

Sistemi Operativi

Ingegneria Informatica - Unipd

Indice

I Introduzione	5
1 Architettura di un Elaboratore	6
1.1 Central Processing Unit	6
1.2 Prestazioni di un Elaboratore	7
1.3 Memoria	7
1.3.1 Cache	8
1.4 Parallelismo	9
1.4.1 Implementazioni	10
1.5 Multiprocessori e MultiComputer	10
1.5.1 Topologia delle Connessioni	11
1.5.2 Prestazioni	12
1.5.3 Tassonomia o Classificazione dei sistemi	12
1.5.4 Implementazioni	12
1.5.5 Coprocessori	13
1.6 High Performance Computing	14
1.6.1 Parallelismo	14
1.6.2 Utilizzo delle GPU	14
1.6.3 TPU	15
1.6.4 FPGA	15
2 Sistemi Operativi	16
2.1 Definizione	16
2.1.1 Modello a Gusci Concentrici	17
2.1.2 Avvio	17
2.2 Gestione dell'I/O	17
2.2.1 Interfaccia Comune	18
2.2.2 Interrupt	18
2.2.3 DMA	19
2.2.4 Operazioni Multimodali	20
2.3 Gestione delle Risorse	20
2.3.1 Gestione dei Processi	21
2.3.2 Gestione della Memoria	22
2.3.3 Strutture Dati del Kernel	23
2.4 Virtualizzazione	23
2.5 Sicurezza e Protezione	24
2.6 Ambiente informatico	25
2.7 Sistemi Operativi Open Source	26
3 Strutture dei Sistemi Operativi	28
3.1 Servizi	28
3.1.1 Interfaccia Utente	29
3.2 Chiamate di Sistema	29
3.2.1 Implementazione	29
3.2.2 Application Programming Interface	30

3.2.3	Tipologie di Chiamate a Sistema	30
3.3	Programmi di Sistema	31
3.3.1	Linkers e Loaders	31
3.4	Implementazione di un Sistema Operativo	32
3.4.1	Design di un Sistema Operativo	33
3.4.2	Monolitici	33
3.4.3	Approccio stratificato	33
3.4.4	Microkernel	34
3.4.5	Kernel Modulari	34
3.4.6	Sistemi ibridi	35
3.5	Debugging e Tuning	36
3.5.1	Tuning	36
4	Affidabilità	37
4.1	Guasti	37
4.2	Affidabilità	37
II	Gestione dei Processi	39
5	Threads e Concorrenza	40
5.1	Thread	40
5.1.1	Visualizzazione	41
5.1.2	User e Kernel Threads	41
5.1.3	Implementazioni	42
5.2	Programmazione multicore	42
5.2.1	Concorrenza	42
5.2.2	Parallelismo	43
5.3	Sistemi di Processi	43
5.3.1	Sistemi Chiusi e Aperti	44
5.3.2	Determinatezza	44
5.3.3	Inferenza	45
5.4	Risorse	45
5.4.1	Grafico dei Processi	45
5.5	Deadlock	45
5.5.1	Grafo delle Risorse	46
5.5.2	Prevenzione dei Deadlock	46
5.5.3	Risoluzione di un Deadlock	47
6	Sincronizzazione	49
6.1	Race Condition	49
6.2	Regioni Critiche	49
6.3	Lock	50
6.4	Semafori	50
6.4.1	Implementazione	51
6.4.2	Problemi di Utilizzo	51
6.5	Monitor	52
6.5.1	Variabili Condition	52
6.5.2	Gestione di signal	53
6.6	Implementazione: Linux	53
7	Scheduling CPU	55
7.1	CPU scheduler	55
7.1.1	Implementazione	55
7.1.2	Meccanismo di Interruzione	56
7.1.3	Dispatcher	57

7.1.4	Scheduler ad alto livello	57
7.2	Comunicazione tra Processi	58
7.3	Algoritmi di Scheduling	58
7.3.1	Selezione	58
7.3.2	Algoritmi Comuni	58
7.4	Scheduling Multiprocessore	60
7.4.1	Implementazioni	62
III	Gestione della Memoria	63
8	Gestione Memoria Principale	64
8.1	Introduzione	64
8.1.1	Frammentazione	65
8.2	Paginazione	65
8.2.1	Implementazione	65
8.2.2	Supporto Hardware	66
8.2.3	Dimensioni della Tabella	67
8.2.4	Implementazioni	69
8.3	Memoria Virtuale	70
8.3.1	Spazio degli Indirizzi Virtuali	71
8.4	Pager	71
8.4.1	Page Fault	72
8.4.2	Prepaging	72
8.4.3	Algoritmi di Paging	73
8.4.4	Allocazione dei Frame	74
8.4.5	Mappatura di File	75
8.5	Trashing	75
8.5.1	Working Set	76
8.5.2	Frequenza di Page Fault	76
8.5.3	Relazione tra Working Set e Frequenza di Page Fault	76
8.6	Allocazione di Memoria al Kernel	77
8.6.1	Allocatore potenza 2	77
8.6.2	Allocazione a Lastra	77
8.7	Implementazioni	78
8.7.1	Linux	78
8.7.2	Windows	78
9	File	79
9.1	Tipo di Dati Astratto	79
9.2	Implementazione	79
9.2.1	Attributi dei File	79
9.2.2	Tipo del File	80
9.2.3	Struttura del File	80
9.3	Apertura di un File	80
9.3.1	Sistemi multiprogrammati	81
9.3.2	Condivisione	81
9.4	Allocazione	81
9.4.1	Spazio Libero	84
9.4.2	Prestazioni	84
9.4.3	Incoerenze	85
9.4.4	Accesso al File	85

10 File System	86
10.1 Directory	86
10.1.1 Montaggio	89
10.1.2 Esempi	89
10.2 Implementazioni	89
10.2.1 File System Virtuali	90
10.2.2 NetWork File System	91
11 Memoria Secondaria	92
11.1 Struttura dei Dischi	92
11.1.1 Formattazione	92
11.1.2 Correzione degli errori	92
11.2 Implementazioni	93
11.2.1 HDD	93
11.2.2 SSD	96
11.2.3 Tape	97
11.2.4 DNA	97
11.3 Misurazione delle Prestazioni	97
11.4 RAID	97
A Sistemi Operativi Moderni	100
A.1 Linux	101
A.1.1 Storia	101
A.1.2 Processi	102
A.1.3 Scheduling	102
A.1.4 Avvio	103
A.1.5 Gestione della Memoria	103
A.1.6 Sicurezza	105
A.2 Android	105
A.2.1 Storia	105
A.2.2 Estensioni di Linux	105
A.3 Windows	107
A.3.1 Storia	107
A.3.2 Struttura del Sistema Operativo	108
A.3.3 Processi	108
A.3.4 Gestione della memoria	110
A.3.5 Gestione dell'I/O	110
A.3.6 File System	110
B Programmazione Concorrente	112
B.1 Download	113
B.2 Sintassi	113
B.2.1 Packages	113
B.2.2 if/switch	113
B.2.3 Cicli	114
B.2.4 Funzioni	114
B.2.5 struct	114
B.2.6 defer	115
B.3 Concorrenza	115
B.3.1 Channels	115
B.3.2 Select	115
B.3.3 WaitGroup	116

Parte I

Introduzione

Capitolo 1

Architettura di un Elaboratore

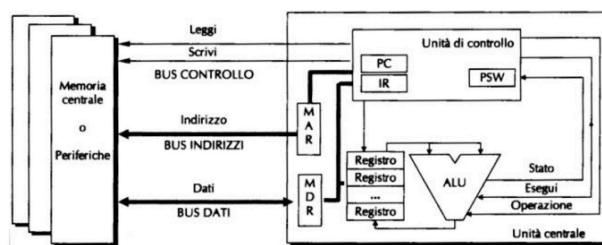
Questo non è il primo capitolo che viene trattato nel corso, tuttavia per mantenere un ordine migliore è importante comprendere il funzionamento dell'hardware in quanto il sistema operativo ci interagisce strettamente.

Parte degli argomenti del capitolo sono già stati studiati durante il corso di Architettura degli Elaboratori vengono poi introdotti i sistemi multiprocessori, la cui gestione occuperà parte significativa del corso.

1.1 Central Processing Unit

Il compito della CPU è quello di **eseguire i programmi** immagazzinati nella memoria centrale, leggendo le loro istruzioni ed eseguendole in sequenza.

La CPU opera in modo ciclico, ripetendo fino alla fine del programma le operazioni di **fetch-decode-execute**: acquisisce l'istruzione, la decodifica e la esegue



Una CPU è composta da:

- **Unità di controllo:** ha il compito di leggere le istruzioni e determinarne il tipo.
 - **Arithmentic and Logic Unit (ALU):** Esegue le operazioni matematiche necessarie per l'esecuzione dell'istruzione.
- I dati devono partire dai registri e il risultato deve tornare nei registri, questo percorso viene detto *data path*, ogni istruzione comporta l'esecuzione di uno o più cicli di *data path*.
- **Registri:** Una memoria di piccole dimensioni che risiede all'interno del processore, viene utilizzata per memorizzare i risultati temporanei e le informazioni di controllo della CPU.

Due registri particolarmente importanti sono l'Instruction Register (IR) che immagazzina l'istruzione corrente e il Program Counter (PC) che punta alla prossima istruzione.

- **MDR:** è un registro a cui la ALU ha accesso diretto e che contiene momentaneamente i dati da/per la CPU.
- **MAR:** è un registro della CPU contenente l'indirizzo della locazione di memoria RAM in cui si andrà a leggere o scrivere un dato.

- **Bus di Controllo:** un bus che permette alla CPU di specificare che operazione deve svolgere sulla memoria.
- **PSW:** *Program Status Word* i bit che forniscono informazioni sul risultato dell'ultima operazione eseguita. (overflow, carry, segno)

Ogni ciclo di esecuzione si compone di alcuni passaggi:

- Il valore di PC viene copiato nel MAR
- La memoria scrive sul MDR il valore all'indice specificato dal MAR
- Il valore di MDR viene copiato sull'IR
- L'istruzione viene eseguita dall'ALU
 - Se l'istruzione richiede degli operandi essi devono essere inseriti nei registri attraverso MAR e MDR, similmente a quanto fatto per l'istruzione
- Terminata l'esecuzione il risultato viene scritto sul MDR e l'indirizzo di destinazione sul MAR, la memoria poi si occupa di copiare il valore.
- Si ricomincia dal primo punto con l'istruzione successiva.

1.2 Prestazioni di un Elaboratore

Sia un processore con una frequenza di clock F e quindi con periodo di clock $T = \frac{1}{F}$. Questo è il tempo necessario per eseguire un ciclo di *data path*.

Istruzioni al secondo

Per svolgere un'istruzione che richiede n cicli di *data path* $t_i = n \cdot T$
Quindi il numero di istruzioni processate al secondo sarà $N = \frac{1}{t_i} = \frac{F}{n}$

Tempo di esecuzione di un processo

$$T_{es} = T \cdot \left(\sum_{i=1}^n N_i \cdot CPI_i \right)$$

Dove N_i è il numero di istruzioni di tipo i e CPI_i (Cicli Per Istruzione) è il numero di cicli richiesto per l'esecuzione di quel tipo di istruzioni. (Questo calcolo si applica per una CPU ad un core senza pipeline.)

Confronto delle Prestazioni tra due Processori

Definiamo la prestazione di un sistema come $P = \frac{1}{T_{es}}$.

Il fattore di speedup tra due sistemi, A e B è $\frac{P_A - P_B}{P_B} = \frac{T_B - T_A}{T_A}$

La *Legge di Amdahl* ci permette calcolare il miglioramento che si ottiene accelerando di un fattore a un determinato sotto insieme di istruzioni (p è la percentuale di operazioni accelerate).

$$T_{es,finale} = \frac{p \cdot T_{es,iniziale}}{a} + (1 - p) \cdot T_{es,iniziale}$$

Misurare le Prestazioni di un Processore

Un'unità di misura sono le *Instruction Per Second*, $IPS = \frac{f}{CPI}$.

Oppure si possono utilizzare i *Mega FLoating point Per Second* (*MFLOPS*), una misura che indica quante operazioni di tipo floating point riesce ad eseguire un elaboratore.

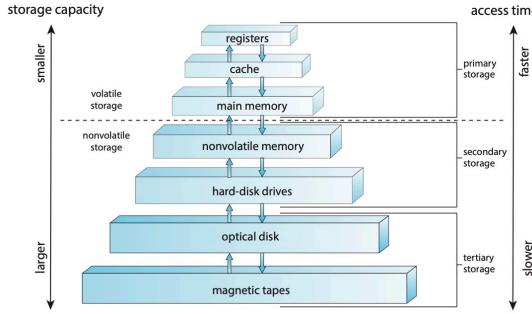
1.3 Memoria

L'architettura pensata da Von Neumann richiede una grande quantità di memoria rapida dove salvare tutti i dati dei programmi, questo non è però economicamente possibile.

La memoria principale dei sistemi moderni costituita da **DRAM** (*Dynamic Random Access Memory*) è molto più lenta della CPU limitandone così le prestazioni.

Le memorie con prestazioni comparabili a quelle della CPU sono le **SRAM** (*Static Random Access Memory*) che sono però troppo costose e quindi possono essere utilizzate solo in quantità limitata.

Per questo motivo si costruisce una piramide attorno alla memoria principale, con cache più rapide e piccole al di sopra e dischi lenti, ma estremamente capienti, al di sotto.



1.3.1 Cache

La cache si inserisce tra il processore e la memoria principale, quando la CPU richiede un valore esso viene prima cercato nella cache:

- **cache hit:** Il valore viene trovato e può essere restituito alla CPU rapidamente.
- **cache miss:** Il valore non viene trovato, quindi sarà necessario attendere la DRAM.
Dopo aver recuperato il valore esso, e i valori adiacenti, vengono salvati nella cache, applicando una politica di rimpiazzo se la cache è piena.

L'obiettivo è quello di rendere il più grande possibile la percentuale di cache hit sul totale delle richieste.

Per ottenere questo è particolarmente importante sviluppare una **politica di rimpiazzo** efficace, due principi efficaci a questo scopo:

- **Località spaziale:** È probabile che la CPU debba accedere alle celle adiacenti, come nel caso della lettura di un vettore. È quindi conveniente copiare non solo la cella, ma un intero blocco di memoria RAM.
- **Località temporale:** È più probabile che venga richiesto un dato che è stato chiesto poco prima.

Livelli di Cache

Spesso per ottimizzare i tradeoff della cache essa viene organizzata in livelli (L1, L2, ...) dove più basso è il livello più la cache è piccola e veloce. Questo viene fatto perché una cache L1 è molto più costosa di una L2 e così via.

Normalmente possiamo dire che la cache di livello $i + 1$ contiene tutti i dati della cache di livello i

Le prestazioni della cache sono particolarmente importanti per prestazioni dell'intero sistema, se viene progettata in modo corretto può garantire che una percentuale dall'80% al 99% degli accessi si limiti ad essa, senza andare alla memoria RAM.

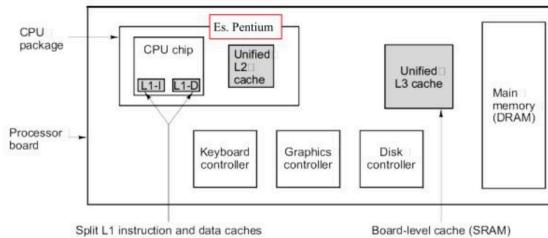
Esempio

Una microarchittetura con 3 livelli di cache può essere strutturata nel seguente modo:

- Una cache L1 interna alla CPU, che contiene una parte per le istruzioni, L1-I (16 Kbyte) e una parte per i dati L1-D (64 Kbyte).

- Una cache L2, esterna alla CPU, ma contenuta comunque nel package. Di dimensioni maggiori (512 Kb - 1 Mb)
- Una cache L3, presente sulla motherboard

Le cache L1 e L2, interne al package, hanno un bus riservato, aumentando così le prestazioni.



1.4 Parallelismo

A partire dal 1971 si è sempre rivelata vera la *Legge di Moore* che afferma: "Il numero di transistor per chip, raddoppia ogni 18 mesi".

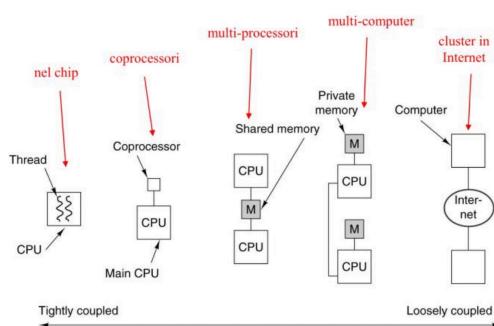
Negli anni i progettisti di processori hanno affrontato con svariate tecniche il problema di tradurre l'aumento di densità dei transistor in un aumento delle prestazioni.

Ognuna di queste tecniche ha pro e contro:

- **Ridurre il numero di cicli**, quindi migliorare gli algoritmi. Questa è ancora una strada percorribile, ma in 60 anni di ricerca nell'ambito siamo riusciti a trovare soluzioni ottime per i problemi più rilevanti sull'ambito.
- **Aumentare la frequenza di clock** si rivela problematico per l'aumento di calore che viene prodotto, i processori moderni già producono la massima quantità di calore che può essere praticamente dissipata.
- Il **Parallelismo** è la strada che si è intrapresa negli ultimi anni, ha lo svantaggio di richiedere dell'overhead, quindi raddoppiare il numero di core non raddoppia le prestazioni, ma è la strada che permette di sfruttare l'aumento di densità nei transistor.
Il parallelismo può avvenire al livello dell'istruzione, permettendo più operazioni all'interno dello stesso core (pipeline). Oppure può avvenire a livello di core, inserendo più core.

I metodi più utilizzati per realizzare il parallelismo sono:

- **Componenti piccole che interagiscono fortemente tra loro.**
Con questa soluzione viene parallelizzata la singola operazione (*parallelismo fine-grained*)
Un esempio si trova nell'hyperthreading che permette ad una CPU di eseguire più processi allo stesso tempo.
- Un **piccolo numero di CPU grandi** dotate di **interconnessioni a bassa velocità**.
Con questa tecnica l'elemento che viene parallelizzato è l'intero processo (*parallelismo coarse-grained*)
Ne sono esempi i sistemi multicore e la realizzazione di chip dedicati a specifiche operazioni.



Gli elementi che caratterizzano il parallelismo, dal punto di vista hardware sono:

- **Natura e numero degli elementi di calcolo**, pochi elementi a prestazioni elevate, oppure molti elementi a basse prestazioni
- **Natura e numero degli elementi di memoria** La memoria viene divisa in moduli per permetterne l'accesso a tutti gli elementi di calcolo, il modo in cui viene divisa influenza il parallelismo
- **Modalità di connessione** Le connessioni possono essere di tipo statico, oppure dinamico, gestite tramite uno switch in grado di instradare i messaggi.

1.4.1 Implementazioni

Il parallelismo può essere quindi implementato su più livelli:

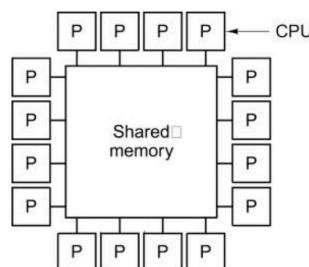
- **A livello di istruzioni** tramite pipeline.
- **Multi-threading** Due thread vengono eseguiti contemporaneamente, una CPU virtualizza due CPU.
- **Multi-core** Consente il completo multi-threading di più processi.
- **Core Eterogenei** all'interno dello stesso chip, ognuno con funzionalità specializzate.

Sia intel (a partire dall'architettura Ice Lake) che Apple (con i suoi Apple Silicon), implementano un'architettura ibrida, con core ad alte prestazioni e altri core per processi di minore importanza.

1.5 Multiprocessori e MultiComputer

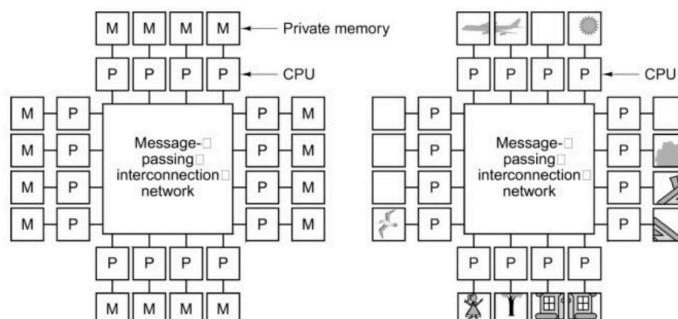
Il modo forse più diretto per implementare il parallelismo è quello di aggiungere più processori, creando un **multiprocessore**.

Se essi condividono la stessa memoria essi possono essere trattati come un unico processore e sono un esempio di un sistema *strongly coupled*.



Per inserire una potenza di calcolo ancora maggiore è necessario unire due o più multiprocessori in un **multicomputer**.

In questo caso risulta essere più complessa la programmazione, in quanto i sistemi sono *loosely coupled* ed è necessario tenere conto degli overhead dello scambio di informazioni, ma a livello hardware sono molto più semplici da costruire a parità di core.



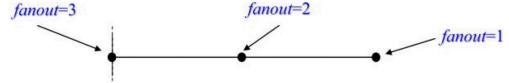
Quando due SoC sono connessi con una connessione con larghezza di banda sufficiente essi possono essere trattati come un unico chip. Ad esempio la serie Ultra dei chip Apple Silicon ha una connessione "DeepFusion" a 2.5 TB/s a bassa latenza.

1.5.1 Topologia delle Connessioni

In un sistema multicomputer la **topologia** delle connessioni tra i vari computer diventa di centrale importanza, si cercano le migliori prestazioni riducendo la quantità di connessioni necessarie.

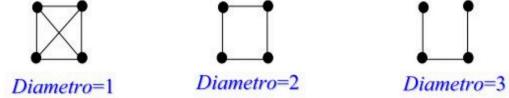
Grado o fanout

Il numero di connessioni che passano per un nodo, un grado maggiore indica maggiori tolleranze a possibili interruzioni di rete.



Diametro

Il numero massimo di passaggi tra un nodo e un'altro in una rete, dà informazioni sul tempo di comunicazione nel caso peggiore.



Dimensionalità

Numero di assi che devono essere percorsi per arrivare dal un nodo ad un'altro (caso massimo).

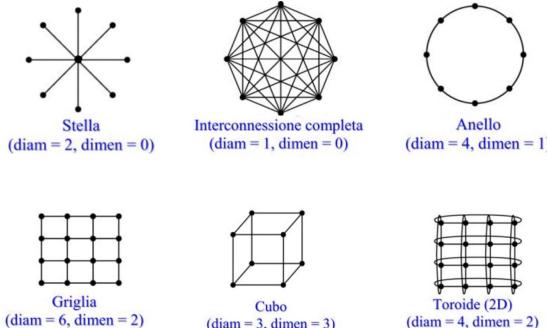
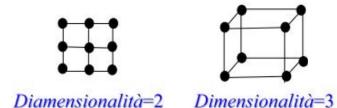


Figura 1.1: Esempi di connessioni

Nei supercomputer nella maggior parte dei casi si opta per una connessione a toroide 3D, quindi ha dimensione 3, che porta un buon compromesso tra diametro e numero di connessioni.

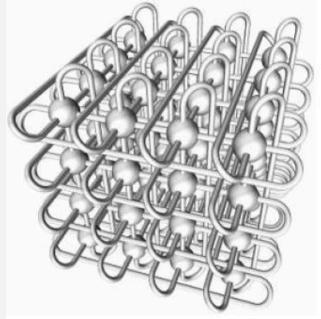


Figura 1.2: Toroide(3D)

1.5.2 Prestazioni

Vogliamo calcolare quanto aumentano le prestazioni se applichiamo n CPU allo stesso problema. Ovviamente è impossibile ottenere un aumento delle prestazioni pari a n .

- Esistono parti dei programmi **intrinsecamente sequenziali**.
- La **comunicazione**, per quanto efficiente, comporta dell'overhead.
- Gli **algoritmi paralleli** sono spesso sub-ottimi rispetto a quelli sequenziali.

$$T_{es,parallelo} = f \cdot T_{es,sequenziale} + \frac{(1-f) \cdot T_{es,sequenziale}}{n}$$

dove f è la frazione di codice sequenziale sul totale.

Prestazioni e Connessioni

Aumentare il numero di CPU spesso non è sufficiente per incrementare le prestazioni. È necessario avere un'architettura **scalabile**.

Delle buone struttura di topologia delle connessioni sono a griglia e a cubo, mentre quella ottimale è l'ipercubo, in quanto il diametro aumenta logaritmicamente rispetto al numero dei processori.

1.5.3 Tassonomia o Classificazione dei sistemi

Quella più utilizzata è quella ideata da Flynn nel 1972 che si basa sulla grandezza delle sequenze di numeri e di dati.

Nome	Sequenze di istruzioni	Sequenze di dati	Esempi
SISD	1	1	Macchina di Von Neumann
SIMD	1	Molte	Computer Vettoriali
MISD	Molte	1	Solo teoriche
MIMD	Molte	Molte	Multiprocessori, Multicomputer

1.5.4 Implementazioni

NUMA

Spesso il fattore limitante al numero di processori che è possibile aggiungere ad un sistema è la dimensione del bus che porta alla memoria principale.

Nell'architettura NUMA si fornisce ad ogni processore una **memoria locale** a cui può accedere con un suo bus separato.

Inoltre tutti i processori condividono gli stessi indirizzi fisici, quindi la comunicazione tra processori può avvenire facilmente tramite una connessione diretta.

Questo rende molto più semplice scalare i sistemi e accelera i tempi di accesso tra il processore e la sua memoria, tuttavia l'accesso alla memoria di un'altro processore è molto rallentato.

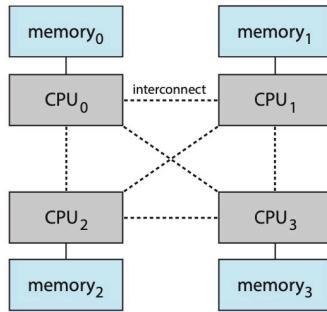


Figura 1.3: Visualizzazione sistema NUMA

Cluster

In modo simile a quanto possiamo fare per più processori possiamo collegare due o più calcolatori completi. I quali possono condividere una memoria secondaria comune. Questi sistemi sono debolmente accoppiati, quindi non tutte le operazioni possono sfruttare la completa potenza di calcolo.

Anche in questo caso è possibile creare cluster **asimmetrici** e **simmetrici**, i primi possiedono un nodo che ha l'unica funzione di gestire gli altri rimanendo in uno stato di attesa attiva, mentre nei secondi tutti i nodi eseguono le applicazioni e si controllano reciprocamente.

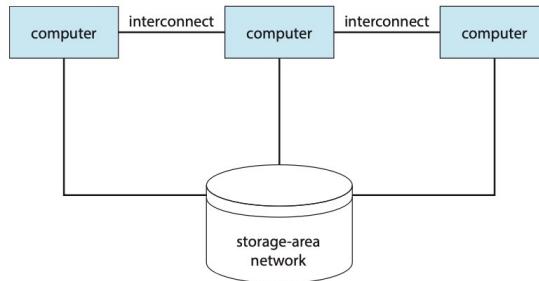


Figura 1.4: Cluster

Questo risulta essere particolarmente utile nel caso di sistemi che devono essere sempre operativi, anche nel caso di un malfunzionamento di uno dei computer il servizio non viene interrotto.

I Cluster vengono utilizzati anche nell'*High performance computing*, ma le applicazioni devono essere scritte appositamente per poter sfruttare tutte le prestazioni messe a disposizione dal cluster.

1.5.5 Coprocessori

Un coprocessore è un processore **indipendente** che esegue compiti specializzati sotto il controllo del processore principale.

- **Coprocessori di rete:** Specializzati per gestire ad alta velocità i pacchetti di rete.
- **Crittoprocessori:** Consentono di cifrare/decifrare velocemente flussi di dati.
- **Graphical Processing Unit (GPU):** Consente di processare una gran quantità di dati video e grafica 3D. Una singola GPU può contenere fino a qualche migliaio di core grafici.

Le GP-GPU, *general purpose GPU* grazie a linguaggi di programmazione quali CUDA e OpenCL possono essere facilmente utilizzate per operazioni floating point, rendendole così di grande importanza nell'high performance computing.

Per ottenere questo tipo di schede grafiche è stato necessario convertire i core per renderli compatibili alle specifiche IEEE per l'aritmetica a singola e doppia precisione. Inoltre è stato necessario fornire accesso in lettura e scrittura alla memoria principale.

- **Tensor Processing Unit (TPU):** Un coprocessore dedicato alle reti neurali per il *deep learning*.
- **Field Programmable Gate Array (FPGA):** È un dispositivo che permette di implementare algoritmi in hardware, tramite software.

Presenta una serie di blocchi logici configurabili e sono ideali per lo sviluppo e per dei prototipi rapidi.

1.6 High Performance Computing

Con *High Performance Computing (HPC)* si intende l'utilizzo di multiprocessori o cluster per effettuare computazioni concorrenti tale che porti a throughput ed efficienza elevati.

Hardware e Software devono essere strettamente collegati per permettere l'utilizzo massivo della parallelizzazione

1.6.1 Parallelismo

Esistono due tipi principali di parallelismo:

- **Parallelismo delle task** Quando diverse task possono essere eseguite indipendentemente, in questo caso possono essere facilmente allocate su diversi core.
- **Parallelismo dei dati** Quando si può operare indipendentemente su porzioni distinte dei dati, in modo da poter suddividere le porzioni su più core.

1.6.2 Utilizzo delle GPU

Mentre per le CPU l'incapacità di aumentare la frequenza e lo spostamento verso sistemi sempre più parallelizzati ha portato ad una riduzione del miglioramento delle performance, le **GPU**, nate con l'obiettivo di **elaborare processi parallelamente** hanno continuato a seguire quanto ipotizzato da Moore.

Grazie all'architettura *CUDA* le schede video NVIDIA hanno reso possibile la semplice scrittura di programmi altamente paralleli.

Ad oggi si utilizzano CPU solo quando non si può ottenere una grande parallelizzazione, mentre GPU quando è possibile distribuire il carico su un enorme numero di core relativamente poco potenti.

Alcune computazioni possono benificiare da un approccio misto tra CPU e GPU, in questo caso il trasferimento dei dati da una memoria all'altra può causare dei rallentamenti anche importanti. Per questo è in alcuni casi conveniente utilizzare una **memoria condivisa** così da eliminare questo overhead.

Un'implementazione della memoria condivisa si vede nei chip Apple Silicon che presentano un'unica pool di memoria da cui CPU e GPU attingono.

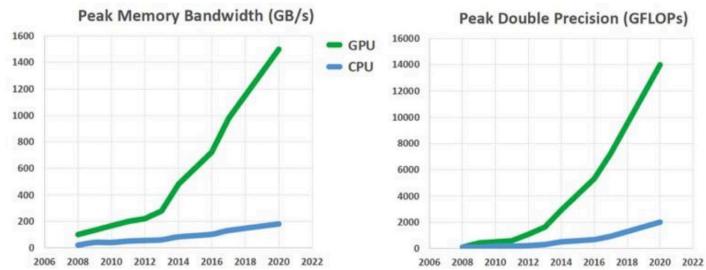


Figura 1.5: Gflops di CPUs e GPUs nel tempo

1.6.3 TPU

Da qualche anno Google si è dedicata allo sviluppo di una nuova tipologia di chip chiamati *Tensor Processing Unit*, i quali hanno un set di operazioni ancora più ridotto rispetto alle GPU, ogni core ha quindi una superficie minore, rendendo così possibile stiparne un numero maggiore sullo stesso chip.

Questi processori sono indirizzati al **Machine Learning**, in particolare all'esecuzione di algoritmi per l'allenamento di reti neurali

1.6.4 FPGA

I *Field Programmable Gate Array* sono dei dispositivi che contengono dei blocchi logici configurabili, essi consentono di programmare dell'hardware per fornirgli particolari funzionalità.

Essi permettono di implementare funzioni in hardware, con tutti i benfici che questo porta, senza dover sostenere i costi di fabbricazione di un chip.

Capitolo 2

Sistemi Operativi

Il capitolo punta a fornire delle conoscenze generali sul funzionamento dei sistemi operativi, alcune sezioni verranno poi approfondite successivamente.

