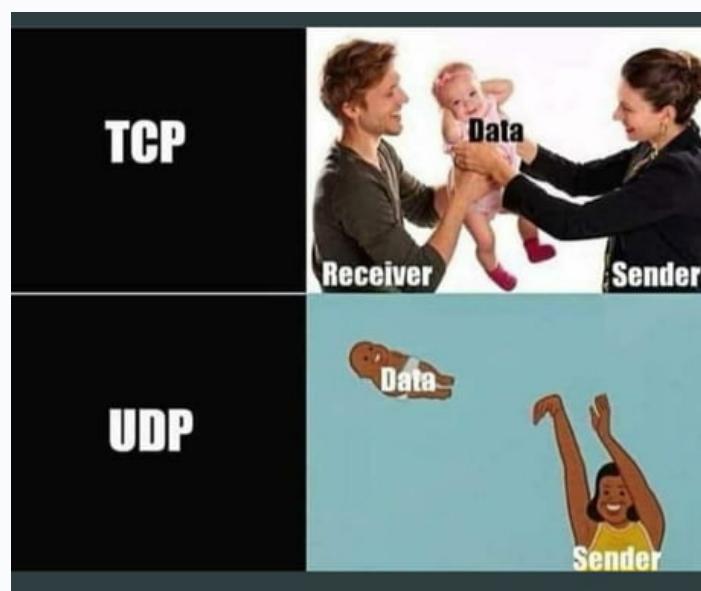
Comunicazione tramite UDP

- Client e server si scambiano *messaggi*
- Servizio non affidabile
 - Possibile perdita di pacchetti che non vengono ritrasmessi

Differenze tra Socket

Differenze:

- Datagram Socket:
 - Orientato ai *messaggi*
 - Non affidabile
- Stream Socket e UNIX socket:
 - Orientato allo *stream*
 - Affidabile



