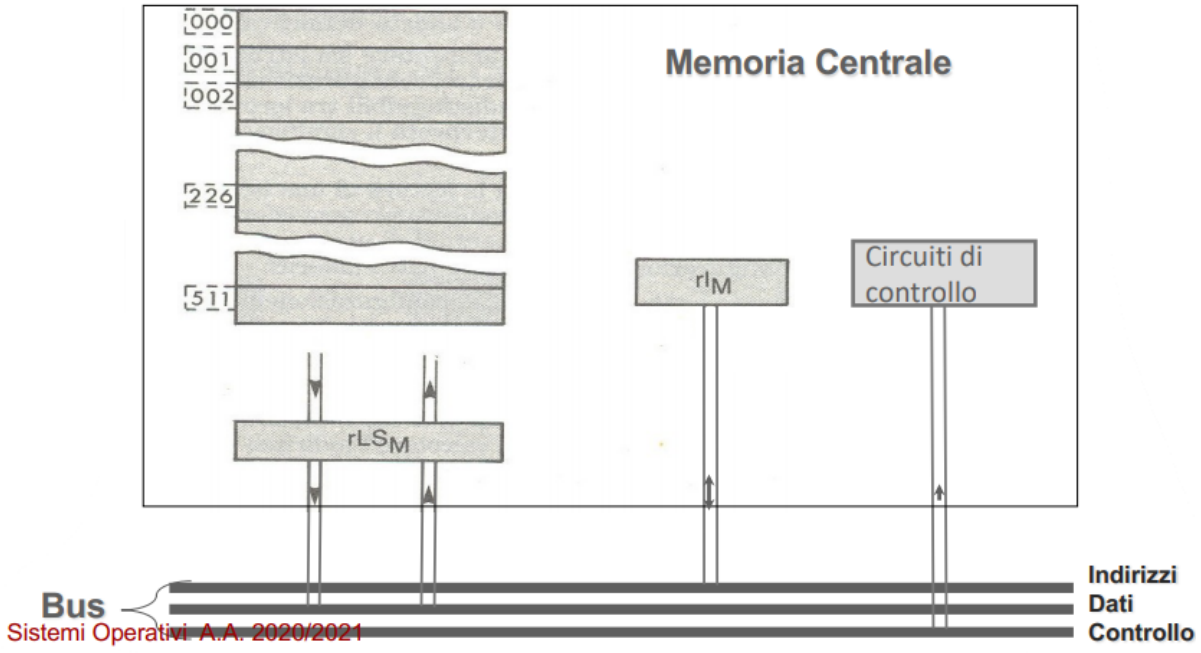
01b - Architettura Hardware

CPU e ciclo Fetch-Decode-Execute

Operazioni di un Computer

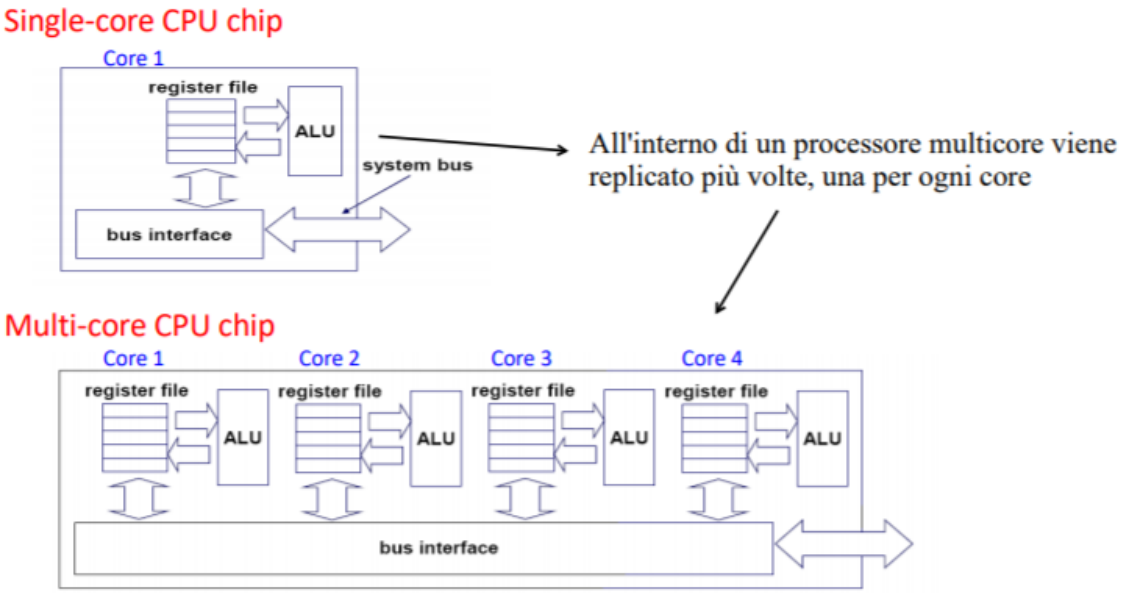
I dispositivi di I/O e le CPU funzionano concorrentemente (architettura master-slave). Ogni controller si occupa solo di un particolare tipo di dispositivo e ha a disposizione un buffer locale. La CPU trasferisce dati dalla memoria centrale ai buffer locali e viceversa, che tra l'altro è un'operazione di I/O. Il controller dei dispositivi e CPU devono essere sincronizzati tra loro.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/21701664-a121-43aa-9154-d289c0c2ceb0/a.pdf



Architettura della Memoria Centrale

Single-Core, Multi-Core, significa che all'interno del processore vi sono o meno diversi core, ciascuno dei quali lavora per il medesimo processore.



Ciclo di funzionamento della CPU

1. **Fetch** (prende l'istruzione)
2. **Decode** (decodifica dell'istruzione)
3. **Execute** (esegue l'istruzione)

(Nelle CPU reali in questo ciclo ci sono anche altri passi di gestione/sicurezza)

- Fetch: prende l'istruzione da eseguire dal PC (Program Counter), passa l'indirizzo al MAR e attraverso il Bus si passa l'informazione alla memoria centrale; alla fine nell' IR abbiamo un'istruzione.
- Decode: legge l'istruzione nell'IR e attiva una serie di segnali di controllo e li invia all'ALU.
- Execute: esegue l'istruzione; aumenta sempre di 1 il PC (Registro Contatore di Programma).

Finita l'esecuzione (qualunque essa sia stata), la CPU ritorna da capo

Esempio ciclo di funzionamento

Fetch:

Immaginiamo di avere nel program counter (PC) l'indirizzo dell'istruzione che vogliamo eseguire, quindi la prima cosa che deve fare la CPU è verificare che all'interno della IR (Instruction Register) ci sia l'istruzione che deve eseguire. La CPU fa partire una richiesta di lettura al controller della memoria centrale. La memoria centrale va a leggere il contenuto della cella il cui indirizzo è contenuto nel registro. Il contenuto della memoria viene memorizzato sul registro lettura e scrittura (Memory Data Register). Questo registro è direttamente collegato al Bus Dati, ed arriva all'IR della CPU, così facendo avremo l'istruzione da eseguire nel MDR. L'ultima fase della Fetch consiste nel ricopiare il contenuto dell'MDR nell'IR della CPU.

Fetch conclusa, inizia la fase DECODE.

Decode:
A questo punto il decoder è agganciato all'IR e quindi riesce a decodificare quell'istruzione. Il che significa inviare una serie di segnali di controllo ad altre parti della CPU stessa o ad altri hardware.

Fine DECODE, INIZIO EXECUTE.

Execute:
A questo punto verrà eseguita l'istruzione e verrà incrementato il registro contatore RCP. Quindi se l'istruzione che ho eseguito sarà la x, l'istruzione successiva molto probabilmente sarà la $x + 1$. Così al termine

dell'esecuzione, riparte da capo con la fetch che andrà a leggere l'istruzione nel registro $x + 1$.

Appena accendiamo la macchina, sarà onere di chi progetta la macchina inserire all'indirizzo 0 della memoria centrale un programmino che gestisca la fase di inizio della macchina. La RAM è volatile, quindi quando accendiamo la macchina non c'è nulla. Per questo motivo la parte bassa della memoria centrale è costituita da dispositivi ROM, che sono a sola lettura e non è volatile. Quindi la memoria centrale di un sistema non è solo composta dalla RAM, ma anche da una parte della ROM, in cui c'è scritto il programma dell'avvio.

Organizzazione della memoria

Gerarchia delle memoria

Le memorie sono organizzate in base al costo, velocità e volatilità (le memorie più veloci sono più costose e volatili).

Caching: consiste nel copiare dati da un livello di memoria ad un livello superiore (più veloce). È una tecnica che serve a tenere dati che sono stati utilizzati più di frequente.
Questo richiede una politica di cache management e un meccanismo di consistenza dei dati che si trovano a più livelli.

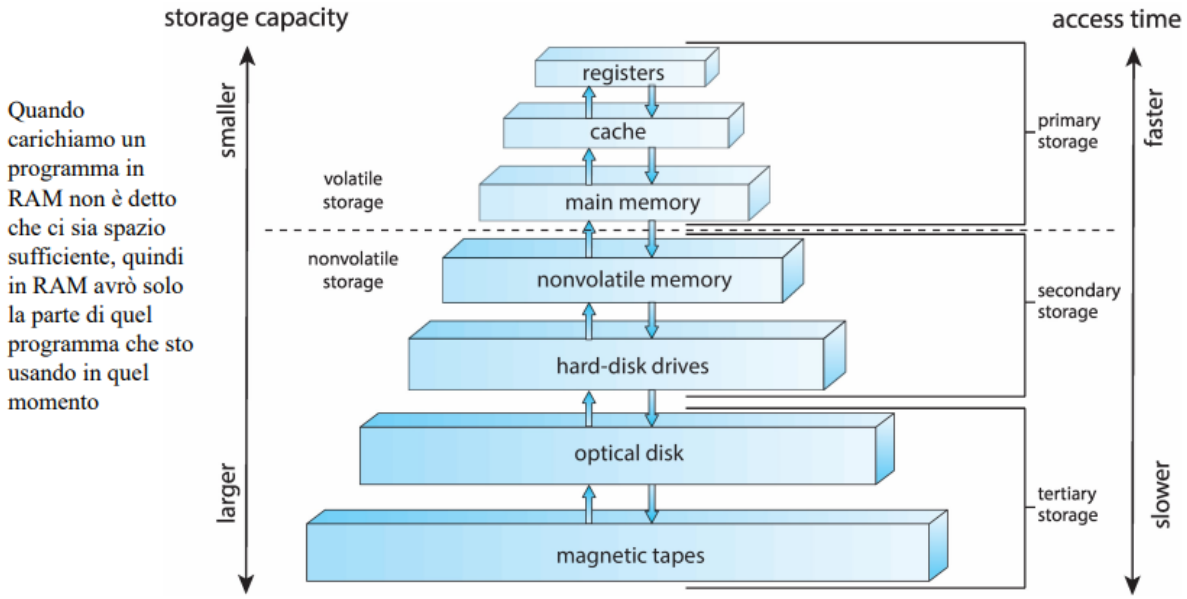
Cache (CPU) → caching di → Memoria Centrale → caching di → Memoria Secondaria
La CPU è volatile, costosissima, velocissima e piccola; la Memoria Centrale è volatile, mediamente costosa, veloce, di media grandezza; la Memoria Secondaria è non volatile, economica, poco veloce, enorme.

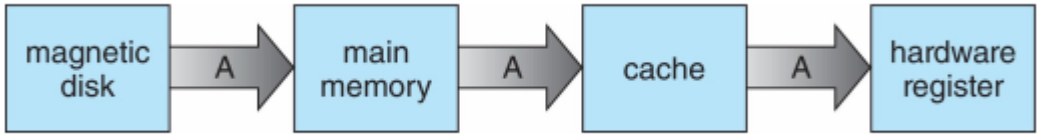
Volatilità: nel momento in cui togliamo l'alimentazione al nostro sistema di elaborazione il contenuto della memoria viene perso (quando riaccendiamo la macchina quello che avevamo memorizzato scompare).

