Architetture

Componenti e Memorie

1) Che tipo di shift ci sono? differenza shift logica e aritmetica

Ci sono due tipi di shift , lo shift logico sposta tutti i bit a destra o a sinistra e le posizioni vacanti vengono riempite con tutti zeri.
Però ho un problema nel caso degli interi **negativi** perchè ovviamente le posizioni vacanti a sinistra hanno un bit ad 1 che rappresenta il segno, che devo mantenere.

Questo è quello che fà l'arithmetic shift cioè mantiene il segno dove necessario , ovvero se il bit più significativo è 1 e shifto a destra allora riempie le posizioni vacanti con 1.
Se il bit più significativo è 0 allora è uguale a fare il logical shift.
L'arithmetic shift left è uguale al logical shift left.
Spiegazione breve : https://www.youtube.com/watch?v=GjmiiLzDzHI

2) Complemento a 2, come faccio a fare 18-5?

Per negare un numero in binario devo invertire tutti i bit e aggiungere 1.
18_2 = 10010
5_2 = 00101
-5_2 = 11010 +1 = 11011

10010+(-00101)=10010+ 11011 =01101 =13

3) Una rete combinatoria cosa é?

Una rete combinatoria, in astratto, è una funzione che mappa determinati input binari a determinati output binari.
In concreto una rete combinatoria è un insieme di porte logiche che soddisfano la funzione.

4) Mappe di Karnaugh, cosa sono?

Le mappe di Karnaugh sono matrici che mi permettono di semplificare le espressioni booleane e di conseguenza di ridurre le componenti elettroniche necessarie per realizzarle su silicio.
L'obiettivo è quello di cerchiare più 1 possibile con cerchi più grandi possibile **e quantità di 1 potenza di 2.**
Poi guardando i bit di input che **non** cambiano nelle caselle del cerchio scrivo l'espressione.

5) Mappa K di questa funzione: una rete combinatoria a 3 ingressi e voglio che la'output mi dica quanti ingressi sono ad 1. Intanto l'uscita deve essere a 2 bit.

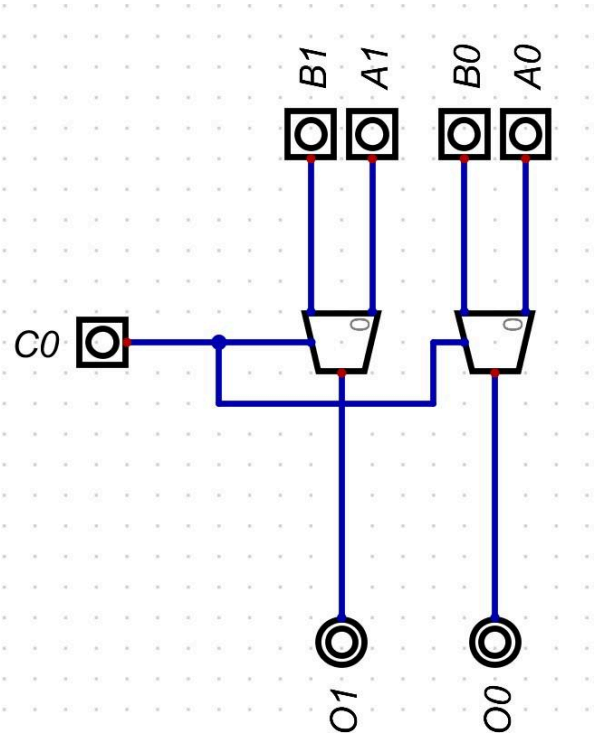
6) Che cos'è una rete sequenziale?

Una rete sequenziale è un circuito che ha necessità di ricordare uno stato precedente per funzionare correttamente. Per ricordare lo stato precedente si può usare un registro.

7) Ritardo di propagazione e contaminazione

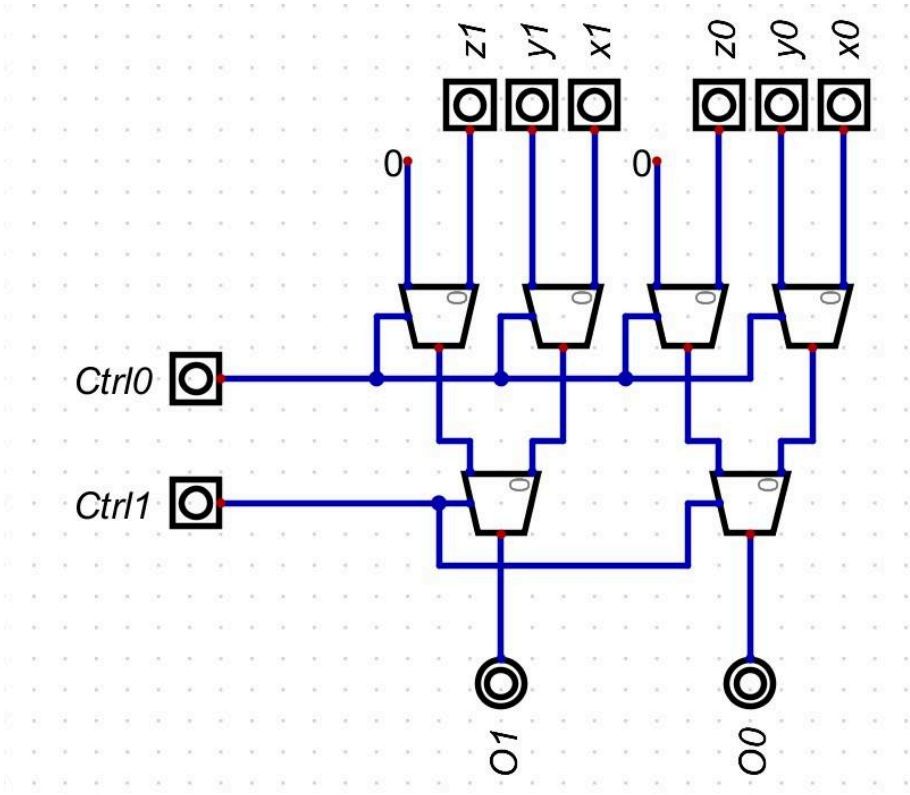
In un circuito, il ritardo di contaminazione è il tempo trascorso tra la stabilizzazione degli input e il momento in cui gli output iniziano a cambiare.
Il ritardo di propagazione è il tempo che intercorre tra la stabilizzazione degli input e la stabilizzazione degli output.
Ovviamente il tempo di prop >= tempo cont SEMPRE

8) Progetta un Multiplexer 2 ingressi a 2 bit



9) E se lo volessi da 3 ingressi a 2bit?

Per convenzione ho deciso che se Ctrl = 11 allora passa zero.



10) Cosa cambia tra architettura 32 bit e 64 bit? Quale è il vantaggio?

Tante cose,
Una architettura a 32 bit ha istruzioni lunghe 4 byte, una a 64 lunghe 8.
Ergo, una Word di sistema è lunga rispettivamente 4 e 8 byte.
Un puntatore è lungo quanto una parola di sistema.
Il PC và ovviamente incrementato appunto di 4 e di 8 nei casi rispettivi.
Tutti i registri sono lunghi una parola di sistema , quindi in un caso 4 l'altro 8.
Il vantaggio di lavorare con 8 byte anzichè 4 è di avere migliore portata e precisione computazionale nei calcoli.
Un Double è lungo 8 byte, posso usare i Long che sono lunghi 8, ora una moltiplicazione tra 2 interi non comporta problemi perchè in 32 bit dovevo utilizzare registri ausiliari nel caso la moltiplicazione risultava maggiore di 2^{32}.

11) T_{Hold} e T_{Setup} nei registri

Generalmente se un registro viene implementato con un FlipFlop allora la scrittura dei dati viene fatta al fronte di salita del clock.
Il T_{Setup} e il T_{Hold} sono intervalli di tempo che riguardano la stabilità degli input/output del suddetto registro.
Il T_{Setup} indica il periodo che precede il fronte di salita del clock entro il quale gli input DEVONO STABILIZZARSI per garantire la corretta lettura da parte della rete .
Se si stabilizzano troppo tardi è impossibile assumere che gli output della rete siano corretti.

Il T_{Hold} è il periodo di tempo dopo il fronte di salita del clock per il quale gli input della rete DEVONO essere mantenuti stabili per garantire la correttezza degli output.
Spiegazione: https://www.youtube.com/watch?v=g8lRqQ-IfYw

12) La differenza tra rete di Mealy e quella di Moore.

Le reti di Mealy hanno output che dipendono sia dagli ingressi che dallo stato interno.
Le reti di Moore solo dallo stato interno.
Entrambi i modelli hanno la stessa capacità espressiva ma a livello implementativo può essere più semplice usare una anzichè l'altra.

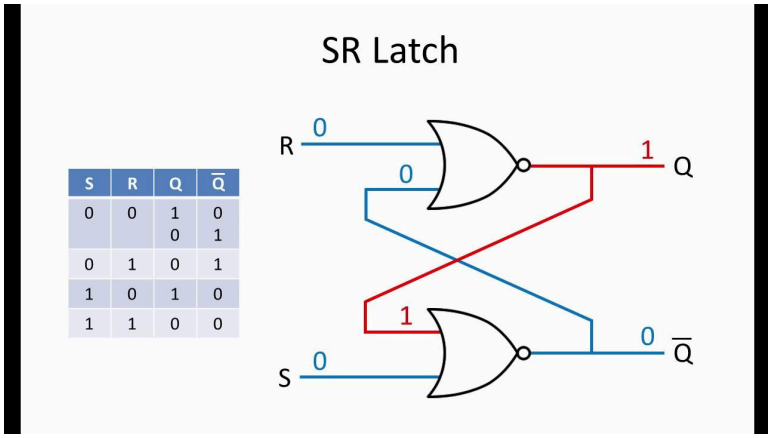
13) Automa, come stimo la lunghezza del ciclo di clock?

T_{CLK} **>=** T_P Registri + T_P Omega + T_P Sigma

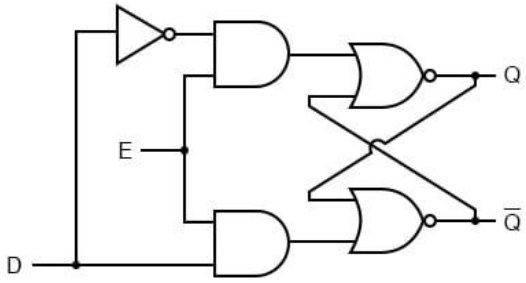
14) Latch SR cosa é? Come é fatto?

Il latch SR è una rete sequenziale che permette di immagazzinare 1 bit.
Ha 2 input e 2 output rispettivamente S, R , Q , Not Q.
S vuol dire Set e imposta Q ad 1, e Not Q ovviamente il contrario di Q cioè 0;
R vuol dire Reset e imposta Q a 0.
Se entrambi R e S sono a 0 allora la rete si ricorda $Q_{Precedente}$ e Not $Q_{Precedente}$;
Se entrambi R e S sono a 1 la rete ha comportamenti non corretti.

Questo circuito ha comportamenti errati quando gli input sono entrambi ad 1.



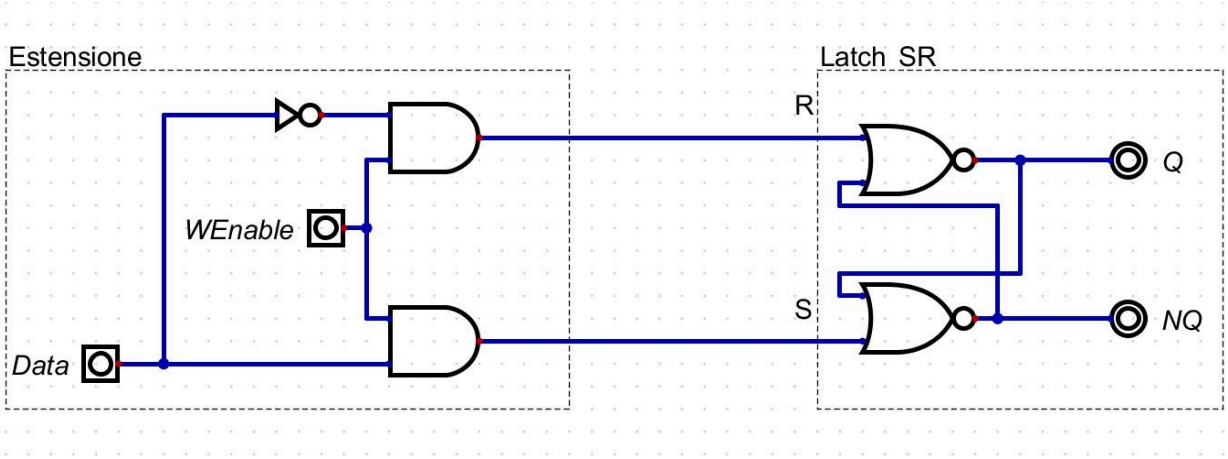
Il latch D estende il lato input del latch SR per impedire la problematica scritta sopra :



E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

D (data) rappresenta il dato che voglio scrivere ed E (enable) rappresenta se la scrittura è abilitata o meno. Di solito E è collegata ad un clock.
Questa configurazione impedisce che il latch SE riceva input entrambi ad 1.

15) Latch D, come é fatto. Cosa risolve?

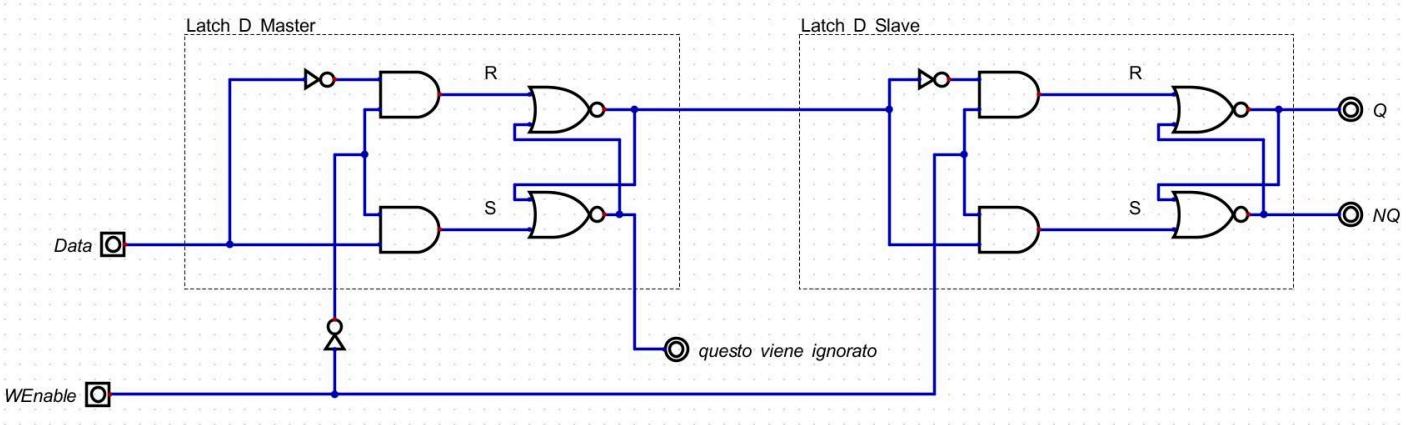


Il latch SR ha comportamenti errati quando sia S che R sono ad 1.
Concettualmente devo fare in modo che questa situazione non accada.
Per farlo estendo il lato input dell'SR con un WriteEnable che permette al dato di propagarsi nel latch evitando che R = S = 1.
Il comportamento è il seguente:

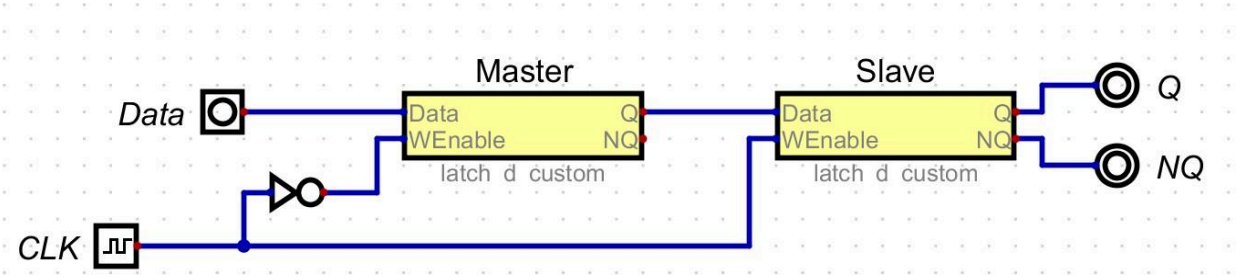
- Se WEnable è a 0 :
  - R riceve 0 e S riceve 0, il dato non viene scritto, indifferentemente se è 0 o 1.
- Se WEnable è a 1 :
  - Se Data è 0 il Reset del latch riceve 1 e il Set riceve 0.
  - Se Data è 1 il Reset riceve 0 e il Set riceve 1.

Notare come la configurazione impedisce sempre che al Latch SR arrivino segnali entrambi uguali ad 1.

16) Come è fatto un Flip Flop D?



Che in forma compatta + CLOCK:



17) Perché il Flip Flop scrive solo nel fronte di salita?

Perchè quando il Clock passa da 0 ad 1 il WE del Master diventa 0, quello dello Slave diventa 1 quindi lo Slave propaga Q_{MASTER} ***a*** Q_{SLAVE} ***e*** NQ_{SLAVE} .
In altre parole il Master diventa ***opaco.***

Quando il Clock passa da 1 a 0 il WE del Master diventa 1 e quello dello Slave 0, il Master ammette la scrittura, si dice che diventa ***trasparente.***

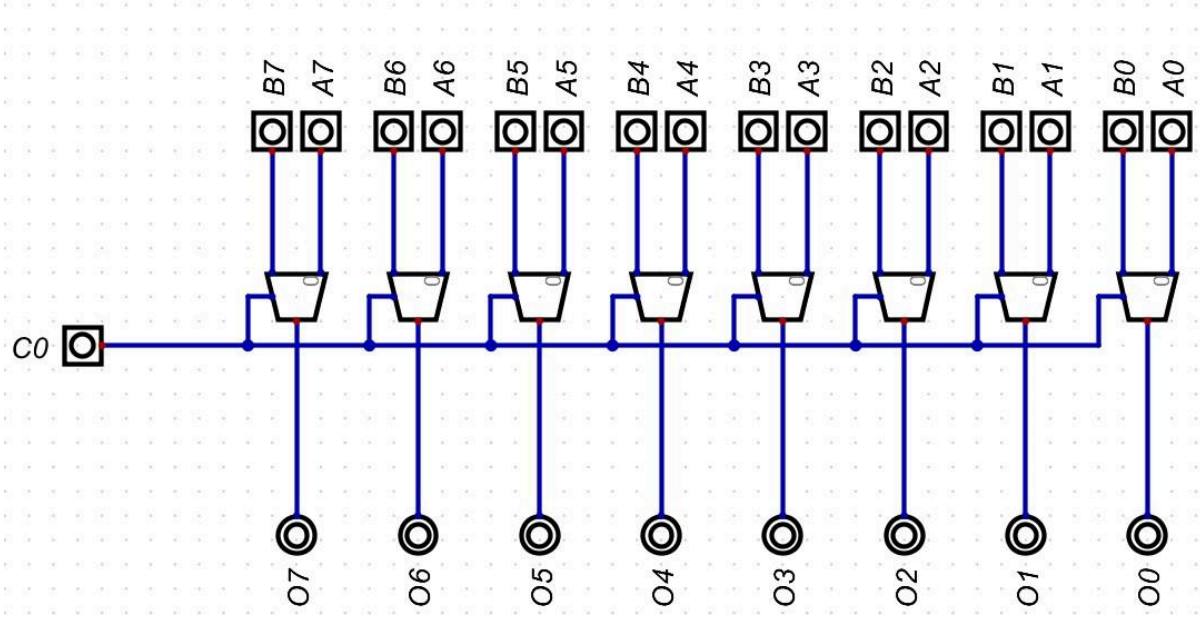
In realtà scriviamo al fronte di salita del clock per convezione. Potevamo anche decidere al fronte di discesa a patto che anche tutti gli altri registri scrivano al fronte di discesa.
L'obiettivo di scrivere al fronte di salita è quello di evitare che il Latch D aggiorni il dato finchè il Clock è 1.