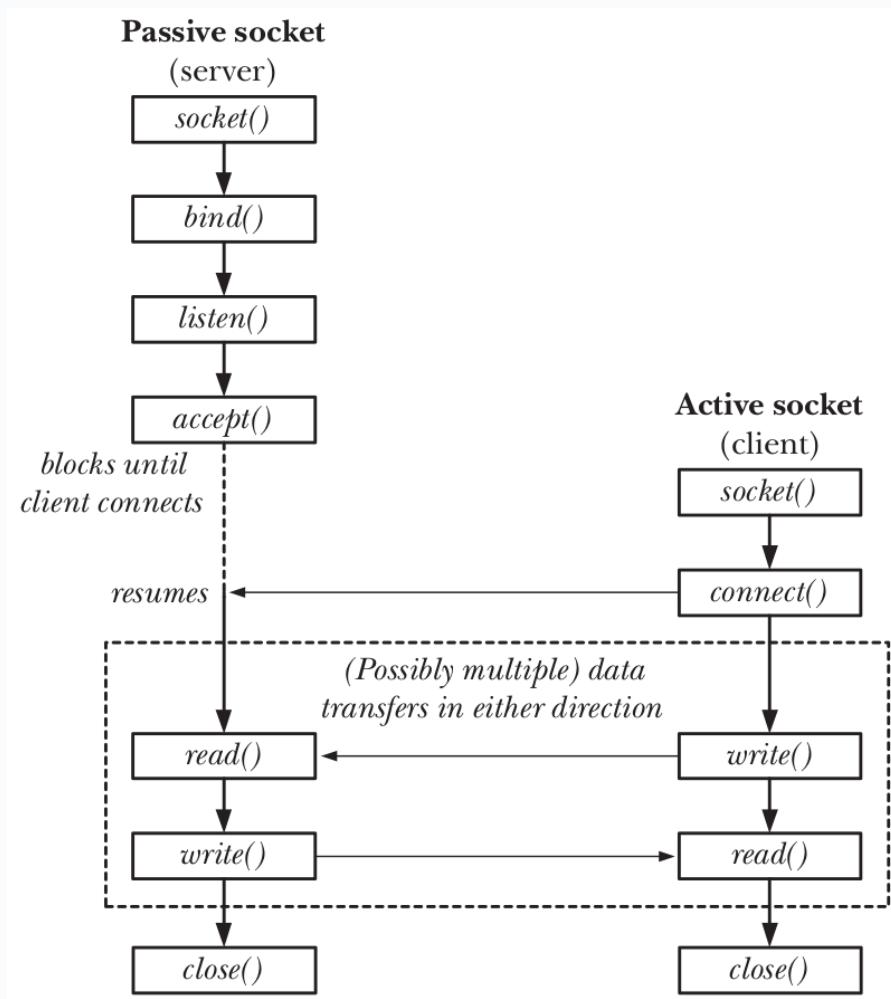
Funzioni e System Call per i Socket

I sistemi Linux/POSIX mettono a disposizione System Call per usare i socket

- Ogni socket è identificato da un File Descriptor
- Similmente ai file aperti o *FIFO* aperti, o *pipe*.
- Si effettua I/O usando le System Call **read** e **write**
 - Tranne che per i *Datagram Socket* (si usano **sendto** e **recvfrom**)
- Per creare un socket, si usano *System Call dedicate*
- Bisogna specificare *indirizzi IP e porte*
- *Attendere* che il kernel stabilisca la connessione

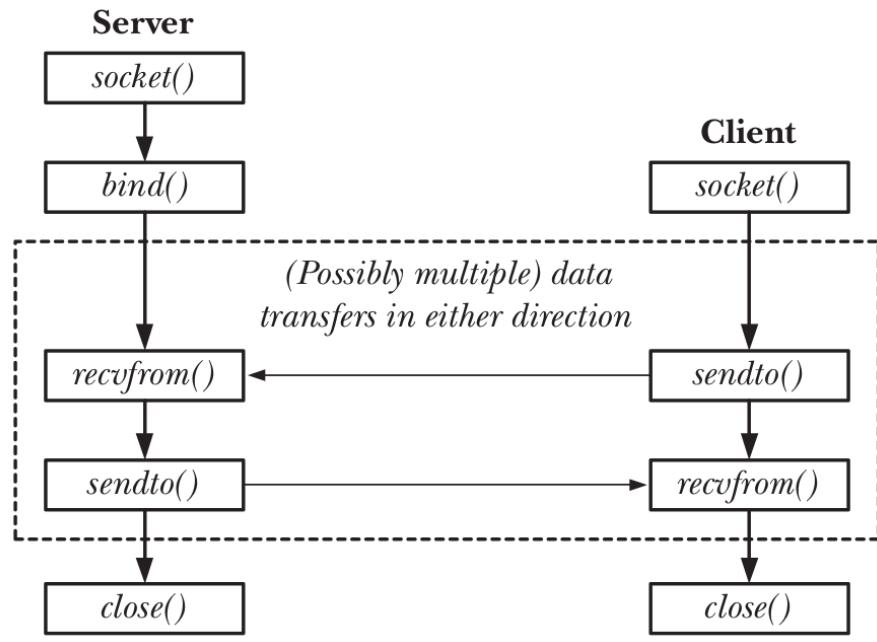
Stream Socket e UNIX Socket

- *Client* usa: **socket** e **connect**
- *Server* usa **socket**, **bind**, **listen** e **accept**
- *Entrambi* usano **read** **write** e **close**



Datagram Socket

- *Client* usa: **socket**
- *Server* usa **socket**, **bind**
- *Entrambi* usano **sendto** e **recvfrom** e **close**



Funzioni e System Call per i Socket

Noi vediamo in dettaglio *solo gli Stream Socket*

- Che utilizzano *TCP*
- Sono *affidabili*
- Orientati alla *connessione*
- Client e server comunicano tramite un stream di byte
- Semantica simile a *pipe*, ma *bidirezionale*

Nelle prossime slide, sono presentate le System Call, ipotizzando di creare uno *Stream Socket*

Creazione di un Socket

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Crea un socket. Gli argomenti **domain** e **protocol** ne *specificano la natura*.

