

# Calcolatori elettronici: Software di base

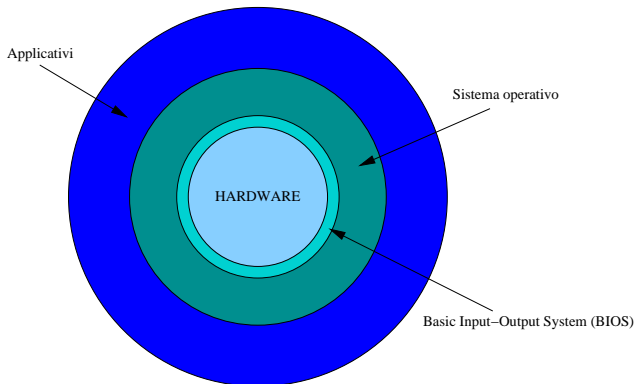
Andrea Passerini  
passerini@dsi.unifi.it

Conoscenze informatiche e relazionali  
Corso di laurea in Scienze dell'Ingegneria Edile

# Sistema operativo (SO)

- E' un'*infrastruttura software* che si pone come *interfaccia* tra l'infrastruttura hardware e l'utente (o il software applicativo).
- Consente di utilizzare le risorse del sistema informatico senza preoccuparsi delle loro caratteristiche fisiche (e.g. tipo di processore, dimensione del bus dati, tipo di schermo)
- Consente l'utilizzo concorrente del sistema da parte di più programmi ed eventualmente utenti (sistemi multiutente) rendendo trasparente la gestione della concorrenza nell'utilizzo delle risorse.
- Virtualizza le caratteristiche dell'hardware, offrendo una visione del sistema come *macchina astratta* (o *virtuale*).

# Sistema operativo



# Funzioni del sistema operativo

- esecuzione applicazioni gestisce il caricamento del programma (istruzioni e dati) in memoria centrale e la sua esecuzione da parte della CPU
- accesso dispositivi I/O maschera i dettagli di basso livello del controllo delle periferiche, consentendo alle applicazioni di operare in termini di istruzioni astratte (lettura, scrittura).
- archiviazione dati e programmi fornisce un'organizzazione logica ai dati nelle memorie di massa (e.g. hard disk) sotto forma di cartelle (*directory*) e file e gestisce le relative operazioni di I/O.

# Funzioni del sistema operativo

- controllo di accesso introduce meccanismi di protezione e risoluzione dei conflitti nei sistemi che condividono risorse tra più applicazioni ed utenti.
- gestione malfunzionamenti rileva e tenta di risolvere in maniera il più trasparente possibile eventuali malfunzionamenti dovuti a guasti hardware o operazioni scorrette compiute da un'applicazione.

# Modello stratificato

- un sistema operativo è tipicamente organizzato in modo stratificato (a buccia di cipolla)
- lo strato più esterno fa riferimento alle funzionalità messe a disposizione dagli strati più interni
- il *kernel* (o nucleo) è lo strato più interno e fa riferimento diretto al BIOS
- l'approccio modulare allo sviluppo dei sistemi operativi tende a ridurre le funzionalità del kernel al minimo indispensabile ed integrare le funzionalità ulteriori tramite *moduli* (e.g. per la gestione delle periferiche).

# Componenti del sistema operativo

- gestione dei processi gestisce i programmi in esecuzione (*processi*) pianificando il loro utilizzo della CPU
- gestione della memoria controlla l'allocazione della memoria ai diversi programmi in esecuzione, garantendo a ciascuno l'accesso ad un'area riservata
- gestione delle periferiche garantisce l'accesso ai dispositivi di I/O mascherandone i dettagli fisici (tramite i *drivers*) e risolvendo gli eventuali conflitti per richieste concorrenti
- gestione dei file (*file system*) gestisce archiviazione e recupero dei dati nelle memorie di massa
- interprete di comandi si interfaccia direttamente agli utenti permettendo di accedere alle funzionalità del sistema (e.g. *shell* UNIX o interfacce utente grafiche)

# Processi e programmi

- un processo (*task*) è un programma in esecuzione sul calcolatore
- un programma è un oggetto statico (una sequenza di istruzioni)
- un processo è dinamico nel senso che è dotato di uno stato interno che cambia nel tempo
- Lo stato di un processo è formato dai valori dei dati contenuti in memoria e nei registri della CPU (in particolare il program counter che contiene l'indirizzo della prossima istruzione da eseguire)
- Lo stesso programma può essere associato a più processi distinti (detti *figli*)

