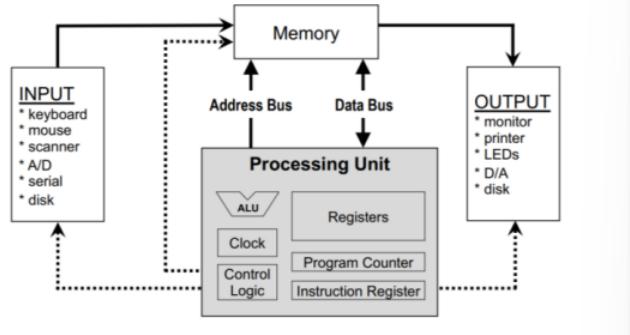


# Capitolo 0 - Architettura

## La macchina di Von Neumann (1946)

Von Neumann (matematico danese) delineò l'architettura dei calcolatori che ancora si utilizza.

In questo schema, la particolarità è che esiste una sola memoria che registra i dati e i programmi indistintamente.



Von Neumann Bottleneck: questo modello soffre di lentezza di memoria; nonostante ciò, il modello in questione è adottato dai computer odierni siccome vengono presi degli accorgimenti per rimediare a questo svantaggio.

Terminologia riguardante la memoria:

- **Address Space**: quantità di dati che possono essere immagazzinati.
- **Addressability**: numero di bits immagazzinati in ogni locazione di memoria.

▫ 1 nibble	= 4 bits
▫ 1 byte	= 8 bits
▫ 1 Kilobyte (KB)	= $2^{10}$ bytes = 1024 bytes
▫ 1 Megabyte (MB)	= $2^{20}$ bytes
▫ 1 Gigabyte (GB)	= $2^{30}$ bytes
▫ 1 Terabyte (TB)	= $2^{40}$ bytes
▫ 1 Petabyte (PB)	= $2^{50}$ bytes
▫ 1 Exabyte (EB)	= $2^{60}$ bytes

## CPU

La CPU (Central Processing Unit) è composta principalmente da:

- **Control Unit (CU)**, o unità di controllo: coordina le attività all'interno della CPU, che controlla le operazioni di ALU e lo spostamento dei dati all'interno della CPU, che controlla le operazioni, indirizzamento, registri, I/O, memoria, interfaccia, elaborazione di interruzioni.  
Essenzialmente è una rete sequenziale (è dotato di memoria). Sono definite *Datapath* le unità che compongono la CPU, esclusa la CU.  $CP = CU + Datapath$ .
- **Arithmetic Logic Unit (ALU)**: è la calcolatrice della CPU, ma i calcoli sono molto semplici. Infatti, svolge operazioni aritmetiche (addizione, sottrazione, moltiplicazione e divisione) e operazioni logiche (and, or, not, xor).
- **Registri**: compongono la memoria della CPU. I principali sono:
  1. **Program Counter (PC)**: contiene la locazione (ovvero l'indirizzo) della prossima istruzione;
  2. **Instruction Register (IR)**: vi immagazzino l'istruzione in fase di elaborazione;
  3. **Memory Address Register (MAR)**: contiene l'indirizzo di memoria del dato a cui la CPU dovrà accedere;
  4. **Memory Buffer Register (MBR), o Memory Data Register (MDR)**: è un registro a due vie che contiene il dato che dev'essere scritto in memoria oppure detiene il dato letto dalla memoria.

## Bus

L'accesso alla memoria avviene tramite i bus. Tipi di Bus:

- **Data Bus**: trasferisce il dato della CPU alla memoria e viceversa. Un dato è portato al Data Bus nel MBR, da cui può essere trasferito in altri registri della CPU.
- **Control Bus**: se un dato dev'essere letto dalla memoria o scritto in memoria, la Control Unit manda un segnale attraverso il Control Bus, che specifica se il dato dev'essere letto o scritto. Il Control Bus connette la CPU al mondo esterno.
- **Address Bus**: bus di indirizzi che serve a specificare l'indirizzo di memoria in cui si vuole accedere; è unidirezionale.

## Codice operativo e Operandi

Anatomia dell'istruzione:

**Opcode (codice operativo)**: è ciò che l'istruzione fa (verbo). Può richiedere o meno uno o più operandi. Essi possono essere:

- **Source Operands (Operandi Sorgente)**: sono gli oggetti che vengono manipolati dall'istruzione;

- **Destination Operands (Operandi Destinazione)**: in cui sono collocati i risultati dell'operazione
  - 0 operands: operand(s) implicitly defined  
ex.: CLA (clear accumulator)
  - 1 operand: 1 operand explicit (the 2<sup>nd</sup> operand is implicit)  
ex.: ADD B (Acc + B → Acc)
  - 2 operands: one of the two operand is source and destination  
ex.: ADD R1,R2 (R1 + R2 → R1)
  - 3 operands: sources and destination explicit  
ex.: ADD R1,R2,R3 (R2 + R3 → R1)

I processori non hanno lo stesso formato d'istruzioni (set d'istruzioni). [Attenzione: non è detto che un programma funzioni se il processore di due calcolatori è lo stesso, perché il sistema operativo, che gestisce l'I/O, potrebbe essere diverso].

Tipi di istruzioni che mi serviranno: data movement, arithmetic, boolean, bit manipulation, I/O, control transfer, special purpose.

In generale, un'operazione potrebbe avere 0, 1, 2 o 3 operandi. In un'istruzione a 0 operandi è l'opcode ad indicare l'operando, es. CLA indica di pulire l'accumulatore.

Istruzioni con 1 Operando: si riferisce sempre all'accumulatore, che è l'ingresso e l'uscita delle operazioni. Pensa alla finestra di una calcolatrice tascabile.

Nei processori moderni l'accumulatore non c'è.

2 operandi: uno dei due operandi è sia una fonte che la destinazione (ADD R1, R2)

3 operandi: ci sono tre registri indicati esplicitamente come fonte e come destinazione (ADD R1, R2, R3 → R2+R2 IN R1).

I processi più moderni sono tutti a 3 operandi.

#### Diversi tipi di istruzioni:

- Spostamento dati tra registri, registri e memoria
- Operazioni aritmetiche
- Operazioni booleane
- Manipolazione dei bit (cambiare un bit in una word, primo bit uguale a 0)
- I/O
- Trasferimento di controllo (loop e salti)
- Istruzioni tipiche del processore che non appartengono a nessuna di queste categorie (Special Purpose)

#### Famiglie di processori

- ARM: dispositivi mobili
- ARM modificata da Apple (MIPS?): tablet
- Intel e AMD: server, portatili in versione leggere
- Processori particolari per Supercomputer

Un processore con 3 operandi produce un codice più ottimizzato e che funziona meglio.

I processori Intel utilizzano istruzioni a 2 operandi per avere compatibilità con i processori degli anni '70, ma ci sono trucchi per averne 3.

ADD R1, R2, R3 → alcuni calcolatori utilizzano questa notazione per essere più vicini ai linguaggi ad alto livello; altri vogliono prima i due registri fonte e poi la destinazione, mentre nell'istruzione citata precedentemente c'è prima l'operando di destinazione e poi i due operandi sorgente.

## Modalità di indirizzamento

Le modalità di indirizzamento specificano dove è locato un operando (se è una costante, un registro o una locazione di memoria). Viene detto **Effective Address** l'indirizzo effettivo tipo dove si trova la stringa di bit su cui operare.

Le modalità di indirizzamento sono:

- **Immediate Addressing**: modalità d'indirizzamento immediato; l'operando (il dato) è parte dell'istruzione.
- **Direct Addressing**: modalità d'indirizzamento diretto; all'interno dell'istruzione è contenuto l'indirizzo dell'operando.
- **Register Addressing**: modalità d'indirizzamento registro; nell'istruzione è contenuto il registro che contiene l'operando.
- **Indirect Addressing**: modalità d'indirizzamento indiretta; nell'istruzione c'è un registro che contiene l'indirizzo dell'indirizzo dell'operando.
- **Index Addressing**: indirizzamento indicizzato; l'indirizzo effettivo dell'operando si ottiene sommando il contenuto del registro indice con l'indirizzo contenuto nell'istruzione. L'indirizzo indice è usato come off-set, ovvero scorrimento (simile all'array).
- **Based Addressing**: indirizzamento basato; molto simile all'Indexed Addressing, ma il registro viene utilizzato come valore di partenza e l'indirizzo rappresenta uno spostamento da questa base.

## Ciclo Fetch-Decode-Operand Assembly-Execute

Un processore, incessantemente, attraversa in loop queste fasi:

- **Fetch**: prelievo della prossima istruzione dalla memoria (da dove indicato dal program counter)
- **Decode**: esamina l'istruzione e determina come eseguirla
- **Operand Assembly**: preparazione (caricamento) degli operandi
- **Execute**: esecuzione dell'istruzione e caricamento dei risultati

#### Operazione di Fetch

Per andare a prendere l'istruzione, devo copiare il PC nel MAR. Dopodiché il MAR viene trasmesso sull'Address Bus, dopodiché l'indirizzo viene

trasmesso sulla memoria.

Dopo che l'area di memoria è stata individuata, attraverso la Control Unit, la CPU manda alla memoria l'informazione che l'operazione è di lettura. La memoria risponde con il contenuto di quella locazione sul Data Bus, che viene caricato nel MBR. Per poter eseguire l'istruzione, essa viene caricata dal MBR all'Instruction Register. Dopo che è stata analizzata, essa viene poi eseguita.

Tempo fa non si faceva caso al consumo energetico e il ciclo si susseguiva anche quando il calcolatore era in stand by. Ora i processori sono in grado di spegnersi o semi spegnersi.

Visto che il ciclo è un loop, quando parte? Con la fase di **bootstrap**: essa è la *fase in cui viene fornita una tensione e parte il processore*.

All'avvio del calcolatore, la control unit si dirama fino alla locazione di memoria che contiene un indirizzo ROM (e non RAM, perché la ROM non è volatile), che contiene il BIOS loader.

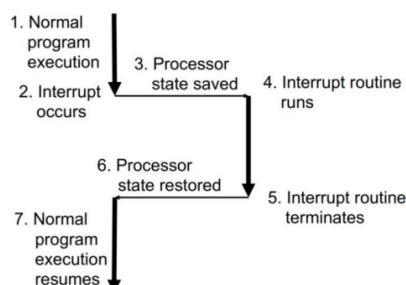
Il BIOS contiene un pezzo di codice, detto POST, che controlla se mancano delle componenti del computer o meno (e le segnala in tal caso).

Dopodiché il BIOS carica il Bootstrap Loader (solitamente dal disco Bootstrap Sector), il quale carica un loader in memoria che viene eseguito, e in seguito carica il sistema operativo.

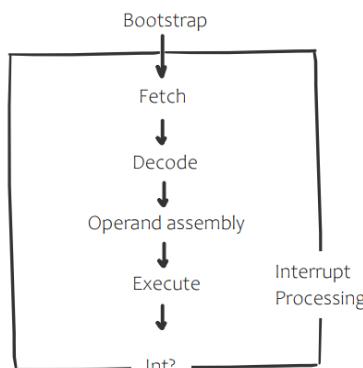
## Sistema delle interruzioni

Il sistema delle interruzioni del ciclo Fetch-Decode-Operand Assembly-Execute: se non ci fosse questo sistema delle interruzioni, la CPU svolgerebbe una sola cosa alla volta, sarebbe quindi meno flessibile e non risponderebbe ad eventi esterni. Per interruzione si intende un qualunque evento esterno dall'arrivo di un byte su una porta seriale, a un tasto premuto sulla tastiera.

Come viene gestita l'interruzione:



*Interrupt Service Routine (ISR)*; è il pezzo di codice che si occupa di gestire l'interruzione. L'ISR appartiene al kernel del sistema operativo.



Il processo d'interruzione necessita a livello hardware di:

- un circuito per salvare lo stato
- un circuito per cancellare l'interruzione
- una locazione dove salvare lo stato
- un circuito per trovare la locazione dell'ISR e caricare l'indirizzo

## Interruzioni (approfondimento)

Un processore che utilizza i cicli non si ferma mai e può eseguire solo un programma alla volta e non può rispondere a eventi esterni.

Interruzione: *evento che richiede attenzione*. Ad esempio, l'arrivo di un byte su un canale seriale.

Consente al processore di accorgersi di un eventi e di attivare un programma esterno chiamato ISR (Interrupt Service Routine).

Gestire un'interruzione:

1. Esecuzione normale del programma
2. Si verifica un'interruzione
3. Lo stato del processore viene salvato, ovvero salvo tutto quello che c'è nei suoi registri interni per riprendere in seguito le operazioni da dove mi sono interrotto
4. La routine dell'interruzione viene eseguita
5. La routine dell'interruzione viene terminata
6. Lo stato del processore viene ristabilito
7. L'esecuzione del programma riprende da dove si era fermata

Posso interrompere anche delle interruzioni per svolgere altre operazioni.

Supporto hardware alle interruzioni.

Circuiti per individuare le interruzioni.

Circuiti per salvare lo stato delle interruzioni.

Un posto per salvare lo stato del programma:

- stack (le interruzioni possono essere annidate)
- registri

Infine, mi servono dei circuiti per trovare la locazione della ISR e per caricare il suo indirizzo nel Program Counter.

Queste ISR al giorno d'oggi fanno parte del Kernel dell'OS, o meglio, l'OS si posa su una base di interruzioni.

#### **Multicore**

Il programma, per sfruttare effettivamente un processore multicore, dev'essere progettato per esso, altrimenti sfrutta unicamente un singolo core.

# Capitolo 1 - Architettura

## Computer Abstractions & Technology

- Progresso dei computer e dei processori (detta dalla Legge di Moore)
- Innovazioni come il World Wide Web, i motori di ricerca, i telefoni cellulari
- I computer diventano pervasivi

Classificazione moderna dei computer:

- *Personal computer*: scopi generici, sono in grado di girare una varietà di programmi di terze parti. Sono soggetti ad un trade-off costo/prestazioni. Sono destinati all'uso di un singolo utente
- *Computer di classe server*: servono a gestire le richieste ricevute da altri computer tramite la rete. Su di essi vengono gestiti grandi archivi di dati, girano i siti web e i servizi di streaming. Hanno gli stessi processori dei PC, sono solo più potenti ed è più importante l'affidabilità (hanno un doppio alimentatore). Essi sono mantenuti in dei contenitori rack. A volte ci sono degli edifici pieni di REC
- *Supercomputer*: computer ad altissime prestazioni che vengono utilizzati per applicazioni per il calcolo scientifico, per le previsioni metereologiche. Rappresentano una piccola parte del mercato dei computer.
- *Computer embedded*: vengono utilizzati come componenti di sistemi. Limiti di costo/prestazioni più mercati. Devono costare poco e non consumare troppa energia

Era Post PC (ma prima della pandemia):

- *Personal Mobile Devices (PMD)*: stanno sostituendo i PC. Essi sono alimentati a batteria, con connessione a internet wireless e un costo non troppo elevato. Su questi dispositivi si possono scaricare ed eseguire applicazioni come nei PC; la differenza è che non possiedono né una tastiera né un mouse e ricevono in input touch screen o attraverso voce.
- *Cloud Computing*: i server sono stati semi sostituiti dai Cloud Computing. Essi consentono di usufruire di servizi tramite server remoto e a risorse software e hardware, inviando i dati ed eseguendo programmi su un altro computer più potente.

### Touchscreen

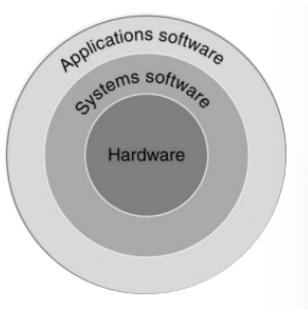
Il dispositivo di I/O più affascinante è lo schermo grafico, in particolare quello a cristalli liquidi (LCD) che ha uno schermo sottile a basso consumo energetico. Tempo fa le Tv erano a tubo catodico (non erano buoni per la salute). I pc utilizzano gli schermi LCD, mentre smartphone e tablet usano degli schermi sensibili al tatto in modo tale che l'utente può direttamente indicare quello a cui è interessato senza l'utilizzo del mouse. In passato, si è adottato il touchscreen resistivo, costituito da due pellicole trasparenti conduttrive caratterizzate da una certa resistenza elettrica. Quando si esercitava una pressione sul display, veniva a crearsi un contatto elettrico sullo strato sottostante che tracciava le coordinate. La tecnologia usata oggi è quella capacitiva. Così come il display resistivo, anche quello capacitivo è composto da più strati, ma quello più esterno è di vetro, ricoperto da ossido di metallo. Viene applicata una tensione che si propaga uniforme su tutta la superficie dello schermo per via dell'ossido di metallo; quando il dito o un materiale conduttore tocca lo schermo, avviene una variazione di capacità. Agli angoli del display, sono posti dei sensori in grado di individuare la caduta di tensione e rilevare le coordinate. Permette, quindi, il multitouch e le gesture.

## 7 idee su cui si basa l'architettura dei calcolatori

1. *Abstraction*: utile per la progettazione sia di hardware che software. I dettagli vengono nascosti a livelli più bassi per offrire un modello più semplice a livelli più alti.
2. *Make a common case fast*: miglioramento delle situazioni comuni, generalmente anche meno complesse, per migliorare le prestazioni.
3. *Parallelism*: migliorare le prestazioni con l'esecuzione di operazioni in parallelo.
4. *Pipelining*: vengono fatte operazioni, una dietro l'altra, come in una catena di montaggio (Pipeline in inglese è la tubatura).
5. *Prediction*: in alcuni casi è più veloce tirare ad indovinare piuttosto che aspettare di sapere con certezza. Necessita di un'accurata precisione nelle predizioni e un meccanismo di recupero per predizioni sbagliate non troppo costoso.
6. *Hierarchy of Memory*: tecnica utilizzata per dare al programmatore l'illusione che la memoria sia di grandi dimensioni e molto veloce. Le memorie più veloci e più costose (cache) si trovano in cima, quelle più lente ed economiche (dischi) si trovano in fondo.
7. *Dependability*: l'introduzione di elementi ridondanti per aumentare l'affidabilità, in modo che possono essere attivati in caso di guasti e aiutare a identificare il guasto. (Es. due alimentatori invece di uno).
8. C'era un'ottava idea, ovvero la *legge di Moore*, che esprimeva il ritmo di crescita del progresso nella tecnologia dei computer. Ma non è più valida perché negli ultimi anni c'è stato un rallentamento. La legge di Moore, da Gordon Moore, uno dei fondatori della Intel, stabiliva che le risorse dei circuiti integrati venivano duplicate ogni 18-24 mesi.

**Legge di Moore**: le risorse messe a disposizione dei calcolatori vengono duplicate ogni 18 mesi. Devo quindi progettare i calcolatori "indovinando" dove sarà la tecnologia a fine progetto e non dove è ora, pena il rilascio di prodotti obsoleti.

Per passare da un'applicazione complessa alle semplici istruzioni che il calcolatore può fare (di basso livello). Abbiamo bisogno di un processo che prevede diversi strati:



- **Application Software**, software applicativo, scritto in un linguaggio ad alto livello.
- **System Software**, software di Sistema, che si divide in:
  - I compilatori: che traducono il programma scritto in un linguaggio ad alto livello in istruzioni eseguibili dall'hardware.
  - Il sistema operativo: principalmente si occupa di gestire I/O, gestire l'immagazzinamento in memoria, pianifica le task ecc.
- **Hardware**.

Linguaggio Assembly: un calcolatore comprende solo due tipi di segnali, on e off, che sono rappresentati da 1 e 0. Il linguaggio macchina è una composizione di numeri in base 2, detti numeri binari, dove ogni cifra è una cifra binaria, detta bit. I calcolatori obbediscono a delle istruzioni che sono semplicemente delle stringhe di bit. È molto difficile comunicare direttamente con il linguaggio macchina, quindi abbiamo bisogno di un Assembler che è un programma che converte una versione simbolica delle istruzioni in una versione binaria comprensibile alla macchina. Il linguaggio assembler costituisce una limitazione perché richiede la scrittura di una linea per ogni istruzione che il calcolatore eseguirà. Si è avuta l'esigenza di linguaggi di programmazione ad alto livello.

## ISA (Instruction Set Architecture)

*Architettura dell'insieme delle istruzioni (ISA)*: insieme delle istruzioni che comprende tutto ciò che i programmati devono sapere per scrivere un programma in linguaggio macchina correttamente funzionante e comprende quindi le istruzioni, i dispositivi di I/O, ecc.

Tipicamente un OS incapsula i dettagli dell'I/O sull'allocazione della memoria e su altre informazioni di basso livello, evitando al programmatore di dover gestire questi dettagli.

La combinazione delle istruzioni di base e dell'interfaccia dell'OS fornita ai programmati per scrivere programmi è detta *interfaccia binaria delle applicazioni (ABI, Application Binary Interface)*. Un'architettura dell'insieme delle istruzioni permette ai progettisti di descrivere le funzionalità di un calcolatore in maniera completamente indipendente dall'hardware che lo implementa.

Un'implementazione di un'architettura è un hardware che realizza la descrizione astratta dell'architettura.

### A SAFE PLACE FOR DATA

Si stanno dirigendo verso sistemi solidi, ma gli hard disk tradizionali continuano ad essere utilizzati. Se sono richiesti tagli piccoli di memoria, si preferiscono utilizzare le SD card.

## Le reti

Ruolo importante al giorno d'oggi.

Si dividono in:

- Local Area Network (LAN): Ethernet
- Wide Area Network (WAN): Internet
- Wireless Network: Wi-Fi, Bluetooth

## Trend della tecnologia

La capacità della memoria principale (RAM Dinamica o DRAM) è aumentata esponenzialmente.

Il grafico è espresso su una scala semilogaritmica, ci permette di vedere più chiaramente numeri grandissimi.

Per sua natura, la memoria dinamica è difficile da gestire, è quindi per il produttore è difficile ridurre gli sprechi di energia.

Anni '70: tempo di accesso di 120 ms

Oggi: tempo di accesso di 40 ms (miglioramento solo di 3 volte)

Le memorie però continuano ad aumentare.

Le memorie statiche sono molto più veloci ma sono più piccole. Per tenere il passo, le memorie dinamiche utilizzano le cache.

I transistor hanno sostituito le valvole a vuoto.

Negli anni 2010 abbiamo l'integrazione a Ultra Large scale.

## Tecnologia dei semiconduttori

Essenzialmente, tutti i CI (Circuiti Integrati) sono basati sul silicio.

Questo materiale si ricava dalla sabbia. Esso è un semiconduttore, conduce la corrente a seconda del droggaggio. Vengono aggiunte zone conduttrici e zone isolanti per costruire i circuiti.

Il processo parte dai lingotti. Lo slicer li taglia in wafer, fette sottilissime, ognuna delle quali viene trattata con processi che possono arrivare a 40 step.

Finiamo per avere un wafer diviso in vari chip.

Il wafer viene poi analizzato per vedere i chip difettosi. Il wafer viene poi diviso ulteriormente in dices (chip). I dices buoni vengono poi chiusi ermeticamente in package. I package vengono testati e tutti quelli che non sono difettosi vengono spediti ai clienti.

*Yield:* proporzione dei dice funzionanti per wafer. Più è complicato il tipo di produzione e più è inevitabile che mi imbatterò in dice difettosi e lo yield si abbassa. Bisogna fare un compromesso tra prestazioni e yield.

I chip con tecnologia più elevata è 7nm. Intel continua a vendere chip meno avanzati.

Costo per dice = costo per wafer/(dices per wafer *yield*)

*Dices per wafer = wafer area/dice area*

*Yield = 1/(1+(difetti per area dice area/2))^2*

Più è grossa l'area, più aumenta il rate dei difetti. L'area del chip dipende dall'architettura e dal design del circuito.

## Misura delle prestazioni

Le prestazioni di un calcolatore possono essere determinate sia attraverso il Response time o Execution time (tempo di risposta o tempo di esecuzione), cioè il tempo totale richiesto per completare una task, sia con il throughput, cioè il numero di operazioni complete per unità di tempo (Es. 3 operazioni in un'ora).

Viene definita:

$$\text{Performance di } X \text{ (prestazione)} = \frac{1}{\text{Execution time di } X}$$

Se bisogna confrontare "X è n volte più veloce di Y":

$$n = \frac{\text{Performance } X}{\text{Performance } Y} = \frac{\text{Execution } Y}{\text{Execution } X}$$

Metodi diversi per misurare i *tempi di esecuzione*:

1. **Elapsed time:** misuro semplicemente il tempo da quando inizia la task fino a quando finisce. Tempo Wall-Clock è un sinonimo di elapsed time.

In questo tempo conto il processing, l'I/O, l'OS overhead, l'idle time (il tempo in cui il programma non è in esecuzione perché un altro programma occupa la CPU. Di solito non si nota perché è molto breve, dando così l'illusione di avere più programmi operativi in contemporanea). Se voglio misurare l'elapsed time devo quindi cercare di non considerare il più possibile questi fattori

2. **CPU Time:** misuro solo il tempo in cui il programma sta nella CPU, togliendo l'I/O e solo parzialmente l'overhead dell'OS. In pratica misura solo tutto quello che il programma fa in proprio e quello che l'OS fa per far funzionare il programma.

Un programma che lavora su archivi deve considerare l'Elapsed time perché dobbiamo considerare anche l'I/O. Tipicamente il CPU time fornisce due numeri: il tempo in cui il programma sta nella CPU e quello che l'OS impiega per il programma.

## Clock

Tutti i calcolatori utilizzano un segnale periodico, il Clock, per sincronizzare le varie operazioni dell'hardware. I relativi intervalli di tempo vengono chiamati cicli di Clock: si tratta di un'unità di misura per valutare le prestazioni di una CPU, che misura il tempo di esecuzione di un'operazione in numero di cicli di clock. Le prestazioni di un calcolatore sono determinate da 3 fattori: numero di istruzioni, durata del ciclo di clock e CPI. Il CPI, Cicli di Clock Per Istruzione, è la media dei cicli di clock per compiere le diverse istruzioni.

$$\text{CPU time} = n \text{ Cicli Clock} * \text{tempo di clock} = \frac{n \text{ Cicli Clock}}{\text{frequenza clock}}$$

Prestazioni migliori: riducendo i cicli di clock, aumentando la frequenza.

In particolare:

$$\text{CPU time} = n \text{ istruzioni nel programma} * n \text{ CPI}$$

$$\text{CPU Time} = \frac{\text{istruzioni}}{\text{programma}} * \frac{\text{ciclo clock}}{\text{istruzione}} * \frac{\text{secondi}}{\text{ciclo clock}}$$

Il numero di istruzioni dipende dal programma in questione, dal set d'istruzioni (ISA) e dal compilatore (e di quanto sia efficiente nel tradurre il linguaggio); il CPI dipende dall'hardware della CPU.

Ricorda: la frequenza da sola non è sufficiente per sapere quale processore è più veloce.

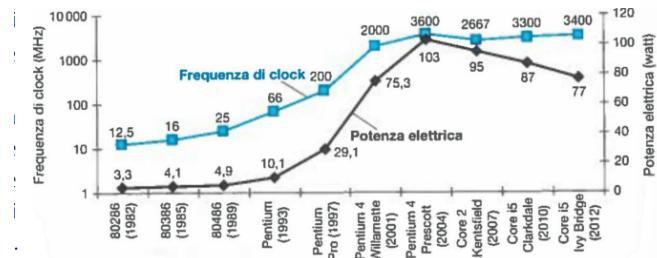
Se istruzioni di classi differenti impiegano numeri di cicli diversi, dovrò considerare la media pesata del CPI. In ogni caso devo cercare di evitare le istruzioni che durano tanto.

Per avere prestazioni migliori, il compilatore deve cercare di avere più istruzioni veloci e meno istruzioni lente.

In generale, le prestazioni dipendono:

- dall'algoritmo (influenza il numero di istruzioni, possibilmente il CPI)
  - dal linguaggio di programmazione: influenza il numero di istruzioni e il CPI (Consumer Price Index)
  - dal compilatore: influenza il numero di istruzioni e il CPI
  - dall'architettura del set di istruzioni: influenza il numero di istruzioni, il CPI e il tempo
- N.B.: non basta migliorare un solo aspetto poiché si influenzano a vicenda.

## Barriera dell'energia



Uno dei metodi utilizzati per aumentare le prestazioni è stato l'aumento della frequenza di clock. La figura mostra l'aumento della frequenza di clock e della potenza elettrica richiesta dalle 8 generazioni di microprocessori Intel prodotte negli ultimi 30 anni. Entrambe sono cresciute rapidamente per anni, ma recentemente hanno smesso di aumentare (Nel 2004, con la versione Prescott dell'Intel Pentium 4, si è raggiunta la massima potenza dissipabile (103W)). Il motivo è che è stata raggiunta la massima potenza dissipabile dai sistemi di raffreddamento montati nei microprocessori.

$$Potenza = Carico Capacitivo * Tensione^2 * Frequenza$$

Infatti, oggi giorno, la questione energetica è molto importante: la durata della batteria può essere un fattore fondamentale per un dispositivo mobile, così come la riduzione dei costi di alimentazione e raffreddamento dei grandi centri server.

Da questo punto in poi, si è cambiato strategia a favore di un approccio Multicore (più di un processore per chip), con una frequenza di clock più bassa, pipeline più semplici e più processori sullo stesso chip. Con l'avvento dei processori multicore, spesso il miglioramento non si verifica nel tempo di esecuzione, ma nel throughput.

In passato, i programmati potevano confidare nelle innovazioni per aumentare le prestazioni dei loro programmi, senza dover modificare una singola linea del loro codice. Con l'avvento del Multicore, i programmi non potranno più essere scritti in maniera sequenziale, ma dovranno sfruttare continuamente tutti i processori, eseguendo sezioni di codice in parallelo, garantendo inoltre che tutti i processori abbiano un carico bilanciato.

Si può valutare oggettivamente le prestazioni di un calcolatore attraverso un insieme di benchmark, ossia programmi campioni, sviluppati dallo SPEC (System Performance Evaluation Cooperative – Cooperativa per la Valutazione delle Prestazioni dei Sistemi), che valutano secondo più parametri, rilasciando un valore rappresentativo. L'ultima versione dello SPEC è lo SPEC CPU2006.

Legge di Amdahl:

$$Tempo_{migliorato} = \frac{Tempo_{potenzialmente\ migliorabile}}{\text{fattore di miglioramento}} + T_{non\ migliorabile}$$

Significa che, per quanto aumentiamo il fattore di miglioramento, c'è un limite al miglioramento ottenibile, siccome è sempre presente una parte non migliorabile.

Di pari passo con la frequenza di clock è aumentata anche la potenza dissipata. Quando si arrivò al Pentium 4, con cui pur avendo una frequenza di clock molto più elevata, dissipava tre volte di più la potenza del processore precedente. Questo perché la potenza dissipata dipende dal carico capacitivo, dalla tensione al quadrato e dalla frequenza. Per questo, con una frequenza pari a 1000 Hz, si poteva arrivare ad un dissipamento pari a x30.

Dopo questo punto di rottura (Power Wall), si decise di passare ai multicore con una frequenza più bassa come strada per poter continuare ad aumentare la potenza senza continuare a dissiparne sempre di più.

Perché le altre strade possibili non erano fattibili?

Supponiamo di voler costruire una nuova CPU con il carico capacitivo pari all'85% della vecchia CPU e anche il voltaggio e la frequenza venivano ridotti del 15%.

$$\frac{P_{(new)}}{P_{(old)}} = \frac{C_{(old)} * 0.85 * (V_{(old)} * 0.85)^2 * F_{(old)} * 0.85}{C_{(old)} * V_{(old)}^2 * F_{(old)}} = 0.85^4 = 0.52$$

Consuma quasi la metà di quello vecchio, ma non possiamo fare di meglio.

Dalla metà degli anni 2000, l'incremento di potenza del singolo processore è diminuito dal 52% ogni anno al 22% ogni anno, non solo per la potenza ma anche per problemi relativi alla memoria (dato che di base è già molto più lenta del processore).

## Multiprocessori

L'idea è che per mantenere l'aumento di potenza dei processori, al posto di incrementare la potenza, aumenta il numero di processori in un solo chip. Negli attuali chip multicore ci sono anche le interconnessioni tra di essi e le memorie cache. La differenza fondamentale con i processori dei tempi precedenti è che l'utente, per poter sfruttare le prestazioni più elevate, deve scrivere i programmi con una programmazione parallela esplicita (divide i compiti tra i vari core), che è una cosa completamente diversa dal metodo tradizionale, ovvero utilizzare il parallelismo a livello di istruzioni. Il parallelismo esplicito pone il compito di distribuire i compiti con i processori, bilanciare il carico del lavoro (nessun core deve stare con le mani in mano) e ottimizzare la comunicazione e la sincronizzazione. Se il numero di core aumenta (8-16 core) devo quindi scrivere dei programmi con il parallelismo esplicito.

I programmi di tipo scientifico, come MATLAB, sono in grado di sfruttare il parallelismo, così come Photoshop e i programmi per la grafica 3D. I thread di Java permettono di sfruttare i core multipli.

## SPEC Benchmark

Benchmark: programma utilizzato per misurare le prestazioni.

Possibilmente questi programmi devono essere rappresentativi del carico di lavoro effettivo che vogliamo andare a fare.

Posso utilizzare i Benchmark per confrontare computer e telefoni differenti per vedere quali hanno le caratteristiche migliori per quello che voglio fare.

SPEC: corporazione che produce benchmark per misurare le prestazioni della CPU, dell'I/O, ricerca web, ecc. Non sono gratis perché sono un

riferimento serio per confrontare modelli differenti. Utilizzano programmi già conosciuti (come il gioco Go, GNU C Compiler), così da avere un numero preciso che indica le prestazioni dell'attività che mi interessa.

Posso avere anche un numero unico facendo la media geometrica di tutti i risultati (SPECRatio).

Esistono anche benchmark per la potenza dissipata.

Xeon: processori Intel che hanno prestazioni più elevate di quelli commerciali, indicati per computer di classe server.

Carico del processore: faccio lavorare il processore con diverse percentuali di tempo, con l'obiettivo di capire quanto consuma al variare del carico.

Possiamo notare che, anche quando il processore non è utilizzato, consuma comunque 80 Watt. Non c'è quindi un abbassamento proporzionale al carico. Bisogna prendere delle precauzioni.

Numeri unici per le prestazioni: rapporto tra la somatoria delle prestazioni e la somatoria della potenza dissipata.

## Legge di Jane Amdahl

*Migliorare un aspetto di un computer non vuol dire avere un miglioramento proporzionale alla performance totale. Per avere miglioramenti effettivi non mi conviene migliorare la parte meno dispendiosa.*

$$T(improved) = \frac{T(affected)}{ImprovedFactor} + T(unaffected)$$

Notiamo dalla formula che *la parte che migliora viene migliorata di un fattore n, mentre la parte non migliorata non cambia.*

Quindi, il possibile incremento totale delle prestazioni indotto dal miglioramento di una caratteristica è limitato dal tempo di utilizzo di quella caratteristica.

Devo migliorare il caso comune.

E' un criterio che si applica sia all'hardware che al software.

## Potenza Dissipata

Rivedendo i benchmark sulla potenza dissipata, i google data center operano principalmente al 10%-50% di carico.

Solo nell'1% del tempo lavorano al 100% del carico.

Sarebbe una buona cosa per il futuro progettare processori la cui potenza dissipata è proporzionale al carico.

MIPS (Millions of Instruction Per Second) come misura per le prestazioni?

Avrebbe senso usarla come metrica se ogni istruzione avesse lo stesso carico. Abbiamo visto che il CPI varia tra programmi differenti.

Il MIPS non fa conto di ISA (Instruction Set Architecture) differenti tra computer e la differenza in complessità tra istruzioni differenti.

## Note finali

- Il rapporto costo/prestazioni sta costantemente migliorando
- Lo sviluppo dei programmi viene realizzato con livelli di astrazione (vale sia per l'hardware che per il software)
- L'ISA si pone come livello intermedio tra hardware e software
- Migliore misura di prestazioni: il tempo che mi serve per fare l'operazione che mi interessa (tempo di esecuzione, che si può migliorare tramite wall-clock o CPU Time)
- La potenza dissipata è un fattore limitante nel miglioramento delle prestazioni; è stato quindi necessario passare ad architetture multicore

## Cos'è il power wall?

Il "Power wall", che in italiano significa "barriera di potenza", indica appunto che l'assorbimento di potenza da parte del processore è una barriera invalicabile. Infatti per fabbricare i circuiti integrati si utilizzano i CMOS i quali assorbono energia elettrica soprattutto durante la fase di commutazione (passaggio da uno stato logico ad un altro: 0 → 1 → 0). Questa energia viene detta "Energia Dinamica" ed è proporzionale alla tensione di alimentazione. La potenza richiesta da ogni CMOS è proporzionale all'energia dinamica. Prima dell'avvento dei multiprocessori, i produttori di processori puntavano ad aumentare la frequenza di clock (la Frequenza di Clock indica il numero di operazioni elementari che la CPU è in grado di eseguire nell'unità di tempo di un secondo).

E questo era possibile grazie al fatto che il numero di transistor che si potevano ottenere su un determinato chip diventava via via maggiore. Il problema è che maggiore era il numero di transistor, maggiore era la potenza assorbita da parte di questi ultimi. Dato che la potenza dissipata dipende anche dalla (*Tensione di Alimentazione*)<sup>2</sup>, i produttori tentarono di diminuirla, ma sapevano 2 che questa soluzione rappresentava un palliativo. Quindi i processori arrivarono a dissipare una potenza che andava oltre la capacità dei sistemi di raffreddamento. Inoltre, si era arrivati ad un punto in cui la ricerca in nuove tecnologie di raffreddamento dei chip era diventata troppo costosa. Queste problematiche portarono i produttori a concentrarsi sul throughput piuttosto che sul tempo di esecuzione di un singolo task: questo fu possibile proprio grazie a dei microprocessori contenenti più processori su un singolo chip, appunto chiamati Multiprocessori.

## Cos'è un benchmark?

I benchmark sono programmi campione, scelti per valutare le prestazioni di un calcolatore. I benchmark sono pensati per fornire un carico di lavoro (work load) che possa essere significativo al fine di stimare le prestazioni sui carichi di lavoro tipici.

Lo SPEC (Cooperativa per la Valutazione delle Prestazioni dei Sistemi) rappresenta uno sforzo congiunto, sovvenzionato e supportato da un certo numero di industrie produttrici di calcolatori, per creare un insieme standard di benchmark per i moderni calcolatori. Per semplificare la valutazione di un calcolatore, il consorzio SPEC ha deciso di riportare anche un singolo valore che riassumesse i risultati ottenuti nel benchmark. Per ottenere tale valore, il tempo di esecuzione viene dapprima normalizzato, dividendolo per il tempo di esecuzione misurato su un calcolatore di riferimento; questa operazione fornisce una misura che ha il vantaggio di assumere un valore tanto maggiore quanto più veloce è l'esecuzione. Il primo benchmark SPEC sull'assorbimento di potenza fu proposto nel 2005 per valutare le applicazioni java nel mondo degli affari.

## Che cos'è la legge di Amdahl?

Il tempo di esecuzione del programma dopo il miglioramento è dato dalla legge di Amdahl: una regola che afferma che il miglioramento delle prestazioni reso possibile da una data modifica è limitato dalla quantità di tempo in cui quella modifica è effettivamente sfruttata. Si può utilizzare la legge di Amdahl per stimare il miglioramento delle prestazioni quando si conosce il tempo in cui viene utilizzata una certa funzionalità e il suo potenziale incremento di velocità. La legge di Amdahl è uno strumento facile per valutare potenziali miglioramenti. Una delle linee guida nella progettazione dell'hardware è descritta da un corollario della legge di Amdahl: rendere veloce la situazione più comune. Essa invita a tenere conto del fatto che in molti casi la frequenza con cui accade un evento può essere molto più elevata di un'altra.

# Capitolo 2 - Architettura

## Le istruzioni: il linguaggio del Computer

### Instruction Set (ISA)

Il set d'istruzioni è l'insieme di istruzioni macchina, visibili a basso livello al programmatore, dell'architettura di un calcolatore. Computer diversi hanno set d'istruzioni diversi, ma con molti aspetti in comune. In passato i computer avevano set d'istruzioni molto semplici. Si è tentato, poi, di usare set d'istruzioni più complessi (con meno istruzioni, ma che impiegassero molto tempo nell'esecuzione), per poi ritornare a set semplici con i calcolatori odierni.

### RISC-V

L'insieme d'istruzioni di seguito è il RISC-V, che è stato inizialmente sviluppato all'Università di Berkeley (2010). Basato sul principio Reduced Instruction Set Computer (RISC), ovvero un'idea di progettazione dell'architettura dei calcolatori semplice e lineare. A differenza di molti altri set d'istruzioni, il RISC-V è pubblicato sotto licenza open source, pertanto non richiede l'acquisto di una licenza per essere utilizzato. Ogni istruzione che coinvolge tre registri è composta dal codice operativo e tre operandi: il primo è un destination operand, gli altri due sono source operands: add a,b,c // la somma di b e c è posta in a. Il fatto di richiedere che tutte le istruzioni abbiano esattamente tre operandi è conforme alla filosofia di mantenere l'hardware "regolare, quindi semplice" e di conseguenza con prestazioni a basso costo. Per svolgere una somma tra quattro variabili (a+b+c+d), con Assembly, con set d'istruzioni RISC-V: Occorrono quindi tre istruzioni per sommare quattro variabili. A differenza di altri linguaggi di programmazione, nel linguaggio Assembly ciascuna linea può contenere al massimo un'istruzione. Un'altra differenza rispetto al cinema e che i commenti terminano sempre alla fine della linea.

### Registri

Nel linguaggio Assembler RISC-V, le istruzioni aritmetiche richiedono che gli operandi siano memorizzati nei registri. Una delle differenze più importanti fra le variabili utilizzate nei linguaggi di programmazione e i registri è il numero limitato di questi ultimi: tipicamente 32 nei calcolatori della classe dei RISC-V (di 32 bit o 64 bit a seconda del tipo d'implementazione). Questo perché un altro principio importante nel RISC-V è "minori sono le dimensioni, maggiore è la velocità": un numero di registri molto elevato potrebbe aumentare la durata del ciclo di clock semplicemente perché i segnali elettrici impiegherebbero un tempo maggiore a compiere il percorso assegnato, dovendo coprire una distanza maggiore. Il progettista deve, quindi, trovare un compromesso fra la richiesta da parte dei programmi di avere più registri a disposizione e il suo desiderio di mantenere veloce il clock. La convenzione RISC-V prevede l'utilizzo di una x seguita dal numero del registro tra 0 e 31, oppure si possono definire tramite una notazione simbolica [più comoda].

x0, <b>zero</b>	Valore costante 0. Registro di sola lettura. Composto da una serie di zeri. Ha utilità nelle operazioni aritmetiche.
x1, <b>ra</b>	Return address. Il registro in cui viene salvato il punto dove ritornare prima di invocare una funzione.
x2, <b>sp</b>	Stack pointer. Che punta all'area stack, ovvero quell'area in cui vengono salvati gli argomenti delle funzioni e variabili locali.
x3, <b>gp</b>	Global pointer. Che punta all'area globale, ovvero definita all'esterno delle funzioni (quindi anche da main).
x4, <b>tp</b>	Thread pointer. ....
x5...x7 e x28...x31, <b>t0..t6</b>	Temporanei.
x8 - x9 e x18...x27, <b>s0..s11</b>	Registri di salvataggio non temporaneo.
x10-x11, <b>a0-a1</b>	Argomenti di funzioni/risultati.
x12...x17, <b>a2-a7</b>	Argomenti di funzioni.

### Load

L'Assembler RISC-V ha delle istruzioni che trasferiscono dati fra la memoria e i registri. L'istruzione di trasferimento che sposta un dato dalla memoria a un registro viene chiamata tradizionalmente load, load word se 32 bit (lw), load doubleword se 64bit (ld). Il formato di load prevede: Id (o lw se a 32 bit) + registro in cui deve essere trasferito il dato + l'offset e il registro contenente l'indirizzo base del dato da trasferire: Id s1,8(s6). L'indirizzo del dato in memoria viene ottenuto dalla somma della costante e del contenuto del secondo registro.

### Memory and Store

In generale, l'accesso ai registri è più veloce dell'accesso alla memoria, quindi i compilatori devono utilizzare i registri per le variabili il più possibile. Di conseguenza, il compilatore cerca di mantenere nei registri le variabili utilizzate più di frequente e mette le altre in memoria: si tratta dello register spilling (versamento dei registri). Inoltre, il compilatore alloca strutture dati, vettori e allocazioni dinamiche in locazioni di memoria (che ha una capacità nettamente maggiore di quella limitata dei registri). L'istruzione complementare a quella di load è chiamata tradizionalmente store e trasferisce un dato da un registro alla memoria. Il formato di store prevede: sd (o sw se a 32 bit) + registro in cui contenente il dato da trasferire in memoria + l'offset e il registro contenente l'indirizzo base dove trasferire il dato: sd s1,8(s6).

### Indirizzamento al singolo byte

La memoria è indirizzata al byte, ovvero ciascun indirizzo identifica un byte a 8 bit. Di conseguenza, l'indirizzo di una parola doppia corrisponde all'indirizzo di uno degli 8 byte che compongono e l'indirizzo di due parole doppie consecutive differisce di 8 unità. L'indirizzamento al byte influenza anche il calcolo degli indici dei vettori. Prendendo l'esercizio precedente (Id s1,8(s6)), per ottenere l'indirizzo in byte corretto, l'offset da sommare

all'indirizzo di base (x22) deve essere uguale a 8x8 bit, ovvero 64: diventa quindi Id s1,64(s6). I calcolatori con indirizzamento relativo al singolo byte si dividono tra quelli che utilizzano il bite più significativo (quello più a sinistra), detti Big Endian, e quelli che utilizzano quello meno significativo (quello più a destra), detti Little Endian, tra cui si colloca RISC-V.

## Operandi immediati o costanti

Molto spesso i programmi utilizzano all'interno di un'operazione una costante (per esempio per incrementare l'indice di un contatore): Dovremmo caricare una costante tramite load dalla memoria per poterla utilizzare. Un'alternativa è costituita da una versione delle istruzioni aritmetiche nella quale uno degli operandi è una costante: l'istruzione di somma in cui lo operando è una costante è chiamata add immediate, o addi. Es. addi s1, s1, 4.

## Rappresentazione interi positivi e negativi

I numeri vengono rappresentati nell'hardware nel calcolatore come una serie di segnali elettrici di livello alto o basso, quindi la rappresentazione naturale è quella binaria. Per indicare un numero binario senza segno (unsigned): Es.  $1011_{\text{due}} = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 8 + 0 + 2 + 1 = 11_{\text{decimal}}$ .

Per esprimere un binario intero signed si potrebbero adottare due rappresentazioni diverse: modulo e segno e il Complemento a due. La seconda è reputata la più conveniente nel nostro caso: prevede che una sequenza di 0 iniziale (il bit più significativo) indichi un valore positivo e una sequenza di 1 un numero negativo. Dando uno sguardo a tutti i valori interi esprimibili con il Complemento a due, vediamo che la successione naturale non è più la stessa (...-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5...), ma diventa: I numeri positivi vanno da  $0_{\text{dec}}$  a  $9\ 223\ 372\ 036\ 854\ 775\ 807_{\text{dec}}$  che corrisponde a  $(2^{64} - 1)$ , meno 1 perché escludiamo lo 0); i numeri negativi vanno da  $-9\ 223\ 372\ 036\ 854\ 775\ 808_{\text{dec}}$  ( $-2^{64}$ ) a  $-1_{\text{dec}}$ . Inoltre,  $-9\ 223\ 372\ 036\ 854\ 775\ 808_{\text{dec}}$  è il numero negativo di valore assoluto maggiore (most negative number) ed è l'unico a non avere un corrispettivo positivo.

### Vantaggi del complemento a due

1. L'hardware deve soltanto controllare il bit (detto bit di segno) più significativo per vedere se è un numero positivo o negativo
2. C'è un modo veloce per cambiare segno ad un numero binario: sapendo che

$$x + \bar{x} = -1 \rightarrow \bar{x} + 1 = -x$$

basta prendere il numero x, cambiare gli 0 con 1 (e viceversa) e sommarlo a 1.

3. L'estensione del segno: operazione utile se bisogna rappresentare un numero usando più bit, preservandone il valore. Consiste nel replicare il bit più significativo alla sinistra in modo da riempire i bit in più. Nel caso in cui il numero è unsigned, viene esteso con degli 0.

## Operazioni logiche

- **Shift:** consiste nello spostare tutti i bits di una parola a destra o a sinistra e riempirli con 0 bits. Queste istruzioni di shift utilizzano il formato I.
  - Shift a sinistra: L'istruzione è slli (shift left logic immediate). L'operazione fornisce un'ulteriore funzionalità; infatti, lo scorrimento di un numero a sinistra di  $i$  cifre produce lo stesso risultato di una moltiplicazione per  $2^i$ . Es. slli s1, a0, a1.
  - Shift a destra. L'istruzione è srli (shift right logic immediate). Dualmente, lo scorrimento di un numero a destra di  $i$  cifre produce lo stesso risultato di una divisione per  $2^i$  (solo per gli unsigned).
- **AND:** è un'operazione logica bit a bit che scrive un 1 nel risultato solamente se entrambi gli operandi hanno un 1 in quella stessa posizione, altrimenti scrive 0. Permette di isolare i campi di una parola e può essere utilizzata per forzare a 0 i bit di una parola fornendo in input all'AND una parola che contiene 0 in quelle posizioni, dato che essa maschera alcuni bit. Es. and s1, a0, a1.
- **OR:** è un'operazione bit a bit che inserisce un 1 nel risultato laddove almeno uno dei due operandi sia uguale a 1. Es. or s1, a0, a2.
- **XOR:** L'ultima operazione logica è la negazione: l'operazione di NOT prende un operando e pone un 1 nel risultato laddove nell'operando era presente uno 0 e viceversa. Per mantenere il formato dell'operazione a tre operandi, i progettisti RISC-V hanno deciso di includere l'istruzione XOR (OR esclusivo) invece del semplice NOT. Dato che XOR produce uno 0 quando due bit sono uguali e 1 altrimenti, si può ottenere l'operatore NOT mediante XOR di un numero con 111...111. Es. xor s1, a0, a2.

## Istruzioni di salto condizionato

Corrispondono all'istruzione if del C.

- **beq:** ovvero salta all'istruzione contrassegnata se i due operatori sono uguali, altrimenti continua sequenzialmente. In C: if(rs1 == rs2) {s1=s2+s3}; In Assembly: beq rs1,rs2,L1 con L1:add s1,s2,s3.
- **bne:** ovvero salta all'istruzione contrassegnata (labeled) se i due operatori non sono uguali, altrimenti continua sequenzialmente. In C: if(rs1!=rs2){s1=s2+s3}; In Assembly: bne rs1,rs2,L1 con L1:add s1,s2, s3.
- **blt:** ovvero salta all'istruzione se il 1° operatore è minore del 2°. In C: if(r1<r2)
- **bltu:** ovvero salta all'istruzione se il primo operatore è minore del 2°, trattando il contenuto dei due registri come numeri senza segno.
- **bge:** salta all'istruzione se il 1° operatore è maggiore o uguale al 2°. In C: if(r1>=r2)
- **bgeu:** salta all'istruzione se il 1° operatore è maggiore o uguale al 2°, trattando il contenuto dei due registri come numeri senza segno.

**Tabelle degli indirizzi di salto:** la maggior parte dei linguaggi di programmazione possiede un costrutto switch che consente al programmatore di far scegliere tra più alternative in base al valore costante che assume l'unica variabile. In modo più efficace, è stata creata una tabella contenente gli indirizzi dell'inizio dei frammenti di codice. La tabella è quindi un vettore di parole che contiene gli indirizzi corrispondenti alle etichette fornite codice. Il programma caricherà la parola corretta dalla tabella di salto in un registro e salterà l'indirizzo caricato nel registro.

## While

Il seguente codice rappresenta un tipico ciclo in C: `while(salva[i] == k) i+=1`. Supponiamo che i corrisponda a s6 e k corrisponda a s8 e che l'indirizzo base del vettore salva sia contenuto in s9. Qual è il codice Assembler RISC-V corrispondente a questo frammento di codice C? Il primo passo consiste nel caricare `salva[i]` in un registro temporaneo, però è necessario prima determinare il suo indirizzo in memoria. E prima di sommare i all'indirizzo di base del vettore salva per comporre l'indirizzo di `salva[i]`, dobbiamo moltiplicare l'indice i per 8 per ottenere l'indirizzamento al byte. Essendo una moltiplicazione per un valore piccolo (per evitare di dover fare il load di una costante per poi moltiplicarla), possiamo utilizzare l'operazione di shift logico a sinistra per moltiplicare per  $2^3$ , cioè 8. Inoltre, è necessario aggiungere l'etichetta Loop alla prima istruzione, in modo che sia possibile ritornare indietro al termine del ciclo.

## Istruzioni di salto condizionato

- **`jal`**: ovvero salta – jump - all'istruzione contrassegnata e salva - link - l'indirizzo della prossima istruzione, ovvero l'indirizzo di ritorno, nel registro. Utilizzata per:
  - la chiamata a procedura: l'istruzione salta all'indirizzo della procedura e contemporaneamente salva l'indirizzo di ritorno nel registro ra: `jal,IndirizzoProcedura`.
  - Salto incondizionato: l'istruzione salta senza eseguire un link: `jal zero,IndirizzoProcedura`.
- **`jalr`**: utilizzata per il ritorno dalla procedura. Restituisce il controllo al programma chiamante e salta all'indirizzo (0+ra) [è un salto indiretto]: `jalr zero, 0(ra)`. Usati anche per i salti alle tabelle per i case dello Switch.

## Rappresentazione delle istruzioni del calcolatore

Le istruzioni codificate in binario sono dette codice macchina. In accordo con il principio di progetto per cui la semplicità favorisce la regolarità, tutte le istruzioni RISC-V sono lunghe 32 bit. Esistono vari tipi d'istruzioni (come si può notare la somma totale dei bit utilizzati è sempre 32):

- **RISC-V formato di istruzioni R**: dove:
  - Opcode: codice operativo;
  - funct7 e funct3: codici operativi aggiuntivi;
  - r1 ed r2: registri sorgente;
  - rd: registro destinazione

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
- **RISC-V, formato di istruzioni I (istantaneo)**: usato per operazioni con costanti, tipo addi o per operazioni di trasferimento dati in memoria, tipo load:
  - rs1: registro con indirizzo sorgente o indirizzo di base;
  - immediate: operando costante o offset da sommare all'indirizzo di base.

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
- **RISC-V, formato di istruzioni S**: usato per store:
  - rs1: registro indirizzo di base;
  - rs2: registro indirizzo sorgente;
  - immediate: offset da sommare all'indirizzo base (diviso in due caselle in modo tale che l'ordine di rs1 e rs2 si mantenga costante tra i formati d'istruzioni).

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
- **RISC-V, formato SB**: usato per le istruzioni di branch:
  - rs1: registro indirizzo di base;
  - rs2: registro indirizzo sorgente;
  - immediate: offset da sommare all'indirizzo base (diviso in due caselle in modo tale che l'ordine di rs1 e rs2 si mantenga costante tra i formati d'istruzioni).

imm[10:5]	rs2	rs1	funct3	imm[4:1]	opcode
nmf12l				immf11l	
- **RISC-V, formato UJ**: usato per le istruzioni di `jal`:
  - opcode;
  - rd, registro destinazione
  - indirizzo target, a cui sono riservati 20 bits. Per salti più grandi, si usa l'istruzione lui

imm[10:5]	rs2	rs1	funct3	imm[4:1]	opcode
nmf12l				immf11l	
- **RISC-V, formato U**, usato dall'istruzione lui.

## Procedure Calling

Per l'esecuzione di una procedura, un programma deve:

1. mettere i parametri nei registri da a0...a7;
2. trasferire il controllo alla procedura;
3. acquisire le risorse necessarie per l'esecuzione della procedura;
4. eseguire il compito richiesto;
5. mettere il risultato in un luogo accessibile al programma chiamante;

6. restituire il controllo al punto d'origine (indirizzo in ra).

Il software RISC-V utilizza i suoi 32 registri secondo queste convenzioni:

- a0...a7 (ovvero x10-x17): otto registri argomento per il passaggio dei parametri o la restituzione dei valori calcolati;
- ra (x1): un registro contenente l'indirizzo di ritorno per ritornare al punto d'origine.

Il linguaggio Assemby RISC-V comprende soluzione apposita per le procedure:

- Chiamata alla procedura: l'istruzione jal.
- Ritorno dalla procedura: l'istruzione jalr.

## Salvataggio in memoria

Quando un programma va in esecuzione, diventa un processo [differenza tra programma e processo è che il programma è un elemento statico, mentre il processo si crea dal momento in cui si coinvolge la memoria RAM e il sistema operativo inizia a dare tempo di CPU].

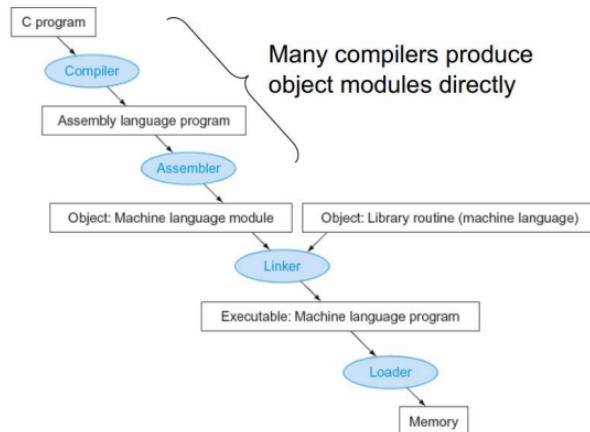
Nel momento in cui viene lanciato, viene predisposta un'apposita zona di memoria, detta spazio d'indirizzamento (address space), organizzata in questa maniera:

- una zona riservata al sistema operativo;
- una zona text (segmento di testo) in cui risiede il codice macchina RISC-V;
- una zona segmento dei dati statici che contiene variabili globali, costanti, variabili statiche, array e stringhe di lunghezza statica;
- una zona in cui c'è sia lo stack che un segmento dati chiamato "heap" (cumulo), che crescono uno verso l'altro, permettendo un uso efficiente della memoria nonostante la dimensione dei due segmenti sia altalenante. Nell'heap si posizionano i dati dinamici ed è usato per posizionare strutture dati come le liste concatenate e allocazione dinamica. Lo stack, oltre per i salvataggi visti in precedenza, è usato anche per memorizzare le variabili locali della procedura che non trovano spazio in registri (vettori o strutture locali). Il segmento dello stack che contiene i registri salvati da una procedura e le variabili locali prende nome di frame della procedura o record d'attivazione. Esiste poi un frame pointer, un puntatore a frame (fp) che punta alla prima parola del frame di una procedura.

## Character Data

Oggi la maggior parte dei calcolatori utilizza 8 bit (1 byte) per la rappresentazione dei caratteri e la codifica ASCII (American Standard Code for Information Interchange) è la più utilizzata. Sono state riservate una serie di istruzioni che consentono di estrarre un byte da una parola e che quindi sono utili per la manipolazione dei caratteri.

## Tradurre e avviare un programma dal C



Quali sono i 4 passaggi fondamentali per tradurre un programma in C, contenuto in un file in memoria, in un programma pronto per essere eseguito su un calcolatore?

- **Compilatore:** trasforma un programma in C in un programma in linguaggio Assembler;
- **Assemblatore:** L'assemblatore può trattare delle pseudo-istruzioni come se fossero delle vere e proprie istruzioni. Traduce il programma in istruzioni macchina (binario): genera un file oggetto, cioè una sequenza d'istruzioni in linguaggio macchina di dati e di informazioni necessarie a collocare le istruzioni in memoria nella posizione opportuna. Tuttavia, non vi è ancora un codice eseguibile.
- **Linker:** prende tutte le procedure in codice macchina, che sono state assemblate indipendentemente tra di loro e le "cuce", producendo un file eseguibile. In particolare: inserisce in memoria in modo simbolico il codice e i moduli, inserisce le librerie (caricamento statico), determina gli indirizzi dei dati e delle etichette, risolve tutti i riferimenti interni ed esterni.
- **Loader:** una volta che il file eseguibile è stato trasferito sul disco, il sistema operativo può leggerlo e trasferirlo in memoria. Nel mondo UNIX, il loader: 1) determina la lunghezza del segmento di testo e del segmento di dati; 2) crea uno spazio di indirizzamento sufficiente a contenere testo e dati; 3) copia le istruzioni e i dati dal file eseguibile in memoria; 4) copia nello stack eventuali parametri passati al programma principale; 5) inizializza i registri del calcolatore impostando uno stack pointer; 6) salta una procedura di start-up e chiama la procedura principale del programma.

## Caricamento dinamico delle librerie

Fino adesso abbiamo descritto l'approccio detto statico, il meno usato, per il caricamento delle librerie, che consiste nel mapparle prima che il programma sia eseguito. Nonostante sia il più veloce, riscontra svantaggi:

- Le funzioni libreria diventano parte del codice eseguibile. Se viene rilasciata una nuova versione corretta della libreria, non è modificabile.
- Vengono caricate tutte le funzioni di libreria utilizzate dal programma e anche quelle non utilizzate.

Questi svantaggi hanno portato allo sviluppo delle librerie a caricamento dinamico (DLL, Dynamically Linked Libraries) e il caricamento viene definito lazy linking (caricamento pigro), per cui le funzioni di libreria non vengono caricate o collegate finché non si inizia l'esecuzione del programma. Qualcosa andrà pur messo al posto della chiamata a funzione per risolvere il codice. Si inserisce una procedura fasulla per ogni funzione diversa, che consiste in un'istruzione di salto. Nella prima chiamata alla funzione di libreria, viene quindi inserita un'istruzione di salto verso un pezzo di codice, detto stub, che serve a identificare il punto in cui sta la libreria dinamica. Lo stub chiama una parte del sistema operativo, detta linker-loader dinamico, che genera una tabella d'indirezione che contiene tutti gli indirizzi delle funzioni chiamate. Successivamente alla prima chiamata, lo stub non viene più utilizzato ma c'è un salto indiretto all'indirizzo che è in tabella.

## Tradurre e avviare un programma da Java

Un programma Java viene inizialmente compilato, diventando Bytecode Java. Successivamente viene interpretato dalla Java virtual Machine. Il vantaggio di utilizzare la JVM come interprete è quello di rendere il programma indipendente dal tipo di calcolatore. A differenza del C, non c'è bisogno di una fase di assemblaggio per la traduzione del codice in linguaggio macchina, siccome questa viene effettuata direttamente dal compilatore o dalla stessa JVM. Viene utilizzato un'ulteriore compilatore detto Just In Time, che traccia il profilo dell'applicazione durante la sua esecuzione in modo da sapere quali siano le istruzioni che consumano più tempo CPU, compilandole singolarmente per velocizzare i tempi.

## Architettura MIPS

L'insieme di istruzioni più vicino a RISC-V è quello del MIPS. I due condividono la stessa filosofia di progetto, anche se il MIPS ha 25 anni in più del RISC-V. L'ISA dei MIPS ha una versione a 32 bit e una a 64 bit, denominate MIPS-32 e MIPS-64. Gli aspetti comuni tra RISC-V e MIPS sono:

- Tutte le istruzioni sono ampie 32 bit in entrambe le architetture;
- Entrambe le architetture hanno 31 registri a uso generale e uno imposto fisso a zero;
- L'unico modo per accedere alla memoria attraverso istruzioni di load e store;
- Non ci sono istruzioni che possono trasferire a o dalla memoria più di un registro, a differenza di altre architetture;
- Hanno entrambe istruzioni di salto;
- Le modalità di indirizzamento di entrambe le architetture possono trasferire dati di tutte le dimensioni.

Una delle differenze principali è sui salti condizionati basati su condizioni diverse dall'uguaglianza o disuguaglianza. I MIPS si devono basare su un'istruzione di comparazione che imposta il contenuto di un registro a 0 oppure a 1 a seconda che il risultato confronto sia vero o falso.

## Architettura x86

Lo scopo dei progettisti era quello di ridurre il numero di istruzioni eseguite da un programma, ma il rischio è che tale riduzione vada a scapito della semplicità aumentando il tempo di esecuzione dei programmi, perché le istruzioni diventano più complesse. L'architettura x86 voi si è evoluta nel corso di quasi quarant'anni ad opera di vari gruppi di progettisti. In particolare, ad offrire miglioramenti nel tempo a tale architettura è l'azienda Intel, e successivamente anche AMD.

- La prima importante differenza è che nel x86 le istruzioni aritmetico-logiche hanno sempre un operando che funge sia da sorgente che da destinazione, a differenza del RISC-V e del MIPS che hanno registri distinti per operandi sorgente e destinazione, rendendo la progettazione più flessibile.
- La seconda importante differenza consiste nel fatto che uno dei due operandi può essere contenuto in memoria; perciò, praticamente tutte le istruzioni possono prendere uno dei due operandi in memoria. I modi d'indirizzamento consentono d'inserire nella stessa istruzione indirizzi di due dimensioni diverse: il cosiddetto spiazzamento può essere di 8 o di 32 bit.
- Nel x86 i salti condizionati si basano sui bit del risultato dei test, detti condition codes o flags. Questi bit vengono posti 1 come conseguenza di un'operazione: la maggior parte di essi viene utilizzata per confrontare un certo risultato con 0. I salti condizionati, quindi, effettuano un controllo su un determinato flag.

### Principio di Design n°1: La semplicità favorisce la regolarità

### Principio di Design n°2: Diminuendo le dimensioni aumento la velocità

### Principio di Design n°3: Un buon design ha bisogno di buoni compromessi

## Fallacie

- Non è vero che avere istruzioni più potenti migliora le prestazioni, perché esse sono difficili da implementare efficientemente e rallentano quelle più semplici. E' meglio lasciare tutto al compilatore, che è bravo ad ottenere un codice veloce da istruzioni semplici
- Se uso l'assembly ottengo prestazioni migliori? Solo se sono tanto bravo, dato che servono più istruzioni aumenta la possibilità di errori e riduce la produttività. Al giorno d'oggi l'assembly serve solo a gestire i registri
- Retrocompatibilità: non vuol dire che il set di istruzioni non viene esteso. Bisogna sempre aggiungerne di nuove per avere nuove funzionalità

## Insidie

- Gli interi non sono ad indirizzi successivi: bisogna incrementare di 4, non di 1
- In C non bisogna mantenere un puntatore ad una variabile automatica dopo che si ritorna da una procedura. Il puntatore diventa invalido dopo che è stato fatto il pop sullo stack

## Commenti finali

Principi di progettazione:

1. *La semplicità favorisce la regolarità*
2. *Più piccolo = più veloce*
3. *Un buon progetto necessita di buoni compromessi*
4. *Bisogna rendere il caso comune il più veloce possibile*

Inoltre, in un processore ci sono vari livelli di software e di hardware (compilatore, assembler, ecc.). Il RISC-V utilizza il tipico ISA del RISC.

## Quali sono i concetti fondamentali di un processore RISC-V?

Il processore RISC-V utilizza un set di istruzioni di tipo RISC (istruzioni semplici che offrono elevate prestazioni). Esistono 3 tipi di istruzioni:

- istruzioni di tipo R → Aritmetico logiche
- Istruzioni di tipo I → Immediate o di trasferimento
- Istruzioni di tipo J → Di salto, condizionato o meno

Ciascuna operazione aritmetica MIPS esegue solo un'operazione e deve contenere esattamente tre variabili. A differenza dei programmi scritti nei linguaggi ad alto livello, gli operandi delle istruzioni aritmetiche del MIPS devono obbedire ad alcune restrizioni: devono essere scelti tra un numero limitato di locazioni particolari, chiamate registri. Il mips ha 32 registri a 32 bit. Le istruzioni aritmetiche del mips nello stesso ciclo di clock possono leggere i due registri su cui eseguire l'operazione e scrivere il risultato. Le istruzioni trasferimento dati del mips invece possono leggere scrivere un solo operando senza poter eseguire nessuna operazione su di esso. Nel MIPS le parole devono iniziare sempre a indirizzi multipli di quattro questo requisito si chiama vincolo di allineamento. Nel posizionamento in memoria dei dati il MIPS utilizza il posizionamento Big Endian. Cioè utilizza il byte più significativo, quello più a sinistra, per specificare l'indirizzo della parola. Nel MIPS le istruzioni hanno tutte la stessa lunghezza però esistono formati differenti di istruzioni:

- formato di tipo R
- formato di tipo I
- formato di tipo j

## Cosa differenzia ARM e MIPS?

Entrambi i processori fanno parte della famiglia dei processori RISC. Entrambi i set di istruzioni hanno dimensioni di istruzioni fisse a 32 o 64 bit. Tuttavia, il MIPS è fornito di 32 registri a 32 bit mentre l'ARM ha soltanto 16 registri a 32 bit e non riserva un registro per il valore 0. Uno svantaggio dei processori ARM è che nelle istruzioni condizionali utilizza i condition code, cosa che un processore RISC non potrebbe fare, in quanto sono dei bit interni alle parole di stato del processore, che producono in un confronto quattro possibili risultati: negativo, zero, riporto e overflow. Questi risultati possono essere impostati in seguito all'esecuzione di una qualsiasi istruzione aritmetica o logica, ma, a differenza delle architetture precedenti, l'impostazione dei bit della parola di stato è facoltativa. La specifica esplicita di queste condizioni crea minori problemi nell'implementazione su pipeline. Ma utilizzarle in un processore moderno non è una grande idea, perché se riesco a fare due o tre operazioni per volta nel senso che dispongo di due o tre ALU, quindi riesco a fare più somme, ognuna di queste tenterà di modificare i condition code e quindi un unico set di condition code non andrà bene; è meglio, quindi, mettere il risultato ogni volta in un registro diverso sarebbe questa l'unica soluzione. Un vantaggio del processore ARM è che, a differenza del MIPS, ogni istruzione può essere condizionale, significa cioè che l'esecuzione di un'istruzione dipende dal risultato di un confronto. Serve ad evitare i branch, così il processore non deve fare previsioni. Inoltre, il MIPS ha soltanto 3 modalità di indirizzamento dei dati, mentre l'ARM dispone di 9 modalità alcune ottenute tramite calcoli complessi. Per esempio, uno dei modi di indirizzamento dell'ARM prevede che si possa spostare il contenuto di un registro di un numero di posizioni arbitrarie, per poi sommarlo a quello di un altro registro per formare un indirizzo, e aggiornare infine il contenuto di un terzo registro con questo indirizzo.

## Cosa differenzia MIPS e il RISC-V?

Entrambe le architetture hanno:

- tutte le istruzioni ampie 32 bit
- 32 registri uso generale con un registro impostato fisso 0
- Un unico modo di accedere alla memoria, cioè, attraverso le istruzioni di load e store
- A differenza di alcune architetture non ci sono istruzioni nel MIPS e nel RISC-V che possano trasferire alla o dalla memoria più di un registro
- istruzioni che possono saltare se il contenuto di un registro è uguale a 0 o è diverso da 0
- che le modalità di indirizzamento in entrambe le architetture possono trasferire dati di tutte le dimensioni.

Una delle differenze principali riguarda i salti condizionati, basati su condizioni diverse da uguaglianza o disuguaglianza. Il RISC-V fornisce semplicemente istruzioni di salto condizionato. Inoltre il MIPS ha la possibilità di utilizzare sia la versione con segno che senza segno dell'istruzione set less than. Infine, l'altra differenza principale è che il MIPS completo ha un insieme di istruzioni molto più ampio del RISC-V.

## **Differenza tra processore MIPS e x86?**

La prima è che nell'X86 le istruzioni aritmetico-logiche hanno sempre un operando che funge sia da sorgente sia da destinazione, mentre il MIPS e l'ARM consentono di definire registri distinti per operandi sorgente e operando destinazione. Questa restrizione rende più complesso il problema del numero limitato di registri, dato che il contenuto di uno dei registri sorgente viene necessariamente modificato. La seconda importante differenza consiste nel fatto che uno degli operandi può essere contenuto in memoria; perciò praticamente tutte le istruzioni possono prendere uno dei due operandi dalla memoria, a differenza del MIPS e dell'ARM.

## **Qual è l'istruzione più lunga nel RISC-V?**

La load, poiché utilizza 5 unità funzionali in sequenza: memoria istruzioni, register file, ALU, memoria dati e register file.

# Capitolo 3 - Architettura

## Aritmetica per Computer

### Addizione, sottrazione e overflow

L'operazione della somma (in complemento a due) di un calcolatore è svolta in maniera del tutto simile a come si svolge a mano: le cifre vengono sommate da destra verso sinistra con il riporto passato alla cifra sinistra.

Un overflow si verifica quando il risultato di un'operazione non può essere rappresentato con l'hardware a disposizione, in questo caso si parla di una parola di 64 bit.

Nell'addizione: la somma tra un negativo e un positivo non darà mai un overflow siccome il modulo della somma diminuirà. Un overflow si può verificare nella somma se sommo due numeri positivi e ottengo un negativo oppure se sommo due numeri negativi e ottengo uno positivo, siccome il modulo aumenta.

Nella sottrazione: non abbiamo rischio di overflow quando i due operandi hanno lo stesso segno (perché è una sottrazione tra due operandi con lo stesso segno corrisponde ad una addizione con due operandi di segno opposto) mentre si potrebbe ottenere overflow se svolgiamo numero positivo-numero negativo e otteniamo un numero negativo oppure se svolgiamo numero negativo-positivo e otteniamo un numero positivo.

Per i numeri senza segno (unsigned) il compilatore può facilmente controllare l'overflow utilizzando un'istruzione di salto condizionato: un'addizione genera overflow se la somma è inferiore a uno dei due addendi mentre una sottrazione genera overflow quando la differenza è maggiore del minuendo. In questo caso l'overflow viene quasi sempre ignorato, poiché i programmi non richiedono di rilevarlo quando si fanno operazioni sugli indirizzi, utilizzo più comune dei numeri interi senza segno.

### Aritmetica nei dati multimediali

Per i dispositivi multimediali non vale la logica adottata finora per l'overflow. Si applica infatti la saturazione: quando si verifica un overflow il risultato assume il valore positivo o negativo più grande rappresentabile, invece del valore del risultato calcolato nell'aritmetica in complemento a due. Per esempio, sarebbe fastidioso se girando la manopola del volume della radio il volume aumentasse man mano per diventare poi improvvisamente bassissimo. Con un meccanismo di saturazione, il dispositivo, una volta raggiunto il volume massimo, continuerebbe a riprodurre il suono a quel volume anche se si continua a girare la manopola.

### Moltiplicazione

Il primo operando è chiamato moltiplicando, il secondo operando è chiamato moltiplicatore e il risultato finale è detto prodotto. L'algoritmo della moltiplicazione può essere semplificato in questo modo:

1. si mette una copia del moltiplicando nella posizione opportuna se la cifra del moltiplicatore è 1.
2. si mette zero nella posizione opportuna se la cifra del moltiplicatore è 0.

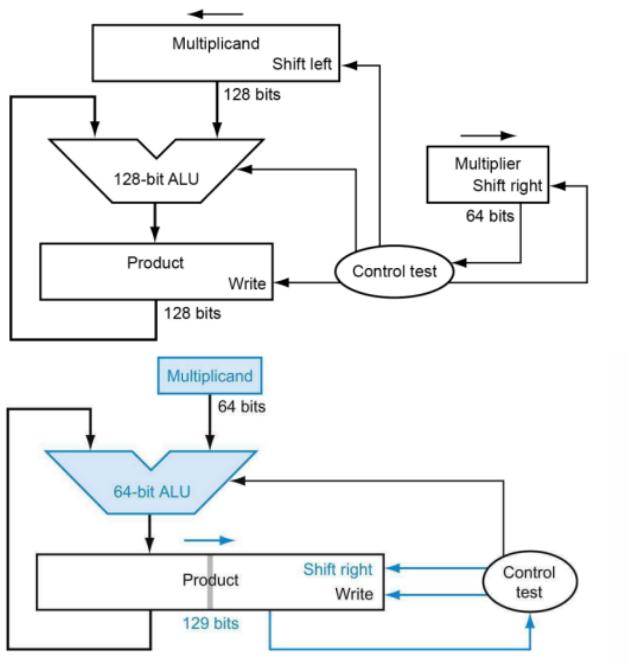
La prima osservazione è che il numero delle cifre del prodotto è considerevolmente maggiore rispetto al numero delle cifre del moltiplicando e del moltiplicatore. Di conseguenza, come per la somma, anche per la moltiplicazione si deve tener conto della possibilità che si verifichi un overflow.

#### Implementazione hardware dell'algoritmo della moltiplicazione

Si suppone che il moltiplicatore si trovi nel registro moltiplicazione di 64 bit e che il registro prodotto di 128 bit sia inizializzato a zero. Sulla base dell'algoritmo che conosciamo di moltiplicazione, è chiaro che sarà necessario spostare il moltiplicando a sinistra di una cifra a ogni passo, in modo che possa essere sommato correttamente agli altri prodotti intermedi.

Dopo 64 passi un moltiplicando su 64 bit si sarà spostato di 64 bit verso sinistra: ci sarà quindi bisogno di un registro moltiplicando di 128 bit (che sarà inizializzato inserendo il moltiplicando nel 64 bit nella metà di destra e una sequenza di zero nella metà sinistra).

La figura mostra i tre passi fondamentali necessarie per il calcolo di ogni bit: 1) il bit meno significativo determina se il moltiplicando debba essere sommato al registro prodotto; 2) si sposta il moltiplicando a sinistra; 3) avviene uno scorrimento a destra che fornisce il prossimo bit del moltiplicatore. Questi tre passi vengono ripetuti 64 volte: se ogni passo corrisponde a un ciclo di clock, l'algoritmo di moltiplicazione richiederebbe quasi 200 colpi di clock (secondo la legge di Amdahl, un'operazione lenta, seppur utilizzando a poco frequentemente, può limitare le prestazioni). L'algoritmo e l'hardware che lo implementa possono essere facilmente raffinati grazie all'esecuzione delle operazioni in parallelo: moltiplicatore e moltiplicando vengono fatti scorrere e contemporaneamente il moltiplicando viene sommato alla somma parziale, somma che avviene solo se il bit corrente del moltiplicatore è pari a uno. Tenendo conto che parte dei registri non viene utilizzata, l'hardware può essere ulteriormente ottimizzato per dimezzare la lunghezza del sommatore e dei registri. La legge di Moore ha consentito di avere sempre più risorse a disposizione tanto che i progettisti dell'hardware possono ora costruire i moltiplicatori molto più veloci rispetto al passato. Guardando ognuno dei 64 bit del moltiplicatore è possibile sapere, già all'inizio della moltiplicazione, se il moltiplicando debba essere sommato o meno. Moltiplicatori veloci si possono realizzare essenzialmente fornendo un sommatore a 64 bit per ogni bit del moltiplicatore: un modo alternativo per organizzare queste 64 addizioni è un albero parallelo e quindi, invece di aspettare 64 addizioni 64 bit, dobbiamo aspettare di meno. In realtà la moltiplicazione può essere ulteriormente velocizzata se si usano i sommatori a salvataggio di riporto. Inoltre, questo schema può essere facilitato se viene implementato in modalità pipeline. Per i numeri senza segno (unsigned), nelle operazioni di scorrimento occorre estendere il segno del numero contenuto in Prodotto. Al termine dell'esecuzione dell'algoritmo la parola meno significativa conterrà il prodotto su 64 bit.



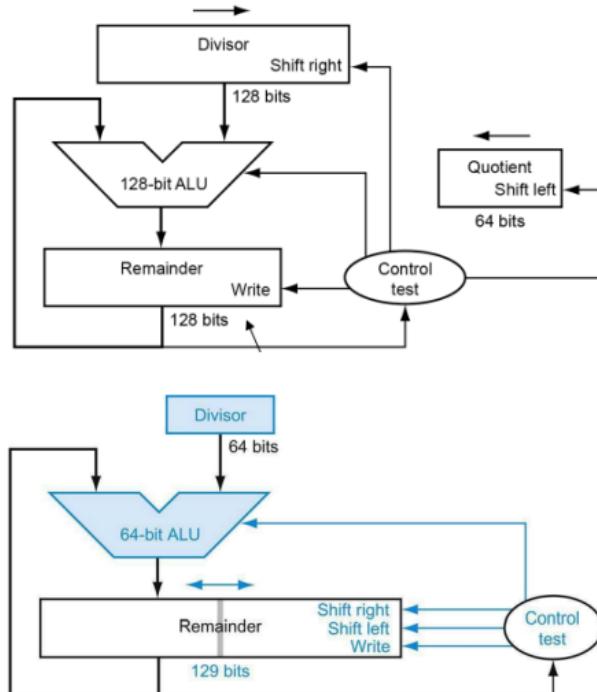
## Divisione

I due operandi, dividendo e divisore, di una divisione, oltre a produrre il risultato, quoziente, producono anche un resto (dove il resto è più piccolo del divisore). Vale la relazione: dividendo=quoziente X divisore + resto. I numeri binari contengono soltanto 0 e 1, quindi anche la divisione binaria è limitata a queste due possibilità quando verifichiamo quante volte il divisore sia nella porzione di dividendo considerata (sarà sempre 0 volte oppure 1).

### Implementazione hardware dell'algoritmo della divisione

La figura mostra i tre passi di questo algoritmo di divisione: 1) deve sottrarre il divisore per verificare se il divisore più piccolo del dividendo; 2) Se il risultato è positivo il divisore più piccolo è uguale al dividendo e quindi viene generato un 1 nel quoziente, se il risultato è negativo, il passo successivo consiste nel ripristinare il valore precedente del resto, sommando il divisore al resto, e nell'inserire uno 0 nel quoziente. 3) Il divisore viene fatto quindi scorrere a destra di una posizione e si ripete l'iterazione. L'algoritmo e l'hardware possono essere migliorati e diventare più veloci e più economici. Identificando le porzioni dei registri non utilizzati si può incrementare la velocità effettuando lo scorrimento degli operandi e del quoziente contemporaneamente la sottrazione. Quindi questa strategia dimezza la larghezza del sommatore dei registri. Si noti che questa implementazione hardware è molto simile a quella del moltiplicatore. Divisione di numeri dotati di segno. Facciamo una considerazione:  $+7:+2 \rightarrow$  Quoziente=+3, Resto=+1 | $-7:+2 \rightarrow$  Quoziente=-3, Resto=-1 | $+7:-2 \rightarrow$  Quoziente=-3, Resto=+1 | $-7:-2 \rightarrow$  Quoziente=-3, Resto=-1

Da questo verifichiamo un algoritmo: una divisione di numeri dotati di segno produce un quoziente negativo se i segni degli operandi sono opposti e fa sì che il segno del resto, quando questo non è nullo, corrisponda a quello del dividendo. La legge di Moore si applica anche all'hardware delle divisioni. La tecnica per velocizzare la divisione più frequente è quella della divisione SRT, che consiste nel predire il valore di un più bit del quoziente ad ogni passo, utilizzando una tabella predefinita che calcola i valori di alcuni bit del quoziente a partire da alcuni bit del dividendo e del resto calcolati prima.



## Floating Point

Oltre agli interi con e senza segno, esistono anche i numeri reali. Una notazione in cui vengono scritti è quella scientifica che prevede che il numero viene scritto con una sola cifra sinistra della virgola. Un numero in notazione scientifica senza zeri davanti alla virgola viene detto normalizzato. Allo stesso modo possiamo operare con i numeri binari, utilizzando come base non più 10 ma 2. L'aritmetica che supporta tali numeri viene detta in virgola mobile perché rappresenta numeri in cui la virgola binaria non è fissa, come per gli interi. Una notazione scientifica in forma normalizzata per i numeri reali offre tre vantaggi: 1) semplifica lo scambio di dati che includono numeri in virgola mobile; 2) semplifica gli algoritmi aritmetici per la virgola mobile, in quanto i numeri vengono rappresentati tutti nella stessa forma; 3) accresce l'accuratezza dei numeri memorizzabili in una parola, in quanto gli 0 alla sinistra del numero possono essere sostituiti da cifre aggiuntive della parte frazionaria. Bisogna trovare un compromesso tra la dimensione della mantissa (il valore, compreso tra 0 e 1, che viene posto dopo la virgola) e quella dell'esponente (il valore assunto dall'esponente nella rappresentazione in virgola mobile dei numeri). Se si aumenta la dimensione della mantissa migliora l'accuratezza del numero, mentre aumentando la dimensione dell'esponente aumenta l'intervallo dei numeri rappresentabili.

Lo standard IEEE 754 ha definito due formati: singola precisione e doppia precisione. In singola precisione, il numero è rappresentato da una parola di 32 bit, in cui 1 bit rappresenta il segno, 8 bit sono per l'esponente e 23 bit sono la mantissa. In realtà, dato che il primo bit nei numeri binari normalizzati, è sempre 1, la mantissa è lunga 24 bit. In doppia precisione, invece, il numero è rappresentato da due parole a 32 bit, quindi in totale 64: 1 bit per il segno, 11 bit per l'esponente e 52 per la mantissa. In realtà la mantissa ha 53 bit (per lo stesso motivo di prima). Anche per quanto riguarda questi numeri si possono verificare overflow e underflow: l'overflow si verifica quando l'esponente, positivo, è troppo grande per poter essere rappresentato nel campo esponente; l'underflow, invece, si verifica quando l'esponente negativo è troppo grande per essere contenuto nel campo esponente.

I progettisti dell'IEEE 754 hanno introdotto una rappresentazione efficiente per implementare il confronto tra numeri reali con le operazioni di ordinamento e confronto che abbiamo introdotto per gli interi. Il posizionamento dell'esponente prima della mantissa semplifica inoltre l'ordinamento dei numeri in virgola mobile attraverso l'utilizzo delle istruzioni di confronto tra interi. In questo modo i numeri con esponente maggiore appaiono più grandi numeri dei numeri con esponente minore purché ambedue esponenti abbiano lo stesso segno. I numeri con esponente negativo rappresentano un problema nell'ordinamento. Bisogna quindi che la notazione desiderata debba rappresentare l'esponente più negativo come 00000... e quello più positivo come 111111.... Questa notazione, infatti, è detta notazione polarizzata e la polarizzazione è il numero sottratto alla rappresentazione normale senza segno per determinare il valore reale. Lo standard IEEE 754 prevede una polarizzazione pari a 127 per la singola precisione (cosicché -1 venga rappresentato come -1+127, ovvero 126=0111 1110) e 1023 per la doppia precisione. La rappresentazione in virgola mobile secondo lo standard IEEE 754 è:

$$(-1)^s * (1 + \text{mantissa}) * 2^{(\text{esponente} - \text{polarizzazione})}$$

Lo standard IEEE 754 definisce anche delle configurazioni speciali utilizzate per rappresentare situazioni particolari. Ad esempio, invece di generare un'interruzione, quando si effettua la divisione per 0, il software può impostare la combinazione di bit associata a  $+\infty$  o  $-\infty$ . Oppure, quando si vogliono effettuare operazioni non valide come  $0/0$  o la sottrazione di infinito da infinito, si stamperà NAN (Not a number).

## Addizione in virgola mobile

Per sommare due numeri con parte frazionaria dopo aver svolto la notazione scientifica bisogna spostare la virgola del numero che ha l'esponente più piccolo in modo da allinearla al numero più grande (in modo da avere lo stesso esponente). Si esegue poi la somma dei significandi, si normalizza la somma e, poiché il significando può essere lungo soltanto quattro cifre (escluso il segno), occorre arrotondare il numero.

## Moltiplicazione in virgola mobile

Diversamente come si fa per la somma, si calcola l'esponente del prodotto semplicemente sommando gli esponenti degli operandi. Si esegue poi la moltiplicazione tra i significandi, si normalizza il prodotto ottenuto, si arrotonda e infine si calcola il segno del prodotto finale guardando il prodotto dei due operandi di partenza.

## Divisione in virgola mobile

Ricordiamo che lo shift a destra equivale a dividere un numero per  $2^i$ ; tuttavia, quest'operazione vale solo per gli interi unsigned.

## Accuratezza aritmetica

A differenza dei numeri interi, i numeri in virgola mobile costituiscono in genere un'approssimazione dei numeri che in realtà non possono essere rappresentati. Lo standard IEEE 754 prevede quattro modalità di arrotondamento: arrotondamento sempre al valore superiore, arrotondamento sempre al valore inferiore, troncamento e arrotondamento al numero pari più vicino (modalità detta "round to even"). Quest'ultima modalità ha definito cosa fare se il numero, ad esempio, si trova esattamente a metà tra due numeri rappresentabili. In binario consiste nell'arrotondare in modo tale che l'ultima cifra sia zero.

## Il parallelismo

Nasce con l'avvento dei supporti per la grafica e l'audio, soprattutto per lo sviluppo di teleconferenze e videogiochi. In questi ambiti, si è notato che si tende ad operare maggiormente su vettori di dati, e non su dati singoli. In questi casi, un processore può sfruttare il parallelismo per eseguire simultaneamente operazioni su vettori. Questo tipo di parallelismo viene spesso incluso nel più ampio concetto di parallelismo a livello di dati. Le architetture con queste estensioni vengono anche chiamate vettoriali o SIMD (Singola istruzione per dati multipli). Il primo esemplare nel mondo x86 fu un coprocessore FP 8087: esso utilizza un sistema per i floating point basato su stack. I registri FP sono a 32 o 64 bits. Lo svantaggio è che i codici sono difficili da ottimizzare, quindi ha una prestazione povera. Nella versione SSE2 vengono aggiunti 4 registri a 128 bit che possono

essere utilizzati in parallelo, ovvero in ognuno dei registri si possono inserire o due operandi da 64 bit (2x64) o quattro da 32 (4x32) e le istruzioni operano su di essi simultaneamente

## Perché i processori utilizzano la rappresentazione dei numeri in complemento a 2?

La rappresentazione in complemento a 2 viene utilizzata per rappresentare numeri interi in informatica e sistemi digitali. In questa rappresentazione, un bit viene utilizzato per segnalare il segno del numero (0 per positivo, 1 per negativo) e gli altri bit vengono utilizzati per rappresentare la magnitudo del numero.

I processori utilizzano la rappresentazione dei numeri in complemento a 2 perché questa rappresentazione ha alcuni vantaggi rispetto ad altre rappresentazioni.

In particolare:

- Semplicità delle operazioni aritmetiche: le operazioni di addizione e sottrazione possono essere effettuate utilizzando solo operazioni di bit a livello hardware, il che rende queste operazioni più veloci ed efficienti
- Risparmio di spazio: la rappresentazione in complemento a 2 consente di rappresentare sia numeri positivi che negativi utilizzando lo stesso numero di bit, il che significa che non è necessario dedicare spazio extra per il segno del numero
- Compatibilità con la logica binaria: la rappresentazione in complemento a 2 si adatta perfettamente alla logica binaria utilizzata nei circuiti elettronici dei processori, rendendo facili le operazioni di bitwise.

Tutte queste caratteristiche rendono la rappresentazione in complemento a 2 adatta per l'utilizzo dei processori.

N.B.: *per ottenere la rappresentazione in complemento a 2 di un numero negativo si parte dalla rappresentazione binaria del valore assoluto (che avrà il bit di segno = 0) e si prende il complemento a 1 di ciascun bit, quindi si aggiunge 1 al risultato.*

*Nel caso di numeri positivi, il complemento a 2 è uguale alla rappresentazione segno-grandezza.*

## Perché l'esponente della rappresentazione a virgola mobile non è mai rappresentato in complemento a 2?

L'esponente, nella rappresentazione a virgola mobile, non viene rappresentato in complemento a 2, ma utilizzando un formato di codifica noto come *formato di codifica dell'esponente normalizzato o formato di codifica dell'esponente intero*.

Questo formato utilizza un insieme di bit specifici per rappresentare l'esponente, solitamente in modo non complementare. La scelta di rappresentare l'esponente in questo modo dipende dalle esigenze specifiche dell'architettura hardware e dalla precisione richiesta per le operazioni in virgola mobile. La codifica dell'esponente normalizzato è un formato comune per rappresentare l'esponente nella rappresentazione a virgola mobile di un numero in informatica. In questo formato, l'esponente viene rappresentato come un numero intero che viene codificato utilizzando un insieme di bit specifico. La codifica dell'esponente normalizzato ha un valore di bias, che è un numero che viene sottratto dall'esponente effettivo prima della codifica. Questo valore di bias è scelto in modo da garantire che l'esponente codificato possa rappresentare sia numeri positivi che negativi. Ad esempio, supponiamo che la rappresentazione a virgola mobile utilizzi 8 bit per rappresentare l'esponente e che il valore di bias sia 127. In questo caso, l'esponente effettivo più piccolo che può essere rappresentato è -127 e l'esponente effettivo più grande che può essere rappresentato è 128. La codifica dell'esponente normalizzato è utilizzata perché consente di effettuare rapidamente le operazioni di confronto e trasferimento dei dati tra unità di elaborazione e di memoria utilizzando solo operazioni di bit a livello hardware. Questo rende la rappresentazione a virgola mobile molto efficiente per le operazioni matematiche.

## Come fai a capire come un numero in virgola mobile è più grande di un altro?

Per capire se un numero in virgola mobile è più grande di un altro è sufficiente confrontare le loro mantisse e i loro esponenti. Il numero con la mantissa più grande è il numero più grande. Se le mantisse sono uguali, allora il numero con l'esponente più grande è il numero più grande.

## Commenti finali

I bit non hanno un significato preciso, ma una stringa di bit potrebbe essere un'istruzione, un numero intero, un floating point, un numero senza segno, ecc.

Le rappresentazioni dei numeri che si utilizzano nei calcolatori:

- Hanno un range e una precisione finita. Il range è molto grande per i numeri in virgola mobile e molto piccolo per gli interi, ma i primi sono affetti da un evitabile errore di rappresentazione per i limiti del range. I nostri programmati ne devono tenere conto
- I set di istruzioni supportano l'aritmetica per i numeri signed e unsigned. Nei processori più avanzati ci sono operazioni specifiche pure per i floating point per fare operazioni sui numeri reali
- Per via del range e delle precisioni limitate, le operazioni possono essere affette da overflow (numero troppo grosso per essere rappresentato) e da underflow (numero troppo piccolo che viene rappresentato come 0)

## Come vengono scritti gli esponenti nella notazione in virgola mobile?

Gli esponenti in virgola mobile si rappresentano come potenza di 2 su 8 bit nella notazione a "virgola mobile a singola precisione", 11 invece nella notazione a doppia precisione (la doppia precisione si utilizza per evitare che si verifichino underflow ed overflow, dato che si prevede un esponente più grande).

## Rappresentazione normalizzata (numeri in virgola mobile)

Un numero in notazione scientifica senza zeri davanti alla virgola viene detto normalizzato ed è questo il modo usuale per rappresentare i numeri in tale notazione. Per poter scrivere un numero binario in forma normalizzata, si deve disporre di una base da poter incrementare o decrementare dello stesso numero di bit di cui il numero deve essere scalato, arrivando ad avere una sola cifra non nulla alla sinistra della virgola. Solo una base pari a 2 soddisfa questo vincolo. Poiché la base non è 10, è anche necessario un nuovo nome per la virgola: virgola binaria. L'aritmetica dei calcolatori che supporta tali numeri è detta aritmetica in virgola mobile perché rappresenta numeri in cui la virgola binaria non è fissa, come per gli interi.

## Perché l'uso della polarizzazione per l'esponente?

Lo Standard IEEE 754 prevede una rappresentazione dell'esponente in eccesso 127 per la singola precisione, ed una rappresentazione dell'esponente in eccesso 1023 per la doppia precisione. Questa rappresentazione si applica attraverso la "polarizzazione dell'esponente".

N.B. 127 per la singola precisione perché avendo 8 bits, si può rappresentare un intervallo di valori che va da 0 a 256, ma dato che voglio rappresentare sia numeri negativi che positivi, utilizzo il primo bit per il segno, così da poter rappresentare un intervallo di valori che vanno da -128 a 127.

Attraverso la polarizzazione dell'esponente si può fare in modo che il numero più grande che si possa rappresentare è 11111111, ed il numero più piccolo è 00000000, quindi nel momento in cui andiamo a confrontare due esponenti, non c'è bisogno di controllare prima il bit più significativo (per controllare il segno del numero), basta che li confrontiamo come due interi qualsiasi senza segno. Polarizzazione pari a 127 per la singola precisione: -125, per esempio, viene rappresentato come  $-125+127 = 2 = 00000010$  (in base 2) -1, per esempio, viene rappresentato come  $-1+127 = 126 = 01111110$  (in base 2) 0, per esempio, viene rappresentato come  $0+127 = 127 = 01111111$  (in base 2) +1, per esempio, viene rappresentato come  $+1+127 = 128 = 10000000$  (in base 2) +125, per esempio, viene rappresentato come  $+125+127 = 252 = 01111110$  (in base 2).

## Cos'è la rappresentazione per eccessi? Perché si usa questa e non il complemento a 2?

L'unica differenza tra i due consiste nel fatto che viene invertito il bit di segno. Nella rappresentazione per eccesso i numeri negativi si determinano come somma di se stessi e K con  $K=2^{l-1}$  dove l è il numero di bit utilizzati. Si noti che il sistema è identico al complemento a due con il bit di segno invertito. I numeri compresi in  $[-2^{l-1}, 2^{l-1}-1]$  sono mappati nell'intervallo  $[0, 2^l-1]$ . In tale rappresentazione, il numero binario che rappresenta  $-2^{l-1}$  sarà associato allo zero, mentre i valori minori di  $2^{l-1}$  ai numeri negativi e quelli maggiori a quelli positivi. Nel caso di  $l=8$  i numeri appartenenti a  $[-128, 127]$  sono mappati nell'intervallo  $[0, 255]$ .

## Perché il processore utilizza la rappresentazione a 2 e non il modulo e segno?

Il complemento a due è stato messo in pratica per permettere al processore di fare facilmente le somme algebriche: infatti il modulo e segno utilizza un bit di segno per mancanza di altri simboli nel calcolatore, ma questo porta a dei numeri negativi scritti in modo "errato". Nel complemento a due, pur mantenendo il primo bit di segno come il modulo e segno, la codifica dei numeri negativi è diversa dalla semplice anteposizione del bit 1: è più complessa, ma permette al processore di fare la somma algebrica semplicemente sommando i numeri.

# Capitolo 4 - Architettura

## Il processore

Il **datapath** è un insieme di unità di calcolo, come ad esempio le unità di elaborazione (ALU), i registri e i moltiplicatori necessari all'esecuzione delle istruzioni nella CPU. Il passaggio di due operandi attraverso la ALU e la memorizzazione del risultato in un nuovo registro viene detto ciclo di data path. Tale ciclo definisce ciò che è in grado di fare una macchina: più veloce è il ciclo del datapath, più è veloce la macchina. Ogni istruzione ISA viene eseguita in uno o più cicli di datapath; diversi cicli sono necessari per istruzioni più complesse, come la divisione.

In architettura non parallela, il ciclo di datapath corrisponde al ciclo di clock (misurato in nanosecondi), ossia l'intervallo di tempo utilizzato per sincronizzare le diverse operazioni del processore. La velocità con cui viene compiuto un ciclo di datapath contribuisce significativamente a determinare la velocità della CPU.

**Fase di fetch di un'istruzione:** si preleva dalla memoria l'istruzione che dev'essere eseguita.

### Costruzione di un processore

Le istruzioni di base sono:

- le istruzioni di accesso alla memoria (ld, sd);
- le istruzioni aritmetico-logiche (add, sub, and, or);
- le istruzioni di salto condizionato (beq).

Molti componenti per la realizzazione di queste funzioni sono gli stessi, indipendentemente dal tipo di istruzione.

I primi due passi sono identici:

1. Inviare il contenuto del Program Counter (che contiene in ogni momento l'indirizzo della prossima istruzione) alla memoria che contiene il programma e da essa prelevare l'istruzione (fetch).
2. Leggere il contenuto di uno o due registri utilizzando i campi dell'istruzione per selezionare i registri. Per load ci vuole un solo registro, ma la maggior parte delle istruzioni richiede la lettura di due registri.

Il resto delle azioni dipende dal tipo di operazione da compiere, anche se a grandi linee sono le stesse indipendentemente dal codice operativo dell'istruzione.

Infatti, tutti i tipi di istruzioni utilizzano la ALU dopo aver letto i registri per scopi diversi a seconda dell'istruzione:

- per un'istruzione di tipo aritmetico-logico, il risultato dell'ALU dev'essere scritto in un registro;
- per un'istruzione di load o store, il risultato dell'ALU viene utilizzato come indirizzo, rispettivamente per leggere dalla memoria il dato da scrivere nel register file o per scrivere il contenuto di un registro in memoria;
- per i salti condizionati, l'uscita dell'ALU determina l'indirizzo dell'istruzione successiva.

Dopo che il PC viene utilizzato per accedere ad un'istruzione, esso verrà incrementato di 4 per puntare all'istruzione successiva.

Tutte le istruzioni iniziano utilizzando il PC per fornire il loro indirizzo alla memoria istruzioni. Dopo che l'istruzione è stata caricata (fetch), il numero d'ordine dei registri contenenti gli operandi può essere letto nei campi opportuni dell'istruzione stessa. Una volta caricati gli operandi, si può: elaborare il loro contenuto per determinare un indirizzo di memoria (nel caso del load o dello store), per eseguire un calcolo effettivo (nel caso di operazioni aritmetico logiche su interi) o per eseguire un confronto (salto condizionato).

- Se l'operazione è un load o uno store, il risultato dell'ALU viene scritto come indirizzo per leggere dalla memoria un dato da scrivere nel register file o per scrivere il contenuto di un registro in memoria.
- Se l'istruzione è di tipo aritmetico logica, il risultato dev'essere scritto in un registro.
- I salti condizionati richiedono l'utilizzo dell'uscita dall'ALU per determinare l'indirizzo dell'istruzione successiva: tale indirizzo può provenire dall'ALU, nella quale PC e offset vengono sommati, oppure da un sommatore apposito che incrementa il PC di 4.

Altre istruzioni:

PC (program counter) -> instruction memory: preleva (fetch) istruzioni dalla memoria. Il registro PC contiene l'indirizzo della prossima istruzione. Numeri dei registri -> register file (castata dai registri del processore), legge i registri per prelevare informazioni.

### Fattori di prestazioni di una CPU

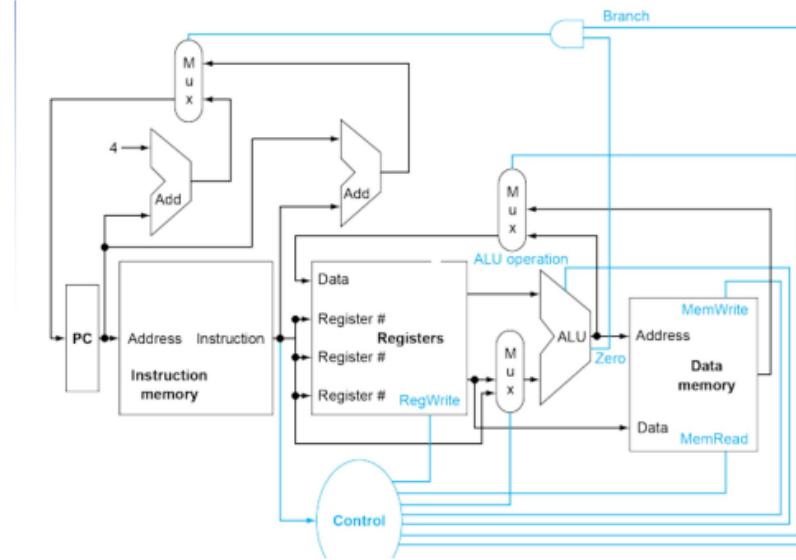
- *Instruction count*: determinato da ISA (set istruzione del processore) e dal compilatore. Ci sono dei processori, come il RISC-V, che hanno istruzioni semplici e che quindi ne richiedono di più per fare operazioni semplici, mentre ce ne sono altri che con una singola istruzione riescono a fare molteplici cose, quindi il numero di istruzione si riduce (ma non il tempo di esecuzione). Il compilatore può essere più o meno bravo a tradurre le istruzioni in linguaggio macchina
- *CPI (numero di colpi di clock)* e *Cycle Time (inverso della frequenza)*: determinato dall'hardware della CPU. Il RISC-V fa tutto in un singolo colpo di clock con frequenze abbastanza basse

Esamineremo due implementazioni RISC-V:

1. una versione semplificata che esegue istruzioni in sequenza

## Multiplexer

Ci sono due aspetti da considerare. Innanzitutto, nell'implementazione di prima ci sono dei dati che provengono da due diverse sorgenti e arrivano alla stessa unità funzionale. Abbiamo dunque bisogno di un circuito logico (multiplexer) che selezioni da quale, fra le differenti sorgenti collegate, debba essere preso il dato, connettendo la sorgente opportuna con la sua destinazione. In particolare, abbiamo bisogno di 3 multiplexer. L'altro aspetto da considerare è la necessità di avere delle linee di controllo per le principali unità funzionali. L'unità di controllo, che ha come ingresso l'istruzione, viene utilizzata per determinare come impostare le linee di controllo per le unità funzionali e per due dei tre multiplexer.



- Il multiplexer in alto controlla ciò che viene scritto nel PC: PC+4 o l'indirizzo di destinazione del salto condizionato; questo multiplexer viene controllato da una porta AND tra il segnale di zero in uscita dalla ALU e un segnale di controllo che indica che l'istruzione è un'istruzione di salto condizionato.
- Il multiplexer centrale, la cui uscita ritorna al register file, viene utilizzato per portare l'uscita della ALU (in caso di istruzioni aritmetico-logiche) o l'uscita dalla memoria dati (in caso di load) al register file per la scrittura.
- Infine, l'ultimo multiplexer viene utilizzato per determinare se il secondo ingresso dell'ALU provenga da registri (per istruzioni aritmetico-logica o per istruzioni di salto condizionato) o dal campo costante dell'istruzione stessa (per operazioni di load e store). Le linee di controllo (in figura in blu) determinano le operazioni che devono essere eseguite dall'ALU, oppure se la memoria debba essere letta o scritta, o ancora se si debba eseguire un'operazione di scrittura sul register file.

Gli elementi funzionali che costituiscono un'unità di elaborazione sono costituiti da due diverse classi di elementi logici: elementi che operano sui dati ed elementi che contengono lo stato. I primi sono di tipo combinatorio: un ogni istante i loro output dipendono solo dagli input ricevuti nello stesso istante. I secondi sono elementi di stato detti anche sequenziali e sono capaci di contenere uno stato però solo se hanno al loro interno elementi di memoria. Un elemento di stato ha due ingressi: il valore da scrivere nell'elemento e il clock, e un output, il valore contenuto al suo interno. L'esempio più semplice è il flip-flop D.

Nei componenti logici che contengono lo stato, le uscite dipendono quindi sia dagli ingressi che dal valore del loro stato interno.

## Metodologia di temporizzazione

La metodologia di temporizzazione definisce quando i segnali possono essere scritti e quando possono essere letti. Per semplicità, useremo la temporizzazione sensibile ai fronti (edge-triggered), nella quale i cambiamenti di stato avvengono su un fronte del segnale di clock (di salita o di discesa).

Asserito: segnale si trova nello stato logico alto.

Non asserito: segnale si trova nello stato logico basso.

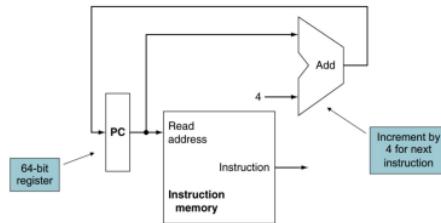
*E' l'operazione più lenta a determinare la massima frequenza del clock. Il periodo dev'essere, quindi, proporzionale all'operazione più lenta.*

## Costruzione del datapath

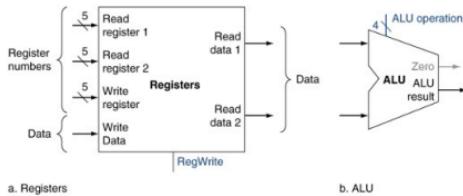
**Datapath:** tutti gli elementi che compongono la CPU, elementi che quindi elaborano dati, esclusa la Control Unit. Comprende i registri, l'ALU, i multiplexer, le memorie, ecc.

- Prelevamento istruzioni:** il primo elemento di cui abbiamo bisogno è un'unità di memoria in cui salvare le istruzioni, il Program Counter (PC) ed infine, è necessario un sommatore (elemento combinatorio) per incrementare di 4 il PC e ottenere l'indirizzo dell'istruzione successiva. Per eseguire una qualunque istruzione occorre anzitutto prelevare l'istruzione stessa dalla memoria. Per prepararsi ad eseguire l'istruzione

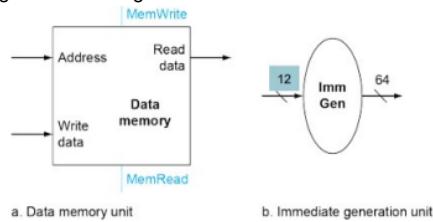
successiva, bisogna incrementare il PC di 4 byte per puntare all'istruzione successiva.



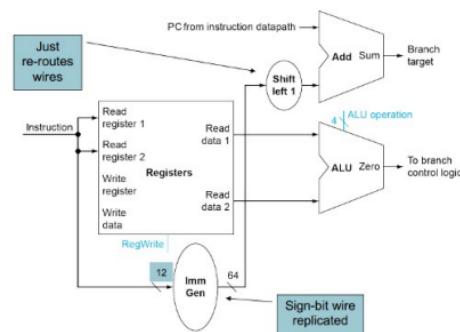
- Istruzioni aritmetico logiche** (dette anche istruzioni di tipo R): i due elementi richiesti per queste istruzioni sono: il register file, un insieme di registri in cui ciascun registro può essere letto o scritto specificando il numero a esso associato all'interno dell'insieme e la ALU. Il register file contiene tutti i registri e dispone di 2 porte di lettura e 1 di scrittura. In ogni istante, il register file fornisce in uscita il contenuto dei registri corrispondenti agli ingressi senza richiedere alcun segnale di controllo. Viceversa, la scrittura di un registro dev'essere indicata esplicitamente asserendo il segnale di controllo della scrittura. La scrittura avviene sul fronte attivo del clock per cui tutti gli ingressi interessati, cioè il dato da scrivere, il numero di registro e il segnale di controllo, devono essere validi sul fronte attivo del clock. Essendo la scrittura attiva sui fronti, si può tranquillamente leggere e scrivere lo stesso registro nello stesso ciclo di clock. La lettura fornirà il dato contenuto nel registro scritto in un ciclo di clock precedente, mentre il valore scritto potrà essere letto a partire dal ciclo di clock successivo.



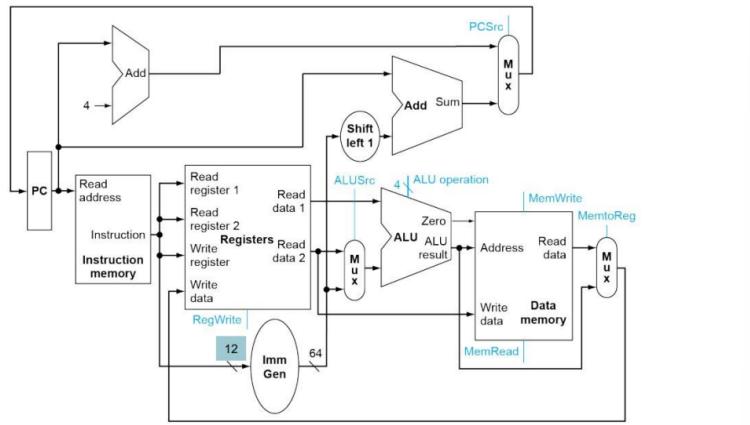
- Istruzioni di Load/Store**: le unità necessarie, oltre al register file e alla ALU, sono un'unità di memoria dati e un'unità di estensione del segno. L'unità di memoria è un elemento di stato (sequenziale) che ha come ingressi l'indirizzo del dato e il dato da scrivere e possiede una sola uscita per il dato letto. I segnali di controllo per la scrittura e la lettura sono separati, anche se all'interno di un ciclo di clock solo uno dei due può essere asserito. L'unità di estensione del segno riceve in ingresso un numero a 16 bit e lo fornisce in uscita esteso a 32 bit.



- Istruzioni di branch**: per i salti condizionati, l'unità di elaborazione impiega la ALU per valutare la condizione di salto e un sommatore a parte per determinare l'indirizzo di destinazione del salto, calcolato come *somma del PC incrementato di 4 + 16 bit meno significativi dell'istruzione (offset)*, estesi a 32 bit con segno e fatti prima scorrere a sinistra di 1 bit in modo tale che non codifichi lo spiazzamento in numero di byte ma in half-word (non si può saltare ad un indirizzo dispari). L'ALU determinerà se la condizione di salto è vera; si parla di salto condizionato eseguito (branch taken), o meno, e in funzione dell'uscita Zero dell'ALU, se nel PC debba essere caricato il contenuto del PC incrementato di 4 oppure l'indirizzo di destinazione del salto.



A questo punto possiamo combinare tra loro tutti i pezzi e creare una semplice unità di elaborazione:



Il primo datapath è in grado di eseguire un'istruzione in un periodo di clock.

- Ogni datapath può eseguire una funzione per volta
  - Ho bisogno quindi di due memorie separate
- Utilizzo i multiplexer per determinare da dove prendere i dati quando fonti alternate sono usate per istruzioni differenti.

## ALU

A seconda del tipo di istruzione, la ALU eseguirà una delle operazioni: per le istruzioni load e store la ALU deve eseguire una somma per calcolare l'indirizzo di memori; per le istruzioni di tipo R deve invece eseguire una delle quattro operazioni (AND, OR, somma e sottrazione), in funzione del valore dei 7 bit del campo funz7 e i 3 bit del campo funz3 dell'istruzione; per le istruzioni di salto condizionato, l'ALU deve eseguire una sottrazione tra i due operandi e controllare se il risultato è 0.

I 4 bit di controllo dell'ALU possono essere generalizzati usando una piccola unità di controllo che riceve in ingresso il campo funzione dell'istruzione e un campo di controllo di 2 bit, chiamato ALUOp.

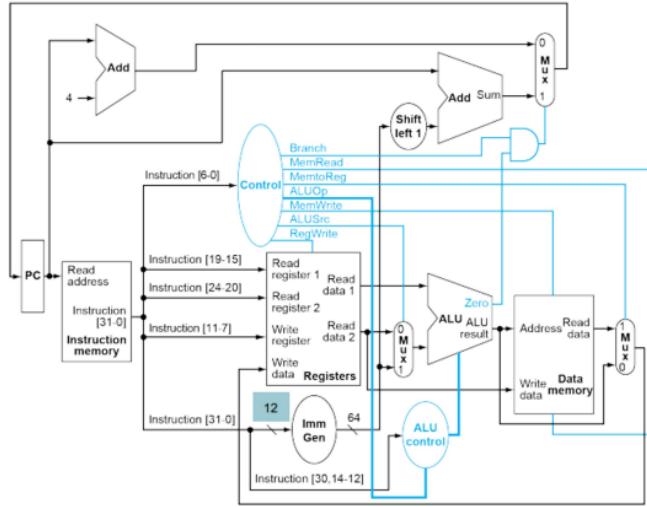
Il valore dei bit di controllo della ALU dipende, per le istruzioni di tipo R, dai bit di ALUOp e dal codice presente nel campo funzione. Il codice operativo (nella prima colonna) determina il valore dei bit di ALUOp. Quando il codice ALUOp vale 00 o 01, l'operazione che la ALU deve eseguire non dipende dal contenuto del campo funzione; in questo caso si dice che il contenuto del campo funzione è indifferente e viene indicato con XXXXXX. Invece, quando la ALUOp vale 10 viene utilizzato il campo funzione per determinare il valore degli ingressi di controllo della ALU.

## Unità di controllo (main control unit)

Ora che abbiamo visto il funzionamento di ciascuno sei segnali di controllo, dobbiamo specificare che l'unità di controllo può impostare tutti i segnali, tranne uno, basandosi solo sul codice operativo dell'istruzione stessa. L'eccezione è fatta da PCSrc, che viene asserito se 1) l'istruzione è un branch e anche 2) l'uscita Zero della ALU (per generare il segnale PCSrc, dunque, bisogna mettere in AND un segnale proveniente dall'unità di controllo, che verrà chiamato branch, con l'uscita Zero della ALU).

Nome del segnale	Effetto quando non asserito	Effetto quando asserito
RegWrite	Nullo	Il dato viene scritto nel register file nel registro individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 12 bit del campo immediato dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola il valore di PC + 4	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nullo	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea "Dato letto"
MemWrite	Nullo	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea "Dato scritto"
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati

## Datapath di controllo



Da notare l'AND: attiva il multiplexer quando c'è un branch e quando il risultato dell'ALU è 0.

Laddove non è un branch o il risultato del branch non è 0, al PC viene sommato 4.

Nel disegno vengono ingrigite le parti non utilizzate per ciascuna istruzione:

- Nelle R-type ignorano la memoria
- Nella Load serve la memoria, ma ignora il read register 2, il write data e l'instruction per l'ALU control. E' il percorso più lungo
- Nella beq ignorano la memoria, il write register/data e il risultato dell'ALU

## Prestazioni

Essendo l'operazione più lunga quella di Load, è quella a determinare il periodo di clock.

Percorso critico: memoria di istruzioni -> register file -> ALU -> memoria dei dati -> register file

Se tutte le operazioni durano lo stesso tempo, non posso migliorare le prestazioni seguendo il principio di rendere il caso più comune il più veloce.

\*Per migliorare le prestazioni viene sfruttato, quindi, il **pipelining**.