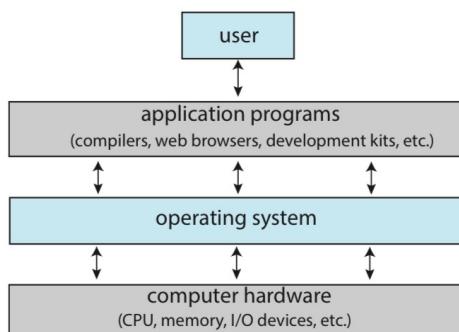
2.1 Definizione

Il sistema operativo è quel programma che è **sempre in esecuzione** quando il computer è acceso, questo viene anche detto il *kernel*, il suo compito è di gestire le interazioni tra l'utente e l'hardware del computer su cui opera.

Dato che il sistema operativo si interfaccia direttamente con l'hardware è importante comprendere a fondo la struttura dei moderni computer quali CPU, memoria e dispositivi di I/O. È infatti responsabilità del sistema operativo allocare queste risorse ai programmi.

Assieme al *kernel* ci sono altri due tipi di programmi:

- I **programmi di sistema**, che fanno parte del sistema operativo, ma non sono necessariamente parte del kernel.
- Le **applicazioni**, ovvero tutti i programmi che non sono associati al sistema operativo.
- Il **middleware**, esclusivo ai sistemi operativi per dispositivi mobile, permette di semplificare lo sviluppo delle applicazioni fornendo una collezione di ambienti software per gli sviluppatori.



Il sistema operativo ha diversi compiti rispetto all'utente e all'hardware, questo significa che deve bilanciare prestazioni e semplicità di utilizzo.

Lato Utente

Il sistema operativo deve **massimizzare le prestazioni** che vengono attribuite a uno o più utenti in modo che esso possa completare il lavoro più velocemente.

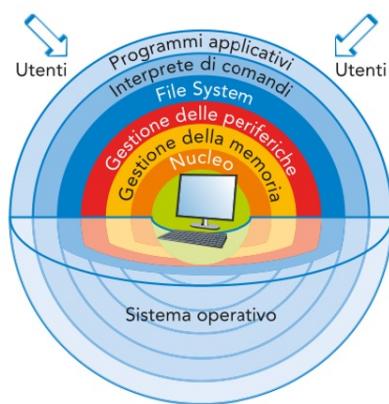
Lato Hardware

Dal punto di vista dell'hardware il sistema operativo ha il compito di **allocare le risorse** ai programmi dell'utente, **gestire eventuali errori** hardware e software e deve limitare l'utilizzo improprio delle risorse.

2.1.1 Modello a Gusci Concentrici

La struttura del sistema di calcolo può essere schematizzata mediante un "modello a cipolla" o a gusci concentrici.

I gusci circondano l'hardware, ogni livello propone al successivo un'interfaccia di sempre più alto livello, fino ad arrivare all'interfaccia utilizzata dagli utenti.



2.1.2 Avvio

All'accensione il computer compie una serie di operazioni per arrivare al sistema operativo:

Legacy BIOS

1. **BIOS (Basic I/O System)**: Si trova nella ROM del computer ed è il primo ad essere avvistato. esegue check dei componenti hardware. fornisce una serie di funzioni di base per l'accesso all'hardware
2. **MBR (Master Boot Record)**: Il Bios cerca nei dischi un bootloader a cui passare il controllo. Questi, se presenti, si trovano nei primi 512 byte del disco, ovvero l'MBR.
3. **Boot Loader**: Ha al compito di inizializzare il kernel e, di conseguenza, il sistema operativo. Alcuni esempi sono GRUB per linux e BootX per macOS.

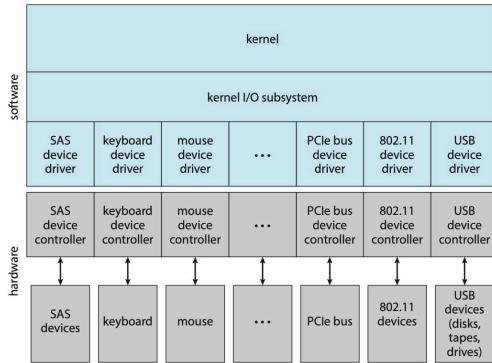
UEFI BIOS

1. **BIOS UEFI (Unified Extensible firmware Interface)**
2. **EFI Boot Loader**: Sostituisce MBR e Bootloader, si trova su una partizione del disco di avvio (identificata dalla flag boot) Carica Kernel al Sistema operativo

2.2 Gestione dell'I/O

I computer moderni, secondo il modello di Von Neumann, hanno la CPU e i controller hardware connessi direttamente alla memoria principale del sistema.

Per ogni dispositivo hardware esiste un **controller**. Il sistema operativo ha poi un **driver** per ogni controller in grado di leggere i dati dal dispositivo e riportarli in un formato standard.



Una cospicua percentuale del codice di un sistema operativo è dedicato alla gestione di I/O, questo per la grande importanza di questi dispositivi rispetto alle prestazioni e l'affidabilità del sistema, inoltre c'è una grande variabilità tra i vari dispositivi.

2.2.1 Interfaccia Comune

Uno dei compiti del sistema operativo consiste nel nascondere all'utente le caratteristiche dei dispositivi di I/O.

Questo viene fatto grazie ad una serie di strategie:

- Il **Buffering**, ovvero l'uso di un'area di memoria temporanea per immagazzinare dei dati. Per esempio la CPU può inserire dei dati in un buffer in attesa che vengano scritti sul disco.
- Il **Caching** consiste nel conservare temporaneamente delle copie dei dati frequentemente utilizzati, anche questa tecnologia viene ampiamente utilizzata nei dispositivi di archiviazione.
- Lo **Spooling** è una tecnica che consiste nell'inserire le richieste di I/O in una coda che viene poi gestita in modo sequenziale. Questo consente di gestire più attività contemporaneamente, permette inoltre di continuare l'esecuzione dei programmi del sistema operativo.
- Un'**Interfaccia** generale per i dispositivi.

2.2.2 Interrupt

Un interrupt è il metodo che un dispositivo hardware ha per comunicare con il suo driver.

Quando un interrupt si verifica il sistema cerca nel vettore degli interrupt l'indirizzo della subroutine che ha il compito di gestirlo. Successivamente salva lo stato dell'istruzione interrotta e trasferisce il controllo alla subroutine.

Durante la gestione di un interrupt, altri eventuali interrupt giunti al sistema sono momentaneamente disabilitati.

Gli interrupt sono spesso utilizzati nella gestione dei dispositivi I/O per la loro semplicità, tuttavia quando un dispositivo ha necessità di trasferire grandi quantità di dati l'utilizzo di interrupt può rallentare in modo significativo il sistema.

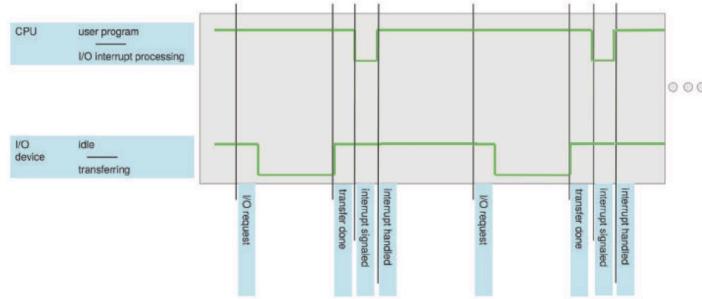


Figura 2.1: Timeline per un interrupt

Interrupt sincrono

È l'implementazione più semplice, dopo la richiesta di dati da parte del sistema esso rimane in idle finché non riceve l'interrupt.

In questo modo viene permessa una sola richiesta alla volta, se ne avvengono due allo stesso istante una delle due dovrà attendere che la gestione del precedente termini.

Interrupt asincrono

Dopo la richiesta di dati la CPU continua a svolgere altri processi, controllando periodicamente per l'arrivo dell'interrupt, permettendo così di sprecare meno tempo per ogni operazione I/O.

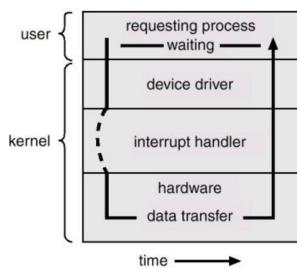


Figura 2.2: Interrupt Sincrono

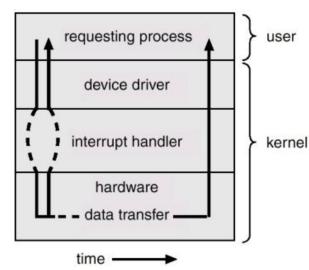


Figura 2.3: Interrupt Asincrono

Trap

Una trap (o eccezione) è un interrupt generato via software, causato da un errore o da una richiesta utente (*System Call*).

Le *system call* sono dei meccanismi che possono essere usati da un processo a livello utente o livello applicativo, per richiedere un servizio a livello kernel.

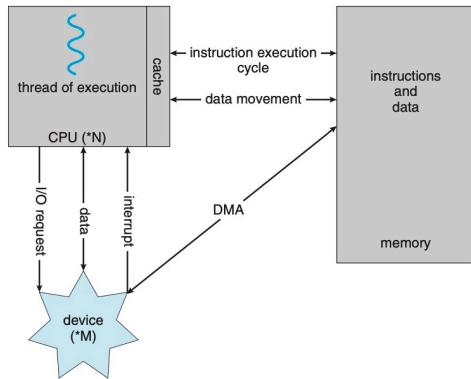
2.2.3 DMA

Il DMA (*Direct Memory Access*) è una strategia che permette di **trasferire rapidamente grandi quantità di dati**.

Dopo che il controller si accorda col sistema rispetto a dove devono essere scritti i dati in memoria principale, il controller può trasferire interi blocchi di dati senza alcun intervento da parte della CPU.

Viene solo generato un interrupt per ogni blocco trasferito per comunicare che l'operazione è avvenuta con successo.

Utilizzando questa strategia la CPU è libera di svolgere altre operazioni mentre il controller trasferisce i dati.



Esiste anche Remote Direct Memory Access che consente di trasferire dati tra la memoria di due computer, senza sovraccaricare il sistema ricevente.

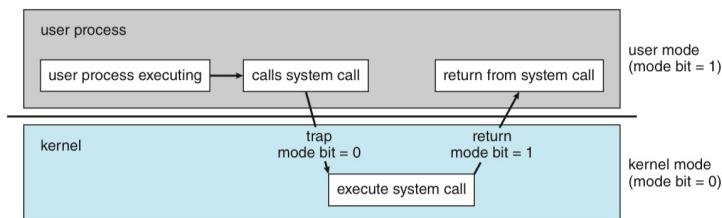
Il RDMA è particolarmente utile, se non indispensabile, per processi eseguiti in parallelo tra computer ad alte prestazioni.

2.2.4 Operazioni Multimodali

Proprio come l'hardware anche l'esecuzione del sistema operativo è scandita dagli interrupt. Nel sistema operativo gli interrupt cambiano la modalità di esecuzione dell'hardware.

Tutti i sistemi operativi utilizzano almeno 2 modalità di operazione:

- **User mode:** Modalità dove vengono eseguiti i processi dell'utente, questi processi non hanno completa libertà sul sistema, alcune operazioni sono illegali a questo livello. Se vengono eseguite l'hardware ritorna il controllo al sistema operativo.
Inoltre anche l'accesso alla memoria ed a altre risorse viene limitato in questa modalità.
- **Kernel mode:** Modalità privilegiata che permette l'accesso completo all'hardware e che viene utilizzata solo dal sistema operativo



Nelle implementazioni reali vengono utilizzare molte più di 2 modalità, uno tra questi è spesso per le macchine virtuali, mentre altri vengono utilizzati per implementare i livelli di privilegio.

Timer

Per prevenire che alcuni processi eseguano cicli infiniti o che semplicemente non restituiscano il controllo al Sistema Operativo gli si fornisce un limite di tempo sull'esecuzione.

Alla creazione di un processo utente si crea anche un timer che se arriva a zero genera un interrupt che ritorna automaticamente il controllo al sistema.

2.3 Gestione delle Risorse

Come visto precedentemente uno dei compiti più importanti del sistema operativo è la gestione delle risorse di sistema, come la memoria, i dispositivi I/O, ecc..

2.3.1 Gestione dei Processi

Un programma è un'entità "passiva", memorizzata sul disco, mentre il processo è "attivo". I programmi dientano processi quando vengono caricati in memoria principale.

Stato del processo

- **New:** il processo viene creato, diventerà ready appena il sistema operativo lo inserirà nella ready queue.
- **Ready:** il processo risiede in memoria principale ed è in attesa di essere assegnato ad un processore.
- **Running:** Vengono eseguite le istruzioni del processo, i processi vengono selezionati dal dispatcher tra quelli nella ready queue e ci possono ritornare nel caso in cui lo scheduler trovi un processo più importante oppure per un interrupt esterno.
- **Waiting:** il processo è in attesa di un evento, questo avviene da parte del processo stesso nel caso di una richiesta ad un dispositivo I/O
- **Terminated:** il processo ha terminato la propria esecuzione. Può essere dovuto dal processo che ha terminato (o ricevuto un errore), oppure può essere dettato dal sistema operativo per un utilizzo scorretto delle risorse.

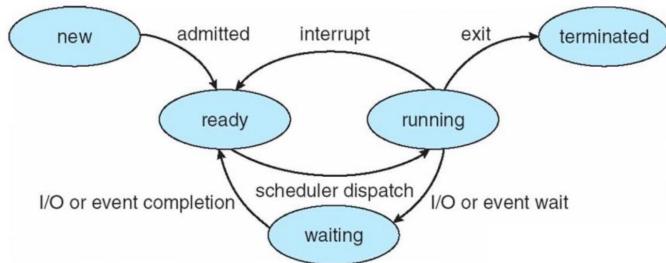


Figura 2.4: Process lifecycle

Informazioni del processo

- **Informazioni sul processo** quali stato, priorità e numero identificativo.
- Una **sezione di testo** dove è contenuto il codice del programma da eseguire.
- Il **program counter**, l'indirizzo di memoria della prossima istruzione da eseguire ed il **contenuto dei registri** della CPU.
- Lo **stack**, ovvero i dati temporanei, parametri per i sottoprogrammi, indirizzi di rientro e variabili locali.
- Una **sezione dati**, le variabili globali.
- Un **heap**, memoria dinamicamente allocata durante l'esecuzione del task.
- Informazioni sulla **contabilizzazione** delle risorse (tempo di utilizzo della CPU, ecc...).

Queste informazioni, o i riferimenti a tali informazioni vengono salvati nel *Process Control Block* (PCB), che a sua volta è salvata nella tabella dei processi.

Un programma non ha necessariamente un solo processo, ma esso descrive un insieme di processi.

Threads

Un processo deve sempre avere almeno un thread (sè stesso), ma può generarne altri in caso abbia bisogno di eseguire delle operazioni simultaneamente sfruttando il multithreading.

Tutti i thread di un processo condividono le risorse che sono state assegnate al processo, risiedono nello stesso spazio di indirizzamento e hanno accesso agli stessi dati.

Ogni thread ha un program counter, uno stato, uno spazio di memoria, uno stack e un descrittore.

Un processore dotato di Hyper-Threading può eseguire 2 thread contemporaneamente, è in grado di eseguire più istruzioni su dati separati in parallelo.

Multiprogrammazione e Multitasking

La multiprogrammazione è un aspetto fondamentale di un sistema operativo. Infatti un singolo processo non è in grado di mantenere sempre occupata la CPU, inoltre spesso gli utenti vogliono essere in grado di eseguire più programmi contemporaneamente.

Il sistema memorizza più processi nella memoria principale rendendo possibile così lo switching tra un contesto ed un'altro. Uno specifico algoritmo di scheduling ha il compito di decidere quando e come effettuare questo switch.

La multiprogrammazione rende possibile il multitasking, dove il sistema cambia velocemente da un processo all'altro al punto che all'utente sembra che non ci siano interruzioni.

2.3.2 Gestione della Memoria

La memoria centrale ha un'importanza centrale nell'architettura di Von Neumann ed è l'unica memoria di grandi dimensioni a cui la CPU ha accesso diretto (accedere al disco richiede un'operazione di I/O)

Il Sistema operativo, rispetto alla memoria principale, ha il compito di:

- Tener traccia di quali parti sono utilizzate e da chi.
- Decidere quali processi caricare in memoria quando c'è dello spazio disponibile.
- Allocare e deallocare spazio di memoria secondo necessità.

Il sistema operativo è anche responsabile di gestire in modo efficiente la memoria secondaria, in quanto spesso non è possibile caricare l'intero programma in memoria principale

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Inoltre bisogna considerare che, soprattutto nei sistemi multiprocessore, l'aggiornamento di un dato deve essere riflesso non solo nel file in cui risiede, ma anche in memoria principale e nelle cache degli altri dispositivi.

File System

Un file è una **collezione di informazioni** correlate contenenti dati e programmi.

I file vengono organizzati dal file system in directory, questo permette di effettuare controlli di accesso per verificare che l'utente abbia accesso a quelle risorse.

Il sistema operativo è responsabile delle seguenti attività per la gestione di file:

- Creazione e cancellazione di file e directory
- Supporto alle funzioni elementari per la manipolazione di file e directory
- Associazione dei file ai dispositivi di memoria secondaria
- Backup di file su dispositivi stabili di memorizzazione

2.3.3 Strutture Dati del Kernel

Un argomento centrale all'implementazione di un sistema operativo sono le strutture dati, qui vengono descritte le strutture dati utili per la costruzione di sistemi operativi. Le strutture dati vengono qui solo citate in quanto sono state già approfondite nel corso di Dati e Algoritmi.

Liste, Pile e Code

La struttura dati più comune per implementare le liste sono le liste concatenate (standard, doppiamente concatenate e circolari).

Le liste vengono spesso utilizzate per implementare Pile (implementate con politica LIFO) e Code (con politica FIFO). In particolare la Pila viene utilizzata per implementare lo Stack, utilizzato in diverse situazioni nel sistema operativo.

Alberi

Sono utilizzati per rappresentare la relazione causale padre-figlio oppure per implementare degli algoritmi di ricerca binaria efficienti.

Hashtable

Vengono utilizzate per memorizzare insiemi di dati con la possibilità di reperire un qualsiasi elemento con una complessità in tempo pari (o quasi) a $O(1)$

Le funzioni di hashing banali soffrono del problema del "paradosso del compleanno". Ovvero in un gruppo di 23 persone la probabilità che 2 di esse compia gli anni lo stesso giorno è $>50\%$, questo significa che se su una tabella di 365 elementi si inseriscono più di 23 elementi allora c'è una probabilità $>50\%$ di una collisione.

2.4 Virtualizzazione

Una tecnologia che permette di **astrarre l'hardware** di un computer e creare diversi ambienti di esecuzioni. Rende quindi possibile eseguire un sistema operativo come applicazione all'interno di un altro sistema.

L'utilizzo di una macchina virtuale garantisce la completa **protezione delle risorse**, sia per la macchina virtuale che per il sistema ospitante.

La condivisione delle risorse rimane possibile, rendendo possibile la creazione di sistemi estremamente efficienti, dove le risorse di una VM in idle vengono sfruttate dalle altre.

Dal punto di vista implementativo la macchina virtuale realizza un'**interfaccia** indistinguibile da quella di un sistema dedicato. Le risorse del computer fisico vengono condivise dall'hypervisor o VMM (Virtual Machine Manager) che può sfruttare tutte le funzioni del sistema ospitante per creare l'illusione che ogni utente abbia il proprio sistema.

Emulazione

Un altro aspetto della virtualizzazione è l'emulazione, ovvero la **simulazione di componenti hardware in software**.

Più in generale è possibile emulare un intero sistema operativo scritto per un'architettura diversa. L'emulazione però viene ad un costo importante dal punto di vista delle prestazioni, non ci si può aspettare che il codice abbia le stesse performance su tutte le architetture.

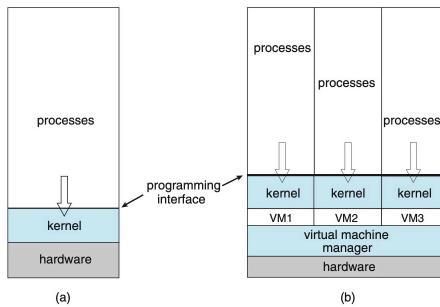


Figura 2.5: Virtual Machine Manager

Nota di Ingegneria del Software

Un'applicazione delle macchine virtuali si trova proprio nello sviluppo e nel test di applicazioni, con una macchina virtuale è possibile testare l'applicazione su varie combinazioni di hardware e software.

Apple

Negli ultimi anni Apple ha costruito diversi strumenti per l'emulazione, a partire dal software Rosetta 2, utilizzato per agevolare la transizione tra mac con processori intel a mac con architettura ARM. A partire da macOS Sonoma apple ha introdotto il Game Porting Toolkit, uno strumento che permette agli sviluppatori di traddurre giochi scritti in DirectX12 a Metal 3, aumentando notevolmente le prestazioni con un minimo sforzo.

2.5 Sicurezza e Protezione

Nel caso in cui più utenti devono usufruire dello stesso elaboratore l'accesso alle risorse deve essere limitato da regole imposte dal sistema operativo.

È compito delle strategie di protezione quello di fornire le specifiche dei controlli da attuare e gli strumenti per la loro applicazione.

La sicurezza comprende tutti i meccanismi di difesa per proteggersi da attacchi interni ed esterni. Alcuni tipi di attacco possono essere:

- **Denial of Service (DoS o DDoS):** L'attacco informatico si concentra sull'esaurire deliberatamente le risorse di un sistema con l'obiettivo di renderlo incapace di erogare il servizio per cui era stato progettato
- **Trojan:** Programmi con una funzione legittima, per la quale l'utente esegue il programma, ed una funzione dannosa nascosta.
- **Worm:** Malware in grado di autoreplicarsi e distribuirsi in modo estremamente rapido.
- **Virus:** Porzioni di codice dannoso che si legano ad altri programmi per diffondersi.

Hacker e Cracker

Un hacker è semplicemente colui che sfrutta le proprie capacità informatiche per esplorare, divertirsi e apprendere senza creare danni reali.

Mentre il cracker sfrutta le sue abilità per distruggere, ingannare o arricchirsi.

La funzione di sicurezza del sistema operativo si basa sull'identificazione degli utenti così da poter per determinare chi può fare cosa.

2.6 Ambiente informatico

Negli ultimi anni c'è stata una grande evoluzione nell'ambiente informatico e i diversi ambienti (la casa, l'ufficio) sono ora una complessa rete di computer.

I dispositivi mobile si sono arricchiti nelle funzionalità fino al punto da essere indistinguibili dai computer.

Questi dispositivi, ognuno con il suo processore e memoria locali, è connesso da una rete di comunicazione:

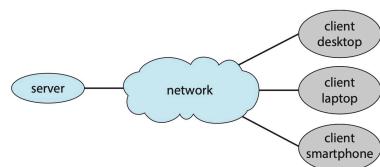
- Wide Area Network (WAN)
- Metropolitan Area Network (MAN)
- Local Area Network (LAN)
- Personal Area Network (PAN)

Un sistema distribuito ha parecchi vantaggi, infatti può fornire agli utenti un'elaborazione più rapida e può fornire una maggior quantità di dati.

Modello Client-Server

I dispositivi sono i client e richiedono servizi a server.

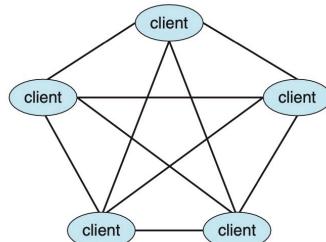
I server permettono l'accesso a servizi e risorse.



Modello Peer to peer

Un modello distribuito peer to peer non fa distinzioni tra client e server, è invece costituita da nodi equivalenti che fungono sia da client che da server rispetto agli altri elementi della rete.

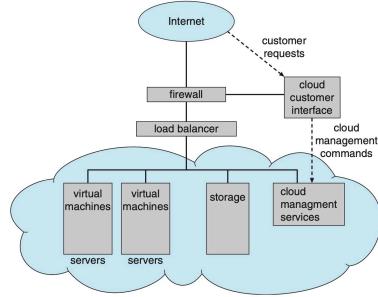
Quando un nuovo dispositivo vuole entrare in un network peer to peer deve registrare il suo servizio in un registro centralizzato di consultazione della rete. Quando un nodo vuole connettersi con un'altro della rete deve prima contattare il registro centralizzato.



Cloud Computing

Il cloud computing è una tecnologia che permette di accedere a risorse computazionali, di storage e di applicazione attraverso i servizi di rete. Risulta particolarmente utile e comodo su dispositivi mobili, le cui risorse sono fortemente limitate.

- **Cloud Pubblico:** liberamente disponibile tramite internet a chi si abbona al servizio
- **Cloud Privato:** gestito da un'azienda per utilizzo interno.
- **Cloud Ibrido:** contiene componenti pubbliche e private.
- **Infrastructure as a service (IaaS):** server o memoria disponibili attraverso l'internet.
- **Platform as a service (PaaS):** un ambiente software il cui hardware viene gestito e che permette di costruire applicativi facilmente.
- **Software as a Service (SaaS):** un'applicazione fruibile tramite internet (es. google docs).



Sistemi embedded

Sono gli elaboratori largamente più diffusi, si possono trovare ovunque nelle nostre case. Sono sistemi rudimentali, hanno compiti precisi, funzionalità limitate e un'interfaccia utente minimale, se presente.

Possiamo trovare una grande variabilità tra i sistemi embedded:

- Alcuni sono **general-purpose** con un sistema operativo standard e delle applicazioni create appositamente per implementare una funzionalità.
- Microprocessori con un sistema operativo **special-purpose** che implementa solamente la funzionalità.
- Altri ancora hanno solamente dei **circuiti** costruiti appositamente per la funzionalità e sono privi di sistema operativo.
- **Sistemi Operativi Real-time**

Questi ultimi sono la tipologia più diffusa di sistemi embedded, si utilizza quando il tempo a disposizione per la lettura, l'elaborazione dei dati e l'esecuzione è fissato.

Nel caso in cui il sistema non riesca a produrre una risposta nel tempo prefissato essa verrà considerata errata.

2.7 Sistemi Operativi Open Source

Sono sistemi operativi disponibili in **formato sorgente** anziché come codice binario.

Questo approccio ha il vantaggio di aprire il sistema ad una vasta rete di programmatore, i quali sono liberi di contribuire al software aggiungendo funzionalità e risolvendo bug e falle di sicurezza.

È da notare che software opensource e free non sono sinonimi. Il software libero permette l'utilizzo, la redistribuzione e la modifica senza costo, mentre quello opensource può avere delle licenze che limitano la libertà degli sviluppatori.

Storia del Software Libero

Il codice viene inizialmente concepito come libero, i primi sviluppatori dell'MIT lasciavano i loro codici sorgenti nei cassetti per lasciare che altri continuassero il lavoro.

Durante gli anni 70 e 80 però le aziende iniziano a cercare modi per permettere l'esecuzione del loro software ai soli clienti paganti, quindi iniziano a distribuire i file compilati invece del codice sorgente.

Per combattere questo spostamento verso il software proprietario, nel 1984 *Richard Stallman* inizia a sviluppare un sistema operativo libero, compatibile e simile a Unix (il sistema operativo più diffuso al tempo), chiamato GNU (acronimo ricorsivo di *GNU's Not Unix!*).

Stallman non è contrario all'idea di mettere un prezzo al proprio software, ma crede che i clienti debbano avere 4 libertà:

- **Eseguire** liberamente il programma
- Studiare e **modificare** il codice sorgente
- **Ridistribuire** o vendere il codice senza modifiche
- **Ridistribuire** o vendere il codice con delle modifiche

Nel 1985 Stallman pubblica il manifesto di GNU che suggerisce che tutto il software dovrebbe essere libero e fonda la *Free Software Foundation (FSF)* con lo scopo di incoraggiare l'uso e lo sviluppo di software libero.

Stallman applica il "copyleft", ovvero l'esatto opposto del copyright, a tutto il suo lavoro che lo rende completamente libero, con l'unica condizione che debba essere ridistribuito come software libero.

Questo ideale viene poi sintetizzato nella *GNU General Public License (GPL)*

Nel 1991 il progetto GNU aveva sviluppato molti tool per lo sviluppo, ma non era ancora arrivato ad un kernel completo. Linus Torvalds, uno studente Finlandese, inizia a sviluppare un sistema operativo usando gli strumenti sviluppati dal progetto GNU.

Grazie anche all'avvento di internet, che permette a sviluppatori da tutto il mondo di scaricare il codice sorgente e di contribuire attivamente al progetto, nasce il sistema operativo *Linux*.

Dal kernel nascono moltissime distribuzioni, ognuna con i suoi obiettivi e le sue particolarità.

MINIX è uno dei sistemi operativi più diffusi, anche se raramente ne sentiamo parlare. È stato sviluppato dal professor Andrew Tanenbaum come supporto didattico al suo corso universitario, con la caratteristica di essere estremamente modulare.

Il sistema operativo è così diffuso perché intel lo utilizza per L'Intel Management Unit, un software che viene eseguito da tutti i processori intel. Il sistema viene eseguito al livello di privilegio più elevato, con permessi pressoché illimitati sull'hardware.

Capitolo 3

Strutture dei Sistemi Operativi

Analizziamo ora le regole e gli strumenti necessari per l'implementazione di un sistema operativo. Esso è un sistema complesso, è quindi fondamentale definire correttamente i requisiti. Essi si possono definire da vari punti di vista: i servizi che dovrà fornire, l'interfaccia messa a disposizione agli utenti e ai programmatore.

3.1 Servizi

Alcune classi di servizi che il sistema operativo può fornire nei confronti dell'utente:

- **Interfaccia Utente:**

Quasi tutti i sistemi operativi sono dotati di un'interfaccia utente, che può prendere molte forme diverse, la linea di comando, un'interfaccia grafica, ...

Si può interagire con essa in molti modi, con mouse e tastiera, con uno schermo touch, ...

- **Esecuzione di Programmi:**

Il sistema deve essere in grado di caricare un programma in memoria ed eseguirlo, inoltre deve essere in grado di rilevare e gestire situazioni di errore.

- **Operazioni I/O:**

Il sistema operativo fornisce ai programmi dell'utente un modo per accedere ai dispositivi I/O.

- **Gestione del File-System:**

I programmi utente devono essere in grado di creare, leggere, scrivere ed eliminare file e devono potersi muovere nella struttura dell'archiviazione.

- **Comunicazioni:**

I programmi devono essere in grado di comunicare ad altri processi presenti sullo stesso sistema oppure ad altri sistemi connessi via rete. La comunicazione può essere implementata tramite memoria condivisa, oppure come scambio di messaggi

- **Rilevamento di Errori:**

Il sistema operativo deve tenere sotto costante controllo CPU, memoria e dispositivi I/O per rilevare e gestire gli errori che sorgono durante l'esecuzione dei programmi utente.

Ci sono altri servizi che il sistema operativo deve eseguire per assicurarsi che il sistema funzioni in modo efficiente:

- **Allocazione di Risorse:**

Quando ci sono più processi o più utenti diventa fondamentale gestire in modo efficiente le risorse hardware tra di essi, in particolare la CPU e la memoria. I dispositivi I/O spesso vengono gestiti in modo meno rigido utilizzando delle regole generali.

- **Logging:**

Risulta essere importante registrare quali programmi e in quali quantità utilizzano le risorse del sistema. Queste statistiche sono preziose per gli amministratori che possono capire l'effettiva necessità di risorse.

- **Protezione e Sicurezza:**

Ai possessori di informazioni su un sistema multiutente o distribuito deve essere garantito i propri dati da accessi indesiderati.

La protezione riguarda il controllo di tutti gli accessi alle risorse di sistema.

La sicurezza si basa sull'obbligo di autenticazione mediante password o sistema equivalente.

3.1.1 Interfaccia Utente

Vista la sua importanza dal punto di vista dell'utente può essere utile approfondire le interfacce che permettono agli utenti di comunicare con il sistema operativo.

Command Line Interpreter (CLI)

L'interfaccia a linea di comando permette di impartire comandi direttamente al sistema operativo. Viene talvolta implementata dal kernel oppure attraverso programmi di sistema, inoltre può essere anche installata dall'utente, il sistema operativo può offrire più *shell*.

I comandi impartiti dall'utente possono essere eseguiti secondo due modalità:

- Se il codice del comando è relativo all'interprete viene chiamata direttamente la funzione richiesta dall'utente.
- Se invece il codice del comando viene implementato da un altro programma di sistema l'interprete utilizza il comando per accedere al file contenente il codice per l'esecuzione e lo inizializza.

Spesso nei sistemi operativi non sono disponibili tutte le funzionalità tramite l'interfaccia grafica, in questi casi si può accedervi tramite l'interprete a linea di comando.