## Elaborazione parallela

- L'architettura di Von Neumann si basa sul principio di esecuzione sequenziale di operazioni.
- Per molti problemi reali è facile trovare situazioni in cui certi passi possano essere eseguiti in parallelo
- Si parla di elaborazione parallela a livello di:
  - dati nel caso in cui si debba svolgere la stessa operazione indipendentemente su un insieme di dati (e.g. l'aggiornamento dei pixel in un'immagine)
  - istruzioni per istruzioni indipendenti da svolgere su dati distinti
  - processi per processi diversi che potrebbero essere in esecuzione allo stesso momento (e.g. usare un programma multimediale per ascoltare musica mentre si utilizza un programma di videoscrittura)

# Parallelismo e Multitasking

- Il parallelismo relativamente a dati e istruzioni è possibile solo utilizzando architetture di elaborazione parallela, basate su:
  - La disponibilità di più unità di elaborazione
  - La presenza di *pipeline* che funzionano come catene di montaggio per istruzioni (non realizzano parallelismo sui dati)
- Il parallelismo a livello di processo può essere gestito direttamente dal sistema operativo (*multitasking*)

# Multitasking

- I tempi di esecuzione di tipi diversi di istruzioni sono molto diversi.
- Le istruzioni aritmetico logiche sono ordini di grandezza più veloci delle istruzioni di I/O.
- Le istruzioni di I/O hanno tempi spesso non prevedibili a priori (e.g. la pressione di un tasto da parte dell'utente)
- La maggior parte dei programmi interattivi sono del tipo *I/O bound*, ossia impiegano la maggior parte del loro tempo in operazioni di I/O, intervallate da brevi periodi di elaborazione.
- Sarebbe assolutamente inefficiente che nel momento in cui il processo attualmente in esecuzione dovesse fare un'operazione di I/O, la CPU aspettasse la fine di tale operazione rimanendo inoperosa.

# Multitasking

- In un calcolatore sono attivi (ossia caricati in memoria centrale) più processi contemporaneamente
- In ogni istante un solo processo si trova realmente in esecuzione (la CPU può eseguire una sola istruzione alla volta)
- Gli altri processi si possono trovare in uno dei due stati:
  - pronto in grado di essere eseguito non appena la CPU diviene disponibile (una certa politica decide quale dei processi pronti mandare in esecuzione)
  - in attesa non in grado di essere eseguito poiché in attesa del verificarsi di un evento esterno (e.g. la pressione di un tasto della tastiera) per passare allo stato di pronto

# Interruzioni

- Se un processo A in esecuzione ha bisogno di accedere ad una periferica (e.g. al disco fisso) passa del tempo prima che la periferica sia effettivamente in grado di comunicare i dati
- In questo caso il processo genera un *interrupt interno* ed il controllo passa al kernel, che mette A nello stato di attesa
- Il kernel manda in esecuzione un altro processo B tra quelli che si trovano nello stato pronto.
- Quando la periferica è pronta, viene generato un *interrupt esterno* (hardware) avvisando il SO che il processo A può essere risvegliato

## Interruzioni interne

- L'interruzione interna avviene per mezzo di una particolare chiamata al sistema operativo da parte del processo (*supervisor call*)
- In assenza di tale meccanismo la CPU dovrebbe rimanere in un ciclo "idle" attendendo la risposta della periferica, spreco di tempo.
- E' importante che il SO salvi il contesto del processo in esecuzione (contenuto dei registri) prima di sospenderlo, altrimenti non sarebbe possibile riportarlo correttamente in esecuzione
- Il contesto viene salvato in un'area speciale di memoria (descrittore del processo)

## Interruzioni interne

- Dopo il salvataggio del contesto il SO sceglie uno dei processi pronti e lo manda in esecuzione, caricando il suo contesto nei registri della CPU
- In particolare, viene caricato il valore del Program Counter che permette di far ripartire l'esecuzione del processo dall'istruzione successiva all'ultima precedentemente eseguita
- L'attività di sospendere un processo, salvarne il contesto, scegliere un altro processo ed attivarlo si chiama *context-switching*
- Il componente del kernel che si incarica di queste operazioni si chiama *scheduler*

