RIASSUNTI SISTEMI OPERATIVI

Capitolo 1 e 2

Definizione di Sistema Operativo:
- Un SO sfrutta le risorse hardware della cpu per offrire dei **servizi** agli utenti. L'SO gestisce la memoria e i dispositivi di I/O.
- Rappresenta un'**interfaccia** tra hardware e software.
- Software che dipende solamente dall'hardware e non da altri software, il quale considera attività e assegna risorse.
- Controlla il trasferimento, la memorizzazione e l'elaborazione dei dati.

Elementi di base: cpu (controlla ed elabora operazioni), **memoria principale** (memorizza dati e programmi), **moduli I/O** (trasferiscono dati tra pc e l'esterno)

Registri: PC: contiene l'indirizzo della prossima istruzione. **IR:** contiene l'istruzione da eseguire.

Obiettivi del SO:
- convienenza: facile uso del computer
- efficienza: buona gestione delle risorse
- capacità di **evoluzione**: introduzione di nuove funzionalità

Schedulazione e gestione delle risorse:
Una politica di allocazione delle risorse e di schedulazione deve considerare 3 fattori:
- equita: tutti i processi dovrebbero avere le stesse possibilità di accesso ad una risorsa.
- tempo di risposta differenziale: l'OS discrimina alcuni processi rispetto ad altri piu importanti.
- efficienza: minimizzazione del tempo di risposta.

Code: il SO gestisce delle code, le quali rappresentano una lista di **processi in attesa** di una risorsa. La coda **a breve termine** si compone di processi che risiedono in memoria principale e sono pronti per l'esecuzione, mentre la coda **a lungo termine** include i job in attesa di utilizzare il sistema. Il dispacher o lo schedulatore hanno il compito di scegliere quale processo eseguire dalla coda.

Servizi offerti dal SO:
- **creazione ed esecuzione di programmi**
- **accesso** ai dispositivi di **I/O**
- **accesso** e **protezione** del sistema e dei **file**
- rilevazione di **errori** e **contabilità**

Gerarchia di un SO:
Livello **1, 2, 3, 4**: circuiti elettronici | istruzioni del processore in linguaggio macchina | chiamate di sistema | interruzioni.
Livello **5, 6, 7**: gestione processi tramite registri | memoria secondaria | trasferimento blocchi di dati | organizzazione dello spazio degli indirizzi virtuali quindi pagine e segmenti.
Livello **8, 9, 10, 11**: canale logico per il flusso dei dati tra processi | memorizzazione dei file con nome | accesso ai dispositivi esterni | gestione identificatori.
Livello **12**: gestione ordinata dei processi
Livello **13**: interfaccia utente (shell), essa accetta comandi, li interpreta e crea e controlla processi.

Esecuzione istruzioni:
Due passi: fetch ed esecuzione
1. la cpu preleva una istruzione dalla memoria
2. il PC viene incrementato cosi da avere l'indirizzo della prossima istruzione
3. l'istruzione viene caricata nell'IR
4. la cpu esegue l'istruzione richiesta

Interrupts:
- Si generano da: **un risultato di una esecuzione, un timer, un controller di I/O, errori.**
- Un interrupt permette al processore di eseguire altre istruzioni durante una operazione di I/O.
1. Il modulo di I/O manda un segnale di richiesta di interruzione alla cpu
2. La cpu finisce l'istruzione corrente e segnala la ricezione dell'interruzione
3. La cpu salva il PSW e il PC correnti, e carica un nuovo valore sul PC
4. Si eseguono le azioni richieste dall'interruzione
5. Si ripristinano i vecchi valori del PSW e PC

Gestione **interruzioni multiple:** o si **disabilitano** le interruzioni durante l'elaborazione di un'interruzione, quindi l'nterruzione resta in **sospeso**, oppure si definisce un sistema di **priorità**.

Cache: serve per avere accesso alla memoria in maniera più **veloce**. La cache contiene una piccola **porzione** della memoria principale. Quando la cpu legge una parola dalla memoria, un test determina se esiste già in cache. Se **si**, la invia direttamente al processore (senza passare per la memoria principale), se **no**, un blocco di memoria principale viene copiato in cache, e viene poi inviato alla cpu.

Tecniche di comunicazione dell'I/O
I/O programmato:
- Il modulo di I/O, dopo aver eseguito l'azione richiesta, non interrompe la cpu, la quale a sua volta deve controllare periodicamente lo stato del modulo per verificare che l'operazione sia completata.

Interrupt-driven I/O:
- La cpu invia ad un modulo un comando di I/O, proseguendo intanto con altre operazioni. Sarà il modulo di I/O stesso ad interrompere il processore.

Accesso diretto a memoria:
- La cpu delega il compito di leggere/scrivere un blocco di dati al modulo DMA. Il processore quindi può continuare con altre istruzioni. Alla fine del trasferimento il modulo DMA manda un'interruzione al processore.

Stack: uno stack è un insieme ordinato di elementi che possono essere acceduti uno alla volta. Per le sue operazioni sono necessari tre indirizzi: base, puntatore, limite.

Kernel: contiene le funzioni del sistema operativo utilizzate più di frequente.

Capitolo 3
Processi: rappresentano i programmi in esecuzione da parte del processore. Un processo è realizzato come una struttura dati. Nel contesto di ogni processo è indicato il suo stato, il quale da informazioni sul fatto se il processo sia o meno in esecuzione e in che modo.

Multiprocessore simmetrico (SMP): si tratta di una architettura hardware che implementa più di un processore, i quali possono effettuare le stesse operazioni e condividono la stessa memoria principale. L'SMP schedula i processi su tutti i processori. Alcuni vantaggi possono essere l'esecuzione in parallelo e la disponibilità di funzionare anche con prestazioni ridotte.

Multiprogrammazione: esecuzione alternata di un insieme di programmi. Il processore salta alternativamente tra i vari processi risiedenti in memoria principale.

Modello a due stati: il processo è eseguito o meno dal processore (**running**, **not-running**)
Modello a cinque stati: running (in esecuzione), **ready** (pronto per l'esecuzione), **blocked** (non è eseguito fino alla verifica di un evento), new (appena creato), exit (appena terminato)

Creazione dei processi: il sistema operativo costruisce le **strutture dati** e alloca lo spazio di indirizzamento. Il primo processo è creato al boot time e non termina mai.

1. si assegna al processo un PID unico
2. si alloca lo spazio per il processo e per il suo pcb
3. inizializzazione pcb e stato del processo a ready o ready-suspend
4. si pone il processo nella coda appropriata
5. crea o estende strutture dati

Cambio di processo (switch)
Per interrompere l'esecuzione di un processo si può interrompere con un **interrupt**, con una **trap** o con una **syscall** esplicita. Passi per effettuare lo switch:
1. si salva il contesto del processore
2. si aggiorna il pcb, si toglie lo stato running
3. sposta il pcb nella coda appropriata
4. seleziona un altro processo
5. aggiornamento del pcb e strutture dati
6. ripristino del contesto del processore

Terminazione dei processi: avviene attraverso una **system call** da parte del processo, oppure quando si verifica un **errore**. Gli errori possono essere: superamento del tempo limite di esecuzione, memoria non disponibile, errore aritmetico, timeout, ecc.