Inoltre la linea di comando risulta essere utile per l'esecuzione di comandi ripetuti in quanto è programmabile

Graphical User Interface (GUI)

L'interfaccia grafica user-friendly realizza la metafora della scrivania (*desktop*).

L'utente può muovere un cursore su icone che rappresentano programmi, file, cartelle e funzioni di sistema.

Questo rende semplice l'interazione con il sistema tramite mouse, tastiera e monitor.

L'interfaccia grafica nasce dal lavoro del centro di ricerca *Xerox Palo Alto Research Center (PARC)*, le cui idee vennero poi riprese da Steve Jobs nella creazione del software per il primo Macintosh.

Gli sviluppatori di apple non hanno però mai avuto accesso al codice, quindi provarono ad ricrearlo partendo da una semplice dimostrazione, nella maggior parte delle situazioni riusciranno addirittura a migliorare le funzionalità del sistema della Xerox.

La storia ha un triste epilogo con la Xerox che provò a fare causa ad apple sul copyright dell'interfaccia, ma una corte statunitense porrà fine alla causa un anno dopo scagionando apple.

3.2 Chiamate di Sistema

Le chiamate di sistema forniscono un'interfaccia tra i programmi e i servizi offerti dal sistema operativo.

Questo permette al programmatore di chiamare le funzioni da linguaggi di programmazione ad alto livello (C o C++) anche se l'implementazione può essere stata fatta ad esempio in assembly.

3.2.1 Implementazione

Dal punto di vista implementativo viene assegnato un numero ad ogni *system call* e il sistema operativo tiene una tabella che associa numeri al codice che gestisce la chiamata a sistema.

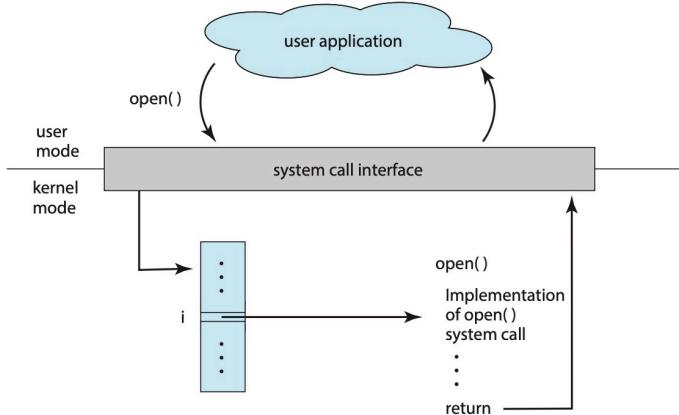


Figura 3.1: La gestione della chiamata a sistema `open()`

Spesso è necessario fornire più informazioni della semplice chiamata. Il tipo e la quantità di dati che devono essere forniti alla *system call* possono variare molto.

Ci sono diverse soluzioni a questo problema:

- **Utilizzare i registri**, per quanto sia il metodo più semplice, limita numero e dimensione dei parametri.
- Memorizzazione dei parametri su un **blocco di memoria** e passaggio dell'indirizzo del blocco utilizzando un registro (Linux e Solaris).
- push dei parametri nello **stack** da parte del programma e poi pop da parte del sistema operativo.

3.2.2 Application Programming Interface

Gli sviluppatori spesso non vedono l'implementazione di queste funzioni, che spesso risultano essere estremamente complesse anche per azioni relativamente semplici.

I programmi e i loro sviluppatori interagiscono con delle *Application Programming Interface (API)*, un set di funzioni che sono disponibili al programmatore.

L'utilizzo di un'API permette all'utente di non preoccuparsi dei dettagli implementativi e permette una maggiore portabilità del programma ad altri sistemi operativi.

Alcune delle API più diffuse ed utilizzate sono la *Win64 API* per Windows, *POSIX API* per Unix, Linux e MacOS e la *Java API* per la *Java Virtual Machine*.

3.2.3 Tipologie di Chiamate a Sistema

Le chiamate a sistema possono essere approssimativamente raggruppate in 6 categorie principali:

- **Controllo dei Processi:**
Permette di creare, caricare, eseguire, far attendere e terminare un processo. Inoltre permette di leggere e modificare gli attributi del processo (priorità, tempo di esecuzione, ...), assegnare/rilasciare memoria e inviare segnali.
- **Gestione delle informazioni:**
Permette di ottenere informazioni di sistema, la lettura/modifica dell'ora e della data, la lettura/modifica degli attributi di processi, file e dispositivi.
- **Comunicazione:**
Permette l'apertura/chiusura di una connessione, l'invio/ricezione di messaggi, inserimento/e-sclusione di dispositivi remoti, condivide informazioni sullo stato dei trasferimenti e gestisce la condivisione della memoria.

- **Gestione dei file:**
Permette la creazione/eliminazione/apertura/chiusura/lettura/scrittura di file e condivide gli attributi dei file.
- **Gestione dei dispositivi I/O:**
Permette la richiesta e rilascio di un dispositivo, la lettura e scrittura dei dati e condivide gli attributi di un dispositivo.

3.3 Programmi di Sistema

I programmi di sistema forniscono un ambiente conveniente per lo sviluppo e l'esecuzione di programmi.

In alcuni casi possono essere semplici interfacce per chiamate a sistema, altre volte possono essere più complesse.

Alcuni possibili scopi dei programmi di sistema sono:

- **Gestione di file**
- **Modifica di file**
- **Informazioni di stato**
- **Supporto a linguaggi di programmazione** (compilatori, debugger, ...)
- **Caricamento ed esecuzione dei programmi**
- **Comunicazioni**
- **Servizi background**

Sono programmi che vengono lanciati al boot, alcuni terminano dopo aver completato alcune azioni, altri continuano ad essere eseguiti fino allo spegnimento.

Supportano servizi quali il controllo del disco, scheduling dei processi, logging degli errori, stampa, connessioni di rete, ...

- **Programmi applicativi**

Sono Programmi che non fanno parte del sistema operativo (es. browser, editor di testo, ...), ma sono importanti per l'esperienza degli utenti.

3.3.1 Linkers e Loaders

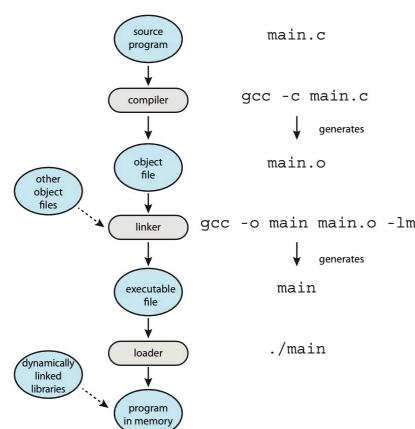
Dal file sorgente un **compilatore** crea un file oggetto, il quale è progettato per essere caricato in qualsiasi locazione di memoria.

Poi il **linker** crea il file eseguibile, combinando i file oggetto e le librerie necessarie.

Il file così generato è pronto per essere caricato in memoria da un **loader**.

I file oggetto ed eseguibili devono avere un formato standard per poter comunicare al sistema operativo come caricare ed eseguire il programma.

Per i sistemi UNIX-like il formato standard è l'*Executable and Linkable Format (ELF)*, per Windows è il formato *PE*, per macOS *Mach-O*.



I moderni sistemi operativi *general purpose* non compilano le librerie nei file eseguibili, utilizzano piuttosto delle librerie collegate dinamicamente (es. DLL per Windows) secondo le necessità e condivise da tutti i programmi.

Le applicazioni sono OS-specific

Le applicazioni compilate per un sistema **non** possono, in generale, essere eseguite su altri sistemi. Se questo fosse possibile non dovremmo scegliere il sistema operativo in base agli applicativi che sono disponibili, ma in base alle funzionalità del sistema.

Parte del problema, che rende difficile l'interoperabilità tra sistemi, è che ognuno ha il suo **set di chiamate** a sistema, i propri formati di file eseguibili, ...

Ci sono alcune soluzioni per scrivere codice che può essere eseguito su più sistemi:

- Scrivere in un **linguaggio interpretato**, come Python, in questo caso è sufficiente che esista un interprete disponibile per quel sistema operativo.
- Scrivere in un linguaggio che **include una VM** all'interno della quale avviene l'esecuzione, come Java.
- Scrivere in un **linguaggio standard**, come C, e compilare poi per ogni sistema.

ABI

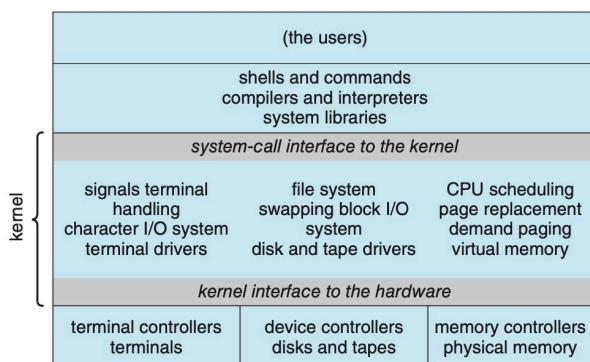
Proprio come le API permettono al programmatore di non dover programmare le implementazioni di semplici funzioni in codice macchina, le ABI forniscono al compilatore il set di istruzioni che possono essere eseguite sulla specifica architettura per cui si sta compilando.

Il problema è che le ABI spesso non forniscono una gran quantità di interoperabilità, quindi il codice deve essere compilato **per ogni sistema operativo e per ogni architettura** su cui verrà eseguito.

3.4 Implementazione di un Sistema Operativo

Tradizionalmente i sistemi operativi venivano scritti in assembly, successivamente vennero utilizzati dei linguaggi specifici e ad oggi si utilizza principalmente C o C++ per scrivere il *kernel*, assieme a frammenti in assembly per gli elementi di basso livello.

Vista la complessità del codice richiesto da un sistema operativo è importante pensare alla struttura da impartire ad esso, vediamo ora le strutture più comune che vengono usate per la creazione dei sistemi operativi.



3.4.1 Design di un Sistema Operativo

Abbiamo già visto come vengono definiti gli obiettivi del sistema operativo, vedendo cosa gli utenti e l'hardware si aspettano da esso.

Nella progettazione di un software complesso come il sistema operativo è importante mantenere separati i due concetti di politiche e meccanismi.

Le **politiche** definiscono i compiti e servizi che il sistema operativo deve fornire, mentre i **meccanismi** definiscono come queste politiche andranno implementate all'atto pratico.

In macos e Windows politiche e meccanismi sono fissati a priori e scritti nel sistema, in Linux la separazione tra le due è più evidente.

3.4.2 Monolitici

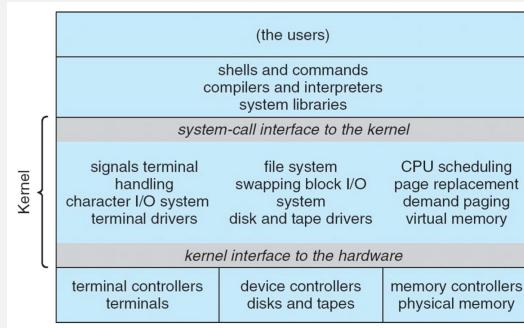
La struttura più semplice è l'**assenza di una struttura**. Tutta la funzionalità del kernel viene implementata in un unico eseguibile, con tutte le funzioni strettamente collegate.

Questo significa che un singolo bug in uno dei sistemi può comportare il blocco dell'intero sistema, inoltre lo sviluppo e l'estensione risultano essere particolarmente complicati per un kernel monolitico.

Tuttavia quando costruito in modo sicuro la stretta integrazione rende il sistema **estremamente efficiente**.

MS-DOS non implementava una struttura modulare, le applicazioni fanno chiamate a system call direttamente, aprendo così la strada a diverse vulnerabilità.

Anche UNIX implementava un kernel parzialmente monolitico, con l'obiettivo di migliorare le performance. Esso presentava solo due livelli, il kernel e le applicazioni utente. Lo strato del kernel si ritrova con una gran quantità di responsabilità.



3.4.3 Approccio stratificato

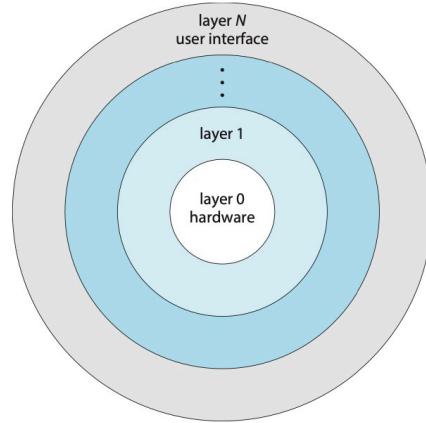
L'alternativa all'approccio monolitico è un approccio dove i vari sistemi sono separati in componenti o moduli relativamente piccoli che assieme compongono il kernel.

Uno dei modi per implementare un approccio modulare è quello stratificato, in presenza di hardware appropriato è possibile suddividere le funzioni del sistema operativo su vari **livelli**. Il livello 0 è l'hardware, il livello N è l'interfaccia utente.

Ciascuno strato impiega esclusivamente funzioni degli strati di livello inferiore.

Questa struttura ha il vantaggio di facilitare la realizzazione e messa a punto del sistema operativo. Gli svantaggi invece sono legati ai tempi di attraversamento dei vari strati (si pensi ad una system call che deve essere intercettata e riportata da molteplici livelli) e alla difficoltà di definire gli strati, in quanto essi possono usare solo le funzioni degli strati inferiori (si pensi al driver della memoria).

virtuale, che dovrebbe essere sopra allo scheduler, perché deve essere possibile interromperne l'esecuzione in caso di page fault. Ma allo stesso tempo lo scheduler deve essere a conoscenza delle informazioni del driver per poter gestire al meglio i processi).



3.4.4 Microkernel

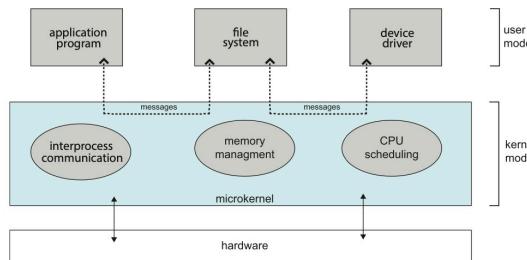
Un microkernel offre la **minima quantità di servizi** di gestione dei processi, della memoria e di comunicazione.

Tutte le funzionalità non essenziali al kernel sono implementate come programmi utente, c'è quindi necessità di intensa comunicazione tra le varie componenti, cosa che viene mediata dal kernel.

Il microkernel è semplice da modificare e rende semplice aggiungere e modificare le funzionalità, in quanto esse sono implementate a livello utente.

Inoltre i sistemi operativi con microkernel sono più sicuri (meno codice viene eseguito a livello kernel) e sono più facili da portare a nuove architetture.

Il più grande svantaggio è l'**overhead** di prestazioni causato dalle continue comunicazioni tra i moduli.



Utilizzato da alcuni sistemi operativi più vecchi, come le prime versioni di Windows NT o macOS su kernel darwin.

3.4.5 Kernel Modulari

L'implementazione di tutti i moderni sistemi operativi utilizza i *Loadable Kernel Module (LKMs)*. Si utilizza un approccio object-oriented dove ciascun modulo implementa un'interfaccia che definisce una funzione del kernel.

Ciascun modulo può comunicare con gli altri moduli mediante l'interfaccia comune, inoltre ogni modulo può essere caricato o meno in memoria in base alle necessità.

Questo approccio è simile a quello del microkernel con solo alcuni moduli caricati inizialmente che hanno il compito di caricare gli altri in base alla necessità, ma vengono risolti i problemi di comunicazione.

3.4.6 Sistemi ibridi

Nei sistemi operativi moderni si utilizza spesso un approccio misto rispetto a quelli visti precedentemente.

Linux

Linux ha un kernel principalmente monolitico per poter fornire prestazioni elevate, tuttavia esso può essere anche esteso in modo dinamico.

Windows

Anche Windows è in larga parte monolitico, ma conserva alcune caratteristiche tipiche dei sistemi microkernel tramite il supporto per sottosistemi (*personalities*) che vengono eseguiti a livello utente.

macOS e iOS

Anche se i sistemi possono sembrare molto differenti e vengono eseguiti su architetture e dispositivi diversi, essi condividono una struttura simile.

- **User experience (GUI):** Permette l'interazione col software, diversa per macOS e iOS in base allo stile di input.
- **Ambienti applicativi:** Cocoa fornisce le API per i linguaggi di programmazione Objective-C e Swift.
- **Framework di base:** Ambienti che supportano le API grafiche e di base, come OpenGL.
- **Ambiente kernel:** Darwin include il kernel BSD UNIX, è estendibile grazie ai kext utili allo sviluppo di driver ed estensioni del kernel.

Le applicazioni su questi sistemi possono essere progettate per sfruttare le funzionalità di user experience o per aggirarle completamente (disponibile ai developer solo su macOS).

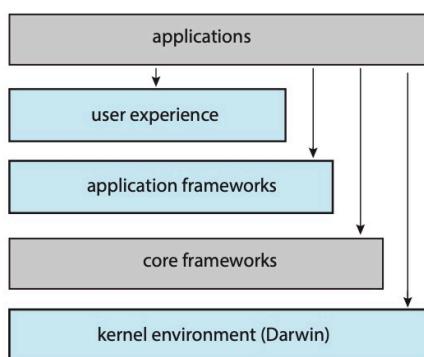


Figura 3.2: Struttura dei sistemi macOS e iOS

Android

Sviluppato dalla *Open Headset Alliance* di cui Google fa parte, ma non è l'unica grande azienda, Android è un sistema operativo open source che gestisce una gran quantità di dispositivi mobili.

Android nasce da una versione modificata del kernel linux, le applicazioni android sono scritte in Java e vengono poi eseguite sull'*Android RunTime (ART)*. Per migliorare le prestazioni

ART non esegue una compilazione a tempo di esecuzione, ma viene eseguita all'installazione dell'applicazione.

Gli sviluppatori possono anche scegliere di utilizzare JNI, l'interfaccia nativa con un accesso più diretto all'hardware.

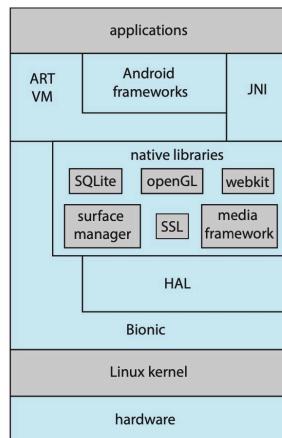


Figura 3.3: Struttura sistema android

Poiché i dispositivi android hanno specifiche hardware pressoché illimitato c'è un livello di astrazione tra l'hardware e le librerie native (*Hardware Abstraction Layer, HAL*)

3.5 Debugging e Tuning

Il **debugging** è l'attività di individuazione e risoluzione dei problemi, il che comprende sia la risoluzione dei bug che il performance tuning.

Il sistema operativo aiuta nel processo con la generazione di **file di log** che danno informazioni sugli errori rilevati durante l'esecuzione. Inoltre il sistema operativo fornisce un **core dump**, ovvero una copia della memoria impiegata dal processo al momento della terminazione anomala.

Kernel

Quando avviene un errore al kernel viene chiamato **crash**. Quando questo avviene viene creato un **crash dump** che viene inserito in un'apposita sezione della memoria.

È necessario di scrivere su un'area riservata di memoria perché quando si ottiene un crash lo stato del sistema non è garantito ed è difficile comprendere quali sezioni della memoria sono libere.

3.5.1 Tuning

Anche i problemi che condizionano le prestazioni sono considerati bachi e vanno quindi trovati e risolti. Il performance tuning è l'insieme delle tecniche atte ad ottimizzare le prestazioni del sistema ed ad eliminare i colli di bottiglia.

Per fare ciò ci sono varie tecniche, è possibile eseguire del codice che misuri le prestazioni del sistema e poi ne salvi i risultati su un file di log.

Oppure possono essere utilizzati degli strumenti grafici del sistema operativo, ad es. task manager di Windows.

Infine esiste anche il profiling che punta a visualizzare quali chiamate a sistema vengono utilizzate maggiormente così da ottimizzarle, quando possibile.

Capitolo 4

Affidabilità

Il capitolo viene svolto alla fine del corso nelle ultime slide, viene posizionato qui in quanto non rientra in nessuno dei macroargomenti del corso, ma riguarda il sistema in generale.

4.1 Guasti

I guasti possono avere una di due caratteristiche rispetto al tempo:

- **Temporanei:** Quindi guasti non sempre presenti in tutte le condizioni. Questi guasti possono essere transitori (avvengono una sola volta) oppure intermittenti (avvengono ad intervalli irregolari).
- **Permanenti:** Guasti che, una volta verificati, rimangono presenti finché il componente non viene sostituito o riparato.

Invece rispetto alla loro importanza definiamo:

- **Guasti significativi:** guasti che degradano significativamente le prestazioni di un sistema.
- **Guasti maggiori:** Impediscono il completamento della missione.

Tolleranza ai Guasti

Una caratteristica importante di un sistema è la sua tolleranza ai possibili guasti che possono emergere, tipicamente richiede una qualche forma di ridondanza, con lo scopo di aumentare l'affidabilità del sistema.

Ridondanza

- **Spaziale:** Richiede l'utilizzo di diversi componenti in grado di svolgere la stessa operazione, in modo da averne uno disponibile in caso di guasto.
- **Temporale:** Ritentare l'operazione quando si incontra un errore, permette di risolvere i guasti temporanei.
- **Informazione:** I dati vengono scritti in modo che possano verificare e correggere i possibili errori.

4.2 Affidabilità

Si definisce affidabilità di un dispositivo la probabilità che esso funzioni correttamente, per un dato tempo, date certe condizioni.

- **Affidabilità logistica:** Si riferisce alla probabilità che nessun guasto si verifichi
- **Affidabilità di missione:** Probabilità che non si verifichino guasti "gravi", ovvero tali da pregiudicare le funzionalità del sistema.
- **Sicurezza:** Probabilità che non si verifichino guasti con conseguenze catastrofiche, tali da produrre danni a persone e cose.

Esempio

dispositivo	Ore di funzionamento	Minuti di riparazione
1	490	140
2	760	130
3	2350	80
4	1400	90
5	1560	110
6	970	150
7	2300	70
8	1190	90
9	1130	110
10	300	120
Somma:	12450	1090

Affidabilità:

$$R(t) = \frac{n_{prod}(t)}{n_{prod}(t_o)}$$

dove $n_{prod}(t)$ è il numero di dispositivi funzionanti al tempo t .

In questo caso $R(1000) = \frac{6}{10} = 0.6$

MTTF (Mean Time to Failure)

$$MTTF = \frac{\sum_{n=1}^N T_n}{N}$$

dove N è il numero di dispositivi

In questo caso $MTTF = \frac{12450}{10} = 1245$ ore

MTTR (Mean Time to Repair)

$$MTTR = \frac{\sum_{n=1}^N R_n}{N}$$

dove N è il numero di dispositivi

In questo caso $MTTR = \frac{1090}{10} = 109$ minuti

Availability

Frazione di tempo in cui il dispositivo funziona correttamente.

$$A = \frac{MTTF}{MTTF + MTTR}$$

In questo caso $A \approx 99.86\%$

Sia la funzione di affidabilità $R(t)$ che quella di guasto $Q(t) = 1 - R(t)$ sono delle distribuzioni di probabilità.

Parte II

Gestione dei Processi

Capitolo 5

Threads e Concorrenza

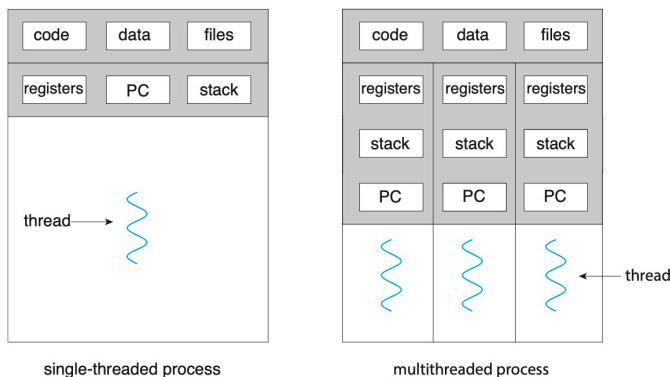
Abbiamo già visto nell'introduzione una prima definizione di processo, l'entità attiva di un programma. Vediamo ora più nel dettaglio l'evoluzione di esso nel tempo.

5.1 Thread

Un thread è l'unità fondamentale della computazione, può essere generato da un processo aprendo la possibilità alla **parallelizzazione** della task.

I thread appartenenti allo stesso processo condividono codice, dati e risorse (ad es. una modifica ad una variabile globale oppure l'apertura di un file sono visibili a tutti i thread)

Queste situazioni vanno però gestite correttamente, altrimenti si rischia di avere conflitti nell'accesso alle risorse.



Il thread comprende:

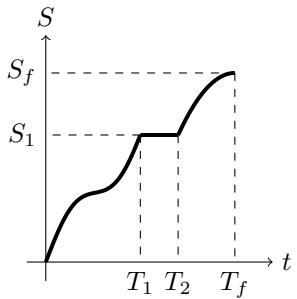
- un identificatore di thread
- un contatore di programma
- un insieme di registri
- uno stack

Il processo tradizionale, che viene eseguito su un singolo processo, si definisce "*heavyweight process*", mentre i thread sono detti anche "processi leggeri".

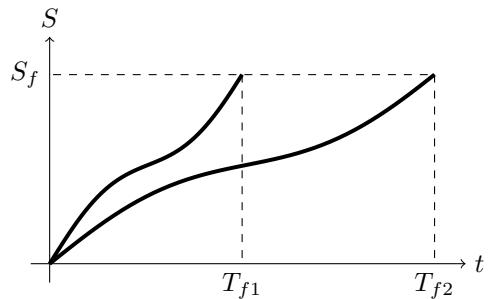
Un thread può essere utilizzato dal browser per rappresentare una singola pagina web, per generare velocemente icone per una serie di immagini oppure possono essere usati da un webserver, un thread per ogni richiesta.

5.1.1 Visualizzazione

Per visualizzare lo stato di un processo in relazione al tempo useremo dei grafici con il tempo sull'asse x, e lo stato del processo sull'asse y.



In questo caso particolare si può notare un periodo di tempo, da T_1 a T_2 dove il processo rimane nello stato S_1 , questo indica che si trovava in un *bottleneck*, ovvero il processo stava attendendo una qualche risorsa.



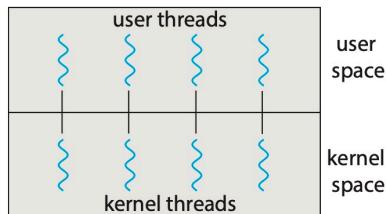
In questo caso lo stesso processo viene eseguito su due processori diversi, T_{f1} è il processore più veloce, mentre T_{f2} quello più lento.

5.1.2 User e Kernel Threads

Nei moderni sistemi operativi esistono threads al livello utente e altri al livello kernel. Quando un thread a livello utente fa una chiamata a sistema è necessario trovare un thread a livello kernel che sia pronto ad accettarla, altrimenti si rischia di introdurre dei tempi di attesa. Va quindi trovata una relazione tra le due tipologie di threads.

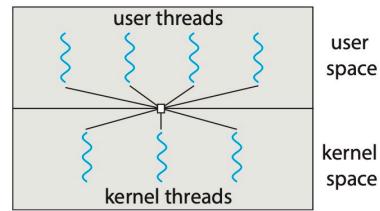
One-to-One

Mappa ogni thread a livello utente con un suo corrispettivo a livello kernel, questo assicura che ogni *syscall* venga gestita correttamente e rapidamente. Tuttavia così facendo si creano molti thread che potrebbero portare ad una riduzione delle prestazioni.



Many-to-Many

Questo modello mappa n thread utente a k thread kernel, dove $n \ll k$. Questo modello ha una migliore utilizzazione delle risorse, ma è di difficile realizzazione la funzione che associa thread utente a thread kernel. Ad oggi nessun sistema operativo moderno lo utilizza, si preferisce usare il modello one-to-one.



5.1.3 Implementazioni

Linux

Linux implementa la Process Control Block mediante una *task struct*, la ready queue diventa quindi una lista concatenata di *task struct*.

Quando un processo crea uno o più figli può scegliere se continuare la sua esecuzione in parallelo con essi o se attendere la loro conclusione.

Inoltre può scegliere se eseguire lo stesso codice del genitore o un altro programma.

In linux, come in molti altri Sistemi Operativi, tutti i processi sono figli di un processo iniziale, nel caso di linux è il processo con ProcessID = 1.

Un nuovo processo viene generato con la funzione `texttfork()` che restituisce l'id del nuovo processo al parent e restituisce 0 al figlio, questo è importante perché permettere di distinguere se ci si trova nel parent o nel children.

Quando il genitore non attende l'esecuzione dei figli e termina prima di essi essi sono detti **orfani** e alla loro terminazione diventano degli **zombie**.

iOS

Nelle prime versioni era previsto un solo processo utente in esecuzione.

A partire da iOS 4 vengono permessi anche dei processi di background, a priorità inferiore rispetto a quello di foregorund che ha il controllo della GUI.

Android

Le applicazioni che vogliono continuare l'esecuzione in background devono utilizzare un servizio.

Quando dei processi devono essere terminati la selezione avviene secondo l'ordine:
processi vuoti -> background (non evidente all'utente) -> di servizio (evidente all'utente) ->
visibile (usati da un processo foregorund) -> foregorund.

Chrome

Google chrome usa un processo di gestione del browser, dell'interfaccia utente, dell'I/O, poi utilizza un processo renderer per ogni pagina di navigazione.

5.2 Programmazione multicore

L'utilizzo di più thread si adatta naturalmente ai sistemi multicore, ma non solo, essa può trovare delle applicazioni anche in sistemi con un solo processore.

5.2.1 Concorrenza

La Concorrenza è l'esecuzione di più attività contemporaneamente, questo può essere ottenuto con un sistema multicore, oppure tramite uno scheduling efficiente dei cicli di un singolo processore.

La concorrenza permette, almeno dal punto di vista dell'utente, di eseguire più processi contemporaneamente anche se questo può non essere vero.

Questo si ottiene tramite una grande quantità di *context switch* da parte della CPU, che dedica ad ogni task una quantità di tempo infinitesimale.

Risulta quindi particolarmente importante scegliere correttamente le task che possono essere eseguite contemporaneamente e come sincronizzare.

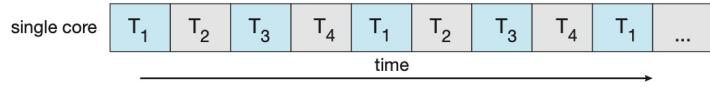


Figura 5.1: Concorrenza su un singolo processore

5.2.2 Parallelismo

Il parallelismo invece è la semplice esecuzione due o più task contemporaneamente. I due thread non devono necessariamente lavorare alla risoluzione dello stesso problema.

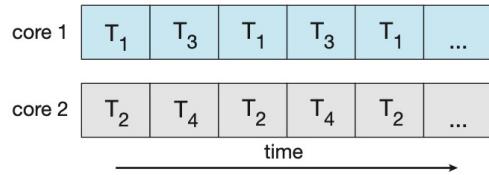


Figura 5.2: Parallelismo su due processori

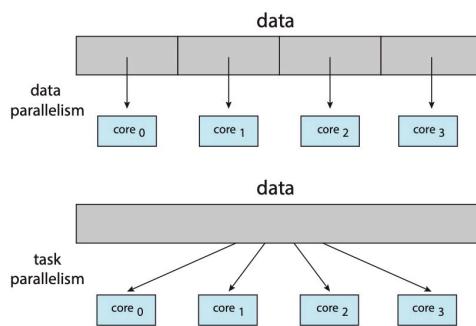
Il parallelismo può essere implementato in vari modi:

- **Parallelismo dei dati:**

La stessa operazione viene eseguita contemporaneamente su processori diversi su segmenti diversi dei dati. (es. per analizzare 10 ore di registrazione, divido in elementi da 1 ora ciascuno)

- **Parallelismo delle task:**

Significa distribuire i threads tra più processori, ogni thread esegue un'operazione differente. I thread possono lavorare sugli stessi dati, ma non è necessario.



5.3 Sistemi di Processi

È spesso conveniente scomporre un processo in n sottoprocessi dando vita ad un sistema di processi, questo può permetterci di parallelizzare le operazioni pur rispettando l'ordine di esecuzione.