## Interruzioni esterne

- Quando la periferica termina la sua operazione essa genera un interrupt esterno che si verifica in modo asincrono rispetto all'esecuzione delle istruzioni nella CPU
- Al verificarsi di questo evento il processo in esecuzione deve essere sospeso (salvandone il contesto) per gestire l'interrupt
- All'interrupt è associato un numero intero  $N$  che lo identifica e che può essere letto dalla CPU sul bus
- All'arrivo del segnale di interrupt la CPU modifica il program counter con un valore calcolato sulla base di  $N$ , corrispondente all'indirizzo in memoria di una porzione speciale di codice detta *routine di servizio dell'interrupt*

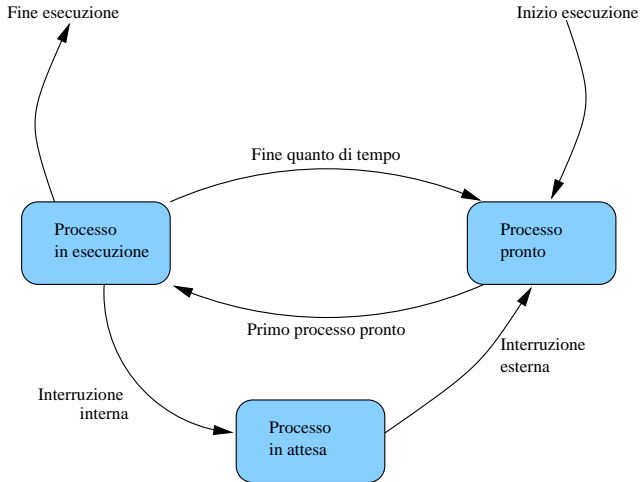
## Interruzioni esterne

- La routine di servizio dell'interrupt provvede a raccogliere i dati forniti dalla periferica oppure a continuare l'invio di altri dati alla periferica
- Terminata la routine di servizio, il processo che aveva generato la richiesta di I/O viene spostato dallo stato di attesa allo stato di pronto (se la routine non implica una operazione di I/O)
- Il controllo torna al kernel che decide quale processo far tornare in esecuzione

# Scheduling

- Un processo può essere sospeso anche perché è scaduto un certo intervallo di tempo ad esso assegnato
- In questo modo si garantisce che tutti i processi in memoria possano usare la CPU in maniera paritaria, evitando monopolizzazioni da parte di singoli processi
- Il processo in esecuzione viene sospeso e messo nello stato di pronto ed un altro processo viene messo in stato di esecuzione
- La scelta su quale processo eseguire tra quelli pronti viene effettuata dallo scheduler
- Tipicamente si usa una coda con priorità (certi processi possono avere priorità maggiore di altri)
- UNIX, ad esempio, assegna maggiore priorità ai processi interattivi

# Grafo delle transizioni



# Politica di gestione

## Round Robin

- Il kernel ha una coda dei processi pronti ed assegna a ciascuno un quanto di tempo  $T$
- La coda viene gestita in modo *FIFO* (*First In First Out*)
- Il primo processo in coda viene messo nello stato di esecuzione per un tempo  $T$  e poi interrotto
- Il quanto di tempo  $T$  deve essere ampio rispetto al tempo di context-switching
- Nella coda entrano anche i processi che dallo stato di attesa vanno in stato di pronto per effetto di un'interruzione esterna
- A seconda dei sistemi è possibile assegnare una priorità ai processi (e.g. con il comando `nice` in ambiente UNIX)

# Foreground e Background

- Relativamente all'interazione con l'utente, un processo può essere in due modalità:

in foreground quando il processo è abilitato all'interazione con l'utente attraverso i dispositivi di I/O quali video e tastiera

in background quando, pur essendo attivo, il processo non è in grado almeno temporaneamente di interagire direttamente con l'utente

# Foreground e Background

- La maggior parte dei processi generati dai programmi interattivi è fatta per lavorare in foreground (e.g. la finestra di un programma di videoscrittura)
- Un utente è in genere in grado di interagire con un solo processo alla volta, per cui gli altri processi si troveranno tipicamente in modalità background.
- Nel sistemi con interfaccia utente grafica, si associa in genere ad ogni processo una finestra sullo schermo, ed una sola finestra è attiva (in foreground) in un certo istante, mentre è possibile attivare un'altra finestra (disattivando automaticamente quella precedentemente attiva) tramite il mouse o con una combinazione di tasti.

