

UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso 2° anno - 15 CFU

Architettura e sistemi operativi

Professori:

Prof. Gabriele Mencagli
Prof. Massimo Torquati

Autori:

Matteo Giuntoni
Filippo Ghirardini

Anno Accademico 2023/2024

Contents

1 Elementi basilari	5
1.1 Astrazione	5
1.1.1 Tre Y	5
1.1.2 Astrazione digitale	5
1.2 Sistemi numerici	5
1.2.1 Unità di misura	6
1.2.2 Somma binaria	6
1.2.3 Segno	6
1.2.4 Confronto	6
1.3 Porte logiche	7
1.3.1 NOT	7
1.3.2 Buffer	7
1.3.3 AND	7
1.3.4 OR	7
1.3.5 XOR	7
2 Progetto di reti logiche combinatorie	8
2.1 Mappe di Karnaugh	8
2.2 Blocchi costruttivi	9
2.2.1 Multiplexer	9
2.2.2 Decoder	10
2.3 Temporizzazioni	10
2.3.1 Ritardi	10
2.3.2 Alee	11
3 Progetto di reti logiche sequenziali	12
3.1 Latch e Flip-Flop	12
3.2 Macchine a stati finiti	12
4 Assembler	13
4.1 Numeri frazionari	13
4.1.1 Virgola fissa	13
4.1.2 Virgola mobile	13
4.2 Mappa di memoria	14
4.2.1 Testo	14
4.2.2 Dati globali	14
4.2.3 Dati dinamici	14
4.2.4 Eccezioni, OS, I/O	14
4.3 Codifica	14
4.3.1 Elaborazione dati	14
4.3.2 Accesso alla memoria	15
4.3.3 Salto	16
4.3.4 Literal	16
4.3.5 NOP	16
5 Microarchitetture	17
5.1 Premesse	17
5.1.1 Stato architetturale	17
5.1.2 Progettazione	17
5.1.3 Tipologie	18
5.1.4 Prestazioni	18
5.2 Ciclo singolo	18
5.3 Multiciclo	19
5.4 Pipeline	19

6 Memory Hierarchy	20
6.1 Memory technologies	20
6.2 Cost vs Capacity vs Access Time	20
6.3 Von Neumann architecture	21
6.4 Terminologia	21
6.4.1 AMAT	21
6.5 The locality principle	22
6.5.1 Locality characterization	22
6.6 Traferimento dati	23
7 Input output	24
7.1 Legge di Amdahl	24
7.2 Connessione bus	25
7.2.1 Bus design	26
7.3 Gestione dell'I/O	27
7.3.1 Invio dei comandi ai dispositivi	27
7.3.2 Ottenerlo lo stato dell'I/O	27
7.4 Data transfers (DMA)	28
8 Dischi rigidi	31
8.1 Dischi magnetici	31
8.1.1 Settore	31
8.1.2 Disk performance	32
8.2 Dishi SSD	32
8.2.1 Flash translation layer	32
8.2.2 File system flash	33
9 Cache	34
9.1 Gestione del movimento dei dati	34
9.2 Utilizzo	34
9.3 Performance	34
9.4 Design	35
9.4.1 Direct mapped	35
9.4.2 Associative	38
9.4.3 Comparazione	38
9.5 Cache miss	38
9.5.1 Multi-level cache	39
9.6 Gestione delle scritture	40
9.6.1 Write-Through	40
9.6.2 Write-Back	40
9.6.3 Ottimizzare le scritture	41
9.6.4 Cache replacement policy	41
9.7 Designing the Memory System	42
9.8 Problemi cache	43
9.8.1 Problemi di corerenza	44
9.9 Software	44
9.9.1 Ottimizzazione	44
10 Address translation	45
10.1 Virtual base and Bounds	45
10.1.1 Variable partitions	46
10.1.2 Frammentazione	46
10.1.3 Allocazione	46
10.1.4 Conclusione	46
10.2 Segmentazione	46

11 Kernel	47
11.1 Booting	47
11.2 Processi	47
11.2.1 Fork	47
11.2.2 Exec	48
11.2.3 Terminazione	48
11.2.4 Shell	49
11.2.5 I/O Unix	49
12 Concorrenza	50
12.1 Inter-Process Communication	50
12.2 Thread	50
12.2.1 Protezione	50
12.2.2 Astrazione	51
12.2.3 Implementazione	51
12.2.4 API	52
12.2.5 Ciclo di vita	52
12.2.6 User-level thread	52
12.2.7 Kernel-level thread	53
12.2.8 Switch	53
12.3 Cooperazione	54
12.4 Sincronizzazione	55
12.4.1 Lock	55
12.4.2 Condition variables	56
12.4.3 Semantica Hoare	57
12.4.4 Semantica Mesa	57
12.4.5 Banker's Algorithm	58
12.4.6 Esempi	59

Architettura e Sistemi Operativi

Realizzato da: Giuntoni Matteo e Ghirardini Filippo

A.A. 2023-2024

1 Elementi basilari

1.1 Astrazione

Definizione 1.1.1 (Astrazione). *La tecnica che consiste nel nascondere dettagli quando questi non sono importanti. Permette di dividere un sistema in più livelli.*

1.1.1 Tre Y

Per gestire la complessità i progettisti utilizzano tre principi:

- **Gerarchia:** dividere un sistema in moduli e successivamente dividere questi ultimi finché i pezzi non sono facili da comprendere
- **Modularità:** i moduli hanno funzioni e interfacce ben definite in modo da connettersi semplicemente e senza effetti indesiderati
- **Regolarità:** punta all'uniformità dei moduli, riutilizzando quelli più comuni e riducendo il carico di progettazione

1.1.2 Astrazione digitale

I sistemi digitali rappresentano le informazioni con variabili **discrete** che possono avere un numero finito di valori.

Definizione 1.1.2 (Quantità di informazione). *La quantità di informazione D associata a una variabile a valori discreti con N stati distinti è misurata in termini di bit come:*

$$D = \log_2 N \text{ bit} \quad (1)$$

Nell'ambito dei calcolatori il numero di valori che può assumere una variabile è due: 0 o 1. Queste variabili si basano sulla **logica Booleana**, dove 0 rappresenta *FALSO* e 1 rappresenta *VERO*.

1.2 Sistemi numerici

Il sistema da noi utilizzato è quello **posizionale**, dove ogni possibile valore assume un peso diverso in base alla posizione.

Proposizione 1.2.1 (Conversione da base X a base 10). *Per convertire da una base generica (ad esempio 2 o 16) è sufficiente moltiplicare ogni numero per il valore della base elevato alla posizione del numero (partendo da destra).*

$$2ED_{16} = 2 \cdot 16^2 + E \cdot 16^1 + D \cdot 16^0 = 749_{10}$$

Proposizione 1.2.2 (Conversione da base 10 a base X). *La conversione da base 10 ad una base generica consiste nella scomposizione in fattori del numero originale, dalla quale si mantengono i resti:*

$$\begin{aligned} 333_{10} &= \\ 333 \% 16 &= 13 = D_{16} & \frac{333}{16} &= 20 \\ 20 \% 16 &= 4 = 4_{16} & \frac{20}{16} &= 1 \\ 1 \% 16 &= 1 = 1_{16} & \frac{1}{16} &= 0 \\ && &= 14D_{16} \end{aligned}$$

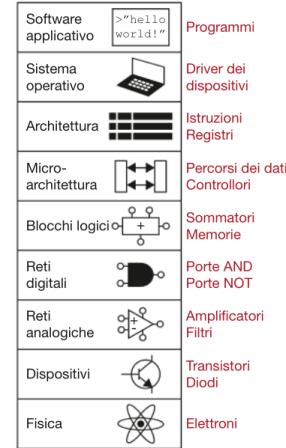


Figure 1: Astrazione di un computer

1.2.1 Unità di misura

Definizione 1.2.1 (Byte). *Insieme di 8 bit. Rappresenta $2^8 = 256$ possibilità.*

Definizione 1.2.2 (Nibble). *Un gruppo di 4 bit o mezzo byte. Rappresenta $2^4 = 16$ possibilità.*

Definizione 1.2.3 (Word). *Sono gruppi di bit utilizzati dai microprocessori la cui grandezza dipende dall'architettura. Ad oggi generalmente sono di 64 bit.*

Osservazione 1.2.1. All'interno di un gruppo di bit, il bit che si trova nella colonna di peso 1 viene chiamato bit **meno significativo** (*lsb*, Least Significant Bit) e il bit che si trova all'estremità opposta viene chiamato bit **più significativo** (*msb*, Most Significant Bit). Lo stesso discorso vale per i *byte* (*LSB* e *MSB*).

1.2.2 Somma binaria

La somma binaria avviene esattamente come quella decimale: nel momento in cui due cifre sommate superano il valore massimo 1, viene scritto 0 ed effettuato un riporto.

Osservazione 1.2.2. I computer usano sempre un numero fisso di cifre. Di conseguenza se una somma è troppo grande per essere rappresentata, avviene un **overflow**.

1.2.3 Segno

Per rappresentare il segno di un numero esistono due tecniche:

- **Modulo e segno:** utilizza il *msb* per esprimere il segno e i restanti per il valore del modulo del numero

$$5_{10} = 0101_2 \quad -5_{10} = 1101_2$$

Con questo metodo la somma binaria vista in precedente non funziona e abbiamo inoltre due rappresentazioni per lo stesso numero: -0 e 0.

- **Complemento a due:** risolve i problemi dell'altro metodo. I numeri positivi sono rappresentati in maniera classica, con 0 come *msb*. Quelli negativi si calcolano invertendo bit e aggiungendo 1.

$$+2_{10} = 0010_2 \quad -2_{10} \rightarrow 0010_2 \rightarrow 1101_2 \rightarrow 1101_2 + 1 = 1110_2$$

Osservazione 1.2.3. Sommare due numeri in complemento a due di segno opposto non causa mai **overflow**.

Proposizione 1.2.3 (Estensione del segno). *Quando un numero in complemento a due viene esteso a un numero maggiore di bit, il bit che dà il segno deve essere copiato in tutte le posizioni più significative.*

$$-3_{10} = 1011_2 \longrightarrow 1111101_2$$

1.2.4 Confronto

Vediamo la **variabilità** delle rappresentazioni viste fin'ora:

Sistema	Range
Senza segno	$[0, 2^N - 1]$
Modulo e segno	$[-2^{N-1}, 2^{N-1} - 1]$
Complemento a due	$[-2^{N-1}, 2^{N-1} - 1]$

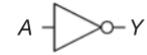
1.3 Porte logiche

Le porte logiche (logic gates) sono circuiti digitali che utilizzano uno o più ingressi binari per produrre un'uscita binaria. La relazione tra ingressi e uscite può essere descritta con una **tavola delle verità** o con una **espressione booleana**.

1.3.1 NOT

Ha un ingresso A e un'uscita Y : è l'esatto contrario del suo ingresso.

$$Y = \bar{A} = \neg A$$



1.3.2 Buffer

Ha un ingresso A e un'uscita Y e riproduce il valore in ingresso. È utile in particolare dal punto di vista elettrico per erogare grandi quantità di corrente o trasmettere il valore a tante porte logiche diverse.

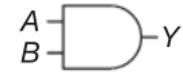
$$Y = A$$



1.3.3 AND

Ha due ingressi, A e B , e restituisce *VERO* solamente se entrambi sono *VERO*, altrimenti *FALSO*.

$$Y = AB = A \wedge B$$



Esiste anche la versione ad N ingressi, dove se tutti sono *VERO* allora il risultato è *VERO*, altrimenti è *FALSO*.

1.3.4 OR

Ha due ingressi, A e B , e restituisce *VERO* se almeno uno dei due è *VERO*, altrimenti *FALSO*.

$$Y = A + B = A \vee B$$



Esiste anche la versione ad N ingressi, dove se almeno un ingresso è *VERO* allora il risultato è *VERO*, altrimenti è *FALSO*.

1.3.5 XOR

Ha due ingressi, A e B , e restituisce *VERO* se almeno uno dei due è *VERO* ma non entrambi, altrimenti *FALSO*.

$$Y = A \oplus B$$



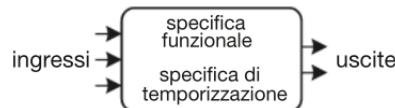
Osservazione 1.3.1. Qualunque porta può essere seguita da un pallino per invertire le sue operazioni, ad esempio **NAND** e **NOR**.

2 Progetto di reti logiche combinatorie

Definizione 2.0.1 (Circuito). *Un circuito è una rete elettrica che elabora variabili a valori discreti.*

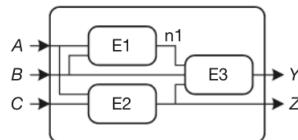
Un circuito contiene:

- Uno o più **ingressi** a valori discreti
- Una o più **uscite** a valori discreti
- Una **specifiche funzionale** che descrive la relazione tra ingressi e uscite
- Una **specifiche temporizzazione** che descrive il ritardo tra il cambio degli ingressi e la risposta delle uscite



All'interno è composta da **elementi**, ovvero a loro volta reti logiche, e **nodi**, ovvero contatti elettrici la cui tensione trasmette un valore discreto. Questi si dividono in:

- **Ingressi**: ricevono valori dal mondo esterno
- **Uscite**: emettono valori all'esterno
- **Interni**: tutti gli altri



Le reti digitali vengono divise in:

- **Combinatorie**: le uscite dipendono esclusivamente dai valori presenti in ingresso
- **Sequenziali**: le uscite dipendono sia dai valori di ingresso sia da quelli precedenti. è quindi dotata di **memoria**

2.1 Mappe di Karnaugh

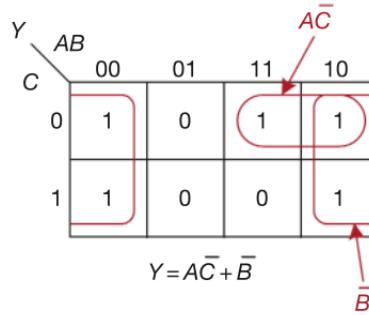
Per semplificare le espressioni booleane usate per rappresentare le reti, è possibile usare le mappe di Karnaugh. A partire dalla tabella di verità, si costruisce un'altra tabella che ha come righe e colonne tutte le possibili combinazioni dei letterali ordinate tramite il **codice Gray**, ovvero ogni combinazione adiacente deve differire solo per un letterale.

		AB	00	01	11	10	
		C	0	1	0	1	1
		1	1	0	0	1	
Y							

Si costruiscono poi dei *cerchi* che devono:

- Essere in numero minore possibile
- Tutti i riquadri devono contenere 1
- Ogni cerchio deve includere un numero di riquadri che sia una potenza di 2
- Ogni cerchio deve essere il più grande possibile
- Si possono disegnare cerchi che vanno dalla fine della mappa all'inizio (pac-man effect)
- Un riquadro può essere cerchiato più di una volta

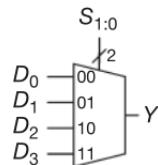
Una volta costruiti i cerchi, per ognuno di essi creiamo un termine della nostra espressione semplificata: ogni termine avrà i letterali che non cambiano tra i riquadri cerchiati e non avrà quelli che cambiano. Questo perché se un termine cambia ma il risultato rimane 1, vuol dire che non è utile (**indifferenza**).



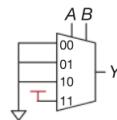
2.2 Blocchi costruttivi

2.2.1 Multiplexer

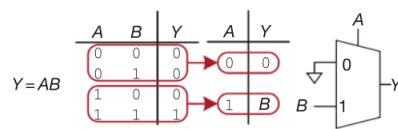
Permettono di scegliere un'uscita a partire da un certo numero di ingressi possibili basandosi sul valore di un **segnale di controllo**.



Possono essere utilizzati come **lookup table** per eseguire funzioni logiche.

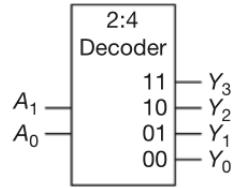


È possibile ridurre il numero di ingressi da 2^N a 2^{N-1} , fornendo uno dei letterali in ingresso come segnale di controllo.

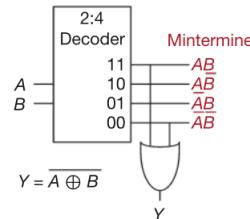


2.2.2 Decoder

Ha N ingressi e 2^N uscite e attiva una di queste in base alla combinazione dei valori in ingresso.



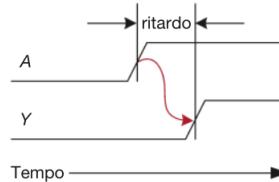
Possono essere utilizzati assieme a delle porte *OR* per costruire funzioni logiche. In particolare una funzione ad N ingressi e M uscite nella tabella di verità, si può rappresentare con un decoder $N : 2^N$ e con una porta *OR* a M ingressi.



2.3 Temporizzazioni

Uno dei problemi più importanti delle reti è come fare in modo che queste funzioni velocemente. Il **diagramma temporale** rappresenta la **risposta transitoria** della rete al cambiare di un ingresso. Ci sono due sezioni:

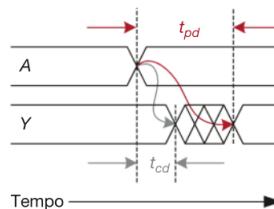
- Fronte di **salita**: passaggio da *BASSO* ad *ALTO*
- Fronte di **discesa**: passaggio da *ALTO* a *BASSO*



2.3.1 Ritardi

Esistono due tipi di ritardi:

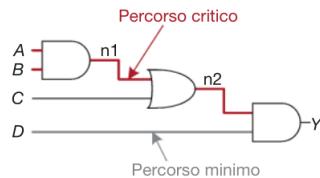
- **Propagazione**, t_{pd} : il tempo massimo che trascorre dal cambiamento di un ingresso al momento in cui l'uscita raggiunge il valore finale
- **Contaminazione**, t_{cd} : il tempo minimo che trascorre dal momento in cui cambia un ingresso al momento in cui una qualsiasi uscita comincia il processo di adattamento del suo valore



Note 2.3.1.1. Le cause del ritardo includono il tempo richiesto per caricare in una rete e la velocità della luce. Inoltre, i due tipi di ritardi possono essere diversi per:

- Diversi ritardi tra salita e discesa
- Ingressi e uscite multipli, alcuni più veloci di altri
- Reti che rallentano al surriscaldamento e si velocizzano al raffreddamento

Il valore dei ritardi è determinato anche dal **percorso critico**, ovvero il percorso seguito dal segnale tra l'ingresso e l'uscita. Il **percorso minimo** è invece quello più breve possibile.



In una rete il ritardo di propagazione è uguale alla somma dei singoli ritardi di ogni elemento del percorso critico, mentre il ritardo di contaminazione è la somma dei ritardi di contaminazione di ogni elemento del percorso minimo.

$$t_{pd} = 2t_{pd_AND} + t_{pd_OR} \quad (2)$$

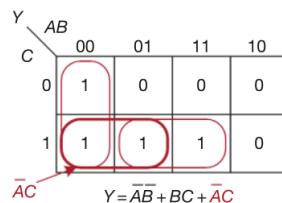
$$t_{cd} = t_{cd_AND} \quad (3)$$

2.3.2 Alee

È possibile che un cambiamento singolo in ingresso possa causare più cambiamenti in uscita indesiderati: le **alee**.

Non rappresentano un problema se si aspetta che il ritardo di *propagazione* si sia esaurito prima di guardare il valore di uscita poiché questa prima o poi torna sul risultato corretto.

Si può evitare aggiungendo un'altra porta facendo riferimento alla mappa di Karnaugh. È sufficiente aggiungere un cerchio quando il cambio di una variabile "attraversa" due cerchi adiacenti (con un costo maggiore a livello HW).



3 Progetto di reti logiche sequenziali

Le reti logiche sequenziali, avendo una memoria, riassume gli ingressi precedenti in **stati** del sistema. Questi sono composti da un insieme di bit detto **variabili di stato**.