- Argomento **protocol**, per noi sempre **0**

Ritorna il File Descriptor, se no –1.

Esempi:

- **Stream Socket** `int fd = socket(AF_INET, SOCK_STREAM, 0)`
- **UNIX Socket** `int fd = socket(AF_UNIX, SOCK_STREAM, 0);`
- **Datagram Socket** `int fd = socket(AF_INET, SOCK_DGRAM, 0))`

Struttura **sockaddr**

Prima di vedere le **System Call** nello specifico, vediamo una struttura dati importante per rappresentare *indirizzi IP e porte*.

```
C  
struct sockaddr {  
    sa_family_t sa_family; /* Address family (AF_* constant) */  
    char sa_data[14];     /* Socket address (size varies  
                           according to socket domain) */  
};
```

La **struct sockaddr** contiene un indirizzo IP, una porta o entrambi.

Deve essere *generica*: supportare protocolli potenzialmente diversi da suite TCP/IP

Il campo **sa_data** deve contenere gli indirizzi e le porte

Quando si usano socket con TCP/IP si utilizza la **struct sockaddr_in**

Viene usata in tutte le System Call che richiedono una **struct sockaddr**.

- Le System Call *solo generiche*
- Se noi usiamo TCP/IP, usiamo **struct sockaddr_in**

Lato Client

```
C  
#include <sys/socket.h>  
int connect(int sockfd, const struct sockaddr * addr, socklen_t addrlen );
```

Rende il socket **sockfd** *attivo* e si connette a indirizzo IP e porta specificati in **addr** e **addrlen**

Ritorna 0 in caso di successo, se no -1

La **struct sockaddr** contiene un indirizzo IP, una porta o entrambi

- Entrambi in questo caso

La **connect** è *bloccante* finché non viene stabilita la connessione (TCP).

Lato Server

1. *Trasformazione in passivo*

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr * addr, socklen_t addrlen);
```

Rende il socket **sockfd** passivo, ovvero *lo mette in ascolto sulla porta specificata* in **addr** e **addrlen**

Ritorna 0 in caso di successo, se no –1

La **addr** punta a una **struct sockaddr**, che sarà sempre di fatto una **struct sockaddr_in**:

- Contenente solo una porta in questo caso
2. *Attivazione di un socket passivo*

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Dopo che un socket **sockfd** è stato specificato come passivo (con **bind**), la **listen** lo *mette effettivamente in ascolto sulla porta* specificata.

Il parametro **backlog** determina *quante connessioni* in attesa *possono accodarsi* prima di essere servite. Di solito è un numero piccolo, tipo 5

Ritorna 0 in caso di successo, se no –1

3. *Accettazione di connessioni*

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr * addr, socklen_t * addrlen);
```

Attende che una connessione arrivi al socket passivo **sockfd**

- *Bloccante* finchè non arriva una connessione

Nel momento in cui arriva una nuova connessione:

- La funzione ritorna
- Il valore di ritorno è un *nuovo socket attivo* (ovvero l'indirizzo IP)
- In **addr** (e **addrlen**) è specificato l'indirizzo del client

Tipicamente messo in un *ciclo while infinito*.

Lettura e Scrittura su Socket

Un socket attivo viene creato:

- Direttamente da un client dopo che si è connessi
- In un server, ogni volta che la **accept** ritorna, e permette la comunicazione con un client

Un socket è *bidirezionale*. In caso di *Stream Socket*:

- Si effettua I/O con **read** e **write**, o volendo con le funzioni specifiche per i socket **send** e **recv**
- Un socket viene chiuso tramite la **close** (col pacchetto *FIN*, *SYN* nel caso di connessione stabilita)

Conversione di Indirizzi IP

Necessarie funzioni per convertire indirizzi IP in stringa e in formato binario su $4B = 32bit$

```
C  
char *inet_ntoa(struct in_addr in);  
int inet_aton(const char *cp, struct in_addr *inp);
```

IP in formato stringa specificato come **char ***

IP in formato binario specificato come **struct in_addr**

- Tipicamente si usa:

```
C  
struct sockaddr_in s;  
inet_aton("1.2.3.4", &s.in_addr);
```

Le varianti **inet_ntop** e **inet_pton** sono equivalenti, ma più moderne

Network Byte Order

Indirizzi IP e porte sono numeri interi su 32 e 16 bit.