# Demoni e agenti

- Molti dei processi relativi alle funzioni interne del sistema operativo vengono:
  - attivati automaticamente dopo l'accensione ed inizializzazione del calcolatore
  - eseguiti in background
- Alcuni di essi, chiamati *demoni* sotto UNIX o *agenti* in altri sistemi, rimangono in attesa che uno specifico evento li mandi in esecuzione,
- Esempi di demoni sono lo spooler di stampa, che gestisce la coda dei processi di stampa, ed i processi che distribuiscono la posta elettronica agli utenti del sistema.

# Gestione della memoria centrale

- Il sistema di gestione della memoria deve essere in grado permettere ad un numero elevato di processi di risiedere in memoria:
  - evitando conflitti tra i processi (e.g. evitando che un processo scriva dei dati nell'area di memoria contenente i dati di un altro processo)
  - ovviando alle limitazioni imposte dalla dimensione della memoria centrale

# Gestione della memoria centrale: soluzioni

- segmentazione della memoria permettere di dividere area istruzioni da area dati, condividendo eventualmente la prima tra processi generati dallo stesso programma
- rilocazione del codice permette di caricare un programma a partire da un qualunque indirizzo di memoria
- swapping, paginazione e memoria virtuale permette di aumentare lo spazio di memoria disponibile utilizzando la memoria di massa

# Segmentazione della memoria

area codice

vi risiedono le istruzioni macchina del programma.

- Se più versioni dello stesso programma sono in esecuzione simultaneamente l'area codice può essere condivisa.
- I programmi che fanno pesante uso dell'interfaccia grafica si basano tipicamente su librerie grafiche che vengono collegate dinamicamente al processo corrispondente, e possono essere condivise tra processi di programmi diversi che si basano sulle stesse librerie.

# Segmentazione della memoria

## area dati

E' l'area privata di ciascun processo contenente i dati su cui opera.

- Se un processo tenta di accedere dati al di fuori della sua area dati il kernel reagisce terminando il processo.
- In alcuni casi è possibile che specifiche aree dati vengano condivise tra più processi (*shared memory*)

## Rilocazione del codice

- Abbiamo visto che nel codice macchina ci sono istruzioni di salto e riferimenti a locazioni di memoria contenenti dati (e.g. `j 158h`)
- I valori degli indirizzi di memoria (per come abbiamo visto le cose) sono assoluti (in genere a partire da 0)
- Questo non funziona in un sistema multitask: infatti l'indirizzo a partire dal quale viene caricato in memoria il programma può variare a seconda delle condizioni di carico della macchina. Di solito il SO sceglie la prima area di memoria libera per caricare il programma.
- Dunque gli indirizzi di salto e di riferimento ai dati devono essere relativi

## Rilocazione del codice

- Invece di usare indirizzi fisici (assoluti) il programma viene compilato usando indirizzi logici (rilocabili)
- Lo *spiazzamento* è la differenza tra l'indirizzo a partire dal quale verrà caricato il programma ed il valore (solitamente 0) a partire dal quale sono stati calcolati gli indirizzi logici.
- Gli indirizzi fisici di un processo vengono calcolati tramite la *rilocazione* che consiste nel sommare all'indirizzo logico lo spiazzamento.

# Rilocazione del codice

Ci sono due approcci alla rilocazione:

statica il kernel trasforma indirizzi logici (contenuti nel programma) in indirizzi fisici associati al processo sommandovi lo spiazzamento al momento di caricare il programma in memoria

dinamica la CPU dispone di un registro speciale (*registro base*) sul quale viene scritto dal kernel lo spiazzamento; gli indirizzi fisici vengono calcolati dinamicamente sommando l'indirizzo logico al contenuto del registro base