Velocità: il tempo che devo impiegare per andare a leggere qualcosa, quanto tempo impiego per recuperare un dato.

Costo: il costo influisce sulla quantità di memoria che si ha a disposizione. Se una memoria è molto costosa probabilmente non se ne avrà in grande quantità

all'interno del sistema di elaborazione; se si devono memorizzare grosse quantità di dati conviene utilizzare delle memoria meno costose.





Il processore preferisce andare a recuperare i dati nella memoria di cache perchè è più veloce, ma essendo più piccola della memoria centrale a volte li recupera da quest'ultima.
Nei sistemi multi-processore ogni processore ha la sua memoria di cache, quindi possono andare a prendere i dati nella cache a patto che siano lì (utilizzano lo stesso bus quindi ci sono tempi di attesa).

Protocolli di coerenza della cache

L'ampio ventaglio dei sistemi di memorizzazione disponibili in un elaboratore può essere ordinato secondo una scala gerarchica, sulla base della velocità e del costo. I gradini più alti ospitano i dispositivi più veloci, ma anche più dispendiosi. Andando verso il basso il costo per bit generalmente diminuisce,

mentre il tempo di accesso tende ad aumentare.
Si tratta di un compromesso ragionevole: se un certo sistema di memorizzazione, a parità di condizioni, fosse più veloce e nel contempo meno costoso rispetto ad altri, verrebbe meno qualunque motivo per usare la soluzione più lenta e costosa. I dispositivi nei quattro livelli superiori possono essere materialmente costruiti con la memoria a semiconduttori.
Oltre a caratterizzarsi per velocità e costo, i sistemi di memorizzazione si suddividono in
volatili e **non volatili**.
Come si è accennato, la
memoria volatile comporta la perdita dei dati nel caso di interruzione dell'alimentazione. Qualora non si disponga di sistemi di salvataggio automatico dei dati (sistemi di backup) basati su dispendiose batterie o generatori elettrici, affinché siano preservati è necessario salvare i medesimi su
memoria non volatile.
Nella gerarchia della memoria,
i dispositivi di memorizzazione al di sopra del disco RAM sono volatili, mentre gli ultimi tre in basso sono non volatili. Un disco RAM si può progettare per essere volatile o non volatile. Durante il normale funzionamento, il disco RAM memorizza i dati in un capiente vettore DRAM, che è volatile. Tuttavia, molti dischi RAM contengono, in posizione nascosta, un disco rigido magnetico e un alimentatore di riserva per la batteria. Nel caso di un'interruzione della corrente elettrica, il controllore del disco RAM copia i dati dalla RAM sul disco magnetico, per poi eseguire l'operazione inversa al ripristino della corrente elettrica. Una variante del disco RAM è rappresentata dalla memoria flash, impiegata in vari prodotti, dalle macchine fotografiche agli assistenti digitali personali (PDA) e ai robot; la sua diffusione come memoria rimovibile nei computer di uso generale è in costante aumento.

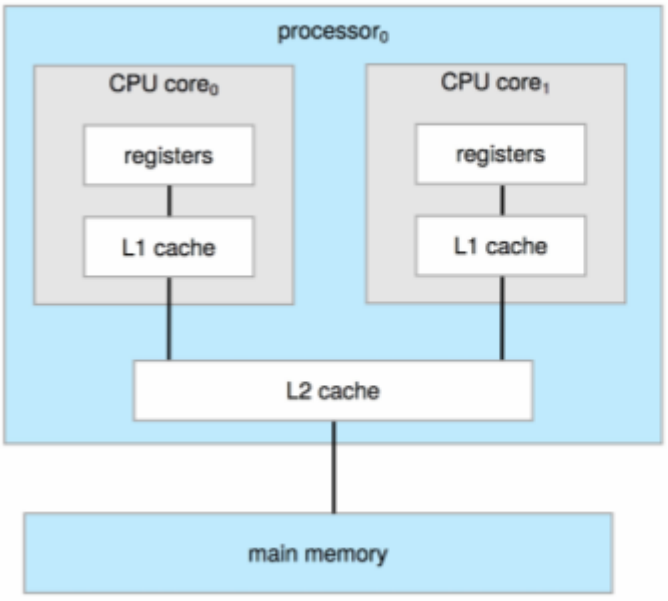
Multi-Core

I processori multi-core si distinguono in base al fatto se hanno la cache condivisa o privata: nel primo caso lo svantaggio consiste nel fatto dimensionale, mentre nel secondo caso si verifica una minor contesa sul bus.

- Cache privata
  - Affinità
  - Minore contesa sul bus
- Cache condivisa
  - Condivisione dati tra core diversi
  - Cache più grandi per thread se ci sono pochi thread

Se due (o più) core chiedono contemporaneamente di accedere in memoria centrale, uno dei due dovrà aspettare l'altro, quindi il tempo di accesso alla memoria centrale (come minimo) raddoppia, indipendentemente da dove si vuole andare a lavorare. Di solito il SO fa in modo che i processi lavorino in parti diverse della memoria in modo che non si diano fastidio fra di loro (salvo alcuni casi), quindi tipicamente non vanno a chiedere operazioni sulla stessa cella di memoria. Rimane il fatto che il bus è quello e quindi uno dei due esegue l'operazione mentre l'altro aspetta.

→ **SOLUZIONE** : ogni core è dotato di una sua cache privata, dove tiene i dati che servono a lui.

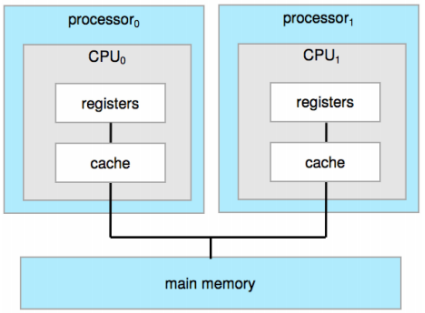


L2 cache:
memoria cache di secondo livello condivisa da tutti i core. Si mette per velocizzare le operazioni

Multi Processori

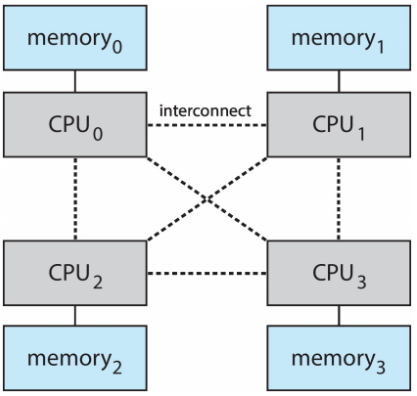
- **UMA (Uniform Memory Access)**: Supporta al max 16-32 CPU, e prevede che diversi processi con una cache privata possano accedere a moduli di memoria attraverso una interconnessione di rete.

  L'accesso alla memoria è uniforme : dal processore si accede a uno qualsiasi di questi blocchi di memoria con lo stesso tempo (ci possono essere dei conflitti nell'accesso alla memoria). All'aumentare del numero dei processori aumenta la probabilità che ci siano dei conflitti, cioè che un altro nodo ha la necessità di utilizzare la memoria, quindi i tempi di attesa aumentano. Oltre un certo numero di nodi non si può andare, altrimenti i tempi di accesso diventano mediamente troppo lunghi (numero di nodi limitato superiormente).

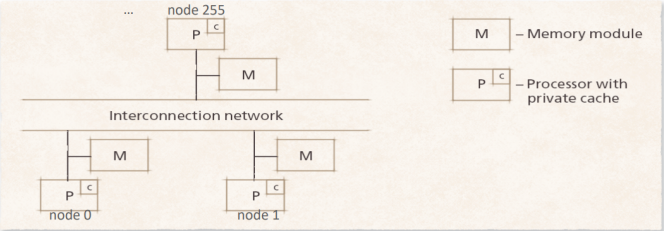


UMA

- **NUMA (Non Uniform Memory Access)**: Essa prevede diversi processori con una cache privata e con un collegamento diretto ad un modulo di memoria riservata, ogni processore interconnesso con gli altri. Questa architettura prevede un massiccio scambio di messaggi al fine di mantenere traccia di quale cache contiene un dato blocco. Rilassiamo il vincolo di avere il tempo di accesso alla memoria uguale per tutti: organizziamo la nostra architettura su 2 livelli di bus, uno di sistema e uno privato per ogni nodo, dove su quest'ultimo è connesso un modulo di memoria riservato.



NUMA



Coerenza dalla cache

Problema

Nel momento in cui abbiamo un'architettura con più nodi, non possiamo escludere la possibilità che più nodi contemporaneamente debbano andare a lavorare sulla stessa porzione di memoria.

Finché di questa porzione di memoria ne esiste una sola copia ed è in memoria centrale, l'unico problema che si può creare è un problema di conflitto nell'accesso a questa porzione di memoria.

Nel momento in cui ogni nodo ha una sua memoria di cache e quindi può portare quella porzione di memoria nella sua memoria di cache (e quindi andare a modificare la sua copia personale di quel blocco di dati), si crea un secondo problema: evitare che ogni nodo lavori con una copia diversa dello stesso blocco di memoria (*mantenere coerente l'informazione*).

Bisogna introdurre dei meccanismi per mantenere la coerenza dei dati presenti nelle varie memorie di cache dei vari nodi.

Meccanismi di coerenza

Per meccanismo di coerenza si intende come fa un nodo ad accorgersi di quale è lo stato del blocco di memoria mantenuto nella propria memoria di cache.

- **Una cache per ogni dato**: un solo processore alla volta può avere in cache un particolare dato.

  Utizzabile in *UMA* e *NUMA* (è più semplice, ma anche inefficiente e inefficace).

- Se si vuole dare la possibilità di tenere i dati in più nodi, a questo punto si deve considerare lo stato della memoria. Ci sono tre metodi:

1. **Bus snooping** (“ficcare il naso nel bus”): il controller della cache di ogni nodo va a controllare (“spiare”) quali sono i comandi che passano sul bus; quando vede che c’è il riferimento ad un’operazione di modifica ad una porzione di memoria che è anche nella memoria cache, a quel punto sa che la sua copia del dato è obsoleta e deve fare qualcosa. Questo significa che ogni controller della cache di ogni nodo, oltre ad avere nella cache i blocchi di memoria, ha anche un flag che gli dice se il blocco può utilizzarlo o no. Il controller della cache di ogni nodo ha una sua tabellina che tiene lo stato solo dei blocchi che in quel momento sono nella sua cache e per ogni nodo dice il suo stato.
   Utizzabile in
   UMA e *NUMA*, ma in NUMA potrebbe perdere di efficienza.
2. **Centralized directory**: il sistema mantiene una directory centralizzata per sapere in quali cache è copiato e quale è lo stato di un determinato dato.

   Utizzabile in *UMA* e *NUMA* a patto che i nodi non siano troppi.
3. **Home-based directory**: la directory non è centralizzata ma gestita in ogni nodo, ogni nodo gestisce i propri blocchi di memoria. Per ogni nodo c’è scritto in quali nodi quel blocco si trova in quel momento.

   Utilizzabile solo in *NUMA*.

Protocolli di coerenza

Una volta che siamo in grado di sapere se quella pagina che un nodo sta modificando ce l'ha qualcun altro, che approcci possiamo utilizzare? Come possiamo intervenire ?

Il protocollo riguarda come viene cambiato lo stato della cache.

Ogni nodo associa ad un blocco di memoria uno stato che indica la disponibilità del blocco stesso per quel nodo.

Il controllore di ogni cache si deve attenere al protocollo scelto :
- osservando gli
eventi : richieste che provengono dal proprio processore o richieste che transitano sul bus (richieste che provengono da altri nodi).
- eseguendo
azioni sul bus.