18) Ho un flip flop D, come ci faccio un registro?

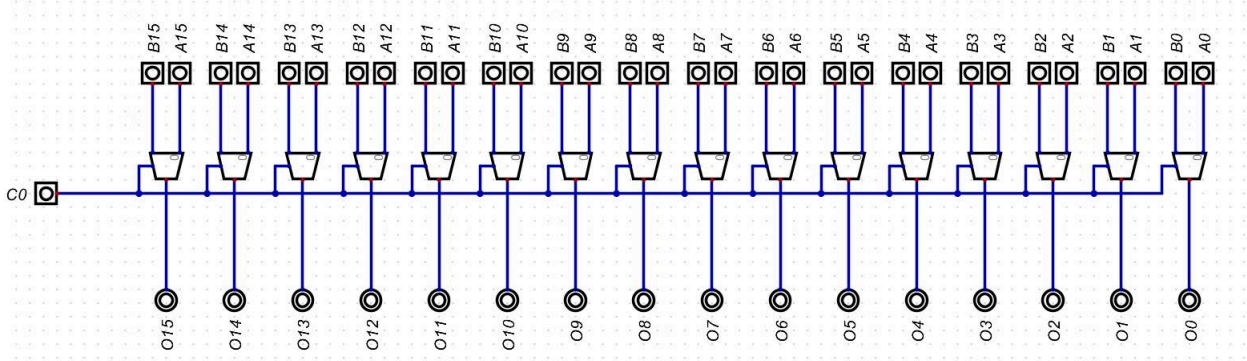
Ne piglio 8, 16, 32 etc. e li configuro così.

19) Il T_P di un MUX cresce in modo logaritmico in base al numero di ingressi o al numero di bit dell'input?

È il **numero di ingressi** che causa un aumento del tempo di propagazione, non i bit.
In figura ho riportato lo **schema di un MUX a due ingressi da 8 bit**.

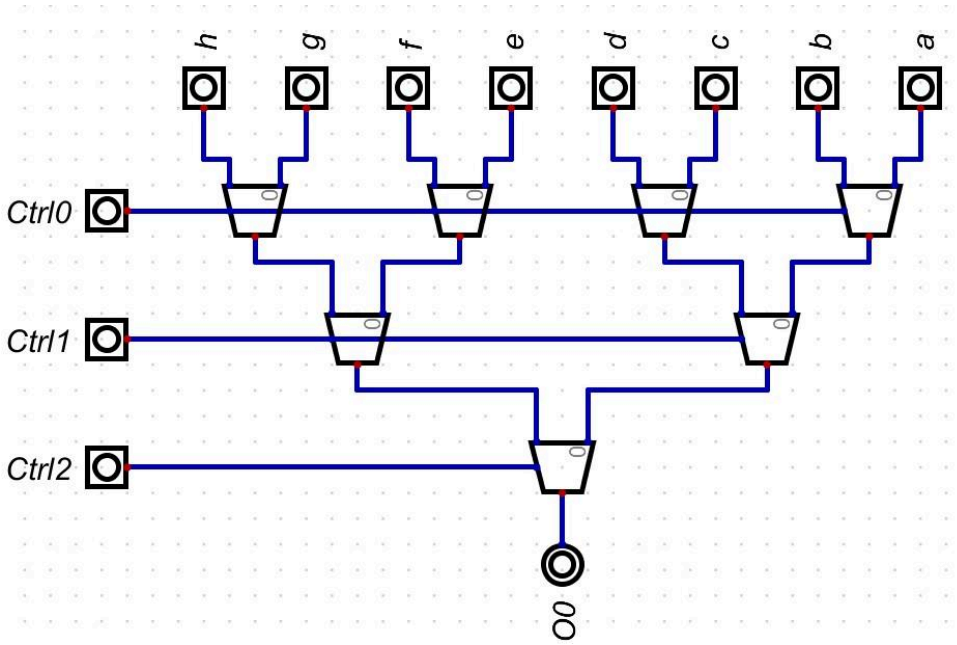


Tutti i multiplexer sono in parallelo quindi si stabilizzano allo stesso tempo.
Il livello dell'albero rimane uno, infatti per controllo C0 mi serve solo un bit perchè mi basta commutare tra l' *i*-esimo bit di A e l'*i*-esimo bit di B.
Se ho **8 bit** mi servono **8 MUX** binari, **complessità lineare**.
Giusto per enfatizzare la cosa riporto pure il MUX a 16 bit:



Se invece ho un mux con tre ingressi ho bisogno di almeno $Math.Ceil(\ Log_{\ 2}\ 3\ ) = 2$ livelli

Ecco a voi riportata invece l'immagine di un MUX a 8 ingressi tutti da un bit:



Notare come ho dovuto impiegare **n-1** MUX dispiegati su 3 livelli nonostante tutti gli ingressi siano ad 1 solo bit. Il tempo di propagazione è quelli di 1 mux moltiplicato per il numero di livelli.

Questa cosa è generalizzabile in base al rapporto tra il numero di ingressi e il numero di uscite. Cioè:

$$N * B_{IN} / 2^X = K * B_{OUT}$$

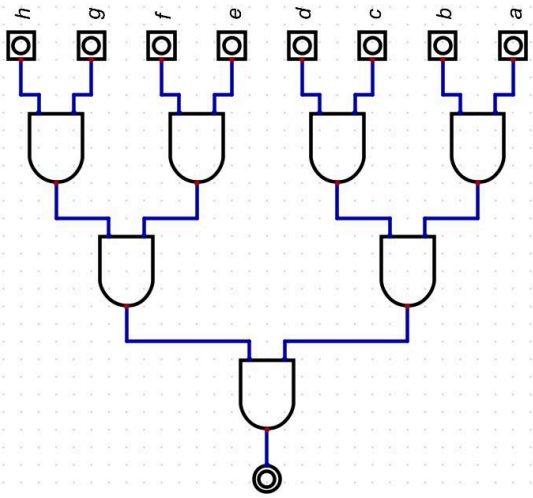
La X è l'incognita e N, B e K sono i parametri dell'equazione.

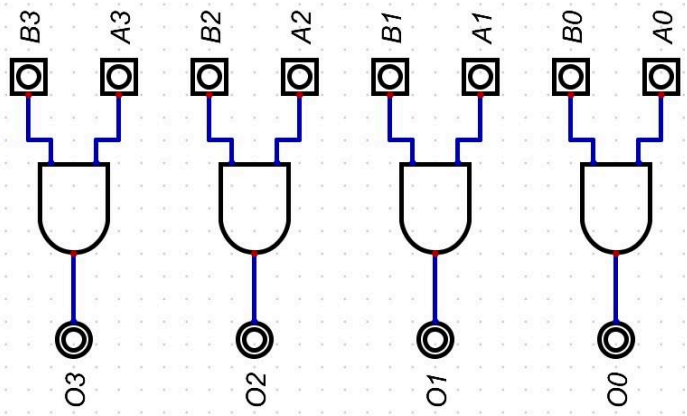
- N è il numero di entrate della rete.
- B_{IN} è il numero di bit di un'entrata (assumo che tutte le entrate hanno lo stesso bittaggio).
- B_{OUT} è il numero di bit dell'uscita.
- K è il numero di uscite.
- X è il numero di livelli dell'albero della rete.

Se l'uscita ha lo <u>stesso numero di bit</u> delle entrate ($B_{IN} == B_{OUT}$) l'espressione è ulteriormente semplificabile:

$$N / 2^X = K \quad \Rightarrow \quad N = K * 2^X \quad \Rightarrow \quad N / K = 2^X \quad \Rightarrow \quad X = \text{Log}_2(N / K)$$

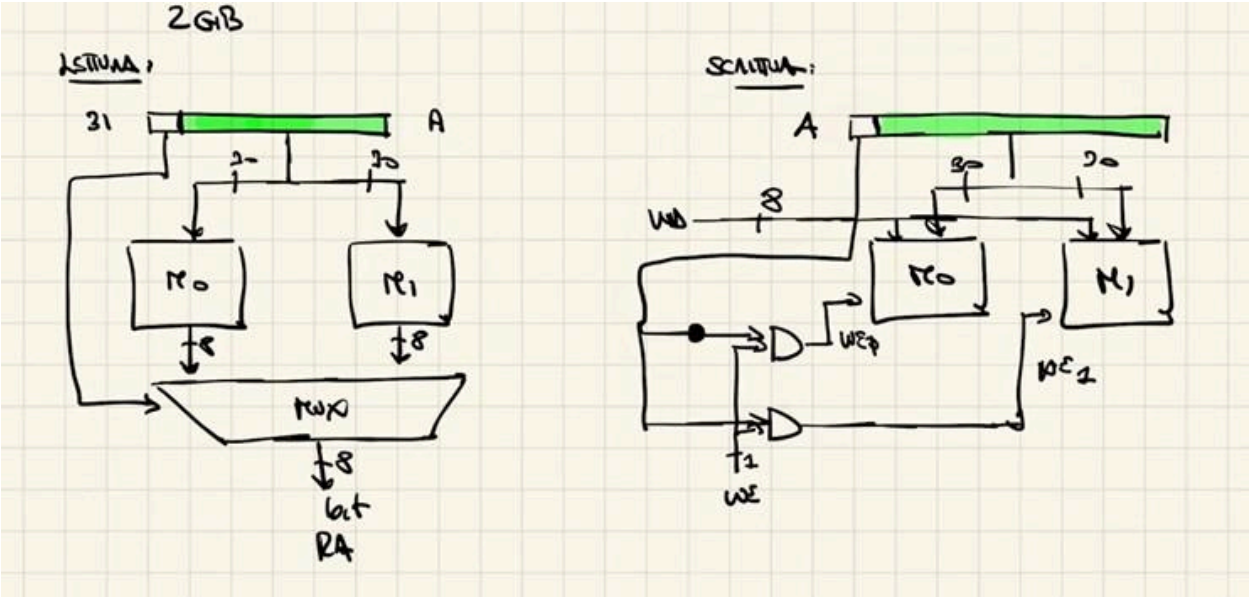
Se un mux da 8 bit con due ingressi produce una uscita da 8 bit ⇒ X è uguale ad 1.
Se una AND ad 8 ingressi da un bit produce 1 bit in uscita ⇒ X = 3.
Se una AND a 2 ingressi da 4 bit fà l'AND bit a bit e produce 4 bit in uscita ⇒ X = 1





20) Voglio una memoria da 1gb con 8 moduli interlacciata/sequenziale che in ingresso mi dà l'indirizzo e come output il valore. Come faccio a realizzarla con i componenti che conosciamo? Come faccio a leggerla?

La figura mostra quella sequenziale con 2 moduli da 1 GB.
Per farla interlacciata mi basta prendere come “parte verde” dell’indirizzo la porzione più significativa, tutto qui.
Notare che nella figura a destra il WE poteva essere benissimo distribuito con un Demultiplexer a 2 ingressi con bit di controllo la “parte bianca” dell’indirizzo.



21) Ho bisogno di componenti diversi tra memoria interlacciata e sequenziale?

No, i componenti sono gli stessi l’unica cosa che cambia è la partizione dell’indirizzo.

22) Ram di 1 Gb con 8 moduli , sequenziale vs interlacciata.

Come in figura sopra, però di moduli ne ho 8 e ciascuno di 2^{27} byte.

Architettura (Assembler)

1) Assembler: fattoriale ricorsivo.

Soluzione

2) Assembler: esponenziale ricorsivo.

Soluzione

3) Codice assembler che alloca un vettore di 1Mb e lo riempie del numero del ciclo corrente (cioè `vector[i] = i`).

MOV r0,#1
LSL r0,#20 @pusho a sinistra l'uno di venti posizioni perchè 1Mb = 2^{20}
Mov r2,r0 @copio il limite in r2 così non lo perdo
Bl malloc @ ora in r0 ho il ptr al vettore
Mov r1,#0
Label start:
Str r1,[r0,r1]
Add r1,r1,#1
Cmp r1,r2
BLT start

4) Assembler: Esiste una funzione " f " che ritorna 1 se l'argomento è una vocale altrimenti 0. Si scriva una codice assembler che scorre una stringa e utilizzi questa funzione per contare quante vocali ci sono nella stringa.

5) Tipo di stack in ARM

Lo stack in ARM è a 32 bit e può avere le seguenti configurazioni:

1. Full descending :
   Si riempie dall'alto verso il basso e il puntatore dello stack punta all'ultima posizione riempita. Un inserimento comporta un decremento dell indice e successivamente la copia dei dati. Questa configurazione si chiama pre-decrement.

2. Empty descending:

Uguale a sopra ma lo stack pointer punta alla prossima posizione libera, quindi un inserimento comporta una copia dei dati e DOPO un decremento. Questa si chiama post-decrement.

6) Funzione che prende un vettore, size e un carattere, restituisco 0 se c'è 1 altrimenti

7) Funzione che dato un array di caratteri e un carattere di input, restituisce la posizione della prima ricorrenza del carattere

// REMINDER: per convenzione i parametri di chiamata a funzione vengono inseriti rispettivamente in r0,r1,r2,r3 ed eventuali parametri aggiuntivi vengono inseriti nello stack.
// in r0 ho il puntatore a stringa come primo parametro
// in r1 ho il codice ascii del carattere come secondo parametro
// in r2 decido di mettere l'indice con cui esploro la stringa
// in r3 decido di inserire il valore della stringa all'indice r2
// REMINDER 2: sempre per convenzione il risultato di ritorno di una funzione viene inserito in r0

```
Mov r2,0

Label Start:
Ldrb r3, [r0,r2]
Cmp r3,r1
Beq Fine
Add r2,#1
B Start

Label Fine:
Add r0,r2
Mov pc, lr
```

8) Quante varianti della load si hanno

La load standard carica dalla memoria un numero di byte in base alla lunghezza della parola di sistema, un architettura a 32 bit (4 byte) carica 4 byte, una a 64 bit carica 8 byte etc. [1]
Però capita di voler caricare un solo byte dalla memoria anzichè 4 o 8 , questo lo posso fare scrivendo LDRB e carico un solo byte.

[1] La direzione in cui questi byte vengono presi dipende se l'architettura è big endian (scorro l'indirizzo incrementalmente) o little endian (scorro l'indirizzo decrementalmente).

9) (Di tale domanda chiede l'idea) LDR R0, [R1,R2,LSL 2] in linguaggio macchina

Carico in R0 il contenuto dell'indirizzo di memoria di valore R1 + (R2 * 4) perchè Logical Shift Left è come moltiplicare per 4 **NEGLI INTERI .** (i float non funzionano così, vedere calcolo numerico)

10) ALU flags, cosa sono.

I flag della ALU sono dei bit che vengono attivati dalle operazioni eseguite dalla ALU e l'architettura ARM ne prevede 4 :

- N : Se l'ultima operazione ha prodotto un numero negativo.
- Z : Se l'ultima operazione ha prodotto uno zero
- V: Se l'ultima operazione ha prodotto un numero troppo grande ed è andato in overflow
- C: Se l'ultima operazione ha prodotto un carry in uscita

11) Com'è fatta un'istruzione operativa in ARM?

http://www.riscos.com/support/developers/asm/instrset.html

12) Dimmi il formato di un'istruzione di memoria.

(Vedere domanda 11)

Microarchitettura

1) Come funziona il single cycle.

Il Single Cycle è una microarchitettura che ad ogni ciclo di clock completa una singola istruzione.
L'intervallo del clock deve essere tarato in modo tale da accomodare l'operazione che dura di più, cioè quasi sempre la LDR.

2) Differenza tra single cycle e multi cycle

Il single cycle è una microarchitettura che esegue una istruzione alla volta e il tempo di clock deve essere tarato in modo da accomodare l'istruzione più lunga eseguibile (nel nostro caso la LDR) .
In altre parole le istruzioni più veloci non offrono alcun vantaggio , c'è un bottleneck fisiologico causato dalla LDR.
Voglio migliorare la situazione.
Alla luce di questo posso fare alcune considerazioni:

1. Ovviamente non posso modificare il tempo di clock dinamicamente in base all'istruzione, quello DEVE essere costante altrimenti le cose non funzionano.
2. Un fatto importante da notare è che non tutte le istruzioni toccano tutti gli stadi, cioè una ADD farà *Fetch->Decode -> Execute* mentre una LDR deve fare *F->D->E->Mem->WB.*

Detto questo posso separare gli stadi tra loro ficcando nel mezzo dei registri non-architetturali (invisibili a livello assembler) permettendo alle istruzioni più corte di fermarsi allo stadio appropriato senza che il clock debba aspettare inutilmente memory e writeback.

Questo mi permette inoltre di diminuire l'intervallo del clock in base allo stadio più lungo invece che all'istruzione più lunga. Purtroppo però il CPI aumenta, perché ora una istruzione impiega più cicli.

NOTA: Il multi-cycle è una microarchitettura sequenziale, viene fatta avanzare una sola ed unica istruzione alla volta, la prossima entra in azione quando la precedente ha terminato.
Traduzione: **NON è UN PIPELINE**

ATTENZIONE: questo non vuol dire necessariamente che le prestazioni siano migliori di prima , se il fattore con cui ho incrementato il CPI è maggiore o uguale del fattore con cui ho decrementato l'intervallo di clock non ho guadagnato nulla

3) Pipeline senza ottimizzazioni, quali sono i vantaggi rispetto al single cycle e multi cycle.

Il pipeline permette di completare gli stadi delle operazioni in parallelo e iniziare un'istruzione prima che la precedente sia andata a termine.
Il CPI è leggermente superiore a 1 perchè il CPI mi dice ogni quanti cicli inizio l'istruzione successiva, siccome inizio una istruzione (idealmente) ogni ciclo avrò il CPI circa 1.

