SISTEMI OPERATIVI
Architetture e Sistemi Operativi - II Semestre

Corso A

Autore
Giuseppe Acocella
2024/25

Ultima Compilazione - February 24, 2025

Contents

1 Caching **4**

1.1 Gerarchia di Memorie 4
1.2 Hit/Miss Rate, Miss Penalty, AMAT 5
1.3 Principio di Località 6
1.4 Tempi CPU (CPI e Miss Penalties) 6
1.5 Tipi di Indirizzamento 7
1.5.1 Indirizzamento Diretto 7
1.5.2 Indirizzamento Associativo 9
1.5.3 Indirizzamento Set Associativo 10
1.6 Accenni di Gestione Stalli Miss Penalties 11
1.7 Tipologie di Miss 12
1.8 Scrittura in Cache 12
1.8.1 Problemi in Scrittura 12
1.8.2 Inconsistenza Cache/Memoria - Write Back/Write Through 12
1.9 Politiche di Rimpiazzamento - Sequenziale/LRU 13
1.10 Cache Coherence 14
1.10.1 Cache Coherence Protocols 14
1.10.2 False Sharing 14

2 Gestione I/O e Periferiche **15**

2.1 Legge di Amdahl 15
2.2 Prima Descrizione Gestione I/O 15
2.3 Protocolli di Arbitraggio del Bus 17
2.4 Meccanismi di Gestione Bus 18
2.4.1 Memory Mapped I/O 18
2.4.2 Interruzioni 18
2.5 Metodi di Query sullo Stato dell'I/O 19

3 Introduzione ai Sistemi Operativi **21**

3.1 Struttura Kernel - Monolitica vs a Micro Kernel 22
3.2 Computazioni nel Tempo - (Batch, Spool, Time Sharing) 22
3.3 Protezione del Calcolatore 23
3.3.1 Dual Mode - User/System 23
3.3.2 Meccanismi Necessari alla Gestione del Dual Mode 23
3.4 Descrizione di un Processo 25
3.4.1 Thread e Processi 25
3.4.2 PCB - Process Control Block 25
3.5 Descrizione Interruzioni 26
3.5.1 Handler di Interruzioni 26
3.6 Esempi d'Implementazione su Architettura ARM 27
3.6.1 CPSR ed Interrupt Masking 27
3.6.2 Modalità di Esecuzione 27
3.6.3 Banked Registers 28
3.7 System Calls 28

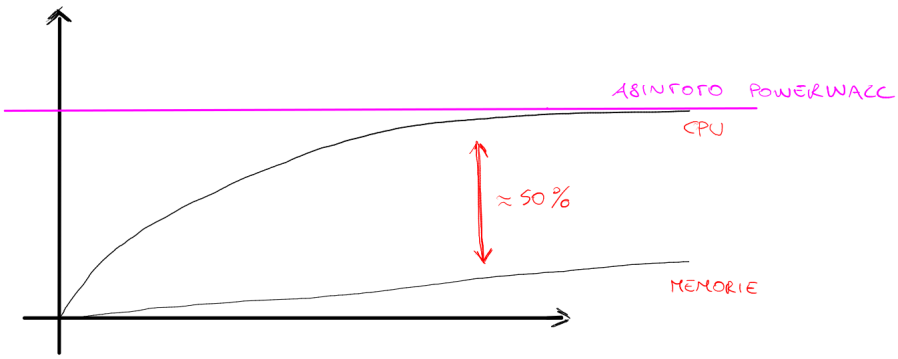
3.7.1 Call Handler in ASM ARM v7 29
3.8 Upcalls - Segnali 29
3.9 Kernel Boot 29
3.10 Sommario Specifiche Hardware Necessarie all'OS 30

1 Caching

Il Caching ha come obiettivo quello di astrarre sulle gerarchie di memoria e creare l'**illusione** di avere la stessa quantità di memoria "lenta" ma alla velocità della memoria "veloce". Questo è solo un esempio che cerca di illustrare l'obiettivo del caching. Si vuole creare una gerarchia di memorie, classificate per velocità e "vicinanza" alla CPU, permettendo l'accesso a dati ricorrenti in tempo breve.

1.1 Gerarchia di Memorie

Classifichiamo le memorie secondo la loro velocità. Più veloci saranno e più saranno costose e piccole in termini di spazio in *byte*. Oltre a questo dobbiamo tenere in considerazione il modo in cui si stanno evolvendo le CPU e le memorie:



Questo è un altro motivo per cui si vuole gestire in maniera più intelligente la comunicazione tra processore e memoria.

Istanza di Caching Assumiamo di avere quattro memorie:

$$A1,\ A2,\ A3,\ A4$$

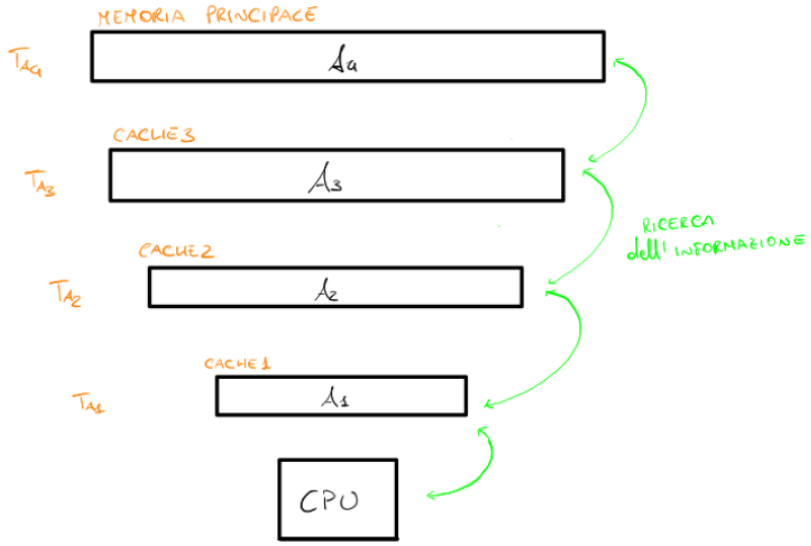
Ed ognuna di queste memorie sarà caratterizzata da un tempo, ordinate in tempo crescente:

$$T_{A1},\ T_{A2},\ T_{A3},\ T_{A4}$$

E' necessario tenere in considerazione come modello di riferimento quello di *Von Neumann*, di conseguenza abbiamo una **CPU** che interagisce con una **memoria** grazie ad un canale di comunicazione. Spesso l'utilizzo di questo canale causa un effetto di **bottleneck**. Si vuole quindi trovare un compromesso che possa velocizzare la comunicazione tra **CPU** e **memoria**. Le prestazioni di una memoria dipendono fortemente dalla tecnologia del supporto[1].

[1] La classificazione di tecnologia delle memorie è presente nel primo modulo di appunti: Architetture degli Elaboratori.

Illustriamo la gerarchia delle memorie e la loro "distanza" dalla CPU.



Possiamo assumere queste tempistiche delle memorie:

$$T_{A_1} = 4\tau \quad T_{A_2} = 10\tau$$

$$T_{A_3} = 20\tau \quad T_{A_4} = 60/80\, nsec$$

1.2 Hit/Miss Rate, Miss Penalty, AMAT

Una volta impostata la gerarchia rappresentata sopra, è necessario capire se un **indirizzo** dato sia o meno presente ad uno specifico **livello** tra quelli illustrati.

1. **Hit Rate**: Percentuale di "Hit", ossia dati **trovati**.
2. **Miss Rate**: Percentuale di "Miss", complementare a quella di "Hit", ossia dati **non trovati**.
3. **Miss Time**: Tempo impiegato per effettuare la fetch con esito di "Miss".
4. **Miss Penalty**: Tempo impiegato per risalire la gerarchia delle cache fino al ritrovamento del dato cercato.
5. **AMAT**: Average Memory Access Time:

$$AMAT = HIT\_TIME + (MISS\_RATE * MISS\_PENALTY)$$

La differenza tra **Miss Time** e **Miss Penalty** può variare anche di ordini di grandezza. L'**Hit Rate** deve essere superiore al 90% per rendere efficiente lo schema delle cache così impostate.