## Ottimizzazioni con la Pipeline

Abbiamo realizzato quindi un'implementazione a singolo ciclo cioè un'implementazione nella quale un'istruzione viene eseguita in un unico ciclo di clock. Sebbene un processore a singolo ciclo funzioni correttamente non viene più utilizzato nelle implementazioni moderne a causa della sua efficienza. Infatti, in questa implementazione il periodo di clock deve avere la stessa durata per tutte le istruzioni, quindi è determinato dal cammino di elaborazione più lungo all'interno del processore, (che sarà molto probabilmente associato a un'istruzione di load dato che essa utilizza 5 unità funzionali in sequenza: la memoria istruzioni, il register file, la ALU, la memoria dati e il register file). Il prezzo da pagare è significativo, ma potrebbe essere considerato accettabile per un insieme limitato di istruzioni. Dato che dobbiamo assicurare che la durata del ciclo di clock sia pari alla durata dell'istruzione più lunga da eseguire, non si possono sfruttare quelle tecniche che consentono di ridurre il tempo di esecuzione delle istruzioni più comuni: un'implementazione a un solo ciclo è in contrasto quindi con una delle grandi idee già introdotte che richiede di rendere veloci le situazioni più comuni. Si possono avere miglioramenti con l'uso delle pipeline.

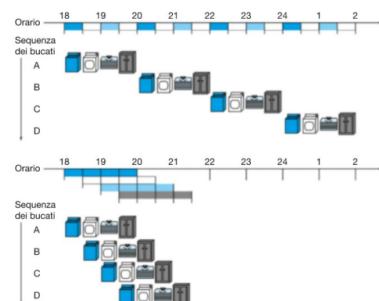
## Pipelining

Esempio della lavanderia:

l'idea è che devo lavare, asciugare, stirare e mettere a posto una certa quantità di indumenti. Ho 4 carichi di indumenti.

Ho due possibilità:

1. Opero sui 4 carichi separatamente -> se ogni fase dura mezz'ora e inizio alle 18, finisco per terminare il tutto alle 2 del mattino
2. Metto il secondo carico nella lavatrice mentre il primo sta asciugando e così via -> in questo modo finisco alle 21:30. Questo metodo funziona meglio perché ogni fase ha funzioni separate



Sono andato:  $8/3,5 = 2,2857 \approx 2.3$  volte più veloce.

Se non mi fermo mai e carico roba ogni mezz'ora, termine ogni mezz'ora con un nuovo pacco di indumenti pronto.

$$\frac{2n}{0.5} + 1.5 \approx 4n$$

**Con il pipelining la latenza rimane la stessa, ma il throughput viene migliorato.**

*Impiego lo stesso tempo per completare un'operazione, ma il tempo tra la produzione di un risultato e un altro (i pacchi di indumenti) è molto minore.*

Che succede se uno stadio è più lento degli altri? Bisogna comunque aspettare che quello più lento avrà finito: il throughput si assesta sulla velocità dello stadio più lento.

(Le code del supermercato non conta come pipeline, è semplicemente una coda unica. Il pipeline consiste nello scomporre l'operazione che c'è da fare in tante fasi che vengono eseguite da entità diverse. Il supermercato sarebbe pipelined se ogni commesso si occupasse di un singolo prodotto da incartare).

*La pipeline è una tecnica di implementazione hardware di un insieme di istruzioni che prevede la sovrapposizione temporale dell'esecuzione delle diverse istruzioni. Sfrutta il parallelismo tra le istruzioni di un programma software scritto in maniera sequenziale.*

Ci sono 5 stadi:

1. **IF**: Instruction Fetch (dalla memoria) -> caricamento dell'istruzione dalla memoria
2. **ID**: Instruction Decode & Register Read -> lettura dei registri e decodifica dell'istruzione
3. **EX**: Execute Operation or Calculate Address -> esecuzione di un'operazione o calcolo di un indirizzo
4. **MEM**: Access Memory Operand (non è sempre presente) -> accesso a un operando nella memoria dati
5. **WB**: Write result Back to register (non c'è nel caso del branch) -> scrittura del risultato in un registro

Senza pipeline dovremmo impiegare 5 cicli di clock per ogni istruzione (un clock per ogni passo), ma tramite la pipeline si avrà:

$$\text{Tempo tra 2 istruzioni con pipeline} = \frac{\text{Tempo tra 2 istruzioni senza pipeline}}{n \text{ di stadi della pipeline}}$$

## Prestazioni

La tecnica della pipeline migliora le prestazioni aumentando il throughput delle istruzioni, ma non riduce il tempo di esecuzione della singola istruzione. L'insieme delle istruzioni del RISC-V è stato progettato espressamente per l'esecuzione in pipeline.

Infatti:

- *Tutte le istruzioni hanno la stessa lunghezza, cioè 32 bit*: tale proprietà semplifica notevolmente la fase di fetch delle istruzioni nel primo stadio della pipeline e la loro decodifica nel secondo stadio.
- *Le istruzioni adottano un numero ridotto di formati diversi e in tutti formati, i registri sorgente sono sempre specificati nella stessa posizione*, qualunque sia l'istruzione: tale regolarità permette al secondo stadio di iniziare a leggere il register file mentre l'unità di controllo sta determinando il tipo dell'istruzione letta; altrimenti se il formato delle istruzioni non fosse regolare, sarebbe necessario spezzare in 2 questo secondo stadio portandola pipeline ad un totale di sei stadi.
- *Gli operandi residenti in memoria possono comparire solo nelle istruzioni di load e store*: questo vincolo permette di utilizzare lo stadio di esecuzione per calcolare l'indirizzo di memoria, per accedere poi alla memoria nello stadio successivo.

Nella versione pipelined, una volta finito il fetch di un'istruzione, inizio il fetch della prossima. Il periodo di clock dev'essere quello che permette di eseguire la fase più lenta (nel nostro caso è 200 ps). Il clock non va più proporzionato sulla singola istruzione, ma sulla singola fase.

Il pipelining è uno dei motivi per l'aumento delle frequenze di clock dei processori moderni (oltre all'avanzamento tecnologico).

## Speedup della pipeline

*Speedup = numero di passi della pipeline.*

Caso migliore: gli stadi sono bilanciati, ovvero durano tutti lo stesso tempo.

$$\text{TempoTraIstruzioni} = \frac{\text{TempoTraIstruzioni}}{\text{NumeroDiStadi}}$$

Se sono bilanciati, lo speedup è minore.

Lo speedup, in ogni caso, è dovuto ad un throughput incrementato.

Come detto prima, la latenza non diminuisce.

## Pipeline e design ISA

Gli ISA (Instruction Set Architecture) del RISC-V, al contrario di altri processori, è stato pensato per il pipeline:

- *Tutte le istruzioni sono a 32 bit*. E' più semplice fare il fetch e decifrare in un ciclo. Nota: l'x86 sfrutta la frammentazione delle istruzioni in istruzioni più semplici (1-17 byte) per permettere il pipelining. Questa frammentazione aumenta i consumi e peggiora le prestazioni
- *Pochi formati di istruzioni e regolari*. Può decifrare e leggere registri in un unico step. Nell'x86 ci sono ancora istruzioni che non si utilizzano più, ritrovandoci con un set più grosso di quello che si serve. La retrocompatibilità costa!
- *Indirizzamento load/store*. Posso calcolare gli indirizzi nel 3° stadio e accedere alla memoria nel 4°. Non posso quindi operare direttamente sulla memoria, ma in compenso posso fare pipelining separando le varie operazioni che posso fare. Nei processori con accesso diretto ho due accessi alla memoria, il che non permette la realizzazione del pipelining

## Hazards

*Situazioni in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente successivo.*

Possono essere di 3 tipi:

- *Hazard di struttura*: una risorsa di cui ho bisogno è occupata
- *Hazard di dati*: ho bisogno di aspettare che l'istruzione precedente completi la sua scrittura/lettura dei dati
- *Hazard di controllo*: decidere le azioni di controllo dipende dalle istruzioni precedenti. Esempio -> potrei dover aspettare il branch per vedere l'azione successiva

### Hazard strutturale

Si verifica quando le risorse hardware presenti non sono in grado di supportare la combinazione di istruzioni che si vorrebbe eseguire nello stesso ciclo di clock (perché impegnate in qualche altra operazione). Si verifica, ad esempio, quando non ho due memorie distinte e nella prima istruzione devo accedere ai dati in memoria e nella quarta deve essere prelevata della stessa memoria nello stesso ciclo di clock.

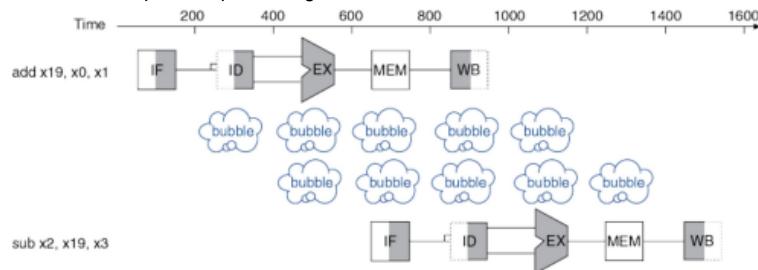
### Hazard sui dati

Si verifica quando un'istruzione non può essere eseguita in un certo ciclo di clock perché i dati di cui ha bisogno l'istruzione non sono ancora disponibili. Si verifica quando un'istruzione dipende dal risultato di un'istruzione precedente che si trova ancora all'interno della pipeline: avremo una situazione di stall (detta anche bolla).

Supponiamo, ad esempio, di avere un'addizione e una sottrazione che utilizza il risultato della somma.

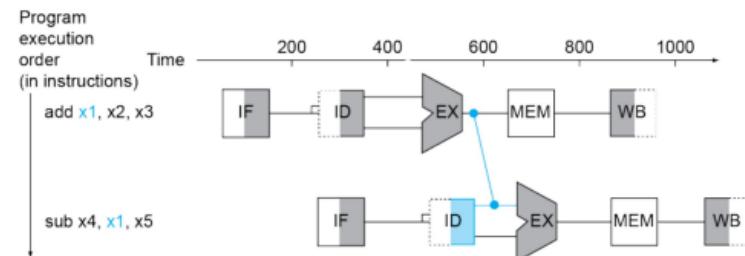
Esempio: add x19, x0, x1; sub x2, x19, x3 -> dopo che ho sommato, devo sottrarre il contenuto di x3.

Devo aspettare di scrivere il risultato in x19 prima di poter eseguire la sottrazione. Introduco così delle "bolle" nella pipeline, ovvero delle attese.



Se non si prendessero delle contromisure, gli hazards vita ti potrebbero causare tanti stalli della pipeline: L'istruzione add, infatti, non scrive il proprio risultato prima del quinto stadio e ciò significa che si dovrebbero perdere 3 cicli di clock.

La soluzione più utilizzata è chiamata propagazione o bypassing, in cui, nell'esempio preso in considerazione



non appena il risultato della somma viene generato dalla ALU, questo potrebbe essere già utilizzato come ingresso alla ALU per l'operazione di sottrazione. La tecnica che prevede l'aggiunta di un circuito che collega in un punto dell'esecuzione il dato mancante, prendendolo da una risorsa interna. La propagazione funziona solamente se lo stadio a cui il dato viene propagato è successivo nel tempo allo stadio dal quale viene prelevato. Questa tecnica funziona molto bene ma non è in grado di evitare tutti gli stalli di una pipeline. Infatti, se ad esempio, la prima istruzione è un load, il dato desiderato sarebbe disponibile solo dopo il quarto stadio della prima istruzione, ossia troppo tardi per essere utilizzato come input del terzo stadio della sottrazione. Di conseguenza, anche disponendo della propagazione, dovremmo imporre uno stall della durata di un ciclo di clock a causa di un hazard sui dati di una load.

Quindi, Forwarding (Bypassing): uso il risultato quando è stato computato. NON aspetto di caricarlo in un registro. Questa soluzione richiede una connessione extra nel datapath.

### Hazard sul controllo (o Hazard sui salti condizionati)

Si verifica quando bisogna prendere una decisione in funzione del risultato dell'esecuzione di un'istruzione e altre istruzioni sono già state avviate all'esecuzione.

Una soluzione potrebbe essere quella di mettere in stall la pipeline immediatamente dopo aver caricato dalla memoria un'istruzione di branch e attendere fino a che la pipeline non abbia determinato il risultato del confronto prima di calcolare l'indirizzo dell'istruzione successiva (stall on branch). Il costo di questa soluzione è però troppo elevato.

Un'altra soluzione è quella basata sulla predizione: si basa sull'ipotesi che il confronto associato a un salto condizionato dia un certo risultato; l'esecuzione procede basandosi su questa ipotesi invece di aspettare che sia verificato il risultato effettivo del confronto.

Ci sono due tecniche di previsione:

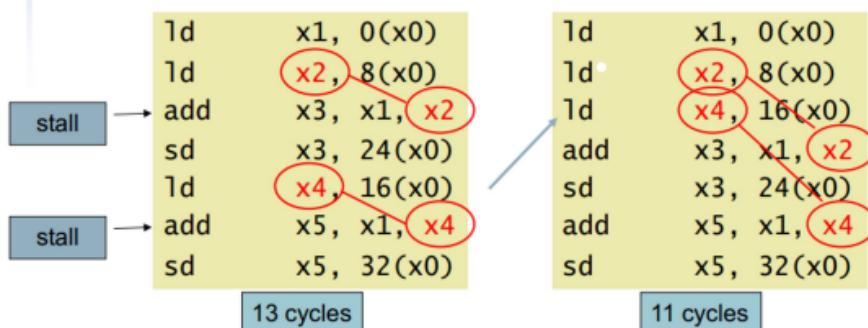
- *Predictors hardware statici*: l'approccio consiste nel predire sempre che il salto venga non eseguito. Una versione più sofisticata consente di prevedere se un salto debba essere eseguito o meno. Ad esempio, i salti che si trovano alla fine del ciclo e che sono utilizzati per ritornare all'inizio del ciclo, sono eseguiti la maggior parte delle volte. Quindi il processore potrebbe decidere che i salti verso indirizzi minori vadano sempre eseguiti.

- Predictors hardware dinamici*: determinano la predizione per ciascun salto in funzione del comportamento precedente di quell'istruzione di salto, e possono modificare la predizione di ogni salto durante l'esecuzione del programma (più utilizzati e più efficienti rispetto a quelli statici). Un'implementazione diffusa consiste nel memorizzare la storia di ciascun salto, ricordando quando è stato preso o no, e nell'utilizzare il comportamento recente per predire il futuro. Quando la predizione è errata, l'unità di controllo della pipeline deve garantire che le istruzioni che seguivano quelle di salto e sono già state caricate nella pipeline non abbiano alcun effetto, e deve far ripartire la pipeline dall'indirizzo corretto. Un modo per implementare la strategia di predizione dinamica dei salti consiste nell'utilizzare un buffer di predizione dei salti, detto anche tabella della storia dei salti: è una piccola memoria che contiene un bit che indica se il salto era stato eseguito o meno nell'ultima esecuzione. Questo semplice schema a un bit ha però un inconveniente: anche se un salto venisse quasi sempre effettuato, la predizione risulterebbe sbagliata almeno due volte (la prima e l'ultima) invece che una volta sola, quando il salto non viene effettuato. Per rimediare, si utilizzano degli schemi di previsione a 2 bit, in cui la predizione deve essere sbagliata due volte di seguito prima di essere modificata. Questo approccio viene rappresentato da una macchina a stati finiti. Un predictor ci dice se un salto debba essere effettuato o meno ma il calcolo dell'indirizzo di destinazione del salto rimane un problema. In una pipeline a cinque stadi questo calcolo richiede un ciclo di clock. Ciò significa che i salti che vengono effettuati avrebbero il costo di un ciclo. I salti ritardati sono un approccio possibile per eliminare questo costo. Un altro approccio consiste nell'utilizzare una memoria cache per salvare l'indirizzo di destinazione del salto o l'istruzione di destinazione, detta anche buffer degli indirizzi di salto.

#### Scheduling del codice per prevenire lo stall

Riordino il codice per evitare di usare il risultato di load nella prossima istruzione.

Laddove l'hardware non riesce a risolvere questi stalli, bisogna intervenire via software.



#### Hazard di controllo

Sono legati ai branch, i quali determinano il flusso di controllo.

Se l'unità di fetch è abbastanza intelligente da notare un fetch incondizionato, non crea problemi di controllo e passa direttamente al prossimo ciclo.

Nel caso di branch condizionato, il fetch della prossima istruzione dipende dal risultato del branch.

La pipeline non può sempre fetchare l'istruzione corretta.

Nella pipeline del RISC-V:

devo confrontare i registri e computare prima l'obiettivo nella pipeline e quindi devo complicare l'hardware aggiungendo dei componenti.

#### Sintesi della pipeline

La pipeline è una tecnica che sfrutta il *parallelismo* tra le istruzioni di un programma software scritto in maniera sequenziale. Rispetto alla programmazione dei multiprocessori, ha il grande vantaggio di essere sostanzialmente invisibile al programmatore.

La pipeline aumenta il numero di istruzioni che vengono eseguite contemporaneamente e la frequenza con cui viene iniziata e terminata l'esecuzione delle istruzioni. Questa tecnica non riduce il tempo richiesto per completare l'esecuzione della singola istruzione, chiamato anche latenza. Per esempio, una pipeline a cinque stadi richiede sempre cinque cicli di clock per completare l'esecuzione di un'istruzione.

La pipeline migliora il throughput delle istruzioni ma non il tempo di esecuzione, o latenza, della singola istruzione. Gli insiemi di istruzioni possono semplificare o rendere particolarmente difficile il compito ai progettisti della pipeline, i quali devono comunque gestire gli hazard strutturali, sul controllo e sui dati. I meccanismi di predizione dei salti condizionati e la propagazione aiutano a rendere veloce l'elaborazione mantenendo un funzionamento corretto del calcolatore.

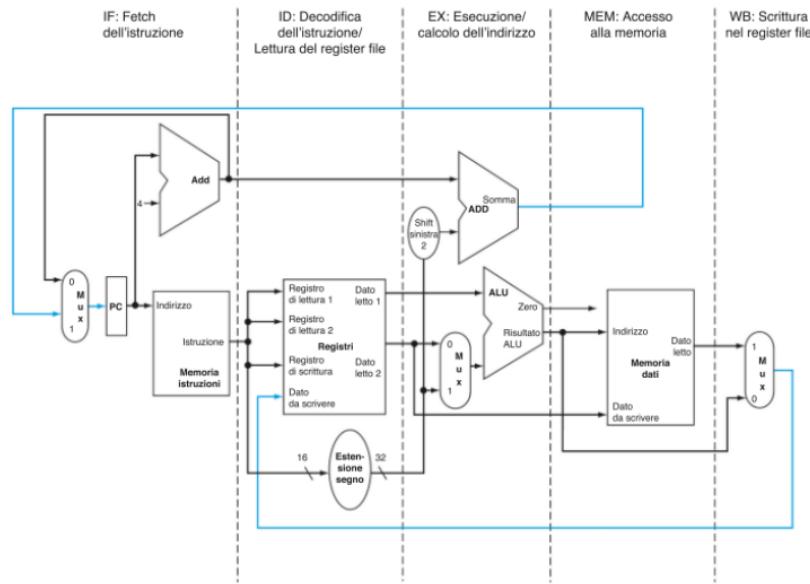
La pipeline migliora le prestazioni, incrementando il throughput delle istruzioni.

- Esegue istruzioni multiple in parallelo (mentre uno fa una cosa ce n'è un'altra che fa altro)
- Ogni istruzione ha la stessa latenza

E' vulnerabile agli hazard (strutturali, di dati, di controllo).

L'instruction set architecture (ISA) ha effetto sulla complessità dell'implementazione della pipeline. Più il set è semplice e regolare, più l'implementazione della pipeline risulta meno complicata.

#### Datapath (Unità di elaborazione) con pipeline e unità di controllo associata



**Figura 4.33.** L'unità di elaborazione a singolo ciclo del Paragrafo 4.4 (simile a quella di Figura 4.17). Ciascun passaggio dell'esecuzione di un'istruzione può essere messo in corrispondenza con una parte dell'unità di elaborazione, seguendo il flusso di elaborazione da sinistra verso destra. Le sole eccezioni, mostrate in blu, sono costituite dall'aggiornamento del PC e dalla scrittura del risultato: nella fase di WB il risultato calcolato dalla ALU o il dato letto dalla memoria dati viene portato alla sinistra dello schema per scrivere nel register file (di solito utilizziamo il colore blu per evidenziare i segnali di controllo, ma in questa figura vogliamo evidenziare linee dati).

La figura mostra l'unità di elaborazione a singolo ciclo con i cinque stadi della pipeline ben identificati.

La suddivisione di un'istruzione in cinque fasi implica una pipeline a cinque stadi, il che significa che in un singolo ciclo di clock ci potranno essere fino a cinque istruzioni in esecuzione contemporaneamente.

Occorre quindi separare l'unità di elaborazione in cinque parti, ciascuna delle quali prende il nome dalla fase di esecuzione dell'istruzione corrispondente:

1. IF: fetch dell'istruzione
2. ID: decodifica dell'istruzione e lettura del register file
3. EX: esecuzione o calcolo dell'indirizzo
4. MEM: accesso alla memoria
5. WB: scrittura nel register file

Durante l'esecuzione, istruzioni e dati si spostano generalmente da sinistra a destra, attraversando le cinque fasi.

Vi sono, tuttavia, due eccezioni al flusso di esecuzione da sinistra verso destra:

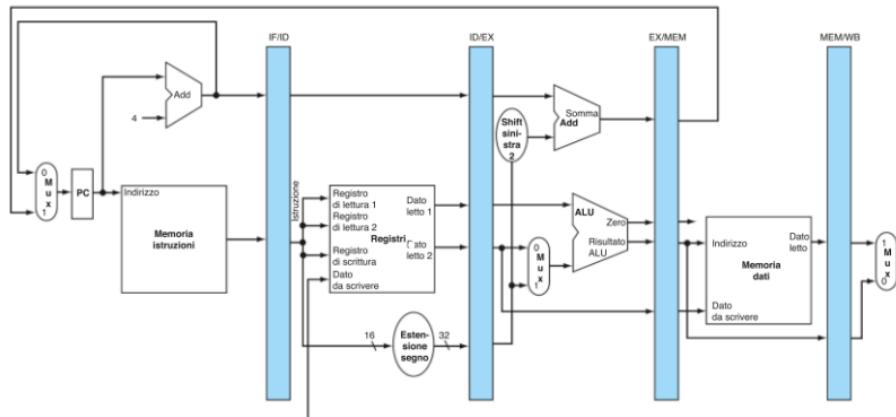
- lo stadio di scrittura del risultato, nel quale viene scritto il risultato dell'esecuzione nel register file al centro dell'unità di elaborazione;
- la selezione del valore successivo del PC, che viene scelto tra il valore del PC incrementato di 4 e l'indirizzo di destinazione del salto proveniente dallo stadio MEM

Il flusso dei dati da destra a sinistra non ha effetto sull'istruzione corrente: questo spostamento all'indietro dei dati influenzerà solo le istruzioni successive presenti nella pipeline.

Si noti che la freccia da destra a sinistra posta nella parte inferiore dello schema può causare hazard sui dati, mentre la seconda freccia nella parte superiore dello schema può provocare hazard sul controllo.

Un modo per descrivere ciò che succede nell'esecuzione con pipeline è quello di ipotizzare che ciascuna istruzione disponga di una propria unità di elaborazione e di inserire tale unità nella corretta posizione lungo l'asse dei tempi, visualizzando le relazioni tra le diverse istruzioni.

La figura:



**Figura 4.35.** La versione con pipeline dell'unità di elaborazione di Figura 4.33. I registri di pipeline, evidenziati in blu, separano i diversi stadi della pipeline e per etichettarli si usa il nome dei relativi stadi che separano; per esempio, il primo registro viene etichettato con IF/ID, poiché separa lo stadio IF, di prelevamento dell'istruzione, dallo stadio ID, di decodifica dell'istruzione. I registri devono avere un'ampiezza sufficiente a memorizzare tutti i dati corrispondenti alle linee che li devono attraversare. Il registro IF/ID, per esempio, deve essere ampio 64 bit, poiché deve contenere sia l'istruzione prelevata dalla memoria su 32 bit sia il contenuto del PC incrementato di 4, sempre su 32 bit. Gli altri registri verranno esaminati più in dettaglio nel seguito del capitolo; anticipiamo solamente che gli altri tre registri di pipeline contengono rispettivamente 128, 97 e 64 bit.

mostra l'unità di elaborazione con pipeline in cui sono evidenziati i registri della pipeline.

In ogni ciclo di clock, tutte le istruzioni procedono da un registro di pipeline a quello successivo.

I registri prendono il nome dai due stadi che separano; per esempio, il registro di pipeline inserito tra gli stadi IF e ID è chiamato IF/ID.

*Si noti che non c'è alcun registro di pipeline alla fine dello stadio di WriteBack.*

Tutte le istruzioni devono modificare qualche elemento dello stato del processore.

Ogni istruzione aggiorna il PC, incrementandolo di 4 o scrivendo l'indirizzo di destinazione del salto.

Il PC può essere visto come il registro di pipeline che alimenta lo stadio IF.

Tuttavia, a differenza dei registri di pipeline, il PC fa parte dello stato dell'architettura, visibile all'esterno: il suo contenuto dev'essere salvato quando si verifica un'eccezione, mentre il contenuto dei registri di pipeline veri e propri viene perso.

Le Figure dalla 4.36 alla 4.38 costituiscono la prima sequenza di schemi e mostrano (evidenziata in blu) la parte dell'unità di elaborazione attiva nei diversi stadi di esecuzione di un'istruzione di load.

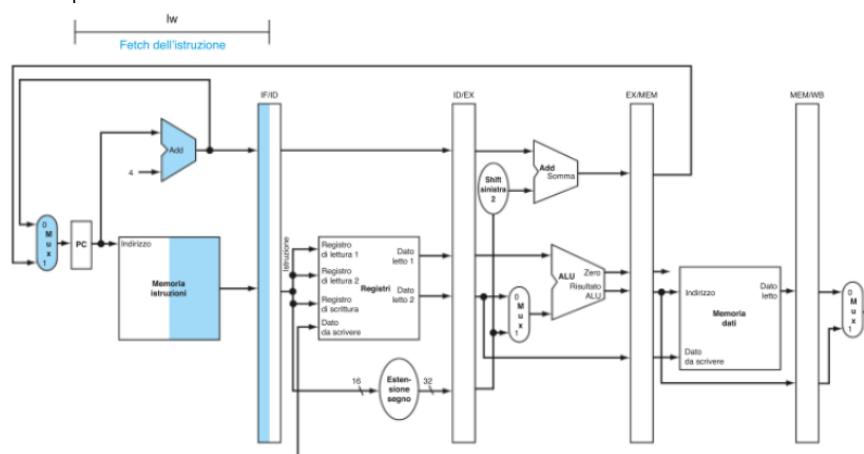
Questa istruzione viene esaminata per prima, essendo attiva in tutti e cinque gli stadi.

Come nelle Figure 4.28-4.30, verrà evidenziata la metà destra dei registri o della memoria quando queste unità vengono lette e la metà sinistra quando vengono scritte.

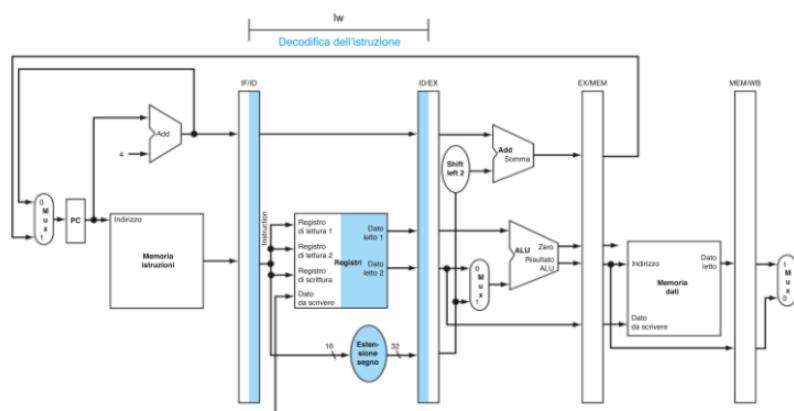
In ciascuna figura si riporterà il nome dell'istruzione, *lw*, insieme al nome dello stadio attivo della pipeline.

I cinque stadi sono:

1. **Fetch dell'istruzione.** La parte superiore della Figura 4.36 mostra la fase in cui l'istruzione viene prelevata dalla memoria istruzioni (utilizzando l'indirizzo contenuto nel PC) e scritta nel registro di pipeline IF/ID. L'indirizzo contenuto nel PC viene incrementato di 4 e viene anche salvato nel registro di pipeline IF/ID, perché potrebbe essere richiesto da un'istruzione successiva. Il calcolatore non può prevedere quale istruzione verrà prelevata dalla memoria, per cui deve preparare i dati per l'esecuzione di tutte le possibili istruzioni, trasportando lungo la pipeline tutte le informazioni potenzialmente necessarie.

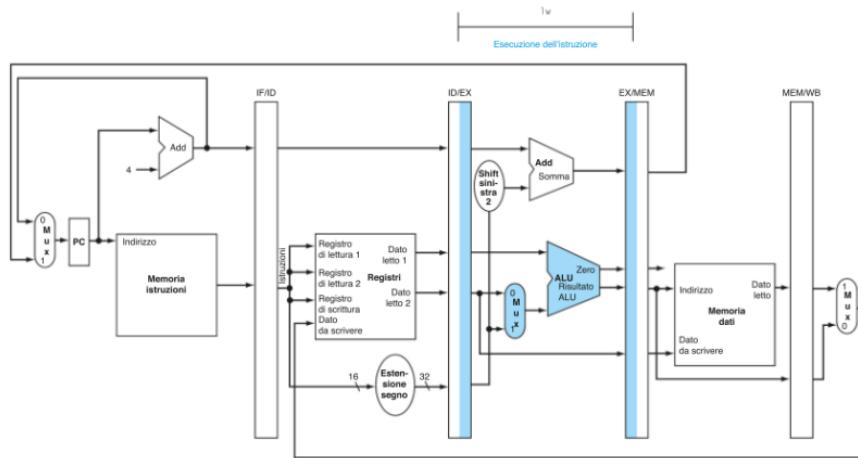


2. **Decodifica dell'istruzione e lettura del register file.** La parte inferiore della Figura 4.36 mostra che la parte contenente l'istruzione del registro di pipeline IF/ID fornisce il campo immediato di 16 bit, convertito a 32 bit mediante estensione del segno e fornisce anche il numero dei due registri da leggere. Il contenuto dei due registri e il campo immediato esteso a 32 bit vengono memorizzati nel registro di pipeline ID/EX insieme al valore del PC incrementato di 4. Anche in questo stadio viene trasferito tutto ciò che potrebbe essere necessario per l'esecuzione delle fasi successive di tutte le possibili istruzioni.



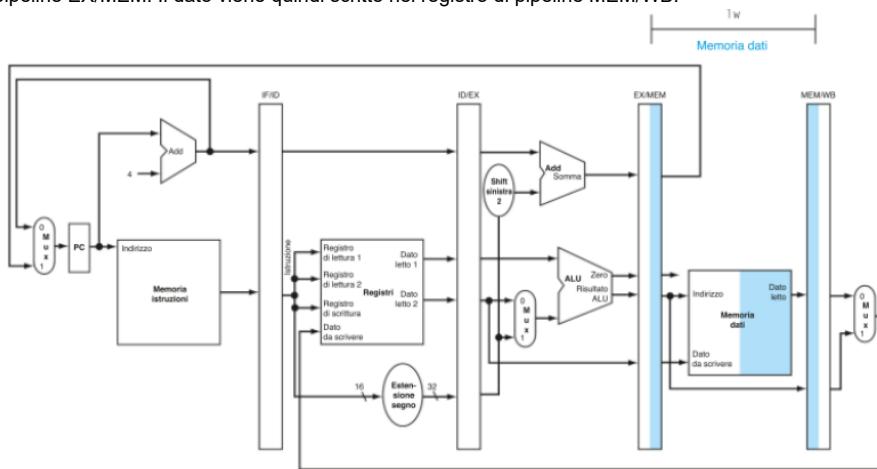
**Figura 4.36. IF e ID:** il primo e il secondo stadio della pipeline durante l'esecuzione di un'istruzione di load; le parti attive dell'unità di elaborazione mostrata in Figura 4.35 sono evidenziate in blu. La convenzione per le parti evidenziate è la stessa utilizzata in Figura 4.28. Come visto nel Paragrafo 4.2, non si può confondere la lettura con la scrittura dei registri, in quanto il loro contenuto varia solamente in corrispondenza del fronte del clock. Sebbene la load nello stadio 2 richieda solo il contenuto del primo registro, il processore non può sapere quale istruzione sta decodificando, per cui estende il segno del contenuto del campo immediato da 16 a 32 bit e legge il contenuto di entrambi i registri scrivendolo nel registro di pipeline ID/EX. Tre operandi non servono, ma la realizzazione dell'unità di controllo risulta più semplice se tutti e tre gli operandi vengono comunque scritti nel registro ID/EX.

3. **Esecuzione o calcolo dell'indirizzo.** L'istruzione di load legge dal registro di pipeline ID/EX il contenuto del registro 1 e il contenuto del campo immediato, esteso a 32 bit e dotato di segno, li somma utilizzando la ALU e scrive il risultato della somma nel registro di pipeline

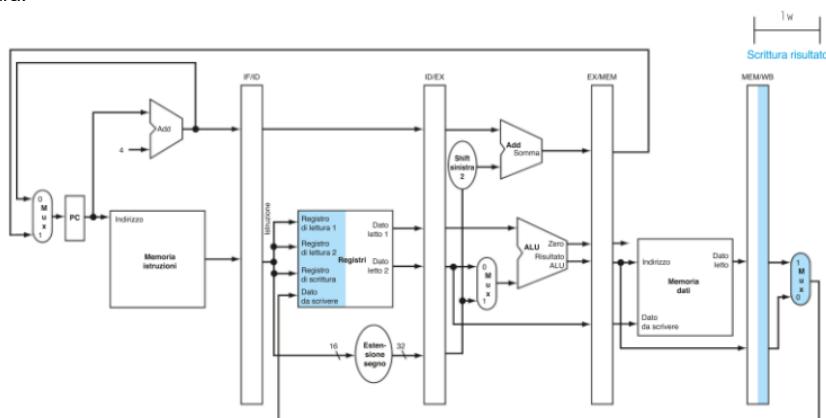


**Figura 4.37.** EX: il terzo stadio della pipeline durante l'esecuzione di un'istruzione di load: le parti attive dell'unità di elaborazione mostrata in Figura 4.35 sono evidenziate in blu. Il contenuto del registro viene sommato al contenuto del campo immediato dotato di segno, esteso a 32 bit, e il risultato della somma viene scritto nel registro di pipeline EX/MEM.

4. **Accesso alla memoria dati.** Nella figura viene mostrata l'istruzione di load mentre legge la memoria dati utilizzando come indirizzo il valore letto dal registro di pipeline EX/MEM. Il dato viene quindi scritto nel registro di pipeline MEM/WB.



5. **Scrittura del risultato.** Viene riportata la fase finale dell'esecuzione: il dato viene letto dal registro di pipeline MEM/WB e scritto nel register file posto al centro della figura.



**Figura 4.38.** MEM e WB: il quarto e il quinto stadio della pipeline durante l'esecuzione di un'istruzione di load: le parti attive dell'unità di elaborazione mostrata in Figura 4.35 sono evidenziate in blu. La memoria dati viene letta utilizzando l'indirizzo contenuto nel registro di pipeline EX/MEM e il dato letto viene scritto nel registro di pipeline MEM/WB. Poi, nello stadio WB, il dato viene letto dal registro di pipeline MEM/WB e scritto nel register file, al centro della figura. Si noti che c'è un baco in questo schema che verrà corretto nella Figura 4.41.

Questa carrellata sull'esecuzione dell'istruzione di load mostra che tutte le informazioni necessarie a uno stadio successivo della pipeline devono essere trasportate attraverso i registri di pipeline.

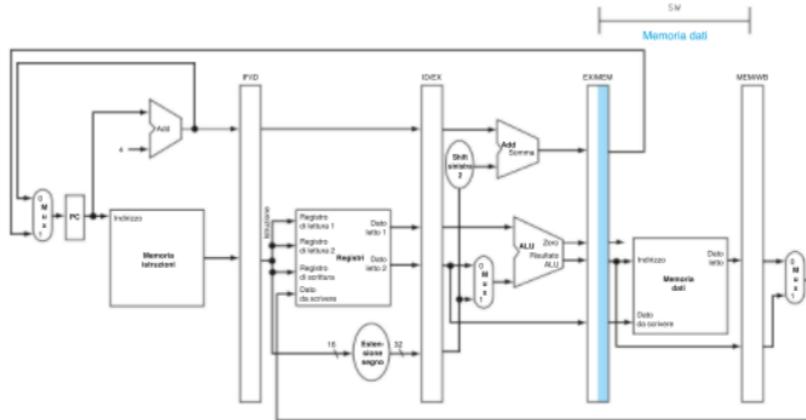
La stessa descrizione dell'esecuzione di un'istruzione di store evidenzia le analogie con la lw nell'esecuzione e nel passaggio delle informazioni agli stadi successivi.

I cinque stadi di pipeline per un'istruzione di store:

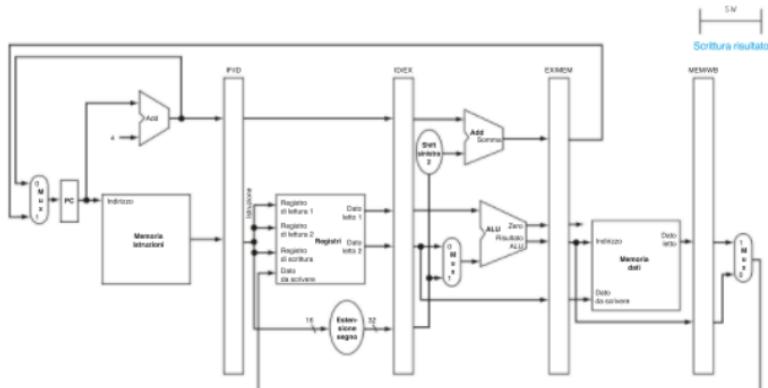
- Fetch dell'istruzione.** L'istruzione viene prelevata dalla memoria utilizzando l'indirizzo contenuto nel PC e viene quindi scritta nel registro di pipeline IF/ID. Questo stadio viene eseguito prima che l'istruzione possa essere identificata, per cui la parte superiore della figura 4.36 è valida non solo per la load, ma anche per lo store.
- Decodifica dell'istruzione e lettura del register file.** L'istruzione presente nel registro di pipeline IF/ID fornisce il numero dei due registri da leggere e il campo immediato di 16 bit dotato di segno. Il contenuto dei due registri e il campo immediato esteso a 32 bit vengono memorizzati nel registro di pipeline ID/EX, insieme al valore del PC incrementato di 4. La parte inferiore della figura 4.36 relativa all'istruzione di load

descrive il funzionamento del secondo stadio anche per le istruzioni di store. I primi due stadi funzionano allo stesso modo per tutte le istruzioni, perché è troppo presto per capire quale sia il tipo di istruzione.

3. **Esecuzione o calcolo dell'indirizzo.** A differenza del terzo stadio di esecuzione dell'istruzione di load, il contenuto del secondo registro viene scritto nel registro di pipeline EX/MEME, per essere utilizzato nello stadio successivo. Sebbene non sia sbagliato scrivere sempre il contenuto del secondo registro nel registro di pipeline EX/MEM, effettueremo questa operazione solo per le istruzioni di store, in modo da rendere più facile la comprensione della pipeline.
4. **Accesso alla memoria dati.** La parte superiore della figura 4.40 mostra in che modo avviene la scrittura del dato in memoria. Si noti che il registro contenente il dato da scrivere era stato letto nel precedente stadio di decodifica e il suo contenuto era stato memorizzato nel registro ID/EX. L'unico modo per rendere questo dato disponibile anche nello stadio MEM è trasportarlo nel registro ID/EX al registro EX/MEM durante lo stadio EX, scrivendolo in EX/MEM esattamente come viene scritto l'indirizzo della memoria.



5. **Scrittura del risultato.** La parte inferiore della Figura 4.40 riporta la fase finale dell'esecuzione dell'istruzione: si vede che nel caso della store non succede nulla. Poiché l'esecuzione di tutte le istruzioni che seguono la store dev'essere ancora completa, non c'è alcun modo per accelerarle; quindi un'istruzione deve sempre passare attraverso tutti gli stadi, anche se a volte non ci sono operazioni da compiere, poiché le istruzioni successive stanno già procedendo alla massima velocità possibile.



**Figura 4.40. MEM e WB: il quarto e il quinto stadio della pipeline durante l'esecuzione di un'istruzione di store.** Nel quarto stadio il dato viene scritto nella memoria dati; si noti che esso proviene dal registro di pipeline EX/MEM e che nulla viene modificato nel registro di pipeline MEM/WB. Una volta scritto il dato in memoria, per l'istruzione di store non rimane più nulla da fare, per cui nello stadio 5 non accade niente.

*Il funzionamento dell'istruzione di store sottolinea ancora una volta come, per trasportare un'informazione da uno stadio della pipeline a quello successivo, quell'informazione debba essere scritta in un registro di pipeline; se così non fosse, l'informazione verrebbe persa nel momento in cui l'istruzione successiva raggiungesse quella fase.*

Le istruzioni di load e di store permettono di fare una seconda considerazione: ciascun componente funzionale dell'unità di elaborazione, come la memoria istruzioni, le porte di lettura del register file, l'ALU, la memoria dati e la porta di scrittura del register file, può essere utilizzato in un solo stadio della pipeline; in caso contrario, si incorreggerebbe in un hazard strutturale.

Di conseguenza, questi componenti funzionali e i loro segnali di controllo possono essere associati ad un unico stadio della pipeline.

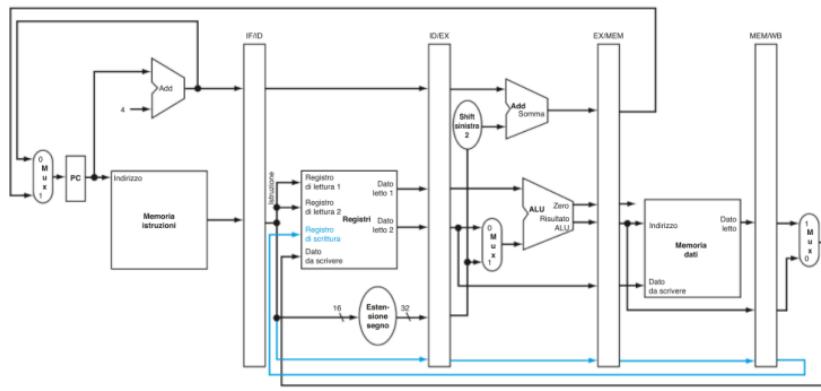
*Quale registro viene modificato nello stadio finale della load? In particolare, quale istruzione fornisce il numero del registro da scrivere?*

In questo schema, il numero del registro da scrivere è contenuto nel relativo campo dell'istruzione, presente nel registro di pipeline IF/ID. Tuttavia tale numero di registro si riferisce a un'istruzione che è successiva alla load!

Quindi, è necessario preservare il numero del registro di destinazione dell'istruzione di load: così come per l'istruzione di store avevamo trasportato il contenuto del registro *rt* dal registro di pipeline ID/EX al registro EX/MEM (per poterlo utilizzare nello stadio MEM), così per l'istruzione di load dobbiamo trasportare il numero del registro da scrivere da ID/EX, attraverso EX/MEM, fino a MEM/WB (per poterlo poi utilizzare nello stadio di WB).

Un modo alternativo di vedere questo passaggio del numero del registro da scrivere è pensare che affinché un'unità di elaborazione dotata di pipeline possa essere condivisa da più istruzioni, occorre preservare alcuni campi dell'istruzione prelevata durante lo stadio IF, in modo tale che ciascun registro di pipeline contenga la parte di istruzione che dev'essere utilizzata da quello stadio e dagli stadi successivi.

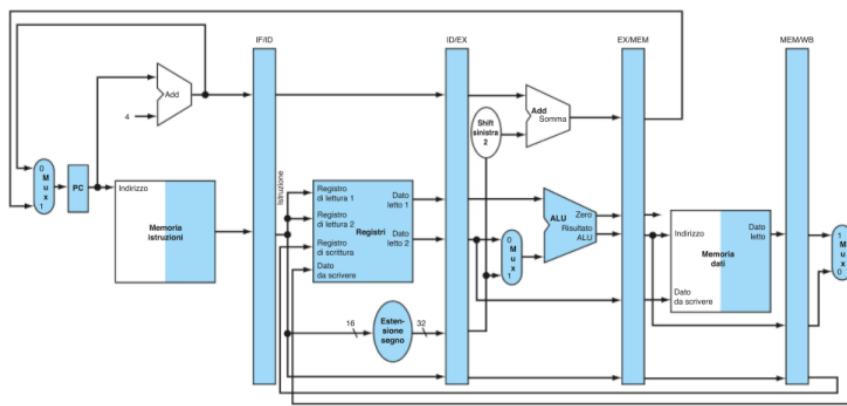
La figura:



**Figura 4.41.** L'unità di elaborazione con pipeline modificata per elaborare correttamente l'istruzione load. Il numero del registro da scrivere ora proviene dal registro di pipeline MEM/WB, così come il dato da scrivere. Il numero del registro è stato trasportato dallo stadio ID fino al registro di pipeline MEM/WB; abbiamo quindi aggiunto cinque bit agli ultimi tre registri di pipeline. Il nuovo cammino è mostrato in blu.

mostra la versione corretta dell'unità di elaborazione: il numero del registro da scrivere viene dapprima scritto nel registro di pipeline ID/EX, poi nel registro EX/MEM e infine nel registro MEM/WB; viene infine utilizzato nello stadio WB per scrivere il dato caricato dalla memoria.

La figura:



**Figura 4.42.** La parte di unità di elaborazione della Figura 4.41 utilizzata per l'esecuzione dei cinque stadi di un'istruzione load word.

mostra in un unico schema l'unità di elaborazione corretta ed evidenzia i componenti funzionali utilizzati nei cinque stadi di esecuzione dell'istruzione load word, riportati nelle Figure 4.36-4.38.

## Hazard sui dati: propagazione o stallo

### Hazard sui dati e propagazione

A questo punto, è necessario abbandonare la situazione ideale e rendersi conto di quello che può accadere durante l'esecuzione di un programma reale.

L'ultimo hazard potenziale può essere risolto progettando opportunamente l'hardware che implementa il register file: che cosa succede quando un registro viene letto e scritto nello stesso ciclo di clock? Si può assumere che la scrittura avvenga nella prima metà del ciclo di clock e la lettura nella seconda metà: in questo modo si può leggere nello stesso ciclo di clock il valore che viene scritto.

Questa situazione è tipica di molte implementazioni hardware del register file, e in questo caso non si hanno hazard sui dati.

Una notazione che assegna dei nomi ai campi dei registri di pipeline permette di formalizzare le dipendenze con maggior precisione. La prima parte del nome, a sinistra del punto, è il nome del registro di pipeline, mentre la seconda parte corrisponde al nome di un campo all'interno del registro.

Utilizzando questa notazione, le due coppie di condizioni che generano un hazard sui dati si possono esprimere come segue:

1a. EX/MEM.RegistroRd = ID/EX.RegistroRs

1b. EX/MEM.RegistroRd = ID/EX.RegistroRt

2a. MEM/WB.RegistroRd = ID/EX.RegistroRs

2b. MEM/WB.RegistroRd = ID/EX.RegistroRt

### Rilevamento delle dipendenze e propagazione

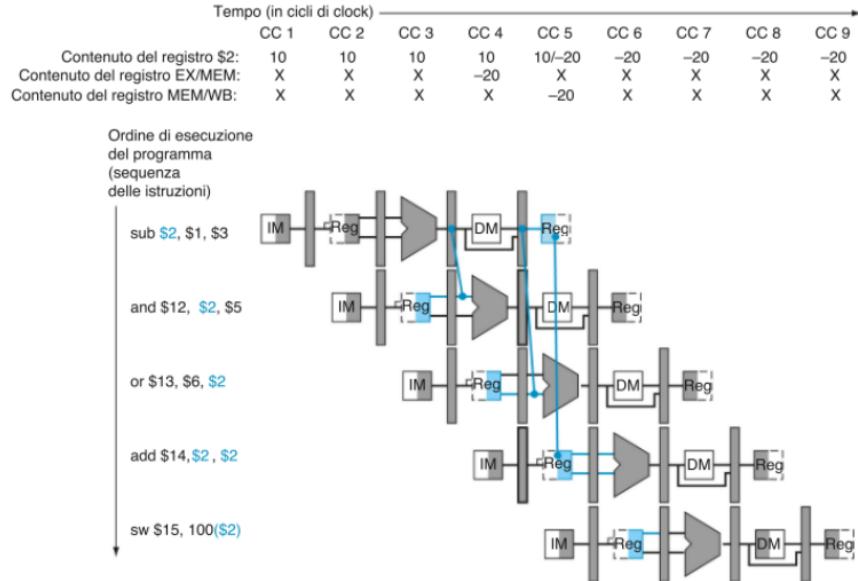
Dato che non tutte le istruzioni scrivono il risultato nel register file, questa strategia non è precisa: in alcuni casi si propagherebbero dei dati anche quando non si deve.

Una possibile soluzione consiste nel verificare se il segnale RegWrite è attivo; a tal fine, si può controllare se sia asserito il segnale RegWrite contenuto nella porzione dei registri di pipeline EX/MEM e MEM/WB riservata ai segnali di controllo attivi nello stadio di WB.

Il RISC-V richiede che il registro x0 contenga sempre il valore 0 quando viene utilizzato come operando. Nel caso in cui un'istruzione nella pipeline abbia x0 come registro destinazione, si cerca di evitare che il risultato dell'operazione, eventualmente non nullo, sia propagato in avanti; in questo modo si evita che il programmatore assembler ed il compilatore debbano considerare il registro x0 come registro destinazione.

Le condizioni riportate sopra funzionano quindi correttamente se si aggiunge la condizione EX/MEM.RegistroRd ≠ 0 alla prima condizione di hazard e MEM/WB.RegistroRd ≠ 0 alla seconda.

La Figura:



**Figura 4.53.** Le dipendenze tra i registri di pipeline si muovono in avanti nel tempo; è così possibile fornire alla ALU gli ingressi corretti per l'esecuzione dell'istruzione and e dell'istruzione or, propagando il risultato della sub presente nei registri di pipeline. Il contenuto dei registri di pipeline mostra che il dato desiderato è disponibile prima che esso venga scritto nel register file. Si suppone che il register file propaga i valori scritti al suo interno nello stesso ciclo di clock, quindi la add non deve essere messa in stallo perché riceve regolarmente i dati dal register file e non serve la propagazione dai registri di pipeline. La propagazione all'interno del register file avviene perché è possibile leggere il dato che viene scritto nello stesso ciclo di clock; questo è il motivo per cui nel quinto ciclo di clock il registro \$2 contiene il valore 10 all'inizio e -20 alla fine. Come nel resto del paragrafo, vengono gestiti tutti i meccanismi di propagazione tranne la propagazione del dato da scrivere in memoria richiesta dalle istruzioni di store.

mostra le dipendenze tra i registri di pipeline e gli ingressi dell'ALU

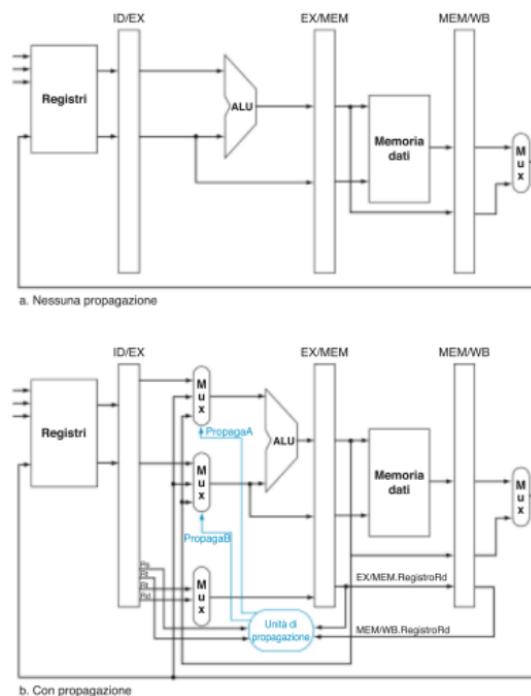
La differenza consiste nel fatto che le linee di dipendenza partono dai registri di pipeline, anziché dallo stadio WB nel quale viene scritto il register file. Queste linee di dipendenza evidenziano che i dati necessari, che devono essere propagati, sono contenuti nei registri di pipeline e sono quindi disponibili in tempo utile per le istruzioni successive.

Se possiamo prendere gli input dell'ALU non solo dal register ID/EX ma anche dagli altri registri di pipeline, allora possiamo propagare i dati corretti.

Aggiungendo dei multiplexer sugli ingressi dell'ALU e utilizzando dei segnali di controllo adeguati, è possibile far funzionare la pipeline alla velocità massima anche in presenza delle dipendenze tra i dati che abbiamo evidenziato.

Faremo inizialmente l'assunzione che la propagazione sia necessaria solamente per le quattro istruzioni del formato R: add, sub, and e or.

La Figura:



**Figura 4.54.** Nella parte superiore della figura sono mostrate la ALU e i registri di pipeline prima di aggiungere l'hardware che esegue la propagazione. Nella parte inferiore della figura sono stati inseriti dei multiplexer per la selezione dei cammini di propagazione e viene mostrata l'unità di propagazione. L'hardware aggiunto è evidenziato in blu. La figura è semplificata, in quanto non contiene alcuni dettagli dell'unità di elaborazione completa, come il circuito per l'estensione del segnale. Si noti che il segnale ID/EX.RegistroRt compare due volte: la prima volta viene collegato al multiplexer e la seconda viene collegato all'unità di propagazione (ovviamente si tratta dello stesso segnale). Come detto in precedenza, questa unità di elaborazione è in grado di gestire tutti i meccanismi di propagazione tranne la propagazione del dato da scrivere nella memoria dati richiesta dalle istruzioni di store. Osserviamo, inoltre, che questo meccanismo di propagazione funziona anche per le istruzioni di s t.

mostra un dettaglio dell'ALU e dei registri di pipeline prima e dopo l'aggiunta della propagazione.

Per determinare se propagare o meno i dati è necessario trasportare il numero dei due registri utilizzati come operandi dallo stadio ID al registro di pipeline ID/EX. E' già disponibile in ID/EX il campo Rt (bit 20-16).

Questo è diventato necessario per poter verificare se c'è un hazard sui dati: occorre quindi aggiungere Rs (bit 25-21) al registro ID/EX.

Si possono ora scrivere sia le condizioni per rilevare gli hazard sia i segnali di controllo per risolvere:

#### 1. Hazard nello stadio EX:

```
if (EX/MEM. RegWrite and (EX/MEM. RegistroRd ≠ 0) and (EX/MEM. RegistroRs = ID/EX. RegistroRs)) PropagaA = 10
if (EX/MEM. RegWrite and (EX/MEM. RegistroRd ≠ 0) and (EX/MEM. RegistroRs = ID/EX. RegistroRt)) PropagaB = 10
```

Si noti che il campo EX/MEM.RegistroRd contiene il numero del registro di scrittura sia per le istruzioni che operano sull'ALU (nel qual caso esso proviene dal campo Rd dell'istruzione stessa) sia per le istruzioni di load (nel qual caso esso proviene dal campo Rt).

In questo schema di risoluzione degli hazard il risultato che arriva dalla precedente istruzione viene propagato a uno dei due ingressi dell'ALU; se l'istruzione precedente deve scrivere sul register file e il numero del registro di scrittura coincide con quello del registro da cui viene letto il dato A o B in ingresso all'ALU (purché questo non sia il registro 0), allora il multiplexer viene pilotato in maniera da prelevare il dato dal registro di pipeline EX/MEM.

#### 2. Hazard nello stadio MEM:

```
If (MEM/WB. RegWrite and (MEM/WB. RegistroRd ≠ 0) and (MEM/WB. RegistroRs = ID/EX. RegistroRs)) PropagaA = 01 if (MEM/WB. RegWrite and (MEM/WB. RegistroRd ≠ 0) and (MEM/WB. RegistroRs = ID/EX. RegistroRt)) PropagaB = 01
```

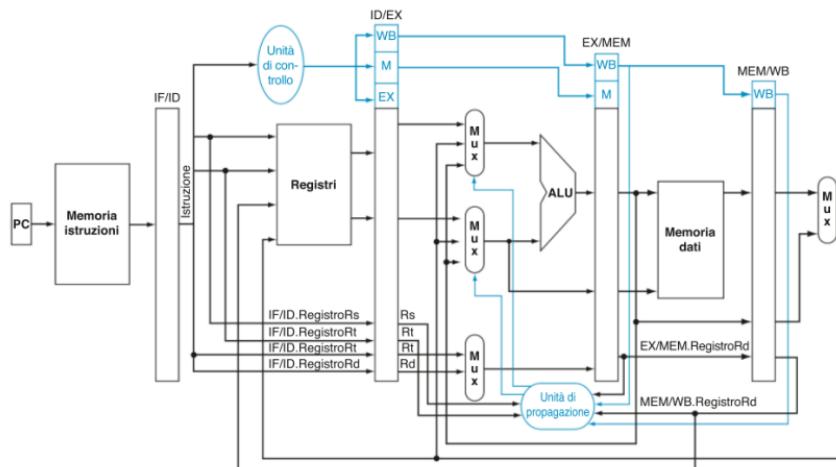
Come già accennato in precedenza, non si può verificare un hazard nello stadio WB, poiché si suppone che il register file fornisca in uscita il dato corretto anche quando l'istruzione nello stadio ID deve leggere il contenuto dello stesso registro che viene scritto in quel ciclo di clock dall'istruzione che si trova nello stadio WB.

Un register file di questo tipo implementa una particolare forma di propagazione che si verifica al suo interno. Una possibile fonte di problemi è costituita dagli hazard sui dati che si possono verificare tra il risultato di un'istruzione nello stadio WB, il risultato di un'istruzione nello stadio MEM e l'operando sorgente di un'istruzione nello stadio EX.

In questo caso il risultato viene propagato dallo stadio MEM, dal momento che quel risultato è il più recente. Quindi il test da eseguire per rilevare un hazard nello stadio MEM sarà il seguente:

```
if (MEM/WB. RegWrite and (MEM/WB. RegistroRd ≠ 0)
and not(EX/MEM. RegWrite and (EX/MEM. RegistroRd ≠ 0)
and (EX/MEM. RegistroRd ≠ ID/EX. RegistroRs)
and (MEM/WB. RegistroRd = ID/EX. RegistroRs)) PropagaA = 01
if (MEM/WB. RegWrite and (MEM/WB. RegistroRd ≠ 0)
and not(EX/MEM. RegWrite and (EX/MEM. RegistroRd ≠ 0)
and (EX/MEM. RegistroRd ≠ ID/EX. RegistroRt)
and (MEM/WB. RegistroRd = ID/EX. RegistroRt)) PropagaB = 01
```

La Figura:



**Figura 4.56.** L'unità di elaborazione dopo le modifiche apportate per risolvere gli hazard tramite la propagazione. Se si confronta questo schema con l'unità di elaborazione della Figura 4.51, si nota che le aggiunte sono costituite dai multiplexer sugli ingressi della ALU. La figura rappresenta uno schema stilizzato, in cui mancano alcuni dettagli relativi all'implementazione completa che contiene anche la logica dei salti e dell'estensione del segnale.

mostra l'hardware necessario a implementare la propagazione per le istruzioni che utilizzano nello stadio EX il risultato di istruzioni precedenti. Si noti che il campo EX/MEM.RegistroRd contiene il numero del registro di scrittura sia per le istruzioni che operano sull'ALU (nel qual caso esso proviene dal campo Rd dell'istruzione stessa) sia per le istruzioni di load (nel qual caso esso proviene dal campo Rt).

## Riepilogo Propagazione

*Il forwarding (propagazione) consiste nella propagazione in avanti dei dati.*

Esempio di codice: sub x2, x2, x3; and x12, x2, x5; or x13, x6, x2; add x14, x2, x2; sw x15, 100(x2)

Il risultato dell'istruzione sub è disponibile alla fine del ciclo di clock 3 (stadio EX) e questo succede per tutte le istruzioni di tipo R. And e or hanno bisogno del dato all'inizio dello stadio di EX, ossia nei cicli di clock 4 e 5 rispettivamente.

Si può eseguire il codice senza stalli propagando il dato non appena disponibile a qualsiasi unità che lo richieda prima che venga scritto nel registro destinazione.

Il valore richiesto dall'istruzione and è già disponibile nel registro di pipeline EX/MEM dell'istruzione sub. Analogamente al ciclo di clock successivo, l'ingresso dell'ALU per l'istruzione or si trova nel registro di pipeline MEM/WB dell'istruzione sub. E' possibile fornire all'ALU gli ingressi richiesti

dalle istruzioni and e or propagando in avanti i risultati che si trovano nei registri di pipeline, senza aspettare che siano scritti nel banco dei registri. Questa tecnica, che utilizza risultati temporanei invece di attendere che i registri siano scritti, si chiama *propagazione in avanti dei risultato (forwarding) o bypassing*.

Aggiungendo multiplexer agli ingressi dell'ALU in modo da prelevare gli ingressi da qualsiasi registro di pipeline e non solo dal registro ID/EX, la pipeline può procedere senza stalli anche in presenza di conflitti di dati.

Quando un'istruzione nel suo stadio EX deve utilizzare un dato di un registro che non è stato ancora scritto da un'istruzione precedente nella sua fase di WB è necessario portare il dato all'ingresso corretto della ALU. La propagazione avviene solo se il segnale RegWrite è asserito nello stadio che propaga (da EX/MEM in avanti).

## Hazard sui dati e stallo

Un caso nel quale la propagazione non può risolvere il problema è quello in cui un'istruzione tenta di leggere il registro che verrà scritto da un'istruzione di load che la precede.

Durante il quarto ciclo di clock il dato dev'essere ancora letto dalla memoria, ma l'ALU si appresta già ad eseguire l'operazione prevista per l'istruzione successiva. Occorre quindi mettere in stallo la pipeline quando un'istruzione di load è seguita da un'istruzione che utilizza il dato letto. E' quindi necessaria, oltre all'unità di propagazione, anche un'unità di rilevamento degli hazard che, durante lo stadio ID di un'istruzione, possa inserire uno stallo tra la lettura del dato e il suo utilizzo.

Per le istruzioni di load, l'unità di rilevamento degli hazard implementa questa condizione logica:

*if (ID/EX. MemRead and ((ID/EX. RegistroRt = IF/ID. RegistroRs) or (ID/EX. RegistroRt = IF/ID. RegistroRt))) metti in stallo la pipeline*

La prima condizione stabilisce se l'istruzione nello stadio EX è una load: questa, infatti, è l'unica istruzione che legge dati dalla memoria. Le successive due condizioni controllano se il numero del registro di scrittura della load coincide con uno dei registri sorgente dell'istruzione nello stadio ID.

Se la condizione è verificata, l'istruzione nello stadio ID viene messa in stallo per un ciclo di clock. Dopo questo stallo, la dipendenza sui dati può essere gestita dalla logica di propagazione e l'esecuzione può quindi procedere.

Se non ci fosse la propagazione, le istruzioni richiederebbero lo stallo per un secondo ciclo di clock.

Se l'istruzione nello stadio ID viene messa in stallo, anche l'istruzione nello stadio IF dev'essere messa in stallo, altrimenti l'istruzione prelevata nella fase di fetch precedente andrebbe persa.

Impedire a queste due istruzioni di procedere può essere ottenuto semplicemente impedendo la commutazione del registro PC e del registro di pipeline IF/ID.

Se il valore di questi registri viene conservato attraverso il fronte del clock, nel ciclo di clock successivo l'istruzione che viene letta nello stadio IF verrà prelevata dallo stesso indirizzo contenuto nel PC nel ciclo di clock precedente, e il numero dei registri da leggere nello stadio ID, contenuti nei campi del registro IF/ID, sarà lo stesso del ciclo di clock precedente.

Naturalmente, gli stadi a valle dello stadio ID, a partire dallo stadio EX, continuano a lavorare ma come nel caso dell'asciugatrice occorre che eseguano istruzioni che non producono effetti: queste istruzioni sono chiamate *nop (not operation)*.

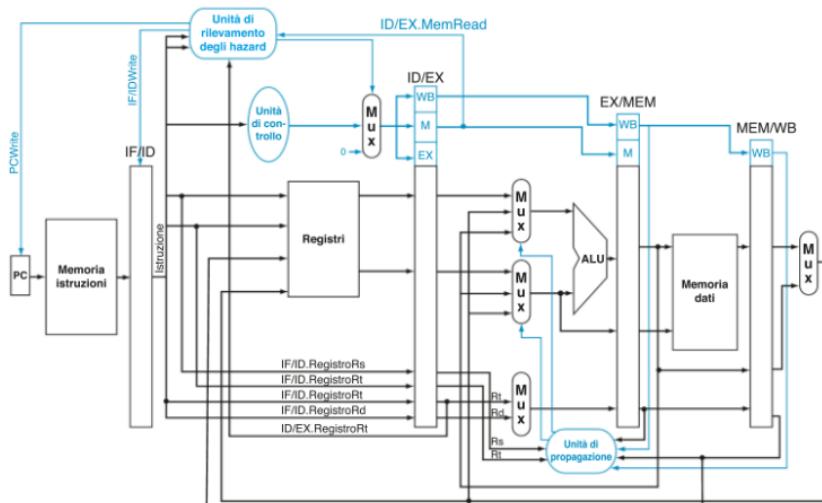
*Come possiamo inserire una nop, che si comporta come una "bolla d'aria", all'interno della pipeline?*

Impostando a 0 tutti e nove i segnali di controllo degli stadi EX, MEM e WB è possibile creare un'istruzione che non fa nulla, cioè una nop. Se identifichiamo l'hazard nello stadio ID, possiamo inserire nella pipeline una bolla mettendo a 0 i campi EX, MEM e WB del registro di pipeline ID/EX, i quali contengono i segnali di controllo; questi vengono poi trasportati in avanti nella pipeline a ogni colpo di clock, producendo l'effetto desiderato: se i segnali di controllo sono tutti a 0 non vengono scritti né i registri né la memoria dati.

Ciò che avviene effettivamente nell'hardware:

- l'esecuzione dell'istruzione *and* nello stadio EX della pipeline è trasformata nell'esecuzione di una nop e l'esecuzione di tutte le istruzioni successive a quella di load, and compresa, viene ritardata di un ciclo di clock
- come una bolla d'aria in un tubo pieno d'acqua, la bolla associata allo stallo ritarda tutto ciò che la segue e precede stadio dopo stadio fino a giungere alla fine della pipeline
- uno stallo implica la ripetizione di una certa quantità di lavoro, anche se il suo effetto è di allungare il tempo di esecuzione delle istruzioni *and* e *or* e di ritardare il caricamento dell'istruzione *and*

La Figura:



**Figura 4.60.** Panoramica sul controllo della pipeline: nella figura compaiono i due multiplexer per la propagazione, l'unità di rilevamento degli hazard e l'unità di propagazione. Sebbene gli stadi ID e EX siano stati semplificati (mancano la logica per l'estensione del segno del campo immediato e la logica dei salti), lo schema mostra gli elementi essenziali dell'hardware per la propagazione.

mette in evidenza le connessioni dell'unità di rilevamento degli hazard e dell'unità di propagazione all'interno della pipeline. Come in precedenza, l'unità di propagazione controlla i multiplexer all'ingresso dell'ALU per sostituire eventualmente il valore proveniente da uno dei registri del register file con il contenuto dell'opportuno registro della pipeline. L'unità di rilevamento degli hazard controlla la scrittura del PC e del registro IF/ID oltre al multiplexer che sceglie se scrivere nel registro ID/EX 0 oppure i segnali di controllo dell'istruzione in fase di decodifica.

L'unità di rilevamento degli hazard mette in stallo la pipeline e imposta a 0 i segnali di controllo se si verifica la condizione di hazard per l'utilizzo del dato di una load riportata in precedenza.

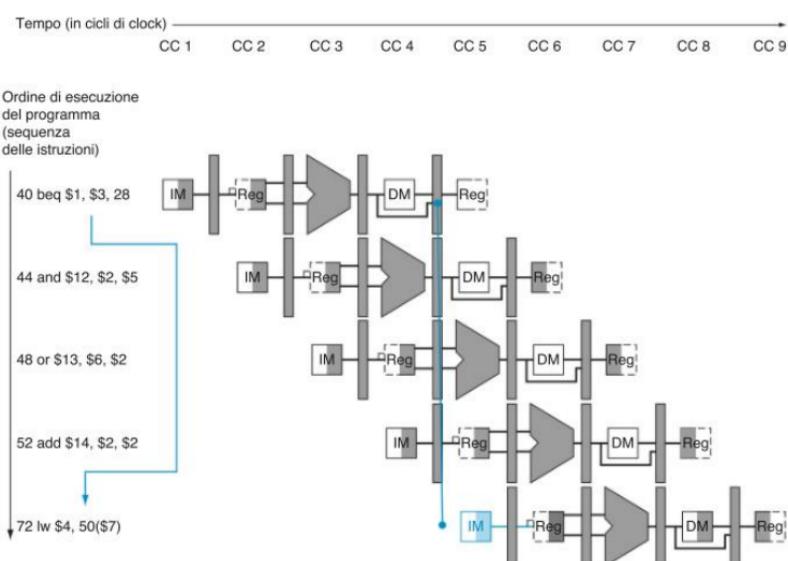
Sebbene il compilatore generalmente si basi sull'hardware per risolvere gli hazard e per assicurare l'esecuzione corretta del codice, esso deve comprendere il funzionamento della pipeline per ottenere prestazioni ottimali; altrimenti potrebbero verificarsi stalli inaspettati che riducono le prestazioni del codice compilato.

## Hazard sul controllo

Finora ci siamo preoccupati soltanto degli hazard che coinvolgono le operazioni aritmetiche e di trasferimento dati.

Tuttavia, ci sono anche hazard della pipeline che riguardano i salti condizionati.

La Figura:



**Figura 4.61.** Impatto della pipeline sulle istruzioni di salto condizionato. I numeri alla sinistra di ciascuna istruzione (40, 44,...) rappresentano l'indirizzo delle istruzioni. Dato che l'istruzione di salto condizionato decide se occorre saltare nello stadio MEM, corrispondente al quarto ciclo di clock dell'istruzione, le tre istruzioni che la seguono vengono comunque prelevate dalla memoria istruzioni e viene avviata la loro esecuzione. Se non si intervenisse, l'esecuzione di queste tre istruzioni inizierebbe prima che il programma salti eventualmente alla 1w contenuta all'indirizzo 72. Nella Figura 4.31 si supponeva che esistesse dell'hardware aggiuntivo per ridurre l'hazard sul controllo a un solo ciclo di clock, mentre in questa figura viene rappresentata l'unità di elaborazione non ottimizzata.

mostra una sequenza di istruzioni e indica in quale punto della pipeline verrebbe eseguito il salto.

Per alimentare la pipeline è necessario prelevare un'istruzione a ogni ciclo di clock, ma nell'implementazione hardware vista finora le decisioni relative ai salti condizionati non vengono prese prima dello stadio MEM della pipeline.

Tale ritardo nella determinazione dell'istruzione successiva da prelevare è chiamato hazard sul controllo o hazard sui salti condizionati, per distinguerlo dagli hazard sui dati appena esaminati.

Gli hazard sul controllo sono relativamente semplici da comprendere, si verificano molto meno frequentemente e non c'è nulla di veramente efficace per risolverli.

Le modifiche circuitali sono quindi più semplici.

## Ipotizzare che il salto condizionato non sia eseguito

Mettere in stallo la pipeline fino a quando l'esecuzione dell'istruzione di salto condizionato non è stata completata fa perdere troppo tempo. Una tecnica frequentemente utilizzata per risolvere il problema consiste nel predire che il salto non debba essere eseguito, continuando quindi a eseguire le istruzioni secondo il loro normale flusso.

Se poi risultasse che il salto doveva essere eseguito, le istruzioni che si trovano nella fase di fetch, di decodifica e di calcolo devono essere scartate e l'esecuzione deve riprendere a partire dall'istruzione contenuta all'indirizzo di destinazione del salto. Se i salti non devono essere eseguiti anche solo nella metà dei casi e se costa poco scartare le istruzioni, questa ottimizzazione permette di dimezzare il costo degli hazard sul controllo.

*Per scartare le istruzioni bisogna semplicemente cambiare il valore dei segnali di controllo, forzandoli a 0, analogamente a quanto si è fatto per ottenere lo stallo nel caso degli hazard per l'utilizzo del dato di una load.*

La differenza consiste nel fatto che, quando l'istruzione di salto condizionato raggiunge lo stadio MEM, è necessario anche eliminare le tre istruzioni presenti negli stadi ID, ID ed EX. Se per le istruzioni di stallo legate all'utilizzo del dato di una load era sufficiente mettere a 0 il valore dei segnali di controllo nello stadio ID, lasciando che questi si propagassero lungo la pipeline, ora bisogna eliminare le istruzioni presenti negli stadi IF, ID ed Ex della pipeline; in termini tecnici occorre effettuare un *flush (svuotamento)* di queste istruzioni.

## Ridurre i ritardi associati ai salti condizionati

Un modo per migliorare le prestazioni sui salti condizionati consiste nel ridurre il costo nel caso in cui il salto venga erroneamente eseguito.

Finora abbiamo supposto (per i salti condizionati) che l'indirizzo successivo da caricare nel PC venga preso dallo stadio MEM; ma se anticipassimo la decisione sul salto si potrebbe scartare un numero inferiore di istruzioni.

*Anticipare la decisione su un salto condizionato richiede che due azioni vengano eseguite anticipatamente:*

1. *calcolare l'indirizzo di destinazione del salto*
2. *valutare la condizione logica sulla base della quale il salto condizionato viene effettuato o meno*

La parte facilmente realizzabile di questa modifica è l'anticipazione del calcolo dell'indirizzo di destinazione del salto; il contenuto del PC e il campo immediato sono già disponibili nel registro IF/ID della pipeline ed è quindi sufficiente *spostare il sommatore che calcola l'indirizzo di salto dallo stadio EX allo stadio ID*; naturalmente, il calcolo dell'indirizzo di destinazione del salto sarà effettuato per tutte le istruzioni, ma verrà utilizzato solamente quando richiesto.

La parte più difficile consiste invece nella decisione sul salto. Se il salto è del tipo branch equal, si deve confrontare il contenuto dei due registri letti durante lo stadio ID e verificare se il loro contenuto è uguale.

Il test di uguaglianza può essere eseguito mettendo innanzitutto in OR esclusivo i bit corrispondenti dei due operandi e inserendo poi in un NOR l'uscita degli OR.

*Spostare il confronto nello stadio ID implica l'aggiunta di altra logica di propagazione e di rilevazione degli hazard; dato che la decisione sul salto dipende dai dati contenuti all'interno della pipeline, dovremo assicurare che il branch funzioni correttamente anche con questa ottimizzazione.*

Ci sono due elementi che complicano le cose:

3. Durante lo stadio ID dobbiamo decodificare l'istruzione, decidere se occorre propagare uno dei due operandi all'ingresso del comparatore e valutare l'uguaglianza, in modo che se l'istruzione richiede di saltare si possa scrivere nel PC l'indirizzo di destinazione del salto. Finora la propagazione degli operandi veniva gestita dall'unità di propagazione anche per i salti condizionati, ma l'introduzione del test di uguaglianza nello stadio ID richiede della logica di propagazione aggiuntiva. Si noti che gli operandi di un'istruzione di salto condizionato possono eventualmente essere prelevati dai registri EX/MEM o MEM/WB della pipeline
4. Poiché i dati su cui fare il confronto sono richiesti già nello stadio ID, ma possono essere prodotti più tardi nella pipeline, è possibile che si verifichi un hazard sui dati e che la pipeline debba essere messa in stallo. Per esempio, se un'istruzione che opera sulla ALU si trova subito prima del salto e produce uno degli operandi per il confronto, ci sarà bisogno di uno stallo, poiché lo stadio EX dell'istruzione che opera sulla ALU è successivo allo stadio ID dell'istruzione di salto condizionato. Di conseguenza, se una load è seguita da un'istruzione di salto condizionato e il dato letto dalla memoria è uno dei due operandi, viene richiesto uno stallo di due cicli di clock, dato che il risultato dell'istruzione di load compare nella pipeline solamente alla fine dello stadio MEM, ma viene richiesto all'inizio dello stadio ID dell'istruzione di salto

Malgrado queste difficoltà, *spostare l'esecuzione di un salto condizionato nello stadio ID rappresenta un miglioramento, perché riduce la penalità di un'istruzione di salto condizionato a una sola istruzione da scartare (cioè quella che si trova nella fase di fetch) nel caso in cui il salto debba essere eseguito.*

Per eliminare l'istruzione in esecuzione dallo stadio IF viene aggiunto un segnale di controllo, denominato IF.flush, che azzerza la parte del registro di pipeline IF/ID che contiene l'istruzione. Azzerando questa parte del registro si trasforma l'istruzione appena caricata in una nop, ossia in un'istruzione che non esegue alcuna operazione e non modifica quindi lo stato del sistema.

## Predizione dinamica dei salti

Quando si suppone che un salto condizionato non debba essere eseguito si realizza una forma rossa di predizione del salto; nel caso in cui la supposizione si riveli errata, occorre eliminare le istruzioni contenute nella pipeline.

Per una semplice pipeline a cinque stadi questo approccio, probabilmente accoppiato a qualche tecnica di predizione utilizzata dal compilatore, risulta probabilmente adeguato.

Con pipeline più grandi il costo dei salti condizionati, misurato in numero di cicli di clock, aumenta.

In maniera simile nei processori a esecuzione parallela il costo dei salti condizionati cresce, in termini di numero di istruzioni perse. Ciò significa che in una pipeline "aggressiva" questo semplice schema di predizione statica causerebbe una perdita di prestazioni eccessiva.

Con più hardware a disposizione è possibile cercare di prevedere il comportamento dei salti durante l'esecuzione stessa del programma.

Un possibile approccio consiste nel *verificare se l'ultima volta che un'istruzione di salto è stata eseguita, il salto sia stato effettivamente eseguito: in caso positivo vengono caricate le istruzioni a partire da quella presente all'indirizzo di destinazione del salto*. Questa tecnica viene chiamata

*predizione dinamica dei salti.*

Un modo per implementare questa strategia consiste nell'utilizzare un *buffer di predizione dei salti*, detto anche *tavella della storia dei salti*.

*Un buffer di predizione dei salti è una piccola memoria indicizzata attraverso la parte inferiore dell'indirizzo dell'istruzione di salto: questa memoria contiene un bit che indica se il salto era stato eseguito o meno durante l'ultima esecuzione.*

Questo è tipo di buffer più semplice: non possiamo sapere se la predizione sia corretta o meno; per esempio, la predizione potrebbe riguardare un'altra istruzione di salto condizionato la cui parte inferiore dell'indirizzo coincide con quella dell'istruzione di salto che stiamo analizzando.

Tuttavia, un errore sulla predizione non influenza la correttezza dell'esecuzione: la predizione è solamente un suggerimento che si spera risulti corretto, su cui è basato il prelevamento dell'istruzione successiva.

Se il suggerimento si rivela sbagliato, le istruzioni caricate erroneamente saranno eliminate, verrà invertito il bit di predizione e l'esecuzione ripartirà dal prelevamento dell'istruzione corretta.

Questo schema di predizione ad 1 bit ha un inconveniente: anche se un salto venisse quasi sempre effettuato, la predizione risulterebbe sbagliata almeno due volte invece che una volta sola, quando il salto non viene effettuato.

## Cicli e predizioni

Si consideri un salto condizionato all'interno di un ciclo e si supponga che il salto venga eseguito nove volte di seguito e poi alla decima volta non venga eseguito.

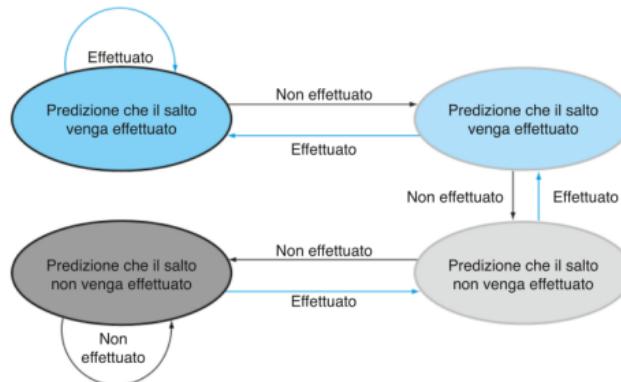
*Qual è l'accuratezza fornita dalla previsione per questo salto, assumendo che il bit di previsione nel buffer di predizione non venga alterato dalle altre istruzioni?*

A regime la previsione non risulterà corretta nella prima e nell'ultima iterazione del ciclo. L'errore sull'ultima iterazione è inevitabile, in quanto la previsione ci dirà che il salto dev'essere eseguito, essendo stato eseguito nove volte di seguito fino ad allora.

L'errore sulla prima iterazione si verifica perché il bit di predizione commuta in occasione della precedente esecuzione dell'ultima iterazione del ciclo, quando il salto non era stato eseguito. Di conseguenza, l'accuratezza della previsione per quel salto è pari all'80% (due previsioni sbagliate ed otto corrette).

In linea di principio si vorrebbe che per questi salti altamente regolari l'accuratezza della predizione corrispondesse alla frequenza di esecuzione del salto. Per rimediare a questa inadeguatezza, si utilizzano spesso degli schemi di previsione a 2 bit. In uno schema a 2 bit la predizione deve risultare sbagliata due volte di seguito prima di essere modificata.

La Figura:



**Figura 4.63.** Gli stati possibili in uno schema di predizioni su 2 bit. Utilizzando 2 bit anziché 1 bit, una branch che abbia una forte tendenza a eseguire il salto o a non eseguirlo, cosa che accade molto frequentemente, avrà una previsione errata soltanto una volta. I 2 bit sono richiesti per codificare i 4 stati della macchina a stati finiti. Lo schema di predizioni su 2 bit è un esempio di previsione basata sui contatori. Questi vengono incrementati quando la previsione si rivela accurata e decrementati in caso contrario; ai contatori vengono associati dei predittori, che utilizzano il valore medio di conteggio per decidere se il salto dovrà essere effettuato o meno.

mostra la macchina a stati finiti che implementa questo schema.

*Un buffer di predizione dei salti può essere realizzato come un piccolo buffer speciale, accessibile tramite l'indirizzo dell'istruzione durante lo stadio IF della pipeline.*

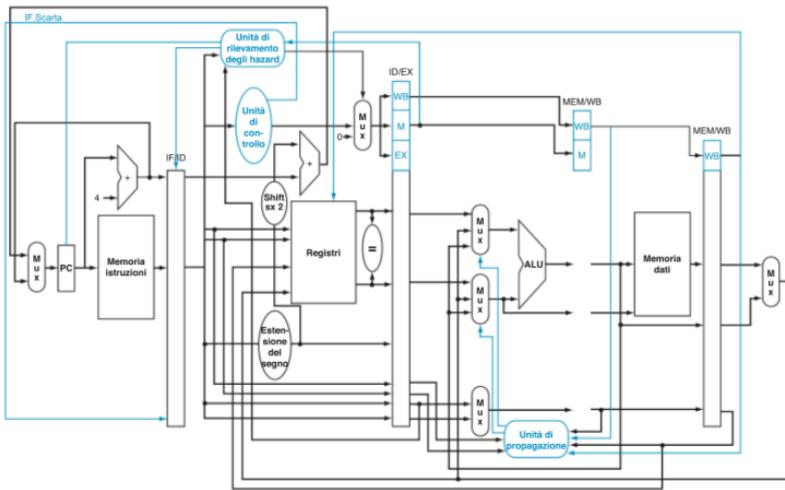
*Se si prevede che il salto debba essere eseguito, il caricamento delle istruzioni successive sarà effettuato a partire dall'indirizzo di destinazione del salto non appena questo viene scritto nel PC; questo può avvenire già nello stadio ID.*

*In caso contrario, il prelevamento e l'esecuzione delle istruzioni continueranno in modo sequenziale. Se la previsione si rivela sbagliata, i bit di previsione saranno modificati.*

## Riepilogo sulla pipeline

Siamo partiti dal bucato, mostrando i principi della pipeline applicati a un'attività comune della vita quotidiana. Utilizzando questo esempio abbiamo spiegato l'esecuzione in pipeline, passo dopo passo, a partire dall'unità di elaborazione a singolo ciclo a cui abbiamo aggiunto via via i registri di pipeline, i cammini di propagazione, le unità di rilevamento delle criticità, la predizione dei salti e l'eliminazione delle istruzioni richiesta quando si verificano eccezioni.

La Figura:



**Figura 4.65.** L'unità di elaborazione finale e la relativa unità di controllo della pipeline vista in questo capitolo. Si noti che questo è uno schema stilizzato e l'unità di elaborazione non è dettagliata; per esempio, manca il multiplexer pilotato dal segnale di controllo ALUSrc presente nella Figura 4.57 e i segnali di controllo di Figura 4.51.

mostra l'unità di elaborazione finale con la relativa unità di controllo.

## Le eccezioni

L'unità di controllo di un processore è la parte più difficile da far funzionare correttamente ed è anche la più difficile da rendere veloce. Una delle sfide maggiori è rappresentata dalla gestione delle eccezioni e degli interrupt, eventi distinti dai salti condizionati e incondizionati, che alterano il normale flusso sequenziale di esecuzione delle istruzioni.

Inizialmente erano stati concepiti per gestire eventi inaspettati che si verificavano all'interno del processore, come l'overflow aritmetico; lo stesso meccanismo di base fu poi esteso alla comunicazione tra i dispositivi di I/O e il processore.

Molte architetture e molti ricercatori non fanno distinzione tra interrupt ed eccezioni, utilizzando spesso il termine più vecchio di interrupt per indicare entrambi i tipi di eventi.

Con il termine **eccezione** (interrupt) ci si riferisce a *un qualsiasi cambiamento non previsto del flusso di controllo, senza distinguere se questo abbia cause esterne o interne al processore*; utilizzeremo il termine *interrupt* per riferirci solo agli eventi che hanno cause esterne.

Molti requisiti per il supporto delle eccezioni provengono dal contesto specifico che le causa.

Il rilevamento di condizioni eccezionali e la scelta dell'azione opportuna si trovano spesso sul cammino critico di un processore, che determina il periodo del clock e quindi le prestazioni.

Occorre porre particolare attenzione alle eccezioni durante la progettazione dell'unità di controllo, perché aggiungere a posteriori la gestione delle eccezioni a un'implementazione hardware complessa può comportare una significativa riduzione delle prestazioni; inoltre, diventerebbe più difficile realizzare un'implementazione hardware che funzioni correttamente.

## La gestione delle eccezioni nelle architetture RISC-V

I soli tipi di eccezione che possono essere generati nell'implementazione hardware vista finora sono:

1. l'esecuzione di un'istruzione non valida
2. i malfunzionamenti hardware

L'operazione fondamentale che il processore deve compiere al verificarsi di un'eccezione consiste nel salvare l'indirizzo dell'istruzione che l'ha generata nel registro *cause* del supervisore delle eccezioni (SEPC, supervision exception cause manager) e trasferire il controllo a un indirizzo specifico del sistema operativo.

Il sistema operativo potrà quindi intraprendere le azioni più opportune, che potrebbero consistere nel fornire alcuni servizi al programma utente, eseguire un'azione predeterminata in risposta al malfunzionamento o terminare l'esecuzione del programma segnalando un errore.

Dopo aver compiuto le azioni necessarie per rispondere all'eccezione, il sistema operativo può terminare il programma o riprenderne l'esecuzione utilizzando il SEPC per determinare il punto da cui ripartire.

Per poter gestire correttamente un'eccezione, il sistema operativo deve conoscerne la causa, oltre a sapere quale istruzione l'ha generata.

Ci sono due metodi per comunicare la causa di un'eccezione:

1. il metodo utilizzato nell'architettura RISC-V è quello di prevedere un registro dedicato (detto registro *cause* di supervisione dell'eccezione, SCAUSE) contenente un campo che indica la causa dell'eccezione
2. adottare *interrupt vettorizzati*, nei quali l'indirizzo a cui si deve trasferire il controllo viene determinato dalla causa dell'eccezione stessa, che fornisce un numero che viene sommato eventualmente a un registro base che punta a un'area di memoria contenente le istruzioni di risposta agli interrupt vettorizzati

Il sistema operativo capisce il motivo dell'eccezione dall'indirizzo da cui inizia la risposta all'eccezione. Gli indirizzi sono separati da 32 byte, equivalenti a 8 istruzioni, con le quali il sistema operativo deve registrare il motivo dell'eccezione ed eseguire qualche semplice elaborazione. Quando l'eccezione non è vettorizzata, il sistema operativo inizia a rispondere a tutte le eccezioni dallo stesso indirizzo e deve decodificare il registro *cause* per capire che cosa abbia generato l'eccezione.

Si possono implementare le operazioni richieste per la gestione delle eccezioni aggiungendo all'implementazione base già vista alcuni registri e alcuni segnali di controllo ed estendendo leggermente l'unità di controllo.

Supponiamo di implementare il metodo di gestione delle eccezioni con un singolo indirizzo di risposta, l'indirizzo

0000 0000 1C09 0000<sub>esa</sub>

Occorrerà introdurre due registri aggiuntivi alla nostra unità di elaborazione RISC-V:

- **SEPC**: è un registro a 64 bit utilizzato per memorizzare l'indirizzo dell'istruzione che ha generato l'eccezione; questo registro serve anche nel caso di eccezioni vettoriali
- **SCAUSA**: un registro utilizzato per memorizzare la causa dell'eccezione. Nell'architettura RISC-V tale registro è a 64 bit, sebbene alcuni bit siano, ad oggi, inutilizzati

## Le eccezioni e loro gestione nella pipeline

In un'unità di elaborazione dotata di pipeline le eccezioni vengono trattate come se fossero una forma di hazard sul controllo.

Per esempio, supponiamo che si verifichi un overflow aritmetico in un'istruzione di somma, dobbiamo eliminare dalla pipeline le istruzioni che seguono l'add e iniziare a prelevare le istruzioni dal nuovo indirizzo, che è quello di risposta all'eccezione.

Utilizzeremo quindi lo stesso meccanismo dei salti condizionati, con la differenza che in questo caso i segnali di controllo devono essere deasserriti. Quando dovevamo gestire gli errori sulle predizioni dei salti, abbiamo visto come eliminare un'istruzione nello stadio IF convertendola in una nop. Per eliminare un'istruzione che si trovava nello stadio ID, invece, utilizziamo il multiplexer già presente che mette a 0 i segnali di controllo, realizzando così di fatto lo stallo della pipeline.

Un nuovo segnale di controllo, chiamato ID.Flush, viene messo in OR con il segnale di stallo che proviene dall'unità di rilevamento degli hazard per eliminare l'istruzione nello stadio ID.

Per eliminare un'istruzione nello stadio EX, viene utilizzato un nuovo segnale di controllo, chiamato EX.Flush, che pilota un nuovo multiplexer utilizzato per mettere a 0 i segnali di controllo di questo stadio.

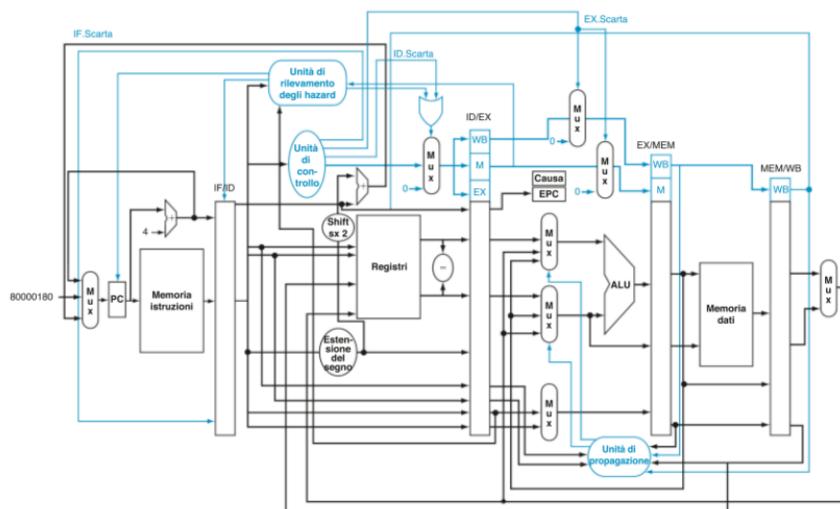
Per prelevare l'istruzione successiva dall'indirizzo

0000 0000 1C09 0000<sub>esa</sub>

che nel RISC-V contiene l'indirizzo della prima istruzione della procedura del sistema operativo di risposta alle eccezioni, occorre semplicemente aggiungere un'altra linea in ingresso, che porta il valore

0000 0000 1C09 0000<sub>esa</sub>

al multiplexer che seleziona il nuovo valore del PC.



**Figura 4.66.** L'unità di elaborazione in grado di gestire le eccezioni, con i relativi segnali di controllo. I cambiamenti più importanti sono l'aggiunta di un nuovo ingresso con valore 8000 0180<sub>esa</sub> al multiplexer che fornisce il nuovo indirizzo al PC, di un registro causa che memorizza la causa dell'eccezione e di un registro EPC per salvare l'indirizzo dell'istruzione che ha provocato l'eccezione. L'ingresso 8000 0180<sub>esa</sub> del multiplexer rappresenta l'indirizzo della procedura del sistema operativo di risposta alle eccezioni. Il segnale di overflow della ALU è uno degli ingressi dell'unità di controllo, anche se in questo schema non viene mostrato.

Questo esempio mostra un problema nella gestione delle eccezioni: se non si interrompe l'istruzione prima della fine della sua esecuzione, il programmatore non sarà più in grado di vedere il valore originale del registro x1, dato che tale registro sarà sporco, poiché è anche il registro destinazione della stessa istruzione add che ha generato l'overflow.

Tuttavia, se supponiamo che l'eccezione venga riconosciuta nello stadio EX si può utilizzare il segnale EX.Flush per prevenire che l'istruzione ora nello stadio Ex scriva poi il registro destinazione nello stadio WB.

Molte eccezioni richiedono di completare normalmente l'esecuzione dell'istruzione che ha causato l'eccezione; il modo più semplice per fare ciò è eliminare l'istruzione e farla eventualmente ripartire dall'inizio dopo che l'eccezione è stata gestita.

L'ultimo passo consiste nel salvare l'indirizzo dell'istruzione che ha causato l'eccezione nel *program counter del supervisore delle eccezioni* (SEPC).

Con cinque istruzioni attive a ogni ciclo di clock, la sfida è associare ciascuna eccezione all'istruzione corretta; inoltre, esiste la possibilità che nello stesso ciclo di clock più eccezioni si verifichino simultaneamente.

*La soluzione utilizzata consiste nell'associare una priorità alle eccezioni, in modo tale che sia semplice determinare quale eccezione debba essere gestita per prima.*

Nella maggior parte delle implementazioni RISC-V, l'hardware ordina le eccezioni in modo da interrompere la prima istruzione che ha generato un'eccezione.

Le richieste da parte dei dispositivi di I/O e i malfunzionamenti hardware non sono associati a istruzioni specifiche e quindi, in questi casi, è consentita una maggiore flessibilità su quando interrompere la pipeline.

Ne consegue che il meccanismo utilizzato per le eccezioni funziona bene anche in questi casi.

*Il registro SEPC mantiene l'indirizzo delle istruzioni interrotte e il registro SCAUSA, quando si verificano più eccezioni in uno stesso ciclo di clock, memorizza l'eccezione a priorità più elevata.*

## Parallelismo a livello di istruzioni

*La pipeline strutta il parallelismo potenziale esistente tra le istruzioni.*

Questa forma di parallelismo viene chiamata Parallelismo a Livello di Istruzioni (ILP, Instruction-Level Parallelism).

Ci sono due metodi principali per incrementare potenzialmente la quantità di parallelismo a livello di istruzioni:

1. *aumentare la profondità della pipeline per sovrapporre più istruzioni.* La pipeline passerebbe in questo caso da 4 a 6 stadi e, per ottenere il massimo aumento di velocità, si renderebbe necessario bilanciare nuovamente i restanti stadi in modo tale che abbiano la stessa durata. La quantità di parallelismo è più alta, poiché più istruzioni vengono sovrapposte nel tempo e le prestazioni sono potenzialmente migliori (dato che il ciclo di clock può essere ridotto)
2. *replicare i componenti interni del calcolatore in modo tale che sia possibile lanciare l'esecuzione di più istruzioni in ogni stadio della pipeline.* Il nome generico che si dà a questa tecnica è *esecuzione parallela (multiple issue)*. Lo svantaggio di questo approccio consiste nel lavoro addizionale necessario a mantenere impegnate tutte le macchine e a trasferire i carichi di buco da uno stadio della pipeline al successivo. Lanciare in esecuzione più istruzioni nello stesso stadio fa sì che il tasso di esecuzione delle istruzioni sia maggiore della frequenza di clock o, in altri termini, che il CPI sia minore di 1. In alcuni casi è utile invertire l'unità di misura e utilizzare il numero di istruzioni per ciclo di clock, anche detto IPC

Tuttavia, ci sono molti vincoli sulle istruzioni che possono essere eseguite in parallelo e su che cosa fare quando si verificano delle dipendenze.

Esistono due modalità principali per realizzare processori a esecuzione parallela, la cui principale differenza consiste nel modo in cui viene suddiviso il lavoro tra il compilatore e l'hardware: le decisioni possono essere prese staticamente (cioè durante la compilazione) o dinamicamente (cioè durante l'esecuzione).

Questi due approcci sono detti *parallelizzazione statica dell'esecuzione* (static multiple issue) e *parallelizzazione dinamica dell'esecuzione* (dynamic multiple issue).

Ci sono due compiti principali, ben distinti, che una pipeline a esecuzione parallela dev'essere in grado di eseguire:

1. *impacchettare le istruzioni in slot di esecuzione (issue slot).* Come fa a sapere il processore quali e quante istruzioni possono essere lanciate in esecuzione in un dato ciclo di clock? Nella maggior parte dei processori dotati di parallelizzazione statica, questo processo viene parzialmente gestito dal compilatore. Nelle architetture dotate di parallelizzazione dinamica, in genere, è il processore a decidere durante l'esecuzione stessa, anche se il compilatore spesso cerca comunque di ottimizzare l'ordine delle istruzioni per aumentare la velocità di esecuzione
2. *gestire gli hazard sui dati e sul controllo.* Nei processori dotati di parallelizzazione statica alcune o tutte le conseguenze degli hazard sui dati e sul controllo vengono gestite staticamente dal compilatore, mentre la maggior parte dei processori dotati di parallelizzazione dinamica tenta di attenuarne gli effetti (almeno di alcuni tipi di hazard) utilizzando tecniche hardware che operano durante l'esecuzione stessa

Benché questi due approcci vengano qui descritti come distinti, in realtà le tecniche di parallelizzazione sono spesso ibride, perché ciascun approccio prende in prestito alcune delle tecniche tipiche dell'altra modalità; in altri termini, *nessun approccio alla parallelizzazione è solamente statico o solamente dinamico.*

## Concetto di speculazione

*Uno dei metodi più importanti per scoprire e sfruttare al massimo l'ILP (Parallelismo a Livello di Istruzioni).*

La speculazione è basata sulla grande idea della predizione ed è un approccio che permette al compilatore o al processore di "indovinare" le caratteristiche di un'istruzione in modo da permettere l'inizio dell'esecuzione di altre istruzioni che dipendono da quell'istruzione.

Per esempio, si può speculare sul risultato del confronto associato ad un salto condizionato in modo tale che l'esecuzione delle istruzioni successive possa essere anticipata. Oppure, si può speculare se una store, posta subito prima di una load, operi su un indirizzo di memoria differente, il che permetterebbe alla load di essere eseguita prima dello store, eliminando il possibile hazard dovuto all'utilizzo del dato caricato nella CPU dalla load.

Il problema è che la speculazione può rilevarsi sbagliata. Perciò, ogni meccanismo di speculazione deve comprendere anche un meccanismo per verificare se la speculazione è corretta e un meccanismo per poter tornare indietro o eliminare gli effetti delle istruzioni eseguite in base alla speculazione. L'eliminazione degli effetti delle istruzioni eseguite in modo speculativo rende la progettazione della pipeline più complicata. *La speculazione può essere fatta dal compilatore oppure dall'hardware.*

Per esempio, il compilatore può utilizzare la speculazione per riordinare le istruzioni, spostando un'istruzione dopo un salto condizionato oppure invertendo tra loro una load e una store.

Il processore può operare le stesse modifiche durante l'esecuzione.

I meccanismi di correzione utilizzati in caso di speculazione errata sono abbastanza differenti tra loro.

- Nel caso di speculazione software, il compilatore di solito inserisce istruzioni aggiuntive che controllano l'accuratezza della speculazione e fornisce una procedura di riparazione da utilizzare quando si verifica un errore
- Nel caso di speculazione hardware, il processore di solito memorizza e mantiene in buffer dedicati i risultati dell'esecuzione speculativa, finché non "capisce" se la speculazione era corretta o meno. Se la speculazione risulta corretta, l'esecuzione delle istruzioni viene completata scrivendo il risultato nei registri o in memoria; se la speculazione si rivela errata, l'hardware elimina il contenuto di questi buffer e riprende l'esecuzione con la sequenza corretta di istruzioni

La speculazione introduce un altro genere di problema: essa può introdurre eccezioni che precedentemente non erano presenti.

Per esempio, si supponga che sulla base di una speculazione una load venga spostata e che l'indirizzo a cui fa riferimento non sia più valido se la speculazione risultasse sbagliata; in questo caso verrebbe generata un'eccezione di indirizzo non valido.

Il problema è reso complicato dal fatto che, quando una load non è oggetto di speculazione, se l'indirizzo risulta non valido, l'eccezione dev'essere sicuramente generata.

*Nella speculazione basata su compilazione, questo problema viene risolto introducendo uno speciale supporto alla speculazione che permette di ignorare queste eccezioni finché non è chiaro se debbano essere sollevate realmente.*

*Nella speculazione hardware, invece, le eccezioni vengono semplicemente memorizzate in un buffer finché la correttezza della speculazione, in base alla quale l'istruzione era stata lanciata in esecuzione, non è stata verificata; solo allora l'esecuzione dell'istruzione può essere terminata. A questo punto l'eccezione viene generata e viene iniziata la procedura di gestione dell'eccezione.*

Se c'è un'istruzione in un'istruzione speculata errata, invece, bisogna tornare indietro facendo finta di nulla.

*Dato che la speculazione può migliorare le prestazioni quando viene fatta in maniera appropriata, ma può diminuirle quando è implementata in modo poco accurato, un grosso sforzo è dedicato a decidere quando sia opportuno speculare.*

## Parallelizzazione statica dell'esecuzione

Tutti i processori dotati di parallelizzazione statica dell'esecuzione utilizzano il compilatore per formare i pacchetti di istruzioni e gestire gli hazard. In questo tipo di processore le istruzioni che vengono lanciate in esecuzione in parallelo nello stesso ciclo di clock costituiscono un *pacchetto di istruzioni (issue packet)*, che si può intendere come un'*istruzione ampia contenente più istruzioni al suo interno*.

Dal momento che, in un processore dotato di parallelizzazione statica, la combinazione di istruzioni che possono essere lanciate in esecuzione nello stesso ciclo di clock è in generale limitata, è utile pensare al *pacchetto di istruzioni* come a un'*istruzione singola nella quale diverse operazioni vengono codificate in campi predefiniti*; questa versione ha prodotto il nome originale di questo approccio: *Very Long Instruction Word (VLIW)*. La maggior parte dei processori con parallelizzazione statica si appoggia al compilatore per la gestione degli hazard sui dati e sul controllo. *Le responsabilità del compilatore comprendono la predizione statica dei salti e il riordino del codice per ridurre o prevenire gli hazard.*

### Un esempio: parallelizzazione statica dell'ISA del RISC-V

Per un'idea di come funziona un processore dotato di parallelizzazione statica dell'esecuzione, consideriamo un semplice processore RISC-V dotato di due cammini di esecuzione, detti anche *vie*; il primo cammino può eseguire istruzioni che operano sull'ALU interna o istruzioni di salto condizionato, il secondo può eseguire istruzioni di load e store.

Una pipeline di questo genere viene utilizzata in alcuni processori RISC-V di tipo embedded.

Lanciare in esecuzione due istruzioni per ciclo di clock richiede il fetch e la decodifica di istruzioni a 64 bit.

In molti processori a parallelizzazione statica (e praticamente in tutti i processori VLIW) esistono dei vincoli su quali istruzioni si possano inserire all'interno della coppia di istruzioni eseguite contemporaneamente, in modo da semplificare la fase di decodifica e di invio all'esecuzione.

Quindi, richiederemo che le istruzioni vengano accoppiate e allineate a parole di 64 bit, con l'istruzione che utilizza l'ALU interna o l'istruzione di salto per prima.

Inoltre, se una delle due istruzioni non può essere utilizzata, dev'essere rimpiazzata da una *nop*.

*Le istruzioni vengono quindi lanciate in esecuzione sempre in coppia, eventualmente con una *nop* al posto di una delle due.*

*I processori dotati di parallelizzazione statica si differenziano in base a come trattano i potenziali hazard sui dati e sul controllo.*

In alcune architetture il compilatore ha la piena responsabilità di rimuovere tutti gli hazard, riordinare il codice e inserire delle istruzioni *nop*, in modo tale che il codice venga eseguito senza alcun bisogno di rilevare hazard o generare stalli.

In altre architetture, è l'hardware a rilevare gli hazard e a generare gli stalli tra due pacchetti di istruzioni consecutivi, mentre viene richiesto al compilatore di eliminare tutte le possibili dipendenze all'interno della stessa coppia di istruzioni.

Anche così, un hazard in genere obbliga a mettere in stallo l'intero pacchetto di istruzioni che contiene l'istruzione con la dipendenza. Sia che il software debba gestire tutti gli hazard sia che debba solo cercare di ridurre il numero di hazard tra i diversi pacchetti di istruzioni, l'idea di un'*istruzione ampia* che codifica più operazioni al suo interno ne esce rafforzata.

Per lanciare in esecuzione una coppia di istruzioni, una che esegue un'operazione aritmetico-logica e una di trasferimento dati, il primo componente hardware aggiuntivo, oltre alla logica di rilevamento degli hazard e di messa in stallo simile a quella già vista, è una coppia addizionale di porte di lettura del register file.

In un singolo ciclo di clock, infatti, occorre leggere due registri per l'istruzione che opera sull'ALU e altri due per l'istruzione di trasferimento dati. Servono, inoltre, due porte di scrittura, una per l'operazione dell'ALU e l'altra per l'operazione di trasferimento dati dalla memoria.

Dato che l'ALU è impegnata nell'esecuzione dell'operazione aritmetico-logica, occorre anche un sommatore separato per il calcolo dell'indirizzo della memoria per il trasferimento dei dati. Senza queste risorse aggiuntive, il funzionamento di questa pipeline a due vie sarebbe ostacolato da un hazard di tipo strutturale.

Charamente questo semplice processore a due vie può migliorare le prestazioni al massimo di un fattore 2.

Tuttavia, con questo semplice parallelismo, si raddoppia il numero di istruzioni in esecuzione che si sovrappongono nel tempo e questo fa crescere le perdite di prestazioni dovute agli hazard sui dati e sul controllo.

Per esempio, nella nostra pipeline a cinque stadi le istruzioni di load hanno una latenza di utilizzo di un ciclo di clock, per cui un'altra istruzione non può utilizzare il risultato della load senza uno stallo.

Questo significa che in una pipeline a cinque stadi e due vie il risultato di un'istruzione di load non può essere utilizzato nel ciclo di clock successivo e le due istruzioni seguenti non possono utilizzare il risultato della load senza uno stallo.

Inoltre, le istruzioni che utilizzano l'ALU e che non hanno latenza di utilizzo nella pipeline semplice ora presentano una latenza di utilizzo di un'istruzione, poiché il loro risultato non può essere utilizzato da un'eventuale load o store con cui sono accoppiate.

*In un processore a esecuzione parallela, per sfruttare il parallelismo in modo efficace sono necessari compilatori più sofisticati o tecniche di riordinamento hardware del codice. I microprocessori a parallelizzazione statica richiedono che il compilatore svolga questi compiti.*

## Loop unrolling

**Il riordinamento di un semplice frammento di codice per la parallelizzazione statica dell'esecuzione:**

una tecnica importante adottata dai compilatori per ottenere prestazioni migliori nell'esecuzione dei cicli è *l'espansione dei cicli (loop unrolling)*. Con questa tecnica vengono generate più copie del corpo del ciclo e istruzioni appartenenti a iterazioni diverse possono essere avviate all'esecuzione contemporaneamente, inserendole nella stessa VLIW.

*In ogni iterazione del loop replica il corpo del ciclo per aumentare il parallelismo e diminuire l'overhead.*

Il loop unrolling può esserci utile nel multiple issue: posso infatti esporre più parallelismo.

Per farlo, uso registri differenti per replicazione. In altre parole, *cambio il nome dei registri (register renaming) per organizzare meglio le istruzioni e avere un IPC più alto*.

*E' un'operazione che viene fatta via hardware dal compilatore nel multiple issue (parallelizzazione) hardware: utilizza quindi più registri in maniera tale da tenere impegnate tutte le unità e avere parallelismo più elevato.*

Facciamo un esempio: per semplicità, si supponga che l'indice del ciclo sia multiplo di 4.

Per riordinare le istruzioni del ciclo in modo che non si verifichino ritardi, occorre fare quattro copie del corpo del ciclo.

Dopo averlo espanso e aver eliminato le istruzioni di controllo, che risultano inutili, il ciclo conterrà quattro copie delle istruzioni lw, addu e sw, e una copia delle istruzioni addi e bne.

Durante il processo di espansione, il compilatore ha introdotto alcuni registri addizionali (t1, t2, t3).

L'obiettivo di questa operazione, detta *ridenominazione dei registri*, è di *eliminare le false dipendenze tra i dati che potrebbero generare hazard o limitare il compilatore nel riordino del codice*.

Le diverse ripetizioni di queste tre istruzioni, malgrado utilizzino tutte t0, sono in realtà completamente indipendenti: non esiste alcun flusso di dati tra una coppia di istruzioni in un'iterazione e la stessa coppia di istruzioni in un'altra iterazione. Si parla in questo caso di *falsa dipendenza*, o dipendenza nominale, che *implicherebbe un riordinamento delle istruzioni forzato dall'utilizzo di uno stesso nome e non da una reale dipendenza*, chiamata anche dipendenza effettiva.

*Il processo di ridenominazione dei registri durante l'espansione dei cicli permette al compilatore di inserire una dopo l'altra queste istruzioni tra loro indipendenti, in modo da ottenere una migliore riorganizzazione del codice.*

*La ridenominazione dei registri elimina le false dipendenze ma non quelle vere.*

L'espansione del ciclo e la riorganizzazione del codice in coppie di istruzioni hanno prodotto un miglioramento delle prestazioni di quasi un fattore 2, dovuto in parte alla riduzione del numero di istruzioni di controllo del ciclo e in parte all'esecuzione parallela di coppie di istruzioni. Il costo di questo miglioramento consiste nell'uso di quattro registri temporanei, invece che di uno solo, e in un significativo incremento nella dimensione del codice.

## Processori dotati di parallelizzazione dinamica dell'esecuzione (PROCESSORI SUPERSCALARI)

I processori dotati di parallelizzazione dinamica dell'esecuzione sono anche conosciuti come *processori superscalari*.

Nei processori superscalari più semplici, le istruzioni vengono eseguite in ordine e il processore decide se nessuna, una o più istruzioni possono essere avviate all'esecuzione nello stesso ciclo.

Ovviamente, per ottenere buone prestazioni con questo tipo di processore occorre che il compilatore riordini le istruzioni in modo da rimuovere le dipendenze per aumentare il numero di istruzioni che possono essere lasciate in parallelo.

Anche con questo supporto del compilatore c'è una differenza importante tra questo semplice processore superscalare e un processore VLIW: è *l'hardware che garantisce che il codice, riordinato o meno, venga eseguito correttamente*.

Inoltre, *il codice in linguaggio macchina verrà eseguito sempre in modo corretto, indipendentemente da quante istruzioni vengono eseguite in parallelo o dalla struttura della pipeline del processore*.

In alcuni processori VLIW ciò può non essere vero, e diventa quindi necessario ricompilare i programmi quando ci si sposta su modelli diversi dello stesso processore; in altri processori VLIW, il codice può venire eseguito correttamente anche su modelli diversi, ma con prestazioni così basse da rendere di fatto necessaria una ricompilazione del codice.

*Molti processori superscalari estendono la versione base dell'unità che garantisce la parallelizzazione dell'esecuzione in modo da includere un riordinamento dinamico delle istruzioni (dynamic scheduling).*

Questo schema permette di decidere quali istruzioni eseguire in un dato ciclo di clock e, allo stesso tempo, di evitare hazard e stalli.

La riorganizzazione dinamica del codice nelle pipeline permette di evitare gli hazard completamente o parzialmente.

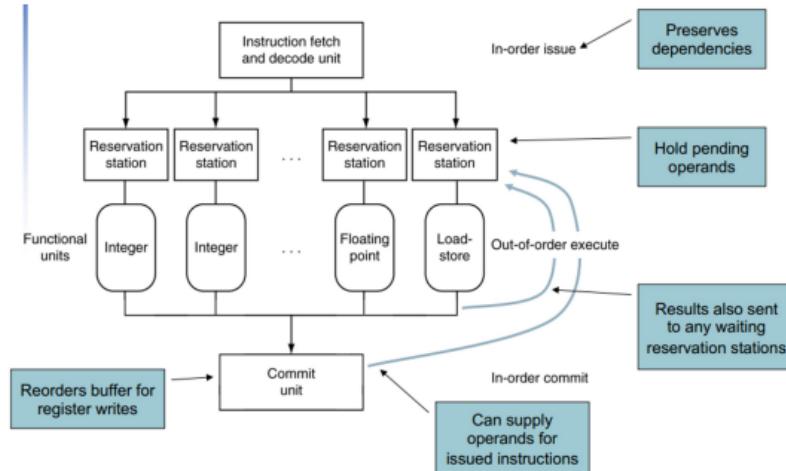
## Riorganizzazione dinamica del codice in una pipeline

Nella riorganizzazione dinamica del codice si sceglie quali istruzioni eseguire nei cicli di clock successivi, eventualmente riordinandole in modo da evitare stalli.

In un processore di questo tipo (superscalare) la pipeline viene suddivisa in tre unità principali:

- un'unità che preleva l'istruzione dalla memoria e la avvia all'esecuzione
- un gruppo di più unità funzionali

- un'unità di consegna (commit unit)



La prima unità preleva le istruzioni, le decodifica e invia ciascuna di esse alla corrispondente unità funzionale per l'esecuzione. Ciascuna unità funzionale è dotata di buffer, chiamati stazioni di prenotazione (reservation stations), che conservano gli operandi e l'operazione.

Non appena la stazione di prenotazione contiene tutti gli operandi richiesti e l'unità funzionale è pronta per l'esecuzione, viene eseguita l'operazione e inviato il risultato alle stazioni di prenotazione che lo stanno attendendo e contemporaneamente all'unità di consegna.

Questa conserva il risultato dell'operazione fino a quando non avrà il via libera per salvarlo nel register file o, nel caso di un'istruzione di store, nella memoria.

Il buffer chiamato nell'unità di consegna è chiamato buffer di riordino e viene anche utilizzato per fornire gli operandi, un po' come faceva la logica di propagazione nelle pipeline con riorganizzazione statica del codice.

Una volta scritto il risultato nel register file, esso può essere prelevato da lì come in una normale pipeline.

La combinazione del salvataggio degli operandi nei buffer delle stazioni di prenotazione e dei risultati nel buffer di riordino consente una forma di ridenominazione dei registri del tutto analoga a quella utilizzata dal compilatore nel caso dell'esempio precedente.

Per comprenderne meglio il funzionamento da un punto di vista concettuale, consideriamo i seguenti passi.

1. Quando inizia l'esecuzione di un'istruzione, essa viene copiata nella stazione di prenotazione associata all'unità funzionale appropriata e i suoi operandi disponibili copiati dal register file o dal buffer di riordino. L'istruzione rimane all'interno della stazione di prenotazione finché tutti gli operandi non saranno disponibili e l'unità funzionale non sarà pronta. Non è più necessario copiare dai registri gli operandi per l'istruzione che viene avviata all'esecuzione; inoltre, se occorre scrivere i registri che contengono gli operandi, il loro contenuto può essere tranquillamente sovrascritto.
2. Se un operando non si trova nel register file o nel buffer di riordino, l'istruzione deve aspettare che esso sia prodotto da una delle unità funzionali. In questo caso, il lavoro di quell'unità funzionale viene seguito e il risultato prodotto viene copiato direttamente dall'unità funzionale nella stazione di prenotazione, senza aspettare che venga scritto nel registro di destinazione.

*In questi due passaggi si utilizzano le stazioni di prenotazione e il buffer di riordino per implementare la ridenominazione dei registri.*

Concettualmente, possiamo pensare ad una pipeline a riordinamento dinamico del codice come a un'entità che analizza la struttura del flusso dei dati di un programma e poi esegue le istruzioni in un ordine che può anche essere diverso, preservando però il flusso dei dati prodotti.

Questo stile di esecuzione è chiamato esecuzione fuori ordine (*out of context execution*), poiché le istruzioni possono essere eseguite in un ordine diverso da quello con cui erano state prelevate dalla memoria istruzioni.

Per fare in modo che i programmi si comportino come se le istruzioni fossero eseguite in ordine, l'unità di fetch e decodifica invia in esecuzione le istruzioni nella sequenza originale, cosa che consente di identificare le dipendenze, e quella di consegna dovrà scrivere i risultati nei registri o in memoria seguendo l'ordine con cui il programma ha prelevato le istruzioni dalla memoria.

Questo modello conservativo di esecuzione è detto *completamento in ordine (in order commit)*. Quindi, nel caso in cui si verifichi un'eccezione, il calcolatore può fare riferimento all'ultima istruzione eseguita, e gli unici registri aggiornati saranno solo quelli scritti dalle istruzioni che precedono l'istruzione che ha generato l'eccezione.