# Swapping

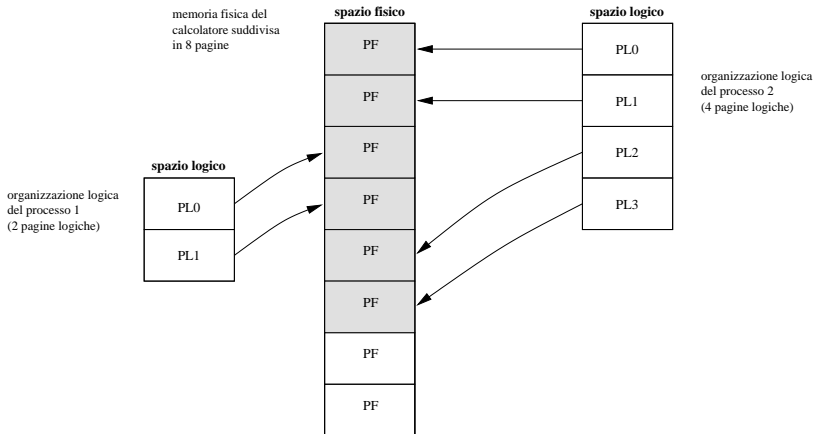
- In un certo momento è possibile che un processo in esecuzione richieda ulteriore memoria per caricare dati, o che si debba attivare un nuovo processo, e che la memoria non contenga spazio libero a sufficienza.
- E' possibile ovviare a questa situazione trasferendo parte del contenuto della memoria in un'area apposita della memoria di massa (detta *area di swap*), liberando lo spazio necessario.
- Tale tecnica, detta *swapping*, si applica trasferendo su disco i dati relativi a processi in stato di attesa o di pronto.
- Per motivi di efficienza vengono messi in esecuzione solo processi residenti in memoria centrale.
- Un processo del sistema operativo si occupa di trasferire alcuni processi in stato di pronto o attesa da memoria centrale ad area di swap e nel contempo caricare dei processi pronti da disco a memoria.

# Paginazione

- Si ottiene suddividendo il programma in sezioni di dimensioni fisse ed uguali tra loro, dette *pagine logiche*.
- La memoria viene corrispondentemente organizzata in *pagine fisiche* della stessa dimensione delle pagine logiche.
- Tale approccio permette di:
  - estendere la dimensione di un processo utilizzando zone di memoria non contigue.
  - tenere in memoria solo la porzione del programma attualmente utilizzata (servendosi dell'area di *swap* per il resto)
- In pratica lo spazio totale occupato dai processi può essere superiore alla dimensione della memoria centrale, per cui si parla di *memoria virtuale* e *spazio di indirizzamento virtuale*

# Paginazione

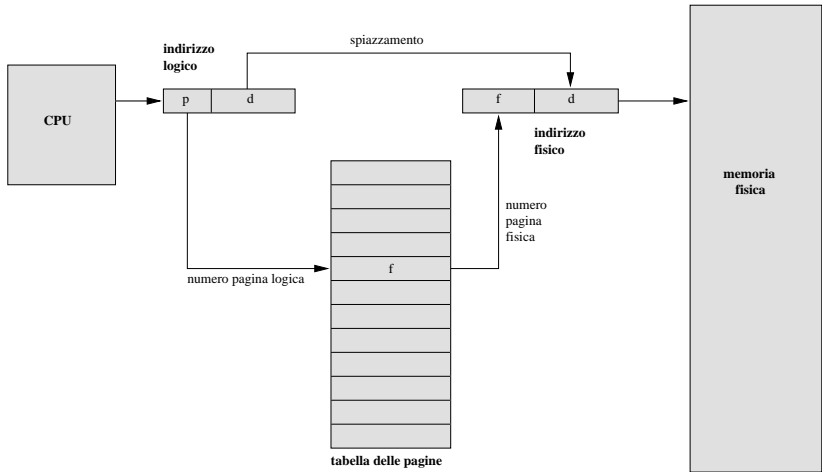
Esempio di corrispondenza tra pagine logiche contigue e pagine fisiche non contigue



## Calcolo degli indirizzi fisici

- E' in genere necessario un dispositivo hardware aggiuntivo (*Memory Management Unit*, MMU) in grado di convertire indirizzi logici in indirizzi fisici.
- Ogni indirizzo logico è suddiviso in due parti:
  - un *numero di pagina* ( $p$ ) serve da indice per la *tabella delle pagine* che contiene l'indirizzo base per ogni pagina della memoria fisica.
  - un *offset di pagina* ( $d$ ) viene associato all'indirizzo base di pagina per definire l'indirizzo fisico.