4) Cosa cambia tra Multi Cycle e Pipeline?

Cambia il throughput. Nel Pipeline ne ho uno migliore perchè il mio CPI è sicuramente più basso del multiciclo.
Il multiciclo impiega più cicli per istruzione perchè

5) Quale é il problema? Cosa sono le dipendenze? Come faccio il forwarding?

Il problema che il pipeline ha è chiamato “dipendenze”.
Occorrono quando il risultato di un'operazione dipende direttamente dall’esecuzione e **dal completamento** di un’altra istruzione.
In pipeline però le istruzioni vengono completate in stadi; se ho una dipendenza devo

6) Cosa fa la Hazard Unit?

La hazard unit gestisce le dipendenze e manipola il flusso del datapath per fare inoltri o stalli.

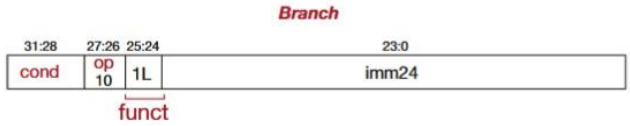
7) Come si accorge la Hazard unit?

Deve comparare i registri tra la decode e execute.

8) Cosa è il program counter?

Il Program Counter , concettualmente, è un indice che rappresenta l’istruzione corrente.
Dal punto di vista implementativo è un registro che contiene questo indice e viene usato come puntatore a memoria.

9) Com'è fatto un salto in istruzioni macchina?



L'istruzione è a 32 bit:

- I 24 bit meno significativi contengono l'offset tra il PC corrente e il Branch Target Address espresso in numeri di istruzioni
- I primi 4 bit più significativi sono comuni a tutte le istruzioni e mi dicono se l'operazione è condizionata
- Il campo OP da 2 bit indica che l'istruzione è di salto ,
- Il campo Funct da 2 bit dove il bit più significativo è sempre 1 e il secondo bit è 0 se è un branch semplice , 1 se è un branch and link.

10) Branch : Perchè al program counter devo inviare un offset moltiplicato per 4 ?

Perchè nell'istruzione di Salto l'offset è espresso in numero di istruzioni ma la devo convertire in numero di byte.

Spiegazione:
Noi dobbiamo immaginare il *Program Counter* come un indice di un array di singoli byte che però devo scorrere a passi di 4 byte perchè ogni istruzione è lunga 4 (cioè 32 bit perchè ARM è un'architettura a 32 bit).
Infatti quando lo avanzo non lo incremento di 1 ma di 4.

L'argomento della branch viene convertito in un numero dal compilatore assembler e indica il **numero di istruzioni** che distano da PC+8 e il mio *Branch Target Address* , cioè la mia destinazione.
Però non va bene!
Io non voglio la distanza in **numero di istruzioni** , la voglio in **numero di byte**.
Siccome so che ogni istruzione è lunga 4 byte e sono contigue nel codice allora devo moltiplicare questo numero per 4.
(Cioè se voglio andare all'istruzione 5 in realtà devo leggere i byte a partire dall'indirizzo 20)
Questo è l'offset **in byte** che devo sommare al PC + 8 per raggiungere BTA, e viene immagazzinato nei 24 bit dell'istruzione di branch.

11) Control path: A cosa serve la separazione tra i segnali e le condizioni.

Mi conviene separare la parte di Controllo e la parte Condizionale perchè viene sia impostata che interrogata solo lì.

12) Un esempio di un istruzione che ha effetto solo se i flag sono verificati.

ADDEQ

13) Quanti cicli di clk si necessitano per la branch e per la store nel multiciclo?

Per la branch 3.
Per la store me ne servono 4.
Ricordiamo che il multiciclo ha la memoria dati e istruzioni fuse insieme.

14) Cos'è il forwarding e quando non basta?

Il forwarding è una tecnica usata nelle microarchitetture pipeline per risolvere eventuali dipendenze indotte da una istruzione.
Un forwarding semplice non basta nel casi di una dipendenza innescata dalla LDR.
Il seguente frammento:
LDR r0, [r1, #Imm]
ADD r5,r4,r0

La ADD legge un registro che dipende direttamente dalla scrittura di una LDR && la distanza è 1.
In questo caso il fowarding non basta, devo implementare uno stallo.

15) Come viene realizzato il salto nel multicycle?

16) Tecnica del fowarding: scrivere codice in cui c'è dipendenza e risolverla

17) A livello di rete, cosa va controllato per fare il fowarding?

Devono essere controllate le fonti della ALU e del Register File con dei multiplexer controllati dalla hazard unit.

18) Cosa sono push e pop

In assembler ARM push e pop sono delle keyword che rispettivamente inseriscono o tolgono valori a 32 bit dallo stack.

19) Quando è conveniente il multicycle?

Il multicycle conviene rispetto al single cycle quando il tempo di completamento del multicycle è inferiore al tempo di completamento del single.
Nel single cycle il CPI è a 1 (singola istruzione per ciclo) e nel multicycle il CPI possiamo assumere sia circa 4.
Inoltre assumiamo che ClockSC > ClockMC.
$T_C^{(SC)} = N_{operazioni} * CPI_{SC} * ClockSC \Rightarrow N_{OP} * 1 * ClockSC \Rightarrow N_{OP} * ClockSC$
$T_C^{(MC)} = N_{OP} * CPI_{MC} * ClockMC$
Vinco quando :

$T_C^{(MC)} < T_C^{(SC)}$ cioè

$CPI_{MC} * ClockMC < ClockSC$

20) LDR: come vengono risolte le dipendenze?

Stallo. Viene eseguito un flush del registro non-architetturale tra Decode/Execute e viene spento il WriteEnable dei registri non-architetturali tra Fetch/Decode e il PC.
In questo modo viene creata una “bolla”

21) Quando serve lo stallo nel pipeline? Come si realizza nel data path?

Lo stallo della pipeline avviene nel caso di una dipendenza ReadAfterWrite innescata da una LDR di distanza 1 * poichè dovrei avere un sistema di forwarding che legga l'output della memoria per poi inoltrarlo allo stato Execute indietro, facendo questo però dovrei in qualche modo rallentare lo stato di execute per dare tempo alla memoria di recuperare il dato che mi serve.
Però per questo caso infrequente sto modificando tutto il blueprint con un aumento PERMANENTE del ciclo di clock poichè il pipeline richiede un intervallo di clock abbastanza lungo per accomodare lo stadio della pipeline più lento.
Per evitare questo introduco lo stallo, cioè faccio in modo che il flusso di istruzioni rimanga un ciclo indietro.

Per implementare lo stallo devo prima di tutto potenziare la mia Hazard Unit.
Se la Hazard Unit , tramite un comparatore, si accorge che un registro di scrittura della LDR è lo stesso registro di lettura della prossima operazione allora eseguirà questi passaggi in un colpo solo:

- Spegne per un ciclo il WriteEnable del ProgramCounter
- Spegne per un ciclo il WriteEnable del registro non-architetturale tra Fetch/Decode.
- Azzera il registro non-architetturale tra Decode/Execute.

Il meccanismo così descritto farà in modo che al prossimo ciclo di clock venga Fetchata, Decodificata ed eseguita la stessa identica istruzione di prima col vantaggio però di aver creato un "void" (o una "bolla vuota" se vogliamo) nel pipeline che equivale ad un'istruzione Nop interposta tra l'istruzione che soffre la dipendenza e la LDR.
Dopo questo passo riprendo con l'esecuzione regolare e posso fare un semplice forwarding dal quinto al terzo stadio risolvendo la dipendenza.

*nel caso di distanza 2 non ho problemi, inoltro dallo stato di writeback all'execute normalmente.

22) Spiega il pipeline? Quanti cicli al max per istruzione?

Il pipeline è una microarchitettura che permette l'esecuzione parallela delle istruzioni suddivise in 5 stadi di Fetch, Decode , Execute , Memory e WriteBack .

23) Come si gestiscono i salti nel pipeline?

Un salto nel pipeline è complesso perché richiede di buttar via le operazioni attualmente nel flusso della CPU.
Posso assumere prima di tutto che il salto non venga preso.
Se vinco la scommessa non devo flushare nulla.

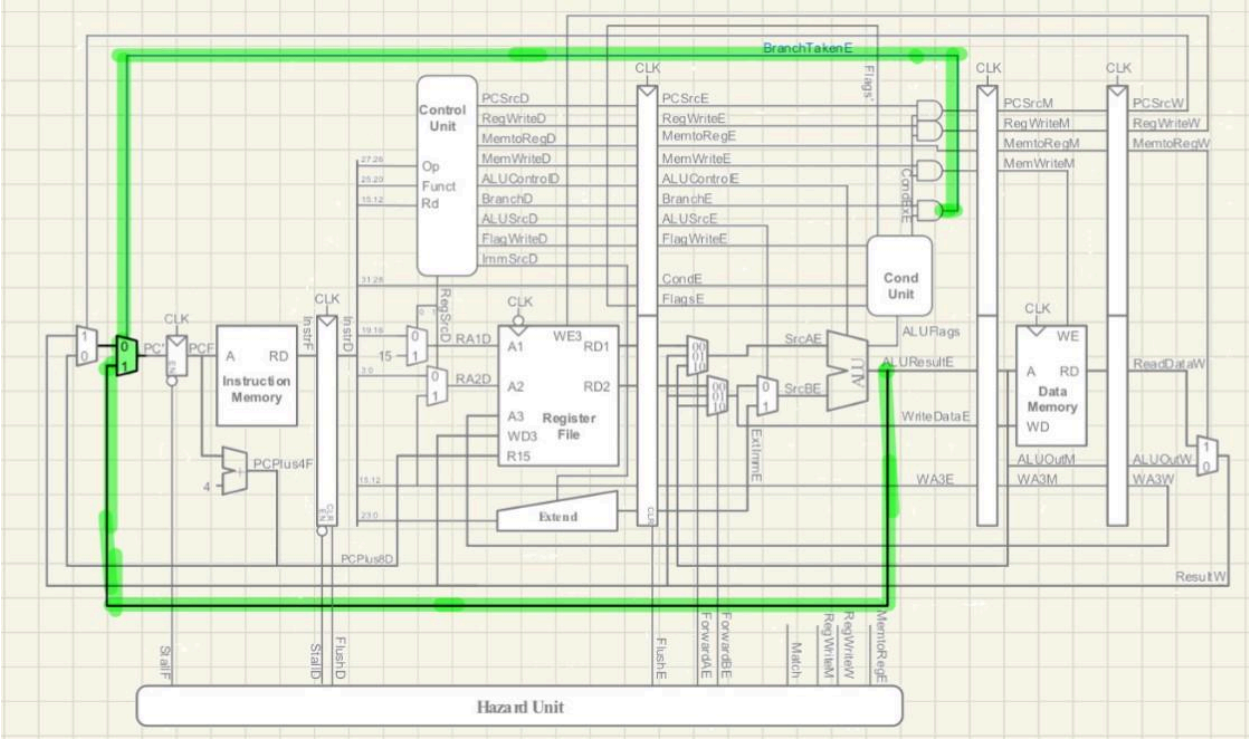
Se la perdo ho una penalità di 4 cicli se utilizzo l'approccio naive, 2 se uso l'anticipazione di salto.

L'anticipazione di salto è una tecnica che mi permette di "far skippare" lo stadio Memory e WriteBack all'istruzione della branch per modificare il Program Counter subito dopo che l'offset è stato calcolato dalla ALU.

Praticamente effettuo il Salto nello stadio di Execute.

In Figura:

Notare che il MUX evidenziato prende 1 **solo se il salto viene preso.**



Ma se ho skippato solo 1 stadio perché perdo 2 cicli e non 3? .

Perchè la valutazione di branch preso / non preso è una informazione che mi posso ricavare immediatamente dallo stadio di Execute e in quel momento ci sono solo 2 istruzioni dietro alla branch.

Assumendo che il salto venga preso devo cancellare i contenuti dei registri non architetturali tra Fetch/Decode **e** quelli tra Decode/Execute, solo 2.

Prima ne avevamo 4 dietro la branch.

24) Quante nop mi servono per l'approccio naive del branch?

Se uso anticipazione di salto solo 2. Se uso approccio "naive" me ne servono 4.

25) Altri modi per fare salto oltre alla branch

Un modo grezzo per implementare è impostare R15 (program counter) ad un numero per spostarlo in un indirizzo di memoria assoluto.
Questo ha poco senso a meno che non sia in un sistema embedded dove ad un certo indirizzo è fissata una funzione nota.
Un altro modo è quello di aggiungere ad R15 un numero positivo o negativo per spostarlo ad un indirizzo relativo all'inizio del programma.

26) Prima di fare un branch and link cosa devo salvare?

(I registri r0,r1 r2 r3 e r12 nel caso mi servano)

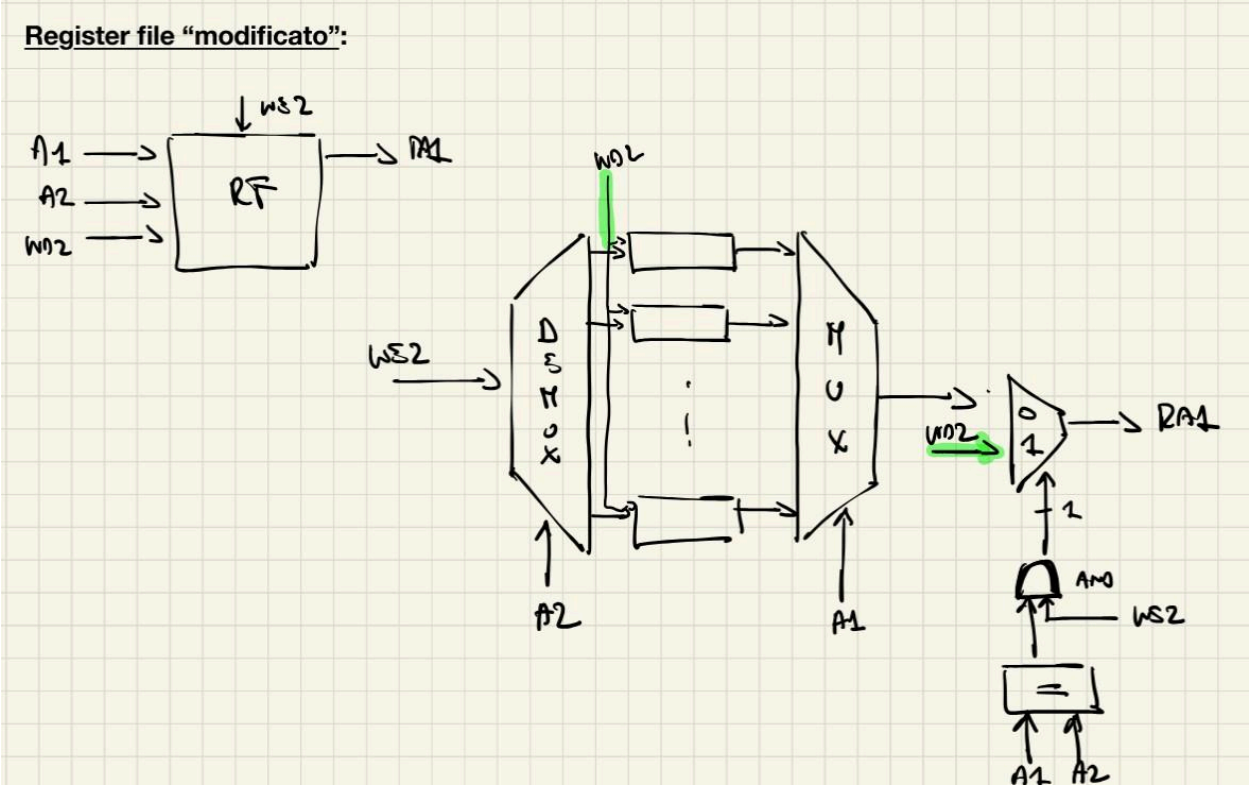
27) Load literal. Cosa sono? Come vengono implementati?

Un load literal in ARM si riferisce ad una operazione che carica un dato dalla memoria del codice.
Il dato è sempre un puntatore , equivale a fare LDR r0, [PC , #offset] dove offset è la distanza tra il PC corrente e l'indirizzo da caricare.
Cioè in altre parole carico una valore hardcoded taggato da una label nel segmento .data .

28) Le dipendenze di distanza 3 ci creano problemi. Perché? Come risolvo?

Non creano problemi, ho il Register File modificato che mi aiuta.
Le dipendenze di distanza 3 caratterizzano la lettura di un registro che attualmente è in WriteBack, quindi faccio in modo che questa lettura prenda il dato che viene scritto.

Estendo il mio register file con un comparatore semplice.
Se l'indice dell'indirizzo in lettura è lo stesso di quello in scrittura allora passo in uscita il dato che sto scrivendo.
Ovviamente modulando tutto in una porta AND col WriteEnable perchè questa cosa deve accadere solo quando scrivo.



Sistemi operativi

Cache e IO

1) Cosa vuol dire “politica di Write Through” ?

Nelle memorie cache multilivello è possibile che una memoria della gerarchia abbia un insieme di dati non aggiornati. (incoherency problem)
Questo accade quando la CPU assegna un nuovo valore nella coppia Indirizzo - Valore all’interno della cache che però non è ancora stato scritto nel livello inferiore.
La politica di write-through impone che nel momento di una write hit venga aggiornato anche il valore nella memoria sottostante immediatamente.
Questo meccanismo garantisce che ad ogni store le memorie siano coerenti SEMPRE, di conseguenza ogni volta che scrivo non ho bisogno di controllare se nel livello inferiore è presente il dato giusto, lo è per la coerenza costante. (cosa che non è assicurata nel WriteBack).
Generalmente questo approccio impiega un ciclo di clock ma è molto lento, devo aspettare la scrittura della memoria sottostante per procedere.

2) E “WriteBack”?

Nelle cache Write Back la CPU scrive solo nella prima cache e l'aggiornamento del livello di memoria sottostante viene eseguito solo quando rimpiazzo una linea modificata.

Comparato al write through questo approccio migliora la velocità media delle STR ma non garantisce la coerenza costante e rallenta la procedura di rimpiazzo perché potrei dover scrivere in memoria prima di rimpiazzare in modo da non perdere dati.

Come faccio però a tenere traccia delle linee modificate?
Tengo traccia delle linee modificate con un bit in più, il “dirty” bit : 0 se non ho modificato nulla, 1 se ho modificato qualcosa nel blocco.

Al rimpiazzo una linea di cache SE il bit è a 1 (*dirty victim*) allora devo aggiornare il livello sottostante , altrimenti non devo fare nulla (*clean victim*).
Se la linea “vittima” è pulita allora la linea di cache è rimasta coerente col livello inferiore, quindi procedo subito con il rimpiazzo senza problemi.
Se la linea “vittima” è stata modificata allora vuol dire che prima di toglierla devo scriverla nel livello sottostante perché **non è ancora coerente** .
Allora prima scrivo per ristabilire la coerenza e dopo rimpiazzo.