Esecuzione del Sistema Operativo

Kernel non implementati con processi:
L'esecuzione del kernel avviene al di fuori di qualsiasi processo. Quando un processo in esecuzione viene interrotto, il suo contesto viene salvato ed il controllo passa al kernel. Il kernel ha la sua propria regione di memoria e il suo stack. Inefficiente in quanto si devono riconfigurare ad ogni mode switch le memory table.

Esecuzione all'interno dei processi utente:
Consiste nell'eseguire quasi tutto il software del SO nel contesto di un processo utente. Chiamate e ritorni sono gestiti da uno stack del kernel separato. I dati del SO sono comuni a tutti i processi. Non si verificano cambi di processi ma solo cambi di modo dentro lo stesso processo. Si verifica un process switch solo nel caso venga schedulato un nuovo processo. E' efficiente in quanto non è richiesta la riconfigurazione della memory table.

Sistemi basati sui processi (micro-kernel):
Consiste nell'implementare il SO come un **insieme di processi** di sistema. Le principali funzioni del kernel sono organizzate in processi separati. Il **kernel ha solo** le funzioni di schedulazione, gestione degli spazi di indirizzamento e la comunicazione tra processi. Tutti gli altri servizi sono forniti da processi di sistema. E' **estendibile** e **flessibile** per l'aggiunta o la rimozione di nuovi servizi. E' una sorta di architettura **client/server** nello stesso calcolatore.

Linux: ha un approccio ibrido: il sistema adotta l'esecuzione all'interno dei processi utente, ma alcuni compiti sono assegnati a processi speciali chiamati kernel threads.
Windows: approccio ibrido, ma nel kernel è contenuta anche l'intera interfaccia grafica.

Stati dei processi Unix:
user running, kernel running, ready in memoria, asleep in memoria, ready swapped, sleeping swapped, created (new), zombie

Processi e Threads:
L'elemento che viene **allocato** è chiamato **thread**, mentre l'elemento che possiete le **risorse** viene chiamato **processo** o task.

Multithreading: è la capacità di un SO di supportare thread di esecuzioni multipli per ogni processo.

Processi sospesi (Swapping):
Nel momento in cui tutti i processi sono nello stato Blocked, l'SO può **sospendere** un processo ponendolo nello stato **Suspend** e **trasferirlo sul disco**. Lo spazio che si libera viene utilizzato per caricare un nuovo processo. Vengono introdotti i nuovi stati: **blocked-suspend:** il processo è in memoria *secondaria* in attesa di un evento, **ready-suspend:** il processo è in memoria *secondaria* ma è pronto per l'esecuzione appena viene caricato in memoria *principale*. Senza suspend significa che sono già in memoria principale.

Il sistema operativo gestisce 4 tabelle: **memoria, I/O, file, processi**. In queste tabelle sono contenute informazioni: allocazione della memoria principale / secondaria, protezione ecc.

Controllo dei processi
Tutte le **informazioni** di un processo, le quali sono richieste dal SO, sono definite nel **PCB**, quindi si tratta di una delle strutture più importanti del sistema operativo.
Immagine del processo: insieme di dati utente, programma da eseguire, stack del processo, pcb.

Process Control Block (PCB) contiene:
- ID del processo
- stato del processore: registri
- informazioni sul processo: priorità, stato, ecc
- privilegi del processo
- gestione della memoria

Modi di esecuzione: *user-mode e kernel-mode*
Servono in particolare per **proteggere** il sistema. I processi dell'utente sono avviati in user-mode, mentre le funzioni di sistema importanti sono eseguite in kernel-mode.
Per **passare** da una modalità ad un altra: da user a kernel, si usa l'interrupt o una syscall, setta la cpu in modalità privilegiata e salva lo stato della cpu. Da kernel a user viene abilitata dal kernel e risetta la cpu in modalità non privilgiata.

Funzioni del kernel:
- gestione dei processi (crea/termina processi, scheduling, gestione pcb)
- gestione della memoria (allocazioni, paging, segmentation)
- gestione dell'I/O (gestione buffer)
- gestione delle interruzioni

Capitolo 7
Gestione della memoria
Il compito principale del gestore della memoria è quello di portare i programmi nella memoria principale cosi che il processore li possa eseguire.

Tecniche di gestione della memoria

Partizionamento fisso: la memoria è divisa in **partizioni statiche** ed i processi sono caricati in partizioni maggiori o uguali alla dimensione del processo. E' semplice da implementare ma presenta **frammentazione interna**.

Partizionamento dinamico: le **partizioni** sono create **dinamicamente** in base alla dimensione del processo. No frammentazione interna ma è necessario **compattare** da parte della cpu per ovviare alla **frammentazione esterna**. La compattazione slitta i blocchi in modo da porli tutti in modo contiguo.

Paginazione: la memoria è divisa in **frame** (blocchi) di dimensione fissa. Ogni processo è diviso in **pagine** (pacchetti di processo) della stessa dimensione dei frame. Per caricare un processo si caricano tutte le sue pagine nei frame, quindi **non c'è frammentazione esterna**, leggera frammentazione interna. Se non ci sono abbastanza frame contigui, il processo viene ugualmente caricato in quanto si utilizzano gli indirizzi logici (numero di pagina + offset). Il SO ha una **page table** per ogni processo in cui contiene la **locazione** del frame corrispondente ad ogni pagina.

Traduzione indirizzo (paging): si estrae il numero di pagina e lo si usa come indice nella page table del processo per trovare il numero del frame. L'indirizzo fisico è costruito semplicemente concatenando i bit del numero di frame con l'offset.

Segmentazione: ogni processo è diviso in **segmenti** non tutti necessariamente della stessa dimensione. Per caricare un processo si caricano tutti i suoi segmenti in partizioni dinamiche non necessariamente contigue, quindi **non c'è frammentazione interna**. Presenta poca frammentazione esterna. Per gestire i segmenti c'è una **segment table** per ogni processo e una **lista** di blocchi liberi. Ogni entry della tavola indica l'indirizzo iniziale del segmento nella memoria e la sua lunghezza.

Traduzione indirizzo (segmentation): si estrae il numero di segmento e lo si usa come indice nella segment table per trovare l'indirizzo fisico iniziale del segmento. Si confronta l'offset con la lunghezza del segmento: se maggiore allora l'indirizzo non è valido. L'indirizzo fisico è costruito come la somma dell'indirizzo fisico con l'offset.

Paginazione con memoria virtuale: come la paginazione, ma non è necessario caricare tutte le pagine di un processo. No frammentazione esterna, overhead per gestione della memoria virtuale.
Segmentazione con memoria virtuale: come la segmentazione, ma non è necessario caricare tutti i segmenti di un processo. No frammentazione interna, overhead per gestione della virtuale.