3.1 Latch e Flip-Flop

3.2 Macchine a stati finiti

Le reti sequenziali **sincrone** possono essere rappresentate tramite **Finite State Machine**. Una FSM ha M ingressi, N uscite e k bit di stato con 2^k possibili stati diversi. Riceve un segnale di *clock* e a volte di *reset*. Si compone di:

- Logica di **stato prossimo**
- Logica di **uscita**
- **Registro di stato**

4 Assembler

4.1 Numeri frazionari

Supponiamo di avere il numero 2.375_{10} . Per convertirlo in base 2, prima convertiamo la parte intera ($2 = 10$) e poi la parte frazionaria:

$$2.375_{10} = 10.011$$

$$0.375 \cdot 2 = 0.75$$

$$0.75 \cdot 2 = 1.5$$

$$0.5 \cdot 2 = 1$$

Per fare la conversione inversa:

$$10.011 = 1 * 2^1 + 1 * 2^{-2} + 1 * 2^{-3} = 2 + \frac{1}{4} + \frac{1}{8} = 2.375$$

4.1.1 Virgola fissa

4.1.2 Virgola mobile

Lo standard IEEE 754 divide i numeri in due categorie:

- Singola precisione
- Doppia precisione

4.2 Mappa di memoria

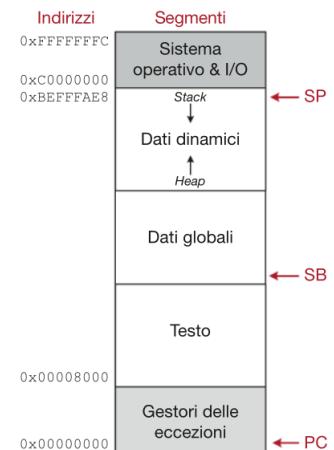
ARM utilizza indirizzi a 32 bit e di conseguenza ha 2^{32} byte di spazio di indirizzamento. Gli indirizzi di parola sono a multipli di 4. Lo spazio viene diviso in cinque segmenti.

4.2.1 Testo

Contiene il programma il linguaggio macchina. È a **sola lettura** e può contenere anche delle costanti dette *literal*.

4.2.2 Dati globali

Memorizza le variabili che sono accessibili da tutto il programma e che sono in **lettura/scrittura**. Si accede a queste variabili a partire da un indirizzo base statico che punta all'inizio del segmento, per convenzione *R9*, chiamato **SB**.



4.2.3 Dati dinamici

Contiene **stack** e **heap**. I dati qui vengono allocati e deallocati dinamicamente durante l'esecuzione del programma.

- **Stack:** *SP* viene inizializzato con l'indirizzo in cima al segmento in quanto poi cresce verso il basso (LIFO). Contiene dati temporanei e variabili locali che sono troppo grandi per stare nei registri. Viene anche usato per salvare e ripristinare i registri.
- **Heap:** memorizza i dati allocati dal programma durante le esecuzioni. Qui gli elementi possono essere salvati ed eliminati in qualsiasi ordine. Cresce verso l'alto a partire dall'inizio del segmento

Per evitare che *stack* e *heap* si sovrappongano con conseguente perdita di dati, il programmatore deve assicurarsi di lanciare errori di tipo *out-of-memory*.

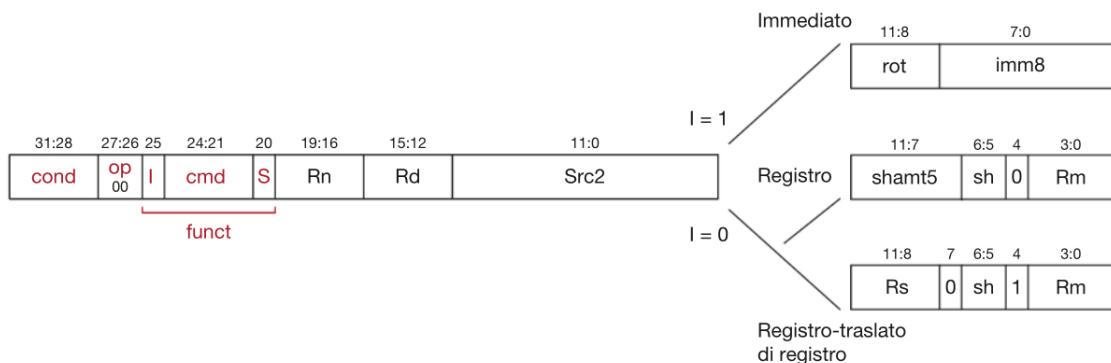
4.2.4 Eccezioni, OS, I/O

Contiene il vettore delle eccezioni e i gestori delle eccezioni. Oltre che il sistema operativo e l'ingresso/uscita mappato in memoria.

4.3 Codifica

Le istruzioni ARM sono codificate in 32 bit che ha un formato variabile in base al tipo di istruzione da eseguire.

4.3.1 Elaborazione dati

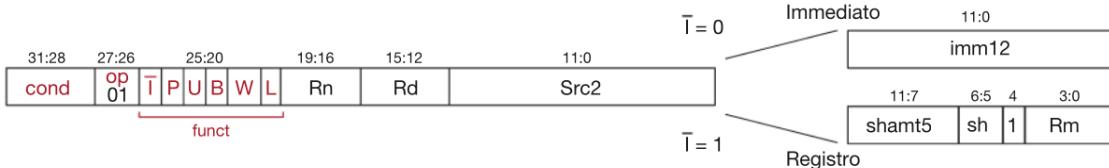


Osservazione 4.3.1. Le istruzioni di elaborazione dati hanno una rappresentazione basata su un **immediato imm8** ruotato a destra di $2 \cdot \text{rot}$. Questo permette di codificare in maniera comoda molte costanti utili, come piccole potenze di due, in pochi bit.

Vediamo lo scopo di ogni campo:

- **cond:** *condition*, identifica l'eventuale condizione da aggiungere all'istruzione. Se non ve ne sono $\text{cond} = 1110_2$
- **op:** *OPeration CODE*, che nel caso delle operazioni di elaborazione dati $\text{op} = 00_2$
- **funct:** *function*, identifica l'istruzione
 - **I:** vale 1 quando Src2 è un immediato
 - **cmd:** valore specifico per ogni istruzione
 - **S:** vale 1 quando l'istruzione modifica le flag di condizione
- **Rn:** registro del primo operando sorgente
- **Rd:** registro destinazione
- **Src2:** registro del secondo operando sorgente
 - *Immediato:*
 - * **rot:** vedi 4.3.1
 - * **imm8:** vedi 4.3.1
 - *Registro traslato di una costante:*
 - * **shamt5:** immediato da utilizzare per la traslazione
 - * **sh:** indica il tipo di traslazione da effettuare
 - * **Rm:** registro di partenza
 - *Registro traslato del contenuto di un altro registro:*
 - * **Rs:** registro che contiene il valore per effettuare la traslazione
 - * **sh:** vedi prima
 - * **Rm:** vedi prima

4.3.2 Accesso alla memoria

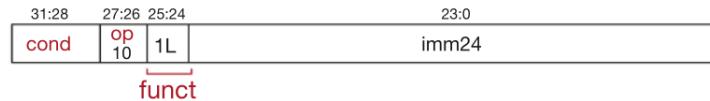


- **op:** nel caso delle operazioni di accesso alla memoria dati $\text{op} = 01_2$
- **funct:** *function*, identifica l'istruzione
 - \bar{I} : indica se lo spiazzamento è un immediato o un registro
 - **U:** indica se lo spiazzamento deve essere sommato o sottratto
 - **L e B:** indicano il tipo di accesso alla memoria
 - **P e W:** indicano in che modo gestire l'indice
- **Src2:** definisce l'offset
 - *Immediato:* **imm12**, unsigned a 12 bit
 - *Registro traslato di una costante*

L	B	Istruzione
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

P	W	Modi di gestione
0	0	Post-indice
0	1	Non supportato
1	0	Spiazzamento
1	1	Pre-indice

4.3.3 Salto



- **op**: nel caso delle operazioni di accesso alla memoria dati $op = 10_2$
- **funct**: due bit, di cui il più significativo è sempre 1 e l'altro è **L**, ovvero il tipo di salto:
 - 1 per **BL**
 - 0 per **B**
 - **U**: indica se lo spiazzamento deve essere sommato o sottratto
- **imm24**: indirizzo di destinazione del salto scritto come differenza con $PC + 8$

4.3.4 Literal

Quando è necessario caricare dei *literal* a 32 bit, non si può usare *MOV* dato che prende una sorgente a 12 bit. Si usa quindi:

```
LDR Rd, =literal
LDR Rd, =label
```

che può prendere o una costante (literal) o un indirizzo ad una zona di memoria contenente il literal (label). In entrambi i casi il valore viene salvato nel **serbatoio dei literal** nel segmento *testo* della memoria. Questo deve stare a meno di 4096 byte di distanza dall'istruzione LDR in modo che il caricamento si possa eseguire correttamente. Di conseguenza il programma deve evitare il serbatoio e non considerarlo come istruzioni.

4.3.5 NOP

Istruzione che non fa nulla, tradotta come:

```
MOV R0, R0
```

Può servire per introdurre ritardi e allineare istruzioni.

5 Microarchitettura

Una microarchitettura consiste nella specifica combinazione di *registri*, *ALU*, *macchine a stati finiti*, *memorie* e altri blocchi logici.

5.1 Premesse

5.1.1 Stato architetturale

Lo stato architetturale definisce l'architettura assieme al set di istruzioni. Nel caso del processore ARM, è definito da 16 registri a 32bit e da un registro di stato. Ogni istruzione quindi produrrà un nuovo stato architetturale. Noi vedremo le seguenti:

- **Elaborazione dati:** *ADD*, *SUB*, *AND* e *ORR* con indirizzamento a *registro* e *immediato* senza traslazioni
- **Accesso alla memoria:** *LDR* e *STR* con spiazzamento immediato positivo
- **Salto:** *B*

5.1.2 Progettazione

Dividiamo la microarchitettura in due parti:

- **Percorso dati:** opera su parole di dati ed è costituito da memorie, registri, ALI e multiplexer. Nel nostro caso sarà a 32bit
- **Unità di controllo:** riceve l'istruzione corrente dal percorso dati e gli comunica come eseguirla attivando i giusti ingressi di selezione, le abilitazioni dei registri e i segnali di lettura e scrittura in memoria

L'hardware necessario per gli elementi di stato è il seguente:



- **Program counter:** concettualmente è parte del banco dei registri, ma dato che viene letto e scritto ad ogni ciclo indipendentemente dall'operazione, conviene realizzarlo separatamente.
 - *PC*: istruzione corrente
 - *PC'*: istruzione successiva
- **Memoria istruzioni:**
 - *A*: indirizzo di istruzione a 32bit
 - *RD*: l'istruzione
- **Banco di registri:** contiene 15 elementi da 32bit
 - Due porte di lettura *A1* e *A2* che ricevono indirizzi a 4 bit per specificare il registro il cui contenuto viene poi emesso su *RD1* e *RD2* a 32 bit

- Una porta di **scrittura** *A3* che riceve l'indirizzo del registro a 4 bit e *WD3* che riceve il dato a 32 bit ed è abilitata da *WE3*
- **R15**, ovvero il *PC*, la quale lettura restituisce *PC + 8*

- **Memoria dati:**

- Singola porta di lettura e scrittura *A*, abilitata da *WE*
- *WD*: eventuale dato da scrivere
- *RD*: se non c'è l'abilitazione alla scrittura, restituisce il dato della lettura

5.1.3 Tipologie

Esistono tre tipi di microarchitetture ARM:

- **Ciclo singolo:** esegue una istruzione in un ciclo. Di conseguenza non ha bisogno dello stato architettonico ma il tempo di ciclo è deciso dall'istruzione più lenta
- **Multiciclo:** esegue le istruzioni in sequenze di cicli brevi. Quelle semplici sono eseguite in meno cicli di quelle complesse. Riduce il costo HW riutilizzando componenti come il sommatore
- **Pipeline:** può eseguire più istruzioni contemporaneamente ma necessita di più logica per gestire le dipendenze tra le istruzioni

5.1.4 Prestazioni

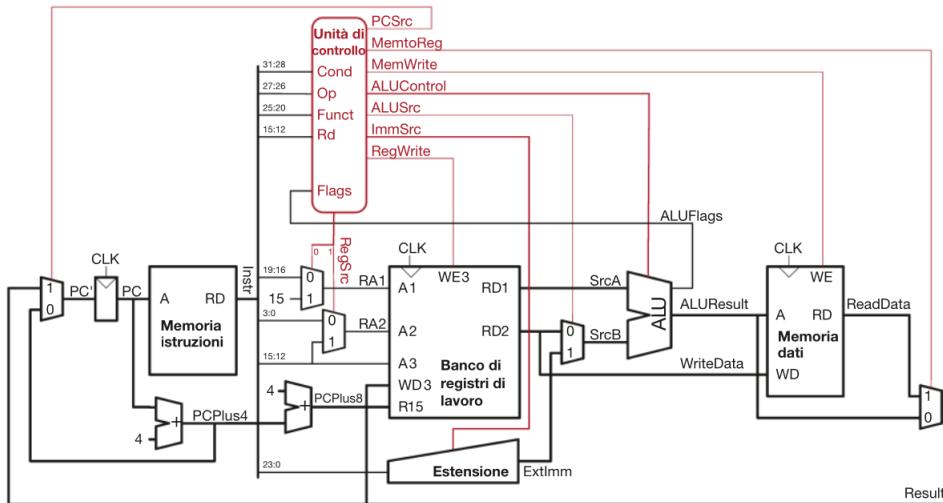
L'analisi consiste nel misurare il tempo di esecuzione di un dato programma su architetture diverse.

$$t_{esecuzione} = (n_{istruzioni}) \left(\frac{cicli}{istruzione} \right) \left(\frac{secondi}{ciclo} \right) \quad (4)$$

Analizziamo le singole componenti:

- **Numero di istruzioni:** dipende dall'architettura presa in considerazione e dalla bravura del programmatore
- **Cicli per istruzione (CPI):** il numero di cicli di clock richiesti in media per eseguire un'istruzione. È il reciproco della potenza di elaborazione (**IPC**). Dipende dalla microarchitettura.
- **Secondi per ciclo, *T_c*:** è il periodo del clock. È determinato dal percorso critico attraverso i circuiti del processore. Dipende dal progetto delle reti logiche e dei circuiti (e.g. sommatore ad anticipazione di riporto più veloce di quello a onda di riporto)

5.2 Ciclo singolo



5.3 Multiciclo

5.4 Pipeline

6 Memory Hierarchy

I principi fondamentali che andremo a vedere sono legati alle tecnologie con cui vengono costruite le memorie, la gerarchia delle varie tipologie di memorie, le memorie caches ed in fine come andare a misurare e migliorare le performance delle caches.

6.1 Memory technologies

Partiamo dicendo che esistono varie tipologie di memorie, che possono essere distinte in primo luogo in **memorie volatili**, e **memorie non volatili**. Quelle volatili sono:

- Latches, flip-flops, register files (o semplici registri).
- SRAM (Static Random-Access Memory).
- DRAM (Dynamic Random-Access Memory).

Fra le memorie non volatili invece ci sono:

- ROM
- NVRAM ¹
- Flash memory
- Magnetic disks

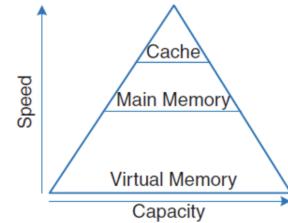


Figure 2: Gerarchia memoria

6.2 Cost vs Capacity vs Access Time

Un altro confronto interessante da fare fra le memorie è relativamente ai costi, le capacità ed il tempo di accesso.

Memoria	Access time (ns)	Bandwidth (GB/s)	Price(\$/GB)	Usage
<i>SRAM</i>	0.5 – 1	25+	5000	Register and caches
<i>DRAM</i>	10 – 50	10	7	RAM
<i>Flash</i>	20.000	0.5	0.40	SSD disks
<i>Magnetic</i>	5.000.000	0.75	0.05	HDD Disk

Da questa classificazione possiamo trarre alcune regole generali:

- Le memorie di grandi dimensioni sono solitamente lente e economiche.
- Le memorie di piccole dimensioni sono più veloci ma anche più costose.

Da qui possiamo capire che nella selezione della memoria va trovato un compromesso fra i parametri visti precedentemente per andare ad avere memorie sufficientemente grandi per contenere i dati richiesti ma allo stesso tempo sufficientemente veloci per evitare il **von Neumann Bottleneck**.

¹Memoria non volatile nel formato di un banco DRAM che può fornire accesso tramite byte-address.

6.3 Von Neumann architecture

Le performance dei computer sono limitate nella velocità della CPU dal trasferimento di dati fra le memorie esterne dall'unità di calcolo.

Per mitigare questo problema inseriamo memorie più piccole e veloci vicino al processore, mentre più ci si allontana più si avranno memorie grosse e lente. In questo modo facciamo lavorare il processore alla velocità della memoria più vicina.

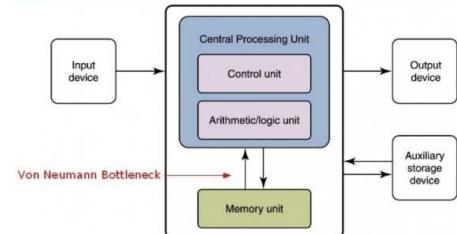


Figure 3: Von-Neumann

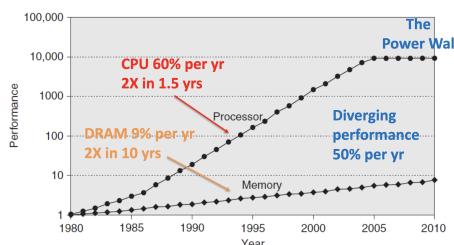


Figure 4: Von-Neumann Bottleneck

L'obiettivo è quindi di fornire l'**illusione** di avere una memoria grande quanto la memoria più lontana e veloce quanto la memoria più vicina. Per farlo facciamo risiedere i dati inizialmente nel livello più lontano e più capiente. Per far accedere il processore bisognerà spostare i dati tra i vari livelli di gerarchia.

Questo metodo presenta alcune problematiche: serve un meccanismo che, se un dato è già presente un certo livello, determini quello giusto e un meccanismo che permetta di rimpiazzare certi dati per poter fare spazio.

6.4 Terminologia

Introduciamo ora un po' di terminologia che ci servirà successivamente.

Definizione 6.4.1 (Hit e Miss). *Se i dati richiesti dal processore compaiono in qualche blocco nel livello di memoria più vicino, si parla di **hit**. In caso contrario, si parla di **miss** e si accede al livello di memoria successivo per recuperare il blocco contenente i dati richiesti.*

Definizione 6.4.2 (Hit rate). *L'**hit rate** (frequenza di successi) è la frazione di accessi alla memoria rilevati nel livello superiore (ovvero, più vicino alla CPU), utilizzata come misura delle prestazioni della gerarchia.*

Definizione 6.4.3 (Miss rate). *Il **miss rate** è la frazione di accessi alla memoria non trovati nel livello superiore.*

Definizione 6.4.4 (Miss penalty). *La **miss penalty** è il tempo necessario per sostituire un blocco nel livello n con il blocco corrispondente dal livello $n - 1$.*

Definizione 6.4.5 (Miss time). *Il **miss time** è il tempo per ottenere l'elemento in caso di tempo di miss.*

$$\text{miss time} = \text{miss penalty} + \text{hit time}$$

Il **miss rate** e l'**hit rate** si calcolano con le seguenti formule:

$$MR = \frac{\text{Number of misses}}{\text{Number of total memory access}} = 1 - HR \quad (5)$$

$$MR = \frac{\text{Number of hits}}{\text{Number of total memory access}} = 1 - MR \quad (6)$$

6.4.1 AMAT

Definiamo l'Average Memory Access Time:

$$AMAT = t_{M0} + MR_{M0} * (t_{M1} + MR_{M1} * (t_{M2} + MR_{M2} * (t_{M3} + \dots))) \quad (7)$$

t_{M0} = hit time, MR_{M0} = miss rate, $(t_{M1} + MR_{M1} * (t_{M2} + MR_{M2} * (t_{M3} + \dots)))$ = miss penalty.