Sebbene la parte iniziale della pipeline (fetch e avvio all'esecuzione) e la parte terminale (consegna) eseguano il codice in ordine, le unità funzionali sono invece libere di iniziare e terminare l'esecuzione non appena i dati di cui hanno bisogno si rendono disponibili.

Oggi tutte le pipeline dotate di riordinamento dinamico del codice utilizzano il completamento in ordine.

Il riordinamento dinamico del codice viene spesso esteso includendo la speculazione hardware, specialmente per i salti condizionati. Prevedendo la direzione di un salto, un processore con riorganizzazione dinamica del codice può prelevare ed eseguire le istruzioni lungo il ramo di esecuzione scelto.

Una pipeline speculativa dotata di riordinamento dinamico del codice può anche supportare la speculazione sugli indirizzi delle load (non funziona benissimo nei processori Intel, poiché ne resta traccia -> sfruttato per accesso alle memorie da hacker), permettendo il riordino delle istruzioni di load-store; l'unità di consegna verrà utilizzata per evitare di scrivere i risultati prodotti dalle istruzioni eseguite quando la speculazione si rivela errata.

## Capire le prestazioni dei programmi

Dato che i compilatori possono riordinare il codice anche quando ci sono delle dipendenze tra i dati, potreste chiedervi perché un processore superscalare debba utilizzare il riordinamento dinamico del codice. I motivi sono principalmente tre.

1. Non tutti gli stalli possono essere predetti. In particolare, i miss delle cache nella gerarchia delle memorie possono causare degli stalli non prevedibili. La riorganizzazione dinamica del codice consente al processore di nascondere alcuni di questi stalli continuando ad eseguire altre istruzioni in attesa che lo stallo dovuto alla lettura abbia termine.
2. Se il processore speculasse sulla direzione dei salti condizionati, utilizzando la predizione dinamica dei salti, non potrebbe comunque sapere qual è l'ordine corretto di esecuzione delle istruzioni all'atto della compilazione, poiché questo dipende dal comportamento predetto e da quello effettivo del salto, durante l'esecuzione. Incorporare la speculazione dinamica per sfruttare un maggiore parallelismo a livello di istruzioni (ILP) senza incorporare il riordino dinamico del codice ridurrebbe grandemente i benefici della speculazione.
3. Poiché la latenza della pipeline e il numero di istruzioni che vengono lanciate in esecuzione in parallelo possono variare molto da un processore all'altro, il modo migliore per compilare una sequenza di istruzioni può cambiare. La struttura della pipeline influenza sia il numero di volte che un ciclo deve essere espanso per evitare stalli, sia il processo di ridenominazione dei registri da parte del compilatore. Il riordinamento dinamico del codice consente all'hardware di nascondere la maggior parte di questi dettagli, per cui gli utenti ed i distributori di software non devono preoccuparsi di avere versioni diverse dello stesso programma per le differenti implementazioni hardware dello stesso insieme di istruzioni. Analogamente, i vecchi programmi godranno dei miglioramenti delle nuove architetture senza che debba essere ricompilato il codice.

## Quadro d'insieme

Sia la pipeline che l'esecuzione parallela incrementano il throughput di picco delle istruzioni e tentano di sfruttare il parallelismo a livello di istruzioni (ILP). Le dipendenze tra i dati e sul controllo nei programmi pongono, tuttavia, un limite superiore a prestazioni elevate, poiché il processore a volte deve aspettare che le dipendenze vengano risolte.

Gli approcci basati sul software si affidano all'abilità del compilatore nel trovare e ridurre gli effetti di queste dipendenze, mentre gli approcci basati sull'hardware si affidano a estensioni della pipeline e a meccanismi hardware di parallelizzazione.

La speculazione, attuata dal compilatore o dall'hardware, può incrementare, attraverso la predizione, la parte di ILP che può essere sfruttata, benché si debba fare attenzione alle speculazioni sbagliate che hanno conseguenze assai dannose sulle prestazioni.

## Interfaccia hardware/software

I microprocessori moderni ad alte prestazioni sono capaci di lanciare in esecuzione molte istruzioni a ogni ciclo di clock; sfortunatamente, mantenere costante questo tasso di riempimento dei pacchetti di istruzioni è molto difficile.

I motivi sono principalmente due:

1. nella pipeline il collo di bottiglia principale è rappresentato dalle dipendenze che non possono essere eliminate; queste riducono il parallelismo tra le istruzioni e quindi il tasso di riempimento dei pacchetti di istruzioni. Anche se si può fare poco contro le dipendenze vere, spesso il compilatore o l'hardware non sono in grado di stabilire con precisione se una dipendenza sia reale o meno, e quindi devono ipotizzare, in modo conservativo, che la dipendenza ci sia. Per esempio, il codice basato sui puntatori, soprattutto quando essi puntano ad aree di memoria parzialmente sovrapposte, porta a dipendenze più complesse da analizzare. Al contrario, la maggiore regolarità dell'accesso agli elementi dei vettori consente spesso al compilatore di capire quando non esistono dipendenze reali. In maniera simile, i salti condizionati che non possono essere predetti accuratamente durante l'esecuzione, o durante la compilazione, limitano fortemente la possibilità di sfruttare l'ILP. Spesso sarebbe utile utilizzare un ILP maggiore, ma l'abilità del compilatore o dell'hardware nel trovare istruzioni che possono essere molto distanti, a volte anche migliaia di istruzioni, è limitata;
2. i miss che scaturiscono dalla gerarchia delle memorie limitano anch'esse la capacità di mantenere la pipeline sempre piena. In alcuni sistemi di memoria gli stalli possono essere nascosti, ma un ILP minor limita anche la capacità di nascondere gli stalli.

## Efficienza energetica nelle pipeline avanzate

L'incremento del parallelismo a livello di istruzioni mediante parallelizzazione dinamica del codice ha come rovescio della medaglia il potenziale abbassamento dell'efficienza energetica.

Ciascuna innovazione introdotta nel tempo ha sfruttato il crescente numero di transistor per incrementare le prestazioni, ma spesso ciò è avvenuto in modo scarsamente efficiente.

Ora che la barriera dell'energia è stata raggiunta, si cominciano a produrre architetture con più processori sullo stesso chip, nelle quali ciascun processore ha una pipeline meno profonda ed è dotato di una speculazione meno aggressiva rispetto ai precedenti.

Nonostante i processori più semplici non siano tanto veloci quanto i loro sofisticati fratelli, infatti, essi hanno prestazioni migliori per Joule e possono quindi garantire prestazioni più elevate per chip quando i vincoli sul progetto sono dettati più dall'energia assorbita che dal numero di transistor.

*Con l'avvento dei processori multicore vi è stata una diminuzione del numero di stadi di pipeline e della potenza dissipata.*

Il Pentium è il primo processore superscalare, l'ISA è sempre lo stesso ma era in grado di fare multiple issue a due istruzioni per volta. Pescava comunque le istruzioni in-order. Il parallelismo delle varie unità portò ad un raddoppio dell'energia assorbita.

In breve: la complessità dello scheduling dinamico richiede potenza (ci sono un sacco di transistor). Avere core multipli potrebbe essere migliore. Se uso lo scheduling dinamico, ogni processore può fare il possibile per eseguire velocemente qualsiasi programma sequenziale.

Nel caso dei core, invece, devo per forza programmare in parallelo.

## Hyperthreading (piccola parentesi)

Idea: per tenere le pipeline occupate, pescavo le istruzioni da più programmi.

Con questa tecnica, con due unità di fetch, riesco a tenere occupato l'hardware in maniera più efficiente.

E' chiamata multithreading da AMD.

Questa tecnica corrisponde ad un processore con due unità di fetch che pescano da due programmi separati.

Resta comunque un unico processore.

Aumento l'esecuzione del singolo programma, ma la coppia dei due programmi è eseguita in un tempo totale minore.

Come viene gestita la doppia unità di fetch a livello di OS? Dal suo punto di vista viene visto come un doppio processore.

Posso disabilitare l'HT nelle opzioni del BIOS.

## La pipeline del Cortex-A53 ARM e del Core i7 Intel

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1 <sup>st</sup> level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2 <sup>nd</sup> level caches/core	128-2048 KiB	256 KiB (per core)
3 <sup>rd</sup> level caches (shared)	(platform dependent)	2-8 MB

Morale: avrò sempre dei compromessi. Non posso progettare un processore adatto sia al mercato mobile che a quello dei server.

### Cortex-A 53

Pipeline del Cortex-A53: i primi tre stadi leggono le istruzioni dalla memoria e le inseriscono in una coda di 13 elementi. L'unità di generazione degli indirizzi (AGU) utilizza un predictor indiretto e uno stack degli indirizzi di ritorno per predire i salti condizionati e cercare di mantenere piena la coda delle istruzioni. La decodifica delle istruzioni richiede tre stadi, così come l'esecuzione.

Altri due stadi sono riservati alle operazioni in virgola mobile e alle istruzioni SIMD.

Performance Cortex-A53: anche se il CPI (Cost Perform Index) ideale è 0,5, il valore migliore ottenuto 1,0, il valore mediano è 1,3 ed il valore peggiore 8,6. Per il caso mediano il 60% degli stalli è dovuto a hazard della pipeline e il 40% alla gerarchia delle memorie. Gli stalli della pipeline sono causati da predizioni errate sui salti condizionati, da hazard strutturali e da dipendenze tra dati di coppie di istruzioni. Data la natura statica della pipeline del Cortex-A53, è compito del compilatore cercare di evitare gli hazard strutturali e le dipendenze tra i dati.

### Core i7 920 di Intel

Pipeline del Core i7: la profondità totale della pipeline è di 14 stadi, con un costo degli errori di predizione di 17 cicli di clock. Questo processore può inserire nel buffer 48 load e 32 store. I sei cammini di esecuzione indipendenti possono iniziare l'esecuzione di un'operazione RISC pronta a ogni ciclo di clock.

Cosa sono le micro-op?

I programmati utilizzano l'ISA del 386, ma il processore prende le istruzioni e le traduce in istruzioni più semplici simili a quelle RISC, così da far funzionare meglio la pipeline (cosa impossibile con i processori CISC) -> le istruzioni complesse devono essere scisse in micro-op semplici -> set CISC che si trasformano in RISC.

Nelle function unit ho delle unità che si occupano di load/store (due per lo store) con un buffer che precede l'accesso alla memoria.

Ho anche delle function unit che si occupano delle moltiplicazioni e delle divisioni floating point.

Performance Core i7: il Core i7 combina una pipeline a 14 stadi con una parallelizzazione aggressiva dell'esecuzione per ottenere prestazioni elevate. Mantenendo bassa la latenza per le operazioni che vengono eseguite una di seguito all'altra, l'impatto della dipendenza tra i dati viene ridotto.

Quali sono i maggiori colli di bottiglia per le prestazioni dei programmi eseguiti su questo processore?

- L'utilizzo delle istruzioni x86, che non si possono trasformare in poche semplici micro-op
- I salti condizionati, che sono difficili da predire; questi causano stalli quando la predizione si rivela errata, con conseguente svuotamento e riavvio della pipeline
- Lunghe dipendenze. Queste sono tipicamente causate da istruzioni con elevato tempo di esecuzione o dalla gerarchia delle memorie e portano a stalli
- Ritardi nell'esecuzione dovuti all'accesso alla memoria, che causano lo stallo del processore

## Considerazioni finali

- Il pipelining è facile? No! Rilevare gli hazard sui dati è difficile.
- Il pipelining è dipendente dalle tecnologie? No! Ho bisogno di tanti transistor per rendere le tecniche avanzate più fattibili.
- Il design ISA deve tenere conto del trend della tecnologia? Più è complicato il set di istruzioni, più è difficile implementare il pipelining. Per farlo funzionare correttamente con set complessi, ho bisogno di un overhead significativo. I moduli di indirizzamento complesso e i salti ritardati

- possono complicare ancora più la pipeline.
- L'ISA influenza il design del datapath e del controllo (e la cosa è mutua). Avere un set di istruzioni moderno e fatto di istruzioni semplici porta giovamento.
  - Il pipelining migliora il throughput delle istruzioni attraverso il parallelismo. La latenza non viene ridotta, ma esegue più istruzioni per secondo.
  - Il pipelining è affetto da una serie di problematiche dette hazard che possono essere strutturali, di dati e di controllo. Le ultime due sono particolarmente problematiche per l'accesso sequenziale alle istruzioni.
  - Posso ridurre l'effetto degli hazard grazie al multiple issue e allo scheduling dinamico (ILP). Ho comunque il problema delle dipendenze dei programmi che limita la quantità di parallelismo che riesco a spingere. La complessità eccessiva dell'hardware dovuta al multiple issue porta ad un'esplosione dell'energia assorbita (power wall). Questo power wall ha portato ad una diminuzione delle frequenze di clock e lo sfruttamento del parallelismo a livello di core (non a livello di istruzioni).

## Cos'è un processore superscalare?

Un processore superscalare ha più unità separate, le quali vengono gestite con una tecnica di multiple issue dinamico. Ad ogni colpo di clock decide quali sono le istruzioni che possono partire, riuscendo ad individuare ed evitare hazard di struttura e di dati.

Lo scheduling dinamico delle pipeline permette alla CPU di eseguire le istruzioni fuori ordine per evitare gli stalli.

Per definizione, è un sistema con più core, dove ognuno di essi ha un'unità di fetch e 4 stazioni di prenotazione con funzioni diverse (Integer, Float, Load/Store). Vi è un register remaining, dove faccio avvenire le operazioni solo quando gli operandi sono disponibili. Quando i dati sono nelle stazioni di prenotazione, i registri possono essere riscritti. Se gli operandi non sono disponibili, ci sarà una function unit a portarli alle station. In questo caso il register remaining non serve. Ad operazioni compiute, la commit unit manda i risultati in memoria. La difficoltà è far funzionare tutte le station per ogni programma.

Un processore superscalare è una forma **intermedia: istruzioni diverse trattano i propri operandi contemporaneamente, su diverse unità hardware all'interno dello stesso chip**. In questo modo nello stesso ciclo di clock possono essere eseguite più istruzioni.

Altra risposta:

I processori dotati di parallelizzazione dinamica dell'esecuzione sono anche conosciuti come processori superscalari, o semplicemente superscalari. Nei processori superscalari più semplici, le istruzioni vengono eseguite in ordine e il processore decide se nessuna, una o più istruzioni possono essere avviate all'esecuzione nello stesso ciclo. Ovviamente, per ottenere buone prestazioni con questo tipo di processore occorre che il compilatore riordini le istruzioni in modo da rimuovere le dipendenze per aumentare il numero di istruzioni che possono essere lanciate in parallelo. C'è una differenza importante tra questo semplice processore superscalare e un processore VLIW(very long instruction word): è l'hardware che garantisce che il codice, riordinato o meno, venga eseguito correttamente. Molti processori superscalari estendono la versione base dell'unità che gestisce la parallelizzazione dell'esecuzione in modo da includere un riordinamento dinamico delle istruzioni. Questo schema permette di decidere quali istruzioni eseguire in un dato ciclo di clock e, allo stesso tempo, di evitare hazard e stalli.

Come mai si adottano processori superscalari? Perché hanno preso piede i processori dinamici e non quelli statici.

## Cosa succede quando avviene un'eccezione in un processore superscalare?

Per fare in modo che i programmi si comportino come se le istruzioni fossero eseguite in ordine, l'unità di fetch e decodifica invia in esecuzione le istruzioni nella sequenza originale, cosa che consente di identificare le dipendenze, e quella di consegna dovrà scrivere i risultati nei registri o in memoria seguendo l'ordine con cui il programma ha prelevato le istruzioni dalla memoria.

Questo modello conservativo di esecuzione è detto *completamento in ordine (in order commit)*. Quindi, nel caso in cui si verifichi un'eccezione, il calcolatore può fare riferimento all'ultima istruzione eseguita, e gli unici registri aggiornati saranno solo quelli scritti dalle istruzioni che precedono l'istruzione che ha generato l'eccezione.

Ergo, in un processore superscalare, l'eccezione può interrompere solo alcune delle istruzioni in esecuzione, mentre le altre possono continuare ad essere elaborate. Questo dipende dalla capacità del processore di gestire le eccezioni e della progettazione del sistema operativo.

## Processore con Very Long Instruction Word (VLIW)

Il Very Long Instruction Word (VLIW) è una tecnologia di elaborazione utilizzata in alcuni processori per migliorare l'efficienza dell'elaborazione. In un processore VLIW, molte istruzioni sono incluse in un'unica parola di istruzione lunga che viene elaborata dal processore in parallelo. Ciò significa che il processore può eseguire molte operazioni in un solo ciclo di clock, aumentando così l'efficienza dell'elaborazione. Questo approccio è diverso dai processori con pipeline tradizionali che eseguono una sola istruzione per ciclo di clock.

Altra risposta:

Tutti i processori dotati di parallelizzazione statica dell'esecuzione utilizzano il compilatore per formare i pacchetti di istruzioni e gestire gli hazard. In questo tipo di processore le istruzioni che vengono lanciate in esecuzione in parallelo nello stesso ciclo di clock costituiscono un pacchetto di istruzioni (issue packet), che si può intendere come un'istruzione ampia contenente più istruzioni al suo interno.

Dal momento che, in un processore dotato di parallelizzazione statica, la combinazione di istruzioni che possono essere lanciate in esecuzione nello stesso ciclo di clock è in generale limitata, è utile pensare al pacchetto di istruzioni come a un'istruzione singola nella quale diverse operazioni vengono codificate in campi predefiniti; questa visione ha prodotto il nome originale di questo approccio: Very Long Instruction Word (VLIW). La maggior parte dei processori con parallelizzazione statica si appoggia al compilatore per la gestione degli hazard sui dati e sul controllo. Le responsabilità del compilatore comprendono la predizione statica dei salti e il riordino del codice per ridurre gli hazard.

## Cos'è la predizione dei salti?

Per evitare hazard di controllo e velocizzare il processore delle pipeline evitando la formazione di bolle posso cercare di predire se un salto avverrà o meno.

Predizione del branch statico vs predizione del branch dinamico.

Predire utilizzando 1 bit vs predire utilizzando 2 bit.

## Che effetto ha l'istruzione più lunga sulla pipeline?

Se un'istruzione è particolarmente lunga, il ciclo di clock deve essere abbastanza lungo da contenere l'istruzione più lunga.

## Cos'è la pipeline?

La pipeline è una tecnica di implementazione hardware di un insieme di istruzioni che prevede la sovrapposizione temporale dell'esecuzione delle diverse istruzioni. La pipeline è una tecnica che sfrutta il parallelismo tra le istruzioni di un programma software scritto in maniera sequenziale. Ha il grosso vantaggio di essere sostanzialmente invisibile al programmatore. Un approccio basato sul concetto di pipeline richiede molto meno tempo. La pipeline aumenta il numero di istruzioni che vengono eseguite contemporaneamente e la frequenza con cui viene iniziata e terminata l'esecuzione delle istruzioni. Il RISC-V è stato progettato espressamente per l'esecuzione in pipeline.

## Cosa sono gli hazard?

In una pipeline si possono verificare situazioni in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente successivo. Tali eventi sono detti hazard (o criticità) e possono essere di tre tipi:

- Hazard strutturale: si verifica quando le risorse hardware presenti non sono in grado di supportare la combinazione di istruzioni che si vorrebbe eseguire nello stesso ciclo di clock. L'insieme di istruzioni MIPS è stato progettato appositamente per essere eseguito in pipeline, facendo in modo che per i progettisti fosse semplice evitare gli hazard strutturali nel progetto dell'unità di elaborazione.
- Hazard sui dati: si verifica quando la pipeline deve essere messa in stallo perché uno stadio deve attendere che termini l'elaborazione in un altro stadio della pipeline. In un calcolatore dotato di pipeline, gli hazard sui dati nascono quando un'istruzione dipende dal risultato di un'istruzione precedente che si trova ancora all'interno della pipeline.
- Hazard sul controllo o hazard sui salti condizionati: avvengono quando bisogna prendere una decisione in funzione del risultato dell'esecuzione di un'istruzione e altre istruzioni sono già state avviate all'esecuzione.

## Cos'è la predizione di un salto?

Una soluzione per risolvere gli hazard sul controllo è basata sulla "Predizione". Con la predizione dei salti si cerca di prevedere se il salto sarà intrapreso oppure no. L'approccio più semplice consiste nel predire sempre che il salto venga non eseguito. Se la predizione si rivela corretta, la pipeline procede al massimo della velocità, e solo quando il salto doveva essere eseguito si verifica uno stallo. Una versione più sofisticata di predizione dei salti consente di prevedere se un salto debba essere eseguito o meno. I predictor hardware dinamici, invece, determinano la predizione per ciascun salto in funzione del comportamento precedente di quell'istruzione di salto, e possono modificare la predizione di ogni salto durante l'esecuzione del programma.

## Predizione ad 1 bit e a 2 bit

Con la predizione ad 1 bit non possiamo sapere se la predizione sia corretta o meno; per esempio, la predizione potrebbe riguardare un'altra istruzione di salto condizionato la cui parte inferiore dell'indirizzo coincide con quella dell'istruzione di salto che stiamo analizzando. Tuttavia, un errore sulla predizione non influenza la correttezza dell'esecuzione: la predizione è solamente un suggerimento che si spera risulti corretto, su cui è basato il prelevamento dell'istruzione successiva. Se il suggerimento si rivela sbagliato, le istruzioni caricate erroneamente saranno eliminate, verrà invertito il bit di predizione e l'esecuzione ripartirà dal prelevamento dell'istruzione corretta. Questo semplice schema di predizione a 1 bit ha un inconveniente: anche se un salto venisse quasi sempre effettuato, la predizione risulterebbe sbagliata almeno due volte invece che una volta sola, quando il salto non viene effettuato. Per rimediare a questa inadeguatezza, si utilizzano spesso degli schemi di previsione a 2 bit. In uno schema a 2 bit, la predizione deve risultare sbagliata due volte di seguito prima di essere modificata. Utilizzando 2 bit anziché 1 bit, una branch che abbia una forte tendenza ad eseguire il salto o a non eseguirlo, avrà una predizione errata soltanto una volta.

[La predizione a un bit fa sì che se il bit di predizione è settato, il salto è predetto come "preso"/"taken" altrimenti "non preso"/"not taken". In caso di errore della predizione, lo stato del bit viene ribaltato e così anche la direzione della successiva predizione. Nei cicli singoli, tutto funziona correttamente per tutta la durata del ciclo, finché non si deve uscire. Nei loop annidati, uno schema un bit causa due errori di predizione per il loop interno: uno alla fine del loop, quando l'iterazione esce dal ciclo invece che looppare ancora, e una quando, eseguendo la prima iterazione del loop, predice esci invece di continuare dentro il ciclo. Questi tipi di errori vengono superati con lo schema successivo, quello a 2bit, rendendo quindi questo sistema poco usato. La differenza fondamentale tra questo schema e quello ad 1 bit è che qui si ha un comportamento più "prudente", nel senso che prima di cambiare stato si vede non solo quello che è successo nel branch precedente ma anche in quello ancora prima. Quindi vengono tenuti in memoria i risultati ottenuti nei due branch che precedono quello che si sta esaminando.]

## Le informazioni sui salti dove vengono mantenute?

Le informazioni sui salti vengono salvate in un buffer di predizione dei salti, detto anche tabella della storia dei salti. Si tratta di una piccola memoria indicizzata dalla porzione meno significativa dell'indirizzo dell'istruzione di salto, contenente uno o più bit che indicano se in precedenza il salto è stato effettuato.

## Cos'è il renaming dei registri?

La ridenominazione dei registri è una tecnica di ottimizzazione utilizzata dai microprocessori per incrementare il livello di parallelismo dei programmi, eliminando alcuni vincoli durante l'esecuzione delle istruzioni nelle pipeline. Durante l'esecuzione di più istruzioni in parallelo all'interno di una pipeline alcune istruzioni non possono essere eseguite perché devono attendere anche alcuni registri utilizzati da altre istruzioni siano liberi. Per evitare di dover attendere l'esecuzione delle istruzioni in corso per la presenza di registri occupati si usa la ridenominazione degli stessi. Quando il processore individua delle istruzioni che non possono essere eseguite per la presenza di registri già occupati il processore assegna all'istruzione altri registri, dei registri temporanei da poter utilizzare fino a quando i registri di destinazione non siano liberi.

## Come funzionano le eccezioni? Com'è la gestione delle eccezioni in MIPS? Differenza tra eccezioni ed interruzione.

[L'operazione fondamentale che il processore deve compiere al verificarsi di un'eccezione consiste nel salvare l'indirizzo dell'istruzione che l'ha generata nel program counter delle eccezioni (EPC, exception program counter) e trasferire il controllo a un indirizzo specifico del sistema operativo. Il sistema operativo potrà quindi intraprendere le azioni più opportune, che potrebbero consistere nel fornire alcuni servizi al programma utente, eseguire un'azione predeterminata o terminare l'esecuzione del programma segnalando un errore. Dopo aver compiuto le azioni necessarie per rispondere all'eccezione, il sistema operativo può terminare il programma o riprenderne l'esecuzione utilizzando l'EPC per determinare il punto da cui ripartire. Per poter gestire correttamente un'eccezione, il sistema operativo deve conoscerne la causa, oltre a sapere quale istruzione l'ha generata. Ci sono due metodi per comunicare la causa di un'eccezione: il metodo utilizzato nell'architettura MIPS è quello di prevedere un registro dedicato (detto registro causa) contenente un campo che indica la causa dell'eccezione. Un metodo alternativo consiste nell'adottare interrupt vettORIZZATI, nei quali l'indirizzo a cui si deve trasferire il controllo viene determinato dalla causa dell'eccezione stessa. Quando l'eccezione non è vettORIZZATA, il sistema operativo inizia a rispondere a tutte le eccezioni dallo stesso indirizzo e deve decodificare il registro causa per capire che cosa abbia generato l'eccezione.]

Sono entrambi degli eventi inaspettati che alterano il normale flusso di eventi di un programma. Con la differenza che:

- Eccezioni hanno origine all'interno della CPU (per esempio si parla di eccezione in caso di overflow, di syscall, di un opcode non definito ossia di un'istruzione non valida, ecc...);
- Interrupts possono essere generate solo all'esterno del processore (per esempio da devices esterni).

Come si gestisce un'eccezione o un interrupt? [In MIPS un'eccezione viene gestita dal System Control Coprocessor (CP0), ossia un coprocessoRE che come un qualsiasi altro coprocessoRE ha il compito di sgravare il processore principale da determinati compiti, in questo caso dalla gestione delle eccezioni].

In RISC-V, nel momento in cui si verifica un'eccezione:

- 1 → L'indirizzo dell'istruzione che l'ha causata (ossia l'istruzione alla quale punta il PC) viene salvato in un apposito registro chiamato EPC (Exception Program Counter) (in modo tale che se dopo la gestione dell'eccezione si vuole riprendere la normale esecuzione del programma dal punto in cui essa è stata interrotta, tramite l'EPC si può fare) ed il controllo viene subito trasferito al sistema operativo, non all'OS in generale, ma ad uno specifico indirizzo dell'OS che dipende dalla causa che ha generato l'eccezione;
- 2 → Ecco perché la causa dell'eccezione (quindi non l'indirizzo dell'istruzione, ma la causa vera e propria) viene salvata in un registro detto "registro di causa".

In altre architetture (non RISC-V) si utilizza la tecnica dell' "interrupt vettORIZZATO", ossia l'indirizzo del OS al quale si deve trasferire il controllo viene determinato sulla base della causa dell'eccezione stessa. Nel momento in cui l'OS deve gestire l'eccezione, soltanto conoscendo l'indirizzo al quale è stato trasferito il controllo, conoscerà già la causa dell'eccezione. Sta all'OS, poi, gestire l'eccezione, potrebbe decidere anche di terminare il programma segnalando l'errore.

## Cos'è un interrupt e chi lo genera?

È un'eccezione che ha origine all'esterno del processore. Alcune architetture utilizzano il termine interrupt per indicare tutti i tipi di eccezioni.

## Le istruzioni RISC-V quanti passi richiedono e quali sono?

Le istruzioni MIPS richiedono 5 passi:

1. fetch dell'istruzione (cioè caricamento dell'istruzione dalla memoria);
2. lettura dei registri e decodifica dell'istruzione: il formato regolare delle istruzioni MIPS permette la simultaneità tra lettura e decodifica;
3. esecuzione di un'operazione o calcolo di un indirizzo;
4. accesso a un operando nella memoria dati;
5. scrittura del risultato in un registro.

## Cos'è un processore con scheduling dinamico?

Molti processori superscalari estendono la versione base dell'unità che gestisce la parallelizzazione dell'esecuzione in modo da includere un riordinamento dinamico delle istruzioni. Questo schema permette di decidere quali istruzioni eseguire in un dato ciclo di clock e allo stesso tempo di evitare hazard e stalli.

# Capitolo 5 - Architettura

## La gerarchia delle memorie

### Principi di località

Per la memoria l'obiettivo principale è creare l'illusione di avere a disposizione in un calcolatore una memoria di grandi dimensioni a cui poter accedere con la stessa velocità con cui si accede ad una memoria di piccole dimensioni.

Il principio di località sta alla base del comportamento dei programmi di un calcolatore ed è del tutto simile al modo di cercare informazioni in una biblioteca. Questo principio afferma che un programma, in un certo istante di tempo, non usa uniformemente tutta la memoria, ma accede soltanto ad una piccola porzione del suo *spazio di indirizzamento*.

Il principio di località vale sia per i dati che per le istruzioni, anche se vale principalmente per quest'ultime.

Esistono due tipi di località:

- **Principio di località temporale:** il programma tende ad accedere più volte alla stessa area di memoria. Una variabile viene usata più volte per riaggiornare il suo valore, leggere il suo valore, ecc. Per quanto riguarda le istruzioni, ci ritorno sopra grazie ai cicli (ricorda che sono presenti in praticamente tutti i programmi). In breve, *ritorno sempre sugli item a cui ho avuto accesso recentemente*. **Principio secondo cui se si accede a una determinata locazione di memoria, è molto probabile che vi si acceda di nuovo dopo poco tempo.**
- **Principio di località spaziale:** sono andato in una locazione di memoria. Per i dati ho una possibilità abbastanza alta di andare in una locazione immediatamente vicina (si pensi agli array). Per il codice questa località spaziale è garantita grazie alle sequenzialità delle istruzioni. Per questo motivo, il principio di località spaziale è *più forte per il codice che per i dati*. **Principio secondo cui se si accede a una determinata locazione di memoria, è molto probabile che si acceda alle locazioni vicine ad essa dopo poco tempo.**

Il principio di località viene sfruttato sfruttando la memoria di un calcolatore in forma gerarchica. *La gerarchia delle memorie consiste in una struttura che utilizza più livelli di memoria, ciascuno caratterizzato da una diversa velocità e dimensione: a parità di capacità, le memorie più veloci hanno un costo più elevato per singolo bit di quelle più lente, perciò di solito sono più piccole. La memoria più veloce è posta vicino al processore e quella più lenta, meno costosa, è posizionata più lontano.*

Organizzo delle memorie via via più piccole (e quindi più veloci) vicino al processore, così da dover accedere al disco solo quando non posso sfruttare la località.

L'obiettivo è di fornire all'utente una quantità di memoria pari a quella disponibile nella tecnologia più economica, consentendo allo stesso tempo una velocità di accesso pari a quella garantita dalla memoria più veloce.

In pratica, **quando è necessaria un'informazione, essa è cercata nei vari livelli di memoria a partire da quello più alto. Se il dato richiesto dal processore è presente in uno dei blocchi nel livello superiore, la richiesta ha successo (hit), altrimenti la richiesta fallisce (miss)** e quindi per trovare il blocco che contiene il dato occorre accedere a un livello inferiore della gerarchia (tutto ciò avviene a livello hardware).

*Quanto più ci si allontana dal processore, tanto più aumenta il tempo necessario per accedere ai dati.*

Terminologia importante:

- E' denominata *blocco di cache* (o *linea*) la *unità della copiatura*. Per sfruttare la località spaziale non può essere a singole word, poiché mi devo portare dietro anche ciò che sta intorno. Devo stare attento a non portare troppi dati in memoria: rischio di aumentare troppo il tempo di accesso alla memoria e di avere troppi dati che non mi servono nella cache. Quando il processore non trova il blocco che gli serve nella cache (avviene un miss) lo va a prendere nella memoria principale e lo copia nella cache
- E' chiamata *hit rate* la frequenza di hit, ovvero la *frequenza di successi nel trovare il dato nei primi livelli della gerarchia*; spesso l'hit rate viene utilizzato come indice delle prestazioni della gerarchia delle memorie
- E' chiamata *miss rate* la frequenza dei miss, che pari a  $1 - \text{hit rate}$  ed è la *frequenza di insuccessi nel trovare il dato nei primi livelli della gerarchia*

Perché il motivo principale che ha condotto all'organizzazione gerarchica delle memorie è l'aumento delle prestazioni, la velocità degli accessi è importante sia in caso di successo che in caso di fallimento.

- Il *tempo di hit* è il *tempo di accesso al livello superiore della gerarchia delle memorie*, e comprende anche il tempo necessario a stabilire se il tentativo di accesso si risolve in un successo o in un fallimento, cioè se produce una hit o un miss
- La *penalità di miss* è il *tempo necessario a sostituire un blocco del livello superiore con un nuovo blocco, caricato dal livello inferiore della gerarchia, e a trasferire i dati contenuti in questo blocco al processore*. Siccome il livello superiore è più piccolo ed è costruito utilizzando componenti più veloci, il tempo di hit sarà molto inferiore al tempo necessario ad accedere al secondo livello della gerarchia, che rappresenta la componente principale della penalità di miss

$$\text{hit ratio} = \frac{\text{hit}}{\text{accessi}}$$

Se l'hit ratio è molto alto, vuol dire che la memoria della memoria principale non influenzera il processore.

$$\text{miss ratio} = \frac{\text{miss accessi}}{\text{accessi}} = 1 - \text{hit ratio}$$

### Tecnologie delle memorie

Al giorno d'oggi vengono utilizzate quattro principali tecnologie per costruire la gerarchia delle memorie.

- **Static RAM (SRAM)**

- 0.5ns – 2.5ns, \$500 – \$1000 per GB

- **Dynamic RAM (DRAM)**

- 50ns – 70ns, \$3 – \$6 per GB

- **Magnetic disk**

- 5ms – 20ms, \$0.01 – \$0.02 per GB

Sintesi dello schema in figura:

- RAM statiche: sono fatte come i flip-flop. Servono un certo numero di transistor per memorizzare un numero di bit. Non è economica, quindi non posso creare memorie grandi, ma sono molto veloci. Raggiungono quasi la velocità dei tempi di esecuzione delle istruzioni del processore. Sono dette statiche perché i dati rimangono memorizzati finché è presente una carica elettrica.
- RAM dinamiche: utilizzate per la memoria principale. Sono dette dinamiche perché le celle di memoria non sono flip-flop ma capacità parassite (transistor) che vengono caricate o meno a seconda se sto caricando 0 o 1. Di tanto in tanto devo fare il refresh per mantenere le cariche memorizzate. Sono molto più lente dei processori, ma più economiche delle SRAM.
- Dischi magnetici: sono un milione di volte più lenti delle RAM, ma sono molto economiche, quindi sono adatte a creare memorie grandi.

Memoria ideale: tempo di accesso della SRAM e rapporto costo/GB del disco magnetico.

1. **DRAM (Dynamic Random Access Memory)** con cui è realizzata la memoria principale.

- Al contrario della memoria SRAM, in cui il dato rimane memorizzato per tutto il tempo in cui l'alimentazione è attiva, in una memoria DRAM *il dato viene memorizzato come una carica in un condensatore* e un solo transistor è sufficiente per leggere il dato o per sovrascriverlo, quindi sono molto più dense e meno costose delle SRAM.
- Nonostante negli anni si sia raggiunto un tempo d'accesso di 50-70 ns, le DRAM sono molto più lente delle SRAM. Tuttavia, le DRAM sono progettate non per essere le più veloci, ma per dimensioni sempre maggiori, cercando ovviamente di avere un compromesso a livelli di tempistiche d'accesso.
- Tuttavia, in questo tipo di memoria occorre rinfrescare periodicamente il dato (refresh), cioè occorre leggerne il contenuto e riscrivere, poiché l'informazione viene memorizzata in un condensatore, quindi non rimane indefinitamente. Le DRAM utilizzano un'architettura a due livelli che consente di fare il refresh non bit per bit, ma per un'intera riga, mediante un ciclo di lettura seguito immediatamente da un ciclo di riscrittura.
- Fattori per migliorare le prestazioni: (1) Le DRAM utilizzano un buffer di riga per leggere e aggiornare più word in parallelo; (2) Per migliorare ulteriormente l'interfaccia con i processori, alle DRAM è stato aggiunto il clock per cui vengono chiamate SDRAM: il vantaggio sta nel fatto che il clock elimina il tempo necessario a sincronizzare la memoria con il processore. L'aumento di velocità proviene dalla possibilità di trasferire gruppi di bit adiacenti a raffica (in burst) senza dover specificare altri indirizzi. La versione attuale più veloce è DDR (Double Data Rate, frequenza doppia dei dati). Questo significa che il trasferimento dei dati avviene sia sul fronte di salita che di discesa del clock, ottenendo così il doppio della larghezza di banda; (3) Per sostenere una tale larghezza di banda abbiamo bisogno di un'organizzazione intelligente all'interno della DRAM: possiamo organizzarle non più con un buffer di riga, ma in banchi (bank), ciascuno con il suo buffer di riga. Inviare un indirizzo a più banchi consente di leggere o scrivere simultaneamente. Per esempio, con quattro banchi occorre un unico tempo di accesso alla memoria, dopodiché si possono indirizzare sequenzialmente a rotazione i quattro banchi, ottenendo così il quadruplo della larghezza di banda. Questo schema di accesso sequenziale a rotazione viene chiamato indirizzamento con interleaving.
- Per i server vengono comunemente vendute delle schedine chiamate *DIMM (Dual Inline Memory Modules*, moduli di memoria in linea doppia, cioè fronte e retro). Il sottoinsieme di chip di una DIMM che condividono le linee di indirizzamento è detto rango della memoria.

Serie di trucchi per accedere alle DRAM:

- *Burst mode*: ricavo word successive da una riga con latenza ridotta
- *DDR (Double Data Rate) DRAM*: trasferisco i dati sia sul fronte di salita che sul fronte di discesa del clock (non è lo stesso clock del processore).
- *Quad data rate (QDR) DRAM*: ho degli input DDR diversi per input e output.

Generazioni di DRAM:

- Buffer delle righe: ci permette di fare il refresh e la lettura di molte word in parallelo
- DRAM sincrona: permette accessi consecutivi in burst senza aver bisogno di inviare indirizzi multipli
- DRAM banking: permette accessi simultanei a più DRAM

2. **SRAM (Static Random Access Memory)** con cui sono composti i livelli più vicini al processore (cache).

- Sono dei circuiti integrati organizzati come vettori di memoria che di solito hanno una sola porta di accesso che può fornire sia la lettura che la scrittura.
- Hanno uno stesso tempo di accesso per tutti i dati anche se i tempi di accesso in lettura e scrittura possono essere diversi.
- Non hanno bisogno del refresh, per cui il tempo di accesso è molto prossimo al periodo di clock
- Utilizzano da 6 a 8 transistor per bit per evitare che l'informazione possa essere disturbata quando viene letta.
- Hanno bisogno della potenza massima per mantenere la carica quando si trovano in modalità stand-by.
- In passato, la maggior parte dei PC e dei server utilizzavano chip di SRAM diversi per diversi livelli di cache. Oggi le cache di tutti i livelli sono integrate nel chip del processore, per cui il mercato delle singole SRAM è praticamente svanito.

3. **Memorie flash**, utilizzate nei dispositivi mobili. Sono un tipo di memoria a sola lettura, cancellabile elettricamente e programmabile.

- Non volatili.
- I bit delle memorie flash si deteriorano dopo un certo numero di scritture (dopo migliaia di accessi, i bit si danneggiano e perdo i contenuti; non sono quindi adatte per RAM o per dischi); per far fronte a questa limitazione, la maggior parte dei dispositivi che utilizzano memorie flash contiene un controllore che distribuisce le scritture consentite ri-mappando i blocchi di memoria che sono stati scritti più spesso sui blocchi che sono stati scritti meno di frequente -> *livellamento dell'usura*.
- Ci sono due tipi di memorie flash: *NOR flash*, che consente di accedere alla singola locazione di memoria, costosa, utilizzata nei sistemi embedded; *NAND flash*, che utilizza un accesso a blocchi, meno costosa, utilizzata per chiavette USB, ecc.

4. **Dischi magneticci**, impiegati per implementare il livello di memoria più capiente e lento della memoria.

- E' formato da un gruppo di dischi metallici, detti piatti, che ruotano saldati, ciascuno dei quali ricoperto da materiale magnetico registrabile. Per scrivere e leggere i dati su un disco rigido, al di sopra di ognuna delle superfici di ogni disco, è posizionato un braccio mobile contenente una piccola bobina elettromagnetica chiamata testina di lettura/scrittura. La superficie di ogni disco è divisa in cerchi concentrici chiamati tracce: ci sono tipicamente decine di migliaia di tracce per ogni lato del piatto. Ogni traccia è a sua volta suddivisa in migliaia di settori, i quali possono contenere da 512 a 4096 byte. Le testine di lettura/scrittura associate a ognuna delle superfici dei piatti sono collegate insieme e si muovono in modo congiuntamente, così che ogni testina si trovi in corrispondenza della stessa traccia su tutte le superfici. Il termine cilindro viene utilizzato per riferirsi a tutte le tracce dei diversi piatti che si trovano sotto le testine in un certo istante di tempo.
- Per accedere ai dati, il sistema operativo deve guidare il disco attraverso un processo in tre passi: (1) posizionare la testina sulla traccia giusta: l'operazione è detta seek e il tempo necessario per spostare la testina sopra la traccia desiderata viene chiamato tempo di ricerca (seek time); (2) quando la testina di lettura/scrittura ha raggiunto la traccia corretta, deve aspettare che il settore desiderato passi sotto. Questo tempo di attesa viene chiamato latenza di rotazione o ritardo di rotazione; (3) l'ultima componente del tempo di accesso al disco, il tempo di trasferimento, è il tempo impiegato per trasferire un blocco di bit; questo tempo è funzione della dimensione del settore, della velocità di rotazione e della densità di memorizzazione delle tracce.
- I dischi magneticci hanno un tempo di accesso superiore rispetto alle memorie flash perché sono dispositivi meccanici, ma hanno un costo per bit più basso perché offrono una grandissima capacità di memoria ad un costo moderato.
- I dischi magneticci non sono volatili, come le memorie flash, e non ci sono problemi di usura.
- Tuttavia, le memorie flash sono molto più robuste e quindi più adatte a sopportare gli urti inerenti all'utilizzo dei dispositivi mobili.



#### Accedere ai settori dei dischi magneticci

- Incoda il ritardo se altri accessi sono in attesa
- Cerca: muovo la testina
- Latenza della rotazione (mediante mezzo giro)
- Trasferimento dei dati (banda passante del dispositivo)
- Controllo dell'overhead

#### Esempio

Ipotesi: settore di 512 byte, 150000 rpm (rotation per minute, fascia alta), 4 ms il tempo di ricerca medio (per spostare la testina, tempo meccanismo -> tempo dei millisecondi invece che nanosecondi), 100 MB/s transfer rate, 0.2 ms di controllo dell'overhead, idle risk.

Normalmente l'rpm è molto più basso per ridurre la potenza necessaria per farli girare.

#### Problemi di prestazioni

I produttori danno il tempo di seek medio; in realtà, se gestisco bene il disco e mantengo le informazioni che mi servono nella stessa zona, non devo spostare di tanto le testine. Si fa attraverso la località e l'OS scheduling. Tutti gli OS hanno dei sistemi ottimizzati per l'allocazione dei dati su un disco.

Se il disco è pieno, l'OS è limitato nello spazio disponibile -> è un caso da evitare.

Controlli su disco: allocano i settori fisici sul disco.

SCSI, ATA, SATA: interfacce standard usate dai dischi. SCSI è pensato per i dischi a fascia alta.

I dischi hanno delle cache, specialmente quelli di fascia bassa, ovvero quelli lenti nella rotazione.

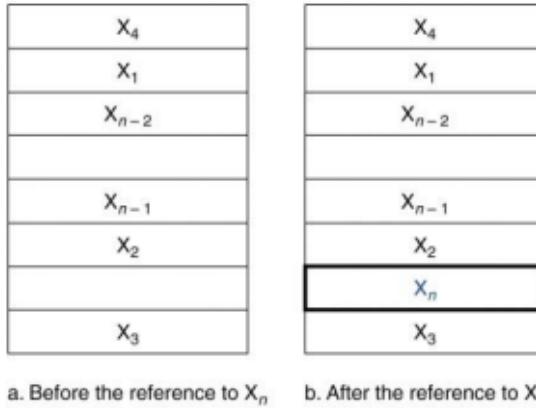
Evito così di attendere i giri per leggere i dati. Evito quindi il ritardo di seek e il delay.

## Principi base delle memorie cache

La memoria cache e il suo utilizzo sono generalmente nascosti agli occhi del programmatore.

Il termine cache viene usato per indicare *il livello della memoria gerarchica che si trova tra il processore e la memoria principale*.

Quando il processore richiede la parola X, che non è presente in cache, si produce un miss e la parola X viene prelevata dal livello inferiore della memoria e portata nella cache.



a. Before the reference to  $X_n$       b. After the reference to  $X_n$

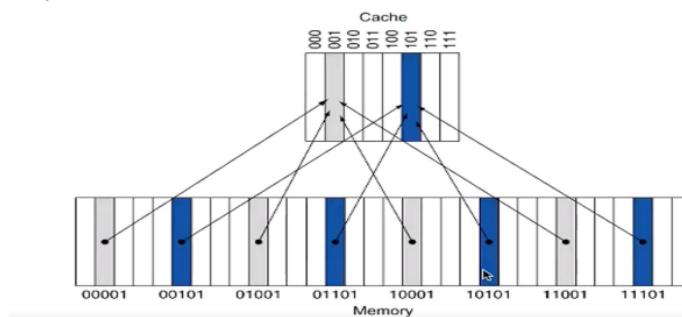
La cache sfrutta il *principio di località spaziale*, siccome è strutturata in modo da contenere blocchi possibilmente di indirizzi vicini (tipo nel caso di un array).

Ma come fa il calcolatore a sapere se un dato è presente nella cache? E se c'è, come facciamo a trovarlo? Le risposte sono collegate. La maniera più semplice per associare una locazione della cache a ogni parola della memoria principale consiste nel definire una corrispondenza univoca tra i due. Questa organizzazione della cache è detta *mappatura diretta*, dato che ogni locazione della memoria principale corrisponde, in modo univoco, a una locazione della cache.

- Abbiamo una sequenza di bit, detta *campo indice*, all'interno dell'indirizzo, che è utilizzata per selezionare il blocco della cache. Il numero di bit da cui sarà composta dipenderà dalla dimensione della cache. In una cache con una capacità di

$$2^n$$

bit sarà necessario un campo indice di  $n$  bit: una cache di 8 blocchi utilizzerà i 3 bit meno significativi per indirizzare il blocco, dato che  $8=s^3$ ; una cache di  $2^{10}$  bit ne utilizzerà 10.



Nell'immagine: esempio di cache a mappatura diretta contenente 8 elementi e corrispondenza tra gli indirizzi della memoria principale, riferiti alla parola e compresi tra 0 e 31, e le locazioni della cache.

Dato che la cache contiene 8 parole, un indirizzo  $X$  della memoria principale viene mappato sull'elemento  $X$  modulo 8 della cache a mappatura diretta. Cioè, i

$$\log_2(8) = 3$$

bit meno significativi dell'indirizzo vengono utilizzati come indice della cache. Perciò gli indirizzi 00001, 01001, 10001 e 11001 vengono mappati tutti sull'elemento 001 della cache, mentre gli indirizzi 00101, 01101, 10101 e 11101 vengono mappati tutti sull'elemento 101.

- Poiché ogni elemento della cache può contenere dati provenienti da diverse locazioni della memoria principale, come si riesce a capire se il dato presente nella cache corrisponde effettivamente alla parola desiderata? In altre parole, come faccio a sapere se la parola richiesta si trova nella cache oppure no?

E' possibile risolvere questo problema aggiungendo alla cache un insieme di bit che costituiscono il campo *tag*. I *tag* contengono le informazioni necessarie a verificare se una parola della cache corrisponde o meno alla parola cercata. Un tag contiene solamente la parte superiore dell'indirizzo della parola nella memoria principale, in particolare i bit dell'indirizzo che non vengono utilizzati come indice per individuare il blocco all'interno della cache. Per verificare se il dato contenuto nella cache sia effettivamente quello richiesto, dopo aver trovato il blocco corrispondente al campo indice, si confrontano i due tag e se coincidono abbiamo trovato il nostro dato.

- E' necessario anche disporre di un metodo per sapere quando un blocco della cache non contiene informazioni valide. Per esempio, quando un processore viene avviato, la cache è vuota e i numeri contenuti nei campi tag non hanno alcun significato. Anche dopo aver eseguito molte istruzioni, alcune delle locazioni della cache possono ancora essere vuote. Occorre perciò sapere quando il campo tag associato a queste locazioni deve essere ignorato. La procedura più comune consiste nell'aggiungere un *bit di validità* per indicare se il corrispondente elemento della cache contenga dati validi. Se il bit non è impostato a 1, la richiesta di lettura dell'elemento associato non può avere successo, perché il contenuto del blocco della cache sarebbe privo di significato.

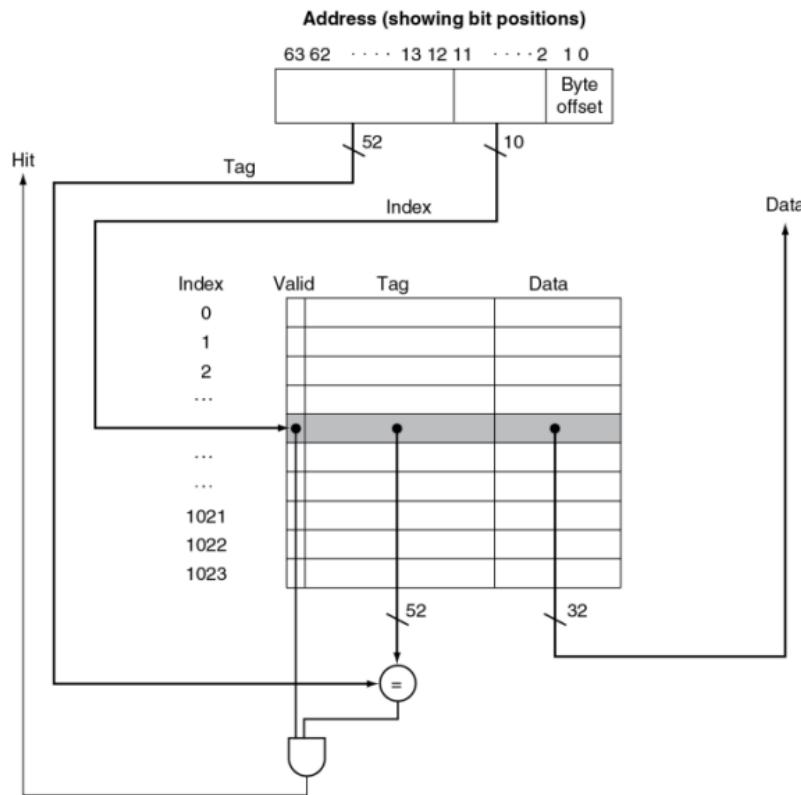
Riassumendo, l'indirizzo della memoria principale viene suddiviso in:

- un campo *tag*, che viene confrontato con il contenuto del campo tag del blocco della cache
- un campo *indice*, utilizzato per selezionare il blocco della cache. L'indice di un blocco della cache, insieme al contenuto del campo tag, specifica in maniera univoca l'indirizzo della memoria principale associato alla parola contenuta in quel blocco della cache
- byte offset (mi serve per identificare il numero di byte -> log in base 2 del numero di byte per blocco)

Il blocco della cache viene invece suddiviso in:

- un campo *tag* che viene confrontato con il tag dell'indirizzo della memoria principale
- bit di validità
- il dato

## Accesso alla cache



L'immagine indica com'è organizzato l'accesso alla cache.

Ho un processore con indirizzi a 64 bit.

Ci sono 4 byte per blocco (una sola word per blocco).

La cache è organizzata in 1024 blocchi per word.

Ho due bit di offset che servono per identificare il byte (quindi devo fare log in base 2 del numero di byte per poterli identificare tutti univocamente), i 10 bit successivi che sono l'indice (perché il log in base 2 di 20124 è 10) e i 52 bit finali sono il tag.

*Usiamo una AND per verificare se c'è un hit.*

*Per vedere se il tag corrisponde, devo usare un comparatore. L'output del comparatore è messo in input ad un AND insieme al bit di validità. Se il bit di validità è alto e il comparatore ha un output positivo, allora si è verificato un hit.*

L'accesso alla cache e l'indirizzamento sono tutte operazioni fatte in hardware.

## Considerazioni sulla grandezza dei blocchi della cache

Per sfruttare la località spaziale, *una cache deve avere blocchi di dimensioni superiori ad una parola e all'aumentare della dimensione dei blocchi diminuisce la frequenza di miss* e aumenta quindi l'efficienza della cache.

Tuttavia, *la frequenza di miss può tornare a crescere se la dimensione dei blocchi diventa troppo grande rispetto alla dimensione della cache, perché in questo caso il numero dei blocchi che possono essere memorizzati nella cache diventa piccolo, mentre cresce la competizione per occuparli.*

Inoltre, se i blocchi aumentano la loro grandezza, a parità di grandezza totale della cache, diminuiscono in numero e quindi c'è più competizione delle stesse posizioni.

Un problema ancora più serio, associato all'aumento della dimensione dei blocchi, è la *crescita del costo di un miss*: più grande è il blocco, più tempo ci vorrà a portarlo nella cache. Aumenta quindi la penalità di un miss, infatti, che è determinata dal tempo necessario a prelevare un blocco dal livello inferiore della gerarchia e a scriverlo nella cache, costituito da 2 parti:

1. la latenza per ottenere la prima parola del blocco
2. il tempo di trasferimento del resto del blocco

Il risultato è che l'incremento della penalità di miss ha un impatto superiore alla diminuzione della frequenza delle miss e quindi le prestazioni della cache si abbassano.

Come in ogni cosa in informatica, bisogna trovare un ragionevole compromesso nella scelta della grandezza dei blocchi.

## Gestione dei miss della cache in lettura

**Miss della cache:** la richiesta di un dato alla cache che non può essere soddisfatta perché il dato non è presente nella cache.

Vediamo ora in che modo l'unità di controllo gestisce i miss della cache: a seguito di un hit, la CPU processa normalmente; al miss della cache, invece, avviene ciò:

1. Ad ogni miss della cache possiamo mettere in stallo l'intero processore. Se l'accesso a un'istruzione si traduce in un miss, il contenuto del registro istruzioni non sarà più valido
2. Si cerca quindi il blocco da gerarchie di memoria inferiori
3. Una volta ottenuto, si fa ripartire l'esecuzione.

Riassumendo i passi:

- *Inviare il valore originale del program counter (PC corrente - 4) alla memoria.* Dal momento che il PC viene incrementato di 4 nel primo ciclo di clock di esecuzione, l'indirizzo dell'istruzione che ha generato il miss sarà uguale al contenuto del PC meno 4
- *Comandare alla memoria principale di eseguire un'operazione di lettura e attendere che la memoria completi la lettura;* si deve poi attendere che la memoria abbia terminato la lettura, dato che l'accesso richiede più cicli di clock
- Infine, occorre attendere che la parola contenente l'istruzione desiderata venga scritta nel blocco della cache, aggiornare il campo tag corrispondente scrivendovi i bit più significativi dell'indirizzo presi direttamente dall'ALU e impostare il bit di validità a 1
- *Far ripartire l'esecuzione dell'istruzione dall'inizio,* ripetendo la fase di fetch che questa volta troverà l'istruzione all'interno della cache

Se il miss riguarda l'istruzione, faccio ripartire il fetch. Se il miss si è verificato quando c'è un accesso ai dati, una volta che la cache è stata caricata, posso completare l'accesso ai dati.

In ogni caso, la pipeline si blocca per decine di colpi di clock.

## La gestione della scrittura

La scrittura funziona in modo un po' diverso. Supponiamo che un'operazione di store scriva il dato solamente nella cache dei dati, senza modificare la memoria principale; al termine della scrittura, la memoria principale avrebbe un contenuto diverso da quello della cache. In questo caso, si dice che la memoria e la cache sono incoerenti. La maniera più semplice per conservare la coerenza tra memoria e cache consiste nello scrivere sempre il dato in entrambe le memorie.

### Write-through

*Write-through: uno schema secondo cui un elemento viene scritto sia in cache sia nel livello inferiore della gerarchia delle memorie, assicurando così che i dati presenti nelle due memorie siano sempre coerenti tra loro.*

Lo schema più semplice viene chiamato write-through, in cui i dati vengono aggiornati simultaneamente nella cache e nella memoria. Viene utilizzato quando non ci sono scritture frequenti nella cache (il numero di operazioni di scrittura è inferiore) e semplice da implementare.

Dopo aver caricato il blocco e averlo scritto in cache, questa parola viene scritta anche nella memoria principale utilizzando il suo indirizzo completo.

Questo meccanismo però non offre buone prestazioni quando ci sono scritture frequenti nella cache. L'utilizzo di questo schema comporta che a ogni scrittura la parola venga salvata anche nella memoria principale. Questa operazione richiede molto tempo, almeno 100 cicli di clock del processore e può quindi rallentare il sistema in maniera considerevole.

Una possibile soluzione consiste nell'utilizzare una memoria tampone, chiamata *buffer di scrittura*, che *memorizza i dati in attesa che essi vengano scritti in memoria*: dopo aver salvato il dato nella cache e nel buffer di scrittura, il processore può proseguire l'esecuzione. Una volta completata la scrittura di un dato nella memoria principale, il corrispondente spazio nel buffer di scrittura viene liberato.

Anche questo metodo ha dei difetti: se il buffer di scrittura è pieno e il processore deve eseguire un'operazione di scrittura, il processore viene messo in stallo finché non si libera spazio nel buffer.

Ovviamente, se la velocità con cui la memoria principale completa le scritture è inferiore a quella con cui il processore genera i dati da scrivere, per quanto grande possa essere il buffer di scrittura, prima o poi il processore dovrà comunque essere messo in stallo, poiché i dati da scrivere vengono generati più velocemente di quanto il sistema di memoria sia in grado di accettarli.

### Write-back

*Write-back: uno schema in cui un elemento viene scritto solo nel blocco di cache; il blocco corrispondente del livello inferiore della gerarchia viene scritto solamente quando il blocco della cache viene sostituito.*

Per ridurre il numero di stalli, lo schema alternativo al write-through è chiamato write-back, nel quale i dati vengono aggiornati solo nella cache e aggiornati nella memoria in un secondo momento. I dati vengono aggiornati in memoria solo quando il blocco della cache è pronto per essere sostituito.

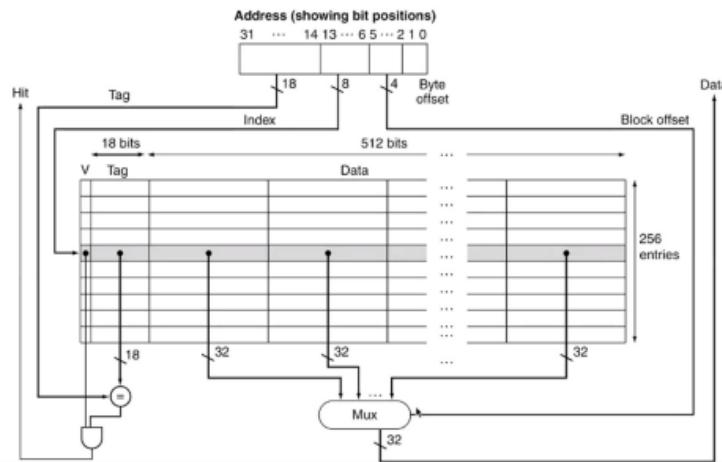
Questo schema può produrre un miglioramento delle prestazioni, ma è comunque più complesso da implementare.

Ogni blocco nella cache necessita di un bit per indicare se i dati presenti nella cache sono stati modificati (dirty) o non modificati (clean). Se è clean non è necessario scriverlo in memoria.

E' progettato per ridurre le operazioni di scrittura su una memoria.

## Un esempio di memoria cache: il processore FastMATH Intrinsity

Microprocessore embedded veloce che utilizza l'architettura MIPS e una semplice implementazione della cache.



La dimensione dell'indirizzo in questo calcolatore è di soli 32 bit.

Questo processore ha una pipeline a 12 stadi. Quando opera a pieno regime, il processore può chiedere sia una parola di dati che una di istruzioni ad ogni ciclo di clock.

Per soddisfare queste richieste senza generare stalli nella pipeline, vengono utilizzate una cache di dati e una cache istruzioni separate. Ciascuna cache è di 16 KiB, ossia 4096 parole, con 16 parole per blocco.

Per le operazioni di scrittura, il FastMATH Intrinsic offre sia la modalità write-back che la modalità write-through, lasciando al sistema operativo il compito di decidere quale strategia utilizzare per l'applicazione corrente.

SPEC2000 (benchmark) miss rates:

- I-cache: 0.4%
- D-cache: 11.4 %
- Media pesata: 3.2%

In generale, le cache funzionano meglio con le istruzioni che con i dati (sempre per la struttura sequenziale del nostro codice).

Indirizzi a 32 bit:

- 2 bit di byte offset (per individuare il byte all'interno della word). Se accedo ad una word multipla di 4 questi bit saranno uguali a 0
- 4 bit di block offset (log in base 2 di 16, dato che ci sono 16 word per blocco)
- 8 bit per l'indice (log in base 2 di 256, i numero dei blocchi nella cache)
- 18 bit come tag

Con l'indice si accede nella cache per capire la linea dove vedere e accedervi. Uso il comparatore per verificare che il tag è lo stesso e faccio l'AND con il bit di validità per verificare se è avvenuto un hit.

Utilizzo poi un multiplexer per determinare quale delle 16 word devo pescare, pilotato dal block offset. Il multiplexer ha come output i dati.

N.B.: *tutto questo è fatto in hardware!*

## Cache di supporto per la memoria principale

Uso le DRAM per la memoria principale.

- Dimensione prefissata (fixed width) -> 1 word
- Connesso da un bus con una dimensione prefissata alla CPU con un proprio clock (clocked bus). Non è il clock del processore. Tipicamente il clock del bus è più lento del clock della CPU, per colpa delle capacità parassite che deformano il segnale

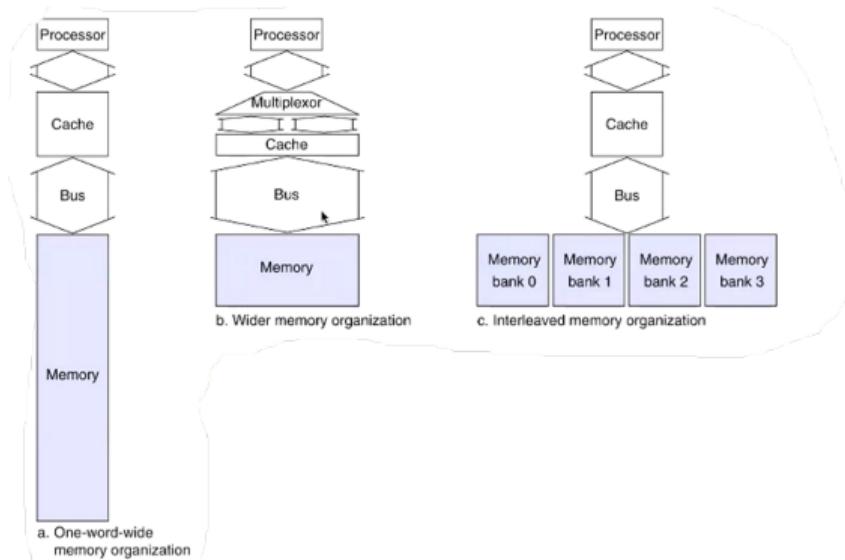
Esempio di lettura di un blocco dalla cache:

- 1 ciclo di bus per il trasferimento dell'indirizzo
- 15 cicli per l'accesso alla DRAM (posso aspettare anche fino a 20/45 colpi di clock del processore, per far capire quanto è lenta la DRAM)
- 1 ciclo per il trasferimento dei dati

Per un blocco di 4 word e una DRAM spessa 1 word:

- miss penalty:  $1 + 4 \times 15 + 4 \times 1 = 65$  cicli di bus
- Larghezza di banda passante = 16 bytes / 65 cicli = 0.25 byte/cicli

## Incrementare la banda passante



#### A) Organizzazione della memoria one-word

B) Organizzazione più larga della memoria (4-word): memoria capace di leggere 128 (32 4) bit alla volta

- miss penalty =  $1+15+1 = 17$  cicli di bus
- banda passante =  $16 \text{ byte}/17 \text{ cicli} = 0.94 \text{ Byte/cicli}$

Svantaggio: difficile e costoso da realizzare

C) Interleaving della memoria\*: ho 4 banche di memoria

- miss penalty =  $1+15+4 \times 1 = 20$  cicli di bus
- banda passante =  $16 \text{ byte}/20 \text{ cicli} = 0.8 \text{ byte/cicli}$

Prestazioni peggiori della memoria larga ma molto più semplice da realizzare. E' quella che viene utilizzata nella pratica

## Come misurare e migliorare le prestazioni di una cache

Esistono due diverse tecniche per migliorare le prestazioni di una cache:

1. riduzione della frequenza dei miss, attraverso la riduzione della probabilità che due differenti blocchi di memoria principale entrino in conflitto per la stessa locazione della cache
2. riduzione della penalità dei miss, introducendo nella gerarchia un livello aggiuntivo: cache multilivello

Le prestazioni di una cache si misurano così:

$$\begin{aligned} \text{Cicli di stall di memoria} &= \frac{n \text{ accessi in memoria}}{\text{programma}} * \text{miss rate (frequenza di miss)} * \text{miss penalty} \\ &= \frac{\text{istruzioni}}{\text{programma}} * \frac{n \text{ miss}}{\text{istruzioni}} * \text{miss penalty} \end{aligned}$$

Fino ad ora abbiamo considerato il fattore di hit come un fattore non significativo nel calcolo delle prestazioni di una cache. Chiaramente, se il tempo di hit aumentasse, il tempo totale richiesto per accedere a una parola del sistema di memoria aumenterebbe, facendo crescere quindi verosimilmente la durata del periodo di clock del processore. I progettisti utilizzano a volte il tempo medio di accesso alla memoria (AMAT, Average Memory Access Time).

$$AMAT = \text{durata di una hit} + \text{frequenza di miss} * \text{penalità di miss}$$

Tutto ciò fa comprendere come sia fondamentale tener conto della struttura della cache quando si progetta e si migliora un processore, siccome esso sarà fortemente influenzato dalla memoria.

### Sintesi delle prestazioni

Più aumentano le prestazioni della CPU (parallelismo, pipeline, clock ad alta frequenza), più esse diventano veloci.

I tempi di accesso alla memoria negli anni scende di poco, per colpa delle tecnologie delle RAM.

Col tempo il miss penalty diventa sempre più significativo.

Se decremento il CPI spendo una quantità di tempo maggiore nello stall della memoria.

Più aumenta la frequenza di clock, più gli stalli producono un effetto maggiore sulle prestazioni.

Morale: non posso ignorare il comportamento della cache per valutare le prestazioni del sistema.

### Dove si trova il sistema di cache?

Storicamente era tutto esterno, solo il primo livello era interno al processore.

A seconda della scheda madre le prestazioni del sistema era migliore o peggiore.

Attualmente le cache sono tutte interne al processore.

Ogni core ha la sua cache di primo livello, ognuno di loro ha una cache di 2° livello e tutti hanno una cache di 3° livello in comune.

La scheda madre permette l'accesso alla memoria (anche collegamenti alle altre unità e al sistema video), quindi ha comunque un effetto sulle prestazioni.

Se sono scarse possono comunque penalizzare le prestazioni del processore.

**Fregatura della cache directed mapped:** se per caso scrivo un programma in cui mi servono gli indirizzi compresi tutti nelle "zone blu" (stessa distanza di potenze di 2), a causa del meccanismo del direct mapping, questi blocchi non possono stare contemporaneamente nella cache, dato

che vanno a collidere nella stessa posizione, anche se c'è lo spazio per farli entrare tutti. Il resto della cache rimane inutilizzato mentre c'è un via vai dai blocchi nella stessa posizione.

Un esempio di questo caso è l'accesso agli array.

Andrei meglio se ci fosse un mapping più libero, anche se questo renderà l'hardware più complicato.

## Posizionamento più flessibile dei blocchi

Fino ad ora abbiamo considerato lo schema a mappatura diretta, siccome è la più semplice da implementare.

Tuttavia, ha dei difetti: alcune parti della memoria saranno inutilizzate e altre saranno utilizzate troppo spesso. In realtà esistono altri schemi di posizionamento dei blocchi.

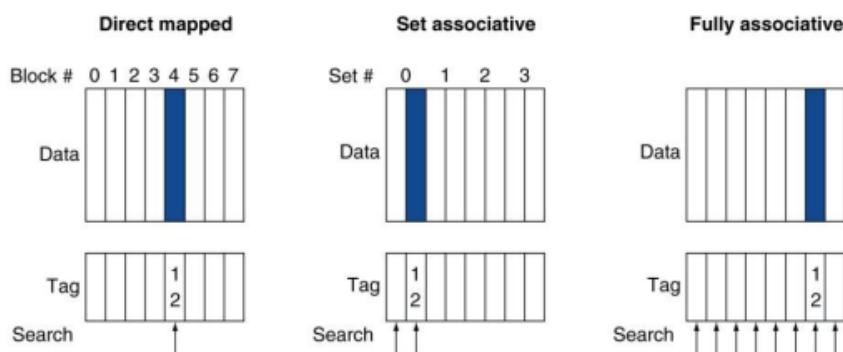
- **Cache completamente associativa:** una cache in cui *un blocco della memoria principale può essere scritto o letto da qualsiasi blocco della cache*. Per trovare un blocco in una cache di questo tipo, la ricerca dev'essere effettuata su tutti gli elementi della cache, dato che il blocco può trovarsi in una qualsiasi posizione. Per renderla efficace, *la ricerca viene svolta in parallelo utilizzando un comparatore per ogni blocco della cache*: mi serve quindi avere un comparatore per ogni posizione al posto di averne uno globale (molto costoso). E' più flessibile della mappatura diretta, tuttavia tali comparatori incrementano significativamente il costo dell'hardware, rendendo realizzabile tale schema solo per cache con un numero ridotto di blocchi
- **Cache set-associativa a n-vie:** adottano schemi intermedi tra una mappatura diretta e una cache completamente associativa. Tale cache è suddivisa in insiemi. All'interno di ogni insieme c'è associaattività totale. Quindi, non vi è la libertà assoluta di inserire il blocco della memoria in qualsiasi posizione della cache, ma tra un numero prefissato di posizioni n all'interno dell'insieme: per questo viene chiamata cache a n-vie. Ad esempio, in una cache a 4 vie bisognerà cercare il blocco in solo 4 posizioni (non in una sola, come nella mappatura diretta e non in tutte, come nella cache completamente associativa). Se avessi un set con tanti blocchi, ho una cache completamente associativa (Fully associative). Se ho tanti set con un solo blocco (one-way associative) ho il direct mapped.

N.B.: nella direct mapped, sulla base dei bit che compongono l'indirizzo, sui bit bassi vado ad individuare la posizione dove potrebbe stare quello che sto cercando nella cache. Controllo poi i bit alti (il tag) e il bit di validità per vedere se corrisponde effettivamente a quello che stavo cercando nella DRAM (hit).

Se la cache non è direct mapped, non posso utilizzare i bit bassi per individuare la posizione. Utilizzo tutto l'indirizzo (tranne i primissimi bit inferiori che sono utilizzati per il byte offset) per vedere se c'è un confronto positivo tra questo tag grande e uno dei comparatori.

Nella cache set-associativa ad n vie rappresento l'insieme con i bit più bassi, nei quali ho due posizioni possibili (distinti dal tag). Faccio (numero di blocco) % (#set nella cache) per individuare l'insieme, faccio poi delle n ricerche parallele.

![[Pasted image 20230505191936.png | 1/2]]



A parità di grandezza della cache, queste sono le possibili mappature della cache (ovviamente le più convenienti sono quelle set-associative).

Quando si verifica un miss in una cache a mappatura diretta, il blocco richiesto può essere scritto in un'unica posizione e quello che occupava precedentemente questa posizione deve essere sostituito.

In una cache associativa, possiamo scegliere dove scrivere il blocco richiesto e quindi dobbiamo decidere quale sia il blocco da sostituire.

In una cache completamente associativa, tutti i blocchi costituiscono un potenziale candidato per la sostituzione. Se la cache è set-associativa, possiamo scegliere tra i blocchi contenuti nello stesso insieme (linea) della cache: si sfrutta uno schema di sostituzione detto *utilizzo meno recente (LRU, Least Recently Used)*, in cui il blocco sostituito è quello che è rimasto inutilizzato più a lungo. Tuttavia, è difficile da implementare. Si riescono ad ottenere simili risultati scegliendo Random ed è molto semplice da implementare.

## Esempio di associaattività

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

Devo accedere a questi blocchi in sequenza: 0, 8, 0, 6, 8.

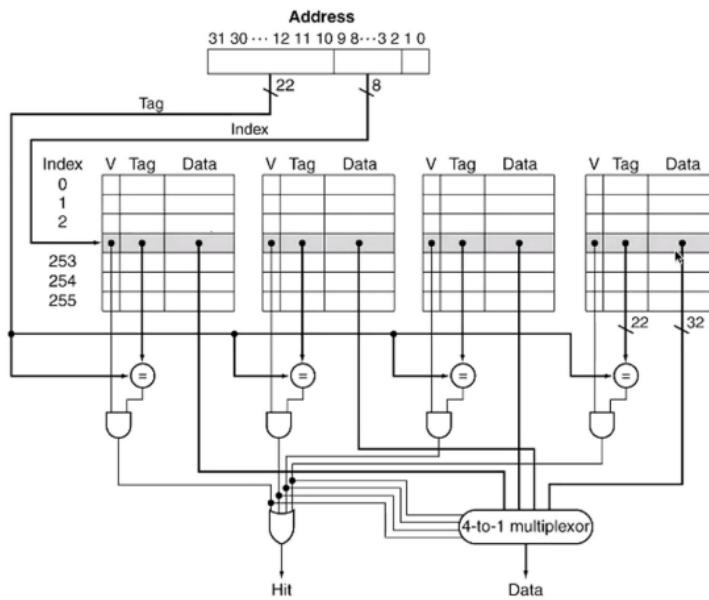
La cache, che nell'esempio è Direct mapped, va a disturbare sempre la stessa posizione e ottengo tutti miss.

Con una set associativa a 2 vie, dato che 0 e 8 possono coesistere, ottengo un hit.

Nella fully associative, ho un hit in più perché può starci pure il 6.

Quindi, *più aumenta l'associatività, più diminuisco il miss rate*. Però, l'associatività non porta sempre un miglioramento che giustifica il costo.

## Organizzazione di una cache set-associativa a 4 vie



Ogni set ha 256 indirizzi.

- 2 bit di offset (posizione all'interno della word a 32 bit)
  - 8 indici per individuare la posizione della cache ( $\log_2 256 = 8$ )
  - 22 bit che mi restano  $\rightarrow$  tag
- 4-way: 4 comparatori e 4 porte AND. Per verificare l'hit mi serve anche una porta OR che ha come ingressi gli output delle porte AND.  
La ricerca è fatta in parallelo sulle 4 vie.

## Politica di rimpiazzo

Che succede se ho un hit in tutte e 4 le vie e devo aggiungere un nuovo blocco nella cache in quella posizione?

Non ho bisogno di una replacement policy nella direct mapped, essendo la posizione del blocco predestinata.

Nella set-associativa:

- se c'è un'entry non valida ho una posizione libera da usare
  - altrimenti devo scegliere una posizione nei set
- Quello che posso scegliere di buttare fuori dalla cache, dato che l'obiettivo è ridurre i cache miss, può essere quello che per più tempo non mi servirà, che è una cosa che non posso sapere, perché non so gli accessi futuri in memoria (contiamo anche che stiamo lavorando in hardware).

### Strategia LRU (Least Recently Used)

Scelgo il blocco che non ho utilizzato da più tempo. Non è una garanzia ma è funzionale. E' semplice da implementare per i 2-way (basta ricordarsi se è il bit di sinistra o quello di destra quello più "antico"), maneggiabile per i 4-way, ma troppo difficile se si va oltre. Tutto ciò sempre perché si fa tutto in hardware.

### Strategia Random (o di rimpiazzo delle cache)

Dà approssimativamente le stesse prestazioni di LRU per associazività alte, ma con spreco di componenti minore.

N.B.: i processori del laptop sono diversi dal processore di un computer da scrivania, anche se hanno lo stesso nome. *La batteria si danneggia!*

## Cache multilivello

Le cache multilivello costituiscono una tecnica che può essere utilizzata per ridurre la penalità di miss.

Un primo livello si trova a stretto contatto con la CPU. Un secondo livello viene utilizzato quando si verifica un miss all'interno della cache primaria.

Se il secondo livello della cache contiene il dato desiderato, la penalità di miss per il primo livello di cache sarà essenzialmente il tempo di accesso al secondo livello di cache, il quale è molto inferiore del tempo di accesso alla memoria principale.

Le considerazioni da fare in fase di progettazione sono molto diverse perché la presenza di una seconda cache modifica le specifiche (una cache a due livelli permette di progettare la cache primaria focalizzandosi sulla minimizzazione del tempo di hit e permette di progettare la cache secondaria focalizzandosi sulla frequenza dei miss per evitare l'accesso alla memoria principale). La cache primaria di una cache multilivello è spesso di dimensioni ridotte ma più veloce e la cache secondaria, invece, è più grande e più lenta, ma comunque più veloce della memoria principale.

Albori delle cache: memorie piccolissime (64 KB) attaccate al processore.

Via via le cache sono state integrate all'interno del chip del processore e differenziate per tipo.

In tutti i processori che fanno uso di pipelining abbiamo una cache principale attaccata alla CPU, piccola (16 KB), ma veloce. Può anche essere separata tra dati e istruzioni.

Abbiamo poi una cache di livello 2 che serve per i miss della cache principale. È più grande, più lenta, ma comunque più veloce della memoria principale.

Al giorno d'oggi ci sono anche cache di livello 3 e 4.

Nel momento in cui qualcosa viene caricato dalla memoria, questo passa per tutti i livelli di cache?

Dipende dall'inclusività: se voglio aumentare l'aggressività del sistema posso evitare di copiare gli elementi dalla cache L2 a quello di L3 per fare in modo di poter memorizzare più dati. Questo modello non inclusivo, ideato per la prima volta da AMD, fa in modo che l'accesso alla memoria esterna sia davvero l'ultima spiaggia.

### Considerazioni sulle cache multilivello

1. Cache principale: è progettata per avere un hit time minimo e servire il prima possibile la memoria principale. I tempi di accesso sono, quindi, il focus
2. Cache secondaria: devo evitare a tutti i costi gli accessi alla memoria principale. Mi concentro quindi su un miss rate. L'hit rate ha così un impatto minore

Quindi, la cache L1 è più piccola di una cache singola ed è anche più piccola di quella L2.

## Interazioni con CPU avanzate

Le CPU superscalari eseguono uno schema out-of-order, così da eseguire altre istruzioni in caso di una cache miss (Hyper Threading = possono venire anche da altri programmi).

- C'è una coda di scritture in attesa che restano nell'unità load/store
- Le istruzioni dipendenti aspettano nelle stazioni di prenotazione, quelle indipendenti possono andare avanti

Così la CPU continua ad operare.

Gli effetti dei miss dipendono dal flusso dei dati del programma. Significa che è praticamente impossibile sapere quanto tempo ci vuole ad eseguire un programma. L'unico modo per fare analisi è una simulazione del sistema nel momento in cui i progettisti ideano una nuova microarchitettura, per vedere se la maggior parte dei programmi gira meglio.

## Interazioni con il software

Il comportamento delle cache dipende molto da com'è scritto il codice.

A seconda che il programma procede con un pattern di accesso alla memoria sequenziale o un pattern "a pettine", avrà un numero diverso di miss: pochi nel primo caso, tanti nel secondo.

E' il programmatore che deve capire cosa sta facendo e che esistono le cache. Deve quindi capire il pattern di accesso alla memoria.

Non possiamo vivere in un mondo astratto dove non pensiamo alla memoria. Se non sappiamo come funziona una cache, non sapremo mai il pattern di accesso da utilizzare.

### Codice cache friendly

Anche l'algoritmo di un programma si può modificare per migliorare il comportamento della cache.

Prendiamo l'esempio seguente:

#### ■ cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

sappiamo che il C lavora a righe (è per questo che incrementando o decrementando un puntatore si può scorrere, ad esempio, un array).

Nell'esempio abbiamo un codice che somma i numeri di una matrice con M righe e N colonne.

Il primo codice somma prima il contenuto di una riga e poi passa alla riga successiva; il secondo codice somma prima il contenuto di una colonna e poi passa a quella successiva.

Il più conveniente, secondo il principio di località spaziale, è il primo: infatti, occorrerà accedere alla memoria una volta per riga, mentre nel secondo occorrerà accedere alla memoria per ogni elemento della colonna.

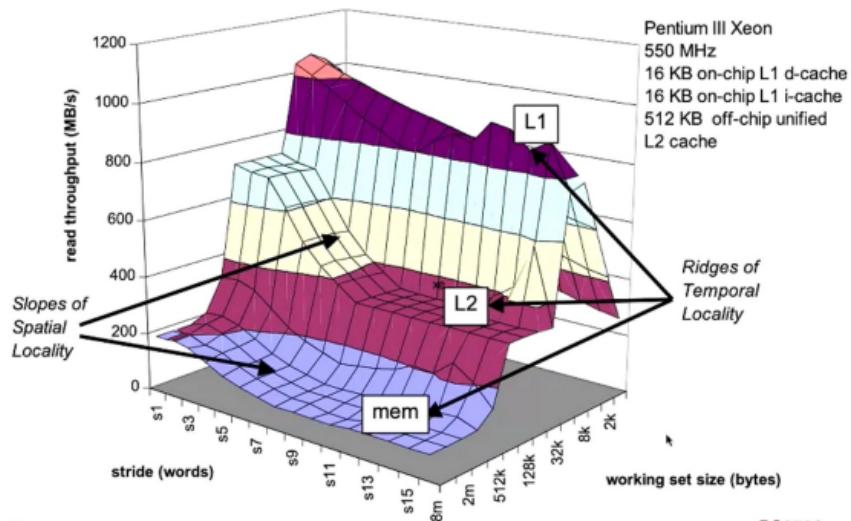
**Gli array C allocano in row-major order -> ogni riga è in locazioni di memoria contigue.**

Esempio 1: ogni volta che accedo ad un elemento mi porto dietro anche i tre elementi successivi. Il primo sarà un miss, i prossimi tre un hit.

Esempio 2: dato che l'accesso successivo non è affianco a quello corrente, non riesco a sfruttare la località spaziale e ho un miss rate del 100%.

Un programma sintatticamente corretto e che fa quello che deve fare è sbagliatissimo per l'accesso alla memoria.

## La montagna di memoria



Diagrammi che rappresentano la misurazione della banda passante di accesso alla memoria. Più è alta, più la banda passante è elevata.

Asse x: stride

Asse y: working set (parte della memoria con cui sto lavorando)

Asse z: banda passante (read throughput MB/s)

I livelli più alti si ottengono con Stride-1 e con un working set piccolo.

Ai piani alti posso vedere che tutto quello che mi serve si trova in L1. Non appena il working set diventa un po' più grande, si crolla verso la zona in cui i dati vanno a finire nella L2 (che ha una banda passante meno elevata ed è meno sensibile agli stride più grandi).

Quando il working set è ancora più grande, la banda passante si riduce fortemente.

L'obiettivo è collocarsi nella parte alta della montagna, lavorando con working set piccoli e stride pari ad 1.

### Osservazioni conclusive

Il programmatore deve rendersi conto di come sono organizzate le strutture di dati, prestando attenzione ai loop innestati.

Il blocking (lavoro su sotto-insiemi minori, andando meglio con le cache) è una tecnica che aumenta molto le prestazioni.

Tutti i sistemi favoriscono un codice "cache friendly".

Le prestazioni ottimali dipendono dalla piattaforma specifica: devo conoscere le dimensioni delle cache, le grandezze delle linee, l'associatività.

In generale, devo scrivere codice cache friendly: avere un working set piccolo sfruttando la località temporale e usare piccoli stride per sfruttare la località spaziale.

## Macchine virtuali

Tradicionalmente, sopra al processore gira il sistema operativo che accede alle sue risorse.

Per motivi storici, negli anni '70, si inventò l'idea delle macchine virtuali: sopra l'hardware c'è un software detto *HyperVisor* (detto anche *Virtual Machine Monitor*, VMM), il quale dà l'illusione ai vari OS che stanno sopra di avere a disposizione l'hardware completo.

Fondamentalmente, *isolo gli OS che lavorano per parte loro*. Ogni OS è isolato l'uno dall'altro; posso quindi far girare OS diversi sulla stessa macchina.

Questo concetto ritornò negli anni '90 poiché i processori erano tornati così veloci da riuscire a gestire più OS resi disponibili.

Ci sono stati degli esperimenti interessanti, tipo come proteggere il processore dai crash.

Può essere utile, ad esempio, per consolidare un unico sistema che fa l'hosting di tanti web server.

Il vantaggio di questa emulazione è l'*isolamento dalle risorse*: ogni OS lavora per conto proprio.

Evito problemi di sicurezza e affidabilità, insieme alla condivisione delle risorse.

E' il VMM a gestire i device di I/O, emulando un device I/O virtuale generico per il guest OS.

## Protezione alla memoria

Oggiorno, la *funzione più importante della memoria virtuale* è forse quella di *consentire a più processi di condividere un'unica memoria principale, fornendo allo stesso tempo un meccanismo di protezione della memoria ai diversi processi e al sistema operativo*.

Il meccanismo di protezione deve garantire che, anche se più processi condividono la stessa memoria principale, non sia possibile per un processo scrivere nello spazio d'indirizzamento di un altro processo utente o nello spazio riservato al sistema operativo, di fatto non rispettando le regole in modo intenzionale o accidentale.

Il *bit di accesso in scrittura* del TLB permette di proteggere la pagina associata, evitando che questa possa essere scritta. Senza questo livello di protezione, i virus dei calcolatori sarebbero ancora più diffusi.

Occorre anche evitare che un processo legga i dati di un altro processo.

Se si permetti di condividere la memoria principale, occorre fornire ad ogni processo la possibilità di proteggere i propri dati dalla lettura e scrittura da parte di altri processi.

Ricordiamo che ciascun processo ha il proprio spazio di indirizzamento virtuale; perciò se il sistema operativo organizza le tabelle delle pagine in modo che pagine virtuali indipendenti corrispondano a pagine fisiche differenti, un processo non sarà in grado di accedere ai dati di un altro processo. Ovviamente, questo richiede anche che un processo utente non possa alterare la traduzione delle pagine virtuali contenuta nella tabella

delle pagine.

Il sistema operativo può garantire la protezione se può impedire ai processi utente di modificare la propria tabella delle pagine; tuttavia, il sistema operativo deve poter verificare le tabelle delle pagine. Inserendo le tabelle delle pagine nello spazio di indirizzamento protetto dal sistema operativo, si possono soddisfare entrambi i requisiti.

Quando i processi vogliono condividere delle informazioni, devono essere assistiti dal sistema operativo, dato che accedere alle informazioni di un altro processo implica cambiare la tabella delle pagine per il processo che effettua l'accesso. Si può utilizzare il bit di accesso in scrittura per limitare la condivisione alla sola lettura; tale bit, come il resto della tabella delle pagine, può essere modificato solo dal sistema operativo.

Per consentire a un processo, per esempio P1, di leggere una pagina del processo P2, P2 dovrebbe chiedere al sistema operativo di creare un nuovo elemento nella tabella delle pagine contenente una pagina virtuale nello spazio di indirizzamento di P1; questo nuovo elemento punterà alla pagina fisica che P2 vuole condividere. Il sistema operativo può usare il bit di protezione in scrittura per impedire a P1 di scrivere dei dati in questa pagina fisica, se questo è il desiderio di P2. Tutti i bit che determinano i diritti di accesso a una pagina devono essere contenuti sia nella tabella delle pagine sia nel TLB, dato che la tabella delle pagine viene consultata solo in corrispondenza dei miss del TLB.

**TLB (Translation Lookaside Buffer):** una cache che tiene traccia degli indirizzi virtuali tradotti più di recente per evitare l'accesso alla tabella delle pagine.

Riformulazione del concetto di protezione alla memoria:

Al giorno d'oggi ci sono più processi (programmi in esecuzione) allo stesso tempo. Nella presenza di più utenti ci potrebbe essere anche programmi in esecuzione appartenenti ad utenti diversi. In breve, task differenti condividono parte dello spazio degli indirizzi virtuali. Un programma potrebbe essere in grado di leggere dei dati da altri programmi, sia nell'ambito dello stesso utente sia nell'ambito di utenti multipli. Deve quindi essere vietato far modificare l'area di memoria dell'O.S. ad un programma. Pensa al meccanismo "segmentation call" nei programmi C quando si prova ad usare un puntatore per accedere ad un'area di memoria in cui non dovremmo mettere mano (come quando il puntatore non è inizializzato e ha come indirizzo la posizione 0). Ogni indirizzo di memoria dovrebbe essere modificato (via hardware) per capire da che programma proviene. Durante l'esecuzione di istruzione c'è almeno un accesso alla memoria (possibilmente 2 se è un'operazione load/store). Essendo nel corso d'esecuzione di un'istruzione, mi serve qualcosa a livello più basso (hardware) per vedere se l'indirizzo di memoria appartiene all'area di memoria dedicata al programma in esecuzione. Se non accade c'è un'eccezione (*segmentation call*) e il programma viene abortito. Serve quindi un supporto hardware per la protezione dell'O.S. Così non posso modificare l'O.S.

Oltre alla protezione di memoria, nasce l'esigenza di avere una differenziazione a due livelli del processeore: ho un set di istruzioni disponibili a tutti i tipi di programmi e un set di istruzioni che devono essere ristrette al solo sistema operativo (Dual mode). Queste sono tipicamente istruzioni ( dette privilegiate) che hanno effetti globali drastici o che riguardano le informazioni di stato. Queste istruzioni privilegiate sono disponibili solo in kernel mode (o modalità supervisore). Un programma che tenta di eseguire queste istruzioni viene immediatamente abortito. Per passare da modalità utente a modalità supervisore si utilizzano le eccezioni (System call exception). La Dual mode non era presente nei primi processori: un programma poteva quindi scrivere abusivamente contro l'O.S. Era questa l'origine degli Blue Screen of Death.

## Schema comune per la gerarchia delle memorie

Principi comuni si applicano a tutti i livelli di questa gerarchia basati sulla nozione del caching.

A tutti i livelli abbiamo:

- delle problematiche di dove piazzare il blocco (block placement), ovvero come fare il mapping del blocco alla cache di quel livello
- il problema di trovare il blocco (avviene più o meno in parallelo, a seconda dell'associatività della cache)
- il problema del replacement su un miss che viene fuori quando l'associatività è maggiore di 1 (devo decidere quale blocco buttare fuori)
- write policy: devo utilizzare il write-through o il write-back?

### 1. Dove può essere posizionato il blocco? (Block Placement)

Dipende dal livello di associatività.

Nel caso del direct mapping (set associative one-way) non ho scelta: i bit inferiori determinano dove andrò a posizionare il blocco.

Set associative n-way: ho n scelte all'interno di un set. I bit inferiori determinano semplicemente il set. Lo svantaggio è che devo fare la ricerca in parallelo quando devo ripescare il blocco dalla cache.

Associatività completa: posso mettere il blocco in ogni posizione.

Avere un'associatività alta ridurre il miss rate, ma incrementa anche la complessità, i costi e il tempo di accesso.

### 2. Come si individua un blocco?

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

**Direct mapped:** utilizzo l'indice per trovare la locazione, devo fare un'unica comparazione di tag.

**N-way set associative:** indice del set, poi cerco all'interno del set, devo fare n comparazioni di tag.

**Fully associative:** devo cercare in tutte le posizioni.

Le cache hardware riducono il numero di confronti per ridurre il costo.

### 3. Quale blocco dev'essere sostituito in caso di miss della cache?

Due scelte dell'entry da sacrificare se ho un miss:

1. *Least Recently Used (LRU)*: ragionevole ma complessa e il costo dell'hardware aumenta con l'associatività
2. *Random*: è molto vicino alle prestazioni dell'LRU per associatività alta ed è più semplice da implementare

### 4. Come vengono gestite le scritture?

Due possibilità:

1. *Write through*: scrivo sia nei livelli superiori della cache che nei livelli inferiori. Semplifica il replacement ma ho un write buffer che può avere un effetto sulle prestazioni.
2. *Write back*: soluzione più complicata da implementare ma più efficiente: aggiorno il livello più basso solo quando il blocco viene aggiornato. Nella cache viene mantenuto solo l'informazione della versione aggiornata del blocco.

## Modello delle 3 C

Un modello di cache in cui tutti i miss vengono classificate in base a 3 categorie: miss obbligati, miss di capacità e miss di conflitto.

- **Miss Obbligati (Compulsory Misses)**: miss causati dal primo accesso a un blocco della cache che non era mai stato caricato in precedenza
- **Miss di Capacità (Capacity Misses)**: miss generati quando la cache non è in grado di contenere tutti i blocchi di memoria necessari durante l'esecuzione di un programma; si verificano quando un blocco viene sostituito per essere ricaricato più tardi
- **Miss di conflitto (Conflict Misses)**: miss generati nella cache a mappatura diretta o set-associativa quando più blocchi competono per la stessa linea; possono essere eliminate utilizzando una cache completamente associativa della stessa dimensione

Design change	Effetto sul miss rate	Effettivi negativi sulla performance
Incremento della dimensione della cache	Decremento la capacità di miss	Potrebbe incrementare il tempo di accesso
Incrementare l'associatività	Decremento i miss di conflitto	Potrebbe incrementare il tempo di accesso
Incremento al dimensione dei blocchi	Decremento miss compulsivi	Incremento la miss penalty. Per blocchi molto grandi, potrei incrementare il miss rate a causa della pollution

Siccome i miss di conflitto nascono direttamente dalla competizione per lo stesso blocco della cache, incrementare il grado di associatività ne riduce l'incidenza. Tuttavia, l'aumento dell'associatività può causare un aumento del tempo di accesso, provocando una diminuzione delle prestazioni complessive.

I miss di capacità si possono facilmente ridurre aumentando la dimensione della cache; infatti, la capacità delle cache di secondo livello negli ultimi anni è cresciuta con un tasso costante. Ovviamente, quando si progettano cache più grandi, occorre anche prestare attenzione all'aumento del tempo di accesso, che può provocare una diminuzione delle prestazioni globali; per questo motivo la dimensione della cache di primo livello è cresciuta con un tasso molto inferiore o addirittura è rimasta invariata.

Dato che i miss obbligati sono provocati dal primo accesso a un blocco di memoria, il sistema migliore che si può adottare per ridurre questo tipo di miss consiste nell'aumentare la dimensione del blocco. Questo accorgimento permette di ridurre il numero di accessi ai blocchi di memoria che contengono un programma, visto che il programma sarà formato da un minor numero di blocchi. Aumentare troppo la dimensione dei blocchi, però, può avere un effetto negativo sulle prestazioni, perché determina anche un aumento della panalità di miss.

## Parallelismo e gerarchie delle memorie: coerenza delle cache

In un sistema con più core, come funziona la faccenda?

L1 e L2 sono private, L3 e L4 sono cache condivise.

Quando più processi che lavorano sulla stessa memoria, le modifiche sulle stesse variabili dovrebbero essere comunicate tra i vari core. Questo è detto problema di coerenza tra le cache.

E' una cosa da fare tutta in hardware: vengono implementati dei protocolli di coerenza.

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Supponiamo inizialmente che nessuna delle due cache contenga dei dati e che la locazione X della memoria principale contenga il valore 0.

Supponiamo anche che la cache sia di tipo write-through; una cache di tipo write-back aggiunge alcuni problemi, ma della stessa natura. Dopo che la CPU A ha scritto il valore 1 nella locazione X della memoria, la memoria e la cache A contengono entrambe il valore aggiornato, mentre la cache B contiene ancora 0: se B richiedesse il contenuto della locazione X, leggerebbe quindi il valore 0.

## Cache on-chip multilivello

Le cache di livello 1 sono divise tra istruzioni e dati sia per ARM che per Intel-

Size L1: configurabile per ARM, mentre è fissa a 23 Kb ad ogni core per Intel.

Associatività L1: 2-way per le istruzioni, 4-way per i dati in ARM (questo perché i miss sono minori per le istruzioni, mentre per i dati sono più

frequenti). Nel core i7 ho un 4-way per le istruzioni e 8-way per i dati.

*Rimpiazzo all'interno di L1:* random in ARM, LRU approssimato in Intel (si cerca di sostituire il blocco meno utilizzato da più tempo ma non c'è garanzia).

*Dimensioni del blocco in L1:* 64 byte da entrambe le parti (16 word consecutive a 32 bit; Intel sono 8 a 64 bit essendo un processore a 64 bit).

*Write policy L1:* write back per entrambi, ma il primo ha write allocate e il secondo no.

*Hit time L1:* due colpi di clock per ARM; 4 cicli di clock per Intel ma è pipelined.

*Organizzazione L2:* unificata per entrambi (niente separazione dati e istruzioni).

*Dimensione L2:* da 128 KB a 22MB per ARM, 256 KB per Intel.

*Associatività L2:* 16 way ARM, 8 way Intel.

*Rimpiazzo L2:* da entrambe le parti abbiamo un rimpiazzo LRU approssimativo

*Dimensioni del blocco in L2:* 64 byte da entrambe le parti

*L2 write policy:* write back con write allocate da entrambe le parti

*L2 hit time:* 12 cicli di clock per ARM e 10 per Intel

Intel ha inoltre una cache L3: unificata, 8 MiB, 16 way associative, LRU approssimato, 64 byte di dimensione blocchi, write back con write allocative, 35 cicli di clock.

## Supporto al multiple issue

Le cache devono consentire di accedere a più roba contemporaneamente.

Devono essere quindi strutturate a banche.

Ulteriori ottimizzazioni: restituisco la word richiesta per prima.

Cache non bloccante: hit under miss e miss under miss.

## Commenti conclusivi

Le memorie veloci sono piccole, le memorie grandi sono lente -> vogliamo memorie grandi ma veloci. Le cache ci danno questa illusione.

Princípio di località: i programmi utilizzano una piccola parte dello spazio di memoria, ma frequentemente.

Gerarchia delle memorie: L1 cache -> L2 cache -> ... -> DRAM Memory -> Disco rigido.

Il sistema di memoria è fondamentale per i multiprocessori.

La DRAM viene utilizzata da tutti i core e quindi la sua banda passante dev'essere bilanciata.

Le frequenze elevate sono possibili proprio perché i multiprocessori si reggono sull'utilizzo della cache.

## Indirizzamento cache direct mapped

Metodo più semplice perché mappa un indirizzo di memoria nella sua posizione prefissata nella cache.

La posizione è unica e prefissata ed è dominata dalla funzione modulo:

*Indirizzo di blocco modulo numero di blocchi nella cache*

Se il blocco è occupato si sovrascrive.

Bit inferiori -> index

Bit superiori -> tag

Bit validità -> 1 buono, 0 cattivo

## Unità minima di accesso alla cache e grandezza

L'unità minima di accesso alla cache è nota come linea di cache o blocco di cache. La grandezza della linea di cache varia a seconda dell'architettura della cache e delle specifiche del sistema. Solitamente, la grandezza del blocco di cache è compresa tra 64 e 256 byte.

Grazie alla località spaziale, se ho blocchi più grandi vuol dire che ho un miss rate ridotto.

Compromesso: avere blocchi più grandi vuol dire avere meno blocchi in totale, che è una cosa che può annullare il miss rate ridotto. Avere blocchi più grandi può portare anche al cache pollution, ovvero dati che non mi servono.

La cache pollution aumenta il tempo di prendere dati che effettivamente mi servono.

Altra risposta:

Il blocco è l'unità minima di informazioni che può essere presente o assente in una cache (dimensione di un blocco della cache: 8-16 word/32-64 byte).

## Differenza tra write through e write back

*Write through:* scrivo direttamente sulla cache e sulla memoria in maniera tale da non avere inconsistenze. Svantaggio: ci impiego troppo tempo.

Soluzione: potremmo usare un buffer di scrittura e lo svuotiamo quando passiamo alla prossima istruzione. Lo stallo avviene comunque se il buffer si riempie.

*Write back:* soluzione migliore perché ignora la sincronizzazione tra cache e memoria principale e salva in quest'ultima solo quando ha finito di aggiornarsi il blocco nella cache. Per indicare un blocco "sporco", ovvero che è stato modificato, utilizzo un altro bit. Aggiorno semplicemente il blocco nella cache: quando un blocco sporco è rimpiazzato lo scrivo in memoria e abbasso il bit di sporcizia.

Riduco così il numero delle scritture nella memoria principale. Copio solo i dati se sono stati modificati rispetto a quando sono entrati nella cache.

Altra risposta:

La maniera più semplice per conservare la coerenza tra memoria e cache consiste nello scrivere sempre il dato in entrambe le memorie. Questo schema viene chiamato write-through («scrivere attraverso»). L'altro elemento fondamentale della scrittura è rappresentato dalla gestione delle

miss in scrittura. Dapprima occorre caricare dalla memoria principale le parole appartenenti al blocco interessato. Dopo avere caricato il blocco e averlo scritto in cache, possiamo sovrascrivere la parola del blocco che aveva causato la miss; questa parola viene scritta anche nella memoria principale utilizzando il suo indirizzo completo. Sebbene il meccanismo sopra descritto consenta di gestire le scritture con molta semplicità, non offre buone prestazioni. L'utilizzo dello schema write through comporta che ad ogni scrittura la parola venga salvata anche nella memoria principale. Questa operazione richiede molto tempo e può quindi rallentare il sistema in maniera considerevole. Lo schema alternativo al write-through è chiamato write-back. In questo schema, quando si verifica una scrittura, il dato viene scritto solamente nel blocco corrispondente della cache e il blocco modificato viene salvato nel livello inferiore della gerarchia solo quando deve essere rimpiazzato. Lo schema write-back può produrre un miglioramento delle prestazioni, specialmente quando il processore può generare le scritture velocemente o più velocemente di quanto le scritture possano essere gestite dalla memoria principale.

## Cos'è una cache completamente associativa?

Ogni blocco può andare in qualsiasi posizione della cache. Questo richiede di effettuare una ricerca su tutte le posizioni allo stesso tempo; mi serve quindi avere un comparatore per ogni posizione al posto di averne uno globale

## Come funziona una cache associativa ad n-vie?

Per non fare tante comparazioni, posso dividere la cache in set (insiemi), ognuno dei quali diviso in  $n$  blocchi ( $n$ -way). Con i bit bassi rappresento l'insieme, faccio poi  $n$  ricerche parallele. Questo tipo di caso si chiama  $n$ -way set associative. Ho quindi più libertà e in più non peggioro troppo la ricerca, avendo bisogno solo di  $n$  comparatori. Il set con 1 blocco sarebbe il direct mapped.

## Tipi di cache miss che si possono verificare

- Compulsory miss (miss obbligatorie): non c'è niente nella cache della memoria che sto utilizzando. All'inizio, quando ancora non è stata utilizzata, pago pegno per forza
- Miss di capacità: ho un working set che eccede le dimensioni della cache. Un blocco rimpiazzato viene acceduto più tardi. Pensa alla montagna della memoria: devo ridurre il working set utilizzando meccanismi di blocking
- Miss di conflitto: nelle cache ho dati che mi servono ma al loro posto devo mettere qualcos'altro. E' presente nelle cache non full associative. Una buona programmazione cache friendly aiuta anche in questo caso. Devo cercare di evitare di mettere roba esattamente a posizioni di potenze di 2.

Altra risposta:

I miss vengono classificati in base alle tre seguenti categorie (modello delle tre C):

- miss obbligate (compulsory misses): questi miss sono causati dal primo accesso a un blocco della cache che non era mai stato caricato in precedenza;
- miss di capacità (capacity misses): questi miss sono generati quando la cache non è in grado di contenere tutti i blocchi di memoria necessari durante l'esecuzione di un programma; i miss di capacità si verificano quando un blocco viene sostituito per essere ricaricato più tardi;
- miss di conflitto (conflict misses): questi miss sono generati nelle cache a mappatura diretta o set-associative quando più blocchi competono per la stessa linea. I miss di conflitto possono essere eliminati utilizzando una cache completamente associativa della stessa dimensione. Questi miss sono anche chiamati miss di collisione (collision misses).

## Diverse tipologie di cache

Le cache sono memorie di accesso veloce utilizzate per migliorare le prestazioni del sistema.

Ci sono diverse tipologie di cache, tra cui:

- Cache di livello 1: questa è la cache più veloce e più vicina al processore. Ha dimensioni ridotte rispetto alle cache di livello 2 e 3, ma è molto più veloce
- Cache di livello 2: questa è una cache più grande rispetto alla cache L1 ma anche più lenta. E' utilizzata per alleviare la pressione sulla cache L1 e offre una capacità di memorizzazione maggiore
- Cache di livello 3: questa è la cache più grande, ma anche la più lenta. E' utilizzata per alleviare la pressione sulle cache L1 ed L2, offrendo una capacità di memorizzazione ancora maggiore
- Cache di sistema: questa è una cache condivisa da tutti i core del processore. E' utilizzata per migliorare le prestazioni del sistema complessivo
- Cache di disco rigido: questa è una cache utilizzata dai dischi rigidi per migliorare le prestazioni del sistema. Memorizza i dati più frequentemente utilizzati, rendendo più veloci le operazioni di lettura e scrittura sul disco
- Cache di rete: questa è una cache utilizzata dai dispositivi di rete per migliorare le prestazioni del sistema. Memorizza i pacchetti di rete più frequentemente utilizzati, rendendo più veloci le operazioni di rete

## Perché non si fanno le cache completamente associative?

Perché servono tanti comparatori e il tutto sarebbe troppo costoso. Ricordiamo sempre che lavoriamo in hardware.

Altra risposta:

Per ricercare un blocco nella cache è necessario cercarlo in tutte le linee della cache. La ricerca sequenziale è troppo lenta. Per velocizzare la ricerca è necessario effettuarla in parallelo, associando un comparatore a ciascuna posizione della cache. COSTO MOLTO ELEVATO!

## Cos'è il caching e come funziona?

Il caching è una tecnica che prevede la memorizzazione temporanea di alcune risorse, all'interno di uno spazio apposito chiamato – appunto – Cache, per permetterne un recupero più rapido.

## Che cos'è la cache pollution?

La cache pollution descrive le situazioni in cui un programma per computer in esecuzione carica i dati nella cache della CPU inutilmente, causando così l'eliminazione di altri dati utili dalla cache ai livelli inferiori della gerarchia di memoria, degradando le prestazioni. Con la cache pollution prendo una word e quelle successive ma se è grande, carico in cache cose che non mi occorrono. Ho come effetto un grande blocco e l'accesso alla memoria potrebbe rallentare il tutto, provocando un miss rate.

## Come si misura l'affidabilità?

L'affidabilità è una misura della continuità con cui viene fornito il servizio a partire da un certo istante; ossia, in maniera equivalente, è il tempo che intercorre prima che si verifichi il primo malfunzionamento. Quindi, il tempo medio di funzionamento prima di un malfunzionamento (MTTF, Mean Time To Failure) è una misura di affidabilità. Una misura collegata è la frequenza annua media dei malfunzionamenti (AFR) che rappresenta la percentuale di dispositivi che, in un anno, presentano un malfunzionamento, dato un certo tempo medio di funzionamento prima di un malfunzionamento o MTTF. Quando quest'ultimo diventa grande può essere fuorviante e quindi la frequenza annua media dei malfunzionamenti rappresenta una misura migliore (AFR).

## Qual è la parte più alta della gerarchia delle memorie e la parte di mezzo per cosa viene usata?

Per la gerarchia delle memorie la parte più alta è costituita dal processore, mentre tra il processore e la memoria principale possiamo trovare uno o più livelli di cache, che ci permettono di evitare in maniera ripetitiva l'accesso in memoria, in quanto prevedono una memorizzazione di un sottoinsieme di dati da poter riutilizzare in modo efficiente

## Cos'è una cache direct mapped e com'è fatta? Qual è la regola per trovare un blocco in una cache direct mapped?

La maniera più semplice per associare una sola locazione della cache a ogni parola della memoria consiste nel definire una corrispondenza tra l'indirizzo in memoria della parola e la locazione nella cache. Questa organizzazione della cache è detta mappatura diretta, dato che ogni locazione della memoria principale corrisponde, in modo univoco, a una locazione della cache. La corrispondenza tra indirizzi di memoria e locazioni della cache è molto semplice. Quasi tutte le cache a mappatura diretta utilizzano la seguente operazione per trovare il blocco che corrisponde a un dato indirizzo della memoria principale:

$$(Indirizzo\ del\ blocco) \bmod (numero\ di\ blocchi\ nella\ cache)$$

## Cos'è una cache full associative?

Si tratta di una cache in cui un blocco della memoria principale può essere scritto o letto da un qualsiasi blocco della cache. Risolve teoricamente lo svantaggio della direct mapped. In questo caso, la logica di controllo della cache interpreta un indirizzo di memoria semplicemente come un campo tag e un campo dati. Tuttavia, è estremamente costosa da implementare perché non essendoci un campo index, l'intero indirizzo deve essere usato come tag, aumentando la dimensione della cache e perché il dato può essere ovunque nella cache, quindi è necessario controllare il tag di ogni blocco.

## Cos'è una cache set associative?

È una cache che adotta uno schema intermedio tra la mappatura diretta e quella completamente associativa. In una cache set-associativa ogni blocco della memoria principale può essere caricato in un numero prefissato di posizioni alternative (almeno due), e una cache con n possibili scelte è chiamata set-associativa a n vie. Una cache set-associativa a n vie è costituita da un certo numero di linee (o insiemi), ciascuna costituita da n blocchi. Ciascun blocco della memoria principale viene mappato su un'unica linea della cache, individuata dal campo indice, e il blocco di dati può essere scritto in uno qualsiasi dei blocchi costituenti la linea. Quindi una cache set-associativa combina la mappatura diretta con il posizionamento completamente associativo di un blocco sulla linea: un blocco di memoria principale viene associato ad una linea della cache in maniera diretta, e tutti i blocchi della linea vengono esaminati per verificare se contengono l'elemento cercato.

## Cosa sono le località? Cos'è il principio di località?

Il principio di località sta alla base del comportamento dei programmi in un calcolatore. Questo principio afferma che un programma, in un certo istante di tempo, accede soltanto a una porzione relativamente piccola del suo spazio di indirizzamento.

Esistono due diversi tipi di località:

- località temporale (località nel tempo): quando si fa riferimento ad un elemento, c'è la tendenza a fare riferimento allo stesso elemento dopo poco tempo;
- località spaziale (località nello spazio): quando si fa riferimento ad un elemento, c'è la tendenza a fare riferimento poco dopo ad altri elementi che hanno l'indirizzo vicino a esso.

## Cosa sono un hit e un miss? E i loro tempi?

Una gerarchia delle memorie può essere composta da più livelli, ma i dati vengono di volta in volta trasferiti solo tra due livelli vicini. Il livello superiore, più vicino al processore, è più piccolo e veloce del livello inferiore, poiché utilizza una tecnologia più costosa. La più piccola quantità di informazione che può essere presente o assente in questa gerarchia su due livelli è denominata blocco o linea. Se il dato richiesto dal processore è contenuto in uno dei blocchi presenti nel livello superiore, si dice che la richiesta ha avuto successo e si denota questo evento con il termine inglese hit. Se il dato non viene trovato nel livello superiore della gerarchia, si dice che la richiesta fallisce, e si indica questo evento con il termine miss. In questo secondo caso, per trovare il blocco che contiene il dato richiesto occorre accedere al livello inferiore della gerarchia. Il tempo di hit è il tempo di accesso al livello superiore della gerarchia delle memorie, e comprende anche il tempo necessario a stabilire se il tentativo di accesso si risolve in un successo o in un fallimento, cioè se produce una hit o un miss.

## Come si effettua una ricerca in un blocco di una cache set associative? E in una cache full associative?

Nella cache set-associativa, la linea che contiene il blocco viene individuata da:

$$(Numerodelblocco) \bmod (Numerodellelineedellacache)$$

Nella cache full-associativa, la ricerca di un blocco avviene in maniera lineare scorrendo tutti i blocchi.

## Differenza tra SRAM e DRAM?

Le SRAM sono semplicemente dei circuiti integrati organizzati come vettori di memoria che (di solito) hanno una sola porta di accesso che può fornire sia la lettura sia la scrittura. Le SRAM hanno uno stesso tempo di accesso per tutti i dati, anche se i tempi di accesso in lettura e scrittura possono essere diversi. Le SRAM non hanno bisogno di refresh, per cui il loro tempo di accesso è molto vicino al periodo del clock; utilizzano da sei a otto transistor per bit per evitare che l'informazione possa essere disturbata quando viene letta. Oggi, grazie alla Legge di Moore, le cache di tutti i livelli sono integrate nel chip del processore, per cui il mercato delle SRAM singole è praticamente svanito. In una memoria SRAM il dato rimane memorizzato per tutto il tempo in cui l'alimentazione è attiva. In una memoria RAM dinamica (DRAM), invece, il dato viene memorizzato come carica in un condensatore e un solo transistor è sufficiente per leggere il dato o per sovrascriverlo. Dato che le DRAM utilizzano un solo transistor per bit memorizzato, sono molto più dense e il costo per bit è inferiore a quello delle SRAM. Tuttavia, dato che nelle DRAM l'informazione viene memorizzata in un condensatore, non rimane indefinitamente e occorre rinfrescarla periodicamente. Per rinfrescare una cella, occorre semplicemente leggerne il contenuto e riscriverlo. La carica può essere mantenuta per pochi millisecondi, per cui, se dovessimo leggere e riscrivere i diversi bit di una DRAM uno per uno, passeremmo tutto il tempo a fare il refresh della DRAM, senza avere il tempo per accedervi. Fortunatamente, le DRAM utilizzano un'architettura a due livelli che consente di rinfrescare un'intera riga, mediante un ciclo di lettura seguito immediatamente da un ciclo di riscrittura. L'organizzazione per righe aiuta nel refresh della memoria e aiuta anche nelle prestazioni: le DRAM utilizzano un buffer per memorizzare una riga per consentire accessi ripetuti. Il buffer si comporta come una SRAM: cambiando opportunamente l'indirizzo si può accedere ai diversi bit del buffer fino a quando non si deve accedere ai dati di un'altra riga. Questa funzionalità migliora significativamente il tempo di accesso, dato che il tempo di accesso a bit diversi della stessa riga è molto più basso. Anche costruire il chip più ampio migliora l'ampiezza di banda del chip. Quando la riga si trova nel buffer, il suo contenuto può essere trasferito specificando gli indirizzi successivi di gruppi di bit qualsiasi sia l'ampiezza della DRAM (tipicamente 4, 8 o 16 bit) o specificando l'indirizzo di partenza e che si intende trasferire l'intero blocco. Per migliorare ulteriormente l'interfaccia con i processori, alle DRAM è stato aggiunto il clock, per cui queste memorie sono chiamate più propriamente DRAM Sincrone o SDRAM (synchronous DRAM). Il vantaggio delle SDRAM sta nel fatto che il clock elimina il tempo necessario a sincronizzare la memoria con il processore. L'aumento di velocità delle DRAM sincrone proviene dalla possibilità di trasferire gruppi di bit adiacenti a raffica (in burst) senza dovere specificare altri indirizzi. La versione attuale più veloce è chiamata DDR (Double Data Rate – frequenza doppia dei dati) SDRAM. Questo nome significa che il trasferimento dei dati avviene sia sul fronte di salita sia di discesa del clock, ottenendo così il doppio della larghezza di banda rispetto alla banda calcolata a partire dall'ampiezza dei dati e dalla frequenza del clock.

## Cosa si intende per Time To Failure?

Il tempo medio di funzionamento prima di un malfunzionamento (MTTF, Mean Time To Failure) ed è una misura di affidabilità.

# Capitolo 6 - Architettura

## Gestione I/O

### Introduzione

Poiché un guasto al dispositivo di massa può creare la perdita dei dati, gli standard di affidabilità sono molto più elevati per i dispositivi di massa che per quelli di calcolo.

Anche le reti vengono progettate in modo da poter gestire l'interruzione della comunicazione e contengono meccanismi per rilevare le interruzioni e ripartire non appena la connessione viene ripristinata.

Per i sistemi di I/O si fa più attenzione all'affidabilità e al costo, mentre per i processori e le memorie sono più importanti le prestazioni e il costo. I sistemi di I/O devono essere progettati anche in modo da poter essere espandibili.

Anche se le prestazioni hanno un peso minore nell'I/O, esse sono più complesse da valutare.