Frammentazione interna: rappresenta la perdita di spazio all'interno di una partizione a seguito del fatto che un blocco di dati caricato è di dimensione minore rispetto alla partizione.
Frammentazione esterna: dopo che la memoria viene allocata e deallocata molte volte si vengono a creare nella memoria dei buchi che spesso non vengono riallocati perchè troppo piccoli. Per risolvere tale fenomeno si ricorre alla **compattazione.**

Algoritmi di allocazione: rappresentano i modi per riempire i buchi in memoria in modo efficiente. Esistono 3 tipi di algoritmi di allocazione:
- best fit: sceglie il blocco di dimensione più simile a quella richiesta. Nonostante il nome, è il peggiore in quanto si vengono a creare blocchi vuoti molto piccoli.
- first fit: sceglie il primo blocco disponibile sufficientemente grande. Semplice e efficiente, ma tende ad allocare la prima parte della memoria.
- next fit: sceglie il blocco disponibile a partire dall'ultimo blocco allocato. Tende ad allocare verso la fine della memoria, lasciando l'inizio più vuoto.

Buddy system:
Si tratta di una tecnica di gestione della memoria la quale funziona cosi: l'intero spazio per l'allocazione è trattato come un **singolo blocco** di dimensione 2^u. Se viene fatta richiesta di dimensione s compresa tra 2^{u-1} e 2^u allora viene allocato l'intero blocco, altrimenti viene **diviso** in due sottoblocchi di uguale dimensione 2^{u-1}. Se vengono deallocati due blocchi compagni, allora essi vengono fusi insieme per creare un blocco grande quanto la somma dei due. Ad esempio partendo da un blocco di 1MB, se viene fatta richiesta di 800KB viene allocato l'intero blocco, mentre se la richiesta è di 300KB il blocco viene diviso in due compagni da 512KB, e in uno vengono allocati i 300KB. Se dealloco i 300KB, si fondono i due blocchi da 512 per ricreare 1MB.

Rilocazione:
L'immagine di un processo la prima volta viene caricata in una partizione della memoria principale. Quando il processo viene deallocato e riallocato, può essere assegnato ad un'altra partizione di

memoria rispetto alla precedente. In pratica **le locazioni cambiano** ogni volta che un processo viene caricato in memoria. Per risolvere tale problema si utlizzano 3 tipi di indirizzi: **indirizzo logico** il quale fa riferimento ad una locazione di memoria, un **indirizzo relativo** che rappresenta una locazione rispetto all'inizio del programma ed un **indirizzo fisico** cioè la locazione effettiva in memoria principale. Quando un processo è in stato Running, il **registro base** è caricato con l'indirizzo iniziale del processo in memoria, mentre un **registro limite** ne indica la sua locazione finale. Durante l'esecuzione del processo, il valore del registro base viene sommato all'indirizzo relativo per produrre un **indirizzo assoluto.** L'indirizzo risultante è **confrontato** con il valore del registro limite: se esso è entro i limiti allora si esegue l'istruzione, altrimenti si invia un interrupt al sistema operativo, che lo dovrà gestire. Tale meccanismo funge anche da **protezione** in quanto ogni immagine dei processi è isolata tramite i registri base e limite.

Capitolo 8
Memoria virtuale

La memoria virtuale rappresenta la memoria allocata sul disco. Tale memoria **permette** di:
- caricare in memoria **solo alcune parti di** processo, quindi caricare **più processi** in memoria.
- avere una **memoria più grande della memoria principale**, di dimensioni pari a quelle del disco.

Resident set: porzione di processo presente in memoria principale.
Thrashing: fenomeno che si presenta quando il processore passa più tempo a trasferire pezzi piuttosto che eseguire operazioni.
Page fault: interruzione generata quando un processo accede ad un indirizzo che non si trova in memoria principale. Il SO pone il processo in stato **blocked** e aspetta la pagina dal disco, intanto un altro processo viene schedulato, poi quando quello blocked ha ricevuto la pagina, tornerà ready. **Minor page fault:** non è richiesto un input dal disco, **major** è richiesto accesso al disco.

Disk cache: ha l'obiettivo di tenere in memoria i file acceduti di frequente. Alcuni file vengono utilizzati da più processi, quindi si **mappano in memoria i file** o parti di essi per un accesso molto più veloce.

Paginazione
La memoria è divisa in **frame** di dimensione fissa. Ogni processo è diviso in **pagine** della stessa dimensione dei frame. Anche con la memoria virtuale è necessaria una **page table**, ma stavolta è leggermente più complessa in quanto contiene delle informazioni in più oltre la locazione del frame corrispondente ad ogni pagina. Le entry della page table contengono **un bit aggiuntivo** che indica se una pagina **è presente o meno** in memoria principale: se presente, allora la entry contiene anche il frame corrispondente alla pagina. **Un altro bit** viene utilizzato per indicare se la pagina è stata **modificata** o meno dal momento in cui è stata caricata in memoria.
La **lettura di una parola** dalla memoria riguarda la traduzione di un indirizzo virtuale o logico in un indirizzo fisico. Può capitare che la quantità di memoria dedicata solo alle **page table** potrebbe essere **troppo** alta, quindi anche le page table potrebbero essere memorizzate nella memoria virtuale, e quindi subiscono anch'esse la paginazione.
La **dimensione della pagina** influenza la percentuale di page fault. Al **crescere** della dimensione della pagina, dopo un certo tempo, ogni pagina conterrà locazioni sempre più distanti dai riferimenti recenti, perciò **aumentano** i page fault.

Page table all'interno dell'immagine del processo: solitamente ogni **processo** ha la sua page table, ma può capitare che la quantità di **memoria** occupata dalle tabelle potrebbe essere **troppo** alta. Per ovviare a tale problema, anche la **page table** potrebbe essere soggetta alla **paginazione**, cosi che il **vantaggio** è quello di non dover avere in memoria necessariamente l'intera page table. Nel caso le page table siano molto grandi, si può anche far uso di un sistema a **più livelli**. Lo **svantaggio** sta nel fatto che se la page table è a più livelli, si deve fare un accesso in più per ogni livello, i quale genera un page fault.

Inverted Page Table
In questo approccio la parte contenente il numero della pagina di un indirizzo virtuale viene mappato in una tavola hash, la quale contiene un puntatore alla page table invertita che contiene le entry della tabella delle pagine.

Translation Lookaside Buffer
Rappresenta una **cache** che contiene le entry della page table che sono state usate più di **recente**. Il processore controlla per prima cosa il TLB, se è presente la entry allora si recupera il numero di frame per formare l'indirizzo reale, altrimenti si accede alla page table usando il numero di pagina come indice. Se il present bit è a 1, allora la pagina è nella memoria principale, altrimenti avviene un page fault il quale viene gestito dal SO che caricherà la pagina necessaria.

Segmentazione
Ogni processo è diviso in **segmenti** non tutti necessariamente della stessa dimensione. I vantaggi sono la migliore gestione delle strutture dati crescenti, modifica e ricompilazione indipendente dei programmi, condivisione tra processi e protezione. Anche con la memoria virtuale è necessaria una **segment table** che contiene altre informazioni oltre l'indirizzo iniziale del segmento in memoria e la sua lunghezza. E' presente **un bit** che indica se il segmento è in **memoria** ed **un altro bit** che indica se i contenuti del segmento sono stati **modificati** dall'ultimo caricamento in memoria. La **lettura di una parola** dalla memoria richiede la traduzione di un indirizzo virtuale o logico in un indirizzo fisico.

Paginazione e Segmentazione combinate
In un sistema che combina paginazione e segmentazione lo spazio di indirizzi è diviso in un certo numero di segmenti. Ogni segmento è a sua volta diviso in pagine di dimensione fissa, lunghe quanto i frame in memoria principale. Ogni processo ha una **segment table** e per ogni segmento c'è una **page table**.