Possiamo utilizzare un grafo per rappresentare le relazioni di precedenza tra i processi, i nodi del grafo sono i sottoprocessi e un arco rappresenta una relazione di precedenza.

Esempio 4.1.

Vogliamo risolvere l'equazione $f = (a + b) \cdot (c + d) + e$, possiamo risolverlo:

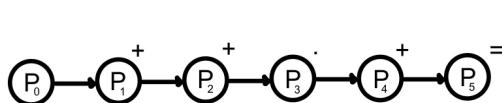


Figura 5.3: In modo sequenziale

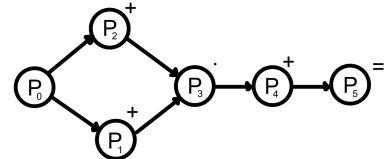


Figura 5.4: In parallelo

Esempio 4.2.

Conoscendo i tempi di esecuzione di ogni singolo processo possiamo anche calcolare la quantità di tempo che viene risparmiata parallelizzando l'operazione.

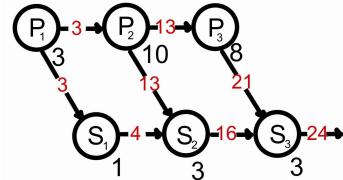
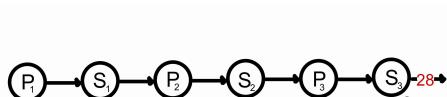
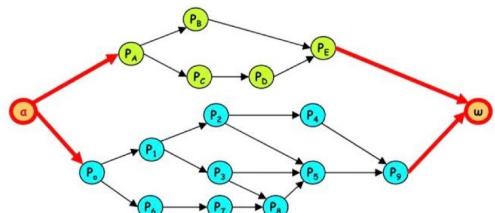
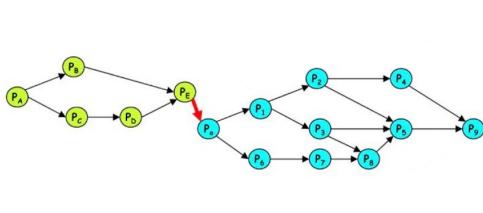


Figura 5.5: Aumento di efficienza ottenuto parallelizzando il processo

5.3.1 Sistemi Chiusi e Aperti

Un sistema si definisce **chiuso** se esiste un singolo processo iniziale e un singolo processo finale. Il sistema è **aperto** se non è chiuso.

Due sistemi di processi possono essere combinati in **serie** o in **parallelo**.



I grafi ci permettono di trovare anche il **massimo grado di parallelismo** che un sistema raggiunge, si può trovare osservando il numero di processi nella stessa colonna.

Nel caso rappresentato qui sopra ci sono al massimo 3 processi ponendo i sistemi in serie e 5 processi ponendo i sistemi in parallelo.

5.3.2 Determinatezza

Un sistema si dice **determinato** se le diverse velocità dei processi e l'ordine di esecuzione dei processi non influenzano il risultato. Altrimenti si dice che il sistema è **indeterminato**.

5.3.3 Inferenza

Possiamo associare ad ogni processo un **dominio** contenente i dati su cui esso lavora e un **codominio** che contiene i risultati del processo.

La **funzione mappa** trasforma elementi del dominio in elementi del codominio.

Due processi si dicono **non inferenti** se almeno una delle due osservazioni è corretta:

- Uno è il successore all'altro
- Non si intersecano i condomini e neppure i domini con i condomini.

Proprietà

- Condizione necessaria e sufficiente affinché un sistema sia **determinato** è che sia composto da processi **non inferenti**.
- Due sistemi sono **equivalenti** se sono costituiti dallo *stesso insieme di processi*, sono *determinati* e partendo dallo *stesso input* producono lo *stesso output*.

5.4 Risorse

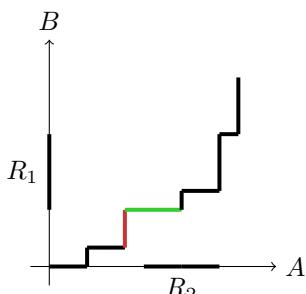
Diciamo **Risorsa** un'entità astratta, necessaria ad un processo per svolgere il proprio lavoro. Quando la risorsa non è disponibile il processo dovrà attendere per utilizzarla.

Alcune risorse che sono **condivisibili** e possono essere usate da più processi in parallelo. Mentre alcune risorse possono essere usate da un solo processo allo stesso tempo.

Un'altra classificazione delle risorse separa quelle non consumabili (es. un core), mentre altre sono **consumabili** (es. un dispositivo a batteria).

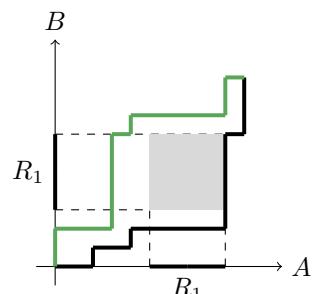
Utilizziamo il termine **preemption** per definire la rimozione forzata di una risorsa ad un processo.

5.4.1 Grafico dei Processi



In questo caso i due processi A e B si alternano. Nel tratto rosso viene eseguito il processo B, mentre A è in attesa. Nel tratto verde accade l'opposto, il processo B che attende in favore di A.

Le risorse R_1 e R_2 sono risorse non condivisibili, A usa R_1 e B R_2 .



In questo caso invece i due processi devono condividere la stessa risorsa R_1 , questo significa che il grafico non può mai entrare nell'area grigia.

I due percorsi, nero e verde, sono entrambi validi per l'esecuzione dei due processi, entrambi passano al di fuori dell'area vietata.

5.5 Deadlock

Quando due processi richiedono la stessa risorsa non condivisibile, uno dei due attende l'altro.

In alcuni casi però si può arrivare ad una situazione più complicata, ovvero quando due o più processi si attendono a vicenda, questo viene detto **deadlock** o **stallo**.

Le condizioni necessarie (ma non sufficienti) per il verificarsi di un **deadlock**:

1. Tutte le risorse devono essere non condivisibili.
2. Il processo non ha un'unica operazione in cui richiede delle risorse.
3. Al processo non può essere rimossa la risorsa, solo esso stesso può rilasciarla.
4. L'attesa avviene in modo circolare, se ci sono n processi, P_1 attende $P_2, \dots, P_{(n-1)}$ attende P_n e P_n attende P_1 .

In questo caso ogni processo attende una risorsa che non verrà rilasciata, tutti attendono e nessuno può procedere.

Livelock

Un sottoinsieme dei deadlock sono i livelock o stalli attivi, rappresenta una situazione dove i thread non sono effettivamente bloccati, ma effettivamente non progrediscono (es. due persone che si incontrano e si spostano ripetutamente da un lato all'altro del corridoio)

5.5.1 Grafo delle Risorse

Una visualizzazione utile per evidenziare possibili deadlock in fase di progettazione è il grafo delle risorse.

I nodi del grafo rappresentano thread (T_1, T_2, \dots, T_n) e risorse (R_1, R_2, \dots, R_n), spesso vengono visualizzati in modo differente per distinguerli.

Diciamo *arco di richiesta* $T_i -> R_j$ e un *arco di assegnazione* $R_i -> T_j$

A questo punto possiamo dire che se il grafo non contiene cicli chiusi allora non ci saranno deadlock. Quando invece il grafo contiene un ciclo e vi è una sola istanza per ogni risorsa allora ci sarà un deadlock, se almeno una risorsa ha più di un'istanza allora il deadlock è solamente possibile.

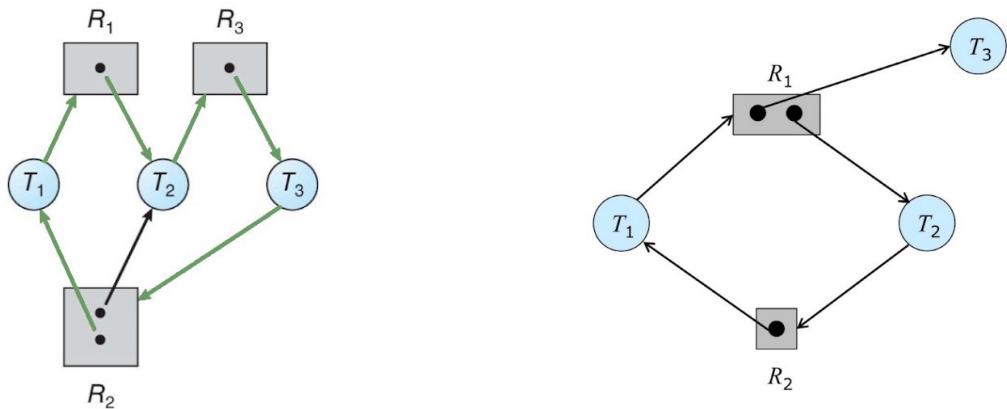


Figura 5.6: Situazione di deadlock

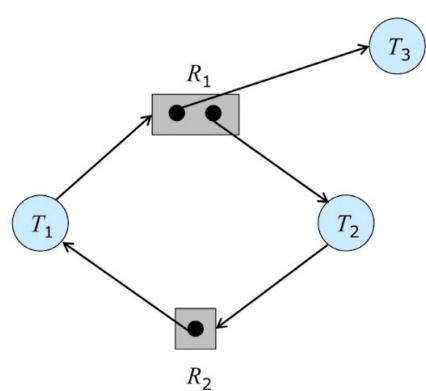


Figura 5.7: Situazione priva di deadlock

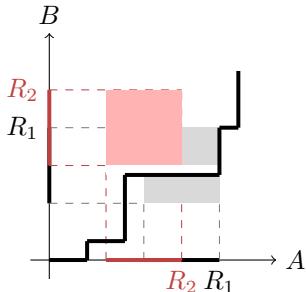
5.5.2 Prevenzione dei Deadlock

Allocazione globale

Tutte le risorse richieste da un processo devono essere allocate prima che inizi l'esecuzione. Questo garantisce l'assenza di deadlock, tuttavia è estremamente inefficiente reclamare tutte le risorse per tutto il periodo di esecuzione.

Allocazione gerarchica

Si assegna un ordine di importanza alle risorse, un processo può richiedere solo risorse più elevate di quelle che ha attualmente. Se un processo necessita di una risorsa di priorità inferiore deve rilasciare quella attualmente in suo possesso per poi tornare a richiederla.



Algoritmo del Banchiere

Si impone che il thread T_i inserisca nel grafo delle risorse tutti gli **archi di reclamo** a lui necessari prima di iniziare l'esecuzione.

Un arco di reclamo di un thread verso una risorsa indica al sistema che il thread utilizzerà in futuro quella risorsa. Un arco di reclamo diventa poi arco di richiesta in futuro.

Definiamo **sequenza sicura** un ordinamento di n processi tali per lui le richieste di P_i siano soddisfacibili dai processi P_j con $j < i$

Uno **stato sicuro** è uno stato che contiene almeno una sequenza sicura.

Per implementare l'algoritmo del banchiere vanno mantenute diverse strutture dati:

- Un **vettore** V che contiene il numero di istanze disponibili per ogni risorsa.
- Una **matrice** M che memorizza la massima quantità di istanze per ogni risorsa che viene usata dal thread.
- Una **matrice** R che memorizza le istanze per ogni risorsa attualmente allocate ad ogni thread.
- Una **matrice** $N = M - R$, che memorizza le risorse mancanti ad ogni thread.

Tutte queste matrici devono essere aggiornate sia nei valori che nelle dimensioni.

Per verificare che lo stato corrente sia sicuro deve essere possibile trovare almeno una sequenza di processi tale per cui sia possibile assegnare le risorse del vettore per completare ogni processo (le risorse necessarie al processo i si trovano su $N[i]$)

Algoritmo dello Struzzo

L'algoritmo del banchiere deve essere eseguito frequentemente per poter rilevare i deadlock prima che si rivelino problematici, questo però richiede parecchie risorse del sistema.

Per questo motivo molti sistemi *consumer non* gestiscono i deadlock, essi avvengono in media una volta l'anno e in quel caso il sistema dovrà essere riavviato.

5.5.3 Risoluzione di un Deadlock

Una volta rilevato un deadlock per risolverlo ci sono due strategie spesso utilizzate.

Terminazione dei Processi

- Terminazione di tutti i processi che fanno parte del deadlock.
Un sistema che risolve sicuramente il deadlock, ma viene anche ad un gran costo computazionale in quanto tutti i processi dovranno venir eseguiti da zero.
- Terminazione di un processo per volta finché il deadlock non viene rotto.

La scelta del processo da terminare è anch'essa gestita da un algoritmo che prende in considerare molti fattori, come la priorità del thread, il tempo di computazione trascorso, quantità di risorse che utilizza.

Prelazione di Risorse

Per risolvere il deadlock è anche possibile rimuovere risorse forzatamente ad uno o più processi. Alcuni fattori da considerare quando si applica questa strategia sono:

- **Selezione della vittima**, è necessario scegliere un processo a cui rimuovere le risorse.
- **Rollback** dopo la rimozione della risorsa il thread non può continuare la sua esecuzione, è necessario tornare indietro a quando quella risorsa è stata acquisita.
- **Starvation** è importante non prelare le risorse sempre dallo stesso processo, altrimenti si potrebbe arrivare alla situazione in cui un processo non procede mai.

Capitolo 6

Sincronizzazione

Quando due o più thread vengono eseguiti in parallelo devono essere sincronizzati correttamente per evitare l'insorgenza di race condition.

6.1 Race Condition

Quando più processi accedono in concorrenza e modificano i dati condivisi l'esito finale dell'esecuzione è indeterminato. Dipende dall'ordine in cui le loro istruzioni vengono eseguite. Questa situazione viene chiamata **race condition** e va gestita per evitare di ottenere risultati erronei.

Esempio

```
// Thread 1                                // Thread 2
for (int i = 0; i < 5; i++) {                for (int j = 0; j < 5; j++) {
    x++;                                    x--;
}
```

In questo caso, se i due thread vengono eseguiti concorrentemente, il valore finale di *x* non è noto, potrebbe variare tra -5 e +5.

Operazioni Atomiche

Per risolvere il problema della *race condition* è necessario svolgere le operazioni di lettura e modifica delle variabili condivise senza interruzioni. Queste operazioni sono definite **atomiche**.

6.2 Regioni Critiche

Definiamo **Regioni Critiche** di un processo quelle regioni dove si accede ai dati in comune. Ci si deve quindi assicurare che solo un processo per volta possa entrare nella sua sezione critica.

È importante gestire correttamente la sincronizzazione dei processi, ogni processo deve richiedere il permesso di entrare nella sezione critica e deve attendere che questo permesso gli sia garantito.

Il protocollo di cooperazione tra processi deve garantire:

- **Mutua Esclusione:** un solo processo nella sezione critica per volta
- **Progresso:** Quando uno o più processi richiedono di entrare nella sezione critica la decisione deve essere fatta solo da processi che stanno già eseguendo la loro sezione critica o che potrebbero eseguirla in futuro. Ricordando sempre che la decisione non può essere rimandata indefinitamente.
- **Attesa Limitata:** Per evitare la *starvation* è necessario limitare il tempo massimo di attesa.

Kernel

Anche le strutture dati del kernel, comprese quelle incaricate di gestire le risorse condivise, sono a loro volta suscettibili a race condition.

È possibile risolvere questo problema utilizzando un kernel senza diritto di prelazione, quindi con processi che non escono dalla modalità kernel finché essi non si bloccano o restituiscono volontariamente il controllo alla CPU.

Questa è la soluzione utilizzata da Windows XP e Windows 2000, ma è particolarmente inefficiente per i sistemi multiprocessore e real-time. Linux supporta la prelazione a partire dalla versione 2.6

6.3 Lock

In generale qualsiasi soluzione al problema della sezione critica utilizza il **lock**. Per accedere ad una sezione critica il processo deve acquisire un lock, che restituisce poi all'uscita dalla sezione critica.

```
{  
    //acquisisce il lock  
    sezione critica  
    //restituisce il lock  
    sezione non critica  
}
```

Supporto Hardware

Molte architetture forniscono il supporto hardware per implementare efficacemente i lock, inoltre alcune forniscono anche direttamente delle istruzioni atomiche.

Supporto Sistema Operativo

Dato che le implementazioni hardware sono spesso inaccessibili ai programmati, il sistema operativo fornisce degli strumenti per risolvere il problema.

Il più semplice è il **lock mutex**, permette di proteggere le sezioni critiche aggiungendo `acquire()` prima del segmento e `release()` al termine

```
acquire() {  
    while(!available) {}; // busy wait  
    available = false;  
}  
  
release() {  
    available = true;  
}
```

6.4 Semafori

I semafori sono uno strumento più avanzato per gestire la cooperazione di più processi.

Un semaforo a N valori è una variabile s intera e non negativa sulla quale si può effettuare:

- **Wait:** decrementa s di 1 se $s > 0$, altrimenti sospende il processo. (Oppure *Probeer te verlagen*, P)
- **Signal:** Risveglia uno dei processi in attesa, se nessuno sta attendendo incrementa s di 1. (Oppure *Verhogen V*)

```
wait(S) {  
    while(S <= 0) {}; // busy wait
```

```

        S--;
    }

signal(S) {
    S++;
}

```

Definiamo **Semaforo Contatore** il semaforo a valori in un dominio non limitato. Utile nel controllo di un numero finito di risorse, il semaforo viene inizializzato al numero di risorse disponibili. **Semaforo Binario** assume solamente i valori 0 e 1, simile al lock mutex.

6.4.1 Implementazione

L'utilizzo del busy waiting rappresenta un problema per un processo multiprogrammato perché l'attesa spreca cicli di CPU che potrebbero essere usati da un'altro processo.

Per risolvere questo problema si può associare ad ogni semaforo una coda di processi in attesa, spesso implementata tramite una queue (First In First Out).

```

struct semaphore {
    int value;
    struct process* list; //puntatore al primo elemento della struttura dati
};

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}

```

In questo caso *valore* può essere < 0, in quelle situazioni $|valore|$ è il numero di processi in attesa.

6.4.2 Problemi di Utilizzo

I semafori possono dare luogo a situazioni inaspettate che devono essere gestite attentamente.

Deadlock

Si possono trovare situazioni in cui uno o più processi attendono risorse che possono essere liberate solo da uno degli altri processi in attesa.

<pre> wait(S); wait(Q); ... signal(S); signal(Q); </pre>	<pre> wait(Q); wait(S); ... signal(Q); signal(S); </pre>
--	--

Linux fornisce uno strumento, lockdep, per rilevare i deadlock all'acquisizione di un lock.

Starvation

Gestendo incorrettamente la lista associata al semaforo, per esempio utilizzando una struttura dati LIFO, è possibile che un processo attenda all'infinito nella coda senza mai venir rimosso.

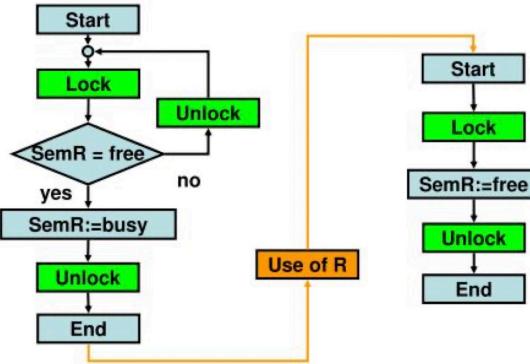


Figura 6.1: Visualizzazione di un semaforo

6.5 Monitor

I semafori sono spesso utilizzati per sincronizzare l'utilizzo delle risorse, tuttavia il loro utilizzo incorretto può generare errori difficili da individuare.

Un monitor è una struttura di più alto livello che contiene una serie di procedure e dei dati (condivisi tra i processi). La caratteristica di un monitor è che al suo interno un solo processo per volta può essere attivo.

```

monitor monitor_name {
    /* dichiarazioni delle variabili condivise */
    function P1 () {}
    function P2 () {}
    ...
    function PN () {}
}

```

6.5.1 Variabili Condition

Le funzionalità appena descritte non sono sufficienti per modellare gran parte degli schemi di sincronizzazione.

Per questo vanno introdotte le variabili condition, da inserire tra le variabili condivise, sulle quali si possono effettuare solamente le operazioni di `wait()` e `signal()`.

Un processo può usare una variabile condition per attendere il verificarsi di una determinata condizione, uscendo dal monitor nell'attesa per lasciarlo libero ad altri processi.

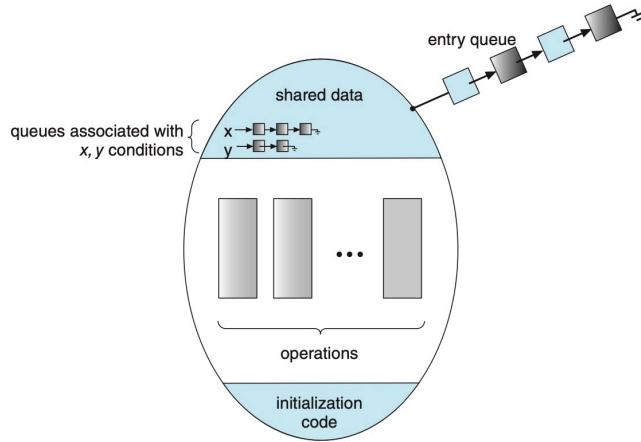


Figura 6.2: Monitor con variabili condition

6.5.2 Gestione di signal

Quando un processo P invoca `signal()` su una risorsa che è attesa da un altro processo Q si ottiene una situazione che può essere gestita in due modi:

- **Segnalare e attendere:** Il processo P , dopo la chiamata a `signal` esce dal monitor e lascia che Q esegua la sua parte di codice.
- **Segnalare e procedere:** Il processo P completa la sua esecuzione e poi Q viene eseguito.

La seconda opzione, segnalare e procedere, sembra essere più ragionevole finché non si considera che la condizione che P segnala potrebbe non essere più valida quando Q arriva ad essere eseguito.

6.6 Implementazione: Linux

Prima della versione 2.6 (2004) Linux utilizzava un kernel senza diritto di prelazione. Per ottenere delle sezioni critiche venivano disabilitati momentaneamente gli interrupt.

Ad oggi Linux fornisce diversi strumenti per la sincronizzazione:

- **Lock Mutex:** Una variabile booleana che indica se il lock è disponibile, modificata tramite chiamate a sistema.

```
#include <pthread.h>
/* create and initialize the mutex lock */
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);

pthread_mutex_lock (&mutex); /* acquire the mutex lock */
/* critical section */
pthread_mutex_unlock (&mutex); /* release the mutex lock */
```

- **Spinlock:** Un ciclo di busy waiting che verifica periodicamente se il lock è libero. È particolarmente inefficiente per i sistemi single core, in quanto altri processi non vengono eseguiti e quindi la risorsa attesa non viene sbloccata. Mentre può fornire alcuni vantaggi su sistemi multicore in quanto il processo non esce dall'esecuzione.
- **Semafori:** Ne esistono di due tipi, con e senza nome. I semafori con nome sono condivisi da tutti i processi imparentati, mentre quelli senza nome possono essere utilizzati solo nel contesto del processo.

```

#include <semaphore.h>
/* Create the semaphore and initialize it to 1 */
sem_t *sem;
sem = sem_open ("SEM", O_CREAT, 0666, 1);

sem_wait (sem); /* acquire the semaphore */
/* critical section */
sem_post (sem); /* release the semaphore */

```

- **Interi atomici:** Un tipo le cui operazioni sono tutte definite in modo atomico. Essi non risentono dell'overhead introdotto dai tradizionali meccanismi di lock.

```

atomic_t counter;
atomic_set (&counter, 5); /* counter = 5 */
atomic_add (10, &counter); /* counter = 15 */
atomic_sub (4, &counter); /* counter = 11 */
atomic_inc (&counter); /* counter = 12 */
int value = atomic_read (&counter); /* value = 12 */

```

- **Variabili condition:** Sono le variabili utilizzate per la realizzazione di monitor, forniscono un wait e un signal che gestiscono in autonomia un lock.

Il wait acquisisce periodicamente il lock, verifica la condition e poi libera il lock.

Capitolo 7

Scheduling CPU

I processi vengono sempre eseguiti finché non richiedono di attendere un'altra risorsa. A questo punto un sistema semplice attende la risorsa, sprecando cicli della CPU, con la multiprogrammazione puntiamo a ridurre i cicli sprecati al minimo.

7.1 CPU scheduler

Abbiamo visto in 2.3.1 il lifecycle di un processo, lo scheduler della CPU deve prendere una decisione in queste situazioni:

1. **Suspend:** Avviene principalmente come risultato di una richiesta di I/O, è quindi richiesto dal processo stesso.
2. **Resign:** Il processo viene forzatamente interrotto dallo scheduler.
3. **Resume:** La condizione attesa diventa vera e il processo è pronto ad utilizzare nuovamente risorse di sistema.
4. **Termina**

Uno scheduler che effettua delle decisioni solo nel caso 1 e 4 si dice **senza diritto di prelazione**, il processo quindi viene eseguito fino al termine o finché non restituisce volontariamente il controllo al sistema operativo.

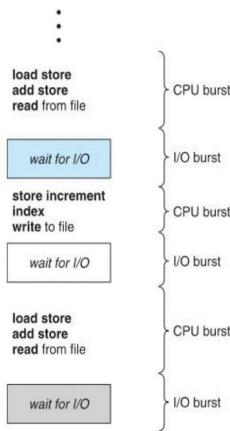
Gli scheduler dei sistemi operativi moderni sono tutti **con diritto di prelazione**, effettuano quindi tutte e 4 le decisioni per portare a migliori prestazioni.

I kernel con diritto di prelazione devono gestire anche tutte le situazioni critiche:

- Possibili **Race condition** quando avviene l'accesso a dati condivisi tra processi.
- La possibilità che anche i processi del kernel vengano prelati interrompendo così operazioni cruciali effettuate dal sistema.

7.1.1 Implementazione

Il massimo utilizzo delle risorse si ottiene implementando lo scheduler in modo tale che ogni processo esegua continui cicli di CPU-I/O burst.



Ogni algoritmo di scheduling ha diversi criteri che prova ad ottimizzare, alcuni di questi sono:

- **Utilizzo della CPU**
- **Throughput:** Numero di processi che completano la loro esecuzione nell'unità di tempo.
- **Tempo di turnaround:** Tempo impiegato per l'esecuzione di un determinato processo
- **Tempo di attesa** nella ready queue
- **Tempo di risposta:** il tempo che intercorre tra una richiesta e una risposta.

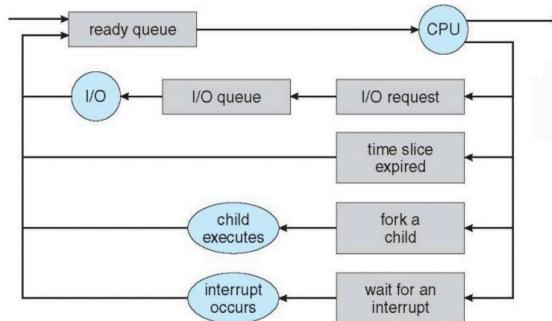


Figura 7.1: Diagramma di scheduling

7.1.2 Meccanismo di Interruzione

Con **Meccanismo di Interruzione** intendiamo il meccanismo messo a disposizione dell'elaboratore per salvare lo stato del processo interrotto per poi trasferire il controllo ad una subroutine che gestisce l'interruzione.

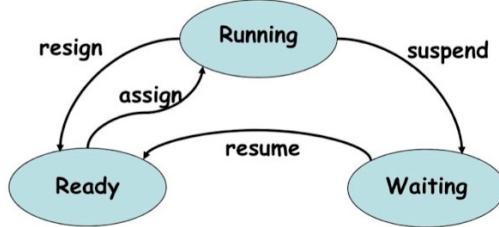
Il kernel del sistema è composto da 3 sezioni, il *First Interrupt Handler (FLIH)*, il dispatcher e l'implementazione degli strumenti di sincronizzazione.

First Interrupt Handler

Il *First Interrupt Handler (FLIH)* ha il compito di determinare l'origine dell'interruzione e attivare il segmento di codice che la gestisce.

Nel caso di un sistema che implementa la multiprogrammazione ci possono essere più processi in attesa, è quindi necessario costruire una coda delle interruzioni.

La coda delle interruzioni funziona assieme a la coda dei processi pronti ad eseguire, quando un processo esce dalla coda delle interruzioni viene inserito tra gli altri processi pronti in attesa di risorse computazionali.



Implementazione Strumenti di Sincronizzazione

Le primitive wait e signal sono implementate dal kernel in quanto possono essere utilizzate da tutti i processi, wait deve poter avere accesso diretto al dispatcher per comunicarci.

Ogni semaforo ha una coda, in genere FIFO, che contiene tutti i processi che si sono fermati sul semaforo.

Non tutti i semafori sono implementati tramite FIFO, essi in generale possono avere diverse implementazioni e organizzazione.

7.1.3 Dispatcher

Il dispatcher ha il compito di assegnare le risorse di computazione al processo selezionato dallo scheduler. Esso gestisce il context switch tra vari processi.

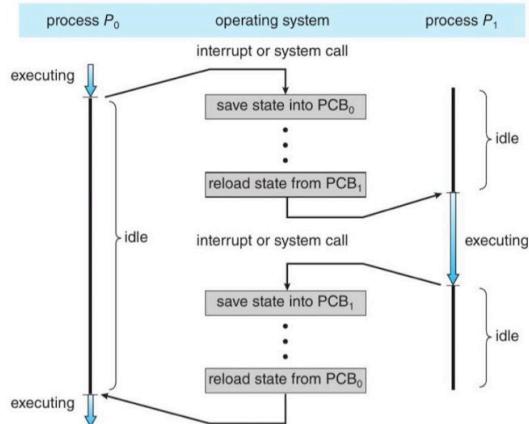


Figura 7.2: Context switch $P_0 \rightarrow P_1 \rightarrow P_0$

Definiamo **Latenza di dispatch** il tempo impiegato dal dispatcher per scambiare un processo con un’altro. Nello schema qui sopra è il periodo in cui entrambi i processi sono in *idle*.

7.1.4 Scheduler ad alto livello

Il compito di attribuire il livello di priorità dei processi non è compito del dispatcher, ma dello scheduler di alto livello.

È possibile utilizzare un heap, così da avere buone prestazioni di inserzione ($O(\log_2 n)$) e rendere banale l'estrazione del processo a priorità maggiore.

7.2 Comunicazione tra Processi

La comunicazione tra processi è necessaria in diverse situazioni e viene implementata mediante diverse tecniche:

- **Scambio di Messaggi:** Richiede un overhead da parte del sistema operativo, ma non presenta conflitti ed è di più semplice utilizzo.
- **Memoria Condivisa:** Richiede l'intervento del kernel solo nell'allocazione iniziale, tuttavia gli accessi successivi devono essere gestiti direttamente dai processi.
- **Pipes Ordinarie:** Permettono la comunicazione unidirezionale tra due processi (produttore-consumatore).
- **Pipes Nominated:** Permettono la comunicazione bidirezionale a molteplici processi, questo avviene grazie ad un file di tipo speciale. Sono supportate sia da Unix che da Windows.

7.3 Algoritmi di Scheduling

7.3.1 Selezione

Prima di vedere alcuni degli algoritmi più utilizzati dai sistemi operativi moderni vediamo il metodo che viene utilizzato per confrontare e migliorare gli algoritmi.

Modelli Deterministici

Nella valutazione analitica, si seleziona un set predeterminato di processi, e si simulano più algoritmi calcolandone il tempo medio di attesa nei vari casi.

È possibile utilizzare la **Formula di Little** per avvicinarsi di più alla realtà: Sia n la lunghezza media della coda, W il tempo medio di attesa e λ il numero di processi in arrivo per unità di tempo. Allora $n = W \cdot \lambda$.

Simulazione

La simulazione implica la creazione di un modello del sistema di calcolo e un processo di logging dei dati.

7.3.2 Algoritmi Comuni

First Come First Serve

Un semplice algoritmo che fornisce le risorse al primo processo che le richiedono. Il criterio poi segue l'implementazione della coda FIFO, quando le risorse si liberano vengono assegnate al primo processo della coda.

Questa strategia non è efficace per i computer consumer in quanto un processo molto lungo può fermare l'esecuzione di altri processi brevi che l'utente richiede con maggiore urgenza.

Highest Priority/Shortest Job First

Viene fornito un valore di priorità ad ogni processo, le risorse vengono assegnate al processo con la priorità più alta.

Questo algoritmo presenta spesso il problema della *starvation* dove i processi a priorità bassa non vengono mai eseguiti, ma introducendo l'*aging* la priorità di un processo aumenta all'aumentare del suo tempo di attesa.