Osservazione 6.4.1. Se l'hit rate è abbastanza alto, la gerarchia della memoria ha un tempo di accesso effettivo vicino a quello del livello più alto (e più veloce) e una dimensione uguale a quella del livello più basso (e più grande).

Esempio 6.4.1. Consideriamo una memoria con una gerarchia su 3 livelli con i seguenti valori:

Livello	Miss rate	Hit time
L1	5%	t_{L1}
L2	2%	t_{L2}
L3	100%	t_{L3}

Quindi il valore AMAT sarà:

$$AMAT = t_{L1} + 0.05 * (t_{L2} + 0.02 * t_{L3}) = t_{L1} + 0.05 * t_{L2} + 0.001 * t_{L3}$$

6.5 The locality principle

Il principio di località di riferimento (o locality principle) si riferisce al fenomeno per il quale un programma tende ad accedere alla stessa locazione di memoria per un determinato periodo.

Possiamo osservare che, se il programma fa riferimento ad una locazione di memoria allora:

- la stessa locazione di memoria verrà riutilizzata a breve con alta probabilità
- gli elementi "vicini" alla posizione di memoria appena raggiunta saranno presto referenziati con un'alta probabilità

Il principio di località è la forza trainante che rende la gerarchia della memoria funzionante. Esso infatti incrementa la probabilità di riutilizzare dei blocchi di dati che erano stati precedente mossi da un livello n ad un livello $n - 1$, riducendo il miss rate.

Il programmatore dovrà comunque fare attenzione ad implementare questo principio.

6.5.1 Locality characterization

Andiamo a distinguere due tipologie di località.

- **La località temporale** (o riuso di dati): i dati riferiti precedentemente probabilmente verranno riferiti nuovamente in un breve lasso di tempo.

Esempio 6.5.1 (Località temporale). Consideriamo il seguente codice:

```
for(int i=0; i<10; i++)
    s1 += i; s2 -= 1;
```

In questo caso le locazioni di memoria che contengono $s1$ ed $s2$ hanno località temporale.

Dunque se all'interno della gerarchia della memoria teniamo i dati più recenti, secondo il principio di località ci riaccorderò nuovamente dopo poco tempo.

- **Località spaziale:** dati vicini a quelli a cui sto facendo riferimento saranno probabilmente utilizzati a breve.

Esempio 6.5.2 (Località spaziale). Consideriamo il seguente codice:

```
for(int i=0; i<10; i++)
    func(A[i]);
```

In questo caso le locazioni di memoria dell'array hanno località spaziale, visto che sono implementate in modo contiguo.

Dunque se all'interno della gerarchia della memoria teniamo i dati vicini a quelli in utilizzo, secondo il principio di località ci saranno grosse probabilità di accedervi con la CPU.

6.6 Traferimento dati

I dati si trasferiscono solamente attraverso due memorie adiacenti. Per ottimizzare il caricamento dei dati esso viene fatto come **blocchi** di dimensione granulare (parole) in modo da poter sfruttare la località spaziale. La dimensione dei blocchi può cambiare attraverso i livelli.

Per la cache i blocchi vengono chiamati *cache line* o *cache block* (tipicamente 64-128 bytes, 8-16 parole). Per le RAM invece abbiamo *pagine di segmenti*, mentre per i dischi abbiamo *blocchi di dischi*.

Consideriamo il seguente codice in C e il suo corrispettivo in assembly.

```
// Sum and A are global variables
int i;
for(int i=0; sum=0; i<N, i++){
    sum += A[i];
}
```

```
 @ r0=&A, r1=&sum, r2=N, r3=i
loop: cmp r3, r3
      beq end
      ldr r12, [r0, r3, lsl #2]
      ldr r4, [r1]
      add r4, r4, r12
      str r4, [r1]
      add r3, r3, #1
      b loop
end: ...
```

In questo frammento di codice il loop viene eseguito N volte, e quindi ogni struttura viene richiesta N volte in maniera sequenziale. In questo caso sia la localizzazione temporale che spaziale viene utilizzata.

'Sum' è ripetutamente letta e scritta, quindi utilizza la località temporale, 'A' è salvata come un insieme contiguo di celle di memoria, quindi utilizza la località spaziale.

7 Input output

Quando andiamo a parlare dei sistemi di input output le cose principale da considerare sono i dispositivi di I/O i device controller, i buses, la gestione dell'I/O ed i device drivers.

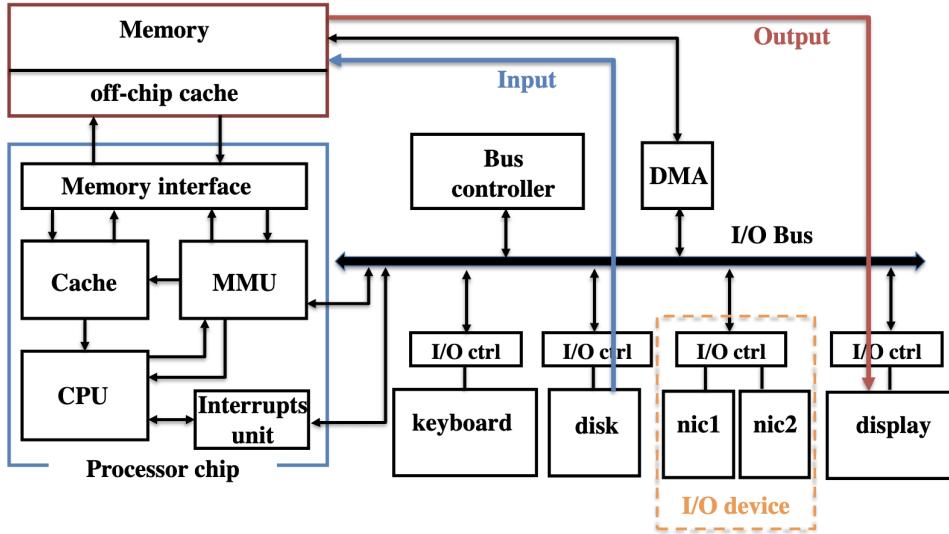


Figure 5: Input/output e I/O devices

Solitamente le operazioni di I/O hanno un grosso impatto sul tempo di esecuzione dei programmi. Supponiamo per esempio di andare a calcolare il tempo di esecuzione con $T_{exe} = T_{cpu} + T_{I/O} = \frac{1}{10}T_{cpu}$ perciò $T_{exe} = \frac{11}{10}T_{cpu}$.

Se andiamo ad aumentare il tempo T_{cpu} di 10 volte lasciando inalterato il $T_{I/O}$ abbiamo un $T_{cpu}^{enhanced} = \frac{1}{10}T_{cpu}$ così $T_{exe} = \frac{1}{10}T_{cpu} + \frac{1}{10}T_{cpu} = \frac{1}{5}T_{cpu}$ Consideriamo copi che l'aumento di velocità uguale a:

$$\frac{T_{exe}\text{before enhancement}}{T_{exe}\text{after enhancement}} = \frac{11}{5} = 5.5$$

7.1 Legge di Amdahl

Proposto da Gene Amdahl nel 1967. Si occupa della potenziale velocità massima raggiungibile da un programma parallelo quando si aumenta il numero di processori da 1 a N. Può essere applicato a qualsiasi processo di ottimizzazione. Si consideri un programma in cui solo la frazione f può essere ottimizzata (ad esempio, parallelizzata utilizzando N processori), mentre la frazione (1-f) rimane inalterata (ad esempio, è intrinsecamente sequenziale).

$$\text{Speedup} = \frac{\text{Tempo di esecuzione prima del miglioramento}}{\text{Tempo di esecuzione dopo il miglioramento}} = \frac{T(1-f) + Tf}{T(1-f) + \frac{Tf}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

Osservazione 7.1.1. L'accelerazione è vincolata dalla frazione sequenziale (1-f), cioè dalla parte del processo che non posso (o non sono in grado) di valorizzare!

Le prestazioni del sottosistema I/O sono molto importanti, ma non sono tutto. Altri aspetti importanti sono:

- **Affidabilità** gestite da metriche del tempo medio al guasto (MTTF), prevenzione dei guasti (componenti migliori), tolleranza ai guasti (introduzione di un certo livello di ridondanza).
- **Disponibilità** invece gestite da Tempo medio di riparazione (MTTR), e dalla formula $\frac{MTTF}{MTTF+MTTR}$

I dispositivi I/O hanno due tipologie di porte: porte di controllo, e porte data.

- Controllo: sia comandi che rapporti di stato, come diciamo al dispositivo cosa fare, come il dispositivo ci racconta le sue caratteristiche, come il dispositivo ci informa sullo stato operativo.
- Data: Alla/dalla memoria del dispositivo.

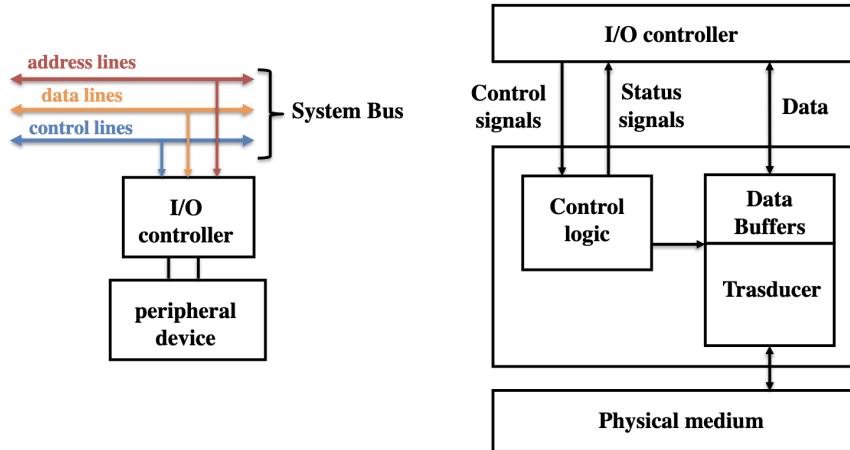


Figure 6: I/O device generico

I/O funzioni di controllo abbiamo invece il control and timing, la comunicazioni fra processi (command decoding, data exchange, status reporting, address recognition), comunicazione fra i device, il data buffering (per ottimizzare il trasferimento dei dati e compensare le differenze di velocità).

7.2 Connessione bus

Servono per il controllo del trasferimento da un dispositivo al processore:

1. la CPU controlla lo stato del dispositivo del modulo I/O.
2. Il controller I/O restituisce lo stato se pronta.
3. La CPU richiede il trasferimento dei dati tramite un comando al controllore.
4. Il controller I/O riceve i dati dal dispositivo periferico.
5. il controller I/O trasferisce i dati al processore.

Questi passaggi richiedono una o più azioni di arbitrato del bus per implementare il protocollo di comunicazione. Andiamo ora a definire l'interconnessione fra bus, essa si definisce come una raccolta di linee di dati trattate insieme come un singolo segnale logico utilizzato anche per indicare una raccolta condivisa di linee con più dispositivi connessi (chiamati rubinetti), si definiscono su essi alcuni fattori di prestazione: lunghezza fisica, numero di prese collegate.

Un bus è un mezzo di trasmissione condiviso, solo un dispositivo alla volta può trasmettere con successo. Il bus di sistema collega i principali componenti (processore, memoria, bus I/O), centinaia di linee separate, linee classificate in tre gruppi: dati, indirizzo, controllo.

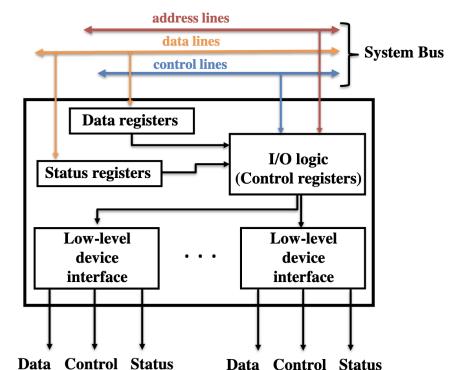


Figure 7: I/O device generico

Alcuni tipi di bus sono:

- **System bus** Collega processore e memoria, I/O interfacciato con adattatori: breve, pochi tocchi e quindi più veloce e larghezza di banda elevata, inoltre non è standard (ovvero specifico del sistema).
- **I/O bus** Connnette dispositivi I/O, nessuna connessione diretta con processore e memoria: più tempo, più tocchi è uguale a più lentezza e larghezza di banda inferiore. E' uno standard di settore (ad es. ATA/SCSI).
- **Backplane bus** Connnette CPU, memoria, dispositivi I/O. Lunghe, molti tocchi allora lento e larghezza di banda medio/bassa. Gestisce diversi dispositivi con diverse larghezze di banda ed è standard di settore.

7.2.1 Bus design

Il design di un bus si basa sul principio di soddisfare gli obiettivi di avere un alta **performance**, **standardizzazione** e bassi costi. Per esempio il bus di sistema enfatizza le prestazioni, l'I/O e i bus backplain enfatizzano i costi e la standardizzazione.

I punti cruciali nel design di un bus sono:

- I cavi sono **condivisi** o separati? I bus più ampi (maggior larghezza di banda) sono più costosi e più suscettibili allo *skew*.
- Come viene acquisito e rilasciato il **controllo** del bus? Tramite due metodologie:
 - *Atomic transactions*: fornisce una bassa utilizzazione ma anche una bassa complessità
 - *Split-transactions*: le richieste/risposte possono essere intervallate; ciò significa che fornisce un utilizzo più elevato ma al costo di una complessità maggiore
- I bus sono **sincronizzati**?
 - *Sincrono*. Tutti i dispositivi collegati al bus condividono lo stesso bus clock. Gli eventi si verificano all'estremità del segnale di clock. Un protocollo possibile è che al ciclo X l'unità di I/O scrive una richiesta READ sul bus; al ciclo $X + \Delta$ l'unità può leggere i dati dal bus. Δ è il tempo massimo per scrivere i dati sul bus da parte di un'unità collegata. Potenzialmente c'è il problema di skew dell'orologio. Viene principalmente utilizzata nei bus brevi (e.g. bus di sistema).
 - *Asincrono*. Il bus non ha il clock. Nessun disallineamento dell'orologio, si occupa di dispositivi con velocità diverse: può essere più lungo, quindi più lento. Richiede protocolli di **handshaking**. Un possibile protocollo (3 linee di controllo, 1 linea dati in cui i dati e l'indirizzo sono multiplexati):
 1. UIO1 scrive una richiesta READ (WRITE) ReadReq (WriteReq) nella linea di controllo e l'indirizzo nella linea dati
 2. UIO2 legge il indirizzo e scrive un ACK nella linea di controllo a UIO1
 3. UIO2 scrive i dati sul bus e scrive la riga DataReadycontrol per notificare UIO14
 4. UIO1 legge i dati e invia un ACK nella linea di controllo a UIO2
- Come si decide chi prende un bus per trasferire dati (**arbitraggio**)? Le comunicazioni bus devono essere gestite da un protocollo di comunicazione. I due concetti principali sono:
 - **Bus master**: un'unità che può far partire una richiesta ad un bus. La configurazione più semplice prevede solo un master ma di solito ce ne sono molteplici.
 - **Arbitration**: il master viene scelto tra più richieste cercando di implementare la priorità e l'equità (prevenendo la starvation). C'è solo un master alla volta (tutti gli altri ascoltano il bus). Il ruolo dell'arbitro è gestire le richieste del bus e assegnare le sovvenzioni considerando le priorità e l'equità. Esistono due modi per implementare l'arbitro:

- * **Centralizzato:** un dispositivo dedicato (Arbiter) raccoglie le richieste e poi decide (potenziale problema di collo di bottiglia). Si può usare il metodo *Daisy Chain*, non equo in quanto i dispositivi ad alta priorità intercettano quelli a bassa priorità. Un altro metodo è tramite linee di *richiesta/risposta* indipendenti.
- * **Distribuito:** ogni dispositivo vede le richieste contemporaneamente e partecipa alla selezione del master successivo. Più complesso da realizzare e necessita di molte linee di controllo.

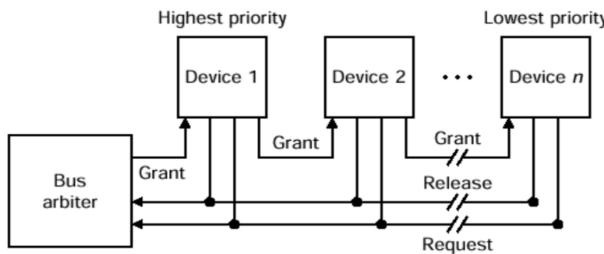


Figure 8: Daisy chain arbiter

7.3 Gestione dell'I/O

7.3.1 Invio dei comandi ai dispositivi

Soltanto il sistema operativo può mandare comandi ai dispositivi di I/O ed il programmatore deve quindi utilizzare delle chiamate di sistema. Abbiamo due opzioni:

- **Istruzioni I/O:** istruzioni ISA che indirizzano i registri del dispositivo I/O. Sono delle istruzioni privilegiate disponibili solamente in kernel mode (e.g. Intel IA-32).
- **Memory-mapped I/O:** una porzione degli indirizzi fisici è riservata all'I/O. I registri interni dei dispositivi sono mappati su locazioni della memoria principale (a *indirizzi fisici riservati*). I comandi di I/O sono quindi letture/scritture di memoria standard. Le operazioni in quelle posizioni vengono *reindirizzate* ai controller del dispositivo dalla MMU. Gli accessi in modalità utente alle aree mappate in memoria generano *eccezioni* di violazione della memoria.

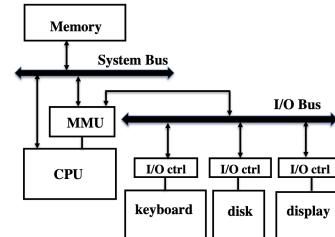


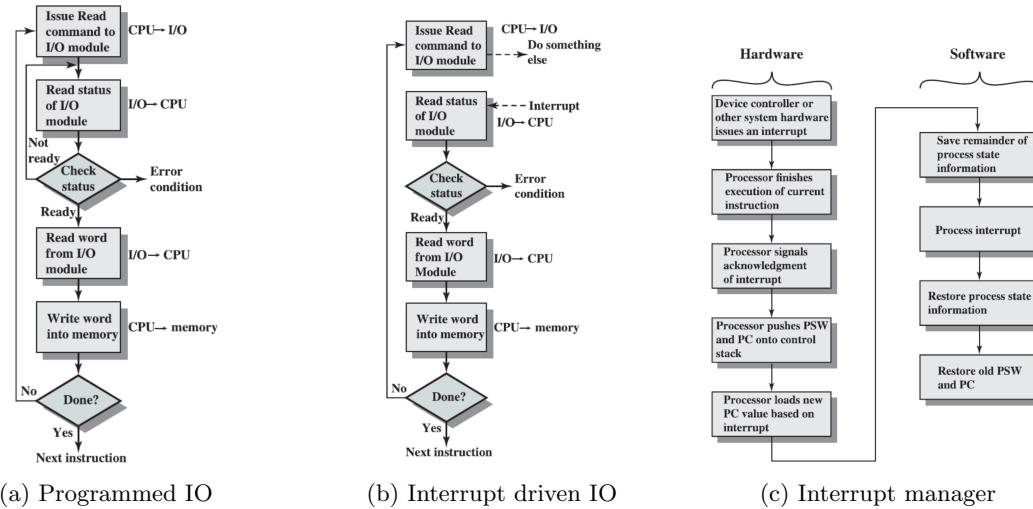
Figure 9: Memory-mapped

7.3.2 Ottenere lo stato dell'I/O

Come fa la CPU a sapere quando i dispositivi di I/O hanno completato le operazioni? Introduciamo due concetti:

- **Programmed I/O:** la CPU gestisce direttamente le operazioni (read write, transfer) e gli status di I/O. Il dispositivo è quindi *passivo*. La CPU è detta in uno stato di **polling**, ovvero periodicamente controlla lo stato del dispositivo. È semplice da implementare ma sposta molti cicli di clock.
- **Interrupt-driven I/O:** un interrupt è un evento asincrono proveniente da un dispositivo (e.g. modulo I/O, timer, controller DMA). È un'alternativa al polling. La CPU invia dei comandi ad un dispositivo e nel frattempo fa qualcosa' altro. Al termine di ogni ciclo fetch-decode-execute, controlla se sono arrivati interrupt. In caso positivo salva lo stato corrente (passando in privileged-mode), esegue il programma, e poi ripristina il contesto.