Fetch Policy (strategia di fetch)
Determina **quando** una pagina deve essere **caricata** nella memoria principale. Ci sono due metodi: **demand paging**: la pagina è caricata in memoria solo quando si fa riferimento a tale pagina e **prepaging**: vengono caricate in memoria, oltre alle pagine necessarie, anche altre pagine contigue ad esse, così che, se necessarie, non devono venir ricaricate.

Positioning Policy (strategia di posizionamento)
Determina **dove** deve risiedere un pezzo di **processo** nella memoria. In un sistema a **segmentazione** pura si usano algoritmi come best-fit, first-fit, ecc, in sistemi con **paginazione o** paginazione **combinata** a segmentazione il posizionamento è **irrilevante** in quanto la traduzione di indirizzi è effettuata via hardware con pari efficienza per ogni pagina-frame.

Replacement Policy (strategia di sostituzione)
Determina, nel momento in cui una nuova pagina deve essere caricata, **quale** delle **pagine** presenti in memorie verrà **sostituita**. Si cercherà di rimuovere la pagina a cui in futuro si accederà di meno.

Frame bloccati: si possono bloccare dei frame nella memoria per impedirne la sostituzione. Tali frame sono solitamente appartenenti al **kernel**. E' possibile il blocco del frame associando al frame un **lock bit** settato a 1.

Algoritmi utilizzati per determinare le pagine da sostituire

Algoritmo ottimo: seleziona la pagina alla quale ci si riferirà dopo il tempo più lungo, causando così il minor numero di page fault in assoluto. Tale algoritmo è solamente teorico in quanto dovrebbe predire il futuro.

Last recently used (LRU): seleziona la pagina a cui il processo non si è riferito per il tempo più lungo. Secondo il principio di località, dovrebbe essere la pagina a cui il processo non farà riferimento successivamente. Richiederebbe di associare ad ogni pagina con un timestamp e ciò provocherebbe un eccessivo overhead.
First in first out (FIFO): tratta i frame allocati per le pagine di un processo come un buffer circolare e le pagine sono rimosse utilizzando il metodo round-robin. E' una strategia di semplice implementazione. Sostituisce la pagina che è stata a più lungo in memoria.
Orologio (clock policy o di seconda chance): ad ogni frame si assegna un **use bit**. Quando una pagina è **caricata** per la prima volta lo use bit del frame è a **1**. I frame candidati per la sostituzione sono trattati come un **buffer circolare** con associato un puntatore. Quando si deve sostituire una pagina il SO scorre il buffer e cerca un frame con use bit a 0 il quale verrà **sostituito**, se prima incontra use bit a 1, li setta a 0. Tale algoritmo approssima la strategia LRU.
Inoltre tale algoritmo può essere **migliorato** usando un **bit di modifica**. L'algoritmo preferirà quindi sostituire una pagina **non modificata** e non usata recentemente in quanto se non è stata ancora modificata allora non necessita di essere riscritta in memoria secondaria.
Aging Policy (politica di invecchiamento): per ogni pagina si tiene un valore di **età** (age), il quale più è **piccolo** e più è **vecchia** la pagina. **Sostituisce** le pagine con il valore più piccolo di age. Tale valore di age ha valore massimo appena la pagina viene caricata, così che una nuova pagina non venga subito sostituita. Ad esempio il valore iniziale di una pagina potrebbe essere 1111, col passare del tempo si **shifta a destra** tale cifra, cosi da ottenere 0111. Più passa il tempo più tale pagina sarà probabile candidata per la sostituzione.

Gestione del resident set: il SO deve decidere quante pagine caricare e quindi quanta memoria allocare per un processo. Esistono 2 strategie: **allocazione fissa**: assegna ad un processo un numero fisso di pagina per la sua esecuzione, **allocazione variabile**: un processo durante la sua esecuzione può variare il suo numero di frame allocati.

Working set: indicato come **W(t,Δ)** rappresenta l'insieme delle pagine di un processo a cui ci si è riferiti nell'intervallo [t – Δ + 1, t]. Delta rappresenta una window, più è grande e maggiore è il working set.

Page fault frequency (PFF): algoritmo che fa uso di un **use bit** per ogni pagina. Quando si **accede** ad una pagina si assegna lo use bit a **1**. Quando avviene un page fault si prende nota del **tempo** trascorso dall'ultimo page fault per il processo. Si definisce una **soglia F**: se il **tempo** dall'ultimo page fault è **minore** di F allora si **aggiunge** una pagina al resident set, altrimenti si scartano le pagine con use bit a 0. Tale strategia si può **migliorare** con due soglie: una per l'aumento del resident set, ed una per ridurlo.

Cleaning Policy: determina quando una pagina modificata deve essere **scritta** in memoria **secondaria**. Ci sono due metodi: **demand cleaning**: una pagina è scritta in memoria secondaria solo quando è stata scelta per la sostituzione, **precleaning:** scrive le pagine modificate prima che i loro frame siano necessari, cosi da scriverle a gruppi. Una miglioramento può essere effettuato tramite i **buffer per le pagine:** si fa cleaning solo sulle pagine sostituibili, quindi si creano **due liste** in cui vanno rispettivamente le pagine **modificate** e quelle **non modificate**.

Load control (controllo del carico): determina il **numero dei processi** che saranno **in memoria** principale. Se ci sono pochi processi in memoria, ci sono processi bloccati e si passa tempo a trasferirli sul disco. Mentre se ci sono troppi processi si verificheranno molti page fault. Per superare tale problema **l'algoritmo** del working set e il PFF **incorporano** il controllo del carico, **eseguendo solo** i processi con un **resident set** abbastanza **grande**.

Sospensione dei processi
Si sospendono i processi, ossia vengono **scaricati su disco**, per ridurre il livello di multiprogrammazione. Solitamente si sospendono i **processi**: con **bassa priorita**, che **provocano page fault** (tale processo non ha il suo working set in memoria e sarà cmq bloccato), **ultimi attivati** (processo propenso ad avere il working set in memoria)

Capitolo 9
Scheduling

Obiettivo: lo scopo dello scheduling è quello **di assegnare i processi** che il processore deve eseguire in modo da soddisfare obiettivi di sistema come throughput, tempo di risposta e efficienza.

Tipi di scheduling
A lungo termine: si aggiunge un processo all'insieme dei **processi che devono essere eseguiti**. Controlla il livello di multiprogrammazione gestendo quali processi sono permessi per il processamento da parte del sistema.
A medio termine: si aggiunge un processo all'insieme dei processi che sono parzialmente o completamente in memoria. Ha la funzione di **sospendere** i processi e trasferirli sul disco (**swap**).
A breve termine: si decide **quale processo** disponibile **sarà eseguito** dal processore. E' anche detto allocatore o dispatcher e decide quale processo eseguire per prossimo e per quanto **tempo**. Tale scheduler è richiamato sul verificarsi di tali **eventi**: interruzioni di clock o di I/O oppure system calls.