1.3 Principio di Località

Risulta necessario stabilire il criterio con il quale vengono caricati questi blocchi di memoria più performanti e vicini alla CPU. Le informazioni caricate sulla cache seguono il cosiddetto **principio di località**:

1. **Località Spaziale**: Solitamente per un certo periodo di tempo, molti dati potrebbero risultare vicini tra di loro. In questo caso si stabilisce quindi la grandezza di un blocco e questo verrà caricato tutto sulla cache. Questo permetterà probabili accessi successivi in tempi ridotti, anche di svariati ordini di grandezza. Secondo questo principio, accedendo a $M[PC]$, molto probabilmente accederò a $M[PC+4]$, $M[PC+8]$ ...

2. **Località Temporale**: Spesso un dato corrente potrebbe essere riutilizzato in breve tempo. Di conseguenza si preferisce mantenere il dato in cache. Secondo questo principio, accedendo a $M[PC]$, probabilmente in breve tempo potrei riaccedere a $M[PC]$.

Grazie a questo principio si stabilisce il criterio con cui vengono caricati i blocchi in cache. Questo procedimento permette in 1τ[2] di caricare in cache un blocco di grandezza arbitraria. E' necessario ricordare che questi principi possono essere applicati a tutti i tipi di dato (codice oppure effettivo dato in memoria).

1.4 Tempi CPU (CPI e Miss Penalties)

Elenchiamo diverse formule di calcolo dei tempi CPI (clock per instructions), illustrando anche l'influenza su questi tempi delle penalties causate dalle informazioni non trovate nella cache.

$$CPU_{TIME} = (CPI_{PERFECT} + CPI_{MISS\_PENALTY}) * LEN\_CICLO\_CLOCK$$

$$CPI_{PERFECT} = \frac{IC_{CPU}}{IC} CPI_{CPU} + \frac{IC_{MEM}}{IC} CPI_{MEM}$$

$$CPI_{MISS\_PENALTY} = \frac{IC_{MEM}}{IC} * MISS\_RATE * MISS\_PENALTY$$

$$CPI_{STALL} = CPI_{STALL\_IST} + CPI_{STALL\_DATA}$$

$$CPI = CPI_{PERFECT} + CPI_{STALL}$$

$$CPI_{STALL} = MISS\_RATE_{IST} * MISS\_PENALTY$$

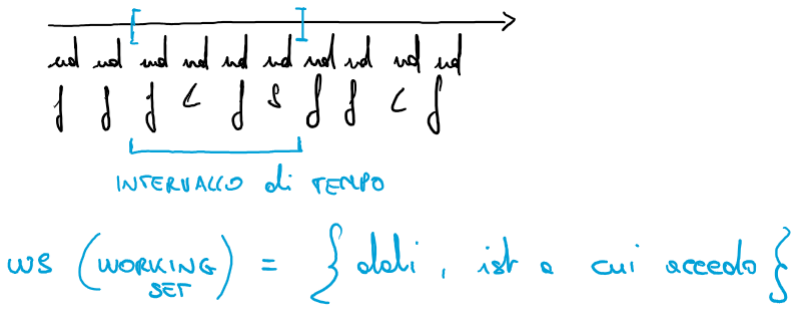
L'identificatore IC corrisponde al contatore di istruzioni. Un caching efficiente tende a minimizzare la percentuale di miss rate. Questa minimizzazione spesso è forzata da tecniche di **prefetching**, ossia si manda in background il caricamento della cache cercando di "prevedere" i dati che verranno utilizzati in caso siano validi i principi di località.

[2]Tempo breve generico, abbiamo assunto un ciclo di clock, per rendere l'idea che un blocco viene caricato tutto sulla cache in un singolo tempo, essendo appunto in "blocco".

1.5 Tipi di Indirizzamento

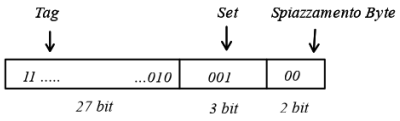
Abbiamo definito i criteri e l'organizzazione fisica del caching. Ma non abbiamo ancora definito come possiamo effettivamente trovare/non trovare un determinato indirizzo dato in cache. Se assumiamo che la cache sia un sottoinsieme della memoria principale, allora abbiamo la certezza (conoscendo le dimensioni di entrambe le memorie citate) che degli indirizzi possono collidere. Sicuramente almeno due indirizzi della memoria principare avranno la stessa locazione in cache. Di conseguenza elenchiamo tre tipi di indirizzamento caratterizzati da proprietà diverse.

Working Set Insieme di istruzioni/dati consecutivi in un certo intervallo di tempo. L'analisi della località di questi set permette la gestione logica delle cache.

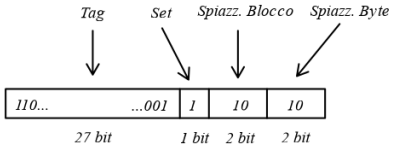


1.5.1 Indirizzamento Diretto

Immaginiamo di avere una memoria principale di dimensioni N, la cache a disposizione della CPU sarà invece di dimensioni N^I con $N >> N^I$. Alla base dell'indirizzamento diretto c'è l'interpretazione di parte dell'indirizzo cercato come **tag** da cercare nelle linee della cache.



Indirizzamento Diretto Singola Word

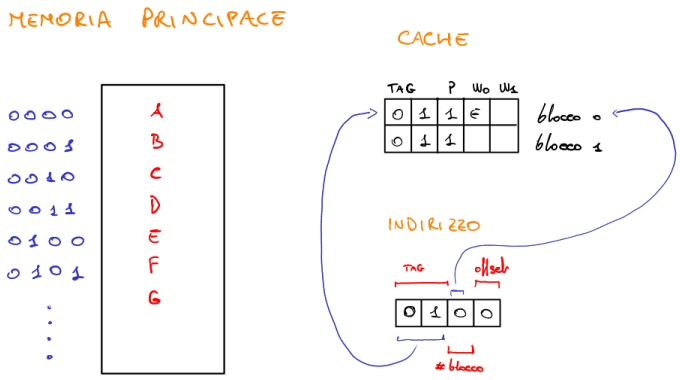


Indirizzamento Diretto con Blocco Words

Trashing A causa delle differenze di dimensioni tra memoria e cache, sicuramente più indirizzi avranno la stessa chiave, di conseguenza se dovesse capitare di dover caricare in cache più indirizzi con la stessa chiave si andrebbe in contro ad un continuo **caricamento** e **scaricamento** di dati tra la cache e la memoria. Questo fenomeno è detto **trashing** ed è un punto di debolezza delle cache ad indirizzamento diretto. Effettuare questi spostamenti continui di dati invaliderebbe infatti tutti i vantaggi ottenuti dall'utilizzo delle cache.

Mostriamo a pagina successiva la logica di questo tipo di indirizzamento e la sua implementazione fisica su rete combinatoria.

1. **Implementazione Logica**: Mostriamo uno schema della logica dietro l'indirizzamento diretto ed analizziamolo a step.

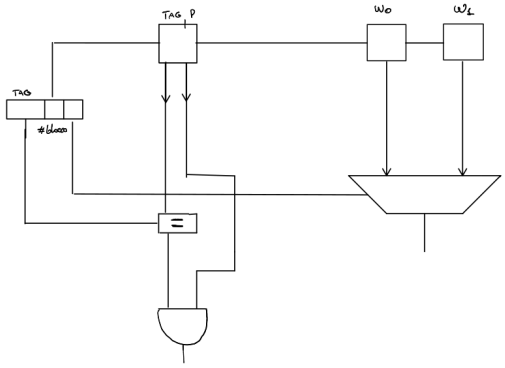


   (a) Ogni dato in memoria avrà un **indirizzo**. Assumiamo un indirizzo composto da 4 bit e diamo un interpretazione alle cifre.

   (b) Ogni indirizzo sarà dunque composto da un **tag** (bit 0-1), un **blocco** (bit 2) e da un **offset** (bit 3).

   (c) Nella cache saranno presenti vari blocchi (linee) e ciascuna di queste avrà un **tag**, un **bit di presenza** (p) e varie **word**, se saranno selezionabili grazie all'**offset** determinato dall'indirizzo cercato.

2. **Implementazione Fisica**: Mostriamo la rete combinatoria dietro l'indirizzamento diretto ed analizziamola a step.



   (a) Tre moduli rappresentanti la cache, il primo mantiene le informazioni riguardanti il **tag** ed il **bit di presenza**.

   (b) L'indirizzo in input è interpretato come nell'implementazione logica.