Note 7.3.2.1 (Eccezioni). A volte le **eccezioni** sono anche chiamate interruzioni (o comprendono interruzioni). In generale, le eccezioni sono eventi **sincroni** che interrompono l'esecuzione del programma (e.g. overflow aritmetico, istruzione non definita).



Esempio 7.3.1. Facciamo ora un esempio, assumiamo di avere una CPU ad una velocità di 500MHz ed un costo di polling di 400 cicli. Abbiamo poi un mouse come dispositivo a banda lenta. Vogliamo aver un poll di 30 volte al secondo.

Per calcolare i cicli al secondo per polling scriviamo $(30 \text{ poll/s}) * 400 = 120000 \text{ cycles/s}$, e la percentuale di cicli spesi per polling si calcola facendo $12K/500M = 0.002\%$. Abbiamo dunque un overhead accettabile.

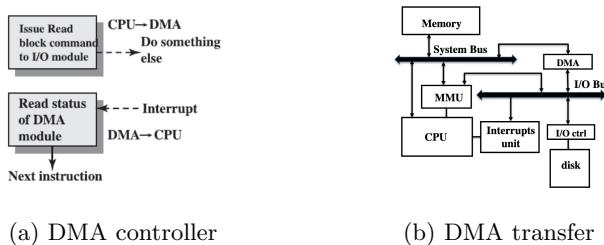
Prendiamo ora un disco (che è un dispositivo ad alta lunghezza di banda) che ha 4MB/s di transfer rate e 16B interface.

Per non mancare i dati dobbiamo eseguire il poll abbastanza spesso, quindi $(4\text{MB/s})/16\text{B} = 250 \text{ K/s}$, vista come percentuale di cicli abbiamo $100\text{M} / 500 \text{ M} = 20\%$, questo risultato non è accettabile perché Abbiamo speso il 20% dei cicli solo per controllare i registri I/O, non per il trasferimento dei dati

7.4 Data transfers (DMA)

Andiamo ora a rispondere all'ultima domanda che ci eravamo posti, come i device di I/O eseguono il trasferimento dei dati? Sappiamo ora che l'I/O guidato da interrupt elimina i problemi di polling, tuttavia, il sistema operativo deve trasferire i dati una parola alla volta. Questo va bene solo per dispositivi I/O a bassa larghezza di banda (ad es. mouse), per ovviare a questo problema introduciamo la direct memory access (DMA).

Esso è meccanismi che forniscono a un dispositivo di controllo I/O la capacità di trasferire i dati



direttamente da e verso la memoria principale senza coinvolgere la CPU. DMA disaccoppia il protocollo CPU-memoria dal protocollo memoria-dispositivo I/O. Interrupt utilizzati dal dispositivo I/O

per comunicare con la CPU solo al completamento del trasferimento I/O o quando si verifica un errore.

DMA è implementato con un controller specializzato. Per eseguire DMA, un dispositivo I/O è collegato a un controller DMA: è possibile collegare più dispositivi allo stesso controller. Il controller stesso è visto come un dispositivo I/O mappato in memoria, inoltre il controller DMA si occupa dell'arbitrato del bus e del trasferimento dei dati: diventa il master del bus. Il controller DMA e la CPU si contendono il bus di memoria. Ci sono tre passi nel transfer DMA:

1. La CPU imposta DMA fornendo identità, op, indirizzo di memoria e numero di byte da trasferire.
2. DMA avvia l'operazione sul dispositivo e arbitra la connessione. Trasferisce i dati dal dispositivo o dalla memoria.
3. Una volta completato il trasferimento DMA, il controller invia un interrupt alla CPU.

Esempio 7.4.1. Facciamo un esempio, assumiamo i seguenti dati: 500Mhz CPU, il device di un disco ha un tempo di trasferimento 4MB/s, 16B di interfacce e un utilizzo di 50%, la gestione degli interrupt richiede 400 cicli, il trasferimento dei dati richiede 100 cicli, l'installazione DMA richiede 1600 cicli, trasferisce un blocco da 16 KB alla volta.

Overhead del processore per I/O guidato da interrupt senza DMA (il processore è coinvolto per ogni trasferimento di dati (16 B)) si calcola facendo $0.5 * (4MB/s) / (16B) * [(400 + 100) \text{ cicli} / 500M \text{ cicli/s}] = 12.5\%$.

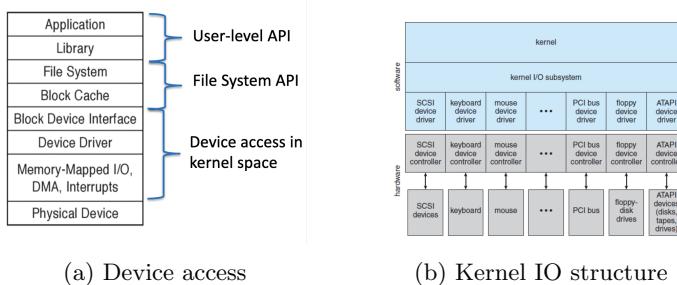
Overhead del processore per I/O guidato da interrupt con trasferimento DMA (Il processore è coinvolto una volta per trasferimento a blocchi (16 KB)) si calcola invece facendo $0.5 * (4M B/s) / (16KB) * [(1600 + 400) / (500M \text{ cicli/s})] = 0.05\%$

Notiamo da questo esempio che Senza DMA: il processore avvia tutti i trasferimenti di dati, tutti i trasferimenti passano attraverso la traduzione degli indirizzi (MMU), i trasferimenti possono essere di qualsiasi dimensione e attraversare i limiti della pagina virtuale, nessun impatto sulla gerarchia della cache, le cache non contengono mai dati obsoleti.

Mentre con DMA: il controller DMA avvia i trasferimenti di dati, esiste un altro percorso per il sistema di memoria, potenziali problemi: i controller DMA utilizzano indirizzi virtuali o fisici? Cosa succede se scrivono i dati in una posizione di memoria cache?

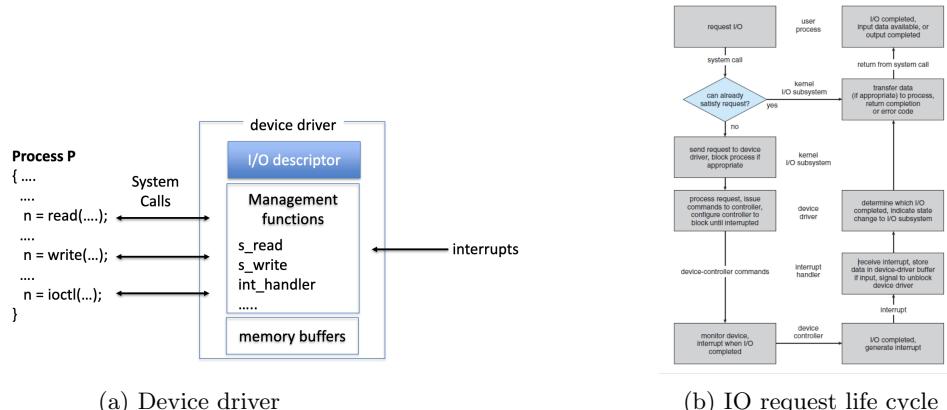
Per rispondere alla domanda, quali indirizzi il processore specifica al controller DMA? Abbiamo la virtual DMA e la DMA fisica:

- Virtual DMA: Il controller DMA deve eseguire internamente la traduzione degli indirizzi utilizzando una piccola cache (TLB) inizializzata dal sistema operativo quando richiede un trasferimento I/O. Complesso ma flessibile (ad esempio, grandi trasferimenti tra pagine).
- Physical DMA: Il controller DMA funziona con indirizzi fisici. Trasferimenti alla granularità della pagina, il sistema operativo suddivide i trasferimenti di grandi dimensioni in blocchi delle dimensioni della pagina. Più semplice, ma meno flessibile.



La DMA ha anche un'integrazione con la cache essendo che la memorizzazione nella cache riduce le istruzioni della CPU e la latenza di accesso ai dati e riduce l'utilizzo della memoria da parte della CPU, lasciando così più larghezza di banda di memoria/bus per i trasferimenti DMA. Ma le cache introducono un problema di coerenza per DMA: se il DMA scrive nella memoria principale, la versione dei dati nella cache è obsoleta. Per le cache write-back, il controller DMA potrebbe leggere i vecchi valori dei dati dalla memoria mentre i dati nella cache hanno il set di bit sporchi. La soluzione è lo svuotamento/invalidamento selettivo delle linee memorizzate nella cache coinvolte nei trasferimenti DMA: richiede una logica aggiuntiva per la coerenza della cache HW!

L'accesso a device è gestito tramite uno stack software che fornisce modi per accedere a un'ampia gamma di dispositivi I/O. Un'interfaccia comune, in POSIX equivalente all'accesso ai file, in termini di chiamate di sistema di apertura/chiusura, lettura/scrittura. Driver di dispositivo per ogni dispositivo specifico, parte superiore HW, indipendenti (per migliorare la portabilità), parte inferiore HW-dipendente.



8 Dischi rigidi

Alcuni tipi di storage device sono per esempio i dischi magnetici e le flash memory.

Definizione 8.0.1 (Dischi magnetici). *I dischi magnetici è un tipo di archiviazione che raramente viene danneggiata. Grande capacità a basso costo. Accesso casuale a livello di blocco. Prestazioni lente per l'accesso casuale. Migliori prestazioni per l'accesso in streaming.*

Definizione 8.0.2 (Flash memory). *Archiviazione che raramente viene danneggiata. Buona capacità a costo intermedio ($2 \times$ disco). Accesso casuale a livello di blocco. Buone prestazioni per le letture; peggio per le scritture casuali.*

8.1 Dischi magnetici

I dischi magnetici hanno circa 1 micron di larghezza, lunghezza d'onda della luce è di circa 0.5 micron, risoluzione dell'occhio umano: 50 micron, 100K su un tipico disco da 2,5 pollici. Separato da regioni di guardia inutilizzate, riduce la probabilità che le tracce vicine vengano danneggiate durante le scritture (ancora una piccola possibilità diversa da zero).

La lunghezza della traccia varia a seconda del disco, all'esterno: più settori per traccia, maggiore larghezza di banda, il disco è organizzato in regioni di tracce con lo stesso numero di settori/traccia, viene utilizzata solo la metà esterna del raggio. La maggior parte dell'area del disco nelle regioni esterne del disco

8.1.1 Settore

I settori contengono sofisticati codici di correzione degli errori, il magnete della testina del disco ha un campo più ampio della traccia, nasconde le corruzioni dovute alle scritture di tracce vicine

- **Sector sparing.** Rimappa i settori danneggiati in modo trasparente ai settori di riserva sulla stessa superficie.
- **Slip sparing.** Rimappa tutti i settori (quando c'è un settore danneggiato) per preservare il comportamento sequenziale.
- **Track skewing.** Numeri di settore spostati da una traccia all'altra, per consentire il movimento della testina del disco per operazioni sequenziali.

A basso livello il controller accede ai singoli settori. La dimensione tipica del settore è di 256/512 byte. Identificato da una tripla: $\langle \#cilindro, \#faccia, \#settore \rangle$.

Al livello superiore, il driver del disco raggruppa un insieme di settori contigui in un blocco. La dimensione tipica del blocco è di 2/4/8 Kbyte, identificata da un puntatore su uno spazio di indirizzi contiguo (da 0 a max-blocks).

Dato un numero di sector b , ed una tripla $\langle c, f, s \rangle$ abbiamo che.

$$b = c(\#faces \cdot \#sectors) + f(\#sectors) + s$$

#faces sono il numero di facce in un disco mentre #sectors sono il numero di settori per track. Di conseguenza a questa formulaabbiamo che:

$$c = b \text{ div } (\#faces \cdot \#sectors) \quad f = (b \text{ div } (\#faces \cdot \#sectors)) \text{ div } \#sectors$$

$$s = (b \text{ div } (\#faces \cdot \#sectors)) \text{ mod } \#sectors$$

Esempio 8.1.1. Disco con 100 cilindri, 4 facce, 2000 settori per traccia. Un file utilizza i blocchi 94421, 94422, 94423. Qual è il $\langle c, f, s \rangle$ per ogni blocco? Possiamo calcolare facendo:

$c = 94421 \text{ div } (4 \cdot 2000) = 1$ (rimangono 6421). $f = 6421 \text{ div } 2000 = 3$ (rimangono 421) quindi $s = 421$. Pertanto le triple sono: $\langle 11, 2, 421 \rangle, \langle 11, 3, 422 \rangle, \langle 11, 3, 423 \rangle$.

8.1.2 Disk performance

Fra i punti principali nel calcolo delle performance di un disco è la sua latenza. Per calcolare la disk latency:

$$\text{Disk latency} = \text{Seek time} + \text{Rotation time} + \text{Transfer Time}$$

In questa formula possiamo notare 3 componenti:

- **Seek time.** Tempo per spostare il braccio del disco sulla traccia (1-20ms). Regolazione fine della posizione necessaria affinché la testina si "assesti". Tempo di cambio testina = tempo di cambio traccia (su dischi moderni).
- **Rotation time.** Tempo di attesa per la rotazione del disco sotto la testina del disco. Rotazione del disco: 4, 15 ms (a seconda del prezzo del disco).
- **Transfer time.** Tempo di trasferimento dei dati su/fuori dal disco. Velocità di trasferimento testina disco: 50-100 MB/s (5-10 usec/settore). Velocità di trasferimento host dipendente dal connettore I/O (USB, SATA, ...).

Esempio 8.1.2. Quanto ci vuole per completare 500 letture casuali del disco, in ordine FIFO? Ricerca: media 10,5 msec. Rotazione: media 4,15 msec. Trasferimento: 5-10 usec. $500 * (10,5 + 4,15 + 0,01) / 1000 = 7,3$ secondi.

Esempio 8.1.3. Quanto tempo occorre per completare 500 letture sequenziali del disco? Tempo di ricerca: 10,5 ms (per raggiungere il primo settore). Tempo di rotazione: 4,15 ms (per raggiungere il primo settore). Tempo di trasferimento: (traccia esterna) 500 settori * 512 byte / 128 MB/sec = 2 ms Totale: $10,5 + 4,15 + 2 = 16,7$ ms.

Potrebbe essere necessaria una testina aggiuntiva o un interruttore di traccia (+1ms). Il buffer di traccia può consentire la lettura fuori servizio di alcuni settori dal disco (-2ms)

Esempio 8.1.4. Disco con 100 cilindri, 4 facce, 2000 settori per traccia. Un file usa i blocchi 94421, 94422, 94423 ($< 11, 3, 421 > < 11, 3, 422 > < 11, 3, 423 >$).

Un settore viene letto in 0,01 ms, il tempo di ricerca tra due tracce consecutive è di 0,01 ms e il tempo medio per raggiungere il settore desiderato è la metà del tempo di rotazione.

Considerando che il braccio è al cilindro 22 e che il controller DMA ha abbastanza buffer, calcola il tempo per leggere i blocchi di file. $(22-11)*0.01 + 20/2 + 3*0.01 = 10.14\text{ms}$

8.2 Dishi SSD

Un Solid StateDrive (SSD) è un dispositivo di archiviazione non volatile basato sulla tecnologia di memoria flash. Gli SSD sono più affidabili e più veloci dei dischi rigidi (HD) perché non hanno parti mobili e nessun tempo di ricerca, gli SSD usano comunemente una semplice pianificazione FCFS policy. Consumano meno energia e sono più costosi per MB di dati. La capacità dell'HD è in genere maggiore. In alcuni sistemi, gli SSD vengono utilizzati come ulteriore livello di cache nella gerarchia della memoria.

All'interno delle flash memory abbiamo che le scritture devono essere per "pulire" le celle; nessun aggiornamento in atto. Cancellazione di blocchi di grandi dimensioni richiesta prima della scrittura. Blocco di cancellazione fra 128 ed i 512 KB. Tempo di cancellazione: diversi millisecondi, scrittura/lettura pagina (2-4 KB) in circa 50-100 usec.

8.2.1 Flash translation layer

Per evitare il costo di una cancellazione per ogni scrittura, le pagine vengono cancellate in anticipo. Pagine pulite sempre disponibili per una nuova scrittura, questo significa che una scrittura non può essere indirizzata ad una pagina arbitraria in memoria, ma solo ad una pagina precedentemente cancellata, cosa succede se si riscrive un blocco di un file? La pagina che contiene il blocco non può da riscrivere subito... Deve essere prima cancellato ma con le pagine circostanti! Viene utilizzata una pagina pulita per applicare la scrittura, ma questa pagina è da qualche parte nel disco... La vecchia pagina va nella spazzatura per il riciclaggio. Come sapere dove sono le pagine del mio file?

Il firmware del dispositivo flash associa la pagina logica # a una posizione fisica: sposta le pagine

live secondo necessità per la cancellazione. Garbage collect blocco di cancellazione vuoto copiando le pagine live in una nuova posizione, abbiamo dunque livellamento dell’usura. Puoi scrivere ogni pagina fisica solo un numero limitato di volte: evita le pagine che non funzionano più. Trasparente per l’utente del dispositivo.

8.2.2 File system flash

I file system sui dischi magnetici non hanno bisogno di dire al disco quali blocchi sono in uso: quando un blocco non è più utilizzato viene contrassegnato come libero nella bitmap, il file system lo riutilizza quando vuole. Quando questi FS sono stati utilizzati per la prima volta su unità flash, le prestazioni sono diminuite drasticamente nel tempo.

Il Flash Translation Layer si è dato da fare con la garbage collection: i blocchi live devono essere rimappati in una nuova posizione, per compattare le pagine libere per poter procedere con la cancellazione dei blocchi. Questo anche con una grande quantità di spazio libero, ad esempio, se FS sposta un file di grandi dimensioni da un intervallo di blocchi a un altro, lo storage non sa che i vecchi blocchi possono andare nella spazzatura, a meno che FS non lo dica!

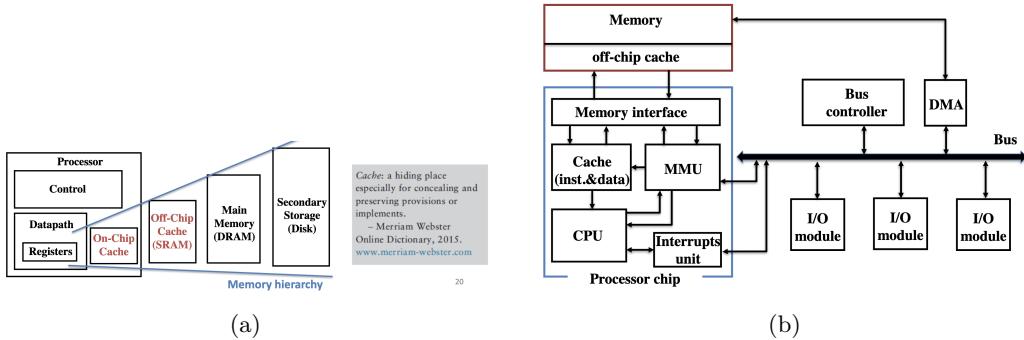
Comando TRIM: il file system indica al dispositivo quando le pagine non sono più in uso. Aiuta il livello di traduzione dei file a ottimizzare la raccolta dei rifiuti. Introdotto tra il 2009 e il 2011 nella maggior parte dei sistemi operativi.