Diverse architetture usano *convenzioni diverse* per l'ordine delle cifre

Necessario mettersi d'accordo quando si trasmettono via rete!

In rete si usa *Big Endian*, anche detto *Network Byte Order*: ovvero mettiamo *prima* le cifre più significative, poi alla fine le *cifre meno significative*.

	2-byte integer		4-byte integer			
	address N	address N + 1	address N	address N + 1	address N + 2	address N + 3
Big-endian byte order	1 (MSB)	0 (LSB)	3 (MSB)	2	1	0 (LSB)
Little-endian byte order	0 (LSB)	1 (MSB)	0 (LSB)	1	2	3 (MSB)

MSB = Most Significant Byte, LSB = Least Significant Byte

Diverse architetture usano convenzioni diverse. Per ovviare a questi problemi, abbiamo le seguenti funzioni per convertire i numeri *Little-Endian* in *Big-endian* (o *Big-Endian* a *Big-Endian*)

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Convertono da formato dell'architettura corrente (**h**) a Network Byte Order (**n**), numeri su 32bit (**l**) e su 16bit (**s**), e viceversa

Esempio:

```
uint16_t port_h = 12345;
uint16_t port_n = htons(port_h);
```

Modicare le Opzioni di un Socket

C

```
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname,
               void *restrict optval, socklen_t *restrict optlen);
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

Manipolano le opzioni per il socket **sockfd**.

Modificano comportamenti di default:

- Forzare la bind a una certa porta: **SO_REUSEADDR**
- Parametri di funzionamento di TCP
- Molte altre

Flusso Tipico per Socket Scream

LATO CLIENT.

C

```
// Creazione
int fd = socket(AF_INET, SOCK_STREAM, 0);

/* Connessione: specifica indirizzo IP
   e porta del server */
connect(fd,
        (struct sockaddr*)&address,
        sizeof(address)));

// Input/Output
write(fd, buffer, n);
read(fd, buffer, SIZE);

// Chiusura
close(fd);
```

LATO SERVER.

```

// Creazione
int fd = socket(AF_INET, SOCK_STREAM, 0);

// Bind: specifica porta
bind(fd, (struct sockaddr*)&address, sizeof(address));

// Listen: specifica lunghezza della coda in attesa
listen(fd, 3);

// Servizio ai client
while (1){

    /* Attesa di un client: ottiene indirizzo IP
       e porta del client */
    int active_fd = accept(fd,
                           (struct sockaddr*)&address,
                           (socklen_t*)&addrlen)
    );

    // Input/Output
    write(active_fd, buffer, n);
    read(active_fd, buffer, SIZE);

    // Chiusura
    close(active_fd);
}

// Chiusura
close(fd);

```

Risoluzione DNS

Esistono funzioni di libreria per effettuare risoluzioni DNS:

```

#include <netdb.h>
struct hostent *gethostbyname(const char *name);

```

Effettua una risoluzione DNS per il dominio **name**.

Ritorna una **struct hostent**, una struttura molto complessa che contiene i risultati della risoluzione

E' deprecata, ora si usa la simile **getaddrinfo**

Non vediamo in dettaglio

Esercizio sui Socket

Esercizio.

Il server 45.79.112.203 alla porta TCP 4242 offre un servizio di **echo**.

Se un client vi si connette e manda un messaggio, il server risponde con lo stesso messaggio.

Si crei un programma che si connette al suddetto endpoint, manda un messaggio e stampa la risposta un messaggio.

SOLUZIONE.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#define SIZE 1024
#define MESSAGGIO "Ciao Mondo!\n"

int main(int argc, char *argv[]){
    int fd, n;
    char buffer[SIZE];
    struct sockaddr_in address;

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_port = htons(4242);
    if (inet_aton("45.79.112.203", &address.sin_addr) < 0){
        perror("convert server ip failed");
        exit(EXIT_FAILURE);
    }

    if ((connect(fd, (struct sockaddr*)&address,sizeof(address)))< 0){
        perror("connect failed");
        exit(EXIT_FAILURE);
    }

    write(fd, MESSAGGIO, sizeof(MESSAGGIO));
    printf("Tramesso: %s\n", MESSAGGIO);

    n = read(fd, buffer, SIZE);
    buffer[n] = 0;
    printf("Ricevuto: %s\n", buffer);
    close(fd);

}
```