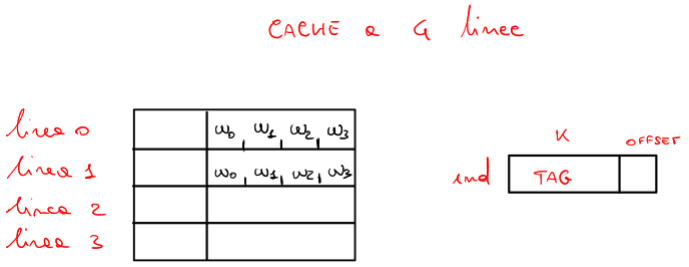
   (c) Il confrontatore prende in input il tag dell'indirizzo in ingresso ed il tag della cache.

   (d) La porta **AND** produce in output l'esito di **hit/miss**, mentre il multiplexer permette di selezionare la word cercata, se presente.

1.5.2 Indirizzamento Associativo

Questo tipo di indirizzamento interpreta parte dell'indirizzo in ingresso come chiave e dedica, ad ogni possibile chiave, una linea di cache. Questo permette di avere una cache molto flessibile e non presenterà problemi causati dalle collisioni. Questa gestione della cache però è davvero costosa, dato che saranno presenti un numero molto alto di confrontatori. Mostriamo le implementazioni di questo schema di cache:

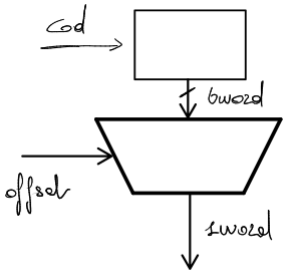
1. **Implementazione Logica**: Illustriamo la gestione delle linee di cache e dell'indirizzo in ingresso.



   (a) Ogni cache avrà due colonne, la prima rappresenterà i **tag** mentre la seconda le *kword* per ogni linea.

   (b) Ogni indirizzo avrà quindi una corrispondente linea in cache, e la word singola in output sarà determinata dalla parte di **offset**.

2. **Implementazione Fisica**: Illustriamo l'implementazione fisica di una **singola** linea di cache.

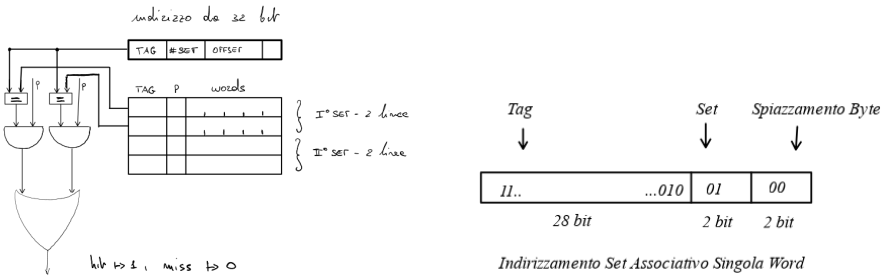


   Questa illustrazione permette di capire il costo di questa gestione della cache. Ogni **singola linea** di cache richiederà un **confrontatore** ed un multiplexer. Solitamente una cache contiene una quantità di linee che si aggira sull'**ordine delle migliaia**.

1.5.3 Indirizzamento Set Associativo

Le due precedenti gestioni di cache offrivano pro e contro. Di conseguenza si preferisce un ibrido tra indirizzamento diretto ed associativo, in modo tale da poter bilanciare rispettivamente sia i problemi causati dalle collisioni sia i costi eccessivi. Illustriamo la gestione di questo tipo di indirizzamento:

1. **Implementazione Logica**: Descriviamo per step il funzionamento di questo schema:



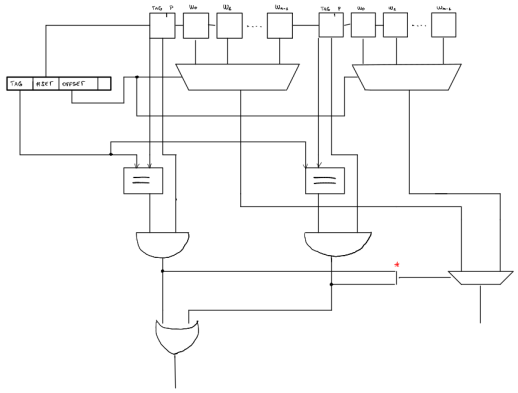
   (a) Ogni **indirizzo** in **ingresso** sarà interpretato con **tag**, **numero set** ed **offset**.

   (b) Le **linee** di cache vengono raggruppati in **set**. Di conseguenza si effettua un indirizzamento **associativo** sul **set** di appartenenza.

   (c) Una volta selezionato il **set** si effettua un indirizzamento **diretto** utilizzando il **tag** tra le varie linee del **set** corrente.

   (d) Infine, se si ha un segnale di **hit**, allora si utilizza l' **offset** per ricavare la **singola word** tra le **kword**.

2. **Implementazione Fisica**: Descriviamo la composizione di questa rete di una cache a 2 vie.



   (a) Ogni via è caratterizzata da un modulo contenente **tag** e **bit** di **presenza** ed i restanti le **word** di quella linea.

(b) L'**indirizzo** in ingresso è caratterizzato da **tag**, **numero** di **set** ed **offset**.

(c) Tutti i moduli contenenti **word** danno in input ad un **multiplexer** il loro contenuto, il cui segnale di controllo è rappresentato dall'**offset** dell'**indirizzo** in ingresso.

(d) Viene utilizzato un numero di comparatori pari al numero di **set**, questi avranno in input il **tag** del primo modulo del corrente **set** ed il **tag** dell'**indirizzo** in **ingresso**.

(e) Delle porte **AND** prendono in ingresso il **bit** di **presenza** e l'output dei **comparatori** per ogni **set** e producono un segnale di **hit/miss** entrando in una porta **OR**.

(f) Infine, l'output delle porte **AND** viene portato ad un multiplexer come segnale di controllo, permettendo la scelta tra le uscite dei **multiplexer** di **ciascun set**. In questo particolare caso non è necessario utilizzare un codificatore prima dell'ingresso di controllo sull'ultimo multiplexer in basso, essendo il caso di soli due set, altrimenti sarebbe stato necessario anche quel componente.

Confronto Caratteristiche Indirizzamenti Riassumiamo le caratteristiche di ciascun tipo di indirizzamento sopra illustrato:

	#vie	#insiemi
diretto	1	#linee cache
associativo	#linee cache	1
set associativo	#vie	$\frac{\#linee}{\#vie}$

1.6 Accenni di Gestione Stalli Miss Penalties

Nel momento in cui accade una **miss penalty** il processore entra in **stallo** per un periodo di tempo abbastanza ampio. Per questo motivo si preferisce attuare delle politiche grazie alle quali si "nasconde" il tempo di stallo in questione:

1. **Out of Order Execution**: Si eseguono operazioni future non in ordine e che permettano di non invalidare le Condizioni di Bernstein.

2. **Esecuzione Speculativa Salti**: Si prevede un salto, assumendo che porti a qualcosa di già fatto.

3. **Prefetching**: Si cerca di prevedere cosa verrà caricato in cache, avviando veri e propri caricamenti in cache in **background**. Questo non sempre può essere realizzato, ma quando possibile permette un efficace minimizzazione degli stalli causati dalle miss penalty.

1.7 Tipologie di Miss

I **miss** si dividono in **specifiche categorie** rispetto alla "causa" che li genera:

1. **Fisiologici**: Cronologicamente il primo miss necessario al caricamento della cache.
2. **Capacità**: La grandezza del **Workspace** supera quella della **cache**.
3. **Conflitto**: Causato dal tipo di indirizzamento. Questa tipologia può essere risolta a tempo di compilazione.

1.8 Scrittura in Cache

Elenchiamo tutte le caratteristiche e le problematiche della scrittura in cache e della sua consistenza in relazione ai dati presenti in memoria principale.

1.8.1 Problemi in Scrittura

Elenchiamo le possibili problematiche causate dalla scrittura in cache:

1. **Scrittura 1 word/b word**: Assumiamo di avere un indirizzo su cui vogliamo effettuare una STR in una cache set associativa a 2 vie. Non posso semplicemente scrivere il **dato (singola word)** e segnare a quella linea il bit di presenza ad 1 perché starei sprecando tutte le $b-1$ word presenti in quella linea su cui potrei ancora scrivere. Dunque si effettua lo stesso procedimento delle LDR, favorendo il caricamento di tutti i dati adiacenti a quello corrente nella linea di cache.