9 Cache

La cache memory è la memoria più vicina al processore, solitamente sono le SRAM, ma alcune volte sono implementate anche come DRAM. Ad oggi tutte le architetture hanno alcuni livelli di cache integrati nel chip, essa può essere più o meno grande e può avere più di un livello.

9.1 Gestione del movimento dei dati

Fra il primo livello di cache e i registri il trasferimento è gestito dal compilatore. Il trasferimento fra caches e RAM viene invece gestito dalla microarchitettura. Infine la gestione dei trasferimenti fra RAM e storage viene fatta dal sistema operativo.



9.2 Utilizzo

L'organizzazione della cache avviene non a blocchi ma a linee. Ogni linea contiene blocchi di memoria (8-16 memory words). La prima volta che il processore richiede la memory word avviene una **cache miss**. A quel punto il blocco contenente la parola viene trasferito dentro la cache.

La richiesta successiva può essere di due tipologie:

- **Cache hit:** se il dato è presente nel blocco.
- **Cache miss:** se il dato non è presente dentro il blocco. In questo caso il blocco che contiene il dato viene trasferito dentro la cache line.

Vediamo ora l'effetto della cache sull'AMAT. Innanzitutto l'utilizzo di grandi cache nella gerarchia delle memorie aiuta a ridurre il bottleneck di von Neumann.

Esempio 9.2.1. Vediamo un esempio quantitativo stabilendo dei valori:

$t_M = 50\text{ns}$ (main memory service time), $t_{L1} = 1\text{ns}$ (L1 hit time, cache hit service time).

Abbiamo un Miss rate (MR_{L1}) del 5%.

Senza cache $AMAT = 50\text{ns}$ mentre con L1 cache $AMAT = t_{L1} + MR_{L1} * t_M = 1 + 0.05 * 50 = 3.5\text{ns}$

9.3 Performance

$$CPU_{time} = \text{ClockCycles} \cdot \text{ClockCycleTime} = IC \cdot CPI \cdot \text{ClockCycleTime} \quad (8)$$

dove

- **IC** (Instruction Count) è il numero di istruzioni che vengono effettivamente eseguite
- **CPI** (ClockCycles Per Instruction) è definito come $\frac{\text{clockcycles}}{IC}$

Possiamo inoltre dividere il CPU_{time} tra $CPI_{Perfect}$, ovvero il tempo che la CPU impiega per eseguire un'istruzione senza *misses*, e CPI_{Stall} , ovvero il tempo che la CPU impiega per aspettare la memoria.

$$CPU_{time} = (CPI_{Perfect} + CPI_{Stall}) \cdot \text{ClockCycleTime} \quad (9)$$

Le due parti le possiamo calcolare come segue:

$$\begin{aligned}
 CPI &= \left(\frac{IC_{CPU}}{IC} \right) \cdot CPI_{CPU} + \left(\frac{IC_{MEM}}{IC} \right) \cdot CPI_{MEM} \\
 CPI_{MEM} &= CPI_{MEM-HIT} + \text{Miss Rate} \cdot CPI_{MEM-MISS} \\
 CPI_{\text{Perfect}} &= \frac{IC_{CPU}}{IC} \cdot CPI_{CPU} + \frac{IC_{MEM}}{IC} \cdot CPI_{MEM-HIT} \\
 CPI_{\text{Stall}} &= \frac{IC_{MEM}}{IC} \cdot \text{Miss Rate} \cdot \text{Miss Penalty}
 \end{aligned} \tag{10}$$

Osservazione 9.3.1. IL CPI_{Stall} è definito come la somma dei cicli di stallo in lettura e scrittura. Per semplicità abbiamo assunto che i miss rate e le miss penalties fossero identiche per lettura e scrittura e che gli stalli per scrivere nei buffer fossero trascurabili.

Esempio 9.3.1. Assumiamo che abbiamo un miss rate del 2% per la cache delle istruzioni mentre un 4% per la cache dei dati, ed una miss penalty di 100 cicli per ogni mancanza. Una frequenza del 36% per le *ldr*, e le *str*. Se la CPI è 2 senza memory stalls, dobbiamo determinare quanto va più veloce un processore con una cache perfetta rispetto ad una cache reale (che ha le caratteristiche elencate sopra).

$$CPI_{Stall-Instr} = 1 \cdot 0.02 \cdot 100 = 2 \text{cycles}$$

$$CPI_{Stall-Data} = 0.36 \cdot 0.4 \cdot 100 = 1.44 \text{cycles}$$

$$CPi_{stall} = 2 + 1.44 = 3.44$$

spendiamo in media 3.44, e quindi in totale il nostro processore $CPI = 2 + 3.44 = 5.44$.

$$\frac{CPU_{\text{time with stalls}}}{CPU_{\text{time Perfect}}} = \frac{(CPI_{\text{Perfect}} + CPI_{\text{Stall}}) \cdot ClockcycleTime}{CPI_{\text{perfect}} \cdot ClockCycleTime} = \frac{5.44}{2}$$

Come abbiamo visto nell'esempio è quindi importante tenere bassi:

- **Hit-time**, che deriva dalla tecnologia utilizzata per la cache
- **Miss penalty**, che deriva dall'architettura utilizzata
- **Miss rate**, che può essere abbassato anche tramite tecniche di programmazione

9.4 Design

Una delle prime domande che dobbiamo porci è come i dati sono organizzati.

Partiamo da una cache di una capacità C, organizzata come S sets in cui ciascuno contiene B block (o linee), dove b è il numero di parole per blocco. Tutto questo in un sistema a 32bit (quindi al massimo possiamo indirizzare 2^{30} parole).

Da qui possiamo distinguere vari metodi di organizzazione:

1. **Direct mapped**: in questo caso $\#S = \#B$
2. **N-way set-associative**, in cui N definisce il numero di blocchi contenuti in un set quindi $S = B/N$
3. **Fully associative**, in cui nell'insieme c'è uno unico blocco contenente tutta la cache, quindi $S = 1$

9.4.1 Direct mapped

Supponiamo di avere $b = 1^2$ e una cache di 8 parole, quindi $S = B = 8$, $C = B \cdot b = 8$. Per indirizzare le 8 parole ci serviranno $\log_2 8 = 3$ bits.

²Nessuna cache avrà mai b=1, è solo come esempio

In questo tipo di design, tutti gli indirizzi con il quintultimo, quartultimo e terzultimo bit in comune saranno associati ad un set nella cache predefinito.

In aggiunta nella cache sarà presente anche un **tag** che permette di identificare univocamente il dato e un valore di **controllo** che mi garantisca che quel dato sia effettivamente valido per quella posizione di cache (ad esempio appena acceso il computer potrebbe esserci un valore random non corretto).

Vediamo un caso realistico dove $b > 1$. Prendiamo $C = 8$ e $b = 4$, con $B = C/b = 2$ e $S = 2$. Dobbiamo quindi andare ad "affettare l'indirizzo" aggiungendo un **block-offset** che mi permetta di selezionare la parola corretta tra le $\log_2 4$ opzioni

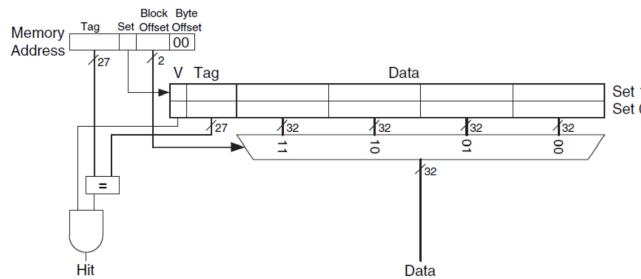
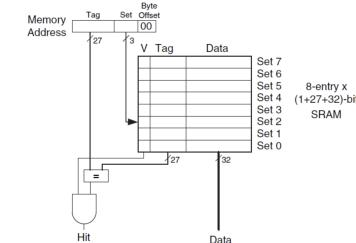
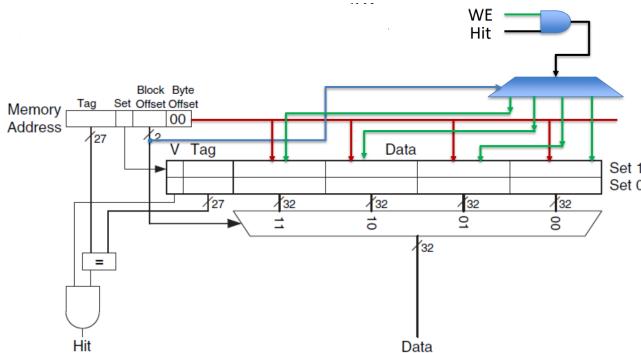


Figure 15: Direct mapped, $b = 1$

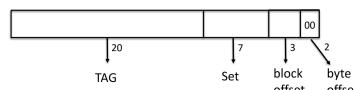


Per implementare anche la scrittura in un indirizzo, aggiorniamo lo schema come segue:



Esempio 9.4.1. Supponiamo di avere indirizzi a 32bit, una cache ad accesso diretto con $S = B = 128$ ed ognuna contiene $b = 8$. Dobbiamo trovare la struttura con cui organizzare gli indirizzi.

Ci servono 2bit per l'offset del dato, $\log_2 8 = 3$ bit per l'offset delle parole nel blocco, $\log_2 129 = 7$ bit per identificare il *set* e 20bit per il tag del dato.



Esempio 9.4.2. Consideriamo la stessa struttura dell'esempio 9.4.1. Dobbiamo calcolare la linea di cache e l'offset all'interno della linea di cache che conterrà l'indirizzo 0xFFAC.

$$0xFFAC = 0\ldots01111\textcolor{red}{111}0101\textcolor{blue}{01}\textcolor{green}{100}$$

L'indirizzo si dividerà quindi in:

- Tag
- Set

- **Block offset**
- **Byte offset**

Quindi la linea di cache ha indice 117, ovvero 118esima. L'offset nel blocco è 3 quindi la quarta parola di memoria.

Esempio 9.4.3. Consideriamo il seguente codice in C:

```
for(i=0; i<16, i++)
    C[i] = A[i] + B[i];
```

eseguito in un processore a 2Ghz con un mapping diretto L1 ai dati dalla cache con $C = 128$, $b = 8$, $t_M = 100$ cicli e $t_{L1} = 4$ cicli.

'A' inizia all'indirizzo 0x00000000, 'B' inizia all'indirizzo 0x00000040 e 'C' inizia all'indirizzo 0x00000080. Calcolare il numero di cache misses e l'AMAT considerando solo le *ldr*.

Il primo passo è verificare se A e B finiscono nello stesso set: A verrà caricato nel set 0 e nel set 1 mentre B nel set 2 e nel set 3, quindi non vanno in conflitto. Se supponiamo la cache sia vuota: avremo 4 cache misses, due per ogni word di A e B, e 32 cache hits (12 per A e 12 per B).

Il calcolo dell'AMAT sarà

$$\begin{aligned} MissRate &= \frac{4}{32} \\ ClockCycleTime &= \frac{1}{2GHz} = 0.5ns \\ AMAT &= \left(4 + \frac{4}{32} \cdot 100\right) \cdot ClockCycleTime = 8.25ns \end{aligned}$$

In generale, se non c'è **località temporale**, il numero di cache misses è $\frac{N}{b}$, dove N è il numero di istruzioni (in questo caso $N = 2 \cdot 16$).

Per riassumere possiamo dire che i pro sono:

- La realizzazione è semplice
- E' molto veloce in caso di hits

mentre i contro:

- La funzione di mapping a posizione fisxe può generare potenzialmente molti conflitti.
- La troppa rigidità potrebbe avere un grosso impatto sul numero dei conflitti nella cache (**trashing**) in quanto essi dipendono dal posizionamento della memoria e dall'utilizzo delle strutture dati.

Definizione 9.4.1 (Working set). *Data una gerarchia di memoria M1-M2, il working set di un programma è l'insieme dei dati che, se simultaneamente presenti in M1, massimizza la probabilità di hits in M1.*

Esempio 9.4.4 (Working set). Prendiamo per esempio il seguente codice:

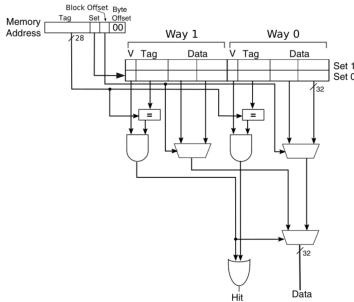
```
// A and B are two arrays of size N
// (the cache block b << N)
for(i=0; i<N; ++i)
    for(j=0; j<N; ++j)
        A[i] = F(A[i], B[j])
```

in cui l'array A ha località spaziale e B quella spaziale (sul lungo termine anche temporale). Il nostro WS sarà l'elemento corrente di A e l'intero array B. Quindi, se la cache contenesse il WS avremmo solamente $O(\frac{N}{b})$ fallimenti.

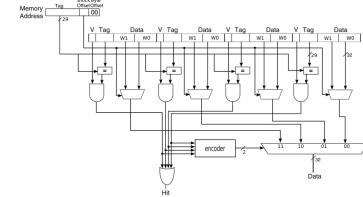
9.4.2 Associative

Esistono due tipi di cache associative:

- **Fully associative:** i blocchi possono essere posizionati in qualunque *cache line*. Per trovarlo bisogna scansionare tutte le linee; questo implica un comparatore per ogni linea (più costoso).
- **Set associative:** i blocchi possono essere posizionati in un certo numero di set, dove ogni set contiene un certo numero di linee che possono essere utilizzate indiscriminatamente (purché del set giusto).



(a) Fully associative



(b) N-way associative

9.4.3 Comparazione

Data una capacità C , dobbiamo decidere la dimensione del blocco b , il numero di blocchi B (cache lines, $\frac{C}{b}$) e il numero di vie (numero di blocchi in un Set).

Organizzazione	Numero di vie N	Numero di Set S
Direct mapped	1	B
Set associative	$1 < N < B$	$\frac{B}{N}$
Fully associative	B	1

9.5 Cache miss

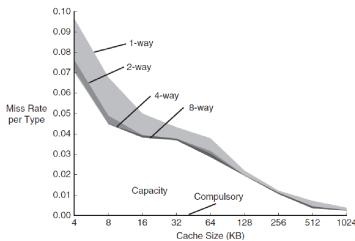
Le tipologie di cache miss sono le seguenti:

- **Compulsory miss:** causato dal primo accesso al blocco che non è mai stato in cache.
- **Capacity miss:** causato dalla cache che non contiene tutti i blocchi necessari al programma per essere eseguito (working set).
- **Conflict miss** solo per la mappatura diretta e per i set associativi

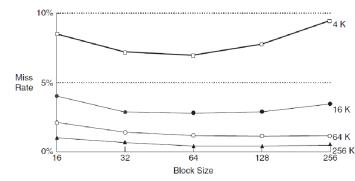
A questo punto, visti i vari strumenti della cache e le tipologie di miss possiamo vedere come ridurre i cache miss tramite l'utilizzo di alcune tecniche:

- **Aumentare la dimensione dei blocchi.** Andando così ad aumentare la località spaziale ma aumentando anche la **miss penalty**.
- **Aumentare l'associatività.** Meno conflitti ma un maggiore tempo di hit.
- **Aumentare dim. dei case.** Porta a meno capacity miss e conflitti ma aumenta l'hit time.
- Usare algoritmi **cache-oblivious** per sfruttare al massimo la località.

In caso comunque avvengano dobbiamo andarli a gestire. Un errore nella cache blocca l'intero processore, congelando il contenuto di tutti i registri durante l'attesa della memoria: in effetti, un processore più avanzato consente l'esecuzione fuori ordine di altre istruzioni durante l'attesa della memoria. Gli steps che vengono eseguiti per la gestione di un cache miss (**fault management**):



(a) Aumentare dimensione cache e associatività



(b) Aumentare la dimensione dei blocchi

1. Indica al livello di memoria successivo nella gerarchia di leggere il valore mancante.
2. Attende che la memoria risponda (questo può richiedere più cicli).
3. Aggiorna la riga della cache corrispondente con i dati ricevuti.
4. Riavvia l'esecuzione dell'istruzione (ora è un hit della cache).

Esempio 9.5.1 (Migliorare performance della CPU). Partiamo dal caso dell'esempio 9.3.1 e supponiamo di raddoppiare la velocità di clock:

$$CPI_{Stall-Instr} = 1 \cdot 0.02 \cdot 200 = 4cycles$$

$$CPI_{Stall-Data} = 0.36 \cdot 0.4 \cdot 200 = 2.88cycles$$

$$CPi_{stall} = 4 + 2.88 = 6.88$$

$$CPI = 2 + 6.88 = 8.88$$

Il tempo che stiamo fermi in attesa della memoria cresce al 77%:

$$\frac{6.88}{8.88} = 0.77$$

e abbiamo quindi un miglioramento totale di un fattore di solo:

$$\frac{CPU_{time1}}{CPU_{time2}} = \frac{(CPI_{Perf} + CPI_{Stall1}) \cdot ClockCycleTime}{(CPI_{Perf} + CPI_{Stall2}) \cdot \frac{ClockCycleTime}{2}} = \frac{5.44}{\frac{8.88}{2}} = 1.23$$

Come si vede da questo esempio, migliorare la velocità del processore o la sua architettura non migliora in maniera efficiente il tempo impiegato nell'attesa della memoria, che per l'esempio è relativamente di

$$\epsilon = \frac{1.23}{2} = 61\%$$

9.5.1 Multi-level cache

Per ovviare a questo problema vengono utilizzate cache su più livelli, dove quelle dei livelli inferiori vengono interrogate in caso di cache miss ed evitano quindi un accesso alla memoria principale (che si rende necessario solo se non si trova il dato neanche sull'ultimo livello).

Esempio 9.5.2 (Multi-level cache). Assumiamo i seguenti valori di service time

$$t_M = 50ns \quad t_{L1} = 1ns \quad t_{L2} = 6ns \quad t_{L3} = 10ns$$

e con dei miss rate rispettivamente del 10%, 1.5% e 0.4% rispettivamente per L1, L2 e L3. I valori dell'AMAT saranno:

- Senza cache: $AMAT = 50ns$
- L1: $AMAT = 1 + 0.1 * 50 = 5ns$
- L1 e L2: $AMAT = 1 + 0.1 \cdot (6 + 0.15 \cdot 50) = 1.675ns$

- L1, L2 e L3: $AMAT = 1 + 0.1 \cdot (6 + 0.015 \cdot (10 + 0.004 \cdot 50)) = 1.6153ns$

In un processore che sfrutta una cache su più livelli, il CPI_{Stall} si calcola come segue:

$$CPI_{Stall} = MissRate_{L1} \cdot MissPenalty_{L1} + GlobalMissRate_{L2} \cdot MissPenalty_{L2} + GlobalMissRate_{L3} \cdot MissPenalty_{L3} + \dots \quad (11)$$

dove

$$GlobalMissRate_{LN} = MissRate_{L1} \cdot MissRate_{L2} \cdot \dots \cdot MissRate_{LN}$$

$$MissPenalty_{LN} = HitTime_{LN+1}$$

Esempio 9.5.3 (Multi-level cache CPI). Supponiamo di avere un sistema con una cache a 2 livelli con:

$$MissRate_{L1} = 2\% \quad MissRate_{L2} = 20\%$$

e con un *access time* per L2 di 20 cicli e di 200 cicli per la memoria principale. Calcoliamo il CPI_{Stall} come segue:

$$GlobalMissRate_{L2} = MissRate_{L1} \cdot MissRate_{L2} = 0.02 \cdot 0.2 = 0.04 = 4\%$$

Un miss nella cache L1 può essere soddisfatto o dalla cache L2 o dalla memoria principale. La miss penalty è 20 se abbiamo un hit in L2 o 200 nell'altro caso. Quindi:

$$MissPenalty = 20 + 0.2 \cdot 200 = 60cycles$$

$$CPI_{Stall} = MissRate_{L1} \cdot MissPenalty = 0.02 \cdot 24 = 1.2cycles$$

oppure

$$CPI_{Stall} = 0.02 \cdot 20 + (0.02 \cdot 0.2) \cdot 200 = 1.2cycles$$

9.6 Gestione delle scritture

Distinguiamo le scritture in due casi:

- **Write hits** quando c'è lo stesso TAG: se il dato viene scritto solo nella cache, allora la memoria e la cache sono inconsistenti
- **Write miss** quando abbiamo tag diversi

9.6.1 Write-Through

Questa tecnica prevede che gli hit di scrittura aggiornino sempre sia la cache che il successivo livello di memoria.