- cambiando lo
stato locale del blocco interessato (un blocco può trovarsi in stati diversi su nodi diversi).

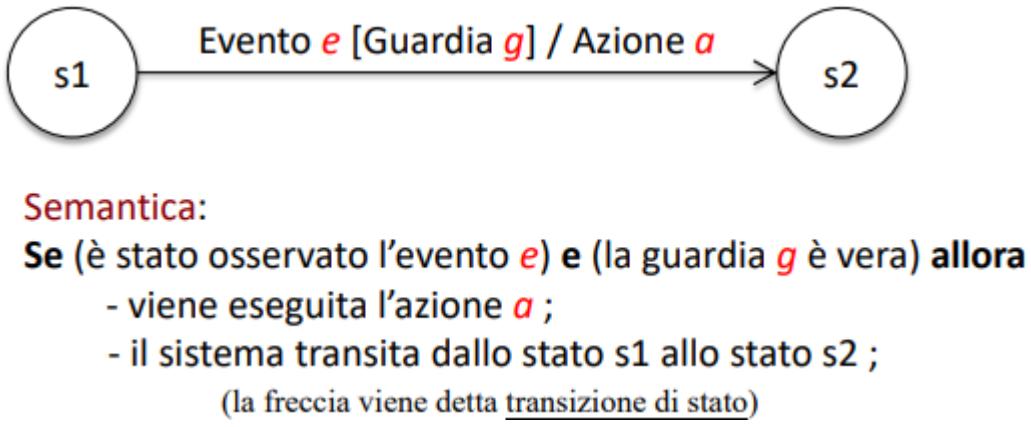
I protocolli sono di due categorie:

- Protocolli basati sul concetto di **invalidazione**: quando un blocco viene modificato da un nodo, lo stato di quel blocco negli altri nodi è settato ad *invalido*. (*MSI* e *MESI*)
- Protocolli basati sull'idea di **aggiornamento**: quando un processore modifica una locazione condivisa il suo valore viene aggiornato nelle cache di tutti i processori che possiedono quel blocco; per questo motivo i protocolli basati sull'aggiornamento hanno un miss-rate più basso ma generano più traffico nel bus, risultando più complessi nella gestione. (*DRAGON*)

Diagrammi di stato

I protocolli possono essere formalizzati mediante i diagrammi di stato.

Un cambiamento di stato viene rappresentato mediante una transizione di stato.



Evento: qualcosa che viene osservato dal sistema (es: input che viene dato al programma) che stiamo descrivendo mediante questo diagramma di stato e come conseguenza dell'aver osservato quell'evento può esserci un cambiamento di stato. Nel nostro caso specifico l'evento è andare a vedere che cosa è successo sul bus (che tipo di segnale di controllo sta passando in quel momento sul bus), oppure ricevere una richiesta dal processore.

Guardia: espressione booleana che serve nei diagrammi di stato perché un nodo può essere interessato a sapere in che stato si trovano gli altri nodi

(es: è inutile mandare un segnale di invalidazione se in quel momento sono l'unico ad avere una copia di quel blocco di memoria).
-Evento: io voglio modificare quel blocco di memoria.
-Guardia: condizione "non ci sono altri nodi che hanno una copia di quel blocco di memoria".
(Vado a finire in uno stato in cui dichiaro di essere l'unico ad avere una copia di quel blocco di memoria)

Azione: quello che deve fare il nostro sistema, una volta che ha osservato un evento *e* ed ha visto che la condizione *g* è vera
(Risultato: il sistema finisce nello stato s2)

Protocolli di invalidazione

`Pr` significa che l'evento/azione viene osservato/eseguita sul proprio processore.

`Bus` significa che l'evento/azione viene osservato/eseguita da un altro processore.

- Richieste del processore:

- `PrRd` : lettura.
- `PrWr` : scrittura.

- Azioni sul bus:

- `BusRd` : quando si verifica una lettura mancata nella cache di un processore, il controllore della sua cache invia l'indirizzo al bus e chiede una copia del blocco che non intende modificare;
- `BusRdX` : quando si verifica una scrittura mancata nella cache di un processore, il controllore della cache invia l'indirizzo al bus e chiede una copia del blocco che intende modificare, richiesta che invalida il blocco nelle cache degli altri processori.
- `BusUpgr` : quando c'è un hit di scrittura nella cache di un processore, invia una richiesta *BusUpgr* sul bus per invalidare il blocco nelle cache degli altri processori.

- `Flush` : un intero blocco della cache viene inviato attraverso il bus, viene ricevuto dal controller della cache che lo ha richiesto e dal controller della memoria centrale.

Protocollo MSI

Tipo di protocollo di invalidazione, che prevede 3 stati.

Ogni controllore della cache segue questo protocollo di coerenza.

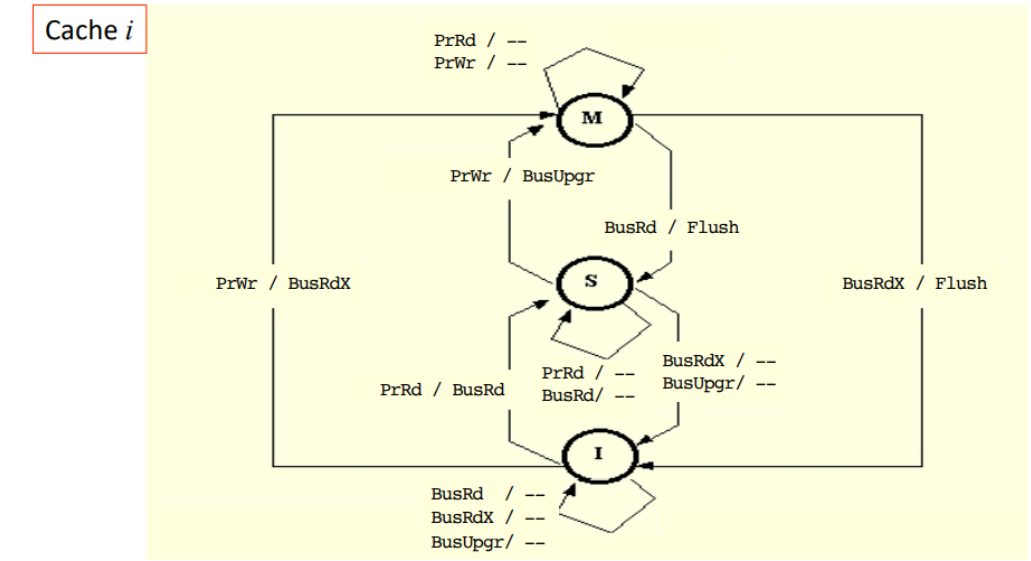
Stati: utilizza tre stati per identificare la situazione di un blocco di memoria in una cache:

- **I (invalid)** : il blocco di memoria non è valido; io ho una copia obsoleta del dato (perché qualcun altro l'ha modificato) oppure non ho proprio la copia del dato nella mia cache.
- **S (shared valid)** : il blocco è presente in uno stato non modificato in una o più cache; le copie in memoria centrale e nelle altre cache in S sono tutte aggiornate.
- **M (modified)** : solo un processore ha una copia valida nella sua cache, la copia in memoria centrale è obsoleta e nessun'altra cache ha una copia valida.

Eventi: `PrRd` , `PrWr` , `BusRd` , `BusRdX` , `BusUpgr`

Guardie: in questo protocollo non vengono utilizzate guardie.

Azioni: `BusRd` , `BusRdX` , `BusUpgr` , `Flush`



Le frecce indicano le transizioni di stato: sono etichettate in qualche modo. Sono state etichettate con una prima stringa di caratteri, una barra e un'altra stringa. La prima stringa è un evento: ovvero il controller osserva che il suo processore ha fatto una certa operazione, oppure che nel bus sta passando un certo comando. L'action invece è un'azione fatta proprio dal controller, rappresentata dalla seconda stringa.

-Da I a S:
Il controller osserva che il suo processore ha fatto richiesta di leggere quel dato (
`PrRd`). Il controller invia quindi una richiesta di lettura di quel dato (`BusRd`). Quel dato dunque va a finire nello stato S in quanto le copie di quel dato diventano almeno 2.

-Da I a M:
Il controller osserva che il suo processore ha chiesto un blocco di memoria per farci una modifica (
`PrWr`). Il processore non ha una copia aggiornata di quel dato, dunque prima di effettuare un'operazione di scrittura deve ottenere la copia aggiornata. Allora il controller effettua una operazione di read esclusiva (`BusRdX`), invalidando quel blocco di memoria nelle caches degli altri processori. Un blocco di memoria rimane *Invalid* fino a quando non mi serve.
Una volta che eseguo un'operazione e il blocco è nello stato di *Invalid*, il suo stato finale non potrà mai ricadere nello stato di *Invalid*. Qualora il processore richieda un dato che non è presente nella sua cache, lo carica in memoria e il suo stato iniziale è S: il comando sarà un `BusRd`. Se si vuole modificare quel dato, esso viene letto con un diverso comando, ovvero `BusRdX` e il blocco viene posto in stato di *Modified*.

Il problema del protocollo MSI è che quando sono nello stato *(S)* in realtà potrei essere l'unico ad avere una copia di quel blocco, così come potrebbero esserci anche altri nodi che hanno una copia di quel blocco.

Questo è un problema perchè quando modifico quel blocco mando un messaggio di `BusUpgr` per invalidare le copie degli altri nodi e in alcuni casi questo può essere del tutto inutile se sono l'unico ad avere una copia.
Si è pensato di dividere lo stato (S) (shared valid) in due stati diversi:

- (S) (shared o valid)
- (E) (exclusive)

Protocollo MESI

Tipo di protocollo di invalidazione, che prevede 4 stati.

Stati: il protocollo MESI utilizza quattro stati:

- **I (invalid)**: come per MSI, ovvero il blocco di memoria non è valido.
- **S (shared o valid)** : due o più processori hanno questo blocco nella loro cache in uno stato non modificato. La memoria centrale è aggiornata.
- **E (exclusive**): solo una cache ha una copia del blocco ed essa non è stata modificata. La memoria centrale è aggiornata; quindi se il controllore della cache riceve una richiesta di scrittura dal proprio processore (`PrWr`) non deve invalidare nessuno (non deve eseguire `BusUpgr`), risparmiando tempo.
- **M (modified)**: come per MSI, ovvero solo un processore ha una copia valida del blocco nella sua cache; la copia in memoria centrale è obsoleta e nessun altra cache ha una copia valida.