   (a) **Eccezione**: Questo comportamento può causare problematiche nel caso in cui si voglia inizializzare un array con elementi arbitrari (da modificare successivamente) al suo interno. In quel caso non avrebbe senso caricare in cache tutti i dati, dato che verrebbero riscritti poco dopo. Questi tipi di ottimizzazioni vengono controllate dal compilatore, se è in grado di accorgersi di queste problematiche.

1.8.2 Inconsistenza Cache/Memoria - Write Back/Write Through

La scrittura in cache causa un altro tipo di problematica, ossia la **consistenza** dei dati in **cache** ed in **memoria principale**. Tutte le locazioni paralleli dovrebbero essere "informate" di una potenziale modifica. Esistono 2 approcci diversi che risolvono questa problematica:

1. **Strategia Write Back**: Si aggiunge un nuovo **bit flag** alle linee di cache, ossia $M \in \{0,1\}$, se il contenuto della linea è stato modificato allora la flag va ad 1, altrimenti resta a 0. Sarà dunque necessario ad un certo punto un "refresh" della memoria, trasportando tutti i dati modificati ai livelli superiori.
2. **Strategia Write Through**: Non utilizzo un bit flag, ma ogni modifica effettuata in cache viene messa in coda ad un **buffer** e l'aggiornamento dei livelli superiori avverrà in maniera **asincrona**. Questo approccio dipende fortemente dalla dimensione del buffer, questo dovrà essere abbastanza grande da contenere tutte le modifiche in coda fino all'effettivo refresh.

3. **Strategia Write Back Bufferizzata**: Versione asincrona del write back, si compone di queste fasi:

   (a) Si copia la modifica nel buffer di scrittura.

   (b) Senza attendere la fase precedente, carico le b parole nuove in cache.

   (c) Finisce l'accesso in cache.

1.9 Politiche di Rimpiazzamento - Sequenziale/LRU

Che criterio segue ogni tipologia di indirizzamento di cache nel caso in cui bisogna aggiungere un nuovo dato in una cache già piena?

1. **Indirizzamento Diretto**: Viene sostituita in maniera secca la linea di cache a cui sta facendo riferimento il nuovo indirizzo.

2. **Indirizzamento Associativo**: Dato che questo tipo di indirizzamento non provocava collisioni, siamo costretti a scegliere una "vittima" in cache da sostituire con l'indirizzo corrente, essendo certi di aver finito lo spazio.

3. **Indirizzamento Set Associativo**: Viene scelta una "vittima" da sostituire come nel caso precedente ma in uno specifico set.

Il focus diventa quindi la gestione del **working set**, in modo tale da ottimizzare il caricamento/scaricamento delle cache in relazione al corrente working set.

1. **Politica Sequenziale - Indirizzamento Diretto**: Questo approccio è quello più semplice possibile, infatti dopo il primo miss in cache si carica in cache un blocco di b words. Appena sarà richiesta una word fuori da quel blocco viene effettuato un nuovo caricamento di blocco di words in cache.

2. **Politica LRU[3] Puro - Indirizzamento Set Associativo**: L'idea è quella di mantenere un tempo di accesso t_a per sostituire quello che servirà tra più tempo. Questo porta però ad un costo molto alto, dato che per ogni linea andrebbe controllato il tempo "mancante" al prossimo accesso. Si preferisce quindi un compromesso, ossia l'LRU Approssimato.

3. **Politica LRU Approssimato - Indirizzamento Set Associativo**: Ogni linea di cache si porta dietro una flag d'accesso $A \in \{0, 1\}$ che viene alzata ad ogni accesso a quella linea. Ogni k cicli di clock viene effettuata un operazione di **azzeramento**, dove tutte le A vengono portate a 0. Questa logica non è molto costosa dal punto di vista implementativo, ma è fortemente influenzata dall'ampiezza del periodo di refresh, dato che un periodo troppo lungo porterebbe a troppi bit $A = 1$, mentre troppo breve porterebbe alla scelta random di locazioni.

[3]LRU sta per Least Recent Used, dunque utilizzato meno recentemente.

1.10 Cache Coherence

Assumiamo di essere in un contesto multicore, dove ciascun core dispone di almeno un livello di cache completamente dipendente dal core stesso. La gestione concorrente dell'esecuzione complessiva dei programmi va attentamente gestita, dato che facilmente può crearsi inconsistenza tra copie dello stesso dato in cache di core diversi.

Mantenimento Coerenza Abbiamo bisogno di due meccanismi per mantenere la coerenza delle cache:

1. Meccanismo di **tracciamento** copie di un determinato dato x in tutte le cache.
2. Meccanismo di **informazione** a tutte le copie di un dato x sulla **modifica** apportata. Potenzialmente esistono due approcci:
   (a) Propagazione del nuovo valore di x a tutte le copie in tutte le cache.
   (b) Invalidazione di tutte le altre copie di x.

1.10.1 Cache Coherence Protocols

Esistono due protocolli che permettono la sincronizzazione dei dati tra tutte le cache di primo livello dei vari core. Elenchiamoli:

1. **Snoopy Based**: Tutti i core sono in ascolto su un **bus**, quando un **core** effettua una **write** in una sua **cache** manda un segnale sul **bus** che permette una sincronizzazione tra tutti i core.
2. **Directory Based**: Si mantiene una **tabella** nel primo livello di **cache comune** tra tutti i core che contiene tutte le informazioni sui dati e sulle locazioni di tutte le sue copie. In questo modo, utilizzando questa **mappa** è possibile **sincronizzare** tutte le copie nelle varie **cache.**

Ogni tipo di protocollo scelto influisce sull'**AMAT** complessivo.

1.10.2 False Sharing

In specifici contesti, questi protocolli di sincronizzazione causano una quantità molto grande di operazioni "inutili" derivate dalla logica alla base della coerenza implementata in questo modo. Mostriamo un esempio:

1. Assumiamo di avere un array molto grande, sull'ordine dei k in grandezza.
2. Assumiamo di avere una macchina con 4 core, suddividiamo l'array in quattro parti e ciascun core calcolerà la somma di ogni quarto di array, depositando il risultato in una specifica posizione di un array risultato di 4 posizioni.

3. Ogni core caricherà in cache una copia dell'array risultato e cercheranno di sincronizzare tutti i valori di quest'ultimo ad ogni operazione effettuata. Ma impostato in questo modo, l'array risultato è "impropriamente condiviso", infatti ogni core si occupa di **una sola** cella. Questo fenomeno è detto **false sharing** ed è un effetto collaterale della **cache coherence** che causa inutile traffico di sincronizzazione non desiderato. Una soluzione è quella di posizionare, grazie al **padding**, i risultati di ciascun core in memoria che **non risulti** condivisa.

2 Gestione I/O e Periferiche

Per rendere un calcolatore in grado di acquisire dati dall'interno/esterno è necessario gestire l'**I/O**. Assumiamo che un programma impiega un tempo t per terminare. Considerando quest'ultimo, si aggiunge circa un $\frac{1}{10}t$ al tempo complessivo. Un concetto fondamentale è anche quello della differenza tra la velocità delle operazioni di I/O e la velocità di esecuzione delle istruzioni della CPU. E' fondamentale che non si crei un collo di bottiglia al processore alla velocità delle operazioni di I/O, la CPU deve essere in grado di eseguire altre istruzioni mentre lavorano le periferiche esterne.

2.1 Legge di Amdahl

Questa legge stabilisce il rapporto "all'infinito" del tempo parallelizzabile e non parallelizzabile, in questo caso rispettivamente tempo generico di esecuzione della **CPU** e tempo dedicato all'**I/O**.

$$T_{seq} = fT_{seq} + (1-f)T_{seq} =$$

$$\lim_{x \to \infty} \frac{T_{seq}}{fT_{seq} + \frac{(1-f)T_{seq}}{n}} =$$

$$= \frac{T_{seq}}{fT_{seq}} = \frac{1}{f}$$

2.2 Prima Descrizione Gestione I/O

Ogni genere di dispositivo di I/O ha la necessità di gestire:

1. **Controllo**: Gestione degli ordini e dei segnali per gli esiti.
2. **Dati**: Dati di input/output.