Networking in Linux

Interfacce di Rete

La gestione della rete cambia a seconda di distribuzione Linux/POSIX, ma ci sono dei concetti generali.

Ogni *interfaccia di rete* è identificata da un nome.

- Scheda Ethernet: **eth0** o **eno1**
- Scheda WiFi: **wifi0**
- Interfaccia di loopback: **lo**

Informazioni sulla Rete

1. **ifconfig** è il comando storico per avere informazioni.

In realtà è obsoleto, ora si usa il comando **ip addr**

Esempio:

SHELL

```
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    group default qlen 1000
        link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel
    state UP group default qlen 1000
        link/ether 2c:f0:5d:c3:7b:b5 brd ff:ff:ff:ff:ff:ff
        altname enp0s31f6
        inet 140.105.50.104/24 brd 140.105.50.255 scope global dynamic
            noproxyroute eno1
            valid_lft 101209sec preferred_lft 101209sec
        inet6 fe80::bf0b:ea7e:b8a9:d363/64 scope link noproxyroute
            valid_lft forever preferred_lft forever
```

2. **ip route**: Quando viene generato un pacchetto, il sistema usa la *routing table* per decidere *su quale interfaccia trasmetterlo*

```
$ ip route
default via 140.105.50.254 dev eno1 proto dhcp metric 100
140.105.50.0/24 dev eno1 proto kernel scope link src 140.105.50.104 metric 100
```

La routing table viene creata in automatico quando si configurano le interfacce di rete, inserendo indirizzo IP, netmask e default gateway.

Configurazione della Rete

Storicamente, rete configurata tramite file di configurazione.

- **/etc/network/interfaces**: indirizzo IP, subnet mask e default gateway
- **/etc/resolv.conf**: resolver DNS

Ora si usa il demone **Netplan**, che ha file di configurazione in **/etc/netplan/...**

YAML

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      addresses: [172.16.86.5/24]
      gateway4: 172.16.86.1
      nameservers:
        addresses: [8.8.8.8, 8.8.4.4]
```

Si applica la configurazione col comando:

```
netplan apply
```

I sistemi desktop hanno meccanismi di più alto livello per queste configuzioni

- Ubuntu Desktop ha **Network Manager** per configurare la rete tramite interfaccia grafica
- **Network Manager** scrive i file di configurazione per noi
- Attenzione a cambiare i file manualmente, rischio conflitto

Comandi Misti

Risoluzioni DNS:

`host <dominio>` o `dig <dominio>`

Troubleshooting:

`ping <destinazione>` e `traceroute <destinazione>`

Richieste HTTP:

`curl <URL>` o `wget <URL>` per scaricare pagine Web

Listare tutti i socket nel sistema:

Si usa il comando `netstat`, che ha molte opzioni:

- `-l`: Stampare solo socket passivi
- `-t`: Solo TCP
- `-p`: Stampare il PID e il nome del processo associato al socket

Utile per sapere se un programma server è attivo:

SHELL

```
$ netstat -nplt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
PID/Program name
tcp      0      0.0.0.0:22          0.0.0.0:*
                  LISTEN      1411/sshd
tcp      0      0.0.0.0:80          0.0.0.0:*
                  LISTEN      950293/nginx:
maste
tcp      0      0.0.0.0:443         0.0.0.0:*
                  LISTEN      950293/nginx:
maste
tcp      0      0.0.0.0:5000         0.0.0.0:*
                  LISTEN      4014584/docker-
prox
```

Creazione di Socket da Comando

Il comando `nc` permette di creare e usare in maniera semplice un socket da riga di comando

Client:

SHELL