Round Robin o Scheduling Circolare

A ciascun processo vengono assegnate le risorse per un periodo di tempo limitato, solitamente 10-100 ms (un periodo più breve ha troppo overhead di switching, con uno più grande si ottiene un algoritmo FIFO)

Questo algoritmo ha in media un tempo medio di attesa maggiore di un algoritmo basato sulla priorità, ma ha un miglior tempo di risposta, cosa importante nei sistemi consumer.

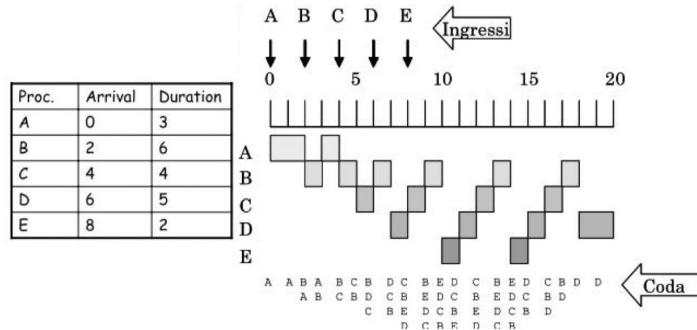


Figura 7.3: Esempio di esecuzione dell'algoritmo

Spesso negli esercizi capita che un nuovo processo venga inserito nella coda ready allo stesso momento in cui un processo termina il suo quanto di tempo. Si incorre quindi in una situazione dove non è chiaro quale dei due processi verrà inserito prima nella ready queue.
In questo caso è possibile scegliere la soluzione che si preferisce, con l'unico limite di restare consistenti all'interno dell'esercizio.

Highest Response Ratio

Definiamo un valore da utilizzare come priorità, la priorità del processo P_i sarà $p_i = \frac{a_i + e_i}{e_i}$. Dove a_i è il tempo di attesa del processo nello stato ready e e_i è il tempo di esecuzione stimato.

Questo significa che i processi brevi avranno una priorità maggiore, ma implementa anche l'*aging* per evitare la *starvation*.

Code Multiple

Come spesso accade la migliore soluzione vede l'utilizzo di più algoritmi allo stesso tempo, è possibile infatti costruire più code, tutte con algoritmi di gestione diversi. Poi sarà sufficiente allocare una certa percentuale di risorse ad ognuna delle code.

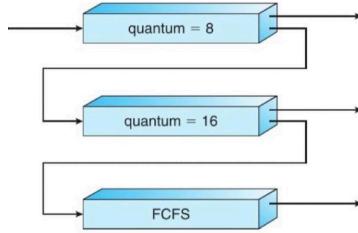
Esempio 1.

Questa strategia si può ad esempio utilizzare per dividere i processi che eseguono in foreground e quelli in background. La coda in foreground avrà maggiore disponibilità di risorse e sarà gestita in Round Robin per assicurarsi un ottima esperienza utente, quella background avrà meno risorse a disposizione e sarà gestita in FIFO.

Esempio 2.

Le code multiple si possono utilizzare anche per implementare l'*aging*, ad esempio possiamo implementare 3 code nel seguente modo:

La prima e la seconda in Round Robin a 8 ms ed a 16 ms e l'ultima in FIFO. In questo modo i processi ricevono tutti rapidamente 8 e poi 16 ms di tempo di esecuzione, risolvendo così i processi semplici, mentre i processi più lunghi finiscono nella coda FIFO e verranno eseguiti quando possibile.



Scheduling con Priorità Proporzionale alla frequenza

Un caso tipico industriale è quello in cui i processi devono essere eseguiti ad intervalli regolari. In questo caso è conveniente definire una priorità proporzionale alla frequenza. Quindi processi che utilizzano la CPU con più frequenza avranno priorità maggiore.

Questo è un algoritmo ottimale, quindi se esso non riesce a pianificare l'esecuzione di una serie di processi rispettando i vincoli temporali allora nessun altro algoritmo ci riuscirà.



È possibile calcolare a priori se i processi potranno essere eseguiti rispettando i vincoli tramite la seguente formula:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m \cdot (2^{\frac{1}{m}} - 1)$$

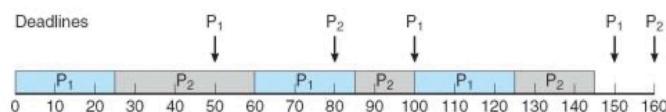
dove m è il numero di processi P_i il periodo del processo i e C_i la sua durata.

Earliest Deadline First

Assegna le priorità dinamicamente a seconda delle scadenze. Ogni processo per poter essere eseguito deve annunciare la sua prossima scadenza allo scheduler, più questa scadenza si avvicina più la priorità del processo aumenta.

Esempio

Siano due processi P_1 e P_2 con periodo di 50 e 80 e con tempo di esecuzione di 25 e 35.



Dalla figura si può vedere come a t=100 il processo P_1 interrompe P_2 in quanto la sua deadline è a t=150 mentre P_2 ha come deadline t=160

7.4 Scheduling Multiprocessore

Il problema dello scheduling diventa più complesso quando nel sistema di calcolo sono presenti più processori.

Multielaborazione asimmetrica

Un singolo processore, detto *master server* ha il compito di gestire lo scheduling, l'I/O e le altre attività di sistema.

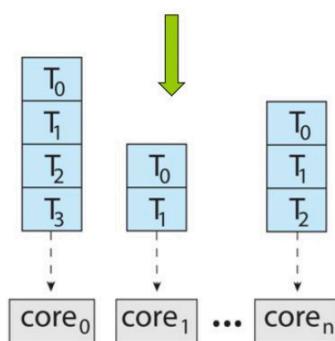
Questo riduce la necessità di condividere dati, un solo processore deve accedere alle strutture dati.

Multielaborazione simmetrica

Quando più processori lavorano su strutture dati comuni è possibile incorrere in delle situazioni critiche, ad esempio quando due processori non possono eseguire lo stesso processo e nessun processo deve subire *starvation*.

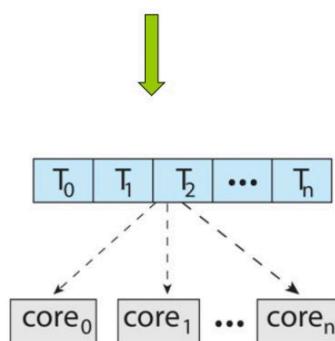
Ci sono due filosofie sull'implementazione di questo tipo di scheduling:

- Una coda per ogni processore, questo ha problemi quando è necessario ribilanciare le code in quanto l'overhead in questo caso è significativo.



In questo caso risulta essere importante ripartire uniformemente il carico tra i vari processori. Un controllo delle lunghezze delle code avviene sia periodicamente sia quando un determinato processore diventa inattivo.

- Una singola coda per tutti i processori, questo risolve il problema del bilanciamento, ma genera delle race condition nell'accesso alla struttura dati.



Esistono alcuni processi che hanno una predilezione per essere eseguiti sempre nello stesso core, questo permette loro di sfruttare al meglio la cache, ma non è sempre supportata dal sistema operativo.

Bilanciamento del Carico

È il procedimento che permette di ripartire uniformemente il carico di lavoro tra più processori, può essere implementata in due modi:

- **Migrazione Guidata:** Un processo dedica controlla periodicamente il carico dei processori per correggere anticipamente gli equilibri
- **Migrazione Spontanea:** Un processore inattivo sottrae ad uno sovraccaricato un processo tra quelli in attesa.

7.4.1 Implementazioni

Linux

Prima della versione 2.5 il kernel Linux implementava delle code multiple gestite perlopiù in Round Robin. La priorità viene calcolata periodicamente in base a quante risorse un processo ha consumato dall'ultimo controllo, minori sono le risorse utilizzate, maggiore sarà la priorità.

Dalla versione 2.6.23 il kernel Linux implementa due code, *default* e *real-time*, con uno scheduler *CFS* (*Completely Fair Scheduler*) che assegna ad ogni task una percentuale del tempo di CPU. A ciascuna task viene associata una priorità (-20 a +19)

Lo scheduler associa ad ogni task il suo tempo di esecuzione virtuale, una quantità che tiene conto sia del tempo di esecuzione che della priorità della task. Quindi una task a bassa priorità avrà un **vruntime** maggiore di una a priorità più alta, anche se sono state entrambe eseguite per lo stesso tempo. Lo scheduler poi esegue la task con il valore **vruntime** più basso.

Lo scheduler Linux è ottimizzato anche per sistemi NUMA, implementando delle strategie per ridurre al minimo il numero di migrazione dei processi da un processore all'altro.

Le CPU Alder Lake causano non pochi problemi allo scheduler Linux < 5.16, infatti esso non si aspetta di vedere processori con livelli di prestazione diversi, ottimizzati per diverse situazioni.

Windows 10

Implementa livelli di priorità da 0 a 31, ognuno con la sua coda. Lo scheduler poi seleziona l'ultimo processo dalla coda a priorità più alta non vuota.

I thread che hanno una priorità compresa tra 0 e 15 vengono gestiti dinamicamente, quando uno di essi viene portato in foreground, oppure riceve un input dall'utente, la sua priorità viene aumentata.

Inoltre la priorità dinamica viene diminuita di un livello ogni volta che il processo accede alle risorse della CPU, fino a tornare alla priorità base.

Windows 11

Windows 11 è stato scritto pensando alle nuove CPU intel, le quali hanno dei core ad alte prestazioni e degli altri ad alta efficienza.

Lo scheduler del sistema interagisce attivamente con un microcontrollore integrato nella CPU, chiamato *Thread Director*.

Esso comunica al sistema operativo dei consigli sulla gestione dei processi processi prendendo in considerazione la situazione attuale del processore.

Un'altra funzionalità introdotta da Windows 11 è il GPU accelerated scheduler, esso permette alla GPU di autogestire alcune task e la sua VRAM, alleggerendo il carico sulla CPU.

Sistemi Real Time

I sistemi operativi real time si dividono in:

- **Hard Real Time:** Richiedono il completamento di alcuni processi critici entro limiti specifici.
- **Soft Real Time:** Richiedono solo che ad alcuni processi venga fornita una priorità maggiore.

Parte III

Gestione della Memoria

Capitolo 8

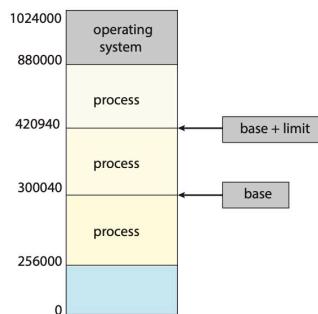
Gestione Memoria Principale

Abbiamo già visto nei capitoli precedenti l'importanza della memoria principale nella gestione dei processi. Il sistema operativo ha il compito di gestirla in modo efficiente così da permettere l'esecuzione dei programmi.

8.1 Introduzione

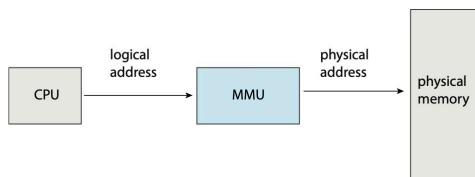
Alla compilazione di un programma il compilatore genera un modulo oggetto dove tutti gli indirizzi sono in **riferimento** a quello dell'inizio del modulo. Questo significa che c'è libertà di inserire il processo in qualsiasi punto della memoria e ci sono quindi decisioni che il sistema operativo è responsabile di prendere.

Per mantenere la sicurezza dei dati è importante che ogni processo possa accedere solamente ai dati ad esso assegnati, per questo motivo assegniamo ad ogni processo due indirizzi logici, uno base ed uno limite. Ora il processo può accedere solo agli indirizzi tali che $base \leq indirizzo \leq base + limite$.



Nei sistemi moderni il processo può essere rilocato nella memoria a *runtime*, questo grazie all'utilizzo della memoria virtuale. Questo ci permette di tradurre gli indirizzi fisici della memoria in indirizzi virtuali, non è quindi necessario allocare ad un programma un segmento contiguo di memoria, ma è possibile rendere contigui segmenti fisicamente lontani in memoria.

Questa traduzione avviene a livello hardware da parte della memory mapping and management unit (MMU).



File di Paging

Una funzionalità introdotta da Windows XP, che estende la memoria RAM del sistema con un file presente in memoria secondaria, delle stesse dimensioni.

8.1.1 Frammentazione

L'allocazione e la rimozione della memoria comporta la frammentazione della memoria.

- La **frammentazione interna**, quando la memoria allocata ad ogni programma è leggermente maggiore di quella richiesta.
- La **frammentazione esterna**, quando lo spazio per un nuovo processo è disponibile, ma non è contiguo, in questo caso può essere necessario rilocare dei processi per compattare lo spazio libero.

8.2 Paginazione

La paginazione risolve il problema posto dalla frammentazione esterna consentendo la non contiguità degli indirizzi fisici, mantenendo invece la contiguità negli indirizzi logici.

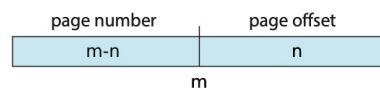
8.2.1 Implementazione

Si divide la memoria fisica in blocchi di dimensione fissa, detti frame o pagine fisiche. Allo stesso modo si divide anche la memoria logica in blocchi della stessa dimensione.

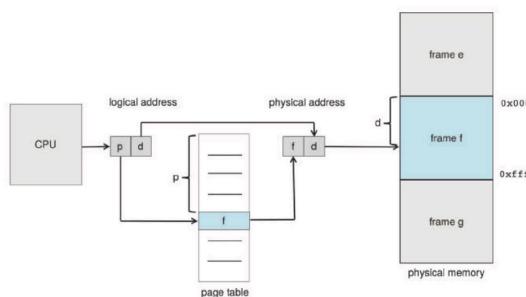
La **tabella delle pagine**, tiene traccia delle relazioni tra memoria fisica e logica e tiene traccia delle pagine libere.

Gli indirizzi logici generati dalla CPU vengono suddivisi in:

- **Numero di pagina:** indice che verrà usato nella tabella delle pagine.
- **Offset nella pagina:** definisce l'offset a partire dall'inizio della pagina



Se lo spazio logico ha dimensione 2^m allora si hanno 2^{m-n} pagine di dimensione 2^n . La scelta di m ed n detta anche la dimensione della tabella delle pagine.



Frammentazione

Si ottiene della frammentazione interna quando un processo non occupa completamente un frame, ad esempio un processo di 72766 byte occupa 36 pagine da 2048 byte, ma l'ultima pagina viene riempita circa a metà.

Dimensioni dei frame

Un'importante considerazione che deve essere fatta nell'architettura dell'elaboratore è la dimensione delle pagine.

Pagine più piccole permettono una minore frammentazione in quanto si possono meglio adattare alle dimensioni del programma, esse migliorano inoltre il principio di località.

Al contrario pagine più grandi riducono le dimensioni della page table e riducono il numero di page fault incontrati.

La soluzione adottata da molti sistemi operativi è quella di costruire pagine di diverse dimensioni così da potersi adattare ad ogni processo.

Pagine Bloccate

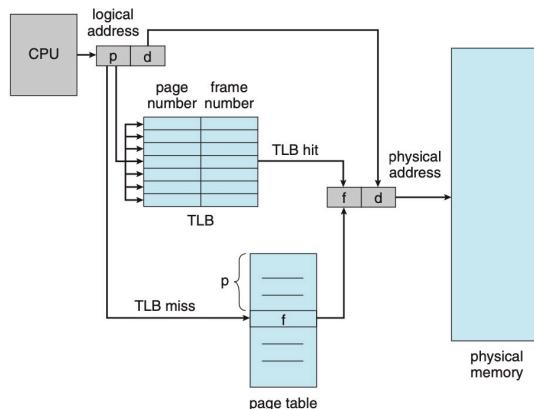
In alcuni casi occorre mantenere delle pagine perennemente in memoria, come nel caso di operazioni di I/O tramite DMA, per questo motivo è possibile segnalare al pager delle pagine bloccate (*pinned*) che non possono essere selezionate come vittime dall'algoritmo di sostituzione.

8.2.2 Supporto Hardware

Si utilizza una tabella delle pagine per ogni processo, essa risiede nella memoria principale e il processo ha un puntatore ad essa (PTBR) ed alla sua lunghezza (PTLR).

Uno degli svantaggi di inserire la tabella delle pagine nella memoria principale è l'**aumento del tempo di accesso dati**, infatti per ogni richiesta devono essere fatte due richieste alla memoria, una alla tabella ad una all'indirizzo fisico.

Una soluzione è il *translation look-aside buffer* (TLB) che permette la ricerca in parallelo su una piccola tabella (64-1024 elementi). Essa può essere utilizzata come cache per la tabella delle pagine. La TLB è più efficiente e sicura quando non viene condivisa tra più processi, ad ogni *context switch* andrebbe fatto un *flush* della memoria. È possibile aggiungere un valore *Address Space IDentifier (ASID)* che identifica il processo a cui appartiene l'associazione, questo permette di mantenere la TLB di un processo anche in un context switch.



Page Fault

Se un processo fa riferimento ad una pagina che al momento non è in memoria principale viene generato un **interrupt** di tipo page fault.

Il processo viene quindi sospeso finché il sistema operativo non carica la pagina mancante in memoria principale. Se la memoria è piena è necessario utilizzare un algoritmo di cammino pagina per scegliere quale pagina rimuovere.

Protezione della Memoria

Viene implementata grazie ad uno o più **bit di protezione** che determinano se una è possibile accedere una determinata pagina in lettura oppure lettura/scrittura.

Inoltre viene aggiunto un **bit di validità**, esso permette di individuare le pagine che non appartengono agli indirizzi *logici* del processo, questo viene poi segnalato all'utente tramite un'interrupt.

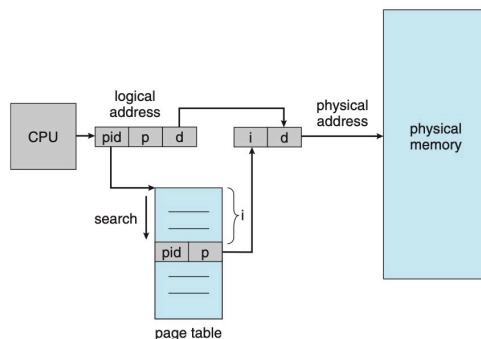
8.2.3 Dimensioni della Tabella

La tabella delle pagine può arrivare facilmente a dimensioni importanti, soprattutto quando se ne crea una per ogni processo, una rappresentazione naturale, ma inefficiente.

Tabella delle Pagine Invertita

Una soluzione è costruire una tabella delle pagine invertita, la quale ha un elemento per ogni frame fisico e salva il frame logico e un identificatore del processo.

Questo riduce lo spazio totale usato dalle tabelle delle pagine ma incrementa il tempo di ricerca nella tabella. Inoltre rende difficile la condivisione delle pagine fisiche tra processi.



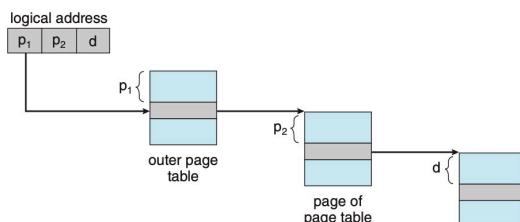
Paginazione Gerarchica

L'idea con questa soluzione è quello di dividere lo spazio degli indirizzi logici in più tabelle delle pagine. Il caso più semplice è quello ottenuto utilizzando una paginazione a due livelli.

page number	page offset
p_1	p_2
10	10

12

p_1 viene utilizzato per indicizzare la tabella delle tabelle, mentre p_2 per indicizzare la tabella delle pagine.



Per le architetture a 64 bit questo tipo di paginazione necessita di troppi livelli di paginazione, è quindi da considerarsi inadeguata.

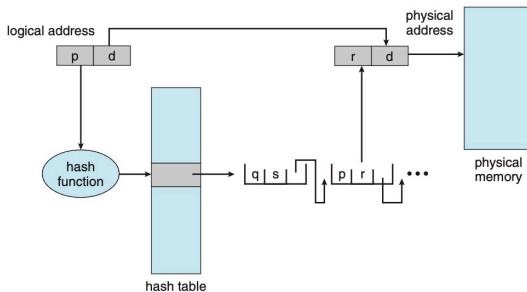
Tabella delle Pagine Hash

Comunemente utilizzato per architetture a 64 bit.

L'elemento che viene inserito nella funzione di hash è il numero della pagina virtuale. Gli elementi della tabella sono liste concatenate, questo per gestire le eventuali collisioni dovute alla funzione di hash.

In particolare ciascun elemento della tabella è composto da:

- Numero della pagina virtuale
- Numero della pagina fisica
- Puntatore al frame successivo



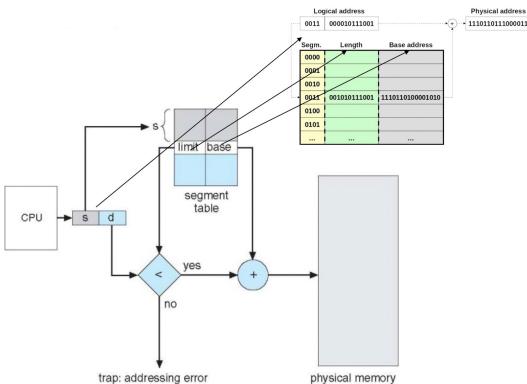
Per gli spazi a 64 bit si utilizza una tabella delle pagine a gruppi, ovvero una tabella hash dove ogni elemento contiene riferimenti alle pagine fisiche corrispondenti ad un gruppo di pagine virtuali contigue.

Segmentazione

Gestisce la memoria in segmenti di dimensione variabile, questo è più vicino a come l'utente immagina l'organizzazione della memoria. Ogni segmento è un'unità logica:

- Procedure e funzioni
- Variabili
- Oggetti

Si ha quindi una **tabella dei segmenti** per ogni processo che mappa gli indirizzi logici in coppie base, limite.

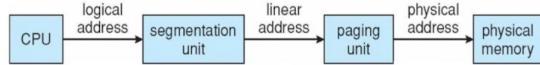


Due registri sono di particolare importanza in questo caso: STRB (*Segment Table Base Register*) e STLR (*Segment Table Length Register*).

8.2.4 Implementazioni

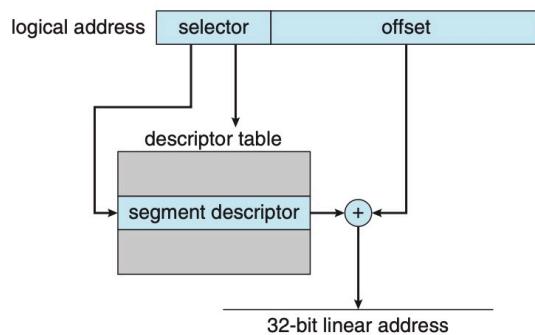
Intel 32

Il pentium supporta sia la segmentazione pura che la segmentazione mista a paginazione.



I segmenti possono essere di dimensione variabile, fino a 64 Kbyte, e possono essere al massimo 16K per processo, di questi al massimo 8K possono essere privati (Informazioni nella *Local Descriptor Table*, *LDT*) e altri 8K condivisi (Informazioni nella *Global Descriptor Table*, *GDT*).

Il selettor è di 16 bit, dove 13 bit rappresentano l'indice del segmento ($2^{13} = 8192$), 1 bit per indicare LDT o GDT e 2 bit di protezione. Altri 16 bit sono di offset.



Nella tabella vengono salvate anche altre informazioni, come un bit per indicare se il segmento è in memoria principale e uno per indicare se è stato modificato.

L'indirizzo ottenuto dalla segmentazione può essere paginato. Le pagine per il pentium possono essere di 4MB (1 livello) o 4KB (2 livelli) e viene usata una paginazione a due livelli.



La dimensione della descriptor table viene aumentata nel tempo, prima a 24 e poi a 32 bit

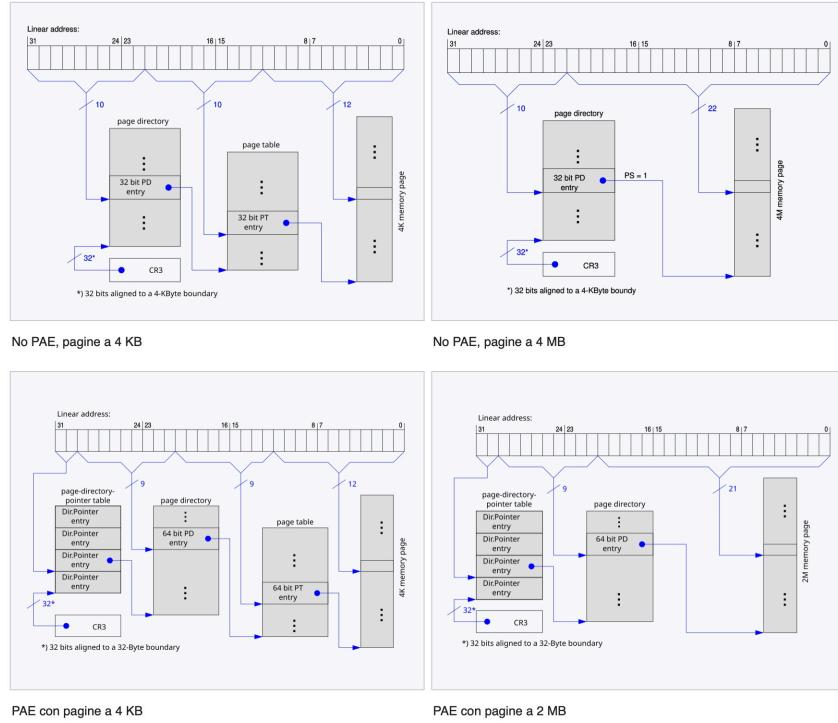
PAE

Anche detto **Page Address Extension**, permette di raggiungere i 64 Gbyte di memoria.

Le principali modifiche all'architettura sono due: Viene inserito un ulteriore livello di paginazione, una page directory pointer table di 4 elementi e gli elementi della descriptor table arrivano a 64 bit, di cui solo 36 vengono utilizzati, arrivando così ad indicizzare 64 Gbyte di memoria.

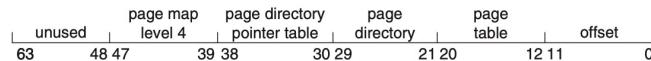
Le singole applicazioni possono, in generale, accedere solamente a 4GB di memoria, ma esse possono essere inserite in 64 Gbyte di memoria.

Inoltre è possibile per le applicazioni utilizzare delle system call per spostare il proprio spazio di indicizzazione, permettendo così di accedere ad una maggiore quantità di memoria RAM.



Intel x86-64

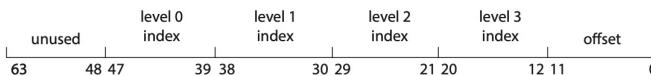
Nei sistemi a 64 bit in realtà non se ne usano così tanti ($2^{64} = 17,179,869,184$ GB), gli indici sono tutti a 48 bit, lasciando inutilizzati i primi 16 bit.
L'architettura supporta pagine da 4KB, 2MB, 1GB.



È possibile utilizzare la PAE anche qui con l'obiettivo di portare gli indirizzi fisici da 48 a 52 bit.

ARMv8

Anche i processori ARM lavorano su 64 bit ed anche loro utilizzano fino a 4 livelli di paging ottenendo così pagine da 4KB, 2MB, 1GB.



8.3 Memoria Virtuale

Abbiamo visto come la paginazione può permettere di condividere la memoria tra più processi, tuttavia è raro che un intero processo venga inserito nella memoria principale prima che venga eseguito.

Questo perché non tutte le sezioni del programma vengono eseguite allo stesso tempo, è quindi conveniente caricarle in modo dinamico.

In questo modo le dimensioni totali del programma non sono più limitate dalle dimensioni della memoria fisica e viene lasciato più spazio nella memoria per la multiprogrammazione. Inoltre l'utilizzo della memoria virtuale rende più semplice la condivisione di librerie o dati.

8.3.1 Spazio degli Indirizzi Virtuali

La memoria virtuale separa la memoria logica utilizzata dagli sviluppatori da quella fisica, rendendo così più semplice il lavoro del programmatore che non deve gestire manualmente le limitazioni fisiche della memoria principale.

Nello spazio degli indirizzi virtuali i programmi sono allocati in modo contiguo, a partire da un certo indirizzo iniziale 0 fino a quello finale *Max*.

Nella memoria fisica invece il programma può essere allocato in pagine sparse. La MMU si occupa della conversione degli indirizzi logici in fisici.

Condividere librerie è ora estremamente semplice, basterà mappare le pagine logiche agli stessi frame fisici.

In questo modo entrambi i processi vedono gli indirizzi della libreria comune come propri, ma in realtà i dati non vengono duplicati.

8.4 Pager

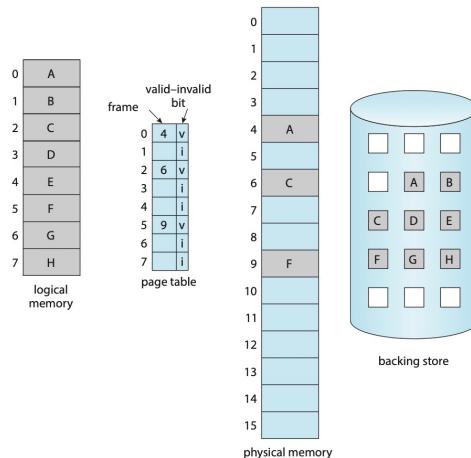
Un vantaggio della memoria virtuale è quello di poter caricare in modo dinamico i frame fisici dalla memoria secondaria. Il modulo del sistema operativo che si occupa della paginazione su richiesta è il *pager*.

Il termine *swapping* indica l'atto di inserire o rimuovere delle pagine dalla memoria. Quando una pagina viene rimossa dalla memoria essa viene riportata in un dispositivo di memoria secondaria, all'interno di un file (Windows) o una partizione (linux)

Spesso si utilizza uno swapper pigro, che carica le pagine in RAM solo quando esse vengono richieste.

Inoltre spesso si utilizza una partizione di swap sulla memoria secondaria per ampliare ulteriormente le dimensioni della memoria principale, essa permette di ottenere delle prestazioni migliori in quanto non incorre nella frammentazione dei dischi.

Per sapere quali pagine si trovano già in memoria e quali invece devono essere caricate il pager utilizza una tabella delle pagine, essa associa la memoria logica a quella fisica. Inoltre comprende un bit di validità che specifica se il frame si trova in memoria o meno.

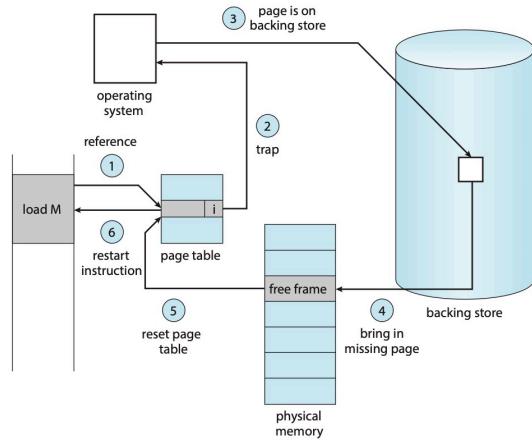


8.4.1 Page Fault

Quando il programma richiede un indirizzo appartenente ad una pagina non caricata in memoria causa una trap al sistema operativo, una *page fault*.

A questo punto il sistema operativo consulta una sua tabella delle pagine e invia al programma un abort se l'indirizzo non è valido.

Se viene trovata la pagina nella memoria secondaria allora essa viene caricata nella RAM e l'esecuzione del programma può riprendere.



Se non sono disponibili dei frame liberi nella memoria principale sarà necessario rimuovere prima una pagina secondo un algoritmo di selezione. Si ricerca un algoritmo che riduca al minimo il numero di page fault in quanto esse sono un fattore importante sulle prestazioni del sistema.

Viene impiegato un bit di modifica, o *dirty bit*, per ridurre l'overhead dei trasferimenti di pagine. Solo le pagine modificate vengono riscritte al disco, le altre possono essere semplicemente sovrascritte.

Prestazioni

Il tempo effettivo di accesso alla memoria (EAT) si può calcolare nel seguente modo:

$$EAT = (1 - p) \cdot \text{accesso alla memoria} + p \cdot \text{overhead page fault}$$

Dove p è la probabilità di ottenere un page fault.

Assumendo dei valori realistici, il tempo di accesso alla memoria è di circa 200ns, mentre l'overhead per un page fault 8ms.

Sotto queste ipotesi se si ha un page fault ogni 1000 accessi l'accesso alla memoria viene mediamente rallentato di un fattore di 40.