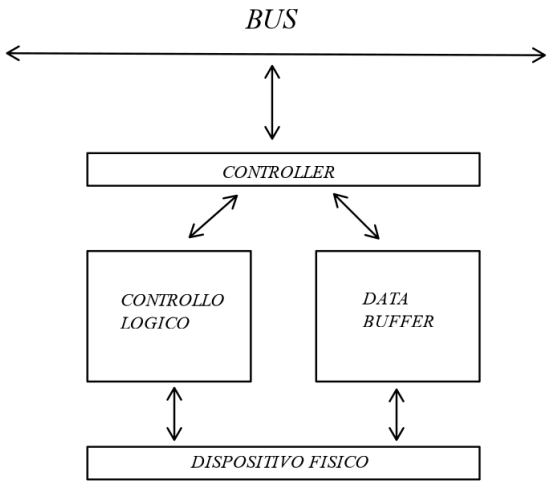
Descriviamo sommariamente i passi necessari alla comunicazione tra **CPU** e device **I/O**:

1. CPU controlla stato periferica, l'I/O risponde al controllo.
2. CPU se e solo se il dispositivo è libero manda il comando alla periferica.
3. Periferica esegue l'istruzione.

(a) Se c'è necessità di trasferimento dati alla CPU viene eseguito in questo momento.

4. I/O "fa rapporto" sull'esito dell'operazione appena eseguita.

Device I/O La comunicazione del device in maniera approssimativa si mostra in questo modo:



La rappresentazione del buffer è **approssimativa**, infatti non dobbiamo pensare che sia un semplice cavo da 1 bit, ma un vero e proprio **insieme di cavi** che permettono il trasporto d'informazioni riguardo **indirizzo**, **controllo**, **dati** ed **arbitraggio**.

Due **tipologie** comuni di **bus** sono ad esempio **USB** o **PCIe**.

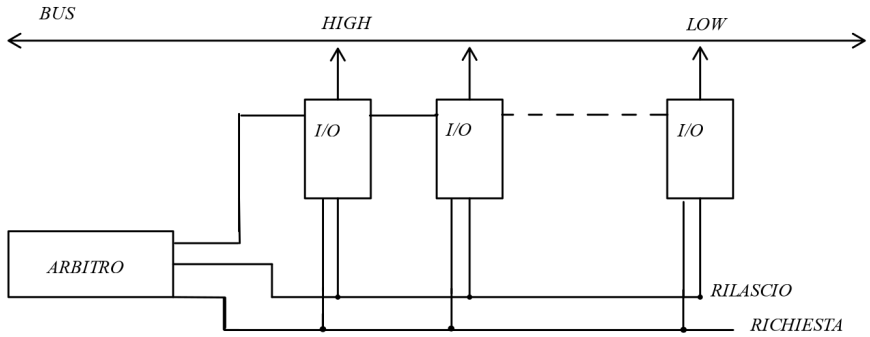
Comunicazioni sul Bus Il bus è condiviso, le informazioni vengono dunque inviate in broadcast a tutti i device, ogni dispositivo però potrebbe avere clock diversi:

1. **Gestione Sincrona**: Il clock è uno solo ed ogni device si adatta a quest'ultimo.
2. **Gestione Asincrona**: Ogni device ha un proprio clock, generando un effetto **skewed**. Questo fenomeno va "ammortizzato" cercando di generare sincronizzazione solo ad uno specifico momento di comunicazione tra dispositivi e CPU.

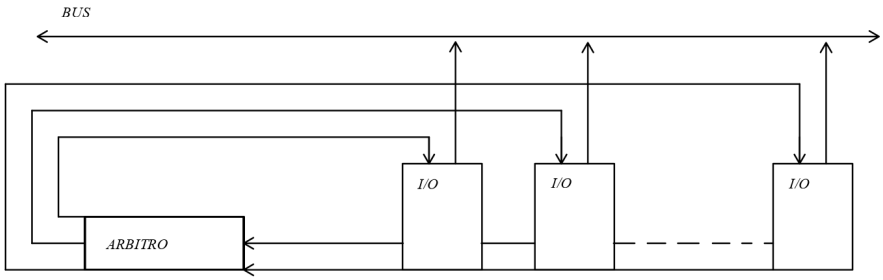
2.3 Protocolli di Arbitraggio del Bus

Dopo aver definito il **bus** come mezzo di **comunicazione condiviso** tra tutti i dispositivi, è necessario stabilire come questi ultimi vengano arbitrati in modo tale da **non operare** in **conflitto** tra loro.

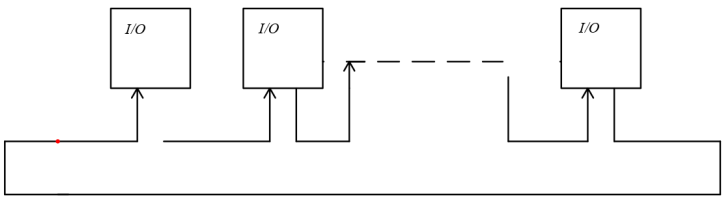
1. **Daisy Chaining**: Sul bus è settata una priorità per device:



2. **Richiesta Indipendente**: Una componente **arbitro** sceglie chi può o meno comunicare sul bus tra tutte le periferiche esterne:



3. **Token Passing**: Esiste una sorta di comunicazione "ad anello" su cui gira un **token**. Chi acquisisce il token ha il permesso di comunicare sul bus e al termine della loro operazione dovranno restituire il token:

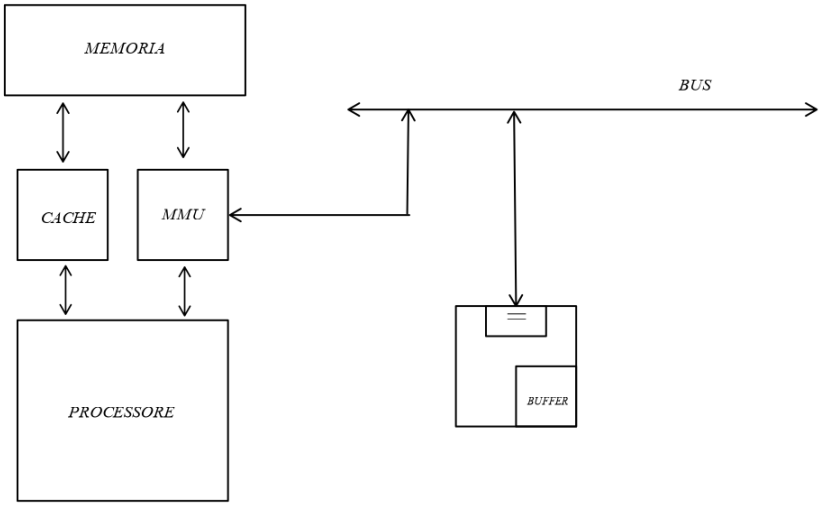


2.4 Meccanismi di Gestione Bus

Una volta definita la composizione e gli arbitraggi del bus bisogna descrivere in che modo vengono inviati i dati ed i segnali grazie a specifici meccanismi.

2.4.1 Memory Mapped I/O

Immaginiamo di star lavorando con un architettura a 32 bit, **riserviamo** una quantità di questi **bit** in ogni indirizzo come **flag** che indichi se bisognerà o meno **dirottare** il contenuto verso la **memoria principale** o la **memoria** di qualche **periferica**.



L'unità che ci permette di effettuare questa scelta è la *Memory Managment Unit* (**MMU**) che ha accesso sia alla memoria sia alle periferiche. Abbiamo quindi un modo per reindirizzare operazioni di load/store anche sul buffer delle periferiche.

2.4.2 Interruzioni

Un altro meccanismo fondamentale è quello della gestione delle interruzioni, ossia dei segnali che permettono la comunicazione tra periferiche e CPU. Immaginiamo che la CPU stia eseguendo questo loop:

```
1    while (true){
2        fetch
3        decode
4        execute
5        memory
6        write back
7        interruzione?
8    }
```

Questo ci permette di ottenere un segnale sul chi e perchè sia stata lanciata un interruzione (se è stata lanciata), in modo che la CPU possa agire di conseguenza.

2.5 Metodi di Query sullo Stato dell'I/O

Assumiamo che la CPU abbia ordinato l'esecuzione di un istruzione ad una periferica. Come fa la CPU a sapere quando la periferica ha terminato l'operazione in questione?

Presentiamo le due principali metodologie:

1. **Programmed I/O**: Metodologia sincrona, la CPU effettua continue richieste alla periferica (nello specifico al campo *ready* del suo buffer) per capire se l'operazione è conclusa. Elenchiamo nello specifico le fasi della CPU:

   (a) Controlla se la periferica è in *idle*. Se lo è, assegna l'operazione (passando *cmd* e *params*).

   (b) Controlla attivamente lo stato *ready* della periferica in questione.