```
nc <indirizzo> <porta>
```

Server:

```
nc -l <porta>
```

Quando il socket è connesso, si può scrivere e leggere nel socket usando il terminale

Esercizio: usare **nc** per scambiare messaggi tra due PC

Domande

Un server, per compiere pienamente le sue funzioni, usa:

- **Socket Passivi**
- **Socket Attivi**
- **Socket Passivi e Attivi**

Risposta: *Socket Passivi e Attivi*

Un client, per compiere pienamente le sue funzioni, usa:

- **Socket Passivi**
- **Socket Attivi**
- **Socket Passivi e Attivi**

Risposta: *Socket attivi*

Un Socket Stream è:

- **Monodirezionale**
- **Bidirezionale**

Risposta: *Bidirezionale*

E' possibile usare anche le funzioni **read** e **write** per effettuare I/O su Socket Stream?

- **Sì**
- **No**

Risposta: *Sì*

A cosa serve il comando **ifconfig**?

- **Configurare il comportamento di un socket**
- **Configurare le interfacce di rete**
- **Inviare pacchetti di configurazione**

Risposta: *Configurare le interfacce di rete*

u8-s3-package

Sistemi Operativi

Unità 8: Altri Argomenti

Gestione dei Pacchetti Software

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Perchè sono necessari
 2. Package Manager
 3. Pacchetti **deb** e Package Manager **apt**
 4. Package Manager **snap**
-

Perchè sono necessari: Utenti e Programmi

Appena installato, un SO contiene *solo i programmi di default*

- Utility per gestione del SO: **ls**, **ps**, **free**

Un utente vuole far girare le applicazioni che preferisce. Ha due opzioni:

1. *Scrive un programma, lo compila e lo esegue*
2. Usa un programma *scritto da qualcun altro*

L'opzione 2 è di gran lunga la più usata

Per usare un programma, ci sono diverse opzioni:

- Scaricare il *file binario del programma* ed eseguirlo
- Oppure un *installer*
 - In Windows: Scarico **installer.exe** e installo
- *Scaricare e compilare* il codice sorgente
- Usare un *Package Manager*
 - Come *AppStore* su iOS o Google Play di *Android*

Package Manager

Definizione di Package Manager

Un *Package Manager* è un software che si occupa di organizzare i software in uso in un sistema.

Ha l'obiettivo di:

- Permettere l'*installazione/rimozione* di pacchetti software
- *Verificare* che il software non sia corrotto e arrivi da fonti sicure
- *Gestire* eventuali *conflitti e dipendenze* tra pacchetti
- *Controllare gli aggiornamenti* dei software installati

Un Package Manager scarica i pacchetti da un *Repository* pubblico

Tipologie di Package Manager

Ci sono più tipologie di package manager. Ne elenchiamo tre.

Monolitici: l'applicazione e le tutte dipendenze *sono nello stesso pacchetto*

- Come MacOS o Docker
- **Vantaggi:** ogni pacchetto si porta dietro tutto ciò che gli serve
- **Svantaggi:** troppo spazio sprecato

A Pacchetti specifici: ogni pacchetto contiene *un singolo software/libreria*.

Quando si scarica l'applicazione, il Package Manager *controlla e scarica eventuali dipendenze*.

- Usato tipicamente in Linux: **apt**, **yum**
- **Vantaggi:** installazione veloce, no spreco di spazio su disco
- **Svantaggi:** gestire le dipendenze aumenta di *molto* la complessità!

Source-Based: il Package Manager *scarica e compila il codice sorgente* di ogni pacchetto

- Come **brew** usato su MacOS
- **Vantaggi:** programmi portabili su diverse architetture
- **Svantaggi:** molto lento, dato che la compilazione richiede tanto tempo

Package Manager Principali

Elenchiamo alcuni *package manager* specifici, molti di cui sono noti

1. Preinstallati nei SO