Processi I/O bound: un processo che usa prevalentemente dispositivi di I/O
Processi CPU bound: un processo che usa prevalentemente il processore

Criteri di scheduling (utente)
Tempo di risposta: intervallo di tempo dall'invio della richiesta al momento in cui la richiesta non inizia ad essere ricevuta.
Tempo di turnaround: intervallo di tempo dall'invio del processo al suo completamento.
Prevedibilità: un processo dovrebbe essere eseguito nello stesso tempo e allo stesso costo indipendentemente dal carico del sistema.

Criteri di scheduling (sistema)
Throughput: massimizzazione del numero di processi per unità di tempo.
Utilizzo cpu: tempo in cui il processore è occupato.
Fairness: i processi devono essere trattati in modo equo.
Priorità: si devono favorire i processi con più alta priorità.

Priorità
Ad ogni processo è assegnata una priorità e lo scheduler deve sempre scegliere i processi con priorità maggiore. In questo modo i processi con bassa priorità potrebbero soffrire di starvation (sempre idle) quindi per superare tale problema la priorità di un processo potrebbe cambiare nel corso della sua esecuzione o in base al tempo speso in coda.

Modalità di decisione
Non preemptive (senza prerilascio): una volta che il processo è in esecuzione, continua fino a che non è terminato o fino a quando non riceve un blocco da un dispositivo di I/O.
Preemptive (con prerilascio): il processo in esecuzione può essere interrotto e spostato dallo stato Running allo stato Ready. Kernel moderni implementano la possibilità di interrompere anche processi eseguiti in kernel mode.

Strategie di scheduling
First Come First Served (FCFS o FIFO): non appena un processo diventa Ready viene inserito nella coda relativa. Quando il processo al momento in esecuzione si sospende, viene **selezionato** per l'esecuzione il **processo più vecchio** della coda dei Ready. Tale strategia **favorisce** i processi **cpu bound** rispetto a quelli I/O bound. Per migliorare la straegia si combina con l'uso di priorità.

Round robin (RR o time slicing): fa uso del pre-rilascio basato sul clock. Viene generata **un'interruzione** di clock **ad interrvalli** regolari. Quando avviene un interruzione, il processo in esecuzione viene inserito nella coda dei Ready, e viene scelto con strategia **FCFS** il prossimo processo da eseguire. Ha lo **svantaggio** di favorire i processi cpu bound in quanto questi ultimi utilizzato solitamente l'intero quanto di tempo, mentre i processi I/O bound ne utilizzano solo una parte essendo più brevi.

Shortest Process Next (SPN): i processi **brevi** vengono portati **in testa** alla coda dei processi Ready in modo che vengano eseguiti per primi. Ha lo svantaggio che i processi **lunghi** potrebbero soffrire di **starvation** nel caso arrivino regolarmente processi brevi, in quanto **non** presenta il **prerilascio**. Una stima del tempo che ogni processo viene eseguito può essere fatto attraverso la media esponeziale.

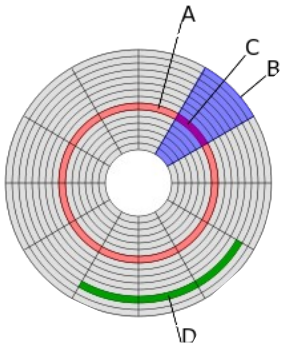
Shortest Remaining Time (SRT): rappresenta una **SPN** ma **con** il **prerilascio**. Lo scheduler sceglierà il processo che ha il **minor tempo di esecuzione** e nel momento in cui un nuovo processo ha tempo minore del processo in esecuzione, allora avviene il prerilascio.

Feedback: fa uso di un meccanismo di **priorità dinamico**. Quando un processo entra per la **prima volta** è posto nella **coda Ready 0**, ossia la coda con maggiore priorità. Dopo che è scaduto un quanto di tempo e/o c'è un altro processo nel sistema, il processo viene posto nella **coda** con **priorità minore**, e cosi via. Così facendo un processo breve completerà presto la sua esecuzione non finendo troppo in basso nelle code, mentre i processi lunghi scenderanno maggiormente. All'interno di **ogni coda** si fa uso della strategia **FCFS**, tranne nella coda con più bassa priorità in cui si usa il **round robin** in quanto il processo viene ripetutamente inserito in questa coda. Per evitare che i processi lunghi subiscano **starvation** si può adottare una strategia di **promozione** di un processo alla coda di priorità maggiore nel caso tale processo abbia aspettato **troppo tempo** nella sua coda oppure sia andato in blocco.

Linux scheduling policies: feedback con prerilascio (anche kernel mode), stima del cpu-burst, priorità dinamiche, FCFS, RR, priorità, prerilascio.

Capitolo niente
Disk Scheduling e RAID

Hard disk drive
A: traccia, cilindro (tutto)
B: settore
C: settore di una traccia
D: cluster, settori contigui



Piatto: un disco rigido si compone di uno o più dischi paralleli, di cui ogni superficie, detta "piatto" e identificata da un numero univoco, è destinata alla memorizzazione dei dati.
Traccia: ogni piatto si compone di numerosi anelli concentrici numerati, detti tracce, ciascuna identificata da un numero univoco.
Cilindro: l'insieme di tracce alla stessa distanza dal centro presenti su tutti i dischi è detto cilindro. Corrisponde a tutte le tracce aventi il medesimo numero, ma diverso piatto.
Settore: ogni piatto è suddiviso in settori circolari, ovvero in "spicchi" radiali uguali ciascuno identificato da un numero univoco.
Blocco: l'insieme di settori posti nella stessa posizione in tutti i piatti.

Per leggere o scrivere la testina deve essere posizionata sulla traccia desiderata e all'inizio del settore desiderato. Operazioni preliminari **per trasferire dati**: aspetta il device, aspetta il canale,

cerca il dato, ritardo di rotazione, trasferimento dati. Avviene un trasferimento dati quando il settore si muove sotto la testina. Il trasferimento avviene nel disk buffer, poi nel controller e poi nella memoria principale.
Tempo accesso > tempo trasferimento

Parametri delle performance di un disco
Seek time (tempo di ricerca): tempo necessario per posizionare la testina sulla traccia desiderata
Rotational delay (ritardo rotazionale): ritardo dovuto alla rotazione del disco per portare il settore desiderato sotto la testina.
Access time (tempo di accesso): seek time + rotational delay: tempo necessario per iniziare un trasferimento di dati effettivo.

Il **disk scheduling** riguarda l'insieme delle richieste da eseguire da parte del disco. In input si hanno le tracce da ricercare, lo stato del disco / testina e lo stato di altri algoritmi. In output la prossima richiesta da eseguire.

Obiettivi: massimo **throughput**, **fairness**/equità (evitare starvation o attese lunghe).
Starvation: richiesta mai servita
Unfairness: richieste che attendono molto tempo

Tipi di disk scheduling

FIFO: il primo job che giunge viene servito, non avviene nessuna starvation, se arrivamo molti processi velocemente non è efficiente.
LIFO: possibilità di starvation in quanto un job non ritorna mai alla testa della coda.
SSTF (shortest service time first): seleziona la richiesta che richiede il minore movimento da parte della testina partendo dalla sua posizione attuale. Ha un throughput ottimale ma è unfair in quanto avviene starvation quando ci sono molte richieste consecutive sulla stessa traccia.
Elevator (SCAN o LOOK): il braccio si muove solo in una direzione, soddisfando tutte le richieste che incontra finchè non raggiunge l'ultima traccia in tale direzione. Inverte la direzione quando è finito il settore o la coda è vuota. Ha un buon throughput ma è unfair in quanto il tempo massimo di attesa delle tracce sui bordi è doppio rispetto a quello sul centro. Ha starvation solo per le letture continue sulla stessa traccia.
One way elevator (C-SCAN): il braccio si muove in una sola direzione, soddisfando tutte le richieste che incontra finchè non raggiunge l'ultima traccia in tale direzione, dove ritorna immediatamente all'inizio senza fare lo scan. Una volta nella posizione iniziale rinizia lo scan. Ha un buon throughput ed è fair. Ha starvation solo per le letture continue sulla stessa traccia.