# Calcolo degli indirizzi fisici



# Page fault

- La MMU controlla il verificarsi dell'evento di *page fault*, che si verifica quando l'indirizzo fisico calcolato corrisponde ad una pagina non in memoria.
- Il page fault viene affrontato liberando lo spazio necessario in memoria e caricando la pagina richiesta.
- Un opportuno algoritmo di sostituzione delle pagine decide quali pagine scaricare su disco per liberare spazio, mirando a minimizzare il numero di page fault.

# Multiutenza

- I SO più evoluti (UNIX, NT) consentono la gestione di più utenti, permettendo accesso simultaneo al sistema e garantendo la protezione dei dati
- Utenti diversi possono avere permessi diversi (ad esempio il permesso di eseguire comandi speciali)
- Esiste sempre un utente privilegiato che corrisponde all'amministratore del sistema (root sotto UNIX, administrator sotto NT).
- Il kernel mantiene informazioni sull'utente che ha lanciato un processo o che accede ad una risorsa del sistema.

# Gestione delle periferiche

- Il SO fornisce le funzionalità che consentono di effettuare operazioni di lettura e scrittura con le periferiche mediante comandi indipendenti dalla struttura hardware delle periferiche.
- Tali comandi *ad alto livello* utilizzano meccanismi di gestione di basso livello quali:
  - controller dispositivi hardware che effettuano a livello fisico le operazioni di trasferimento dati con le periferiche
  - driver programmi software per la gestione delle periferiche. Sono parte del sistema operativo anche se spesso realizzati dai produttori delle periferiche o da sviluppatori indipendenti.

# Driver

- Hanno lo scopo di mascherare le caratteristiche specifiche dei controller.
- Forniscono un insieme di primitive (comandi) ad alto livello per la gestione delle operazioni di I/O utilizzabili dai programmi applicativi e dagli utenti
- Si incaricano anche di ripetere più volte un'operazione di I/O non andata a buon fine, segnalando eventualmente il tipo di malfunzionamento
- Permettono di virtualizzare la presenza di più periferiche intrinsecamente non condivisibili, tramite la tecnica dello *spooling*
- I sistemi operativi più recenti hanno funzionalità dette di *Plug&Play* che permettono di configurare automaticamente il driver corretto per la nuova periferica collegata (che deve essere concepita per farsi *riconoscere*)

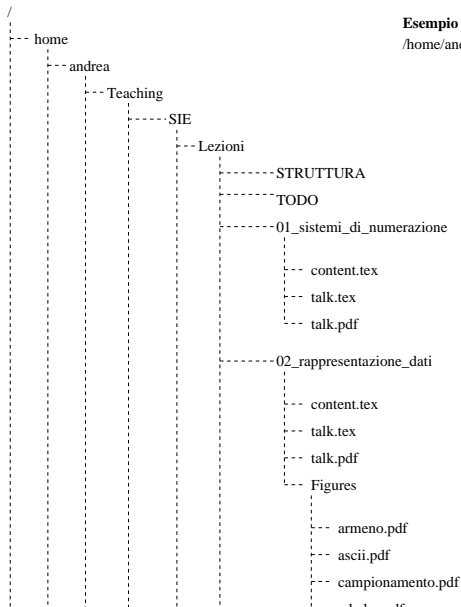
## Esempio: driver di stampa

- Riceve dai processi i file da stampare
- Accoda i file in una apposita directory (coda) di spooling
- Invia uno alla volta alla stampante i file contenuti nella coda di spooling
- Permette di cancellare file non ancora stampati rimuovendoli dalla coda di spooling.

# File System

- Scopi:
  - creare, leggere e scrivere files
  - collocare un file in uno spazio opportuno del disco (mascherando l'organizzazione fisica del disco in tracce e settori)
  - organizzare gerarchicamente i files
- I file sono inclusi all'interno di cartelle (directory) che generalmente sono organizzate ad albero (con radice)

# Esempio di struttura ad albero (UNIX)



## Esempio di pathname:

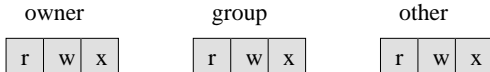
/home/andrea/Teaching/SIE/Lezioni/02\_rappresentazione\_dati/talk.pdf

# Funzioni del File System

- Creazione di un file o di una directory
- Elencazione dei files in una directory
- Cambiamento di directory corrente
- Copia di files o concatenamento
- Modifica del nome di un file
- Recupero della data di creazione, modifica, accesso
- Protezione