Per ottenere un fattore di rallentamento accettabile (<0.1) è necessario arrivare ad un page fault ogni 400,000 richieste.

8.4.2 Prepaging

Per ridurre il gran numero di page fault che avvengono all'avvio del processo si può utilizzare il prepaging, ovvero si possono copiare un certo numero di pagine in memoria prima dell'esecuzione del programma.

Ad esempio, se un programma è stato sospeso per una richiesta I/O, quando esso riprende l'esecuzione è possibile riportare in memoria il suo ultimo working set.

È importante bilanciare le prestazioni di questa operazione ed assicurarsi che il vantaggio in riduzione di page fault sia superiore al costo di prepaging.

8.4.3 Algoritmi di Paging

Il paging richiede diversi algoritmi per un funzionamento corretto, in particolare sono necessari per l'allocazione dei frame e per la sostituzione delle pagine.

Alcune pagine devono rimanere bloccate (pinned) in memoria, non possono essere selezionate come vittime dagli algoritmi.

Gli algoritmi vengono valutati su una particolare stringa di riferimenti in memoria, ovvero 7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1.

Algoritmo Ottimo

L'algoritmo ideale è quello che rimuove la pagina che non verrà utilizzata per il periodo di tempo più lungo, in questo caso sulla stringa l'algoritmo genera 9 page fault.

Tuttavia questo non è possibile saperlo senza conoscere il futuro, cosa che il sistema non è ovviamente in grado di fare.

FIFO

L'algoritmo più semplice è forse quello FIFO, possiamo creare una coda con tutte le pagine di memoria fisica e sostituendo la prima entrata.

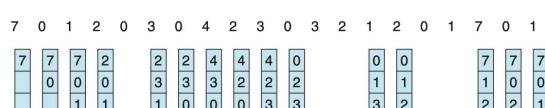
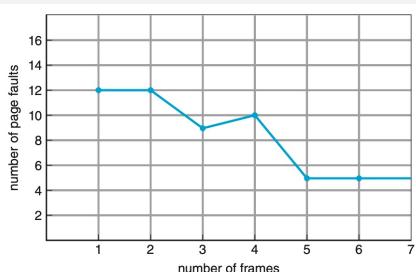


Figura 8.1: Example when usign a 3 frame memory

Non sempre aumentare le dimensioni della memoria porta ad una riduzione del numero di page fault, questo fatto è noto come *Anomalia di Belady*.

In particolare usando una coda FIFO con stringa dei riferimenti 1 2 3 4 1 2 5 1 2 3 4 5



LRU

L'algoritmo rimpiazza la pagina che non è stata utilizzata per più tempo, provando a prevedere il futuro conoscendo il passato.

Questo algoritmo genera 12 page fault sulla stringa di riferimento.

LRU viene considerato un buon algoritmo per questa applicazione, tuttavia risulta esserne difficile l'implementazione. Ci sono diverse soluzioni per l'implementazione, ma tutte richiedono dell'hardware dedicato per renderle praticabili.

LFU

Least Frequently Used, si rimpiazza la pagina meno utilizzata.

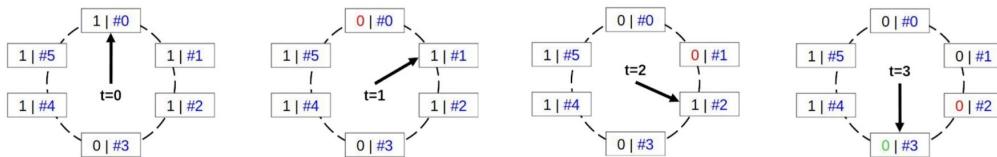
MFU

Most Frequently Used, si basa sull'idea che la pagina utilizzata più di recente non deve essere ricoppiata in memoria per essere rimpiazzata.

Seconda Chance

L'algoritmo si basa su un algoritmo FIFO, ma viene aggiunto un bit di riferimento ad ogni pagina. Il bit di riferimento di tutte le pagine viene inizialmente impostato a 0, quando vengono utilizzate viene impostato ad 1.

Quando è necessario sostituire una pagina si cerca nella coda FIFO, se si incontra uno 0 quella pagina verrà rimossa. Altrimenti se si incontra un 1 si dà alla pagina una seconda chance, impostando il bit a 0 e spostandosi oltre.



L'algoritmo può essere migliorato tenendo conto anche del bit di modifica, si consideri la coppia (bit di riferimento, bit di modifica).

La pagina migliore è quella con valore (0, 0), seguita da (0, 1) che ha il difetto di dover essere salvata sulla memoria secondaria.

Pool di Frame Liberi

L'algoritmo richiede che venga sempre mantenuto un certo numero di frame liberi in memoria. Questi vengono utilizzati da buffer, dopo la selezione del frame vittima il nuovo frame viene scritto in uno dei frame liberi e infine il frame vittima viene copiato in memoria secondaria.

Un'altra ottimizzazione disponibile è quella di controllare la memoria secondaria e quando esso è inattivo utilizzarlo per copiare dei frame modificati, resettando il loro bit di modifica.

8.4.4 Allocazione dei Frame

Il numero minimo di pagine che viene allocato ad un processo è deciso dall'architettura, mentre il massimo è dettato dalla dimensione della memoria primaria.

Allocazione Uniforme

Semplicemente divide lo spazio della memoria per i programmi in esecuzione, quindi se si hanno 100 frame e 5 processi ognuno otterrà 20 frame.

Allocazione Proporzionale

Uno schema di allocazione proporzionale alla dimensione del processo, un processo con più frame riceverà più risorse.

Allocazione con Priorità

Si impiega uno schema basato sulla priorità, questa stessa priorità viene utilizzata anche per la scelta dei frame da rimuovere, un processo a priorità maggiore rimuoverà frame di processi a priorità inferiore.

Allocazione locale e globale

L'allocazione globale permette ai processi di prendere frame da qualsiasi altro processo in esecuzione, quella locale invece limita le scelte ai frame che appartengono già al processo.

Nei sistemi moderni si preferisce l'allocazione globale in quanto spesso garantisce un numero minore di page fault.

8.4.5 Mappatura di File

La mappatura di file in memoria permette l'accesso in modo completamente simile a quanto si fa per qualsiasi altro dato, l'accesso iniziale avviene tramite un page fault grazie al quale viene copiata una pagina di dati alla memoria principale.

Questo permette anche di condividere il file tra più programmi in modo semplice dato che si trova in memoria virtuale.

Purtroppo ci sono anche degli svantaggi, legati principalmente a mantenere l'integrità del file anche in presenza di concorrenza.

Il file viene aggiornato effettivamente in memoria solo alla chiusura o quando il pager decide di resettare il bit di modifica.

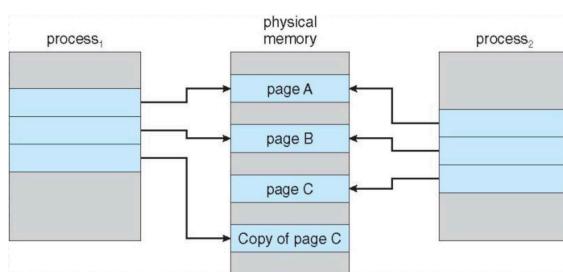
Tutti i moderni sistemi operativi supportano la mappatura di file in memoria principale, spesso tramite una chiamata di sistema (`mmap()`). Alcuni sistemi utilizzano la mappatura di default, come Solaris, rimuovendo così l'overhead causato dalla system call.

Mappatura dell'I/O

È possibile condividere informazioni tra processo e dispositivo I/O grazie alla mappatura in memoria, ad esempio un processo può scrivere su degli indirizzi di memoria, dai quali la scheda video prenderà i dati da mostrare all'utente.

Copy On Write

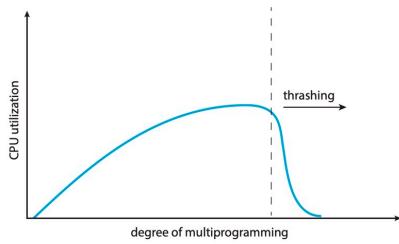
Le pagine di memoria vengono condivise tra due o più processi, quando uno vuole scrivere su una pagina la copia prima in una nuova locazione di memoria e poi la modifica.



8.5 Trashing

Definiamo trashing la situazione in cui un processo è costantemente occupato a spostare pagine dal disco al posto di utilizzare le risorse del processore.

Possiamo dire che c'è un certo numero di processi oltre il quale la multiprogrammazione diventa uno spreco di risorse, in quanto viene perso troppo tempo in page fault.



Vogliamo quindi trovare il massimo numero di processi che possiamo eseguire in multiprogrammazione senza incorrere nel *trashing*, per decidere questo utilizziamo il modello di località.

Il modello afferma che un processo si sposta da una località ad un'altra durante la sua esecuzione, una località è un insieme di indirizzi che sono spesso utilizzati insieme.

8.5.1 Working Set

Possiamo approssimare la località di un processo guardando ai frame che esso ha utilizzato nelle ultime Δ chiamate alla memoria, chiamiamo il set dei frame usati dal processo i nel periodo Δ *Working Set* (WSS_i) .

Δ deve essere della giusta dimensione, troppo piccolo e non conterrà tutta la località, troppo grande e conterrà più località

Rappresentiamo il Working set con due numeri: $WS(t, \Delta)$ dove il primo è l'istante che stiamo analizzando, mentre il secondo la dimensione della località.

Otteniamo quindi $D = \sum_i WSS_i$, il numero totale di frame attualmente nella località dei processi in esecuzione, se questo numero supera il numero totale dei frame disponibili nella memoria principale si rischia di incorrere nel *trashing*.

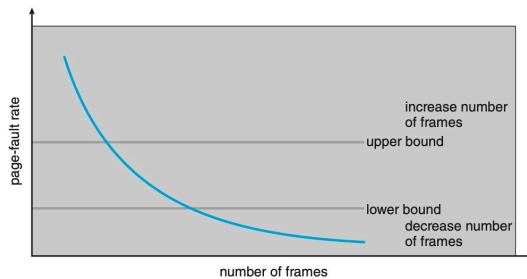
A questo punto è conveniente terminare alcuni processi o sottoporli a swapping.

Il working set è un insieme in costante cambiamento, quindi per alleggerirne l'utilizzo è conveniente effettuare il controllo su un determinato periodo e salvare quali frame sono presenti in almeno un working set in un bit di stato (1 se presente, 0 se non presente).

Quando sarà necessario rimuovere un frame si cercherà quello con il maggior numero di 0 nei suoi bit di stato, ovvero quello che non appare nei working set da più tempo.

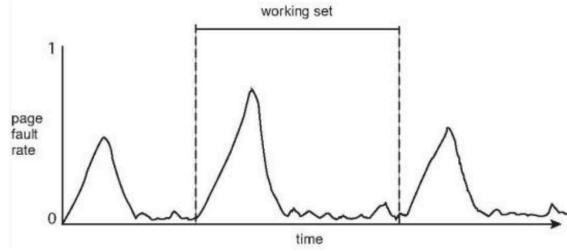
8.5.2 Frequenza di Page Fault

Un approccio più diretto è quello di stabilire una certa frequenza di page fault "accettabile" e aumentare o diminuire i frame di conseguenza.



8.5.3 Relazione tra Working Set e Frequenza di Page Fault

Il cambiamento di località comporta un picco di page fault



8.6 Allocazione di Memoria al Kernel

Il kernel utilizza della memoria separata da quella destinata agli utenti, questa memoria molto spesso non è soggetta a paginazione in quanto esso opera con oggetti di piccole dimensioni che sprecerebbero le pagine.

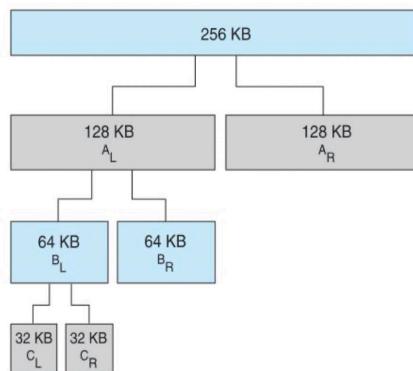
Inoltre alcune parti della memoria del kernel devono essere contigue per permettere a dispositivi I/O, che non hanno accesso alla memoria fisica, di scriverci.

Vedremo ora due tecniche per gestire la memoria che viene riservata al kernel.

8.6.1 Allocatore potenza 2

Utilizza un segmento di memoria a dimensione fissa e fisicamente contigua, che viene allocato in blocchi di dimensione pari a potenze del 2.

Quando viene richiesta una quantità di memoria essa viene arrotondata alla più piccola potenza che la contiene, se si richiede una quantità di memoria minore il segmento viene diviso a metà.

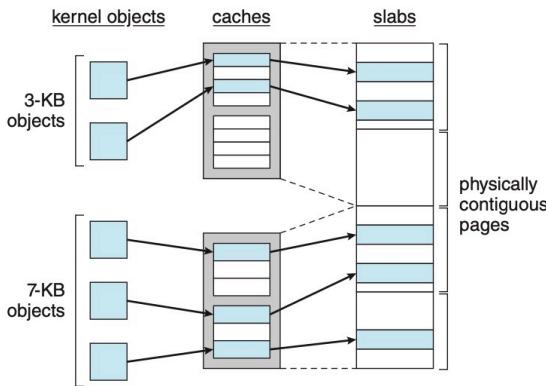


8.6.2 Allocazione a Lastra

Il kernel ha la necessità di allocare e distruggere oggetti frequentemente, per questo motivo è conveniente mantenere una cache per ogni tipo di oggetto così da rimuovere il costo di inizializzazione.

Le cache contengono istanze della struttura dati del kernel a cui sono assegnate e vengono inserite in una lastra (*slab*), dei segmenti di memoria fisicamente contigui.

Quando il kernel richiede un nuovo oggetto esso viene preso da quelli segnati come liberi. Se non dovessero essere disponibili oggetti liberi ne vengono allocati altri in una lastra vuota. Nel caso in cui non dovessero esserci lastre libere verrà allocato spazio per far crescere la cache di una lastra.



8.7 Implementazioni

8.7.1 Linux

Per quanto riguarda la **Memoria del Kernel** linux utilizza il sistema Buddy, unallocatore di potenza di 2. Invece per le strutture dati, a partire dal kernel 2.4, viene utilizzato l'allocatore SLOB o SLUB.

- **SLOB:** Utilizzato per sistemi con poca memoria, mantiene 3 liste, per oggetti grandi, medi e piccoli.
- **SLUB:** Allocatore a lastre ottimizzata per sistemi multicore.

Invece la **Memoria Utente** viene gestita tramite paginazione su richiesta. La politica di sostituzione è detta Clock: vengono mantenute due liste, `active_list` e `inactive_list`, la prima che contiene le pagine in uso e la seconda contiene pagine non utilizzate di frequente che possono essere rimosse.

Ogni pagina ha un bit di accesso.

- Quando una pagina viene **allocata** o **riferita** il bit viene impostato su 1 e viene inserita in coda a `active_list`.
- La pagina utilizzata meno di recente si sposta in testa alla lista `active_list`, dalla quale viene **spostata** verso la lista `inactive_list` per mantenere le due liste bilanciate.
- Inoltre i bit di accesso vengono resettati periodicamente, riportando il sistema ad uno stato iniziale con tutti i processi in `inactive_list`.
- Il deamon `kswapd` si risveglia periodicamente e se la memoria non è sufficiente **termina** dei programmi dalla `inactive_list`.

8.7.2 Windows

Per quanto riguarda la **Memoria del Kernel** Windows scrive i dati su un page file, ma quando questo non è possibile i dati vengono scritti su una pool di memoria non paginata. (ad es. quando Windows non può fare page fault perché sta gestendo un page fault.)

Invece la **Memoria Utente** viene gestita tramite paginazione con clustering, quindi quando avviene un page fault viene caricato non solo la pagina richiesta, ma anche le pagine adiacenti.

Ad un processo viene assegnato una dimensione minima e massima del working set, il minimo è il numero di pagine garantite dal sistema operativo, il numero massimo è il limite oltre il quale il processo non può allocare.

Quando l'occupazione della memoria scende si effettua una riassegnazione dei working set aumentando i tetti massimi per processo.

Windows fa inoltre uso di compressione delle pagine più vecchie, prima di ricorrere alla memoria virtuale. Quando poi i dati tornano ad essere utilizzati devono prima essere decompressi.

Capitolo 9

File

Il *file* viene definito come uno spazio di indirizzi logici contigui. È quindi un insieme di informazioni registrate nella memoria secondaria con delle informazioni correlate.

Per gli utenti il file è la più piccola porzione di memoria secondaria indicizzabile e rappresentano l'unico modo di scrivere delle informazioni nella memoria secondaria.

9.1 Tipo di Dati Astratto

Un file è un tipo di dati astratto su cui sono definite le operazioni di:

- **Creazione**

Reperisce dello spazio per memorizzare il file nella memoria e crea un nuovo elemento nella directory.

- **Scrittura**

Viene fatta una *system call* che specifica il file su cui scrivere e i dati da scrivere. Il sistema operativo mantiene un puntatore di scrittura per continuare una scrittura sequenziale.

- **Lettura**

system call con riferimento al file e indirizzo di memoria principale su cui scrivere i dati. Anche in questo caso il sistema operativo mantiene un puntatore di lettura per le letture sequenziali.

- **Posizionamento del File**

Riposiziona il puntatore di lettura/scrittura ad un diverso punto del file.

- **Cancellazione**

Si rilascia lo spazio allocato al file e lo si rimuove dalla directory. (I bit non vengono azzerati quindi i dati del file possono essere ancora letti)

- **Troncamento**

Elimina il contenuto del file, ma ne mantiene tutti gli attributi

- **Impostazione degli attributi**

Reperimento/aggiornamento delle informazioni del file nel relativo elemento della directory

Da queste operazioni elementari è possibile poi costruirne di più complesse facendone una combinazione.

9.2 Implementazione

9.2.1 Attributi dei File

- **Nome:** Identificativo del file per l'utente.
- **Identificativo:** Etichetta unica utilizzata dal file system.
- **Tipo:** Specifica il tipo dei dati contenuti nel file, ad esempio testi, immagini, video oppure programmi.
- **Locazione:** Puntatore alla posizione del file sul dispositivo di memoria secondaria.
- **Dimensione:** Attuale dimensione del file.

- **Protezione:** Parametri di controllo su lettura/scrittura/esecuzione del file.
- **Ora, data, identificazione dell'utente:** Informazioni utili alla protezione del file.

Gli attributi estesi sono una serie di informazioni aggiuntive che possono essere associate ad un file, essi sono una serie di coppie nome-valore e possono essere definiti dall'utente.
Non vengono salvati nel file, ma vengono salvati separatamente e gestiti dal file system.

9.2.2 Tipo del File

Il tipo del file è un attributo che indica la struttura interna e permette al sistema operativo di gestire il file in modo corretto.

Le estensioni possono essere specificate in vari modi:

- Meccanismo delle estensioni (Windows)
- Attributo nella directory (MacOS)
- Valore posto all'inizio del file (Unix)

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

9.2.3 Struttura del File

Strettamente collegato al tipo del file, il quale può suggerirci la sua struttura interna.

- Nessuna Struttura (sequenza di byte)
- Struttura a record semplice (Linee a lunghezza fissa/variabile simili alle tabelle sql)
- Struttura complessa (documento formattato o eseguibile rilocabile)

Spesso è inefficiente lasciare al sistema operativo il compito di gestire ogni tipo di file, è più conveniente che esso sappia gestire solo i tipi base (spesso solamente i file eseguibili) e il resto viene lasciato a programmi di sistema o di terze parti.

Unix e DOS implementano questa scelta, i file sono gestiti solamente come stringhe di byte.

9.3 Apertura di un File

Prima di un qualsiasi accesso ai dati del file occorre trovare l'indirizzo fisico dei dati, ovvero è necessario "aprire" il file.

Il sistema operativo tiene poi in memoria una tabella dei file aperti che permette di accedere rapidamente ai file in uso.

Le system call che gestiscono questa funzione sono:

- **open(F)**: ricerca il file nella directory del disco e copia il puntatore ai dati nella tabella dei file.
- **close(F)**: copia il contenuto residente nella memoria principale alla memoria secondaria e rimuove il file dalla tabella.

9.3.1 Sistemi multiprogrammati

Nei sistemi multiprogrammati la gestione dei file aperti è più complicata in quanto file possono essere aperti da più processi.

Il sistema operativo mantiene due tipi di tabelle dei file, una a livello del sistema e un'altra per ogni processo.

Tabella di Sistema

La tabella di sistema mantiene riferimenti a tutti i file aperti nel sistema, per ognuno essa salva:

- **Posizione** del file nel disco
- **Dimensione** del file
- Date di **ultimo accesso** e **ultima modifica**
- **Contatore** di aperture

In particolare il **contatore di aperture** permette di assicurarsi di chiudere il file al livello del sistema operativo solamente quando tutti i processi hanno smesso di utilizzarlo.

Tabella del Processo

La tabella associata al processo mantiene riferimenti a tutti i file aperti dal processo, per ognuno essa salva:

- Puntatore alla posizione corrente nel file

Inoltre alcuni sistemi operativi forniscono un **lock** per mediare sull'accesso dei file. Esso può essere **obbligatorio** oppure **consigliato**.

Nei sistemi in cui il lock è obbligatorio (Windows) non è possibile accedere ai file che sono già aperti da un'altro processo, invece nei sistemi con un lock consigliato (Unix) viene passato allo sviluppatore il compito di mantenere l'integrità dei dati.

9.3.2 Condivisione

Nei sistemi multiutente è utile che i file vengano condivisi tra più utenti, tuttavia questo deve avvenire seguendo un preciso schema di protezione.

Il modello più diffuso prevede un **proprietario** di ogni file, il suo creatore, che può cambiare gli attributi tra i quali si trova il gruppo di utenti che possono accedere al file.

Coerenza

Quando più utenti possono apportare modifiche allo stesso file si possono verificare delle situazioni simili alle race condition che avvengono nella sincronizzazione fra processi.

Spesso si impone che le scritture di un utente risultino immediatamente visibili agli altri utenti, il file ha quindi un'unica immagine a cui tutti gli utenti accedono.

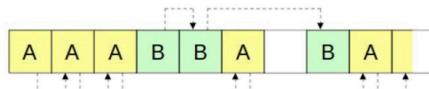
9.4 Allocazione

L'allocazione del file su disco può avvenire in più modi.

Allocazione Contigua

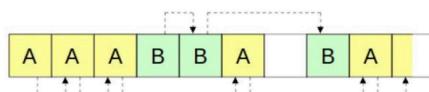
Il file viene scritto in locazioni contigue di memoria, ha il vantaggio che è necessario solo un indirizzo iniziale e la lunghezza, ma spesso non è semplice conoscere la lunghezza a priori e si rischia di generare frammentazione.

I blocchi logici non devono essere uguali a quelli fisici, se ho blocchi fisici da 512 byte posso dividerlo in 10 blocchi logici da circa 51 byte, riducendo così la frammentazione. Con dei puntatori è possibile anche distribuire i file nel disco, senza che siano fisicamente contigui.



Allocazione Concatenata

Il file può essere sparso in tutto il disco, ogni blocco contiene un puntatore al blocco successivo.



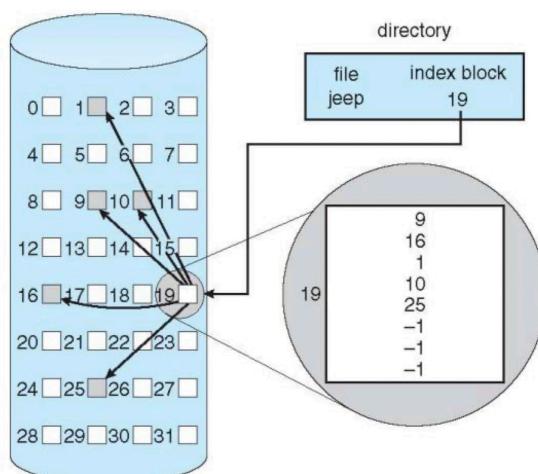
Esempio di questa tecnica è il file system FAT (File Allocation Table), sviluppato per MS-DOS. Essa mantiene una tabella con un identificativo del file e l'indirizzo di inizio del file sul disco.

La prima versione, FAT32 è limitata nella dimensione massima del disco, e dei file, a 4GB. exFAT elimina questa limitazione utilizzando indirizzi a 64 bit ed è supportata da tutti i sistemi operativi moderni.

Windows ora si è spostato su NTFS con piani per arrivare ad utilizzare ReFS sempre più avanzati per soddisfare le richieste degli utenti.

Allocazione Indicizzata

Per ogni file si stabilisce un blocco indice, il quale contiene tutti i puntatori ai blocchi utilizzati dal file. Introduce quindi l'overhead per leggere il blocco indice, ma poi rimuove la frammentazione esterna del disco.



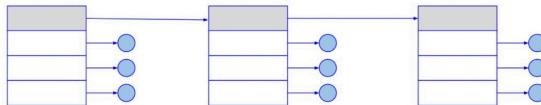
Qualora i 512 puntatori non dovessero essere sufficienti si possono utilizzare delle tabelle concatenate con 511 puntatori a blocchi di memoria ciascuna.

Mapping

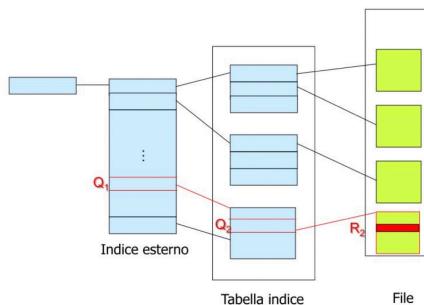
È possibile astrarre ulteriormente introducendo degli indirizzi logici dove il file risulta avere indirizzi contigui anche se gli indirizzi fisici non lo sono.

Questo si può ottenere in più modi, tramite uno schema concatenato oppure tramite indici multilivello.

Il primo, più semplice, prevede dei blocchi che puntano direttamente ai blocchi logici del file. Questi blocchi di indirizzi possono essere concatenati per ottenere una dimensione massima maggiore.

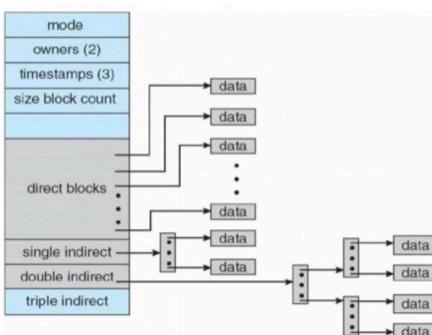


Il secondo utilizza più livelli per indicizzare una quantità maggiore di memoria.



Schema Combinato

Una soluzione ibrida viene spesso utilizzata nei sistemi operativi, ad esempio in unix, dove si mantiene un blocco per ogni file (inode) che contiene informazioni su di esso e un primo livello di indirizzi, alcuni sono diretti, altri sono multilivello a 2 o più livelli.



- 0-10 indirizzi: indirizzamento diretto
- 11° indirizzo: 1 livello
- 12° indirizzo: 2 livelli
- 13° indirizzo: 3 livelli

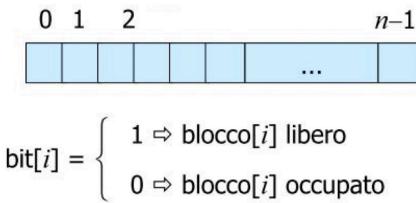
In generale l'allocazione continua ha buone prestazioni sia per l'accesso casuale sia per quello sequenziale, l'allocazione concatenata è ottimale per l'accesso sequenziale.

L'allocazione indicizzata è più complessa, un singolo accesso al disco può richiederne molti, è quindi importante gestire correttamente il clustering dei dati.

9.4.1 Spazio Libero

Per scegliere dove inserire i nuovi file è importante tenere traccia dello spazio libero nel disco. Poi quando si crea un nuovo file si cercano blocchi liberi dalla struttura dati e quando si eliminano si raggiungono i blocchi nella struttura.

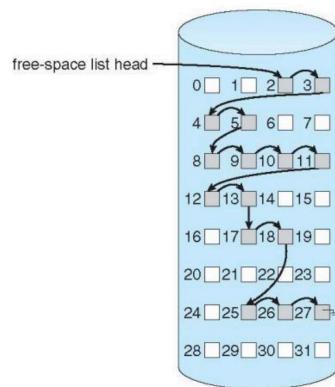
Bitmap



È possibile mantenere un vettore dove ogni bit rappresenta un blocco, questa struttura dati ha buone prestazioni quando il vettore può essere mantenuto in memoria principale.

Lista Concatenata

Tutti i blocchi liberi si collegano mediante puntatori e si mantiene il puntatore della testa della lista in memoria centrale. È inefficiente per trovare spazi contigui di memoria.



Grouping

Creazione di n blocchi, sul primo vengono salvati gli indirizzi dei blocchi liberi, che inizialmente sono $n - 1$

Conteggio

Se lo spazio viene allocato e liberato in modo contiguo è possibile mantenere semplicemente un indirizzo sul disco e il numero di blocchi liberi contigui successivi all'indirizzo.

TRIM è un comando ATA che consente al Sistema Operativo di comunicare ai dischi quali blocchi di dati possono essere eliminati.
Questo permette agli SSD di meglio gestire i dati, senza copiare file che verranno poi eliminati.

9.4.2 Prestazioni

Per ottimizzare le prestazioni i dati e metadati vengono mantenuti in locazioni di memoria vicine nel disco. Anche le dimensioni dei puntatori vanno valutate attentamente, puntatori di dimensioni

maggiori permettono di indicizzare una maggior quantità di memoria, ma richiedono più spazio di memorizzazione.

Per le letture/scritture asincrone è possibile utilizzare delle cache, inizialmente si utilizzava una cache sul dispositivo ed una sulla memoria principale.

Da quando le cache sono aumentate di dimensione è possibile utilizzare solo quella del dispositivo.

L'algoritmo di rimpiazzo LRU per questa cache non è efficiente (una volta letto un dato difficilmente viene letto nuovamente), si preferisce quindi un algoritmo free-behind (che rilascia la pagina più lontana dall'indirizzo corrente) oppure read-ahead (che carica sia la pagina richiesta che quelle successive)

9.4.3 Incoerenze

I file sono mantenuti parzialmente in memoria ram e nel disco, con un sistema di sincronizzazione non istantaneo, per questo motivo in caso di malfunzionamento del sistema si possono generare delle incoerenze che vanno correttamente gestite.

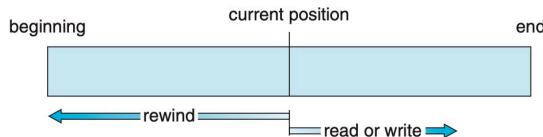
Per risolvere questo problema si possono utilizzare dei programmi (es. `chkdsk` in UNIX) che confrontano i dati sul disco e quelli sulla directory alla ricerca di possibili incoerenze.

Inoltre è possibile effettuare dei backup dei dati del disco da cui recuperare file persi o danneggiati.

9.4.4 Accesso al File

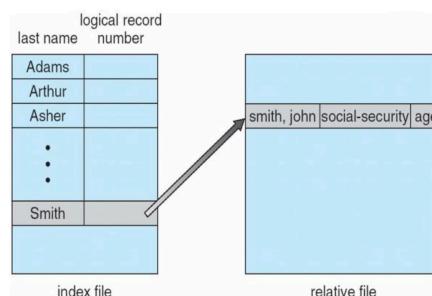
Approfondiamo le strategie che possono essere utilizzate per accedere ai file, in particolare vediamo:

- **Accesso Sequenziale:** I dati vengono letti un blocco alla volta e il puntatore avanza ad ogni lettura o scrittura in modo regolare.
- **Accesso Diretto:** I dati vengono letti da posizioni arbitrarie senza dover rispettare un determinato ordine.



- **Accesso Indicizzato:** Strategia costruita sull'accesso diretto, in aggiunta viene costruito un file di indice per l'accesso al file stesso.

Il file contiene dei puntatori al file stesso, prima di accedere al file serve quindi fare una ricerca nell'indice.

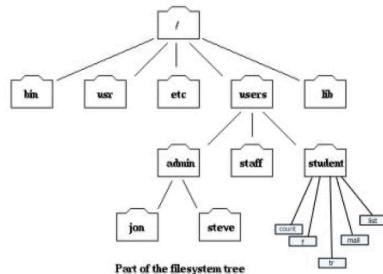


Capitolo 10

File System

La struttura logica con la quale il sistema operativo gestisce i file prende il nome di *file system*. Essa fornisce meccanismi di registrazione, accesso e protezione ai dati e programmi degli utenti e del sistema.