- **Pro:** soluzione semplice, facile da implementare. I dati sono sempre coerenti tra i due livelli di memoria.
- **Contro:** la **velocità** di scrittura dipende dalla velocità di scrittura del livello di memoria inferiore. Maggiore **traffico di memoria**, per ogni singola scrittura potrebbero esserci più scritture in ogni livello di memoria.

9.6.2 Write-Back

Gli hit di scrittura aggiornano solo la cache, quindi il blocco modificato viene scritto nel livello di memoria inferiore quando viene sostituito.

- **Pro:** la **velocità** delle scritture è quella della cache: minore traffico di memoria rispetto a Write-Through, le successive scritture sullo stesso blocco di cache non producono traffico con il livello di memoria inferiore più costoso.
- **Contro:** Abbiamo bisogno di tenere traccia dei blocchi modificati (bit sporchi). Più **complesso** da implementare rispetto al Write-Through. La sostituzione della riga della cache è più costosa.

Esempio 9.6.1. Supponiamo che una cache abbia un blocco di 4 parole. Quanti accessi alla memoria principale sono necessari in base alle due politiche di accesso per il seguente codice?

```
MOV R5, #0
STR R1, [R5]
STR R2, [R5, #12]
STR R3, [R5, #8]
STR R4, [R5, #4]
```

Tutte e 4 le istruzioni scrivono sullo stesso blocco di cache. Con la tecnica **write-through**, ogni istruzione scrive nella memoria principale e richiede quindi 4 scritture. Con la tecnica write-back serve solamente un accesso alla memoria principale per pulire il blocco.

Il **write-back** può migliorare le prestazioni specialmente quando la CPU genera istruzioni di memorizzazione più velocemente di quanto la memoria principale può gestire, tuttavia, il costo delle scritture nella cache è maggiore se si verifica un errore di scrittura, dobbiamo prima riscrivere il blocco in memoria (se il dirty bit è 1). Ciò richiede almeno due cicli anche per un hit di scrittura: un ciclo per verificare un hit seguito da un ciclo per eseguire effettivamente la scrittura. In alternativa, possiamo usare un buffer di scrittura per trattenere temporaneamente i dati da scrivere mentre il blocco della cache è controllato per un colpo. Il processore esegue la ricerca nella cache e inserisce i dati nel buffer di scrittura durante il normale ciclo di accesso alla cache. Supponendo un riscontro nella cache, i nuovi dati vengono scritti dal buffer di scrittura nella cache al successivo ciclo di accesso alla cache inutilizzato (pipelining degli accessi).

9.6.3 Ottimizzare le scritture

Poiché la scrittura nella memoria off-chip è costosa, gli archivi di memoria sono bufferizzati, un **buffer di scrittura** viene utilizzato per conservare i dati in attesa di essere scritti nella memoria. L'esecuzione continua immediatamente dopo la scrittura dei dati nella cache e nel buffer di scrittura, i salvataggi nella memoria principale vengono eseguiti in parallelo con il calcolo della CPU. La CPU va in stallo solo se il buffer di scrittura è pieno, quindi la larghezza di banda richiesta dalla memoria principale è un fattore critico, in particolare per il modello di cache Writhe-Through.

9.6.4 Cache replacement policy

Per decidere quali blocchi rimpiazzare dobbiamo distinguere per i tipi di cache:

- **Direct mapped:** se il blocco da rimpiazzare è stato modificato e usiamo una policy di **write-back** dobbiamo aggiornare la memoria di livello più basso
- **Associative cache:**
 - *Fully associative*: tutti i blocchi sono candidati per essere sostituiti
 - *N-way set-associative*: dobbiamo scegliere tra gli N blocchi nel set selezionato

Lo schema più utilizzato è **Least Recent Used (LRU)**. Sfrutta la località temporale: il blocco sostituito è quello che è rimasto inutilizzato per il tempo più lungo. Per una cache set-associativa a 2 vie, può essere implementato con 1 bit (usare bit -U), per un 4 vie è ancora fattibile con 2 bit, per più di 4 vie diventa abbastanza complicato (pseudo-LRU). Per le cache altamente associative, una politica **Random** offre all'incirca le stesse prestazioni di LRU.

Esempio 9.6.2 (Direct mapped). Considera una piccola cache con 4 blocchi, $b = 1$. Trova il numero di miss per una cache **direct-mapped** per la seguente sequenza ordinata di indirizzi di blocco: 0, 8, 0, 6, 8.

Block address	Cache block
0	$0 \% 4 = 0$
6	$6 \% 4 = 2$
8	$8 \% 4 = 0$

Address	Hit or miss	Contents after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Esempio 9.6.3 (2-way set-associative). Considera una piccola cache con 4 blocchi, $b = 1$. Trova il numero di miss per una cache 2-way **set-associative** per la seguente sequenza ordinata di indirizzi di blocco: 0, 8, 0, 6, 8 assumendo la **LRU** replacement policy.

Block address	Cache block
0	$0\%2 = 0$
6	$6\%2 = 2$
8	$8\%2 = 0$

Address	Hit or miss	Contents after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Esempio 9.6.4 (Fully-associative). Considera una piccola cache con 4 blocchi, $b = 1$. Trova il numero di miss per una cache **fully-associative** per la seguente sequenza ordinata di indirizzi di blocco: 0, 8, 0, 6, 8.

Address	Hit or miss	Contents after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

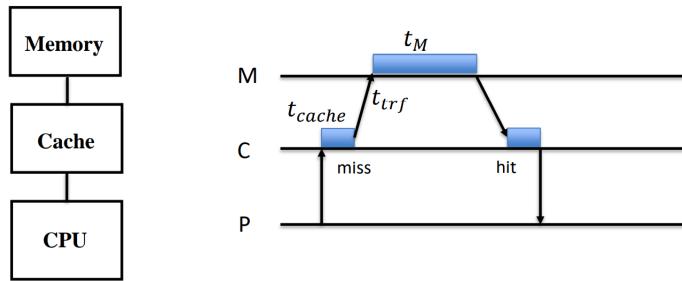
9.7 Designing the Memory System

La miss penalty può essere ridotta aumentando la larghezza di banda del bus tra DRAM e cache. Ci sono tre possibili organizzazioni:

- **Semplice**: una parola alla volta viene letta dalla memoria.
- **Ampia memoria**: N parole alla volta vengono lette dalla memoria.
- **Interleaved**: K banchi di memoria indipendenti in grado di servire K richieste contemporaneamente.

Consideriamo il tempo di trasferimento del blocco della cache per l'organizzazione della memoria Simple contro quella Interleaved.

Il caso **semplice** è il seguente:



Sapendo che il memory time e il cache time sono rispettivamente t_M e t_{cache} , la larghezza di banda della memoria sarà $B_M = \frac{1}{t_M}$ e $b = 1$. Il memory access time è

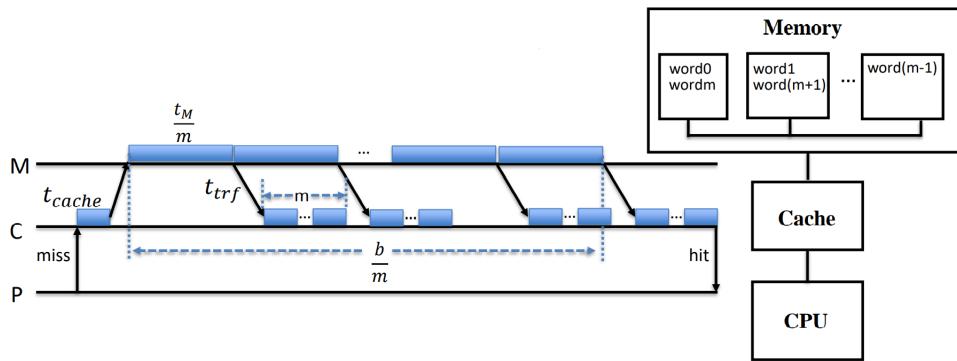
$$t_a = 2 \cdot t_{trf} + 2 \cdot t_{cache} + t_M$$

Se la cache line contiene più di un blocco, il suo costo di trasferimento in caso di cache miss sarà

$$t_{miss} = 2 \cdot t_{trf} + 2 \cdot t_{cache} + b \cdot t_M \approx b \cdot t_M$$

Esempio 9.7.1. Se consideriamo $b = 8$, $t_M = 80$, $t_{cache} = 1$ e $t_{trf} = 6$ cicli di clock, allora il costo del cache miss sarà molto alto, circa > 650 cicli di clock.

Invece per quanto riguarda il caso **interleaved**:



aumentiamo la larghezza di banda di un fattore del numero di moduli m

$$B_M = \frac{m}{t_M}$$

avendo quindi un costo di

$$t_{miss} = 2 \cdot t_{trf} + (\sim 1 + 1) \cdot t_{cache} + \frac{b}{m} \cdot t_M \cdot t_M$$

che, se poniamo $b = m$, diventa $t_{miss} \approx t_M$.

9.8 Problemi cache

La memorizzazione nella cache è essenziale per ridurre il collo di bottiglia di von Neuman e ottenere prestazioni ragionevoli sui sistemi moderni. Tuttavia, la memorizzazione nella cache introduce alcuni problemi nei sistemi multiprocessore/multicore.

Problema di **coerenza** della cache e **falsa condivisione** (due variabili non correlate sono collocate nello stesso blocco della cache e accesso in modalità lettura/scrittura da thread diversi).

9.8.1 Problemi di corerenza

Supponiamo un'architettura SMP (Symmetric Multiprocessors), ovvero con caching di dati privati e condivisi. I dati core privati vengono memorizzati nella cache in L1, riducendo così l'AMAT e le comunicazioni di memoria off-chip. Quando i dati condivisi vengono memorizzati nella cache, i valori condivisi possono essere replicati in più core cache private, riducendo anche la contesa di memoria. Cosa succede se i dati condivisi vengono scritti?

Viene introdotto un nuovo livello dell'architettura che forza a mantenere la consistenza dei dati per linee diverse di cache. Uno dei protocolli più semplici è il **write invalidate**, che consiste nell'invalidare tutte le altre linee di cache non aggiornate tramite un bus di tipo *snoopy*. Esistono anche protocolli di aggiornamento ma sono più complicati.

9.9 Software

I *cache misses* dipendono anche dal comportamento dell'algoritmo e dall'ottimizzazione che viene (o meno) fatta dal programmatore. Ad esempio per quanto il *Radix Sort* sia avvantaggiato sul *Quick sort*, fa molti più miss.

9.9.1 Ottimizzazione

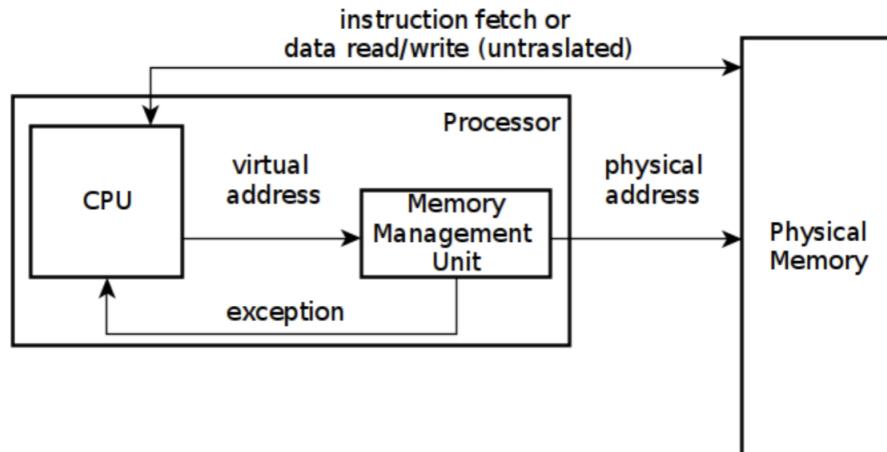
L'obiettivo è di massimizzare il riuso dei dati nella cache sfruttando la località spaziale e temporale. Esistono principalmente due tecniche:

- **Loop interchange**: usare esternamente i loop più grandi in modo da ridurre gli accessi. Migliora la località spaziale.
- **Data blocking**: si concentra sulla località temporale. Ad esempio nella moltiplicazione da matrice si lavora per blocchi invece che per colonne.

10 Address translation

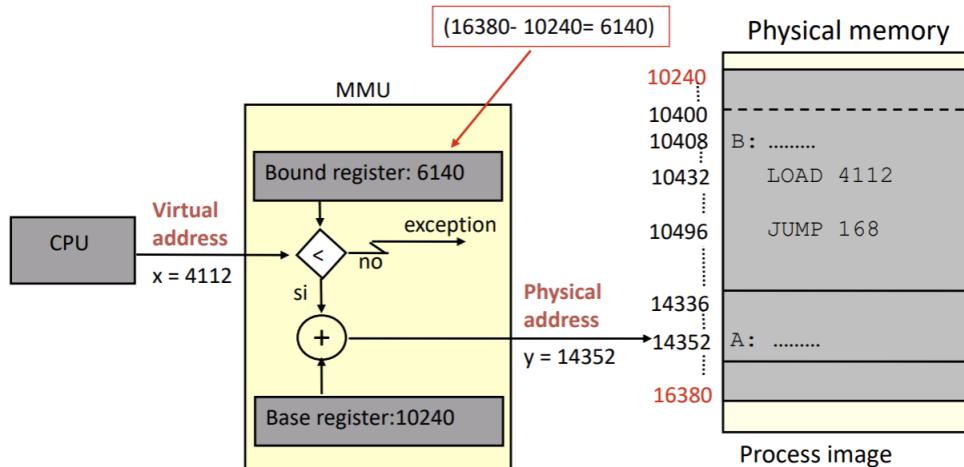
Gli obiettivi sono:

- **Protezione** della memoria
- **Condivisione** della memoria
- Piazzamento in memoria **flessibile**
- Indirizzi **sparsi**
- **Ricerca efficiente** a runtime
- Tabelle di traduzione **compatte**
- **Portabilità**



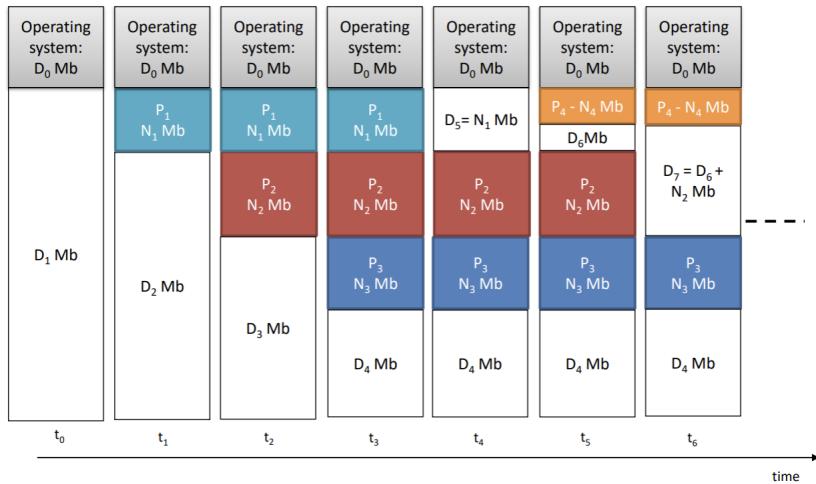
10.1 Virtual base and Bounds

Ad ogni processo si assegna un **base register**, ovvero un limite inferiore alla zona di memoria, e un **bound register**, ovvero un limite superiore. La **Memory Management Unit** si occupa di verificare che l'indirizzo richiesto sia nel range che è stato assegnato al programma in esecuzione ed effettua poi la traduzione dell'indirizzo in quello effettivo della memoria fisica.



10.1.1 Variable partitions

Se la gestione avviene con le partizioni di memoria, questa può diventare **frammentata**. Bisogna quindi decidere dove allocare ogni volta una nuova partizione.



10.1.2 Frammentazione

La frammentazione della memoria può presentarsi in due modi:

- **Interna:** la memoria viene allocata ad una partizione ma non viene usata dal processo. Avviene con le partizioni di dimensione **fissa**.
- **Esterna:** le partizioni libere di memoria sono troppo piccole per essere usate per altre allocazioni anche se la memoria totale libera sarebbe sufficiente. Si presenta con le partizioni **variabili**.

Definizione 10.1.1 (Deframmentazione). *La deframmentazione della memoria è il processo di spostare le partizioni allocate l'una vicina all'altra in modo da avere lo spazio libero tutto contiguo.*

10.1.3 Allocazione

Esistono due tecniche di ottimizzazione per allocare una nuova partizione:

- **First-fit:** tra le partizioni libere, seleziona la *prima* grande a sufficienza
- **Best-fit:** tra le partizioni libere, seleziona la *più piccola* che è grande a sufficienza

10.1.4 Conclusione

È un metodo **semplice** e veloce che però non ha *coarse-grain protection*, non permette la condivisione tra processi e non può far crescere a richiesta *stack* e *heap*.

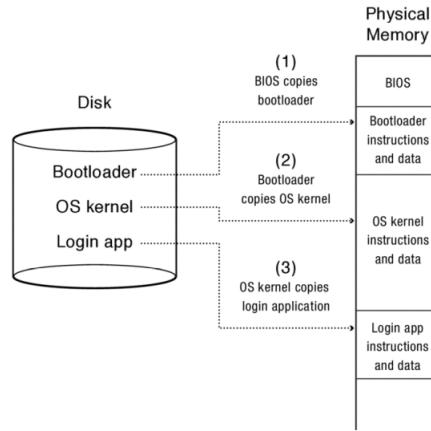
10.2 Segmentazione

Un **segmento**

11 Kernel

11.1 Booting

Il boot è un aspetto critico messo a disposizione dalla microarchitettura: all'avvio il processore esegue il codice nella ROM per caricare il **first-stage bootloader**. Questo inizializza il controller della memoria e alcune periferiche di I/O per permettere al **second-stage bootloader** di essere caricato. Quest'ultimo carica il kernel e il filesystem, a cui poi è ceduto il controllo.



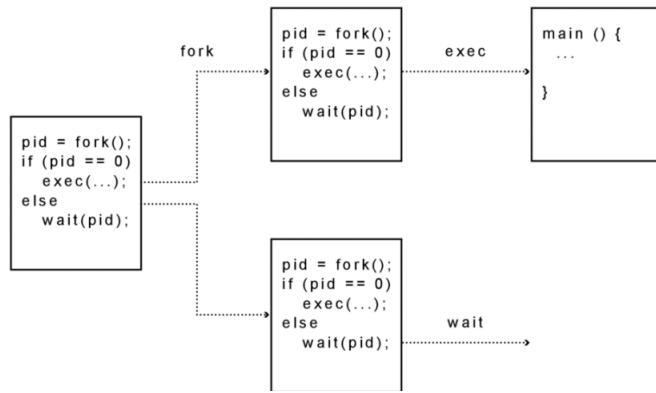
Note 11.1.0.1. Il BIOS risiede in una memoria volatile e permette di fare dei controlli basilari di sistema e caricare il bootloader. La versione moderna è lo UEFI. Si occupa anche di fare controlli di **sicurezza** per assicurarsi che il kernel caricato non sia modificato.

11.2 Processi

Un processo è una struttura dati (PCB) su cui vengono memorizzate tutte le informazioni riguardanti memoria, file aperti, sottoprocessi.