# Protezione nel File System (UNIX)

- Si distingue tra:
  - proprietario del file
  - utente appartenente allo stesso gruppo del proprietario
  - altro utente
- Si distinguono i permessi di:
  - scrittura
  - lettura
  - esecuzione
- In totale 9 flags specificano i permessi di un file:



# Esempio (UNIX)

```
[andrea@praha 08_software]$ ls -lha
total 220K
drwxr-xr-x  3 andrea ai  4.0K 2007-02-25 17:14 .
drwxr-xr-x 12 andrea ai  4.0K 2007-02-25 17:12 ..
-rw-r--r--  1 andrea ai   35K 2007-02-25 17:12 content.tex
drwxr-xr-x  2 andrea ai  4.0K 2007-02-25 17:12 Figures
-rw-r--r--  1 andrea ai 161K 2007-02-25 17:14 talk.pdf
-rw-r--r--  1 andrea ai  2.6K 2007-02-25 17:14 talk.tex
```

# Significato dei permessi

Il significato dei permessi differisce se si tratta di file o di directory

file

lettura è possibile leggere il contenuto del file

scrittura è possibile modificare il contenuto del file

esecuzione è possibile eseguire il file (nel caso in cui il file contenga un programma)

directory

lettura è possibile recuperare l'elenco dei file contenuti nella directory

scrittura è possibile creare un nuovo file nella directory

esecuzione è possibile entrare nella directory o attraversarla per entrare in una sua sottodirectory

# Shells

- Una *shell* è un interprete di comandi che serve da interfaccia tra l'utente ed il SO
- UNIX: `sh`, `csh`, `bash`, `tcsh`, etc.
- DOS: `command`
- Windows: "Prompt dei comandi"
- Nei sistemi privi di interfaccia utente grafica, dopo l'avvio all'utente si presenta un'interfaccia testuale a riga di comando (*Command Line Interface* o CLI) sulla quale è possibile scrivere direttamente i comandi di shell.
- Nei sistemi con interfaccia utente grafica, è sempre possibile ottenere un'interfaccia a riga di comando nella quale inserire comandi di shell (il terminale).

## Esempi di comandi di shell UNIX(dos/windows)

- `ls (dir)` elenca il contenuto di una directory
- `cd (cd)` cambia la directory corrente
- `cp (copy)` copia un file in un altro
- `mv (move)` sposta un file in un altro
- `rm (del)` cancella un file
- `mkdir (md)` crea una directory
- `cat (type)` visualizza il contenuto di un file sul terminale

## Dispositivi standard di I/O

- I programmi scritti per terminali a carattere (compresi i comandi per shell) usano 3 dispositivi standard di I/O:

`stdin` (input)

`stdout` (output)

`stderr` (error)

- Normalmente `stdin` è collegato alla tastiera, mentre `stdout` e `stderr` sono collegati al terminale video a caratteri
- I dispositivi possono essere “rediretti” su file o in ingresso ad altri comandi tramite gli operatori di redirezione `>`, `>>`, `<`, `|`.

## Esempi di redirectione

- `ls > pippo.txt`  
(sovrascrive `pippo.txt`)
- `ls >> pippo.txt`  
(appende a `pippo.txt`)
- `sort < pippo.txt`  
(ordina il contenuto di `pippo.txt` e manda in `stdout`)
- `ls mydir | sort`  
(manda in `stdout` la lista ordinata dei file contenuti in `mydir`)
- `cat file.txt | sort | uniq > file2.txt`  
(ordina il contenuto di `file.txt`, ne elimina le righe ripetute e scrive il risultato su `file2.txt`)

# Interfaccia grafica

- I moderni SO mettono a disposizione un'interfaccia a finestre per l'interazione con l'utente (*Graphical User Interface* o GUI)
- Le interfacce grafiche si basano su uno stile di interazione detto WIMP (*Window, Icon, Menu, Pointing device*) dall'insieme degli elementi tipici di tale interazione.
- L'interfaccia definisce uno standard per i vari *widgets*, ossia elementi di controllo quali menu, bottoni, toolbars, scrollbars, finestre di dialogo, campi di testo, etc.
- Tali oggetti grafici elementari possono essere utilizzati nei programmi applicativi dotati di interfaccia grafica mediante chiamate alle API (*Application Programming Interface*)