Request merging: le richieste di blocchi adiacenti vengono trattate come una **singola** richiesta, così da evitare di aumentare il tempo di **accesso**. Migliora le performance nel caso venga usato nella FIFO, è indifferente nella SSTF.

Il problema del write-starving-reads
Quando un processo deve eseguire una operazione di **scrittura**, tale processo **non** deve **attendere** che la scrittura venga completata, ma la delega al SO che deciderà quando scrivere. Il discorso è differente quando avviene una operazione di **lettura** perchè il processo **deve** effettivamente rimanere in **attesa** del dato da leggere se questo non è disponibile, quindi il processo potrebbe essere ad esempio sospeso e riattivato quando il dato è disponibile.

Linux disk scheduling
Noop scheduler: ingloba la strategia FIFO con il request merging.
Deadline scheduler: impone una tempo limite per evitare starvation. Linux lo implementa insieme al C-SCAN. Per evitare il write-starving-reads problem le **letture** non possono durare più di **500ms** e le **scritture** non più di **5s**, ma per soddisfare la deadline cerca avanti e indietro.

Anticipatory scheduling: adotta il request merging e un meccanismo di deadline. Per evitare il write-starving-reads dopo una operazione di read **aspetta** 6ms per vedere se ne giunge un'altra (non lo fa sempre). Buono per streaming, non per i DBMS.
Complete fair queuing (CFQ): adotta il **request merging** e il **C-SCAN**. Ha un approccio simile al **round robin**: schedula le richieste per un certo quanto di tempo di qualche ms, se non avvengono più richieste per il processo schedulato **aspetta** pochi ms per attendere ulteriori richieste, altrimenti passa alla successiva. Supporta il meccanismo di **priorità** dell'I/O col comando ionice. E' buono per ambienti **multiutente** e di solito è settato di **default** sui sistemi linux.

Disk scheduling switching: l'algoritmo di scheduling su linux può cambiare / switchare durante il runtime o secondo il device.

Dischi Solid-state
Non è presente **nessun seek time** perchè non c'è più un meccanismo meccanico. Il problema del write-starving-reads è ancora presente ma è risolvibile via software. Lo scheduler consigliato è il **noop**. Sono ottimi per i DBMS, Google li adotta per migliorare le prestazioni.

Drive virtuale: sono visti dal sistema operativo come dischi fisici normali. Non essendo fisici può portare vantaggi prestazionali in termini di velocità. Possono anche essere utlizzati per emulare un intera macchina (macchina virtuale).

Macchina virtuale: software che attraverso un processo di virtualizzazione crea un ambiente che emula il comportamento di una macchina fisica e nella quale possono essere eseguite applicazioni relative a tale macchina. Se la macchina virtuale andasse fuori uso, il sistema di base che ha emulato tale macchina non ne risentirebbe alcun effetto.

RAID (redundant array of indipendent disks)
Insieme di dischi fisici visti dal sistema operativo come un singolo drive logico. Un RAID ha **l'obiettivo** di migliorare la tolleranza ai guasti, le prestazioni e l'integrità dei dati.

L'insieme dei dischi fisici viene denominato **array**. Quando un disco si **rompe**, l'array passa allo stato **degradato** e le prestazioni e la ridondanza ovviamente diminuiscono. Quando si **sostituisce** un disco, questo deve subire un processo di **rebuiling** per essere aggiornato con i dati degli altri dischi. Durante tale processo si hanno prestazioni ovviamente minori. La **sostituzione** di un disco può essere **automatizzata** nei sistemi che hanno dischi inutilizzati disponibili: nel momento in cui si rompe un disco, entra subito in funzione quello prima inattivo (hot-spare disk), tale tecnica si chiama **hot-spare.**

Tecniche RAID
Mirroring (duplicazione): i dati sono duplicati su almeno due dischi. Per ulteriore affidabilità ogni disco ha un proprio controller (**duplexing**).
Parity (Hamming error correction): ci si serve di un bit aggiuntivo per la rilevazione degli errori e attraverso il codice di Hamming avviene la relativa correzione.
Striping: i dati vengono partizionati in segmenti di uguale lunghezza e scritti su dischi differenti.

Tipi di richieste
Sequential I/O: file di grandi dimensioni, bulk, streaming
Random I/O: i blocchi sono sparsi nel disco, molta richiesta per pochi dati (OLTP)
Lettura e Scrittura.

Tipi di RAID
RAID 0: divide i dati equamente tra i dischi (**striping**) con nessuna informazione di parità o ridondanza. Ha il **vantaggio** di **dividere** equamente le **operazioni** su tutti i dischi e non spreca nessuno spazio aggiuntivo oltre a quello dei dati. Ha il grande **svantaggio** che la **rottura** di un

singolo disco manda giù **l'intero** array. **L'affidabilità** misurata come **MTBF** è inversamente proporzionale al numero di dischi (due dischi, metà affidabilità).

RAID 1: i dati vengono **copiati** interamente sugli altri dischi (**mirroring**). Ha il vantaggio di effettuare **letture** in parallelo su più dischi. La **capienza** massima è limitata a quella del disco più piccolo. **L'affidabilità** è direttamente proporzionale con il **numero di dischi** presenti: se un disco si **rompe**, tutti gli altri continuano tranquillamente a funzionare. Le scritture vanno effettuate su tutti i dischi, overhead legato al mirroring, bassa scalabilità.

RAID 2: divide i dati al livello di **bit** e usa un codice di Hamming per la **correzione** d'errore. Tutti i dischi sono **sincronizzati**, cioè la testina si trova nella stessa posizione in ogni disco. Viene usato solo in ambienti dove si verificano molti errori in lettura o scrittura, ma non è molto usato in altri casi in quanto richiede un hardware costoso. Si tratta essenzialmente di un RAID 0 con maggiore affidabilità.

Calcolo della parità: in caso un disco si **guasti**, si accede al disco di parità per la ricostruzione dei dati. Una volta sostituito il disco rotto, si avvia il **rebuilding** sul nuovo disco che permette di ricostruire i dati del disco rotto. In un **esempio** di 4 dischi A B C P, di cui P il disco di parità, la parità viene calcolata mettendo in XOR i bit degli altri dischi: A XOR B XOR C. Nel caso si rompa il disco A, si effettua B XOR C XOR P. Il calcolo della parità può essere effetuato in modo più efficiente anche in questo modo: P' = A' XOR A XOR P, si fa uso di meno dischi.