Nel caso del rimpiazzo di una linea modificata si impiegano 2 cicli di clock.

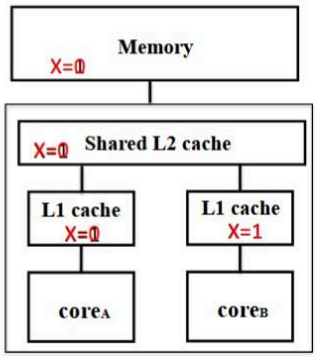
3) *Write Invalidate Protocol*: cos'è e a cosa serve.

Il write invalidate protocol è un protocollo di scrittura che risolve l'inconsistenza delle cache dovuta all'esecuzione di più thread su core diversi che usano gli stessi dati.

Ogni core ha la sua cache, ma un thread può venire eseguito su qualsiasi core disponibile ed è possibile che questo thread lavori su dati condivisi con altri thread del processo.
Immaginiamo che un processo ha una variabile X e nella cache del core 0 ho il dato X=5.
Poi un thread del processo inizia a lavorare su un'altro core, per esempio core 7 , e modifica la variabile X e la setta a 1500.
Ora però il processo ha due valori diversi per la stessa variabile!
Uno vale 5 , l'altro 1500.
Ho bisogno di un meccanismo di comunicazione tra le cache che per invalidare le linee di cache in comune che sono state modificate .

Write-through caches

Time	Event	L1-core1	L1-core2	M
0				X=1
1	A reads X	X=1		X=1
2	B reads X	X=1	X=1	X=1
3	A writes X	X=0	X=1	X=0



Write-back caches

Event	Bus activity	L1-core1	L1-core2	M
				X=1
A reads X	Miss for X	X=1		X=1
B reads X	Miss for X	X=1	X=1	X=1
A writes X	Invalidation for X	X=0	not valid	X=1
B reads X	Miss for X	X=0	X=0	X=0

4) Cosa è la MMU?

La memory management unit è un'unità aggiuntiva presente nella CPU che assiste nelle scritture in memoria , sia nelle cache che in memoria principale.

5) Write-Allocate vs No-Write-Allocate

Sono due metodi per gestire i write-miss nelle cache.

La write allocate recupera la linea di cache dal livello sottostante , la aggiorna e la immagazzina nella L0.
Questo metodo è più lento ma assume che la linea di cache verrà riutilizzata a breve.

La no-write-allocate invece non si preoccupa di recuperare la linea ma passa al livello sottostante direttamente l'operazione di write. Questo metodo è sicuramente più immediato ma se dovessi riaccedere alla linea di cache avrò un rallentamento che non avrei avuto con write-allocate.

WA è principalmente usato nelle WriteBack, NWA nelle WriteThrough.

6) Cosa è il Write Buffer nel contesto della write-through?

Fino ad ora abbiamo detto che le scritture in Write Through comportano un costo non banale in termini di latenza e tempo di clock.
Per velocizzare le scritture posso installare un write buffer cioè una piccola area di memoria "tampone" che immagazzina i dati che la CPU vuole scrivere, poi ci penserà il sottosistema di memoria in parallelo a scriverli nei livelli sottostanti.
Nel write through il write buffer è praticamente **obbligatorio**, non voglio che la CPU attenga il tempo di scrittura della intera gerarchia, è ovviamente preferibile che scriva nel buffer e continui subito le sue operazioni.
La CPU stalla solo quando questo write buffer si riempie, cioè quando la CPU sta producendo più velocemente di quanto il sottosistema riesce a consumare.

7) A cosa serve il write buffer nel Write Back?

Nel write back il write buffer è comunque consigliato.
Per rimpiazzare una linea di cache devo prima scegliere una linea "vittima" da togliere, che può essere sporca o pulita.
Se la linea "vittima" è pulita allora la linea di cache è rimasta coerente col livello inferiore, quindi la sovrascrivo e basta .
Se la linea "vittima" è stata modificata allora vuol dire che prima di toglierla devo scriverla nel livello inferiore perché **non è ancora coerente** .
Il problema è che per scrivere devo aspettare il livello inferiore 2 volte!
Questo perché devo prima attendere di scrivere la linea di cache modificata, poi dopo aspetto di nuovo per recuperare quella nuova.

Per velocizzare il processo la CPU scrive subito la dirty victim nel write buffer e recupera la linea di cui necessito.

La dirty victim verrà scritta in parallelo dal sottosistema di memoria dopo che la CPU finisce il fetch della linea nuova.

8) WriteBack workflow con e senza write buffer

Workflow della WriteBack standard SENZA write buffer.

Scenario A: Write Hit

1. La CPU richiede una scrittura in un indirizzo.
2. La linea di cache associata all'indirizzo viene trovata in cache (TAG coincide).
3. Il dato viene scritto
4. Il Dirty Bit della linea di cache viene impostato ad 1.
5. La CPU riprende l'esecuzione

Scenario B: Read Miss oppure Write Miss

1. La CPU richiede una scrittura / lettura di un indirizzo.
2. La linea di cache associata all'indirizzo **non** viene trovata.
    a. Se la cache ha spazi liberi la CPU esegue solo i passi ***5 e 7*****.**
    b. Altrimenti esegue tutti quelli elencati.
3. Viene selezionata una linea di cache "vittima" da rimpiazzare
4. Controllo il Dirty Bit della linea vittima
    a. Se la linea è pulita allora è consistente con la memoria principale. La sovrascrivo e basta.
    b. Se la linea è sporca la CPU si deve fermare e aspettare che venga scritta in memoria.
5. La CPU aspetta che la la linea di cache nuova venga fetchata dalla memoria.
6. La linea nuova viene sovrascritta al posto della vittima.
7. La CPU riprende l'esecuzione e ripete l'operazione originale
    a. Se era una read allora leggo la linea
    b. Se era una write allora ci scrivo sopra e la marchio come "dirty"

Notiamo che alcuni passi possono essere migliorati con un write buffer.

Workflow della WriteBack CON write buffer.
I cambiamenti sono evidenziati in rosso.

Scenario A: Write Hit

Identico a prima

Scenario B: Read Miss oppure Write Miss

1. La CPU richiede una scrittura / lettura di un indirizzo.
2. La linea di cache associata all'indirizzo **non** sta in cache.
    a. Se la cache ha spazi liberi la CPU esegue solo i passi ***5 e 7*****.**
    b. Altrimenti esegue tutti quelli elencati.
3. Viene selezionata una linea di cache "vittima" da eliminare
4. Controllo il Dirty Bit della linea vittima
    a. Se la linea è pulita allora è consistente con la memoria principale. La sovrascrivo e basta.
    b. Se la linea è sporca **la CPU copia la dirty victim nel write buffer.**
    Ora lo slot della vittima è immediatamente considerato libero.
    La scrittura della vittima verrà fatta in background più tardi dal sottosistema di memoria.
5. La CPU aspetta che la linea di cache nuova venga fetchata dalla memoria.
6. La linea nuova viene sovrascritta al posto della vittima.
7. La CPU riprende l'esecuzione e ripete l'operazione originale che ha causato il miss
    a. Se era una read allora leggo la linea
    b. Se era una write allora ci scrivo sopra e la marchio come "dirty"

Considerazioni finali:
Col write buffer la mia cache WriteBack **si ferma solo una volta** per aspettare la linea dalla memoria. É una ottimizzazione non banale.

9) LRU nelle cache.

LRU è una euristica (metodo) per rimpiazzare le linee di cache scegliendo quella utilizzata meno recentemente.
Nelle cache ad accesso diretto non ho scelta perchè la linea può andare solo nella posizione che coincide con i bit dell'indirizzo indicati come "Set" bits.
Nelle cache associative ho scelta perchè posso decidere di rimpiazzare uno degli N blocchi del set (Ricordiamo che nelle N-way un set ha esattamente N blocchi disponibili mentre nelle completamente associative il set è solo uno con tanti blocchi, basta che ne scelgo uno).
Ma quale scelgo? Quello più vecchio.
Chi è quello più vecchio???? Tenere traccia del blocco più vecchio potrebbe non essere efficiente, in una cache a 2 vie potrei mettere un bit U (Use this) per indicare quale dei due blocchi nella via rimpiazzare per prossimo.
Nelle cache a più vie diventa complicato.

10) [Domanda Trabocchetto] Quale algoritmo di rimpiazzo posso usare nelle cache ad accesso diretto?

Nessuno, nelle cache ad accesso diretto **non ho scelta** , ogni linea di cache può andare esattamente in una sola posizione.

11) Qual'è il difetto principale delle cache ad accesso diretto? Fammi un esempio.

Le cache ad accesso diretto soffrono maggiormente di **thrashing** poichè ogni linea di cache può alloggiare in un solo Set.
Quando faccio accessi a due indirizzi di memoria che hanno lo stesso set il sistema dovrà tutte le volte interrogare le cache sottostanti e impiegherà più tempo per cercare indirizzi piuttosto che eseguire istruzioni.

12) Cosa è il working set?

Il working set è un insieme di dati usati dal programma in certo intervallo di tempo.

13) Cache a 2 vie : 1 KB di capacità, 32 bit Word, una linea ha 64byte, numero di set? il numero di parole totale della cache

Dall'equazione :

C = B * b * w dove C è la capacità totale, B num blocchi totali, b parole per ogni blocco e w lunghezza in byte della parola

Riempo i parametri:

1024 = B * 64

Se una parola è a 32 bit allora w = 4 , una linea è 64 byte allora vuol dire:

b * w == 64 ⇒ b * 4 = 64 ⇒ b= 16

Quindi

1024 = B * 16 * 4 ⇒ B = 16

Num totale delle parole possibili in cache :

B * b ⇒ 16 * 16 = 256

Reminder : 2 vie vuol dire ogni linea di cache ha due blocchi (S = B / N)

13.1) Consideriamo la parola 0xF4 , che indice ha **DENTRO** il blocco?

5.

14) Gerarchia a tre livelli. Quante cache ho?

2. (La memoria é inclusa nella piramide)

15) AMAT & GLOBAL MISS RATE: CPU da 1 GHz , T di accesso ad L0 10 cicli , L1 50 cicli, L2 200 cicli e T_M 500 cicli, miss rate rispettivi 10%, 5%,1%

Formula generale AMAT:

CicliHit*DurataCiclo * + MissChance*(MissPenalty*DurataCiclo)

Ricordiamo che la misspenalty equivale al tempo di accesso al livello inferiore

1 ciclo dura $1/10^9$ = 1 nanosecondo = 10^{-9}

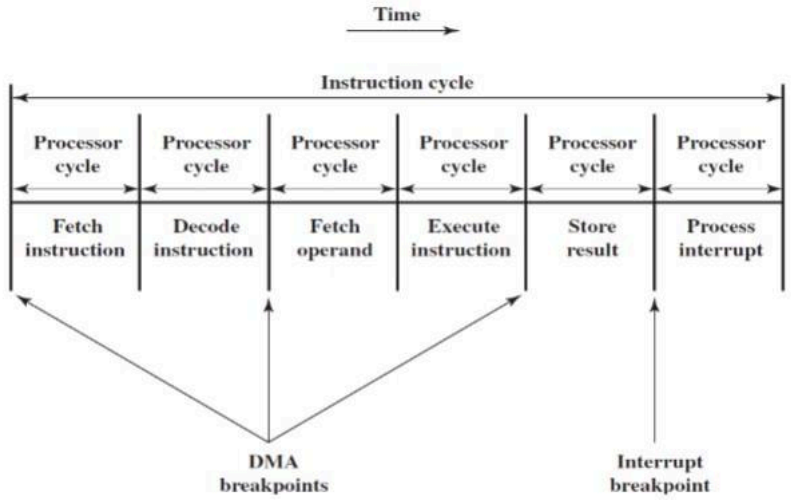
$10*10^{-9} + 0.1(50*10^{-9} + 0.05(200*10^{-9} + 0.01(500*10^{-9})))$ = boh fatelo voi il calcolo

16) Memory Mapped IO, cosa vuol dire?

É un metodo di gestione dei dispositivi input e output secondo la quale i registri dei dispositivi IO sono mappati in una regione specifica della memoria principale e la comunicazione tra i dispositivi e la CPU sono implementate tramite normali Load e Store a questi indirizzi.
Fun Fact: in alcune vecchie console da gioco i pixel dello schermo riflettono i valori a 16 bit di una pagina di memoria che aveva la stessa dimensione dello schermo.

17) Cosa è il DMA? Principali problematiche.

Il DMA è un meccanismo che permette alla CPU di delegare le operazioni di trasferimento dati tra dispositivi Input/Output e la memoria **in parallelo*.**
Il DMA è composta da un controller che è visto dalla CPU come un dispositivo **memory mapped**.
Quando è necessario eseguire un trasferimento la CPU invia in questi indirizzi mappati in memoria tutte le informazioni necessarie: quale è il dispositivo in questione, se l'operazione è una read o write, quanti byte trasferire e in che indirizzo metterli.
Quando il trasferimento è completo notifica la CPU con un interrupt
*(in realtà la risposta inizia qui) Asterisco perché in realtà il DMA non opera in vero parallelo: per trasferire dati ha bisogno di usare il bus in comune con la CPU. Quindi in un certo senso **potrebbero** andare in conflitto , si contendono il bus.
Normalmente la collisione **potrebbe** accadere in 3 punti rappresentati in figura perchè magari la CPU deve usare il bus per trasferire dati.
In quel caso la CPU deve attendere un ciclo nel quale il DMA rilascia il bus.



Poi il DMA introduce due ulteriori problemi **di coerenza con le cache**:

1. **Problemi in Disco → Memoria:**
   Fino ad ora abbiamo visto che **la coerenza delle cache è mantenuta a senso unico dall'alto verso il basso** (dalle cache verso la memoria) **ma non il contrario**!

Siccome il DMA può scrivere dati in memoria allora è possibile che siano le cache ad avere dati vecchi.
Chiarimento: **questo problema esiste sia in WB che WT.**
Ora ho bisogno di un meccanismo che mantenga la coerenza **dalla memoria verso le cache,** per esempio invalidando le linee di cache con lo stesso tag.

2. **Problemi in Memoria → Disco:**
   Nelle cache WriteBack i livelli più vicini alla CPU potrebbero avere dati più aggiornati rispetto alla memoria principale.
   Ora immaginiamo che il DMA debba scrivere questi dati sul disco.
   Come fà il DMA a leggere i dati nuovi se stanno nelle cache?
   Il DMA può solo leggere e scrivere dalla memoria![2]
   Devo forzare la propagazione delle modifiche verso i livelli sottostanti con un flush di tutti i blocchi “dirty” che servono al DMA.

[2] Questo perché il DMA ha il bus che comunica solo con la memoria, non con le cache.

18) Come fa il kernel a riconoscere un dispositivo? Come si identifica un dispositivo? Come recupera queste informazioni?

Il kernel dispone di una serie di gestori ad istruzioni privilegiate chiamati driver che hanno il compito di interpretare i comandi inviati dalla CPU per farli capire al dispositivo.
I driver formano un livello importante nello stack di Device Access.
Il kernel ha poi un subsystem software che gestisce tutti questi driver in base alla necessità del dispositivo; questo perché non tutti i dispositivi hanno una interfaccia uniforme.
Basti pensare per esempio ad un microfono che potrebbe avere driver specifici in base a se è USB-A , oppure USB-C oppure con il jack classico.
Un esempio di device con interfaccia uniforme e comune a tutti i Sistemi Operativi è il disco rigido, che oramai opera con driver SATA.

19) Interrupt driven IO e polling. Pro e Contro.

Il polling è una tecnica di “Status Checking” che la CPU impiega per interrogare lo stato di un certo dispositivo, più specificamente controlla il registro di stato del dispositivo..
Il polling è anche chiamato “attesa attiva” perchè la CPU interroga attivamente il device in questione in un ciclo.
Un po ' come Ciuchino in Shrek 2 quando continua a chiedere se sono arrivati.

L’interrupt driven IO invece agisce in modo diverso, la CPU ora non deve attivamente controllare di continuo lo stato del dispositivo ma può inviare una richiesta al dispositivo ed eseguire altre operazioni in parallelo; sarà il dispositivo ad avere la responsabilità di notificare la CPU che l’operazione è terminata.
In questo evento la CPU salverà lo stato corrente e farà partire in Interrupt Handler che gestirà il dispositivo.

20) Quando ha senso usare attesa attiva? indicare gli svantaggi

Generalmente l’attesa attiva di una risorsa è considerata come uno spreco di cicli, tranne in un caso: **se** il dispositivo ha bassa banda **allora** ha senso fare attesa attiva (polling).

Questa però è una situazione che va bene per il mouse, tastiera e per esempio le stampanti perchè sono abbastanza lente quindi mi basta fare una query di stato ogni tanto.
L’attesa attiva comporta un utilizzo della CPU inutile perchè poteva fare benissimo altre operazioni anzichè essere presa in ostaggio da un dispositivo.

21) Interrupt. Cosa fà l'hardware?

Concettualmente, a prescindere dal modello architetturale e la relativa implementazione micro-architetturale, l'hardware deve interrompere l'esecuzione corrente, gestire l'interrupt il più velocemente possibile e poi riprendere l'esecuzione da dove si era interrotta.

Per soddisfare questi 3 passi vengono eseguite operazioni diverse in base alla microarchitettura, ma in modo più generale possibile:

1. Per interrompere l'esecuzione devo prima aspettare che l'istruzione attuale si concluda perchè non voglio errori dovuti ad una esecuzione parziale delle istruzioni.
2. Spesso a questo punto vengono disabilitate le interruzioni (**Interrupt masking**) perchè non voglio che ulteriori interrupt si accavallino mentre ne gestisco uno.
3. L'hardware poi deve salvare il **contesto minimale base** che include **sempre** ProgramCounter , StackPointer & ParolaDiStato (PSW) del processo corrente e li salva[3].
4. Per proseguire l'esecuzione usando il KernelStack la CPU dovrà muovere nel registro StackPointer l'indirizzo del KernelStackPointer (la CPU sà dove trovarlo) così che d'ora in poi l'esecuzione utilizzi il KS.
5. Entro in modalità Kernel cambiando i bit di stato.
   É necessario entrare in Kernel Mode poiché i gestori degli interrupt (Interrupt Handler) richiedono un accesso privilegiato per isolarli da potenziale codice malevolo a livello utente.
6. Identifico l'InterruptHandler appropriato nell'InterruptVector.[4]
7. Salto nell'InterruptHandler e procedo la gestione. Eventuali registri usati nella subroutine verranno salvati nello Stack attuale per poi riesumarli al termine dell'Handler.
8. Il resto dei passi sono fatti a software

In ARM le interrupt vengono ascoltate dopo fetch decode e execute.

22) ARM come gestisce gli interrupt?

ARM ha dei registri ausiliari visibili solo in certe modalità Kernel.