Tutti i moderni sistemi operativi utilizzano un file system **gerarchico** a cui gli utenti sono abituati. L'organizzazione gerarchica non viene rispecchiata fisicamente nel disco, è solo una particolare astrazione dei dati.



Il file system ha il compito di fornire all'utente una serie di funzioni ad alto livello che permettono di lavorare sui file mascherando i dettagli implementativi. Tutto questo deve essere svolto a prescindere dalle caratteristiche fisiche delle memorie secondarie.

10.1 Directory

Tipicamente tutte le informazioni sui file vengono conservate nella struttura **directory**, che risiede sullo stesso dispositivo dei dati. Al suo interno viene salvato, per ogni file, un identificatore e altri attributi:

- **Nome**
- **Tipo**
- **Indirizzo**
- **Lunghezza attuale**
- **Lunghezza massima**
- **Data ultimo accesso**
- **Data ultima modifica**
- **ID del proprietario**
- **Informazioni di protezione**

Sulla directory è possibile fare anche delle operazioni:

- **Ricerca**
- **Creazione** di un file
- **Eliminazione** di file

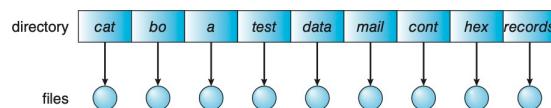
- **Elenco** di tutti i file
- **Ridenominazione** di un file
- **Attraversamento** del file system

La struttura dati della directory deve essere in grado di garantire buone prestazioni e deve permettere di raggruppare i file secondo specifiche caratteristiche.

Directory Monolivello

La struttura più semplice è quella che pone tutti i file allo stesso livello.

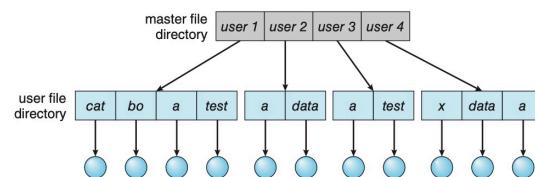
Tuttavia questa struttura non supporta nessun raggruppamento logico e presenta problemi nel caso di due file con lo stesso nome.



Directory a Due Livelli

Ogni utente ha la sua directory ad un livello separata.

Questa struttura ha un possibile raggruppamento (per utente) ed non presenta problemi se due utenti hanno un file con lo stesso nome, tuttavia risulta essere ancora estremamente limitata.



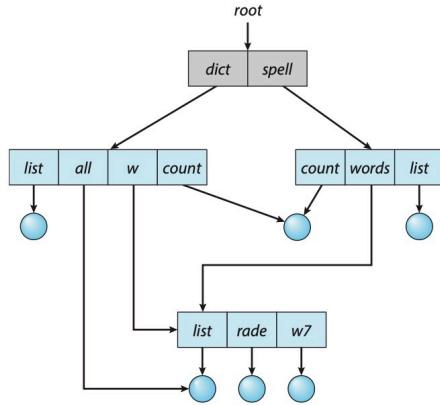
Directory con struttura ad albero aciclico

Una struttura ad albero è naturale per risolvere il problema della directory, infatti essa permette una ricerca rapida, un'ottima capacità di raggruppamento e permette file con lo stesso nome in directory diverse.

Tutte le azioni (creazione di file, creazione di directory, cancellazione di directory) avvengono a partire da una directory.

Un altro vantaggio della struttura ad albero è la possibilità di utilizzare più nomi diversi per indicare lo stesso file. Questo può essere implementato solo come reference (soft link) oppure duplicando il file in memoria (hard link).

Questo porta a dei problemi per l'eliminazione del file, è necessario infatti eliminare anche tutte le reference. Viene quindi tenuto un counter delle reference ed il file non viene eliminato finché il counter non arriva a 0.

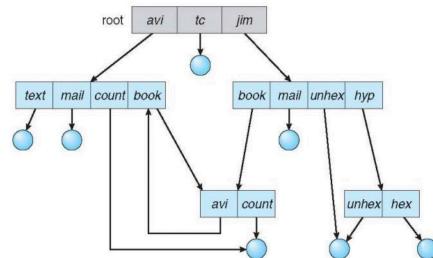


Directory con struttura ad albero generale

Se permettiamo anche la presenza di cicli nel grafo si ottengono dei problemi per l'attraversamento, vogliamo assicurarci che non ci siano file che vengono visitati due volte.

Per fare questo in fase di attraversamento è necessario marcare tutti i file visitati.

Un'altra complicazione si trova nel fatto che un blocco può avere counter delle reference diverso da 0 pur non avendo nessun blocco che lo punta, per questo viene utilizzato un algoritmo di *garbage collection* il quale attraversa l'albero e poi elimina tutti i file non contrassegnati come visitati.



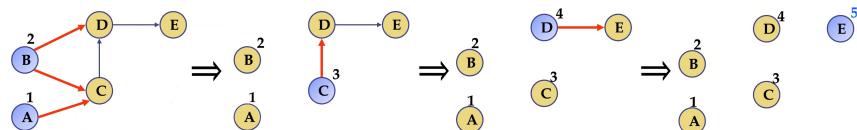
Stabilire se un grafo è aciclico

Per stabilire se un grafo è aciclico possiamo sfruttare il fatto che un grafo si dice aciclico se ha (almeno) un ordinamento topologico. Ovvero se è possibile assegnare ai nodi degli indici in modo che

$$\forall \text{ ramo orientato } e = (u, v) \in G, \text{index}(u) < \text{index}(v)$$

Algoritmo per trovare un ordinamento topologico:

- Assegniamo (a caso) numeri crescenti ai vertici privi di rami entranti che non siano ancora numerati (se non ce ne sono, il grafo ha almeno un ciclo)
- Eliminiamo dal grafo i rami uscenti dai vertici a cui abbiamo assegnato un numero
- Ripetiamo finché ci sono vertici non numerati



Implementazione

La struttura dati può essere implementata tramite una lista o una lista concatenata, ma questa soluzione ha importanti problemi sulle prestazioni.

Si preferisce utilizzare una tabella hash con chiave: nome del file, valore: puntatore all'identificatore del file. Essa porta a prestazioni migliori, anche se può avere necessità di rehash.

10.1.1 Montaggio

Per poter accedere ad un file system è necessario montarlo, ovvero attaccarne la struttura dati ad un punto di montaggio.

La procedura richiede di fornire al sistema operativo il nome del dispositivo e il punto di montaggio, ovvero una directory a cui viene agganciato il file system.

Il punto di montaggio non deve essere una directory vuota, ma questo è preferibile in quanto i file in essa contenuti non saranno visibili fino all'operazione di umount.

I sistemi macOS e Windows rilevano tutti i dispositivi all'avvio e montano automaticamente tutti i loro file system.

Nei sistemi Unix è invece necessario montare i file system manualmente dopo l'avvio.

I sistemi Linux sono simili a quelli unix, ma forniscono anche un file di configurazione che permette di descrivere quali dispositivi montare in modo automatico.

10.1.2 Esempi

Unix/Linux

Viene costituito un grafo generale di directory, che ospita tutti i file e le directory. A ciascun utente viene associata una directory come sottodirectory di `/usr`.

Ciascun file viene identificato univocamente da un *pathname*, questo significa che tutti i file e sottodirectory devono avere nomi distinti.

Ogni utente che interagisce con il file system ha un proprio contesto, visualizzabile con `pwd` che può essere modificato grazie al comando `cd`.

Esistono due tipi di link in linux, l'hard link che crea un nuovo elemento nella directory che punta allo stesso file fisico e il soft link che punta solamente ad un'altro elemento della directory.

Alla cancellazione dell'elemento gli hard link rimangono, mentre i soft link vengono tutti eliminati.

Il comando `chmod` permette di modificare il livello di protezione assegnato al file. È possibile utilizzare il comando per assegnare dei permessi sovrascrivendo quelli presenti o per modificare quelli già esistenti.

A partire da Fedora 33 è stato selezionato Btrfs come file system di default, esso fornisce dei servizi di pooling (risorse pronte per essere utilizzate), snapshots (una copia read only generata in O(1)) e checksum.

10.2 Implementazioni

Il file system è stratificato, ogni livello si serve di funzioni dei livelli inferiori per realizzare funzioni utilizzate dai livelli superiori.

- **Dispositivi Hardware**
- **Controllo I/O:** Forniscono comandi che permettono di leggere specifiche locazioni di memoria del disco (leggi dal disco 1, cilindro 72, traccia 2, settore 10 nella locazione di memoria 1060) e li traducono in sequenze di bit che fanno spostare la testina alla locazione.
- **File System di Base:** Gestisce buffer e cache del dispositivo, fornisce comandi di più alto livello (leggi il blocco 123)

- **Modulo di organizzazione dei file:** Traduce indirizzi logici in indirizzi fisici
- **File System logico:** Gestisce le strutture dati del file system.

Strutture Dati su Disco

- **Blocco di controllo di avviamento:** È la partizione di boot, contiene le informazioni necessarie all'avvio del disco, normalmente inizia dal primo blocco del disco.
 - **Blocco di controllo dei volumi:** Contengono dettagli riguardo la partizione quali: numero dei blocchi (e dimensione), contatore blocchi liberi (e i loro puntatori)
 - **Struttura delle directory**
 - **Blocchi di controlli dei file:** Contengono i dettagli di ogni file quali: permessi, date (creazione, modifica), owner, dimensione, puntatore.
- NTFS utilizza una struttura dati in stile DB relazionale per queste informazioni.

Strutture Dati su Memoria

- **Tabella di montaggio:** Contiene informazioni su ogni volume montato
- **Struttura delle directory:** Una cache della struttura dati fisica.
- **Tabella dei file aperti**
- **Tabella dei file aperti per ciascun processo**

ISO 9660

Utilizzato dalle CD-ROM

UFS

Unix File System, creato sul File System Berkley Fast (FFS)

Windows

Supporta FAT, FAT32, FAT64, NTFS

Linux

Fornisce ext2, ext3, ext4 (extended file system), ma supporta altri 40 tipi diversi.

GoogleFS

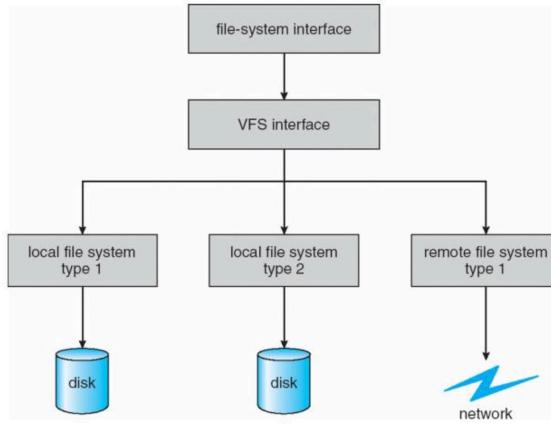
Detto anche BigFiles è un file system ottimizzato per lo storage di file di grandi dimensioni (>100GB) con poche modifiche o eliminazioni.

OracleASM

Gestisce file, directory, volumi grazie a direttive SQL.

10.2.1 File System Virtuali

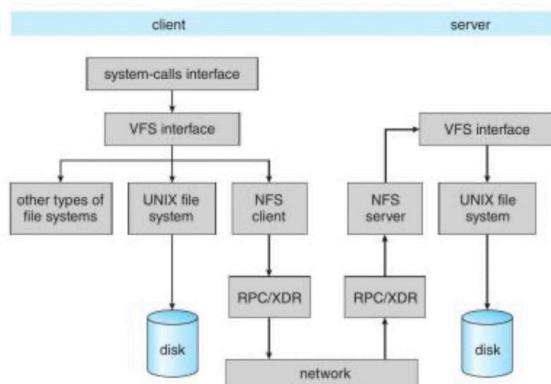
Fornisce un'interfaccia comune per l'accesso a dispositivi con file system differenti.



10.2.2 NetWork File System

Permette di unificare file system indipendenti di computer indipendenti nella rete, permette a qualsiasi utente di accedere al file system di qualsiasi altro computer.

In particolare rende possibile montare una directory remota specificando la posizione del calcolatore e quella del montaggio.



FTP

È possibile condividere file system anche attraverso la rete, il *File Transfer Protocol (FTP)* permette di visualizzare ed accedere al file system di un altro calcolatore.

Questo è un modello client-server multi-a-molti, un server può gestire le richieste di più utenti ed un utente può accedere a molti server. In questo caso per gestire l'autenticazione vengono utilizzate delle chiavi di cifratura, che introducono un'altra serie di complicazioni.

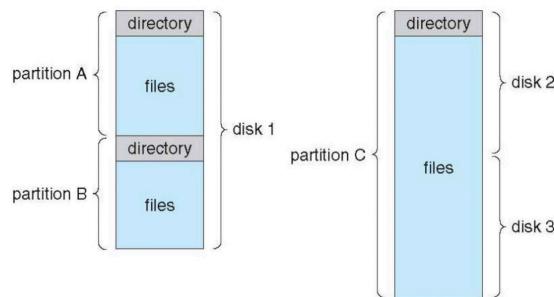
Capitolo 11

Memoria Secondaria

11.1 Struttura dei Dischi

Da una parte è possibile suddividere un singolo disco in più dischi virtuali, detti **partizioni** o **minidischi**.

Una partizione che contiene un file system è detta **volume** e possiede una directory di dispositivo che contiene informazioni di tutti i file della partizione.



Dall'altra è possibile unire più dischi per creare un disco virtuale con specifiche proprietà mediante RAID che vedremo alla fine del capitolo.

11.1.1 Formattazione

Oltre alla formattazione logica, ovvero la creazione di un file system sul disco, esiste la **formattazione fisica** o di basso livello.

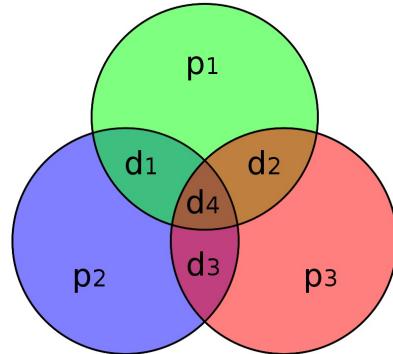
La formattazione prevede il salvataggio di una struttura dati che associa ogni settore fisico ad un settore logico, permettendo così anche la gestione degli errori, quando un settore è difettoso esso viene rimosso dalla struttura dati, rendendo ancora possibile l'utilizzo del disco.

11.1.2 Correzione degli errori

Le memorie fisiche utilizzano tecniche di memorizzazione che possono generare errori. Definiamo C come la codifica corretta e C' la codifica effettiva, la **distanza di Hamming** $H(C, C')$ è il numero di bit sbagliati nella codifica effettiva.

Per rilevare questi errori è necessario aggiungere dei bit di parità alla codifica. In particolare vogliamo aggiungerne un numero tale da permetterci di rilevare non solo se ci sono errori, ma anche quali bit sono errati. In questo modo è possibile non solo rilevare gli errori, ma anche correggerli.

Per fare ciò utilizziamo dei bit di parità che coprano ognuno un diverso gruppo di bit, così da poter individuare il dato errato. Per sapere su quali bit di parità influenza il bit k basta riscriverlo come somma di potenze di 2, quindi $7 = 1 + 2 + 4$



In questo caso se, ad esempio, i bit p_1 e p_3 ci segnalano un errore sappiamo che sicuramente il dato d_2 è errato.

Per rilevare d errori è necessario utilizzare $d + 1$ bit, mentre per correggere lo stesso numero di errori vanno utilizzati $2d + 1$ bit.

I bit di parità vengono poi confrontati tra la sequenza iniziale e quella effettivamente scritta, se ci sono discrepanze sappiamo che ci sono degli errori e, in caso, possiamo trovare i bit colpevoli.

Esempio

Se vogliamo rilevare gli errori sulla sequenza: 0 1 1 0

Inseriamo dei bit di parità in posizione 1, 2, 4: - - 0 - 1 1 0
Il bit 3 influenza 1 e 2 ($3 = 1+2$), ($5 = 1+4$), ($6 = 2+4$), ($7 = 1+2+4$)

Per calcolare i bit di parità: 1 se il numero di 1 è dispari, 0 se pari.

Quindi, ad esempio, il bit 1 è influenzato dai bit 3, 5, 7 (0 1 0 \Rightarrow 1)

Otteniamo: 1 1 0 0 1 1 0

11.2 Implementazioni

La memoria secondaria permette di mantenere una grande quantità di dati in modo permanente. I **dischi magnetici** e i **dischi a stato solido** sono i due principali mezzi per la memorizzazione di massa.

L'unità disco è connessa al calcolatore mediante un bus di I/O, il quale può essere di diversi tipi:

- **ATA** (Advanced Technology Attachment)
- **SATA** (Serial ATA)
- **USB**
- **Thunderbolt**

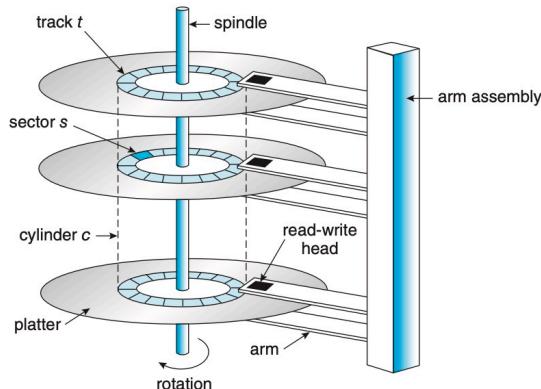
Sia dal lato del calcolatore che da quello del disco sono inseriti dei controllori per gestire il flusso di dati ed il disco.

Quando viene richiesta un'operazione I/O al disco i dati vengono prima copiati nella cache del disco e poi vengono inviati alla RAM del sistema.

11.2.1 HDD

Un Hard Disk è composto da molti dischi paralleli impilati (piatti), su ogni piatto sono presenti numerosi anelli concentrici (tracce), ogni traccia viene divisa in settori.

Diremo cilindro l'insieme di tutte le tracce poste alla stessa distanza dal centro, mentre un blocco è l'insieme di tutti i settori posti nella stessa posizione in tutti i piatti.



Ogni piatto ha una testina che ha il compito di lettura/scrittura sul disco. La maggior parte della latenza dei dischi rigidi è dovuta allo spostamento di questo elemento.

Negli anni quasi tutte le proprietà del disco sono state migliorate sostanzialmente, con l'eccezione del tempo di accesso, dettato da limitazioni fisiche.

Parameter	Started with (1957)	Developed to (2019)	Improvement
Capacity (formatted)	3.75 megabytes ^[17]	18 terabytes (as of 2020) ^[18]	4.8-million-to-one ^[19]
Physical volume	68 cubic feet (1.9 m ³) ^{[c][6]}	2.1 cubic inches (34 cm ³) ^{[20][d]}	56,000-to-one ^[21]
Weight	2,000 pounds (910 kg) ^[6]	2.2 ounces (62 g) ^[20]	15,000-to-one ^[22]
Average access time	approx. 600 milliseconds ^[6]	2.5 ms to 10 ms; RW RAM dependent	about 200-to-one ^[23]
Price	US\$9,200 per megabyte (1981) ^[24]	US\$0.024 per gigabyte by 2020 ^{[25][26][27]}	383-million-to-one ^[28]
Data density	2,000 bits per square inch ^[29]	1.3 terabits per square inch in 2015 ^[30]	650-million-to-one ^[31]
Average lifespan	c. 2,000 hrs MTBF ^[citation needed]	c. 2,500,000 hrs (~285 years) MTBF ^[32]	1250-to-one ^[33]

Implementazioni

- **CAV Constant Angular Velocity**, la velocità di rotazione resta costante per ogni traccia, così come la quantità di dati per ogni traccia. Questo significa che tracce più interne devono avere maggiore densità.
- **CLV Constant Linear Velocity** La densità rimane costante su tutto il disco, questo significa che la velocità angolare deve aumentare quando ci si sposta da tracce esterne a quelle interne.

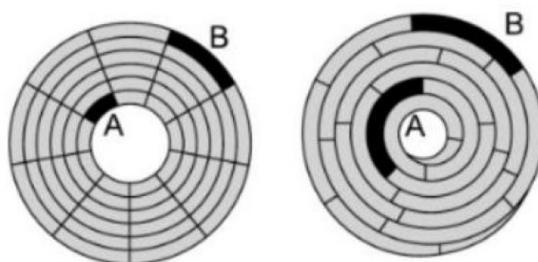


Figura 11.1: CAV a sinistra, CLV a destra

Un esempio visivo di encoding dei dati su un disco si può trovare nel paracadute usato dalla sonda *Perseverance* durante il suo atterraggio su Marte nel 2021.
Su di esso è presente un codice binario che si può leggere come un disco magnetico e riporta il motto "dare mighty things".

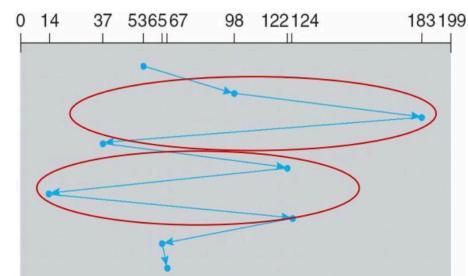


Scheduling

Le prestazioni degli algoritmi di scheduling vengono misurate su una coda di richieste per i cilindri 98, 183, 37, 122, 14, 124, 65, 67, con la testina inizialmente impostata sul cilindro 53.

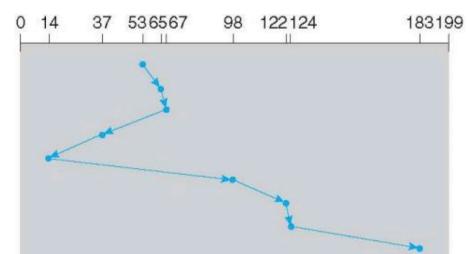
First Come First Serve

Un algoritmo spesso inefficiente in quanto può risultare in grandi oscillazioni, in questo caso muove la testina di 640 cilindri.



Shortest Seek Time First

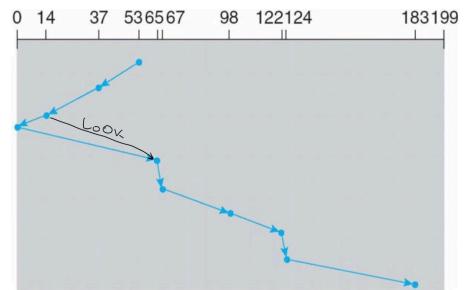
Si muove sempre verso l'elemento più vicino, questo può causare l'attesa indefinita di alcune richieste e non è ancora ottimale. In questo caso muove la testina di 236 cilindri.



Scan

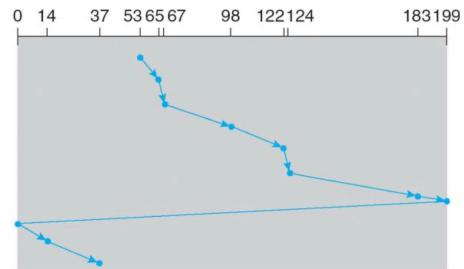
Il braccio della testina si muove da un estremo all'altro, servendo le richieste che incontra.

Una variante di questo algoritmo, detta look, ferma lo scanning prima della fine, all'ultimo cilindro richiesto. Questo richiede dell'overhead per controllare la coda, ma riduce il numero di spostamenti sprecati.



C-Scan

Simile a Scan, ma la testina compie richieste solo in una direzione, per poi ritornare velocemente all'altro estremo. Questo fornisce un tempo di attesa più uniforme.



C-Look

Una versione ottimizzata di c-scan che si ferma all'ultima richiesta senza arrivare alla fine dei cilindri.



Nei sistemi operativi moderni spesso non vale la pena di ottimizzare in modo particolare gli algoritmi di scheduling, viene quindi spesso utilizzato Shortest Seek Time First, con qualche accorgimento per evitare lo starving.

11.2.2 SSD

Sono delle memorie a stato solido, costituite a da transistor MOSFET i quali sono in grado di mantenere una determinata carica elettrica per un lungo tempo.

Nei nuovi dischi è possibile memorizzare nei transistor più livelli di tensione (oltre al solito 0 o 5V). In questo modo si aumenta notevolmente la densità, ma aumenta anche la complessità.

I dischi a stato solido hanno tempo di accesso costante per ogni cella, quindi la frammentazione del disco non è più problematica.

Al contrario ai dischi magnetici le memorie a stato solido non possono essere sovrascritte, è necessario riportare tutti i bit a 0 e poi scrivere nuovi dati. A questo si aggiunge il fatto che la scrittura dei dati è la principale causa di fallimento del disco, tuttavia grazie alle nuove tecnologie si è riusciti a limitare questo problema.

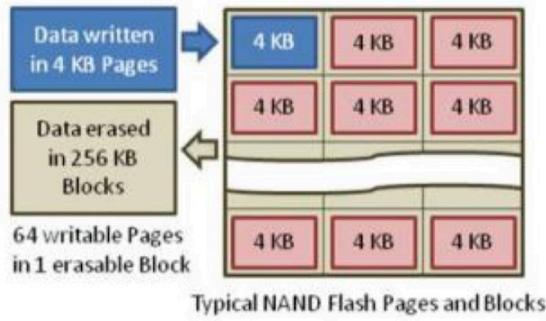


Figura 11.2: Disco rigido con pagine da 4KB e blocchi di eliminazione da 256KB

Per il fatto che i blocchi di eliminazione sono più grandi delle pagine può essere necessario copiare alcuni elementi in un’altro blocco prima di effettuare l’eliminazione.

11.2.3 Tape

I nastri magnetici sono ancora utilizzati per alcune specifiche applicazioni, in particolare di backup. Essi infatti, se tenuti correttamente, possono durare per 100 anni e sono estremamente capienti (i più nuovi fino a 580 Tb).

Essi sono prettamente sequenziali, quindi il tempo di accesso è spesso molto elevato.

11.2.4 DNA

Sembra che il futuro della memorizzazione dei dati si trovi nel DNA, il quale può essere utilizzato per archiviare enormi quantità di dati per un periodo di tempo estremamente lungo (215 petabyte in 1 grammo per migliaia di anni).

Per ora è una tecnica che ha ancora costi proibitivi, per scrivere 2 megabyte sono necessari 7000 dollari e altri 2000 per leggerli.

11.3 Misurazione delle Prestazioni

L’interfaccia SATA inserisce le richieste I/O in una singola coda con lunghezza massima di 256. Per questo motivo i dischi a stato solido (SSD) utilizzano l’interfaccia PCIexpress, che fornisce 64000 code ognuna con lunghezza massima di 640000.

Questo ci permette di comprendere meglio gli indicatori utilizzati per misurare le prestazioni i quali utilizzano degli indicatori: [Seq] [block size] Q[queue size] T[thread number]

- Seq indica se la lettura avviene in modo sequenziale, se non è presente si assume che gli accessi siano casuali
- È possibile specificare la dimensione dei blocchi che vengono letti
- Dopo la lettera Q troviamo la quantità di elementi che sono inseriti nella queue
- Dopo la lettera T troviamo il livello di multiprogrammazione utilizzato.

Definiamo il **tempo medio di I/O** come:

$$T_{\text{medio, I/O}} = \text{seek time} + \text{latenza media} + \frac{\text{quantità di dati da trasferire}}{\text{velocità di trasferimento}} + \text{overhead}$$

11.4 RAID

Redundant Array of Independent Disks (RAID) è una tecnologia che permette di introdurre della **ridondanza** nel salvataggio dei dati, rendendo il sistema in grado di gestire il fallimento di alcune componenti hardware.

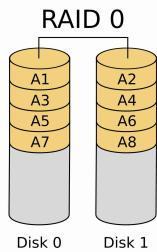
Un'altra applicazione si trova nella **parallelizzazione** delle richieste su più dischi, aumentandone così le prestazioni.

In un RAID si utilizzano sempre la stessa tipologia di dischi (HDD, SSD), in quanto il RAID funziona sempre alla velocità del disco più lento.

Il RAID può essere implementato in software, rischiando di intaccare le prestazioni dell'intero sistema, oppure in hardware mediante un apposito controller che gestisce le letture/scritture e fornisce al sistema un'interfaccia standard.

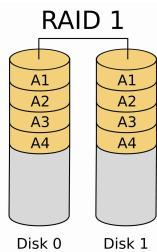
RAID 0

Il RAID 0 o *data striping* tratta un gruppo di dischi come un'unica unità, ogni blocco di dati è diviso tra di essi, permettendo così la lettura e la scrittura parallela.

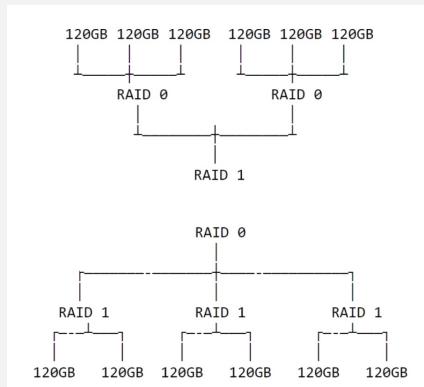


RAID 1

Prevede la duplicazione di tutti i dati in più dischi, in questo modo si introduce una *ridondanza* che aumenta notevolmente l'affidabilità del sistema. La lettura avviene in parallelo, ma la scrittura avviene alla velocità del disco più lento.



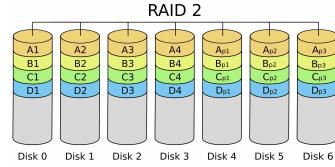
RAID 0 e 1 vengono spesso utilizzati in combinazione per ottenere una combinazione dei loro vantaggi.



RAID 2

Architettura poco diffusa a livello commerciale, striping a livello di bit, movimento parallelo delle testine. Gli errori vengono rilevati mediante codici di Hamming. Questo permette di rilevare sia l'errore che il disco problematico.

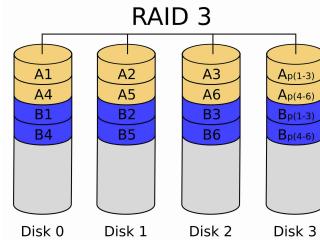
Il numero di dischi di parità è uguale al \log_2 del numero totale di dischi, arrotondato per eccesso.



RAID 3

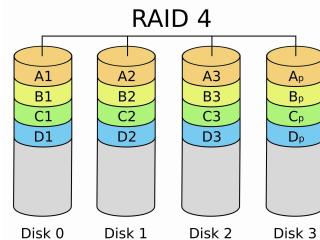
Funzionamento simile a RAID 2, striping a livello di bit e movimento parallelo delle testine.

Un disco viene utilizzato per la parità, permettendo al sistema di accettare una perdita di un disco.



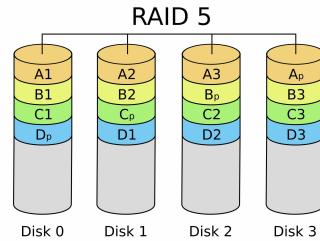
RAID 4

Striping a livello di blocco, testine indipendenti e utilizzo di un solo disco di parità. In questo modo le letture piccole richiedono un solo disco, e possono essere svolte in modo parallelo. Mentre le scritture richiedono anche l'accesso al disco di parità.



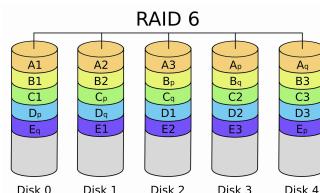
RAID 5

Sia RAID 3 che RAID 4 hanno un solo disco di parità, il che significa che è un bottleneck in quanto limita le operazioni parallele che possono essere svolte dai dischi. Il RAID 5 distribuisce questi bit su più dischi, migliorando le prestazioni. Inoltre ha striping a livello di blocco con testine indipendenti.



RAID 6

La limitazione del RAID 5 è che è in grado di tollerare la perdita di un solo disco, il RAID 6 aumenta questo margine a 2 dischi.



Appendice A

Sistemi Operativi Moderni

Questa parte copre le dispense opzionali fornite dal professore sui moderni sistemi operativi, quindi Linux, Windows e Android.

Questa è una sintesi estremamente compatta, non comprende tutto ciò che viene scritto nelle 230 pagine di dispense, e non pretende di essere completa.

Fornisce però una spiegazione di come vengono applicati i concetti visti a lezione nei vari sistemi operativi e una breve storia di essi, ovvero ciò che viene chiesto più di frequente all'esame.