I dispositivi di I/O sono incredibilmente diversi tra loro, ma si possono classificare in base alle seguenti caratteristiche:

- *Comportamento*: ingresso (input), dispositivo da cui si leggono i dati; uscita (output), dispositivo su cui si possono solamente scrivere dati (non fornisce dati in ingresso); memoria, dispositivo da cui si può leggere e su cui, spesso, si può anche scrivere.
- *Partner*: può essere un uomo o una macchina; si trova dall'altra parte di un dispositivo di I/O e fornisce in ingresso i dati da leggere o legge i dati inviati in uscita
- *Velocità di trasferimento dei dati*: la velocità di picco con cui vengono trasmesse le informazioni tra un dispositivo di I/O e la memoria principale o il processore. Quando si progetta un sistema di I/O è utile conoscere quale sia la massima velocità di trasferimento richiesta da un dispositivo

Per alcune applicazioni sono più importanti le prestazioni che la larghezza di banda e viceversa. In base a quale delle due sia più importante in questo ambito, verrà scelta la periferica più appropriata.

Se le richieste I/O sono per grandi quantità di dati, il tempo di risposta dipenderà principalmente dalla larghezza della banda di trasmissione.

In molti ambiti applicativi la maggior parte degli accessi richiede blocchi di dati di piccole dimensioni, quindi il sistema di I/O con la più bassa latenza per singolo accesso consentirà di ottenere il tempo di risposta inferiore.

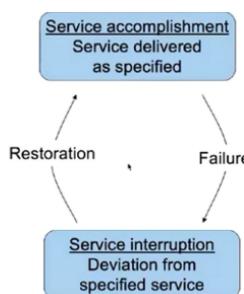
Molte applicazioni richiedono sia un'elevata velocità di trasferimento dei dati sia un tempo di risposta breve.

I tre tipi di calcolatori (desktop, server e embedded) sono sensibili all'affidabilità e al costo dei dispositivi di I/O. Per i desktop e i calcolatori embedded sono più importanti i tempi di risposta e la capacità di supportare un insieme eterogeneo di dispositivi di I/O, mentre per i server contano maggiormente il throughput e l'espandibilità dei dispositivi di I/O.

### Misurare le prestazioni

- Latenza: tempo di risposta
- Throughput: banda passante, quante azioni riesco a compiere in un'unità di tempo
- Desktop e sistemi embedded: mi interessa principalmente avere una grande varietà di device collegati e che i tempi di risposta siano brevi
- Server: mi interessa principalmente il throughput (deve smaltire velocemente le richieste) e espandere le capacità dei device (espandere la memoria)

### Dependability (affidabilità)



Qualsiasi dispositivo si muove tra due stati: funzionante e rotto.

Può quindi compiere il servizio richiesto o interrompere il servizio a causa di un guasto (fault). In ogni caso posso ritornare allo stato precedente riparando il guasto.

### Misurare la dependability

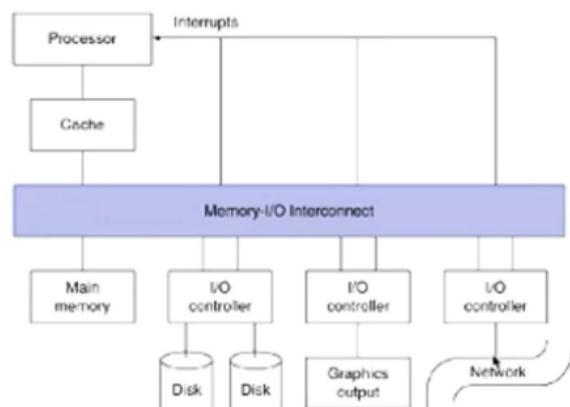
- Reliability (affidabilità misurata): mean time to failure (MTTF) -> mediamente quanto tempo passa prima che avvenga un guasto
- Service interruption: mean time to repair (MTTR) -> quanto tempo ci metto in media per riparare un guasto
- Mean time between failures: MTTF + MTTR
- Availability (disponibilità) = MTTF/(MTTF + MTTR)
- Migliorare la disponibilità: devo aumentare l'MTTF (fault avoidance, fault tolerance, fault forecasting) o diminuire l'MTTR (migliorare i tool e i processi di diagnosi e riparazione)

I produttori ci dicono il numero di accensione e spegnimento del disco, che sono le fasi che rovinano di più i cuscinetti a sfera.

E' il motivo per cui è da evitare spegnere il disco non appena non c'è attività in certi programmi.

MTBF di 5 anni con modelli appena entrati nel mercato: un produttore per tirare fuori le informazioni sui guasti si basa sull'effetto di un utilizzo sovra-elongato del disco (es: 10 ore al giorno per un prodotto che di media viene acceso un'ora al giorno). Non sono numeri affidabili, essendo sforzi estenuanti fatti su hard disk giovani.

## La connessione tra i processori, le memorie e i dispositivi di I/O



Abbiamo una coppia processore-cache affacciata su un bus che permette di accedere alla memoria e agli strumenti I/O.

Le unità di I/O di qualsiasi tipo non comunicano direttamente con la coppia processore/processi ma vengono governate da un controller. Quando gli I/O hanno compiuto il loro compito mandano un'eccezione per segnalarlo.

Ogni controller è autonomo: se il processore è impegnato, le operazioni dei controllori (che di base sono molto più lente dell'accesso alla memoria) sono indipendenti e possono procedere per conto loro. Comunicano la fine delle loro attività con le interruzioni.

Comportamento del device I/O: diviso in input, output e storage.

Partner: possono essere operate dalla macchina o da un utente umano.

Dalla rete: byte/sec numeri di trasferimenti al secondo.

*Tutto avviene tramite il bus!*

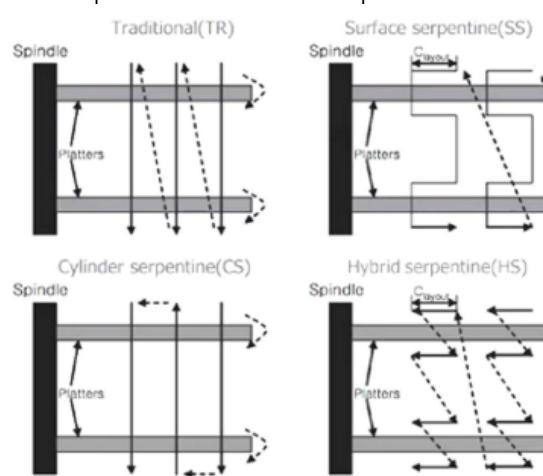
## Layout del settore sui drive moderni

Devo mettere i dati in posizioni del disco dove non devo muovere le testine.

Tradizionalmente, il sistema più conveniente era metterli nello stesso cilindro con la stessa posizione.

Col tempo le cose cambiano: mentre una volta questo cambiamento era veloce, per effetto dell'aumento della capacità dei dischi, l'indice Track Per Inch (TPI) è aumentato molto.

Diventa più conveniente spostare di poco le testine piuttosto che scendere di una posizione sotto.



E' più conveniente il pattern serpantino. I settori numerati conseguentemente sono messi così per minimizzare i tempi d'accesso.

## Componenti interconnessi

Come collego CPU, memoria e controlli I/O?

Lo faccio attraverso fili che servono per il trasferimento di dati e la sincronizzazione per capire quando campionare. Ovviamente il bus può diventare un collo di bottiglia: più aumenta la lunghezza dei fili e il numero di connessioni, più il bus diventa lento e aumentano le capacità parassite.

*Alternativa recente:* utilizzare connessioni seriali (viaggia un bit per volta) con degli switch.

## Bus

In un sistema di calcolo ho dei bus che collegano processo e memoria:

- sono corti e ad alta velocità
- il loro progetto dipende fortemente dall'organizzazione della memoria

## Bus di I/O

- Sono più lunghi e devono consentire connessioni multiple (hanno meno bisogno di alte prestazioni rispetto a quello di memoria-bus)
- Sono governati da standard di interoperabilità
- Connesi alla memoria di processo indirettamente attraverso un bridge

## Segnali di bus e sincronizzazione

*Linee di dati*: portano gli indirizzi e i dati. Possono essere multiplexate o separate

*Linee di controllo*: indicano il tipo di dati e la sincronizzazione delle transazioni e la temporizzazione

I bus possono essere sincroni o asincroni.

*Bus sincrono*: i trasferimenti sono tutti governati da un clock, che è più lento di quello del processore

*Bus asincrono*: non utilizzo un clock ma uso delle linee apposite che mi dicono quando il dato è valido (linee di handshaking).

Il migliore è l'asincrono: se metto un'unità lenta nel caso sincrono, esso uniforma i tempi di accesso sul dispositivo più lento. Con un bus asincrono ognuno segue la propria velocità.

	Firewire	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Intended use	External	External	Internal	Internal	External
Devices per channel	63	127	1	1	4
Data width	4	2	2/lane	4	4
Peak bandwidth	50MB/s or 100MB/s	0.2MB/s, 1.5MB/s, or 60MB/s	250MB/s/lane 1x, 2x, 4x, 8x, 16x, 32x	300MB/s	300MB/s
Hot pluggable	Yes	Yes	Depends	Yes	Yes
Max length	4.5m	5m	0.5m	1m	8m
Standard	IEEE 1394	USB Implementers Forum	PCI-SIG	SATA-IO	INCITS TC T10

*Hot pluggable*: si può connettere al volo in unità accesa (non era il caso della stampante)

*Data width*: misurato in bit.

Firewire venivano usati per le telecamere.

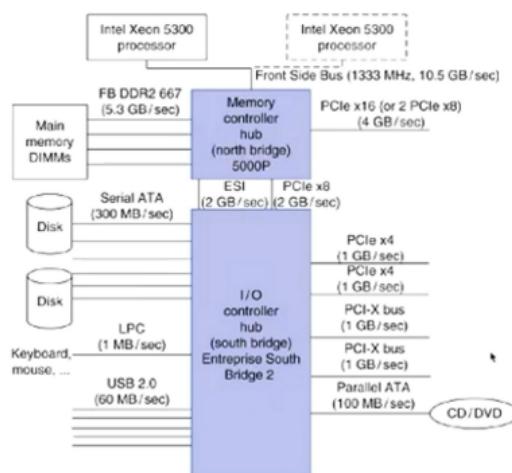
PCI express: connette le varie unità all'interno del PC

Serial ATA (Versione vecchia)

P(arallel)ATA: quello usato negli hard disk

Serial Attached SCSI: usato nei sistemi di tipo server. Costruiti con una corteza costruttiva superiore e sono più affidabili. USB 3.0 è veloce quasi quanto il serial ATA.

## Le interconnessioni di I/O nei processori x86



Il sistema tradizionale di I/O è costituito da un processore che è connesso alle periferiche attraverso due chip principali.

Il *chip più vicino al processore* è un *hub che ha la funzione di connettere e controllare la memoria* e viene chiamato *North Bridge*; l'altro chip, connesso al north bridge, è un *hub che ha la funzione di connettere i dispositivi di I/O e di controllarli* e viene chiamato *South Bridge*.

Il North Bridge è sostanzialmente un controllore DMA che connette il processore alla memoria, eventualmente a una scheda grafica, e al South Bridge.

Il South Bridge connette invece il North Bridge ai molti bus di I/O.

AMD ha integrato il North Bridge all'interno dei suoi processori, riducendo così il numero di chip e la latenza di trasferimento con la memoria e la grafica.

In base alla legge di Moore, il numero di controllori di I/O che erano prima implementati su schede esterne connesse ai bus di I/O e che sono ora implementati nei chipset è destinato ad aumentare.

Queste interconnessioni di I/O forniscono le connessioni elettriche tra i dispositivi di I/O, i processori e la memoria e definiscono anche il livello più basso del protocollo di comunicazione.

Al di sopra di questo livello base occorre definire i protocolli hardware e software per il controllo del trasferimento dati tra dispositivi di I/O e la memoria, e bisogna specificare i comandi che il processore deve inviare ai sistemi di I/O.

#### Ransomware

Delle mail spam possono contenere degli eseguibili (.rar, .zip) che sono in grado di installare software malevoli che potrebbero servire a carpire i dati personali, oppure a lanciare degli attacchi DOS su siti web.

I ransomware sono programmi che prendono i file e tentano di modificarli e criptarli con una chiave.

L'organizzazione criminale che ha creato il ransomware potrebbe poi chiedere un versamento in denaro in cambio della chiave. E' importante avere un disco di backup offline, così che il ransomware non possa accedervi.

## Interfacciamento dei dispositivi di I/O con il processore, la memoria ed il sistema operativo

Il sistema operativo gioca un ruolo cruciale nella gestione dell'I/O: esso costituisce l'interfaccia tra l'hardware e il programma che richiede di effettuare un'operazione di I/O.

I compiti di un sistema operativo derivano da 3 caratteristiche:

1. più programmi che utilizzano lo stesso processore condividono anche il sistema di I/O;
2. i sistemi di I/O utilizzano spesso gli interrupt per comunicare informazioni relative alle operazioni di I/O. Dato che gli interrupt causano il passaggio alla modalità kernel, o di supervisione, devono essere gestiti dal sistema operativo (OS)
3. Il controllo a basso livello di un dispositivo I/O è complesso perché richiede la gestione di un insieme di eventi concorrenti e perché le specifiche per gestire in modo corretto il controllo dei dispositivi sono spesso estremamente articolate

Le tre caratteristiche dei sistemi di I/O suggeriscono diverse funzionalità che il sistema operativo deve fornire:

- Deve garantire che un programma utente acceda soltanto a quella parte del dispositivo di I/O per la quale ha i diritti di accesso. Non potrebbe essere fornito alcun meccanismo di protezione a un sistema con dispositivi di I/O condivisi se i programmi utente fossero in grado di eseguire direttamente le operazioni di I/O
- Fornisce i comandi ad alto livello per accedere ai dispositivi attraverso procedure che gestiscono le operazioni del dispositivo di I/O a basso livello
- Gestisce le interruzioni generate dai dispositivi di I/O proprio come provvede a gestire le eccezioni generate dai programmi
- Cerca di fornire un accesso equo alle risorse di I/O condivise e di riorganizzare gli accessi in modo da incrementare la velocità di trasferimento dei dati nel sistema

Per svolgere queste funzioni, su incarico dei programmi utente, il sistema operativo dev'essere in grado di comunicare con i dispositivi di I/O e deve impedire ai programmi utente di comunicare direttamente con essi.

Sono tre i tipi fondamentali di informazioni da trasmettere:

1. Il sistema operativo dev'essere in grado di inviare comandi ai dispositivi di I/O. Questi comandi riguardano non solo le operazioni di lettura e scrittura, ma anche altre operazioni che devono essere effettuate dal dispositivo
2. Il dispositivo di I/O dev'essere in grado di notificare al sistema operativo che un'operazione di I/O è stata completata oppure che si è verificato un errore
3. Occorre consentire il trasferimento dei dati tra la memoria e il dispositivo di I/O

## Come impartire i comandi ai dispositivi di I/O

Per inviare un comando a un dispositivo di I/O il processore deve poter indirizzare un dispositivo e fornirgli il comando su una o più parole. Ci sono due metodi che vengono utilizzati per indirizzare un dispositivo: I/O mappato in memoria (*memory-mapped I/O*), oppure le *istruzioni di I/O speciali*.

Nel primo caso, alcune aree dello spazio di indirizzamento vengono assegnate ai dispositivi di I/O; una lettura o una scrittura in questi indirizzi di memoria viene interpretata come un comando rivolto ad un certo dispositivo di I/O.

Quando il processore invia l'indirizzo e i dati sul bus di memoria, il sistema di memoria ignora l'operazione, perché l'indirizzamento appartiene alla parte di memoria riservata all'I/O. Il controllore del dispositivo interessato all'I/O, invece, vede l'operazione, memorizza i dati e li trasmette al dispositivo sotto forma di comando.

Ai programmi utente non viene consentito di eseguire direttamente le operazioni di I/O perché il sistema operativo non permette loro di accedere allo spazio di indirizzamento assegnato ai dispositivi di I/O; questi indirizzi sono quindi protetti dal procedimento di traduzione degli indirizzi.

L'I/O mappato in memoria può anche essere utilizzato per trasmettere dei dati, leggendoli e scrivendoli in particolari indirizzi. Il dispositivo utilizza questi indirizzi per determinare il tipo di comando.

L'indirizzo codifica sia l'entità del dispositivo sia il tipo di trasmissione richiesta tra il processore e il dispositivo.

In realtà, effettuare la scrittura o la lettura dei dati richiesti da un programma si traduce, di solito, in una sequenza di operazioni di I/O fra loro distinte.

Inoltre, può succedere che il processore debba verificare lo stato del dispositivo nell'intervallo di tempo che intercorre tra due comandi, per controllare che l'attività associata al primo comando inviato sia stata completata con successo.

- **Memory-mapped I/O:** i controller I/O vengono mappati in zone lasciate opportunamente vuote dell'area di memoria così da potervi accedere tramite le solite istruzioni di load/store della memoria. Era la soluzione adottata dai processori Motorola.  
Vantaggio: utilizzo meno istruzioni.  
Svantaggio: le istruzioni non sono veloci (essendo gli indirizzi dei controller a 32 bit) ma più importante è il fatto che la memoria dovrà avere sempre un'area inutilizzabile per altro se non per mapparci gli indirizzi del controller. Infine, il sistema operativo usa un meccanismo di traduzione di indirizzi per fare in modo che quest'area sia accessibile solamente in modalità kernel.
- **Istruzioni di I/O:** accedo ai controller attraverso delle operazioni particolari (es. IN-OUT) che non usano indirizzi di memoria e che quindi sono più corti. Soluzione adottata da Intel.  
Vantaggio: indirizzi brevi (e quindi istruzioni più veloci) e non occupo la memoria.  
Svantaggio: ho bisogno di istruzioni particolari per leggere e scrivere nei controller.

I computer odierni hanno un'organizzazione diversa.

A partire dai primi pc IBM c'è un'*implementazione ibrida in cui c'è un'area di memoria riservata ma per accedervi utilizzo istruzioni IN e OUT* (avendo processori Intel).

## Come comunicare con il processore (polling e interrupt)

Il controllo periodico del bit di stato per verificare se sia giunto il momento di effettuare la successiva operazione di I/O viene detto *polling* ("interrogazione").

Il polling è il metodo più semplice per far comunicare un dispositivo di I/O con il processore.

Il dispositivo di I/O deve soltanto scrivere le informazioni nel registro di stato, mentre il processore dovrà leggere questo registro e prelevare le informazioni. Il controllo è completamente affidato al processore, il quale svolge tutto il lavoro.

Il polling può essere utilizzato in diversi modi. Le applicazioni real time dei sistemi embedded interrogano ciclicamente tutti i dispositivi I/O e possono farlo perché la velocità di trasferimento è predeterminata, cosa che rende il tempo di I/O prevedibile. Questo consente di utilizzare il polling anche quando la velocità di trasferimento dell'I/O è un po' più alta.

Lo svantaggio del polling è che può far perdere molto tempo al processore, il quale è molto più veloce dei dispositivi di I/O. Il processore può leggere il registro di stato tante volte solamente per scoprire che il dispositivo non ha ancora terminato un'operazione di I/O relativamente lenta, oppure che il mouse non è stato spostato dall'ultima volta in cui è stato interrogato. Inoltre, una volta che il dispositivo ha completato l'operazione, bisogna comunque leggere il registro di stato per verificare che l'operazione sia stata terminata con successo.

I progettisti si sono resi conto molto tempo fa dell'incremento del tempo di esecuzione dovuto al polling, per cui sono stati introdotti gli *interrupt* per comunicare al processore quando un dispositivo di I/O richiede la sua attenzione.

L'I/O controllato da interrupt (*interrupt-driven I/O*) è una tecnica adottata da quasi tutti i sistemi per almeno qualche dispositivo. Questo metodo impiega gli *interrupt* per segnalare al processore che un dispositivo di I/O ha bisogno della sua attenzione o ha determinato qualche operazione, facendo in modo che il processore venga interrotto.

Un interrupt di I/O si comporta come le eccezioni con due importanti differenze:

1. Un interrupt di I/O è asincrono rispetto all'esecuzione delle istruzioni, ossia non è associato ad alcuna istruzione e non impedisce il completamento dell'esecuzione di un'istruzione. Ciò lo rende molto diverso dalle eccezioni. In questo caso, l'unità di controllo deve solo verificare che non ci siano interrupt di I/O pendenti prima di iniziare l'esecuzione di una nuova istruzione
2. Oltre alla notifica che si è verificato un interrupt di I/O, c'è bisogno di trasmettere anche altre informazioni; inoltre, gli interrupt possono provenire da dispositivi che hanno diverse priorità, le cui richieste di interrupt sono associate a diversi livelli di urgenza

Per comunicare con il processore informazioni quali l'identità del dispositivo che ha generato l'interrupt, il sistema può utilizzare gli interrupt vettorizzati oppure il registro Causa delle eccezioni.

Quando il processore rileva l'interrupt, il dispositivo può inviare l'indirizzo del vettore oppure lo stato da scrivere nel registro Causa. Di conseguenza, quando il sistema operativo prende il controllo conosce l'identità del dispositivo che ha provocato l'interrupt e può provvedere immediatamente a gestirlo.

*Il meccanismo di interrupt elimina la necessità di far interrogare periodicamente al processore i dispositivi di I/O e permette al processore stesso di concentrarsi sull'esecuzione dei programmi.*

\*L'interruzione è come un'eccezione:

- ma non è sincronizzata all'esecuzione dell'istruzione
- può provocare un handler tra istruzioni
- l'informazione della causa identifica spesso il device che ha lanciato l'interruzione

## Livelli di priorità degli interrupt

Per accordare una diversa priorità ai dispositivi di I/O, la maggior parte dei meccanismi di interrupt possiede diversi livelli di priorità che indicano l'ordine con cui il processore deve gestire gli interrupt.

Sia le eccezioni generate internamente al processore sia gli interrupt generati dai dispositivi di I/O hanno un livello di priorità associato: tipicamente gli interrupt di I/O hanno una priorità più bassa delle eccezioni generate internamente al processore. Potrebbero esserci diversi livelli di priorità per gli interrupt di I/O, dove si associa la priorità più elevata ai dispositivi ad alta velocità.

Per supportare diversi livelli di priorità, il RISC-V mette a disposizione delle primitive che consentono al sistema operativo di implementare una

*politica di gestione degli interrupt molto simile alla gestione dei miss del TLB.*

*Il registro di stato stabilisce chi può interrompere il computer: se il bit di abilitazione degli interrupt è impostato a 0 non possono verificarsi interrupt.*

Un modo più raffinato per bloccare gli interrupt è rappresentato dalla *maschera degli interrupt*, che contiene un gruppo di bit, ciascuno dei quali corrisponde ad un bit del campo degli interrupt pendenti del registro Causa.

**Per abilitare un certo interrupt deve esserci un 1 nel corrispondente bit della maschera degli interrupt.**

Quando si genera un interrupt, il sistema operativo può determinare la causa analizzando il campo del codice delle eccezioni del registro di stato: 0 indica che c'è stato un interrupt, mentre valori diversi da zero rappresentano le diverse eccezioni.

I passi necessari per la gestione di un interrupt sono i seguenti:

1. AND logico tra il campo degli interrupt pendenti e la maschera degli interrupt per sapere quali, tra gli interrupt abilitati, abbiano generato l'interruzione.
2. Selezione dell'interrupt con la massima priorità: la convenzione del software prevede che l'interrupt più a sinistra abbia la priorità più elevata.
3. Salvataggio della maschera di interrupt del registro di stato.
4. Modifica della maschera di interrupt, in modo da disabilitare tutti gli interrupt che hanno priorità inferiore o uguale a quella che sta per essere servito.
5. Salvataggio dello stato del processore, necessario per la gestione dell'interrupt.
6. Impostazione del bit di abilitazione degli interrupt del registro Causa a 1, per abilitare gli interrupt con priorità maggiore.
7. Chiamata alla procedura di interrupt appropriata.
8. Prima di ripristinare lo stato nella condizione precedente alla gestione dell'interrupt, impostazione a 0 del bit di abilitazione degli interrupt del registro Causa. Questo permette di ripristinare il contenuto della maschera di interrupt.

In che modo il livello di priorità degli interrupt (IPL, Interrupt Priority Level) è in relazione con questi meccanismi?

L'IPL è un'astrazione del sistema operativo; esso viene salvato nella memoria di ogni processo e a ciascun processo viene assegnato un proprio IPL.

All'IPLS più basso, tutti gli interrupt sono permessi. Al contrario, all'IPLS più alto tutti gli interrupt sono bloccati.

Alzare e abbassare l'IPLS significa modificare la maschera degli interrupt del registro di Stato.

## Trasferimento dati tra un dispositivo e la memoria

Ci sono due differenti metodi che permettono di far comunicare un dispositivo con il processore. Queste due tecniche, polling ed interrupt, costituiscono la base per implementare due modalità di trasferimento dati tra i dispositivi di I/O e la memoria. Entrambe funzionano meglio con i dispositivi che hanno una larghezza di banda ridotta, per i quali si è più interessati a ridurre il costo per il dispositivo di controllo che alla velocità di trasferimento.

Sia per il trasferimento gestito dal polling che dall'interrupt, l'onere di gestire ed eseguire il trasferimento dati ricade sul processore.

Possiamo utilizzare il processore per trasferire dati tra un dispositivo e la memoria basandoci sul polling. Nelle applicazioni real time il processore carica i dati dai registri del dispositivo di I/O e li salva nella memoria.

Un meccanismo alternativo è quello di utilizzare gli interrupt per il trasferimento dei dati. In questo caso il sistema operativo trasferisce ancora dati, in piccole quantità, da e verso il dispositivo di I/O. Però poiché l'operazione di I/O è governata da un interrupt, il sistema operativo può lavorare su altri task mentre i dati vengono letti o scritti sul dispositivo.

Quando il sistema operativo capisce che è stato generato un interrupt da un dispositivo, legge lo stato del dispositivo per verificare che non si siano verificati errori. Se non ci sono errori, il sistema operativo può fornire i dati successivi.

Dopo che l'ultimo byte richiesto è stato trasferito e l'operazione di I/O è stata completata, il sistema operativo può informare il programma. In questo caso, il processore ed il sistema operativo compiono tutto il lavoro, accedendo sia al dispositivo sia alla memoria per ogni elemento di dato che dev'essere trasferito.

L'I/O governato dagli interrupt permette di evitare che il processore debba aspettare per ogni evento di I/O, anche se utilizzando questo metodo per trasferire i dati da e verso un disco rigido, il tempo di elaborazione dell'interrupt potrebbe risultare eccessivo.

Per i dispositivi a banda larga il trasferimento riguarda soprattutto blocchi di dati relativamente grandi, quindi i progettisti dei calcolatori hanno inventato un meccanismo per scaricare il processore e far sì che il controllore del dispositivo trasferisca i dati direttamente in memoria.

Questo meccanismo è chiamato Accesso Diretto alla Memoria (DMA, Direct Access Memory). Il meccanismo di interrupt viene ancora utilizzato dal dispositivo per comunicare con il processore, ma solo al fine di segnalare il completamento del trasferimento o in caso di errore.

*Il DMA è implementato mediante un controllore specializzato che trasferisce i dati tra un dispositivo di I/O e la memoria, indipendentemente dal processore.*

Il controllore di DMA diventa master ("padrone") della connessione e gestisce la lettura e la scrittura dei dati tra il dispositivo e la memoria.

Ci sono tre fasi in un trasferimento di DMA:

1. Il processore inizializza il DMA fornendo l'identità del dispositivo, l'operazione che esso deve eseguire, l'indirizzo dei dati da trasferire ed il numero di byte.
2. Il DMA inizia l'operazione sul dispositivo e governa la connessione. Quando il dato è disponibile (nel dispositivo o in memoria), lo trasferisce. Il DMA fornisce l'indirizzo di memoria per la singola lettura o scrittura. Se la richiesta implica più di un trasferimento, l'unità di controllo genera l'indirizzo successivo ed inizia il relativo trasferimento del dato. Utilizzando questo meccanismo un'unità DMA può completare un intero trasferimento senza far intervenire il processore. Molti controllori DMA contengono della memoria per permettere loro di gestire in maniera flessibile i ritardi nel trasferimento o quelli dovuti all'attesa di diventare master della connessione.
3. Una volta terminato il trasferimento DMA, il controllore interrompe il processore, il quale può verificare se l'operazione sia stata completata con successo.

Nello stesso calcolatore è possibile trovare DMA multipli.

Diversamente da quanto avviene negli schemi I/O basati su polling o interrupt, *il DMA può essere utilizzato per interfacciare un disco rigido senza che il processore debba utilizzare tutti i suoi cicli di clock per gestire un singolo I/O*.

Ovviamente, se il processore deve accedere alla memoria, il processo in esecuzione verrà ritardato se la memoria è occupata in un trasferimento DMA. D'altra parte, utilizzando la cache, il processore può evitare di dover accedere alla memoria tutte le volte che deve leggere dati, lasciando quindi libera gran parte della banda di trasferimento con la memoria per essere utilizzata dai dispositivi di I/O.

## L'accesso diretto alla memoria (DMA) e il sistema di memoria

Quando un DMA viene incorporato in un sistema di I/O, cambia la relazione tra il sistema di memoria e il processore.

Senza DMA, tutti gli accessi alla memoria provengono dal processore e quindi procedono attraverso la traduzione dell'indirizzo e l'accesso alla cache, come se fosse il processore a generare i riferimenti alla memoria.

Con il DMA si crea un percorso alternativo che non passa attraverso il meccanismo di traduzione degli indirizzi o la gerarchia delle cache.

Questa differenza genera alcuni problemi sia nei sistemi che utilizzano la memoria virtuale sia in quelli dotati di cache. Questi problemi sono generalmente risolti grazie a una combinazione di tecniche hardware e software.

La difficoltà nell'avere un DMA in un sistema dotato di memoria virtuale nasce dal fatto che le pagine hanno un indirizzo sia fisico sia virtuale.

Il DMA, inoltre, comporta dei problemi nei sistemi dotati di memorie cache, poiché possono esistere più copie dello stesso dato: una nella cache e l'altra nella memoria principale. Dato che il processore DMA invia le richieste di accesso direttamente alla memoria invece di passare attraverso la cache del processore, il contenuto di una locazione di memoria visto dal processore può essere diverso dal contenuto visto dal DMA.

Si consideri una lettura da disco gestita da un'unità DMA che scrive i dati letti direttamente in memoria. Se il DMA scrive i dati in alcune locazioni di memoria il cui contenuto si trova anche nella cache, il processore, quando leggerà quei dati dalla cache, vedrà il loro vecchio valore.

In modo analogo, se la cache è di tipo write-back, il DMA può anche leggere direttamente dalla memoria dei dati quando nella cache esiste un valore più recente di quei dati, che non è ancora stato scritto in memoria.

Questo è chiamato *problema dei dati stanti o problema della coerenza*.

Il problema della coerenza dei dati di I/O viene evitato utilizzando una delle seguenti tre tecniche principali:

1. far passare l'attività di I/O attraverso la cache. Questo assicura che le letture vedano l'ultimo valore memorizzato e le scritture aggiornino i dati nella cache. Questa operazione è dispendiosa e potenzialmente può avere un forte impatto negativo sulle prestazioni del processore, poiché i dati di I/O raramente vengono utilizzati subito e possono espellere dalla cache dati utili al processo corrente in esecuzione
2. fare in modo che il sistema operativo invalidi selettivamente la cache ad ogni richiesta di I/O in lettura o forzi la scrittura nella memoria principale ad ogni richiesta di I/O in scrittura. Questa tecnica, chiamata spesso svuotamento della cache (cache flushing), richiede un piccolo supporto da parte dell'hardware e probabilmente è più efficace se il sistema operativo può svolgere questa funzione facilmente e in modo efficiente. Poiché lo svuotamento della cache serve solo in caso di accesso DMA a blocchi di memoria, esso è piuttosto raro
3. fornire un meccanismo hardware per svuotare o invalidare in maniera selettiva alcune linee della cache. L'invalidazione hardware per assicurare la coerenza della cache è tipica dei sistemi multiprocessore, ma può essere usata anche nell'I/O

Esistono metodi per diminuire il carico della gestione delle operazioni di I/O dal processore a controllori di I/O sempre più intelligenti.

Questi metodi hanno il vantaggio di liberare cicli di clock del processore che possono essere utilizzati per altre operazioni.

Lo svantaggio è che fanno aumentare il costo del sistema I/O. Per questo motivo in un calcolatore si può scegliere la soluzione più appropriata per connettere dispositivi di I/O.

## Misure delle prestazioni dell'I/O

Le prestazioni dell'I/O dipendono:

- Dall'hardware: CPU, memoria, controlli, bus
- Software: sistema operativo, database management system, applicazioni
- Workload: frequenza di richieste e pattern

Ci sono due tipi fondamentali di specifiche che i progettisti dei sistemi di I/O devono considerare:

- i vincoli sulla latenza
- i vincoli sulla larghezza di banda

In entrambi i casi, la conoscenza del tipo di traffico influenza la progettazione e l'analisi.

I vincoli sulla latenza implicano che la latenza richiesta per completare un'operazione di I/O sia inferiore a un certo intervallo di tempo. Nel caso più semplice, il sistema è scarico e il progettista deve assicurare che sia rispettata una latenza massima, perché tale latenza è critica per l'applicazione oppure il dispositivo deve ricevere dei servizi essenziali in un tempo prestabilito (per evitare errori).

Per determinare la latenza basta tracciare le operazioni di I/O e sommare le latenze delle singole operazioni.

Diventa più complicato calcolare la latenza media o la distribuzione delle latenze quando il processore ha un carico di lavoro. Questo problema è affrontato facendo ricorso alla teoria delle code, nei casi in cui il comportamento delle richieste dei carichi di lavoro e degli istanti di tempo delle richieste delle funzioni essenziali di I/O possano essere approssimati con distribuzioni statistiche semplici, oppure tramite simulazioni quando il comportamento degli eventi di I/O è più complesso.

Progettare un sistema di I/O per soddisfare un insieme di vincoli sulla larghezza di banda dato un certo carico di lavoro è l'altro tipico problema che un progettista deve affrontare. In alternativa, si può fornire al progettista un sistema di I/O già potenzialmente configurato e richiedergli di bilanciarlo in modo da ottenere la massima larghezza di banda possibile, tenuto conto dei vincoli forniti dalla parte preconfigurata del sistema.

L'approccio generale alla progettazione di un tale sistema di I/O procede secondo le seguenti fasi:

1. Identificare il collegamento più debole del sistema di I/O; questo sarà il componente che determinerà i vincoli del progetto. A seconda del carico di lavoro, questo componente può trovarsi ovunque. Il carico di lavoro e le limitazioni sulla configurazione possono definire il punto in cui si trova il collegamento più debole.
2. Configurare questo componente per sostenere la larghezza di banda richiesta.
3. Determinare i requisiti del resto del sistema e configurare gli altri componenti in modo da supportare la larghezza di banda massima da essi richiesta.

## Parallelismo e I/O

Le CPU diventano sempre più veloci, non dimentichiamoci dell'I/O visto che è la componente più lenta a dettare la velocità.

Legge di Amdhal: non ignorare l'I/O dato che il parallelismo incrementa le prestazioni di calcolo.

La rivoluzione del parallelismo deve riguardare anche i sistemi di I/O oltre che la parte di calcolo, altrimenti lo sforzo speso nel rendere parallela l'esecuzione verrebbe vanificato ogni qualvolta un programma effettua operazioni di I/O.

L'aumento della velocità dell'I/O fu la motivazione principale che portò l'introduzione degli insiemi di disk (disk array).

Alla fine degli anni Ottanta i dispositivi di memoria di massa ad alte prestazioni disponibili erano costituiti da dischi costosi e di grandi dimensioni. Fu quindi proposto di sostituire pochi dischi grandi con molti dischi più piccoli per aumentare le prestazioni, grazie alla presenza di un numero maggiore di testine di lettura.

Questo cambiamento di struttura si è rivelato particolarmente adatto anche ai processori multipli, poiché *la presenza di diverse testine di lettura/scrittura consente al sistema di memoria di supportare molti accessi indipendenti e di trasferire dati di grandi dimensioni accedendo contemporaneamente a dischi diversi*. In altri termini, *si può ottenere sia un numero elevato di operazioni di I/O al secondo sia un'elevata velocità di trasferimento*. Oltre alle maggiori prestazioni, gli insiemi di dischi sono risultati anche più vantaggiosi in termini di costi, di consumo energetico e di dimensioni, poiché *i dischi più piccoli sono caratterizzati generalmente da una più alta efficienza nello sfruttamento dello spazio*.

La presenza di più dischi rischiava di rendere l'affidabilità del dispositivo molto più bassa. Queste unità disco piccole e meno costose avevano un MTTF più basso rispetto alle unità più grandi.

La soluzione trovata per risolvere il problema dell'affidabilità è stata quella di *aggiungere della ridondanza, in modo tale da far gestire al sistema i guasti del disco senza perdere informazioni*. Utilizzando molti dischi piccoli, il costo della ridondanza aggiuntiva richiesta per migliorare l'affidabilità era comunque piccolo rispetto al costo delle soluzioni che si potevano adottare per pochi dischi di grandi dimensioni.

Una maggiore affidabilità era quindi più facilmente ottenibile attraverso un *insieme ridondante di dischi poco costosi*, da cui il nome *redundant array of inexpensive disks* (insieme ridondante di dischi economici), abbreviato in **RAID**.

La rivoluzione del parallelismo ha fatto riemergere l'esigenza originale dello sviluppo dei RAID, cioè le prestazioni.

### RAID 0 (Nessuna ridondanza)

Semplicemente suddividendo i dati su più dischi, operazione denominata striping ("suddividere in scrisce"), si forza in modo automatico l'accesso a dischi diversi. Lo striping su un insieme di dischi fa sì che il disco RAID appaia al software come un unico grande disco; questo semplifica le operazioni di gestione dei dati memorizzati e migliora le prestazioni per accessi a dati di grandi dimensioni, poiché molti dischi possono lavorare contemporaneamente.

Il RAID 0, non contendo ridondanze, non può essere considerato a pieno titolo un disco RAID. Tuttavia, la scelta del tipo di struttura RAID viene spesso lasciata all'operatore quando crea il sistema di memoria di massa, e il RAID 0 è spesso annoverato tra le opzioni possibili.

### RAID 1 (Mirroring)

Questo schema tradizionale utilizzato per compensare i guasti dei dischi viene chiamato mirroring o shadowing e utilizza il doppio dei dischi della struttura RAID 0. Ogni volta che un dato viene scritto su un disco, lo stesso dato viene scritto anche sul disco ridondante, cosicché esistono sempre due copie della stessa informazione. Se un disco subisce un guasto, il sistema accederà al disco mirror dove leggerà le informazioni desiderate.

Il mirroring è la soluzione più costosa tra le diverse strutture RAID, in quanto richiede il maggior numero di dischi.

### RAID 2 (Riconoscimento degli errori e codice di correzione degli errori)

Il RAID 2 adotta un codice di riconoscimento e correzione degli errori molto utilizzato nelle memorie.

Esso è caduto in disuso.

### RAID 3 (Bit di parità interallacciati)

Il costo della maggiore disponibilità delle informazioni può essere ridotto a  $1/n$ , dove  $n$  è il numero dei dischi contenuti in un gruppo protetto. Invece di avere una copia completa dei dati originati in ciascun disco, vengono solamente aggiunte informazioni ridondanti sufficienti per recuperare i dati persi a causa del guasto.

Le letture e le scritture vengono effettuate sui diversi dischi che costituiscono un insieme, mentre un disco aggiuntivo contiene le informazioni addizionali richieste per ricostruire i dati in caso di guasti. Il RAID 3 è assai diffuso nelle applicazioni con grandi indiemi di dati.

La parità è uno degli schemi utilizzati nei RAID 3.

Quando un disco subisce un guasto, si possono recuperare i dati del disco rotto sottraendo il contenuto del disco di parità al contenuto degli altri dischi; le informazioni rimaste saranno le informazioni mancanti. La parità è semplicemente la somma modulo due e il bit di parità è il bit da

sommare a un numero per ottenere un risultato pari (o dispari).

A differenza dei RAID 1, molti dischi devono essere letti per determinare quali dati siano andati persi. Anche se questa tecnica ha tempi più lunghi per il recupero dei dati, richiede un numero minore di dischi ridondanti, e quindi il sistema ha un costo inferiore (per questo motivo è considerato un buon compromesso).

## RAID 4 (Blocchi di parità interallacciati)

Il RAID 4 utilizza lo stesso rapporto del RAID 3 tra numero di dischi contenenti i dati e di controllo, ma la modalità di accesso ai dati è diversa. La parità viene memorizzata in blocchi ed è associata a insiemi di blocchi di dati.

Poiché molte applicazioni richiedono che si possa accedere ai dati (anche di piccole dimensioni) in parallelo è stato creato il RAID 4, 5, 6.

Poiché l'informazione relativa al riconoscimento degli errori in ciascun settore viene controllata in lettura, quando viene verificata la correttezza dei dati, queste "piccole" letture dai singoli dischi possono avvenire indipendentemente, purché i dati siano confinati all'interno di un settore. Nel contesto dei RAID, con il termine "piccolo accesso" si intende la lettura di un solo disco di un gruppo protetto, mentre con il termine "grande accesso" si intende un accesso a tutti i dischi di un gruppo protetto.

Una "piccola" scrittura richiederebbe la lettura del vecchio dato e della vecchia parità, l'aggiunta delle nuove informazioni e infine la scrittura della nuova parità nel disco di parità e dei nuovi dati nel disco dei dati.

La chiave per ridurre la quantità di lavoro è osservare che la parità è semplicemente la somma delle informazioni: considerando i bit che cambiano quando viene scritto il nuovo dato, dobbiamo scrivere sul disco di parità questi bit.

Occorre leggere i vecchi dati dal disco prima della scrittura, confrontare i vecchi dati con i nuovi per vedere quali bit siano cambiati, leggere i vecchi bit di parità, cambiare i bit di parità corrispondenti e infine scrivere i nuovi dati e la nuova parità. In questo modo, le piccole scritture richiedono quattro accessi a due dischi invece dell'accesso a tutti i dischi.

## RAID 5 (Blocchi di parità interlacciati distribuiti)

Il RAID 4 supporta in modo efficiente sia letture che scritture in grandi e piccole dimensioni.

Un inconveniente di questo schema è la necessità di aggiornare il disco di parità ad ogni scrittura, così esso diventa un collo di bottiglia per le scritture in sequenza.

Per eliminare questo problema, l'informazione sulla parità può essere distribuita su tutti i dischi in modo da evitare che esista un unico collo di bottiglia per le scritture

Questa organizzazione distribuita della parità è definita nella struttura dei RAID 5.

Nei RAID 5 la parità associata a ciascuna riga dei blocchi di dati non è più confinata all'interno di un solo disco. Questa organizzazione consente scritture multiple contemporanee se i blocchi di parità non sono allocati sullo stesso disco.

## RAID 6 (Ridondanza P + Q)

Gli schemi basati sulla parità offrono una protezione contro il verificarsi di un guasto singolo, offrendo un metodo automatico per identificarlo.

Quando la correzione di un singolo errore non è più sufficiente, la parità può essere generalizzata eseguendo una seconda operazione aritmetica sui dati tramite le informazioni contenute in un altro disco di controllo.

Questo secondo blocco di controllo consente di recuperare i dati anche dopo un secondo guasto, ma con un incremento della quantità di memoria doppia rispetto ai RAID 5.

## Riepilogo sui dischi RAID

I RAID 1 e 5 sono molto utilizzati nei server.

Uno dei punti deboli di questo tipo di dischi RAID è la loro riparazione. Innanzitutto, per evitare di rendere i dati inaccessibili durante la riparazione, l'insieme di dischi dev'essere realizzato in modo tale da permettere la sostituzione dei dischi danneggiati senza dover spegnere l'intero sistema.

I RAID hanno ridondanza a sufficienza per poter continuare a lavorare, ma la sostituzione a caldo (hot swapping) dei dischi pone alcuni requisiti fisici ed elettrici particolarmente restrittivi sull'insieme dei dischi e sulle loro interfacce. Inoltre, potrebbe verificarsi un secondo guasto durante la riparazione; di conseguenza, il tempo di riparazione influisce sulla possibilità di perdere dati: più lungo è il tempo di riparazione, maggiore diventa la possibilità che si verifichi un secondo guasto e che vengano persi dei dati. Invece di dover attendere che un tecnico porti un disco funzionante, alcuni sistemi contengono copie di scorta, in modo da poter ricostruire i dati immediatamente, non appena viene rilevato un guasto. In questo modo l'operatore può sostituire il disco danneggiato con più calma.

Il calcolo dell'affidabilità dei dischi RAID è basato sull'ipotesi che due guasti successivi siano indipendenti tra loro, mentre i guasti dovuti alle condizioni operative tendono a essere correlati, poiché si possono verificare contemporaneamente su tutti i dischi dell'insieme. Un altro problema è legato al fatto che il tempo di riparazione di un disco di un'unità RAID è in aumento, perché la larghezza di banda dei dischi sta crescendo ad una velocità inferiore alla loro capacità; questo aumenta la probabilità che si verifichi un secondo guasto.

Dato che un sistema RAID danneggiato verosimilmente continua a fornire dati, il tempo necessario per ricostruire il disco potrebbe allungarsi parecchio. Oltre ad aumentare la durata della ricostruzione, un'altra preoccupazione è che la lettura della grande quantità di dati richiesta per la ricostruzione del disco possa aumentare la probabilità che si verifichi un guasto non più riparabile, con la conseguente perdita dei dati.

Altri elementi di preoccupazione legati al verificarsi contemporaneo di guasti multipli sono l'aumento del numero di dischi nella stessa unità e l'utilizzo dei dischi SATA, i quali sono più lenti e hanno una capacità maggiore dei dischi tradizionalmente utilizzati nei calcolatori aziendali.

## Fattori da considerare per le prestazioni

1. Processore
2. Memoria RAM

3. Sistema I/O
4. GPU (se sei un gamer)

*Non serve a niente avere un processore veloce se la RAM e il sistema di I/O sono lenti.*

## Computer Server

Le applicazioni girano sempre più sui server (web search, office apps, virtual worlds, ecc.). Hanno bisogno di server grandi ospitati in data center: processori multipli, connessione alle reti, memoria grande. Ho dei limiti di spazio e di tempo.

Rack da 19 pollici: servono per contenere i sistemi di tipo server.

Questi server non hanno quindi tastiera, le prese stanno davanti e non dietro: ci sono due alimentatori doppi (per l'affidabilità, sbalzi di tensione tendono a romperli). Ci sono poi le prese di rete.

LED di stato: gestisce il server a distanza.

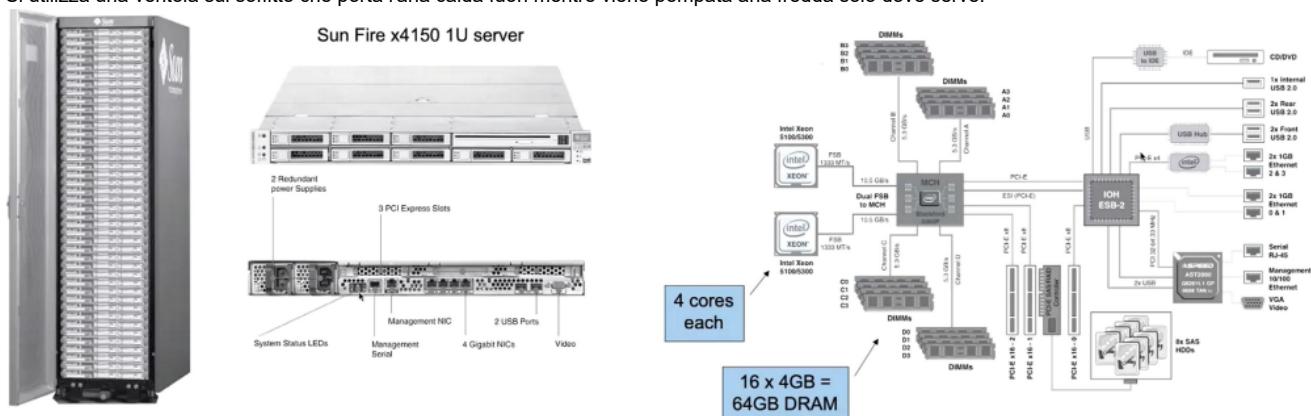
Management NIC: serve per la gestione del sistema.

Ci sono anche 4 gigabit di NIC e un adattatore grafico (video).

Parte dello spazio è dedicato ad uno switch server ed eventuali sistemi di storage.

Importante: il rack prende aria da davanti e l'aria calda esce dietro. In questo spazio viene infatti prodotto una grande quantità di calore e c'è bisogno di un sistema di raffreddamento avanzato.

Si utilizza una ventola sul soffitto che porta l'aria calda fuori mentre viene pompata aria fredda solo dove serve.



Abbiamo due Xeon con 4 core ciascuno. Sono connessi con un FSB ad un bridge che permette le connessioni alle memorie. Utilizza un interleaving a 4.

Dal bridge escono gli altri collegamenti, incluso uno verso un HDD a 8 dischi e i vari collegamenti ai fili (USB, DVD, ecc.).

## Credenze sbagliate sull'affidabilità dei dischi

Se l'MTTF è segnato come 140 anni non vuol dire che durerà tanto, si ricordi che è una media.

Se ho 100 dischi, ecco il numero che fallirà ogni anno:

$$\text{Annual Failure Rate (AFR)} = \frac{1000 \text{ dischi} * 18760 \text{ hrs/disc}}{\text{failure}} = 0.73 \text{ percento}$$

Molto spesso ci sono dei guasti immediati se il disco è uscito difettoso sin dall'inizio (sono molto rari).

In ogni caso, l'MTTF è testato in condizioni estreme (il disco viene utilizzato in maniera eccessiva per il test).

## Difetti dei sistemi I/O

- L'overhead di gestire le richieste dei processori I/O potrebbe essere dominante: anello debole del sistema
- E' più veloce fare operazioni veloci sulla CPU, ma l'architettura I/O potrebbe prevenire ciò
- Anche se dovrebbe essere più semplice, il processore dell'I/O è più lento
- Per renderlo più veloce dovremmo renderlo un componente maggiore del sistema

## Nastri magnetici

I nastri magnetici avevano dei vantaggi: erano rimovibili ed avevano capacità alte; sono affidabili e hanno accesso sequenziale.

Sono stati soppiantati dai dischi, ma sono utili per fare il backup dei sistemi server.

E' comunque meglio replicare i dati (attraverso RAID e il mirroring remoto).

## Prestazioni di picco

Per quanto riguarda l'I/O, sono quasi impossibili da raggiungere.

Altri componenti del sistema limitano la memoria, come ad esempio trasferire la memoria attraverso un bus.

Ci possono essere collisioni con il refresh della DRAM.

## Conclusioni

Le prestazioni di I/O misurano:

- throughput, tempo di risposta
- l'affidabilità e il costo

Le connessioni tra CPU, memoria e controlli I/O venivano fatti con i bus (polling, interrupt, DMA).

Il RAID migliora le prestazioni e l'affidabilità.

## Come vengono gestite le interruzioni?

Il sistema operativo preserva lo stato della CPU. Il program counter viene salvato in hardware, il resto come il valore dei registri viene salvato via software dal sistema operativo.

Modi di determinare il tipo di interruzione:

- Polling: interrogo ogni interfaccia finché non trovo quello richiesto dall'interrupt
- Sistema di interruzione vettorizzato: c'è una lista delle varie routine che gestiscono le interruzioni cui si salta direttamente

Gestione Interrupt I/O: dopo che parte un'operazione di I/O, la CPU viene sbloccata per fare in modo che faccia altro. Il controllo viene ridato al programma utente solo quando l'I/O è andato a buon fine:

- wait instruction: blocca la CPU fino alla prossima interrupt
- wait loop: memoria contesa

Nel caso in cui controllo al programma utente ritorni prima che l'I/O finisca:

- system call: richiede al sistema operativo di aspettare la fine dell'I/O
- device status table: informazioni per ogni device

## Come viene implementata una system call?

Una system call (chiamata di sistema) è una *routine che un processo esegue per richiedere un servizio al sistema operativo*. Questo servizio può essere l'accesso a un dispositivo hardware, l'allocazione di memoria, l'apertura di un file o la creazione di un processo. Una system call è un'operazione privilegiata e, di conseguenza, dev'essere eseguita in modo sicuro dal sistema operativo.

Le system call sono implementate come parte del kernel del sistema operativo e sono accessibili al livello utente tramite librerie che forniscono un'interfaccia per le chiamate di sistema. Ad esempio, la funzione open in C è una system call che viene utilizzata per aprire un file sul sistema.

Implementazione: internamente ogni system call ha un numero associato -> l'interfaccia system call mantiene una tabella indicizzata che segue questi numeri. La sys call interface invoca la sys call intesa nel kernel del sistema operativo e ritorna lo stato della sys call e i valori di ritorno.

Le system call sono importanti perché forniscono un modo per i processori di accedere ai servizi del sistema operativo in modo sicuro e controllato. Inoltre, le system call consentono ai programmatori di scrivere programmi che possono interagire con il sistema operativo e con altri processi sul sistema.

## Come funziona il DMA?

Nei nostri sistemi viene effettuato solo per i dischi.

Il sistema operativo provvede lo starting address in memoria, il controller I/O trasferisce da/alla memoria autonomamente e genera un'interruzione quando ha completato l'operazione o in seguito ad un errore. Il disco si fa dare l'indirizzo dove scrivere nella CPU, li trasferisce nella RAM nei tempi morti e quando ha finito comunica alla CPU tramite una sola interrupt. Viene utilizzato con i dischi perché sono i dispositivi con banda passante più elevata.

Altra risposta:

La comunicazione tra CPU e periferiche può avvenire principalmente attraverso due tecniche:

1. POLLING consiste nella scansione ciclica (e non periodica perché non avviene sempre nello stesso intervallo di tempo), da parte della CPU, di tutte le periferiche per verificare la disponibilità o meno alla comunicazione. E' facile da implementare ma la CPU si ritrova troppo spesso impegnata in operazioni di scansione;
2. INTERRUPT consiste nella richiesta di dialogo da parte della periferica alla CPU mentre quest'ultima è impegnata nello svolgimento di altri compiti.

La tecnica dell'Interrupt era molto buona, ma anche in questo caso si aveva il rischio di "disturbare" il processore con un gran carico di interrupt (e quindi di "richieste") da parte delle periferiche. Ecco perché è stata sviluppata la tecnica del "DMA". DMA è la sigla di "Direct Memory Access".

Praticamente il meccanismo di interrupt viene ancora utilizzato dal dispositivo per comunicare con il processore, ma solo al fine di segnalare il completamento del trasferimento o in caso di errore, per il resto, l'intero trasferimento di blocchi di dati dalla periferica di I/O alla memoria e viceversa viene gestito da un "Controller DMA".

Ci sono tre fasi in un trasferimento di DMA:

1. Il processore inizializza il DMA fornendo l'identità del dispositivo, l'operazione che esso deve eseguire (lettura o scrittura), l'indirizzo dei dati da trasferire ed il numero di byte.
2. Il DMA inizia l'operazione di trasferimento. Il DMA fornisce l'indirizzo di memoria per la singola lettura o scrittura. Se la richiesta implica più di un trasferimento, l'unità di controllo genera l'indirizzo successivo ed inizia il relativo trasferimento del dato. Utilizzando questo meccanismo un'unità DMA può completare un intero trasferimento senza far intervenire il processore

- Una volta terminato il trasferimento DMA, il controllore interrompe il processore il quale può verificare se l'operazione sia stata completata con successo.

(Nello stesso calcolatore è possibile trovare DMA multipli).

## Un processore come esegue le operazioni di I/O?

Un processore esegue le operazioni di I/O utilizzando una combinazione di hardware e software. Il processore stesso non esegue direttamente le operazioni I/O ma le delega a un componente hardware specifico, come un controller di I/O, che è responsabile della gestione delle operazioni I/O. Il processore invia una richiesta di I/O al controller I/O, che a sua volta delega al dispositivo di I/O corrispondente, come un'unità di archiviazione (es.: disco rigido), una scheda di rete o una periferica come una tastiera o un mouse. Il dispositivo I/O effettua l'operazione richiesta e restituisce i risultati al controller I/O, che a sua volta li restituisce al processore.

In questo modo, il processore può concentrarsi sull'esecuzione delle proprie attività principali, mentre le operazioni I/O vengono gestite da componenti hardware separati. Ciò garantisce che le operazioni I/O non interferiscono con le prestazioni dei processori e che il processore sia in grado di continuare a eseguire altre attività durante le operazioni I/O.

## Memory-mapped I/O e I/O instructions

- Memory-mapped I/O:** i controller I/O vengono mappati in zone lasciate opportunamente vuote dell'area di memoria così da potervi accedere tramite le solite istruzioni di load/store della memoria. Era la soluzione adottata dai processori Motorola.  
Vantaggio: utilizzo meno istruzioni.  
Svantaggio: le istruzioni non sono veloci (essendo gli indirizzi dei controller a 32 bit) ma più importante è il fatto che la memoria dovrà avere sempre un'area inutilizzabile per altro se non per mapparci gli indirizzi del controller. Infine, il sistema operativo usa un meccanismo di traduzione di indirizzi per fare in modo che quest'area sia accessibile solamente in modalità kernel.
- Istruzioni di I/O:** accedo ai controller attraverso delle operazioni particolari (es. IN-OUT) che non usano indirizzi di memoria e che quindi sono più corti. Soluzione adottata da Intel.  
Vantaggio: indirizzi brevi (e quindi istruzioni più veloci) e non occupo la memoria.  
Svantaggio: ho bisogno di istruzioni particolari per leggere e scrivere nei controller.

## Differenza tra bus sincrono e bus asincrono

- Bus sincrono:** i trasferimenti sono tutti governati da un clock (più lento rispetto a quello del processore)
- Bus asincrono:** non utilizzo un clock ma uso delle linee apposite che mi dicono quando il dato è valido (linee di handshaking)

Il migliore è l'asincrono perché se metto un'unità lenta nel caso sincrono, esso uniforma tutti i tempi di accesso sul dispositivo più lento. Con un bus asincrono ognuno segue la propria velocità.

## Cos'è il polling?

Il polling è un modello di gestione delle operazioni di I/O in cui il processore regolarmente controlla una periferica di I/O per determinare se è disponibile un nuovo dato da elaborare. Questo viene fatto inviando regolarmente una richiesta al dispositivo I/O per verificare se ha dati disponibili. Se il dispositivo ha dati disponibili, il processore li recupera e li elabora.

Il polling è un modello semplice e affidabile per la gestione delle operazioni di I/O, ma può essere inefficiente dal punto di vista delle prestazioni.

Questo perché il processore deve spendere tempo regolarmente per controllare il dispositivo di I/O, anche quando non ci sono dati disponibili.

Esiste un modello alternativo di gestione delle operazioni di I/O, noto come interrupt-driven I/O, che è più efficiente dal punto di vista delle prestazioni, ma più complesso da implementare.

In questo modello, il processore viene notificato solo quando il dispositivo di I/O ha dati disponibili, evitando così il continuo polling e permettendogli di dedicarsi ad altre attività.

## A cosa servono i dischi RAID e in che contesto vengono utilizzati?

I dischi RAID (Redundant Array of Inexpensive Disks) sono un insieme di dischi rigidi utilizzati per migliorare la performance, la disponibilità e/o la tolleranza ai guasti dei dati memorizzati.

In un sistema RAID, i dati vengono distribuiti tra più dischi rigidi, creando un'architettura che fornisce resilienza e affidabilità rispetto ad un singolo disco rigido. Ci sono diverse configurazioni RAID, ognuna delle quali fornisce diverse funzionalità. Ad esempio:

- RAID 0 (Striping):** i dati vengono divisi e distribuiti su più dischi, migliorando la velocità di lettura e scrittura dei dati. Tuttavia, non fornisce alcuna protezione contro la perdita dei dati in caso di guasto a un disco
- RAID 1 (Mirroring):** i dati vengono duplicati su più dischi, garantendo che i dati siano sempre disponibili anche in caso di guasto ad un disco. Tuttavia, non offre alcun miglioramento delle prestazioni

I sistemi seri appartengono ai livelli RAID 5/6/7, in cui i livelli vengono collegati in modo da ricostruire i dati persi attraverso dei codici. Richiedono l'Hot Swapped.

Ai nostri giorni, i RAID servono principalmente per avere maggiori prestazioni nei sistemi server, non per risparmiare.

Altra risposta:

RAID→Redundant Array of Inexpensive (Independent) Disk → Il concetto alla base del RAID, nonché quello che ha permesso di evitare che il grado di affidabilità scendesse, è quello di "avere una ridondanza dei dati" (cioè avere a disposizione sempre una copia per ogni dato).

Perché scelgo RAID?

Il vantaggio principale dei dischi RAID (in particolar modo in ambito server) è che è possibile sostituire un disco (che per esempio si è guastato) "a caldo", cioè senza interrompere il servizio (HOT SWAPPING) e che, naturalmente, il parallelismo tra i dischi aumenta le performance.

## Cosa si guadagna con un DMA?

Il Direct Memory Access (DMA) permette di aumentare le prestazioni del sistema di I/O perché riduce il carico sul processore e lo libera da compiti di gestione dei dati di I/O. Questo è importante perché il processore può concentrarsi sulle proprie attività principali, aumentando la sua efficienza e migliorando le prestazioni complessive del sistema.

## Differenza tra hot plug e hot swap?

Hot plug e hot swap sono due termini che descrivono la capacità di un dispositivo di essere aggiunto o sostituito mentre il sistema è in esecuzione (cioè senza necessità di arrestare il sistema o riavivarlo). Tuttavia, ci sono alcune differenze tra i due termini:

- Hot plug: descrive la capacità di un dispositivo di essere aggiunto al sistema mentre il sistema è in esecuzione. Ad esempio, la capacità di inserire un nuovo disco rigido in un sistema server senza interrompere il funzionamento del sistema.
- Hot swap: descrive la capacità di un dispositivo di essere sostituito con un altro dispositivo mentre il sistema è in esecuzione. Ad esempio, la capacità di sostituire un disco rigido guasto in un sistema server con un disco rigido funzionante senza interrompere il funzionamento del sistema.

In sintesi, hot plug descrive la capacità di aggiungere un dispositivo mentre il sistema è in esecuzione, mentre hot swap descrive la capacità di sostituire un dispositivo mentre il sistema è in esecuzione.

## Cosa si intende per dependability?

Per "Dependability" si intende l'affidabilità.

Il concetto di affidabilità è importante soprattutto per i dispositivi di memorizzazione di dati.

L'affidabilità viene "misurata" attraverso diversi parametri:

- Reliability → Attendibilità/Fedeltà → Tempo medio prima di un "failure" di un componente → MTTF (Mean Time To Failure);
- Service interruption → Interruzione di un servizio → Tempo medio per la riparazione → MTTR (Mean Time To Repair);
- Availability → Disponibilità → Si calcola come

$$\frac{MTTF}{(MTTF + MTTR)}$$

- Tempo medio tra failures → MTBF (Mean Time Before Failures).

# Capitolo 1 - Sistemi Operativi

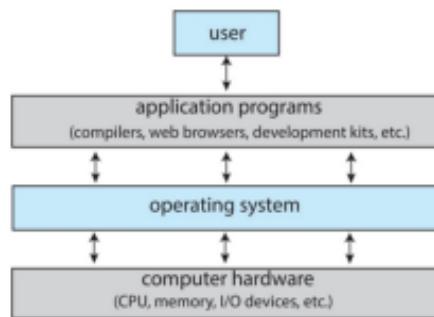
## Introduzione sui sistemi operativi

Un sistema operativo è un programma che agisce come intermediario tra l'utente e l'hardware di un calcolatore. Scopo di un sistema operativo è fornire un ambiente nel quale un utente possa eseguire programmi in modo conveniente ed efficiente.

### Che cosa fa un sistema operativo

Un sistema di elaborazione si può dividere in 4 componenti:

- *hardware*: che fornisce le operazioni di calcolo di base; comprende CPU, memoria, dispositivi I/O
- *sistema operativo*: controlla e coordina l'uso dell'hardware tra le varie applicazioni che vengono eseguite e i loro utenti (non è detto che si parli di un hardware diretto ad un singolo utente)
- *programmi di tipo applicativo*: definisco i modi in cui vengono utilizzate le risorse del sistema (memoria, CPU time, spazio su disco) per risolvere i problemi degli utenti. Esempi: compilatori, web, browser, database
- *utenti*: persone, macchine (sistemi di riproduzione automatizzata), altri computer (che si collega in remoto per ottenere delle informazioni)



Cosa fanno i sistemi operativi?

Dipende dal punto di vista.

- Dal punto di vista dell'utente gli obiettivi principali che deve soddisfare un OS sono: essere *facile da utilizzare* e avere *buone prestazioni*. Non gli interessa dell'utilizzo delle risorse
- Nel caso di un sistema condiviso, come un mainframe o un minicomputer, l'OS deve mantenere felici tutti gli utenti, ovvero tutti devono essere accontentati in tempi ragionevoli. L'utilizzo delle risorse diventa un dato importante: il sistema operativo utilizza quindi un allocatore di risorse e un programma di controllo per fare un uso efficiente dell'hardware e gestire l'esecuzione di più programmi
- Utenti di un sistema dedicato (workstation) hanno bisogno di accedere frequentemente a risorse condivise presenti su un server
- Device mobili come gli smartphone sono poveri di risorse, vengono ottimizzati per la facilità di utilizzo e per ridurre il consumo della batteria. Ho device di I/O particolari come touchscreen e riconoscimento della voce
- I sistemi embedded non hanno un'interfaccia utente: vanno in esecuzione senza il loro intervento

Il sistema operativo ha quindi tanti ruoli in contesti differenti.

Ci sono una miriade di sistemi operativi diversi e di design: dal tostapane, ai videogiochi, ai server.

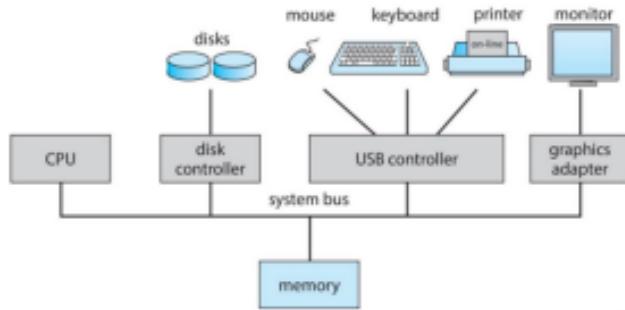
Non c'è quindi una definizione riconosciuta universalmente; sono talmente tanto diversi tra di loro che quello che mi offrono è quello che definisce le sue capacità. Da un sistema all'altro, il sistema operativo può essere tanto diverso.

**Kernel:** il programma del computer che è sempre in memoria e che è sempre pronto a mettersi in esecuzione quando è necessario.

Oltre al kernel c'è un'infrastruttura di programmi di sistema (sono moltissimi ma non fanno parte del kernel) e di programmi applicativi che non sono associati al sistema operativo.

Nei sistemi operativi moderni sono inclusi dei *middleware* (*framework di ambienti software che offrono servizi addizionali*), che sono metà tra il software e l'hardware. Offrono servizi addizionali come i database, i multimedia, le grafiche.

### Organizzazione di un sistema elaborativo



Un moderno calcolatore general-purpose è composto da una o più CPU e da un certo numero di controllori di dispositivi connessi attraverso un canale di comunicazione comune (*bus*) che permette l'accesso alla memoria condivisa del sistema.

Ciascuno di questi controllori si occupa di un particolare tipo di dispositivo fisico (per esempio, unità disco, dispositivi audio e unità video) e può gestire una o più unità ad esso connesse.

Un controllore di dispositivi dispone di una propria memoria interna (*buffer*) e di un insieme di registri specializzati. Il controllore è responsabile del trasferimento dei dati tra i dispositivi periferici ad esso connessi e la propria memoria interna.

I sistemi operativi possiedono in genere per ogni controllore di dispositivo un *driver* del dispositivo che gestisce le specificità del controllore e funge da interfaccia uniforme con il resto del sistema. La CPU e i controllori possono eseguire operazioni in parallelo, competendo per i cicli di memoria.

Per garantire un accesso ordinato alla memoria condivisa di un controllore della memoria sincronizza gli accessi.

## Interruzioni

Sono un caso particolare delle eccezioni; si definiscono come *un qualsiasi fenomeno che interrompe il programma in esecuzione e fa saltare in un'altra parte*.

L'architettura delle interruzioni deve salvare l'indirizzo dell'istruzione interrotta.

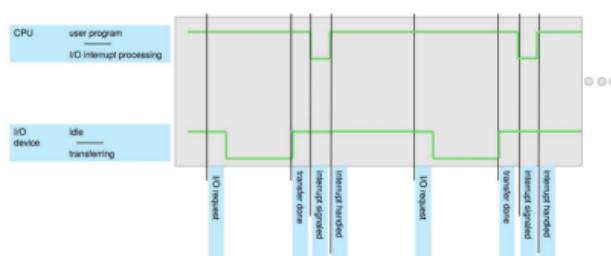
*Trap o exception*: interruzioni software che permettono di attivare il sistema operativo attraverso un errore o una richiesta dell'utente.

*Interrupt vettORIZZATO*: ogni caso di interruzione ha un proprio indirizzo all'interno di una tabella che permette di saltare direttamente alla routine di servizio di quella particolare interruzione.

Un interrupt system è interrupt driven, cioè è guidato dalle interruzioni.

Il kernel si mette in esecuzione se è rilevata un'interruzione da un programma utente (che può richiederlo direttamente tramite una trap) o dal sistema di I/O.

Timeline delle interruzioni:



Ogni volta che una richiesta di I/O completa un trasferimento, viene generata un'eccezione che ferma l'esecuzione del programma utente finché non viene gestita.

### Gestione delle interruzioni

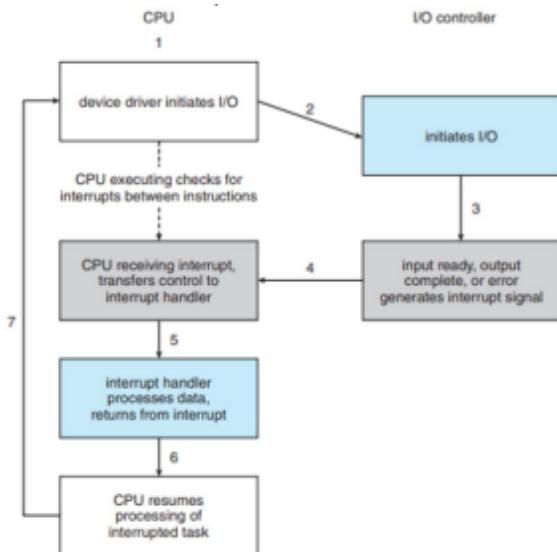
Il sistema operativo preserva lo stato della CPU. Il Program Counter viene salvato in hardware, il resto, come il valore dei registri, viene salvato via software dal sistema operativo.

Modi di determinare il tipo di interruzione:

- Polling: interrogo ogni interfaccia finché non trovo quella che ha richiesto l'interrupt
- Sistema di interruzioni vettORIZZATO: c'è una lista delle varie routine che gestiscono le interruzioni a cui si salta direttamente

Segmenti di codice separati determinano quale azione dovrebbe essere presa per ogni interrupt.

Ciclo di I/O guidato dalle interruzioni:



### Struttura dell'I/O

Dopo che parte un'operazione dell'I/O, la CPU viene sbloccata per fare in modo che faccia altro. Il controllo viene ridato al programma utente solo quando l'I/O è compiuto.

- *Wait instruction*: ferma la CPU fino all'altra interrupt
- *Wait loop*: l'accesso alla memoria è conteso
- Al più emerge una richiesta di I/O alla volta, non ci sono processi di I/O simultanei

Nel caso in cui il controllo ritorni al programma utente prima che l'I/O finisca:

- *System call*: richiede al sistema operativo di permettere all'utente di aspettare la fine dell'I/O.
- *Device-status table*: contiene un'entry per ogni device di I/O indicandone il tipo, l'indirizzo e lo stato → gli indici del sistema operativo nella tabella determinano lo stato del device e modificano la casella della tabella per includere l'interruzione.

Sull'unità di disco ci potrebbero essere più richieste in pending (che devono essere soddisfatte in seguito).

### Struttura della memoria

*Memoria principale*: solo accesso casuale (RAM), tipicamente è volatile. È una DRAM a cui la CPU può accedere direttamente.

*Memoria secondaria*: dischi. Estensione della memoria principale, non volatile.

*Hard Disk Drive (HDD)*: placche in metallo rigido o vetro ricoperte di materiale magnetico. La superficie dei dischi è divisa in tacche che sono a loro volta divise in settori. La testina (disk controller) determina l'interazione logica tra device e computer.

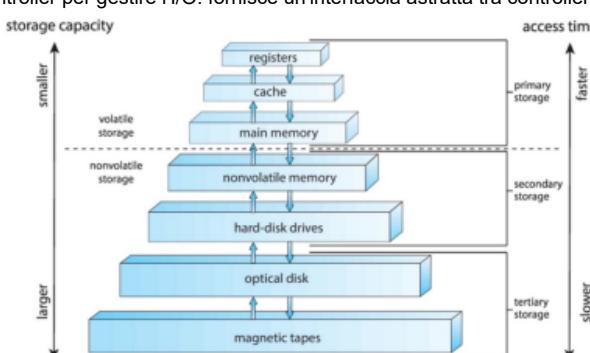
*Device a memoria non volatile (NVM)*: sono più veloci degli hard disk. Stanno diventando popolari per il loro costo basso.

### Gerarchia della memoria

I sistemi di memoria sono organizzati in una gerarchia basata su velocità, grandezza e volatilità.

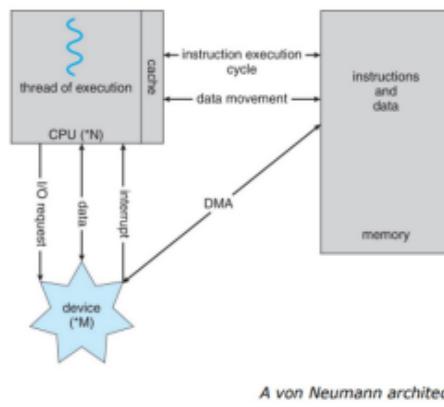
*Caching*: copia informazioni in un sistema di memoria più veloce. La memoria principale può essere vista come una cache per lo storage secondario.

C'è un device driver per ogni device controller per gestire l'I/O: fornisce un'interfaccia astratta tra controller e kernel per accedere all'I/O.



**Memoria terziaria**: viene utilizzata per il backup su larga scala. **Dischi ottici**: DVD per memorizzazione permanente e nastro magnetico utilizzato per i server.

Figura che mostra il funzionamento di un moderno sistema operativo:



A von Neumann architecture

Le unità di I/O, nel caso del DMA, possono scrivere sulla memoria senza passare sul PC grazie ai loro controller.

#### Struttura ad accesso diretto alla memoria (DMA):

usato solo per i device ad alta velocità per trasmettere informazioni con velocità vicine a quelle della memoria.

In questo caso il controller permette un accesso diretto dai buffer alla memoria, ignorando l'intervento della CPU.

Si ha la generazione di un interrupt per grandi blocchi di dati (word) al posto di blocchi piccoli (byte).

E' conveniente quando ho grosse quantità di dati da trasferire.

## Architettura degli elaboratori

### Sistemi monoprocessoresso

Diversi anni fa la maggior parte dei sistemi utilizzava un solo processore contenente un'unica CPU con un unico nucleo di elaborazione (o unità di calcolo, o *core*). Il nucleo di elaborazione è la componente che esegue le istruzioni e utilizza registri per memorizzare i dati localmente. La singola CPU con il suo nucleo di elaborazione è in grado di eseguire un insieme di istruzioni di natura generale, comprese quelle necessarie ai processi utenti. I sistemi monoprocessoresso, inoltre, possiedono altri processori specializzati, deputati a compiti particolari. Essi possono assumere la forma di processori specifici dedicati a un dispositivo, quali i controllori del disco, della tastiera o del video.

Tutti questi processori specializzati sono dotati di un insieme ristretto di istruzioni e non eseguono processi utenti. Talvolta sono guidati dal sistema operativo, che può inviare loro informazioni sul compito da eseguire successivamente, e controllarne lo stato. Questa organizzazione allegerisce la CPU dal sovraccarico di lavoro.

I PC ospitano un microprocessore all'interno della tastiera per convertire la pressione di ciascun tasto nel codice appropriato da trasmettere alla CPU.

In altre circostanze o sistemi, i processori specializzati sono dispositivi di basso livello integrati nell'hardware. Il sistema operativo non può comunicare con questi processori, che svolgono in autonomia il proprio lavoro. L'utilizzo di microprocessori specializzati è comune e non trasforma un sistema monoprocessoresso in un sistema multiprocessoresso. Se è presente una sola CPU, si tratta di un sistema monoprocessoresso. Secondo questa definizione, tuttavia, pochissimi computer moderni sono monoprocessoresso.

Molti sistemi utilizzano un general purpose a singolo processore (sistemi special purpose: sistemi embedded).

### Sistemi multiprocessoresso

Tradizionalmente, un sistema di questo tipo dispone di due o più processori, ciascuno con una CPU dotata di una singola unità di calcolo (single-core), che condividono il bus e talvolta il clock del sistema, la memoria e i dispositivi periferici.

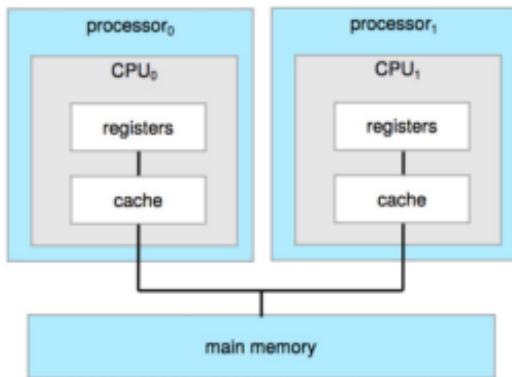
Il vantaggio principale dei sistemi multiprocessoresso è la maggiore capacità elaborativa (throughput). Aumentando il numero di unità di elaborazione è infatti possibile svolgere un lavoro maggiore in meno tempo. Con N unità di elaborazione la velocità non aumenta tuttavia di N volte, ma in misura minore. Infatti, se più unità di elaborazione collaborano nell'esecuzione di un compito, è necessario un certo lavoro supplementare (overhead) per garantire che tutti i componenti funzionino correttamente.

Questo overhead, unito alla contesa per le risorse condivise, riduce il guadagno atteso dalla disponibilità di più unità di elaborazione.

Due tipi di architetture multiprocessoresso:

- *Multiprocessoresso asimmetrico*: ad un processore è assegnata una task specifica per diversi compiti (pensa al telefono, oltre a quello non è molto diffuso al giorno d'oggi)
- *Multiprocessoresso simmetrico*: ogni processore è abilitato all'esecuzione di tutte le operazioni del sistema, incluse le funzioni del sistema operativo e i processi utente

Architettura di multielaborazione simmetrica:



Le CPU possono stare sullo stesso chip o meno, non importa. In linea di principio rimangono collegati alla memoria principale condivisa, ma hanno registri e cache proprie.

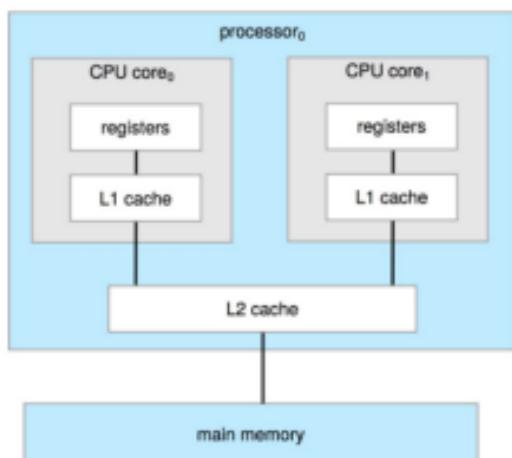
Schema UMA (Uniform Memory Access).

Nel caso di un'architettura dual-core con unità separate e non sullo stesso chip, ogni CPU ha il proprio set di registri e una cache locale (privata).

Tutti i processori condividono, tuttavia, la memoria fisica sul bus di sistema.

Il vantaggio offerto da questo modello è che molti processi sono eseguibili contemporaneamente ( $N$  processi se si hanno  $N$  CPU) senza causare un rilevante calo delle prestazioni. Tuttavia, poiché le unità di elaborazione sono separate, una potrebbe essere inattiva, mentre l'altra è sovraccarica e ciò determina un'inefficienza che sarebbe evitabile se le unità di elaborazione condividessero alcune strutture dati.

Architettura dual-core con due unità sullo stesso chip:



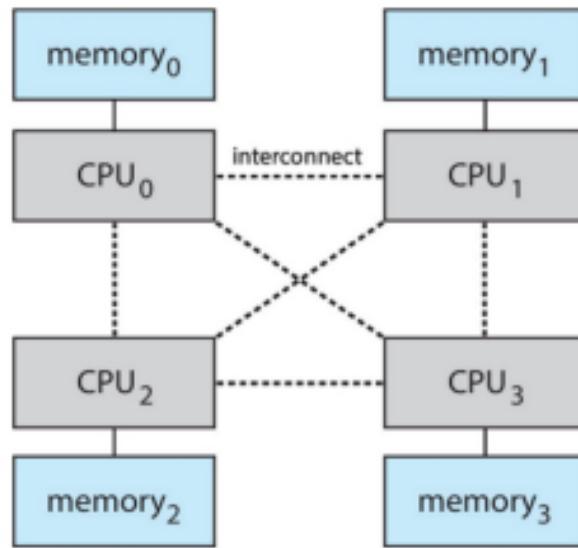
Di solito anche la cache L2 è personale e quella L3 è condivisa.

Questi sistemi possono essere più efficienti rispetto a chip dotati di una singola unità di calcolo, poiché la comunicazione all'interno di un singolo chip è più veloce rispetto a quella tra un chip e l'altro. Inoltre, un chip dotato di diversi core usa molta meno potenza rispetto a diversi chip con un singolo core e ciò è particolarmente importante in dispositivi mobili e laptop.

*Aggiungere nuove CPU a un multiprocessore ne aumenta la potenza di calcolo; tuttavia, il concetto non scala molto bene e una volta aggiunte troppe CPU il conflitto per il bus di sistema diventa un collo di bottiglia e le prestazioni iniziano a peggiorare.*

Un approccio alternativo è quello di fornire a ciascuna CPU (o a ciascun blocco di CPU), la propria memoria locale accessibile per mezzo di un bus locale piccolo e veloce.

Architettura multiprocessore NUMA:



Le CPU sono collegate da un'interconnessione di sistema condivisa, in modo che tutte le CPU condividano uno spazio di indirizzamento fisico: **accesso non uniforme alla memoria (NUMA)**.

Il vantaggio è che quando una CPU accede alla sua memoria locale, non solo è veloce, ma non vi è alcun conflitto sull'interconnessione di sistema. Pertanto, i sistemi NUMA possono scalare in modo più efficace con l'aggiunta di più processori.

Un potenziale svantaggio di un sistema NUMA è la maggior latenza quando una CPU deve accedere alla memoria remota attraverso l'interconnessione di sistema, con un conseguente possibile peggioramento delle prestazioni.

I processori NUMA, grazie alla loro scalabilità, stanno diventando sempre più diffuse nei server.

## Cluster di elaboratori (Sistemi clustered)

Sono utilizzati per operazioni ad alte prestazioni.

Sono sistemi in cui computer indipendenti lavorano insieme. Condividono la memoria tramite uno storage-area-network (SAN).

Come i multiprocessori, possono garantire grande affidabilità e resistenza ai guasti.

Degrado controllato: continuo a fornire un servizio in maniera proporzionale alla quantità dei danni.

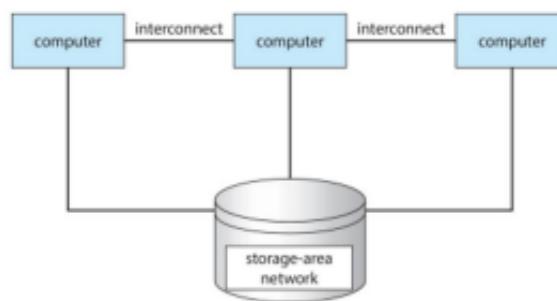
Fault tolerance: passo in più rispetto al degrado controllato -> continuo a far funzionare il cluster come se nulla fosse.

Ci sono due tipi di clustering:

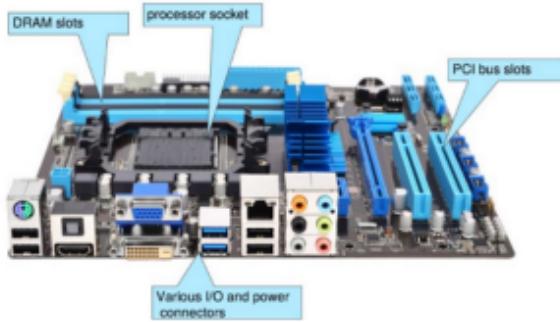
- *Cluster asimmetrico*: ha una sola macchina in hot-standby mode mentre un'altra soddisfa le richieste.
- *Cluster simmetrico*: ha nodi multipli che eseguono le applicazioni e si monitorano a vicenda; ciò comporta una maggiore efficienza.

Alcuni cluster sono utilizzati per il calcolo ad alte prestazioni. Esempi: ricerca pozzi di petrolio e precisione delle previsioni meteo. Le applicazioni devono essere scritte sfruttando la parallelizzazione.

Alcune hanno un distributed lock manager (DLM) per evitare operazioni conflittuali.



## Scheda madre di un PC



Questa scheda diventa un computer perfettamente funzionante una volta che i suoi connettori (slot) vengono popolati.

La scheda dispone di un socket del processore in grado di contenere una CPU, di alcuni connettori DRAM, di connettori PCI e di connetti I/O di vario tipo. Oggi, anche le CPU general-purpose più economiche contengono più unità di calcolo (core). Alcune schede madri contengono più socket del processore e i computer più avanzati consentono di avere più di una scheda di sistema, permettendo così di costruire sistemi NUMA.

## Attività del sistema operativo

Programma di bootstrap: codice semplice che serve ad inizializzare il sistema; carica il kernel lanciando i daemon di sistema. Normalmente è memorizzato nel firmware che fa parte dell'hardware del calcolatore.

I *deamon* sono dei programmi d'utente che servono a gestire le connessioni esterne e i server dei siti web, ecc. Rimangono attivi finché non spengo il computer.

Il kernel si appoggia quindi ai servizi forniti esternamente dal sistema operativo.

Il kernel è guidato dalle interruzioni, che siano hardware o software:

- *Hardware interrupt*: avviene quando viene lanciata un'interrupt da uno dei device
- *Software interrupt (exception trap)*: causati da errori del software (es. divisione per zero) o da una richiesta da parte di un programma utente (system call)

## Multiprogrammazione e multitasking

Fra le principali caratteristiche dei sistemi operativi vi è la multiprogrammazione. In generale, un singolo programma non è in grado di tenere costantemente occupati la CPU e i dispositivi di I/O. Inoltre, gli utenti vogliono solitamente eseguire più programmi allo stesso tempo. La multiprogrammazione, oltre a soddisfare questa esigenza degli utenti, consente di aumentare la percentuale di utilizzo della CPU, organizzando i programmi in modo tale da mantenerla in continua attività. In un sistema multiprogrammato un *programma in esecuzione* è chiamato *processo*.

L'idea su cui si fonda questa tecnica è la seguente: il sistema operativo tiene contemporaneamente in memoria centrale diversi processi. Il sistema operativo ne sceglie uno e inizia l'esecuzione: a un certo punto il processo potrebbe trovarsi nell'attesa di qualche evento, come il completamento di un'operazione di I/O.

In questi casi, in un sistema non multiprogrammato, la CPU rimarrebbe inattiva. In un sistema con multiprogrammazione, invece, il sistema operativo passa semplicemente ad un altro processo e lo esegue. Quando quest'ultimo deve a sua volta attendere, la CPU passa ancora a un altro processo. Quando il primo processo ha terminato l'attesa, la CPU ne riprende l'esecuzione. Finché c'è almeno un processo da eseguire, la CPU non è mai inattiva.

Il multitasking è un'estensione logica della multiprogrammazione; la CPU esegue più lavori commutando le loro esecuzioni con una frequenza tale da permettere a ciascun utente di usufruire di tempi di risposta rapidi.

Con questo sistema si dà l'illusione che i processi siano in esecuzione contemporaneamente. Il tempo di risposta dovrebbe essere inferiore ad un secondo.

Poiché in un sistema multitasking il sistema operativo deve garantire tempi di risposta accettabili, un metodo comune per ottenere questo risultato è la *memoria virtuale*, tecnica che consente l'esecuzione di programmi anche se non interamente caricati nella memoria. In tal modo, i programmi possono avere dimensioni maggiori della memoria fisica; inoltre, essa astrae la memoria centrale in un grande e uniforme vettore, separando la memoria logica, vista dall'utente, dalla memoria fisica, sollevando i programmatori dai problemi legati ai limiti della memoria.

## Duplicate modalità di funzionamento (dual-mode e multimode operation)

Dal momento che il sistema operativo e i suoi utenti condividono le risorse hardware e software del sistema, un sistema operativo progettato correttamente deve assicurarsi che un programma errato (o volutamente dannoso) non possa causare il malfunzionamento di altri programmi o del sistema operativo stesso.

Al fine di garantire la corretta esecuzione del sistema dobbiamo essere in grado di distinguere tra l'esecuzione del codice del sistema operativo e l'esecuzione del codice utente.

L'approccio adottato dalla maggior parte dei sistemi elaborativi è quello di fornire un supporto hardware che consenta la differenziazione tra le varie modalità di esecuzione.

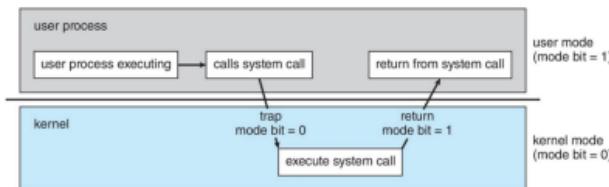
Sono necessarie due diverse modalità:

1. *Modalità utente*
2. *Modalità di sistema (kernel, supervisore, monitor o privilegiata)*

Per indicare quale sia la modalità attiva, l'architettura della CPU dev'essere dotata di un bit, chiamato *bit di modalità*:

- kernel (0)
- utente (1)

Questo bit consente di stabilire se l'istruzione corrente si esegue per conto del sistema operativo o per conto di un utente.



All'avviamento del sistema, il bit è posto in modalità di sistema. Si carica il sistema operativo che provvede all'esecuzione delle applicazioni utenti in modalità utente.

Ogni volta che si verifica un'interruzione o un'eccezione, l'hardware passa dalla modalità utente a quella di sistema, cioè pone a 0 il bit di modo. Perciò, quando il sistema operativo riprende il controllo del calcolatore, esso si ritrova in modalità di sistema. Prima di passare il controllo al programma utente, il sistema ripristina la modalità utente riportando a 1 il valore del bit.

**Istruzioni privilegiate:** *istruzioni di macchina che possono causare danni allo stato del sistema.*

Poiché l'hardware consente l'esecuzione di queste istruzioni solo nella modalità di sistema, se si tenta di far eseguire in modalità utente un'istruzione privilegiata l'hardware non la esegue e passa il controllo con un'eccezione al sistema operativo o la ignora.

Esempio: istruzione per passare alla modalità kernel.

Al giorno d'oggi ci sono ancora più livelli separati, anche per l'avvento delle macchine virtuali (gestore macchina virtuale, VMM).

## Timer

Occorre assicurare che il sistema operativo mantenga il controllo della CPU, che consiste nell'*impedire che un programma utente entri in un ciclo infinito o non richieda servizi del sistema senza più restituire il controllo al sistema operativo.*

A tale scopo si può usare un *timer*, programmabile affinché invii un segnale di interruzione alla CPU a intervalli di tempo specificati, che possono essere fissi o variabili.

*Un timer variabile di solito si realizza mediante un clock a frequenza fissa e un contatore.*

Il sistema operativo assegna un valore al contatore, che si decrementa a ogni impulso e quando raggiunge il valore 0 si genera un segnale d'interruzione.

Prima di restituire all'utente il controllo dell'esecuzione, il sistema assegna un valore al timer. Se esso esaurisce questo intervallo genera un'interruzione che causa il trasferimento del controllo al sistema operativo, che può decidere se gestire l'interruzione come un errore fatale o concedere altro tempo al programma. Ovviamente, anche le istruzioni usate dal sistema per modificare il valore del timer si possono eseguire soltanto in modalità privilegiata.

## Gestione delle risorse

Il sistema operativo è un gestore di risorse che deve gestire la CPU del sistema, lo spazio di memoria, la memoria secondaria e le periferiche di I/O.

### Gestione dei processi

Processo: programma in esecuzione -> entità attiva.

Programma: entità passiva.

Un processo ha bisogno di risorse: tempo di CPU, spazio di memoria, I/O, file aperti e dati inizializzati. Alla terminazione del processo, il sistema operativo recupera le risorse e il processo viene eliminato.

Program counter: specifica la prossima istruzione da eseguire per i programmi a singolo thread.

I programmi multithread hanno un program counter per ciascun thread.

Tipicamente, i sistemi possono avere più processi in memoria che si contendono l'esecuzione della CPU.

Il sistema operativo come gestisce i processi?

Esso deve creare e cancellare sia i processi utente che i processi del sistema (come i deamon), deve sospenderli e recuperarli.

Inoltre, deve provvedere ai meccanismi per la sincronizzazione, la comunicazione e la gestione dei deadlock.

*Un processo è l'unità fondamentale di lavoro in un sistema operativo. La gestione dei processi comprende aspetti come la loro creazione e cancellazione, nonché la messa a punto di meccanismi per la comunicazione reciproca e la sincronizzazione dei processi.*

## Gestione della memoria

Per eseguire un programma, le sue istruzioni devono stare in memoria. Però, non tutti i dati devono essere per forza memorizzati.

La gestione della memoria da parte del sistema operativo determina quali processi sono allocati in memoria, quanta memoria occupano, in quale spazio si trovano, ecc.

Attività del sistema operativo connesse alla gestione della memoria:

- Tenere traccia delle parti occupate
- Decidere quando un processore dev'essere mosso dentro o fuori dalla memoria
- Allocare e deallocare lo spazio di memoria necessario

*Un sistema operativo gestisce la memoria mantenendo traccia di quali parti di essa vengono usate e da chi. E' sempre al sistema operativo, inoltre, che spetta l'allocazione dinamica e il rilascio dello spazio di memoria.*

## Gestione dei file

Il sistema operativo gestisce una vista logica di tutte le unità di memorizzazione. Noi vediamo un'*unità logica di memorizzazione detta file*. I file verranno memorizzati nel file system.

Il sistema operativo come gestisce il file management system?

- I file vengono organizzati all'interno di directory
- Access control per capire chi può accedere i file e/o modificarli
- Il sistema operativo crea i file e le directory, le cancella, fornisce primitive per manipolarli. A livello più basso fornisce dei meccanismi per mappare i file su un altro livello di memoria secondario e fare il backup su una memoria non volatile

*Il sistema operativo gestisce l'archiviazione dei dati: ciò comprende la realizzazione del file system per i file e le directory, e la gestione dello spazio sui dispositivi per la memorizzazione di massa.*

## Gestione della memoria di massa

La memoria di massa è una memoria terziaria che viene utilizzata per il back-up; comprende dischi e blue-ray.

E' tutto sotto il controllo del sistema operativo, ma lo vedremo dopo.

## Gestione dell'I/O

Il sistema operativo nasconde le caratteristiche dei dispositivi in modo tale che l'utente possa lavorarvi in maniera uniforme.

- Questo comprende operazioni di buffering (memorizzare dati temporaneamente mentre vengono trasferiti), di caching (memorizzare parti in una memoria più veloce), spooling (output overlappante per un lavoro di input di un altro job, I/O asincrono)
- Driver per device hardware specifici
- Interfaccia generale device-driver

BSoD (Blue Screen od Death) erano dovute a dei driver che danneggiavano l'integrità del sistema. E' il motivo per cui parti l'allarme per vedere se il driver è certificato/sicuro. Questo perché vanno a far parte integrante del sistema operativo e quindi possono fare danni al sistema.

Servono quindi dei driver certificati per poterli includere nel driver del sistema senza problemi (problema grande per Linux).

## Sicurezza e protezione

Protezione: ogni meccanismo per controllare l'accesso di processi o utenti alle risorse (es. login e password).

Sicurezza: proteggere il sistema da attacchi esterni e interni (virus, malware, furto d'identità).

I sistemi generalmente devono poter distinguere i vari utenti tramite gli user ID; questi ultimi sono associati ai file e ai processi.

E' buona norma non ricorrere mai alla modalità amministratore per accedere perché è una privilege escalation: è pericoloso se non si sa cosa si sta facendo.

*I sistemi operativi forniscono meccanismi di protezione e per la sicurezza del sistema e degli utenti. La protezione controlla l'accesso, da parte dei processi o degli utenti, alle risorse che il sistema mette a disposizione.*

## Virtualizzazione

La virtualizzazione consiste nell'*astrazione dell'hardware di un computer in molteplici distinti ambienti di esecuzione*.

Essa nasconde le caratteristiche dell'hardware, in maniera tale da eseguire programmi tramite altri tipi di CPU.

*Emulazione*: tecnica utilizzata quando il tipo di CPU origine è diverso dal tipo di CPU destinazione. Metodo molto lento.

*Interpretazione*: il linguaggio macchina non è compilato in codice nativo.

Per virtualizzazione si intende dare la possibilità di installare un sistema operativo sopra un altro sistema operativo. Mi dà l'illusione di lavorare su un hardware separato con una sola CPU.

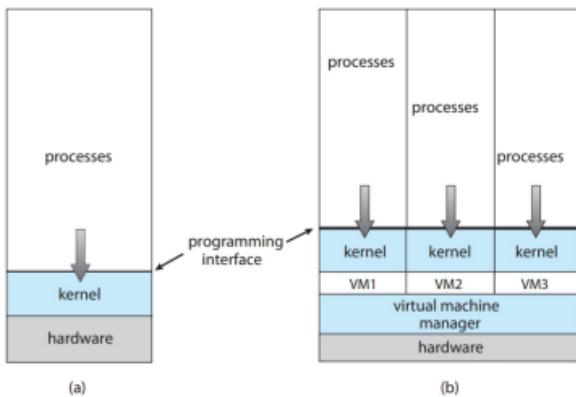
Viene utilizzato soprattutto in ambito server: le società che sviluppano software diversi per diversi sistemi operativi possono utilizzare la virtualizzazione per eseguire tutti questi sistemi operativi su un unico server fisico destinato allo sviluppo.

La virtualizzazione evita l'emulazione.

VMM (Virtual Machine Manager): provvede ai servizi di virtualizzazione.

Container (caratteristica di Linux): permette di avere degli ambienti di esecuzione separati sotto un unico sistema operativo.

Differenza tra ambiente virtualizzato e ambiente hardware:



Ogni macchina virtuale ha i propri processori e un sistema operativo diverso, anche se condividono tutti lo stesso hardware.

*La virtualizzazione è una tecnica che permette di astrarre l'hardware di un singolo computer (la CPU, la memoria, le unità disco, le schede di interfaccia di rete) in diversi ambienti di esecuzione, creando così l'illusione che ogni distinto ambiente sia in esecuzione sul suo proprio computer. Questi ambienti possono essere visti come diversi sistemi operativi in esecuzione contemporaneamente e in grado di interagire l'uno con l'altro. Un utente di una macchina virtuale può passare da un sistema operativo a un altro esattamente come può commutare tra processi in esecuzione contemporaneamente in un singolo sistema operativo.*

## Sistemi embedded real-time

In termini quantitativi, i *computer facenti parte di altri sistemi* (*embedded computer*) attualmente costituiscono la tipologia predominante di elaboratore. Questi dispositivi si ritrovano dappertutto, dai motori delle auto ai robot industriali. Di solito hanno compiti molto precisi: i sistemi su cui sono installati sono spesso rudimentali, per cui offrono funzionalità limitate. Essi presentano un'interfaccia utente scarsamente sviluppata, se non assente, dato che sono spesso concepiti per la sorveglianza e la gestione di dispositivi meccanici, quali i motori delle automobili e i bracci dei robot.

Ciò che contraddistingue i sistemi embedded è la loro grande *variabilità*. Talvolta possono essere elaboratori general-purpose con sistemi operativi standard - come Linux - che sfruttano applicazioni create appositamente per implementare una funzionalità. Altri sono dispositivi hardware che ospitano un sistema operativo special-purpose, che consente di ottenere proprio la funzionalità desiderata. Inoltre, esistono dispositivi hardware che hanno al loro interno circuiti integrati per applicazioni specifiche (ASIC), capaci di svolgere il loro lavoro senza un sistema operativo.

La diffusione dei sistemi embedded è in continua espansione. Indubbiamente, sono destinate a crescere anche le potenzialità di questi congegni, sia come unità indipendenti sia in qualità di membri delle reti e di Internet.

E' già oggi possibile l'automatizzazione di intere abitazioni, così che un computer centrale - che si tratti di un computer general-purpose o di un sistema embedded - possa essere in grado di controllare il riscaldamento, l'illuminazione o i sistemi di allarme.

I sistemi embedded funzionano quasi sempre con *sistemi operativi real-time*. Essi si utilizzano quando siano stati imposti rigidi vincoli di tempo alle funzioni del processore o al flusso dei dati: per questa ragione sono spesso adoperati come dispositivi di controllo per applicazioni dedicate. I sensori trasmettono i dati al computer, che deve analizzarli e, a volte, prendere le misure adatte per il controllo del sistema.

Questo tipo di sistema ha vincoli di tempo fissi e definiti con precisione. L'elaborazione deve avvenire entro i limiti prestabiliti: in caso contrario, il sistema andrà in crisi.

*Un sistema real-time funziona correttamente solo se esso genera il risultato corretto entro precise scadenze.*

## Sistemi operativi liberi e open-source (nato con Linux)

Lo studio dei sistemi operativi è stato facilitato dalla disponibilità di un vasto numero di programmi liberi e open-source.

Sia i sistemi operativi liberi che i sistemi operativi open-source sono disponibili in formato sorgente anziché come codice binario compilato.

Si noti, tuttavia, che software libero e software open-source sono due idee diverse sostenute da diversi gruppi di persone.

- Il software libero non solo rende il codice sorgente disponibile, ma è anche dotato di una licenza che consente l'uso, la ridistribuzione e la modifica senza costi.
- Il software open-source non offre necessariamente tale licenza.

Pertanto, sebbene tutto il software libero sia open-source, alcuni software open-source non sono liberi.

GNU/Linux è il sistema operativo open-source più famoso, con alcune distribuzioni libere e altre solo open-source.

Microsoft Windows è un esempio dell'approccio opposto -> closed-source. Windows è un software proprietario, Microsoft lo possiede, ne limita l'uso e ne protegge attentamente il codice sorgente.

Il sistema operativo macOS di Apple adotta un approccio ibrido: contiene un kernel open-source chiamato Darwin, ma include anche componenti chiuse e proprietarie.

Avere a disposizione il codice sorgente permette al programmatore di produrre il codice binario, eseguibile dal sistema. Il processo inverso, chiamato processo di reverse engineering, che permette di ricavare il codice sorgente partendo dal binario, è molto più oneroso; molti elementi utili, per esempio i commenti, non possono essere ripristinati.

Si può sostenere che i sistemi open-source sono più sicuri poiché hanno più occhi puntati sopra, dato che su di esso ci lavorano un gran numero di programmatore.

## **Cosa sono le istruzioni privilegiate?**

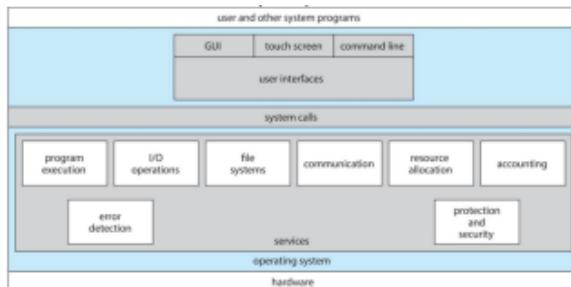
Le istruzioni privilegiate sono istruzioni di sistema che possono essere eseguite solo da processi che hanno privilegi elevati, come i processi del sistema operativo o i processi con privilegi da amministratore. Queste istruzioni hanno accesso a funzionalità o risorse del sistema che non sono disponibili per i processi normali e possono compiere azioni che influiscono sul funzionamento del sistema, come la gestione della memoria, la configurazione della rete e l'accesso ai dispositivi di sistema.

# Capitolo 2 - Sistemi Operativi

## Strutture dei sistemi operativi

### Servizi di un sistema operativo

Un sistema operativo offre un ambiente in cui eseguire i programmi e fornire i servizi.



Servizi che offrono funzionalità utili agli utenti:

- **Interfaccia con l'utente (UI):** può assumere diverse forme, ma la più diffusa è la GUI (interfaccia utente grafica), ossia un sistema a finestre dotato di un dispositivo puntatore per comandare operazioni di I/O e selezionare opzioni dai menu. I sistemi mobili sono dotati di un'interfaccia touch screen. Altra possibilità: interfaccia a riga di comando (CLI), che utilizza comandi di testo e un mezzo per inserirli.
- **Esecuzione di un programma** (i sistemi operativi sono dati per questo): il sistema deve poter caricare un programma in memoria ed eseguirlo. Il programma deve poter terminare la propria esecuzione in modo normale o anomalo (indicando l'errore).
- **Operazioni di I/O:** un programma in esecuzione può richiedere un'operazione di I/O che implica l'uso di un file o di un dispositivo di I/O. Per particolari dispositivi possono essere necessarie funzioni speciali, come la registrazione su un CD o DVD. Per motivi di efficienza e protezione, di solito un utente non può controllare direttamente i dispositivi di I/O, quindi il sistema operativo deve fornire mezzi adeguati.
- **Comunicazioni** (una delle ultime caratteristiche aggiunte): in molti casi un processo ha bisogno di scambiare informazioni con un altro processo. Ciò avviene principalmente in due modi: tra processi in esecuzione nello stesso calcolatore e tra processi in esecuzione in calcolatori diversi collegati per mezzo di una rete. La comunicazione si può realizzare tramite una *memoria condivisa*, che permette a due o più processi di leggere e scrivere in una porzione di memoria che condividono, o attraverso lo scambio di messaggi, in questo caso il sistema operativo trasferisce pacchetti d'informazioni in un formato predefinito tra i vari processi (prima serviva un'applicazione di terze parti).
- **Rilevamento di errori:** il sistema operativo dev'essere sempre capace di rilevare e correggere eventuali errori che possono verificarsi nella CPU nei dispositivi di memoria, nei dispositivi di I/O e in un programma utente. Per assicurare un'elaborazione corretta e coerente il sistema operativo deve saper intraprendere l'azione giusta per ciascun tipo di errore. Talvolta l'unica scelta possibile è l'arresto del sistema, altre volte è possibile terminare il processo che è causa d'errore o restituire un codice d'errore a un processo in modo che da solo cerchi di rilevare e correggere l'errore.

Funzioni del sistema operativo che non riguardano direttamente l'utente:

- **Allocazione delle risorse:** se sono attivi più utenti o sono contemporaneamente in esecuzione più processi, il sistema operativo provvede all'assegnazione delle risorse necessarie a ciascuno di essi. Alcune di queste risorse, come i cicli di CPU, la memoria centrale e la memoria del file system, possono avere un software di gestione molto specifico, mentre altre, come i dispositivi di I/O, possono avere routine di richiesta e di rilascio più generali. Per esempio, per determinare come utilizzare al meglio la CPU, i sistemi operativi impiegano le procedure di scheduling della CPU, che tengono conto della velocità, dei processi da eseguire, del numero di registri disponibili. Esistono anche procedure per l'assegnazione di stampanti, driver di memorizzazione USB e altre periferiche.
- **Logging:** vogliamo mantenere traccia di quali programmi usano il calcolatore, segnalando quali e quante risorse impiegano.
- **Protezione e sicurezza:** i proprietari di informazioni memorizzate in un sistema elaborativo multiutente o in rete possono voler controllare l'uso di tali informazioni. Quando più processi separati sono in esecuzione concorrente essi non devono influenzarsi o interferire con il sistema operativo. La protezione assicura che l'accesso alle risorse del sistema sia controllato. La sicurezza di un sistema comincia con la richiesta dell'identificazione da parte dell'utente, di solito attraverso password, per permettere l'accesso alle risorse; si estende fino a difendere i dispositivi di I/O dai tentativi di accesso illegali e provvede al loro rilevamento.

### Interfaccia con l'utente del sistema operativo

Vi sono due modi fondamentali per gli utenti di comunicare con il sistema operativo. Uno si basa su un'interfaccia a riga di comando o interprete dei comandi, che permette agli utenti di inserire direttamente le istruzioni che il sistema deve eseguire.

L'altro sfrutta un'interfaccia grafica con l'utente o GUI, che serve da tramite tra utente e sistema.

#### Interprete dei comandi (riga di comando)

La funzione principale dell'interprete dei comandi consiste nel raccogliere ed eseguire il successivo comando impartito dall'utente. A questo livello, la maggioranza dei comandi riguarda la gestione dei file: creazione, cancellazione, elenco, stampa, copia, esecuzione e così via.

- E' implementato all'interno del kernel, altre volte è implementata attraverso un programma di sistema (principalmente nella famiglia UNIX)
- A volte diversi CLI offrono funzionalità diverse. La shell di Linux si chiama bash
- Fa il fetch di un comando dall'utente e lo esegue
- A volte ci sono dei comandi che l'interprete può fare da solo, per tutto il resto si affida ai programmi di sistema. Nell'ultimo caso, per aggiungere nuove funzioni non c'è bisogno di modificare la shell

La versione open source è la bash (bourne again shell). Le CLI sono programmabili.

## Interfaccia grafica con l'utente (GUI)

Metafora del desktop: ho uno spazio virtuale dove ci sono le varie icone che rappresentano file, programmi e azioni.

Attraverso il mouse posso muovermi tra gli oggetti dell'interfaccia per causare varie azioni (provvedere informazioni, opzioni, eseguire funzioni, aprire directory).

Invenzione recente portata da Windows e inventata nei laboratori PARC della Xerox.

## Interfaccia touch screen

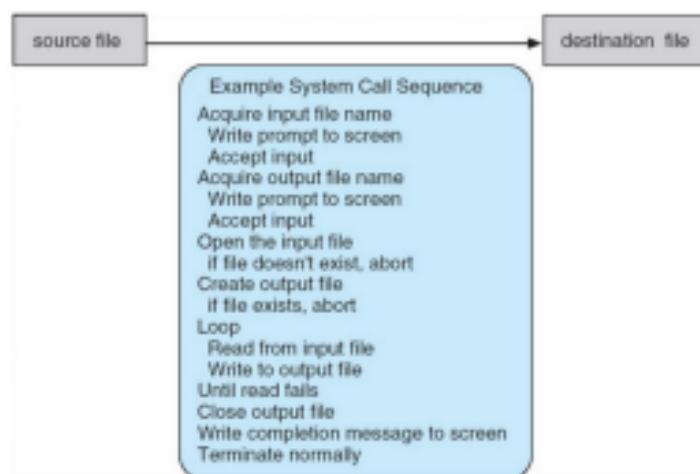
Non utilizzo il mouse, ma il tatto e una tastiera virtuale.

## System call (chiamate di sistema)

Le system call costituiscono un'*interfaccia per i servizi resi disponibili dal sistema operativo*. Tali chiamate sono generalmente disponibili sotto forma di routine scritte in C o C++, sebbene per alcuni compiti di basso livello, come quelli che comportano un accesso diretto all'hardware, possa essere necessario il ricorso al linguaggio assembly.

### Esempio di system call

Scrittura di un programma che legge i dati da un file e li trascina in un altro.



## Interfaccia per la programmazione di applicazioni (API)

Una API (Application Programming Interface) specifica un insieme di funzioni a disposizione del programmatore e dettaglia i parametri necessari all'invocazione di queste funzioni, insieme ai valori restituiti.

### Esempio di API standard

Consideriamo la funzione read() disponibile in Unix e Linux. L'API per questa funzione si può ottenere digitando:

```
man read
```

da riga di comando.

Una descrizione di questa API è la seguente:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

dove:

- ssize\_t è il valore restituito
- read è il nome della funzione
- fd, \* buf, count sono i parametri

Un programma che utilizza la `read()` deve includere il file `unistd.h` che, tra le altre cose, definisce i tipi di dato `ssize_t` e `size_t`. I parametri passati alla `read()` sono i seguenti:

- `int fd` - il descrittore del file da leggere
- `void * buf` - un buffer nel quale vengono messi i dati letti
- `size_t count` - il massimo numero di byte da leggere e inserire nel buffer

Quando una `read()` è completata con successo viene restituito il numero di byte letti.

La `read()` restituisce 0 in caso di fine del file e -1 quando si è verificato un errore.

Nei sistemi UNIX, il manuale completo è disponibile attraverso il comando `man`.

Comando `ps`: mostra i processi che sono in esecuzione.

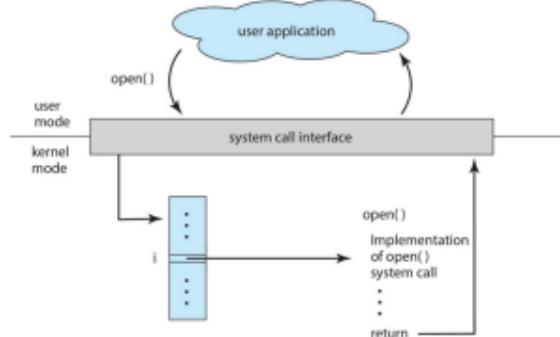
## Implementazione delle system call

Internamente ogni system call ha un numero associato ad esso.

L'interfaccia system call mantiene una tabella indicizzata che segue questi numeri. Questa interfaccia invoca la system call intesa nel kernel OS e ritorna lo stato della sys call e i valori di ritorno.

Il chiamante non ha alcuna necessità di conoscere l'implementazione della chiamata di sistema o i dettagli della sua esecuzione: gli è sufficiente essere conforme alla specifica dell'API e conoscere l'effetto dell'esecuzione della chiamata di sistema operativo. Ne consegue che la gran parte dei dettagli relativi alle chiamate di sistema è nascosta al programmatore delle API e gestita dal sistema di supporto all'esecuzione.

Le relazioni fra un'API, l'interfaccia alle chiamate di sistema e il sistema operativo sono illustrate in figura:



ove si mostra come il sistema operativo tratti l'invocazione della chiamata di sistema `open()` da parte di un'applicazione.

Le system call si presentano in modi diversi, secondo il calcolatore in uso. Spesso sono richieste maggiori informazioni oltre alla semplice identità della chiamata di sistema desiderata. Il tipo e la quantità delle informazioni variano secondo lo specifico sistema operativo e la specifica chiamata di sistema.

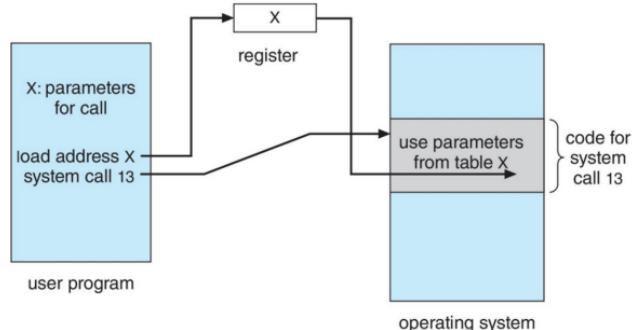
Per ottenere l'immissione di un dato, per esempio, può essere necessario specificare il file o il dispositivo da usare come sorgente e anche l'indirizzo e la dimensione del buffer in memoria in cui depositare i dati letti. Naturalmente, il dispositivo o il file e la dimensione possono essere impliciti nella chiamata di sistema.

Per passare parametri al sistema operativo si usano tre metodi generali:

1. *Passare i parametri in registri*: il caso più semplice; si possono però presentare casi in cui ci sono più parametri che registri
2. *Passare come parametro l'indirizzo del blocco*: si memorizzano i parametri in un blocco e si passa l'indirizzo del blocco, come parametro, in un registro
3. *Push dei parametri nello stack da cui sono prelevati dal sistema operativo*

Linux fa uso di una combinazione dei primi due approcci. Se sono presenti al massimo cinque parametri vengono utilizzati i registri, altrimenti viene utilizzato il metodo del blocco.

Alcuni sistemi operativi preferiscono i metodi del blocco o dello stack, perché non limitano il numero o la lunghezza dei parametri da passare.



## Categorie di chiamate di sistema

Le chiamate di sistema sono classificabili approssimativamente in 6 categorie principali:

1. controllo dei processi

2. *gestione dei file*
3. *gestione dei dispositivi*
4. *gestione delle informazioni*
5. *comunicazione*
6. *protezione*

## **1. Controllo dei processi**

- Creazione e arresto di un processo (fork() ed exit())
- Caricamento, esecuzione
- Terminazione normale e anomala (end(), abort())
- Esame e impostazione degli attributi di un processo
- Attesa per il tempo indicato (wait())
- Attesa e segnalazione di un evento
- Assegnazione e rilascio di memoria (serve per il debugging)
- Debugger per determinare bugs, single step execution
- Locks per gestire accessi ai dati condivisi tra i processi (servono ad evitare accessi concorrenti non desiderati)

## **2. Gestione dei file**

- Creazione e cancellazione di file (open())
- Apertura, chiusura (open(), close())
- Lettura, scrittura, posizionamento (read(), write())
- Esame e impostazione degli attributi di un file

## **3. Gestione dei dispositivi**

- Richiesta e rilascio di un dispositivo
- Lettura, scrittura, posizionamento (read(), write(), set())
- Esame e impostazione degli attributi di un dispositivo
- Inserimento logico ed esclusione logica di un dispositivo

## **4. Gestione delle informazioni**

- Esame e impostazione dell'ora e della data (alarm())
- Esame e impostazione dei dati del sistema (getpid())
- Esame e impostazione degli attributi dei processi, file e dispositivi

## **5. Comunicazione**

- Creazione e chiusura di una connessione
- Invio e ricezione di messaggi
- Informazioni sullo stato di un trasferimento
- Inserimento ed esclusione di dispositivi remoti

## **6. Protezione**

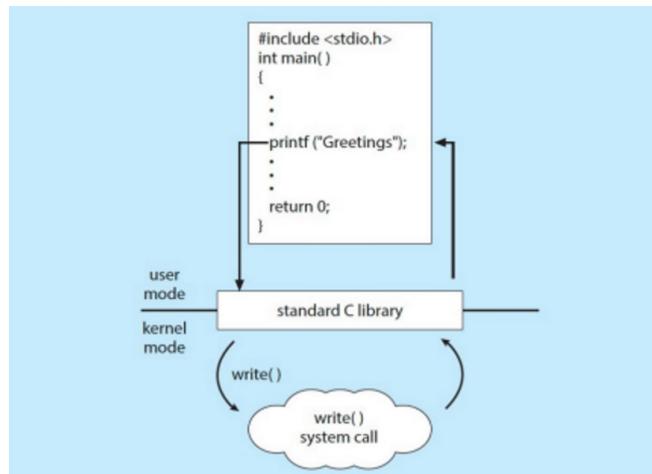
- Visualizzazione dei permessi di un file
- Impostazione dei permessi in un file (chmod(), umask(), chown())

Fondamentalmente, per quanto i sistemi operativi abbiano API diverse, le operazioni sono sempre le stesse.