---

[3] Di solito queste tre informazioni vengono temporaneamente salvate in registri ausiliari e successivamente nel **KernelStack** del processo corrente (ormai tutti i SO allocano sia uno UserStack che KernelStack per ogni processo) una volta impostato lo StackPointer del KS nel prossimo passo. Invece ARM in base alla priorità dell'interrupt non li salva per nulla e attiva dei registri ausiliari, e cerca di usare solo quelli.

[4] Un InterruptHandler è un puntatore a funzione indicizzato in un vettore appunto chiamato InterruptVector. Ogni interrupt ha associato un codice che indica l'indice dell'interrupt handler giusto.

In ARM ci sono più modalità privilegiate come System, Interrupt, Fast Interrupt, Abort , Supervisor etc.
In base alla modalità in cui mi trovo vengono attivati registri aggiuntivi , cosiddetti Banked, che permettono una gestione più celere degli interrupt.
La modalità System è praticamente una modalità Kernel senza però registri aggiuntivi.
La modalità Fast Interrupt ha più Banked di tutti perchè salvare i registri del processo precedente sullo stack richiede tempo prezioso, quindi per risolvere l'interrupt più velocemente installo registri ridondanti nella CPU così lei lavora con quelli senza dover salvare nulla.

Concorrenza

1) Perchè durante la fork è giusto copiare le pagine mentre con la exec no?

Perchè la exec cambia il programma in esecuzione quindi tutti i segmenti dati e codice saranno diversi, non avrebbe senso copiarli.
Nella fork invece verranno copiati solo quando uno dei due processi ci sovrascriverà. (Copy on write)
Non c'è bisogno di copiarli quando li leggo.

2) Cos'è un processo zombie?

Un processo zombie è un sottoprocesso che ha terminato la sua esecuzione ma che permane nella tabella dei processi.
Il processo zombie accade quando viene creato un sottoprocesso il cui exit code non viene mai letto dal processo genitore, impedendo l'eliminazione del processo nella ProcessTable perché ci sono informazioni sono ancora potenzialmente utili che il SO deve mantenere.

3) Stallo e starvation, Le condizioni per il deadlock.

Stallo e starvation sono due cose diverse.
Lo stallo (deadlock) è una attesa circolare tra thread .
Se un thread A ha bisogno di una risorsa che è detenuta dal thread B, ma il thread B ha bisogno di una risorsa detenuta dal thread A **ottengo una dipendenza circolare.**

Condizioni **necessarie** per lo stallo:

1. **Mutua esclusione**: una risorsa attualmente acquisita da un thread può essere usata esclusivamente da quel thread
2. **Risorse non revocabili**, una volta che un thread acquisisce una risorsa il kernel non gliela detrae
3. **Attesa senza rilascio:** Se un thread deve attendere una risorsa non rilascia eventuali risorse già acquisite
4. **Attesa Circolare:** Due thread si attendono a vicenda

4) Lettori/Scrittori

I lettori possono leggere senza mutua esclusione peró se uno scrittore deve modificare il testo si devono fermare tutti finché la scrittura non finisce

5) Implementazione fair + funzione StartWrite

6) Il reordering delle istruzioni impatta il funzionamento della sincronizzazione?

Assolutamente sì.
È comune scrivere un codice con l'assunzione che venga eseguito esattamente per come è scritto, ma per motivi di ottimizzazione il compilatore potrebbe riordinarle.
Questa cosa però non va bene per codice che deve essere sincronizzato.
.
Esempio:
In un buffer Producer/Consumer avrò sicuramente un indice per il produttore.
Questo indice può avere **2 configurazioni:**

1. Indica l'ultimo consumabile inserito
2. Indica la prossima posizione libera

Nella configurazione 1 **prima incremento e poi scrivo** mentre nella 2 **prima scrivo e poi incremento.**
Se il compilatore riordina queste istruzioni non funziona nulla!

7) PTHREAD YIELD, quando si usa maggiormente ? (spoiler: user level thread)

Si usa maggiormente nelle situazioni in cui un

8) Producer consumer con semaforo. Se il buffer é uno quanti semafori necessito?

Mi servono due semafori per buffer, uno per i produttori e uno per i consumatori.

9) Cosa vuol dire Race Condition?

Race Condition è una condizione sull'output di un programma e si manifesta quando la correttezza di suddetto output dipende direttamente dall'ordine di thread che lo modificano.
In altre parole, i thread "fanno a gara" per raggiungere una certa risorsa e l'output del programma cambia in base a quale thread arriva per primo.

10) Algoritmo del banchiere, come funziona?

Il sistema operativo , se conosce il numero di risorse che ogni processo necessita, puó provare a dare ad un processo le risorse e vede se il sistema si ritrova in una situazione unsafe, cioé se non riesce a fare terminare gli altri processi con le risorse rimanenti.

11) Come funzionano i semafori? Sono equivalenti alle c.v.? (**TRUCCO MNEMONICO**)

I semafori di Dijkstra sono strutture dati create per la sincronizzazione e hanno un contatore (quindi non sono memoryless come le CV) e hanno 2 metodi:

- P(), se il contatore è >0 decremento il numero di risorse disponibili. Se il numero era zero allora il thread chiamante si mette in lista di attesa.
- V(), incremento il contatore di 1 e risveglio il primo thread in coda di attesa, se ce n'è uno.

Non c'entrano nulla con le CV.

TRUCCO PER RICORDARE P() E V():

- P : Una risorsa è diventata **P**resa quindi decremento.
- V : Una risorsa è diventata **V**acante (cioè libera) e quindi incremento.

12) Condition variable. Come funziona?

La Condition Variable è una struttura dati per sincronizzazione e prevede 3 metodi.

- **Wait(Lock_t * lock)** : Funzione che interrompe il thread chiamante e rilascia la lock atomicamente, il thread chiamante si mette nella Waiting List della Condition Variable.
- **Signal():** il thread chiamante segnala **ad un solo** thread nella waiting list della CV che ora la condizione è verificata, quello che succede sotto il cofano è che viene semplicemente tolto dalla Waiting List della CV e aggiunto alla Ready List dello scheduler del SO.
- **Broadcast()**: come la Signal() ma notifica tutti i thread in attesa.

LE CONDITION VARIABLE FUNZIONANO SOLO DENTRO LE LOCK PERCHÈ SONO STATE PROGETTATE APPOSTA PER CONDIZIONARE LE LOCK

13) Perché è necessario che la ‘Wait’ della CV stia in un ciclo?

Perchè il thread risvegliato da Signal o Broadcast non rientra immediatamente in esecuzione ed è possibile che un altro thread **che lo precede nella coda Ready** sia entrato in esecuzione e abbia modificato la condizione, **togliendomi la garanzia che sia ancora verificata.**
Quindi per sicurezza quando un thread risvegliato torna in esecuzione **è bene che controlli la condizione finché non c’è la certezza che sia verificata, se non lo è chiama la Wait di nuovo.**

C’è anche un altro motivo però, più sottile.
Esiste il fenomeno degli **Spurious Wakeup,** risvegli spuri , che avvengono per motivi di performance.

14) Produttore consumatore con Condition variable.

15) Semantica Mesa e Hoare + vantaggi e svantaggi

Queste due semantiche caratterizzano le variabili di condizione e differiscono nel modo in cui segnalano il thread in attesa sulla condizione.
Nella semantica Mesa la Signal() non interrompe il chiamante e i thread risvegliati devono necessariamente riacquisire la lock.
Nella semantica Hoare invece la Signal() **interrompe il chiamante e** seleziona un thread in attesa cui **assegna direttamente** la lock. (Inoltre questo meccanismo rende la broadcast ridondante)

Hoare offre un vantaggio non-banale:
La Signal diventa atomica, permettendo al thread risvegliato di operare con la garanzia che la condizione non sia mai stata modificata.
Il problema è che questa semantica richiede algoritmi di Scheduling particolari e complicati.

Mesa invece ha il vantaggio di essere più semplice e risulta più retrocompatibile con gli impianti di scheduling dei sistemi operativi.

16) Differenza Lettore/scrittore e produttore/consumatore.

17) Lock vs spin Lock? Quando mi conviene?

La spinlock è praticamente un ciclo while(true) dove controllo sempre se la sezione critica è libera ed è considerata come uno spreco di cicli, tranne in un caso particolare: **se** le operazioni

che devo fare nella sezione critica costano meno del context switch indotto dalla transizione a "Waiting" del thread **allora** è meglio fare attesa attiva (o spinlock).
Questa però è una situazione abbastanza rara. L'attesa attiva comporta un utilizzo della CPU inutile perchè poteva fare benissimo altre operazioni anzichè essere presa in ostaggio da un processo.

18) Perchè il SO deve usare le spinlock anzichè Lock normali?

Le Lock normali sono bloccanti, interrompono l'esecuzione e spostano il thread nella Waiting List della Lock in modo atomico.
Le spinlock invece non bloccano l'esecuzione e non hanno una lista di attesa.
Sono usate soprattutto nel **Kernel**, per esempio:

- Negli **scheduler** : Lo scheduler non può bloccarsi, altrimenti il PC si impalla.
  Detto questo nel caso un thread dello Scheduler debba accedere ad una risorsa condivisa *(ad esempio la Waiting/Ready List)* non deve avere la possibilità di bloccarsi,
- Negli **Interrupt Handler**: Se un IH deve accedere ad una sezione critica non si deve poter bloccare, devono eseguire fino a completamento.

La cosa tricky delle spinlock è che per essere atomiche devono usare istruzioni speciali di Read-Modify-Write per l'acquisizione e il rilascio.

19) Perchè l'implementazione della Lock necessita una SpinLock?

È meglio spiegato con un esempio:
Possiamo immaginare le Lock strutturate così:

```
struct Lock
{
    bool Free; // se è libera o meno

    // lista d'attesa DELLA LOCK
    Queue LockWaitingList;
}
```

Come sarà fatta la funzione di acquisizione?

```
void Lock::Acquire()
{
    if(Free == false)
    {
        //aggiungo il thread chiamante alla
        //WaitingList della Lock
        LockWaitingList.Add(ThisThread);

        // Il Kernel mi sospende
        // poi mi aggiunge alla WaitingList
        // dello scheduler
        OS::Suspend(ThisThread);
    }
    Free = false;
}
```

Concettualmente se la Lock è occupata metto il thread corrente sulla waiting List della Lock e poi chiedo al SO di mettermi in Waiting List dello scheduler.

Però c'è un problema. Il codice sopra non mi basta.

Domanda prima:
La Lock è un oggetto condiviso???
Sì che lo è! Lo è per definizione!!!!!

Domanda che sorge spontanea:
Ma allora cosa succede se due thread cercano di acquisire la Lock nello stesso identico momento ??????????????
In quel caso sono nei guai! Sto modificando una struttura dati E un valore!

Perdipiù anche `OS::Suspend` modifica delle strutture dati, quelle dello Scheduler.
Abbiamo visto che le risorse condivise vanno protette.
Dovrei racchiudere sia `Free` che `waitingList` in una Lock **però non posso farlo perchè la Lock la sto definendo!!!!!!**
Cioè per definire un oggetto non posso usare l'oggetto stesso!

È come se io chiedessi:
Cosa è una Lock?
Una Lock è un oggetto contiene un'altra Lock…
Si ma cosa è una Lock?
Una Lock è un oggetto contiene un'altra Lock…
Si ma cosa è una Lock?
Una Lock è un oggetto contiene un'altra Lock……

È una definizione ricorsiva infinita che ovviamente non ha senso.
Tutti i compilatori e interpreti del mondo si lamenterebbero.

Come risolvo?
Devo per forza usare una SpinLock.

```
struct Lock
{
    bool Free; // se è libera o meno

    // lista d'attesa DELLA LOCK
    Queue LockWaitingList;
    SpinLock internalLock;
}
```

```
void Lock::Acquire()
{
    //questa deve essere atomica
    internalLock.Acquire();

    if(Free == false)
    {
        waitingList.Add(ThisThread);
        // Il Kernel mi sospende & rilascia la SL atomicamente
        OS::Suspend(ThisThread,internalLock);
    }
    Free = false;

    //pure questa atomica
    internalLock.Release();
}
```

Acquire aggiornata.

E la release?

```
void Lock::Release()
{
    ThreadControlBlock* next;

    // anche qui devo usare SpinLock perchè
    // accedo alla ReadyList dello Scheduler.
    // Anche nella Acquire ci
    // accedevo con la OS::Suspend
    internalLock.Acquire();
    if(LockWaitingList.Count() > 0 )
    { // c'è qualche thread in attesa

        //lo prelevo dalla coda
        next = LockWaitingList.Dequeue();

        //chiedo allo Scheduler di svegliarlo.
        OS::ReadyList.Add(next);
    }
    Free = true;
    internalLock.Release();
}
```

20) Problema dei filosofi, come si risolve?

Una possibile risoluzione del problema dei filosofi a cena è quello di rimuovere la “Wait while holding” cioè ogni filosofo che vuole mangiare prova ad acquisire esattamente due bacchette, se non riesce a prendere entrambe si interrompe e rilascia tutte le bacchette eventualmente detenute.
Un altro modo è quello di creare un vettore di Condition Variable , una per ogni filosofo, e il filosofo che decide di mangiare

21) Lock in Mono-Core vs Lock in Multi-Core.

Il parallelismo non esiste nei monocore, ho un core solo!
Le Lock nei processori mono-core mi servono per bloccare certe risorse condivise tra thread diversi.
Nei multi-core mi servono per la stessa cosa

22) Perchè le Lock funzionano male nei multiprocessori?

Le Lock nei multi-core funzionano male nel senso che si ha un bottleneck significativo sulle performance del parallelismo.
Ricordiamo che le lock permettono solo ad un thread di entrare in sezione critica:
se ho tanti thread su core diversi che cercano di accedere allo stesso Mutex allora tutti tranne uno dovranno aspettare, questa cosa **fà decadere le mie performance a quelle del mono-core, come se non avessi mai avuto il multi-core!**
Immagina di avere un'autostrada enorme a 16 corsie ma che ha un casello autostradale ad 1 corsia soltanto.
Questo è quello che succede sul multi-core.

Per peggiorare le cose ci si mette anche il problema della **coerenza tra cache dei core!**

Il contesto del parallelismo nel multi-core prevede un minimo di condivisione di dati tra thread che potrebbero andare a finire in più cache contemporaneamente!
Questo non era un problema nel Mono-Core perchè tanto la gerarchia delle cache era una sola!
Questi dati prima o poi verranno scritti da un thread che li usa.
Come faccio a dire alle altre cache che i valori vecchi vanno buttati?
I sistemi multi-core infatti hanno già **built-in** dei sistemi per mantenere le cache coerenti nel caso che più core condividano gli stessi dati.

Questi sistemi però sono sfortunatamente molto lenti e le Lock utilizzano questi meccanismi spessissimo, impattando le performance.

Immaginiamo questo scenario:

- Thread A esegue sul core X.
- Thread B esegue sul core Y.
- A e B condividono una Lock e le risorse da essa protette, come ad esempio un intero inizializzato a 10.
- A và in esecuzione, prende la Lock e incrementa l'intero, ora la Cache del core X avrà l'intero con valore 11.
- A rilascia la Lock e la passa a B, che è andato in esecuzione.
- B però gira sul core Y , che ha ancora l'intero **settato come 10!**
- Avviene una comunicazione tra queste cache per invalidare le linee di cache vecchie.
- Ora B dovrà recuperare di nuovo il dato dalla memoria.
- Ora B ha il dato corretto, 11.

23) Cosa è il False Sharing?

Il False Sharing è un fenomeno che si verifica quando due o più thread diversi modificano **variabili distinte ed indipendenti <u>ma che sono capitate nella stessa linea di cache</u>.**

Questo problema deriva dal fatto che le cache hanno una granularità a **blocco**, non a singola variabile; quindi è possibile che più variabili con indirizzo vicini siano capitate nello stesso blocco che è stato riesumato in più cache.
Quando questo accade ho grossi problemi di performance perchè quando modifico un dato che è presente in più cache devo “flushare” o “invalidare” le linee di cache del dato perchè considerate invalide, vecchie.
Siccome però le variabili sono indivisibili dal blocco questo capiterà ogniqualvolta che uno dei thread scrive.

Scheduling

1) Scheduling RoundRobin: pro e contro.

Il Round Robin fornisce un quanto di tempo a ciascun Job, di ugual valore per tutti.
Questo scheduling lo distribuisce in modo FIFO, quindi si avvale di una Queue.
Quando un Job si interrompe **O** esaurisce il TimeSlice viene messo in fondo alla coda.

Il pro è che sicuramente è una politica Fair, previene la Starvation ed è semplice da implementare.

Il contro è che ho tanti context switch che mi bruciano tempo **e** soprattutto i **pro dipendono dalla scelta del quanto di tempo**.
Se è troppo breve decado a SJF, se è troppo duraturo decado a FIFO.
Un altro contro è che i Job che hanno workload **misti** , cioè che hanno CPU-Burst e IO-Burst variabili, si ritrovano male in Round Robin.
Perchè se un Job si interrompe per un'operazione di IO viene messo in fondo alla coda e dovrà aspettare che tutti gli altri job finiscano il loro TimeSlice.
Se l'operazione di IO è lunga, poco male; se è corta (leggo dal mouse) mi costa tantissimo.

2) Funzionamento scheduling FIFO e SJF. quando sono convenienti?

Lo scheduling **FIFO** è una politica di scheduling non-preemptive secondo la quale i Job vengono processati in ordine di arrivo nella coda.

FIFO ha un overhead minimo perchè non deve controllare la durata dei job né deve ordinarli, li mette in coda e basta, quindi funziona bene in situazioni dove sò per certo che i Job sono in numero finito e della stessa lunghezza, inoltre previene la starvation.
Questa politica ha un difetto: i job più corti potrebbero ritrovarsi dietro a job più pesanti , il tempo medio di attesa nella coda potrebbe aumentare significativamente.

La politica **SJF** (che è sempre non-preemptive) impone che i job più corti abbiano la priorità su job più pesanti.
Questo implica che la coda dei job venga riordinata in ordine di durata crescente. (*Coda di priorità*).
La **SJF** migliora il tempo di attesa medio ma **non** previene la starvation dato che job in fondo alla coda potrebbero venire sorpassati da tanti job più brevi.
Secondo esperimenti di benchmark un Time Slice di 20 - 100 millisecondi funziona bene.

3) Da quale punto di vista SJF è ottimo?

SJF ottimizza il tempo medio di risposta però ho la starvation.

4) Perchè Round Robin non è fair per i job IO bound?

Round Robin non è “equa” per i job IO bound perché non tiene conto del fatto che potrebbero essere molto più corti del TimeSlice ad essi assegnato.
Facciamo un esempio:

- Ho un TimeSlice di 100ms.
- Ho un Job IO Bound che si blocca dopo 1ms.
- Ho 10 Job CPU Bound che utilizzeranno tutto la loro TimeSlice.

- Immaginiamo che il Job IO Bound venga eseguito per primo, dopo 1ms si blocca e dà la precedenza al prossimo.
- Lo scheduler lo rimette in fondo alla coda.
- Ora il Job IO bound rientrerà in esecuzione dopo 1 secondo pieno perchè si ritrova davanti 10 job con 100ms di TimeSlice ciascuno.