RAID 3: divide i dati al livello di **byte** e presenta un disco **dedicato** per la **parità.** E' raramente usato in quanto non può eseguire richieste in parallelo. La **read** è buona in quanto viene distribuita su tutti i dischi, mentre con la **scrittura** c'è il disco di parità che funge da collo di bottiglia. **Tollera** al massimo una rottura di un disco.

RAID 4: divide i dati al livello di **blocco** e presenta un disco **dedicato** per la **parità.** Le **letture** sono buone in quanto avvengono parallelamente nel caso vengano richiesti due blocchi su dischi differenti, le **scritture** molto lente a causa della modifica e del calcolo della parità, che funge da collo di bottiglia. **Tollera** la rottura di un disco e supporta la possibilità di dischi **hot-spare**.

RAID 5: divide i dati a livello di **blocco** come il RAID 4 ma la **parità** è **distribuita** su tutti i dischi dell'array quindi a differenza del 4, non funge da collo di bottiglia. Le **letture** sono buone in quanto avvengono **parallelamente** nel caso vengano richiesti due blocchi su dischi differenti, le **scritture** lente in quanto richiedono la modifica e il calcolo della parità. **Tollera** la rottura di un singolo disco e supporta la possibilità di inserire dischi **hot-spare**.

RAID 6: divide i dati a livello di **blocco** ma presenta **doppia parità distribuita** su tutti i dischi. Ha una **tolleranza** molto **elevata** in quanto la doppia parità permette la rottura di **due** dischi senza la perdita di alcun dato. Le operazioni di **lettura** sono pressochè come il RAID 5, mentre le operazioni di **scrittura** sono leggermente peggiori per il doppio calcolo della parità.

Nested RAID: rappresentano dei sistemi RAID che sono costruiti sulla base di altri sistemi RAID. La notazione RAID XY significa che viene costruito il RAID Y sulla base di X.

RAID 01: si tratta di un RAID 1 costruito sulla base del RAID 0. Ad **esempio** se vi sono due RAID 0 con tre dischi ciascuno, nel caso si rompa un disco in uno dei due sistemi RAID 0, allora i dati non sono persi come nel RAID 0 tradizionale, ma possono essere **ripristinati tramite l'altro sistema** RAID 0. Ovviamente nel caso si rompa un disco per ogni RAID 0, i dati sono persi. Nel caso si rompa un disco, devono esserne aggiunti almeno due per compensare lo spazio tra un sistema e l'altro. La ricostruzione utilizza tutti i dischi.

12 12
34 34
56 56
78 78

RAID 10: si tratta di un RAID 0 costruito sulla base del RAID 1. Ad **esempio** se vi sono tre RAID 1 con due dischi ciascuno, si può **rompere un disco per ogni RAID 1** senza la perdità di dati. A differenza del RAID 01, la ricostruzione non necessita di tutti i dischi ma solo quello danneggiato.

11	22	33
44	55	66
77	88	99

RAID 15

11	22	33	pp
44	55	pp	66
77	pp	88	99
pp	1010	1111	1212

RAID 51

15p	15p
2p5	2p6
p37	p37
84p	84p

RAID 50

13	5p	24	6p
79	p11	810	p12
13p	1517	14p	1618
P19	2123	p20	2224

RAID 05

14	25	3p	p6
7p	p10	811	912
1316	1417	15p	p18
19p	p22	2023	2124

Just a Bunch Of Disks (JBOD): non si tratta di una configurazione RAID ma semplicemente di un array di dischi, ognuno dei quali è acceduto in modo indipendente. Ogni differente **disco fisico** è mappato in un differente **volume logico**. Nel caso si rompa un disco, l'intero array viene compromesso.

File management

Parte del sistema operativo il quale si occupa della **gestione dei dati** sul disco. Tali dati sono organizzati tramite **file** e **cartelle**, con essenzialmente una architettura ad albero. Il file management risiede per lo più nel **kernel**, ma potrebbe risiedere in un processo nel caso di architetture microkernel. Il file management ha la proprietà di essere persistente, strutturato e di essere condiviso tra i processi.

Un **file** è una collezione di record dello stesso tipo a cui ci si può **riferire** tramite il suo **nome**. I file possono essere **creati** e **cancellati** e si possono gestire i **permessi** di accesso al file da parte degli utenti o dei programmi.

Operazioni sui file: creare, cancellare, modificare un file; controllo dell'accesso; controllo permessi; accesso tramite nomi; trasferimento; backup; lettura; scrittura; chiusura;
Operazioni sulle cartelle: creare, cancellare, cambiare: al suo interno: creare file, rimuovere file, spostare file.

Obiettivi della gestione dei file: memorizzazione dei dati e possibilità di eseguire operazioni; mantenere i dati validi; ottimizzare le prestazioni; fornire supporto I/O; minimizzare la perdita di dati; affidabilità; memorizzazione indipendente dal dispositivo; sicurezza su sistemi multiutente.

File system
Ha diversi significati in base al suo contesto: regole che descrivono le strutture dati immagazzinate sul disco; montare un dispositivo o una partizione contenente un filesystem; parte del kernel che implementa il filesystem.
Il filesystem necessita di **strutture dati** per tenere traccia di quali blocchi sono allocati ad un certo file, per conoscere i blocchi liberi, per mantenere la struttura dei file e cartelle, per avere un recovery (solo journaled filesystem)
Utenti e programmi interagiscono col filesystem per richiedere qualunque **operazione** sui file. Prima di eseguire le operazioni, il filesystem **identifica e localizza** la posizione del file. La localizzazione richiede delle **cartelle**, le quali descrivono le **posizioni** dei file. Le operazioni avvengono a livello di record.

Directory dei file
Ogni collezione di file ha associata una directory di file. Tale directory contiene informazioni relative ai file tra cui attributi, posizione, permessi. La directory stessa è un file di proprietà del sistema operativo il quale viene accessa con gestori di file.

Pathname del file: percorso che parte dalla root (directory principale) fino al file stesso, il quale percorre tutte le directory intermedie per il raggiungimento del file.

File in UNIX
Unix vede tutti i file come un **flusso di byte**. Si distinguono 4 tipi di file:
ordinari: file che sono stati creati da utenti o applicazioni
directory: hanno una lista dei nomi dei file più i puntatori agli inode. Ha organizzazione gerarchica
speciali: file usati da dispositivi di I/O
file con nome: pipe con nome, code
links e symbolic links: hard links e soft links

Inode: struttura di controllo che contiene **informazioni sui file** necessarie al sistema operativo. Un inode contiene **le seguenti informazioni sui file:** permessi di accesso, numeri di riferimenti al inode, proprietario, dimensione del file, indirizzi, ultimi tempi di modifica o accesso.

Allocazione dei file: è basata sui **blocchi**, ed è **dinamica** in base alle richieste. Per tenere traccia dei file si usano le informazioni di indirizzamento contenuti nell'inode. Un inode contiene **39 byte** di info di **indirizzamento** organizzati in 13 indirizzi, da 3 byte ciascuno. I primi 10 indirizzi puntano ai primi 10 blocchi del file. Se il file è più lungo di 10 blocchi, vengono usati **livelli di indirezione**. L'undicesimo indirizzo dell'inode punta ad un blocco che contiene l'indirizzo successivo (**indiretto singolo**). Il dodicesimo punta ad un blocco che punta ad un blocco che contiene l'indirizzo successivo (**indiretto singolo**). Il tredicesimo punta ad un blocco che punta ad un blocco che punta ad un blocco che contiene l'indirizzo successivo (**indiretto singolo**).
I **vantaggi** di tale implementazione **sono**: l'inode ha dimensione fissa ed è abbastanza piccolo da poter essere mantenuto in memoria per lungo tempo. Si accede a file piccoli con nessuna o con poche indirezioni. La dimensione massima teorica è molto grande.