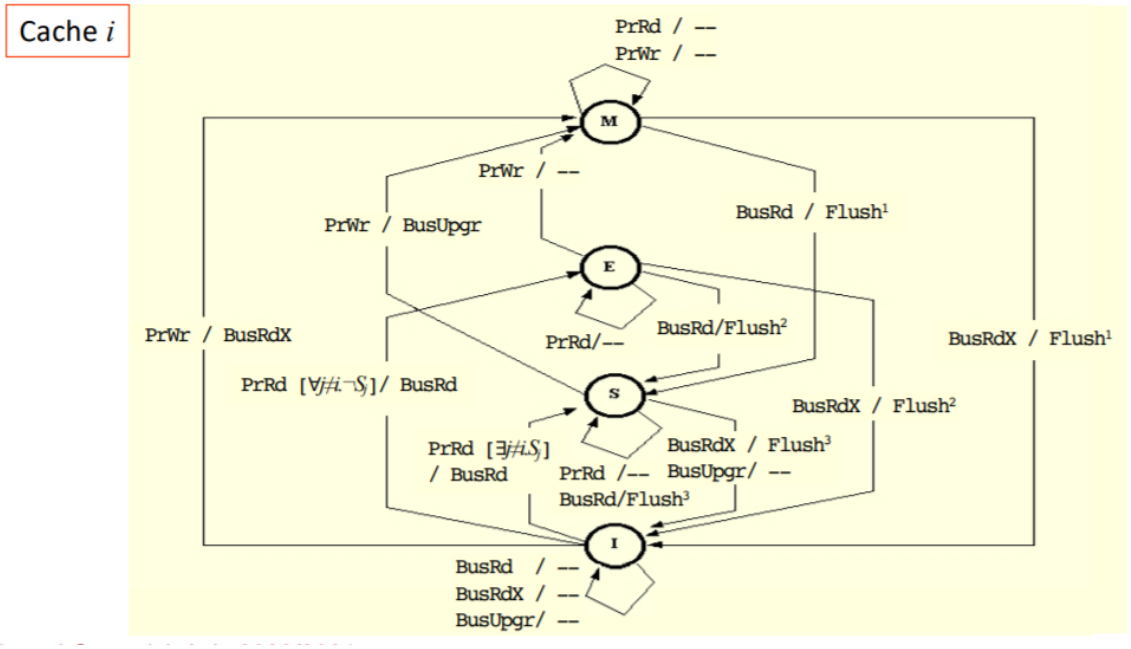
Eventi: `PrRd` , `PrWr` , `BusRd` , `BusRdX` , `BusUpgr`

Guardie:

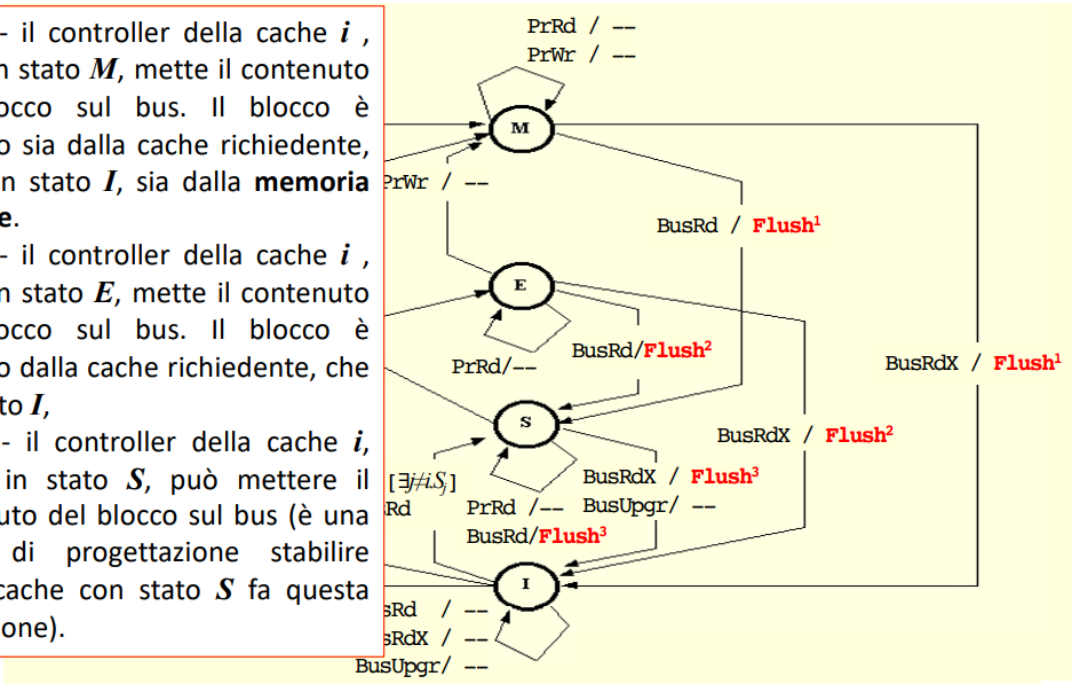
- $[\forall\ j \neq i\ |\neg\ S_j]$: nessun'altra cache ha una copia aggiornata del blocco
- $[\exists\ j \neq i\ |\ S_j]$: esiste almeno un'altra cache con una copia aggiornata del blocco

Azioni: `BusRd` , `BusRdX` , `BusUpgr` , `Flush`

In questo caso lo stato *S* dell'MSI è stato suddiviso in due stati, *S* ed *E*, perchè prima quando lo stato era solo *S* e il processore chiedeva di modificare il valore del blocco *S*, si doveva inviare un segnale sul bus per invalidare i blocchi degli altri. Però tale azione ha senso solo se qualcun altro ha quel blocco. Perciò se gli altri non hanno quel blocco, non si deve inviare nulla (e viene appunto individuato dallo stato *E*) così da ridurre il traffico nel Bus.



- **Flush¹** - il controller della cache ***i*** , che è in stato ***M***, mette il contenuto del blocco sul bus. Il blocco è ricevuto sia dalla cache richiedente, che è in stato ***I***, sia dalla **memoria centrale**.
- **Flush²** - il controller della cache ***i*** , che è in stato ***E***, mette il contenuto del blocco sul bus. Il blocco è ricevuto dalla cache richiedente, che è in stato ***I***,
- **Flush³** - il controller della cache ***i***, che è in stato ***S***, può mettere il contenuto del blocco sul bus (è una scelta di progettazione stabilire quale cache con stato ***S*** fa questa operazione).



Lo stato in cui si può trovare un blocco di memoria
Shared nell'MSI non teneva conto se quel blocco di memoria era effettivamente presente in altre memorie di cache, infatti *Shared* significa che il blocco di memoria è stato caricato sulla memoria cache di un nodo senza modificarlo o senza alcuna richiesta di modifica.
Questa variante suddivide lo
Shared in 2 stati:
- uno si chiama ancora

Shared, solo che significa che il blocco di memoria è sicuramente posseduto da qualche altro nodo.
- l'altro è
l'Exclusive, ovvero quel blocco di memoria è posseduto solo da quel nodo.

Supponiamo che un nodo carichi nella sua cache un blocco: siamo nello stato di *Exclusive*.
Se il controller della cache osserva una richiesta di lettura da parte del proprio processore, cosa deve fare? Nulla, in quanto gia possiede quel blocco! Rimane nello stato
Exclusive.
Ipotizziamo che il controller osservi un'operazione di richiesta di scrittura, allora il blocco diventa
Modified, ma non fa comunque nulla perchè non deve avvertire gli altri nodi che quel blocco deve essere invalidato in quanto è l'unico ad avere quel blocco.
Supponiamo che osservi una richiesta da parte di qualche altro nodo di andare a leggere quel blocco: a quel punto non è piu l'unico ad avere quel blocco, dato che c'è un altro nodo ad averlo: il blocco diventa
Shared.
Supponiamo che il controller osservi una richiesta di modifica da parte di un altro nodo: non solo lui non è piu il proprietario esclusivo di quel blocco, ma a quel punto la sua copia non è piu valida perchè un altro nodo l'ha letto e l'ha modificato.

Protocolli basati sull'aggiornamento

-Richieste del processore:

- `PrRd` : lettura; `PrRdMiss` : lettura di un nuovo blocco (non in cache)
- `PrWr` : scrittura; `PrWrMiss` : scrittura di un nuovo blocco (non in cache)

-Azioni sul bus:

- `BusRd`: il controllore della cache invia l'indirizzo al bus e chiede una copia del blocco che non intende modificare (in caso di ricerca fallita nella cache);
- `BusUpd`: è una nuova transazione che prende una parola specifica scritta dal processore e la trasmette al bus così che tutte le altre cache si possano aggiornare. Ogni `BusUpd` è seguito da un'azione di `Update` eseguita dalle cache che devono aggiornare il blocco.

- `Flush`: il controllore della cache può mettere il contenuto del blocco referenziato sul bus, piuttosto che permettere alla memoria centrale di fornire i dati.

Protocollo DRAGON

Protocollo basato sull'aggiornamento.

Ha 4 stati:

- **M (modified)** : io sono l'unico ad avere la copia modificata.
- **E (exclusive)** : solo quel nodo ha una copia di quel dato, e quella copia non è stata modificata.
- **SC (shared-clean)** : potenzialmente due o più processori hanno questo blocco nella loro cache. La memoria centrale può essere o non essere aggiornata.
- **SM (shared-modified)** : potenzialmente due o più processori hanno questo blocco nella loro cache. La memoria centrale non è aggiornata ed è responsabilità di tale processore aggiornare la memoria centrale quando il blocco è rimpiazzato.

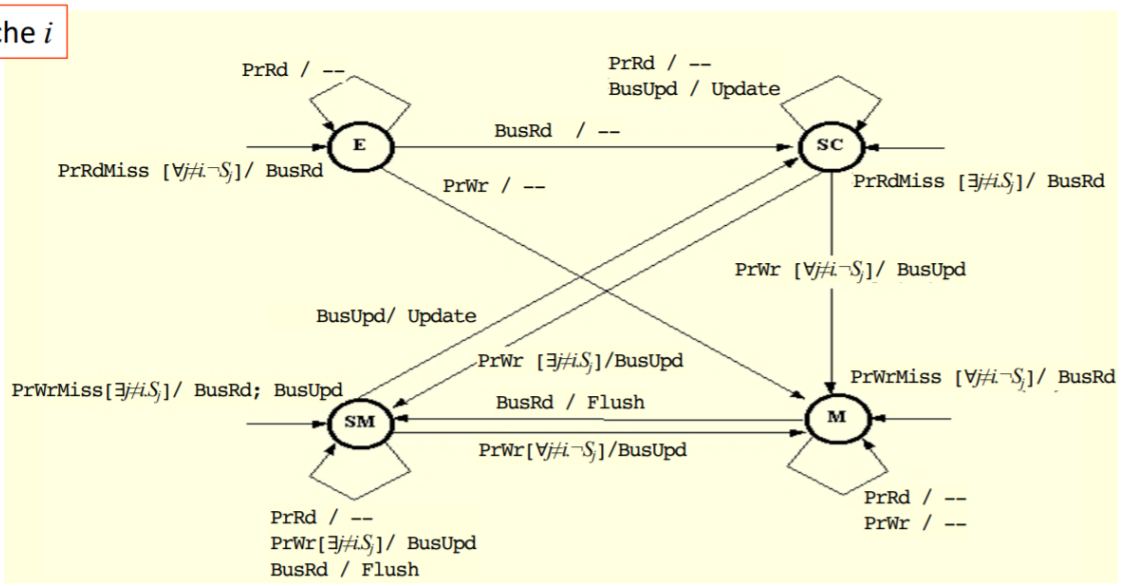
Eventi: `PrRdMiss`, `PrWrMiss`, `PrRd`, `PrWr`, `BusRd`, `BusUpd`

Guardie:

- $[\forall\ j \neq i\ |\neg\ S_j]$: nessun'altra cache ha una copia aggiornata del blocco
- $[\exists\ j \neq i\ |\ S_j]$: esiste almeno un'altra cache con una copia aggiornata del blocco

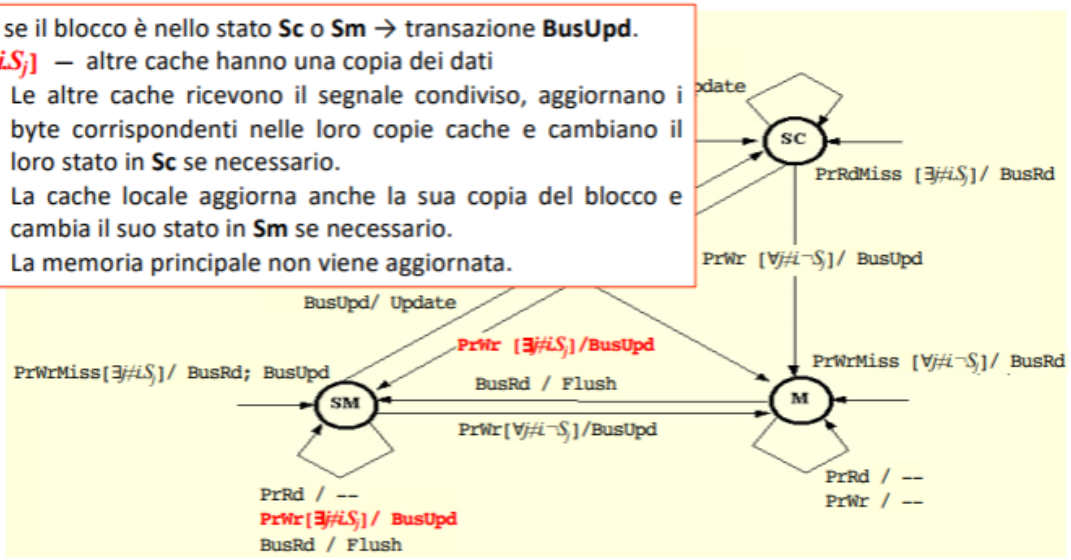
Azioni: `BusRd`, `BusUpd`, `Flush`, `Update`