Note 11.2.0.1. Noi faremo riferimento sempre ai sistemi UNIX che sfrutta diverse semplici chiamate. Sotto Windows la chiamata per creare i processi è molto più articolata.

I comandi principali su UNIX sono: **fork**, **exec**, **wait** e **signal**.



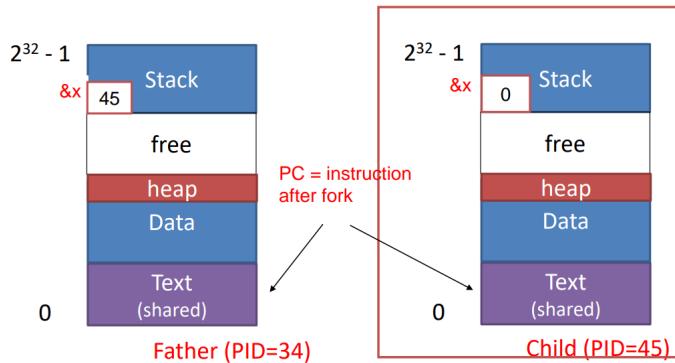
11.2.1 Fork

La fork crea una copia del processo corrente (*figlio*) che condivide il codice e lo stack del *padre*. Al processo padre restituisce l'ID del PCB del figlio (o un eventuale codice di errore), mentre al figlio restituisce 0.

Note 11.2.1.1. Come detto in precedenza la copia del PCB non è totale tra padre e figlio, ma fatta in modo che ci siano solo le parti utilizzate, dato che è possibile che poco dopo venga sovrascritta da una `exec`.

La *fork* può **fallire** in diversi casi, ad esempio quando non c'è sufficiente memoria.
 Quali sono i passaggi della fork?

1. Creata e inizializzato il PCB nel kernel
2. Creato un address space
3. Viene inizializzato l'address space con una copia di tutti i contenuti dello spazio di indirizzo del padre
4. Vengono copiati i parametri nella memoria dell'address space
5. Viene ereditata dal padre il contesto di esecuzione (e.g. tutti i file aperti)
6. Viene informato lo scheduler che c'è un nuovo processo attivo



Esempio 11.2.1. Un esercizio da compitino:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(){
    fork();
    printf("Pid %d\n", getpid());
    fork();
    printf("Pid %d\n", getpid());

    return 0;
}
```

Quanti ne stampa? 4

11.2.2 Exec

Questa chiamata sostituisce il codice eseguito da un processo, rimpiazzandone tutti i dati.

```
int execl(char *pathname, char *arg0, ..., char *argN, (char*)0)
```

La *exec* può **fallire** (e.g. per un parametro sbagliato) ma non ritorna, dato che non ci sarebbe un codice a cui può ritornare.

Se invece ritorna con **successo**, il processo su cui viene eseguita mantiene il PID e il PCB, mentre resetta i *pending signals*. Mantiene anche il *kernel stack* e le risorse (e.g. file aperti).

11.2.3 Terminazione

Un processo può terminare perché ha ricevuto un'eccezione o chiamando *exit*.

Un processo terminato restituisce il valore di ritorno al padre, che lo riceve tramite una *wait*. Se

quest'ultima non è ancora stata chiamata il figlio diventa **zombie**, rimanendo in attesa della chiamata o venendo eliminato dopo la morte del padre.

```
void exit(int status);
int wait(int *status);
```

La *wait* può ritornare immediatamente ad esempio se il processo da aspettare è già terminato.

Note 11.2.3.1. Mai fare assunzioni sulla durata dei processi. Evitare assolutamente di usare la *sleep* per eseguire la sincronizzazione.

11.2.4 Shell

Un aspetto importante dei processi è la **shell**, un programma di controllo che si occupa di eseguire comandi o altri processi e di riportare sullo standard output eventuali risultati o errori.

Questa sfrutterà principalmente chiamate di sistema come *fork*, *exec*, *waitpid*, etc...

Un'implementazione di una shell può essere la seguente:

```
char *prog, **args;
int child_pid;
// Leggo ed elaboro l'input una riga alla volta
while (readAndParseCmdLine(&prog, &args)) {
    child_pid = fork(); // Creo un processo figlio
    if (child_pid == 0) {
        exec(prog, args); // Sono il processo figlio, eseguo il programma
        // Non raggiunto
    } else {
        wait(child_pid); // Sono il processo padre, aspetto il figlio
        // Controllo lo stato di uscita
    }
}
```

Note 11.2.4.1. Se con il codice di esempio proviamo ad eseguire il comando *cd*, restituirà errore. Questo perché è implementato direttamente dalla shell e non dal sistema operativo.

11.2.5 I/O Unix

Le chiamate di sistema in Unix che si occupano dell'I/O sono **byte oriented** (non fortemente tipate). Il kernel inoltre fa della **bufferizzazione** (caricando un blocco invece che un byte), quindi *fopen*, *fread*, etc... sono più veloci di *open*, *read*, etc... mantenendo comunque l'overhead derivante dalle *system call*. La chiamata **open** ha diverse funzionalità e accetta molti parametri:

- Se il file non esiste, restituisce un errore o lo crea
- Se il file esiste, restituisce un errore o apre il file
- Se il file esiste ma non è vuoto, lo sovrascrive e poi lo apre oppure restituisce un errore

Viene tutto fatto da una sola funzione invece che da molteplici poiché è necessario fare questo tipo di operazioni in modo **atomico**.

Esempio 11.2.2. Nel seguente codice

```
if(!exists(name))
    create(name);
open(name);
```

Tra il controllo di esistenza e la creazione, il sistema operativo potrebbe portarsi avanti ed anticipare la creazione, causando problemi.

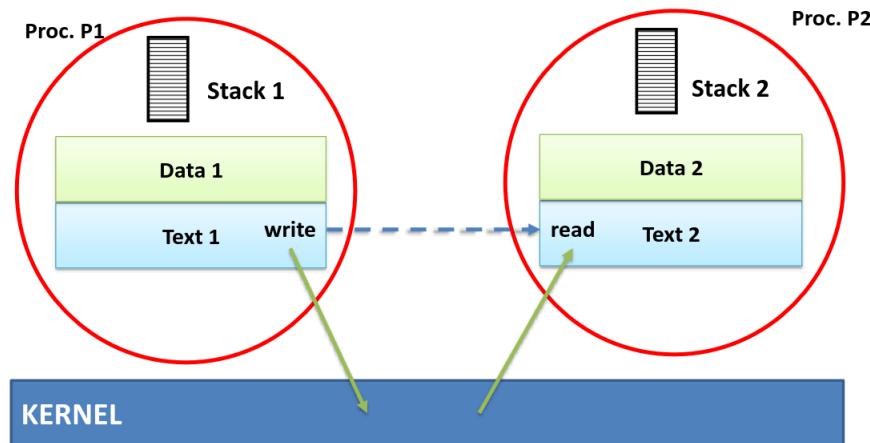
12 Concorrenza

Il sistema operativo deve gestire molte cose contemporaneamente (**MTAO** - Multiple Things at Once): processi, interrupts, manutenzione di background, etc... Non è solo il SO che deve gestire tutti questi aspetti, ma anche *server* (garantire più connessioni simultanee), *programmi* (migliori performance), *interfacce* (aumentare la responsiveness), *network* e *dischi* (ridurre la latenza).

Note 12.0.0.1. Il parallelismo è quando del codice può essere eseguito effettivamente nello stesso momento mentre la concorrenza è più virtuale: è lo scheduler che alterna i processi.

12.1 Inter-Process Communication

La concorrenza porta i processi ad avere un loro spazio di memoria riservato e si rende quindi necessario un **Inter-Process Communication** (IPC) per scambiare le informazioni e sincronizzarsi. C'è quindi un maggiore *overhead* di comunicazione.



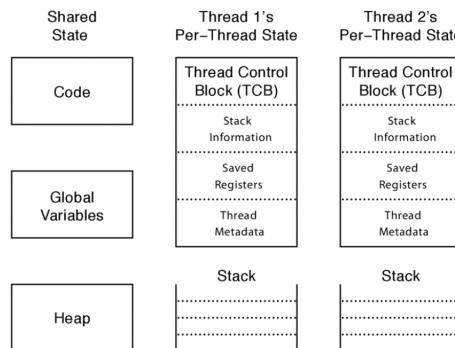
12.2 Thread

I thread sono singole entità che eseguono una **task indipendente** (può essere messa in pausa dal SO). Condividono la memoria del processo che li crea, poiché fanno parte dello stesso spazio di memoria, ed eseguono un pezzo di codice.

Il vantaggio principale è il fatto che non si deve più passare dal SO per comunicare e c'è quindi un vantaggio **prestazionale**. Inoltre migliora l'**organizzazione** della struttura codice.

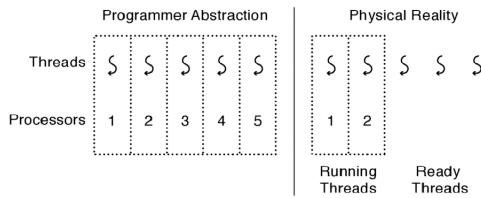
12.2.1 Protezione

Rimane che il thread può accedere solamente alla memoria del suo processo, garantendo così la protezione. C'è però da prestare attenzione ad evitare che un thread acceda ai dati di un altro thread nello stesso processo. In questo caso si SO non rileva errori. Eventuali tecniche di protezione sono implementate allo user-level dal RTS.



12.2.2 Astrazione

Ogni processo può creare virtualmente infiniti thread. Lo sviluppatore non sa in che ordine e con che velocità verrà poi eseguito ognuno di essi, dato che viene gestito dal kernel, e deve quindi progettare il software in modo che possa funzionare indipendentemente dalla schedule.



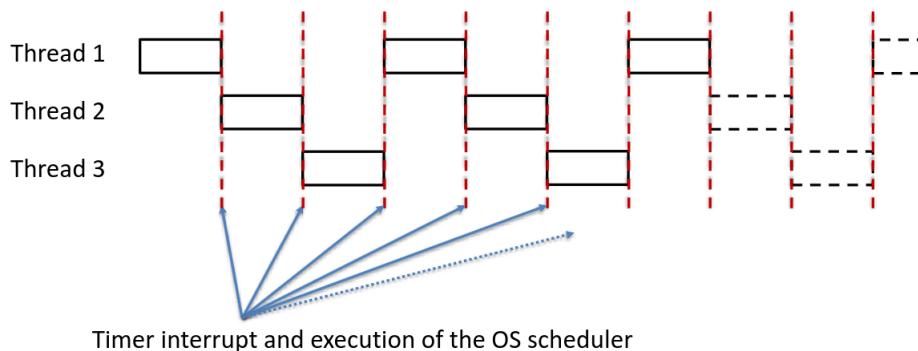
12.2.3 Implementazione

Alla base dei thread troviamo il **Thread Control Block** (TCB), ovvero una struttura dati che contiene le informazioni necessarie al thread., quali:

- Thread ID
- Stato
- Contesto del thread
- Parametri di scheduling
- Riferimenti allo stack

È importante definire le **operazioni** sui thread e uno **scheduler**, ovvero una funzione del sistema operativo che si occupi di assegnargli dei processori.

Lo scheduler viene mandato in esecuzione periodicamente dal SO tramite degli interrupt attivati dal **timer**:



Distinguiamo due tipi di thread:

- **Cooperative**: cooperano tra di loro in modo esplicito, ovvero con particolari istruzioni viene controllato lo scheduler. Finché non viene dato un comando o finché non scade il timer, rimane attivo quel thread
- **Preemptive**: partono e possono essere fermati dal SO in qualunque momento

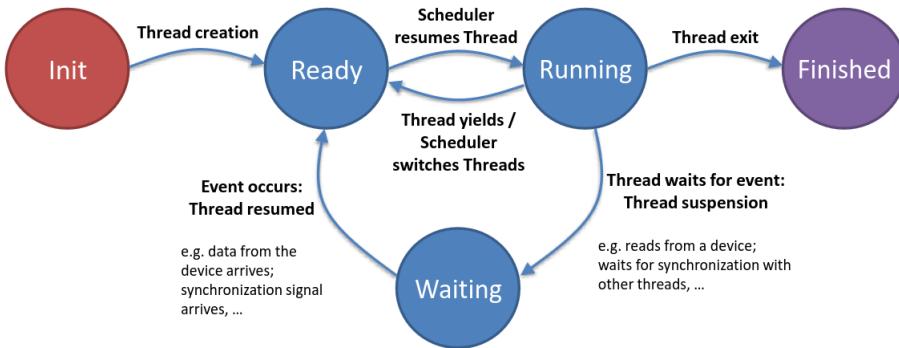
che possono essere implementati in modi diversi:

- Processi multipli **single-threaded**
- Processi multipli all'interno di un singolo processo, dove lo scheduler è gestito da una libreria (**user-level threads**)
- **Misto** tra single e multi-threaded (**kernel-level threads**)
- **Scheduler activation**

12.2.4 API

- **create(thread, func, args)**: crea un thread e lo salva in *thread*, poi esegue *func* con gli argomenti in *args*
- **yield()**: il thread smette volontariamente di usare il processore per dare spazio ad altri
- **join(thread)**: attende che il *thread* finisca e a quel punto restituisce lo stato di uscita
- **exit(status)**: termina il thread corrente con un determinato *status*

12.2.5 Ciclo di vita



Tutti questi stati sono implementati tramite **code** (liste). Ogni stato ha diverse code.

Note 12.2.5.1. Un altro possibile caso in cui il thread passa dallo stato *running* a quello *ready* è quando abbiamo uno scheduler con una politica di tipo *preemptive* e il SO crea un nuovo processo con priorità maggiore.

Nella seguente tabella è indicato dove si trovano i registri in ogni stato del ciclo di vita.

State of thread	Location of TCB	Location of registers
INIT	Being created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List, then Deleted	TCB

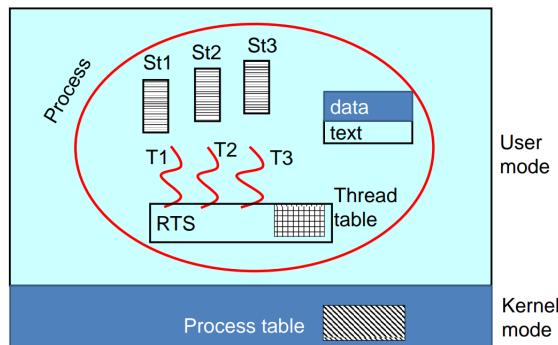
Riassumendo, l'unico caso in cui i registri non sono quelli del TCB, è quando il thread è in esecuzione, poiché in quel momento i registri validi sono nel processore.

12.2.6 User-level thread

Sono implementati a livello utente, ovvero il SO non è a conoscenza del fatto che un determinato processo sia al suo interno formato da più thread logici. Lo scheduling è quindi fatto da una libreria a livello utente: il **Run Time Support** (RTS).

Il modello utilizzato in questo schema è quello **cooperativo** (vedi 12.2.3). Potrebbe essere implementato anche con il modello **preemptive** utilizzando le *Scheduler Activation* di Windows o le *UpCall*. Le problematiche principali sono:

- La mancanza di conoscenza da parte del SO mi impedisce di sfruttare a pieno le risorse del sistema
- Quando invoco una **system-call bloccante**, viene bloccato l'intero processo, dato che il SO non sa cosa c'è dentro al processo



Per quanto riguarda l'implementazione, l'unica differenza è che la libreria fornisce una istruzione che permette di far partire e fermare il thread, oltre che di fare controlli e pulizia.

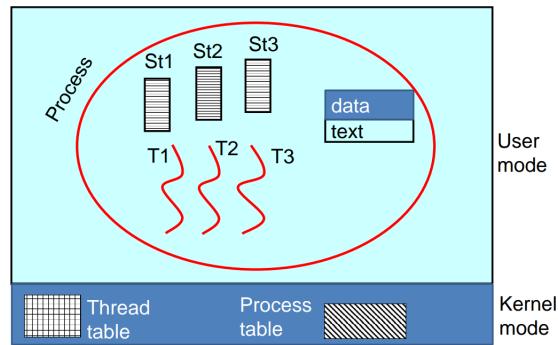
```
stub(func, args);
```

I principali lati positivi dei user-level thread sono:

- Creazione, terminazione e cambio di contesto (cambio di stato) più **efficienti** in quanto basta una chiamata alla libreria e non al SO
- Quando avviene un cambio di contesto l'addressing space rimane lo stesso
- Può essere implementato anche in SO che non supportano il multithreading

12.2.7 Kernel-level thread

In questo caso i thread vengono implementati dal kernel, che conterrà la relativa tabella e gestirà la creazione, terminazione e cambio di contesto. I lati positivi sono che i thread sono effettivamente parallelizzabili e in caso di system call bloccanti si blocca il singolo thread e non l'intero processo.



I principali lati negativi sono che ci sarà un costo maggiore per il cambio di contesto dato che sono necessarie delle system call.

Osservazione 12.2.1 (Asynchronous I/O). Un'alternativa ai thread implementati a livello kernel è la gestione di input e output tramite chiamate asincrone.

12.2.8 Switch

Il cambio di contesto di un thread può avvenire per due cause:

- **Volontariamente:**

- *User-level threads*: vengono salvati i registri sul TCB, viene fatto il cambio di stack e di thread, ripristinati i registri del nuovo thread e *return*
- *Kernel-level threads*: identico allo user-level tranne che invece della *return* c'è un'istruzione apposita

- A causa di un **interrupt** o un'**eccezione** (e.g. timer dello scheduler o interrupt I/O): può essere eseguito in due modi
 - *Semplice*: l'interrupt handler salva i registri nello stack, chiama la funzione per eseguire lo switch
 - *Veloce*: l'interrupt handler salva i registri sul TCB

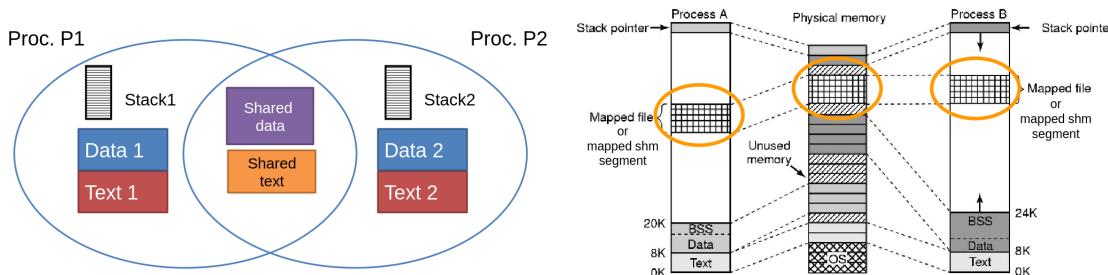
Il cambio di contesto può generare **overhead** a causa di:

- Salvataggio e ripristino dei registri
- Gestione dei TCB
- Memory cache invalidation
- Errori di memoria, quali:
 - Address exceptions
 - Page faults
 - MMU invalidations

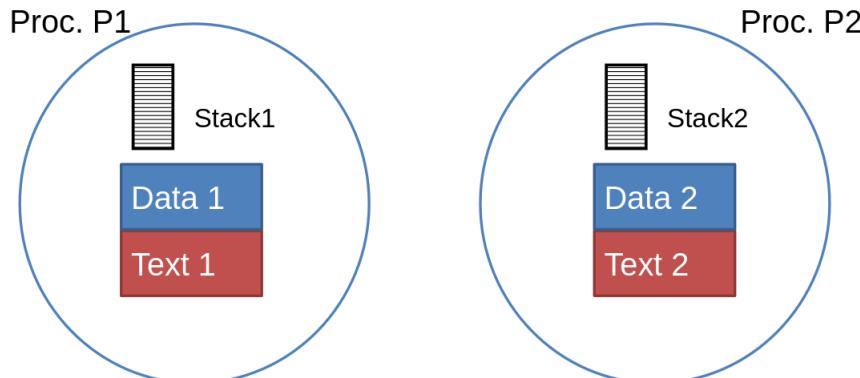
12.3 Cooperazione

Distinguiamo due modelli di cooperazione:

- **Global environment**: processi e thread possono condividere i dati tramite memoria condivisa



- **Local environment**: processi e thread non hanno memoria condivisa e condividono quindi i dati tramite messaggi



12.4 Sincronizzazione

Quando i thread lavorano contemporaneamente su memoria condivisa, il comportamento del programma non può essere determinato in anticipo in quanto la schedule è non deterministica.

Osservazione 12.4.1 (Reordering). I compilatori riordinano le istruzioni per migliorare l'efficienza del codice. Lo stesso vale per la CPU che cerca di fare **write buffering**. Per evitare problemi si possono dare istruzioni speciali al compilatore perché eviti di fare qualunque operazione prima che quella critica sia conclusa.

Definizione 12.4.1 (Race condition). *La corsa critica è quando il risultato di un programma concorrente dipende dall'ordine delle operazioni tra i thread, ammettendo che almeno una di queste sia una write.*

Definizione 12.4.2 (Mutual exclusion). *Si parla di mutua esclusione quando alcune operazioni in una critical section possono essere eseguite solamente da un thread alla volta.*

Definizione 12.4.3 (Lock). *È un meccanismo di sincronizzazione che permette di garantire la mutua esclusione.*

Definizione 12.4.4 (Correctness). *La proprietà di correttezza si compone di:*

- **Liveliness:** garantisce che prima o poi il programma completi il lavoro
- **Safety:** garantisce che il programma non entri mai in un bad state

12.4.1 Lock

Il meccanismo di **lock** prevede due operazioni:

- **acquire:** aspetta finché il lock non è libero e poi lo acquisisce
- **release:** libera il lock permettendo a chiunque fosse in attesa di acquisirlo

Sotto, la lock è un contatore booleano e una coda.

La **correttezza** è garantita in quanto al massimo c'è una persona che lo detiene alla volta (*safety*) e se nessuno lo detiene (o se qualcuno lo libera) è ottenuto da qualcuno (*liveliness*).

È fondamentale assicurarsi di aver acquisito il lock prima di accedere alla memoria condivisa e rilasciarlo appena finito.

Esempio 12.4.1 (Lock with malloc). Una possibile implementazione della **lock** con *malloc* e *free*.