## **Controllo dei processi**

### **La libreria standard del linguaggio C**

La libreria standard del linguaggio C fornisce una parte dell'interfaccia alle chiamate di sistema per molte versioni di UNIX e Linux. Come esempio, supponiamo che un programma C invochi la funzione printf(). La libreria C intercetta la funzione e invoca le necessarie system call: in questo caso la chiamata write(). La libreria riceve il valore restituito da write() e lo passa al programma utente.



Anche il programma C più semplice ha bisogno di sys call.

### FreeBSD

Derivato da Unix Barkley, anche se sono state rifatte un paio di versioni nuove lontane da LINUX.

Esso è un esempio di *sistema multitasking*.

Quando un utente inizia una sessione di lavoro, il sistema esegue un interprete dei comandi (shell) scelto dall'utente. Tuttavia, poiché il FreeBSD è un sistema multitasking, l'interprete dei comandi può continuare l'esecuzione mentre si esegue un altro programma. Per avviare un nuovo processo, la shell esegue la sys call `fork()`; si carica il programma selezionato in memoria tramite la sys call `exec()` e infine si esegue il programma. A seconda di come il programma è stato imparito, la shell attende il termine del processo oppure esegue il processo in background. In quest'ultimo caso la shell richiede immediatamente un altro comando. Se un processo è eseguito in background, non può ricevere dati direttamente dalla tastiera, giacché anche la shell sta usando tale risorsa. L'eventuale operazione di I/O è dunque eseguita tramite un file o tramite una GUI. Nel frattempo l'utente è libero di richiedere alla shell l'esecuzione di altri, di controllare lo svolgimento del processo in esecuzione, di modificare la priorità di quel programma e così via. Completato il proprio compito, il processo esegue una chiamata di sistema `exit()` per terminare la propria esecuzione, riportando al processo chiamante un codice di stato 0 oppure un codice d'errore diverso da 0. Questo codice di stato (o di errore) rimane disponibile per la shell o per altri programmi.

## Servizi di sistema

Un'altra caratteristica importante di un sistema moderno è quella che riguarda l'insieme dei servizi di sistema.

I servizi di sistema, detti anche *utilità di sistema*, offrono un ambiente più conveniente per lo sviluppo e l'esecuzione dei programmi; alcuni sono semplici interfacce per le system call, altri sono considerevolmente più complessi.

In generale, i servizi di sistema si possono classificare nelle seguenti categorie:

- *Gestione dei file*: questi programmi creano, cancellano, copiano, rinominano, stampano, elencano e in genere compiono operazioni sui file e le directory
- *Informazioni di stato*: alcuni programmi richiedono semplicemente al sistema di indicare data, ora, quantità di memoria disponibile o spazio nei dischi, numero degli utenti e simili informazioni di stato. Altri, più complessi, forniscono informazioni dettagliate su prestazioni, accessi al sistema e debug. In genere mostrano le informazioni su terminale, o tramite altri dispositivi per l'uscita dei dati o, ancora, all'interno di una finestra della GUI. Alcuni sistemi comprendono anche un registro, al fine di archiviare e poter consultare informazioni sulla configurazione del sistema (sistemi Windows)
- *Modifica dei file*: diversi editor sono disponibili per creare e modificare il contenuto di file memorizzati su dischi o altri dispositivi, oltre a comandi speciali per l'individuazione di contenuti di file o per particolari trasformazioni del testo
- *Ambienti di supporto alla programmazione*: compilatori, assemblatori, debugger e interpreti dei comuni linguaggi di programmazione sono spesso forniti con il sistema operativo oppure disponibili al download
- *Caricamento ed esecuzione dei programmi*: una volta assemblato o compilato, per essere eseguito, un programma deve essere caricato in memoria. Il sistema può mettere a disposizione caricatori assoluti, caricatori rilocabili, editor dei collegamenti, ecc. Sono necessari anche i sistemi di ausilio all'individuazione e correzione degli errori (debugger) per i linguaggi ad alto livello o per il linguaggio macchina
- *Comunicazioni*: offrono i meccanismi con cui si possono creare collegamenti virtuali tra processi, utenti e calcolatori diversi
- *Servizi in background*: tutti i sistemi general purpose hanno metodi per lanciare alcuni programmi di sistema al momento dell'avvio. Alcuni di questi processi terminano dopo aver completato i loro compiti, mentre altri restano in esecuzione fino a quando il sistema non viene arrestato. I processi di sistema costantemente in esecuzione sono noti come *deamons*. Un sistema tipico ha decine di deamons. I sistemi operativi che eseguono attività importanti in contesto utente invece che in kernel possono utilizzare appositi demoni per eseguire queste attività.

Oltre ai programmi di sistema, con la maggior parte dei sistemi operativi sono forniti *programmi che risolvono problemi comuni o che eseguono operazioni comuni*, noti come *programmi applicativi*. Essi comprendono browser web, word processor, fogli di calcolo, ecc.

## Link e loader

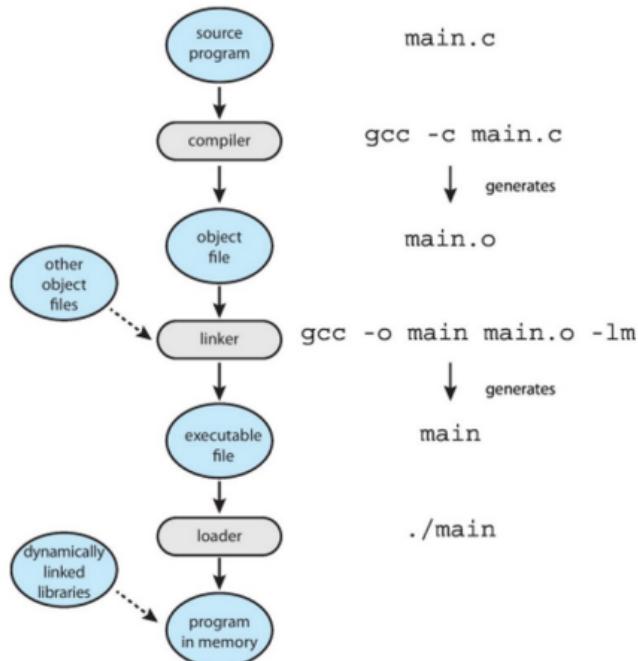
Generalmente un programma risiede su disco in forma di file binario eseguibile (.out o .exe). Per essere eseguito su una CPU, il programma dev'essere caricato in memoria e inserito nel contesto di un processo.

I file sorgenti vengono compilati in file oggetto progettati per essere caricati in una qualsiasi posizione della memoria fisica, noti come file oggetto rilocabili.

In seguito, il *linker* combina i file oggetto rilocabili in un singolo file binario eseguibile. Durante la fase di collegamento (*linking*) possono essere

inclusi altri file oggetto o alcune librerie.

Per caricare il file eseguibile in memoria, dove diventa idoneo per l'esecuzione sul core di una CPU, viene utilizzato un *loader*.



In Figura possiamo osservare che per eseguire il loader è sufficiente scrivere il nome del file eseguibile sulla riga di comando. Quando si immette il nome di un programma sulla riga di comando dei sistemi UNIX (per esempio, digitando `./main`) la shell crea innanzitutto un nuovo processo per eseguire il programma, utilizzando la sys call `fork()` e richiama poi il loader con la sys call `exec()`, passando ad essa il nome del file eseguibile. Il loader carica quindi in memoria il programma specificato, utilizzando lo spazio d'indirizzamento del processo appena creato.

C'è una sys call detta `open`. Il `main` è diviso in sezioni: `open(1)` indica che sono nella prima sezione, ovvero i programmi di sistema. La sys call `open` è invece nella sezione 2; devo indicare specificatamente la sezione quando ci sono queste sovrapposizioni tra nomi di programmi di sistema e system call.

La sezione 3 del manuale è quella delle librerie (library functions manual).

Utility `kill`: serve per uccidere un programma lanciandogli un segnale.

Comando `man kill`: vedo la parte del manuale sull'utility `kill`; non ci sono omologie quindi non specifico la sezione.

Con le librerie dinamiche (.so in UNIX, DLL in Windows) si evita di collegare e caricare librerie che potrebbero non essere utilizzate.

## Perché le applicazioni dipendono dal sistema operativo

Le applicazioni compilate su un sistema operativo non sono eseguibili su altri sistemi operativi.

Perché? Ogni sistema operativo fornisce un insieme univoco di chiamate di sistema. Questo vale anche per le librerie, dato che anche loro utilizzano sys call precompilate.

Se ho l'eseguibile di Windows, avrà un formato differente rispetto all'eseguibile di Linux. C'è inoltre il problema delle system call: non esistono programmi che non le utilizzano.

Non importa se i set di istruzioni del processore sono simili: le sys call rimangono diverse e quindi non è possibile eseguire programmi su un sistema operativo diverso anche se è fatto sulla stessa macchina.

Le applicazioni possono essere anche *multi-operating system*: in questo caso viene lanciato un programma attraverso un emulatore (*WINE* per Linux) che cambia e adatta le system call al nuovo sistema operativo.

Tutto è possibile con le macchine virtuali, a patto di avere abbastanza RAM a disposizione da dedicare alla VM.

## Progettazione e realizzazione di un sistema operativo

Non si dispone di soluzioni complete, ma vari approcci si sono rivelati efficaci.

### Scopi della progettazione

La progettazione del sistema è influenzata dalla scelta dell'architettura fisica e del tipo di sistema: desktop e laptop tradizionali, sistemi mobili, distribuiti o real-time.

D'altra parte, oltre questo livello di progettazione, i requisiti possono essere molto difficili da specificare anche se, in generale, si possono distinguere in due gruppi fondamentali:

1. *obiettivi degli utenti*: il sistema dev'essere facile da utilizzare, affidabile, sicuro e veloce
2. *obiettivi del sistema*: il sistema dev'essere facile da progettare, da realizzare e da manutenere

Non esiste una soluzione unica al problema della definizione dei requisiti di un sistema operativo.

Sono stati sviluppati alcuni principi generali nel campo del software engineering.

## Meccanismi e politiche

Un principio molto importante è quello che riguarda la distinzione tra meccanismi e criteri o politiche.

- *Meccanismi*: determinano *come* eseguire qualcosa
- *Criteri o politiche*: determinano *che cosa* si debba fare

Il timer del sistema, per esempio, è un meccanismo che assicura la protezione della CPU, ma la decisione riguardante la quantità di tempo da impostare nel timer per un utente specifico riguarda le politiche.

La distinzione tra meccanismi e politiche è molto importante ai fini della flessibilità.

Le politiche sono soggette a cambiamenti di luogo o di tempo. Nei casi peggiori, il cambiamento di una politica può richiedere il cambiamento del meccanismo sottostante.

Sarebbe preferibile disporre di meccanismi generali: in questo caso, un cambiamento di politica implicherebbe solo la ridefinizione di alcuni parametri del sistema.

I sistemi operativi basati su microkernel (che vedremo dopo) portano alle estreme conseguenze la separazione dei meccanismi dalle politiche, fornendo un insieme di funzioni fondamentali da impiegare come elementi di base; tali funzioni, quasi completamente indipendenti dalle politiche, consentono l'aggiunta di meccanismi e criteri più complessi tramite moduli del kernel creati dagli utenti o anche tramite programmi utente.

## Realizzazione

I primi sistemi operativi erano scritti in linguaggio assembly. Oggi la maggior parte è scritta in linguaggi ad alto livello come C o C++, con piccole porzioni di codice scritte in linguaggio assembly per gestire i registri.

Di fatto, viene spesso utilizzata una combinazione di diversi linguaggi ad alto livello: i livelli più bassi del kernel potrebbero essere scritte in C e C++ e le librerie di sistema in C++ o anche in linguaggi di più alto livello.

Android fornisce un buon esempio: il suo kernel è scritto principalmente in C, con piccole porzioni in linguaggio assembly, la maggior parte delle librerie di sistema è scritta in C o C++ e i suoi ambienti applicativi, che forniscono l'interfaccia al sistema per gli sviluppatori, sono scritti principalmente in Java.

I vantaggi derivanti dall'uso di un linguaggio ad alto livello per implementare un sistema operativo sono:

- il codice si scrive più rapidamente
- è più compatto
- è più facile da capire e mettere a punto
- il sistema è più facile da adattare a un'altra architettura (porting)

Svantaggi:

- minore velocità di esecuzione
- maggiore occupazione di spazio in memoria (superato nei sistemi moderni)

## Struttura del sistema operativo

Anziché progettare un sistema come un unico blocco, un orientamento diffuso è di \*suddividerlo in piccoli componenti detti **moduli**\*; ciascun modulo deve costituire una porzione ben definita del sistema, con interfacce e funzioni definite con precisione.

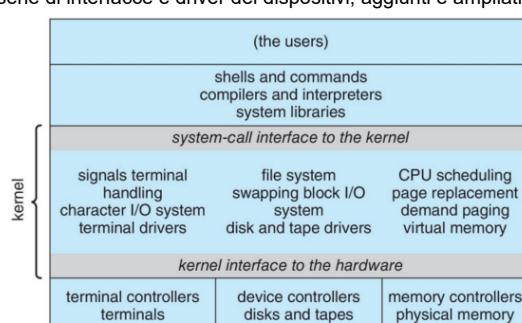
Struttura a livelli -> astrazione.

## Struttura monolitica

La struttura più semplice per l'organizzazione di un sistema operativo è l'assenza di struttura: *tutte le funzionalità del kernel vengono inserite in un singolo file binario statico che viene eseguito in un unico spazio d'indirizzamento*.

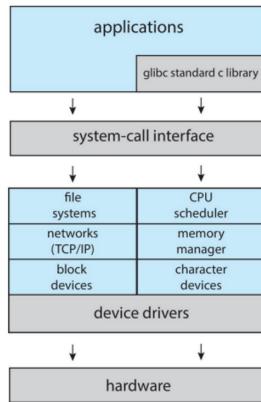
Questo approccio, noto come *struttura monolitica*, è una *tecnica comune di progettazione dei sistemi operativi*.

Un esempio di questa strutturazione è il **sistema operativo UNIX originale**, che consiste di due parti separate: il kernel e i programmi di sistema. Il kernel è ulteriormente suddiviso in una serie di interfacce e driver dei dispositivi, aggiunti e ampliati nel corso dell'evoluzione di UNIX.



Possiamo vedere UNIX come un sistema parzialmente stratificato. Sotto l'interfaccia alle chiamate del sistema e sopra l'hardware si trova il kernel, che fornisce il file system, lo scheduling della CPU, la gestione della memoria e altre funzionalità del sistema operativo attraverso le system call: si tratta di un'enorme quantità di funzionalità diverse combinate in un unico spazio di indirizzamento

Il sistema operativo **Linux** è basato su UNIX ed è strutturato in modo simile.



Le applicazioni utilizzano in genere la libreria standard del linguaggio C, glibc, quando comunicano con il kernel mediante l'interfaccia alle sys call.

Il kernel Linux è monolitico, in quanto viene eseguito interamente in modalità kernel in un unico spazio d'indirizzamento, ma è dotato di una struttura modulare che consente di modificare il kernel durante l'esecuzione: ci sono dei moduli precompilati che vengono aggiunti al kernel momentaneamente per aggiungere delle funzionalità (eseguiti in modalità supervisore).

Differenza con Windows: in Linux la GUI gira in modalità utente, in Windows in modalità sistema.

Svantaggi di kernel monolitici:

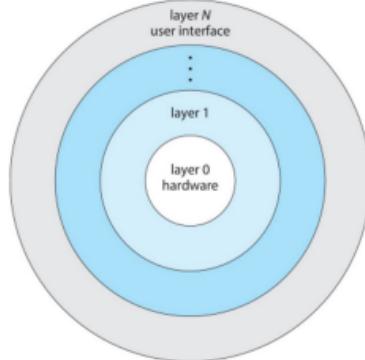
- difficili da implementare
- difficili da estendere

Vantaggi:

- l'interfaccia alle sys call presenta un overhead molto ridotto
- la comunicazione all'interno del kernel è veloce

Pertanto, nonostante gli svantaggi dei kernel monolitici, la loro velocità e la loro efficienza giustificano la presenza di elementi di una struttura di questo tipo.

## Approccio stratificato



Il sistema è suddiviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (strato 0), il più alto all'interfaccia con l'utente (strato N).

Lo *strato di un sistema operativo* è la *realizzazione di un oggetto astratto*, che incapsula i dati e le operazioni che trattano tali dati.

Un tipico strato di sistema operativo è composto da strutture dati e da un insieme di routine richiamabili dagli strati di livello più alto. Lo stato in questione, a sua volta, è in grado di invocare operazioni degli strati di livello inferiore.

Ogni strato si realizza impiegando unicamente le operazioni messe a disposizione dagli strati inferiori, considerando soltanto le azioni che compiono, senza entrare nel merito di come queste sono realizzate. Di conseguenza, ogni strato nasconde a quelli superiori l'esistenza di determinate strutture dati, operazioni e hardware.

Per motivi pratici non è molto efficiente ed offre prestazioni scarse, di conseguenza non viene utilizzato effettivamente nei sistemi operativi.

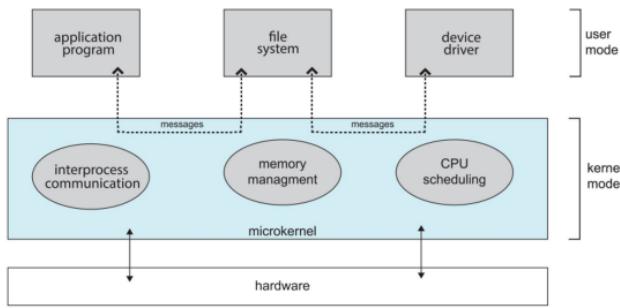
## Microkernel

Idea nata verso la metà degli anni '80.

Si progetta il sistema operativo *rimuovendo dal kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema*.

Ne risulta un kernel di dimensioni assai inferiori.

Non c'è un'opinione comune su quali servizi debbano rimanere nel kernel e quali si debbano realizzare nello spazio utente. Tuttavia, in generale, un *microkernel offre i servizi minimi di gestione dei processi, della memoria e di comunicazione*.



L'esempio più noto di un sistema operativo microkernel è **Darwin**, il componente kernel dei sistemi operativi macOS e iOS.

Lo scopo principale del microkernel è fornire funzioni di comunicazione tra i programmi client e i vari servizi, anch'essi in esecuzione nello spazio utente. La comunicazione viene realizzata mediante scambi di messaggi.

Per accedere a un file, per esempio, un programma client deve interagire con il file server; ciò non avviene mai in modo diretto, ma tramite uno scambio di messaggi con il microkernel.

Svantaggi:

- cali di prestazioni dovuti all'aumento dell'overhead
  - più lento poiché c'è uno scambio di messaggi tra il kernel e l'user mode
- Vantaggi:
- facilità di estensione del sistema operativo -> i nuovi servizi si aggiungono allo spazio utente e non comportano modifiche al kernel
  - offre maggiori garanzie di sicurezza e affidabilità, poiché i servizi si eseguono in gran parte come processi utente e non come processi del kernel -> se un servizio è compromesso, il resto del sistema operativo rimane intatto
  - il kernel è compatto

*Lo scheduling della CPU, la gestione della memoria e la comunicazione tra processi sono le parti fondamentali per far girare i programmi.*

## Come viene implementata una system call?

Una system call (chiamata di sistema) è una routine che un processo esegue per richiedere un servizio al sistema operativo. Questo servizio può essere l'accesso a un dispositivo hardware, l'allocazione di memoria, l'apertura di un file o la creazione di un processo. Una system call è un'operazione privilegiata e, di conseguenza, deve essere eseguita in modo sicuro dal sistema operativo.

Le system call sono implementate come parte del kernel del sistema operativo e sono accessibili al livello utente tramite librerie che forniscono un'interfaccia per le chiamate di sistema. Ad esempio, la funzione open in C è una system call che viene utilizzata per aprire un file sul sistema. Implementazione: internamente ogni system call ha un numero associato -> l'interfaccia system call mantiene una tabella indicizzata che segue questi numeri. La sys call interface invoca la sys call intesa nel kernel del sistema operativo e ritorna lo stato della sys call e i valori di ritorno. Le system call sono importanti perché forniscono un modo per i processi di accedere ai servizi del sistema operativo in modo sicuro e controllato. Inoltre, le system call consentono ai programmatori di scrivere programmi che possono interagire con il sistema operativo e con altri processi sul sistema.

Altra risposta:

Una system call (chiamata di sistema) è una routine che un processo esegue per richiedere un servizio al sistema operativo. Questo servizio può essere l'accesso a un dispositivo hardware, l'allocazione di memoria, l'apertura di un file o la creazione di un processo. Una system call è un'operazione privilegiata e, di conseguenza, deve essere eseguita in modo sicuro dal sistema operativo. Le system call sono implementate come parte del kernel del sistema operativo e sono accessibili al livello di utente tramite librerie che forniscono un'interfaccia per le chiamate di sistema. Ad esempio, la funzione "open" in C è una system call che viene utilizzata per aprire un file sul sistema. Implementazione → Internamente ogni system call ha un numero associato → L'interfaccia system call mantiene una tabella indicizzata che segue questi numeri. La sys call interface invoca la sys call intesa nel kernel del sistema operativo e ritorna lo stato della sys call e i valori di ritorno. Le system call sono importanti perché forniscono un modo per i processi di accedere ai servizi del sistema operativo in modo sicuro e controllato. Inoltre, le system call consentono ai programmatori di scrivere programmi che possono interagire con il sistema operativo e con altri processi sul sistema.

## Cos'è il kernel? Come funziona?

Il metodo tradizionale di implementare un sistema operativo consiste nel mettere sopra la HW uno strato di software detto kernel che in qualche maniera fornisce l'interfaccia per le system call e fornisce diversi servizi i quali: gestione del terminale, sistema di I/O, file system, CPU scheduling, rimpiazzo delle pagine, memoria virtuale; questo software è sempre presente in memoria e viene attivato tramite le interruzioni. Quando arrivano le interruzioni da parte dell'HW oppure quando qualcuno si interfaccia con il kernel mediante delle trap (si tratta di interrupt generate via software che in pratica identificano il metodo che viene utilizzato per implementare le syscall). Il kernel si mette in funzione. Nel momento in cui eseguo una system call attivo il SO che passa in modalità protetta. Il metodo tradizionale di implementare il sistema il SO ogniqualvolta che io richiedo un servizio al SO tutto quello che richiedo attraversa l'interfaccia delle system call e viene eseguita in modalità protetta (tutto quello che si trova al di sotto dell'interfaccia delle syscall viene eseguito in modalità protetta) dopodiché si ritorna dalla syscall e si ritorna dai programmi utente. Altro metodo utilizzato per implementare il sistema operativo consiste nell'inserire sopra l'HW uno strato di software di piccole dimensioni che si chiama microkernel il quale fornisce delle funzionalità minime, il minimo indispensabile che mi permette di far eseguire i programmi e farli comunicare tra di loro.

## Cos'è un microkernel?

Un microkernel è un'architettura del sistema operativo in cui la maggior parte delle funzionalità del sistema operativo sono implementate come processi separati che girano sulla stessa macchina. Il microkernel stesso fornisce solo le funzioni di base, come la gestione della memoria e la gestione della comunicazione tra i processi.

L'obiettivo di un microkernel è di mantenere il sistema operativo il più semplice possibile, in modo da ridurre la quantità di codice che dev'essere eseguito con privilegi elevati e quindi ridurre la quantità di codice che può essere soggetto a vulnerabilità di sicurezza. Inoltre, il design modulare di un microkernel consente una maggiore flessibilità, poiché è possibile aggiungere o rimuovere funzionalità del sistema operativo senza dover modificare il nucleo stesso.

L'esempio più noto di un sistema operativo basato su microkernel è QNX, che viene utilizzato in molte applicazioni embedded industriali. Tuttavia, il design del microkernel non è così diffuso come quello dei sistemi operativi basati sul kernel monolitico, come Windows o Linux, a causa della maggiore complessità e delle prestazioni inferiori rispetto ai sistemi operativi basati su kernel monolitico.

## Differenza tra kernel e microkernel

Il kernel è il nucleo centrale di un sistema operativo che gestisce le risorse hardware, come la memoria, i processi e i dispositivi di I/O. Un microkernel, d'altra parte, è una variante del kernel che fornisce solo le funzionalità di base necessarie a supportare l'interazione tra i componenti di un sistema operativo.

La differenza principale tra i due sta nella loro architettura: un kernel tradizionale contiene molte funzionalità e servizi integrati all'interno del nucleo stesso, mentre un microkernel fornisce solo le funzionalità di base e lascia che altri componenti del sistema operativo forniscano funzionalità di base e lascia che altri componenti del sistema operativo forniscano funzionalità più avanzate come servizi esterni.

Un altro aspetto che distingue i due è la loro flessibilità e la loro capacità di adattarsi a nuove esigenze: un microkernel è più flessibile in quanto consente di modificare o aggiungere funzionalità senza dover riscrivere l'intero kernel, mentre un kernel tradizionale può essere più difficile da modificare a causa della sua architettura integrata.

In sintesi, un kernel tradizionale fornisce una soluzione più completa e integrata per gestire le risorse di un sistema operativo, mentre un microkernel fornisce una soluzione più flessibile e modificabile che consente di adattarsi alle esigenze specifiche di un sistema operativo.

Altra risposta:

Un sistema operativo multi-kernel considera una macchina multi-core come una rete di core indipendenti, come se fosse un sistema distribuito . Non presuppone la memoria condivisa, ma piuttosto implementa le comunicazioni tra processi come passaggio di messaggi. Il microkernel invece è strutturato su un insieme di moduli adatti a funzioni specifiche in modo da isolare il kernel.

Linux è microkernel? No, è monolitico. ciò significa che vi è un'assenza di struttura tutte le funzionalità del terminal vengono inserite in un singolo file binario statico che viene seguito in un unico spazio di indirizzamento. Il kernel è ulteriormente suddiviso in una serie di interfacce i driver dei dispositivi aggiungi ampliati nel corso dell'evoluzione di UNIX (Sistema strettamente accoppiato, perché le modifiche a una parte del sistema possono avere effetti di ampia portata su altre parti )

# Capitolo 3 - Sistemi Operativi

## I processi

### Concetto di processo

I programmi in esecuzione sono detti processi; essi, quindi, sono un'entità dinamica.

Persino in un sistema mono-utente un utente può far eseguire diversi programmi contemporaneamente: un word processor, un browser, un programma di posta elettronica. Anche se l'utente esegue un solo programma alla volta, come avviene sui dispositivi embedded che non supportano il multitasking, il sistema operativo deve svolgere le proprie attività interne, per esempio la gestione della memoria. Queste attività sono simili per molti aspetti, perciò sono denominate *processi*.

### Il processo

Lo stato dell'attività corrente di un processo è rappresentato dal valore del contatore di programma e dal contenuto dei registri del processore. La struttura di un processo in memoria è generalmente suddivisa in più sezioni, che sono le seguenti:

- *Sezione di testo*: contiene il codice eseguibile
- *Sezione dati*: contenente le variabili globali
- *Heap*: memoria allocata dinamicamente durante l'esecuzione del programma
- *Stack*: memoria temporaneamente utilizzata durante le chiamate di funzioni

Si noti che le dimensioni delle sezioni di testo e dati sono fisse, ovvero non cambiano mai durante l'esecuzione del programma, mentre le sezioni di stack e heap possono ridursi e crescere dinamicamente durante l'esecuzione.

Ogni volta che si chiama una funzione, un *record di attivazione* contenente i suoi parametri, le variabili locali e l'indirizzo di ritorno viene inserito nello stack; quando la funzione restituisce il controllo al chiamante, il record di attivazione viene rimosso dallo stack.

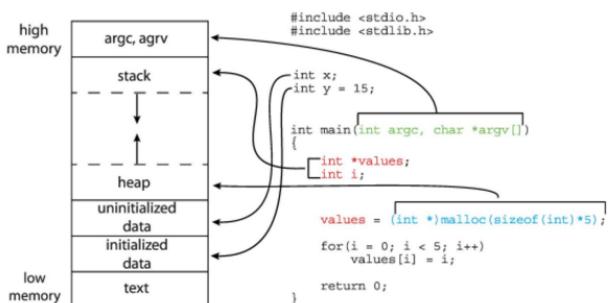
Allo stesso modo, l'heap crescerà quando viene allocata memoria dinamicamente e si ridurrà quando la memoria viene restituita al sistema. Visto che le sezioni di stack e heap crescono l'una verso l'altra, tocca al sistema operativo garantire che non si sovrappongano.

Sottolineiamo che un programma di per sé non è un processo; un programma è un'entità passiva (es. file eseguibile), mentre un processo è un'entità attiva.

Un programma diventa un processo allorquando il file eseguibile è caricato in memoria.

Due tecniche comuni per ottenere questo effetto sono il doppio clic sull'icona del file eseguibile e la digitazione del nome del file eseguibile nella riga di comando.

### Struttura in memoria di un programma C



La Figura mostra la struttura di un programma C in memoria, evidenziando la relazione tra le diverse sezioni di un processo e un programma C reale. Questa figura è simile alla rappresentazione generale di un processo in memoria, con alcune differenze.

- La sezione dei dati globali è suddivisa in sezioni distinte per (a) dati inizializzati e (b) dati non inizializzati
- E' presente una sezione separata per i parametri argc e argv passati alla funzione main()

Commento al codice:

- argc, argv -> argomenti
- stack -> variabili automatiche
- heap -> variabili allocate dinamicamente / liberare con free()
- Le variabili static non vengono esportate dal linker, ma vivono solo nel file .c in cui sono dichiarate
- int x; int y = 15; -> visibili da tutte le funzioni
- main() -> funzione
- int \* values; int i; -> con static diventa dato non inizializzato; variabili visibili solo alla funzione

Il comando GNU size può essere usato per determinare la dimensione (in byte) di alcune di queste sezioni.

## Sommario:

argc e argv fanno parte dell'environment e sono esterni allo spazio indirizzi. Le variabili x e y vanno nell'area dati.

Tutto quello che viene dichiarato nelle funzioni viene allocato in stack. Questo include le variabili dichiarate al loro interno (variabili automatiche).

Hanno visibilità ridotta alla sola funzione.

Variabili static: utilizzate per indicare che una variabile va in area dati globali.

Se metto static davanti ad una funzione, essa non sarà visibile dal linker e sarà limitata al file .c. Questo può essere una precauzione nel caso ci siano delle funzioni con lo stesso nome in file diversi.

Si utilizza la malloc() per lasciare lo spazio a 5 interi: cresce l'heap, a cui viene assegnata questa parte del processo. Quest'area di memoria ritornerà libera in seguito alla free().

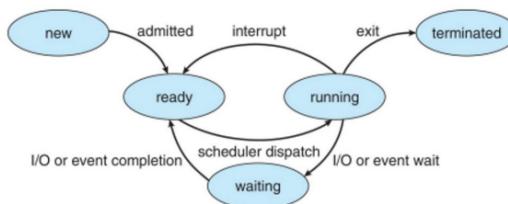
Le variabili globali sono esportate dai linker, mentre le variabili locali non sono visibili agli altri .c.

Per vedere le variabili globali devo chiamarle con la parola chiave "external".

## Stato del processo

Un processo durante l'esecuzione è soggetto a cambiamenti del suo stato, definito in parte dall'attività corrente del processo stesso. Un processo può trovarsi in uno tra i seguenti stati:

- *New*: si crea il processo (tipico dei sistemi batch, ma resta in una coda)
- *Running*: le sue istruzioni vengono eseguite
- *Waiting*: il processo attende che si verifichi qualche evento (come il completamento di un'operazione di I/O)
- *Ready*: il processo attende di essere assegnato ad un'unità di elaborazione (non ha ancora una CPU)
- *Terminated*: il processo ha terminato l'esecuzione



Dispatch: prelievo e assegnazione della CPU.

Scheduler: parte del sistema operativo che decide chi è il prossimo processo ad andare in esecuzione.

*E' importante capire che in ciascuna unità di elaborazione può essere in esecuzione un solo processo per volta, sebbene molti processi possano essere pronti o nello stato di attesa.*

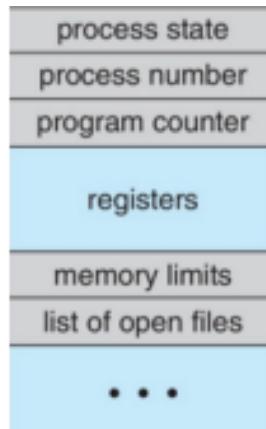
Un processo perde la CPU se:

- è stato terminato
- arriva un interrupt dalle periferiche, dal timer che genera un'interruzione periodica e il programma torna nello stato ready
- il processo attende un evento o un'operazione di I/O; è tirato fuori dallo stato di waiting dopo che l'evento o l'operazione di I/O è completato

## Blocco di controllo del processo (PCB)

Il process control block (PCB) viene preparato quando il processo è nello stato ready.

Ogni processo è rappresentato nel sistema operativo da un blocco di controllo, detto blocco di controllo del processo; le informazioni sono disseminate per svariate parti della memoria.



Un PCB contiene molte informazioni connesse a un processo specifico, tra cui le seguenti:

- *Stato del processo*: lo stato può essere new, ready, running, waiting, terminated
- *Contatore di programma*: contiene l'indirizzo della successiva istruzione da eseguire per tale processo
- *Registri della CPU*: variano in numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, registri indice, stack pointer, registri d'uso generale e registri contenenti i codici di condizione. Quando si verifica un'interruzione della CPU si devono salvare tutte

queste informazioni insieme al contatore di programma, in modo da permettere la corretta esecuzione del processo in un momento successivo, quando viene ri-schedulato

- *Informazioni sullo scheduling di CPU:* comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling
- *Informazioni sulla gestione della memoria:* possono includere elementi quali il valore dei registri di base e di limite, le tabelle delle pagine o le tabella dei segmenti, a seconda del sistema di gestione della memoria usato dal sistema operativo
- *Informazioni di accounting:* comprendono la quota di uso della CPU e il tempo d'utilizzo della stessa, i limiti di tempo, i numeri dei processi, ecc. Dopo un certo tempo i processi vengono abortiti perché evidentemente si sono bloccati
- *Informazioni sullo stato dell'I/O:* comprendono la lista dei dispositivi di I/O assegnati a un determinato processo, l'elenco dei file aperti e così via

In sintesi, il PCB si usa semplicemente come deposito per tutte le informazioni relative ai vari processi.

## Thread

\*Un processo è un programma che si esegue seguendo un unico percorso di esecuzione detto **thread**\*

Se un processo sta, per esempio, eseguendo un browser, l'esecuzione avviene secondo una singola sequenza di istruzioni; quindi il processo può svolgere un solo compito alla volta.

Nella maggior parte dei sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità di avere più percorsi di esecuzione, in modo da permettere che un processo possa svolgere più di un compito alla volta.

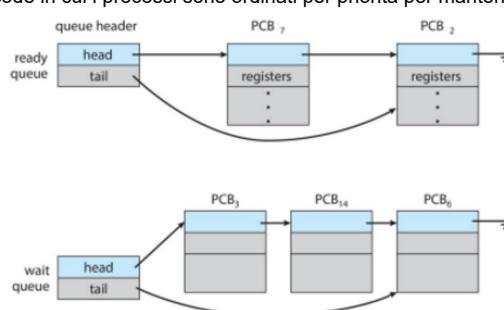
Alcuni sistemi operativi moderni prevedono che all'interno di un processo ci possano essere più esecuzioni contemporanee del codice in area text. Questa funzione è particolarmente utile sui sistemi multicore, in cui più thread possono essere eseguiti in parallelo.

In un sistema che supporta i thread, il PCB viene esteso per includere informazioni su ogni thread. Per supportare i thread sono necessari anche altri cambiamenti nel sistema.

## Scheduling dei processi

Il sistema operativo deve schedulare i processi in maniera tale da massimizzare l'utilizzo della CPU. Il passaggio da un processo all'altro dev'essere veloce.

Lo scheduler dei processi mantiene delle code in cui i processi sono ordinati per priorità per mantenere la CPU occupata.



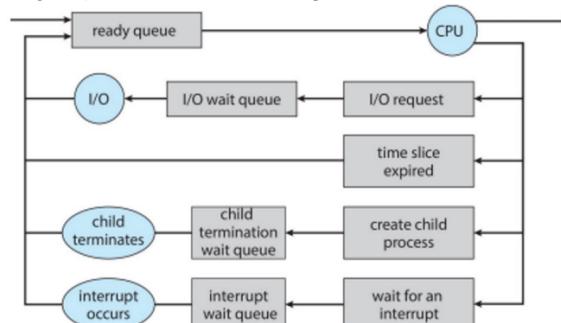
## Code di scheduling

Entrando nel sistema, \*ogni processo è inserito in una coda di processi pronti e in attesa di essere eseguiti, detta **coda dei processi pronti**\*

Questa coda generalmente viene memorizzata come una **coda concatenata**: un'intestazione della coda dei processi pronti contiene i puntatori al primo PCB della lista, e ciascun PCB comprende un campo puntatore che indica il successivo processo contenuto nella coda.

I processi in attesa di un determinato evento, per esempio il completamento dell'I/O, vengono collocati in una **coda d'attesa**.

Una comune rappresentazione dello scheduling dei processi è data da un **diagramma di accodamento** come quello mostrato in Figura:



Sono presenti due tipi di coda: la ready queue e un insieme di wait queue. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso di processi nel sistema.

Un nuovo processo si colloca inizialmente nella ready queue, dove attende finché non è selezionato per essere eseguito (dispatched). Una volta che il processo è assegnato alla CPU ed è in fase di esecuzione, si può verificare uno dei seguenti eventi:

- il processo può emettere una richiesta di I/O e quindi essere inserito in una coda di I/O
- il processo può creare un nuovo processo figlio e attenderne la terminazione

- il processo può essere rimosso forzatamente dalla CPU a causa di un'interruzione, ed essere reinserito nella coda dei processi pronti.

Nei primi due casi, al completamento della richiesta di I/O o al termine del processo figlio, il processo passa dallo stato d'attesa allo stato pronto ed è nuovamente inserito nella ready queue.

Un processo continua questo ciclo fino al termine della sua esecuzione: a questo punto viene rimosso da tutte le code e vengono deallocati il suo PCB e le varie risorse.

N.B.: processo figlio -> processo generato da un altro processo. Il primo processo viene eseguito quando avviene il boot del sistema.

## Scheduling della CPU

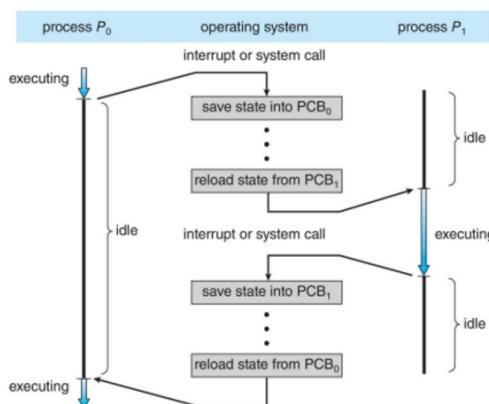
Nel corso della sua esistenza, un processo si sposta ripetutamente tra la coda dei processi pronti e diverse code di attesa. Il ruolo dello scheduler della CPU è *selezionare un processo nella coda dei processi e allocarlo ad un core della CPU*. Questa selezione dev'essere effettuata frequentemente.

Lo scheduling della CPU avviene almeno una volta ogni 100 millisecondi, anche se generalmente ciò avviene molto più frequentemente.

Alcuni sistemi operativi hanno una forma intermedia di scheduling, nota come *swapping*, la cui idea chiave è che a volte possa essere vantaggioso eliminare processi dalla memoria (e dalla contesa per la CPU), riducendo il grado di multiprogrammazione del sistema. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta. Lo swapping è in genere necessario solo quando la memoria è sovrautilizzata e deve essere liberata.

## Cambio di contesto (context switch)

Diagramma del cambio di contesto:



Le interruzioni forzano il sistema a sospendere il lavoro attuale della CPU per eseguire routine del kernel. Le interruzioni sono eventi comuni nei sistemi general-purpose. In presenza di un'interruzione, il sistema deve salvare il contesto del processo corrente, per poi ripristinarlo quando il processo stesso potrà tornare in esecuzione.

Il contesto è rappresentato all'interno del PCB del processo, e comprende i valori dei registri della CPU, lo stato del processo e informazioni relative alla gestione della memoria. In termini generali, si esegue un *salvataggio dello stato corrente della CPU*, sia che essa sia eseguita in modalità utente o in modalità di sistema; in seguito, si attuerà un corrispondente *ripristino dello stato* per poter riprendere l'elaborazione dal punto in cui era stata interrotta.

**Cambio di contesto:** il passaggio della CPU a un nuovo processo implica il *salvataggio dello stato del processo attuale* e il *ripristino dello stato del nuovo processo*.

Lo switch dev'essere il più veloce possibile: quel tempo di passaggio tra registri è tutto overhead.

La durata del cambio di registro dipende molto dall'architettura e da quanti registri devo salvare. Queste parti sono scritte in assembly, dato che solo con questo linguaggio posso manipolare i registri, e sono ottimizzate per minimizzare il tempo di context switch.

I sistemi moderni sono tutti time-shared: ogni millisecondo, se il processo non è stato ancora chiuso, viene chiuso da un'interrupt generata dal timer e viene chiamato un altro processo.

## Operazioni sui processi

Nella maggior parte dei sistemi i processi si possono eseguire in modo concorrente, e si devono creare e cancellare dinamicamente; a tal fine il sistema operativo deve offrire un meccanismo che permetta di creare e terminare il processo.

### Creazione di un processo

Durante la propria esecuzione, un processo può creare numerosi nuovi processi. Come menzionato in precedenza, il *processo creante* si chiama *processo genitore* (o padre), mentre il *nuovo processo* si chiama *processo figlio*. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un *albero di processi*.

La maggior parte dei sistemi operativi identifica un processo per mezzo di un *numero univoco*, solitamente un intero, detto *identificatore di processo* o *pid*. Il *pid* fornisce un valore univoco per ogni processo del sistema e può essere usato come indice per accedere a vari attributi di un processo all'interno del kernel.

Opzioni di condivisione delle risorse:

- padri e figli condividono le risorse (spazio di memoria, proprietà di scheduling, file aperti, ecc.)

- i figli condividono il sottoinsieme delle risorse del padre (caso più comune)
- i padri e figli non condividono risorse

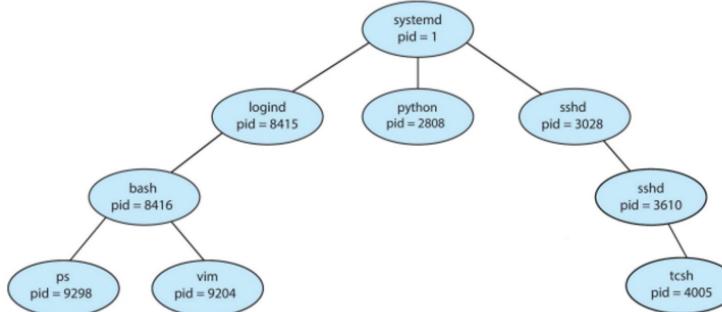
Opzioni di esecuzione:

- il processo genitore continua l'esecuzione in modo concorrente con i propri processi figli (più adottata)
- il processo genitore attende che alcuni o tutti i suoi processi figli terminino (questo significa che non può lanciare più figli alla volta, il che è scomodo)

Ci sono due possibilità per quel che riguarda lo spazio d'indirizzi del nuovo processo:

- il processo figlio è un duplicato del processo genitore (ha gli stessi programmi e dati del genitore) (non molto utile)
- nel processo figlio si carica un nuovo programma

#### Albero dei processi del sistema Linux



Il processo **systemd** (che ha sempre pid pari a 1) svolge il ruolo di processo padre di tutti i processi utente (processo init). Una volta che il sistema si è avviato, il processo **systemd** può creare vari processi utente, per esempio un server web o per la stampa, un server **ssh** e processi simili.

In figura vediamo due processi figli di **systemd**, **logind** e **sshd**. Il processo **logind** è responsabile della gestione dei client che si collegano direttamente al sistema. In questo esempio, un client ha effettuato l'accesso e sta utilizzando la shell **bash**, a cui è stato assegnato pid 8416. Utilizzando l'interfaccia a riga di comando **bash**, questo utente ha creato il processo **ps** e l'editor **vim**. Il processo **sshd** è responsabile della gestione dei client che si connettono al sistema tramite **ssh** (secure shell).

Daemon **systemd**: versione più moderna e performante del daemon.

**Loginid**: gestisce le connessioni locali.

Parentesi:

Connessione non criptata: quando ci connettiamo, la nostra password parte in chiaro. **sshd** permette la connessione in remoto criptata.

Come si crea un processo figlio in un sistema?

Address space:

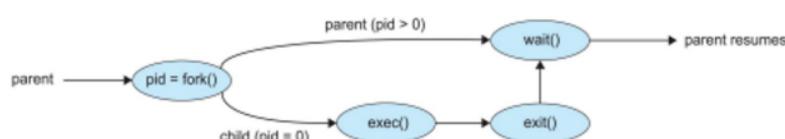
- il figlio è un duplicato del genitore (in Unix)
- il figlio ha un programma diverso caricato al suo interno (in Windows)

Un nuovo processo si crea per mezzo della sys call **fork()**, ed è composto di una copia dello spazio degli indirizzi del processo genitore. Questo meccanismo consente al processo genitore di comunicare senza difficoltà con il proprio processo figlio.

Entrambi i processi (genitore e figlio) continuano l'esecuzione all'istruzione successiva alla chiamata di sistema **fork()**, con una differenza: la chiamata di sistema **fork()** riporta il valore 0 nel nuovo processo (il figlio), ma riporta l'identificatore del processo figlio (il pid diverso da zero) nel processo genitore.

Generalmente, dopo una chiamata di sistema **fork()**, uno dei due processi impiega una chiamata di sistema **exec()** per sostituire lo spazio di memoria del processo con un nuovo programma. La chiamata di sistema **exec()** carica in memoria un file binario, cancellando l'immagine di memoria del programma contenente la stessa chiamata di sistema **exec()**, quindi avvia la sua esecuzione. In questo modo i due processi possono comunicare e poi procedere in modo diverso. Il processo genitore può anche generare più processi figli, oppure, se durante l'esecuzione del processo figlio non ha nient'altro da fare, può invocare la chiamata di sistema **wait()** per rimuovere se stesso dalla coda dei processi pronti fino alla terminazione del figlio.

Poiché la chiamata **exec()** sovrappone lo spazio di indirizzi del processo con un nuovo programma, essa non restituisce il controllo a meno che non si verifichi un errore.



#### Programma C per fare il fork su processi separati

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    execp("./bin/ls","ls",NULL);
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
}

return 0;
}

```

Il programma C in figura illustra le chiamate di sistema UNIX descritte precedentemente. Abbiamo due processi distinti, ciascuno dei quali esegue una copia dello stesso programma. Il valore assegnato alla variabile pid è 0 nel processo figlio e un numero intero > 0 nel processo genitore. Il processo figlio eredita privilegi, attributi di scheduling e alcune risorse, come i file aperti, dal processo genitore. Usando la chiamata di sistema exec() il processo figlio sovrappone il proprio spazio d'indirizzi con il comando /bin/ls di UNIX (che si usa per ottenere l'elenco del contenuto di una directory). Eseguendo la chiamata di sistema wait(), il processo genitore attende che il processo figlio termini. Quando ciò accade (implicitamente o esplicitamente mediante l'invocazione di exit()), il processo genitore chiude la propria fase d'attesa dovuta alla chiamata di sistema wait() e termina usando la chiamata di sistema exit().

Naturalmente, non c'è modo di impedire al figlio di non invocare exec() e restare in esecuzione come una copia del processo genitore. In questo scenario, il genitore e il figlio sono processi concorrenti che eseguono lo stesso codice. Poiché il figlio è una copia del genitore, ogni processo ha la propria copia dei dati.

## Terminazione di un processo

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltre la richiesta al sistema operativo di essere cancellato usando la sys call exit(); a questo punto, il processo figlio può riportare un'informazione di stato al processo genitore, che la riceve attraverso la chiamata di sistema wait(). Tutte le risorse del processo, incluse la memoria fisica e virtuale, i file aperti e le aree della memoria per l'I/O, sono liberate dal sistema operativo.

La terminazione di un processo si può verificare anche in altri casi. Un processo può causare la terminazione di un altro per mezzo di un'opportuna sys call. Generalmente solo il genitore del processo che si vuole terminare può invocare una sys call di questo tipo, altrimenti gli utenti potrebbero causare arbitrariamente la terminazione forzata di processi di chiunque. Occorre notare che un genitore deve conoscere le identità dei propri figli per terminarli, perciò quando un processo ne crea uno nuovo, l'identità del nuovo processo viene passata al processo genitore.

Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli per diversi motivi, tra i quali i seguenti:

- il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate. Ciò richiede che il processo genitore disponga di un sistema che esamini lo stato dei propri processi figli
- il compito assegnato al processo figlio non è più richiesto
- il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza

In alcuni sistemi, se un processo termina si devono terminare anche i suoi figli, indipendentemente dal fatto che la terminazione del genitore sia stata normale o anormale. Si parla di *terminazione a cascata*, una procedura avviata di solito dal sistema operativo.

Nella stragrande maggioranza dei sistemi, come UNIX, si permette al figlio di continuare ad esistere anche se il processo padre è terminato.

Un processo che è terminato, ma il cui genitore non ha ancora chiamato la wait() è detto *processo zombie*.

In questo caso init potrebbe fare wait (operazione di reaping), espellendo il figlio zombie dal sistema. In questo caso, con il comando ps su bash, sotto PPID (parent pid) verrà indicato 1, il che vuol dire che init ha adottato il figlio.

Tutti i processi passano in questo stato quando terminano, ma in genere vi rimangono solo per un breve tempo. Una volta che il genitore chiama la wait() il pid del processore zombie e la sua voce nella tabella dei processi vengono rilasciati.

Se il processo padre termina per errore senza invocare wait() il processo figlio è detto orfano.

## Comunicazioni tra processi (Inter-Process Communication, IPC)

I processi eseguiti concorrentemente nel sistema operativo possono essere indipendenti o cooperanti.

Un processo è *indipendente* se non può influire su altri processi del sistema o subirne l'influsso. Chiaramente, un processo che non condivide dati con altri processi è indipendente.

Un processo è *cooperante* se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Un qualsiasi processo che condivide dati con altri processi è un processo cooperante.

Un ambiente che consente la cooperazione tra processi può essere utile per diverse ragioni:

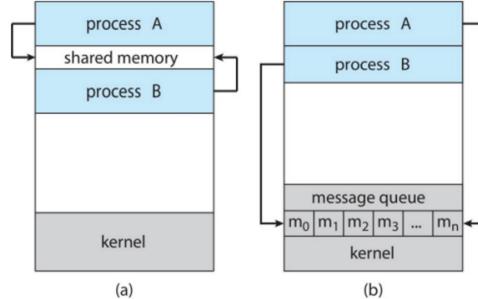
- *Condivisione d'informazioni*: poiché più utenti possono essere interessati alle stesse informazioni (per esempio un file condiviso) è necessario offrire un ambiente che consenta un accesso concorrente a tali risorse
- *Velocizzazione del calcolo*: alcune attività d'elaborazione sono realizzabili più rapidamente se si suddividono in sottoattività eseguibili in parallelo. Un'accelerazione di questo tipo è ottenibile solo se il calcolatore dispone di più core di elaborazione
- *Modularità*: può essere utile la costruzione di un sistema modulare, che suddivide le funzioni di sistema in processi o thread distinti

- **Convenienza**

Per lo scambio di dati e di informazioni i processi cooperanti necessitano di un meccanismo di *comunicazione tra processi (IPC)*. I modelli fondamentali della comunicazione tra processi sono due:

- A *Memoria Condivisa*: si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono comunicare scrivendo e leggendo da tale zona (possibile solo con memoria fisica condivisa)
- A *Scambio di Messaggi*: la comunicazione ha luogo tramite scambio di messaggi cooperanti

(a) Shared memory. (b) Message passing.



Il modello a memoria condivisa è più performante, ma se non c'è fisicamente una memoria disponibile, il modello a scambio di messaggi è l'unica alternativa.

Il passaggio di messaggi non è performante quando c'è una memoria condivisa: è utile principalmente per permettere la comunicazione tra processi su macchine diverse.

Nei sistemi operativi sono diffusi entrambi i modelli; spesso coesistono in un unico sistema (poco conveniente). Lo scambio di messaggi è utile per trasmettere piccole quantità di dati, non essendovi bisogno di evitare conflitti. Lo scambio di messaggi è anche più facile da implementare, rispetto alla memoria condivisa, in un sistema distribuito.

La memoria condivisa può essere più veloce dello scambio di messaggi, che è solitamente implementato tramite chiamate di sistema che impegnano il kernel; la memoria condivisa, invece, richiede l'intervento del kernel solo per allocare le regioni di memoria condivisa, dopo di che tutti gli accessi sono gestiti alla stregua di ordinari accessi in memoria che non richiedono l'assistenza del kernel.

## IPC in sistemi a memoria condivisa

La comunicazione tra processi basata sulla condivisione della memoria richiede che i processi comunicanti allochino una zona di memoria condivisa, di solito residente nello spazio degli indirizzi del processo che la alloca: gli altri processi che desiderano usarla per comunicare dovranno annetterla al loro spazio degli indirizzi. Si ricordi che, normalmente, il sistema operativo tenta di impedire a un processo l'accesso alla memoria di altri processi. La condivisione della memoria richiede che due o più processi raggiungano un accordo per superare questo limite, in modo da poter comunicare tramite scritture e letture dell'area condivisa. Il tipo e la collocazione dei dati sono determinati dai processi, e rimangono al di fuori del controllo del sistema operativo. I processi hanno anche la responsabilità di non scrivere nella stessa locazione simultaneamente.

Per illustrare il concetto di cooperazione tra processi consideri il problema del **Produttore/Consumatore**; tale problema è un **comune paradigma per processi cooperanti**.

*Un processo Produttore produce informazioni che sono consumate da un processo Consumatore.*

Il problema del produttore/consumatore è anche un'utile metafora del paradigma client/server. Si pensa in genere al server come produttore e al client come consumatore.

Una possibile soluzione del problema del produttore/consumatore si basa sulla memoria condivisa. *L'esecuzione concorrente dei due processi richiede la presenza di un buffer che possa essere riempito dal produttore e svuotato dal consumatore. Il buffer dovrà risiedere in una zona di memoria condivisa dai due processi.* Il produttore potrà allora produrre un'unità mentre il consumatore ne consuma un'altra. I due processi devono essere sincronizzati in modo tale che il consumatore non tenti di consumare un'unità non ancora prodotta.

Si possono usare due tipi di buffer:

- **Buffer illimitato:** non pone limiti pratici alla dimensione del buffer -> il consumatore può dover attendere nuovi oggetti, ma il produttore ne può sempre produrre
- **Buffer limitato:** il buffer ha una dimensione fissa -> il consumatore deve attendere se il buffer è vuoto; il produttore deve attendere se il buffer è pieno

Buffer Limitato:

```

item next_produced;

while (true) {
    /* produce an item in next produced */
    while ((in + 1) % BUFFER_SIZE == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

```

Soluzione corretta ma che può trattare solo BUFFER\_SIZE-1 elementi (così i puntatori non si sovrappongono).

Puntatore in: posizione da cui prelevare il prossimo elemento.

Puntatore out: posizione in cui mettere il prossimo elemento.

Buffer circolare: il puntatore in si sovrappone ad out quando il buffer è pieno. Se non voglio sovrapporre i due puntatori quando il buffer è pieno mi fermo alla nona posizione del buffer.

Producer:

```
item next_produced;

while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Il loop è detto spin-loop; finché l'altro processo non cambia out (variabile condivisa), il ciclo continua a girare. In questo caso mi fermo alla nona posizione finché il consumatore non ha finito di elaborare i dati.

Consumer:

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

Caso duale: lo spin-loop continua a girare finché in non viene modificata.

Gli spin-loop hanno senso solo se uso delle variabili condivise.

Devo fare delle operazioni di sincronizzazione per aspettare che l'altro processo finisca la sua operazione prima di operare.

Notare che Producer non modifica out e Consumer non modifica in: questa è una soluzione per fare in modo che i due processi non si pestino i piedi a vicenda.

Quando ci sono variabili accedute in memoria condivisa, bisogna sempre prendere delle precauzioni come questa.

Pensa che l'operazione di decremento e incremento non sono fatte con lo stesso comando in linguaggio macchina, l'operazione che viene fatta per ultima prevale e i dati vengono persi.

## IPC in sistemi a scambio di messaggi

Un altro modo in cui il sistema operativo può far comunicare due processi consente nel fornire ai processi appositi strumenti per lo scambio di messaggi.

Lo scambio di messaggi è un *meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio di indirizzi*. E' una tecnica particolarmente usata nei sistemi distribuiti, dove i processi possono risiedere su macchine diverse connesse da una rete.

Un meccanismo per lo scambio di messaggi deve prevedere almeno due operazioni:

- send(message)
- receive(message)

I messaggi possono avere lunghezza fissa o variabile.

Nel primo caso, l'implementazione a livello di sistema è elementare, ma programmare applicazioni diviene più complicato. Nel secondo caso, l'implementazione del meccanismo a livello di sistema è più complessa, mentre la programmazione utente risulta semplificata.

Se i processi P e Q vogliono comunicare, devono inviare e ricevere messaggi tra loro; deve, dunque, esistere un *canale di comunicazione*.

Ci sono diversi metodi per realizzare a livello fisico un canale di comunicazione e le operazioni send() e receive():

- memoria condivisa
- bus hardware
- network

Ci sono diversi metodi per realizzare a livello logico un canale di comunicazione e le operazioni send() e receive():

- comunicazione diretta o indiretta
- comunicazione sincrona o asincrona
- gestione automatica o esplicita del buffer

## Naming

Con la **comunicazione diretta** ogni processo che intenda comunicare deve nominare esplicitamente il ricevente o il trasmittente della comunicazione.

In questo schema, le funzioni primitive send() e receive() si definiscono come segue:

- send(P, messaggio), invia il messaggio al processo P
- receive(Q, messaggio), riceve il messaggio al processo Q

All'interno di questo schema, un canale di comunicazione ha le seguenti caratteristiche:

- tra ogni coppia di processi che intendono comunicare si stabilisce automaticamente un canale; i processi devono conoscere solo la reciproca identità
- un canale è associato esattamente a due processi
- esiste esattamente un canale tra ciascuna coppia di processi

Questo schema ha una *simmetria all'indirizzamento*, vale a dire che per poter comunicare, il trasmittente e il ricevente devono nominarsi a vicenda. Una variante di questo schema si avvale dell'*asimmetria all'indirizzamento*: soltanto il trasmittente nomina il ricevente, mentre il ricevente non ha bisogno di nominare il trasmittente.

In questo schema le primitive send() e receive() si definiscono come segue:

- send(P, messaggio), invio messaggio al processo P
- receive(id, messaggio), riceve un messaggio da qualsiasi processo; nella variabile id si ottiene il nome del processo con cui è avvenuta la comunicazione

Entrambi gli schemi, simmetrico e asimmetrico, hanno lo svantaggio di una limitata modularità delle restanti definizioni dei processi. La modifica del nome di un processo può infatti implicare la necessità di un riesame di tutte le altre definizioni dei processi, individuando tutti i riferimenti al vecchio nome allo scopo di sostituirli con il nuovo. In generale, tali cablature di informazioni nel codice sono meno vantaggiose di soluzioni indirette.

Con la **comunicazione indiretta** i messaggi s'inviano a delle porte o *mailbox*, che li ricevono. Una *mailbox* si può considerare in modo astratto come un oggetto *in cui i processi possono introdurre e prelevare i messaggi*, ed è identificata in modo univoco.

In questo schema un processo può comunicare con altri processi tramite un certo numero di mailbox e *due processi possono comunicare solo se condividono una mailbox*.

Le primitive send() e receive() si definiscono come segue:

- send(A, messaggio), invia messaggio alla mailbox A
- receive(A, messaggio), riceve un messaggio dalla mailbox A

In questo schema un canale di comunicazione ha le seguenti caratteristiche:

- tra una coppia di processi si stabilisce un canale solo se entrambi i processi della coppia condividono una stessa mailbox
- un canale può essere associato a più di due processi
- tra ogni coppia di processi comunicanti possono esserci più canali condivisi, ciascuno corrispondente ad una mailbox

A questo punto, si supponga che i processi P1, P2 e P3 condividano la mailbox A. Il processo P1 invia un messaggio ad A, mentre sia P2 e P3 eseguono una receive() da A. Sorge il problema di sapere quale processo riceverà il messaggio. La soluzione dipende dallo schema prescelto:

- si fa in modo che un canale sia associato al massimo a due processi
- si consente l'esecuzione di un'operazione receive() a un solo processo alla volta
- si consente al sistema di decidere arbitrariamente quale processo riceverà il messaggio. Il sistema può anche definire un algoritmo per selezionare quale processo riceverà il messaggio (specificando per esempio uno schema, detto *round robin*, secondo il quale i processi ricevono i messaggi a turno) e può comunicare l'identità del ricevente al trasmittente

## Sincronizzazione

Lo scambio di messaggi può essere sincrono (o bloccante) oppure asincrono (non bloccante).

- *Invio sincrono*: il processo che invia il messaggio si blocca nell'attesa che il processo ricevente, o la mailbox, riceva il messaggio
- *Invio asincrono*: il processo invia il messaggio e riprende la propria esecuzione
- *Ricezione sincrona*: il ricevente si blocca nell'attesa dell'arrivo di un messaggio
- *Ricezione asincrona*: il ricevente riceve il messaggio valido o un valore nullo

E' possibile anche avere diverse combinazioni di send() e receive(). Se le primitive sono entrambe bloccanti si parla di *rendezvous* tra mittente e ricevente.

E' banale dare soluzione al problema del produttore e del consumatore tramite le primitive bloccanti send() e receive(). Il produttore, infatti, si limita a invocare send() e attendere finché il messaggio sia giunto a destinazione; analogamente, il consumatore chiama receive(), bloccandosi fino all'arrivo di un messaggio.

```
message next_produced;

while (true) {
    /* produce an item in next_produced */

    send(next_produced);
}

message next_consumed;

while (true) {
    receive(next_consumed)

    /* consume the item in next_consumed */
}
```

L'implementazione è semplice e compatta, ma meno performante del codice visto in precedenza.

## Code di messaggi (buffering)

Se la comunicazione è diretta o indiretta, i messaggi scambiati tra processi comunicanti risiedono in code temporanee. Esistono 3 modi per realizzare queste code:

- **Capacità zero:** la coda ha lunghezza massima 0, quindi il canale non può avere messaggi in attesa al suo interno. In questo caso il trasmittente deve fermarsi finché il ricevente prende in consegna il messaggio (deve aspettare il rendezvous)
- **Capacità limitata:** la coda ha lunghezza finita n, quindi al suo interno possono risidere al massimo n messaggi. Se la coda non è piena, quando s'invia un nuovo messaggio, quest'ultimo è posto in fondo alla coda (il messaggio viene copiato oppure si tiene un puntatore a quel messaggio). Il trasmittente può proseguire la propria esecuzione senza essere costretto ad attendere. Il canale ha tuttavia una capacità limitata; se è pieno, il trasmittente deve fermarsi nell'attesa che ci sia spazio
- **Capacità illimitata:** prettamente teorico; la coda ha lunghezza potenzialmente infinita, quindi al suo interno può attendere un numero indefinito di messaggi. Il trasmittente non si ferma mai.

Il caso di capacità zero è chiamato *sistema a scambio di messaggi senza buffering*; gli altri due, *sistemi con buffering automatico*.

## Cos'è un processo?

Un processo è un programma in esecuzione. Lo stato dell'attività corrente di un processo è rappresentato dal valore del contatore di programma e dal contenuto dei registri del processore. La struttura di un processo in memoria è generalmente suddivisa in più sezioni:

- Sezione di testo: contenente il codice eseguibile.
- Sezione dati: contiene le variabili globali.
- Heap: memoria allocata dinamicamente durante l'esecuzione del programma.
- Stack: memoria temporaneamente utilizzata durante le chiamate di funzioni. Le dimensioni delle sezioni di testo e dati sono fisse, mentre le sezioni stack e heap possono ridursi e crescere dinamicamente durante l'esecuzione. Sottolineiamo che un programma è un'entità passiva, come il contenuto di un file memorizzato su un disco, mentre un processo è un'entità attiva, con un contatore di programma che specifica quale è l'istruzione successiva da eseguire. Un processo durante l'esecuzione è soggetto a cambiamenti di stato. Ogni processo può trovarsi nei seguenti stati:
  - Nuovo: si crea il processo.
  - Esecuzione: Vengono eseguite le istruzioni del relativo programma.
  - Attesa: Il processo attende che si verifichi qualche evento (tipo un'operazione di I/O).
  - Pronto: Il processo attende di essere assegnato ad una unità di elaborazione.
  - Terminato: Il processo ha terminato l'esecuzione.

Un processo è rappresentato nel sistema operativo da un blocco di controllo di un processo(PCB, task control block,). Un blocco di controllo di un processo contiene molte informazioni: - Stato del processo.( vedi sopra 3.1.2) - Contatore di programma: contiene l'indirizzo della successiva istruzione da eseguire. - Registri della CPU . i registri variano il numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, registro indice, puntatori alla cima dello stack, registri d'uso generale e registri contenenti i codici di condizione. - Informazioni sullo scheduling di CPU: comprendono la priorità sui processi. - Gestione della memoria: informazioni che si esprimono attraverso i registri base e limite, tabelle delle pagine o tabelle dei segmenti. - Informazioni di accounting. Queste informazioni comprendono la quota di uso della CPU e il tempo di utilizzo della stessa, i limiti di tempo, i numeri dei processi e così via. - Informazioni sullo stato dell'input dell' output. Queste informazioni comprendono la lista dei dispositivi di input o output assegnato un determinato processo, l'elenco dei file aperti.

## Cos'è un sotto-processo?

Durante la propria esecuzione, un processo può creare numerosi nuovi processi. Il processo creatore si chiama processo genitore, il processo creato si chiama processo figlio. Ciascuno di questi nuovi processi può creare a sua volta altri nuovi processi formando un albero di processi. La maggior parte di un sistema operativo identifica un processo per mezzo di un identificatore del processo (pid). In generale, per eseguire il proprio compito, un processo necessita di alcune risorse (tempo di elaborazione, memoria, file, dispositivo di I/O). Quando un processo crea un sotto-processo, quest'ultimo può essere in grado di ottenere le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato a un sottoinsieme delle risorse del processo genitore. Il processo genitore può avere la necessità di spartire le proprie risorse tra i suoi processi figli, oppure può essere in grado di condividerne alcune, come la memoria o i file.

Quando un processo ne crea uno nuovo, per quel che riguarda l'esecuzione ci sono due possibilità:

1. Il processo genitore continua l'esecuzione in modo concorrente con i propri processi figli;
2. Il processo genitore attende che alcuni o tutti i suoi processi figli terminano.

Ci sono due possibilità, anche per quel che riguarda lo spazio di indirizzi del nuovo processo:

1. Il processo figlio è un duplicato del processo genitore;
2. Nel processo figlio si carica un nuovo programma. In alcuni sistemi quando un processo genitore termina terminano anche i processi figli ottenendo così un effetto a cascata mentre in altri sistemi sia un processo genitore termina i processi figli vengono affidati al processo init, cioè alla radice dei processi

## Cosa fa una syscall fork()?

Un processo si crea per mezzo della chiamata di sistema fork().

## Scheduler di un sistema operativo Batch?

In questa tipologia di sistema accade che si sottopongono più processi di quanti se ne possono eseguire. questi lavori si trasferiscono in dispositivi di memoria secondari i si tengono fino al momento dell'esecuzione. Si hanno due tipi di scheduler:

- Scheduler a lungo termine: sceglie i lavori da questo insieme e li carica in memoria, affinché siano eseguiti.
- Scheduler a breve termine: fa la selezione tra i lavori pronti per l'esecuzione e assegna la CPU ad uno di loro.

Questi due scheduler si differenziano principalmente per la frequenza con cui sono eseguiti. Lo scheduler a breve termine seleziona frequentemente un nuovo processo per la CPU e il processo può essere in esecuzione solo per pochi millisecondi, lo scheduler a lungo termine invece, si esegue con una frequenza molto inferiore; diversi minuti possono trascorrere tra la creazione di un nuovo processo ed il successivo. Lo scheduler a lungo termine controlla il grado di multiprogrammazione, cioè il numero di processi presenti in memoria. Le prestazioni migliori sono date da una combinazione equilibrata di processi con prevalenza di I/O e processi con prevalenza d'elaborazione. In alcuni sistemi operativi, si può introdurre un livello di scheduling intermedio chiamato scheduler a medio termine. L'idea alla base di un tale scheduler è che a volte può essere vantaggioso eliminare processi dalla memoria riducendo il grado di multiprogrammazione del sistema. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta (AVVICENDAMENTO DEI PROCESSI IN MEMORIA, swapping). Il processo viene rimosso e successivamente caricato in memoria dallo scheduler a medio termine. L'avvicendamento dei processi in memoria può servire a migliorare la combinazione di processi, oppure a liberare una parte della memoria se un cambiamento dei requisiti di memoria ha impegnato eccessivamente la memoria disponibile.

## Comunicazione tra processi

I processi eseguiti in modo concorrente nel sistema operativo possono essere indipendenti o cooperanti. Per lo scambio di messaggi e informazioni i processi cooperanti necessitano di un meccanismo di comunicazione tra processi (IPC). I modelli fondamentali della comunicazione tra processi sono: a memoria condivisa e a scambio di messaggi. Nel modello memoria condivisa si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona appunto nel secondo modello la comunicazione al luogo tramite uno scambio di messaggi tra processi cooperanti. La memoria condivisa massimizza l'efficienza della comunicazione, ed è più veloce dello scambio di messaggi che è solitamente implementato tramite chiamate di sistema che impegnano il kernel.

# Capitolo 4 - Sistemi Operativi

## Thread e concorrenza

### Introduzione

Un **thread** è l'**unità di base d'uso della CPU** e comprende un identificatore di thread (ID), un contatore di programma, un insieme di registri e una pila (stack).

Possiamo definire un thread anche come *l'unità più piccola schedulabile all'interno della CPU*.

Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali.

Un processo tradizionale, chiamato anche *processo pesante*, è composto da un solo thread.

Un *processo multithread* è *in grado di svolgere più compiti in modo concorrente*.

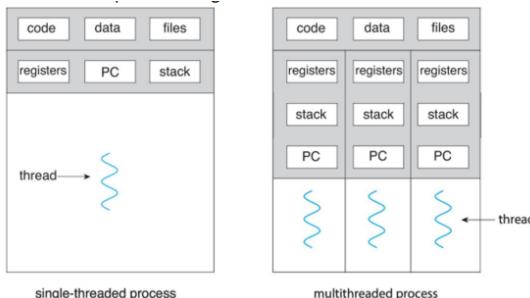


Immagine che mostra la differenza tra un processo single threader e multi-threader.

La creazione di un thread dev'essere più leggera rispetto alla creazione di un processo.

Ho sempre l'opzione di creare semplicemente più processi e con un processore multicore avrei comunque un'esecuzione multipla, ma ognuno di loro avrà uno spazio indirizzi dedicato che renderà più complicata la commutazione tra di essi: c'è molto overhead.

I thread *condividono lo stesso spazio indirizzi del loro processo*: questo rende la loro commutazione leggera, ovvero i tempi sono anche centomila volte più piccoli.

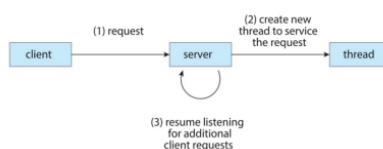
Percorsi multipli: ognuno chiama le proprie istruzioni, ogni thread ha uno stack personale diverso da quello degli altri. I thread, però, condividono l'area dati globale. In questo modo sono schedulabili individualmente (se ho una CPU multicore).

Il codice è unico ma ognuno fa una cosa diversa: iniziano la loro esecuzione quando il main thread li genera. Condividere il codice non vuol dire che fanno tutti la stessa cosa.

La maggior parte delle applicazioni per i moderni computer è multi-thread. Di solito, un'applicazione si codifica come un processo a sé stante comprendente più thread di controllo.

Esempio di applicazioni multi-thread: un web browser può avere un thread per rappresentare sullo schermo immagini e testo e un altro thread per scaricare i dati dalla rete.

Nel caso del server web, se il processo è multi-thread, il server creerà un thread distinto per ricevere le richieste dei client; in presenza di una richiesta, anziché creare un altro processo, si creerebbe un altro thread per soddisfarla:



### Vantaggi

I vantaggi della programmazione multi-thread si possono classificare in quattro categorie principali:

1. **Tempo di risposta**: rendere multi-thread un'applicazione interattiva può permettere a un programma di continuare la sua esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo di risposta all'utente. Questa caratteristica è particolarmente utile nella progettazione di interfacce utente.
2. **Condivisione delle risorse**: i processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa e lo scambio di messaggi. Queste tecniche devono essere esplicitamente messe in atto dal programmatore. Tuttavia, i thread condividono di default la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice e dei dati consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
3. **Economia**: assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestire i cambi di contesto. E' difficile misurare empiricamente la differenza nell'overhead richiesto per creare e gestire un processo invece che un thread, tuttavia la creazione dei thread richiede in generale meno tempo e meno memoria, e il cambio di contesto tra thread è più rapido.

4. **Scalabilità:** i vantaggi della programmazione multi-thread sono ancora maggiori nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo su distinti core di elaborazione. Invece un processo con un singolo thread può funzionare solo su un processore, indipendentemente da quanti ve ne siano a disposizione. Il multithreading su una macchina con più processori incrementa il parallelismo.

## Programmazione multicore

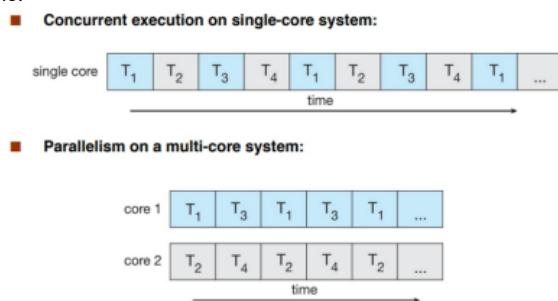
Ricorso all'architettura multicore: ho più core sullo stesso chip, ognuno dei quali viene visto dal sistema operativo come un processore separato. Le prestazioni aumentano, ma è il programmatore a dover sfruttare il parallelismo, programmando effettivamente in parallelo.

Si consideri un'applicazione con quattro thread. In un sistema con singolo core, esecuzione concorrente significa solo che l'esecuzione dei thread è avviceduta nel tempo (interleaved), perché la CPU è in grado di eseguire un solo thread alla volta. Su un sistema multicore, invece, esecuzione concorrente significa che i thread possono funzionare in parallelo, dal momento che il sistema può assegnare thread diversi a ciascun core.

Differenza tra concorrenza e parallelismo:

- **Sistema Concorrente:** supporta più task permettendo a ciascuno di progredire nell'esecuzione
- **Sistema Parallello:** può eseguire simultaneamente più di un task

La concorrenza non implica il parallelismo.



## Le sfide della programmazione

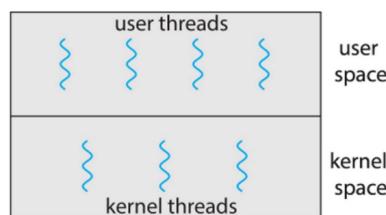
Ecco le cinque grandi aree da tenere in considerazione quando si vogliono sviluppare applicazioni multithreaded:

1. **Identificazione dei task:** analizzo l'applicazione per individuare le aree di codice separabili in task distinte, eseguibili separatamente in parallelo su core distinti.
2. **Bilanciamento:** i task individuati devono eseguire una mole di lavoro comparabile. Se un task non contribuisce al processo tanto quanto gli altri, non vale la pena eseguirlo su un core separato.
3. **Suddivisione dei dati:** proprio come le applicazioni sono divise in task separati, i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati su core distinti.
4. **Dipendenze dei dati:** bisogna esaminare i dati per individuare le dipendenze tra due o più task. Se un task dipende dai dati di un altro, i programmati devono assicurare che l'esecuzione dei task sia sincronizzata in modo da soddisfare queste dipendenze.
5. **Test e debugging:** essendoci più flussi di esecuzione possibili, effettuare i test e il debugging sui programmi di questo tipo è complicato rispetto a semplici programmi single threaded.

## Thread a livello utente vs thread a livello kernel

I thread a livello utente sono gestiti sopra il livello del kernel e senza il suo supporto.

I thread a livello kernel sono gestiti direttamente dal sistema operativo.



Come posso generare questi thread?

Ho delle librerie che schedulano separatamente i thread all'interno del processore. Possono sospendersi spontaneamente ed essere schedulati in maniera timesharing da uno scheduler interno al processo: user thread.

Il sistema operativo non è a conoscenza dei thread, manda semplicemente in esecuzione il processo, non interessandosi di ciò che succede al suo interno.

Pro: non pago per ogni thread eseguito dato che non deve fare syscall aggiuntive, non scomodando così il sistema operativo.

Contro: se il processo viene sospeso (es. operazioni di I/O) non ha la concorrenza perché anche i thread sono sospesi. Il problema venne inizialmente attutito facendo delle chiamate I/O asincrone. Inoltre, non posso sfruttare i core multipli; tutto ciò sempre perché il sistema operativo vede un solo processo.

I sistemi operativi moderni permettono di gestire i thread attraverso delle chiamate al sistema operativo: kernel thread.

E' più lento ma non pago per ogni I/O e posso sfruttare i multicore.

Windows ha i thread, Linux e Android non esattamente (vedremo dopo).

Differenza tra API (primitive per chiamare i thread) e la loro implementazione:

solo vedendo le primitive non posso distinguere se i thread vengono implementati nello spazio kernel o nello spazio utente.

Standard POSIX: potrebbe essere fatto in una maniera o nell'altra (dipende dalla JVM). E' fatto così per sfruttare effettivamente i multicore quando sono disponibili.

Java thread: tranne nei sistemi embedded singlecore sono gestiti a livello kernel. Questo perché la JVM viene quasi sempre eseguita su un sistema operativo che la ospita. Perciò, i thread di Java su Windows sono in effetti implementati mediante la API di Windows. Non c'è comunque garanzia.

Il kernel del sistema, operando, deve eseguire delle attività concorrenti mentre è in attività. Devo ideare un sistema per fare eseguire più cose contemporaneamente. Metodo tradizionale: accodo le attività.

Ma perché non organizzare anche il kernel a thread? Si può fare! Anche il sistema operativo può essere multithread.

## Librerie dei thread

Una libreria dei thread fornisce al programmatore una API per la creazione e la gestione dei thread.

Metodi in cui implementare una libreria dei thread:

1. A livello utente: la libreria è collocata interamente a livello utente, senza fare ricorso al kernel. Il codice e le strutture dati per la libreria risiedono tutti nello spazio utente. Questo implica che invocare una funzione della libreria si traduce in una chiamata locale a una funzione nello spazio utente e non in una chiamata di sistema
2. A livello kernel: consiste nell'implementare una libreria a livello kernel, supportata direttamente dal sistema operativo. In questo caso, il codice e le strutture dati per la libreria si trovano nello spazio del kernel. Invocare una funzione dell'API per la libreria provoca, generalmente, una chiamata di sistema al kernel

Introduciamo due strategie generali per la creazione di più thread: il threading sincrono e il threading asincrono.

Nel threading asincrono, una volta che un genitore crea un thread figlio, riprende la sua esecuzione, in modo che genitore e figlio restino in esecuzione concorrentemente. Ogni thread viene eseguito in modo indipendente rispetto agli altri thread e il thread genitore non ha bisogno di conoscere quando suo figlio termina. Poiché i thread sono indipendenti, vi è di solito poca condivisione dei dati.

Il threading sincrono si verifica quando il thread genitore crea uno o più figli e attende che tutti terminino prima di riprendere l'esecuzione. Tale strategia è detta *fork-join*. In questo caso, i thread creati dal genitore svolgono il lavoro in maniera concorrente, ma il genitore non può continuare fino al completamento di questo lavoro. Una volta che un thread completa il suo lavoro esso termina e si unisce con il genitore. Solo dopo che tutti i figli si sono uniti al genitore, questo può riprendere l'esecuzione. In genere, il threading sincrono comporta una significativa condivisione dei dati tra i thread. Per esempio, il thread genitore può combinare i risultati calcolati dai suoi figli.

## Pthreads (è una specifica, non un'implementazione)

Ci si riferisce allo *standard POSIX che definisce una API per la creazione e la sincronizzazione dei thread*. Non si tratta di una implementazione, ma di una *specificità del comportamento dei thread*; i progettisti di sistemi operativi possono realizzare la specifica come meglio credono. Sono molti i sistemi che implementano le specifiche Pthreads; per la maggior parte si tratta di sistemi di tipo UNIX, tra cui Linux e macOS.

Anche se Windows non supporta nativamente Pthreads, sono disponibili alcune implementazioni di terze parti per Windows.

Il seguente programma C multithread semplifica la API Pthreads tramite il calcolo di una sommatoria eseguito da un thread apposito:

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

In grassetto ho le funzioni per settare gli attributi del thread (`pthread_attr_init`), creare il thread (`pthread_create`), attendere la fine del thread (`pthread_join`).

Dato che non definisco gli attributi, vengono utilizzati quelli di default. Oltre al thread identifier e gli attributi, passo al thread la funzione che dovrà eseguire (`runner`) e il numero intero fornito come intero dalla linea di comando (`argv[1]`).

Avrà quindi il thread padre (`main`) e un thread figlio che esegue `runner`. Quando il figlio termina, il `main` condivide il valore condiviso `sum`. Questo è un esempio di thread sincrono.

La funzione `pthread_exit` termina il run del thread.

## Thread in Windows

Le API di Windows sono simili ai Pthreads, ma hanno nomi diversi delle funzioni.

## Thread Java

Tutti i programmi in Java incorporano almeno un thread di controllo. Anche solo un programma con un singolo main è eseguito come un thread. I thread Java sono disponibili su tutti i sistemi che dispongono di una JVM.

Ci sono due tecniche per la generazione di thread:

1. creo una nuova classe che eredita Thread e faccio l'override del metodo run()
2. definisco una classe che implementa l'interfaccia Runnable, la quale definisce un metodo run() astratto

La creazione di un thread in Java richiede di creare un oggetto Thread e passare all'oggetto un'istanza di una classe che implementa Runnable, per poi invocare il metodo start() dell'oggetto Thread.

```
Thread worker = new Thread(new Task())
worker.start()
```

Il metodo start() ha il duplice effetto di allocare la memoria e inizializzare un nuovo metodo nella JVM e chiamare il metodo run. Il metodo run non viene chiamato in maniera diretta. Ho anche il metodo join() se voglio fare l'unione tra thread (lancia però l'eccezione InterruptedException()).

Nelle nuove versioni di Java sono disponibili nuovi strumenti grazie al package java.util.concurrent.

Interfaccia Executor: si basa sul modello produttore-consumatore. Separo la creazione del thread dalla loro esecuzione.

Interfaccia Callable: simile a Runnable, ma è possibile restituire i risultati.

E' possibile usare il metodo get() dell'interfaccia per recuperare i dati di un thread (in Java non esistono i dati globali, essendo orientato agli oggetti).

## Threading implicito

La correttezza del programma può essere a rischio con thread esplicativi. Creazione implicita lasciata ai compilatori e alle librerie run-time. Il sistema più comunemente utilizzato è OpenMP. Standard industriale che permette di scrivere dei programmi che automaticamente attivano dei thread per fare delle attività. Verranno generati tanti thread quanti sono i core disponibili e ogni iterazione del ciclo verrà gestita da un core diverso.

Intel, per supportare processori nuovi, ha creato un sistema di threading implicito (Intel Threading Building Blocks).

Un modo per gestire al meglio la progettazione di applicazioni parallele e concorrenti è il trasferimento della creazione e della gestione del threading dagli sviluppatori di applicazioni ai compilatori e alle librerie di routine.

Questa strategia si chiama threading implicito ed è diventata, oggi, molto comune. Essa richiede agli sviluppatori di applicazioni di identificare task, e non thread, che possano essere eseguiti in parallelo.

Un task viene solitamente codificato come una funzione, che viene mappata dalla libreria di runtime su un thread separato. Il vantaggio di questo approccio è che gli sviluppatori devono solo individuare i task parallelizzabili, mentre le librerie determinano i dettagli specifici della creazione e della gestione dei thread.

## Problematiche di programmazione multithread

### Chiamate di sistema fork() ed exec()

La syscall fork() viene utilizzata per la creazione di un nuovo processo tramite la duplicazione di un processo esistente.

In un programma multithread la semantica delle chiamate di sistema fork() ed exec() cambia.

Se un thread in un programma invoca la chiamata di sistema fork(), il nuovo processo potrebbe, in generale, contenere un duplicato di tutti i thread oppure del solo thread invocante. Alcuni sistemi UNIX includono entrambe le versioni.

La syscall exec() funziona così: se un thread la invoca, il programma specificato come parametro della exec() sostituisce l'intero processo, inclusi tutti i thread.

L'uso delle due versioni della fork() dipende dall'applicazione. Se s'invoca la exec() immediatamente dopo la fork(), la duplicazione del thread non è necessaria, poiché il programma specificato nei parametri della exec() sostituirà il processo. In questo caso conviene duplicare il solo thread chiamante. Tuttavia, se la exec() non segue immediatamente la fork(), il nuovo processo dovrebbe duplicare tutti i thread del processo genitore.

### Gestione dei segnali

Nei sistemi UNIX si usano i segnali per *comunicare ai processi il verificarsi di determinati eventi*. Un segnale si può ricevere in modo sincrono o asincrono, secondo la sorgente e la ragione della segnalazione dell'evento. Indipendentemente dal modo di ricezione sincrono o asincrono, tutti i segnali seguono lo stesso schema

- all'occorrenza di un particolare evento si genera un segnale
- s'invia il segnale ad un processo
- una volta ricevuto, il segnale dev'essere gestito

Un accesso illegale alla memoria o una divisione per zero generano segnali sincroni. In questi casi, se un programma in esecuzione compie le suddette azioni, viene generato un segnale. I segnali sincroni s'inviano allo stesso processo che ha eseguito l'operazione causa del segnale (questo è il motivo per cui sono considerati sincroni).

Quando un segnale è causato da un evento esterno al processo in esecuzione, tale processo riceve il segnale in modo asincrono. Esempi di segnali di questo tipo sono la terminazione di un processo richiesta con specifiche combinazioni di tasti (come control c) oppure la scadenza di un timer. Di solito un segnale asincrono si invia ad un altro processo.

Ogni segnale si può gestire in due modi:

1. tramite un gestore predefinito di segnali
2. tramite un gestore di segnali definito dall'utente

Per i processi a singolo thread la gestione dei segnali è semplice: i segnali vengono sempre inviati al processo. Per i processi multi-thread si pone il problema del thread cui si deve inviare il segnale. In generale esistono le seguenti possibilità:

1. inviare il segnale al thread cui il segnale si riferisce
2. inviare il segnale a ogni thread del processo
3. inviare il segnale a specifici thread del processo
4. definire un thread specifico per ricevere tutti i segnali diretti al processo

Segnali sincroni: invio il segnale al thread che ha generato l'evento.

Segnali asincroni: invio il segnale, generalmente, a tutti i thread.

## Cancellazione dei thread

E' possibile terminare un thread prima che finisca la sua esecuzione. Esempio: eseguo una ricerca con più thread. Una volta che uno di essi ha trovato il risultato, elimino gli altri.

Comune nei browser: più thread si occupano del caricamento delle risorse di una pagina web. Quando l'utente preme il pulsante di terminazione, tutti i thread che stanno caricando la pagina verranno cancellati.

Il thread da cancellare è detto *thread target*.

La cancellazione può avvenire in due modi diversi:

1. *Cancellazione asincrona*: un thread fa immediatamente terminare il thread target
2. *Cancellazione differita*: il thread target controlla periodicamente se deve terminare, in modo da effettuare la terminazione in maniera ordinata

Difficoltà: se un thread ha una risorsa assegnata o sta aggiornando dei dati che condivide con altri thread.

Il sistema operativo spesso non si riappropria di tutte le risorse di un thread cancellato.

Meglio utilizzare la cancellazione differita: avviene solamente quando il thread target verifica se può essere cancellato o meno; con la cancellazione asincrona non ho garanzie.

L'effettiva cancellazione del thread dipende da come il thread destinazione è impostato per la gestione della richiesta.

Pthreads supporta 3 modalità di cancellazione, ognuna definita da un tipo e uno stato:

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Se la cancellazione è disabilitata, le richieste di cancellazione rimangono in sospeso finché il thread non le riabilita.

Cancellazione differita: cancellazione di default, la cancellazione avviene solo quando un thread raggiunge un punto di cancellazione (cancellation point).

Una tecnica per la creazione del cancellation point è l'invocazione della funzione `pthread_testcancel()`.

In caso di richiesta di cancellazione in attesa, viene chiamata una funzione nota come gestore della pulizia (cleanup handler), che permette di rilasciare tutte le risorse che un thread può aver acquisito prima di eliminarlo.

Nei sistemi Linux la cancellazione di un thread è gestita tramite segnali. La cancellazione dei thread in Java utilizza una tecnica simile alla cancellazione in differita dei Pthreads: si invoca il metodo `interrupt()`, che imposta lo stato di interruzione di un thread a true.

Un thread può controllare il suo stato di interruzione invocando il metodo `isInterrupted()`.

## Thread Windows

I thread nativi esistono. Ogni processo può contenere più thread.

I componenti generali di un thread includono:

- un identificatore (ID) di thread, che identifica univocamente il thread
- un insieme di registri che rappresenta lo stato del processore
- un contatore di programma
- uno stack utente, usato quando il thread è eseguito in modalità utente, e uno stack kernel, usato quando il thread è eseguito in modalità kernel
- un'area di memoria privata, usata da diverse librerie run-time e dinamiche (DLL)

L'insieme dei registri, gli stack e la memoria privata sono detti *contesto* del thread.

Le strutture principali di un thread includono:

- ETHREAD (Executive Thread Block): appartiene al kernel, contiene il puntatore a KTHREAD, puntatore al processo a cui appartiene il thread e l'indirizzo di funzione in cui il thread inizia l'esecuzione
- KTHREAD (Kernel Thread Block): appartiene al kernel, contiene informazioni relative allo scheduling, stack kernel e un puntatore alla struttura TEB
- TEB (Thread Environment Block): appartiene allo spazio utente, contiene ID, stack utente e memoria locale del thread

## Thread Linux

Linux nasce come sistema tradizionale monolitico che punta ad avere prestazioni elevate: non ha quindi il concetto di thread.

I thread, quindi, non sono nativi, ma somigliano ai processi.

*Non esistono thread, ma solo processi che condividono delle cose -> non previsti i thread a livello kernel.*

Se creo un processo che condivide l'area dati globale, i file aperti e il codice ma non lo stack è quasi come se creassi un fthread.

E se facessi una fork() in cui viene fatta una clonazione dello spazio indirizzi e cambio solo il pid?

La fork() chiama quindi la syscall *clone()* per specificare cosa duplicare e cosa no.

In UNIX la fork() non fa tutto da sola proprio per permettere questa flessibilità.

In Linux ho quindi dei processi detti **task** che selettivamente possono condividere qualcosa. Sia i processi che i thread sono detti task, non c'è distinzione tra i due.

Quando clone() è invocata, riceve come parametro dei flag, al fine di stabilire quante e quali risorse dei task genitore devono essere condivise con il task figlio.

Se non viene impostato nessun flag al momento dell'invocazione di clone non si ha alcuna condivisione, e la funzione ottenuta diventa simile a quella fornita dalla chiamata di sistema fork().

Vengono gestiti dal kernel e le prestazioni ne risentono essendo dei thread non nativi.

Controllo dei flag:

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Per ogni task, nel kernel esiste un'unica struttura dati (struct task\_struct).

Invece di memorizzare i dati del task relativo, questa struttura utilizza dei puntatori ad altre strutture dove i dati sono effettivamente contenuti (es. strutture dati che rappresentano l'elenco dei file aperti).

Quando si invoca fork() si crea un nuovo task insieme con una copia di tutte le strutture dati del task genitore.

Quando si invoca clone(), il nuovo task anziché ricevere una copia di tutte le strutture dati, può puntare alle strutture dati del task genitore, a seconda dell'insieme dei flag passati a clone().

Solaris permette di creare sia dei thread a livello utente che a livello kernel.

I thread non sono protetti in memoria tra di loro, condividendo lo stesso spazio indirizzi. Se scrivo per errore sopra un altro thread non avrò la segnalazione del segmentation error -> il debugging è quindi molto problematico.

I thread possono scrivere e leggere contemporaneamente dall'area condivisa? NO! Non posso avere una lettura e scrittura, ma devo operare secondo primitive di sincronizzazione precise.

## Cos'è un thread? Quali aree dati vengono condivise?

Un thread è l'unità di base della CPU e comprende un identificatore di thread (ID), un contatore di programma, un insieme di registri e una pila (stack). Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali. Un processo tradizionale, chiamato anche processo pesante, è composto da un solo thread. Un processo multi-thread è in grado di lavorare a più task in modo concorrente.

## Cosa sono i thread?

I sistemi operativi tradizionali vedevano i processi come un unico thread di esecuzione. All'interno del processo c'è la possibilità di più esecuzioni temporanee del codice. I thread sono quindi l'unità minore schedulabile.

Altra risposta: (+ altra domanda: quali aree vengono condivise?)

Un thread è l'unità di base di uso della CPU e comprende un identificatore di thread (ID), un contatore di programma, un insieme di registri, e una pila (stack). Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali. Un processo tradizionale, chiamato anche processo pesante, è composto da un solo thread. Un processo multi-thread è in grado di lavorare a più compiti in modo concorrente

## Quali sono i vantaggi del thread?

I vantaggi della programmazione multi-thread si possono classificare rispetto a 4 categorie principali:

1. *Tempo di risposta*: rendere multi-thread un'applicazione interattiva può permettere ad un programma di continuare la sua esecuzione anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo di risposta media all'utente
2. *Condivisione delle risorse*: i processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa o il passaggio di messaggi. Tuttavia, i thread condividono di ufficio la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi
3. *Economia*: assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo a cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto

4. **Scalabilità:** i vantaggi della programmazione multi-thread aumentano notevolmente nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo (uno per ciascun processore)

## Differenza tra thread a livello kernel e thread a livello utente. Qual è il più utile?

I thread a livello utente sono gestiti senza l'aiuto del kernel, il vantaggio consiste nel fatto che quando creo un thread o passo da un thread all'altro non ho bisogno dell'intervento del sistema operativo e lo svantaggio è che i thread possono chiamare syscall facendo così intervenire il sistema operativo bloccando tutti i thread del processo. Inoltre, non è possibile sfruttare il parallelismo dato che quando un processo è assegnato ad uno dei processori, tutti i suoi thread sono eseguiti uno alla volta.

I thread a livello kernel, invece, sono gestiti direttamente dal sistema operativo, cioè tutte le operazioni effettuate su di essi avvengono secondo una chiamata di sistema, a differenza del thread a livello utente, se utilizza una syscall il processo non blocca tutti i thread. Deve esistere una relazione tra i thread a livello utente e i thread a livello kernel.

3 tipologie di collegamenti: molti a uno, uno a uno, molti a molti.

Altra risposta:

- Un thread utente è un tipo di thread che viene eseguito in un'applicazione utilizzando una libreria di sistema come Pthreads o Windows API. Questo tipo di thread è gestito dal sistema operativo, ma non ha la stessa priorità di un thread kernel.
- Un thread kernel, al contrario, è un thread che viene eseguito direttamente dal kernel del sistema operativo e ha una priorità più elevata rispetto ai thread utente. Questi thread sono utilizzati per attività di sistema come la gestione della memoria e delle periferiche I/O.

## Thread in Linux

Linux non prevede i thread. Esso offre la chiamata di sistema fork() per duplicare un processo e la syscall clone() per generare un nuovo thread. Tuttavia, Linux non distingue tra processi e thread, impiegando generalmente al loro posto il termine task.

Quando clone() è invocata, riceve come parametro un insieme di flag, al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio.

Altra risposta:

Linux nasce come sistema tradizionale monolitico che punta ad avere prestazioni elevate. Non ha quindi un concetto di thread.

Se faccio un processo che condivide l'area di dati globali i file aperti e il codice ma non lo stack è quasi come se facessi un thread. Se facessi una fork in cui viene fatto una clonazione dello spazio di indirizzi, e cambia solo il pid? La fork chiama quindi la syscall clone() per specificare cosa duplicare e cosa no. In Unix la fork non fa tutto da sola per permettere questa flessibilità. Il Linux ha quindi dei processi detti task che selettivamente possono condividere qualcosa. Sia i processi che i thread sono detti task, non c'è distinzione tra i due. Quando clone() è invocata, riceve come parametri dei flag, al fine di stabilire quante e quali risorse del task genitore devono essere condivise con il task figlio. Se non viene impostato nessun flag al momento dell'invocazione del clone() non si ha alcuna condivisione e la funzione ottenuta diventa simile a quella fornita dalla chiamata di sistema fork(). Vengono gestiti dal kernel e le prestazioni ne risentono essendo dei thread non attivi.

Altra risposta ancora:

Linux non ha thread. Linux offre la chiamata di sistema fork() per duplicare un processo, e prevede inoltre la chiamata di sistema clone() per generare nuovo thread. Tuttavia, Linux non distingue tra processi e thread, impiegando generalmente al loro posto il termine task. Quando clone() è invocata, riceve come parametro un insieme di indicatori (flag), al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio.

## Cos'è un processore multi-thread?

Dà la possibilità di eseguire più thread su un singolo processore o core.

## Differenze tra multicore e multi-thread?

Per multicore si intende un'architettura con più processori.

Multi-thread significa che si possono eseguire più thread di un processo in parallelo su un singolo processore.

## Differenza tra sistema operativo monolitico e multithread

La differenza principale tra i due è la loro architettura e la loro capacità di gestire più richieste contemporaneamente. Un sistema operativo monolitico funziona come un singolo processo che gestisce tutte le richieste, mentre un sistema operativo multi-thread è in grado di gestire più richieste contemporaneamente, utilizzando più fili di esecuzione. Ciò significa che un sistema operativo multi-thread è più efficiente e più veloce di un sistema operativo monolitico, ma può essere più complesso da implementare e mantenere.

# Capitolo 5 - Sistemi Operativi

## Scheduling della CPU

### Concetti fondamentali

In un sistema dotato di un singolo core di elaborazione si può eseguire al massimo un processo alla volta: gli altri processi, se ci sono, devono attendere che la CPU sia libera e possa essere nuovamente sottoposta a scheduling.

L'obiettivo della multiprogrammazione è avere sempre un processo in esecuzione, in modo da massimizzare l'utilizzo della CPU.

Un processo è in esecuzione finché non deve attendere un evento, generalmente il completamento di qualche richiesta di I/O; durante l'attesa, in un sistema di calcolo semplice, la CPU resterebbe inattiva e tutto il tempo d'attesa sarebbe sprecato.

Con la multiprogrammazione si cerca d'impiegare questi tempi d'attesa in modo produttivo: si tengono contemporaneamente più processi in memoria e quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della CPU per cederlo ad un altro processo.

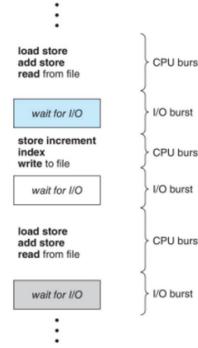
Su un sistema multicore questo concetto di mantenere occupata la CPU è esteso a tutti i core di elaborazione del sistema.

Lo scheduling è una *funzione fondamentale dei sistemi operativi*; si sottpongono a scheduling quasi tutte le risorse di un calcolatore. Naturalmente la CPU è una delle risorse principali e il suo scheduling è alla base della progettazione dei sistemi operativi.

### Ciclicità delle fasi d'elaborazione e di I/O

Il successo dello scheduling della CPU dipende dall'osservazione della seguente proprietà dei processi: l'esecuzione del processo consiste in un *ciclo d'elaborazione* (svolto dalla CPU) e d'attesa del completamento delle operazioni di I/O.

I processi si alternano tra questi due stati.



L'esecuzione di un processo comincia con una *sequenza di elaborazione svolte dalla CPU (CPU burst)*, seguita da una *sequenza di operazioni I/O (I/O burst)*, quindi un'altra sequenza di operazioni della CPU, di nuovo una sequenza di operazioni di I/O e così via.

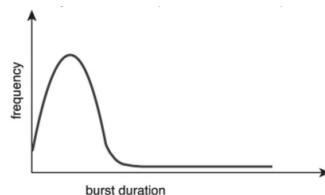
L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione.

La distribuzione e le durate delle fasi di CPU dipendono dal tipo di processo che stiamo esaminando. Un programma con prevalenza di operazioni di CPU è detto *CPU bound*, può produrre operazioni della CPU molto lunghe. Discorso duale per un programma *I/O bound*.

In base alla distribuzione dei due tipi di operazione scelgo l'algoritmo di scheduling più adatto per massimizzare l'utilizzo della CPU.

I burst devono essere abbastanza brevi: inevitabilmente un processo dovrà fare un po' di I/O. Sono quindi molto frequenti.

I burst più frequenti, quindi, sono quelli abbastanza piccoli.



### Scheduler della CPU

Ogniqualvolta la CPU passa nello stato d'inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella ready queue. In particolare, è lo *scheduler a breve termine*, o scheduler della CPU che, tra i processi in memoria pronti per l'esecuzione, sceglie quello a cui assegnare la CPU.

La ready queue non è necessariamente una coda FIFO: si può realizzare come tale, come una coda con priorità, un albero o semplicemente una lista concatenata non ordinata. Tuttavia, concettualmente, tutti i processi della ready queue sono posti nella lista d'attesa per accedere alla CPU. Generalmente, gli elementi delle code sono i PCB (process control block) dei processi.

### Scheduling con e senza prelazione (preemptive e non preemptive)

Lo scheduling con prelazione è il più utilizzato.

Le decisioni riguardanti lo scheduling della CPU vengono prese nelle seguenti quattro circostanze:

1. un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di I/O o invocazione di wait() per la terminazione di uno dei processi figli)
2. un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale d'interruzione)
3. un processo passa dallo stato di attesa allo stato pronto (per esempio, al completamento di un'operazione di I/O)
4. un processo termina

I casi 1 e 4 non danno alternative in termini di scheduling: si deve per forza scegliere un nuovo processo da eseguire, sempre che ce ne siano nella ready queue. In questi casi parliamo di *scheduling senza prelazione (non preemptive)*.

I casi 2 e 3 danno una scelta: lo *scheduling è con prelazione (preemptive)*.

Nello scheduling senza prelazione, quando si assegna la CPU a un processo, questo ne rimane in possesso fino al momento del suo rilascio.

Il problema dello scheduling con prelazione è che può portare ad una race condition quando i dati sono condivisi tra diversi processi. Si consideri il caso in cui due processi condividono dati; mentre uno di questi aggiorna i dati si ha la sua prelazione in favore dell'esecuzione dell'altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato incoerente dal primo processo.

La capacità di prelazione si ripercuote anche sulla progettazione del kernel del sistema operativo. Durante l'elaborazione di una chiamata di sistema, il kernel può essere impegnato in attività per conto di un processo; tali attività possono comportare la necessità di modifiche a importanti dati del kernel, come le code di I/O. Se si ha la prelazione del processo durante tali modifiche e il kernel (o un driver di dispositivo) deve leggere o modificare gli stessi dati, si può avere il caos.

I kernel dei sistemi operativi possono essere progettati con o senza prelazione.

Un kernel senza prelazione attenderà che una chiamata di sistema venga completata o che un processo si blochi in attesa di terminare l'I/O prima di eseguire un cambio di contesto. Questo schema garantisce che la struttura del kernel sia semplice, poiché il kernel non può esercitare la prelazione su un processo mentre le sue strutture dati si trovano in uno stato incoerente.

Questo modello d'esecuzione non è adeguato alle elaborazioni in tempo reale, in cui i task devono essere conclusi entro un intervallo fissato di tempo.

Un kernel con prelazione richiede meccanismi come i lock mutex per prevenire le race condition quando si accede a strutture dati condivise del kernel. La maggior parte dei sistemi operativi moderni è interamente con prelazione durante l'esecuzione in modalità kernel.

Poiché le interruzioni si possono verificare in ogni istante e il kernel non può sempre ignorarle, le sezioni di codice eseguite per effetto delle interruzioni devono essere protette da un uso simultaneo. Il sistema operativo deve ignorare raramente le interruzioni, altrimenti si potrebbero perdere dati in ingresso, o si potrebbero sovrascrivere dati in uscita. Per evitare che più processi accedano in modo concorrente a tali sezioni di codice, queste disattivano le interruzioni al loro inizio e le riattivano alla fine. Le sezioni di codice che disabilitano le interruzioni si verificano tuttavia raramente e, in genere, non contengono molte istruzioni.

## Dispatcher

Scritto in Assembly perché devono maneggiare i registri.

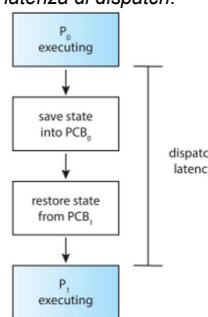
Un dispatcher è una parte di codice che fa lo switch tra due processi.

*Il Dispatcher è un modulo che passa effettivamente il controllo della CPU al processo scelto dallo scheduler a breve termine.*

Questa funzione comprende:

- il cambio di contesto da un processo all'altro
- il passaggio alla modalità utente
- il salto alla giusta posizione del programma utente per riavviare l'esecuzione

Poiché si attiva ad ogni cambio di contesto, il dispatcher dovrebbe essere quanto più rapido possibile. Il *tempo richiesto dal dispatcher per fermare un processo ed avviare l'esecuzione di un altro* è noto come *latenza di dispatch*.



Le parti bianche sono overhead.

Un *cambio di contesto volontario* si verifica quando *un processo cede il controllo della CPU in seguito alla richiesta di una risorsa che al momento non è disponibile* (per esempio, quando è bloccato in attesa di I/O).

Un *cambio di contesto involontario* si verifica quando *la CPU viene sottratta a un processo*, per esempio quando il suo quanto di tempo è scaduto oppure quando è stata esercitata la prelazione da parte di un processo con priorità più alta.

## Criteri di scheduling

Si applicano a processi eseguiti in modalità batch, non per sistemi interattivi come quelli moderni.

Diversi algoritmi di scheduling della CPU hanno priorità differenti e possono favorire una particolare classe di processi.

Di seguito si riportano alcuni criteri che possono incidere sulla scelta dell'algoritmo di scheduling:

- **Utilizzo della CPU:** la CPU dev'essere più attiva possibile. Teoricamente, l'utilizzo della CPU può variare dallo 0 al 100 percento. In un sistema reale dovrebbe variare dal 40 percento, per un sistema con poco carico, al 90 percento, per un sistema con carico elevato.
- **Throughput:** la CPU è attiva quando si svolge del lavoro. Una misura del lavoro svolto è data dal numero di processi completati nell'unità di tempo: il throughput. Per processi di lunga durata il suo valore può essere di un processo all'ora, mentre per brevi transizioni è possibile avere un throughput di 10 processi al secondo.
- **Tempo di completamento:** dal punto di vista di uno specifico processo, il criterio più importante è il tempo necessario per eseguire il processo stesso. L'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato *tempo di completamento*, ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella ready queue, durante l'esecuzione nella CPU e nelle operazioni di I/O. Il tempo di completamento non può essere migliorato dallo scheduler.
- **Tempo d'attesa:** l'algoritmo di scheduling della CPU non influisce sul tempo impiegato per l'esecuzione di un processo o di un'operazione di I/O; influisce solo sul tempo d'attesa nella ready queue. Il tempo d'attesa è la somma degli intervalli d'attesa passati in questa coda.
- **Tempo di risposta:** in un sistema interattivo il tempo di completamento può non essere il miglior criterio di valutazione: spesso accade che un processo emetta dati abbastanza presto, e continui a calcolare i nuovi risultati mentre quelli precedenti sono in fase di output per l'utente. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la effettuazione di una richiesta e la prima risposta prodotta. Questa misura è chiamata tempo di risposta, ed è dato dal tempo necessario per iniziare la risposta, ma non dal suo tempo necessario per completare l'output.

E' auspicabile aumentare al massimo utilizzo e produttività della CPU, mentre il tempo di completamento, di attesa e di risposta si devono ridurre al minimo. Nella maggior parte dei casi si ottimizzano i valori medi.

## Algoritmi di scheduling

Lo scheduling della CPU si occupa di decidere quale dei processi nella ready queue debba essere assegnato al core della CPU.

### Scheduling in ordine di arrivo (scheduling first come, first served)

Si pensi alla fila al supermercato (se facciamo prima, dobbiamo andare prima).

E l'algoritmo di scheduling della CPU più semplice che ci sia. Con questo scheduling la CPU si assegna al processo che la richiede per primo. La realizzazione si basa su una coda FIFO: quando la CPU è libera, viene assegnata al processo che si trova in testa alla coda, rimuovendolo da essa.

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

■ Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:



Un aspetto negativo è che il tempo medio d'attesa per l'algoritmo FCFS è spesso abbastanza lungo. Molto spesso si ottiene un *effetto convoglio*: tutti i processi attendono che un lungo processo liberi la CPU, il che causa una riduzione dell'utilizzo della CPU e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi.

L'algoritmo di scheduling FCFS è non preemptive: una volta che la CPU è assegnata a un processo, questo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o al termine di un'operazione di I/O. L'algoritmo FCFS risulta particolarmente problematico nei sistemi time-sharing, dove è importante che ogni utente disponga della CPU a intervalli regolari. Permettere a un solo processo di occupare la CPU per un lungo periodo condurrebbe a risultati disastrosi.

### Scheduling shortest-job-first (SJF)

Devo sempre eseguire prima il processo con CPU-burst minore (quando è possibile) per migliorare il tempo di attesa medio.

SJF è ottimale, la difficoltà è proprio individuare le lunghezze del burst dei processi. Non ho informazioni attendibili da nessuna parte, l'unica cosa è chiedere all'utente.

Process	Burst Time
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

■ SJF scheduling chart

$P_4$	$P_1$	$P_3$	$P_2$
0	3	9	16

■ Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

SJF è ottimale: il tempo d'attesa medio diminuisce.

Notiamo che lo scheduler è no preemptive, ovvero non c'è limite al burst di CPU.

#### Determinare la lunghezza del prossimo CPU burst

SJF non si può realizzare a livello dello scheduler della CPU a breve termine, poiché non esiste alcun modo per conoscere la lunghezza della successiva sequenza di operazioni della CPU.

Posso fare solo delle stime basate sul comportamento passato e recente e su quello presente.

Faccio quindi uso della *media esponenziale*:

- $t_n \rightarrow$  lunghezza dell'ennesimo CPU burst, contiene informazioni più recenti

- $\tau_{n+1} \rightarrow$  prevede il valore del prossimo CPU burst
- alfa compreso tra 0 ed 1  $\rightarrow$  controlla il peso relativo sulla predizione della storia recente ( $t$ ) e di quella passata ( $\tau$ )

Definisco

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Comunemente, alfa è pari ad 1/2.

Ogni volta devo conservarmi la predizione precedente, la combino con quello che è stato misurato e mi ricavo la lunghezza prevista per l'intervallo di tempo successivo.

Caso alfa = 0:

$$\tau_{n+1} = \tau_n$$

la storia recente non conta.

Caso alfa = 1:

$$\tau_{n+1} = t_n$$

conta solo l'ultimo CPU burst.

Caso con alfa compreso tra 0 e 1:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

I burst misurati vengono misurati con coefficienti via via decrescenti (dato che sia alfa che  $(1-\alpha)$  sono minori di 1) in modo da scordarsi di quello successo in passato e tenendo conto solo della storia recente.

L'algoritmo SJF può essere sia con prelazione che senza. Dipende se quando arriva un nuovo processo nella ready queue un altro è ancora in esecuzione.

Un algoritmo SJF con prelazione sostituisce il processo già in esecuzione, mentre un algoritmo senza prelazione permette al processo correttamente in esecuzione di portare a termine il proprio CPU burst.

Versione con prelazione chiamata: short-remaining-time-first.

I processi non arrivano insieme ma in tempi varianti.

Now we add the concepts of varying arrival times and preemption to the analysis		
Process	Arrival Time	Burst Time
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

■ Preemptive SJF Gantt Chart

■ Average waiting time =  $[(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5$  msec

Esempio: quando arriva  $P_2$ ,  $P_1$  aveva già fatto 1 ms di burst.  $4 < 7$  quindi  $P_2$  vince e si sostituisce al processo  $P_1$ .

La previsione si adatta a quello che succede in tempi recenti.

## Scheduling circolare (round robin-algoritmo dell'orologio)

Algoritmo tipico dei sistemi interattivi, in cui vogliamo avere l'illusione della concorrenza e non posso bloccare per troppo tempo i processi dalla CPU.

Il Round Robin è simile all'algoritmo di scheduling FCFS ma aggiunge la capacità di prelazione in modo che il sistema possa commutare tra i vari processi.

Ogni processo ha una piccola unità di tempo della CPU (*quanto di tempo*), in media pari a 10-100 millisecondi. Dopo che questo tempo è finito, il processo è preempted ed è inserito alla fine della ready queue  $\rightarrow$  lo scheduler della CPU scorre la ready queue, assegnando la CPU a ciascun processo per un intervallo di tempo della durata massima di un quanto di tempo. I tempi brevissimi ci illudono che tutto avvenga allo stesso tempo. Molti sistemi operativi moderni hanno un quanto di tempo variabile.

Per implementare lo scheduling RR si gestisce la ready queue come una coda FIFO. I nuovi processi si aggiungono alla fine della ready queue. Lo scheduler della CPU prende il primo processo dalla ready queue, imposta un timer in modo che invii un segnale d'interruzione alla scadenza del quanto di tempo e attiva il dispatcher per eseguire il processo.

A questo punto si può verificare una delle due seguenti situazioni:

- il processo ha una sequenza di operazioni della CPU di durata minore di un quanto di tempo, quindi il processo stesso rilascia volontariamente la CPU e lo scheduler passa al processo successivo nella ready queue;
- la durata della sequenza di operazioni è più lunga di un quanto di tempo e in questo caso il timer scade e invia un segnale d'interruzione al sistema operativo, che esegue un cambio di contesto e mette il processo alla fine della ready queue. Lo scheduler quindi seleziona il processo successivo nella ready queue

Se ci sono n processi nella ready queue e il tempo di quanto è q, allora ogni processo avrà  $1/n$  del tempo della cpu. Nessun processo aspetterà meno di  $(n-1)q$  unità di tempo.

Dati 5 processi e un quanto di 20 millisecondi, ogni processo usa fino a 20 ms ogni 100 ms.

Se ho un grande quanto di tempo: tutti riescono a fare il CPU burst alla prima botta e seguono praticamente la ready queue.

Se ho un piccolo quanto di tempo: potrei avere un overhead troppo grande.

Esempio di RR con  $q = 4$ :

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

■ The Gantt chart is:



Il tempo di completamento è in media maggiore rispetto al SJF, ma il tempo di risposta è migliore.

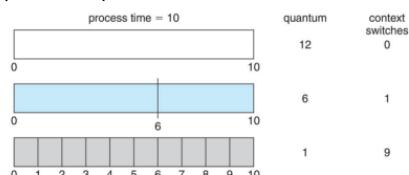
L'algoritmo RR è con prelazione dovuto al context switch successivo ad un processo che dura più del quanto.

Il tempo di completamento dipende dalla dimensione del quanto di tempo. Esso non migliora necessariamente con l'aumento della dimensione del quanto.

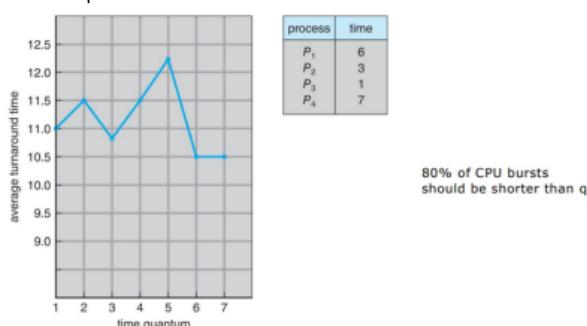
In generale, il tempo di completamento medio può migliorare se la maggior parte dei processi termina la successiva CPU burst in un quanto di tempo.

Come già detto, il quanto dev'essere abbastanza grande rispetto al tempo del context switch (se il quanto è troppo grande però si tende all'FCFS).

Empiricamente, si può stabilire che l'80% delle sequenze di operazioni della CPU debba essere più breve del quanto di tempo.



Tempi di attesa alla variazione del quanto di tempo:



## Scheduling con priorità

Algoritmo proprio dei sistemi dei giorni nostri.