   (c) Quando la periferica ha finito, acquisisce il risultato.

   Polling E' possibile modificare questa attesa attiva grazie al fenomeno del polling, ossia una **frequenza** di **richiesta** al campo *ready* del buffer della periferica. Questa frequenza va settata con cura, dato che richieste troppo frequenti si avvicinerebbero all'attesa attiva, mentre richieste poco frequenti causerebbero poca reattività. Questa metodologia causa quindi un rallentamento della CPU dato che si effettuerebbero molte operazioni inutili, ma si cerca di evitare la piena attesa attiva.

2. **Interrupt Driven I/O**: Metodologia asincrona, la CPU carica l'istruzione alla periferica e attende non attivamente un segnale di terminazione da parte della periferica stessa. Illustriamo per step:

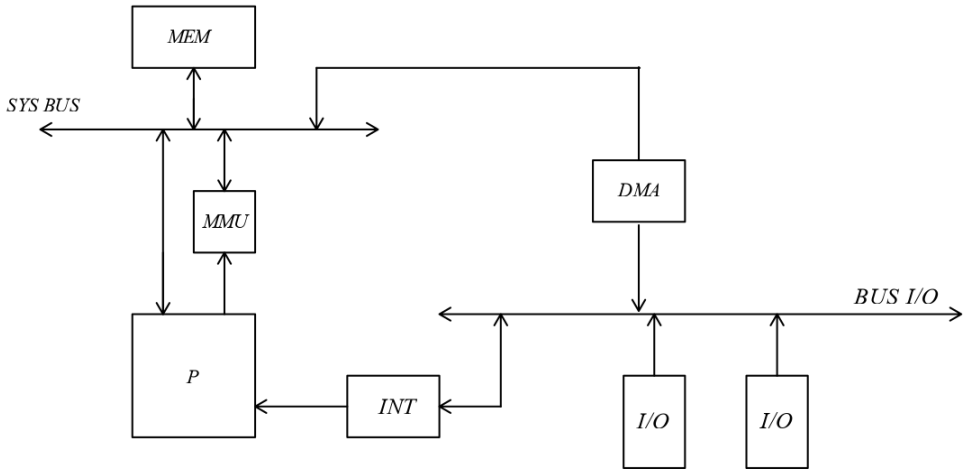
   (a) CPU legge l'*idle* della periferica, se trova 1 carica *cmd* e *params*.

   (b) CPU riprende il suo ciclo standard, controllando ad ogni ciclo di clock se sono presenti segnali d'interruzione.

   (c) Se dovessero incorrere delle interruzioni, la CPU salverebbe il corrente stato dell'operazione e analizzerebbe l'interruzione data.

   Questa metodologia ci permette di evitare il calcolo di un **polling** ottimale dato che sarà la periferica stessa a notificare il completamento dell'operazione grazie all'unità delle **interruzioni**.

System Bus Notiamo che dopo le ultime modifiche è necessario che la memoria non sia collegata solo al processore. Di conseguenza viene posizionato un nuovo bus che permetta l'interazione diretta tra memoria e device esterni.



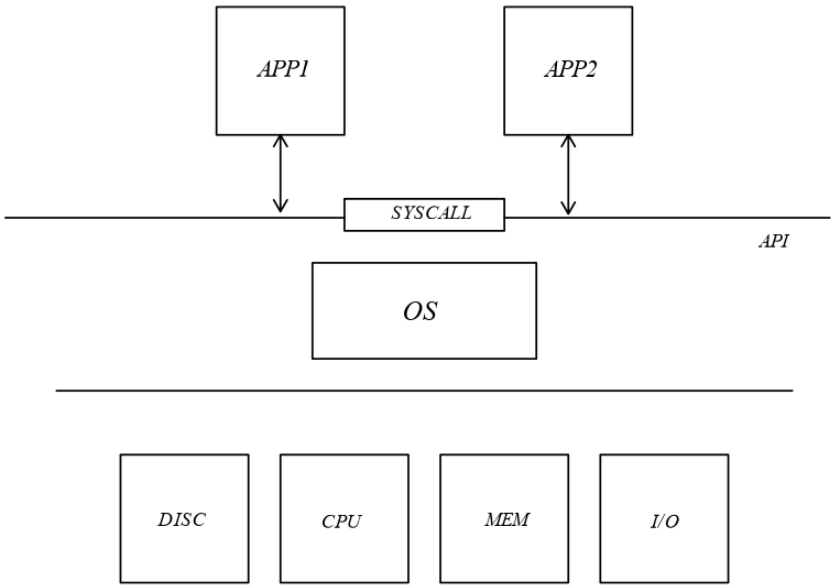
Il modulo *Direct Memory Access* (**DMA**) permette l'interazione stabilita prima.

Priorità tra i due Bus E' necessario stabilire a quale **bus** dare **priorità**. Sappiamo che le operazioni sui dispositivi di I/O sono molto più lente delle operazioni del processore, ma se ricevessimo un segnale d'interruzione risulterebbe molto probabile che il buffer del device che ha inviato l'interruzione è pieno. Di conseguenza, per prevenire perdite di dati dei buffer, si preferisce dare priorità al bus I/O.

DMA e Coerenza Cache La coesistenza di **cache**, **buffer I/O** e **memoria** principale causata dal **DMA** può causare forte inconsistenza. Conosciamo infatti i meccanismi che permettono di tenere coerenti le linee di cache e le informazioni in memoria principale. In presenza del **DMA** si preferisce invalidare la cache, per evitare a monte di avere problemi con i meccanismi di coerenza citati prima.

3 Introduzione ai Sistemi Operativi

Il sistema operativo è il software fondamentale per un calcolatore. Solitamente scritto in un linguaggio ad "alto" livello (es. C), si occupa della gestione del calcolatore (processi, I/O, memoria).



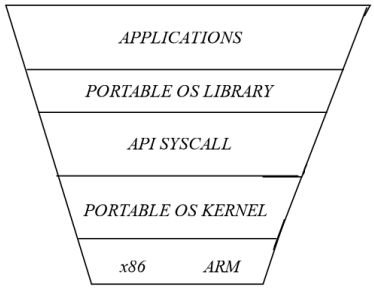
Astrazioni del Sistema Operativo Uno degli obiettivi del'OS è quello di rendere questi tre tipi di astrazione all'user:

1. **Illusionist**: Il sistema operativo permette di immaginare spazi di memoria allocati in maniera contigua. Questo è più un concetto **logico** che fisico, infatti sarà lo stesso **OS** ad occuparsene a basso livello, astraendo l'user dalla gestione fisica della memoria.
2. **Referee**: Il sistema operativo deve gestire le risorse condivise ed il settaggio delle proprietà di ciascun utente.
3. **Glue**: Il sistema operativo mette a disposizione le librerie e le utilities necessarie alla coesistenza di CPU, memoria e disco.

Proprietà Garantite dal Sistema Operativo Per costruire l'astrazione, il sistema operativo deve garantire specifiche proprietà:

1. **Affidabilità**
   (a) Disponibilità
2. **Sicurezza**
3. **Portabilità**

4. **Performance**
   (a) Latenza
   (b) Throughput (Tasso di produzione per unità di tempo)
   (c) Overhead (Quanto lavoro aggiuntivo necessario alla resa dell'astrazione)
   (d) Fairness (In presenza di più utenti, questi devono avere stesse possibilità di utilizzo delle risorse)
   (e) Predictability (Possibilità di prevedere sommariamente il comportamento del sistema operativo in base al contesto)



3.1 Struttura Kernel - Monolitica vs a Micro Kernel

Esistono diversi modi per organizzare il nucleo di un sistema operativo, illustriamo le due metodologie agli antipodi:

1. **Kernel Monolitico**: Tutto ciò che è a disposizione dell'OS è nel suo kernel, di conseguenza ogni modifica o aggiunta di funzionalità integrate comporta l'estensione del kernel stesso.
2. **Micro Kernel**: L'effettivo kernel contiene solo pochissime funzionalità (come lo scheduling di attività o l'acquisizione dell'I/O), il resto delle attività vengono integrate come veri e propri processi.

3.2 Computazioni nel Tempo - (Batch, Spool, Time Sharing)

Elenchiamo tre metodologie di gestione delle elaborazioni nei primi calcolatori:

1. **Sistema Batch**: Assumiamo di avere una lista di *jobs* da elaborare.

   $$Batch = \{job_1,\ job_2,\ job_3\}$$

   Ognuno di questi *job* si compone di parte CPU e parte I/O. Il sistema Batch è il più semplice, infatti eseguirà in maniera sequenziale i lavori.
2. **Spool**: Con lo *spool* si tenta di riempire i "vuoti" di elaborazione durante operazioni di I/O con calcoli con CPU del *job* successivo che non risulti in nessun modo dipendente da quello prima.