Cache *i*



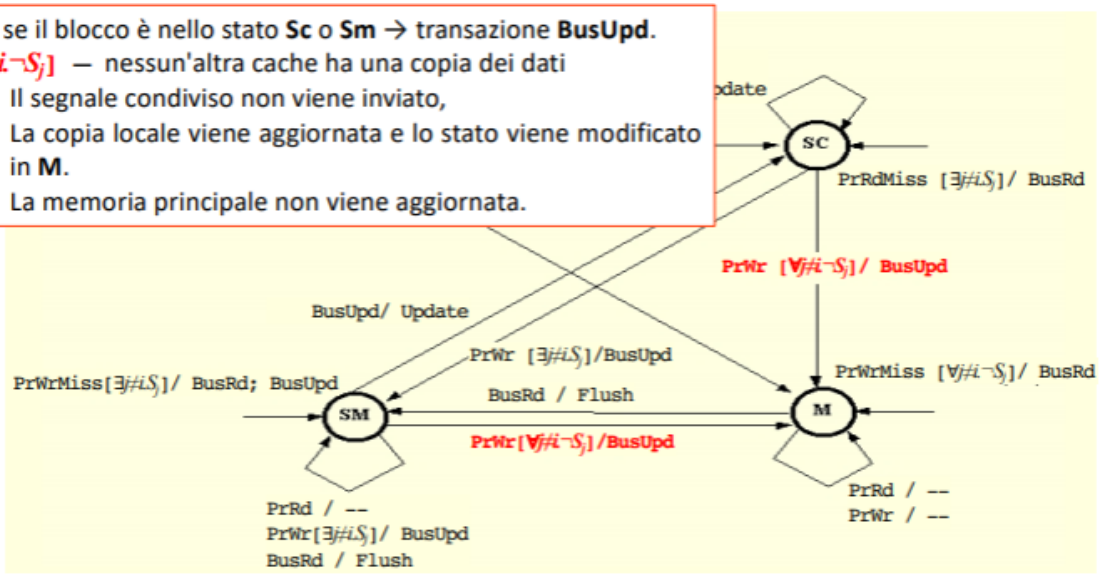
PrWr - se il blocco è nello stato **Sc** o **Sm** → transazione **BusUpd**.

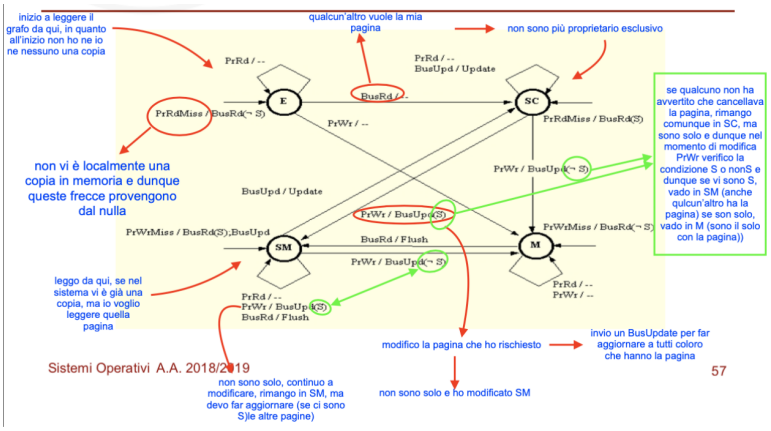
- **[∃j≠i.S_j]** — altre cache hanno una copia dei dati
  - Le altre cache ricevono il segnale condiviso, aggiornano i byte corrispondenti nelle loro copie cache e cambiano il loro stato in **Sc** se necessario.
  - La cache locale aggiorna anche la sua copia del blocco e cambia il suo stato in **Sm** se necessario.
  - La memoria principale non viene aggiornata.



PrWr - se il blocco è nello stato **Sc** o **Sm** → transazione **BusUpd**.

- $[\forall j \neq i \neg S_j]$ — nessun'altra cache ha una copia dei dati
  - Il segnale condiviso non viene inviato,
  - La copia locale viene aggiornata e lo stato viene modificato in **M**.
  - La memoria principale non viene aggiornata.

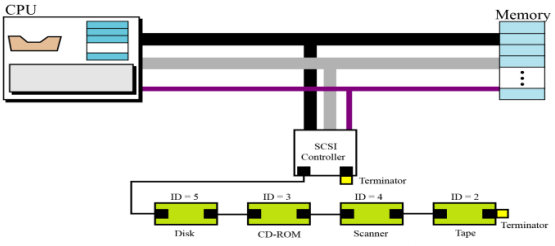




Organizzazione dei Dispositivi di I/O

Architettura dei dispositivi I/O

SCSI CONTROLLER : I vari dispositivi di I/O sono messi in serie, ciò implica che per accedere all'ultimo dispositivo vanno attraversati tutti gli altri. Può avere più porte alle quali possono essere collegate catene di dispositivi messi in sequenza tra loro; questo comporta una grande flessibilità nel progettare un sistema di elaborazione perchè il numero di dispositivi che possiamo agganciare

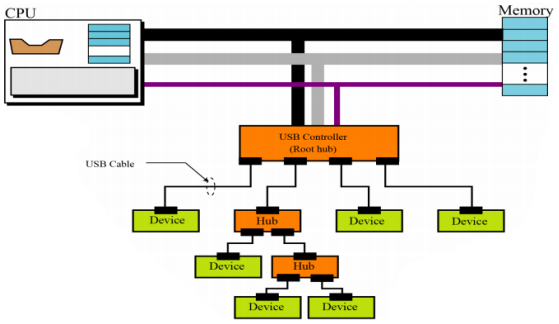


al sistema di elaborazione può essere maggiore al numero di porte presenti sulla macchina.

Svantaggi: i tempi di latenza possono essere maggiori.

FIREWIRE CONTROLLER: i dispositivi sono soggetti a una struttura di tipo albero (Nodo-Radice)

USB CONTROLLER: Si ha una struttura ad albero anche in questo caso, con un accesso ai vari più veloce rispetto al FIREWIRE.
Come lo SCSI ha il vantaggio di agganciare dispositivi periferici diversi. Il vantaggio maggiore rispetto allo SCSI è che i dispositivi non devono essere connessi fra di loro in sequenza, ma ad ogni porta USB possiamo agganciare un dispositivo oppure possiamo agganciare un hub che moltiplica le porte a disposizione e quindi alla fine la struttura è tipo ad albero (in figura: 6 dispositivi connessi con 4 porte a disposizione)



Ogni dispositivo è collegato al bus di sistema tramite un'interfaccia la quale contiene alcuni registri (di controllo, o al fine di mantenere la sincronizzazione CPU-Dispositivo, buffer per contenere i dati in IN/OUT). Questa interfaccia è chiamata **controller**.

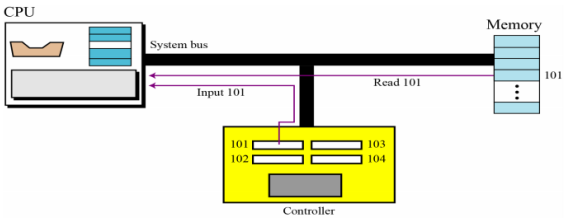
La CPU colloquia con il controller scrivendo/leggendo i suoi registri:

- utilizzando apposite istruzioni **IN** / **OUT** che avranno come parametro un "indirizzo" che identifica un particolare controller

- utilizzando normali istruzioni **LOAD**/**STORE** a indirizzi associati ai registri del controller: questa tecnica si chiama Memory Mapped I/O

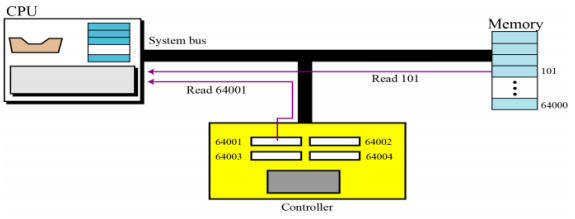
Per comunicare abbiamo due tecniche diverse:

- **ISOLATED I/O ADRESSING**: Se abbiamo delle istruzioni specifiche per i dispositivi periferici, chiediamo un'operazione di Input con un determinato indirizzo che è l'indirizzo che ci permette di identificare il dispositivo associato a quel controller. Nel bus indirizzi viaggia un indirizzo che è l'identificativo del dispositivo. Usa due tipi di istruzione: Input,Read che avranno come parametri l'indirizzo che identifica un particolare controller. Avendo gli stessi indirizzi ma in contesti diversi, devo avere 4 operazioni una per il rapporto CPU-Memory e l'altro per CPU-Periferica.



- **MEMORY MAPPED I/O ADRESSING**:

  Utilizza istruzioni di tipo load/store a indirizzi associati ai registri del controller, ed è come avere 2 classi di indirizzi diversi, con solo due istruzioni riesco a leggere/scrivere tra memoria-cpu o cpu-periferica, con questa soluzione devo poter distinguere, tramite l'indirizzo di memoria o di periferica.



Io ho un unico spazio degli indirizzi in cui inserisco tutti gli indirizzi di tutte le locazioni della memoria centrale e tutti gli identificatori dei vari dispositivi, e ho a disposizione un'unica istruzione

Vantaggio: set di istruzioni ridotto, quindi la costruzione della CPU risulta essere molto più semplice.

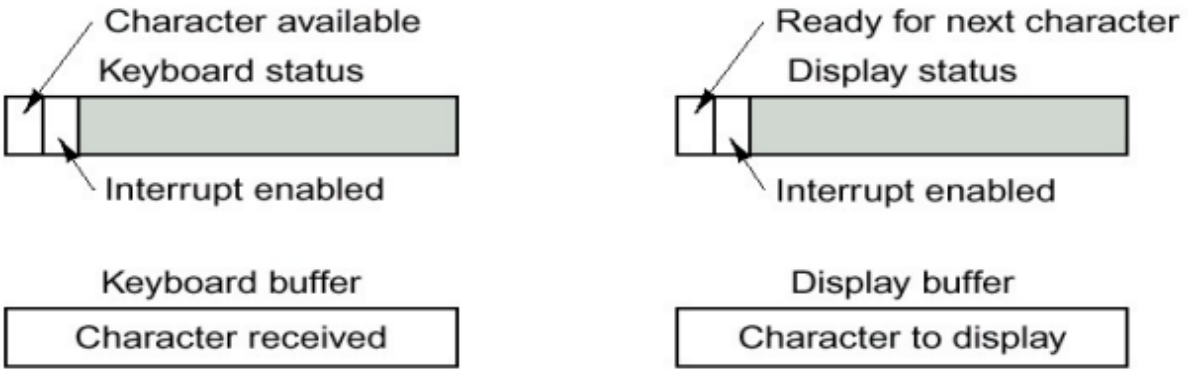
Svantaggio: con l'approccio precedente abbiamo due spazi indirizzi distinti, mentre qui lo spazio degli indirizzi è unico e quindi lo devo dividere tra locazioni di memoria e periferiche.