```

char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}

void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}

```

Esempio 12.4.2. Un'altra possibile implementazione della **lock**.

```

tryget() {
    item = NULL;
    lock.acquire();
    if (nelem>0) {
        item = buf[front];
        front = (front++)%size;
    }
}

```

```

        nelem--;
    }
    lock.release();
    return item;
}

tryput(item) {
    r=false;
    lock.acquire();
    if (nelem < size) {
        buf[last] = item;
        last = (last++)%size;
        nelem++; r=true;
    }
    lock.release();
    return r;
}

```

Esempio 12.4.3 (Uniprocessor e multiprocessor). Implementazione della lock su sistema **single core**.

```

LockAcquire() {
    disableInterrupts ();
    if (value == BUSY) {
        waiting.add(myTCB);
        suspend(); // Sospende il thread corrente, ovvero invoca lo scheduler per fare il
                   content switch, seleziona un altro TCB e abilita gli interrupt
    } else {
        value = BUSY;
    }
    enableInterrupts ();
}

LockRelease() {
    disableInterrupts ();
    if (!waiting.Empty()){
        thTCB = waiting.Remove();
        readyList.Append(thTCB);
    } else {
        value = FREE;
    }
    enableInterrupts ();
}

```

Su un sistema **multi core**, disabilitare gli *interrupt* non è sufficiente. Si rendono quindi necessarie istruzioni **Read-Modify-Write** (RNW).

12.4.2 Condition variables

Permettono di effettuare la sincronizzazione senza l'attesa attiva, sfruttando i mutex e variabili di condizione. Prevede tre azioni **atomiche**:

- *wait*: rilascia automaticamente il lock e rinuncia al processore fino a nuovo segnale
- *signal*: sveglia un singolo thread in attesa, se ce ne sono
- *broadcast*: sveglia tutti i thread in attesa, se ce ne sono

Esempio 12.4.4 (Produttore e consumatore). Il problema prevede un **buffer condiviso** in cui uno o più produttori caricano messaggi. I consumatori estraggono uno alla volta i messaggi per consumarli. È necessario che un messaggio sia consumato una ed una sola volta.

```

get() {
    // Lock
    lock.acquire();
    // Se non ci sono elementi nel buffer, eseguo una wait aspettando che arrivi qualcosa
    while (nelem == 0)
        empty.wait(&lock);
    // Consumo il dato
    item = buf[front];
    front = (front++) % size;
    nelem--;
    // Avviso che ho consumato un elemento
    full.signal();
    // Unlock
    lock.release();
    return item;
}

put(item) {
    // Lock
    lock.acquire();
    // Se il buffer e' pieno, eseguo una wait aspettando che venga consumato qualcosa
    while (nelem == size)
        full.wait(&lock);
    // Producio il dato
    buf[last] = item;
    last = (last++) % size;
    nelem++;
    // Avviso che e' stato prodotto un elemento
    empty.signal();
    // Unlock
    lock.release();
}

```

In questo codice abbiamo tre variabili di condizione: *lock*, *empty*, *full*. È fondamentale seguire il pattern descritto nell'esempio.

Quando un thread viene svegliato da una *wait*, potrebbe non partire subito in quanto qualche altro thread potrebbe acquisire prima il lock. È quindi necessario, soprattutto quando il risveglio avviene tramite *broadcast*, usare un ciclo che rimanga in attesa del risveglio:

```

while (needToWait())
    condition.Wait(lock);

```

Dalla documentazione ufficiale di Java:

When waiting upon a Condition, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.

12.4.3 Semantica Hoare

La semantica Hoare prevede che la *signal* svegli un thread e gli passi la lock, senza lasciarla. Nasce con l'idea di usare un concetto **FIFO** per evitare che i thread si scavalchino a vicenda.

Il problema principale è che non c'è più l'esclusività di essere nella sezione critica e ci potrebbero essere cambiamenti di stato non previsti.

12.4.4 Semantica Mesa

La semantica Mesa prevede che la *signal* mette chi era in attesa in una lista **ready** e questo poi deve prendere la lock (che può anche essergli "rubata" da qualcun'altro). È più facile da implementare della

semantica *Hoare* e c'è un meccanismo per implementare la politica **FIFO**. Una possibile implementazione è la seguente:

```

get() {
    lock.acquire();
    if (!nextGet.empty() || nelem == 0) {
        self = createCondition();
        nextGet.Append(self);
        do self.wait(lock);
        while (nelem == 0);
        nextGet.Remove(self);
        destroyCondition(self);
    }
    item = buf[front];
    front = (front++) % size;
    nelem--;
    if (!nextPut.empty())
        nextPut.first()->signal();
    lock.release();
    return item;
}

```

12.4.5 Banker's Algorithm

L'algoritmo del banchiere prevede che siano definite in anticipo le massime risorse necessarie e che queste vengano allocate strada facendo, controllando però prima che non causino **deadlock**.

Definizione 12.4.5 (Safe state). *Per ogni possibile sequenza di richieste future è possibile garantire tutte le eventuali richieste. Potrebbe implicare dell'attesa anche quando sono disponibili tutte le risorse necessarie.*

Definizione 12.4.6 (Unsafe state). *Una sequenza di richieste di risorse che potrebbe portare ad un deadlock.*

Definizione 12.4.7 (Doomed state). *Tutte le possibili richieste portano al deadlock.*

L'algoritmo del banchiere fornisce la risorsa richiesta solo se può essere garantita in un *safe state*.

Note 12.4.5.1. La somma delle risorse necessarie ai thread correnti può essere maggiore delle risorse disponibili totali, purché ci sia un modo per tutti i thread di non finire in deadlock.

I passi effettuati dall'algoritmo sono:

1. Ogni processo dichiara di quante risorse ha bisogno
2. Ad una richiesta di un determinato processo P il banchiere verifica se la concessione della risorsa mantenga un *safe state*. Per farlo:
 - Considera lo stato S che si otterrebbe concedendo la risorsa
 - Per ogni processo calcola le risorse ancora necessarie R
 - Esegue un ordinamento dei processi in base a R
 - Esegue l'algoritmo
 - Se tutti gli algoritmi sono marcati, procede
3. Se lo stato S è *safe* allora la richiesta è concessa, altrimenti P deve aspettare che ci siano sufficienti risorse

```
// Initially each process Pj is not marked
while (exists non marked processes) {
    if (exists a non-marked Pj that satisfies Ej<=D) {
        mark Pj;
        D = D + Aj ;
    } else ends while, the state is not safe;
}
success: the initial state is safe
```

Esempio 12.4.5. Un sistema con 4 processi P_1, P_2, P_3 e P_4 ha risorse del tipo R_1, R_2, R_3 e R_4 rispettivamente con molteplicità [4, 5, 5, 5]. Le esigenze dei processi:

	R_1	R_2	R_3	R_4
P_1	2	3	1	1
P_2	2	1	1	2
P_3	0	1	0	2
P_4	0	2	5	2

Inizialmente l'assegnazione delle risorse e le loro esigenze rimanenti sono le seguenti:

	R_1	R_2	R_3	R_4
P_1	2	1	1	1
P_2	2	0	1	2
P_3	0	1	0	0
P_4	0	2	2	2

	R_1	R_2	R_3	R_4
P_1	0	2	0	0
P_2	0	1	0	0
P_3	0	0	0	2
P_4	0	0	3	0

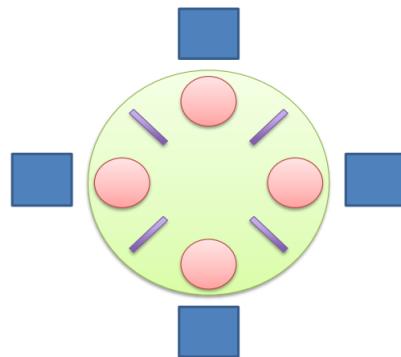
Procedo poi simulando di assegnare ai vari processi le risorse di cui hanno bisogno in base a quelle disponibili e verifico di rimanere in un *safe state*. Ad esempio assegnando R_2 a P_2 :

	R_1	R_2	R_3	R_4
P_1	2	1	1	1
P_2	-	-	-	-
P_3	0	1	0	0
P_4	0	2	2	2

	R_1	R_2	R_3	R_4
P_1	0	2	0	0
P_2	-	-	-	-
P_3	0	0	0	2
P_4	0	0	3	0

12.4.6 Esempi

Esempio 12.4.6 (Filosofi a cena). N filosofi si ritrovano a cena in un ristorante cinese occupando un tavolo circolare, apparecchiando con un piatto per ogni filosofo e un bastoncino posto fra ogni coppia di piatti adiacenti.



Per mangiare, ogni filosofo deve avere a disposizione i due bastoncini che si trovano rispettivamente alla sua sinistra e alla sua destra, entrando così in competizione con i due filosofi ai lati. Il filosofo che non riesce ad ottenere entrambi i bastoncini, rimane in attesa.

Ogni filosofo alterna periodi in cui medita a periodi in cui vuole mangiare: quando decide di mangiare richiede i bastoncini ed eventualmente attende. Dopo aver mangiato torna a pensare rilasciando entrambi i bastoncini.

Il problema può essere risolto associando una variabile di **lock** ad ogni bastoncino e organizzando queste variabili in un vettore.

```
while (true) {
    penso(); // Il filosofo pensa
    // Il filosofo di indice i decide di mangiare
    lockBastoncino[i].Acquire(); // Acquisisce il bastoncino di destra
    // Il filosofo si sospende se non puo acquisire il bastoncino alla sua sinistra
    lockBastoncino[(i+ 1) mod N].Acquire();
    mangia(); // Il filosofo di indice i mangia
    // Rilascia i bastoncini
    lockBastoncino[(i+ 1) mod N].Release();
    lockBastoncino[i].Release();
}
```

Chiaramente questo primo algoritmo porta allo **stallo** poiché ogni filosofo prova a prendere il bastoncino alla sua destra e rimane quindi poi in attesa per quella di sinistra, che non sarà mai libera.

Per evitare lo stallo abbiamo tre possibili soluzioni:

- Eseguiamo un ordinamento ragionato per l'acquisizione delle risorse, in questo caso asimmetrico. I filosofi di indice pari prendono prima la bacchetta di indice $i + 1$ e quelli di indice dispari il contrario.

```
while (true) {
    penso();
    if (i % 2) { // filosofo con indice dispari
        // Prima acquisisce la bacchetta di sinistra e poi quella di destra
        lockBastoncino[i].Acquire(); lockBastoncino[(i+ 1) mod N].Acquire();
        mangia(); // Il filosofo di indice i mangia
        lockBastoncino[(i+ 1) mod N].Release(); lockBastoncino[i].Release();
    } else { // filosofo con indice pari
        // Prima acquisisce la bacchetta di destra e poi quella di sinistra
        lockBastoncino[(i+ 1) mod N].Acquire(); lockBastoncino[i].Acquire();
        mangia(); // Il filosofo di indice i mangia
        lockBastoncino[i].Release(); lockBastoncino[(i+ 1) mod N].Release();
    }
}
```

Questo codice è *asimmetrico*, cosa che potrebbe causarci problemi. Ad esempio con 4 filosofi solo uno riuscirebbe a mangiare.

- Un processo prende le risorse di cui ha bisogno solo se riesce a prenderle tutte assieme, altrimenti rimane in attesa che un altro gliele passi.

```
while (true) {
    penso(); // Il filosofo pensa
    // Il filosofo di indice i decide di mangiare
    PrendiBastoncini(&lockBastoncino[i], &lockBastoncino[((i+ 1) mod N];
    mangia(); // Il filosofo di indice i mangia
    RilasciaBastoncini(&lockBastoncino[i], &lockBastoncino[((i+ 1) mod N);
}
PrendiBasoncini(lock1, lock2) {
    while(true) {
        lock1.Acquire();
```

```

        if (lock2.tryAcquire()) return; // Successo
        lock1.Release(); // Rilascio il bastoncino
        swap(lock1, lock2); // Provo nell'ordine contrario
    }
}

RilasciaBastoncini(lock1, lock2) {
    lock1.Release();
    lock2.Release();
}

```

Questa è una soluzione che evita il *deadlock* ma potrebbe causare **starvation**, che è comunque meglio. Per migliorarla si potrebbe implementare dell'ordinamento casuale per aumentare l'entropia.

- Si tiene conto dello stato dei processi concorrenti e si agisce di conseguenza. Lo stato è rappresentato da un vettore che per ogni singolo filosofo tiene conto se *HaFame*, *Mangia* o *Pensa*. Per gestire la concorrenza viene usato un *mutex*.

```

while (true) {
    penso(); // Il filosofo pensa
    // Il filosofo di indice i decide di mangiare
    PrendiBastoncini(i);
    mangia(); // Il filosofo di indice i mangia
    RilasciaBastoncini(i);
}

prendiBastoncini(i) { // Metodo del monitor
    // Il filosofo di indice i ha deciso di mangiare
    mutex.Acquire();
    stato[i] = HaFame;
    while (stato[(i - 1) mod N] == Mangia) || (stato[(i + 1) mod N] == Mangia) {
        attesaFilosofo[i].wait(&mutex); // Devo attendere che siano liberi entrambi
    }
    stato[i] = Mangia; // Ha ottenuto entrambi i bastoncini
    mutex.Release();
}

rilasciaBastoncini(i) { // Metodo del monitor
    mutex.Acquire();
    stato[i] = Pensa;
    if (stato[(i - 1) mod N] == HaFame) && (stato[(i - 2) mod N] != Mangia) {
        // Riattiva il filosofo (i-1) mod N se puo ottenere entrambi i bastoncini
        stato[(i - 1) mod N] = Mangia;
        attesaFilosofo[(i - 1) mod N].signal();
    }
    if (stato[(i + 1) mod N] == HaFame) && (stato[(i + 2) mod N] != Mangia) {
        // Riattiva il filosofo (i+1) mod N se puo ottenere entrambi i bastoncini
        stato[(i + 1) mod N] = Mangia;
        attesaFilosofo[(i + 1) mod N].signal();
    }
    mutex.Release();
}

```

Esempio 12.4.7 (Lettori e scrittori). La differenza principale dal problema descritto nell'esempio 12.4.4 è che nella sezione critica possano trovarci più thread alla volta. La struttura base dell'implementazione è la seguente:

```

// Lettore
while (true) {
    startRead();
    // Accede in lettura alla struttura condivisa
    doneRead();
}

```

```

    // Usa i dati letti
}

// Scrittore
while (true) {
    // Prepara dati da scrivere
    startWrite();
    // Accede in scrittura alla struttura condivisa
    doneWrite();
}

```

Vediamo alcune soluzioni:

1. Quando la struttura dati non è usata da nessun thread, il primo che ne fa richiesta accede. Se invece la richiesta avviene mentre è utilizzata da uno *scrittore* allora il thread in questione viene messo in attesa. Se è utilizzata da un *lettore*, un altro *lettore* può accedere ma uno *scrittore* viene messo in attesa (potrebbe anche finire in *starvation*).

Quando uno *scrittore* rilascia la struttura dati tutti i *lettori* ottengono l'accesso, se non ce ne sono allora il primo *scrittore* lo ottiene

Quando è il *lettore* a rilasciare la struttura dati, se non è rimasto nessun altro *lettore* allora il primo *scrittore* ottiene l'accesso.

```

// Lettore
startRead();
mutex.Acquire();
waitingReaders++;
while (activeWriters > 0) {
    readGo.Wait(&mutex);
}
waitingReaders--;
activeReaders++;
mutex.Release();
// Lettura
doneRead()
mutex.Acquire();
activeReaders--;
if (activeReaders == 0 &&
waitingWriters > 0) {
    writeGo.Signal();
}
mutex.Release()

// Scrittore
startWrite();
waitingWriters++;
while (activeWriters > 0 || activeReaders > 0) {
    writeGo.Wait(&mutex);
}
waitingWriters--;
activeWriters++;
mutex.Release();
// Scrive
doneWrite();
mutex.Acquire();
activeWriters--;
if(waitingReaders>0)
    readGo.Broadcast();
else
    writeGo.Signal();
mutex.Release();

```

2. Una variante della prima soluzione prevede che il *lettore* che entra nella sezione critica si sospende se ci sono *scrittori* in attesa. Lo scrittore quando rilascia la struttura dati sveglia un altro scrittore se disponibile altrimenti un lettore (rischio di **starvation** per i lettori).
3. La soluzione **fair** prevede che:
 - Se un lettore richiede l'accesso ma vi è almeno uno scrittore in attesa di accedere, il lettore richiedente viene sospeso, evitando l'attesa indefinita per gli scrittori
 - L'ultimo dei lettori che rilascia la struttura dati fa entrare l'eventuale scrittore in attesa
 - Quando uno scrittore rilascia la struttura dati, fa entrare il prossimo in attesa, indipendentemente che sia lettore o scrittore
 - Il lettore in attesa sbloccato dallo scrittore sblocca a sua volta l'ingresso all'eventuale altro lettore in attesa dopo di lui

```

// Lettore
startRead();
ordering.Acquire();
mutex.Acquire();
while (activeWriters > 0) {
    Go.Wait(&mutex);
}
activeReaders++;
mutex.Release();
ordering.Release();
// Lettura
doneRead()
mutex.Acquire();
activeReaders--;
if (activeReaders ==0) {
    Go.Signal();
}
mutex.Release()

// Scrittore
startWrite();
ordering.Acquire();
mutex.Acquire();
while (activeWriters > 0 || activeReaders > 0) {
    Go.Wait(&mutex);
}
activeWriters++;
mutex.Release();
ordering.Release();
// Scrive
doneWrite();
mutex.Acquire();
activeWriters--;
Go.Signal();
mutex.Release();

```