3. **Time Sharing**: La CPU eseguirà frazioni alternate di ogni *job* in coda. Quando viene richiesto dell'I/O da un *job* questo non sarà più contato in questa operazione di interleaving.

3.3 Protezione del Calcolatore

Il sistema operativo deve occuparsi anche di meccanismi di **protezione** del calcolatore. Nello specifico, con protezione intendiamo dei meccanismi per cui ogni operazione sia legata a relativi **diritti** e **permessi**.

3.3.1 Dual Mode - User/System

Esistono vari ruoli in base al sistema operativo utilizzato che permettono o meno di eseguire determinate istruzioni. Mostriamo uno schema semplice basato su due ruoli, ossia **User** e **System**:

1. **User Mode**: L'user avrà **meno privilegi**, di conseguenza potrà eseguire meno istruzioni. Spesso l'user avrà modo di utilizzare funzioni "più ad alto livello" come ad esempio una *fread* oppure una **syscall**, senza avere accesso direttamente alla routine di esecuzione effettuata dal sistema.
2. **System Mode**: Il System avrà **più privilegi**, di conseguenza ha il permesso di eseguire più istruzioni. Una particolarità del "ruolo" System è quello di poter eseguire le routine invocate dalle **syscalls** degli user. Per questo l'invocazione di syscall causa uno switch di modalità, per fare in modo che la sua routine possa essere eseguita dal lato system.

3.3.2 Meccanismi Necessari alla Gestione del Dual Mode

Per garantire una dual mode consistente l'OS deve prima fornire questi meccanismi:

1. **Istruzioni Privilegiate**: Operazioni non eseguibili dall'user, sarà la *CPSW* ad indicare se sarò o meno user. Degli esempi di istruzioni privilegiate:
   (a) Disabilitare le interruzioni.
   (b) Scrivere manualmente nella *CPSW*.
   (c) Scrivere manualmente nel *timer*.

2. **Limitazione sugli Indirizzi di Memoria**: Si limitano gli indirizzi di memoria accessibili dall'user, rendendo visibile all'user solo una memoria virtuale che mappa su quella reale secondo degli schemi specifici, come ad esempio il *Base & Bound*.

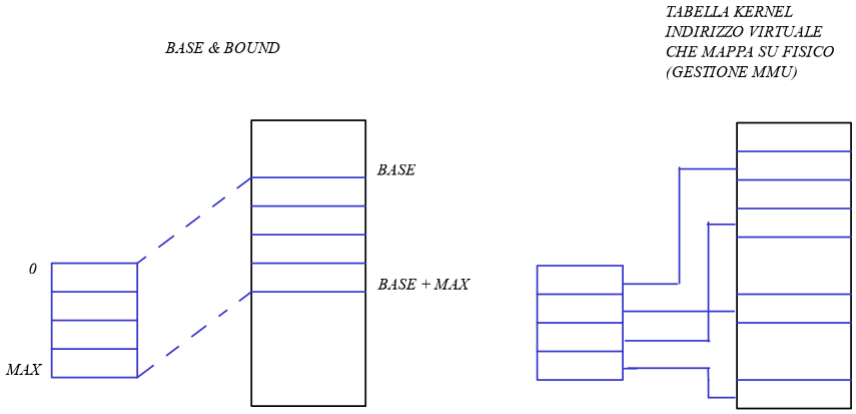


Figure 1: Due esempi di limitazione e mapping di memoria virtuale su quella fisica.

3. **Timer**: Si definisce un modo per gestire interruzioni in caso di **overtime**, se un processo dovesse infatti "bloccarsi" avremmo un modo per gestire in maniera semiautomatica il suo overtime, interrompendolo nel caso in cui sia trascorso troppo tempo.

4. **Safe Ways per Switch Mode**: L'OS deve anche garantire metodologie **sicure** per lo **swap** tra le modalità user e system. In **ARM** una di queste metodologie è la *SVC* (supervisor call), che funziona esattamente come una chiamata di funzione, ossia si aspetta i parametri nei registri R_0, R_1, R_2 ... e nello specifico in R_7 si aspetta il codice della *syscall* che si sta effettuando.

   (a) **Esempio di una read**: Vediamo un esempio di syscall, ossia a cosa corrisponde effettuare una read:

   $$\boxed{read(file\_desc,\ buff,\ len)}$$

```
1    MOV R0, "file_desc"
2    MOV R1, "buff"
3    MOV R2, "len"
4    MOV R7, "#CODICE_READ"
5    SVC 0
```

   Stiamo descrivendo in "pseudo ARM" le operazioni effettuate dall'invocazione della read, ossia una comune *syscall*.

In vari casi si effettuando degli **switch mode**, ad esempio **da user a supervisor** in casi come *syscall*, *interruzioni* o *eccezioni*, mentre **da supervisor a user** in return dalle operazioni elencate prima, in *creazioni di processi* e in *upcall* (ossia se scatta il timer esegui una specifica operazione).

3.4 Descrizione di un Processo

Immaginiamo un processo che "nasce" da queste fasi:

1. Un **programma** (codice sorgente) viene compilato.
2. La **compilazione** produce un **eseguibile**.
3. L'**eseguibile** viene lanciato, creando un **processo**.
   (a) Verrà allocata della memoria durante il lancio dell eseguibile, necessaria al corretto funzionamento del processo.
   (b) Viene generato un **descrittore di processo** durante il lancio dell'eseguibile, per fare in modo che si possano mantenere dei **metadati** riguardanti il processo stesso.

3.4.1 Thread e Processi

Assumendo che ogni **processo** abbia spazio d'indirizzamento indipendente dagli altri, possiamo descrivere le differenze con i **thread** che invece **condividono** tra loro memoria **dati** ed **istruzioni**. Dunque un processo sarà "composto" da più thread, che condivideranno lo spazio allocato per il processo "padre".

3.4.2 PCB - Process Control Block

Il **PCB** descrive un intero processo, portandosi dietro una serie di informazioni utili, elenchiamole:

1. **PID**: ID del processo.
2. **Stato Processo**: Corrente stato del processo (*running*, *new*, *ready*, *wait*, *terminated*).
3. **Registri CPU**: Stato architetturale, ossia contenuto dei registri della CPU durante l'esecuzione del processo.
4. **Informazioni di Scheduling**: Priorità d'esecuzione assegnata al processo in questione.
5. **Puntatore ai Thread**: Puntatore a tutti i thread che compongono il processo.
6. **Informazioni su Memoria Allocata**: Informazioni sulla memoria allocata automaticamente che dovrà essere deallocata a tempo di terminazione del processo.
7. **Informazioni su Risorse Allocate**: Informazioni su tutta la memoria allocata a supporto del funzionamento del processo (ad esempio descrittori di file).

3.5 Descrizione Interruzioni

Descriviamo tutti i meccanismi a supporto del funzionamento delle interruzioni:

1. **Interrupt Vector**: La vera e propria interruzione sarà un semplice **codice** numerico. L'interrupt vector mappa a quel codice l'effettiva routine da eseguire nel caso venga rilevata l'interruzione in questione. Possiamo dunque definirla come zona di memoria che da contesto alle interruzioni in base al loro codice.

2. **Kernel Interrupt Stack**: Nel momento in cui l'OS si accorge di un interruzione ha la necessità di salvare il corrente stato architetturale[4]. Tutti questi dati corrispondono però a dati di **sistema** e non di **user**, di conseguenza non possono essere pushati sullo stack regolare ma devono essere riposti nel **kernel stack**. In questo modo sappiamo dove il sistema dovrà riacquisire i dati dopo la completa elaborazione dell'interruzione.

3. **Interrupt Masking**: Opzione di apertura/chiusura delle interruzioni, possiamo in questo modo disattivare la ricezione di interruzione durante specifiche fasi critiche.

4. **Swap Mode Atomico**: Meccanismo di swap mode safe.

5. **Ripartenza del Ciclo CPU Regolare**: Meccanismo di ripristino della routine standard del processore.