Fatti per l'utilizzo da parte di utenti non esperti

- Windows ⇒ Microsoft Store (precedentemente Windows Store)
- MacOS, iOS ⇒ AppStore
- Android ⇒ Google Play (precedentemente Android Market)

Caratteristiche:

- *Closed-source*: spesso il codice viene *offuscato* per evitare che il codice sorgente venga letto
- *Commerciali*: offrono applicazioni a pagamento

2. Per la programmazione

Ambienti specifici hanno un *Package Manager* dedicato

- Python ⇒ **pip**, **conda**
- Java ⇒ **maven**
- JavaScript ⇒ **npm**
- Go ⇒ **go get**

3. In Linux

Esistono due *formati* di *Package Binari*, ovvero che contengono software compilato:

- Pacchetti *Deb*: usati in Debian, Ubuntu
- Pacchetti *RPM*: usati in Red Hat, CentOS

I *Package Manager* installano *Package Binari* da repository pubblici:

- In Debian, Ubuntu: **apt**
- In Red Hat, CentOS: **yum**, ora rimpiazzato da **dnf**

Vengono usati tipicamente da riga di comando

4. In Mac OS

I software sono tipicamente in immagini *DMG*

- Formato per immagini di disco
- *Contengono tutte le dipendenze* (paradigma monolitico)
Si possono installare *Package Manager* aggiuntivi:
 - **port** o MacPort
 - **brew**

Entrambi scaricano i sorgenti e li compilano (source-based)

Operazioni con Package Manager

Ogni *Package Manager* ha comandi diversi.

- Tipicamente si usano da riga di comando.
- Ma esistono interfacce grafiche per semplificare l'uso

Azioni comuni:

- **install**
- **remove**
- **update**
- **view dependencies**

Pacchetti **deb** e Package Manager **apt**

Approfondiremo il discorso, per quanto riguarda i *package Manager* su *Linux* (in particolare nei sistemi basati su *Debian* e *Ubuntu*)

Nei sistemi basati su Debian e Ubuntu si usa il formato *Deb*. Sono:

- **Package atomici:** ognuno contiene *un singolo software*
- **Binari compilati:** si scaricano programmi già compilati *per la propria architettura*

Un pacchetto *Deb* è un archivio compresso contenente:

- I *file binari*
- *Metadati*: nome, versione
- Lista delle *dipendenze*

- Opzionalmente:
 - File di configurazione
 - Script da eseguire per *installazione* o *disinstallazione*
 - Firma digitale GPG (per evitare eventuali fake da parte di malfattori)

Installazione Manuale dei Pacchetti **deb**

Se scarico pacchetti **.deb** per fatti miei, posso gestirli *manualmente*.

Il comando **dpkg** permette di gestire pacchetti *Deb*

- Installazione: **dpkg -i <file.deb>**
- Informazioni su un pacchetto: **dpkg -l <file.deb>**
- Disinstallazione: **dpkg -r <nome-pacchetto>**
- Lista di pacchetti installati: **dpkg -l**
- Lista dei file installati da un pacchetto installato: **dpkg -L <nome-pacchetto>**

dpkg è un tool *di basso livello*

- Installa pacchetti da file **deb**
- Non risolve le dipendenze
- Non pratico da usare

Devo fare tutto a mano... c'è un modo per evitarsi questi problemi? Ma certo!

Solitamente non si usa **dpkg** direttamente, ma *Advanced package tool* (**apt**): risolve i problemi di cui sopra

Advanced Package Tool **apt**

1. Repository

apt scarica package da *repository online* (della società privata *Canonical*):

- Lista ottenuta dal file: **/etc/apt/sources.list** e da tutti i file nella cartella **/etc/apt/sources.list.d/**
 - Repository *pre-definiti* quando si installa il SO
- Si possono aggiungere repository per package non presenti di default:
 - E.g., *Chrome*, *Dropbox*
- Un repository è identificato da un *URL* e ha dei *tag*
 - Esempio: **deb http://it.archive.ubuntu.com/ubuntu/ focal main restricted**