Hard links e soft links
Un **hard link** è l'associazione del nome di un file al suo contenuto. E' sempre **valido** e possono esserci più hard link che puntano allo stesso file. Nei sistemi linux viene mantenuto un identificatore, cioè **l'inode**, e un **contatore** dei riferimenti a quel file. Solo quando il contatore è 0, il file viene rimosso.
Un **soft link** è un file contenente un collegamento ad un altro file attraverso un pathname. Questi possono essere relativi o assoluti. Non sono sempre validi, ad esempio nel caso si sposti un file e si faccia riferimento ad un pathname non aggiornato.

Linux virtual file system
Linux assume i **file come oggetti** che condividono proprietà indipendentemente dal file system.

Oggetti principali: superblock obj (specifico file montato sul sistema), inode obj (un file), dentry obj (entry di una directory), file obj (file associati a processi).

Ext2: si tratta di un tipo di file system. Lo spazio è suddiviso in blocchi e organizzato in gruppi di blocchi. All'interno di ogni gruppo di blocchi vi sono gli oggetti sopra citati.
Ext3: ext2 con journaling

Journaling: tecnica utilizzata da molti file system per **preservare l'integrità dei dati**. Ogni scrittura su disco viene vista come una **transazione**. Prima di effettuare le operazioni richieste, le memorizza in un **log** e man mano registra su tale log le operazioni effettuate. Quindi nel caso di **crash** basterà analizzare il file di log per vedere quali sono le operazioni che sono state interrotte.

Windows vs Linux
Su Windows le app e i servizi non usano **mai system call dirette** ma fanno uso di **subsystem DLLs** con environment subsystems. Ci sono diversi **processi di supporto** al sistema. Le subsystem DLLs possono fare delle chiamate a Ntdll.dll, interagire con processi di supporto o aggiornare degli stati locali.
DLLs principali: kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll sono equivalenti ai file **.so** di **Linux** e sono condivise tra più processi.

Environment subsystem process: csrss.exe
Il file **win32k**.sys contiene la **grafica** di Windows, tale file è nel **kernel**.
Alcuni **processi di sistema**: csrss.exe, smss.exe (session manager), winlogon.exe, winint.exe, services.exe (service control manager), svchost.exe

Boot
Smss.exe è l'unico processo che fa una **system call diretta** in quanto il windows subsystem non è ancora partito. Tale processo avvia anche autochk, winint.exe il quale avvia csrss.exe che carica la grafica win32k.sys, e services.exe che carica i servizi.

Services
I servizi sono gestiti dal services control manager, cioè il file **services.exe** che è caricato da winint.exe. Tale processo può avviare, stoppare e pausare servizi, i quali sono configurati nel registro. In un unico processo possono risiedere più servizi. Svchost.exe è lo standard generic host service.

Hal: gestisce le differenze hardware delle schede madri.
Kernel: threads, process scheduler, sync, interrupt handlers.
Executive: memory management, process and thread management, I/O, iter-process comm.

Risorse: sono viste dai processi come **executive objects**, il quale è memorizzato in kernel space. In user space gli executive obj appaiono come **handles**, che sono il modo in cui i processi fanno riferimento agli executive objects, equivalgono agli handle di unix.

Executive objects examples: process, thread, job, file, mapped file, desktop.

Executive object: ciascuna **risorsa** è rappresentata dal sistema operativo come un **oggetto** (una struttura dati in kernel space). **L'executive** è la parte del sistema operativo che si occupa della **gestione delle risorse** e quindi alla rappresentazione di queste mediante questo tipo di oggetti.

Object manager: è la parte dell'executive che effettua la **gestione degli oggetti**. In particolare tiene una **tabella** per ciascun processo degli **handle** che tale processo può utilizzare. Assimilabile alla file table di unix.

Object sharing: gli oggetti possono essere **condivisi** tra i processi. Per essere identificati si usa una stringa che è rappresentata un **pathname**. Non è persistente, esiste solo in memoria.

Memory management: il process address space ha uno spazio condiviso del kernel. C'è il kernel space e lo user space, virtual memory, memory mapped files, disk cache, kernel heaps.
C'è un processo che decide la taglia del RS di ogni processo. La **eviction strategy** è di tipo **aging**, ossia la pagina che è stata più a lungo in memoria viene rimossa. Anche parti del kernel possono essere evicted. Su linux la politica di replacement è globale e non locale per processo.
Page buffering: ci sono **due kernel thread**, uno per il cleaning delle pagine ed un per il loro azzeramento. Le pagine vuote sono sempre assegnare a frame azzerati. Sia Linux che Windows hanno il page buffering, la differenza è che in windows c'è una particolare gestione per pagine che devono essere azzerate o che sono state azzerate prima che siano messe a disposizione da altri processi.
Stand by page: pagina della memoria di un processo che è stata **tolta dal resident set** ma che è ancora a disposizione in memoria in un **buffer**.
Zeroing: operazione di azzeramento che viene fatta dal kernel sulle **nuove pagine** allocate dai processi. Esistono dei kernel thread particolari che azzerano pagine in memoria e che tolgono dal resident set dei vari processi le pagine che sono meno usate e le mettono in stand-by.

Disk cache: due tipi di blocchi, letture scritture mappate nel system address space, oppure file mappati in memoria nel process address space. La dimensione della disk cache cambia dinamicamente col balance manager.
Hard/soft page fault: un hard page fault richiede un accesso a disco, il soft page fault no. Corrisondono a major a minor page fault in unix.

Cpu scheduling: ha 32 livelli di priorità, ognuno con la sua coda, ha preemption. Lo user level ha 5 livelli di priorità, i quali possono anch'essi cambiare dinamicamente. La priorità può essere incrementata dopo un I/O completion, sync events, user input oppure dopo troppo tempo che il processo è ready.
Registro: sorta di filesystem per small data elements, è persistente. E' equivalente alla cartella etc/ di Linux in quanto tale cartella contiene molte informazioni di configurazione del sistema. E' strutturato con **key e subkey** che sono cartelle e sottocartelle, ogni key ha un valore di un certo tipo. Link simbolici non sono persistenti e sono ricreati dopo ogni boot.
Il registro ha 6 radici, 3 sono reali, le altre sono reindirizzamenti: HKU, HKCR, HKLM: user contiene configurazione dell'utente, classes root contiene associazioni dei file, e local machine contiene configurazioni software, security, hardware ecc.

NTFS: file system flessibile e efficiente, ha le seguenti caratteristiche:
Capacità di recupero: è in grado di ricostruire volumi di disco e riportarli in uno stato consistente, trattando ogni azione atomicamente, quindi o viene eseguita per intero o per niente. Inoltre usa un meccanismo di memorizzazione ridondante.
Sicurezza, file di grandi dimensioni, flussi multipli.