3.5.1 Handler di Interruzioni

Gli **Handler** sono gestori di interruzioni ed hanno l'obiettivo di eseguire tutte le fasi necessarie all'elaborazione completa di un interruzione. Possono essere visti come **kernel threads**, si fermano solo quando hanno terminato di elaborare l'interruzione e hanno lo scopo di effettuare queste fasi:

1. Salvare lo **stato** a tempo d'acquisizione dell'interruzione.

2. **Trattare l'evento** in base alla **specifica interruzione** ricevuta.

3. **Interagire** con lo schedulatore per dare **priorità** all'evento legato all'elaborazione dell'interruzione.

Dopo la sua esecuzione vanno restabiliti: **stato precedente**, **program stack**, **parola di stato** e **modalità utente**, ripristinando anche il funzionamento delle interruzioni.

Interruzioni in ARM In architettura **ARM** quando il processore coglie un interruzione a fine del suo ciclo standard, va nell'unità **INT**, che ha una "propria memory mapped I/O", ricavando così le informazioni sul chi e sul perchè sia stata sollevata l'interruzione in questione.

[4]Per stato architetturale s'intende il corrente contenuto dei registri.

3.6 Esempi d'Implementazione su Architettura ARM

Mostriamo come meccanismi generici dei sistemi operativi vengano implementati su ARM.

3.6.1 CPSR ed Interrupt Masking

Nel registro **CPSR** (Current Program Status Register) sono presenti delle informazioni riguardanti le interruzioni, infatti:

1. **Bit I (Interrupt)**: Questo bit, solitamente settato a 0 può essere cambiato in 1 effettuando così un mask. Questo ci permetterebbe temporaneamente di ignorare tutte le interruzioni.
2. **Bit F (Fast Interrupt)**: Funziona allo stesso modo, a differenza del normale Interrupt il Fast Interrupt salva parte dello stato architetturale automaticamente in specifici registri ombra.

Un bit tra quelli del **CPSR** è dedicato anche al **Thumb Arm**, settando ad 1 la **flag** si ha infatti la possibilità di **switchare** in direzione di ARM su istruzioni di dimensione dimezzata.

3.6.2 Modalità di Esecuzione

In ARM esistono **diverse** modalità di esecuzione, elenchiamole:

MODE	CODICE NELLA CPSR
User	10000
Supervisor	10011
Abort	10111
Undefined (Reserved)	11011
Interrupt	10010
Fast Interrupt	10001

Nello specifico queste modalità avranno un vettore con locazioni contenenti ciascuna la routine da eseguire in caso di switch di modalità.

Reset
Undefined
Supervisor Call
Prefetch Abort
Data Abort
Reserved
Interruption
Fast Interruption

In questo vettore **non saranno presenti** i puntatori alle routine delle modalità, infatti saranno presenti **direttamente** le routine da eseguire sottoforma di istruzioni ASM.

Fasi di Swap Mode Descriviamo in dettagli tutti i passi necessari ad uno scambio di modalità:

1. In **Banked LR** salvo in corrente **PC**.
2. In **SPSR** (Banked CPSR) salvo la corrente **CPSR**.
3. Cambio i 5 bit che definivano la mia corrente proprietà nella **CPSR**.
4. Cambio **Reg** in **Reg Banked**.
5. Disabilito le **Interrupt**.
6. In **PC** metto il codice del **mode** verso cui voglio switchare

Ognuna di queste fasi viene eseguita in maniera **atomica** per mantenere consistenza.

3.6.3 Banked Registers

In ARM alcuni registri si definiscono **banked** perchè preservati in maniera automatica, anche durante lo swap di modalità.

1. **Int, SVC, Abort, Undefined**: Queste quattro modalità hanno come banked registers SP ed LR.
2. **Fast Int**: Ha come banked registers $R8 - R12$, SP ed LR.

3.7 System Calls

Definiamo le **SYSCALLS**, ossia una sorta di chiamata di procedura ma in direzione del sistema operativo, **caratterizzata** da uno **swap di modalità** durante la sua esecuzione.

Il flusso di una syscall è così definito (esempio su architettura ARM):

1. L'**User** effettua la chiamata al sistema.
2. Si **switcha** verso l'**SVC**.
3. Si **settano** i **parametri** forniti dall'**user** in maniera tale da renderli **leggibili** anche in **modalità system** (scambi con i banked registers oppure effettuando delle pop dal kernel stack).
4. Si chiama la **routine della syscall** indirizzandosi sulla tabella che contiene le routine stesse grazie al **codice assegnato** alla corrente syscall.

Queste operazioni, dallo switch di modalità in poi, sono eseguite dal **Call Handler**.

Mostriamo nel successivo sottocapitolo un esempio di implementazione in ASM ARM v7 di Call Handler.

3.7.1 Call Handler in ASM ARM v7

Mostriamo una possibile implementazione di **Call Handler** commentandola successivamente.

```
1    STMFD SP!, {R0-R12,LR}
2    ADR R8, BASE_SYS_CALL_TABLE
3    @CHECK
4    LDR PC, [R8,R7,LSL #2]
5    LDMFD SP!, {@PARAMETRI PREC.}
6    MOVS PC, LR
```

1. Store multipla dei registri e del link register per **salvare** lo stato **architetturale corrente**.
2. Si mette in r8 la $BASE\_SYS\_CALL\_TABLE$ (base della tabella delle syscalls) grazie all'istruzione ADR.
3. Si cambia il PC grazie ad una load nella **tabella delle syscall** con **base**

   d'indirizzamento in $R8$ e con il **codice di syscall** in $R7$, ricordando di star lavorando su indirizzi che vanno di quattro in quattro (caratteristica dell'ARM v7).
4. **Restore** dello **stato architetturale** pre-call.
5. Si **ristabilisce** il vecchio **Program Counter** per **ritornare** a chi ha invocato la syscall.

3.8 Upcalls - Segnali

In user mode abbiamo la possibilità di registrare un **handler** che verrà invocato dal **kernel** a tempo dell'occorrenza di uno specifico **evento**. Il concetto di **up** deriva dal fatto che sto definendo qualcosa a livello **user** che verrà eseguito dal **kernel** quando necessario. Elenchiamo altre caratteristiche vantaggiose delle upcalls:

1. Gestiscono lo stack **user-space** in **maniera distaccata** dal resto delle informazioni.
2. Hanno meccanismi di **auto salvataggio** dei **registri**.
3. Permettono il **masking** degli **altri segnali** durante l'esecuzione di un handler.

3.9 Kernel Boot

Descriviamo cosa accade all'accensione di un calcolatore:

1. La **CPU** accede ad una locazione di una memoria ROM/EEPROM facendo in modo che il **BIOS** possa caricare il *first stage bootloader*.
2. Questa **prima forma** di **bootloader** inizializza il controllo della memoria, qualche dispositivo di I/O e **carica** il *second stage bootloader*.

3. Questa **seconda forma** di **bootloader** si occupa del caricamento del **kernel** del sistema operativo e della **root** del file system, lasciando infine il controllo al kernel.

3.10 Sommario Specifiche Hardware Necessarie all'OS

Possiamo dunque riassumere tutti i minimi meccanismi necessari alla composizione di un sistema operativo consistente:

1. **Livelli di Privilegio**: E' necessaria l'esistenza di **almeno** due livelli di privilegio (user/system) anche se abbiamo visto che in ARM ne sono presenti diversi (SVC, Abort, User, Interrupt...)
2. **Istruzioni Privilegiate**: Istruzioni eseguibili **solo se non** siamo in **modalità user** per garantire sicurezza.
3. **Traduzione Indirizzi Memoria**: Meccanismo di map di **indirizzi virtuali** disponibili a reali **indirizzi** di memoria **fisici**.
4. **Eccezioni**: Routine eseguite in caso di **violazione di privilegi**.
5. **Interruzioni**: Meccanismo di interruzioni che permetta la **comunicazione** tra **device** e **CPU**.
6. **Mask di Interruzioni**: Meccanismo che ci permetta di disattivare le interruzioni in fasi critiche.
7. **Syscalls**: Interfaccia tra **utente** e **macchina** tramite invocazione di operazioni "standard", dunque possibilità di esecuzione di azioni privilegiate da un programma user.
8. **Boot**: Routine di avvio della macchina.
9. **Operazioni Atomiche su Memoria**: Operazioni in memoria il cui utilizzo **garantisce consistenza** anche in contesti multithread.
10. **Virtualizzazione**: Gestione di **hypervisors** che permettano di creare **diversi ambienti virtuali** anche sulla stessa macchina fisica.

$$\sum_{i=1}^{N} x1^2$$