I vari controller come fanno a sapere se la richiesta riguarda loro o qualcun'altro?

Ovviamente ogni controller sa qual è il range di indirizzi che gli è stato assegnato, va a vedere sul bus indirizzi qual è l'indirizzo : se è uno dei suoi allora sa che è lui che deve intervenire; se invece è al di fuori del range che gli è stato assegnato sa che è una richiesta che deve ignorare perchè ci sarà qualcun altro che risponderà a quella richiesta.

Tipi di I/O

I/O Programmato (con busy waiting)



Il meccanismo di busy waiting è funzionale se un sistema è monoprogrammato (in esecuzione un solo programma alla volta)

```
public static void output_buffer (int buf[], int count) {
    int status, i, ready;
    for (i= 0; i < count; i++) {
        do {
            status = in(display_status_reg);  //
            ready = (status << 7) & 0x01;     //  Busy Wait
ing (attesa attiva)
        } while (ready == 1);               //
        out (display_buffer_reg, buf[i]);
    }
}
```

La CPU, in attesa che venga eseguita l'operazione di I/O, va a controllare continuamente se tale operazione che ha richiesto è stata eseguita oppure no.

Nel momento in cui si è passati a sistemi multiprogrammati questo meccanismo è risultato inefficiente.

Si è pensato a una soluzione alternativa: se in questo momento il processore sta eseguendo le istruzioni del processo 1 ed il processo 1 ad un certo punto chiede l'esecuzione di un'operazione di I/O, il processore svolge la richiesta di un'operazione di I/O come parte delle istruzioni del processo 1. Non appena il processore chiede questa operazione al dispositivo di I/O, il mio processo 1

viene sospeso (viene fatto uscire dalla CPU, è in attesa del risultato dell'operazione che ha richiesto) e assegniamo la CPU ad un altro processo; quindi adesso all'interno della nostra CPU abbiamo in esecuzione le istruzioni del processo 2 (e via dicendo...).

Nel frattempo il dispositivo che stava eseguendo l'operazione richiesta dal processo 1 va avanti con l'operazione; ad un certo punto l'operazione richiesta dal processo 1 verrà eseguita (terminerà l'esecuzione di tale operazione).

Il mio processo 1 però è sospeso (= non è in esecuzione su nessuno dei processori); *come fa a sapere che l'operazione di I/O che aveva richiesto è stata completata? E soprattutto, come fa il SO a sapere che a quel punto può risvegliare il processo 1 e farlo ripartire da dove era stato sospeso?*

I/O Guidato da Interrupt

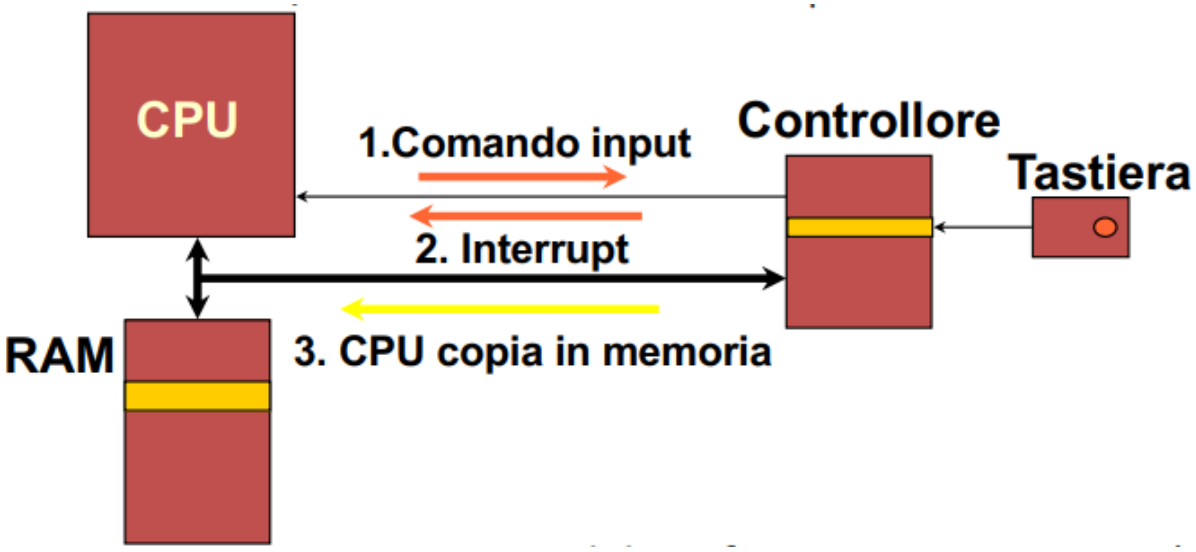
Partendo dai problemi analizzati sopra possiamo affermare quanto segue.

Il dispositivo deve inviare un segnale che permetta al SO di capire che l'operazione chiesta dal processo 1 è terminata; posso dunque risvegliare il processo 1.

Però in questo momento la CPU sta eseguendo le istruzioni (per esempio) del processo 2 (non c'è il SO in esecuzione sul processore); quindi bisogna che in qualche modo vengano eseguite le istruzioni del SO sul processore.

Allora questo segnale, inviato dal dispositivo per dire che ha terminato, interrompe l'esecuzione del processo 2 per permettere alla CPU di eseguire le istruzioni del SO, che a sua volta si rende conto che il segnale che ha ricevuto viene dal dispositivo che stava eseguendo l'operazione del processo 1, riferendogli che l'operazione richiesta è terminata.

Per chiedere il coinvolgimento del SO bisogna interrompere l'esecuzione del processo che in quel momento sta utilizzando la CPU → **Meccanismo basato sull'Interruzione.**



Dato che CPU e dispositivi di I/O hanno velocità molto diverse, è opportuno non tenere la CPU bloccata in un ciclo di controllo dei bit di stato, in attesa che termini l'operazione richiesta. Se si potesse evitare il busy waiting mentre i dispositivi di I/O lavorano, la CPU potrebbe essere impiegata in compiti più utili (per esempio eseguire una altro programma).

Prima era la CPU che controllava nel registro del controller del dispositivo lo stato del dispositivo stesso; se vedeva che il lavoro era completato poteva proseguire e lavorare in un altro dispositivo (*Busy Waiting*).

Nel processo di Interrupt : siamo nella situazione in cui abbiamo assegnato la CPU ad un altro processo, il SO deve essere informato che la richiesta è terminata e deve sapere che può far ripartire il processo che aveva sospeso; il dispositivo allora comunica alla CPU (e di conseguenza al SO) di poter andare avanti, e lo fa attraverso il bus di controllo.

Il problema è che quando il controller del dispositivo invia un segnale alla CPU, la CPU già sta lavorando ad un altro processo. Per gestire tutto ciò è stato introdotto il concetto di **interruzione** : il dispositivo, quando ha terminato, manda un segnale alla CPU che interrompe le funzioni che stava svolgendo; ogni volta che arriva un segnale di interruzione da un qualsiasi dispositivo la CPU si deve interrompere, analizzare l'interruzione ed agire di conseguenza.

Questo comporta una serie di problemi:

- Come funziona nel caso ci siano più dispositivi I/O che possono inviare segnali di interrupt ?

1) i segnali provenienti dai diversi dispositivi vengono messi in OR

2) il dispositivo che ha causato interrupt viene individuato tramite *Polling* oppure utilizzando la tecnica dell'*Interrupt Vettorizzato*.

- **Polling** : vengono interrogati tutti i dispositivi per trovare quello che ha inviato il segnale; l'ordine di scansione determina anche un ordine di importanza.

  Lo svantaggio è dato dal tempo di domanda/risposta tra CPU e dispositivo.

- **Interrupt Vettorizzato** *:* la CPU col codice che ha ricevuto accede ad un vettore (il codice è quindi l'indice del vettore), detto *vettore degli interrupt*, e nell'elemento corrispondente dell'array c'è l'indirizzo della prima istruzione del SO che deve gestire quella interruzione; quell'indirizzo viene caricato nel PC e da quel momento va in esecuzione la parte del SO che deve gestire quell'interruzione (chiamata *Routine di gestione dell'interruzione*).

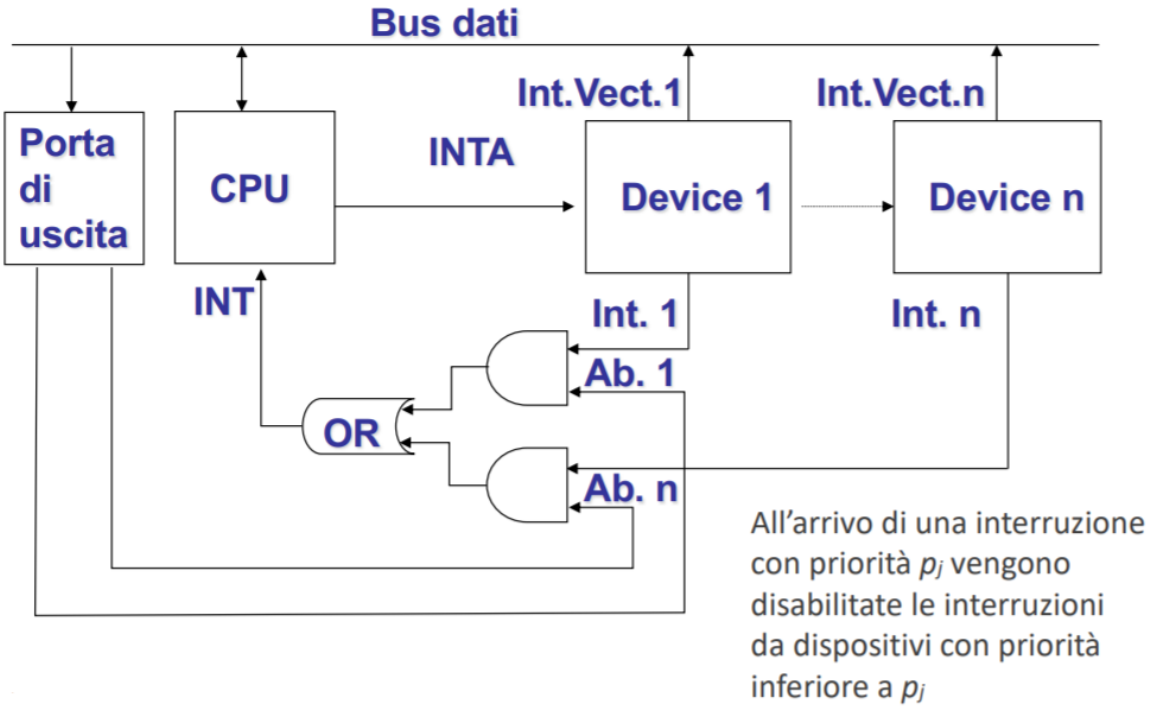
- Cosa accade se arrivano più interrupt contemporaneamente ?

La CPU avverte un solo segnale (*OR* di tutti i segnali di interrupt)

1) se si usa il *Polling*, l'ordine di scansione determina un ordinamento implicito tra i dispositivi.

2) se si usa l'*Interrupt Vettorizzato*, occorre impostare un protocollo di comunicazione tra CPU e dispositivi : quando la CPU ha ricevuto l'interrupt (cioè l'*INT*) ed è pronta a gestirlo, invia un *segnale di Ackowledge* (*INTA*). Quando il controller riceve il segnale invia l'identificatore del dispositivo (vettore di interrupt) sul bus dei dati. Se il segnale di INTA viene propagato sequenzialmente dai controllori dei dispositivi (collegamento *daisy chain*) e assorbito dal primo controller che aveva un interrupt pendente, si ha un ordinamento implicito tra i dispositivi determinato dal collegamento in daisy chain degli stessi.