L'algoritmo SJF è un caso particolare del più generale algoritmo di scheduling con priorità: *si associa una priorità ad ogni processo e si assegna la CPU al processo con priorità più alta; i processi con priorità uguali si ordinano secondo uno schema FCFS*.

Un algoritmo SJF è semplicemente un algoritmo con priorità in cui la priorità  $p$  è l'inverso della lunghezza prevista dalla successiva sequenza di operazioni della CPU. A una maggiore lunghezza corrisponde una minore priorità e viceversa.

In alcuni sistemi priorità più alta = intero minore; in altri, priorità più alta = intero maggiore.

Nel nostro caso, numeri bassi indicano priorità alte.

La priorità può essere definita sia internamente (limiti di tempo, requisiti di memoria, numero di file aperti) o esternamente (tutti ciò al di fuori del sistema operativo, come l'importanza del processo, il tipo e la quantità di fondi pagati per l'uso del calcolatore).

Lo scheduling con priorità può essere sia con prelazione che senza. Quando un processo arriva alla ready queue, si confronta la sua priorità con quella del processo attualmente in esecuzione. Un algoritmo di scheduling con priorità con diritto di prelazione sottrae la CPU al processo attualmente in esecuzione se la priorità del processo appena arrivato è superiore. Un algoritmo senza prelazione si limita a porre l'ultimo processo arrivato in testa alla ready queue.

Un problema importante relativo agli algoritmi di scheduling con priorità è l'*attesa indefinita (starvation)*. Un processo pronto per l'esecuzione, ma che non dispone della CPU, si può considerare bloccato nell'attesa della CPU. Un algoritmo di scheduling con priorità può lasciare processi a bassa priorità nell'attesa indefinita della CPU.

Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'*invecchiamento (aging)*; si tratta di una tecnica di aumento graduale delle priorità dei processi che attendono nel sistema per parecchio tempo.

Per esempio, se le priorità variano da 127 (bassa) a 0 (alta), si potrebbe incrementare di 1 ogni secondo il livello di priorità di un processo in attesa.

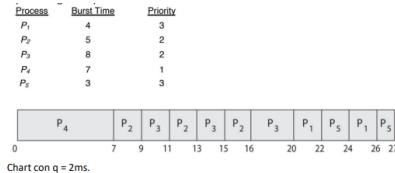
Alla fine anche un processo con priorità iniziale 127 può ottenere la priorità massima nel sistema e quindi essere eseguito: un processo con priorità 127 impiegherebbe poco più di 2 minuti per raggiungere la priorità 0.

Process	Burst Time	Priority
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

■ Priority scheduling Gantt Chart



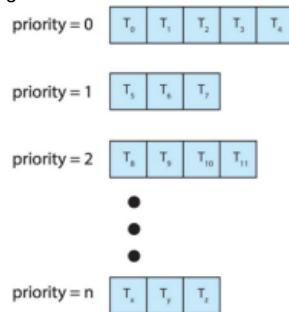
Un'altra opzione consiste nel combinare scheduling circolare e scheduling con priorità in modo tale che il sistema esegua il processo con la priorità più alta ed esegua processi con la stessa priorità utilizzando lo scheduling circolare.



## Scheduling a code multilivello

Con una sola coda di priorità potrebbe essere necessaria una ricerca  $O(n)$  per determinare il processo con maggiore priorità.

E' più conveniente utilizzare una coda per ogni priorità. Eseguo i processi nella coda con priorità più alta e funziona bene anche quando lo scheduling con priorità è combinato con lo scheduling RR.



Un algoritmo di scheduling a code multilivello può anche essere utilizzato per suddividere i processi in diverse code in base al tipo di processo. Per esempio, di solito vengono divisi in *processi in primo piano (interattivi)* e in *processi in background (batch)*. Questi due tipi di processi hanno requisiti diversi in termini di tempo di risposta e possono quindi avere esigenze di scheduling diverse.

Inoltre, i processi in primo piano possono avere una priorità più alta (definita esternamente) rispetto ai processi in background. E' possibile utilizzare code distinte per i processi in primo piano e in background e ciascuna coda potrebbe avere il proprio algoritmo di scheduling.

In questa situazione è inoltre necessario avere uno scheduling tra le code; si tratta comunemente di uno scheduling a priorità fissa con prelazione. Nessun processo della coda con prelazione più bassa può iniziare l'esecuzione finché le code superiori non sono tutte vuote.

Si può anche riservare porzioni di tempo per ogni coda: a ogni coda si assegna una parte del tempo di elaborazione della CPU, suddivisibile a sua volta tra i processi che la costituiscono.

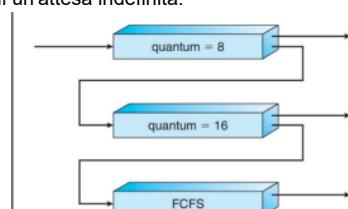
## Scheduling a code multilivello con retroazione

Di solito in un algoritmo di scheduling a code multilivello i processi si assegnano in modo permanente a una coda all'entrata nel sistema e non si possono sposare tra le code. Se, per esempio, esistono code distinte per i processi che si eseguono in foreground e quelli che si eseguono in background, i processi non possono passare da una coda all'altra, poiché non possono cambiare la loro natura di processi in foreground o in background. Questa impostazione è rigida, ma ha il vantaggio di avere un basso carico di scheduling.

Lo scheduling a code multilivello con retroazione, invece, permette ai processi di spostarsi tra le code. L'idea consiste nel separare i processi che hanno caratteristiche in termini di CPU burst. Se un processo usa troppo tempo di elaborazione della CPU, viene spostato in una coda con priorità più bassa.

Questo schema mantiene i processi con prevalenza di I/O e i processi interattivi nelle code con priorità più alta. Inoltre, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo in una coda a priorità bassa.

Questa forma di agiing impedisce il verificarsi di un'attesa indefinita.



Se un processo supera il quanto definito dalla coda, viene degradato alla coda inferiore.

E' l'algoritmo di scheduling più complesso da implementare (la definizione dello scheduler migliore richiede particolari metodi per la selezione dei valori dei diversi parametri) ma è anche quello che costituisce il criterio più generale per lo scheduling della CPU.

Generalmente, uno scheduler a code multilivello con retroazione è caratterizzato dai seguenti parametri:

- numero di code
- algoritmo di scheduling per ciascuna coda
- metodo usato per determinare quando spostare un processo in una coda con priorità maggiore
- metodo usato per determinare quando spostare un processo in una coda con priorità minore
- metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio

## Scheduling dei thread

Un sistema operativo che riconosce i thread può schedularli separatamente dai processi. Una distinzione fra thread a livello utente e a livello kernel riguarda proprio il modo in cui vengono schedulati.

I thread a livello utente vengono schedulati dalla libreria dei thread: si parla di *process-contention scope (PCS)*, perché la contesa per aggiudicarsi la CPU ha luogo fra thread dello stesso processo.

Per determinare quale thread a livello kernel debba essere eseguito sulla CPU, il kernel esamina tutti i thread: si parla di *system-contention scope (SCS)*. Quindi tutti i thread competono per l'utilizzo della CPU.

Nel caso del PCS, lo scheduling è in base alle priorità. Le priorità dei thread a livello utente sono stabilite dal programmatore e la libreria dei thread non le modifica.

## Scheduling per sistemi multiprocessore

Se sono disponibili più unità di elaborazione è possibile la *distribuzione del carico*, ma il problema dello scheduling diventa proporzionalmente più complesso.

Tradizionalmente, il termine multiprocessore faceva riferimento a sistemi che fornivano più processori fisici, in cui ogni processore conteneva una CPU single-core. Tuttavia, la definizione di multiprocessore si è evoluta in modo significativo e sui moderni sistemi di elaborazione il termine multiprocessore si applica ora alle seguenti architetture di sistema:

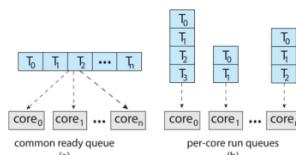
- CPU multicore
- Core multithread (hyper-threading -> 1 CPU con più processi in corso)
- Sistemi NUMA
- Sistemi multiprocessore eterogenei -> a differenza dei primi tre casi non tutti i processori hanno le stesse capacità

## Approcci allo scheduling per multiprocessori

Una prima strategia di scheduling della CPU per i sistemi multiprocessore affida tutte le decisioni, l'elaborazione dell'I/O e altre attività del sistema a un solo processore, il cosiddetto *master server*. Gli altri processori eseguono soltanto il codice utente. Si tratta dell'*elaborazione asimmetrica*, che riduce la necessità di condividere dati grazie all'accesso di un solo processore alle strutture dati del sistema. Lo svantaggio di questo approccio è che il master server costituisce un potenziale collo di bottiglia in grado di ridurre le prestazioni generali del sistema.

L'approccio standard per supportare i multiprocessori è la *multielaborazione simmetrica (SMP)*, in cui ciascun processore è in grado di autogestirsi. Lo scheduling viene realizzato facendo in modo che lo scheduler di ciascun processore esamini la ready queue e selezioni un thread da eseguire. Si noti che questo approccio offre due possibili strategie per organizzare i thread da selezionare per l'esecuzione:

1. tutti i thread possono trovarsi in una ready queue comune
2. ogni processore può avere una propria coda privata di thread



Se viene utilizzata la prima opzione è possibile avere race condition (fenomeno che si presenta nei sistemi concorrenti quando, in presenza di una sequenza di processi multipli, il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguiti) sulla coda condivisa e occorre quindi assicurarsi che due processori distinti non scelgano di eseguire lo stesso thread e che i thread nella coda non vengano persi. E' possibile usare una forma di lock per proteggere la ready queue comune da questa race condition. Il lock sarebbe tuttavia molto conteso, poiché tutti gli accessi alla coda richiederebbero il possesso del lock e l'accesso alla coda condivisa costituirebbe probabilmente un collo di bottiglia per le prestazioni.

La seconda opzione permette a ciascun processore di schedulare i thread da una coda di esecuzione privata e pertanto non risente dei possibili problemi di prestazioni associati ad una coda condivisa. Per questa ragione si tratta dell'approccio più comune: l'uso di code di esecuzione private, una per ogni processore, può portare a un uso più efficiente della memoria cache.

Vi sono anche dei problemi nell'utilizzo di code distinte per ogni processore, in particolare relativi ai diversi carichi di lavoro, ma possono essere utilizzati algoritmi di bilanciamento per distribuire equamente i carichi di lavoro tra tutti i processori.

Praticamente tutti i sistemi operativi moderni supportano SMP, inclusi Windows, Linux, macOS e i sistemi mobili Android e iOS.

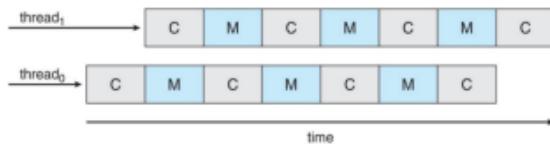
## Processori multicore

Tradizionalmente, i sistemi SMP (a multielaborazione simmetrica) hanno reso possibile la concorrenza tra thread con l'utilizzo di diversi processori fisici. Tuttavia, la pratica recente nel progetto hardware dei calcolatori è di inserire *più core di elaborazione in un unico chip fisico*, dando origine a un processore multicore. Ogni core mantiene il proprio stato e appare dunque al sistema operativo come un processore fisico separato. I sistemi SMP che usano processori multicore sono più veloci e consumano meno energia dei sistemi in cui ciascun processore è costituito da un singolo chip.

I processori multicore possono complicare i problemi relativi allo scheduling.

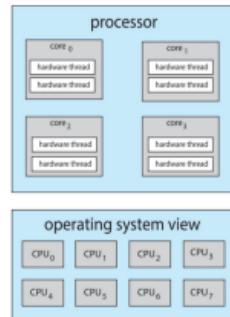
Quando un processore accede alla memoria, una quantità significativa di tempo trascorre in attesa della disponibilità dei dati: *stallo della memoria* che può verificarsi per varie ragioni, per esempio la mancanza dei dati richiesti nella cache.

Per rimediare a questa situazione, molti dei progetti hardware recenti implementano delle unità di calcolo multithread in cui due o più thread hardware sono assegnati a un singolo core. In questo modo, se un thread è in situazione di stallo in attesa della memoria, il core può passare a eseguire un altro thread.



Dal punto di vista del sistema operativo, ogni thread hardware mantiene il suo stato architettonico, come il puntatore alle istruzioni e il set di registri, e appare quindi come una CPU logica che è disponibile per eseguire un thread software.

Questa tecnica, nota come *chip multithreading (CMT)*, è mostrata in Figura:



è mostrato un processore contenente quattro core di elaborazione, ognuno dei quali contiene due thread hardware -> dal punto di vista del sistema operativo sono presenti 8 CPU logiche.

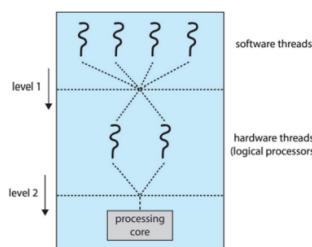
**I processori Intel utilizzano il termine hyper-threading per descrivere l'assegnazione di più thread hardware a un singolo core di elaborazione.**

Ci sono due modi per rendere un core multithread:

- *multithreading a grana grossa*: un thread resta in esecuzione su un processore fino al verificarsi di un evento a lunga latenza, per esempio uno stallo di memoria. A causa dell'attesa introdotta dall'evento a lunga latenza, il processore deve passare a un altro thread e iniziare a eseguirlo. Tuttavia, il costo per cambiare il thread in esecuzione è alto, poiché occorre ripulire la pipeline delle istruzioni prima che il nuovo thread possa iniziare a riempire la pipeline con le sue istruzioni. Una volta che il nuovo thread è in esecuzione inizia a riempire la pipeline con le sue istruzioni.
- *multithreading a grana fine*: passa da un thread a un altro a un livello molto più fine di granularità (tipicamente al termine di un ciclo di istruzioni). Tuttavia, il progetto di sistemi a multithreading fine include una logica dedicata al cambio di thread. Ne risulta così che il costo del passaggio da un thread a un altro è piuttosto basso.

E' importante notare che le risorse del core fisico (come cache e pipeline) devono essere condivise tra i suoi thread hardware e dunque un core di elaborazione può eseguire solo un thread hardware alla volta.

Di conseguenza, un processore multithread e multicore richiede due livelli diversi di scheduling:



1. il sistema operativo decide quale thread software eseguire su una CPU logica
2. ogni core decide quale thread hardware mandare in esecuzione sulla CPU fisica

## Bilanciamento del carico

Sui sistemi SMP è importante che il carico di lavoro sia distribuito equamente tra tutte le unità di elaborazione per sfruttare appieno i vantaggi di avere più processori. Se ciò non avvenisse, alcuni processori potrebbero restare inattivi mentre altri verrebbero intensamente sfruttati su una coda di processi in attesa.

Il bilanciamento del carico tenta di ripartire il carico di lavoro uniformemente tra tutti i processori di un sistema SMP.

Bisogna notare che il bilanciamento è necessario, di norma, solo nei sistemi in cui ciascun processore ha una coda privata di accessi eseguibili. Nei sistemi che mantengono una coda comune, il bilanciamento del carico è superfluo: un processore inattivo passerà immediatamente all'esecuzione di un processo dalla coda comune dei processi eseguibili.

Il bilanciamento del carico può seguire due approcci:

1. *migrazione push*: prevede che un processo apposito controlli periodicamente il carico di ogni processore e se identifica uno sbilanciamento, riporta il carico in equilibrio, spostando i processi dal processore saturo ad altri più liberi o inattivi.
2. *migrazione pull*: si ha quando un processore inattivo prende un processo in attesa a un processore sovraccarico

I due tipi di migrazione non sono mutuamente esclusivi e trovano spesso applicazione contemporanea.

Il concetto di carico bilanciato può avere significati diversi: una prima possibilità è richiedere semplicemente che tutte le code contengano approssimativamente lo stesso numero di thread. In alternativa, il bilanciamento potrebbe richiedere un'equa distribuzione delle proprietà dei thread su tutte le code. Inoltre, in determinate situazioni potrebbe non essere sufficiente alcuna di queste strategie, poiché esse possono essere in contrasto con gli obiettivi dell'algoritmo di scheduling.

## Predilezione per il processore

Tentativo da parte del sistema operativo di mandare un processo in esecuzione su una CPU dov'è già stato per velocizzare l'esecuzione, dato che i dati che il processore ha trattato più recentemente permangono nella cache e, di conseguenza, i successivi accessi alla memoria da parte del processo tendono a utilizzare spesso la cache (sfrutta la warm cache).

Il bilanciamento del carico potrebbe avere un effetto sull'affinity (predilezione), dato che un processo potrebbe essere mosso da un processore all'altro per bilanciare i carichi di lavoro, ma il processo potrebbe perdere i contenuti di quello che aveva nella cache del processore da cui è stato mosso.

*Soft affinity*: il sistema operativo tenta di mantenere il processo in esecuzione sullo stesso processore, ma non ci sono garanzie.

*Hard affinity*: il processo specifica un insieme di processori su cui può essere eseguito.

## Scheduling real-time della CPU

Lo scheduling della CPU nei sistemi real-time ha peculiarità proprie.

- *Sistemi real time soft*: non offrono garanzie sul momento in cui un processo critico sarà eseguito, ma assicurano solamente che sarà data precedenza a quest'ultimo piuttosto che ad altri processi non critici.
- *Sistemi real time hard*: hanno vincoli più rigidi -> i task vanno eseguiti entro una scadenza prefissata ed eseguiti dopo tale scadenza è del tutto inutile

POSIX come Linux prevede processi real time soft. Tutti i sistemi operativi prevedono code di processi basati sulla priorità, in modo da ridurre il più possibile il tempo di context switch. E' utile avere un tempo di scheduling O(1), ovvero indipendente dal numero di processi nel sistema.

Questo obiettivo è stato raggiunto dallo scheduler di Linux nel kernel versione 2.5.

Tutti i sistemi UNIX tendono a degradare i processi che utilizzano troppa CPU, in modo da dare priorità ai processi interattivi per dare soddisfazione all'utente.

I tempi dei processi vanno da 0 a 99.

Nice-value (tempi da 100 a 140): valore dato dall'utente (anche da linea di comando) che cambia la priorità in base alle esigenze dell'utente.

Priority boost: innalzamento della priorità dei processi che sono in attesa di qualcosa dalla rete, al beneficio del sistema in modo da far passare subito i messaggi ricevuti.

## Valutazione degli algoritmi

Ci si può chiedere come scegliere un algoritmo di scheduling della CPU per un sistema particolare.

Il primo problema da affrontare riguarda la definizione dei criteri da utilizzare per la scelta dell'algoritmo.

I criteri si definiscono spesso in termini di utilizzo della CPU, tempo di risposta o throughput. Tra i criteri suggeriti si possono inserire diverse misure. Una volta definiti i criteri di selezione è necessario valutare gli algoritmi considerati.

## Modellazione deterministica

La modellazione deterministica è un tipo di valutazione analitica, che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro.

Per ogni algoritmo, calcolo il tempo di attesa medio minimo. Anche se è un metodo semplice e veloce, richiede un numero esatto di numeri per input e si applica SOLO a quegli input.

## Reti di code

Descrive l'arrivo dei processi e dei burst CPU/I/O in modo probabilistico.

Si ottiene una formula matematica che indica la probabilità di una determinata sequenza di operazioni della CPU.

Comunemente questa distribuzione è di tipo esponenziale ed è descritta dalla sua media.

Analogamente, è possibile caratterizzare anche la distribuzione degli istanti d'arrivo dei processi nel sistema. Da queste due distribuzioni si può calcolare il throughput medio, l'utilizzo della CPU o il tempo di attesa medi, per la maggior parte degli algoritmi.

Il sistema di calcolo si descrive come una rete di server, ciascuno con una coda d'attesa.

La CPU è un server con la propria ready queue, così come il sistema di I/O con le sue code di attesa dei dispositivi.

Se sono noti le distribuzioni degli arrivi e dei servizi, si possono calcolare l'utilizzo, la lunghezza media delle code, il tempo medio d'attesa e così via. Questo tipo di studio si chiama *analisi delle reti di code*.

Si consideri il seguente esempio: sia  $n$  la lunghezza media di una coda, escluso il processo correntemente servito, detti  $W$  il tempo medio d'attesa della coda e  $\lambda$  il tasso medio d'arrivo dei nuovi processi nella coda; si prevede che, nel tempo  $W$  durante il quale un processo attende nella coda, raggiungano la coda  $\lambda W$  processi.

Se il sistema è stabile, il numero dei processi che lasciano la coda dev'essere uguale al numero dei processi che vi arrivano:

$$n = \lambda * W$$

Quest'equazione è nota come **formula di Little** ed è utile soprattutto perché è valida per qualsiasi algoritmo di scheduling e distribuzione degli arrivi.

Essa è utilizzabile per il calcolo delle tre variabili, quando sono note le altre due.

Poiché può essere difficile lavorare matematicamente con distribuzioni e algoritmi complicati, spesso le distribuzioni d'arrivo e servizio vengono definite in maniera matematicamente trattabile, ma non realistica: le reti di code si limitano ad approssimare un sistema reale, rendendo discutibile la precisione dei risultati ottenuti.

## Simulazioni

I modi di incodamento sono limitati: le simulazioni sono più accurate.

- Modelli programmati di un sistema
- Il clock è variabile
- Raccoglie statistiche indicanti le prestazioni dell'algoritmo
- I dati per le simulazioni sono raccolti da: distribuzioni definite matematicamente o empiricamente, traccia red di sequenza di eventi reali in sistemi reali, RNG dovuto alle probabilità

## Scheduler preemptive e non preemptive

Le decisioni riguardanti lo scheduling della CPU si possono prendere nelle seguenti circostanze:

1. Un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di I/O o richiesta di attesa (wait) per la terminazione dei processi figli)
2. Un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale di interruzione)
3. Un processo passa dallo stato di attesa allo stato pronto (per esempio, al completamento di un'operazione di I/O)
4. Un processo termina

I casi 1 e 4 non danno alternative in termini di scheduling: si deve comunque scegliere un nuovo processo da eseguire, sempre che ce ne sia almeno uno nella ready queue. Una scelta si può fare nei casi 2 e 3.

Quando lo scheduling interviene nelle condizioni 1 e 4, si dice che lo schema di scheduling è senza diritto di prelazione (non preemptive) o cooperativo; altrimenti, lo schema di scheduling è con diritto di prelazione (preemptive).

Nel caso dello scheduling senza diritto di prelazione, quando si assegna la CPU ad un processo, questo rimane in possesso della CPU fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al passaggio nello stato di attesa.

Lo scheduling con diritto di prelazione presenta un inconveniente: si consideri il caso in cui due processi condividono dati. Mentre uno di questi aggiorna i dati si ha la sua prelazione in favore dell'esecuzione dell'altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato incoerente al primo processo. Sono quindi necessari nuovi meccanismi per coordinare l'accesso ai dati condivisi.

## Quali sono gli algoritmi di scheduling?

Ci sono diversi algoritmi di scheduling:

- Quello più banale da implementare è il *First Come First Serve* (FCFS). Si basa sulla logica delle code FIFO: il primo processo che arriva nella CPU viene eseguito. Se i processi non sono molto lunghi e non hanno troppe chiamate di sistema bloccanti che fanno entrare in stallo il processo potrebbe anche essere considerato un buon algoritmo di scheduling, ma purtroppo non è così e questo potrebbe comportare lunghi tempi di attesa.
- L'algoritmo per brevità (SJF) invece, idealmente dovrebbe controllare la lunghezza di ogni processo e assegnare la CPU al processo più breve, così da diminuire il tempo medio di attesa. Questo algoritmo potrebbe essere senza prelazione e permetterebbe al processo di permanere nella CPU fino al suo termine, mentre nel caso fosse con diritto di prelazione sostituirebbe il processo già presente nella CPU, nel caso in cui ne arrivasse un altro con un livello di prelazione più alto. Tutto ciò è difficile da implementare perché si dovrebbe sapere il tempo di impiego della CPU prima che ancora il processo entri al suo interno.
- Abbiamo anche un algoritmo che ci permette di gestire i processi in base alla priorità, scheduling con priorità. Questo algoritmo associa la CPU al processo con priorità maggiore. Se ci dovessero essere due processi con la stessa priorità si utilizza il concetto dell'algoritmo FCFS (viene servito il primo che tra i due è arrivato).
- Infine abbiamo l'algoritmo circolare (Round Robin), che viene usato maggiormente per sistemi a tempo ripartito, per ogni processo c'è un quanto di tempo che gli viene assegnato per permanere e sfruttare la CPU; nel momento in cui il quanto di tempo è finito ma il processo non ha ancora terminato la sua esecuzione, il processo viene sottoposto a prelazione e riportato nella coda dei processi pronti. Riguardo alle prestazioni dello scheduler RR, occorre considerare l'effetto dei cambi di contesto; quindi il quanto di tempo dev'essere ampio rispetto alla durata del cambio di contesto ma non eccessivo perché altrimenti si tenderebbe al criterio FCFS.

## Quali sono i criteri di scheduling?

Diversi algoritmi di scheduling della CPU hanno proprietà differenti e possono favorire una particolare classe di processi. Prima di scegliere l'algoritmo da usare in una specifica situazione occorre considerare le caratteristiche dei diversi algoritmi.

Di seguito si riportano alcuni criteri:

1. *Utilizzo della CPU*: la CPU dev'essere più attiva possibile. Teoricamente, l'utilizzo della CPU può variare dal 0 al 100%. In un sistema reale può variare dal 40% per un sistema poco carico, al 90% per un sistema con utilizzo intenso.
2. *Throughput*: la CPU è attiva quando si svolge del lavoro. Una misura del lavoro svolto è data dal numero dei processi completati nell'unità di tempo: tale misura è detta throughput. Per processi di lunga durata, questo rapporto può essere di un processo all'ora, mentre per brevi transazioni è possibile avere una produttività di dieci processi al secondo.
3. *Tempo di completamento*: rappresenta l'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella coda dei processi pronti, durante l'esecuzione della CPU e nelle operazioni di I/O.
4. *Tempo di attesa*: l'algoritmo di scheduling della CPU non influisce sul tempo impiegato per l'esecuzione di un processo o di un'operazione di I/O; influisce solo sul tempo di attesa nella coda dei processi pronti. Il tempo di attesa è la somma degli intervalli di attesa passati nella coda

dei processi pronti.

5. *Tempo di risposta*: in un sistema interattivo il tempo di completamento può non essere il miglior criterio di valutazione. Spesso accade che un processo emetta dati abbastanza presto e continui a calcolare i nuovi risultati mentre quelli precedenti sono in fase di emissione. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta. Questa misura è chiamata tempo di risposta ed è data dal tempo necessario per iniziare la risposta, ma non dal suo tempo di emissione. Generalmente il tempo di completamento è limitato dalla velocità del dispositivo di emissione dei dati.

## Cosa fa il dispatcher?

Si tratta del modulo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine.

Questa funzione riguarda quel che segue:

1. il cambio di contesto;
2. il passaggio alla modalità utente;
3. il salto alla giusta posizione del programma utente per riavviare l'esecuzione.

Poiché si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere quanto più rapido possibile.

# Capitolo 6 - Sistemi Operativi

## Strumenti di sincronizzazione

### Introduzione

I processi possono essere eseguiti in modo concorrente o in modo parallelo.

In questo capitolo viene spiegato come l'esecuzione concorrente o parallela possa contribuire a problematiche che riguardano l'integrità dei dati condivisi da più processi.

Un *processo cooperante* è un processo che può influenzare un altro processo del sistema e può essere influenzato da esso.

Si parla di *sincronizzazione* nel caso in cui due processi lavorano con una memoria comune.

La condivisione della memoria può portare ad un'inconsistenza dei dati.

L'accesso alla memoria condivisa si ha in seguito al permesso del sistema operativo; questo crea dei problemi quando ci sono dei processi/threads che cercano di accedere alla memoria condivisa. Si pensi al modello produttore/consumatore che può portare ad una *race condition*.

Occorre assicurare che un solo processo alla volta possa modificare la variabile del contatore.

Un buffer limitato è utilizzato per permettere ai processi la condivisione della memoria.

Nell'esempio del produttore e del consumatore, abbiamo un buffer limitato e due puntatori.

Il produttore deve badare alla velocità del get del consumatore e il consumatore non può più prelevare elementi già presi o non può prelevare da un buffer vuoto. Quando i due puntatori sono sovrapposti, non c'è nulla da prelevare.

E se volessimo una soluzione in cui i due processi riempiono tutti i buffer? Abbiamo un *integer counter* che tiene traccia di tutti i numeri dei full buffer. Il contatore è inizializzato a 0 ed è incrementato dal produttore dopo che produce un nuovo buffer ed è decrementato quando il consumatore consuma un buffer.

```
Produttore- codice
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

Consumatore- codice
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in next consumed */
}
```

Le due procedure possono non funzionare correttamente se si eseguono in modo concorrente: questa implementazione non funziona -> non posso incrementare/decrementare una variabile in un'unica istruzione a basso livello.

Es.: il valore del contatore è 5. Il produttore e il consumatore eseguono le istruzioni contatore++ e contatore-- in modo concorrente. Terminata l'esecuzione delle istruzioni, il valore della variabile contatore potrebbe essere 4, 5 o 6!

```
■ counter++ could be implemented as
register1 = counter
register1 = register1 + 1
counter = register1

■ counter-- could be implemented as
register2 = counter
register2 = register2 - 1
counter = register2

■ Consider this execution interleaving with "count = 5" initially:
S0: producer execute register1 = counter          (register1 = 5)
S1: producer execute register1 = register1 + 1    (register1 = 6)
S2: consumer execute register2 = counter          (register2 = 5)
S3: consumer execute register2 = register2 - 1    (register2 = 4)
S4: producer execute counter = register1          (counter = 6)
S5: consumer execute counter = register2          (counter = 4)
```

Il solo risultato corretto è contatore == 5, che si ottiene se si eseguono separatamente produttore e consumatore. Anche se il processore è sempre lo stesso, è come se i registri dei processi siano separati -> interleaving della sequenza delle istruzioni.

Le operazioni di scrittura in memoria non sono atomiche (atomica riferito al significato di indivisibile): i processi prendono un'immagine, la processano e la salvano.

Per evitare le situazioni in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (*race condition*), occorre assicurare che un solo processo alla volta possa modificare la variabile contatore.

Questa garanzia richiede una forma di sincronizzazione dei processi.

Tali situazioni si verificano spesso nei sistemi operativi, nei quali diversi componenti del sistema compiono operazioni su risorse condivise. Inoltre, la crescente importanza dei sistemi multicore ha dato maggior enfasi allo sviluppo di applicazioni multi-thread in cui diversi thread, che probabilmente possono condividere dati, sono in esecuzione in parallelo su core distinti.

### Problema della sezione critica

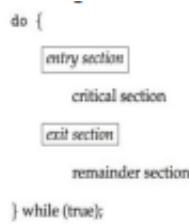
Consideriamo un sistema composto da n processi: ogni segmento ha una parte di codice detta *sezione critica*, in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via.

La caratteristica fondamentale del sistema è che, quando un processo è in esecuzione nella propria sezione critica, non si consente ad alcun altro processo di essere in esecuzione nella propria sezione critica.

Il problema della sezione critica consiste nel progettare un protocollo che i processi possano usare per cooperare. Ogni processo deve chiedere il

permesso per entrare nella propria sezione critica: la sezione di codice che realizza questa richiesta è la **sezione d'ingresso**. La sezione critica può essere eseguita da una **sezione di uscita** e la restante parte del codice è detta sezione non critica.

Struttura generale di un tipico processo:



Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti:

1. **Mutua esclusione**: se il processo P è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. **Progresso**: se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.
3. **Attesa limitata**: se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

Si suppone che ogni processo sia eseguito a una velocità diversa da zero. Tuttavia, non si può fare alcuna ipotesi sulla velocità relativa degli n processi.

In un dato momento, numerosi processi in modalità kernel possono essere attivi nel sistema operativo. Se ciò si verifica, il codice del kernel, che implementa il sistema operativo, è soggetto a molte possibili race condition. Si consideri per esempio una struttura dati del kernel che mantenga una lista di tutti i file aperti nel sistema.

Tale lista dev'essere modificata quando un nuovo file è aperto, e quindi aggiunto all'elenco, oppure chiuso, quindi tolto dall'elenco. Se due o più processi dovessero aprire dei file simultaneamente, potrebbero generare nel sistema una race condition legata ai necessari aggiornamenti della lista.

Altre strutture dati del kernel soggette a problemi analoghi sono quelle per l'allocazione della memoria, per la gestione delle interruzioni e le liste dei processi. La responsabilità di preservare il sistema operativo da simili problemi compete a chi sviluppa il kernel.

Il problema della sezione critica può essere risolto in maniera semplice in un ambiente single-core, impedendo che si verifichino interruzioni durante la modifica di una variabile condivisa. In questo modo, saremmo sicuri che l'attuale sequenza di istruzioni sia eseguita in ordine, senza prelazione. Visto che non verranno eseguite altre istruzioni sarà impossibile apportare modifiche inaspettate a una variabile condivisa. Sfortunatamente questa soluzione non è praticabile in un ambiente multiprocessore. Disabilitare gli interrupt su un multiprocessore può richiedere infatti molto tempo, poiché il messaggio viene passato a tutti i processori. Questo invio di messaggi ritarda l'ingresso in ciascuna sezione critica e l'efficienza del sistema diminuisce. Va inoltre tenuto in considerazione l'effetto sul clock di sistema nel caso in cui il clock venga aggiornato dalle interruzioni.

Le due strategie principali per la gestione delle sezioni critiche nei sistemi operativi sono:

- **kernel con diritto di prelazione**: consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione.
- **kernel senza diritto di prelazione**: non consente di applicare la prelazione a un processo attivo in modalità di sistema -> l'esecuzione di questo processo seguirà finché lo stesso esca da tale modalità, si sblocchi o ceda volontariamente il controllo della CPU.

I kernel senza diritto di prelazione sono immuni da race condition sulle strutture dati del kernel, visto che un solo processo per volta impegnava il kernel. Altrettanto non si può dire dei kernel con diritto di prelazione, motivo per cui bisogna avere cura, nella progettazione, di mantenerli al riparo dai problemi nell'accesso alle strutture dati del kernel. I kernel con diritto di prelazione presentano particolari difficoltà di progettazione quando sono destinati ad architetture SMP (multielaborazione simmetrica), poiché in tali ambienti due processi nella modalità di sistema possono essere eseguiti in contemporanea su core differenti.

Perché, allora, i kernel con diritto di prelazione dovrebbero essere preferiti a quelli senza diritto di prelazione?

I kernel con diritto di prelazione possono vantare una maggiore prontezza nelle risposte, grazie al basso rischio di eseguire processi in modalità di sistema per un tempo eccessivamente lungo, prima di liberare la CPU per i processi in attesa.

Inoltre, i kernel con diritto di prelazione sono più adatti alla programmazione real-time, dal momento che permettono ai processi in tempo reale di effettuare la prelazione di un processo attivo nel kernel.

## Soluzione di Peterson

Implementata via software. Funziona solo su due processi e sui vecchi processori.

A causa del modo in cui i moderni elaboratori eseguono le istruzioni elementari del linguaggio macchina, quali load e store, non è affatto certo che la soluzione di Peterson funzioni correttamente su tali sistemi.

La soluzione di Peterson è limitata a due processi, P0 e P1, ognuno dei quali esegue alternativamente la propria sezione critica e la sezione non critica.

Supponiamo che le operazioni di load e store in memoria siano **atomiche**: nella stragrande maggioranza dei processori è così.

Essa richiede che i processi condividano i seguenti dati:

```

int turn; -> segnala di chi sia il turno d'accesso alla sezione critica
boolean flag[2]; -> desiderio di entrare di un processo (indica se un processo è pronto a entrare nella sez. cr.)

```

Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a turno sia il valore i che il valore j. Soltanto uno dei due permane: l'altro sarà immediatamente sovrascritto. Il valore definitivo di turn stabilisce quale dei due processori sia autorizzato a entrare per primo nella propria sezione critica.

```

while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;
    /* critical section */
    flag[i] = false;
    /* remainder section */
}

```

Dimostriamo ora la correttezza di questa soluzione.

Dobbiamo provare che:

1. la mutua esclusione sia preservata
2. il requisito del progresso sia soddisfatto
3. il requisito dell'attesa limitata sia rispettato
4. La mutua esclusione è garantita dal sistema tie-breaker. Pi entra nella sezione critica solo se flag[j]=false o turn=i. Il flag turn può avere come valore solo i o j in un momento. Il processo con turn coerente eseguirà il while, mentre l'altro aveva da eseguire un'istruzione aggiuntiva (turn == j). Poiché in quel momento e fino al termine della permanenza di Pj nella propria sezione critica restano valide le asserzioni flag[j] == true e turn == j, la mutua esclusione è preservata.
5. Il progresso è garantito dalla variabile flag: qualora il processo Pj non sia pronto a entrare nella sezione critica, flag[j] == false e Pi può accedere alla propria sezione critica. Nel momento di uscita dalla propria sezione critica, Pi porrà flag[i] a false e Pj potrà entrare nella sua sezione critica.
6. In condizioni in cui i processi tentano di accedere a raffica, la parte del codice iniziale garantisce che uno conceda l'accesso all'altro. Pi non modifica il valore di turn durante l'esecuzione dell'istruzione while, quindi entrerà nella sezione critica dopo che Pj abbia effettuato non più di un ingresso.

Come si fa a estendere questa soluzione a più di due processi? Dovrei creare una tabella per tenere traccia dei turni, ma questa soluzione va a minare la mutua esclusione perché non utilizzo una singola variabile booleana.

Non è possibile garantire il funzionamento della soluzione di Peterson su architetture moderne, principalmente perché per migliorare le prestazioni del sistema i processori e/o i compilatori possono riordinare le operazioni di lettura e scrittura che non hanno dipendenze.

Per un'applicazione con un singolo thread questo riordino è irrilevante per quanto riguarda la correttezza del programma, in quanto i valori finali rimangono coerenti con quanto previsto, ma per un'applicazione multi-thread con dati condivisi il riordino delle istruzioni può portare a risultati incoerenti o inattesi.

Esempio:

■ Two threads share the data:

```

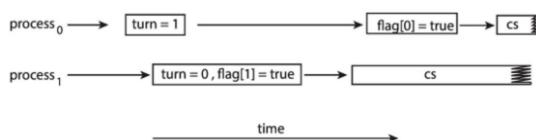
boolean flag = false;
int x = 0;
■ Thread 1 performs
while (!flag)
    ;
print x
■ Thread 2 performs
x = 100;
flag = true

```

Il comportamento previsto è che Thread 1 restituisca il valore 100 per la variabile x. Tuttavia, poiché non ci sono dipendenze tra le variabili flag e x, è possibile che i riordini delle istruzioni di Thread 2 porti l'istruzione flag = true ad essere eseguita prima dell'assegnazione x = 100. In questa situazione, Thread 1 potrebbe restituire 0 come valore di x.

Il processore può anche ordinare le istruzioni di Thread 1 e caricare la variabile 0 anche se le istruzioni eseguite da Thread 2 non fossero state ordinate.

Effetto sulla soluzione di Peterson:



In seguito al riordino delle istruzioni, è possibile avere entrambi i Thread attivi nelle loro sezioni critiche contemporaneamente: servono quindi strumenti di sincronizzazione adeguati.

## Supporto hardware per la sincronizzazione

Nel caso della soluzione di Peterson parliamo di soluzione basata sul software, perché l'algoritmo garantisce la mutua esclusione senza richiedere alcun supporto speciale dal sistema operativo né istruzioni hardware specifiche. Tuttavia, le soluzioni basate sul software non garantiscono il loro funzionamento su architetture elaborate moderne.

## Istruzioni hardware

Molte moderne architetture offrono *particolari istruzioni che permettono di controllare e modificare il contenuto di una parola in memoria, oppure di scambiare il contenuto di due parole di memoria*, in modo atomico, cioè come unità non interrompibile.

Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice.

Anziché discutere una specifica istruzione di una particolare architettura è preferibile astrarre i concetti principali descrivendo le istruzioni *test\_and\_set()* e *compare\_and\_swap()*.

L'istruzione *test\_and\_set()* è così definibile:

```
boolean test_and_set(boolean *obiettivo){  
    boolean valore = *obiettivo  
    *obiettivo = true;  
    return valore;  
}
```

La caratteristica fondamentale di questa istruzione è che viene eseguita *atomicamente*; quindi, *se si eseguono contemporaneamente due istruzioni test\_and\_set(), ciascuna in un'unità di elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario*.

Se si dispone dell'istruzione *test\_and\_set()*, si può realizzare la mutua esclusione dichiarando una variabile booleana globale lock, inizializzata a false:

```
do{  
    while(test_and_set(&lock))  
        ; -> do nothing  
        / sezione critica /  
    lock = false;  
        / sezione non critica /  
} while(true);
```

Anche se ho 100 processi che eseguono la *test\_and\_set()*, solo uno avrà il valore iniziale di lock a false,

Il problema è che, senza sapere com'è fatto lo scheduling, non ho l'attesa limitata assicurata perché non ho nessuna garanzia sulla velocità di esecuzione dei processi.

Posso garantire l'attesa limitata indipendentemente dallo scheduling utilizzando la *compare\_and\_swap()*.

L'istruzione *compare\_and\_swap()* (CAS), proprio come l'istruzione *test\_and\_set()*, opera su due parole atomicamente, ma utilizza un meccanismo diverso che si basa sullo scambio del contenuto delle parole.

Quest'istruzione utilizza tre operandi:

```
int compare_and_swap(int *value, int expected, int new_value){  
    int temp = *value;  
    if(*value == expected)  
        *value = new_value;  
    return temp;  
}
```

L'operando *value* viene impostato a *new\_value* solo se l'espressione (*value == expected*) è vera. A parte in questo caso, la *compare\_and\_swap()* restituisce sempre il valore originale della variabile *value*.

L'importante caratteristica di questa istruzione è che viene eseguita atomicamente. Pertanto, se due istruzioni CAS vengono eseguite simultaneamente (ciascuna su un core diverso), verranno eseguite sequenzialmente in un ordine arbitrario\*.

La mutua esclusione può essere realizzata usando CAS come segue:

```
while (true){  
    while(compare_and_swap(&lock, 0, 1) != 0)  
        ; -> non fare niente  
        / sezione critica /  
    lock = 0;  
        / sezione non critica /  
}
```

Venne dichiarata e inizializzata a 0 una variabile globale (lock). Il primo processo che richiama *compare\_and\_swap()* imposterà lock a 1. Entrerà poi nella sua sezione critica, poiché il valore originale di lock era pari al valore atteso 0. Le chiamate successive di *compare\_and\_swap()* non avranno successo, perché ora lock non è uguale al valore atteso 0. Quando un processo esce dalla sua sezione critica, imposta di nuovo il lock al valore 0, per permettere ad un altro processo di entrare nella propria sezione critica.

Questo algoritmo soddisfa il requisito della mutua esclusione, ma non quello dell'attesa limitata, poiché non conosciamo la velocità dei processi.

Altro algoritmo che sfrutta l'istruzione *compare\_and\_swap()* che soddisfa tutti e tre i requisiti desiderati:

le strutture dati condivise sono:

```

boolean waiting[n];
boolean lock;

```

La variabile `waiting[i]` diventa false solo se un altro processo esce dalla sua attesa critica; solo una variabile di `waiting[]` vale false, così da rispettare la mutua esclusione.

Il progresso è rispettato poiché un processo che esce dalla sezione critica imposta `lock` al valore false oppure `waiting[i] == false`, consentendo ad un processo in attesa di entrare nella sua sezione critica.

Il RISC-V fa un unico accesso in memoria dati per ogni istruzione, e visto che non prevede lettura e scrittura in memoria nella stessa riga di codice è complicato implementare le istruzioni atomiche.

## Sincronizzazione in RISC-V

Vediamo come si fa la sincronizzazione in RISC-V:

due processi condividono un'area di memoria -> P1 scrive e P2 legge.

E' richiesto un supporto hardware per evitare il problema della race condition: serve quindi un'operazione di lettura/scrittura atomica.

Inoltre, non devono essere permessi accessi alla location tra read e write.

Può essere una singola istruzione: atomic swap register <-> memoria.

Due istruzioni a nostra disposizione:

- *Load Reserver*: `l.d rd, (rs1)` (la d sta per double word, senza .d carico una word).
  - Carico dall'indirizzo in rs1 a rd
  - E' reserved perché il processore tiene traccia dell'indirizzo di memoria. Ripeto il load finché non scrivo prima che un altro processo lo faccia prima di me
- *Store Conditional*: `sc.d rd, (rs1), rs2`
  - Scrive il contenuto di rs2 in rs1
  - Rispetto allo store normale ho un terzo registro che mi dice se l'operazione è andata a buon fine o meno (che vuol dire che ho scritto prima di un altro processo)
  - Se l'operazione è andata a buon fine, la location non è cambiata dal ld.r, rd = 0. Se la location è cambiata, rd è diverso da 0

```

■ Example 1: atomic swap (to test/set lock variable)
again: l.r.d x10,(x20)
      sc.d x11,(x20),x23 // x11 = status
      bne x11,x0,again // branch if store failed
      addi x23,x10,0      // x23 = loaded value

■ Example 2: lock
      addi x12,x0,1      // copy locked value
again: l.r.d x10,(x20)    // read lock
      bne x10,x0,again   // check if it is 0 yet
      sc.d x11,(x20),x12 // attempt to store
      bne x11,x0,again   // branch if fails
■ Unlock:
      sd x0,0,(x20)      // free lock

```

Esempio 1: il bne mi permette di ripetere l'operazione finché lo store ha successo.

*L'interesse dei sistemi operativi è spostarsi verso istruzioni che fanno la mutua esclusione in spazio utente per evitare syscall pesanti.*  
I futex di Linux sono uno di questi esempi.

## Lock mutex

Le soluzioni hardware presentate precedentemente (`test_and_set()` e `cmpare_and_swap()`) sono complicate e generalmente inaccessibili ai programmati di applicazioni. In alternativa, i progettisti di sistemi operativi implementano strumenti software per risolvere lo stesso problema.

Il più semplice di questi strumenti è il lock mutex (mutex -> mutual exclusion). Usiamo il lock mutex per *proteggere le regioni critiche e quindi prevenire le race condition*.

*In pratica, un processo deve acquisire il lock prima di entrare in una sezione critica e rilasciarlo quando esce dalla sezione critica.*

La funzione `acquire()` acquisisce il lock e la funzione `release()` lo rilascia:

```

while (true) {
  acquire lock
  critical section
  release lock
}
remainder section

```

Un lock mutex ha una variabile booleana `available`, il cui valore *indica se il lock è disponibile o meno*. Se il lock è disponibile, la chiamata `acquire()` ha successo e il lock viene da questo momento considerato non disponibile.

Un processo che tenta di acquisire un lock indisponibile viene bloccato fino al rilascio del lock.

Definizione della funzione `acquire()`:

```

acquire(){
  while(!available)
    ; / attesa attiva /

```

```

    available = false;
}

```

La definizione di release() è la seguente:

```

release(){
    available = true;
}

```

Le syscall acquire() e release() devono essere eseguite in modo atomico. Posso usare test\_and\_set() e compare\_and\_swap() per implementarle. Il principale svantaggio dell'implementazione che abbiamo fornito è che richiede *attesa attiva (busy waiting)*: mentre un processo si trova nella sua sezione critica, ogni altro processo che cerchi di entrare nella sezione critica deve ciclare continuamente effettuando la chiamata acquire(). Questo continuo ciclare è chiaramente un problema in un sistema multiprogrammato in cui un singolo core della CPU è condiviso tra molti processi. L'attesa attiva spreca inoltre cicli di CPU che altri processi potrebbero essere in grado di utilizzare in modo produttivo.

Il tipo di lock mutex che abbiamo descritto è anche chiamato **spinlock** perché il processo continua a girare in attesa che il lock diventi disponibile. Tuttavia, gli spinlock hanno il vantaggio di non rendere necessario alcun cambio di contesto (operazione che può richiedere molto tempo) quando un processo deve attendere un lock e tornano quindi utili quando si prevede che i lock verranno trattenuti per tempi brevi. Gli spinlock sono spesso impiegati in sistemi multiprocessore in cui un thread può girare su un processore, mentre un altro thread esegue la sua sezione critica su un altro processore.

## Semafori

I semafori sono presenti in qualsiasi sistema operativo moderno, ma non in quelli real-time.

I semafori sono uno strumento di sincronizzazione a più alto livello, *utilizzati per permettere al programmatore di scrivere del codice da eseguire potenzialmente in parallelo*. Il semaforo serve anche per altri problemi di sincronizzazione, non solo per quello della mutua esclusione. L'altro problema può essere la sincronizzazione su condizione: devo aspettare finché una condizione non mi abilita a fare qualcosa (es. scrittura produttore/consumatore).

*Un semaforo S è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: wait() e signal().*

Queste operazioni erano originariamente chiamate P (wait()) e V (signal()).

```

wait(S){
    while(S <= 0)
        ; / attesa attiva (o metodo alternativo che sospende il while)
    S--;
}

signal(S){
    S++;
}

```

Tutte le modifiche al valore del semaforo contenute nelle operazioni wait() e signal() si devono eseguire in modo indivisibile: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore.

Inoltre, nel caso della wait(S) si devono eseguire senza interruzione anche la verifica del valore intero di S ( $S \leq 0$ ) e la sua possibile modifica ( $S--$ ).

## Uso dei semafori

Esistono due tipi di semafori:

- **Semaforo contatore**: il suo valore è un numero intero
- **Semaforo binario**: il suo valore è limitato a 0 o 1 -> simili ai lock mutex e vengono utilizzati al loro posto per la mutua esclusione nei sistemi dove i lock mutex non sono disponibili

I semafori contatore trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di risorse disponibili. I processi che desiderino utilizzare una risorsa invocano wait() sul semaforo, decrementandone così il valore; i processi che restituiscono una risorsa, invece, invocano signal() sul semaforo, incrementandone il valore. Quando il semaforo vale 0, tutte le risorse sono occupate e i processi che ne richiedono l'uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

I semafori sono utilizzabili anche per risolvere diversi problemi di sincronizzazione. Si considerino, per esempio, due processi in esecuzione concorrente: P1 con un'istruzione S1 e P2 con un'istruzione S2. Si supponga di voler eseguire S2 solo dopo che S1 è terminata. Questo schema si può prontamente realizzare facendo condividere a P1 e P2 un semaforo comune, synch, inizializzato a 0, e inserendo nel processo P1 le istruzioni:

```

S1;
signal(synch);

```

e nel processo P2 le istruzioni:

```
wait(synch);
S2;
```

Poiché synch è inizializzato a 0, P2 esegue S2 solo dopo che P1 ha eseguito signal(synch), che si trova dopo S1.

## Implementazione dei semafori

Le definizioni delle operazioni sui semafori wait() e signal() presentano il problema dell'attesa attiva (vedi lock mutex).

Per superare la necessità dell'attesa attiva si possono modificare le definizioni delle operazioni wait() e signal() come segue: quando un processo invoca l'operazione wait() e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell'attesa attiva può bloccare se stesso. L'operazione di bloccaggio pone il processo in una coda d'attesa associata al semaforo e pone lo stato del processo a waiting. Quindi, si trasferisce il controllo allo scheduler della CPU che sceglie un altro processo pronto per l'esecuzione.

Un processo sospeso, che attende un semaforo S, sarà riavviato in seguito all'esecuzione di un'operazione signal() su S da parte di qualche altro processo. Il processo si riavvia tramite un'operazione wakeup(), che *modifica lo stato del processo da waiting a ready*. Il processo entra nella coda dei processi pronti (l'uso della CPU potrebbe essere o meno commutato dal processo in esecuzione al processo appena divenuto pronto, a seconda dell'algoritmo di scheduling).

Si può definire il semaforo così come segue:

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
```

A ogni semaforo sono associati un valore intero (value) e una lista di processo (list) contenente i processi in attesa a un semaforo; l'operazione signal() preleva un processo da tale lista e lo attiva.

L'operazione wait() del semaforo si può definire come segue:

```
wait(semaphore *S){
    S->value--;
    if(S->value < 0){
        aggiungi questo processo a S->list
        sleep();
    }
}
```

L'operazione signal() del semaforo si può definire come segue:

```
signal(semaphore *S){
    S->value++;
    if(S->value <= 0){
        togli un processo P da s->list;
        wakeup(P);
    }
}
```

L'operazione sleep() sospende il processo che la invoca; l'operazione wakeup(P) pone in stato di pronto per l'esecuzione un processo P bloccato. Queste due operazioni sono fornite dal sistema operativo come syscall.

Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, questa definizione può condurre a valori negativi. Se il valore del semaforo è negativo, questa definizione può condurre a valori negativi. Se il valore del semaforo è negativo, il suo valore assoluto rappresenta il numero di processi che attendono quel semaforo:  
es.: se il semaforo vale -3 ci sono 3 processi in attesa.

Ciò è dovuto alla inversione dell'ordine del decremento e dalla verifica nel codice dell'operazione wait().

La lista dei processi che attendono a un semaforo si può facilmente realizzare con un campo puntatore in ciascun blocco di controllo del processo (PCB). Ogni semaforo contiene un valore intero e un puntatore a una lista di PCB. Un modo per aggiungere e togliere processi dalla lista assicurando un'attesa limitata è usare una coda FIFO, della quale il semaforo contiene i puntatori al primo e all'ultimo elemento. In generale, si può usare però qualsiasi criterio di accodamento; il corretto uso dei semafori non dipende dal particolare criterio adottato.

Uso incorreto dei semafori: il processo non esegue wait(mutex) prima di entrare nella sezione critica o signal(mutex) prima di uscirne.

Se questa sequenza non è rispettata, può accadere che i due processi occupino simultaneamente le rispettive sezioni critiche, dato che non ci saranno meccanismi all'inizio o alla fine per delimitarla.

*Le operazioni dei semafori devono essere eseguite in modo atomico!*

I sistemi SMP, per garantire l'atomicità delle operazioni, devono mettere a disposizione altre tecniche di realizzazione dei lock (spinlock, compare\_and\_swap()). Questa definizione delle operazioni wait() e signal() non consente di eliminare completamente l'attesa attiva, ma piuttosto di rimuoverla dalle sezioni d'ingresso dei programmi applicativi.

Altro problema: se ho due processi e fanno wait() contemporaneamente o se omettono wait e/o signal -> può portare ad un blocco di un processo. In tutti questi casi c'è una violazione della mutua esclusione, oppure ci può essere un blocco indefinito del processo se si usa wait() sia all'inizio che alla fine.

Inoltre, sul semaforo non c'è nessun controllo su chi rientra. Nel real time vorrei avere un meccanismo in cui il processo con più alta priorità entra per primo, nel semaforo non posso averlo. Quindi i semafori non vengono utilizzati in ambito real-time.

Queste problematiche sono il motivo per cui esistono altri strumenti per la sincronizzazione più facili da implementare, come i Monitor.

## Monitor

Sono un'astrazione ad alto livello che provvede un meccanismo conveniente ed efficace per processare la sincronizzazione.

Un tipo di dato astratto (ADT) incapsula i dati mettendo a disposizione un insieme di funzioni per operare su di essi; tali funzioni sono indipendenti dalla specifica implementazione del tipo di dato -> sono gli antenati degli oggetti.

Il monitor è un ADT che comprende un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione.

Il tipo monitor contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza del tipo, oltre al corpo delle procedure o funzioni che operano su tali variabili.

Sintassi di un monitor:

```
monitor monitor name{
    function P1(...){...}
    function P2(...){...}
    ...
    function Pn(...){...}
    initialization_code(...){...}
}
```

La rappresentazione di un tipo monitor non può essere usata direttamente dai vari processi. Pertanto, una funzione definita all'interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, situate nel monitor, e ai relativi parametri formali. In modo analogo, alle variabili locali di un monitor possono accedere solo le procedure locali.

Il costrutto monitor assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, sicché non si deve codificare esplicitamente il vincolo di mutua esclusione.

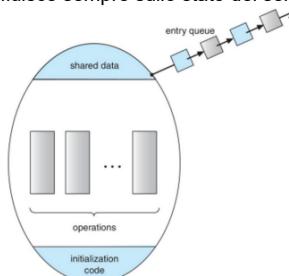
Un programmatore che necessita di implementare un proprio particolare schema di sincronizzazione può definire una o più variabili di tipo condition (variabili interne al monitor, non visibili all'esterno):

```
condition x, y;
```

Le uniche operazioni eseguibili su una variabile condition sono wait() e signal():

l'operazione x.wait() implica che il processo che la invoca rimanga sospeso finché un altro processo non invochi l'operazione x.signal() che risveglia (tecnica FIFO al risveglio dei processi) esattamente un processo sospeso.

Se non esistono processi sospesi l'operazione signal() non ha alcun effetto, vale a dire che lo stato di x resta immutato. Ciò contrasta con l'operazione signal() associata ai semafori poiché questa influisce sempre sullo stato del semaforo.



Buffer condiviso: lo metto dentro ad un monitor e uso le funzioni per accedere alle sue variabili -> lavorerò con il buffer accedendo uno alla volta.

Problemi: quando voglio accedere all'area di memoria condivisa con più processi... ma perché dovrei farlo? Si pensi se più processi vogliono semplicemente leggere i dati nella memoria condivisa: questo non si può fare con i monitor anche se è un'operazione innocua.

Se P invoca x.signal() e il processo Q è sospeso in x.wait() cosa succede?

Non posso eseguire P e Q contemporaneamente in parallelo. Se Q è eseguito, P deve aspettare e viceversa.

Concettualmente, però, entrambi i processi possono continuare l'esecuzione.

Sussistono due possibilità:

1. *Segnalare e attendere*: P attende che Q lasci il monitor o attenda su un'altra variabile condition;
2. *Segnalare e proseguire*: Q attende che P lasci il monitor o attenda su un'altra variabile condition (if sostituiti da while dove ci sono le wait())

Compromesso nel linguaggio Current Pascal: quando il thread P esegue l'operazione signal() lascia subito il monitor; pertanto, Q riprende immediatamente l'esecuzione.

## Ripresa dei processi all'interno di un monitor

Se più processi sono sospesi alla condizione x, e se qualche processo esegue l'operazione x.signal(), è necessario stabilire quale tra i processi sospesi si debba riattivare per primo.

Una semplice soluzione consiste nell'utilizzare un ordinamento FCFS (first come, first served), secondo cui il processo che attende da più tempo viene ripreso per primo.

Tuttavia, in molti casi uno schema di scheduling di questo tipo non risulta adeguato; in questi casi si può usare un costrutto di attesa condizionale della forma x.wait(x), dove con c si indica un'espressione intera che si valuta al momento dell'esecuzione dell'operazione wait(). Il valore di c, chiamato *numero di priorità*, viene poi memorizzato col nome del processo sospeso. Quando si esegue x.signal(), si riprende il processo cui è associato il numero di priorità più basso.

Il concetto di monitor, però, non può garantire che una sequenza di accesso sia rispettata. In particolare, può accadere quanto segue:

- un processo può accedere alla risorsa senza prima ottenere il permesso d'accesso;
- una volta che ne ha ottenuto l'accesso, un processo può non rilasciare più la risorsa;
- un processo può tentare di rilasciare una risorsa che non ha mai richiesto;
- un processo può richiedere due volte la stessa risorsa, senza rilasciarla prima della seconda richiesta

Possibile soluzione: inclusione delle operazioni d'accesso alle risorse all'interno del monitor *assegnazione\_risorse*.

Adottando questa soluzione, tuttavia, si userebbe l'algoritmo di base del monitor per schedulare le risorse invece che uno codificato da noi.

Per garantire che i processi rispettino le sequenze appropriate è necessario controllare tutti i programmi che usano il monitor *assegnazione\_risorse* e la risorsa da esso gestita.

Devo verificare due condizioni: la prima, che i processi utente devono sempre impiegare il monitor secondo una sequenza corretta; la seconda, che è necessario assicurare che un processo non cooperativo non cerchi di aggirare la mutua esclusione offerta dal monitor e tenti di accedere direttamente alla risorsa condivisa senza usare i protocolli di accesso. Soltanto se si rispettano queste due condizioni si può garantire l'assenza di errori di sincronizzazione e che l'algoritmo di scheduling sia rispettato.

Questo controllo è possibile per sistemi statici di piccole dimensioni, mentre non è ragionevolmente applicabile a sistemi di grandi dimensioni o sistemi dinamici.

## Cos'è una race condition?

Sono situazioni in cui più processi accedono e modificano gli stessi dati in modo concorrente. Per evitare ciò è richiesta una forma di sincronizzazione dei processi. Si verificano spesso in sistemi operativi dove i diversi componenti del sistema accedono a risorse condivise.

## Cos'è una sezione critica e come si risolve?

La sezione critica è un segmento di codice appartenente ad un processo, nel quale è possibile modificare delle variabili da parte di più processi. Quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica.

Il problema della sezione critica consiste nel progettare un protocollo che i processi possono usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. Quindi, l'esecuzione delle sezioni critiche da parte dei processi è mutuamente esclusiva nel tempo. La sezione di codice che realizza questa richiesta è la sezione di ingresso. La sezione critica può essere eseguita da una sezione d'uscita e la restante parte del codice è detta sezione non critica.

Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti:

1. *Mutua esclusione*: se il processo P1 è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. *Progresso*: se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica.
3. *Attesa limitata*: se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

Le due strategie principali per la gestione delle sezioni critiche nei sistemi operativi prevedono l'impiego di: kernel con diritto di prelazione e kernel senza diritto di prelazione. Un kernel con diritto di prelazione consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. Un kernel senza diritto di prelazione non consente di applicare la prelazione a un processo attivo in modalità di sistema.

In sostanza, i kernel senza diritto di prelazione sono immuni dai problemi legati all'ordine degli accessi alle strutture dati del kernel, visto che un solo processo per volta impegna il kernel.

Perché, allora, i kernel con diritto di prelazione dovrebbero essere preferiti a quelli senza diritto di prelazione? I kernel con diritto di prelazione sono più adatti alla programmazione real-time, dal momento che permettono ai processi in tempo reale di far valere il loro diritto di precedenza nei confronti di un processo attivo nel kernel. Inoltre, i kernel con diritto di prelazione possono vantare una maggiore potenza nelle risposte.

## Cos'è l'algoritmo di Peterson?

Esso rappresenta un buon algoritmo per il problema della sezione critica che soddisfino i tre requisiti di mutua esclusione, progresso e attesa limitata. La soluzione di Peterson si applica a due processi. Questa prevede che i due processi condividono due dati:

1. una variabile turno che indica di chi è il turno per entrare alla sezione critica

## 2. un'area flag con la quale si manifesta la volontà di un processo di entrare nella sezione critica

L'algoritmo di Peterson garantisce tutti e tre i requisiti. La mutua esclusione è dimostrata perché se entrambi i processi sono pronti, quindi  $\text{flag}[0]=\text{flag}[1]=\text{true}$ , la variabile turno può essere 1 o 0 ma non entrambi. Gli altri due aspetti sono dimostrati perché all'uscita dalla sezione critica il turno viene impostato all'altro processo e in questo modo si ha progresso e attesa limitata perché il processo precedente non ha effettuato più di un ingresso.

Per accedere alla sezione critica, il processo assegna innanzitutto a  $\text{flag}[i]$  il valore true; quindi attribuisce a turno il valore j, conferendo così all'altro processo la facoltà di entrare nella sezione critica. Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a turno sia il valore i che j: soltanto uno dei due permane e l'altro sarà immediatamente sovrascritto.

Altra risposta:

Funziona solo con due processi in contemporanea. Implementa le variabili int turn e boolean flag[2] : con queste riesce a non fare entrare due processi, che in contemporanea fanno richiesta, nella loro sezione critica.

## Cos'è un test\_and\_set()? Perché non si utilizza? Il RISC-V ha un'istruzione simile?

Molte delle moderne architetture offrono particolari istruzioni che consentono di controllare e modificare il contenuto di una parola di memoria, oppure scambiare il contenuto di due parole in memoria, in modo atomico e cioè senza interruzioni. Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice.

L'istruzione test\_and\_set() è eseguita atomicamente, cioè come un'unità non soggetta a interruzioni; quindi se si eseguono contemporaneamente due istruzioni test\_and\_set(), ciascuna in due unità di elaborazione, queste vengono eseguite in modo sequenziale in un ordine arbitrario.

Sì, in RISC-V ci sono due istruzioni simili alla test\_and\_set() che vengono eseguite in modo atomico e sono load linked e store condition: servono rispettivamente a prelevare e caricare una word dalla/nella memoria in maniera atomica.

Altra risposta:

Il RISC-V fa un unico accesso alla memoria e visto che non prevede lettura e scrittura in memoria allo stesso rigo di codice è complicato implementare istruzioni atomiche. Come si fa?

Due processi: P1 legge P2 scrive. E' richiesto un supporto hardware per evitare il problema del data race. Abbiamo due istruzioni a disposizione → lr.d rd,(rs1) Carico dall'indirizzo in rs1 a rd è reserved perchè il processore tiene traccia dell'indirizzo in memoria. Ripeto il load finchè non scrivo prima che un altro processo lo faccia → sc.d rd,(rs1),rs2 Scrive il contenuto di rs2 in rs1 , Rispetto allo store normale ho un terzo registro che mi dice se l'operazione è andata a buon fine o no . Se l'operazione è andata a buon fine , la location non è cambiata dal ld.r,rd=0 se la location è cambiata rd è diverso da 0.

## Cos'è un semaforo?

Un semaforo S è una variabile intera cui si può accedere escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: wait() e signal(). Tutte le modifiche al valore del semaforo contenute nelle operazioni wait() e signal() si devono eseguire in modo invisibile: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Servono a risolvere i problemi delle soluzioni critiche.

Altra risposta:

E' uno strumento di sincronizzazione più ad alto livello per permettere al programmatore di scrivere del codice da eseguire potenzialmente in parallelo. Il semaforo serve anche per altri problemi di sincronizzazione oltre che per la mutua esclusione come la sincronizzazione su condizione. Il semaforo risolve entrambi i problemi, Definizione semaforo variabile S. – Due operazioni atomiche → P() e V(). Struttura di P() e V() – Funzionamento – Mutua esclusione sempre verificata – Tipologia semafori – Implementazione dei semafori con e senza busy wait – Operazioni Block e Wakeup – Implementazione non canonica numeri negativi.

## Come si realizza un semaforo?

Per superare la necessità dell'attesa attiva, si possono modificare le definizioni delle operazioni wait() e signal(): quando un processo invoca l'operazione wait() e trova che il valore del semaforo non è positivo deve attendere, ma anziché restare nell'attesa attiva può bloccare se stesso. L'operazione di bloccaggio pone il processo in una coda d'attesa associata al semaforo e cambia lo stato del processo nello stato d'attesa. Quindi si trasferisce il controllo allo scheduler della CPU che sceglie un altro processo pronto per l'esecuzione. Il processo si riavvia tramite un'operazione di wakeup(), che modifica lo stato del processo da attesa a pronto. Il processo entra nella coda dei processi pronti. A ogni semaforo sono associati un valore intero e una lista di processi, contenente i processi in attesa a un semaforo; l'operazione signal() preleva un processo da tale lista e lo attiva. L'operazione sleep() sospende il processo che lo invoca; l'operazione wakeup(P) pone in stato di pronto per l'esecuzione un processo P bloccato. Queste due operazioni sono fornite dal SO come chiamate di sistema di base. I semafori devono essere eseguiti in modo atomico. Si deve garantire che nessuno dei due processi possa eseguire operazioni wait() e signal() contemporaneamente sullo stesso semaforo. In un contesto monoprocesso lo si può risolvere inibendo le interruzioni durante l'esecuzione di signal() e wait(). Nei sistemi multiprocessore disabilitare le interruzioni di tutti i processori può non essere cosa semplice e causare un notevole calo delle prestazioni. È per questo che -per garantire l'esecuzione atomica di wait() e signal() - i sistemi SMP devono mettere a disposizione altre tecniche di realizzazione dei lock (per esempio, gli spinlock). L'attesa attiva si limita alle sezioni critiche delle operazioni wait() e signal(), che sono abbastanza brevi.

## Che cosa significa in maniera atomica?

Eseguire istruzioni senza interruzioni.

## Un semaforo può avere valore 10?

Sì, se non è un semaforo binario o un lock mutex.

## Cos'è un monitor? Come garantisce la mutua esclusione?

Scopo del monitor è controllare l'accesso a una risorsa da parte di processi concorrenti in accordo a determinate politiche. Le variabili locali definiscono lo stato della risorsa associata al monitor. L'accesso alla risorsa avviene secondo due livelli di controllo:

1. Il primo garantisce che un solo processo alla volta possa aver accesso alle variabili comuni del monitor. Ciò è ottenuto garantendo che le operazioni public siano eseguite in modo mutuamente esclusivo (eventuale sospensione dei processi nella entry queue).
2. Il secondo controlla l'ordine con il quale i processi hanno accesso alla risorsa. La procedura chiamata verifica il soddisfacimento di una condizione logica (condizione di sincronizzazione) che assicura l'ordinamento (eventuale sospensione del processo in una coda associata alla condizione e liberazione del monitor).

Il costrutto monitor assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, sicché non si deve codificare esplicitamente il vincolo di mutua esclusione.

Altra risposta:

Sono un'astrazione ad alto livello che prevede un meccanismo per la sincronizzazione. Tipo di dato astratto (ADT), incapsula dei dati nelle variabili interne che sono accessibili solo dal codice all'interno della procedura. La caratteristica del monitor è che le operazioni definite al suo interno sono caratterizzate dalla mutua esclusione. Il tipo monitor contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza di tipo. Solo un processo alla volta può essere attivo dentro ad un monitor. Buffer condiviso → viene posizionato dentro al monitor e uso le funzioni per accedere alle sue variabili. Entry queue - Sincronizzazione su condizione – Variable condition – x.wait x.signal – Differenza con semaforo – Signal and wait & Signal and continue(JAVA).

## Cos'è un lock mutex?

È uno strumento software che viene utilizzato per proteggere le regioni critiche e quindi prevenire le Race condition. In pratica un processo deve acquisire il lock prima di entrare in una sezione critica e rilasciarlo quando esce dalla sezione critica. Un processo che tenta di acquisire un lock indisponibile viene bloccato fino al rilascio del lock. Il principale svantaggio dell'implementazione che abbiamo fornito e che richiede attesa attiva. L'attesa attiva spreca cicli di clock più che altro i processi potrebbero essere in grado di utilizzare in modo produttivo. Il tipo di lock mutex più comune è lo spin lock chiamato così perché il processo continua a girare in attesa che il lock diventi disponibile. Inoltre, sono spesso utilizzati su sistemi multiprocessore in cui un thread può girare su un processore mentre un altro thread esegue la sua sezione critica su un altro processore.

## Come implementare le condizioni atomiche?

Le condizioni atomiche possono essere implementate in diversi modi:

- Locks: una delle implementazioni più semplici delle condizioni atomiche è utilizzare un lock. Un lock è un meccanismo di sincronizzazione che permette a un solo thread alla volta di accedere a una sezione critica di codice. Quando un thread richiede l'accesso a una sezione critica protetta da un lock, il thread deve acquisire il lock prima di accedere a quella sezione.
- Semafori: un semaforo è un meccanismo di sincronizzazione che permette a più thread di accedere a una sezione critica contemporaneamente, ma con un limite sul numero di thread che possono accedere allo stesso tempo. Questo meccanismo è utile per implementare le condizioni atomiche, poiché permette di limitare il numero di thread che possono accedere a una sezione critica allo stesso tempo.
- Monitor: un monitor è un meccanismo di sincronizzazione che fornisce un modo semplice e sicuro per implementare le condizioni atomiche. Un monitor è una classe o un oggetto che fornisce metodi per acquisire e rilasciare lock e altri meccanismi di sincronizzazione. In genere, i monitor sono utilizzati in combinazione con i lock per implementare le condizioni atomiche.

# Capitolo 7 - Sistemi Operativi

## Esempi di sincronizzazione

### Classici problemi di sincronizzazione

Nelle soluzioni che verranno proposte si impiegano sempre i semafori, ma le implementazioni reali possono utilizzare i lock mutex al posto dei semafori binari.

Problemi classici utilizzati per testare gli schemi di sincronizzazione:

- spiegazione dei bounded buffer e i suoi problemi (produttore/consumatore con memoria limitata)
- problemi del Reader e del Writer
- problema dei Filosofi che cenano

### Produttore/Consumatore con memoria limitata

Il problema del Produttore/Consumatore con memoria illimitata *si usa per illustrare la potenza delle primitive di sincronizzazione*.

Si presenta uno schema generale di soluzione, senza far riferimento ad alcuna realizzazione particolare.

Nel nostro problema, Produttore e Consumatore condividono le seguenti strutture dati:

```
int n;
semaphore mutex = 1; (mutua esclusione)
semaphore empty = n; (posizioni vuote)
semaphore full = 0; (posizioni piene)
```

- n buffer; ognuno può contenere un elemento
- Semaforo mutex inizializzato al valore 1 (è per la mutua esclusione e dev'essere inizializzato a 1)
- Semaforo full inizializzato al valore 0 (conta il numero di posizioni piene nel buffer)
- Semaforo empty inizializzato al valore n (conta il numero di posizioni vuote nel buffer)

Produttore:

```
while (true) {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);

    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
}
```

si dovrebbe fermare se non ci sono posizioni vuote (empty = 0).

Faccio una wait su empty: se ci sono posizioni vuote, esso viene decrementato e il produttore può entrare perché c'è spazio.

Per evitare danni entro uno per volta: questo viene garantito inserendo una wait(mutex) a inizio codice e una signal (mutex) a fine codice. C'è anche una signal(full) per incrementare il semaforo.

Consumatore:

```
while (true) {
    wait(full);
    wait(mutex);

    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);

    ...
    /* consume the item in next consumed */
    ...
}
```

Fa le operazioni duali su full ed empty.

Il codice si può interpretare nel senso di produzione, da parte del produttore, di posizioni piene per il consumatore; oppure di produzione, da parte del consumatore, di posizioni vuote per il produttore.

### Problema dei lettori-scrittori

Si supponga che una base di dati sia da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto della base di dati, mentre altri ne possono richiedere un aggiornamento, vale a dire una lettura e una scrittura. Questi due tipi di processi vengono distinti chiamando lettori quelli interessati alla sola lettura e scrittura gli altri. Naturalmente, se due lettori accedono nello stesso momento all'insieme di dati condiviso, non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro processo (lettore o scrittore) accedono contemporaneamente alla stessa base di dati, ne può derivare il caos.

Per impedire l'insorgere di difficoltà di questo tipo è necessario che gli scrittori abbiano un accesso esclusivo in fase di scrittura alla base di dati condivisa.

Il problema dei lettori-scrittori, da quando è stato enunciato, è stato usato per verificare quasi tutte le nuove primitive di sincronizzazione. Ha diverse varianti, che implicano tutte l'esistenza di priorità.

### I variante

La più semplice, richiede che nessun lettore attenda, a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme di dati condiviso -> nessun lettore deve attendere che altri lettori terminino l'operazione solo perché uno scrittore attende l'accesso ai dati.

Viene data una priorità ai lettori. Essi si aggiungeranno ogni volta che c'è una nuova richiesta, facendo attendere gli scrittori. Bisognerebbe alternare lettori e scrittori per non avere uno stato d'attesa indefinita da parte degli scrittori.

Ci sono quindi varie variazioni possibili del problema, come non far attendere nessun reader a meno che un writer non abbia il permesso di usare i dati condivisi o far scrivere il prima possibile al writer sui dati condivisi quando è pronto. Questo problema è risolta in alcuni sistemi dal kernel che provvede dei lock per i writer e i reader.

### Il variante (non ne verrà trattata la soluzione)

Richiede che uno scrittore, una volta pronto, esegua il proprio compito di scrittura al più presto -> se uno scrittore attende l'accesso all'insieme di dati, nessun nuovo lettore deve iniziare la lettura.

La soluzione del primo problema e quella del secondo possono condurre a uno stato d'attesa indefinita degli scrittori nel primo caso e dei lettori nel secondo.

### Soluzione della I variante del problema

Si prevede la condivisione da parte dei processi lettori delle seguenti strutture dati:

```
semaphore rw_mutex = 1; -> permette di gestire l'accesso alla variabile  
semaphore mutex = 1;  
int read_count = 0;
```

- rw\_mutex: semaforo di mutua esclusione per gli scrittori e serve anche al primo o all'ultimo lettore che esce dalla sezione critica
- mutex: semaforo utilizzato solo per la mutua esclusione al momento dell'aggiornamento di read\_count
- read\_count: variabile condivisa che conta i lettori attualmente attivi. Devo incrementarla per un lettore alla volta per non creare conflitti

Processo scrittore:

```
while (true) {  
    wait(rw_mutex);  
    ...  
    /* writing is performed */  
    ...  
    signal(rw_mutex);  
}
```

Fa una wait su rw\_mutex e poi fa una signal per rilasciare la mutua esclusione. Indipendentemente dal numero di scrittori, ognuno sarà libero di scrivere perché opereranno uno per volta.

Processo lettore:

```
while (true){  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
}
```

Il primo lettore deve fare una wait come lo scrittore per assicurarsi l'accesso esclusivo, poi posso aggiungere quanti lettori voglio al gruppo. La variabile read\_count tiene conto dei lettori: uso mutex per incrementarla in mutua esclusione.

Se read\_count == 1 -> il lettore è il primo e fa una wait su rw\_mutex per richiedere il permesso; fatto questo rilascia il mutex e legge.

Arriva un secondo lettore, incrementa read\_count, bypassa l'if e legge direttamente.

Uscita: dopo la lettura riduco reader\_count e richiedo la mutua esclusione per non perdere i decrementi (essendo la variabile condivisa).

Se read\_count == 1, vuol dire che era l'ultimo lettore rimasto.

Faccio quindi la signal rw\_mutex per rilasciare la mutua esclusione, sbloccando gli scrittori che vogliono scrivere.

In ogni caso, rilascio la mutua esclusione su mutex.

Se uno scrittore si trova nella sezione critica e n lettori attendono di entrarvi, si accoda un lettore a rw\_mutex e n-1 lettori a mutex. Inoltre, se uno scrittore esegue signal(rw\_mutex) si può riprendere l'esecuzione dei lettori in attesa, oppure di un singolo scrittore in attesa. La scelta è fatta dallo scheduler.

Le soluzioni al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire lock di lettura-scrittura. Per acquisire un tale lock è necessario specificarne la modalità, scrittura o lettura: se il processo desidera solo leggere i dati condivisi, richiede un lock di lettura-scrittura in modalità lettura; se invece desidera anche modificarne i dati, lo richiede in modalità scrittura.

E' permesso a più processi di acquisire lock di lettura-scrittura in modalità lettura, ma solo un processo alla volta può avere il lock di lettura-scrittura in modalità scrittura, visto che nel caso della scrittura è necessario garantire l'accesso esclusivo.

I lock di lettura-scrittura sono utili soprattutto nelle seguenti situazioni:

- Nelle applicazioni in cui è facile identificare i processi che si limitano alla lettura di dati condivisi e quelli che si limitano alla scrittura di dati condivisi
- Nelle applicazioni che prevedono più lettori che scrittori. Infatti, i lock di scrittura comportano in genere un carico di lavoro aggiuntivo rispetto ai semafori o ai lock mutex, compensato però dalla possibilità di eseguire molti lettori in concorrenza

## Problema dei cinque filosofi (dining philosophers)

Si considerino cinque filosofi che trascorrono la loro esistenza pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso e la tavola è apparecchiata con cinque bacchette (chopstick). Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino. Quando un filosofo tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare.

*Il problema dei cinque filosofi è considerato un classico problema di sincronizzazione perché rappresenta una vasta classe di sistemi di controllo della concorrenza, in particolare i problemi caratterizzati dalla necessità di assegnare varie risorse a diversi processi evitando situazioni di stallo e di attesa indefinita.*

### Soluzione con uso di semafori

Una semplice soluzione consiste nel *rappresentare ogni bacchetta con un semaforo*: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione `wait()` su quel semaforo e la posa eseguendo operazioni `signal()` sui semafori appropriati.

Quindi, i dati condivisi sono:

```
semaphore chopstick[5];
```

dove tutti gli elementi `chopstick` sono inizializzati a 1.

Struttura del filosofo  $i$ :

```
while(true){
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    ...
    / mangia /
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    ...
    / pensa /
    ...
}
```

$(i+1)\%5$ : si utilizza per fare in modo di prendere la prossima bacchetta ciclicamente; tutte le cose cicliche si fanno tramite la modulazione. Faccio due `wait` per bacchetta: quando sono assegnate sono utilizzabili da un solo filosofo alla volta; poi viene fatta la `signal` per rilasciarla. Questa soluzione garantisce che due vicini non mangino contemporaneamente, ma è insufficiente perché non esclude la possibilità che si abbia una situazione di stallo. Si sopponga che tutti e cinque i filosofi abbiano fame contemporaneamente e che ciascuno afferri la bacchetta di sinistra; tutti gli elementi di `chopstick` diventano uguali a zero, perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo. [Problema -> la soluzione non funziona se tutti cercano di mangiare contemporaneamente. Tutti riusciranno a beccarsi la bacchetta di sinistra, ma la bacchetta di destra sarà presa dal vicino -> deadlock, rimarranno bloccati per sempre]

Tali situazioni di stallo possono essere evitate con i seguenti espedienti:

- solo quattro filosofi possono stare contemporaneamente a tavola
- un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (quest'operazione si deve eseguire in una sezione critica)
- si adotta una soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra

### Soluzione per mezzo di monitor

La soluzione impone il vincolo che un filosofo possa prendere le sue bacchette solo quando siano entrambe disponibili.

Per codificare questa soluzione si devono distinguere i tre diversi stati in cui si può trovare un filosofo: pensante, affamato, mangiante (utilizziamo una enum).

Un filosofo può entrare nello stato eating solo se i suoi vicini non stanno mangiando.

Occorre dichiarare la struttura dati `condition self[5]`; che permette al filosofo  $i$  di ritardare se stesso quando ha fame, ma non riesce a ottenere le bacchette di cui ha bisogno.

La distribuzione delle bacchette è controllata dal monitor `DiningPhilosophers`. Ciascun filosofo, prima di cominciare a mangiare, deve invocare l'operazione `pickup()`; ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione, il filosofo può mangiare; in

seguito, il filosofo invoca l'operazione *putdown()* e comincia a pensare. Il filosofo i deve quindi chiamare le operazioni *pickup()* e *putdown()* nella seguente sequenza: *DiningPhilosophers.pickup(i); ... eat .... DiningPhilosophers.putdown(i);*

Entrambe queste soluzioni evitano i deadlock, ma non la starvation (è da questo problema che prende il nome questa situazione).

## Sincronizzazione Posix

L'API di Posix provvede:

- mutex lock
- semafori
- variabili condizionali

Con i mutex e le variabili condizionali possiamo scrivere qualcosa di simile ad un monitor. Sono entrambe nello standard orientato ai threads. I semafori si possono utilizzare, ma è meglio non farlo perché sono in un altro standard.

Usato in UNIX.

## Sincronizzazione in Java

Java dà la possibilità di creare dei thread. Un metodo può essere synchronized (eseguito uno per volta). All'interno c'è la possibilità di utilizzare due operazioni che sono come *wait()* e *signal()* dei monitor -> ogni oggetto Java ha un mutex, definito internamente per default, e una variabile condition.

Questa è una limitazione rispetto ai monitor veri e propri, dove posso definire quante variabili condition voglio, ma la semantica è la stessa. Ci sono *wait()* e *notify()* che sono equivalenti a *wait()* e *signal()*.

## Quali sono i problemi di sincronizzazione?

I principali problemi di sincronizzazione sono il problema lettore-scrittore e il problema dei cinque filosofi.

Problema lettore-scrittore: si supponga una base di dati da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto dell'oggetto condiviso, mentre altri possono richiedere un aggiornamento, vale a dire una lettura e una scrittura. Questi due processi sono distinti, e si indicano chiamando lettori quelli interessati alla sola lettura e scrittori gli altri. Se due lettori accedono nello stesso momento all'insieme di dati condiviso, non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro processo (lettore o scrittore) accedono contemporaneamente alla stessa base di dati, ne può derivare il caos. Questo problema di sincronizzazione è conosciuto come problema dei lettori-scrittori. Il primo problema dei lettori-scrittori ha diverse varianti, la più semplice cui si fa riferimento come al primo problema dei lettori-scrittori, richiede che nessun lettore attenda, a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme di dati condiviso.

Il secondo problema dei lettori-scrittori si fonda sul presupposto che uno scrittore, una volta pronto esegua il proprio compito di scrittura al più presto: se uno scrittore attende un accesso all'insieme di dati, nessun nuovo lettore deve iniziare le letture. La soluzione al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire lock di lettura-scrittura. Per acquisire tale lock, è necessario specificare la modalità di scrittura o di lettura: se il processo desidera solo leggere i dati condivisi, richiede un lock di lettura-scrittura in modalità lettura; se invece desidera anche modificare i dati, lo richiede in modalità scrittura. E' permesso a più processi di acquisire lock di lettura-scrittura in modalità lettura, ma solo un processo alla volta può avere il lock di lettura-scrittura in modalità scrittura, visto che nel caso della scrittura è necessario garantire l'accesso esclusivo.

## Problema dei cinque filosofi e soluzioni

Si considerino 5 filosofi che condividono un tavolo rotondo circondato da 5 sedie, e la tavola è apparecchiata con 5 bacchette. Quando a un filosofo gli viene fame, tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trovi già nelle mani di un suo vicino. Quando un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Il problema dei 5 filosofi, è considerato un classico problema di sincronizzazione, perché rappresenta una vasta classe di problemi di controllo della concorrenza, in particolare i problemi caratterizzati dalla necessità di assegnare varie risorse a diversi processi evitando situazioni di stallo e di attesa indefinita.

Una semplice soluzione consiste nel rappresentare ogni bacchetta con un semaforo: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione *wait()* su quel semaforo e la posa eseguendo *signal()* sui semafori appropriati. Questa soluzione garantisce che due vicini non mangino contemporaneamente, ma è insufficiente poiché non esclude la possibilità che si abbia una situazione di stallo. Si supponga che tutti e 5 i filosofi abbiano fame contemporaneamente e che ciascuno tenti di afferrare la bacchetta di sinistra; ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo.

Possibili soluzioni:

- solo quattro filosofi possono stare contemporaneamente a tavola
- un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (eseguire in una sezione critica)
- soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

Soluzione con semafori: Una semplice soluzione consiste nel rappresentare ogni bacchetta con un semaforo: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione *wait()* su quel semaforo e la posa eseguendo un'operazione *signal()* sui semafori appropriati. Questa soluzione garantisce che i tuoi vicini mangino contemporaneamente, ma è sufficiente poiché non esclude la possibilità che si abbia una soluzione di stallo. Si supponga che tutti e 5 filosofi abbiano fame contemporaneamente e che ciascuno afferri la bacchetta di sinistra. Tutti gli elementi del semaforo diventano uguale a 0 perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo. Tali situazioni possono essere evitate con i seguenti espiedimenti: solo quattro filosofi possono stare contemporaneamente a tavola; un filosofo può prendere le sue bacchette solo se sono

entrambe disponibili; si adotta una soluzione asimmetrica, cioè un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra invece, un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

Soluzione con monitor: La soluzione impone il vincolo che un filosofo possa prendere le sue bacchette solo quando siano entrambe disponibili. Si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. enum { pensa, affamato, mangia } stato[5]; Il filosofo "i" può impostare la variabile stato[i]=mangia solo se i suoi due vicini non stanno mangiando: ((stato[(i+4)%5] != mangia) && (stato[(i+1)%5] != mangia)). Ciascun filosofo, prima di cominciare a mangiare, deve invocare l'operazione prende(); ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione, il filosofo può mangiare; in seguito, il filosofo invoca l'operazione posa() e comincia a pensare. Il filosofo "i" deve quindi chiamare le operazioni prendi() e posa() nella seguente sequenza: fc.prende(i); ... mangia ... fc.posa(i); Questa soluzione assicura che due vicini non mangino contemporaneamente e che non si verifichino situazioni di stallo. Occorre però notare che un filosofo può attendere indefinitamente.

# Capitolo 8 - Sistemi Operativi

## Stallo dei processi (deadlock)

### Modello di sistema

Storicamente è un problema di chi programma applicazioni concorrenti su una singola CPU che si alterna al sistema operativo: è per questo che lo studiamo.

Un sistema è composto da un numero finito di risorse da distribuire tra più thread in competizione. Le risorse possono essere suddivise in tipi (o classi) differenti, ciascuno formato da un certo numero di istanze identiche. Cicli di CPU, file e dispositivi di I/O (come interfacce di rete e lettori DVD), sono tutti esempi di tipi di risorsa. Se un sistema ha quattro CPU, il tipo di risorsa CPU ha quattro istanze. Analogamente, il tipo di risorsa rete può avere due istanze. Se un thread richiede un'istanza relativa a un tipo di risorsa, l'assegnazione di qualsiasi istanza di quel tipo dovrebbe soddisfare la richiesta. Se ciò non si verifica significa che le istanze non sono identiche e le classi di risorse non sono state definite correttamente.

In un ambiente con multiprogrammazione, più thread possono competere per ottenere un numero finito di risorse: se una risorsa non è correttamente disponibile, il thread richiedente passa allo stato d'attesa. In alcuni casi, se le risorse sono trattenute da altri thread, a loro volta nello stato d'attesa, il thread potrebbe non cambiare più il suo stato. Situazioni di questo tipo sono dette situazioni di *stallo*.

Prima di adoperare una risorsa, un thread deve richiederla e, dopo averla usata, deve rilasciarla. Un thread può richiedere tutte le risorse necessarie per eseguire il compito assegnatogli, anche se ovviamente il numero delle risorse richieste non può superare quello totale delle risorse disponibili del sistema: per esempio un thread non può richiedere due interfacce di rete se il sistema ne ha solo una.

Nelle ordinarie condizioni di funzionamento un thread può servirsi di una risorsa soltanto se rispetta la seguente sequenza:

- *Richiesta*: il thread richiede la risorsa; se non si può soddisfare immediatamente, il thread richiedente deve attendere finché non può acquisire tale risorsa
- *Uso*: il thread può operare sulla risorsa
- *Rilascio*: il thread rilascia la risorsa.

La richiesta e il rilascio di risorse avvengono tramite syscall: ne sono esempi request() e release() di una periferica, open() e close() di un file, allocate() e free() di una porzione di memoria. In modo simile, la richiesta e il rilascio si possono eseguire per mezzo delle operazioni wait() e signal() dei semafori o con l'utilizzo delle funzioni acquire() e release() dei lock mutex. Quindi, *ogni volta che si usa una risorsa gestita dal kernel, il sistema operativo controlla che il thread utente ne abbia fatto richiesta e che questa gli sia stata assegnata*. Una tabella di sistema registra lo stato di ogni risorsa e, se questa è assegnata, indica il thread relativo. Se un thread richiede una risorsa già assegnata a un altro thread, il thread richiedente può essere accodato agli altri thread che attendono tale risorsa.

*Un gruppo di thread si trova in stallo quando ogni thread del gruppo attende un evento che può essere causato solo da un altro thread che si trova nel gruppo.* Gli eventi che interessano maggiormente in questo contesto sono l'acquisizione e il rilascio di risorse. Le risorse sono tipicamente logiche; tuttavia, vi sono altri tipi di eventi che possono causare deadlock, come la lettura da un'interfaccia di rete.

**I deadlock sono per sempre; se prima o poi il processo acquisisce la risorsa, non ci troviamo in un deadlock. C'è un deadlock solo se i processi sono bloccati per sempre e deve intervenire il sistema operativo.**

### Caratterizzazione delle situazioni di stallo

#### Condizioni necessarie

Si può avere una situazione di stallo solo se in un sistema si verificano contemporaneamente le seguenti quattro condizioni:

1. *Mutua esclusione*: almeno una risorsa dev'essere non condivisibile, vale a dire che è utilizzabile solo da un thread alla volta. Se un altro thread richiede tale risorsa, si deve ritardare il thread richiedente fino al rilascio della risorsa.
2. *Processo e attesa*: un thread dev'essere in possesso di almeno una risorsa e attendere di acquisire risorse già in possesso di altri thread.
3. *Attesa con prelazione*: le risorse non possono essere prelazionate, vale a dire che una risorsa può essere rilasciata dal thread che la possiede solo volontariamente, dopo aver terminato il proprio compito.
4. *Attesa circolare*: deve esistere un insieme  $\{T_0, T_1, \dots, T_n\}$  di thread tale che  $T_0$  attende una risorsa posseduta da  $T_1$ ,  $T_1$  attende una risorsa posseduta da  $T_2$ ,  $T_{n-1}$  attende una risorsa posseduta da  $T_n$  e  $T_n$  attende una risorsa posseduta da  $T_0$ .

Occorre sottolineare che tutte e quattro le condizioni devono essere vere, altrimenti non si può avere alcuno stallo. La condizione dell'attesa circolare implica la condizione di possesso e attesa, quindi le quattro condizioni non sono completamente indipendenti.

#### Grafo di assegnazione delle risorse

I deadlock si possono descrivere con maggiore precisione avvalendosi di una rappresentazione detta *grafo di assegnazione delle risorse*. Si tratta di un *insieme di nodi V e archi E*, con l'insieme di nodi V composto da due sottoinsiemi:

1.  $T = \{T_1, T_2, \dots, T_n\}$  che rappresenta tutti i thread del sistema

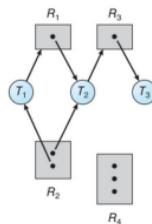
2.  $R=\{R_1, R_2, \dots, R_n\}$  che rappresenta tutti i tipi di risorsa del sistema.

Tipi di archi:

- **Arco di richiesta:** arco orientato da  $T$  a  $R$ . Indica che il processo  $T$  ha chiesto un'istanza del tipo di risorsa  $R$  e attualmente attende tale risorsa.
- **Arco di assegnazione:** arco orientato da  $R$  a  $T$ . Indica che la risorsa  $R$  è stata assegnata al processo  $P$ .

Esempio di grafo di allocazione risorse

- One instance of  $R_1$
- Two instances of  $R_2$
- One instance of  $R_3$
- Three instance of  $R_4$
- $T_1$  holds one instance of  $R_2$  and is waiting for an instance of  $R_1$
- $T_2$  holds one instance of  $R_1$ , one instance of  $R_2$ , and is waiting for an instance of  $R_3$
- $T_3$  holds one instance of  $R_3$



Graficamente, ogni thread  $T$  si rappresenta con un cerchio e ogni tipo di risorsa  $R$  con un rettangolo.

Quando il thread  $T$  richiede un'istanza del tipo di risorsa  $R$ , si inserisce un arco di richiesta nel grafo di assegnazione delle risorse. Se questa richiesta può essere esaudita, si trasforma immediatamente l'arco di richiesta in un arco di assegnazione, che al rilascio della risorsa viene cancellato.

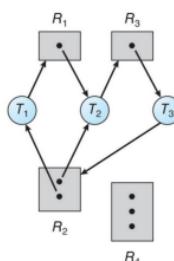
*Se il grafo non contiene cicli, nessun thread del sistema subisce uno stallo; se il grafo contiene un ciclo, può sopraggiungere uno stallo.*

*Se ciascun tipo di risorsa ha esattamente un'istanza, allora l'esistenza di un ciclo implica la presenza di uno stallo; se il ciclo riguarda solo un insieme di tipi di risorsa, ciascuno dei quali ha solo un'istanza, si è verificato uno stallo.*

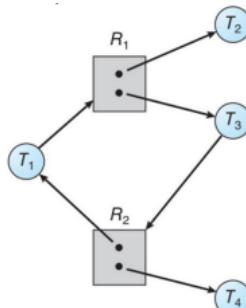
*Ogni thread che si trova nel ciclo è in stallo. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria e sufficiente per l'esistenza di uno stallo.*

*Se ogni tipo di risorsa ha più istanze, un ciclo non implica necessariamente uno stallo. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria ma non sufficiente per l'esistenza di uno stallo.*

Nell'immagine in alto non c'è un ciclo, quindi nessun processo subisce uno stallo: quanto  $T_3$  avrà finito, la risorsa  $R_3$  sarà libera per essere usata da  $T_2$ .



$T_1, T_2$  E  $T_3$  sono in stallo: ognuno attende una risorsa posseduta da un altro processi del gruppo.



Prima o poi  $T_2$  o  $T_4$  si libereranno e  $T_1$  e  $T_3$  riceveranno le risorse richieste. Nota: non è garantito che  $T_4$  non abbia bisogno di altre risorse non indicate nel grafo e non siamo sicuri che in futuro non possa restare coinvolto in un deadlock.

Assunzioni:

- Se un grafo non contiene cicli: niente deadlock.
- Se un grafo contiene un ciclo:
  - se c'è solo un'istanza per risorsa c'è un deadlock
  - se c'è più di un'istanza per risorsa c'è la possibilità di un deadlock

*Un ciclo è una condizione necessaria ma non sufficiente per individuare un deadlock.*

## Metodi per la gestione delle situazioni di stallo

Il problema delle situazioni di stallo si può affrontare in tre modi:

1. Fare in modo di non entrare mai in un deadlock, utilizzando uno di questi due protocolli: deadlock prevention (evito preventivamente di entrare in un deadlock), deadlock avoidance (più soft)
2. Permettere al sistema di entrare in uno stato deadlock, per poi individuarlo e cercare di ripristinarlo (ucciso dei processi per recuperare le risorse)

3. Ignorare il problema e pretendere che le situazioni di stallo non possano mai verificarsi nei sistemi (spero che non avvenga)

I programmatore di kernel si concentrano sulla prima soluzione. I database adottano la seconda soluzione. La terza soluzione è quella adottata dalla maggior parte dei sistemi operativi come Linux e Windows.

Quale atteggiamento scelgo? Dipende dalla frequenza del fenomeno.

Se è molto frequente, devo fare in modo di prevenire i deadlock. Se è meno frequente posso anche ignorare la situazione.

La soluzione di mezzo non è molto efficiente perché devo sfruttare gli algoritmi sui grafi, i quali possono diventare molto complessi se c'è un alto numero di nodi (e ho tanti nodi poiché sono processi e risorse).

Le tre soluzioni proposte possono essere combinati, in modo da permettere la selezione della migliore per ciascuna classe di risorse del sistema.

Per assicurare che non si verifichi mai uno stallo, il sistema può servirsi di metodi di prevenzione o di metodi per evitare tale situazione. *Prevenire le situazioni di stallo* significa far uso di metodi per evitare tale situazione. Prevenire le situazioni di stallo significa far uso di metodi atti ad assicurare che non si verifichi una delle condizioni necessarie per il verificarsi di deadlock.

Per evitare le situazioni di stallo occorre che il sistema operativo abbia in anticipo informazioni aggiuntive riguardanti le risorse che un thread richiederà e userà durante le sue attività. Con queste informazioni aggiuntive il sistema operativo può decidere se una richiesta di risorse da parte di un thread si può soddisfare o se il thread debba invece attendere. In tale thread di decisione il sistema tiene conto delle risorse correntemente disponibili, di quelle correntemente assegnate a ciascun thread e delle future richieste e futuri rilasci di ciascun thread.

Se un sistema non fornisce alcun tipo di meccanismo per l'individuazione degli stalli e il ripristino del sistema, situazioni di stallo possono avvenire senza la possibilità di capire che cos'è successo. Le situazioni di stallo non rilevate portano ad una degradazione delle prestazioni del sistema; infatti vi sono risorse assegnate a processi che non si possono eseguire e un numero crescente di thread richiede tali risorse ed entra in stallo, fino al blocco totale di sistema che dovrà essere riavvita manualmente.

Anche se ignorare la possibilità di deadlock può non sembrare una valida soluzione al problema delle situazioni di stallo, viene comunque utilizzato nella maggior parte dei sistemi operativi. Gli aspetti economici sono importanti: ignorare la possibilità di situazioni di stallo è più conveniente rispetto ad altri approcci. Dal momento che in molti sistemi le situazioni di stallo si verificano raramente, sostenere una spesa aggiuntiva per utilizzare gli altri metodi non sembra essere così conveniente. Inoltre, i metodi utilizzati per risolvere altre situazioni possono essere utilizzati per gestire le situazioni di stallo.

Il sistema deve disporre di meccanismi manuali di ripristino per queste situazioni.

## Prevenzione delle situazioni di stallo

Affinché si abbia uno stallo si devono verificare quattro condizioni necessarie; perciò si può prevenire il verificarsi di un deadlock assicurando che almeno una di queste condizioni non possa capitare.

### Mutua esclusione

Deve valere la condizione di mutua esclusione: almeno una risorsa dev'essere non condivisibile. Le risorse condivisibili, invece, non richiedono l'accesso mutuamente esclusivo, perciò non possono essere coinvolte in uno stallo. I file aperti per la sola lettura sono un buon esempio di risorsa condivisibile; se più thread richiedono l'apertura di un file in sola lettura, possono ottenere un accesso contemporaneo. Un thread non deve mai attendere una risorsa condivisibile. Ma poiché alcune risorse sono intrinsecamente non condivisibili, non si possono prevenire in generale le situazioni di stallo negando la condizione di mutua esclusione. Per esempio, un lock mutex non può essere contemporaneamente condiviso da diversi thread.

### Possesso e attesa

Per assicurare che la condizione di possesso e attesa non si presenti mai nel sistema, occorre garantire che un thread che richiede una risorsa non ne possegga altre. Si può usare un protocollo che ponga la condizione che ogni thread, prima di iniziare la propria esecuzione, richieda tutte le risorse che gli servono e che esse gli siano assegnate. A causa della natura dinamica della richiesta di risorse questa condizione è poco pratica. Un protocollo alternativo è quello che permette a un thread di richiedere risorse solo se non ne possiede: un thread può richiedere risorse e adoperarle, ma prima di richiedere ulteriori risorse deve rilasciare tutte quelle che possiede.

Entrambi i protocolli presentano due svantaggi principali: innanzitutto, l'utilizzo delle risorse può risultare poco efficiente, poiché molte risorse possono essere assegnate, ma non utilizzate, per un lungo periodo di tempo. Il secondo svantaggio è dovuto al fatto che si possono verificare situazioni di attesa indefinita. Un thread che richieda più risorse molto utilizzate può trovarsi nella condizione di attendere indefinitamente la disponibilità, poiché almeno una delle risorse di cui necessita è sempre assegnata a qualche altro thread.

### Assenza di prelazione

La terza condizione necessaria prevede che non sia possibile avere la prelazione su risorse già assegnate. Per assicurare che questa condizione non sussista, si può impiegare il seguente protocollo: se un thread che possiede una o più risorse ne richiede un'altra che non gli si può assegnare immediatamente (e quindi il thread deve attendere) allora si esercita la prelazione su tutte le risorse attualmente in suo possesso. Si ha cioè il rilascio implicito di queste risorse, che si aggiungono alla lista delle risorse che il thread sta attendendo; il thread viene nuovamente avviato solo quando può ottenere sia le vecchie risorse sia quella nuova che sta richiedendo.

In alternativa, quando un thread richiede alcune risorse, se ne verifica la disponibilità: se sono disponibili vengono assegnate, se non lo sono, si verifica se sono assegnate a un thread che attende altre risorse. In tal caso si sottraggono le risorse desiderate a quest'ultimo thread e si assegnano al thread richiedente. Se le risorse non sono disponibili né sono possedute da un thread in attesa, il thread richiedente deve attendere. Durante l'attesa si può avere la prelazione di alcune sue risorse; ciò può accadere solo se un altro thread le richiede. Un thread si può avviare nuovamente solo quando riceve le risorse che sta richiedendo e recupera tutte quelle a esso sottratte durante l'attesa.

Questo protocollo è applicato spesso per risorse il cui stato si può salvare e recuperare facilmente in un secondo tempo, come i registri della CPU e le transazioni su un database, mentre non si può in generale applicare a risorse come i lock mutex e i semafori, ovvero a quei tipi di risorse che più comunemente generano situazioni di stallo.

## Attesa circolare (la più funzionale)

Offre un'opportunità per una soluzione pratica che renda non valida una delle condizioni di stallo.

*Un modo per assicurare che tale condizione d'attesa non si verifichi consiste nell'imporre un ordinamento totale all'insieme di tutti i tipi di risorse e imporre che ciascun thread richieda le risorse in ordine crescente.*

Ad ogni risorsa assegno un numero intero che permette di confrontare due risorse e stabilirne la relazione di precedenza nell'ordinamento. Ogni thread può richiedere risorse solo seguendo un ordine crescente di numerazione.

Esempio: il processo 2 può richiedere solo risorse enumerate dal 3 in poi e le deve richiedere in ordine crescente (può richiedere l'istanza 4 solo dopo aver richiesto l'istanza 3).

Questa soluzione funziona perché così una catena circolare non si potrà mai chiudere.

Si tenga presente che la semplice esistenza di un ordinamento delle risorse non protegge dallo stallo; è infatti responsabilità degli sviluppatori di applicazioni scrivere programmi che rispettino tale ordinamento.