A.1 Linux

A.1.1 Storia

Negli anni '40 e '50 i computer venivano spesso condivisi tra diversi utenti in quanto essi erano principalmente utilizzati da ricercatori per i loro calcoli. Per questo motivo vengono inventati dei sistemi operativi adatti a gestire il time sharing di più utenti, **MULTICS** è uno di essi, sviluppato dai ricercatori di MIT, Bell Labs e General Electric.

Uno dei ricercatori di Bell Labs, Ken Thompson, iniziò a scrivere una versione ridotta di MULTICS, adatta a gestire solamente un utente, per questo motivo venne chiamato ironicamente **UNICS**.

Porting

Nel tempo questo sistema operativo viene portato su diversi computer, per rendere più agevole il processo si cerca di riscriverlo in un linguaggio ad alto livello. Inizialmente Bell Thompson crea il linguaggio **B**, una semplificazione del BCPL, ma non riesce a portare a termine questo progetto a causa di alcune debolezze del B (in particolare la sua mancanza di struct).

Con l'aiuto di Ritchie progetta quindi un successore al B, chiamandolo (ovviamente) **C**, con il quale riescono a riscrivere UNIX in un linguaggio ad alto livello.

Dopo la scrittura in C altro lavoro deve essere fatto per rendere UNIX facilmente portabile, si è scoperto infatti che il sistema faceva alcune assunzioni riguardo all'hardware su cui veniva eseguito.

Separazione e riconciliazione

La licenza per UNIX, assieme al suo codice sorgente, vennero forniti facilmente a diverse università, le quali continuarono il lavoro iniziato e distribuirono ulteriormente il sistema operativo.

In particolare la versione sviluppata all'università di Berkley ha ricevuto parecchi miglioramenti rispetto alla versione di Bell Labs, rendendola la versione più popolare negli ambienti accademici.

A questo punto le versioni dei UNIX si sono però divise e i programmi non sono interscambiabili, rendendo difficile la distribuzione di software. Per ovviare al problema l'IEEE fornisce uno standard di procedure di libreria che ogni sistema operativo avrebbe dovuto supportare, esso viene chiamato **POSIX**.

MINIX

MINIX è un'altro sistema operativo nato dal codice sorgente di UNIX, ma viene creato con l'obiettivo di ottenere le massime prestazioni e di essere interamente comprensibile da un singolo sviluppatore.

Questo ha portato MINIX ad essere estremamente limitato e quando gli utenti chiedevano se fosse possibile aggiungere certe funzionalmente si sentivano rispondere di no perché il codice sarebbe diventato troppo grande.

Linux

Questi continui "no" hanno portato molti utenti a cercare un'alternativa, e la trovarono in **Linux**, un sistema operativo sviluppato da uno studente finlandese, Linus Torvalds.

Il codice di questo sistema operativo esplode rapidamente con molti utenti felici di contribuire con le loro migliorie, arrivando a Linux 5.11 dove il kernel è composto da 30 milioni di righe di codice. Linux nel tempo è rimasto un software libero, il cui codice sorgente può essere letto e modificato da tutti. Esso è coperto dalla licenza **GPL** che specifica cosa può essere fatto e cosa no, in particolare non è possibile modificare il codice e redistribuirlo solamente in forma binaria.

Obiettivi

Linux viene concepito come un sistema operativo scritto da programmatori, per programmatori. Fornendo quindi un sistema semplice, elegante e coerente.

Altre caratteristiche importanti per i programmatori sono il "principio della minima sorpresa", potenza, flessibilità e assenza di rindondanza.

A.1.2 Processi

Ogni processo in linux viene rappresentato attraverso una `task_struct` che contiene tutte le informazioni.

Deamon

I deamon sono dei processi di background che vengono eseguiti periodicamente.

Fork

La chiamata a sistema fork è di particolare importanza, essa permette di creare una copia esatta del processo su cui viene chiamata.

Essa ritorna l'id del processo figlio al genitore e ritorna 0 al figlio, questo permette di distinguere il contesto.

```
pid = fork();
if (pid < 0) {
    handle_error();
} else if (pid > 0) {
    /* codice genitore qui.*/
} else {
    /* codice figlio qui.*/
}
```

I figli utilizzano la tecnica del copy on write per non creare troppo overhead alla creazione di un nuovo thread.

Clone

Una funzione introdotta nel 2000 è clone, la quale ha sfumato la distinzione tra processo e thread. Infatti grazie a questa system call è possibile specificare quali informazioni del genitore devono essere condivisi al figlio e quali no.

Comunicazione

Per la comunicazione tra processi è possibile utilizzare delle **Pipe**, oppure degli interrupt software, detti **Signal**.

A.1.3 Scheduling

Le politiche implementate da linux sono:

- **Real Time FIFO:** priorità massima, non possono essere prelati se non da altri processi a priorità maggiore.

- **Real Time Round Robin:** alta priorità, vengono prelati solo da processi a priorità maggiore o quando termina il quanto di tempo.
- **Sporadici:** bassa priorità, vengono eseguiti quando ci sono delle risorse libere.
- **Time Sharing:** priorità minima, vengono eseguiti con priorità minore di quelli sporadici.

Le categorie Real Time condividono le priorità 0-99, le altre due le priorità 100-139. La priorità può anche essere modificata da un valore "cortese" (nice), che ha valore da -20 a +19. Un processo che sa di essere a bassa priorità può ridurla tramite una chiamata a sistema.

Queste politiche vengono implementate dal **CFS** (*Completely Fair Scheduler*), il quale cerca di dare a tutti i processi lo stesso tempo di CPU. Per implementare la priorità la velocità effettiva del tempo viene modificata, è più lenta per i processi ad alta priorità e più veloce per quelli a bassa priorità.

A.1.4 Avvio

Il bootloader di default di Linux è **GRUB** (GRand Unified Bootloader), a seguito viene avviato il kernel.

Esso inizializza prima l'hardware (identificazione tipo CPU, RAM, attivazione MMU, disattivazione interrupt), e successivamente il software (allocazione buffer dei messaggi, strutture dati kernel, configurazione dei dispositivi I/O).

Poi viene creato il processo 0 il quale svolge altre inizializzazioni software e poi crea il processo 1 (init) e il processo 2 (page deamon).

Infine viene eseguito un programma chiamato getty che mostra all'utente una schermata di login.

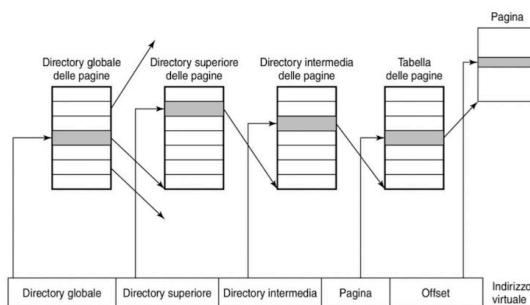
A.1.5 Gestione della Memoria

In linux su sistemi a 64 bit vengono utilizzati solo 48 per l'indirizzamento della memoria principale, fornendo così un limite teorico di 258 Tb di memoria, e 128 Tb per ogni processo.

Le architetture degli elaboratori suddividono la memoria in diverse aree:

- **ZONE_DMA**
- **ZONE_NORMAL**
- **ZONE_HIGHMEM** che non viene mappata in modo permanente

Linux per indicizzare la memoria utilizza uno schema di paginazione a 4 livelli



Allocatore

L'allocatore di memoria utilizzato da linux è una combinazione dell'algoritmo Buddy (potenza a 2) e un algoritmo slab.

La memoria viene prima suddivisa dall'algoritmo buddy, poi le varie zone sono gestite dall'algoritmo slab.

Paginazione

La memoria viene paginata solamente su richiesta, l'algoritmo di reclamo delle pagine suddivide le pagine in 4 categorie:

- **unreclaimable** Pagine bloccate e non recuperabili.
- **swappable** Pagine che devono essere scritte sulla zona di swap del disco prima di poter procedere.
- **syncable** Pagine il cui contenuto è stato modificato, quindi devono essere riportate sul disco.
- **discarable** Pagine che possono essere eliminate direttamente.

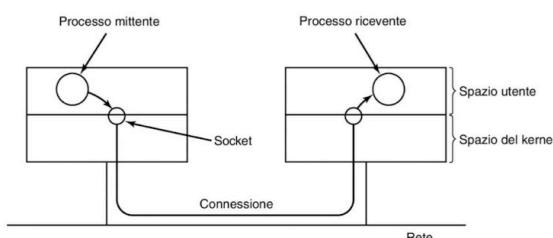
I/O

I dispositivi I/O vengono trattati da Linux come dei **file speciali**, montati nel file system virtuale.

Essi sono divisi in due tipologie:

- **file speciali a blocchi**: sequenze di blocchi numerati ad accesso causale, ideali per i dischi.
- **file speciali a caratteri**: flusso di caratteri, utili per tastiere e stampanti.

Il networking viene implementato mediante dei socket che permettono al sistema di interfacciarsi ad un cavo.



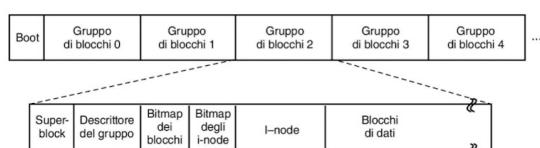
Ogni socket supporta un particolare tipo di rete:

- flusso di byte con connessione affidabile (TCP)
- flusso di pacchetti con connessione affidabile
- flusso di pacchetti con connessione non affidabile (UDP)

File System

ext2

Il primo file system implementato da Linux è ext2, il quale suddivide il disco in blocchi, ognuno dei quali ha una sezione detta **superblocco** che contiene informazioni sulla struttura della sezione. Poi contiene due bitmap, una per i blocchi liberi e una per gli i-node liberi. Infine vengono salvati i blocchi e gli i-node.



ext4

Il file system ext4 introduce il journaling, ovvero le istruzioni vengono salvate dal sistema operativo prima di essere inviate al disco e vengono eliminate quando completate.

Questo significa che in caso di terminazione inaspettata il sistema può riportare il disco allo stato atteso.

A.1.6 Sicurezza

In linux la sicurezza e la condivisione vengono implementate mediante un id dell'utente e un id del gruppo. Ogni file è contrassegnato dall'identificativo dell'utente e del gruppo proprietari.

A questo punto è sufficiente specificare con 9 bit i permessi, 3 per il proprietario, 3 per il gruppo e 3 per gli esterni.

I 3 bit sono:

- read
- write
- execute

A.2 Android

A.2.1 Storia

Android nasce come una startup con l'idea di portare il sistema operativo linux ai nuovi dispositivi mobili, ma viene presto acquisita da Google nel 2005.

Inizialmente si vuole sviluppare un sistema in grado di supportare applicazioni sviluppate in Java, C e C++, ma presto viene selezionato Java con l'obiettivo di semplificare le API di sistema.

Android 1.0

Nel settembre del 2009 viene rilasciato Android 1.0 per un dispositivo mobile con un grande touch screen, il Dream.

Successivi miglioramenti

A seguito del rilascio verranno fatti ben 15 aggiornamenti maggiori in soli 5 anni, nei quali le funzionalità di Android crescono enormemente e si ottiene una separazione completa tra il sistema operativo open source e il codice proprietario Google.

Il sistema operativo android rimane open source, accessibile gratuitamente a tutti i produttori di dispositivi che vogliono utilizzarlo.

Per assicurarsi che i sistemi non si allontanino troppo a causa delle modifiche android fornisce il CDD, uno standard che tutti i dispositivi devono supportare per poter utilizzare il Play Store.

Obiettivi

Android viene costruito con degli specifici obiettivi in mente, i quali sono leggermente diversi rispetto a quelli di Linux.

- Fornire una piattaforma **open source** completa per i dispositivi mobili
- Consentire a tutte le applicazioni di terze parti di **competere allo stesso livello**, il codice open source di Android è progettato per essere neutrale.
- Fornire un livello di **sicurezza** delle applicazioni tale per cui gli utenti non si debbano preoccupare di cosa stanno installando.
- L'utilizzo dei dispositivi mobili è fondamentalmente diverso da quello dei desktop, le applicazioni devono essere più reattive, con **tempi di risposta minimi** (200 ms apertura a "freddo").
- Le applicazioni svolgono tutte azioni specifiche, è necessario che android faccia da **collante** per creare qualcosa di più grande.

A.2.2 Estensioni di Linux

Android si trova ad affrontare problemi fondamentalmente diversi rispetto a quelli affrontati da linux su un ambiente desktop o server.

Per questo motivo sono necessarie delle modifiche al kernel linux/

Wake Lock

È un lock che blocca lo stato di sleep del dispositivo, esso viene tenuto dal sistema operativo quando il dispositivo è attivo, dalle applicazioni in background quando il display è spento. Quando nessuno tiene più il lock il dispositivo va in sleep.

Out-of-memory killer

Una situazione che avviene spesso sui dispositivi mobile e di rado nei desktop è quella di finire la memoria di sistema.

In questo caso Android introduce il suo Out-of-memory killer che opera in modo più aggressivo quando il sistema va oltre un certo utilizzo della memoria

ART

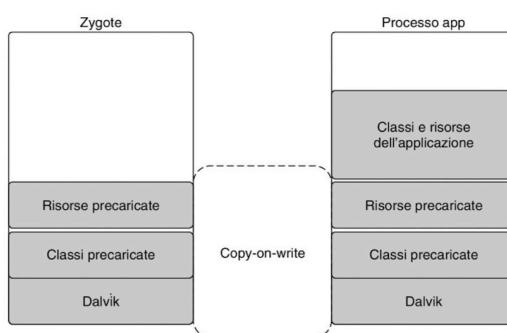
ART (Android RunTime) è uno strumento che implementa in Android l'ambiente del linguaggio Java.

Esso richiede un particolare bytecode, che può essere ottenuto dal bytecode standard di Java, introducendo quindi un ulteriore step di compilazione.

Android non è responsabile dei processi, i quali vengono gestiti come normali processi Linux, con un ambiente di ART.

In questo modo Android può sfruttare la gestione dei processi di Linux e i programmatore possono beneficiare delle API di ART.

Per rimanere al di sotto del massimo di 200 ms di avvio l'ambiente ART non viene inizializzato ad ogni apertura di un'applicazione. Tutte le applicazioni sono figlie di un processo



Binder IPC

Binder IPC (Inter-Process Communication) è un meccanismo di comunicazione tra processi, utilizzato per permettere lo scambio di dati tra diverse applicazioni e servizi in esecuzione su un dispositivo.

Il Binder opera come un driver nel kernel di Linux, offrendo un'interfaccia ad alte prestazioni per la comunicazione tra processi.

Il funzionamento del Binder si basa su un modello client-server, dove i processi client inviano richieste ai processi server. Il server elabora la richiesta e invia una risposta al client.

Una delle caratteristiche chiave del Binder IPC è la sua capacità di trasportare riferimenti a oggetti tra processi, mantenendo la sicurezza e l'integrità dei dati.

Esso gestisce anche i permessi e l'autenticazione, assicurando che solo i processi autorizzati possano comunicare tra loro. Questo è essenziale per la sicurezza del sistema, specialmente in un ambiente mobile dove molte applicazioni potrebbero provenire da fonti non fidate.

Android utilizza il Binder IPC per numerose operazioni interne, come la comunicazione tra i servizi di sistema e le applicazioni. Questo approccio centralizzato permette un controllo rigoroso sulla gestione delle risorse e migliora la stabilità del sistema.

Applicazioni Android

Un'applicazione android non è un file eseguibile, ma un contenitore di tutto ciò che costituisce l'app.

I contenuti fondamentali sono:

- Un manifesto che contiene: cos'è l'applicazione, cosa fa e come deve essere eseguita
- Le risorse
- Il codice
- Informazioni sulla firma digitale

Non viene fornito un'unico punto di ingresso all'applicazione, ma essi si dividono in:

- **Attività:** Una parte dell'applicazione che interagisce direttamente con l'utente, il punto principale di accesso all'app.
- **Ricezione:** È un destinatario di eventi, quando il sistema operativo vuole inviare un evento lo fa ad ogni ricevitore installato.
- **Servizio:** I quali possono avere due sfumature:
 - Un'attività di background mantenuta per un lungo tempo.
 - Un punto di connessione per altre applicazioni.
- **Fornitura Contenuti:** Il principale strumento utilizzato per lo scambio di dati tra diverse applicazioni. Fornisce un'API alla quale altre applicazioni possono fare richieste e ottenere risposte.

Sicurezza

In Android la sicurezza viene implementata con un metodo di minimi permessi. Un'applicazione non ha accesso a nessun dato sensibile se non approvato esplicitamente dall'utente.

A.3 Windows

A.3.1 Storia

Negli anni '80 IBM stava sviluppando un nuovo personal computer, per il quale volevano utilizzare il sistema operativo CP/M di Digital Research.

Tuttavia il presidente non volle incontrare IBM, quindi essi andarono da Microsoft che sviluppò in breve tempo il sistema MS-DOS.

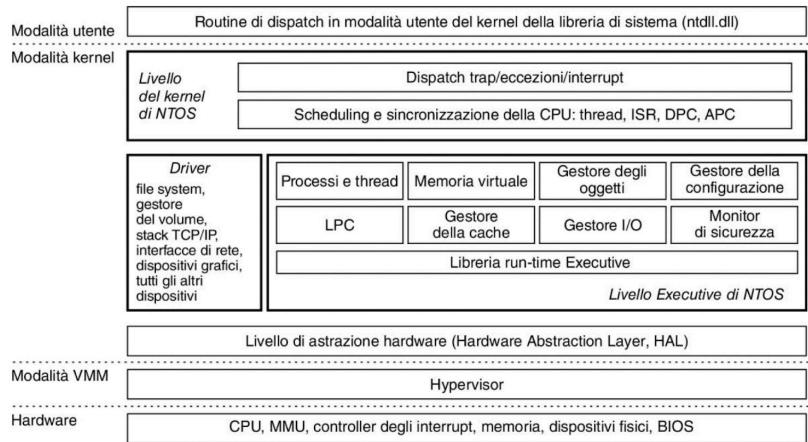
Alla fine degli anni '80 MS-DOS acquisisce un'interfaccia utente e prende il nome di Windows.

Nel 1993 Windows NT (New Technology) viene rilasciato, una completa riscrittura del sistema in quanto MS-DOS non era adatto ai computer del tempo.

Nel 2006 Windows Vista porta una riprogettazione grafica del sistema.

Con Windows 8 Microsoft prova ad avvicinarsi al mondo dei dispositivi mobile, con scarso successo, Windows 10 fornisce queste funzionalità, ma solamente agli utenti che ne necessitano invece che a tutti.

A.3.2 Struttura del Sistema Operativo



Il primo strato sopra all'hardware è formato dall'hypervisor il quale gira sopra l'hardware e supporta l'esecuzione concorrente di più sistemi operativi. Esso sfrutta le estensioni di virtualizzazione delle CPU per ripartire le risorse di CPU, memoria, ecc...

Al di sopra troviamo l'HAL (Hardware Abstraction Layer), esso realizza l'astrazione di dettagli hardware di basso livello, come l'accesso ai registri o il DMA.

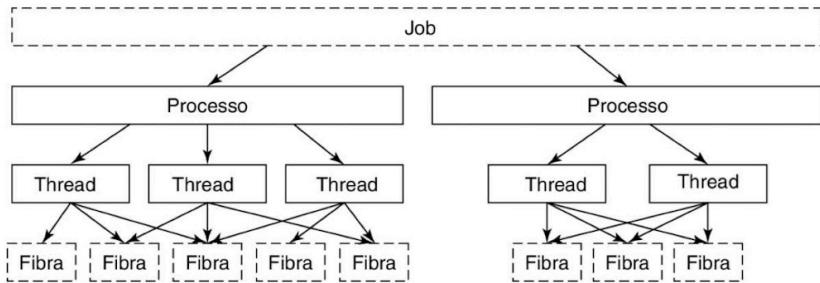
Salendo ancora si arriva al livello kernel di NTOS, esso si occupa di operazioni low level di scheduling e sincronizzazione dei processi. Il resto del kernel, ovvero il livello executive di NTOS, costruisce su questo livello, permettendo una programmazione simile a quella a livello utente.

Infine il livello executive di NTOS è largamente indipendente dall'architettura e supporta una gran quantità di funzionalità:

- **Gestore degli oggetti**: gestisce gli oggetti in modalità kernel come processi, thread, file, semafori, dispositivi, driver di I/O e timer
- **Gestore dell'I/O**: Fornisce la struttura per implementare i driver dei dispositivi e vari servizi del sistema, infatti i driver offrono anche estensibilità al sistema operativo.
- **Gestore dei processi**
- **Gestore della memoria**
- **Gestore della cache**
- **Security Reference Monitor**: Impone complessi meccanismi di sicurezza, evoluti dai requisiti del dipartimento della difesa degli Stati Uniti.
- **Gestore della configurazione**: Implementa il registro di sistema che contiene i dati della configurazione di sistema.

A.3.3 Processi

In Windows oltre a processi e thread vengono introdotti anche i job, che raggruppano più processi e le fibre, contesti di esecuzione schedulati in modo cooperativo.



Al contrario di unix la procedura di creazione di un thread non è così leggera come una chiamata a `fork()`, questo ha dei vantaggi, ma anche un aumento del tempo di esecuzione.
Per questo motivo possono essere usate delle thread pool, un gruppo di thread attivi a cui possono essere fornite delle istruzioni.

Scheduler

Windows utilizza delle classi di priorità sia sui processi che sui thread, ottenendo poi le vere priorità da una tabella.

I processi possono avere come priorità: real-time, alta, sopra al normale, normale, sotto al normale e inattivo.

I thread possono avere come priorità: time critical, altissima, sopra il normale, normale, sotto il normale, bassissima e inattivo.

ThreadPriorities	Process Classes					
	Realtime	High	Above Normal	Normal	Below Normal	Idle
Time Critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above Normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below Normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

Il sistema mantiene poi 32 liste di thread e quando deve selezionare un processo prende il primo processo che trova scendendo le liste.

Dopo che il processo ha eseguito il suo quanto di tempo esso viene rimesso in coda alla stessa lista.

Per migliorare la scalabilità dell'algoritmo più set di 32 liste possono esistere, alcune specifiche ad un singolo processore, altre comuni ad un set di processori.

Emulazione x86

Fino a Windows 7 per eseguire applicazioni a 32 bit su sistemi a 64 si utilizzava una macchina virtuale, tuttavia questo risulta inefficiente e difficile da nascondere all'utente.

L'approccio moderno è quello di utilizzare WoW64, uno strumento che traduce le chiamate a sistema delle applicazioni a 32 bit in chiamate a sistema per x64.

Emulazione x64

Su Windows 11 viene aggiunto il supporto per l'emulazione di applicazioni x64 su sistemi arm64.

I binari di sistema intesi per il caricamento di applicazioni x64 sono in un formato ARM64EC (Emulation Compatible), ovvero codice macchina arm64 compilato con comportamento e tipi per x64.

Il resto dei binari di sistema sono compilati in ARM64X, e contengono sia codice arm64 che codice compatibile con x64, in base al processo richiedente viene caricata la versione corretta.

A.3.4 Gestione della memoria

Windows utilizza 48 bit per l'indicizzazione della memoria, fornendo 258 TB di memoria ai processi, 128 per la modalità utente e 128 per la modalità kernel.

Utilizza una paginazione pura a 4 livelli con pagine di 4KB, 2MB o 1GB.

Le pagine virtuali sono dette **non valide** finché non sono mappate ad un oggetto, vengono poi dette **committed** quando una pagina fisica può venir allocata su richiesta. Infine le pagine virutali possono essere **reserved**, per indicare che sono libere, ma assegnate ad un processo.

Windows ha diverse funzionalità di prefetching:

- Quando avviene un page fault Windows carica sempre 64KB di memoria, includendo pagine vicine al page fault.
- Quando viene avviata un'applicazione Windows salva le pagine che vengono subito utilizzate e le carica in memoria ai riavvii successivi.
- **Superfetch** ovvero il fetching di pagine spesso utilizzate quando il sistema è in idle.

A.3.5 Gestione dell'I/O

Windows include il supporto per il plug-and-play che permette al sistema di rilevare e configurare automaticamente i nuovi dispositivi hardware.

I driver dei dispositivi devono essere conformi con il Windows Drier Model e spesso vengono eseguiti in modalità utente, così da limitare i possibili danni di un errore.

A.3.6 File System

Windows supporta diversi file system, i più importanti dei quali sono FAT-16, FAT-32, NTFS (NT File System) e ReFS (Resilient File System).

ReFS è il file system più nuovo, prende il nome di Resilient File System perché uno dei suoi obiettivi progettuali è ripararsi da solo: può verificarsi e ripararsi automaticamente senza downtime. Ciò è possibile mantenendo metadati di integrità per le strutture dati su disco, oltre che per i dati utente.

NTFS

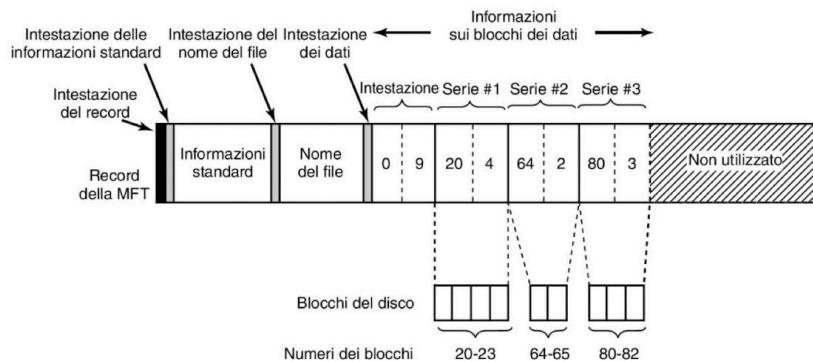
NTFS, come ext4 è un file system Journaled, sviluppato assieme a Windows NT ed è di default da quel momento.

I nomi dei file NTFS sono unicode con un limite di 255 caratteri, quindi tutti possono utilizzare la propria lingua nello scrivere il nome dei file.

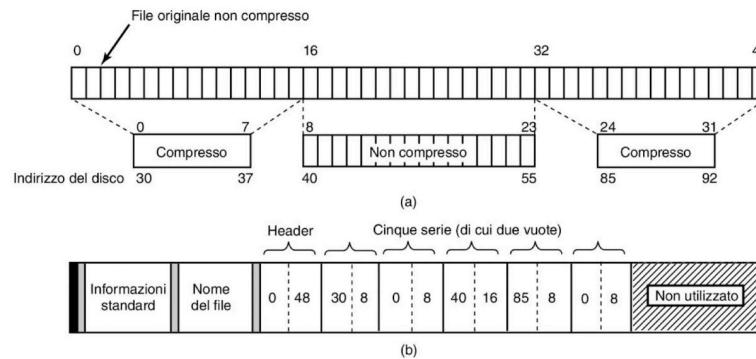
Ciascun volume NTFS è organizzato come una sequenza lineare di blocchi spesso di 4 KB come compromesso tra blocchi grandi (per trasferimenti efficaci) e blocchi piccoli (per la ridotta frammentazione interna).

La struttura dati principale in ciascun volume è la MFT (Master File Table), una sequenza lineare di record di file e directory. Ciascun elemento della tabella contiene gli attributi del file, come il suo nome e i timbri temporali, e la lista degli indirizzi del disco dove sono posizionati i suoi blocchi. Se un file è estremamente grande, qualche volta è necessario usare due o più record della MFT per contenere la lista di tutti i blocchi, e in tal caso il primo record della MFT, chiamato record base, punta agli altri record della MFT.

Vediamo una tipica MFT per un file di 9 blocchi.



NTFS supporta anche la compressione trasparente all'utente, essa funziona nel seguente modo: Il file viene diviso in sezioni di 16 blocchi, su ognuna si prova ad applicare un algoritmo di compressione, se il risultato è di 15 blocchi o meno quella sezione del file viene compressa e scritta.



Windows infine fornisce anche Bitlocker, uno strumento di cifratura del disco, questo è particolarmente utile per i laptop che sono più semplici da smarrire o rubare.

Appendice B

Programmazione Concorrente

Durante il corso viene consigliato lo studio del linguaggio di programmazione Go in quanto rende semplice la creazione di programmi che supportano concorrenza.

B.1 Download

Pagina di Download: <https://go.dev/dl/>

B.2 Sintassi

B.2.1 Packages

I *package* sono alla base di ogni programma in Go, non è infatti possibile creare un programma senza il package main.

La struttura del progetto dovrebbe essere:

```
progetto/
    utils/
        sum.go
    main.go
    go.mod
```

Dove nel file `go.mod` inseriamo una definizione del progetto: `go.mod`:

```
module progetto
go 1.22
```

Il package main deve contenere una funzione main dalla quale l'esecuzione del programma avrà inizio. Nel file `main.go` inseriamo quindi:

```
package main

import (
    "fmt"
    "progetto/utils"
)

func main() {
    fmt.Println(utils.Sum(2, 3))
}
```

Dal file `sum.go` vengono esportate tutte le variabili e funzioni che iniziano con la lettera maiuscola, le altre rimangono locali al file.

B.2.2 if/switch

Gli `if` in go hanno una sintassi simile agli altri linguaggi, viene però aggiunta la possibilità di inserire un piccolo statement prima della condizione.

```
if v := math.Pow(x, n); v < lim {
    return v
}
```

Gli switch possono avere una o più condizioni

```
switch condition {
    ^Icase x:
    ^I^I...
```

```

    ^Icase y, z:
    ^I^I...
    ^Idefault:
    ^I^I...
}

```

B.2.3 Cicli

In Go esiste un solo ciclo, il ciclo for, il quale ha una sintassi che può essere adattata per diventare un while ed un foreach.

```

for i:=1; i<10; i++{ // for loop
    ...
}

for num < 8 { // while loop
    ...

for i := range sl { //foreach
    fmt.Println(i, sl[i])
}

```

B.2.4 Funzioni

In Go le funzioni hanno una sintassi semplice, vediamo la stessa funzione in una forma estesa e poi in una più compatta

```

func calculate(x int, y int) (int, int) {
    var sum = x + y
    min := x - y
    return sum, min
}

func calculate(x, y int) (a, b int) {
    a = x + y
    b = x - y
    return //naked return
}

```

B.2.5 struct

Sono una collezione di campi

```

// Definizione
type Books struct{
    title string
    author string
}

// Costruttore (di default)
b := Books {"titolo", "autore"}

// Accesso ai membri
b.title
b.author

```

B.2.6 defer

Permette di eseguire uno o più statement (in questo caso sono inseriti in uno stack LIFO) al termine della funzione chiamante.

```
defer fmt.Println("primo")
fmt.Println("secondo")
// output => "secondo", "primo"
```

B.3 Concorrenza

Gli strumenti principali per la concorrenza in Go sono le goroutine, ovvero dei thread che vengono avviati da un'altro thread mediante la keyword go.

```
func foo(){
    fmt.Println("foo")
}

func main() {
    go foo() // main avvia foo in un altro thread
    fmt.Println("main")
}
```

In questo caso spesso "foo" non viene stampato in quanto termina prima l'esecuzione della funzione main(), che di conseguenza termina il programma e tutte le sue subroutine.

B.3.1 Channels

I channels sono degli strumenti che permettono la comunicazione tra due threads in Go.

Vengono creati utilizzando la funzione `make(chan int, 100)`, dove il primo argomento specifica il tipo e il secondo il buffer.

Si utilizza -> per inviare e ricevere i dati.

Un thread si blocca fino a che l'istruzione di inviare o ricevere un elemento non si è completata. Da un channel chiuso si ottiene l'elemento zero della classe.

B.3.2 Select

Statement che permette di attendere il primo thread a completare l'esecuzione. La keyword default permette di continuare l'esecuzione quando non sono presenti elementi nei channel, default -> attende una certa quantità di tempo prima di proseguire.

```
func main(){
    c1 := make(chan int)
    c2 := make(chan int)
    ...
    select{
        case x := <- c1:
            ...
        case x := <- c2:
            ...
        case <-time.After(time.Duration(rand.Intn(5))*time.Second)
            ...
    }
}
```

B.3.3 WaitGroup

Per attendere che un gruppo di thread completi l'esecuzione, si può utilizzare WaitGroup.

- `waitgroup.Add(n)` aggiunge all'oggetto un numero n di thread da aspettare
- `waitgroup.Done()` comunica che il thread ha completato l'esecuzione
- `waitgroup.Wait()` attende che tutti i thread abbiano chiamato Done()

```
func worker(id int) {
    fmt.Printf("Worker %d starting\n", id)

    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}

func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        wg.Add(1)

        go func() {
            defer wg.Done()
            worker(i)
        }()
    }
    wg.Wait()
}
```

L'oggetto waitgroup deve essere passato come puntatore