5) Max - Min fairness

Questo algoritmo di scheduling funziona cercando di re-distribuire lo scarto che esiste tra la durata di tempo effettiva di un Job e il termine previsto del suo TimeSlice in modo equo.
Se un Job richiede meno tempo del suo TimeSlice allora il tempo in eccesso viene re-distribuito equamente tra il resto dei Job.
Esempio:
TimeSlice di 100ms.
Ho 3 Task:

- Task A è IO bound e consuma solo 5% del Time Slice.
- Task B è CPU bound e consuma tutto il TimeSlice.
- Task C è CPU bound e anche lui consuma tutto il TimeSlice.
- **Iterazione 1:** Task A richiede solo 5ms del TimeSlice.
- **Soddisfo A:** Assegno a Task A il suo 5%.
- **Re-distribuisco:** Il rimanente TimeSlice è ancora disponibile per il 95%. Rimangono i due task B e C.
- **Iterazione 2:** Divido il rimanente 95% equamente tra B e C. Ognuno si becca 95% / 2 = 47.5% del TimeSlice.
- **Risultato finale:**
    - Task A: 5% (Totalmente responsivo)
    - Task B: 47.5%
    - Task C: 47.5%

6) Parla dell MFQ

Multilevel Feedback Queue è una politica di scheduling generalmente **pre-emptive Round Robin** che opera su un vettore di Ready Queue.

Il vettore delle queue è disposto in ordine di priorità, la coda 1 ha priorità più alta della coda 0, e vengono mandati in esecuzione prima i job con la priorità più alta.
Lo scheduler manderà in esecuzione i thread di una coda **se e solo se** la coda di priorità soprastante è vuota, in modo **pre-emptive** [5] .
Inoltre le code di priorità più alta hanno un Time Slice più breve delle code basse.

- Se un job esaurisce il suo quanto di tempo viene declassato di un rango, praticamente viene accodato ad una coda di priorità sottostante **e** il suo quanto di tempo aumentato.
- Se un job invece si interrompe per I/O o dà la precedenza **durante** il suo quanto di tempo rimane nel livello attuale ***OPPURE*** viene promosso e accodato ad una coda di priorità maggiore. [6]

Il workflow della MFQ non è tecnicamente ottimo per nulla **MA** offre un buon compromesso per accomodare Job con workload sia variabili che fissi.

Un problema è dato dalla priorità che job IO-Bound hanno su job CPU-Bound, quindi è **possibile la starvation** per i Job bassi.
Per affrontare il problema si implementa un meccanismo di *Aging* , cioè i job che sono rimasti per un Tot di tempo in un livello vengono **promossi.**

Problema della *Priority Inversion*

Cosa succede se un Job basso detiene una risorsa che un Job alto vorrebbe usare?
Praticamente si ha un bottleneck di priorità al Job più alto, deve aspettare che il Job basso rilasci la risorsa in un prossimo Time Slice.
Ho un job alto che deve aspettare come se fosse un job basso.
Per risolvere si possono dare boost temporanei di priorità a job bassi per non rallentare job alti.

[5] Se un job basso è in esecuzione e appare un job alto allora il job in esecuzione viene sospeso e fatto partire il job alto.

[6] Un Job può essere promosso di uno o più livelli in base al contesto, e.g. una interruzione per il disco incrementa la priorità di +1, una interruzione per un mutex +2 , un'interruzione per la tastiera +8 etc. Inoltre le priorità possono essere attribuite anche temporaneamente per N Time Slice.

7) Algoritmi di scheduling multiprocessore

In una CPU multi-core ho la possibilità di redistribuire il lavoro tra i diversi core per velocizzare tutto ma al costo di complicare alcuni algoritmi di scheduling, come tener conto del'**affinità di un thread**.
Ecco alcune meccaniche da tenere in mente :

- **Affinità di un thread**: Il concetto di affinità di un thread si basa sull'ottimizzazione delle cache, cioè un thread è affine ad un determinato core poiché i dati che utilizza stanno nella cache di quel core specifico.
  Se il thread viene assegnato ad un core a cui non è affine **la cache non avrà alcun dato del thread e la CPU incontrerà inerzia mnemonica** (deve ripescare tutte le linee di cache che il thread tocca)**.**
  Per evitare questa inerzia mnemonica lo scheduler cerca di schedulare il thread sullo stesso core.
- **Redistribuzione del lavoro:** Un core che si ritrova meno task della media cercherà di "rubare" lavoro agli altri Core così da massimizzare il throughput del sistema.
- **Comunicazione e coordinazione tra thread:** Un'applicazione concorrente su multicore avrà sicuramente delle sezioni di codice specifiche per coordinare i thread dell'applicazione, come ad esempio una **barrier** .

Algoritmi:

- **Oblivious scheduling:** Questo è un algoritmo che non tiene conto dei meccanismi descritti sopra. Schedula ogni thread indipendentemente dai suoi "fratelli".
  Il problema di questo approccio è che c'è un divario abissale tra le assunzioni che il programmatore fà quando scrive il programma concorrente e come effettivamente viene eseguito sotto il cofano
  Un programmatore scrive sempre un programma multi-threaded con l'assunzione che questi thread cooperino tramite una comunicazione tra essi facendo lavoro **di squadra**; assunzione invalidata da questo tipo di scheduling che invece tratta ogni thread **individualmente ,** causando delay nell'esecuzione e coordinazione.
  Esempio in produttore/consumatore:
  - Produttore A e Consumatore B
  - Produttore A viene de-schedulato prima di rilasciare il Mutex
  - Consumatore B prova ad acquisire il Mutex
  - Consumatore B si mette in attesa poichè Mutex occupato
  - Consumatore B deve aspettare più tempo del previsto *(rispetto agli algoritmi descritti sotto)* perché il produttore si è fermato.
- **Affinity Scheduling**: Questo framework di lavoro tiene conto del concetto di affinità descritto sopra.
  Questo algoritmo di scheduling cerca di ottimizzare **la coordinazione spaziale**, cioè

decide **DOVE** il thread verrà eseguito.
Il thread si ritroverà sempre nello stesso spazio in modo che ne abbia “familiarità”.

- **Gang Scheduling:** L’idea è quella di far schedulare i thread di un'applicazione in modo che **vengano eseguiti tutti insieme parallelamente ma su core diversi .**
  I thread formano una singola e atomica unità schedulabile, cioè o vengono schedulati tutti insieme o nessuno di loro viene schedulato. (All or none)
  Il Gang Scheduling ottimizza la **coordinazione temporale** , cioè decide **QUANDO** il thread verrà eseguito.
  Questo approccio migliora la responsività della comunicazione e coordinazione tra i thread poichè ho la garanzia che siano attivi insieme.
  Problemi del Gang Scheduling sono:
    - Potrebbe avere problemi nel commutare da un processo all’altro.
    - Frammentazione delle risorse, cioè la mia gang è troppo grande e non riesco ad assegnare abbastanza core per farla avviare.
- **Space Sharing:** Lo scheduler assegna ad ogni processo un subset dei core disponibili, spartendoli tra i diversi processi.
  Ad esempio ProcessoA riceve i core 0, 3, 7 mentre ProcessoB riceve i core 1,2,4.

Virtual Memory

1)B&B, Segmentation, Paging & Segmented paging: Pro e contro

Format di domanda abbastanza frequente. Ho agglomerato tutto in una domanda sola per coprire tutte le possibili varianti che sono state chieste e tanto vale rispiegare tutto come se fossero appunti.
Per scrivere sta risposta in modo esaustivo servirebbe un documento a parte.
Prima di tutto prendiamo in ottica il problema alla base: vorrei partizionare il mio spazio degli indirizzi per gestire meglio l'allocazione dei miei programmi nel modo più veloce, compatto e versatile possibile.

Base and bound

Il primo approccio (che fa schifo) è quello di assegnare a ciascun processo una singola area di memoria contigua e registrare in una tabella dei processi sia l'indirizzo da dove inizia sia quanto è grossa.

Perchè fa schifo?
Perchè ho agilità inesistente.
I programmi sono dinamici e hanno regioni di memoria che crescono in modo e velocità diverse.
La heap cresce in modo sicuramente più veloce dello stack, cosa succede se la regione allocata per la heap non mi basta più e collide con lo stack?
Dovrei spostare tutta la regione dello stack , abbassarla e poi aumentare il bound della heap, o viceversa prendere tutto l'heap e spostarlo in alto , per poi aggiornare sia la base che il bound.
Perdo un fottio di tempo.
Inoltre se ho più processi che vengono allocato per poi venir deallocati potrei avere tanti "buchi" nella memoria che non mi permettono di allocare alcuna regione contigua per poterci ficcare un nuovo processo **,** questo fenomeno si chiama **frammentazione esterna** .

PRO

- Frammentazione interna trascurabile

CONTRO

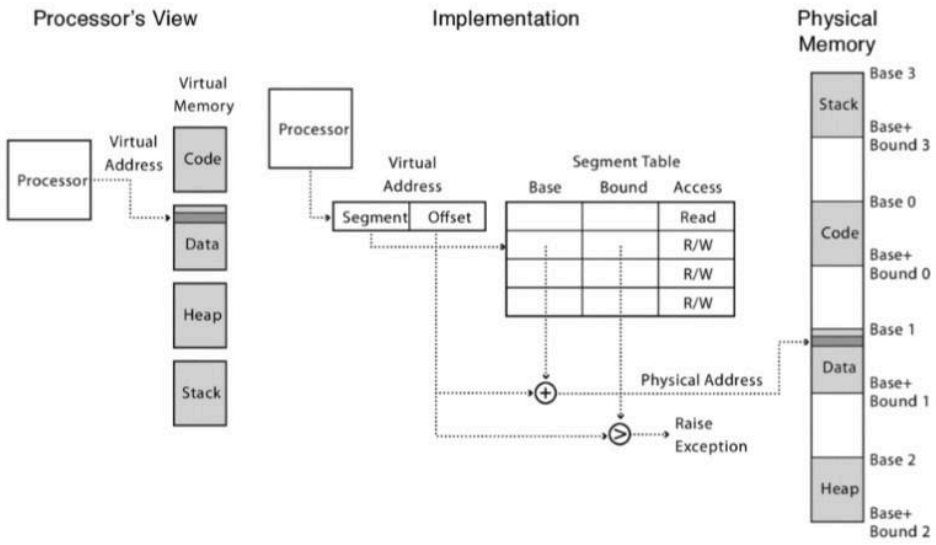
- Frammentazione esterna
- Non tiene conto della natura dinamica delle diverse componenti dei programmi
- I processi non possono condividere la memoria in modo semplice
- Non posso impostare permessi o protezioni per le componenti del programma

Segmentation

Allora anziché dare ad un processo una singola area di memoria contigua gliene dò di più.
Queste aree le chiamo segmenti.
Praticamente suddivido ogni processo in tanti " *organi interni* " [7] con una funzione specifica per il mio programma:

- Segmento codice
- Segmento heap
- Segmento stack
- Segmento dati
- Eventuali altri segmenti per librerie esterne

Ognuno di questi segmenti ha lunghezza variabile proprio come il base and bound.
Per tener traccia di questi segmenti li registro in una *Segment Table* e ne ho una per ogni processo.



Come la implemento?

Interpreto gli indirizzi virtuali in due parti:

- La più significativa indica l'indice della entry nella *Segment Table*
- I rimanenti bit meno significativi mi dicono la dimensione di ciascun segmento e contengonol'offset **nel** segmento.

Esempio: In 32 bit, un'architettura che interpreta 4 bit per il segmento potrà offrire a ciascun processo un massimo di 2^4 segmenti con una lunghezza massima di $2^{32-4} = 2^{28}$.

Se un segmento esaurisce lo spazio verrà aumentato il suo valore "Bound" nella tabella.

ATTENZIONE: la lunghezza delle due partizioni **è fissata** dall'architettura **ed è naturalmente potenza di 2**.

[7] In tante implementazioni l'indirizzo virtuale del Segmento Codice inizia da 0x00000000 mentre lo stack spesso inizia ad indirizzi più alti.

Il problema della **frammentazione esterna** non è stato ancora risolto, dovrei deframmentare la RAM ogni tot tempo senza però compattare troppo i processi per lasciare un pò di spazio per eventuali estensioni dei bound.
Insomma un casino.

PRO

- Frammentazione interna trascurabile
- I segmenti si allineano bene con la struttura e il funzionamento dei programmi
- I processi possono condividere i segmenti, basta che abbiano la stessa entry nella Segment Table
- Posso impostare livelli di protezione e permessi con granularità a segmento

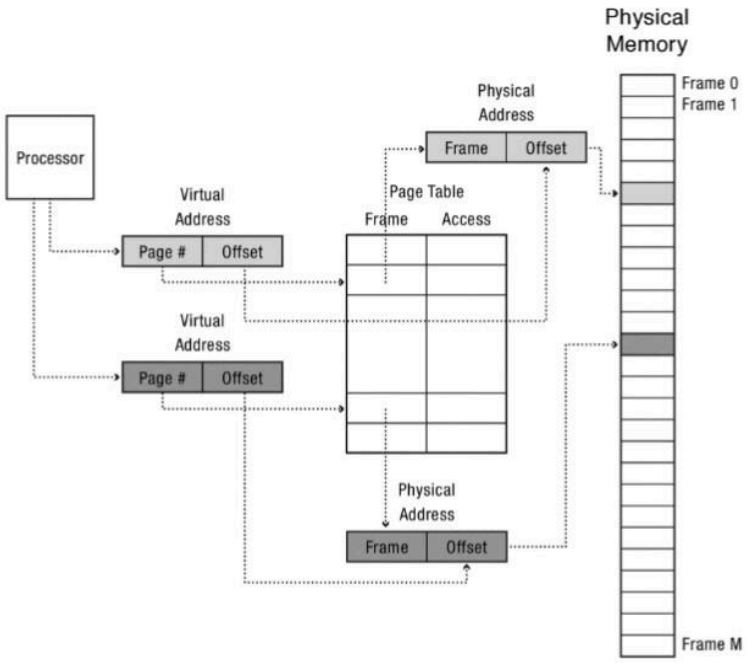
CONTRO

- Frammentazione esterna
- Il numero massimo di segmenti che ogni processo può avere è fissato dall'architettura
- È complicato gestire segmenti che esauriscono lo spazio allocato
- Deframmentazione complicata

Paging

Se possiamo dire che la segmentazione è una pratica che spezzetta il lato software, la **paginazione** è una tecnica che invece spezzetta il lato hardware.

Nel paging la memoria principale viene partizionata in tanti **slot della stessa dimensione** chiamate ***Page Frame*** e viene trattata come un enorme array uniforme di slot e ad ogni processo viene assegnata una *Page Table* che registra le sue pagine .



Anche qui l'indirizzo virtuale viene partizionato in due:

- La più significativa mi dice l'indice della pagine nella *Page Table*
- La meno significativa l'offset all'interno della pagina.

Anche in questo caso le pagine hanno **dimensione fissata** dall'architettura ed è implicitamente **una potenza del 2.**

Importante : Una entry della *Page Table* registra il numero identificativo del Frame e per risalire al suo indirizzo basta moltiplicarlo per la grandezza delle pagine di memoria . Poi da lì aggiungo l'offset e trovo il dato che cerco.

Il numero massimo possibile delle pagine dipende da quanti bit vengono interpretati per l'indice della *Page Table* e la dimensione di ciascuna pagina dipende dai bit che rimangono.

La **forza del paging** risiede nella gestione più uniforme della memoria fisica, trovare spazio libero è molto più semplice e la frammentazione esterna è praticamente impossibile.
Per trovare una pagina libera basta avere una lista di Frame liberi oppure una bitmap.

Inoltre i processi non si accorgono che i loro accessi in memoria vengono reindirizzati verso locazioni fisiche diverse, per loro sembra tutto contiguo.

Vera chicca è che le pagine posso caricarle "**on demand**" dal disco man-mano che il programma si evolve tenendo in memoria solo le pagine a cui ho acceduto nell'ultimo quanto di tempo T, scaricando quelle più vecchie. (Swap Out)
Questo meccanismo si chiama ***On Demand Paging*** che verrà trattato in dettaglio più avanti.
Per implementarlo ho bisogno che la tabella delle pagine abbia un **bit di validità** per ogni entry per capire se la pagina è presente in memoria oppure no.
Quando il processo accede ad una pagina invalida la CPU si ferma e attende che venga caricata dal disco.
Con questo strumento posso iniziare l'esecuzione di un programma senza dover aspettare che tutto il suo codice sia in memoria *(Fast Program Start)* e in più posso tenere in memoria solo le pagine che effettivamente sto usando.

La Page Table però contiene coppie *Indirizzo Virtuale → Numero Frame*, mi farebbe comodo avere una tabella che mantiene la relazione inversa per capire per ogni Frame da quanti indirizzi virtuali è puntato.
Questa tabella si chiama **core map** e la interrogo ogni volta che un processo termina per capire se i Frame a cui puntava sono condivisi da altri processi oppure se li posso rilasciare.

Un altro piccolo vantaggio è che ora la gestione dei permessi è granulare a livello di pagina, ogni entry nella Page Table avrà una serie di bit che rappresentano i permessi di lettura/scrittura/accesso.

NOTA: Al momento di un context switch la tabella del processo uscente viene sostituita con quella del processo entrante. Internamente viene cambiato un registro che contiene l'indirizzo della tabella del processo attuale. Questi puntatori a tabelle stanno nel PCB.

Finora sembra tutto molto bello se non fosse per un paio di edge case che potrebbero complicare le cose drasticamente.

Se le pagine sono troppo grandi rischio di inasprire la **frammentazione interna** , cioè ho spazio disponibile ma non lo uso.
Se invece sono troppo piccole avrò una *Page Table* del processo che riserva spazio per un sacco di entry!

Il problema della Page Table grossa ce l'ho soprattutto se ho uno spazio degli indirizzi a 64 bit.

Caso 32 bit
In 32 bit se ogni pagina è grande 4 KB = 2^{12} allora per ogni Page Table dovrò riservare 2^{20} entry.
L'unità minima che le CPU possono allocare è un byte, quindi per rappresentare un numero a 20 bit mi servono per forza 3 byte , spreco 4 bit ma almeno riesco a lavorarci.
2^{20} entry ognuna da 3 byte richiedono

3* 2^{20} = 3 MB di Malloc

Caso 64 bit

Per enfatizzare assumiamo che ogni pagina sia più grande di prima anziché avere 12 bit di indirizzo ne ha 24, quindi ogni pagina è di 16 MB = 2^{24} byte.
Per ogni Page Table dovrò riservare $2^{64 - 24} = 2^{40}$ entry.
Un numero di 40 bit richiede esattamente 5 byte , 2^{40} * 5
LA TABELLA PESA ALMENO 5 TERABYTE.