```

first_mutex = 1
second_mutex = 5

code for thread_two could not be
written as follows:
}

/* thread one runs in this function */
void <>do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex.lock(&second_mutex);
    /* Do some work
     */
    pthread_mutex.unlock(&second_mutex);
    pthread_mutex.unlock(&first_mutex);

    pthread_exit(0);
}

/* thread two runs in this function */
void <>do_work_two(void *param)
{
    pthread_mutex.lock(&second_mutex);
    pthread_mutex.lock(&first_mutex);
    /* Do some work
     */
    pthread_mutex.unlock(&first_mutex);
    pthread_mutex.unlock(&second_mutex);

    pthread_exit(0);
}

```

Stabilire un ordine tra i lock può essere difficile, specialmente su un sistema con centinaia o migliaia di lock. Per affrontare questa sfida molti sviluppatori Java utilizzano il metodo `System.identityHashCode(Object)`, che restituisce il valore predefinito del codice hash del parametro `Object` che gli viene passato, come funzione per l'acquisizione ordinata dei lock.

E' anche importante notare che imporre un ordinamento sui lock non garantisce l'assenza di situazioni di stallo quando i lock possono essere acquisiti dinamicamente.

*Il metodo della prevenzione delle situazioni di stallo può causare effetti collaterali negativi, come uno scarso utilizzo dei dispositivi e una ridotta produttività del sistema.*

## Evitare situazioni di stallo

Un metodo alternativo per evitare le situazioni di stallo consiste nel richiedere ulteriori informazioni sulle modalità di richiesta delle risorse. Una volta acquisita la sequenza completa delle risorse e dei rilasci di ogni thread, il sistema può stabilire per ogni richiesta se il thread debba attendere o meno, per evitare una possibile situazione di stallo futura. *In seguito a ogni richiesta, il sistema deve esaminare le risorse attualmente assegnate a ogni thread e le richieste e i rilasci futuri per ciascun thread.*

Gli algoritmi differiscono tra loro per la quantità e il tipo di informazioni richieste. Il modello più semplice e più utile richiede che ciascun thread dichiari il numero massimo delle risorse di ciascun tipo di cui necessita. Data questa informazione a priori, si può costruire un algoritmo capace di assicurare che il sistema non entri mai in stallo. Questo algoritmo per evitare lo stallo esamina dinamicamente lo stato di assegnazione delle risorse per garantire che non possa esistere una condizione di attesa circolare. *Lo stato di assegnazione delle risorse è definito dal numero di risorse disponibili e assegnate e dalle richieste massime dei thread.*

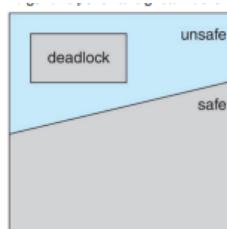
## Stato sicuro

*Uno stato si dice sicuro se il sistema è in grado di assegnare risorse a ciascun thread (fino al suo massimo) in un certo ordine e impedire il verificarsi di uno stallo: un sistema si trova in stato sicuro solo se esiste una sequenza sicura.*

Una sequenza di processi è detta sicura se, per ogni processo  $P_i$  le richieste che  $P_i$  può ancora fare si possono soddisfare impiegando le risorse attualmente disponibili più le risorse possedute da tutti i processi  $P_j$  con  $j < i$ . In questa situazione, se le risorse necessarie al processo  $P_i$  non sono disponibili immediatamente, allora  $P_i$  può attendere che tutti i  $P_j$  abbiano finito, e a quel punto  $P_i$  può ottenere tutte le risorse di cui ha bisogno, completare il compito assegnato, restituire le richieste e così via. Quando  $P_i$  termina,  $P_{i+1}$  può ottenere le risorse richieste e così via.

Il processo può essere terminato con le risorse disponibili o con le risorse tenute dai processi precedenti nella catena: siamo in uno stato sicuro.

Uno stato sicuro non è di stallo; viceversa, uno stato di stallo è uno stato non sicuro; tuttavia non tutti gli stati non sicuri sono stati di stallo.



Uno stato non sicuro può condurre a uno stallo. Finché lo stato rimane sicuro, il sistema operativo può evitare il verificarsi di stati non sicuri e di stallo.

In uno stato non sicuro il sistema operativo non può impedire ai thread di richiedere risorse in modo da causare uno stallo: ciò che accade negli stati non sicuri dipende dal comportamento dei thread.

*Il sistema può passare da uno stato sicuro a uno stato non sicuro se permette ad un processo di richiedere una risorsa prima che uno degli altri processi la liberi.*

Dato il concetto di stato sicuro, si possono definire algoritmi che permettano di evitare le situazioni di stallo. L'idea è semplice: è sufficiente assicurare che il sistema rimanga sempre in uno stato sicuro. *Il sistema si trova inizialmente in uno stato sicuro; ogni volta che un thread richiede una risorsa disponibile, il sistema deve stabilire se la risorsa può essere allocata o se il thread debba attendere. Si soddisfa la richiesta solo se l'assegnazione lascia il sistema in uno stato sicuro.*

In questo modo, se un thread richiede una risorsa attualmente disponibile può essere comunque costretto ad attendere. Quindi, l'utilizzo delle risorse può essere inferiore rispetto a quello che si avrebbe in assenza di un algoritmo per evitare le situazioni di stallo.

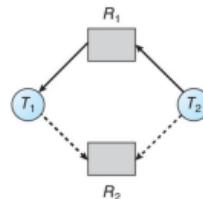
## Algoritmo con grafo di assegnazione delle risorse

Quando il sistema per l'assegnazione delle risorse è tale che *ogni tipo di risorsa ha una sola istanza*, per evitare situazioni di stallo si può far uso di una variante del grafo di assegnazione delle risorse. Oltre agli archi di richiesta e di assegnazione, si introduce l'*arco di rivendicazione*. Un arco di rivendicazione  $T_i \rightarrow R_j$  indica che il thread  $T_i$  può richiedere la risorsa  $R_j$  in qualsiasi momento futuro. Quest'arco ha la stessa direzione dell'arco di richiesta, ma si rappresenta con una linea tratteggiata. Quando il thread  $T_i$  richiede la risorsa  $R_j$ , l'arco di rivendicazione  $T_i \rightarrow R_j$  diventa un arco di richiesta. Analogamente, quando  $T_i$  rilascia la risorsa  $R_j$ , l'arco di assegnazione  $R_j \rightarrow T_i$  diventa un arco di rivendicazione  $T_i \rightarrow R_j$ .

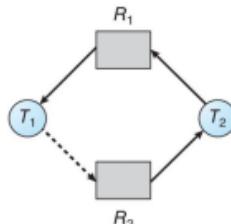
*Occorre sottolineare che le risorse devono essere rivendicate a priori dal sistema.* Ciò significa che prima che il thread  $T_i$  inizi l'esecuzione, tutti i suoi archi di rivendicazione devono essere già inseriti nel grafo di assegnazione delle risorse. Questa condizione si può rendere meno stringente permettendo l'aggiunta di un arco di rivendicazione  $T_i \rightarrow R_j$  al grafo solo se tutti gli archi associati al  $T_i$  sono archi di rivendicazione.

Si supponga che il thread  $T_1$  richieda la risorsa  $R_1$ . La richiesta si può soddisfare solo se la conversione richiesta dell'arco di richiesta  $T_1 \rightarrow R_1$  nell'arco di assegnazione  $R_1 \rightarrow T_1$  non causa la formazione di un ciclo nel grafo di assegnazione delle risorse. Possiamo verificare la condizione di sicurezza con un algoritmo di rilevamento dei cicli, che richiede un numero di operazioni dell'ordine  $n^2$ , dove  $n$  è il numero di thread del sistema.

Se non esiste alcun ciclo, l'assegnazione della risorsa lascia il sistema in uno stato sicuro. Se invece si trova un ciclo, l'assegnazione conduce il sistema in uno stato non sicuro e il thread  $T_1$  deve attendere che si soddisfino le sue richieste.



Si supponga che  $T_2$  richieda  $R_2$ ; sebbene sia attualmente libera,  $R_2$  non può essere assegnata a  $T_2$  poiché questa operazione creerebbe un ciclo nel grafo e un ciclo indica che il sistema è in uno stato non sicuro. Se, a questo punto,  $T_1$  richiedesse  $T_2$ , si avrebbe uno stallo.



## Algoritmo del banchiere

L'algoritmo con grafo di assegnazione delle risorse non si può applicare ai sistemi di assegnazione delle *risorse con più istanze di ciascun tipo di risorsa*. L'algoritmo per evitare le situazioni di stallo descritte nel seguito, pur essendo meno efficiente dello schema con grafo di assegnazione delle risorse, si può applicare a tali sistemi, ed è noto come algoritmo del banchiere.

Questo nome è stato scelto perché l'algoritmo si potrebbe impiegare in un sistema bancario per assicurare che la banca non assegna mai tutto il denaro disponibile, in modo da non poter più soddisfare le richieste di tutti i suoi clienti.

Quando si presenta al sistema, un nuovo thread deve dichiarare il numero massimo delle istanze di ciascun tipo di risorsa di cui potrà avere bisogno. Questo numero non può superare il numero totale delle risorse del sistema. Quando un utente richiede un gruppo di risorse, si deve stabilire se l'assegnazione di queste risorse lasci il sistema in uno stato sicuro. Se si rispetta tale condizione si assegnano le risorse, altrimenti il thread deve attendere che qualche altro thread ne rilasci un numero sufficiente.

La realizzazione dell'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema.

Sia  $n$  il numero di thread del sistema e  $m$  il numero di tipi di risorsa. Sono necessarie le seguenti strutture dati:

- *Disponibili*: un vettore di lunghezza  $m$  che indica il numero delle istanze disponibili per ciascun tipo di risorsa.  $Disponibili[j] = k$ , significa che sono disponibili  $k$  istanze del tipo di risorsa  $R_j$
- *Massimo*: una matrice  $n \times m$  che definisce la richiesta massima di ciascun thread;  $Massimo[i,j] = k$  significa che il thread  $T_i$  può chiedere un massimo di  $k$  istanze del tipo di risorsa  $R_j$

- **Assegnate:** una matrice  $n \times m$  definisce il numero delle istanze di ciascun tipo di risorsa attualmente assegnate a ogni thread;  $\text{Assegnate}[i,j] = k$  significa che al thread  $T_i$  sono correntemente assegnate  $k$  istanze del tipo di risorsa  $R_j$
- **Necessità:** una matrice  $n \times m$  che indica la necessità residua di risorse relativa a ogni thread.  $\text{Necessità}[i,j] = k$  significa che il thread  $T_i$ , per completare il suo compito, può avere bisogno di altre  $k$  istanze del tipo di risorsa  $R_j$ .  $\text{Necessità}[i,j] = \text{Massimo}[i,j] - \text{Assegnate}[i,j]$

## Algoritmo di verifica della sicurezza

L'algoritmo utilizzato per scoprire se il sistema è o meno in uno stato sicuro si può descrivere come segue:

1. Siano Lavoro e Fine vettori di lunghezza  $m$  ed  $n$ , si inizializza  $\text{Lavoro} = \text{Disponibili}$  e  $\text{Fine}[i] = \text{falso}$ , per  $i = 0, \dots, n-1$
2. Si cerca un indice  $i$  tale che valgano contemporaneamente le seguenti relazioni:
  - $\text{Fine}[i] == \text{falso}$
  - $\text{Necessità}_i \leq \text{Lavoro}$   
se tale  $i$  non esiste, si esegue il passo 4
3.  $\text{Lavoro} = \text{Lavoro} + \text{Assegnate}_i$   
 $\text{Fine}[i] = \text{vero}$   
si va al passo 2
4. Se  $\text{Fine}[i] == \text{vero}$  per ogni  $i$ , allora il sistema è in uno stato sicuro

Per determinare se uno stato è sicuro, tale algoritmo può richiedere un numero di operazioni dell'ordine di

$$m * n^2$$

## Algoritmo di richiesta delle risorse

Si descrive ora l'algoritmo che determina se le richieste possano essere soddisfatte mantenendo la condizione di sicurezza. Sia  $\text{Richieste}_i$  il vettore delle richieste per il Thread  $T_i$ . Se  $\text{Richieste}_i[j] == k$ , allora il thread  $T_i$  richiede  $k$  istanze del tipo di risorsa  $R_j$ . Se il thread  $T_i$  effettua una richiesta di risorse, si svolgono le seguenti azioni:

1. Se  $\text{Richieste}_i \leq \text{Necessità}_i$  si va al passo 2, altrimenti si riposta una condizione d'errore perché il thread ha superato il numero massimo di richieste
2. Se  $\text{Richieste}_i \leq \text{Disponibili}$  si esegue il passo 3, altrimenti  $T_i$  deve attendere poiché le risorse non sono disponibili
3. Si simula l'assegnazione al thread  $T_i$  delle risorse richieste modificando come segue lo stato di assegnazione delle risorse:  
 $\text{Disponibili} = \text{Disponibili} - \text{Richieste}_i$   
 $\text{Assegnate}_i = \text{Assegnate}_i + \text{Richieste}_i$   
 $\text{Necessità}_i = \text{Necessità}_i - \text{Richieste}_i$

Se lo stato di assegnazione delle risorse risultante è sicuro, la transazione è completata e al thread  $T_i$  si assegnano le risorse richieste. Tuttavia, se il nuovo stato è non sicuro  $T_i$  deve attendere  $\text{Richieste}_i$  e si ripristina il vecchio stato di assegnazione delle risorse.

### Esempio

Abbiamo cinque processi (da  $P_0$  a  $P_4$ ) e tre tipi di risorse (A, B, C). Il tipo di risorse A ha 10 istanze, il tipo di risorse B ha 5 istanze e il tipo di risorse C ha 7 istanze.

Si supponga che all'istanza T0 si sia verificata la seguente situazione del sistema:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	7 5 3	3 3 2
$P_1$	2 0 0	3 2 2	
$P_2$	3 0 2	9 0 2	
$P_3$	2 1 1	2 2 2	
$P_4$	0 0 2	4 3 3	

Available indica le risorse disponibili dal "banchiere".

Il contenuto della matrice necessità è definito come Massimo-Assegnate:

	<u>Need</u>
	A B C
$P_0$	7 4 3
$P_1$	1 2 2
$P_2$	6 0 0
$P_3$	0 1 1
$P_4$	4 3 1

Possiamo affermare che il sistema si trova attualmente in uno stato sicuro: infatti, la sequenza  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  soddisfa i criteri di sicurezza.

Si supponga ora che il processo  $P_1$  richieda un'altra istanza del tipo di risorsa A e due istanze del tipo C, quindi  $\text{Richieste}_i = (1, 0, 2)$ . Per stabilire se questa richiesta si possa soddisfare immediatamente verifichiamo la condizione  $\text{Richieste}_i \leq \text{Disponibili}$  (vale a dire  $(1,0,2) \leq (3,3,2)$ ), che risulta vera.

A questo punto simuliamo che questa richiesta sia stata soddisfatta e otteniamo il seguente nuovo stato:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	7 4 3	2 3 0
$P_1$	3 0 2	0 2 0	
$P_2$	3 0 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

Occorre stabilire se questo nuovo stato del sistema sia sicuro; a tale scopo si esegue l'algoritmo di verifica della sicurezza da cui risulta che la sequenza  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  rispetta il requisito di sicurezza. Quindi si può soddisfare immediatamente la richiesta del processo  $P_1$ . Tuttavia, dovrebbe essere chiaro che, quando il sistema si trova in questo stato, una richiesta di  $(3,3,0)$  da parte di  $P_4$  non si può soddisfare perché non sono disponibili le risorse. Inoltre, una richiesta di  $(9,2,0)$  da parte di  $T_0$  non si può soddisfare, anche se le risorse sono disponibili, poiché lo stato risultante sarebbe non sicuro.

## Rilevamento delle situazioni di stallo

Se un sistema non si avvale di un algoritmo per prevenire o per evitare lo stallo è possibile che una situazione di stallo si verifichi.

In un ambiente di questo genere, il sistema può fornire i seguenti algoritmi:

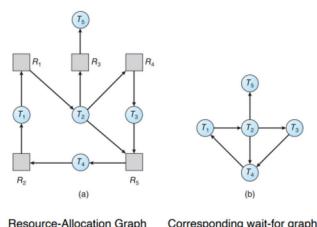
- un algoritmo che esamina lo stato del sistema per stabilire se si è verificato uno stallo;
- un algoritmo che ripristini il sistema dalla condizione di stallo

Occorre notare che uno schema di rilevamento e ripristino richiede un overhead che include non solo i costi per la memorizzazione delle informazioni necessarie e per l'esecuzione dell'algoritmo di rilevamento, ma anche i potenziali costi dovuti alle perdite di informazioni connesse al ripristino da una situazione di stallo.

### Istanza singola di ciascun tipo di risorsa

Se tutte le risorse hanno una sola istanza si può definire un algoritmo di rilevamento di situazioni di stallo che fa uso di una variante del grafo di assegnazione delle risorse, detta *grafo d'attesa*, ottenuta dal grafo di assegnazione delle risorse togliendo i nodi dei tipi di risorse e componendo gli archi tra i thread.

Più precisamente, un arco da  $T_i$  a  $T_j$  del grafo d'attesa implica che il thread  $T_i$  attende che il thread  $T_j$  rilasci una risorsa di cui  $T_i$  ha bisogno. Un arco  $T_i \rightarrow T_j$  esiste nel grafo d'attesa se e solo se il corrispondente grafo di assegnazione delle risorse contiene due archi  $T_i \rightarrow R_q$  e  $R_q \rightarrow T_j$  per qualche risorsa  $R_q$ .



Nel sistema esiste uno stallo se e solo se il grafo d'attesa contiene un ciclo. Per individuare le situazioni di stallo, il sistema deve mantenere aggiornato il grafo d'attesa e *invocare periodicamente un algoritmo che cerchi un ciclo all'interno del grafo*.

L'algoritmo per il rilevamento di un ciclo all'interno di un grafo richiede un numero di operazioni nell'ordine di  $n^2$  quadri, dove con  $n$  si indica il numero dei nodi del grafo.

Lo schema con grafo d'attesa non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa.

### Più istanze di ciascun tipo di risorsa

Questo algoritmo di rilevamento di situazioni di stallo è applicabile ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. Esso si serve di strutture variabili nel tempo (vedi algoritmo del Banchiere):

- *Disponibili*: vettore di lunghezza  $m$  che indica il numero delle istanze disponibili per ciascun tipo di risorsa
- *Assegnate*: matrice  $n \times m$  che definisce il numero delle istanze di ciascun tipo di risorse correttamente assegnate a ciascun thread
- *Richieste*: matrice  $n \times m$  che indica la richiesta attuale di ciascun thread. Se  $Richieste[i,j] = k$ , significa che il thread  $T_i$  sta richiedendo altre  $k$  istanze del tipo di risorsa  $R_j$

Sia qui che nell'algoritmo del Banchiere possiamo trattare le righe della matrice come vettori personali dei thread  $T_i$ .

Come prima, verifica ogni possibile sequenza di assegnazione per i processi che devono ancora essere completati:

1. Siano Lavoro e Fine vettori di lunghezza  $m$  e  $n$ , si inizializza  $Lavoro = Disponibili$ ; per  $i = 0, \dots, n$ , se  $Assegnate \neq 0$ , allora  $Fine[i] = \text{falso}$ , altrimenti  $Fine[i] = \text{vero}$
2. si cerca un indice  $i$  tale che valgano contemporaneamente le seguenti relazioni:
  - $Fine[i] == \text{falso}$
  - $Richieste_{-i} \leq Lavoro$
3.  $Lavoro = Lavoro + Assegnate_{-i}$   
 $Fine[i] = \text{vero}$

si torna al passo 2

4. se  $\text{Fine}[i] == \text{falso}$  per qualche  $i$ ,  $0 \leq i < n$ , allora il sistema è in stallo, inoltre, se  $\text{Fine}[i] == \text{falso}$ , il thread  $T_i$  è in stallo

Per determinare se il sistema è in stallo, tale algoritmo può richiedere un numero di operazioni dell'ordine di

$$m * n^2$$

Notare che al passo 3 si parte dal presupposto che il processo richieda solo le risorse strettamente necessarie al compimento del suo lavoro, e quindi sono immediatamente liberate. Se ciò non fosse vero si potrebbe verificare uno stallo, che sarà rilevato quando si richiamerà nuovamente l'algoritmo di rilevamento.

## Uso dell'algoritmo di rilevamento

Per sapere quando è necessario ricorrere all'algoritmo di rilevamento si devono considerare i seguenti fattori:

1. frequenza presunta con la quale si verifica uno stallo;
2. numero dei thread che sarebbero influenzati da tale stallo

Se le situazioni di stallo sono frequenti, è necessario ricorrere spesso all'algoritmo per il loro rilevamento. Le risorse assegnate a thread in stallo rimangono inattive fino all'eliminazione dello stallo. Inoltre, il numero di thread coinvolti nel ciclo di stallo può aumentare.

Le situazioni di stallo si verificano solo quando qualche thread fa una richiesta che non si può soddisfare immediatamente; questa può essere la richiesta che chiude una catena di thread in attesa. All'estremo, si può invocare l'algoritmo di rilevamento ogni volta che non si può soddisfare immediatamente una richiesta di assegnazione. In questo caso non si identifica soltanto il gruppo di thread in stallo, ma anche lo specifico thread che ha causato lo stallo, anche se, in verità, ciascuno dei thread in stallo è un elemento del ciclo all'interno del grafo di assegnazione delle risorse, quindi tutti i thread sono, congiuntamente, causa dello stallo. Se esistono tipi di risorsa diversi, una singola richiesta può causare più cicli nel grafo delle risorse, ciascuno dei quali viene completato da quest'ultima richiesta, causata da un thread identificabile.

Naturalmente, l'uso dell'algoritmo di rilevamento per ogni richiesta aumenta notevolmente il carico computazionale. Un'alternativa meno dispendiosa è quella in cui l'algoritmo di rilevamento s'invoca a intervalli definiti, per esempio ogni volta che l'utilizzo della CPU scende sotto il 40%, poiché uno stallo rende inefficiente le prestazioni del sistema e quindi porta a una drastica riduzione dell'utilizzo della CPU. Se l'algoritmo di rilevamento viene invocato in momenti arbitrari, poiché nel grafo delle risorse possono coesistere molti cicli, normalmente non si può dire quale fra i tanti thread in stallo abbia causato lo stallo.

## Ripristino da situazioni di stallo

Una volta rilevato uno stallo, questo si può affrontare in diversi modi. Una soluzione consiste nell'informare l'operatore della presenza dello stallo, in modo che possa gestirlo manualmente. L'altra soluzione lascia al sistema il ripristino automatico della situazione di stallo.

Uno stallo si può eliminare in due modi:

1. terminazione di uno o più thread per interrompere l'attesa circolare
2. esercitare la prelazione su alcune risorse in possesso di uno o più thread in stallo

## Terminazione di processi e thread

Per eliminare le situazioni di stallo attraverso la terminazione di processi o thread si possono adoperare due metodi; in entrambi il sistema reupera tutte le risorse assegnate ai processi terminati.

- *Terminazione di tutti i processi in stallo*: chiaramente questo metodo interrompe il ciclo di stallo, ma l'operazione è molto onerosa; questi processi possono aver già fatto molti calcoli i cui risultati si scartano e probabilmente dovranno essere ricalcolati
- *Terminazione di un processo alla volta fino all'eliminazione del ciclo di stallo*: questo metodo comporta un notevole overhead poiché, dopo aver terminato ogni processo, si deve invocare un algoritmo di rilevamento per stabilire se esistono ancora processi in stallo

Procurare la terminazione di un processo può non essere un'operazione semplice: se il processo si trova nel mezzo dell'aggiornamento di un file, la terminazione lascia il file in uno stato scorretto; analogamente, se il processo si trova nel mezzo dell'aggiornamento di dati condivisi mentre è in possesso di un lock mutex, il sistema deve reimpostare lo stato del lock come disponibile, sebbene non sia possibile offrire alcuna garanzia sull'integrità dei dati condivisi.

Se si adopera il metodo di terminazione parziale, occorre determinare quale processo, o quali processi in situazione di stallo devono essere terminati. Analogamente ai problemi di scheduling della CPU, si tratta di seguire un criterio. Le considerazioni sono essenzialmente economiche: si dovrebbero arrestare i processi la cui terminazione causa il minimo costo.

Sfortunatamente, il termine costo minimo non è preciso.

La scelta dei processi è influenzata da diversi fattori, tra cui i seguenti:

1. la priorità del processo;
2. il tempo trascorso in computazione e il tempo necessario per completare il compito assegnato al processo;
3. la quantità e il tipo di risorse impiegate dal processo (per esempio, se si può effettuare facilmente la prelazione delle risorse);
4. la quantità di ulteriori risorse di cui il processo ha ancora bisogno per completare l'esecuzione;
5. il numero di processi che si devono terminare;
6. il processo è iterativo o batch?

## Prelazione delle risorse

Per eliminare uno stallo si può esercitare la prelazione sulle risorse: le risorse si sottraggono in successione ad alcuni processi e si assegnano ad altri finché si ottiene l'interruzione del ciclo di stallo.

Se per gestire le situazioni di stallo si impiega la prelazione, si devono considerare i seguenti problemi:

1. *Selezione di una vittima*: occorre stabilire quali risorse e quali processi si devono sottoporre a prelazione. Come per la terminazione dei processi, è necessario stabilire l'ordine di prelazione allo scopo di minimizzare i costi. I fattori di costo possono includere parametri come il numero delle risorse possedute da un processo in stallo, e la quantità di tempo già spesa durante l'esecuzione del processo.
2. *Ristabilimento di un precedente stato sicuro*: occorre stabilire cosa fare con un processo a cui è stata sottratta una risorsa. Poiché è stato privato di una risorsa necessaria, la sua esecuzione non può continuare normalmente, quindi dev'essere ricondotto a un precedente stato sicuro (rollback) dal quale essere riavviato. Poiché, in generale, è difficile stabilire quale stato sia sicuro, la soluzione più semplice consiste nel terminare il processo e quindi riavivarlo. Benché sia più efficace effettuare il rollback solo fino al punto necessario allo scioglimento della situazione di stallo, questo metodo richiede che il sistema mantenga più informazioni sullo stato di tutti i processi in esecuzione.
3. *Attesa indefinita (starvation)*: è necessario assicurare che non si verifichino situazioni d'attesa indefinita, occorre cioè garantire che le risorse non siano sottratte sempre allo stesso processo.

In un sistema in cui la scelta della vittima avviene soprattutto secondo fattori di costo, può accadere che si scelga sempre lo stesso processo; in questo caso il processo non riesce mai a completare il suo compito; si tratta di una situazione d'attesa indefinita che ogni sistema reale deve saper affrontare. Chiaramente è necessario assicurare che un processo possa essere prescelto come vittima solo un numero finito (e ridotto) di volte; la soluzione più diffusa prevede *l'inclusione del numero di rollback tra i fattori di costo*.

## Cos'è un deadlock?

O situazione di stallo, si ha quando in un ambiente con multiprogrammazione, più processi competono per ottenere una risorsa, ma se essa ed altre risorse non sono disponibili, poiché trattenute da altri processi, essi entrano in uno stato di attesa o stallo. Per evitare una situazione di stallo, dobbiamo garantire la non presenza di queste condizioni contemporaneamente:

- mutua esclusione, cioè almeno una risorsa non dev'essere condivisibile
- possesso e attesa, un processo in possesso di almeno una risorsa attende di acquisire risorse già in possesso di altri processi
- impossibilità di prelazione: non esiste alcun diritto di prelazione sulle risorse
- attesa circolare: deve esistere un insieme di processi, tali che un processo attende una risorsa posseduta da un secondo processo, ma quest'ultimo attenda una risorsa posseduta da un altro processo e così via

Altra risposta:

In un ambiente con multiprogrammazione, più thread possono competere per ottenere un numero finito di risorse -> se una risorsa non è disponibile, il thread richiedente passa allo stato di attesa. In alcuni casi, se le risorse richieste sono trattenute da altri thread, a loro volta nello stato di attesa, il thread potrebbe non cambiare più il suo stato. Queste situazioni sono dette di stallo.

I deadlock sono per sempre, se prima o poi il processo prende la risorsa non ci troviamo in un deadlock. C'è un deadlock solo se i processi sono bloccati per sempre e deve intervenire il sistema operativo.

Per esserci un deadlock devono verificarsi contemporaneamente 4 condizioni:

1. mutua esclusione
2. possesso e attesa
3. assenza di prelazione
4. attesa circolare

## Come evitare i deadlock?

Gli algoritmi di prevenzione delle situazioni di stallo si basano sul controllo delle modalità di richiesta, così da assicurare che non si possa verificare almeno una delle condizioni necessarie perché si abbia uno stallo. Questo metodo può però causare uno scarso utilizzo dei dirpositivi e una ridotta produttività del sistema. Un metodo alternativo per evitare le situazioni di stallo consiste nel richiedere ulteriori informazioni sulle modalità di richiesta delle risorse. Il modello più semplice e più utile richiede che ciascun processo dichiari il numero massimo di risorse di ciascun tipo di cui necessiti. Data un'informazione a priori per ogni processo sul massimo numero di risorse richiedibili per ciascun tipo, si può implementare un algoritmo capace di assicurare che il sistema non entri in stallo. Questo algoritmo definisce un metodo per evitare lo stallo, ed esamina dinamicamente lo stallo di assegnazione delle risorse per garantire che non possa esistere una condizione di attesa circolare.

## Algoritmi di prevenzione degli stalli

Affinché si abbia uno stallo si devono verificare le quattro condizioni necessarie, perciò si può prevenire il verificarsi di uno stallo assicurando che almeno una di queste condizioni non possa capitare.

- Mutua esclusione: deve valere la condizione di mutua esclusione per le risorse non condivisibili. Le risorse condivisibili, invece, non richiedono l'accesso mutuamente esclusivo, perciò non possono essere coinvolte in uno stallo. I file aperti per la sola lettura sono un esempio di risorsa condivisibile.
- Possesso e attesa: per assicurare che la condizione di possesso e attesa non si presenti mai nel sistema, occorre garantire che un processo che richiede una risorsa non ne possieda altre. Si può usare un protocollo che ponga la condizione che ogni processo, prima di iniziare la propria esecuzione, richieda tutte le risorse che gli servono e che esse gli siano state assegnate. Un protocollo alternativo è quello che

permette a un processo di richiedere risorse solo se non ne possiede: un processo può richiedere risorse e adoperarle, ma prima di richiederne altre deve rilasciare tutte quelle che possiede.

- Assenza di prelazione: per assicurare che questa condizione non persista, si può impiegare il seguente protocollo. Se un processo che possiede una o più risorse ne chiede un'altra che non gli si può assegnare immediatamente, allora si eseguita la prelazione su tutte le risorse attualmente in suo possesso. Si ha, cioè, il rilascio implicito di queste risorse, che si aggiungono alla lista delle risorse che il processo sta attendendo
- Attesa circolare: per prevenire il verificarsi di situazioni di stallo si può considerare il seguente protocollo. Ogni processo può richiedere risorse solo seguendo un ordine crescente di numerazione. Ciò significa che un processo può richiedere inizialmente qualsiasi numero di istanza di un tipo di risorsa

## Cos'è uno stato sicuro?

Uno stato si dice sicuro se il sistema è in grado di assegnare risorse a ciascun processo in un certo ordine e impedire il verificarsi di uno stallo. Un sistema si trova in uno stato sicuro solo se esiste una sequenza sicura. Uno stato sicuro non è di stallo; uno stato di stallo è uno stato non sicuro.

## Cos'è l'algoritmo del banchiere?

Quando si presenta, un nuovo processo deve dichiarare il numero massimo delle istanze di ciascun tipo di risorsa di cui necessita. Questo numero non può superare il numero totale delle risorse del sistema.

Quando un utente richiede un gruppo di risorse, si deve stabilire se l'assegnazione di queste risorse lasci il sistema in uno stato sicuro. Se si

rispetta tale condizione, si assegnano le risorse, altrimenti il processo deve attendere che qualche altro processo ne rilasci un numero sufficiente.

La realizzazione dell'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema (vettore di risorse disponibili, matrice del massimo che una risorsa può ottenere, matrice delle risorse assegnate, matrice che indica la necessità residua di risorse)

## A cosa serve l'algoritmo del banchiere?

L'algoritmo del banchiere fa della gestione dei deadlock un particolare caso in cui stiamo approcciando la risoluzione tramite la deadlock avoidance. Quindi il suo scopo è far in modo che il nostro sistema non entri mai in uno stato non sicuro.

Il banchiere deve distribuire i fondi alle persone per realizzare dei progetti, questi soldi alla fine devono tornare al banchiere. Il banchiere quindi non deve mai assegnare tutto il denaro disponibile.

Va controllato che un processo, quando richiede delle risorse, non ne chieda un numero più grande di quelle disponibili e che l'assegnazione di tali risorse non porti il sistema in uno stato non sicuro.

Le strutture dati necessarie sono (sia  $n$  il numero di processi e  $m$  il numero di tipi delle risorse):

1. Disponibili: un vettore di lunghezza  $m$  che indica il numero di istanze disponibili per ciascuna risorsa
2. Massimo: una matrice  $n \times m$  che definisce la richiesta massima di ciascun processo
3. Assegnate: una matrice  $n \times m$  che definisce il numero di istanze di ciascun tipo di risorsa attualmente assegnate ad un processo
4. Necessità: una matrice  $n \times m$  che indica la necessità residua di risorse relativa ad ogni processo

# Capitolo 9 - Sistemi Operativi

## Memoria centrale

### Introduzione

La memoria è fondamentale nelle operazioni di un moderno sistema di calcolo. La memoria consiste in un grande vettore di byte, ciascuno con il proprio indirizzo. La CPU preleva le istruzioni dalla memoria sulla base del contenuto del program counter; tali istruzioni possono determinare ulteriori letture (load) e scritture (store) in specifici indirizzi di memoria.

La memoria non comprende il significato del flusso di indirizzi, non si interessa della loro origine o a che cosa servano (dati e istruzioni). E' possibile ignorare come un programma generi un indirizzo di memoria, così da prestare attenzione solo alla sequenza degli indirizzi di memoria generati dal programma in esecuzione.

La CPU può accedere solo ai registri e alla memoria principale, non al disco; preleva le istruzioni dalla memoria sulla base del contenuto del program counter. I dati devono essere, quindi, caricati in memoria.

L'unità di memoria vede solo una sequenza di indirizzi e le richieste che arrivano dalla CPU:

- richieste di lettura (load)
- richieste di dati e scrittura (store)

L'accesso ai registri è fatto in un solo ciclo del clock della CPU. La memoria principale può metterci molti cicli, dato che si accede ad essa tramite una transazione sul bus di memoria. In tal caso, il processore entra necessariamente in stallo, poiché mancano dei dati richiesti per completare l'istruzione che sta eseguendo.

*Dato che gli accessi alla memoria sono frequenti, si utilizza un buffer intermedio (la cache), in modo da garantire un accesso più rapido -> cosa realizzata totalmente in hardware senza l'input del sistema operativo.*

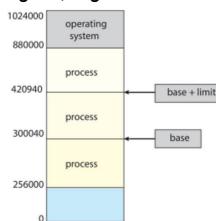
### Hardware di base e protezione

Non basta prestare attenzione alle velocità relative di accesso alla memoria fisica, ma occorre anche assicurare una corretta esecuzione delle operazioni. A tal fine, bisogna proteggere il sistema operativo dall'accesso dei processi utente e, in sistemi multiutente, salvaguardare i processi utenti l'uno dell'altro. Tale protezione dev'essere messa in atto a livello hardware, perché solitamente il sistema operativo, per una questione di prestazioni, non interviene negli accessi della CPU alla memoria.

Innanzitutto, bisogna assicurarsi che ciascun processo abbia uno spazio di memoria separato, in modo da proteggere i processi l'uno dell'altro: ciò è fondamentale per avere più processi caricati in memoria per l'esecuzione concorrente.

Per separare gli spazi di memoria occorre poter determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi.

Si può implementare il *meccanismo di protezione tramite due registri, registro base e registro limite*.



Il registro base contiene il più piccolo indirizzo legale della memoria fisica; il registro limite determina la dimensione dell'intervallo ammesso.

Per mettere in atto il meccanismo di protezione, la CPU confronta ciascun indirizzo generato in modalità utente con i valori contenuti nei due registri. Qualsiasi tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di memoria riservata ad altri utenti comporta l'invio di un'eccezione (trap) che restituisce il controllo al sistema operativo che, a sua volta, interpreta l'evento come un errore fatale.

(Esempio: segmentation fault che segnala il compilatore quando si sfonda un array).

*Questo schema impedisce a qualsiasi programma utente di alterare il codice o le strutture dati del sistema operativo o degli altri utenti.*

Solo il sistema operativo può caricare i registri base e limite, grazie a una speciale istruzione privilegiata. Dal momento che le istruzioni privilegiate possono essere eseguite unicamente in modalità kernel, e poiché solo il sistema operativo può essere eseguito in tale modalità, tale schema gli consente di modificare il valore di questi registri, ma impedisce tale operazione ai programmi utente.

Grazie all'esecuzione in modalità kernel, il sistema operativo ha la possibilità di accedere indiscriminatamente sia alla memoria a esso riservata sia a quella riservata agli utenti. Questo privilegio consente al sistema di caricare i programmi utente nelle aree di memoria a loro riservate e di fornire molti servizi.

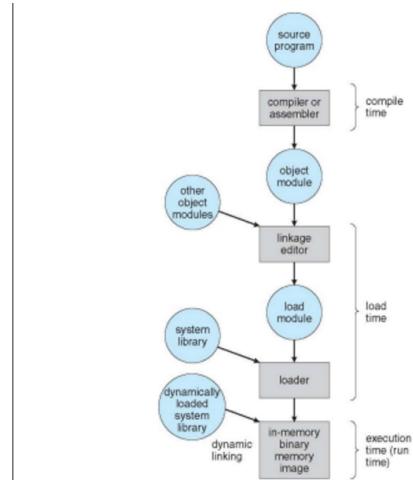
### Associazione degli indirizzi

In genere un programma risiede in un disco sotto forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria e inserito nel contesto di un processo, dove diventa idoneo per l'esecuzione su una CPU disponibile.

Il processo durante l'esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina si dichiara disponibile il suo spazio di memoria.

La maggior parte dei sistemi consente ai processi utente di risiedere in qualsiasi parte della memoria fisica, quindi, anche se lo spazio di indirizzi del calcolatore comincia all'indirizzo 00000, il primo indirizzo del processo utente non deve necessariamente essere quello.

Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari passi. In questi passi gli indirizzi sono rappresentabili in modi diversi.



Generalmente gli indirizzi del programma sorgente sono simbolici. Un compilatore di solito associa (bind) questi indirizzi simbolici a indirizzi rilocabili. L'editore dei collegamenti (o loader) fa corrispondere a sua volta questi indirizzi rilocabili a indirizzi assoluti. Ogni associazione rappresenta una corrispondenza da uno spazio d'indirizzi a un altro.

L'associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi passo del seguente percorso:

- **Compilazione:** se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare **codice assoluto**. Se, per esempio, è noto a priori che un processo utente inizia alla locazione R, anche il codice generato dal compilatore comincia da quella locazione. Se, in un momento successivo, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice.
- **Caricamento:** se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare **codice rilocabile**. In questo caso si ritarda l'associazione finale degli indirizzi alla fase del caricamento. Se l'indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato.
- **Esecuzione:** se durante l'esecuzione il processo può essere spostato da un segmento di memoria all'altro, si deve ritardare l'associazione degli indirizzi fino alla fase d'esecuzione. Per realizzare questo schema è necessario disporre di hardware specializzato. La maggior parte dei sistemi operativi general-purpose impiega questo metodo.

## Spazi di indirizzi logici e fisici

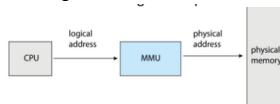
Un *indirizzo generato dalla CPU* è chiamato **indirizzo logico**, mentre un *indirizzo visto dall'unità di memoria*, cioè caricato nel *registro dell'indirizzo di memoria (Memory Access Register, MAR)* è chiamato **indirizzo fisico**.

I puntatori in C, ad esempio, contengono gli indirizzi logici.

I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Nella fase di esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici.

L'insieme di tutti gli indirizzi logici generati da un programma è lo **spazio degli indirizzi logici**; l'insieme di tutti gli indirizzi fisici corrispondenti a tali indirizzi logici è lo **spazio degli indirizzi fisici**.

L'associazione nella fase di esecuzione degli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto **unità di gestione della memoria**.



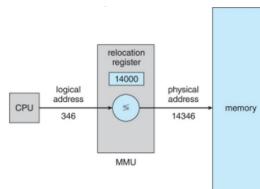
Il registro base è ora denominato *registro di rilocazione*: quando un processo utente genera un indirizzo, prima dell'invio all'unità di memoria, si somma a tale indirizzo il valore contenuto nel registro di rilocazione.

Per esempio, se il registro di rielocazione contiene il valore 14000, un tentativo da parte dell'utente di accedere alla locazione 0 è dinamicamente rilocato alla locazione 14000; un accesso alla locazione 346 è mappato alla locazione 14346.

Il programma utente non vede mai i reali indirizzi fisici. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò sempre come il numero 346. Solo quando assume il ruolo di un indirizzo di memoria (magari in una load o una store indiretta), si riloca il numero sulla base del contenuto del registro di rilocazione. **Il programma utente tratta indirizzi logici, l'architettura del sistema converte gli indirizzi logici in indirizzi fisici.**

La locazione finale di un riferimento a un indirizzo di memoria non è determinata finché non si compie effettivamente il riferimento.

In questo caso esistono due diversi tipi di indirizzi: gli indirizzi logici (nell'intervallo da 0 a max) e gli indirizzi fisici (nell'intervallo da r+0 a r+max per un valore di base r).



Il programma utente genera solo indirizzi logici e pensa che il processo sia eseguito dalle posizioni da 0 a max. Tuttavia questi indirizzi logici devono essere mappati in indirizzi fisici prima di essere usati. Il concetto di spazio di indirizzi logici mappato su uno spazio di indirizzi fisici è fondamentale per una corretta gestione della memoria.

## Linking dinamico

Si carica una procedura solo quando viene richiamata: tutte le procedure si tengono su disco in un formato di caricamento rilocabile. Il vantaggio è che una procedura viene caricata solo quando serve; può essere molto utile per le procedure che servono per gestire casi meno frequenti, come gli errori. Anche se un programma ha dimensioni elevate, posso tenere in memoria solo una piccola parte.

## Allocazione contigua della memoria

Metodo di allocazione della memoria.

La memoria centrale di solito si divide in due partizioni: una per il sistema operativo residente e una per i processi utente. Il sistema operativo si può collocare sia nella parte bassa sia nella parte alta della memoria. Questa decisione dipende da molti fattori, per esempio dalla posizione del vettore delle interruzioni. Molti sistemi operativi (inclusi Linux e Windows) collocano il sistema operativo in memoria alta.

Generalmente si vuole che più processi utenti risiedano contemporaneamente in memoria centrale. Perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda d'ingresso che attendono di essere caricati in memoria. Con l'allocazione contigua della memoria, ciascun processo è contenuto in una singola sezione di memoria contigua a quella che contiene il processo successivo.

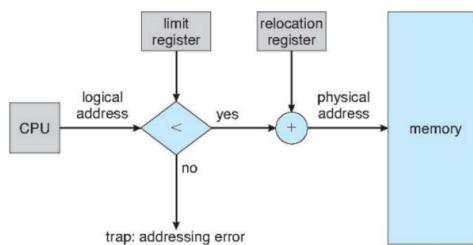
## Protezione della memoria

Se abbiamo un sistema con un registro di rielocazione e un registro limite abbiamo già raggiunto il nostro obiettivo.

Il registro di rielocazione contiene il valore dell'indirizzo fisico minore; il registro limite contiene l'intervallo di indirizzi logici.

Ogni indirizzo logico deve cadere nell'intervallo specificato dal registro limite; la MMU fa corrispondere dinamicamente l'indirizzo fisico all'indirizzo logico sommando a quest'ultimo i valori contenuti nel registro di rielocazione e invia l'indirizzo risultante alla memoria.

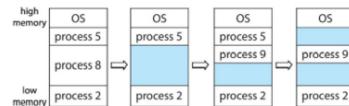
Lo schema con registro di rielocazione consente al sistema operativo di cambiare dinamicamente le proprie dimensioni; può essere utile in certe situazioni come creare spazio di memoria per i driver dei dispositivi (li carico in memoria solo quando mi servono effettivamente) ed eliminarlo quando non mi servono più.



## Allocazione della memoria

Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in più partizioni di dimensione variabile, dove ciascuna partizione può contenere esattamente un processo.

In questo schema a partizione variabile il sistema operativo conserva una tabella in cui sono indicate le partizioni di memoria disponibili e quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utenti; si tratta di un grande blocco di memoria disponibile, un buco. Nel lungo periodo, la memoria contiene una serie di buchi di diverse dimensioni.



Quando i processi entrano nel sistema, il sistema operativo prende in considerazione i loro requisiti di memoria e la quantità di spazio di memoria disponibile e determina a quali processi allocare la memoria. Quando gli viene assegnato dello spazio, un processo viene caricato in memoria e può quindi competere per il controllo della CPU. Quando termina, rilascia la memoria che gli era stata assegnata e il sistema operativo può impiegarla per un altro processo.

Che cosa succede quando non c'è memoria sufficiente per soddisfare le esigenze di un processo in arrivo? Un'opzione semplice è rifiutare il processo e fornire un messaggio di errore appropriato. In alternativa, possiamo inserire il processo in una coda di attesa. Quando, in seguito, la memoria viene rilasciata, il sistema operativo controlla la coda di attesa per determinare se può soddisfare le richieste di memoria di un processo in attesa.

In generale, è sempre presente un insieme di buchi di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un buco di dimensioni sufficienti per contenerlo. Se è troppo grande, il buco viene diviso in due parti: si

assegna una parte al processo in arrivo e si riporta l'altra nell'insieme dei buchi. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nell'insieme dei buchi; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formarne uno più grande.

Questa procedura è una particolare istanza del più generale problema di *allocazione dinamica della memoria*, che consiste nel soddisfare una richiesta di dimensione n data una lista di buchi liberi.

Le soluzioni sono numerose. I criteri più usati per scegliere un buco libero tra quelli disponibili nell'insieme sono i seguenti:

- *First-fit*: si assegna il primo buco abbastanza grande. La ricerca può cominciare sia dall'inizio dell'insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi.
- *Best-fit*: si assegna il più piccolo buco in grado di contenere il processo. Si deve compiere la ricerca in tutta la lista, a meno che questa non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più piccole.
- *Worst-fit*: si assegna il buco più grande. Anche in questo caso si deve esaminare tutta la lista, a meno che non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più grandi, che possono essere più utili delle parti più piccole ottenute con il criterio best-fit.

Con l'uso di simulazioni si è dimostrato che sia first-fit che best-fit sono migliori rispetto a worst-fit in termini di risparmio di tempo e di utilizzo di memoria. D'altra parte, nessuno dei due è chiaramente migliore dell'altro in termini di utilizzo di memoria ma, in genere, first-fit è più veloce.

## Frammentazione

Entrambi i criteri di allocazione dinamica della memoria soffrono di frammentazione.

- *Frammentazione esterna*: caso best-fit e first-fit. Ci sono dei buchi troppo piccoli per essere riempiti: ho lo spazio necessario, ma non in celle di memoria contigue. Lo spazio di memoria non è più continuo; ho memoria disponibile ma non utilizzabile.
- *Frammentazione interna*: la memoria assegnata è più grande di poco della memoria richiesta. Ho memoria libera ma che è segnata come utilizzata.

L'analisi statistica dell'algoritmo first-fit rivela che, pur con alcune ottimizzazioni, per n blocchi assegnati, si perdono altri 0.5 n blocchi a causa della frammentazione, ciò significa che potrebbe essere inutilizzabile un terzo della memoria -> *regola del 50 per cento*.

Il metodo generale per superare il problema della frammentazione interna prevede di suddividere la memoria fisica in blocchi di dimensione fissa, che costituiscono le unità d'allocazione. Con questo metodo la memoria assegnata può essere leggermente maggiore della memoria richiesta.

Una soluzione al problema della frammentazione esterna è data dalla *compattazione*. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grande blocco. La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è effettuata nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si effettua nella fase di esecuzione.

Se gli indirizzi sono rilocati dinamicamente, la rilocazione richiede solo lo spostamento del programma e dei dati, e quindi la modifica del registro di rilocazione in modo che rifletta il nuovo indirizzo di base. Quando è possibile eseguire la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria: tutti i buchi si spostano nell'altra direzione formando un grosso buco di memoria. Questo metodo può essere assai oneroso.

Un'altra possibile soluzione del problema della frammentazione esterna è data dal consentire la non contiguità degli indirizzi logici di un processo, permettendo così di assegnare la memoria fisica ai processi dovunque essa sia disponibile. Questa è la strategia utilizzata nella paginazione, la più comune tecnica di gestione della memoria nei sistemi elaborativi.

## Paginazione

La paginazione è uno *schema di gestione della memoria che consente allo spazio di indirizzamento fisico di un processo di non essere contiguo*. La paginazione evita la frammentazione esterna e la necessità di compattazione.

Visti i numerosi vantaggi offerti, paginazione, nelle sue varie forme, viene utilizzata nella maggior parte dei sistemi operativi, da quelli destinati a server di grandi dimensioni a quelli per dispositivi mobili. L'implementazione della paginazione è frutto della cooperazione tra il sistema operativo e l'hardware del computer.

## Metodo di base

Il metodo di base per implementare la paginazione consiste nel *suddividere la memoria fisica in blocchi di dimensione fissa, detti frame, e nel suddividere la memoria logica in blocchi di pari dimensione, detti pagine*.

Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria o dal file system. La memoria ausiliaria è divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria o di blocchi composti da più frame.

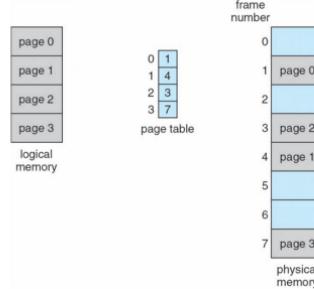
Questa idea piuttosto semplice fornisce grandi funzionalità e ha diverse ramificazioni. Per esempio, ora lo spazio degli indirizzi logici è totalmente separato dallo spazio degli indirizzi fisici e dunque un processo può avere uno spazio degli indirizzi logici a 64 bit anche se il sistema ha meno di  $2^{64}$  byte di memoria fisica.

Ogni indirizzo generato dalla CPU è diviso in due parti: un *numero di pagina (p)* e un *offset di pagina (d)*.

page number	page offset
p	d

- Il *numero di pagina* serve come *indice per la tabella delle pagine relativa al processo*.
- La *tabella delle pagine* contiene l'*indirizzo di base di ciascun frame nella memoria fisica*.
- L'*offset di pagina* è la *posizione del frame a cui viene fatto riferimento*.

- L'indirizzo base del frame viene combinato con l'offset della pagina per definire l'indirizzo di memoria fisica.



Descriviamo di seguito i passaggi adottati dalla MMU (Unità di gestione della memoria) per tradurre un indirizzo logico generato dalla CPU in un indirizzo fisico:

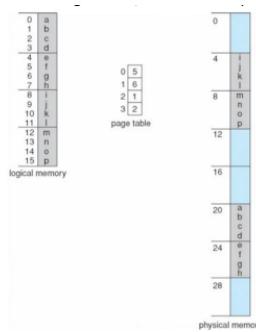
- Estrarre il numero di pagina  $p$  e utilizzarlo come indice nella tabella delle pagine
- Estrarre il numero di frame  $f$  corrispondente dalla tabella delle pagine
- Sostituire il numero di pagina  $p$  nell'indirizzo logico con il numero di frame  $f$

La dimensione di una pagina, così come quella di un frame, è definita dall'hardware ed è, in genere, una potenza di 2 compresa tra 4KB e 1GB, a seconda dell'architettura.

La scelta di una potenza di 2 come dimensione della pagina facilita la traduzione di un indirizzo logico nei corrispondenti numero di pagina e offset di pagina.

Se la dimensione dello spazio degli indirizzi logici è  $2^m$  e la dimensione di una pagina è di  $2^n$  byte, allora gli  $m-n$  bit più significativi di un indirizzo logico indicano il numero di pagina e gli  $n$  bit meno significativi indicano l'offset di pagina.

### Esempio di paginazione



Nell'indirizzo logico,  $n=2$  e  $m=4$ . Utilizzo una pagina di 4 byte e una memoria fisica di 32 byte (8 pagine).

- L'indirizzo logico 0 è la pagina 0 con offset 0; la pagina 0 si trova nel frame 5. Quindi all'indirizzo logico 0 corrisponde l'indirizzo fisico 20 [=  $(5 \times 4) + 0$ ]
- All'indirizzo logico 3 (pagina 0, offset 3) corrisponde l'indirizzo fisico 23 [=  $(5 \times 4) + 3$ ]
- All'indirizzo logico 4 (pagina 1, offset 0) corrisponde l'indirizzo fisico [=  $(6 \times 4) + 0$ ]

La paginazione non è altro che una forma di rilocazione dinamica: ad ogni indirizzo logico l'architettura di paginazione fa corrispondere un indirizzo fisico. L'uso della tabella delle pagine è simile all'uso di una tabella di registri base (o di rilocazione).

Con la paginazione si evita la frammentazione esterna: qualsiasi frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia si può avere la frammentazione interna.

I frame si assegnano come unità. Poiché in generale lo spazio di memoria richiesto da un processo non è multiplo della dimensione delle pagine, l'ultimo frame assegnato può non essere completamente pieno. Il caso peggiore si ha con un processo che necessita di  $n$  pagine più un byte: si assegnano  $n+1$  frame, quindi si ha una frammentazione interna di quasi un intero frame.

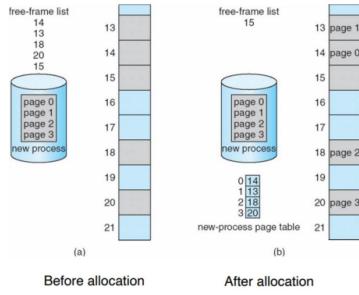
Se la dimensione del processo è indipendente dalla dimensione della pagina, si deve prevedere una frammentazione interna media di mezza pagina per processo.

Questa considerazione suggerisce che conviene utilizzare pagine di piccole dimensioni; tuttavia, a ogni elemento della tabella delle pagine è associato un overhead che si riduce all'aumentare della dimensione delle pagine. Inoltre, con un maggior numero di dati da trasferire, l'I/O sul disco è più efficiente. Generalmente, nel tempo la dimensione delle pagine è cresciuta col crescere dei processi, dei dati e della memoria centrale; attualmente la dimensione tipica delle pagine è compresa tra 4KB e 8 KB.

La paginazione ci consente di utilizzare una memoria fisica che è decisamente più grande rispetto a quella che potrebbe essere indicizzata dalla lunghezza del puntatore agli indirizzi della CPU.

Quando si deve eseguire un processo, il sistema esamina la sua dimensione espressa in pagine. Poiché ogni pagina del processo necessita di un frame, se il processo richiede  $n$  pagine, devono essere disponibili almeno  $n$  frame che, se ci sono, vengono assegnati al processo. Si carica la prima pagina in uno dei frame assegnati e si inserisce il numero del frame nella tabella delle pagine relativa al processo in questione.

La pagina successiva si carica in un altro frame e, anche in questo caso, s'inserisce il numero del frame nella tabella delle pagine e così via:



Un aspetto importante della paginazione è la netta distinzione tra la memoria vista dal programmatore e l'effettiva memoria fisica: il programmatore vede la memoria come un unico spazio contiguo, contenente solo il programma stesso; in realtà, il programma utente è sparso in memoria fisica contenente anche altri programmi. La differenza tra la memoria vista dal programmatore e la memoria fisica è colmata dall'hardware di traduzione degli indirizzi, che fa corrispondere gli indirizzi fisici agli indirizzi logici generati dai processi utente. Questa corrispondenza non è visibile ai programmatori ed è controllata dal sistema operativo.

Si noti che un processo utente non può accedere alle zone di memoria che non gli appartengono. Non ha modo di accedere alla memoria oltre quel che è previsto dalla sua tabella delle pagine e tale tabella contiene soltanto le pagine che appartengono al processo.

Poiché il sistema operativo gestisce la memoria fisica, dev'essere informato dei dettagli dell'allocazione: quali frame sono assegnati, quali disponibili, il loro numero totale e così via. In genere queste informazioni sono contenute in una struttura dati chiamata *tavella dei frame*, contenente un elemento per ogni frame, indicante se sia libero oppure assegnato e, se è assegnato, a quale pagina di quale processo o di quali processi.

Inoltre, il sistema operativo deve sapere che i processi utente operano nello spazio utente e tutti gli altri indirizzi logici si devono far corrispondere a indirizzi fisici. Se un utente usa una syscall e fornisce un indirizzo come parametro, si deve tradurre questo indirizzo nell'indirizzo fisico corretto. Il sistema operativo conserva una copia della tabella delle pagine per ciascun processo, così come conserva una copia dei valori contenuti nel program counter e nei registri. Questa copia si usa per tradurre gli indirizzi logici in indirizzi fisici ogni volta che il sistema operativo deve associare esplicitamente un indirizzo fisico ad un indirizzo logico. La stessa copia è usata anche dal dispatcher della CPU per impostare l'hardware di paginazione quando a un processo sta per essere assegnata la CPU. *La paginazione fa quindi aumentare la durata dei cambi di contesto*.

## Supporto hardware alla paginazione

Poiché ogni processo ha la sua tabella delle pagine, un puntatore alla tabella di pagina viene memorizzato insieme ai valori degli altri registri (come il puntatore alle istruzioni) nel process control block di ciascun processo. Quando lo scheduler della CPU seleziona un processo per l'esecuzione, deve ricaricare i registri utente e i valori appropriati della tabella delle pagine hardware dalla tabella delle pagine memorizzata.

L'implementazione hardware della tabella delle pagine si può realizzare in modi diversi. Nel caso più semplice è implementata come una *serie di registri hardware dedicati ad alta velocità*, così da rendere la traduzione dell'indirizzo molto efficiente. Tuttavia, questo approccio aumenta il tempo dei cambi di contesto, poiché durante un cambio di contesto ogni registro dovrà essere scambiato.

L'uso dei registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, nell'ordine, per esempio, di 256 elementi. Però la maggior parte dei calcolatori utilizza tabelle molto grandi, quindi non si possono impiegare i registri veloci per realizzare la **tavella delle pagine**; quest'ultima \*viene invece mantenuta nella memoria principale e un registro di base della tabella delle pagine\* punta alla tabella stessa. Il cambio della tabella delle pagine richiede soltanto di modificare questo registro, riducendo considerevolmente il tempo dei cambi di contesto.

- *Page table base register (PTBR)*: punta alla tabella
- *Page table lenght register (PTLR)*: indica la lunghezza della tabella

## TLB (Translation Look-aside Buffer)

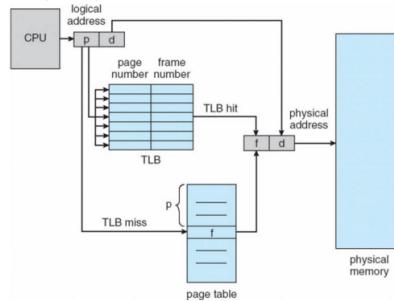
La memorizzazione della tabella delle pagine nella memoria principale può favorire cambi di contesto più rapidi, ma può anche comportare tempi di accesso alla memoria più lenti. Supponiamo di voler accedere alla posizione  $i$ . Per farlo, occorre far riferimento alla tabella delle pagine usando il valore contenuto nel PTBR (registro di base della tabella delle pagine) aumentato dell'offset relativo alla pagina  $i$ , perciò si deve accedere alla memoria. Si ottiene il numero del frame che, associato all'offset rispetto all'inizio della pagina, produce l'indirizzo cercato; a questo punto è possibile accedere alla posizione di memoria desiderata. Con questo metodo sono necessari due accessi alla memoria per accedere ai dati (uno per l'elemento della tabella delle pagine ed uno per il dato effettivo). L'accesso alla memoria è quindi rallentato di un fattore 2, un ritardo considerato intollerabile nella maggior parte delle circostanze.

La soluzione tipica a questo problema consiste nell'impiego del TLB, una *speciale piccola cache hardware*. Il TLB è una *memoria associativa ad alta velocità* in cui ogni elemento consiste di due parti: una *chiave* (o *tag*) e un *valore*. Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. La ricerca è molto rapida e in un hardware moderno è parte della pipeline delle istruzioni: non induce dunque nessuna penalizzazione in termini di prestazioni.

Per poter eseguire la ricerca in uno stadio della pipeline, tuttavia, il TLB dev'essere di dimensioni ridotte, in genere contenute tra le 32 e le 1024 voci. Alcune CPU implementano TLB separate per dati e istruzioni, in modo da poter raddoppiare il numero di voci TLB disponibili, poiché le due ricerche vengono effettuate in diversi stadi della pipeline.

*Il TLB si usa insieme con le pagine nel modo seguente: il TLB contiene una piccola parte della tabella delle pagine; quando la CPU genera un indirizzo logico, si presenta il suo numero di pagina al TLB; se tale numero è presente, il corrispondente numero del frame è immediatamente disponibile e si usa per accedere alla memoria. Se nel TLB non è presente il numero di pagina (TLB miss) si deve consultare la tabella delle pagine in memoria.*

A seconda della CPU, questa operazione può essere effettuata automaticamente a livello hardware oppure per mezzo di un interrupt al sistema operativo. Il numero del frame così ottenuto viene usato per accedere alla memoria. Inoltre, i numeri della pagina e del frame vengono inseriti nel TLB, e al riferimento successivo la ricerca sarà molto più rapida.



Se il TLB è già pieno di elementi, occorre sceglierne uno per sostituirlo. I criteri di sostituzione variano dalla scelta dell'elemento usato meno recentemente(LRU), a una politica round robin, fino alla scelta casuale. Alcuni TLB consentono che certi elementi siano vincolati, cioè che non si possano rimuovere dal TLB; in genere si vincolano gli elementi per il codice chiave del kernel.

Alcuni TLB memorizzano gli *identificatori dello spazio d'indirizzi* (ASID) in ciascun elemento del TLB. Un ASID identifica in modo univoco ciascun processo e si usa per fornire al processo corrispondente la protezione del suo spazio d'indirizzi. Quando tenta di trovare i valori corrispondenti ai numeri delle pagine virtuali, il TLB si assicura che l'ASID per il processo attualmente in esecuzione corrisponda all'ASID associato alla pagina virtuale.

La mancata corrispondenza dell'ASID viene trattata come un TLB miss. Oltre a fornire la protezione dello spazio d'indirizzi, l'ASID consente che il TLB contenga nello stesso istante elementi di diversi processi. Se il TLB non permette l'uso di ASID distinti, ogni volta che si seleziona una nuova tabella delle pagine, per esempio ad ogni cambio di contesto, si deve cancellare il TLB (TLB flush), in modo da assicurare che il successivo processo in esecuzione non faccia uso di errate informazioni di traduzione. Potrebbero altrimenti esserci vecchi elementi del TLB contenenti indirizzi virtuali validi, ma con indirizzi fisici corrispondenti sbagliati o non validi, lasciati dal precedente processo.

E' questo il motivo per cui il context switch è lento: all'inizio avrà solo dei TLB miss.

La percentuale di volte che il numero di pagina di interesse si trova nel TLB è detta *tasso di successi* (hit ratio).

Esempio:

se sono necessari 10 nanosecondi per accedere alle memoria, allora un accesso alla memoria mappata nel TLB richiede 10 nanosecondi. Se, invece, il numero non è contenuto nel TLB, occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero del frame (10 nanosecondi), quindi accedere al byte desiderato in memoria (10 nanosecondi); in totale sono necessari 20 nanosecondi. Stiamo supponendo che una ricerca nella tabella delle pagine richieda un solo accesso alla memoria, ma, come vedremo, potrebbero talvolta essere necessari più accessi. Per calcolare il *tempo effettivo di accesso alla memoria* occorre tener conto della probabilità dei due casi:

$$\text{tempo effettivo d'accesso} = 0.80 * 10 + 0.20 * 20 = 12 \text{ nanosecondi}$$

In questo esempio si verifica un rallentamento del 20 per cento nel tempo medio d'accesso alla memoria (da 10 a 12 nanosecondi).

I TLB sono una caratteristica hardware, quindi cosa c'entrano con lo studio del sistema operativo?

I progettisti hanno bisogno di capire la funzione e le caratteristiche del TLB, che variano a seconda della piattaforma hardware. Per un funzionamento ottimale, i progettisti di un sistema operativo per una data piattaforma devono implementare la paginazione basandosi sul progetto del TLB della piattaforma. Analogamente, un cambiamento nel progetto del TLB (per esempio, tra generazioni diverse di CPU Intel) può rendere necessaria una modifica nell'implementazione della paginazione dei sistemi operativi.

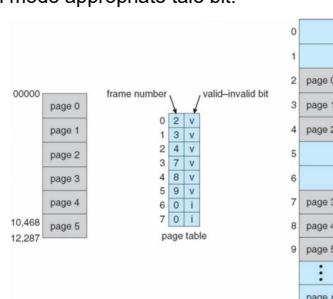
## Protezione

In un ambiente paginato, la protezione della memoria è assicurata dai *bit di protezione associati ad ogni frame*; normalmente, tali bit si trovano nella tabella delle pagine.

Un bit può determinare se una pagina si può leggere e scrivere oppure soltanto leggere. Mentre si calcola l'indirizzo fisico, si possono controllare i bit di protezione per verificare che non si scriva in una pagina di sola lettura. Un tale tentativo genera un'eccezione hardware per il sistema operativo, dato che sarebbe una violazione della protezione della memoria.

Possiamo estendere questo principio per avere un livello di protezione più raffinato: si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto *bit di validità*. Tale bit, impostato a valido, indica che la pagina corrispondente è nello spazio d'indirizzi logici del processo, quindi è una pagina valida; impostato a non valido, indica che la pagina non è nello spazio d'indirizzi logici del processo.

Il bit di validità consente quindi di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso un'eccezione. Il sistema operativo concede o impedisce l'accesso a una pagina impostando in modo appropriato tale bit.



L'immagine mostra un processo di frammentazione interna: lo spazio indirizzi del processo si estende fino all'indirizzo 10,468, ma dato che i

riferimenti alla pagina 5 sono considerati come validi, ciò rende validi gli accessi sino all'indirizzo 12,287. Tale problema è dovuto alla dimensione delle pagine di 2KB.

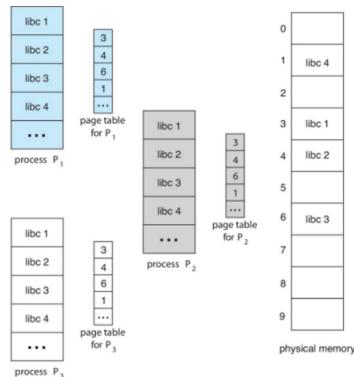
Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, infatti molti processi utilizzano solo una piccola frazione dello spazio d'indirizzi di cui dispongono. In questi casi è un inutile spreco creare una tabella di pagine con elementi per ogni pagina dell'intervallo di indirizzi, poiché una grande parte di questa tabella resta inutilizzata e occupa prezioso spazio di memoria.

Alcune architetture dispongono di *registri di lunghezza della tabella delle pagine (PTLR)*, per indicare le dimensioni della tabella. Questo valore si controlla rispetto a ogni indirizzo logico per verificare che quest'ultimo si trovi nell'intervallo valido per il processo. Un errore causa la generazione di un'eccezione per il sistema operativo.

## Pagine condivise

Un vantaggio della paginazione risiede nella possibilità di condividere codice comune, il che è particolarmente importante in un ambiente con più processi. Si consideri la libreria standard del C, che fornisce parte dell'interfaccia alla chiamata di sistema in molte versioni di UNIX e Linux. Su un tipico sistema Linux, la maggior parte dei processi utente richiede la libreria standard del C libc. Un'opzione pericolosa è che ogni processo carichi la propria copia di libc nel proprio spazio di indirizzamento. Se ho un sistema con 40 processi utente e la libreria è di 2MB, ciò richiederebbe 80MB di memoria.

Se il codice è rientrante può però essere condiviso.



In Figura si osservano tre processi che condividono le pagine della libreria libc.

Il *codice rientrante* è un codice non auto-modificante: *non cambia mai durante l'esecuzione*. Due o più processi possono quindi eseguire lo stesso codice allo stesso tempo. Ogni processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari per la propria esecuzione. I dati per due processi differenti saranno, ovviamente, diversi.

Solo una copia della libreria standard del C dev'essere conservata nella memoria fisica e la tabella delle pagine di ogni processo utente viene mappata sulla stessa copia fisica di libc. Quindi, per supportare 40 processi, abbiamo bisogno di una sola copia della libreria e lo spazio totale richiesto è di 2MB anziché di 80MB: risparmio significativo!

Oltre alle librerie run-time come libc, altri programmi d'uso frequente possono essere condivisi: compilatori, interfacce e finestre, sistemi di database e così via. Le librerie condivise sono in genere implementate con pagine diverse; per poter essere condiviso, il codice dev'essere rientrante. La natura di sola lettura del codice condiviso non deve essere lasciata alla correttezza intrinseca del codice, ma dev'essere fatta rispettare dal sistema operativo.

## Struttura della tabella delle pagine

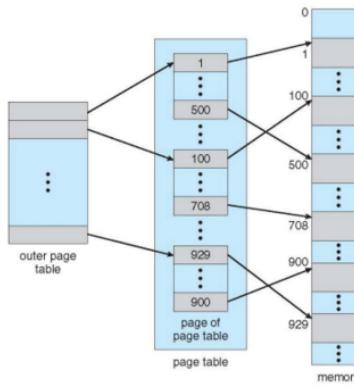
La maggior parte dei moderni calcolatori dispone di uno spazio d'indirizzi logici molto grande. In un ambiente di questo tipo la tabella delle pagine diventa eccessivamente grande, fino a sfiorare il milione di elementi.

Vi sono delle soluzioni alternative per strutturare la tabella delle pagine, al fine di dividere la tabella in tabelle più piccole ed occupare meno spazio in memoria.

## Paginazione gerarchica

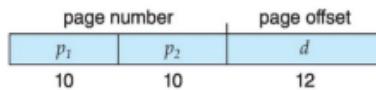
Consiste nell'adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata.

Entro prima nella tabella di primo livello che punta ad una tabella interna, che porta poi alla memoria.



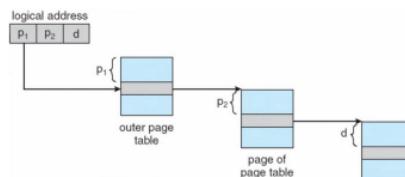
Se ho uno spazio interno vacante, avrò un sacco di tabelle inutilizzate, ma resta il fatto che ho più flessibilità. Il grande svantaggio è che rallenta il tempo di accesso: ogni qualvolta non trovo l'entry nel TLB devo fare tre accessi alla memoria e non due -> funziona comunque grazie all'alto hit rate del TLB.

Ciascun indirizzo logico è suddiviso in un numero di pagina di 20 bit e in un offset di pagina di 12 bit. Paginando la tabella delle pagine, anche il numero di pagina è a sua volta suddiviso in un numero di pagina di 10 bit e un offset di pagina di 10 bit. L'indirizzo logico è strutturato come segue:



dove  $p_1$  è un indice della tabella delle pagine di primo livello (o tabella esterna delle pagine) e  $p_2$  è l'offset all'interno della pagina indicata dalla tabella esterna delle pagine.

Poiché la traduzione degli indirizzi si svolge dalla tabella esterna delle pagine verso l'interno, questo metodo è anche noto come *tabella delle pagine ad associazione diretta*.



Lo schema di paginazione a due livelli non è più adatto nel caso di sistemi con uno spazio di indirizzi logici a 64 bit.

La soluzione ovvia per evitare una tabella tanto grande consiste nel suddividere la tabella in parti più piccole. Questo metodo si adotta anche in alcuni processori a 32 bit allo scopo di fornire una maggiore flessibilità ed efficienza. Questo però mi porta a dover fare ancora più accessi in memoria in caso di miss.

Le tabelle gerarchiche, per questo motivo, non sono considerate appropriate per i sistemi a 64 bit.

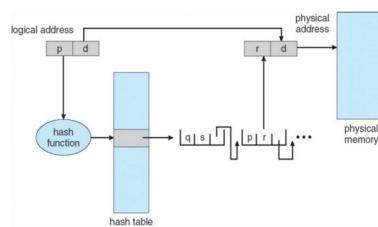
## Tabella delle pagine di tipo hash

Un metodo di gestione molto comune degli spazi di indirizzi oltre i 32 bit consiste nell'impiego di una tabella delle pagine di tipo hash, in cui l'argomento della funzione hash è il numero della pagina virtuale.

Per la gestione delle collisioni, ogni elemento della tabella hash contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa locazione.

Ciascun elemento è composto da 3 campi:

1. il numero della pagina virtuale
2. l'indirizzo del frame corrispondente alla pagina virtuale
3. un puntatore al successivo elemento della lista



L'algoritmo opera come segue:

- si applica la funzione hash al numero della pagina virtuale contenuto nell'indirizzo virtuale, identificando un elemento della tabella
- si confronta il numero di pagina virtuale con il campo 1 del primo elemento della lista concatenata corrispondente; se i valori coincidono, si usa l'indirizzo del relativo frame (campo 2) per generare l'indirizzo fisico desiderato. Altrimenti, l'algoritmo esamina allo stesso modo gli elementi successivi della lista concatenata

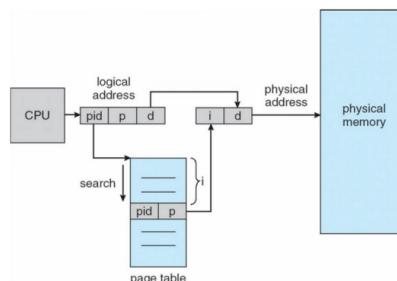
Per questo schema è stata proposta una variante, adatta a spazi di indirizzamento a 64 bit. Si tratta della *tabella delle pagine a gruppi*; la differenza è che ciascun elemento della tabella hash contiene i riferimenti alle pagine fisiche corrispondenti a un gruppo di pagine virtuali contigue. Le tabelle delle pagine a gruppi sono particolarmente utili per gli spazi d'indirizzi sparsi, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio di indirizzi.

## Tabella delle pagine invertita

Generalmente, si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, ossia vi sono elementi corrispondenti a ogni indirizzo virtuale a prescindere dalla validità di quest'ultimo. Questa rappresentazione tabellare è naturale poiché i processi fanno riferimento alle pagine tramite gli indirizzi virtuali delle pagine stesse, che il sistema operativo deve poi tradurre in indirizzi di memoria fisica. Poiché la tabella è ordinata per indirizzi virtuali, il sistema operativo può calcolare in che punto della tabella si trova l'indirizzo fisico associato, e usare direttamente tale valore. Uno degli inconvenienti insiti in questo metodo è che ciascuna tabella delle pagine può contenere milioni di elementi e occupare grandi quantità di memoria fisica, necessaria solo per sapere com'è impiegata la rimanente memoria fisica. Per risolvere questo problema si fa uso della tabella delle pagine invertita.

Una tabella delle pagine inverita *ha un elemento per ogni pagina reale (o frame)*.

Ciascun elemento è quindi costituito dell'indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi, nel sistema esiste una sola tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica.



In Figura sono mostrate le operazioni di una tabella delle pagine invertita.

Nel sistema esiste quindi una sola tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica. Anziché avere una traduzione pagina-frame per ogni processo, ho una tabella unica in cui viene indicato a chi appartiene il frame.

E' detta invertita perché devo scandire (non in sequenza perché è hardware, ho bisogno dell'hashing) nella ricerca nella tabella invece di arrivarci direttamente.

Le tabelle invertite richiedono spesso la memorizzazione di un identificatore dello spazio di indirizzi in ciascun elemento della tabella delle pagine, perché essa contiene di solito molti spazi di indirizzi diversi associati alla memoria fisica; l'identificatore garantisce che una data pagina logica relativa a un certo processo sia associata alla pagina fisica corrispondente.

Ogni elemento della tabella delle pagine invertita è una coppia <id-processo, numero di pagina> dove l'id assume il ruolo di identificatore dello spazio d'indirizzi. Se si trova una corrispondenza nella tabella dell'elemento i, si genera l'indirizzo fisico <i, offset>, in caso contrario è stato tentato un accesso illegale.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta il tempo di ricerca nella tabella quando si fa riferimento ad una pagina. Poiché la tabella invertita è ordinata per indirizzi fisici, mentre le ricerche si fanno per indirizzi virtuali, per trovare una corrispondenza potrebbe essere necessario scorrere tutta la tabella.

Per limitare il problema, però, si può impiegare una tabella hash anche qui. Naturalmente, ogni accesso alla tabella hash aggiunge al procedimento un riferimento alla memoria, quindi un riferimento alla memoria virtuale richiede almeno due letture dalla memoria reale. Posso usare una TLB per migliorare le prestazioni.

Una questione interessante relativa alle tabelle delle pagine invertite riguarda la memoria condivisa. Con la paginazione standard, ogni processo ha una propria tabella delle pagine, il che consente di mappare più indirizzi virtuali sullo stesso indirizzo fisico. Questo metodo non può essere utilizzato con le tabelle delle pagine invertite.

Infatti, poiché esiste una sola pagina virtuale per ogni pagina fisica, una pagina fisica non può avere due o più indirizzi virtuali condivisi.

Con le tabelle delle pagine invertite, in un dato istante si può dunque avere una sola mappatura di un indirizzo virtuale sull'indirizzo fisico. Un riferimento da parte di un altro processo che condivide la memoria provocherà un errore di pagina e sostituirà la mappatura con un indirizzo virtuale diverso.

## Segmentazione

La paginazione permette di ridurre la quantità di memoria sprecata, ma non è l'unica soluzione possibile. Prima di essa, si utilizzò la segmentazione.

Schema di gestione della memoria che supporta la visione della memoria dell'utente. Dà alla memoria una divisione in segmenti in cui in ogni segmento c'è roba che sta bene insieme (non è garantito con la paginazione).

Un *programma* è una *collezione di segmenti*. I *segmenti* sono un'*unità logica* come la memoria, una procedura, una funzione, un metodo, un oggetto, ecc.

Fregatura: i segmenti sono di dimensione variabile -> quando vado ad allargare i vari segmenti in memoria, mi trovo il problema della gestione dinamica della memoria anche se ridotto, poiché i segmenti sono più facili da gestire. NON CI SONO SISTEMI OPERATIVI CHE UTILIZZANO LA SEGMENTAZIONE.

## Architettura per la segmentazione

Essendo i segmenti di dimensioni variabili, non posso più contare sulle potenze di due.

Struttura dell'indirizzo: <numero del segmento, offset>

Tabella dei segmenti: anche qui ho un indirizzo di base e la lunghezza. La base contiene un indirizzo fisico iniziale dove il segmento si trova in

memoria. La lunghezza specifica la lunghezza del segmento.

*Entro con il numero di segmento, prendo l'indirizzo di inizio, sommo l'offset e ottengo l'indirizzo finale.*

Segment Table Base Register (STBR): corrisponde al PTBR, punta allocazione in memoria del segmento della tabella.

Segment Table Length Register (STLR): indica il numero di segmenti utilizzato dal programma; il segmento numero s è legale se  $s > STLR$ .

## Protezione

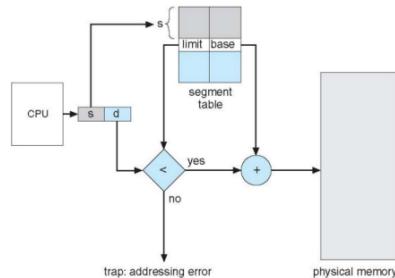
E' semplice da implementare perché posso indicare se un segmento è read-only o anche solo eseguibile.

Bit di validità pari a 0: segmento illegale.

Il bit di protezione è associato ai segmenti, la condivisione del codice avviene a livello di segmento.

Dato che i segmenti variano in lunghezza, l'allocazione in memoria è un problema di allocazione dinamica.

Esempio di segmentazione (traduzione di un indirizzo logico in un indirizzo fisico fatto a segmenti):

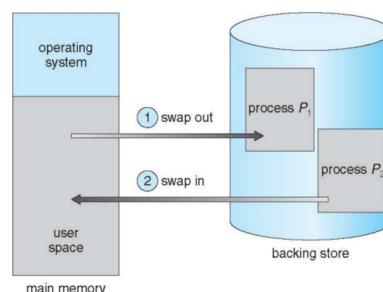


Problema: gestione dell'allocazione dello spazio libero.

## Avvicendamento dei processi (Swapping)

Le istruzioni di un processo e i dati su cui operano per essere eseguiti devono essere in memoria. Tuttavia, un processo o una sua parte, può essere rimosso temporaneamente dalla memoria centrale (mediante un'operazione detta di swap out), spostato in memoria ausiliaria (backing store) e in seguito riportato in memoria (swap in) per continuare la sua esecuzione.

Grazie a questo procedimento, detto **avvicendamento dei processi**, lo spazio totale degli indirizzi fisici di tutti i processi può eccedere la reale dimensione della memoria fisica del sistema, aumentando così il grado di multiprogrammazione possibile.



## Avvicendamento standard

Riguarda spostamenti di interi processi tra la memoria centrale e una memoria ausiliaria solitamente costituita da un veloce dispositivo di memorizzazione secondaria.

Anche le strutture dati associate al processo devono essere scritte all'interno di questa memoria. In un processo multithread devono essere spostate anche tutte le strutture dati relative ad ogni thread.

Il vantaggio dell'avvicendamento standard è che consente di sovrascrivere la memoria fisica, in modo che il sistema possa ospitare più processi rispetto alla quantità di memoria fisica effettivamente a disposizione.

I processi inattivi o raramente attivi sono buoni candidati per l'avvicendamento e la memoria allocata a questi processi inattivi può quindi essere destinata a processi attivi. Se un processo inattivo che è stato rimosso dalla memoria centrale diventa di nuovo attivo, dev'essere portato di nuovo in memoria e ripristinato.

## Avvicendamento con paginazione

L'avvicendamento standard è stato utilizzato nei sistemi UNIX tradizionali, ma in genere oggi non è più utilizzato nei sistemi operativi contemporanei. Il motivo è che la quantità di tempo necessaria per spostare interi processi tra due memorie è proibito.

La maggior parte dei sistemi, compresi Linux e Windows, usa ora una variante dello swapping standard in cui è possibile spostare solo alcune pagine di un processo, piuttosto che il processo intero.

Questa strategia consente tuttavia di sovrascrivere la memoria fisica, ma non comporta il costo di spostamento di interi processi, poiché presumibilmente solo un numero di pagine limitato sarà coinvolto nello scambio.

Un'operazione di page out sposta una pagina dalla memoria centrale a quella ausiliaria, mentre il processo inverso è noto come page in.

In ogni caso, lo swap avviene solo quando la memoria disponibile è molto poca.

## Cambio di contesto con l'avvicendamento

Se il nuovo processo da mettere nella CPU non è in memoria, c'è bisogno di fare uno swap out su un processo e fare lo swap in sul processo da eseguire. Il tempo del cambio di contesto può essere molto elevato in questo caso. Un processo di 100MB viene caricato sull'hard disk con una frequenza di trasferimento di 50 MB/sec. Il tempo di caricamento è di 2000 ms, a cui aggiungo il tempo dello swap in. In totale il tempo del cambio di contesto arriva ad essere di 4 secondi! Se sono a conoscenza della memoria utilizzata, posso ridurre la quantità di memoria trasferita: System call usate per informare il sistema della memoria utilizzata: `request_memory()` e `release_memory()`.

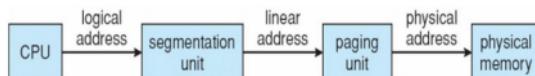
Altro problema relativo all'avvicendamento: Se un processo sta facendo delle operazioni di I/O non può essere caricato sulla memoria ausiliaria, al limite posso trasferire l'I/O sullo spazio kernel, poi al device I/O. Tecnica conosciuta come Double buffering, aggiunge overhead

## Segmentazione con paging

La gestione della memoria nei sistemi IA-32 è suddivisa in due componenti: segmentazione e paginazione.

La CPU genera indirizzi logici che vengono passati all'unità di segmentazione. L'unità di segmentazione produce un indirizzo lineare per ogni indirizzo logico. L'indirizzo lineare viene quindi passato all'unità di paginazione, che genera l'indirizzo fisico nella memoria principale.

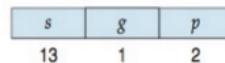
Dunque, l'unità di segmentazione e l'unità di paginazione formano insieme l'equivalente dell'unità di gestione della memoria (MMU).



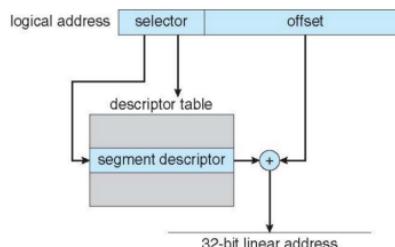
## Segmentazione in Intel IA-32

Nell'architettura IA-32, un segmento può raggiungere la dimensione massima di 4 GB; il numero massimo di segmenti per processo è pari a 16K.

Lo spazio di indirizzi logici di un processo è composto da due partizioni: la prima contiene fino a 8K segmenti riservati al processo; la seconda contiene fino a 8K segmenti condivisi fra tutti i processi. Le informazioni riguardanti la prima partizione sono contenute nella tabella locale dei descrittori (local descriptor table, LDT), quelle relative alla seconda partizione sono memorizzate nelle tabelle globali dei descrittori (global descriptor table, GDT). Ciascun elemento nella LDT e nella GDT è lungo 8 byte e contiene informazioni dettagliate riguardanti uno specifico segmento, oltre agli indirizzi base e limite. Un indirizzo logico è una coppia (selettore, offset), dove il selettore è un numero di 16 bit.

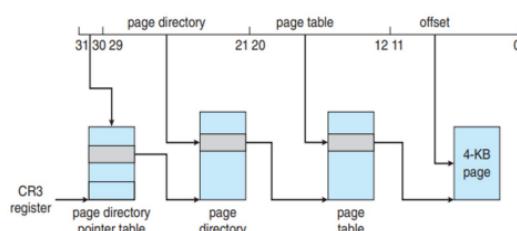


Nella figura: *s* indica il nome del segmento, *g* indica se il segmento si trova nella GDT o nella LDT e *p* contiene informazioni relative alla protezione. L'offset è un numero di 32 bit che indica la posizione del byte all'interno del segmento in questione. La macchina ha sei registri di segmento che consentono a un processo di far riferimento contemporaneamente a sei segmenti; inoltre possiede sei registri di microprogramma di 8 byte per i corrispondenti descrittori prelevati dalla LDT o dalla GDT. Questa cache evita alla macchina di dover prelevare dalla memoria i descrittori per ogni riferimento in memoria. Un indirizzo lineare di IA-32 è lungo 32 bit e si genera come segue: il registro di segmento punta all'elemento appropriato all'interno della LTD della GDT; le informazioni relative alla base e al limite di tale segmento si usano per generare un indirizzo lineare. Innanzitutto si usa il valore del limite per controllare la validità dell'indirizzo; se non è valido, si ha un errore di riferimento alla memoria che causa la generazione di un'eccezione e la restituzione del controllo al sistema operativo; altrimenti si somma il valore al valore della base ottenendo un indirizzo lineare di 32 bit.



## Page Address Extension (PAE)

Non appena gli sviluppatori di software hanno iniziato a soffrire della limitazione della memoria a 4 GB imposta dall'architettura a 32 bit, Intel ha introdotto l'estensione di indirizzo della pagina, che consente ai processori a 32 bit di accedere a uno spazio di indirizzamento fisico più grande di 4 GB punto la differenza fondamentale introdotta dal supporto PAE è era il passaggio nella paginazione di uno schema a due livelli a uno schema a tre livelli, i cui i cui primi due bit fanno riferimento a una tabella di puntatori directory di pagina. L'immagine illustra un sistema PAE compagini di 4 KB. PAE supporta anche pagine di 2 MB. Con PAE è stato inoltre aumentata la dimensione degli elementi della directory delle pagine della tabella delle pagine che passa da 32 a 64 bit permettendo di estendere l'indirizzo di base delle tabelle della pagina e dei frame da 20 a 24 bit. In combinazione con i 12 bit di offset, l'aggiunta del supporto PAE a IA-32 ha aumentato lo spazio di indirizzamento a 36 bit, garantendo il supporto di un massimo di 64 GB di memoria fisica. E' importante notare che per utilizzare PAE è necessario il supporto del sistema operativo. Linux e Mac OS supportano PAE. Tuttavia, la versione a 32 bit dei sistemi operativi Windows per desktop supportano soltanto 4 GB di memoria fisica, anche se PAE è abilitato.

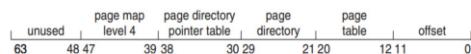


## Architettura x86-64

Lo sviluppo di architetture Intel a 64 bit ha avuto una storia curiosa. La prima di queste architetture era IA-64 (Itanium), ma questa architettura non ha avuto un'ampia diffusione. Nel frattempo, un altro produttore di chip - AMD - ha iniziato a sviluppare un'architettura a 64 bit nota come x86-64, basata sull'estensione del set di istruzioni IA-32 esistente. L'architettura x86-64 supportava spazi di indirizzamento logico e fisico molto più grandi e introduceva diverse altre novità architettoniche. Storicamente AMD aveva spesso sviluppato chip basati sull'architettura Intel, ma in questo caso i ruoli si sono invertiti e Intel ha adottato l'architettura x86-64 di AMD.

Il supporto a uno spazio di indirizzamento a 64 bit permette di indirizzare la straordinaria quantità di  $2^{64}$  byte di memoria. Tuttavia, anche se i sistemi a 64 bit possono potenzialmente indirizzare una tale quantità di memoria, nella pratica vengono utilizzati nei progetti attuali assai meno di 64 bit per la rappresentazione di un indirizzo. L'architettura x86-64 utilizza attualmente un indirizzo virtuale di 48 bit con supporto ai formati di pagina di 4KB, 2MB o 1GB utilizzando una paginazione a 4 livelli.

Rappresentazione dell'indirizzo lineare:



Poiché questo schema di indirizzamento può usare PAE, gli indirizzi virtuali sono di 48 bit, ma supportano indirizzi fisici a 52 bit (Terabyte).

## Come fanno i sistemi operativi a gestire la memoria?

Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in partizione di dimensione variabile. In questo schema a partizione variabile il sistema operativo conserva una tabella in cui sono indicate le partizioni di memoria disponibili e quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utente; si tratta di un grande blocco di memoria disponibile (buco). In ogni dato istante è sempre disponibile una lista delle dimensioni dei blocchi liberi e della coda di ingresso. Tuttavia questo procedimento non è del tutto efficiente. Esistono due tecniche di gestione della memoria: la paginazione e la segmentazione.

## Cos'è la frammentazione e come si può evitare?

Prima di tutto, esistono due tipologie di frammentazione: quella interna e quella esterna. La prima si ha quando viene assegnato un blocco di memoria troppo grande ad un processo e resta memoria inutilizzata. La frammentazione esterna si ha quando non ci sono blocchi disponibili della dimensione necessaria e quindi si assegna al processo un blocco di memoria non contiguo.

Il metodo generale per superare il problema della frammentazione interna prevede di suddividere la memoria fisica in blocchi di dimensione fissa, che costituiscono le unità d'allocazione. Con questo metodo la memoria assegnata può essere leggermente maggiore della memoria richiesta.

Una soluzione al problema della frammentazione esterna è data dalla *compattazione*. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grande blocco. La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è effettuata nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si effettua nella fase di esecuzione.

Se gli indirizzi sono rilocati dinamicamente, la rilocazione richiede solo lo spostamento del programma e dei dati, e quindi la modifica del registro di rilocazione in modo che rifletta il nuovo indirizzo di base. Quando è possibile eseguire la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria: tutti i buchi si spostano nell'altra direzione formando un grosso buco di memoria. Questo metodo può essere assai oneroso.

Un'altra possibile soluzione del problema della frammentazione esterna è data dal consentire la non contiguità dello spazio agli indirizzi logici di un processo, permettendo così di assegnare la memoria fisica ai processi dovunque essa sia disponibile. Questa è la strategia utilizzata nella paginazione, la più comune tecnica di gestione della memoria nei sistemi elaborativi.

## Cos'è la paginazione?

La paginazione è un metodo di gestione della memoria che permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. Il metodo di base per implementarla consiste nel suddividere la memoria fisica in blocchi di dimensioni costanti, detti frame o pagine fisiche e nel dividere la memoria logica in blocchi di pari dimensioni detti pagine. Quando si deve eseguire un processo, si carica le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria. Ogni indirizzo generato dalla CPU è diviso in due parti: un numero di pagina (p) e uno scostamento (offset) di pagina (d). Il numero di pagina serve come indice per la tabella delle pagine, contenente l'indirizzo di base in memoria fisica di ogni pagina. Questo indirizzo di base si combina con l'offset di pagina per definire l'indirizzo della memoria fisica, che si invia all'unità di memoria. La paginazione non è altro che una forma di rielocazione dinamica: a ogni indirizzo logico l'architettura di paginazione fa corrispondere un indirizzo fisico. Con la paginazione si può evitare la frammentazione esterna: qualsiasi frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia si può continuare ad avere la frammentazione interna.

## Cos'è il TLB? Quanto è grande?

La TLB è una memoria associativa ad alta velocità (cache di ricerca) in cui ogni suo elemento consiste di due parti: una chiave (tag) e un valore. Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. Una TLB può contenere dai 64 ai 1024 elementi. La si utilizza quando non sono disponibili le pagine di un processo e prima di prelevarle dalla memoria si controlla all'interno della cache.

## Cos'è la paginazione gerarchica?

E' una delle possibili strutture della tabella delle pagine; algoritmo di paginazione a due livelli in cui la tabella delle pagine è paginata. Paginando la tabella delle pagine, anche il numero di pagina è a sua volta suddiviso in un numero di pagina e in un offset di pagina. Poiché la traduzione degli indirizzi si svolge dalla tabella esterna delle pagine verso l'interno, questo metodo è anche noto come tabella delle pagine ad associazione diretta. Serve per ridurre la tabella delle pagine in un sistema con uno spazio degli indirizzi logici molto grandi (da  $2^{32}$  a  $2^{64}$  elementi).

## Come possono essere le tabelle delle pagine?

Possiamo avere tre tipologie di tabella delle pagine: gerarchica, di tipo hash e invertita.

In quella di tipo hash l'argomento della funzione hash è il numero della pagina. Nella tabella delle pagine invertita ho un elemento per ogni frame ed è costituito dall'indirizzo di pagina virtuale. Lo svantaggio di quest'ultima è che aumenta il tempo di ricerca di un elemento, in quanto è organizzata per indirizzi fisici mentre le ricerche vengono effettuate per indirizzi virtuali.

## Cos'è la segmentazione e perché non viene utilizzata?

La segmentazione è uno schema di gestione della memoria che consente di gestire questa rappresentazione della memoria dal punto di vista dell'utente. Uno spazio di indirizzi logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indirizzi specificano sia il nome sia l'offset all'interno del segmento, quindi l'utente fornisce ogni indirizzo come una coppia ordinata di valori: un nome di segmento e un offset. Questo contrasta con la paginazione in cui l'utente fornisce un indirizzo singolo, che l'architettura di paginazione suddivide in un numero di pagine e un offset, non visibili al programmatore. Per semplicità i segmenti sono numerati e ogni riferimento si compie per mezzo di un numero anziché di un nome; quindi un indirizzo logico è una coppia <numero di segmento, offset>. Non viene utilizzata perché la gestione della memoria è più complessa e subentra di nuovo il problema della frammentazione, dato l'uso del partizionamento dinamico.

## Come avviene la traduzione degli indirizzi?

Generalmente, l'associazione di istruzioni e dati a indirizzi di memoria si può compiere in questo modo:

1. Compilazione: se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare codice assoluto. Se, in un momento successivo, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice
2. Caricamento: se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare codice rilocabile. In questo caso si ritarda l'associazione finale degli indirizzi alla fase del caricamento
3. Esecuzione: se durante l'esecuzione il processo può essere spostato da un segmento di memoria all'altro, si deve ritardare l'associazione degli indirizzi fino alla fase di esecuzione. Per realizzare questo schema sono necessarie specifiche caratteristiche dell'architettura

## Differenza tra frame e pagina?

Le pagine servono per gli indirizzi logici che vengono divisi in page number ed offset. La tabella delle pagine è la struttura che memorizza la corrispondenza tra frame e numeri di pagina. Il frame è l'unità di gestione della memoria fisica.

## Tabella delle pagine

La tabella delle pagine contiene all'interno l'indirizzo di base di ogni frame nella memoria fisica. Combinata con l'indirizzo di base c'è il page offset per definire l'indirizzo fisico che è mandato all'unità di memoria.

## Come entrare nella tabella delle pagine?

La tabella delle pagine è una struttura dati utilizzata dal sistema operativo per gestire la memoria virtuale. Quando un processo accede alla memoria, il sistema operativo utilizza la tabella delle pagine per tradurre l'indirizzo virtuale in un indirizzo fisico.

Per entrare nella tabella delle pagine è necessario che il processo acceda alla memoria. Il sistema operativo utilizzerà quindi la tabella delle pagine per tradurre l'indirizzo virtuale in un indirizzo fisico e, se necessario, caricare la pagina richiesta dal disco. In seguito, la pagina viene mantenuta in memoria, pronta per essere utilizzata in caso di ulteriori accessi.

In sintesi, la tabella delle pagine viene utilizzata automaticamente dal sistema operativo ogni volta che un processo accede alla memoria. Non è necessario che l'utente o il programmatore facciano alcun tipo di interazione con la tabella delle pagine per entrarvi.

## Vantaggio utilizzo tabella delle pagine invertite

L'utilizzo della tabella delle pagine invertita presenta diversi vantaggi rispetto alla tradizionale tabella delle pagine:

1. Riduzione della dimensione della tabella: la tabella delle pagine invertita è più piccola rispetto alla tradizionale tabella delle pagine, poiché ogni entrata rappresenta un intervallo di indirizzi virtuali anziché un singolo indirizzo
2. Maggiore efficienza nell'utilizzo della memoria: più pagine possono essere allocate nella stessa zona di memoria fisica
3. Maggiore velocità nella traduzione dell'indirizzo: utilizza un algoritmo di ricerca più efficiente rispetto alla tradizionale tabella delle pagine, rendendo più veloce la traduzione dell'indirizzo virtuale in un indirizzo fisico
4. Maggiore flessibilità nella gestione della memoria: permette di gestire la memoria in modo flessibile poiché può essere utilizzata per implementare diverse politiche di gestione della memoria

In sintesi, l'utilizzo di una tabella delle pagine invertite permette di migliorare l'efficienza, la velocità e la flessibilità nella gestione della memoria rispetto alla tradizionale tabella delle pagine.



# Capitolo 10 - Sistemi Operativi

## Memoria virtuale

La memoria virtuale è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria.

Il vantaggio principale offerto da questa tecnica è quello di permettere che i programmi siano più grandi della memoria fisica; inoltre la memoria virtuale astrae la memoria centrale in un vettore di memorizzazione molto grande e uniforme, separando la memoria logica, com'è vista dall'utente, da quella fisica. Questa tecnica libera i programmatori dai problemi di limitazione della memoria. La memoria virtuale permette inoltre ai processi di condividere facilmente file e di realizzare memorie condivise e fornisce un meccanismo efficiente per la creazione dei processi. La memoria virtuale è però difficile da realizzare e, se usata scorrettamente, può ridurre di molto le prestazioni del sistema.

### Introduzione

Gli algoritmi di gestione della memoria sono necessari a causa di un requisito fondamentale: le istruzioni da eseguire si devono trovare all'interno della memoria fisica.

La condizione che le istruzioni debbano essere nella memoria fisica sembra tanto necessaria quanto ragionevole, ma purtroppo riduce le dimensioni dei programmi a valori strettamente correlati alle dimensioni memoria fisica.

Da un esame dei programmi reali, però, risulta che in molti casi non è necessario avere in memoria l'intero programma.

Esempi:

- Spesso i programmi dispongono di codice per la gestione di condizioni di errore insolite. Poiché questi errori non si verificano quasi mai, anche il relativo codice si esegue molto poco.
- Spesso ad array, liste e tabelle si assegna più memoria di quanta sia effettivamente necessaria.
- Alcune opzioni e caratteristiche di un programma sono utilizzabili solo di rado.

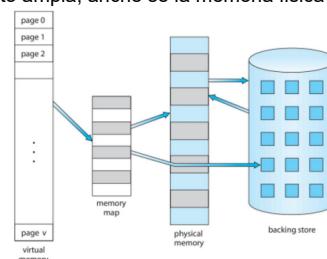
Anche nei casi in cui è necessario disporre di tutto il programma, è possibile che non serva tutto in una volta.

La possibilità di eseguire un programma che si trova solo parzialmente in memoria porterebbe molti benefici:

- Un programma non è più vincolato alla quantità di memoria fisica disponibile. Gli utenti possono scrivere programmi per uno spazio degli indirizzi virtuali molto grande, semplificando il compito della programmazione.
- Poiché ogni programma utente può impiegare meno memoria fisica, si possono eseguire molti più programmi contemporaneamente, ottenendo un corrispondente aumento dell'utilizzo e della produttività della CPU senza aumentare il tempo di risposta o di completamento.
- Per caricare ogni programma utente in memoria sono necessarie meno operazioni di I/O, quindi ogni programma utente è eseguito più rapidamente.

Questa è una forma di caching: la RAM non sarà nient'altro che una cache che contiene quello che mi serve per eseguire il programma.

La memoria virtuale si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica. Questa separazione permette di offrire ai programmatori una memoria virtuale molto ampia, anche se la memoria fisica disponibile è più piccola.



L'espressione *spazio degli indirizzi virtuali* si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Tipicamente, da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico e si estende in uno spazio di memoria contigua.

E' tuttavia possibile organizzare la memoria fisica in frame di pagine; in questo caso i frame delle pagine fisiche assegnati ai processi possono non essere contigui. Spetta all'unità di gestione della memoria (MMU) associare in memoria le pagine logiche alle pagine fisiche.

Uno spazio degli indirizzi virtuali che contiene buchi si definisce *sparsa*. Un simile spazio degli indirizzi è utile, poiché i buchi possono essere riempiti grazie all'espansione dei segmenti heap o stack, oppure se vogliamo collegare dinamicamente delle librerie (o altri oggetti condivisi) durante l'esecuzione del programma.

Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre il vantaggio di condividere i file e la memoria fra due o più processi, mediante la condivisione delle pagine.

### Paginazione su richiesta

La paginazione su richiesta consiste nel caricare le pagine nel momento in cui servono realmente; tecnica adottata comunemente dai sistemi con memoria virtuale.

Secondo questo schema, *le pagine sono caricate in memoria solo quando richieste durante l'esecuzione del programma*: ne consegue che le

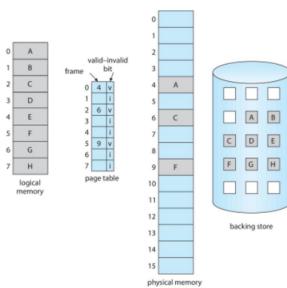
pagine cui non si accede mai non sono mai caricate nella memoria fisica. Un sistema di paginazione su richiesta è analogo a un sistema paginato con avvicendamento dei processi in memoria. I processi risiedono in memoria secondaria (generalmente su disco o su un altro supporto di memoria non volatile).

La paginazione su richiesta mostra uno dei principali vantaggi della memoria virtuale: *caricando solo le parti necessarie dei programmi, la memoria viene utilizzata in modo più efficiente.*

## Concetti fondamentali

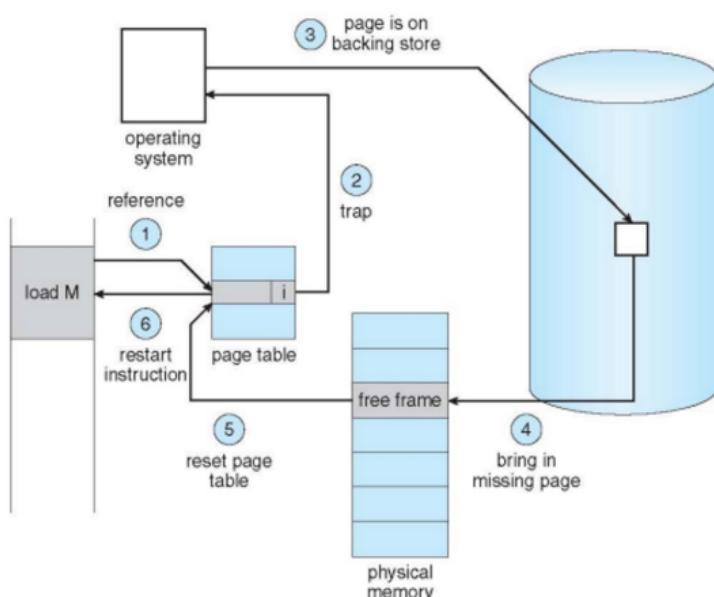
Mentre un processo è in esecuzione, alcune pagine saranno in memoria e altre si troveranno nella memoria secondaria. E' dunque necessaria una forma di supporto hardware per distinguere tra i due casi. A tale scopo si può utilizzare lo schema basato sul bit di validità. Il bit impostato come non valido indica che la pagina non è valida (cioè non appartiene allo spazio di indirizzi logici del processo) oppure è valida, ma è attualmente nella memoria secondaria. L'elemento della tabella delle pagine corrispondente a una pagina caricata in memoria s'impone come al solito, mentre l'elemento della tabella delle pagine corrispondente a una pagina che attualmente non è in memoria è semplicemente contrassegnato come non valido.

Occorre notare che indicare una pagina come non valida non sortisce alcun effetto se il processo non tenta mai di accedervi.



Che cosa succede se il processo tenta l'accesso ad una pagina che non era stata caricata in memoria? L'accesso ad una pagina contrassegnata come non valida causa un evento o un'eccezione *page fault*. L'hardware di paginazione, traducendo l'indirizzo attraverso la tabella delle pagine, nota che il bit non è valido e genera una trap per il sistema operativo; tale eccezione è dovuta a un insuccesso del sistema operativo nella scelta delle pagine da caricare in memoria.

La procedura di gestione dell'eccezione di page fault è lineare:



1. Si controlla una tabella interna per questo processo (in genere tale tabella è conservata insieme al blocco di controllo del processo) allo scopo di stabilire se il riferimento fosse un accesso alla memoria valido o non valido
2. Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua il caricamento
3. Si individua un frame libero, per esempio prelevandone uno dalla lista dei frame liberi
4. Si programma un'operazione sui dischi per trasferire la pagina desiderata nel frame appena assegnato
5. Quando la lettura del disco è completata, si modificano la tabella interna, conservata con il processo, e la tabella delle pagine per indicare che la pagina si trova attualmente in memoria
6. Si riavvia l'istruzione interrotta dall'eccezione. A questo punto il processo può accedere alla pagina come se questa fosse stata sempre presente in memoria

Come caso estremo, è possibile avviare l'esecuzione di un processo senza pagine in memoria. Quando il sistema operativo carica nel program counter l'indirizzo della prima istruzione del processo genera immediatamente un page fault. Una volta portata la pagina in memoria, il processo continua l'esecuzione, generando page fault fino a che tutte le pagine necessarie non si trovino effettivamente in memoria; a questo punto si può eseguire il processo senza ulteriori richieste. Lo schema descritto è una **paginazione su richiesta pura**, vale a dire che una pagina non si trasferisce mai in memoria se non viene richiesta.

In teoria alcuni programmi possono accedere a diverse nuove pagine di memoria all'esecuzione di ogni istruzione (una pagina per l'istruzione e meno per i dati), eventualmente causando più page fault per ogni istruzione. In un caso simile le prestazioni del sistema sarebbero inaccettabili. Fortunatamente, l'analisi dei programmi in esecuzione mostra che questo comportamento è estremamente improbabile.

I programmi tendono ad avere una *località dei riferimenti*, quindi le prestazioni della paginazione su richiesta risultano ragionevoli.

L'hardware di supporto alla paginazione su richiesta è lo stesso che è richiesto per la paginazione e l'avvicendamento dei processi in memoria:

- *tabella delle pagine*: questa tabella ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione;
- *memoria secondaria*: questa memoria conserva le pagine non presenti in memoria centrale. Generalmente, la memoria secondaria è costituita da un disco ad alta velocità detto dispositivo di swap; la sezione del disco usata a questo scopo si chiama *area di avvicendamento*.

Uno dei requisiti cruciali della paginazione su richiesta è la possibilità di rieseguire una qualunque istruzione dopo un page fault. Avendo salvato lo stato del processo interrotto (registro, codici di condizione, contatore di programma) al momento del page fault, occorrerà riavviare il processo esattamente dallo stesso punto e con lo stesso stato, eccezion fatta per la presenza della pagina desiderata in memoria. Nella maggior parte dei casi questo requisito è facile da soddisfare. Un page fault si può verificare per qualsiasi riferimento alla memoria. Se si verifica durante la fase di fetch (prelievo) di un'istruzione, l'esecuzione si può riavviare effettuando nuovamente il fetch. Se si verifica durante il fetch di un operando bisogna effettuare nuovamente fetch e decode dell'istruzione, quindi si può prelevare l'operando.

I processi destinati alla memoria virtuale non fanno modifiche permanenti alla memoria, dato che ogni volta devo rifare le istruzioni fino a che non l'ho terminata. E' l'unico modo per garantire che non avvengano page fault!

Visto che l'operazione di paginazione su richiesta richiede un disco è un'operazione lenta: la CPU nel frattempo passa ad un altro processo per evitare di attendere senza far nulla.

## **Lista dei frame liberi**

Quando si verifica un page fault, il sistema operativo deve spostare la pagina desiderata dalla memoria secondaria alla memoria principale. Per risolvere il page fault la maggior parte dei sistemi operativi mantiene una lista dei frame liberi, ovvero *un insieme di frame disponibili e utilizzabili per soddisfare le richieste*.

I frame liberi devono essere allocati anche quando lo stack o l'heap di un processo si espandono.

I sistemi operativi allocano generalmente i frame liberi usando una tecnica come *zero-fill-on-demand* (riempimento con zeri su richiesta): i frame vengono azzerati su richiesta prima di essere allocati, cancellando così il loro precedente contenuto (si considerino le potenziali implicazioni sulla sicurezza della non eliminazione del contenuto di un frame prima di riassegnarlo).

All'avvio di un sistema tutta la memoria disponibile viene inserita nella lista dei frame liberi. Man mano che vengono richiesti frame liberi, la dimensione della lista dei frame liberi si riduce. A un certo punto la lista diventa vuota, oppure la sua dimensione scende al di sotto di una certa soglia fissata: quando ciò si verifica la lista deve essere ripopolata.

## **Prestazioni della paginazione su richiesta**

La paginazione su richiesta può avere effetto sulle prestazioni di un calcolatore. Il motivo si può comprendere calcolando il tempo di accesso effettivo. Per calcolarlo è necessario conoscere il tempo necessario alla gestione di una page fault. In tal caso si deve eseguire la seguente sequenza:

1. trap per il sistema operativo
2. salvataggio dei registri utente e dello stato del processo
3. verifica che l'interruzione sia dovuta ad una page fault
4. controllo della correttezza del riferimento alla pagina e determinazione della locazione della pagina nel disco
5. lettura del disco e trasferimento in un frame libero:
  - a) attesa nella coda relativa a questo dispositivo finché la richiesta di lettura non sia servita
  - b) attesa del tempo di posizionamento e latenza del dispositivo
  - c) inizio del trasferimento in un frame libero
6. durante l'attesa, l'allocazione della CPU a un altro processo utente (scheduling della CPU, facoltativo)
7. ricezione di un'interruzione del controllore del disco (I/O completato)
8. salvataggio dei registri e dello stato dell'altro processo utente (se è stato eseguito il passo 6)
9. verifica della provenienza dell'interruzione dal disco
10. aggiornamento della tabella delle pagine e di altre tabelle per segnalare che la pagina richiesta è attualmente presente in memoria
11. attesa che la CPU sia nuovamente assegnata a questo processo
12. ripristino dei registri utente, dello stato del processo e della nuova tabella delle pagine, quindi ripresa dell'istruzione interrotta

Se queste operazioni vengono fatte velocemente il page fault non è così terribile. Il problema sorge quando il OS genera una raffica di page fault. In un sistema con paginazione su richiesta bisogna tenere basso il tasso di page fault.

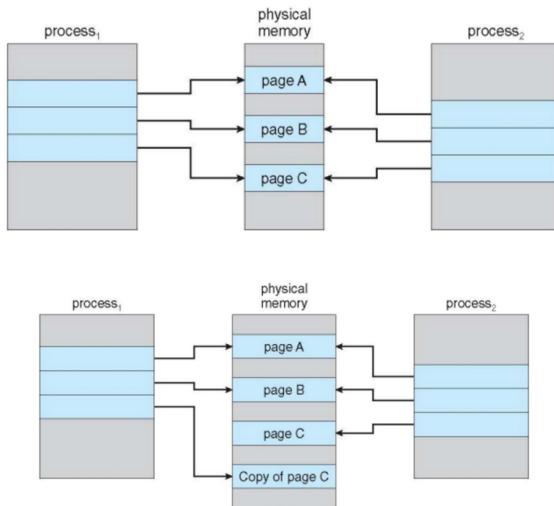
## **Copertura su scrittura (Copy-on-Write)**

Si è visto come un processo possa cominciare rapidamente l'esecuzione richiedendo solo la pagina contenente la prima istruzione.

La generazione dei processi tramite fork() può inizialmente evitare la paginazione su richiesta per mezzo di una tecnica simile alla condivisione delle pagine che garantisce la celere generazione dei processi riuscendo anche a minimizzare il numero di nuove pagine allocate al processo appena creato.

Si ricordi che la syscall fork() crea un processo figlio come duplicato del genitore. Nella sua versione originale la fork() creava per il figlio una copia dello spazio d'indirizzi del genitore, duplicando le pagine appartenenti al processo genitore. Considerando che molti processi figli eseguono subito dopo la loro creazione la chiamata di sistema exec(), questa operazione di copiatura può essere inutile.

Come alternativa, si può utilizzare una tecnica nota come *copy-on-write*, *il cui funzionamento si fonda sulla condivisione iniziale delle pagine da parte dei processi genitori e dei processi figli*. Le pagine condivise si contrassegnano come pagine da copiare su scrittura, a significare che, se un processo (genitore o figlio) scrive su una pagina condivisa, il sistema deve creare una copia di tale pagina.



Si consideri per esempio un processo figlio che cerchi di modificare una pagina contenente parti dello stack, quando le pagine sono contrassegnate come copy-on-write. Il sistema operativo crea una copia della pagina nello spazio degli indirizzi del processo figlio. Il processo figlio modifica la sua copia della pagina e non la pagina appartenente al processo genitore.

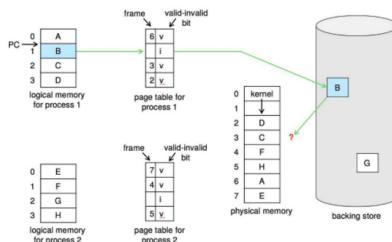
Adoperando la tecnica di copiatura su scrittura si copiano soltanto le pagine modificate da uno dei due processi, mentre tutte le altre sono condivisibili dai processi genitore e figlio.

Si noti che soltanto le pagine modificabili si devono contrassegnare come copy-on-write, mentre quelle che non si possono modificare (per esempio, le pagine contenenti codice eseguibile) sono condivisibili dai processi genitore e figlio.

## Sostituzione delle pagine

Si consideri che la memoria del sistema non si usa solo per contenere pagine di programmi: le aree di memoria per l'I/O impiegano una rilevante quantità di memoria. Ciò può aumentare la difficoltà degli algoritmi di allocazione della memoria. Decidere quanta memoria assegnare all'I/O e quanta alle pagine dei programmi è un problema complesso. Alcuni sistemi riservano una quota fissa di memoria per l'I/O, altri permettono sia ai processi utente sia al sottosistema di I/O di competere per tutta la memoria del sistema.

**Sovrallocazione:** durante l'esecuzione di un processo utente si verifica un page fault. Il sistema operativo determina la locazione del disco in cui risiede la pagina desiderata, ma poi scopre che la lista dei frame liberi è vuota: tutta la memoria è in uso.



A questo punto il sistema operativo può scegliere tra diverse possibilità, per esempio può terminare un processo utente. Tuttavia, la paginazione su richiesta è un tentativo che il sistema operativo fa per migliorare l'utilizzo e la produttività del sistema di calcolo. Gli utenti non dovrebbero sapere che i loro processi sono eseguiti su un sistema paginato. La paginazione deve essere logicamente trasparente per l'utente, quindi la terminazione del processo non costituisce la scelta migliore.

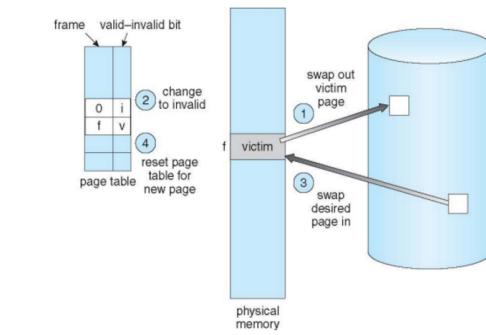
Il sistema operativo può scaricare dalla memoria un intero processo, liberando tutti i suoi frame e riducendo il livello di multiprogrammazione.

Tuttavia, lo swapping standard non è più utilizzato nella maggior parte dei sistemi operativi a causa del sovraccarico indotto dalla copia di interi processi tra la memoria e lo spazio di swap. La maggior parte dei sistemi operativi combina ora lo swapping con la sostituzione di una pagina.

## Sostituzione di pagina

**La sostituzione delle pagine segue il seguente criterio: se nessun frame è libero, ne viene liberato uno attualmente inutilizzato.**

E' possibile liberarlo scrivendo il suo contenuto nell'area di swap e modificando la tabella delle pagine (e tutte le altre tabelle) per indicare che la pagina non si trova più in memoria.



Il frame liberato si può utilizzare per memorizzare la pagina che ha causato il fault. Si modifica la procedura di servizio dell'eccezione di page fault in modo da includere la sostituzione della pagina:

1. si individua la locazione su disco della pagina richiesta;
2. si cerca un frame libero:
  - se esiste, lo si usa;
  - altrimenti si impiega un algoritmo di sostituzione delle pagine per scegliere un frame vittima;
  - si scrive la pagina vittima nel disco;
3. si scrive la pagina richiesta nel frame appena liberato; si modificano le tabelle delle pagine e dei frame;
4. si riprende il processo dal punto in cui si è verificato il page fault.

Occorre notare che, se non esiste alcun frame libero, sono necessari *due trasferimenti di pagine*, uno fuori e uno dentro la memoria. Questa situazione raddoppia il tempo di servizio del page fault e aumenta di conseguenza anche il tempo eddettivo d'accesso.

Questo sovraccarico si può ridurre utilizzando un *bit di modifica*. In questo caso l'hardware del calcolatore dispone di un bit di modifica, associato a ogni pagina (o frame), che viene posto a 1 ogni volta che nella pagina si scrive un byte, indicando che la pagina è stata modificata. Quando si sceglie una pagina da sovrascrivere si esamina il suo bit di modifica; se è a 1, significa che quella pagina è stata modificata rispetto a quando era stata letta dal disco; in questo caso la pagina deve essere scritta nel disco. Se il bit di modifica è rimasto a 0, significa che la pagina non è stata modificata da quando è stata caricata in memoria, quindi non è necessario scrivere nel disco la pagina di memoria: c'è già. Questa tecnica vale anche per le pagine di sola lettura, per esempio pagine di codice binario. Queste pagine non possono essere modificate, quindi si possono rimuovere in ogni momento.

*Questo schema può ridurre in modo considerevole il tempo per il servizio del page fault, poiché dimezza il tempo di I/O, se la pagina non è stata modificata.*

*La sostituzione di una pagina è fondamentale al fine della paginazione su richiesta, perché completa la separazione tra memoria logica e memoria fisica. Con questo meccanismo si può mettere a disposizione del programmatore una memoria virtuale enorme con una memoria fisica più piccola.*

Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali:

- occorre sviluppare un **algoritmo di allocazione dei frame**
- occorre sviluppare un **algoritmo di sostituzione delle pagine**

Ossia, se sono presenti più processi in memoria, occorre decidere quanti frame vadano assegnati a ciascun processo. Inoltre, quando è richiesta una sostituzione di pagina, occorre selezionare i frame da sostituire.

L'idea è quella di ottenere le migliori prestazioni possibili. Dovrei essere in grado di prevedere le pagine che mi serviranno in futuro in modo da minimizzare il page fault. Quindi, dovrei evitare di cacciare fuori pagine che mi serviranno in futuro.

Quello che mi aspetto è che nel momento in cui ho un numero maggiore di frame, ho un numero minore di page fault.

## Sostituzione delle pagine secondo l'ordine di arrivo (FIFO)

Algoritmo di sostituzione delle pagine più semplice: *associa ad ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo.*

Occorre notare che non è strettamente necessario registrare l'istante in cui si carica una pagina in memoria; infatti si può creare una coda FIFO di tutte le pagine presenti in memoria. In questo caso si sostituisce la pagina che si trova in testa alla coda. Quando si carica una pagina in memoria, la si inserisce nell'ultimo elemento della coda.

L'algoritmo FIFO di sostituzione delle pagine è facile da capire e da programmare; tuttavia le sue prestazioni non sono sempre buone. La pagina sostituita potrebbe essere un modulo di inizializzazione usato molto tempo prima e che non serve più, ma potrebbe contenere una variabile molto usata, inizializzata precedentemente, e ancora in uso.

Occorre notare che anche se si sceglie una pagina da sostituire che è in uso attivo, tutto continua a funzionare correttamente. Dopo aver rimosso una pagina attiva per inserirne una nuova, quasi immediatamente si verifica un'eccezione di page fault per riprendere la pagina attiva. Per riportare la pagina attiva in memoria è necessario sostituire un'altra pagina. *Quindi, una cattiva scelta della pagina da sostituire aumenta il tasso di page fault e reallenta l'esecuzione del processo, ma non causa errori.*

**Anomalia di Belady:** *con alcuni algoritmi di sostituzione delle pagine, il tasso di page fault può aumentare con l'aumento del numero di frame assegnati ai processi.* Impedisce al sistema di reagire ad un tasso di page fault molto elevato, anche se per un breve lasso di tempo.

L'anomalia si verifica quando l'algoritmo non è in grado di prevedere le pagine vittime e di conseguenza sceglie sempre quelle sbagliate.

L'anomalia è dovuta proprio alla natura FIFO dell'algoritmo.

A prima vista sembra logico supporre che fornendo più memoria a un processo le prestazioni di quest'ultimo migliorino. In alcune delle prime ricerche sperimentali si notò invece che questo presupposto non è sempre vero.

## Sostituzione ottimale delle pagine (teorico)

In seguito alla scoperta dell'anomalia di Belady si è ricercato un algoritmo ottimale di sostituzione delle pagine. Tale algoritmo è quello che fra tutti gli algoritmi presenta il tasso minimo di page fault e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato OPT o MIN.

Consiste semplicemente nel: **sostituire la pagina che non verrà usata per il periodo di tempo più lungo**.

L'uso di questo algoritmo di sostituzione delle pagine assicura il tasso minimo di page fault per un dato numero di frame.

L'algoritmo ottimale di sostituzione delle pagine è difficile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti. Quindi, esso si impiega soprattutto per studi comparativi.

## Sostituzione delle pagine usate meno recentemente (LRU)

Possiamo interpretare questo algoritmo come l'algoritmo ottimale ma con ricerca all'indietro invece che nel futuro: sostituisco la pagina che non è stata usata per il periodo più lungo.

La sostituzione LRU associa ad ogni pagina l'istante in cui è stata usata per l'ultima volta.

Quando occorre sostituire una pagina, l'algoritmo LRU sceglie quella che non è stata usata per il periodo più lungo.

Il criterio LRU si usa spesso come algoritmo di sostituzione delle pagine ed è considerato valido. Il problema principale riguarda la sua implementazione. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza da parte dell'hardware (pochi sistemi si possono permettere un tale sovraccarico per la gestione della memoria). Il problema consiste nel determinare un ordine per i frame definito dal momento dell'ultimo uso.

Si possono realizzare le due seguenti soluzioni:

- **Contatori:** a ogni elemento delle pagine si associa un campo *momento di utilizzo*, e alla CPU sia aggiunge un contatore che si incrementa ad ogni riferimento alla memoria. Ogni volta che si fa un riferimento a una pagina, si copia il contenuto del registro contatore nel campo momento di utilizzo nella voce della page table relativa a quella pagina. In questo modo è sempre possibile conoscere il momento in cui è stato fatto l'ultimo riferimento ad ogni pagina. Si sostituisce la pagina con il valore associato più piccolo. I riferimenti temporali si devono mantenere anche quando, a seguito dello scheduling della CPU, si modificano le tabelle delle pagine. Occorre infine considerare l'overflow del contatore.
- **Stack:** un altro metodo per la realizzazione della sostituzione delle pagine LRU prevede l'utilizzo di uno stack dei numeri delle pagine. Ogni volta che si fa un riferimento a una pagina, la si estrae dallo stack e la si colloca in cima a quest'ultima. In questo modo, in cima allo stack si trova sempre la pagina usata per prima, mentre in fondo si trova la pagina usata meno recentemente. Poiché gli elementi si devono estrarre dal centro dello stack, la migliore realizzazione si ottiene utilizzando una lista doppiamente concatenata, con un puntatore all'elemento iniziale e uno a quello finale. Ogni aggiornamento è un po' costoso, ma per la sostituzione non si deve compiere alcuna ricerca: il puntatore dell'elemento di coda punta alla pagina LRU.

Né la sostituzione ottimale né quella LRU sono soggette all'anomalia di Belady. Entrambe appartengono ad una classe di algoritmi di sostituzione delle pagine, chiamati *algoritmi a stack*, che non presenta l'anomalia di Belady.

*Un algoritmo a stack è un algoritmo per il quale è possibile mostrare che l'insieme delle pagine in memoria per n frame è sempre un sottoinsieme dell'insieme delle pagine che dovrebbero essere in memoria per n+1 frame.*

## Sostituzione delle pagine per approssimazione a LRU (usati nella realtà)

Molti sistemi possono fornire un *bit di riferimento*.

Il bit di riferimento a una pagina è impostato automaticamente dall'hardware del sistema ogni volta che si fa riferimento a quella pagina, che sia una lettura o una scrittura su qualsiasi byte della pagina. I bit di riferimento sono associati a ciascun elemento della tabella delle pagine.

Inizialmente, il sistema operativo azzera tutti i bit. Quando si inizia l'esecuzione di un processo utente, l'hardware imposta a 1 il bit associato a ciascuna pagina cui si fa riferimento. Dopo qualche tempo è possibile stabilire quali pagine sono state usate semplicemente esaminando i bit di riferimento. Non è però possibile conoscere l'ordine d'uso. Questa informazione è alla base di molti algoritmi per la sostituzione delle pagine che approssimano LRU.

## Algoritmo con bit supplementari di riferimento

Ulteriori informazioni sull'ordinamento si possono ottenere registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella in memoria con, per esempio, un byte per ogni pagina.

A intervalli regolari, per esempio di 100 millisecondi, un segnale d'interruzione del timer trasferisce il controllo al sistema operativo. Questo inserisce il bit di riferimento per ciascuna pagina nel bit più significativo, shiftando gli altri bit a destra di 1 bit e scartando il bit meno significativo.

Questi registri a scorrimento di 8 bit contengono la storia dell'utilizzo delle pagine relativo agli ultimi otto periodi di tempo.

Interpretando queste successioni di bit come interi senza segno, la pagina cui è associato il numero minore è la pagina LRU e può essere sostituita. Si noti che l'unicità dei numeri non è garantita. Si possono sostituire tutte le pagine con il valore minore, oppure si può ricorrere ad una selezione FIFO.

Il numero dei bit può essere variato: si sceglie secondo l'hardware disponibile per accelerarne al massimo la modifica. Nel caso limite tale numero si riconduce a zero, lasciando soltanto il bit di riferimento.

In questo caso l'algoritmo è noto come algoritmo di sostituzione delle pagine con seconda chance.

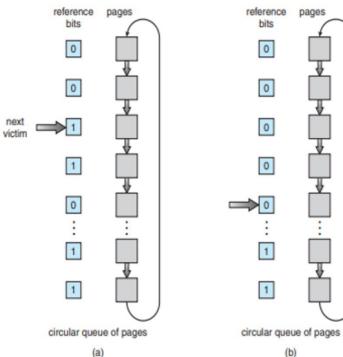
## Algoritmo con seconda chance (algoritmo dell'orologio)

L'algoritmo di base per la sostituzione con seconda chance è un algoritmo di sostituzione di tipo FIFO. Tuttavia, dopo aver selezionato una pagina, si controlla il bit di riferimento: se il suo valore è 0, si sostituisce la pagina; se il bit di riferimento è impostato a 1, si dà una seconda chance alla pagina e si passa alla successiva pagina FIFO.

Quando una pagina riceve la seconda chance, si azzera il suo bit di riferimento e si aggiorna il suo istante d'arrivo al momento attuale. In questo modo, una pagina cui si offre una seconda chance non viene mai sostituita finché tutte le altre pagine non siano state sostituite, oppure non sia stata data loro una seconda chance.