2. Comandi

Per installare pacchetti con **apt** si usa il comando **apt** o **apt-get** (più *vecchio* ma *analogico*)

- Installazione: **apt install <nome-pacchetto>**
- Disinstallazione: **apt remove <nome-pacchetto>**
- Aggiornamento delle *liste* di pacchetti disponibili: **apt update**
 - **NOTA!** Questo comando *non*
- Ricerca di pacchetti nei repository: **apt-cache search**

3. Dipendenze

Ogni volta che si installa un pacchetto, **apt** risolve le dipendenze (o almeno ci prova)

- Installa in automatico le librerie i software da cui dipende
- Problema complesso: generato un *grafo delle dipendenze*

Possono nascere *conflitti*, per problemi di versione: ad esempio voglio scaricare **pacchetto1** che richiede **pacchetto2** versione 2.0, ma ho l'ho già installata in versione 1.0 c'è un problema

The following packages have unmet dependencies:

package1 : Depends: package2 (> 1.8) but 1.7.5-1ubuntu1 is to be installed

Tipicamente i pacchetti nei repository di sistema *non hanno questi problemi*

4. Risoluzione dei problemi

In caso di *dipendenze non risolte o altri problemi*, si può dire ad **apt** di fare pulizia

- **apt autoclean**: elimina i pacchetti **.deb** scaricati relativi a versioni vecchie
 - **Nota:** rimuove *l'archivio Deb*, che è inutile dopo installazione, ma viene tenuto in *cache*. Non rimuove l'*installazione*
- **apt clean**: elimina tutti i pacchetti **.deb** in *cache*
- **apt autoremove**: *disinstalla i pacchetti orfani*, ovvero dipendenze installate per l'installazione di un'applicazione che poi *rimuovete*, così non sono più necessarie

Da **apt** a **snap**

apt funziona molto bene ed è usato con successo nella maggior parte dei sistemi Linux

- *Economizza lo spazio*: i pacchetti hanno *dipendenze*
- Le dipendenze sono *installate e condivise* da tutto il sistema (come visto, ciò è complesso da fare)

Nei sistemi Ubuntu, ora a fianco di **apt** si usa anche **Snap** (monolitico)

- Installato di default su Ubuntu, installabile anche su altre distribuzioni

Package Manager **snap**

Snap installa pacchetti *self-contained*

- Contengono il *programma* e *tutte le dipendenze*: librerie, altro software
- Di fatto contengono un *File System in formato SquashFS* (ovvero il software con le sue dipendenze)
- Le applicazioni girano in una *SandBox*, con limitato accesso al sistema
- *Concettualmente* simile a un *container*
 - Simile a Docker, ma pensato anche per utenti non esperti

- Evita casini di accesso, di vulnerabilità, eccetera...

Vantaggi:

- Risolve problemi di *dipendenze*
- Maggiore sicurezza grazie a SandBox

Svantaggi:

- Si usa *più spazio su disco*
- Pacchetti sono *più grandi da scaricare dalla rete*
- *Più pesante per il sistema:*
 - Il File System di un pacchetto viene *montato* ad ogni avvio

Per riassumere: ho molti vantaggi, ma pagando un prezzo salato...

Domande

A cosa serve un Package Manager?

- A instradare i pacchetti di rete
- A installare i pacchetti software da repository pubblici
- A installare i programmi creati dall'utente

Risposta: A installare i pacchetti software da repository pubblici

Un pacchetto Deb contiene le tutte sue dipendenze:

- Si
- No

Risposta: No

Un pacchetto Deb contiene i file sorgenti:

- Si
- No

Risposta: No

Il Package Manager **apt** installa le dipendenze:

- Automaticamente
- Mai
- Su richeista

Risposta: Automaticamente

Un pacchetto Snap contiene le tutte sue dipendenze:

- Si
- No

Risposta: Sì