Oss. 1) I dispositivi sono messi in un determinato ordine che determina la loro importanza: i più importanti vengono messi prima nella catena, quelli meno importanti in fondo. Questo vuol dire che nel momento in cui ci sono due dispositivi che hanno inviato nello stesso momento il segnale di interruzione, uno dei due due riceverà il segnale di acknowledge prima dell'altro e quindi sarà lui che risponderà e quindi la sua interruzione verrà gestita per prima.



Oss. 2) I segnali INT e INTA viaggiano sul bus di controllo (non è altro che una serie di segnali che mettono in comunicazione tra loro i vari dispositivi o i vari segnali).
Il codice che viene restituito dal dispositivo che ha inviato l'interruzione, essendo un vero e proprio dato, viene inviato sul bus dati.

Oss. 3) Se prima nel caso peggiore (con n dispositivi) avevamo un tempo di risposta che era proporzionale a $2n$ (segnale di richiesta e di risposta per ogni dispositivo, $2n$ segnali), adesso nel caso peggiore abbiamo n segnali (da CPU a dispositivo1, da dispositivo1 a dispositivo2,...).
Nel caso medio prima avevamo un tempo di risposta
$2n/2 = n$, adesso abbiamo un tempo $n/2$ (tempi dimezzati rispetto al Polling).

- Cosa accade se arriva un secondo interrupt mentre è in esecuzione la procedura di trattamento di un (primo) interrupt ?

La CPU vede che c'è un segnale di interruzione: interrompe il processo le cui istruzioni sono in esecuzione in questo momento, dopodiché invia un segnale di acknowledge che arriva al dispositivo1, il quale risponde tramite il bus dati con il codice relativo a quella interruzione. Esistono due possibili gestioni:
1) Deve essere possibile disabilitare gli interrupt per prevenire la perdita di

interrupt e più in generale quando deve essere eseguita una porzione di codice senza interruzioni.
In questo caso gli interrupt pendenti vengono ignorati fino alla riabilitazione degli interrupt. Non si possono avere interrupt annidati.
2)
Se vi sono segnali di interrupt con vincoli stringenti sui tempi di risposta, è opportuno poter definire diversi livelli di priorità e permettere ad un interrupt ad alta priorità di poter essere gestito anche se è in esecuzione una procedura di gestione di un altro interrupt (a più bassa priorità). Si possono avere interrupt annidati.

Le interruzioni che arrivano dai dispositivi in generale sono ***asincrone*** rispetto a quello che sta facendo la CPU in quel momento.

Cosa succede quando la CPU riceve un interrupt :

1. L'istruzione macchina in corso viene completata.
2. Si salva in memoria lo stata della computazione interrotta (registri e PC) nel PCB.
3. Si determina il tipo di interruzione che è avvenuta (tramite polling o tecnica dell'interrupt vettorizzato).
4. Si salta ad una posizione di memoria dove è contenuto l'*interrupt handler* (ovvero il gestore di quel tipo di interruzioni). Infatti segmenti di codice separati determinano quale azione deve essere eseguita per ogni tipo di interrupt.
5. Dopo aver gestito l'interrupt, viene ripristinato lo stato del programma interrotto e la sua esecuzione.

Tutto ciò deve essere TRASPARENTE rispetto a chi viene interrotto (il processo deve ripartire esattamente da dove era stato bloccato).

Quando è il momento giusto per interrompere l'esecuzione del programma che aveva il controllo della CPU in quel momento?

L'unico momento giusto per prendere in considerazione l'interruzione è prima della fase di *fetch*: ho appena finito l'esecuzione di un'istruzione, devo iniziare

l'esecuzione dell'istruzione successiva; qui in mezzo vado a controllare se ci sono interruzioni pendenti e se ci sono posso interrompere l'esecuzione del programma senza danneggiarlo, in modo tale da essere in grado di riprendere l'istruzione più tardi.

Timer : ogni determinato numero di cicli di clock invia un segnale di interruzione. L'importanza sta nel fatto che così siamo sicuri che ogni tanto arrivi un'interruzione.

Se noi vogliamo dare la possibilità periodicamente al SO (o meglio al Kernel) di riprendere il controllo della CPU per vedere se sta andando tutto bene o c'è qualcosa di anomalo, l'unico in grado di fare questo è il timer. Periodicamente va in esecuzione la routine di gestione di questo interrupt (che non è altro che un pezzo del kernel del nostro SO) e quindi il controllo ritorna momentaneamente nelle mani del SO che può fare tutti i controlli che vuole.

Direct Memory Access Structure (DMA)

Utilizzato dagli I/O ad alta velocità in grado di trasferire dati ad una velocità vicina a quella della memoria.

Il controller del dispositivo trasferisce blocchi di dati dal buffer direttamente alla memoria centrale senza l'intervento della CPU.

Viene generato un solo interrupt per blocco piuttosto che un interrupt per byte.

Poiché il bus di collegamento tra memoria e il controller DMA è condiviso, occore un arbitro che gestisca i conflitti evitando il furto di cicli (Cycle Stealing).
Per i dispositivi veloci, c'è il rischio di impegnare la CPU troppo tempo a gestire gli interrupt. Si abbandona la stretta struttura master-slave, dando più autonomia ai controller.
Aumentando i dispositivi DMA si possono avere degli intasamenti del bus di sistema.

I/O con Accesso diretto alla memoria

Se io devo scrivere una sequenza di caratteri, per ogni tasto che io premo c'è un'interruzione. Questo vuol dire che, quando il mio programma chiede un'operazione di lettura di una stringa di caratteri dalla tastiera, quel programma si sospende e viene data la CPU ad un altro programma; però poi arriva un'interruzione dovuta alla pressione di un carattere, quel programma

viene sospeso e viene letto quel carattere.
Dopodiché non è stata ancora completata la lettura della stringa di caratteri, quindi viene ridato il controllo al nostro programma; dopo un pò viene interrotto, viene letto il secondo carattere e viene ridato il controllo al programma ... fino a quando non viene premuto Invio: a questo punto risveglio il programma1 e magari la CPU viene data a questo programma.

Questo può andare bene fino a quando abbiamo a che fare con dispositivi lenti come la tastiera.

Consideriamo lo stesso meccanismo con un dispositivo molto più veloce, come la memoria di massa : siccome il dispositivo è molto più veloce, il programma2 non fa in tempo a riprendere l'esecuzione che arriva un'altra interruzione, quindi è più il tempo che la CPU passa a fare questi cotext switch che il tempo che passa a eseguire l'istruzione del programma2.

Paradossalmente quando il dispositivo è troppo veloce si creano problemi maggiori, perchè si rischia che a chiunque assegni la CPU quel processo non ha il tempo di iniziare a lavorare che viene subito interrotto.

Si è preferito fare dei controller con **accesso diretto alla memoria**: l'operazione di trasferire un carattere alla volta o una stringa alla volta in memoria centrale fino a quando l'operazione non è stata completata, invece di farla fare alla CPU (il che significa dover interrompere il processo a cui è stata data la CPU), diamo al controller la possibilità di andare a scrivere direttamente in memoria centrale; poi quando avrà finito di trasferire tutti i dati, e solo in quel momento, verrà inviata un'interruzione alla CPU per segnalare che tutti i dati relativi all'operazione richiesta sono stati trasferiti in memoria centrale e possono essere trovati lì; l'operazione è completata.

A questo punto il processo2 a cui è stata assegnata la CPU ha la possibilità di lavorare senza interruzioni fino a quando il trasferimento non è stato completato.

Rimanendo nell'ipotesi di un sistema monoprocessore, se prima avevamo solo la CPU che poteva chiedere delle operazioni alla memoria centrale (quindi era solo la CPU che occupava il bus indirizzi e il bus dati), adesso abbiamo due dispositivi (la CPU e il controller DMA) che possono chiedere, magari anche nello stesso momento, di utilizzare il bus per trasferire dei dati; il bus è unico, quindi il primo dei due che arriva prende il controllo del bus, mentre il secondo deve aspettare che il bus si liberi.

Interruzione software (Trap)

Un evento è di solito segnalato da un'interruzione dell'attuale sequenza d'esecuzione della CPU, che può essere causata da :

- un dispositivo fisico → si parla di **interrupt**.
- da un programma. → si parla di **trap**.

Una *trap* può essere causata da un programma in esecuzione a seguito di un evento eccezionale, riconosciuto tramite l'architettura della CPU (per esempio un errore: una divisione per zero o un accesso alla memoria non valido), oppure a seguito di una richiesta specifica effettuata da un programma utente per ottenere l'esecuzione di un servizio del SO, attraverso una speciale istruzione detta *chiamata di sistema* (**system call**) o chiamata supervisore (**supervisor call**, SVC).

Le trap vengono trattate in modo del tutto analogo agli interrupt: ciascun tipo di trap ha un identificatore che viene usato per cercare nell'*interrupt vector* l'indirizzo della prima istruzione del corrispondente interrupt handler.

Gli *interrupt* sono **asincroni** rispetto al programma in esecuzione; le *trap* sono **sincrone** in quanto causate dal programma in esecuzione.

Interruzione generata dal software vuol dire che nella CPU ad un certo punto va in esecuzione un'istruzione (fase di *execute*) che ha l'effetto di interrompere il processo che ha eseguito quell'istruzione.
In questa maniera un processo può interrompere se stesso.

In fase di *execute* :
- se tutto va bene, si passa all'istruzione successiva;
- se c'è un qualche errore (es: accedo ad un'area di memoria a cui non devo accedere, faccio una divisione per 0), viene generata un'interruzione software.
La fase di
execute di un'istruzione ha già in sé una serie di controlli che se non vengono superati fanno sì che venga generata un'interruzione software.

Per seguire da vicino tutte le operazioni che devono essere fatte per gestire un'operazione di I/O, proprio per evitare che sia il programmatore a doversene occupare, esistono delle funzioni del Kernel che fanno questa operazione al posto nostro.

System Call

Visto che il SO è stato organizzato a rispondere a delle interruzioni, come fa un programma a invocare una funzione del SO?

Il modo più semplice è generare un'interruzione.

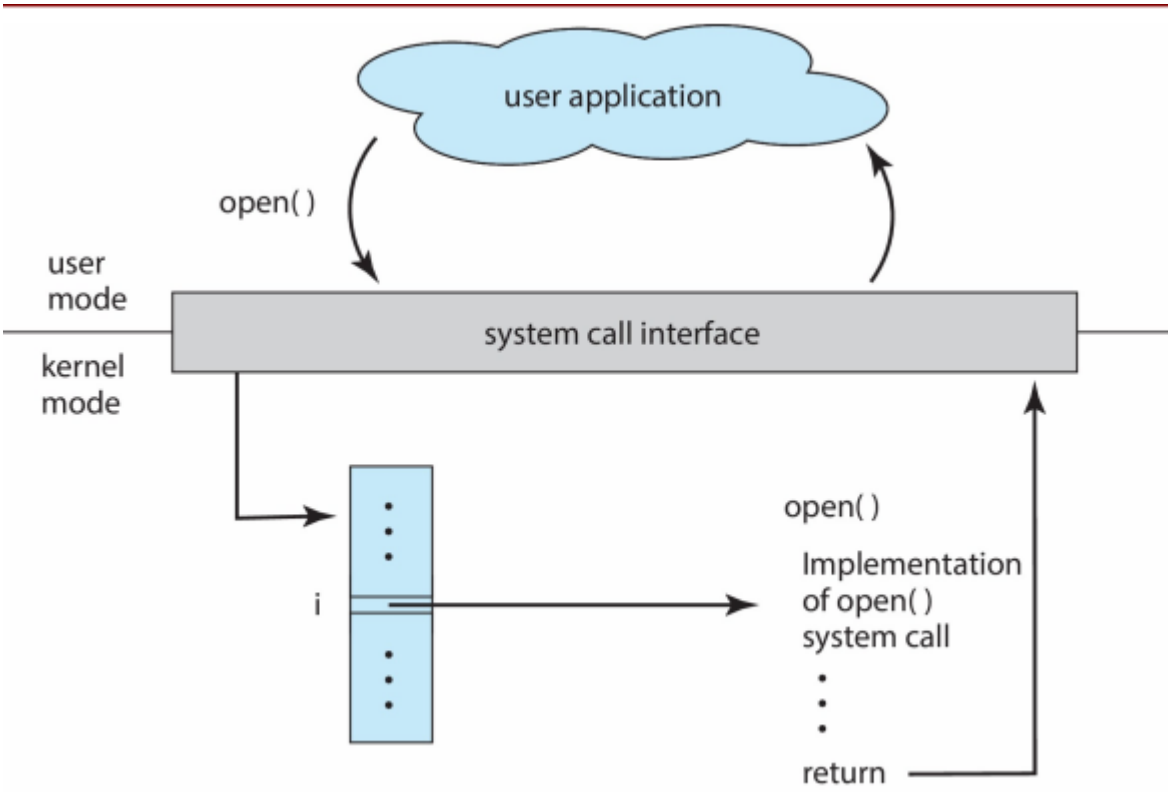
Quindi il mio programma, quando deve eseguire un'operazione di input, non si mette lui a farlo materialmente, ma invoca la funzione del Kernel che fa questa operazione al posto suo. Per invocare questa funzione del Kernel, visto che a quel punto il processo si deve sospendere e lasciare il posto al Kernel e attendere che gli dica che l'operazione è completata, il modo più semplice è fargli generare un'interruzione software che ha:

- come primo effetto il fatto che lui viene sospeso;
- come secondo effetto, dandogli il codice giusto, viene invocata quella funzione del Kernel che si prende in carico la gestione dell'operazione di input.

Questo meccanismo viene chiamato **SYSTEM CALL** e vengono gestite mediante interruzioni software.

Le interruzioni software vengono utilizzate principalmente per due motivi :
1)
gestire eccezioni che si possono verificare durante l'esecuzione di una qualche istruzione;
2)
permettere ad un processo di invocare le funzioni del SO (system call).

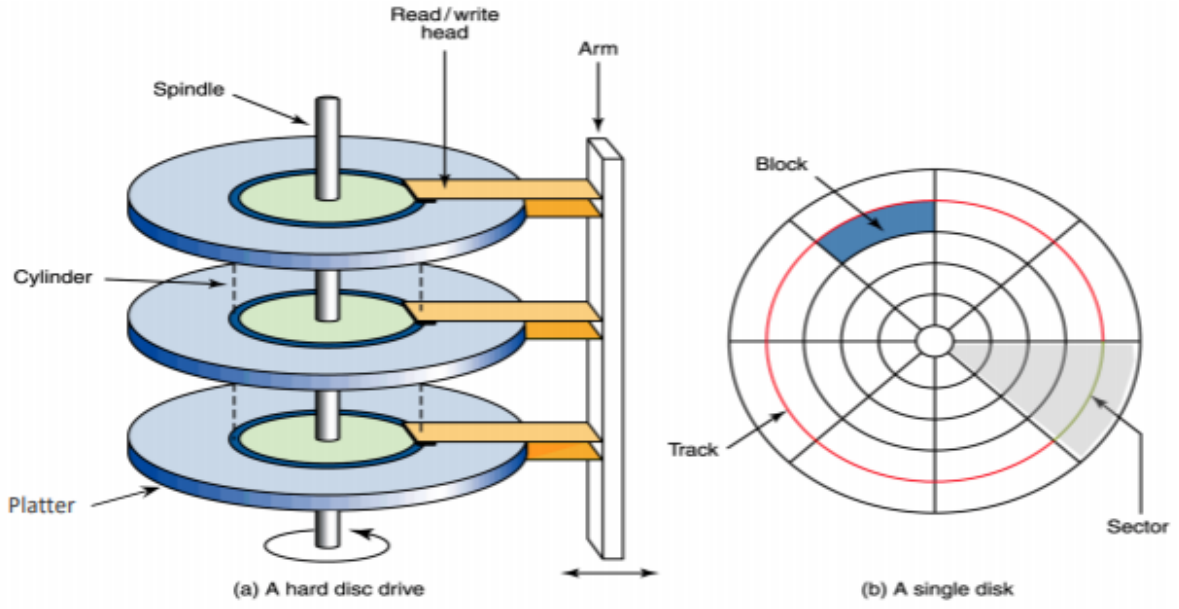
Le system call passano attraverso l'interfaccia tra un programma in esecuzione ed i servizi di un SO; vengono implementate nelle trap.
Esistono tre metodi per il passaggio dei parametri:
- I valori dei parametri vengono memorizzati nei registri;
- I valori dei parametri vengono memorizzati in una tabella in memoria e l'indirizzo della tabella viene memorizzato in un registro.
- I valori dei parametri vengono memorizzati dal programma con una operazione di
push nello stack (lo stack dei parametri risulta adatto ad esecuzioni ricorsive) e letti dal SO con una operazione di *pop*.



Memoria secondaria (di massa)

La *memoria centrale* è un supporto di memorizzazione (volatile) che può essere acceduto direttamente dalla CPU.

La *memoria secondaria* è un'estensione della memoria centrale, in grado di fornire un supporto di memorizzazione non volatile e di maggiori dimensioni. Un disco magnetico è un insieme di piatti ricoperti di materiale magnetico. La quantità di informazioni per blocco è costante: blocchi più vicini al centro riescono ad immagazzinare la stessa quantità di dati rispetto ai blocchi periferici.



Protezione dell'hardware

Sono dei meccanismi usati per evitare che, a seguito di un errore di programmazione, un processo danneggi altri processi o blocchi l'intero sistema.

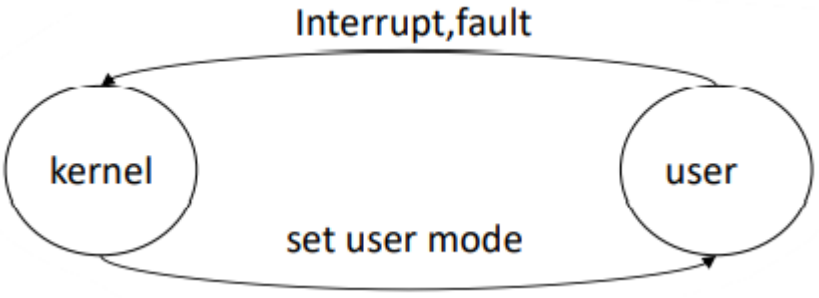
Funziamento in Dual Mode

Per garantire il corretto funzionamento del sistema è necessario distinguere tra l'esecuzione di codice del SO e di codice definito dall'utente. Il metodo seguito da molti sistemi operativi consiste nel disporre di specifiche caratteristiche dell'architettura del sistema che consentono di gestire differenti modalità di funzionamento.

Sono necessarie almeno due diverse modalità: *modalità utente* e *modalità di sistema* (detta anche *modalità kernel*, modalità supervisore, modalità monitor o modalità privilegiata).
Per indicare quale sia la modalità attiva,
l'architettura della CPU deve essere dotata di un bit, chiamato appunto *bit di modalità*: di sistema (0) o utente (1). Questo bit consente di stabilire se l'istruzione corrente si esegue per conto del sistema operativo o per conto di un utente. Quando l'elaboratore agisce per conto di un'applicazione utente, il sistema è in modalità utente. Tuttavia, quando l'applicazione utente rivolga una richiesta di servizio al SO (tramite una chiamata di sistema), per soddisfare la richiesta questi deve passare dalla modalità utente alla modalità di sistema.

All’avviamento del sistema, il bit è posto in modalità di sistema. Si carica il sistema operativo che provvede all’esecuzione dei processi utenti in modalità utente. Ogni volta che si verifica un’interruzione o un’eccezione si passa dalla modalità utente a quella di sistema, cioè si pone a 0 il bit di modalità. Perciò quando il SO riprende il controllo del calcolatore si trova in modalità di sistema. Prima di passare il controllo al programma utente, il sistema ripristina la modalità utente riportando a 1 il valore del bit.

La duplice modalità di funzionamento (dual mode) consente la protezione del SO rispetto al comportamento degli utenti e viceversa. Questo livello di protezione si ottiene definendo come istruzioni privilegiate le istruzioni di macchina che possono causare danni allo stato del sistema. Poiché la CPU consente l’esecuzione di queste istruzioni soltanto nella modalità di sistema (kernel mode), se si tenta di far eseguire in modalità utente un’istruzione privilegiata, la CPU non la esegue, ma la tratta come un’istruzione illegale inviando un segnale di eccezione al SO.



All'interno della CPU c'è un bit, chiamato bit di modo, che quando è settato a 0 allora possono essere eseguite tutte le istruzioni, privilegiate e non privilegiate. Questa modalità di funzionamento viene chiamata modalità kernel, perchè l'unico programma che è autorizzato ad eseguire tutte le istruzioni è il kernel. Quando questo bit di modo è settato a 1, ovvero in modalità utente, allora la CPU può eseguire SOLO istruzioni non privilegiate; significa che se il bit è settato in modalità utente e sta eseguendo un programma che presenta un'istruzione privilegiata, nel momento in cui la CPU prova ad eseguire tale istruzione privilegiata, vede che il bit di modo è settato in modalità utente e viene generato un errore (trap).
Se i processi utente (che possono lavorare solo in modalità utente) hanno

bisogno di svolgere delle funzioni per le quali è richiesta l'esecuzione di istruzioni privilegiate, il nostro processo utente fa una chiamata di sistema (esegue un'interruzione software e manda in esecuzione un pezzetto del Kernel).
Un altro degli effetti dell'interruzione è che cambia il bit di modo.
L'ultima cosa che fa il pezzo di kernel che va in esecuzione prima di lasciare il controllo di nuovo a un processo utente è quella di spostare il bit di modo da modalità kernel a modalità utente.

Protezione dell'I /O

Tutte le istruzioni di I/O sono istruzioni privilegiate. Questo è il motivo per cui per eseguire operazioni di I/O bisogna eseguire delle system call.

Il funzionamento in Dual-Mode protegge i dispositivi in I/O, perchè quando un processo che lavora in modalità utente e ha bisogno di un dispositivo di I/O, fa una chiamata di sistema; a quel punto risponde il Kernel (che può lavorare in modalità kernel) e sarà lui a decidere quando potrà essere eseguita l'operazione che il processo ha richiesto.

Protezione della Memoria

Ogni processo è dotato della propria zona di memoria, che nessun altro utente può andare a modificare. Serve quindi proteggere la memoria da accessi indesiderati, al meno per il vettore di interrupt e gli interrupt handler. Si può implementare il meccanismo di protezione tramite due registri, detti registri base e registri limite. Il registro base contiene il più piccolo indirizzo legale della memoria fisica; il registro limite determina la dimensione dell'intervallo ammesso. La memoria al di fuori di tale intervallo è protetta.

Se il SO opera in modalità kernel, ha accesso illimitato sia alla memoria del monitor sia alla memoria dei programmi utente. Le istruzioni di load per i registri base e limite sono istruzioni privilegiate.

Protezione della CPU

Occorre assicurare che il SO mantenga il controllo dell'elaborazione, cioè impedire che un programma utente entri in un ciclo infinito o non richieda servizi del sistema senza più restituire il controllo al sistema operativo. A tale scopo si può usare un **timer**, programmabile affinché invii un segnale d'interruzione alla CPU a intervalli di tempo specificati, che possono essere fissi. È un contatore che viene decrementato dopo ogni clock tick, e, quando

raggiunge il valore 0, viene generato un interrupt. I timer sono comunemente usati per realizzare il time sharing o per calcolare l'ora. Load-timer è un'operazione privilegiata.