Un metodo per implementare l'algoritmo con seconda chance è basato sull'uso di una *coda circolare*, in cui il puntatore (lancetta) indica qual è la prima pagina da sostituire. Quando serve un frame si fa avanzare il puntatore finché non si trovi in corrispondenza di una pagina con il bit di riferimento 0; a ogni passo si azzera il bit di riferimento appena esaminato. Una volta trovata una pagina vittima, la si sostituisce e si inserisce la nuova pagina nella coda circolare nella posizione corrispondente. Si noti che nel caso peggiore, quando tutti i bit sono impostati a 1, il puntatore percorre un ciclo su tutta la coda, dando a ogni pagina una seconda chance. Prima di selezionare la pagina da sostituire, azzera tutti i bit di riferimento. Se tutti i bit sono a 1, la sostituzione con seconda chance si riduce a una sostituzione FIFO.

Non è una garanzia ma perlomeno è facile da implementare.



## Algoritmo con seconda chance migliorato

L'algoritmo con seconda chance descritto precedentemente si può migliorare considerando i bit di riferimento e di modifica come una coppia ordinata, con cui si possono ottenere le seguenti quattro classi:

- (0,0) né recentemente usato né modificato - migliore pagina da sostituire;
- (0,1) non usato recentemente, ma modificato - pagina non così buona poiché prima di essere sostituita viene dev'essere scritta in memoria secondaria;
- (1,0) usato recentemente ma non modificato - probabilmente la pagina sarà presto usata nuovamente;
- (1,1) usato recentemente e non modificato - probabilmente la pagina sarà presto ancora usata e dovrà essere scritta in memoria secondaria prima di essere sostituita;

Ogni pagina rientra in una di queste quattro classi. Alla richiesta di una sostituzione di pagina, si usa lo stesso schema impiegato nell'algoritmo a orologio, ma anziché controllare se la pagina puntata ha il bit di riferimento impostato a 1, si esamina la classe a cui la pagina appartiene e si sostituisce la prima pagina che si trova nella classe minima non vuota. Si noti che si può dover scandire la coda circolare più volte prima di trovare una pagina da sostituire.

La differenza principale tra questo algoritmo e il più semplice algoritmo dell'orologio è che qui si dà la preferenza alle pagine modificate, al fine di ridurre il numero di I/O richiesti.

## Sostituzione delle pagine basata su conteggio (usata poco)

Si potrebbe usare un contatore del numero dei riferimenti fatti a ciascuna pagina e sviluppare i seguenti schemi:

- *Algoritmo di sostituzione delle pagine meno frequentemente usate*: richiede che si sostituisca la pagina con il conteggio più basso. Si ha però un problema quando una pagina è usata molto intensamente durante la fase iniziale di un processo, ma poi non viene più utilizzata. Una soluzione può essere quella di spostare i valori dei contatori a destra di un bit a intervalli regolari, misurando l'utilizzo con un peso esponenziale decrescente.
- *Algoritmo di sostituzione delle pagine più frequentemente usate*: è basato sul fatto che, probabilmente, la pagina con il contatore più basso è stata appena inserita e non è stata ancora usata.

Queste soluzioni non sono molto comuni, poiché la loro realizzazione è molto onerosa. Inoltre, tali algoritmi non approssimano bene la soluzione ottimale.

## Algoritmi con buffer delle pagine

I sistemi hanno generalmente un gruppo di frame liberi. Quando si verifica un page fault, si sceglie innanzitutto un frame vittima, ma prima che la vittima sia scritta in memoria secondaria, si trasferisce la pagina richiesta in un frame libero del gruppo. Questa procedura permette al processo di ricominciare al più presto, senza attendere che la pagina vittima sia scritta in memoria secondaria. Quando nel seguito si scrive la vittima in memoria secondaria, si aggiunge il suo frame nel gruppo dei frame liberi.

Quest'idea si può estendere conservando una lista delle pagine modificate: ogni qualvolta il dispositivo di paginazione è inattivo, si sceglie una pagina modificata, la si scrive nel disco e si resetta il suo bit di modifica. Questo schema aumenta la probabilità che, al momento della selezione

per la sostituzione, la pagina non abbia subito modifiche e non debba essere scritta in memoria secondaria.

Un'altra modifica consiste nell'usare un gruppo di frame liberi, ma ricordare quale pagina era contenuta in ciascun frame. Poiché quando il contenuto di un frame viene scritto su disco tale contenuto non cambia, la vecchia pagina è ancora utilizzabile prendendola dal gruppo dei frame liberi, se ce n'è bisogno prima che sia riusato quel frame. In questo caso non è necessario alcun I/O. Se si verifica un page fault si controlla prima se la pagina richiesta si trova nel gruppo dei frame liberi; se non c'è si deve individuare un frame libero e trasferirvi la pagina.

Alcune versioni di UNIX adottano questo metodo insieme all'algoritmo con seconda chance. In effetti, si tratta di un'utile integrazione a qualunque algoritmo di sostituzione, al fine di ridurre il prezzo pagato per l'eventuale errata scelta della pagina da sostituire.

Abbiamo un impegno di memoria superiore, ma c'è ottimizzazione.

## Allocazione dei frame

Consideriamo ora il problema dell'allocazione. Occorre stabilire un criterio per l'allocazione della memoria libera ai diversi processi. Come esempio, se abbiamo 93 frame e due processi, quanti frame assegnamo a ciascun processo?

Si consideri un sistema che disponga di 128 frame. Il sistema operativo può occuparne 35, lasciando 93 frame per il processo utente. In condizioni di paginazione su richiesta pura, tutti i 93 sono inizialmente posti nella lista dei frame liberi. Quando comincia l'esecuzione, il processo utente genera una sequenza di page fault. I primi 93 page fault ricevono i frame liberi dalla lista. Una volta esaurita quest'ultima, per stabilire quale tra le 93 pagine presenti in memoria si debba sostituire con la 94esima, si può usare un algoritmo di sostituzione delle agine. Terminato il processo, si reinseriscono i 93 frame nella lista dei frame liberi.

Vi sono molte variazioni di questa semplice strategia. Si può richiedere che il sistema operativo assegna tutto lo spazio richiesto dalle proprie strutture dati attingendo dalla lista dei frame liberi. Quando questo spazio è inutilizzato dal sistema operativo può essere sfruttato per la paginazione utente. Un'altra variante prevede di riservare sempre tre frame liberi, in modo che quando si verifica un page fault sia sempre disponibile un frame libero in cui trasferire la pagina richiesta. Mentre ha luogo il trasferimento, si può scegliere una pagina da rimpiazzare, che viene poi scritta nel disco mentre il processo utente continua l'esecuzione.

La strategia di base è chiara: *al processo utente si assegna qualsiasi frame libero*.

## Numero minimo di frame

Le strategie di allocazione dei frame sono soggette a parecchi vincoli. Non si possono assegnare più frame di quanti siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre è necessario assegnare almeno un numero minimo di frame.

Una delle ragioni per allocare sempre un numero minimo di frame è legata alle prestazioni. Ovviamente, *al decrescere del numero dei frame allocati a ciascun processo aumenta il tasso di page fault, con conseguente rallentamento dell'esecuzione dei processi*.

Inoltre va ricordato che, quando si verifica un page fault prima che sia stata completata l'esecuzione di un'istruzione, quest'ultima dev'essere riavviata. Di conseguenza, *i frame disponibili devono essere in numero sufficiente per contenere tutte le pagine cui ogni singola istruzione può far riferimento*.

*Il numero minimo di frame è definito dall'architettura del calcolatore*. Per esempio, se l'istruzione di move in una data architettura, per alcune modalità di indirizzamento, è costituita da più di una parola, la stessa istruzione può stare a cavallo tra due pagine. Inoltre, ciascuno dei suoi due operandi può essere un riferimento indiretto, per un totale di sei frame.

Il minimo numero di frame dipende dunque anche dalla complessità delle istruzioni.

*Il numero minimo di frame per ciascun processo è definito dall'architettura, mentre il numero massimo è definito dalla quantità di memoria fisica disponibile. In mezzo vi è un ampio spazio di scelta*.

## Algoritmi di allocazione

Il modo più semplice per suddividere m frame tra n processi è quello per cui a ciascuno si dà una parte uguale,  $m/n$  frame (ignorando per ora i frame di cui il sistema operativo ha bisogno). I frame lasciati liberi si potrebbero usare come buffer di frame liberi.

Questo schema è chiamato *allocazione uniforme*.

*Allocazione proporzionale*: la memoria disponibile si assegna a ciascun processo secondo la propria dimensione.

Sia nell'allocazione uniforme che in quella proporzionale, l'allocazione a ogni processo può variare rispetto al livello di multiprogrammazione. Se tale livello aumenta, ciascun processo perde alcuni frame per fornire la memoria necessaria per il nuovo processo. D'altra parte, se il livello di multiprogrammazione diminuisce, i frame allocati al processo rimossi si possono distribuire tra quelli che restano.

Occorre notare che sia con l'allocazione uniforme sia con l'allocazione proporzionale, un processo a priorità elevata è trattato come un processo a bassa priorità anche se, per definizione, si vorrebbe che al processo con elevata priorità fosse allocata più memoria per accelerarne l'esecuzione, a discapito dei processi a bassa priorità. Una soluzione prevede l'uso di uno schema di allocazione proporzionale in cui il rapporto dei frame non dipende dalle dimensioni relative dei processi, ma dalle priorità degli stessi oppure da una combinazione di dimensioni e priorità.

## Allocazione globale e allocazione locale

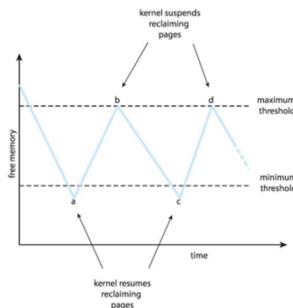
Gli algoritmi di sostituzione delle pagine si possono classificare in due categorie principali:

- *sostituzione globale*: permette che per un processo si scelga un frame per la sostituzione dall'insieme di tutti i frame, anche se quel frame è al momento allocato a un altro processo; un processo può dunque sottrarre un frame a un altro processo. Può accadere che per un certo processo si selezionino solo frame allocati ad altri processi, aumentando così il numero di frame assegnati a quel processo, purché altri non

scelgano per la sostituzione i suoi frame. Risente di un problema: un processo non può controllare il proprio tasso di page fault, infatti l'insieme di pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi.

- **sostituzione locale:** richiede che per ogni processo si scelga un frame solo dal proprio insieme di frame. Il numero di blocchi di memoria assegnati a un processo non cambia. Può penalizzare un processo, non rendendogli disponibili altre pagine di memoria meno usate. Generalmente, la sostituzione globale genera una maggiore produttività del sistema, e perciò è il metodo più usato.

Ci soffermeremo ora su una possibile strategia per implementare una politica globale di sostituzione delle pagine. In questo approccio soddisfiamo tutte le richieste di memoria mediante la lista dei frame liberi, ma piuttosto che aspettare che la lista si svuoti prima di iniziare a selezionare le pagine per la sostituzione, attiviamo la sostituzione delle pagine quando la dimensione della lista scende al di sotto di una certa soglia. Questa strategia cerca di garantire che ci sia sempre sufficiente memoria libera per soddisfare nuove richieste.



Come accennato, lo scopo è di mantenere la quantità di memoria libera al di sopra di una soglia minima: quando si scende al di sotto di questa soglia, viene avviata una routine del kernel che inizia a recuperare pagine da tutti i processi nel sistema (in genere escludendo il kernel). Queste routine del kernel sono note come **reaper** e possono applicare qualsiasi algoritmo di sostituzione delle pagine (in genere utilizza un algoritmo che approssima LRU).

Quando la quantità di memoria libera raggiunge la soglia massima, la routine reaper viene sospesa, per poi essere riavviata quando la memoria libera scende nuovamente al di sotto della soglia libera.

Consideriamo ciò che potrebbe accadere quando la routine reaper non è in grado di mantenere la lista dei frame liberi al di sopra della soglia minima. In queste circostanze, la routine inizia a recuperare le pagine in modo più aggressivo, per esempio sospendendo l'algoritmo con seconda chance e utilizzando una tecnica FIFO pura.

Un altro esempio più estremo si può verificare in Linux: quando la quantità di memoria libera scende a livelli molto bassi, una routine nota come *Out Of Memory killer* seleziona un processo da terminare, liberando così la sua memoria. Ogni processo è dotato del punteggio OOM, dove un punteggio più alto aumenta la probabilità che il processo possa essere terminato dalla routine OOM. I punteggi OOM sono calcolati in base alla percentuale di memoria utilizzata da un processo: maggiore è la percentuale, maggiore è il punteggio OOM.

In generale, non è soltanto l'aggressività della routine reaper a variare, ma possono essere modificati anche i valori delle soglie minima e massima. Questi valori possono essere impostati su valori predefiniti, ma alcuni sistemi consentono a un amministratore di sistema di configurarli in base alla quantità di memoria fisica presente nel sistema.

## Thrashing

Si consideri un qualsiasi processo che non disponga di un numero di frame sufficiente. Se non vi sono abbastanza frame per ospitare le pagine del working set, il processo incorre rapidamente in un page fault.

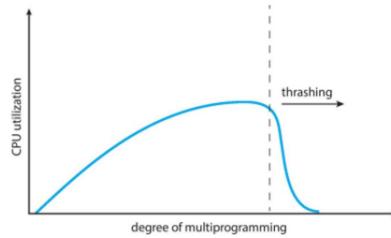
A questo punto si deve sostituire qualche pagina; ma, poiché tutte le sue pagine sono attive, si deve sostituire una pagina che sarà subito necessaria, e di conseguenza si verificano parecchi page fault poiché si sostituiscono pagine che saranno immediatamente riportate in memoria. Questa intensa paginazione è nota come **thrashing**. Un processo di thrashing spende più tempo per la paginazione che per l'esecuzione dei processi -> causa gravi problemi di prestazioni.

## Cause del thrashing

Si consideri il seguente scenario, basato sul comportamento effettivo dei primi sistemi di paginazione.

Il sistema operativo controlla l'utilizzo della CPU. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione delle pagine globale, che sostituisce le pagine senza tenere conto del processo al quale appartengono. Ora si ipotizzi che un processo entri in una nuova fase d'esecuzione e richieda più frame; se ciò si verifica si ha una serie di page fault, cui segue la sottrazione di frame ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch'essi dei page fault, con conseguente sottrazione di frame ad altri processi. Per effettuare il caricamento e lo scaricamento delle pagine per questi processi si deve usare il dispositivo di paginazione. Mentre si mettono i processi in coda per il dispositivo di paginazione, la coda dei processi pronti per l'esecuzione si svuota, quindi l'utilizzo della CPU diminuisce.

Lo scheduler della CPU rileva questa riduzione dell'utilizzo della CPU e aumenta il grado di multiprogrammazione. Si tenta di avviare il nuovo processo sottraendo pagine ai processi in esecuzione, causando ulteriori page fault e allungando la coda per il dispositivo di paginazione. Come risultato, l'utilizzo della CPU scende ulteriormente e lo scheduler della CPU tenta di aumentare ancora il grado di multiprogrammazione. Si è in una situazione di thrashing che fa precipitare la produttività del sistema. Il tasso dei page fault aumenta enormemente, e di conseguenza aumenta il tempo effettivo d'accesso alla memoria. I processi non svolgono alcun lavoro, poiché si sta spendendo tutto il tempo per l'attività di paginazione.



Aumentando il grado di multiprogrammazione aumenta anche l'utilizzo della CPU, anche se più lentamente, fino a raggiungere un massimo. Se a questo punto si aumenta ulteriormente il grado di multiprogrammazione, l'attività di paginazione degenera e fa crollare l'utilizzo della CPU. In questa situazione, per aumentare l'utilizzo della CPU e bloccare il thrashing occorre ridurre il grado di multiprogrammazione (numero di processi attivi nel sistema).

Gli effetti di questa situazione si possono limitare usando un **algoritmo di sostituzione locale (algoritmo di sostituzione per priorità)**.

Con la sostituzione locale, se un processo entra in thrashing, non può sottrarre frame a un altro processo e quindi provocarne a sua volta la degenerazione. Tuttavia il problema non è completamente risolto. I processi in thrashing rimangono nella coda d'attesa del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio di un page fault aumenta a causa dell'allungamento della coda media d'attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo d'accesso in memoria aumenta anche per gli altri processi.

Per evitare di verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame servano a un processo si impiegano diverse tecniche. L'approccio del working set comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo approccio definisce il **modello di località** d'esecuzione del processo.

Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine usate attivamente insieme. Generalmente un programma è formato di parecchie località diverse, che sono sovrapponibili. Generalmente un programma è formato di parecchie località diverse, che sono sovrapponibili. Per esempio, quando s'invoca una procedura, essa definisce una nuova località. In questa località si fanno riferimenti alla memoria per le istruzioni della procedura, per le sue variabili locali e per un sottoinsieme delle variabili globali.

Quando la procedura termina, il processo lascia questa località, poiché le variabili locali e le istruzioni della procedura non sono più usate attivamente. Potrà tornare più tardi questa località.

Quindi, le località sono definite dalla struttura del programma e dalle relative strutture dati. Il modello di località sostiene che tutti i programmi mostrino questa struttura di base di riferimenti alla memoria. Si noti che il modello di località è il principio non dichiarato sottostante all'analisi fin qui svolta sul caching. Se gli accessi ai vari tipi di dati fossero casuali, anziché strutturati in località, il caching sarebbe inutile.

Si supponga di allocare a un processo un numero di frame sufficiente per sistemare le sue località attuali. Finché tutte queste pagine non si trovano in memoria, si verificano le assenze delle pagine relative a tali località; quindi, finché le località non vengono modificate, non hanno luogo altri page fault. Se si assegnano meno frame rispetto alla dimensione della località attuale, la paginazione del processo degnerà, poiché non si possono tenere in memoria tutte le pagine che il processo sta usando attivamente.

## Modello del working set

Basato sull'ipotesi di località.

Questo modello usa un parametro, delta, per definire la finestra del working set. L'idea consiste nell'esaminare i più recenti delta riferimenti alle pagine. L'insieme di pagine nei più recenti delta riferimenti è il working set. Se una pagina è in uso attivo si trova nel working set; se non è più usata esce dal working set delta unità di tempo dopo il suo ultimo riferimento. Quindi, *il working set non è altro che un'approssimazione della località del programma*.

La precisione del working set dipende dalla scelta del valore di delta:

- se delta è troppo piccolo non include l'intera località
- se delta è troppo grande può sovrapporre più località
- se delta è infinito il working set coincide con l'insieme di pagine cui il processo fa riferimento durante la sua esecuzione

La caratteristica più importante del working set è la sua dimensione. Calcolandone la dimensione WSS<sub>i</sub>, per ciascun processo pi<sub>i</sub> del sistema, si può determinare la richiesta totale di frame, cioè D:

$$D = \sum WSS_i$$

Ogni processo usa attivamente le pagine del proprio working set. Quindi, il processo necessita di WSS<sub>i</sub> frame. Se la richiesta totale è maggiore del numero totale di frame liberi ( $D > m$ ), si avrà il thrashing, poiché alcuni processi non dispongono di un numero sufficiente di frame.

Una volta scelto delta, l'uso del modello del working set è abbastanza semplice. Il sistema operativo controlla il working set di ogni processo e gli assegna un numero di frame sufficiente, rispetto alle dimensioni del suo working set. Se i frame ancora liberi sono in numero sufficiente, si può iniziare un altro processo. Se la somma delle dimensioni dei working set aumenta, superando il numero totale dei frame disponibili, il sistema operativo individua un processo da sospendere. Scrive in memoria secondaria le pagine di quel processo e assegna i suoi frame ad altri processi. Il processo sospeso può essere ripreso successivamente.

Questa strategia impedisce il thrashing, mantenendo il grado di multiprogrammazione più alto possibile, quindi ottimizza l'utilizzo della CPU.

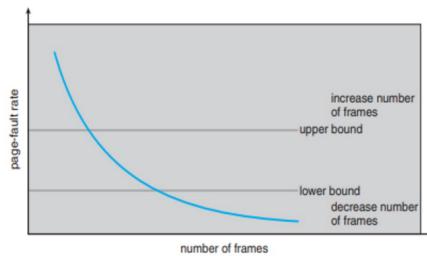
Poiché la finestra del working set è una finestra dinamica, la difficoltà insita in questo modello consiste nel tener traccia degli elementi che compongono il working set stesso. A ogni riferimento alla memoria, a un'estremità appare un riferimento nuovo e il riferimento più vecchio fuoriesce dall'altra estremità. Una pagina si trova nel working set se esiste un riferimento a essa in qualsiasi punto della finestra del working set.

Si può approssimare il modello con un interrupt da timer a intervalli fissi e un bit di riferimento. Le pagine con almeno un bit attivo si considerano appartenenti al working set.

## Frequenza dei page fault

Il modello del working set ha avuto successo e la sua conoscenza può servire per la prepaginazione, ma appare un modo alquanto goffo per controllare il thrashing. La strategia basata sulla frequenza dei page fault è più diretta.

Il problema specifico è la prevenzione del thrashing. La frequenza dei page fault in tale situazione è alta, ed è proprio questa che si deve controllare. Se la frequenza è eccessiva, significa che il processo necessita di più frame. Analogamente, se la frequenza dei page fault è molto bassa, il processo potrebbe disporre di troppi frame. Si può fissare un limite inferiore e un limite superiore per la frequenza desiderata dei page fault.



- Se la frequenza effettiva dei page fault per un processo oltrepassa il limite superiore, occorre allocare a quel processo un altro frame
- Se la frequenza effettiva dei page fault per un processo scende sotto il limite inferiore, si sottrae un frame a quel processo

Quindi, *per prevenire il thrashing, si può misurare e controllare direttamente la frequenza dei page fault.*

Come nel caso della strategia del working set, può essere necessario lo swapping di un processo. Se la frequenza dei page fault aumenta e non ci sono frame disponibili, occorre selezionare un processo e spostarlo nel backing store. I frame liberati si distribuiscono ai processi con elevata frequenza di page fault.

## Regole correttamente adottate

Il thrashing e l'avvicendamento dei processi hanno un pessimo impatto sulle prestazioni. La miglior regola adottata attualmente nella realizzazione dei computer è quella di includere una quantità di memoria fisica sufficiente a evitare il thrashing e l'avvicendamento, quando possibile. Sia quando si parla di smartphone che nel caso di mainframe, fornire una quantità di memoria sufficiente a mantenere tutti gli insiemi di lavoro contemporaneamente in memoria, eccetto in condizioni estreme, offre all'utente la miglior esperienza d'uso possibile.

## Prepaginazione

Rappresenta un *tentativo di prevenire un alto livello di paginazione iniziale*: do un numero iniziale di pagine al processo per evitare che si verifichi il page fault sin dall'inizio. In un sistema che usa il modello del working set, per esempio, a ogni processo si può associare una lista delle pagine contenute nel suo working set.

In alcuni casi la prepaginazione può essere vantaggiosa. La questione riguarda semplicemente il suo costo, che dev'essere inferiore al costo per servire i corrispondenti page faults. Può accadere che molte pagine trasferite in memoria dalla prepaginazione non siano usate.

Si supponga che siano prepaginate  $s$  pagine e sia effettivamente usata una frazione  $\alpha$  di queste  $s$  pagine ( $0 \leq \alpha \leq 1$ ). Occorre sapere se il costo delle  $\alpha s$  eccezioni di page fault risparmiate sia maggiore o minore del costo di prepaginazione di  $(1-\alpha)s$  pagine non necessarie. Se il parametro  $\alpha$  è prossimo allo 0, la prepaginazione non è conveniente; se  $\alpha$  è prossimo a 1, la prepaginazione lo è.

La prepaginazione di un programma eseguibile può essere difficile, perché non sempre è chiaro quali pagine debbano essere portate in memoria. La prepaginazione di un file è spesso maggiormente prevedibile, poiché l'accesso ai file è solamente sequenziale.

## Dimensione delle pagine

In linea di massima non esiste una dimensione migliore; più fattori sono a sostegno delle diverse dimensioni. Un primo fattore da considerare è la tabella delle pagine: diminuendo la dimensione aumenta il numero delle pagine e di conseguenza la dimensione della tabella. D'altra parte la memoria è utilizzata meglio se le pagine sono piccole: la frammentazione interna è ridotta. Un altro problema è il tempo per leggere o scrivere una pagina: il tempo di trasferimento è proporzionale alla quantità trasferita. Il tempo di I/O è la somma dei tempi di posizionamento, latenza e trasferimento. Il tempo di trasferimento è piccolo se confrontato rispetto alla latenza e tempo di posizionamento. Per ridurre il tempo di I/O occorre avere pagine di dimensioni maggiori. Tuttavia con pagine di piccole dimensioni si riduce l'I/O totale. Infine per ridurre il numero di page fault occorre sono necessarie pagine di grandi dimensioni.

## TLB reach

La portata del TLB è un parametro che esprime la quantità di memoria accessibile dal TLB ed è dato dal numero di elementi moltiplicato per la dimensione delle pagine. Idealmente il TLB dovrebbe contenere il working set del processo. Raddoppiando il numero di elementi del TLB si raddoppia la portata; un altro modo per aumentarla è aumentare la dimensione delle pagine.

Struttura del programma: se accedo a pettine ho molte più page fault rispetto a un programma che accede alla memoria in maniera sequenziale -> è la stessa idea dietro alla scrittura dei programmi cache friendly.

## Vincolo di I/O e vincolo delle pagine

A volte, quando si usa la paginazione su richiesta, occorre vincolare alla memoria alcune pagine. Una tipica situazione è quando l'I/O si esegue verso e dalla memoria d'utente. Esempio: se si vuole eseguire un'operazione di scrittura su di un'unità a nastro, al controllore si indica il numero di byte da trasferire e l'indirizzo di memoria per la scrittura. Quando il processo emette la richiesta di I/O viene messo in coda. Nel frattempo può succedere che la pagina contenente l'indirizzo di memoria per l'operazione di I/O venga scaricata dalla memoria. Quando la richiesta raggiunge la prima posizione della coda, l'operazione di I/O avviene all'indirizzo specificato, ma il blocco di memoria è ora impiegato per una pagina di un altro processo. Il problema può essere risolto in due modi: eseguire l'I/O tra il dispositivo e la memoria di sistema; questa soluzione introduce un sovraccarico inaccettabile. La seconda soluzione prevede che le pagine siano vincolate alla memoria. Ad ogni blocco si associa un bit di vincolo: se tale bit è attivo, la pagina contenuta in tale blocco non può essere selezionata per la sostituzione.

## Come funziona la memoria virtuale?

La memoria virtuale è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria, sfruttando il disco (come RAM). La memoria virtuale si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica. È possibile organizzare la memoria fisica in frame di pagine; in questo caso i frame delle pagine fisiche assegnati ai processi possono non essere contigui. Spetta all'unità di gestione della memoria (MMU) associare in memoria le pagine logiche alle pagine fisiche. Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre, per due o più processi, il vantaggio di condividere i file e la memoria, mediante la condivisione delle pagine.

## Cos'è il page fault?

Se il processo tenta l'accesso a una pagina che non è stata caricata precedentemente in memoria, l'accesso a una pagina contrassegnata come non valida causa un'eccezione di pagina mancante.

## Cos'è la paginazione su richiesta e perché viene utilizzata?

E' una strategia comunemente utilizzata nei sistemi con memoria virtuale e consiste nel caricare in memoria le pagine nel momento in cui servono realmente. Secondo questo schema, le pagine sono caricate in memoria solo quando richieste durante l'esecuzione del programma. Un sistema di paginazione su richiesta è analogo ad un sistema paginato con avvicendamento dei processi in memoria.

Anziché caricare in memoria l'intero processo, si può seguire un criterio di avvicendamento "pigro" (lazy swapping): non si carica mai in memoria una pagina che non sia necessaria.

Quando un processo sta per essere caricato, il paginatore ipotizza quali pagine saranno usate. E' necessario che l'architettura risponda di un qualche meccanismo che consenta di distinguere le pagine presenti in memoria da quelle nei dischi. Durante l'esecuzione, il processo accede alle pagine residenti in memoria, e l'esecuzione procede come di consueto. Se il processo tenta l'accesso a una pagina che non era stata caricata in memoria, l'accesso a una pagina contrassegnata come non valida causa una page fault.

## Perché si usa la sostituzione delle pagine e quali sono gli algoritmi che la definiscono?

Nel momento in cui durante l'esecuzione di un processo utente si verifica un'assenza di pagina, il sistema operativo deve caricare la pagina mancante, ma se la lista dei frame liberi è vuota, si deve scegliere un frame da sacrificare e sostituire. Per effettuare una sostituzione ottimale sono stati definiti diversi algoritmi (vedi sopra).

## Cos'è l'anomalia di Belady?

L'anomalia di Belady si ha quando alcuni algoritmi di sostituzione delle pagine aumentano la frequenza di page fault, perché vanno a sostituire pagine necessarie che devono essere ricaricate in memoria poco dopo.

Altra risposta:

L'anomalia di Belady è un fenomeno osservato in alcuni algoritmi di sostituzione di pagine in sistemi operativi che gestiscono la memoria virtuale. L'anomalia di Belady afferma che, in alcuni casi, aumentare la dimensione della cache può portare a un aumento del numero di page fault (quando una pagina richiesta non è presente in memoria e deve essere caricata dal disco) rispetto a una cache più piccola. Questo può sembrare controtintuitivo, poiché ci si aspetterebbe che una cache più grande sia in grado di contenere più pagine e quindi ridurre il numero di fallimenti di pagina. L'anomalia di Belady si verifica quando un algoritmo di sostituzione di pagine non è in grado di prevedere correttamente quali pagine saranno richieste in futuro e, di conseguenza, sceglie di sostituire le pagine sbagliate, portando a una maggiore frequenza di page fault. Inoltre, la presenza di dipendenze cicliche tra le pagine richieste può peggiorare ulteriormente la situazione. Per evitare l'anomalia di Belady, è necessario utilizzare algoritmi di sostituzione di pagine più sofisticati che prendono in considerazione informazioni sulle preferenze future delle pagine, come ad esempio l'algoritmo di sostituzione di pagine Optimal (OPT) o l'algoritmo di sostituzione di riga.

## Cos'è il thrashing?

Degenerazione della paginazione su richiesta, con annesso aumento dei page fault e degradazione delle prestazioni del sistema operativo.

Altra risposta:

Il thrashing è una situazione in cui il sistema operativo non è in grado di fornire una quantità sufficiente di memoria ad un'applicazione che ne richiede. Ciò può causare una serie di problemi, tra cui una riduzione della velocità del sistema, una maggiore probabilità di crash e una diminuzione delle prestazioni complessive. Il thrashing si verifica quando la memoria disponibile è insufficiente per soddisfare le richieste delle applicazioni in esecuzione e il sistema operativo è costantemente impegnato a spostare i dati tra la memoria RAM e la memoria virtuale sul disco rigido. Questo processo, noto come swapping, può rallentare drasticamente le prestazioni del sistema e causare un eccessivo utilizzo del disco.

rigido. L'idea è monitorare il numero di pagine richiesto dal processo su intervallo di tempo. Utilizzo un delta che tenta di carpire la dimensione del working set in modo da dare il giusto numero di pagine. Casi del delta – Sistema Unix.

## Cos'è il working set?

Il working set è un concetto utilizzato in informatica per descrivere la quantità di memoria necessaria per eseguire un'applicazione o un processo in modo efficiente. Il working set rappresenta la quantità di memoria necessaria per mantenere in RAM i dati e le istruzioni più frequentemente utilizzati da un processo in un determinato periodo di tempo. Il working set viene utilizzato per ottimizzare le prestazioni del sistema operativo e dei processi in esecuzione. Ad esempio, il sistema operativo può tenere traccia del working set di ciascun processo e allocare la memoria in modo da mantenere la quantità di memoria richiesta da ciascun processo in RAM. In questo modo, il sistema operativo può ridurre la quantità di swapping tra la memoria RAM e la memoria virtuale sul disco rigido, migliorando le prestazioni complessive del sistema. In generale, un working set più grande significa che un processo richiede più memoria per eseguirsi in modo efficiente, mentre un working set più piccolo significa che un processo richiede meno memoria. Tuttavia, un working set troppo grande può anche causare problemi di memoria, come il thrashing. Pertanto, è importante che il sistema operativo tenga traccia del working set di ciascun processo e allochi la memoria in modo equilibrato e ottimizzato.

# Capitolo 13 - Sistemi Operativi

## Interfaccia del file system

Il file system è l'aspetto più visibile di un sistema operativo. Esso fornisce il meccanismo per la memorizzazione in linea di dati e programmi appartenenti al sistema operativo e a tutti gli utenti del sistema elaborativo.

Il file system consiste di due parti distinte:

- un *insieme di file*, ciascuno dei quali contiene i dati
- una *struttura della directory*, che organizza tutti i file nel sistema e fornisce le informazioni relative

Alcuni file system hanno un terzo componente: le partizioni, utilizzate per separare fisicamente o logicamente grandi gruppi di directory.

La maggior parte dei file system risiede su dispositivi di memoria secondaria.

### Concetto di file

Per rendere agevole l'utilizzo del calcolatore il sistema operativo astrae le caratteristiche fisiche dei propri dispositivi di memoria definendo un'unità di memoria logica, il file.

Il file è un *insieme di informazioni, correlate e registrate nella memoria secondaria, cui è stato assegnato un nome*. Dal punto di vista dell'utente un file è la più piccola porzione di memoria secondaria logica, cioè i dati si possono scrivere nella memoria secondaria solo all'interno di un file.

In un file può essere contenuto: un programma sorgente (sequenza di procedure e funzioni), programmi oggetto (sequenza di byte comprensibili al modulo di collegamento), immagini, registrazioni, testi, ecc.

Perché nella dimensione corrente di un file, sul disco, viene occupata più memoria? Es. 11KB, 11.7 KB su disco.

Perché lo spazio sul disco viene allocato a blocchi di 512 byte (a volte più grossi). Significa che, attraverso una minima segmentazione interna, viene assegnato più spazio di quello strettamente necessario.

### Attributi dei file

Un file ha attributi che possono variare secondo il sistema operativo, ma che tipicamente comprendono i seguenti:

- *Nome*: il nome simbolico del file è l'unica informazione in forma umanamente leggibile.
- *Identificatore*: si tratta di un'etichetta unica, di solito un numero, che identifica il file all'interno del file system; è il nome impiegato dal sistema per il file.
- *Tipo*: necessario ai sistemi che gestiscono tipi di file diversi.
- *Locazione*: si tratta di un puntatore al dispositivo e alla locazione del file in tale dispositivo.
- *Dimensione*: si tratta della dimensione corrente del file (in byte, parole o blocchi) ed eventualmente della massima dimensione consentita.
- *Protezione*: le informazioni di controllo degli accessi controllano chi può leggere, scrivere o eseguire il file.
- *Ora, data e identificazione dell'utente*: queste informazioni possono essere relative alla creazione, l'ultima modifica e l'ultimo uso. Questi dati possono essere utili ai fini della protezione, della sicurezza e del monitoraggio del suo utilizzo.

Poiché le directory, come i file, devono essere non volatili, si devono registrare sul dispositivo di memorizzazione di massa e caricare in memoria centrale un po' per volta, secondo la necessità.

### Operazioni sui file

Un file è un tipo di dato astratto (ADT).

Per definire adeguatamente un file è necessario considerare le operazioni che si possono eseguire su di esso. Il sistema operativo può offrire chiamate di sistema per creare, scrivere, leggere, spostare, cancellare e troncare un file.

- *Creazione*: è necessario compiere due passaggi. In primo luogo si deve trovare lo spazio per il file nel file system; inoltre, per un file si deve creare un nuovo elemento nella directory.
- *Scrittura*: per scrivere in un file viene effettuata una syscall che specifica il nome del file e le informazioni che si vogliono scrivere. Dato il nome del file, il sistema ricerca la directory per individuare la posizione del file. Il file system deve mantenere un puntatore di scrittura alla locazione nel file in cui deve avvenire l'operazione di scrittura successiva. Il puntatore si deve aggiornare ogni qualvolta si esegue una scrittura.
- *Lettura*: per leggere da un file è necessaria una chiamata di sistema che specifichi il nome del file e la posizione in memoria dove collocare il blocco del file da leggere. Anche in questo caso si cerca l'elemento corrispondente nella directory e il sistema deve mantenere un puntatore di lettura alla locazione nel file in cui deve avvenire la successiva operazione di lettura. Una volta completata la lettura, si aggiorna il puntatore. Di solito un processo o legge o scrive in un file, e la posizione corrente è mantenuta come un puntatore alla posizione corrente nel file specifico del processo. Sia le operazioni di lettura che quelle di scrittura adoperano lo stesso puntatore, risparmiando spazio e riducendo la complessità del sistema.
- *Riposizionamento in un file*: si ricerca l'elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente nel file. Il riposizionamento non richiede alcuna operazione di I/O. Questa operazione è anche nota come posizionamento o ricerca nel file.
- *Cancellazione*: per cancellare un file si cerca l'elemento della directory associato al file disegnato, si rilascia lo spazio associato al file (in modo che possa essere adoperato per altri) e si elimina l'elemento della directory.

- *Troncamento*: si potrebbe voler cancellare il contenuto di un file, ma mantenere i suoi attributi. Invece di forzare gli utenti a cancellare il file e quindi ricrearlo, questa funzione consente di mantenere immutati gli attributi (a esclusione della lunghezza del file) pur azzerando la lunghezza del file e rilasciando lo spazio occupato.

La maggior parte delle operazioni citate richiede una ricerca dell'elemento associato al file nella directory. Per evitare questa continua ricerca si può utilizzare la syscall open() la prima volta che si utilizza il file. Il sistema operativo mantiene una piccola tabella contenente informazioni riguardo i file aperti: tabella dei file aperti.

Quando si richiede un'operazione su un file si evita qualsiasi ricerca, individuando tale file tramite un indice nella tabella.

Quando il file viene chiuso, il sistema operativo rimuove l'elemento associato dalla tabella. La syscall open() riporta di solito un puntatore all'elemento della tabella; questo puntatore si adopera al posto dell'effettivo nome del file in tutte le operazioni di I/O.

In un ambiente multietente (come Unix), la realizzazione delle operazioni open() e close() è più complicata perché più utenti possono aprire un file contemporaneamente. Di solito il sistema operativo introduce due livelli di tabella interne: una tavola di sistema e una tabella per ciascun processo. La tabella di un processo contiene i riferimenti a tutti i file aperti da quel processo. Ciascun elemento della tabella punta a sua volta a una tabella di sistema dei file aperti che contiene le informazioni indipendenti dal processo come la posizione dei file nei dischi, le date degli accessi, la dimensione. Quando un file è stato aperto da un processo, una open() eseguita da un altro comporta l'aggiunta di un nuovo elemento nella tabella dei file aperti associata a quel processo, che punta al corrispondente elemento della tabella del sistema.

Tipicamente, la tabella dei file aperti ha un contatore delle aperture associato a ciascun file che indica il numero di processi che ha aperto quel file. Ogni close() decremente il contatore; quando esso raggiunge il valore 0 il file non è più in uso e viene eliminato dalla tabella.

Riassumendo, a ciascun file aperto sono associate le diverse seguenti informazioni:

- *Puntatore al file*: nei sistemi che non prevedono un offset come parametro delle chiamate di sistema read() e write(), il sistema deve tenere traccia dell'ultima posizione di lettura e scrittura sotto forma di un puntatore alla posizione corrente nel file. Questo puntatore è unico per ogni processo che opera sul file e quindi dev'essere tenuto separato dagli altri attributi del file residenti nel disco.
- *Contatore dei file aperti*: man mano che si chiudono i file, per evitare di esaurire lo spazio associato alla propria tabella dei file aperti, il sistema operativo deve riutilizzare gli elementi. Poiché più processi possono aprire uno stesso file, prima di rimuovere l'elemento corrispondente, il sistema deve attendere l'ultima chiusura del file. Questo contatore tiene traccia del numero di open() e close() e raggiunge il valore 0 dopo l'ultima chiusura, momento in cui il sistema può rimuovere l'elemento dalla tabella.
- *Posizione nel disco del file*: la maggior parte delle operazioni richiede al sistema di modificare i dati contenuti nel file. L'informazione necessaria per localizzare il file (ovunque si trovi, sia esso su una memoria di massa, su un file server in rete, o su un'unità RAM) è mantenuta in memoria, per evitare di doverla prelevare dal disco a ogni operazione.
- *Diritti d'accesso*: ciascun processo apre un file in una delle modalità d'accesso. Questa informazione è contenuta nella tabella del processo in modo che il sistema operativo possa permettere o negare le successive richieste di I/O.

Alcuni sistemi operativi offrono la possibilità di applicare lock a un file aperto (o a parti di esso). Quando un processo intende proteggere un file dall'accesso concorrente di altri processi, ci serve il lock.

L'utilità dei lock dei file emerge nel caso di file condivisi da diversi processi: un file di log, per esempio, può subire modifiche da parte di molti processi del sistema.

I lock dei file sono basati su una funzionalità simile ai lock di lettura-scrittura.

Un *lock condiviso* consente a più processi concorrenti di appropriarsene.

Un *lock esclusivo* permette ad un solo processo per volta di acquisire il lock.

Non in tutti i sistemi operativi vengono forniti entrambi i tipi di lock; alcuni forniscono solo i lock esclusivi dei file.

Inoltre, il sistema può fornire meccanismi di *lock dei file obbligatori oppure consultivi*. Se un lock è obbligatorio, il sistema operativo impedirà a qualunque altro processo di accedere al file interessato una volta che il suo lock sia stato acquisito.

Se il lock è obbligatorio, il sistema operativo assicura l'integrità dei dati soggetti a lock; se il lock è consultivo, è compito dei programmati garantire la corretta acquisizione e cessione dei lock.

In linea generale, i sistemi operativi Windows adottano i lock obbligatori, mentre Unix adotta i lock consultivi.

## Struttura dei file

Un sistema operativo che riconosce il tipo di un file ha la possibilità di trattare il file in modo ragionevole. Una tecnica comune per realizzare la gestione dei tipi di file consiste nell'includere il tipo nel nome del file. Quindi il nome viene suddiviso in due parti: il nome e l'estensione. Il SO usa l'estensione per stabilire il tipo di file e le operazioni che si possono eseguire su di esso. I tipi di file si possono anche adoperare per indicare la struttura interna dei file. Inoltre alcuni file devono rispettare una determinata struttura comprensibile al SO. Per un SO la localizzazione di uno scostamento all'interno di un file può essere complicata.

Nella stragrande maggioranza dei casi un file è una sequenza di byte che viene interpretata dal processo.

Struttura complicata: gestibile mettendo dei caratteri di controllo.

## Struttura interna dei file

I dischi hanno una dimensione dei blocchi ben definita; tutti gli I/O su disco si eseguono in unità di un blocco. E' improbabile che la dimensione del record fisico corrisponda esattamente alla lunghezza del record logico desiderato che può essere variabile. Una soluzione diffusa consiste nell'impaccamento di un certo numero di record logici in un blocco fisico. Il file si può considerare come una sequenza di blocchi; tutte le funzioni di I/O operano in termini di blocchi. Tutti i file system soffrono di frammentazione interna che aumenta con l'aumentare della dimensione dei blocchi.

## Metodi di accesso

I file memorizzano informazioni; al momento dell'uso è necessario accedere a queste informazioni e trasferirle in memoria. Esistono molti metodi per accedere alle informazioni dei file; alcuni sistemi consentono un solo metodo d'accesso ai file, mentre altri offrono diversi metodi di accesso: in questo caso, la scelta del metodo giusto per una particolare applicazione è un importante problema di progettazione.

## Accesso sequenziale

Il metodo di accesso più semplice: *le informazioni del file si elaborano ordinatamente*, un record dopo l'altro; il più comune, usato dagli editor e dai compilatori.

Un'operazione di lettura legge la prima porzione e fa avanzare automaticamente il puntatore del file che tiene traccia della locazione di I/O.

Analogamente, un'operazione di scrittura fa un'aggiunta in coda al file e avanza fino alla fine.

## Accesso diretto

Abbiamo detto che un file è formato da elementi logici (record) di dimensione fissa.

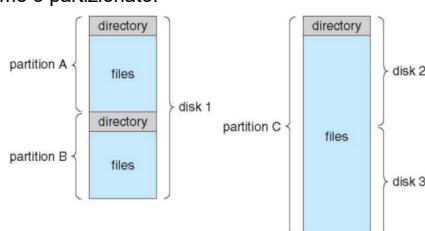
Questo metodo consente di leggere o scrivere i record di un file in modo arbitrario: non esistono limiti all'ordine di lettura o scrittura.

I file ad accesso diretto sono utili quando è necessario accedere immediatamente a grandi quantità di informazioni. Spesso le basi di dati sono di questo tipo: data una query, occorre stabilire quale blocco contiene la risposta alla richiesta e quindi leggere direttamente quel blocco ottenendo così le informazioni richieste. Per questo metodo occorre modificare le operazioni sui file inserendo il numero di blocco in forma di parametro.

Questo numero è un numero di blocco relativo il uso permette al sistema operativo di decidere dove posizionare il file e aiuta a impedire che l'utente acceda a porzioni del file system che possono non far parte del suo file.

## Struttura di un disco

Un disco può essere visto come un unico volume o partizionato.



Associato a un disco c'è una tabella delle partizioni che individua le posizioni di inizio e di fine (espressa in cilindri) dei file separati.

Il vantaggio delle partizioni è assicurare una migliore protezione e un miglior backup.

Calibro meglio lo spazio e impedisce l'accesso a certe directory dove ci sono i file di sistema.

Backup dati degli utenti: evito di prendere dati di sistema.

Il backup dell'intero sistema è meno utile e dev'essere fatto di meno rispetto al backup dei dati utente.

La tabella delle partizioni è utile anche per avere più sistemi operativi sulla stessa macchina.

Ci sono dei programmi che gestiscono le partizioni da gestire con cura.

Esistono dei programmi che maneggiano la tabella delle partizioni, che fa sì che un disco rigido venga visto come più dischi fisici, e quindi con più unità separate.

Utilizzata anche dai costruttori di computer per fare partizioni in cui c'è un minimo di software più o meno nascosto, che può essere usato per il recovery.

Sui sistemi seri, viene fatto per protezione, poiché si può calibrare lo spazio, impedire l'accesso a certe directory o volumi fisici in cui ci sono file di sistema.

Le directory possono essere maneggiate solo dal sistema operativo!

## Struttura delle directory

La directory si può considerare come *una tabella di simboli che traduce i nomi dei file negli elementi in essa contenuti*.

La stessa directory si può organizzare in modi diversi; l'organizzazione deve rendere possibile l'inserimento di nuovi elementi, la cancellazione di elementi esistenti, la ricerca di un elemento, e l'elenco di tutti gli elementi della directory.

Nel considerare una particolare struttura della directory si deve tenere presente l'insieme delle seguenti operazioni che si possono eseguire in una directory:

- *Ricerca di un file*: dev'esserci la possibilità di scorrere una directory per individuare l'elemento associato a un particolare file. Poiché i file possono avere nomi simbolici, e poiché nomi simili possono indicare relazioni tra file, deve esistere la possibilità di trovare tutti i file il cui nome soddisfi una particolare espressione.
- *Creazione di un file*: deve essere possibile creare nuovi file e aggiungerli alla directory.
- *Cancellazione di un file*: quando non serve più, si deve poter rimuovere un file dalla directory.
- *Elencazione di una directory*: deve esistere la possibilità di elencare tutti i file di una directory, ed il contenuto degli elementi della directory associati a ciascun file nell'elenco.
- *Ridenominazione di un file*: poiché il nome di un file rappresenta per i suoi utenti il contenuto del file, questo nome deve poter essere modificato quando il contenuto o l'uso del file subiscono cambiamenti. La ridefinizione di un file potrebbe anche permettere la variazione della posizione del file nella directory.

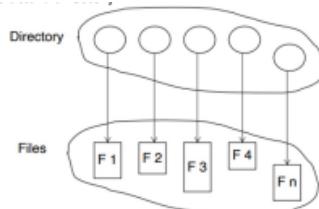
- **Attraversamento del file system:** si potrebbe voler accedere a ogni directory e a ciascun file contenuto in una directory. Per motivi di affidabilità, è opportuno salvare il contenuto e la struttura dell'intero file system a intervalli regolari. Questo salvataggio consiste nella copiatura di tutti i file in un nastro magnetico; tale tecnica consente di avere una copia di riserva (backup) che sarebbe utile nel caso in cui si dovesse verificare un guasto nel sistema. Inoltre, se un file non è più in uso, si può copiarlo su nastro e liberare lo spazio da esso occupato nel disco, rendendolo riutilizzabile per altri file.

## Directory a un livello

Struttura logica della directory più semplice.

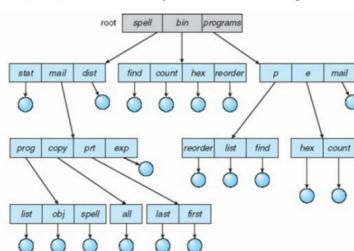
Tutti i file sono contenuti nella stessa directory. Questa struttura presenta dei limiti all'aumentare dei file e se il sistema è usato da più utenti. In quest'ultimo caso, poiché tutti i file si trovano nella stessa directory, gli utenti non possono attribuire lo stesso nome ai loro file di dati. Anche con un solo utente, diventa difficile ricordare i nomi dei file, con l'aumentare del loro numero.

La soluzione più ovvia prevede una directory separata per ogni utente.



## Directory a due livelli

Nella struttura a due livelli, ogni utente dispone della propria directory d'utente. In ogni directory sono elencati solo i file del proprietario.



Quando un utente fa riferimento ad un file particolare, il sistema operativo esegue la ricerca solo nella directory di quell'utente. In questo modo utenti diversi possono avere file con lo stesso nome, purché tutti i nomi di file all'interno di ciascuna directory siano unici. Nella struttura a due livelli esiste una directory principale in cui ogni elemento punta alla relativa directory d'utente.

Sebbene si risolva il problema della collisione dei nomi, la struttura di directory a due livelli presenta ancora dei problemi.

Questa struttura isola un utente dagli altri utenti; questo è uno svantaggio quando gli utenti vogliono cooperare ed accedere a file di altri utenti; questo è uno svantaggio quando gli utenti vogliono cooperare ed accedere a file di altri utenti. Alcuni sistemi non permettono l'accesso a file di utenti locali da parte di altri utenti. Se l'accesso è autorizzato, occorre indicare sia il nome dell'utente che il nome del file; essi compongono il *path name*.

La struttura a due livelli può essere vista come una struttura ad albero di altezza 2.

## Directory con struttura ad albero (fortemente utilizzata)

In una struttura ad albero di altezza n, è possibile creare proprie sottodirectory nelle quali si possono organizzare i file. L'albero ha una root directory e ogni file di sistema ha un unico percorso. Il path name descrive il percorso che parte dalla radice, passa attraverso le sottodirectory e arriva ad un file specifico. Una directory o una sottodirectory contiene un insieme di file o sottodirectory. La distinzione tra file e directory è data dal bit, rispettivamente 0 ed 1, di ogni elemento della directory.

Per creare e cancellare directory si fa uso di system calls. Lo stesso vale per cambiare directory corrente (essa dovrebbe contenere la maggior parte dei file di interesse corrente per il processo).

I path name possono essere:

- assoluti: comincia dalla radice e segue un percorso che lo porta fino al file specificato
- relativi: definisce un percorso a partire dalla directory corrente

Un discorso importante è quello che riguarda la cancellazione di una directory. Se la directory è vuota è sufficiente cancellare l'elemento che la designa nella directory che la contiene.

Se non è vuota è possibile procedere in due modi:

1. non cancellare una directory a meno che non sia vuota; per cancellarla occorre cancellare tutti i file e le sottodirectory in essa contenuti
2. come per il comando rm di UNIX, cancellare automaticamente tutti i file e sottodirectory contenute in essa

La struttura ad albero non ammette la condivisione di file o directory.

## Directory con struttura a grafo aciclico

Una struttura a grafo aciclico permette alle directory di avere sottodirectory e file condivisi: lo stesso file o la stessa sottodirectory possono essere in due directory diverse.

Il fatto che il file o la directory siano condivise non significa che ci siano due copie del file. Se il file è condiviso esiste un solo file effettivo, perciò tutte le modifiche sono immediatamente visibili.

La condivisione può essere realizzata in diversi modi:

- un metodo, diffuso soprattutto tra i sistemi UNIX, prevede la creazione di un nuovo elemento di directory chiamato collegamento (link). Esso è un puntatore a un file o a un'altra directory. Un collegamento si può realizzare con un nome di percorso assoluto o relativo
- un altro metodo comune prevede la duplicazione di tutte le informazioni relative ai file in entrambe le directory di condivisione. In questo caso la copia e l'originale sono resi indistinguibili: sorge il problema di mantenere la coerenza se il file viene modificato.

Una struttura a grafo aciclico è più complessa rispetto ad una struttura ad albero. Infatti, un file può avere più nomi di percorso assoluti. Quando si percorre tutto il file system, il problema diviene più grave poiché non si devono attaversare più di una volta le strutture condivise.

Un altro problema è la cancellazione, poiché è necessario stabilire in quali casi è possibile riassegnare lo spazio assegnato ad un file condiviso. In un sistema in cui la condivisione si realizza tramite collegamenti simbolici la gestione di questa situazione è relativamente semplice. La cancellazione del collegamento non influisce sul file originale; se si cancella il file, si libera lo spazio corrispondente lasciando in sospeso il collegamento. A questo punto si cercano tutti i collegamenti e si rimuovono. Se questi non esistono, la ricerca può essere abbastanza onerosa. In alternativa, si possono lasciare i collegamenti finché non si tenta di usarli.

Un altro criterio prevede la conservazione del file fino a che non siano stati cancellati tutti i riferimenti ad esso. In questo caso è sufficiente un contatore del numero di riferimenti; nel momento in cui il suo valore è uguale a 0 si può cancellare il file.

Il sistema operativo UNIX usa questo metodo per i collegamenti non simbolici o collegamenti hard link.

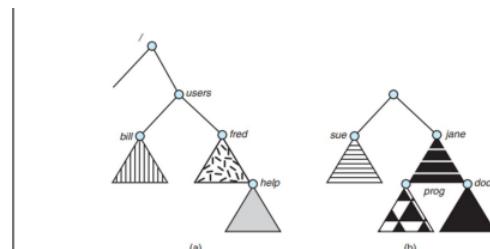
## Directory con struttura a grafo generale

Il vantaggio principale della struttura a grafo aciclico sta nella semplicità degli algoritmi necessari ad attraversarlo e per determinare quando non ci sono più riferimenti a un file.

Se si permette che nella directory esistano dei cicli, è preferibile evitare la duplice ricerca di un elemento. Una soluzione è quella di limitare il numero di directory cui accedere durante la ricerca. Una soluzione è quella di limitare arbitrariamente il numero di directory cui accedere durante una ricerca.

Un problema analogo si presenta al momento di stabilire quando sia possibile cancellare un file. Come con le strutture delle directory a grafo aciclico, la presenza di uno 0 nel contatore dei riferimenti significa che non esistono più riferimenti al file o alla directory, e quindi il file può essere cancellato. Tuttavia, se esistono cicli, è possibile che il contatore dei riferimenti possa essere non nullo, anche se non è più possibile far riferimento a una directory o a un file. Questa anomalia è dovuta alla possibilità di autoriferimento nella struttura delle directory. In questo caso è generalmente necessario utilizzare un metodo di ripulitura (garbage collector) per stabilire quando sia stato cancellato l'ultimo riferimento e quando sia possibile riallocare lo spazio dei dischi.

## File System Mounting (Montaggio)



Nel mondo Windows, nel momento in cui inserisco un'unità, mi ritrovo per ognuna di essa un volume logico.

Nel mondo Unix, invece, tutte le directory vengono incastrate in un unico albero, mediante delle operazioni di montaggio.

Quindi, a differenza del mondo Windows, dove ci sono le unità logiche (A:, B:, ecc), che sono unità separate, in UNIX tutto viene incastrato in un unico albero, secondo i nostri comandi.

In particolare, c'è un albero di testa in cui c'è la directory root, dove ci sono i file di sistema, dopodiché ci sono delle directory appositamente predisposte (tipicamente vuote), che sono detti PUNTO DI MONTAGGIO (sottodirectory di mnt in Unix, /volumes in macOS).

Storicamente, queste unità andavano montate e smontate a mano usando il comando mount per inserire e inastrarlo nell'albero, umount per rimuovere l'unità e scaricando sull'unità tutto quello che stava ancora nella cache, così da non perdere dati.

Tutti i sistemi attuali hanno dei daemon (programmi di sistema) che automaticamente, non appena rilevano la presenza di una nuova unità, la montano in un punto predisposto.

Nel montaggio, prendo e incastro tutto in un unico albero -> con questo sistema ho tutto visibile e le cose funzionano bene.

Se la directory che è punto di montaggio fosse piena, i contenuti vengono momentaneamente nascosti perché viene sostituita dalla radice di quella che sto montando.

## File sharing

In UNIX, le modifiche ai file condivisi sono visibili immediatamente da tutti gli utenti.

Nei file di rete non funziona bene poiché ognuno si fa una copia locale dei file.

## File System remoti

Le reti permettono la condivisione di risorse; un esempio di risorsa da condividere sono i dati nella forma di file. Un primo metodo consiste nel trasferimento dei file attraverso programmi come l'ftp. Un secondo metodo è quello del file system distribuito (DFS) che permette la visibilità nel calcolatore locale delle directory remote. Il terzo metodo è quello del World Wide Web che da un certo punto di vista è il contrario del primo metodo, si usa un programma di consultazione per accedere ai file remoti.

## Semantica della coerenza

E' un'importante criterio per la valutazione di qualsiasi file system che consenta la condivisione.

Si tratta di una caratterizzazione del sistema che specifica la semantica delle operazioni in cui più utenti accedono contemporaneamente a un file condiviso. Questa semantica deve specificare quando le modifiche ai dati apportate da un utente possono essere osservate da altri utenti. La semantica è realizzata come codice facente parte del file system. Il file system dello Unix ha la seguente semantica:

- Le scritture in un file aperto da parte di un utente sono immediatamente visibili ad altri utenti che hanno aperto contemporaneamente lo stesso file.

Esiste un metodo di condivisione in cui gli utenti condividono il puntatore alla locazione corrente del file.

## Protezione

Le informazioni contenute in un sistema elaborativo devono essere protette dai danni fisici (questione dell'affidabilità) e da accessi impropri (questione della protezione).

Generalmente, l'affidabilità è assicurata da più copie dei file. Molti calcolatori hanno programmi di sistema che copiano i file dai dischi ai nastri a intervalli regolari, per esempio una volta al giorno, alla settimana o al mese; quest'operazione di copiatura può essere automatica o controllata dall'intervento di un operatore.

Lo scopo è quello di conservare copie di riserva utili nei casi in cui il file system andasse accidentalmente distrutto. I danni possono essere causati da problemi hardware (di lettura o scrittura), sovraccarichi o cadute della tensione elettrica, rottura delle testine, sporcizia, ecc.

I file possono inoltre essere cancellati accidentalmente, e anche errori di programmazione possono causare la perdita del contenuto dei file.

## Tipi di accesso

La necessità di proteggere i file deriva direttamente dalla possibilità di accedervi. I sistemi che non permettono l'accesso ai file di altri utenti non richiedono la protezione; quindi si può ottenere una completa protezione proibendo l'accesso. In alternativa si può permettere un accesso totalmente libero senza alcuna protezione. Questi orientamenti sono entrambi eccessivi, quindi non si possono applicare in generale; ciò che serve in realtà è un accesso controllato.

Il controllo offerto dai meccanismi di protezione si ottiene limitando i possibili tipi di accesso. Gli accessi si permettono o si negano secondo diversi fattori, innanzitutto i tipi d'accesso richiesti.

Si possono controllare diversi tipi di operazione:

- *Lettura*: lettura da file
- *Scrittura*: scrittura o riscrittura di file
- *Esecuzione*: caricamento di file in memoria ed esecuzione
- *Aggiunta*: scrittura di nuove informazioni in coda ai file
- *Cancellazione*: cancellazione di file e liberazione del relativo spazio per un possibile riutilizzo
- *Elencazione*: elencazione del nome e degli attributi dei file

Si possono controllare anche altre operazioni, come ridenominazione, copiatura o modifica dei file. Tuttavia, in molti sistemi queste funzioni di livello superiore si possono realizzare tramite un programma di sistema che compie alcune syscall di livello inferiore, quindi la protezione viene garantita a livello inferiore.

Sono stati proposti molti meccanismi di protezione. Come sempre, ogni meccanismo presenta vantaggi e svantaggi, e dev'essere appropriato alla sua particolare applicazione.

## Controllo degli accessi

L'approccio più comune al problema della protezione è *rendere l'accesso dipendente dall'identità dell'utente*. Utenti differenti possono richiedere diversi tipi di accesso a un file o a una directory.

Lo schema più generale per realizzare gli accessi dipendenti dall'identità consiste nell'associare una **lista di controllo agli accessi** a ogni file o directory; in tale lista sono specificati i nomi degli utenti e i relativi tipi di accesso consentiti. Quando un utente richiede un accesso a un particolare file il sistema operativo esamina la lista di controllo degli accessi associata a quel file; se tale utente è presente nella lista per quel tipo di accesso, viene autorizzato, altrimenti si verifica una violazione della protezione e si nega l'accesso al file.

Questo sistema ha il vantaggio di permettere complessi metodi d'accesso. Il problema maggiore delle liste di controllo degli accessi è la loro lunghezza: per permettere a tutti di leggere un file, la lista deve contenere tutti gli utenti con accesso per la lettura. Questa tecnica comporta due inconvenienti:

- la costruzione di una lista di questo tipo può essere un compito noioso e non gratificante, soprattutto se la lista degli utenti del sistema non è nota a priori;
- l'elemento della directory, precedentemente di dimensione fissa, dev'essere di dimensione variabile, quindi anche la gestione dello spazio è più complicata.

Questi problemi si possono risolvere una versione condensata della lista di controllo degli accessi. Per condensarne la lunghezza, molti sistemi raggruppano gli utenti di ogni file in tre classi distinte:

- *Proprietario*: è l'utente che ha creato il file
- *Gruppo*: si tratta di un insieme di utenti che condividono il file e hanno bisogno di tipi di accesso simili
- *Universo*: tutti gli altri utenti del sistema

Il più comune orientamento recente prevede la combinazione delle liste di controllo degli accessi con lo schema di controllo degli accessi per proprietario, gruppo e universo (più facile da realizzare).

Affinché questo schema funzioni correttamente, è necessario uno stretto controllo dei permessi e delle liste di controllo degli accessi.

Nel sistema UNIX solo un utente con compiti di gestione può creare e modificare i gruppi, quindi questo controllo si ottiene con la partecipazione umana. Per definire la protezione, data questa più limitata classificazione, occorrono solo tre campi.

Normalmente ogni campo è formato da un insieme di bit, ciascuno dei quali permette o impedisce l'accesso che gli è associato.

Nel sistema UNIX sono definiti tre campi di tre bit ciascuno: rwx, dove r controlla l'accesso per la lettura, w quello per la scrittura e x per l'esecuzione. Tre campi separati sono riservati al proprietario del file, al gruppo proprietario e a tutti gli altri utenti. In questo schema, per registrare le informazioni di protezione sono necessari nove bit per file.

## Quali sono i metodi di accesso al file?

Accesso sequenziale: l'informazioni del file si elaborano ordinatamente, un record dopo l'altro. Un'operazione di lettura legge la prima porzione e fa avanzare automaticamente il puntatore del file che tiene traccia della locazione di I/O; analogamente, un'operazione di scrittura fa un'aggiunta in coda al file e avanza fino alla fine delle informazioni appena scritte, che costituisce la nuova fine del file. L'accesso sequenziale funziona nei dispositivi ad accesso sequenziale così come nei dispositivi ad accesso diretto.

Accesso diretto: un file è formato da elementi logici (record) di lunghezza fissa; ciò consente ai programmi di leggere e scrivere rapidamente tali elementi senza un ordine particolare. Il metodo ad accesso diretto si fonda su un modello di file che si rifà al disco. Non esistono limiti all'ordine di lettura o scrittura di un file ad accesso diretto.

Spesso le basi di dati sono di questo tipo: quando si presenta un'interrogazione riguardante un oggetto particolare, occorre stabilire quale blocco contiene la risposta alla richiesta e quindi leggere direttamente quel blocco, ottenendo così le informazioni richieste. Il numero del blocco fornito dall'utente al SO è normalmente un numero di blocco relativo.

Non tutti i SO gestiscono ambedue i tipi di accesso; alcuni permettono il solo accesso sequenziale, altri solo quello diretto. Alcuni sistemi richiedono che si definisca il tipo di accesso al file al momento della sua creazione; a tale file si può accedere soltanto nel modo definito. Tuttavia, si può facilmente simulare l'accesso sequenziale a un file ad accesso diretto mantenendo una variabile cp, che definisce la posizione corrente. D'altra parte, è estremamente goffo ed inefficiente simulare l'accesso diretto a un file che di per sé è ad accesso sequenziale.

## Come possono essere le directory?

- Livello singolo: Tutti i file sono contenuti nella stessa directory, facilmente gestibili e comprensibile. Una directory a livello singolo presenta però limiti notevoli che si manifestano all'aumentare del numero dei file in essa contenuti, oppure se il sistema è usato da più utenti.
- Due livelli: ogni utente dispone della propria directory utente (user file directory, UFD). Tutte le directory utente hanno una struttura simile, ma in ciascuna sono elencati solo i file del proprietario. Quando un utente fa un riferimento a un file particolare, il SO esegue la ricerca solo nella directory di quell'utente. In questo modo utenti diversi possono avere file con lo stesso nome, purché tutti i nomi di file all'interno di ciascuna directory utente siano unici. Per creare un file per un utente, in SO controlla che non ci sia un altro file con lo stesso nome, soltanto della directory di tale utente.
- Ad albero: Questa generalizzazione permette agli utenti di creare proprie sottodirectory e di organizzare i file di conseguenza. Una directory o una sottodirectory, contiene un insieme di file o di sottodirectory. Le directory sono semplicemente file, trattati però in modo speciale. Tutte le directory hanno lo stesso formato interno. La distinzione tra file e directory è data dal bit, rispettivamente 0 e 1, di ogni elemento della directory. Per creare e cancellare le directory si adoperano speciali chiamate di sistema. La directory corrente deve contenere la maggior parte dei file di interesse corrente per il processo.
- Grafo aciclico: Nel sistema esiste una directory condivisa o un file condiviso in due o più posizioni alla volta. Un grafo aciclico permette alla directory di avere sottodirectory e file condivisi. Lo stesso file o la stessa sottodirectory possono essere in due directory diverse. Il fatto che un file sia condiviso, o che lo sia una directory, non significa che ci siano due copie del file. Esiste un solo file effettivo, perciò tutte le modifiche sono immediatamente visibili. La condivisione è di particolare importanza se applicata alle sottodirectory; Un nuovo file appare automaticamente in tutte le sottodirectory condivise. Un metodo diffuso, esemplificato da molti tra i sistemi UNIX, prevede la creazione di un nuovo elemento di directory, chiamato collegamento. Un collegamento (link) è un puntatore a un altro file o un'altra directory.

## open() di un file nel sistema operativo

Aprire un file in un sistema operativo consiste nel recuperare i dati associati a un file dalla memoria del sistema e renderli disponibili per la lettura o per la modifica.

Il processo di apertura di un file in un sistema operativo può essere suddiviso in 3 fasi:

1. Allocazione di un descrittore di file: il sistema operativo alloca un descrittore di file, che rappresenta una struttura dati che contiene informazioni sul file, come il nome, la posizione su disco, i permessi di accesso e altro
2. Verifica dei permessi di accesso: il sistema operativo verifica che l'utente che richiede l'apertura del file abbia i permessi necessari, la richiesta di apertura del file viene neagata

3. Caricamento del file in memoria. il sistema operativo carica i dati del file dalla posizione su disco alla memoria, rendendoli disponibili per la lettura o la modifica

Una volta aperto il file, l'utente può eseguire diverse operazioni, come lettura, scrittura o modifica dei dati del file. Quando l'utente ha terminato di utilizzare il file, il sistema operativo lo chiude, liberando la memoria utilizzata e salvando eventuali modifiche apportate al file.

# Capitolo 14 - Sistemi Operativi

## Realizzazione del file system

### Struttura del file system

I dischi costituiscono la maggior parte della memoria secondaria in cui si conservano i file system. Hanno due caratteristiche importanti che ne fanno un mezzo adatto a questo scopo:

1. si possono riscrivere localmente; si può leggere un blocco dal disco, modificarlo e quindi scriverlo nella stessa posizione;
2. è possibile accedere direttamente a qualsiasi blocco di informazioni del disco, quindi risulta semplice accedere a qualsiasi file, sia in modo sequenziale sia in modo diretto, e passare da un file all'altro spostando le testine di lettura e scrittura e attendendo la rotazione del disco.

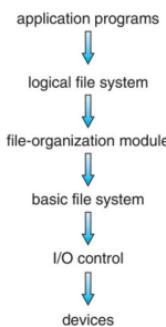
I dispositivi NVM sono sempre più utilizzati per l'archiviazione dei file e su di essi vengono quindi creati dei file system. Questi dispositivi differiscono dagli hard disk, in quanto non possono essere riscritti direttamente e presentano differenti performance.

Per migliorare l'efficienza dell'I/O, i trasferimenti tra memoria centrale e dischi si eseguono per **blocchi**. Ciascun blocco è composto da uno o più settori. A seconda dell'unità a disco, la dimensione dei settori è compresa tra 32 byte e 4096 byte; di solito è pari a 512 byte o 4096 byte.

Per fornire un efficiente e conveniente accesso al disco, il sistema operativo fa uso di uno o più file system che consentono di memorizzare, individuare e recuperare facilmente i dati. Un file system presenta due problemi di progettazione molto diversi. Il primo riguarda la definizione dell'aspetto del file system agli occhi dell'utente.

Questo compito implica la definizione di un file e dei suoi attributi, delle operazioni permesse su un file e della struttura delle directory per l'organizzazione dei file. Il secondo riguarda la creazione di algoritmi e strutture dati che permettono di far corrispondere il file logico ai dispositivi fisici di memoria secondaria.

Lo stesso file system è generalmente composto da molti livelli distinti:



Questa struttura è un esempio di struttura stratificata e ogni livello si serve delle funzioni inferiori per creare nuove impostazioni dai livelli superiori.

- *Controllo dell'I/O*: costituito dai driver dei dispositivi e dai gestori dei segnali d'interruzione. Un driver si occupa del trasferimento delle informazioni tra la memoria centrale e quella secondaria. Si può pensare ad un driver come un traduttore che riceve comandi ad alto livello ed emette specifiche istruzioni di basso livello per i dispositivi.
- *File system di base*: deve inviare dei generici comandi all'appropriato driver di dispositivo per leggere o scrivere blocchi fisici nel disco.
- *Modulo di organizzazione dei file*: è a conoscenza dei file, dei blocchi logici e fisici. Conoscendo il tipo di assegnazione dei file e la loro locazione, esso può tradurre gli indirizzi dei blocchi logici negli indirizzi dei blocchi fisici.
- *File system logico*: gestisce i metadati, ossia tutte le strutture del file system, eccetto gli effettivi dati. Il file system logico gestisce la struttura di directory per fornire al modulo di organizzazione dei file le informazioni di cui necessita. Esso mantiene le strutture dei file tramite i descrittori di file (FD), noti anche come *file control block* che contengono informazioni sui file.

Nei file system stratificati la duplicazione di codice è ridotta al minimo. Il controllo dell'I/O e, talvolta, il codice di base del file system, possono essere utilizzati da più di un file system. Ogni file system ha i propri moduli che gestiscono il file system logico e l'organizzazione dei file.

Sfortunatamente, la stratificazione può comportare un maggior overhead del sistema operativo, che può generare un conseguente decadimento delle prestazioni.

L'utilizzo della stratificazione e le scelte sul numero di strati da impiegare e sulle loro funzionalità rappresentano una grande sfida per la progettazione di nuovi sistemi.

I sistemi operativi dispongono, in genere, di più file system; Unix usa lo Unix file system (UFS) e Windows 2000 usa Windows NT file system (NTFS).

### Operazioni del file system

Per realizzare un file system si utilizzano parecchie strutture dati, sia nei dischi sia in memoria. Queste strutture variano a seconda del sistema operativo e del file system, ma esistono dei principi generali. Nei dischi, il file system mantiene informazioni su come eseguire l'avviamento di un sistema operativo memorizzato nei dischi stessi, il numero totale di blocchi, il numero e la locazione dei blocchi liberi, la struttura delle directory e i singoli file.

Fra le strutture presenti nei dischi ci sono le seguenti:

- *Il blocco di controllo dell'avviamento (boot control block)*: contiene le informazioni necessarie all'avviamento del sistema operativo. Nell'UFS si chiama boot block.
- *Il blocco di controllo del volume (volume control block)*: ciascuno di essi contiene i dettagli riguardanti il relativo volume (o partizione), come il numero e la dimensione dei blocchi nella partizione, il contatore dei blocchi liberi e i relativi puntatori, il contatore degli FCB liberi e i relativi puntatori. Nell'UFS si chiama super blocco; nell'NTFS si chiama tabella principale dei file (MFT).
- *La struttura della directory*: una per file system, usata per organizzare i file. Nel caso dell'UFS comprende i nomi dei file e i numeri di inode associati. Nel caso dell'NTFS è memorizzata nella tabella principale dei file.
- *Il blocco di controllo del file (FCB)*: contenente molti dettagli del relativo file. Ha un identificatore unico per poterlo associare a una voce della directory. Nell'NTFS, queste informazioni sono memorizzate all'interno della tabella principale dei file, che si serve di una struttura base di dati relazionale, con una riga per ciascun file.

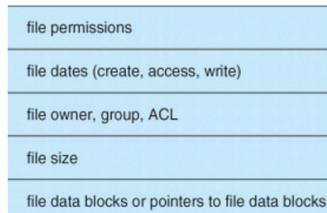
Le informazioni tenute in memoria servono sia per la gestione del file system sia per migliorare le prestazioni attraverso l'uso di cache. I dati si caricano al momento del montaggio, si aggiornano mentre si opera sul file system e si eliminano allo smontaggio.

Le strutture che vi possono essere incluse sono di diverso tipo:

- *La tabella di montaggio*: in memoria, che contiene informazioni relative a ciascun volume montato;
- *Una cache della struttura della directory*: tenuta in memoria, contenente le informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente (per le directory che costituiscono dei punti di montaggio, può essere prelevante un puntatore alla tabella dei volumi);
- *La tabella di sistema dei file aperti*: contenente una copia dell'FCB per ciascun file aperto, insieme con altre informazioni;
- *La tabella dei file aperti per ciascun processo*: contenente un puntatore al corrispondente elemento della tabella generale dei file aperti, insieme con altre informazioni;
- *I buffer*: conservano blocchi del file system durante la loro lettura o scrittura sul disco.

Le applicazioni, per creare un nuovo file, eseguono una chiamata al file system logico, il quale conosce il formato della struttura della directory. Per creare un nuovo file, esso crea un nuovo FCB. (In alternativa, nel caso dei file system che creano tutti gli FCB al momento della loro installazione, esso alloca semplicemente un FCB libero). Il sistema carica quindi la directory appropriata in memoria, la aggiorna con il nome del nuovo file e con l'FCB associato, e la scrive nuovamente sul disco.

Tipica struttura di FCB:



Alcuni sistemi operativi, compreso UNIX, trattano le directory esattamente come i file, distinguendole con un campo che indica che si tratta di una directory. Altri, tra cui il sistema operativo Windows, dispongono di chiamate di sistema distinte per i file e le directory e trattano le directory come entità separate dai file. Indipendentemente da tali questioni strutturali, il file system logico può basarsi sul modulo che si occupa dell'organizzazione dei file per far corrispondere l'I/O su directory ai numeri di blocchi di disco, che poi si passano al file system di base e al sistema per il controllo dell'I/O.

## Utilizzo

Una volta creato un file, per essere usato per operazioni di I/O deve essere aperto. La chiamata di sistema open() passa un nome di file al file system logico. Per controllare se il file sia già in uso da parte di qualche processo, la chiamata open() dapprima esamina la tabella di sistema dei file aperti. In caso affermativo, aggiunge un elemento alla tabella dei file aperti del processo che punta alla tabella dei file aperti in tutto il sistema. Questo algoritmo può eliminare significativi overhead. Se il file non è già aperto, se ne ricerca il nome all'interno della directory. Alcune porzioni della struttura delle directory sono di solito tenute in memoria per accelerare le operazioni sulle directory. Una volta trovato il file, si copia l'FCB nella tabella di sistema dei file aperti tenuta in memoria. Questa tabella non solo contiene l'FCB, ma tiene anche traccia del numero di processi che in quel momento hanno il file aperto.

Successivamente, si crea un elemento nella tabella dei file aperti del processo con un puntatore alla tabella di sistema e con alcuni altri campi. Questi altri campi possono comprendere un puntatore alla posizione corrente nel file (per successive operazioni read() o write()) e il tipo di accesso specificato dall'apertura del file. La open() riporta un puntatore all'elemento appropriato nella tabella dei file aperti del processo, sicché tutte le operazioni sul file si svolgeranno usando questo puntatore.

Il nome del file potrebbe non essere contenuto nella tabella dei file aperti, visto che, una volta che il corrispondente FCB è stato individuato nei dischi, il sistema non ne ha bisogno. Tuttavia, potrebbe venir memorizzato in una cache per risparmiare tempo sulle aperture successive dello stesso file. Il nome dato all'elemento della tabella è detto **descrittore di file** in UNIX, e **handle del file** in Windows.

Quando un processo chiude il file, si cancella il relativo elemento nella tabella dei file aperti del processo e si decrementa il contatore associato al file nella tabella di sistema. Se tutti i processi che avevano aperto il file lo hanno chiuso, si riscrivono i metadati aggiornati nella struttura della directory nei dischi e si cancella il relativo elemento nella tabella di sistema dei file aperti.

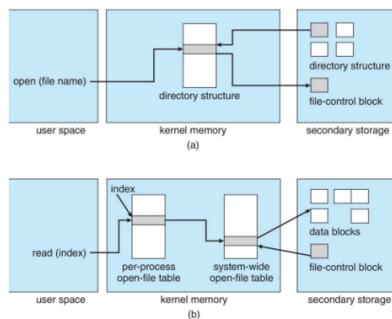
Alcuni sistemi complicano ulteriormente lo schema descritto, usando il file system come interfaccia per altri aspetti del sistema, come la comunicazione in rete. Per esempio, nell'UFS, la tabella generale dei file aperti contiene gli *inode* e altre informazioni su file e directory, ma contiene anche informazioni simili per le connessioni di rete e i dispositivi. In questo modo si può usare un unico meccanismo per molteplici fini.

Le questioni concernenti l'uso delle cache per queste strutture non vanno però trascurate. La maggior parte dei sistemi mantiene in memoria tutta l'informazione su un file aperto, eccetto i suoi effettivi blocchi di dati. Il sistema UNIX BSD è noto per il suo uso di cache ovunque sia possibile.

risparmiare su operazioni di I/O nei dischi.

La sua frequenza media di successi nella cache, pari all'85%, dimostra l'utilità di queste tecniche.

Figura che riassume le strutture dati che si usano nella realizzazione di un file system:



## Realizzazione delle directory

La selezione degli algoritmi di allocazione e degli algoritmi di gestione delle directory ha un grande effetto sull'efficienza, le prestazioni e l'affidabilità del file system.

### Lista lineare

Il più semplice metodo di realizzazione di una directory è basato sull'uso di una lista lineare contenente i nomi dei file con puntatori a blocchi di dati. Questo metodo è di facile programmazione, ma la sua esecuzione è onerosa in termini di tempo. Per creare un nuovo file occorre prima esaminare la directory per essere sicuri che non esista già un file con lo stesso nome, quindi aggiungere un nuovo elemento alla fine della directory. Per cancellare un file occorre cercare nella directory il file con quel nome, quindi rilasciare lo spazio che gli era stato assegnato. Esistono vari metodi per riutilizzare un elemento della directory: si può contrassegnare l'elemento come non usato (attribuendogli un nome speciale, come un nome blank, oppure includendo un bit d'uso in ogni elemento), oppure può essere aggiunto a una lista di elementi di directory liberi; una terza possibilità prevede la copiatura dell'ultimo elemento della directory in una locazione liberata e la diminuzione della lunghezza della directory. Per ridurre il tempo di cancellazione di un file si può usare anche una *lista concatenata*.

Il vero svantaggio dato da una lista lineare di elementi di directory è dato dalla ricerca lineare di un file. Le informazioni sulla directory vengono usate frequentemente, e gli utenti si accorgono se l'accesso a tali informazioni è lento. In effetti, molti sistemi operativi impiegano una cache software per memorizzare le informazioni di directory usate più recentemente. La presenza nella cache delle informazioni richieste ne evita la continua rilettura dai dischi. Una lista ordinata permette una ricerca binaria e riduce il tempo medio di ricerca, tuttavia il requisito dell'ordinamento può complicare la creazione e la cancellazione di file, poiché, per tenere ordinata la lista, può essere necessario spostare quantità notevoli di informazioni di directory. In questo caso, può essere d'aiuto una struttura dati più raffinata, come un albero bilanciato.

Un vantaggio della lista ordinata è che consente di produrre l'elenco ordinato del contenuto della directory senza una fase d'ordinamento separata.

### Tabella hash

Un'altra struttura dati che si usa per realizzare le directory è la tabella hash. In questo caso una lista lineare contiene gli elementi di directory, ma si usa anche una struttura dati hash. La tabella hash riceve un valore calcolato a partire dal nome del file e riporta un puntatore al nome del file nella lista lineare. Questa struttura può diminuire notevolmente il tempo di ricerca nella directory. L'inserimento e la cancellazione sono abbastanza semplici, anche se occorre prendere provvedimenti per gestire le **collisioni**, cioè *situazioni in cui da due nomi di file si ottiene un riferimento alla stessa locazione*.

Le maggiori difficoltà legate a una tabella hash sono la sua dimensione, che in generale è fissa, e la dipendenza della funzione hash da tale dimensione. Si supponga, per esempio, di realizzare una tabella hash di 64 elementi con gestione lineare delle collisioni; la funzione hash converte i nomi di file in interi da 0 a 63, per esempio, usando il resto di una divisione per 64. Per creare in un secondo tempo un sessantacinquesimo file occorre allungare la tabella hash della directory, per esempio fino a 128 elementi. Occorre quindi una nuova funzione hash per associare i nomi di file all'intervallo compreso tra 0 e 127, e gli elementi esistenti nella directory si devono riorganizzare in modo da riflettere i loro nuovi valori della funzione hash.

Alternativamente, ciascun elemento della tabella hash, anziché un singolo valore, può essere una lista concatenata; ciò consente di risolvere le collisioni aggiungendo il nuovo elemento alla lista concatenata. Le ricerche vengono alquanto rallentate, poiché la ricerca per nome può richiedere l'attraversamento di una lista concatenata degli elementi in collisione della tabella hash; tuttavia tale metodo è verosimilmente più veloce di una ricerca lineare nell'intera directory.

## Metodi di allocazione

La natura ad accesso diretto de dischi dà flessibilità nella realizzazione dei file. In quasi tutti i casi, molti file si memorizzano nello stesso disco. Il problema principale consiste dunque nell'allocare lo spazio per questi file in modo che lo spazio nel disco sia usato efficientemente e l'accesso ai file sia rapido. Esistono tre metodi principali per l'allocazione dello spazio di un disco; può essere contigua, concatenata o indicizzata. Ciascuno di questi metodi presenta vantaggi e svantaggi. Anche se alcuni sistemi dispongono di tutti e tre i metodi, più spesso un sistema usa un unico metodo per tutti i file all'interno di un certo tipo di file system.

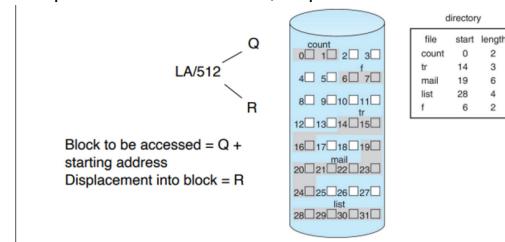
### Allocazione contigua

Ogni file deve occupare un insieme di blocchi contigui del disco. L'assegnazione contigua dello spazio per un file è definita dall'indirizzo del primo blocco e dalla lunghezza. L'elemento di directory per ciascun file indica queste due informazioni.

Problemi: assegnazione dinamica della memoria. Occorre soddisfare una richiesta di dimensione  $n$  a partire da una lista di buchi liberi. Questi algoritmo di assegnazione soffre di frammentazione esterna: la memoria viene frammentata in tanti buchi, nessuno dei quali è in grado di contenere i dati. Una soluzione a questo problema è quella della compattazione dello spazio libero in uno spazio contiguo.

Un altro problema è la determinazione della quantità di spazio necessaria per un file. Esiste il problema di conoscere la dimensione del file da creare. Se un file riceve poco spazio può essere impossibile estenderlo; a proposito esistono due possibilità:

- il programma utente termina in modo che l'utente assegna più spazio al programma
- si trova un buco più grande nel quale copiare il contenuto del file; in questo caso non occorre informare esplicitamente l'utente

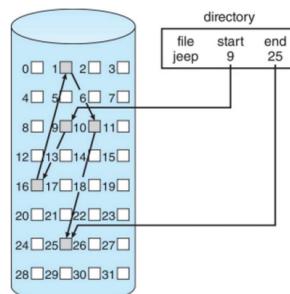


Inoltre, per un file che cresce lentamente si deve assegnare spazio sufficiente per la sua dimensione finale, anche se molto di questo spazio può rimanere inutilizzato per parecchio tempo (frammentazione interna).

## Allocazione concatenata

Risolve tutti i problemi dell'allocazione contigua. Ogni file è composto da una lista concatenata di blocchi i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo ed ultimo blocco del file. Ogni blocco contiene un puntatore al blocco successivo. Un'operazione di scrittura determina la ricerca di un blocco libero attraverso il sistema di gestione dello spazio libero, la scrittura di tale blocco e la sua concatenazione alla fine del file.

Per leggere un file occorre leggere i blocchi seguendo i puntatori. Con l'assegnazione concatenata non esiste la frammentazione esterna, non è necessario dichiarare la dimensione di un file al momento della sua creazione.



Svantaggi: per accedere all' $i$ -esimo blocco di un file occorre partire dall'inizio e seguire i puntatori.

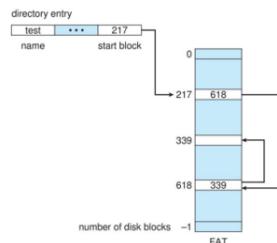
Un altro svantaggio riguarda lo spazio richiesto per i puntatori. Ogni file richiede un po' più di spazio di quanto ne richiederebbe altrimenti. La soluzione più comune è riunire un certo numero di blocchi in gruppi (*cluster*) e nell'assegnare i cluster anziché i blocchi. Così i puntatori usano una quantità di spazio che si riduce in modo proporzionale al numero di blocchi per gruppo. Il costo di questo metodo è dato da un incremento della frammentazione interna, poiché se un gruppo di blocchi è parzialmente pieno si spreca più spazio di quanto se ne sprecherebbe con un solo blocco parzialmente pieno.

Un altro problema dell'allocazione concatenata riguarda l'affidabilità. Poiché i file sono tenuti insieme da puntatori sparsi per tutto il disco, s'immagini cosa accadrebbe se un puntatore andasse perduto o danneggiato.

Un errore di programmazione del sistema operativo oppure un errore hardware di un'unità a disco potrebbero causare la lettura di un puntatore errato. Questo errore, a sua volta, potrebbe causare il collegamento alla lista dei blocchi liberi oppure a un altro file. Una soluzione parziale a tale problema consiste nell'usare liste doppiamente concatenate oppure nel memorizzare il nome del file e il relativo numero di blocco in ogni blocco; questi schemi però richiedono un overhead ancora maggiore per ogni file.

## FAT

Una variante del metodo di allocazione concatenata consiste nell'uso della *tavella di assegnazione dei file* (FAT, File Allocation Table), tipica dei vecchi sistemi Microsoft, ora utilizzata per le unità removibili.



Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascun volume; la FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; si usa essenzialmente come una lista concatenata. L'elemento di directory contiene il numero del primo blocco del file. L'elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero del blocco successivo del file.

Questa catena continua fino all'ultimo blocco, che ha come elemento della tabella un valore speciale di fine del file. I blocchi inutilizzati sono indicati

nella tabella da un valore 0. L'allocazione di un nuovo blocco a un file richiede semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine del file precedente con l'indirizzo del nuovo blocco; lo 0 è quindi sostituito con il valore di fine del file.

Lo schema di allocazione basato sulla FAT, se non si usa una cache, può causare un significativo numero di posizionamenti della testina. La testina del disco deve spostarsi all'inizio del volume per leggere la FAT e trovare la locazione del blocco in questione, quindi raggiungere la locazione del blocco stesso; nel caso peggiore sono necessari ambedue i movimenti per ciascun blocco.

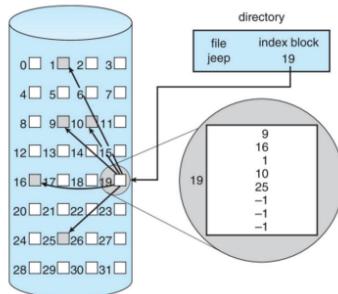
Un vantaggio è dato dall'ottimizzazione del tempo d'accesso diretto, poiché la testina del disco può trovare la locazione di ogni blocco leggendo le informazioni contenute nella FAT.

La FAT, dunque, è una componente fondamentale del file system di MS-DOS che utilizza un metodo di allocazione dei file. Questo metodo utilizza una tabella che tiene traccia di tutto lo spazio disponibile in hard disk. Ogni elemento di directory contiene il numero del primo cluster che compone il file. Una volta letto il contenuto di questo cluster, il suo numero viene usato come indice all'interno della FAT in modo da scoprire quale sia il cluster che contiene il pezzo di file successivo; questa operazione viene ripetuta finché non viene scoperto l'EOF.

## Allocazione indicizzata

L'allocazione concatenata risolve il problema della frammentazione esterna e quello della dichiarazione delle dimensioni dei file, presenti nell'allocazione contigua. Tuttavia, in mancanza di una FAT, l'allocazione concatenata non è in grado di sostenere un efficiente accesso diretto, poiché i puntatori ai blocchi sono sparsi, con i blocchi stessi, per tutto il disco e si devono recuperare in ordine.

L'allocazione indicizzata (soluzione adottata da UNIX), risolve questo problema, raggruppando tutti i puntatori in una sola locazione: il *blocco indice*.



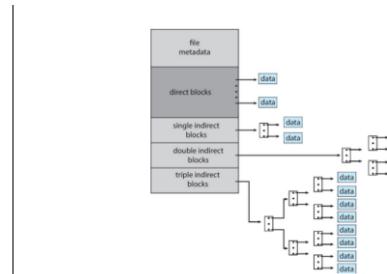
Ogni file ha il proprio blocco indice: si tratta di un array d'indirizzi di blocchi del disco. L'i-esimo elemento del blocco indice punta all'i-esimo blocco del file. La directory contiene l'indirizzo del blocco indice. Per individuare e leggere l'i-esimo blocco occorre usare il puntatore che si trova nell'i-esimo blocco indice. Questo schema è simile a quello della paginazione. Una volta creato il file, tutti i puntatori del blocco indice sono impostati a null. Quando si scrive l'i-esimo blocco per la prima volta, il gestore dei blocchi liberi fornisce un blocco: l'indirizzo di questo blocco viene inserito nell'i-esimo elemento del blocco indice. Poiché ogni blocco libero del disco può soddisfare una richiesta di maggiore spazio, l'allocazione indicizzata consente l'accesso diretto senza soffrire di frammentazione esterna.

L'allocazione indicizzata soffre tuttavia di un overhead maggiore: lo spazio aggiuntivo richiesto dai puntatori del blocco indice è generalmente maggiore dello spazio aggiuntivo necessario per l'allocazione concatenata. Si consideri il comune caso di un file con uno o due blocchi; con l'allocazione concatenata si perde il solo spazio di un puntatore per blocco, complessivamente uno o due puntatori; con l'allocazione indicizzata occorre allocare un intero blocco indice, anche se solo uno o due puntatori sono diversi da null.

Questo punto solleva la questione della dimensione del blocco indice. Ogni file deve avere un blocco indice, quindi è auspicabile che questo sia quanto più piccolo è possibile; ma se il blocco indice è troppo piccolo non può contenere un numero di puntatori sufficiente per un file di grandi dimensioni, quindi è necessario disporre di un meccanismo per gestire questa situazione.

Fra i possibili meccanismi vi sono i seguenti:

- *Schema concatenato*: per permettere la presenza di file lunghi è possibile collegare tra loro parecchi blocchi indice.
- *Indice a più livelli*: un blocco indice di primo livello punta ad un insieme di blocchi indice di secondo livello che a loro volta puntano ai blocchi del file.
- *Schema combinato*: utilizzata nei sistemi basati su UNIX, consistente nel tenere i primi 15 puntatori del blocco indice nell'*inode* del file. I primi 12 di questi puntatori puntano a blocchi diretti che contengono dati del file, cioè contengono direttamente gli indirizzi di blocchi contenenti dati del file. Gli altri tre puntatori puntano a blocchi indiretti. Il primo è un puntatore a un blocco indiretto singolo; si tratta di un blocco indice che non contiene dati, ma indirizzi di blocchi che contengono dati. Il secondo è un puntatore a un blocco diretto doppio contenente l'indirizzo di un blocco che a sua volta contiene gli indirizzi di blocchi contenenti puntatori agli effettivi blocchi di dati. Il terzo è un puntatore a un blocco indiretto triplo.



Con questo metodo il numero dei blocchi che si può allocare a un file supera la quantità di spazio che possono indirizzare i puntatori a file di 4 byte usati da molti sistemi operativi.

Un puntatore a file di 32 bit consente di arrivare a soli  $2^{32}$  byte, 4GB.

Gli schemi d'allocazione indicizzata soffrono di alcuni dei problemi di prestazioni dell'allocazione concatenata. In particolare, i blocchi indice si possono caricare in memoria, ma i blocchi dei dati possono essere sparsi per un intero volume.

Inode: una directory contiene una tabella che elenca i file contenuti nella directory stessa, dove ai nomi dei file vengono assegnati i corrispondenti numeri di inode. Un inode è un file speciale, progettato per essere letto dal Kernel al fine di conoscere alcune informazioni su ciascun file. Esso specifica i permessi del file, il proprietario del file, la data di creazione, quella dell'ultimo accesso e la posizione fisica dei blocchi di dati su disco che contengono un file. Si possono avere più voci che puntano allo stesso inode. Ogni inode ha un contatore (link count) che contiene il numero di riferimenti che sono stati fatti ad esso. Solo quando questo contatore si annulla i dati del file vengono rimossi dal disco. In realtà non si cancella (unlink) i dati del file, ma si limita ad eliminare la relativa voce da una directory e decrementare il numero di riferimenti all'inode. Quando si cambia nome ad un file senza cambiare file system, il contenuto del file non viene spostato fisicamente, ma viene creata semplicemente una nuova voce per l'inode in questione e rimossa la vecchia.

Hard e Soft link: l'utilità del link è varia. Esempio: se si devono modificare diversi file uguali, in diverse directory, con l'uso del link la modifica fatta su un file diventa immediatamente effettiva anche sugli altri file. Un'hard link è fondamentalmente un altro nome per lo stesso file, in pratica vi sono due nomi che puntano alla stessa area su disco, in termini tecnici due elementi di directory puntano allo stesso i-node. Un soft link è un nome che punta ad un altro nome, e solo quest'ultimo punta all'i-node del file su disco. Notiamo che cancellando il nome di un hard link non si cancella il file dal disco ma solo l'hard link indicato. Solo quando tutti gli hard link ad un file sono stati rimossi, il file su disco non è più utilizzabile e l'area di disco può essere riutilizzata per nuovi file. Nel caso di soft link, la rimozione del vero file rende il soft link inutilizzabile, in quanto quest'ultimo punta al nome del file e non direttamente al suo i-node.

## Gestione dello spazio libero

Poiché la quantità di spazio dei dischi è limitata, è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere nuovi file, se possibile (i dischi ottici a una sola scrittura permettono una sola scrittura in qualsiasi settore e quindi il riutilizzo è fisicamente impossibile). Per tener traccia dello spazio libero in un disco, il sistema conserva una lista dello spazio libero; vi sono registrati tutti gli spazi liberi, cioè non allocati ad alcun file o directory. Per creare un file occorre cercare nella lista dello spazio libero la quantità di spazio necessaria e assegnarla al nuovo file, quindi rimuovere questo spazio dalla lista. Quando si cancella un file, si aggiungono alla lista dello spazio libero i blocchi di disco a esso assegnati. A dispetto del suo nome, la lista dello spazio libero potrebbe non essere realizzata come una lista.

### Vettore di bit

Spesso la lista dello spazio libero si realizza come una mappa di bit, o vettore di bit. Ogni blocco è rappresentato da un bit: se il blocco è libero, il bit è 1, se è assegnato il bit è 0.

Il vantaggio principale di questa tecnica è la semplicità ed efficienza nel trovare il primo blocco libero o gli n blocchi liberi consecutivi nel disco. Sfortunatamente, i vettori di bit sono efficienti solo se tutto il vettore è mantenuto nella memoria centrale e viene di tanto in tanto scritto nella memoria secondaria.

### Lista concatenata

Un altro metodo di gestione degli spazi liberi consiste nel collegarli tutti, tenere un puntatore al primo di questi in una speciale locazione del disco e caricarlo in memoria. Il primo blocco libero contiene un puntatore al successivo, e così via.

Questo schema non è efficiente; per attraversare la lista è infatti necessario leggere ogni blocco, con un notevole tempo di I/O. Fortunatamente la necessità di attraversare la lista dello spazio libero non è frequente.

Di solito il sistema operativo ha semplicemente bisogno di un blocco libero perché possa assegnarlo a un file, quindi si usa il primo blocco della lista. Il metodo che fa uso della FAT include la lista dei blocchi liberi nella struttura dati per l'allocazione; non è necessario un metodo separato.

### Raggruppamento

Una possibile modifica del metodo della lista dello spazio libero prevede la memorizzazione degli indirizzi di n blocchi liberi nel primo di questi. I primi n-1 di questi blocchi sono effettivamente liberi; l'ultimo blocco contiene gli indirizzi di altri n blocchi liberi, e così via. Con questo metodo, diversamente dall'ordinaria lista concatenata, è possibile trovare rapidamente gli indirizzi di un gran numero di blocchi liberi.

### Conteggio

Un altro metodo è quello del conteggio: anziché tenere un elenco di n indirizzi liberi, è sufficiente tenere l'indirizzo del primo blocco libero e il numero n blocchi liberi contigui che seguono il primo blocco.

### TRIM dei blocchi non utilizzati (per dischi a stato solido)

Gli HDD e altri supporti di memorizzazione che consentono la sovrascrittura dei blocchi per effettuare aggiornamenti necessitano solo di una lista di blocchi liberi per poter gestire lo spazio libero. I blocchi non devono essere trattati in modo speciale una volta liberati e un blocco liberato conserva in genere i suoi dati (ma senza alcun puntatore al blocco) finché i dati non vengono sovrascritti una volta che il blocco viene successivamente riassegnato.

I dispositivi di archiviazione che non consentono la sovrascrittura basati su flash, risentono negativamente se vengono applicati gli stessi algoritmi. Tali dispositivi devono essere cancellati prima di poter essere scritti nuovamente, e che le cancellazioni devono essere fatte su grandi raggruppamenti di dati (blocchi, formattati da pagine) e richiedono un tempo relativamente lungo rispetto a letture o scritture.

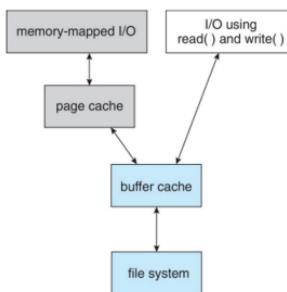
E' necessario quindi un nuovo meccanismo per consentire al file system di segnalare al dispositivo che una pagina è libera e può essere considerata per la cancellazione (una volta che il blocco contenente la pagina è completamente libero). La tecnica utilizzata varia in base al controllore del dispositivo di archiviazione: sulle unità con collegamento ATA viene utilizzato il comando TRIM, mentre per l'archiviazione basata su NVMe si utilizza il comando unallocates. Qualunque sia il comando specifico del controllore, questo meccanismo mantiene lo spazio di archiviazione disponibile per la scrittura. Senza una tale funzionalità, il dispositivo di archiviazione si riempirebbe e richiederebbe la garbage collection e la cancellazione dei blocchi, con conseguente riduzione delle prestazioni dell'I/O di scrittura. Con TRIM o funzionalità simili, la garbage collection e la cancellazione possono essere eseguite prima che il dispositivo arrivi a essere quasi del tutto pieno, consentendo al dispositivo di fornire prestazioni più uniformi.

## Efficienza e prestazioni

I dischi tendono ad essere il principale collo di bottiglia per le prestazioni di un sistema, essendo i più lenti tra i componenti principali di un calcolatore.

L'uso efficiente di un disco dipende fortemente dagli algoritmi utilizzati per l'assegnazione del disco e la gestione delle directory. Ad esempio, l'assegnazione preventiva degli inode migliora le prestazioni del file system, come il tentativo di mantenere i blocchi di dati di un file vicini al blocco che contiene l'inode allo scopo di ridurre il tempo di posizionamento. Un altro aspetto sono i tipi di dati contenuti in un elemento di una directory. Ogni qualvolta si legge un file si devono aggiornare i campi delle directory; questo può essere inefficiente per file a cui si accede frequentemente; quindi nella fase di progettazione del file system è necessario confrontare i benefici con i costi rispetto alle prestazioni. Ci sono diversi modi per migliorare le prestazioni. Alcuni sistemi riservano una sezione separata come cache del disco, dove tenere i blocchi in previsione di un loro riutilizzo entro breve tempo. Altri sistemi impiegano una cache delle pagine per i file; si tratta di una soluzione che impiega tecniche di memoria virtuale per la gestione dei dati dei file come pagine anziché come blocchi di file system.

Questo metodo è noto come **memoria virtuale unificata**. Alcune versioni di UNIX prevedono la *buffer cache unificata*.



Non utilizzando la buffer cache unificata si verifica il fenomeno del double caching: la tecnica memory-mapped prevede la lettura dei blocchi dal file system e la loro memorizzazione nella buffer cache (buffer cache = cache per accesso al disco).

Poiché il sistema di MV non può interfacciarsi con la buffer cache, si deve copiare nella cache delle pagine il contenuto del file presente nella buffer cache. Un processo che scrive in un disco, in realtà scrive semplicemente il contenuto del file presente nella buffer cache. Un processo che scrive in un disco, in realtà scrive semplicemente nella cache, successivamente il sistema scrive i dati nel disco in modo asincrono. Il processo utente in questo modo percepisce scritture rapide.

- Le *scritture sincrone* avvengono nell'ordine in cui le riceve il sistema e non subiscono la memorizzazione transitoria. Quindi la procedura chiamante prima di proseguire deve attendere che i dati raggiungano l'unità a disco.
- Nella maggior parte dei casi si utilizzano *scritture asincrone*: i dati si memorizzano nella cache e si restituisce immediatamente il controllo alla procedura chiamante.

Alcuni sistemi ottimizzano la cache delle pagine utilizzando diversi algoritmi di sostituzione. Il rilascio all'indietro rimuove una pagina dalla memoria non appena si verifica una richiesta della pagina successiva. Con la lettura anticipata si leggono e si mettono nella cache la pagina richiesta e parecchie pagine successive: è probabile che queste pagine siano richieste una volta terminata l'elaborazione della pagina corrente.

Generalmente, parte delle informazioni contenute nelle directory è mantenuta in memoria centrale allo scopo di accelerare gli accessi. Nel caso di un crollo del sistema la tabella dei file aperti viene persa e con essa qualsiasi modifica alle directory alle quali i file aperti appartengono. Al riavvio del sistema si esegue uno speciale programma: il *verificatore della coerenza*. Esso confronta i dati delle directory con quelli contenuti nei blocchi dei dischi tentando di correggere ogni incoerenza. La scelta dell'algoritmo di assegnazione e di gestione dello spazio libero influiscono sull'operazione di ripristino. Se si è utilizzato un algoritmo di assegnazione indicizzato la perdita di un elemento della directory potrebbe essere disastroso poiché ogni blocco di dati non contiene alcuna informazione sugli altri blocchi di dati. Per questo motivo UNIX gestisce tramite cache gli elementi delle directory per letture, mentre le operazioni di scrittura vengono eseguite in modo sincrono.

Per il ripristino del sistema è possibile usare programmi che consentono di fare delle copie di riserva (backup) dei dati residenti nei dischi in altri dispositivi di registrazione dei dati.

Un'altra possibile soluzione è il *file system con annotazione delle modifiche*. Il metodo che abbiamo visto prima presenta diversi problemi tra cui la possibilità che l'incoerenza sia irrisolvibile e che richiede molto tempo e risorse. Il file system con annotazione delle modifiche registra tutte le modifiche dei metadati in un giornale delle modifiche (file di registrazione). Ogni insieme di operazioni che eseguono uno specifico compito prende il nome di transazione. Una volta che le modifiche sono riportate nel file di registrazione, le operazioni si considerano portate a termine con successo (committed) e la chiamata del sistema può restituire il controllo al processo utente. Nel frattempo si applicano alle effettive strutture del file system le operazioni scritte nel giornale delle modifiche. Se si verifica un crollo del sistema, nel giornale delle modifiche ci potranno essere 0 o più transazioni. Le transazioni presenti non sono mai state completate nel file system; esse si possono eseguire a partire dalla posizione corrente del puntatore fino al completamento. Nel caso in cui una transazione sia fallita si devono annullare tutti i cambiamenti che erano stati applicati al file system.

NFS: è un file system di rete (network file system). Esso usa un metodo client-server per permettere agli utenti di accedere a file e directory in calcolatori remoti come se fossero in file system locali. Affinché una directory remota sia accessibile a un calcolatore, un client di quel calcolatore deve prima eseguire un'operazione di montaggio. Una directory remota si monta in corrispondenza di una directory di un file system locale. Una volta terminata l'operazione di montaggio, la directory montata assume l'aspetto di un sottoalbero del file system locale e sostituisce il sottoalbero che discende dalla directory locale; questa a sua volta, rappresenta la radice della directory appena montata. Montando un file system remoto il client non acquisisce l'accesso ai file system che erano montati sopra il primo file system: il meccanismo di montaggio non ha la proprietà transitiva. La definizione del NFS distingue tra i servizi offerti da un meccanismo di montaggio e gli effettivi servizi d'accesso ai file remoti. Di conseguenza sono definiti due protocolli. Il primo è il protocollo di montaggio stabilisce una connessione logica iniziale tra un server e un client. L'operazione comprende il nome della directory remota da montare e il nome del calcolatore server in cui tale directory è memorizzata. Il server conserva una lista di esportazione che specifica i file system locali esportati per il montaggio e i nomi dei calcolatori ai quali è permessa tale operazione. Quando il server riceve una richiesta di montaggio, riporta al client un file handle (maniglia di file) da usare come chiave per ulteriori accessi ai file che si trovano all'interno del file system remoto. Il file handle è composto da un identificatore di file system e da un numero di inode per identificare la directory montata all'interno del file system esportato. Il protocollo NFS offre una serie di RPC per operazioni su file remoti quali: ricerca di un file in una directory, lettura di un insieme di elementi di una directory, accesso di attributi di file, lettura e scrittura dei file. Queste procedure si possono invocare soltanto dopo aver stabilito un file handle per la directory montata in modo remoto. Una caratteristica importante dei server NFS è l'assenza dell'informazione di stato: i server non conservano informazioni sui client tra un accesso e l'altro.

## Ripristino

Poiché i file e le directory sono mantenuti sia in memoria centrale sia nei dischi, è necessario aver cura di assicurare che il verificarsi di un malfunzionamento nel sistema non comporti la perdita di dati o la loro incoerenza.

Una caduta del sistema può causare incoerenze tra le strutture dati del file system su disco, come le strutture delle directory, i puntatori ai blocchi liberi e i puntatori agli FCB liberi. Molti file system applicano delle modifiche direttamente a queste strutture. Operazioni comuni come la creazione di un file possono comportare molti cambiamenti strutturali all'interno del file system di un disco. Le strutture delle directory vengono modificate, gli FCB e i blocchi di dati allocati e i contatori degli elementi liberi per tutti questi blocchi diminuiti. Quando queste modifiche sono interrotte da una caduta del sistema, ne possono derivare incoerenze tra le strutture. Per esempio, il contatore degli FCB liberi potrebbe indicare che un FCB è stato allocato, ma la struttura della directory potrebbe non avere un puntatore a quel FCB. L'utilizzo della cache che i sistemi operativi adottano per ottimizzare le prestazioni di I/O aggrava questo problema. Alcuni cambiamenti potrebbero andare direttamente sul disco, mentre altri possono essere memorizzati nella cache. Se i cambiamenti nella cache non raggiungono il disco prima che si verifichi una caduta, è possibile che la situazione peggiori ulteriormente.

Inoltre, anche i bachi nell'implementazione del file system, i controllori del disco, e persino le applicazioni utente possono indurre incoerenze nel file system. I file system hanno svariati metodi per affrontare queste circostanze, a seconda delle strutture dati e degli algoritmi che utilizzano.

## Verifica della coerenza

Quale che sia la causa degli errori, un file system deve prima scoprire i problemi e poi correggerli. Per scoprire gli errori vengono esaminati tutti i metadati su ogni file system per verificare la coerenza del sistema. Per scoprire gli errori vengono esaminati tutti i metadati su ogni file system per verificare la coerenza del sistema. Sfortunatamente, questo procedimento può richiedere diversi minuti, o addirittura delle ore, e dovrebbe avvenire tutte le volte che il sistema si avvia. In alternativa, un file system può registrare il suo stato all'interno dei metadati. All'inizio di ogni serie di modifiche dei metadati è impostato un bit di stato per indicare che i metadati sono in stato di modifica. Se tutti gli aggiornamenti dei metadati si completano con successo, il file system può azzerare quel bit. Se tutti gli aggiornamenti dei metadati si completano con successo, il file system può azzerare quel bit. Se tuttavia il bit dello stato rimane settato, entra in funzione un verificatore della coerenza.

Il verificatore della coerenza confronta i dati delle directory con i blocchi dati dei dischi, tentando di correggere ogni incoerenza. Gli algoritmi di allocazione e di gestione dello spazio determinano il genere di problemi che questo programma può riconoscere e con quanto successo riuscirà a risolverli. Per esempio, se si adotta uno schema di allocazione concatenata con un puntatore da ciascun blocco al successivo, si può ricostruire l'intero file e ricreare il corrispondente elemento nella directory analizzando i blocchi di dati. Al contrario, la perdita di un elemento di una directory in un sistema ad allocazione indirizzata potrebbe essere catastrofica, poiché ogni blocco di dati non contiene alcuna informazione sugli altri blocchi di dati. Per questo motivo, UNIX gestisce tramite cache gli elementi delle directory per le letture, mentre qualsiasi operazione di scrittura che produca l'allocazione di spazio, o altre modifiche dei metadati, è svolta in modo sincrono, prima della scrittura dei corrispondenti blocchi di dati. Naturalmente possono ancora insorgere dei problemi se una scrittura sincrona viene interrotta da una caduta del sistema.

Alcuni dispositivi di memoria NVM contengono una batteria o super condensatori ad alta capacità in grado di fornire energia durante una perdita di potenza o trasferire dati dai buffer di dispositivo ai supporti di memorizzazione per evitare la perdita di dati. Tuttavia anche queste precauzioni non evitano i danni di una caduta del sistema.

## File system con log delle modifiche

Gli algoritmi per il ripristino sono basati sulla registrazione (*log*) delle modifiche sono stati applicati al problema della verifica della coerenza, realizzando i file system orientati alle transazioni e basati sul log delle modifiche, noti anche come file system annotati.

Si osservi che l'approccio alla verifica della coerenza esaminato precedentemente permette in sostanza alle strutture di esibire incoerenze successivamente corrette nel ripristino. Questa strategia comporta tuttavia vari problemi. Per esempio, l'incoerenza potrebbe rivelarsi irreparabile: il verificatore della coerenza potrebbe non essere in grado di ripristinare le strutture, con una conseguente perdita di file o addirittura di intere directory. Ancora, il verificatore della coerenza potrebbe richiedere l'intervento umano per risolvere i conflitti, il che causa inconvenienti: in mancanza di assistenza da parte di qualcuno, il sistema potrebbe rimanere inutilizzabile finché una persona non gli indichi come procedere. Infine, il verificatore della coerenza sottrae risorse al sistema: per controllare terabyte di dati possono essere necessarie molte ore.

Fondamentalmente, tutte le modifiche dei metadati si registrano in modo sequenziale in un file di log. Ogni insieme di operazione che esegue uno specifico compito si chiama *transazione*. Una volta che le modifiche sono riportate nel file di log, le operazioni si considerano portate a termine con

successo (committed) e la chiamata di sistema può restituire il controllo al processo utente, permettendogli di proseguire la sua esecuzione. Nel frattempo, si applicano alle effettive strutture del file system le operazioni scritte nel log, e man mano che si eseguono si aggiorna un puntatore che indica quali azioni sono state completate e quali sono ancora incomplete. Quando un'intera transazione è stata completata, se ne rimuovono le annotazioni dal log, che è in realtà un buffer circolare. I *buffer circolari* scrivono fino alla fine dello spazio disponibile, e poi ricominciano dall'inizio, sovrascrivendo i vecchi contenuti. Naturalmente, si devono prendere delle misure per evitare che dati non ancora salvati siano sovrascritti. Il log si può mantenere in una sezione separata del file system, o anche in un disco separato. E' più efficiente, anche se è più complesso, averlo sotto testine di lettura e scrittura separate, poiché si riducono le situazioni di contesa della testina e i tempi di ricerca (*seek time*).

Se si verifica una caduta del sistema, nel log ci potranno essere zero o più transazioni. Le transazioni presenti non sono mai state ultimate nel file system, anche se il sistema operativo le definisce portate a termine con successo, e quindi si devono completare.

Le transazioni si possono eseguire a partire dalla posizione corrente del puntatore fino al completamento, in modo che le strutture del file system rimangano coerenti. L'unico problema che si può presentare è il caso in cui una transazione sia fallita (*aborted*), cioè non sia stata dichiarata terminata con successo prima della caduta del sistema. In questo caso, si devono annullare tutti i cambiamenti che erano stati applicati al file system della transazione, mantenendo anche in questo caso la coerenza del file system. Questo ripristino è tutto ciò che è necessario fare dopo una caduta del sistema, eliminando tutti i problemi della verifica della coerenza.

Un vantaggio indiretto dell'uso del logging degli aggiornamenti dei metadati dei dischi è che gli aggiornamenti sono molto più rapidi di quando si applicano direttamente alle strutture dati nei dischi. La ragione di questo miglioramento sta nel vantaggio, dal punto di vista delle prestazioni, dell'I/O ad accesso sequenziale rispetto a quello ad accesso diretto. Le onerose operazioni di scrittura sincrona ad accesso diretto sui metadati vengono sostituite con molte meno operazioni di scrittura sequenziali sincrone nell'area di log di un file system annotato. I cambiamenti determinati da quelle operazioni si riportano successivamente in modo asincrono nelle strutture appropriate nei dischi attraverso operazioni di scrittura ad accesso diretto. Il risultato complessivo è un significativo guadagno in termini di prestazioni per le operazioni orientate ai metadati, come la creazione e la cancellazione dei file su disco rigido.

#### **Appunti esterni:**

Caso della chiavetta staccata a volo o componente mancata: stato inconsistente del file system. Dei blocchi vengono markati come occupati ma non ci ho scritto niente. Servono delle utility per ripristinare il file system.

Windows: chkdsk (check disk), recupera dei file illeggibili. Unix: fsc

Queste utility vanno in ogni directory per controllare tutti file: è un processo inefficiente e lento.

FS basato sul log: Molto più efficiente mettere il FS nella condizione in cui i metadati sono al sicuro. I dati non scritti vengono persi per sempre, ma i metadati sono salvati come transazione. Sono appunto trascritti in un log.

Il disco di appoggio usato per la Memoria Virtuale inizialmente era un file grosso con un'estensione .swp (swap). È un sistema non efficiente: dovrei entrare nella directory e usare i metodi di accesso ai file. Questo file non sarà quindi ottimizzato. Dato che non dimensioni variabili, posso formattare un sistema partizionando lo swap: oltre al disco normale viene creata una partizione su disco usata unicamente per la memoria virtuale. Uso uno spazio doppio rispetto alla memoria disponibile per la partizione. E' usato in Linux.

## **Cos'è l'allocazione concatenata?**

L'allocazione concatenata risolve tutti i problemi dell'allocazione contigua. Con questo tipo di allocazione infatti, ogni file è composto da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Per creare nuovo file si crea semplicemente un nuovo elemento nella directory. Con l'allocazione concatenata, ogni elemento della directory ha un puntatore al primo blocco del file. Questo puntatore si inizializza a null (valore del puntatore di fine lista) a indicare un file vuoto; anche il campo della dimensione si imposta a zero. Un'operazione di scrittura nel file determina la ricerca di un blocco libero attraverso il sistema di gestione dello spazio libero, la scrittura in tale blocco, e la concatenazione di tale blocco alla fine del file. Un file può continuare a crescere finché sono disponibili blocchi liberi, di conseguenza non è mai necessario compattare lo spazio del disco. L'allocazione concatenata presenta comunque alcuni svantaggi. Il problema principale riguarda il fatto che può essere usata in modo efficiente solo per i file ad accesso sequenziale. Un altro svantaggio dell'allocazione concatenata riguarda lo spazio richiesto per i puntatori.

## **Cos'è la FAT? E' un tipo di allocazione? Vantaggi e svantaggi?**

Una variante importante del metodo di allocazione concatenata consiste nell'uso della tabella di allocazione dei file (file allocation table, FAT). Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascun volume; la FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; si usa essenzialmente come una lista concatenata. L'elemento di directory contiene il numero del primo blocco del file.

L'elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero del blocco successivo del file. I blocchi inutilizzati sono indicati nella tabella da un valore 0. L'allocazione di un nuovo blocco a un file implica semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine del file precedente con l'indirizzo del nuovo blocco.

## **Cos'è un file system con journaling?**

È una tecnica utilizzata da molti file system moderni per preservare l'integrità dei dati da eventuali cadute di tensione. Derivata dal mondo dei database, il journaling si basa infatti sul concetto di transazione dove ogni scrittura su disco è interpretata dal file system come tale.

## **Com'è fatta una FAT?**

Una File Allocation Table (FAT) è un sistema di gestione dei file utilizzato in molte partizioni di archiviazione, come floppy disk, schede di memoria flash e dischi rigidi. La FAT mantiene traccia dell'allocazione dei file sul disco e consente a un sistema operativo di accedere ai file.

La FAT è organizzata come una tabella di dimensioni fisse che contiene informazioni sulle posizioni dei file sul disco. Ogni entrata della tabella rappresenta una unità di archiviazione sul disco, nota come cluster. Ogni cluster può contenere un singolo file o una parte di un file.

La FAT è un componente importante per la gestione dei file, poiché fornisce informazioni sulle posizioni dei file sul disco e consente a un sistema operativo di accedere ai file in modo efficiente. Inoltre, la FAT è spesso utilizzata per la recupero dei file in caso di errori o danneggiamenti del disco, poiché mantiene informazioni complete sulle posizioni dei file sul disco.