PRO

- Niente frammentazione esterna
- Trovare memoria libera è più semplice e veloce rispetto alla Segmentation
- I processi possono condividere le pagine se hanno la stessa entry nella Page Table
- ***On demand paging***

TRADE OFFS

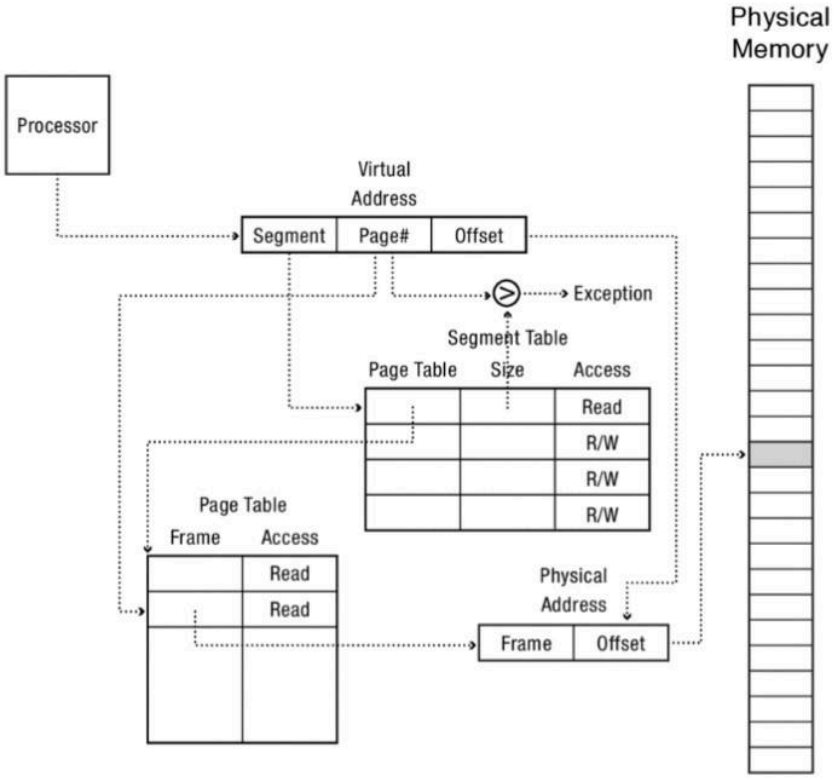
- Pagine piccole riducono la frammentazione interna ma richiedono più entry nella Page Table, viceversa pagine grandi riducono la Page Table ma accentuano la frammentazione interna.
- Gestione dei permessi a grana più sottile rispetto ai segmenti, però ora cambiare i permessi di un "organo" del processo richiede accedere a tante entry.

CONTRO

- Uno spazio degli indirizzi a 64 bit non è proponibile
- Il numero massimo pagine e la loro grandezza sono valori fissati dall'architettura

Segmented paging

Prendiamo lo suddivisione del processo della Segmentation e prendiamo lo suddivisione della memoria fisica del Paging.
Li fondo insieme.
Ottengo un framework di gestione ibrido che divide i processi in “organi interni” logici e poi ognuno di questi organi viene impaginato.



Qui l’indirizzo virtuale viene partizionato in 3, sempre in potenze di 2:

- La parte alta indicizza il segmento del programma
- La parte interna indicizza la pagina che compone il suddetto segmento
- La parte meno significativa indica l’offset nella pagina

Il meccanismo è praticamente lo stesso dei due metodi descritti prima ma concatenato.
Ho bisogno di più tabelle ora, una *Segment Table* per processo + una *Page Table* per ogni segmento.
Ogni processo referenzia solo la Segment Table nel PCB.

Ogni segmento ha associato una *Segment Table* che contiene tre informazioni:

1. L’indirizzo di memoria della *Page Table* che compone il segmento in questione
2. Il numero di pagine registrate in questa Page Table.
3. Permessi applicati all’intero Segmento

La *Page Table* contiene:

1. Il numero identificativo del Frame
2. Permessi mirati al singolo Frame

Workflow del Segmented Paging

1. Il processo accede ad un certo indirizzo
2. La CPU invia l'indirizzo al sottosistema di memoria
3. L'indirizzo viene partizionato come in figura
4. L'indice di segmento estratto è il numero del segmento che mi interessa, è la entry della *Segment Table* da interrogare.
5. La *Segment Table* acceduta a quell'indice contiene l'indirizzo di memoria di una *Page Table* contenente le pagine che compongono il segmento in questione
6. L'indice di pagina estratto dalla partizione interna dell'indirizzo è la entry della Page Table da interrogare
7. La entry nella *Page Table* indica il numero identificativo del Frame
8. Per calcolare l'indirizzo in memoria del suddetto Frame moltiplico il suo numero per la grandezza di una singola pagina
9. Il sottosistema di memoria aggiunge l'Offset all'indirizzo appena calcolato
   Espressione risultante :

```
return Mem [ FrameNum *sizeof(Frame_t) + offset]
```

Problema del 64 bit

Nel paging avevamo il problema degli indirizzi a 64 bit che richiedevano un'allocazione mostruosa di byte per una tabella.
Questo problema è attenuato dal fatto che ora l'indirizzo è separato in tre parti che mi offre più modularità: magari i primi 16 bit li uso per la Segment Table che peserà circa 64KB , i prossimi 24 per la *Page Table* che peserà 16MB e i restanti 24 per l'offset (quindi ho pagine grandi 16MB).
Quando un processo viene creato il SO è inizialmente sempre obbligato ad allocare solo la *Segment Table* , poi allocare dinamicamente le Page Table in base ai segmenti acceduti nel corso del tempo.
Questo concetto è generalizzabile e si chiama **Multilevel Paging** .

PRO

- Niente frammentazione esterna (Ereditato dal paging)
- Alloco solo Page Table per i segmenti attivi
- La struttura logica del programma è organizzata in macro-aree specializzate (Ereditato dal Segmentation)

- Trovare memoria libera è più semplice e veloce (Paging)
- I processi possono condividere sia interi segmenti sia le singole pagine (Fusione di entrambi)
- ***On demand paging*** (Paging)
- Spazi di indirizzamento a 64 bit ora sono supportati.

TRADE OFFs

- Pagine piccole riducono la frammentazione interna ma richiedono più entry nella Page Table, viceversa pagine grandi riducono la Page Table ma accentuano la frammentazione interna.

CONTRO

- Il numero massimo segmenti, pagine e la loro grandezza sono valori fissati dall'architettura

Breve commento sulle Inverted Page Tables

E se anzichè usare tutti i metodi visti sopra e trattassi le pagine di memoria come un grosso vettore alla base di una hash table?
Prendo la parte significativa dell'indirizzo virtuale e ne calcolo un hash che mi dice il numero identificativo del Frame.
Si può fare e velocizza la traduzione degli indirizzi perchè risalire ad un indirizzo fisico con un calcolo è sicuramente più veloce di un accesso in memoria.
Il problema è che implementare una buona funzione hash in hardware è complicatissimo.

2) On-Demand-Paging

L' ODP è un meccanismo per la gestione della memoria secondo il quale le pagine di memoria del processo **vengono copiate dal disco e piazzate in memoria solo quando il processo ci accede** (sia in lettura che scrittura).

Questo meccanismo offre un grande vantaggio, quello di **far girare programmi molto più grandi di quanto la memoria principale permetterebbe se fossero caricati interamente,** tengo solo le pagine effettivamente usate e scarico quelle meno usate, **usando algoritmi di rimpiazzo.**

Funzionamento

Prima di tutto ho bisogno di un bit di "Presenza" o "Validità" per ogni entry della Page Table per capire se la Page è in memoria o sul disco.
Inizialmente tutte le pagine della Page Table sono marcate come "Invalide" cioè non presenti , quando il processo prova ad accedere ad una invalida si innesca un "**Page Fault"**, che sospende il processo .
Viene fatto partire l'handler dei Page Fault , viene cercata nella Free Bitmap o nella CoreMap la prima pagina libera disponibile.
Se non viene trovata pagina libera devo **sfrattare** una pagina occupata in base a dei criteri.

Una volta ricavato un Frame libero il SO ci carica la pagina del programma, imposta il bit di presenza ad 1 e il processo viene fatto ripartire ripetendo l'istruzione che ha causato il Page Fault.

Come faccio a trovare la pagina giusta sul disco

Abbiamo detto che un Page Fault causa una copia di una pagina **dal disco.**
Ma come fà il Kernel a sapere dove sta questa pagina ricercata?
Ho 2 possibilità:

1. La Pagina che cerco è nello **Swap Space**[8]**,** a quel punto la Taglio dal disco e la Incollo in memoria principale. (Swap In)
2. Non è nello Swap Space (non è stata mai caricata oppure è stata cancellata), a quel punto devo cercare nel disco.

Nel caso di pagine **dinamiche** come il segmento Heap e Stack ovviamente ha senso cercarle solo nello Swap Space.
Prendiamo in esame il secondo caso **per i dati statici come il segmento codice `.text` e variabili inizializzate staticamente del segmento `.data`.**
Quando un eseguibile viene caricato in memoria il Kernel può leggere informazioni importanti sull'.exe in questione, incluso **l'offset e dimensione** di questi segmenti **nel FILE**.
Il Kernel salva queste informazioni di mappatura in una tabella del PCB.
In base al segmento che ha causato il PF e l'offset, il Kernel capisce quale pagina caricare dal disco.

Quale è il piccolo problema di questo approccio?
A volte la CPU potrebbe essere interrotta perchè il sottosistema di memoria deve caricare la pagina dal disco.
Il problema è attenuato da algoritmi di **pre-paging**[9] che sfruttano la località degli accessi per "prevedere" in modo euristico quale sarà la prossima pagina a cui accederò.

[8] In realtà lo Swap Space non esiste più da anni.
I SO moderni trattano le pagine da spostare sul disco come dei **file** veri e propri senza richiedere che venga sacrificato spazio sul disco per allocare questo famigerato Swap Space.
Seppur attualmente errato chiamare questa area di memoria "Swap Space" il nome rende bene l'idea di cosa succede sotto il cofano.

[9] Il pre-paging è una tecnica che richiede di sapere in anticipo quale sarà la prossima pagina del Working Set a cui accederò, quindi non può essere usata per tutte le pagine, solo per alcune come lo stack che cresce in una direzione soltanto.

Come funziona lo sfratto

Al momento di un Page Fault se la memoria è piena devo necessariamente sfrattare una pagina tra le tante, scegliendo in modo furbo da evitare il **thrashing** (Il sistema impiega più tempo per swappare/caricare le pagine che eseguire istruzioni).

Ogni entry della Page Table ha informazioni che ci possono aiutare nel lavoro.
Abbiamo il bit di Validità o Presenza, bit di Utilizzo, bit del permesso di lettura, bit del permesso di scrittura, bit di Modifica.
Ovviamente non posso sfrattare una pagina che ha il bit di Presenza a 0, quindi questa evenienza non ha senso trattarla.

Immaginiamo di aver selezionato una pagina da eliminare con un algoritmo tipo LFU, LRU, FIFO…etc.
La pagina selezionata potrebbe essere riferita da Page Table di altri processi, per localizzarle uso la core map, poi imposto la entry della pagina sfrattata come **invalida** cioè **non presente.**
Invalido eventuali riferimenti a questa pagina anche nel **TLB. (TLB Shootdown)**

Se la pagina da sfrattare è PULITA, vuol dire che i suoi contenuti sono identici a quelli sul disco, la rimuovo e basta.

Se la pagina selezionata è “dirty” vuol dire che i contenuti sono stati modificati dall'ultima volta che è stata caricata dal disco.
Per non perdere queste modifiche **la pagina viene spostata sul disco sotto forma di file**, in vecchie architetture veniva solo copiata in una area del disco chiamata *Swap Partition***.**

3) Algoritmi di sfratto

MIN: Questo è il Buddha degli algoritmi di sfratto , è teoreticamente perfetto ma praticamente irraggiungibile.
Dice di sfrattare la pagina di memoria che non utilizzerò per più tempo nel futuro.
Inutile a meno che uno non abbia una macchina del tempo.

FIFO: Sfratto la pagina di memoria che **è rimasta in memoria per più tempo**.
Chi mi dice però che la pagina più vecchia è la migliore da togliere?
Se è ancora utilizzata la toglierei per poi doverla rimettere più tardi.
Inoltre soffre della **Anomalia di Belady (** *Belady's Anomaly* ***)*** secondo la quale un aumento della capacità della memoria non comporta necessariamente a meno page fault.

LRU: Least Recently Used, sfratto la pagina che **ho usato più tempo fà possibile.**
Approssima bene MIN ma ha un overhead pesante e richiede più spazio per le entry:
Per tenere traccia delle entry usate meno recentemente ho per forza bisogno di un ***time stamp*** di generalmente 8 byte che aggiorno ad ogni accesso alla entry che richiede una query ad un orologio di sistema ad alta risoluzione (e.g. Epoch, Cronometro di sistema etc.) e tenendole ordinate per timestamp crescente.

Second Chance: Un FIFO modificato. Ciascun numero delle pagine è ordinato in una linked list dove in testa ho la pagina più vecchia.
Periodicamente viene scorsa la lista e viene esaminato il bit di utilizzo.
Se è 0 la pagina viene rimossa.

Se è 1 la pagina è stata usata dall'ultimo *sweep.*
Il bit di Uso viene impostato a 0 e la pagina viene spostata in fondo alla lista.
Il problema è che spostare puntatori/nodi è costoso.

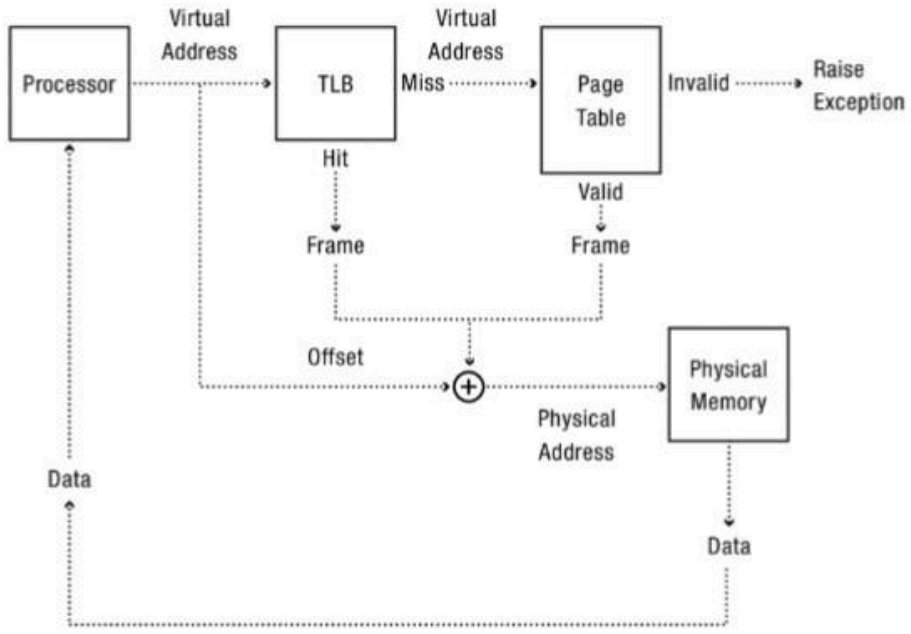
Clock: Uguale a Second Chance ma la lista non è ordinata ed è *circolare* per spostare meno nodi.
Durante lo sweep se il bit di Uso è 0 la pagina viene rimossa e il selettore fatto avanzare al prossimo nodo della linked list.
Se è 1 invece viene solo settato a 0, non viene spostato il nodo in fondo.
Questo algoritmo è eseguibile sia in modo sincrono che asincrono.

- Se sincrono i Page Fault avranno overhead aggiuntivo.
- Se gira in parallelo su un thread Kernel avrò meno overhead ma gestione più complessa.

Questo thread mantiene una lista di pagine recentemente non utilizzate.
Al momento di un Page Fault questo thread rimuoverà una pagina della lista.

4) Parlare della TLB.

La gestione dell memoria virtuale richiede un sacco di accessi a tabelle in memoria.
2 accessi con Segmentation puro e Paging puro (Accedo alla tabella poi in memoria)
3 accessi con la Paged Segmentation.
Tutti questi accessi mi rallentano tantissimo.
Posso migliorare la situazione? Sì. Installo un TLB.
Il TLB è una cache **totalmente associativa** (la più veloce e flessibile) che mi serve per tenere traccia delle ultime traduzioni indirizzo logico -> frame così da non dover accedere tutte le volte nella tabella e quindi in memoria.
Guardando il disegno il TLB è interposto tra la PageTable e la CPU; con questa configurazione quando la CPU emette un indirizzo di memoria il TLB lo intercetta per primo e controlla se l'indirizzo è contenuto in essa:
Se sì ho un TLB hit e procedo con un solo accesso in memoria diretto.
Se no ho un miss e devo fare la traduzione come l'abbiamo descritta fin'ora, accedo in memoria verso la Page Table e poi ci ri-accedo per il dato.



5) Quali sono i possibili problemi del TLB?

In realtà i problemi del TLB sono legati ai problemi generali delle cache.
Tutte le cache offrono uno “snapshot” discretamente aggiornato di una memoria che può essere alterata da fattori esterni come DMA, Memory Mapped IO, etc.
Il TLB offre uno snapshot delle traduzioni recentemente effettuate da VAddress→Frame.

Cosa succede se una pagina di memoria viene spostata o i permessi vengono modificati?
Succede che ora il TLB non ha uno snapshot aggiornato e la entry vecchia deve essere eliminata. (**TLB Shootdown)**

Nel context switch?
In un context switch il TLB avrà tutte traduzioni inutili perché il processo nuovo utilizzerà pagine completamente diverse (a meno che non abbia qualche pagina condivisa col processo precedente).
In un primo approccio si potrebbe pensare ad un flush totale del TLB, però una flush comporta un overhead abbastanza lungo quindi è meglio optare per un **Tagged TLB** cioè ogni entry del TLB è taggata col PID del processo che ha innescato la traduzione.
In questo modo quando faccio una traduzione confronto il PID come se fosse un TAG e se non coincide allora non è valido.Cosa cambia se ho virtual cache e physical cache.28) Una tabella di secondo livello é grande 2k entrate. Quante tabelle di primo livello ho e quante di secondo ho? Ho 11 bit di offset.

6) Perché si usa la paginazione multilivello.

Con la paginazione monolivello dovrei allocare una tabellona, con multilivello devo allocare solo le tabella di cui ho bisogno.

7) Algoritmo del Working Set delle pagine di memoria.

Reminder: Working Set è l’insieme delle pagine riferite nell’ultimo periodo T.

L’algoritmo del Working Set è una procedura di rimpiazzo per le pagine che cerca di mantenere in memoria il Working Set dei processi e buttare via ciò che non serve , sempre evitando il **thrashing** come la peste.

L’ambiente di lavoro dell’algoritmo è l’insieme delle pagine effettivamente in memoria chiamato **Resident Set,** quindi praticamente lavora sulla *core map*.
Quindi in un certo senso cerca di spazzare via le pagine inutili a livello globale, per tutti i processi.
Ovviamente posso anche avere un algoritmo che si scorre tutte le Page Table di tutti i processi anche se sperimentalmente funziona un pò peggio.

Funzionamento

Questo è un algoritmo progettato per essere innescato a causa di un Page Fault o ridurre il numero di pagine di un processo su richiesta (PFF algorithm) e eseguito periodicamente da un Daemon.

L'algoritmo ha bisogno di un parametro T che indica un'età limite sopra la quale le pagine vengono sfrattate.

L'algoritmo si avvale del bit di Used (a.k.a. Referenced) della pagine e il TLR, Time of Last Reference che è il timestamp dell'ultimo utilizzo della pagina , infine ricava il timestamp del tempo attuale.

ATTENZIONE: La Core Map mi aiuta solo a non dover scorrere tutte le page table di tutti i processi.
Per ogni entry della Core Map riesco a ricavare il numero di pagina virtuale & i PID dei processi che referenziano il Frame in questione.
Per ogni processo ricavo la Page Table e con il numero identificativo della pagina virtuale riesco ad accedere immediatamente alla entry che cerco.
Sono più passaggi ma almeno non devo scorrere tutte le tabelle di tutti i processi.

L'algoritmo calcola poi l'età della pagina , cioè la differenza tra il timestamp attuale e il TLR.

Per ogni pagina:

- **Se bit U = 1** , l'algoritmo imposta il TLR della pagina uguale al timestamp attuale e imposta U = 0;

- **Se bit U = 0** , se l'età della pagina **è maggiore o uguale** di T la pagina viene rimossa perchè decade al di fuori della definizione di Working Set, se invece è minore viene mantenuta in memoria

Questo algoritmo è spesso implementato come Working Set Clock, cioè un misto tra WS e Clock, dove ho i vantaggi di entrambi.
Ha lista circolare di page Frame, vengono scorse tutte e a ciascuna viene applicato l'algoritmo discusso sopra.

8) Leasy recently used

9) Frequenza dei fault (Page-Fault Frequency algorithm)

Mentre l' On Demand Paging è il braccio che mi porta le pagine in memoria, PFF è la mente che offre **l'ingegno** per decidere quante pagine ogni processo dovrebbe avere per lavorare in modo efficiente **ed evitare thrashing**.

Premessa abbastanza ovvia: generalmente nei sistemi operativi è bene che ogni processo abbia diritto ad un numero **non infinito** di Page Frame per varie ragioni come:

- **Stabilire un livello di fairness sulla spartizione di pagine tra i processi,** non voglio che un processo a bassa priorità ottenga X pagine dopo X faults (X molto grande) quando magari c'è un altro processo a più alta priorità che ha una necessità di pagine che non riesce a soddisfare per mancanza di spazio *(Starvation)* causando questi due processi a contendersi i frame in memoria.
- **Lo spazio degli indirizzi non è illimitato,** non dispongo di bit infiniti in una macchina. Concetto discusso nella sezione **Paging** e **Segmented Paging.**

Le domande che sorgono spontanee:

1. Come faccio a stabilire questo limite per ogni processo? E se è troppo grande? Se è troppo basso?
2. È possibile determinare un **range di pagine** che ogni processo **dovrebbe avere** per stabilire **fairness** sul numero limitato di Frame?
3. Se sì, allora quante pagine di memoria dovrei effettivamente concedere ad ogni processo per diminuire **thrashing** ?
4. Inversamente, è possibile che un processo abbia un numero troppo alto di pagine, sprecando memoria?

La risposta a queste domande la offre il Page Fault Frequency Algorithm, un algoritmo basato sulla frequenza di page fault generati da un processo in una unità di tempo che mi permette di stabilire un'euristica per concedere **dinamicamente** più o meno pagine di memoria con l'obiettivo di ridurre il **thrashing .**
Se un processo ha fault inferiori ad una soglia X vuol dire che il Working Set è ridotto rispetto al Resident Set, forse ha un limite di pagine troppo alto, gliele tolgo alcune.
Se ha fault superiori ad una soglia Y, il Working Set è più grande del Resident Set, di pagine ne ha troppe poche.[10]

[10] Le soglie inferiori e superiori di frequenza sono stabilite sperimentalmente da benchmark.

Il rationale dietro PFF è quello di monitorare i fault di ogni processo, e impostare una quota per l'On Demand Paging per adattare il Resident Set al Working Set nel modo più efficiente ed equo possibile.
L'On Demand Paging ***da solo*** non sà se un programma ha un fabbisogno di pagine maggiore superiore o inferiore alle pagine finora a lui concesse.

Se un processo raggiunge un fault rate alto e continua a richiedere pagine l'On Demand Paging PURO sfratterà una pagina che però fà sicuramente parte del Working Set, **non ho scelta**.
Il PFF invece **tenta** di dare più pagine per questo specifico processo così da non rallentare tutto il sistema con **thrashing**.

Invece se un programma ha tanti page frame in memoria che non usa ha un Resident Set più grande del Working Set, li devo rilasciare con un algoritmo tipo Working Set Clock per evitare starvation di altri processi.

Aggiungendo questo ulteriore strato di gestione della memoria mi aiuta a determinare le esigenze correnti di ciascun processo con le seguenti funzionalità:

- **Monitorare le esigenze di ciascun processo,** percependo thrashing ed eventuali sovra-allocazioni è possibile stabilire se il numero di pagine presenti in memoria (Resident Set) è appropriato per il Working Set. Fornendo anche soluzioni estreme per situazioni estreme.

- **Quota di pagine flessibile ed equilibrata:** Se un processo ha raggiunto il "limite massimo" di pagine ma continua a causare thrashing vuol dire che ha un Working Set << Resident Set.
  Il sistema prova ad alzare il limite massimo per rilasciare pressione, a quel punto l'On Demand Paging allocherà pagine rispettando il nuovo limite anziché sfrattare pagine che forse servivano.

- **Sovra allocazioni potrebbero causare starvation,** cioè ho un Working Set minore del Resident Set, in quel caso viene innescato manualmente uno sweep di Working Set Clock per sfrattare eventuali pagine non usate.

- **Sfratto un intero processo in situazioni disperate,** nel caso in cui il Working Set globale sovrasta il numero pagine fisiche totali disponibili in RAM ho una situazione che ha il termine tecnico di **super-mega thrashing schifoso** e bisogno di sospendere un processo e *swapparlo* dalla memoria verso il disco interamente, tutte le pagine.

10) Se ho indirizzi a 32 bit e pagine da 4k quanto è grande la tabella? (2^20)

11) Cosa c'è in una entry della page table?

Nella page table sono presenti entry che mantengono la relazione IndirizzoLogico→IndirizzoFisico insieme ad altre informazioni come i diritti di accesso alla pagina in questione.

12) Cos'è la core map? quante entry ha? (spoiler: quante le pagine fisiche)

La core map è una tabella che ha una entry per ogni pagina fisica del sistema e funziona come controimmagine della PageTable, cioè anzichè mantenere relazioni VAddr⇒Frame ha entry con la relazione inversa cioè Frame⇒VAddr insieme ad altre informazioni, come ad esempio quant

13) Cos'è e dove si usa il Copy On Write?

Si usa nel contesto dei software multiprocesso e più precisamente a seguito di una fork().
Il Sistema Operativo non copia tutte le pagine del processo padre ma copierà solo quelle che il processo figlio modifica.
Questo perchè se il processo figlio **non modifica ma <u>legge e basta</u>** il SO evita di sprecare tempo a copiare tutte le pagine.
Dopo la fork() il processo figlio avrà nella tabella delle pagine/segmenti delle entry che puntano alle stesse pagine di memoria del processo padre, cioè ora le pagine sono condivise, **però sono state impostate come ReadOnly.**
Nel caso il processo figlio decida di modificare queste pagine allora il SO dovrà copiare le pagine di memoria del segmento interessato

14) Quale é la condizione in cui non devo ricopiare la pagina nel disco?

Se non l'ho modificata non la devo ricopiare.

File System

Premessa: In questo capitolo la parola “Blocco” viene usata intercambiabilmente con la parola “Cluster”, così come “Cartella” e “Directory”.

1) Cos'è una directory? Cosa c'é dentro?

Una directory è un file che contiene un'associazione FileName → FileData che può cambiare a seconda del FileSystem di fondo.
In FAT32 le directory contengono FileName→StartingBlockNumber cioè il numero del **primo blocco** del file, insieme ad altre informazioni.
In FFS è immagazzinata l’associazione FileName → i-nodo viene chiamata **Hard Link,** le directory contengono Hard Link.
In NTFS ho sempre un’associazione FileName → Master File Record e si chiama anche in questo caso **Hard Link** ma le directory contengono un sacco di altre cose.
Se una directory è molto piccola si utilizza una organizzazione a lista lineare.
Se è molto grande viene implementata una struttura chiamata B-Tree che ha ricerca logaritmica.

2) Dove sono salvate le directory in FAT32? Come le navigo? Stanno nella file table? Se sì/no spiega il perchè.

Abbiamo detto che le directory sono file, però questi file sono particolari in FAT32.
Infatti vengono salvati nella regione dei **dati,** cioè la stessa regione dove vengono salvati i blocchi dei file.

Ogni directory contiene coppie FileName → ClusterNumber e dato che le directory sono file può contenere anche coppie DirectoryName → ClusterNumber.
Il fatto sopra descritto è cruciale per permettere la navigazione dell’albero N-ario delle cartelle per raggiungerne una.
Assumendo che sia fornito un **path,** per accedere ad una specifica directory devo tracciare il percorso nell’albero N-ario delle directory partendo dalla **directory radice** (il cui blocco è puntato nel **boot record**) e cercare nella directory corrente **il prossimo token del path.**
Se lo trovo mi ci sposto dentro usando il ClusterNumber.
Se non lo trovo il path è invalido o il file non esiste,
In sostanza: La FAT registra tutti i blocchi in entry con una struttura **a nodi linkati** .
I nodi **dell’albero N-ario delle directory contengono entry** che puntano alla testa di queste Linked List.

I contenuti delle directory, cioè i nomi dei file e il relativo numero del blocco da dove inizia, stanno dentro un cluster (a.k.a. blocco).
Questo blocco **deve necessariamente** essere registrato nella FAT, questo perché la FAT registra tutti i blocchi del volume.

Tuttavia sono due strutture completamente diverse e interagiscono quando si cerca o si accede ad un file o una directory.

3) Perchè i nomi dei file NON stanno all'interno del file stesso?

Facciamo finta che i nomi dei file siano esclusivamente dentro sè stessi.
Immediatamente ho un paradosso.

1. **Voglio accedere ad un file.**
2. **Per accedervi mi serve sapere come si chiama il file.**
3. **Il nome del file sta dentro il file.**
4. **Per sapere come si chiama il file ci devo accedere.**
5. **Tornare al punto 1.**

Cioè come faccio ad accedere ad un file se il nome del file sta dentro lo stesso????
Fun Fact: Sto problema mi ricorda quando ho lasciato le chiavi di casa dentro casa
(Daje Cristo)

SOLUZIONE: Creo un riferimento esterno al file con lo stesso **nome** di questo file. Problema Risolto.

Domanda spontanea 1: Ma se ora ho un riferimento al file con lo stesso nome del file ma allora a cosa mi serve avere sto nome pure DENTRO al file???????
Nulla.
Spreco byte e basta.

Domanda spontanea 2:
Cosa succede se due directory puntano allo stesso file ma con nomi diversi?
Modifico il nome interno al file a quello più recente?
Faccio che le directory si mettono d'accordo?
Faccio che un file ha più spazio interno per accomodare più nomi?
Implemento un BFS inverso per modificare tutti i riferimenti ????????
Il problema può essere tranquillamente schivato. Vedere sotto.

SOLUZIONE : ***Tolgo i byte occupati dal nome e li uso per i dati utili.***
Tra l'altro ora ho uno strato di flessibilità in più perchè ora un file può avere identità multiple.
Un file può essere puntato come "Pippo.txt" e magari un'altra directory lo punta come "Pablo.txt" ma il file è sempre lo stesso.

In parole povere:
Il motivo per cui il nome del file non può esistere esclusivamente dentro il file è per un motivo prettamente logico.

Il motivo per cui il nome del file viene tolto dai suoi contenuti è per motivi di “Design Hygiene” [11], flessibilità ed evitare pallottole implementative.

4) FAT32, come si calcola la dimensione della fat, dove si trova e il problema della frammentazione della FAT.

Ha tante entry quanti blocchi ho sul disco. La frammentazione siccome ho blocchi allora ho frammentazione interna. La piú critica é esterna. Se tolgo blocchi frammento la memoria storage. Su hard disk della fat se il blocco é lontano la testina si deve muovere tantissimo.
Su SSD, non ho problemi.

5) FFS. Cosa è un i-node? Cosa c'è nei metadati? E nei blocchi dei puntatori indiretti? Dov'è contenuto il nome del file?

La FFS sta per Fast File System ed è il file system che in passato era presente sulle macchine linux.
FFS opera con gli i-nodi cioè delle strutture dati che contengono metadati , puntatori diretti a blocchi del file o puntatori a blocchi pieni di altri puntatori.
Ogni file è in corrispondenza biunivoca con un i-nodo sul disco. Se ho 200 i-nodi vuol dire che ho 200 file.
Gli i-nodi sono registrati sul disco in una area specifica che si chiama appunto **i-node list.**

In FFS nei metadati ci sono le informazioni descrittive relative al file come:

- Proprietario del file
- Il gruppo di appartenenza
- **HARD LINK COUNT**
- I permessi
- Data di creazione etc.

ATTENZIONE: Il nome del file non sta nei metadati, il nome sta nella directory che è a sua volta un file.

I puntatori indiretti puntano ad un blocco di memoria in cui sono allocati puntatori diretti.

6) Hard Link count, cosa è e a cosa serve?

Il contatore di Hard Link è **importantissimo** negli i-nodi perchè mi fà capire **il numero di directory che puntano all’i-nodo .**

Un Hard Link è una associazione FileName → I-Node.

[11] Evitare cose ridondanti o che potrebbero confondere gli utenti

Ogni volta che viene creato un file viene allocato un nuovo i-nodo e **il contatore degli hard link è impostato ad 1,** perché i dati di fondo dell'i-nodo sono raggiungibili dalla directory in cui ho creato il file.

Quando si elimina un file in FFS, in realtà non stiamo eliminando i dati di fondo ma **stiamo decrementando il contatore di hard link.**
Se il Kernel vede che il contatore è diventato **0** allora procederà con l'eliminazione vera dell'i-nodo.
Ciò per assicurarsi che non venga effettivamente eliminato un insieme di dati ancora puntato da altre directory.

Un Hard Link è un "arco in entrata" per l'i-nodo. Se l'i-nodo non ha archi in entrata è praticamente irraggiungibile e quindi eliminabile.

7) Soft link e hard link.

Un Hard Link è una ***entry della directory*** **(** non è un file, non è "tipo il collegamento di windows" , non è un puntatore) che registra una relazione File Name → i-nodo (FFS) oppure FileName → Master File Record (NTFS).
Un Soft Link è sempre una entry della directory ma registra una relazione File Name→File Name.
Soft e Hard link potrebbero sembrare inutili a prima vista perchè uno si potrebbe chiedere "Perchè mai dovrei fare un soft link quando l'hard link và direttamente al sodo?" ma in realtà non è così semplice; entrambi sono in stati progettati per funzioni diverse e anche per completarsi a vicenda un pò.
Non posso creare Hard Link nè a mount point esterni nè a directory, perchè altrimenti l'albero del FS non sarebbe un DAG ma un grafo con cicli (problema).
Posso però fare queste cosa con i Soft Link.

8) Path: cosa fa la open?

Open è una System Call UNIX e dato una path e i diritti di accesso ritorna un File Descriptor, un intero con segno che rappresenta l'indice di una entry nella tabella dei file descriptor del processo.

Esegue i seguenti passi :

- Switch in Kernel-Mode
- Copia il path della chiamata in Kernel space
- Il Kernel controlla che il path sia valido e che il file esista
    - Se il file non esiste ma il flag di creazione è attivo allora viene creato e aperto
    - Altrimenti errore

- Check dei permessi, se non ho accesso ho una eccezione
- La directory corrente contiene il numero dell'i-node del file
    - Se l'i-node è presente in RAM (come una cache) procedo
    - Se no lo devo recuperare dal disco, ci sarà una regione specifica dell'HDD o SSD dove viene mantenuta la tabella degli i-node, viene trovato e messo in una regione specifica per il Kernel in RAM
- Inserzione di una entry che raccoglie le informazioni del file aperto nella tabella globale dei file aperti (File Descriptor)
- Inserzione del file descriptor nella tabella dei file aperti del processo

- In quella entry ho un'indice della tabella dei file aperti (System wide) , e in quella entry directory trovo pippo.dat ,

  come faccio a risalire alla fat? ci sarà una entry nella fat che mi dice dove inizia pippo.dat

9) Cosa contiene la tabella globale dei file aperti?

Contiene i metadati dei file aperti e altre informazioni che servono per fare la lettura, come il puntatore Read/Write , i flag degli accessi , puntatore di memoria all'i-nodo del file in questione e il contatore di riferimenti (se più file usano lo stesso descrittore)

10) HDD best fit o first fit?

First fit perché é meglio per la località.

11) Quando è rilevante che il FFS sfruttti la località usando i gruppi? (spoiler: con HHD) spiegare i gruppi

12) Difetti principali del FFS

FFS ha i seguenti difetti:

- **File piccoli causano frammentazione interna:** i file molto più piccoli del blocco su disco occupano uno spazio molto maggiore di quanto dovrebbero perché allocano un i-nodo **+** almeno un blocco.
- **Scalabilità limitata per directory giganti:** FFS cerca di piazzare file della stessa directory in posizioni più vicine possibili sul disco. Se una directory ha tantissimi file (magari è una directory speciale di file temporanei o file di sistema) avrò problemi.

- **Richiede un sacrificio di spazio in cambio di performance:** é stato provato sperimentalmente che FFS performa meglio e sfrutta meglio la località **se** viene lasciata libera una porzione del disco di almeno 10% .

13) Come funziona NTFS?

NTFS utilizza una unica tabella che si chiama Master File Table ed è essenzialmente un file.
La master table la carico in memoria? No, sta sul disco e carico le entry necessarie.

Ogni singolo file del sistema è in corrispondenza biunivoca con **una entry** della *Master File Table* e la entry è chiamata *Master File Record* che generalmente è grande 1kb e contiene gli "**attributi**" del file.
Un **attributo** è un componente base del file che può contenere sia metadati che dati in base agli *header* del componente, cioè una serie di byte che identifica il formato del contenuto.
Ad esempio (STO INVENTANDO) l'header **`0xF0FFAB0CCA`** indica che i prossimi byte sono dell'attributo FileName, poi magari **`0xDEADBEEF`** indica che ora inizia l'attributo della data di creazione etc.
Gli attributi includono la data di creazione, chi l'ha creato, dimensione, **e stavolta anche il nome,** e anche i dati stessi.

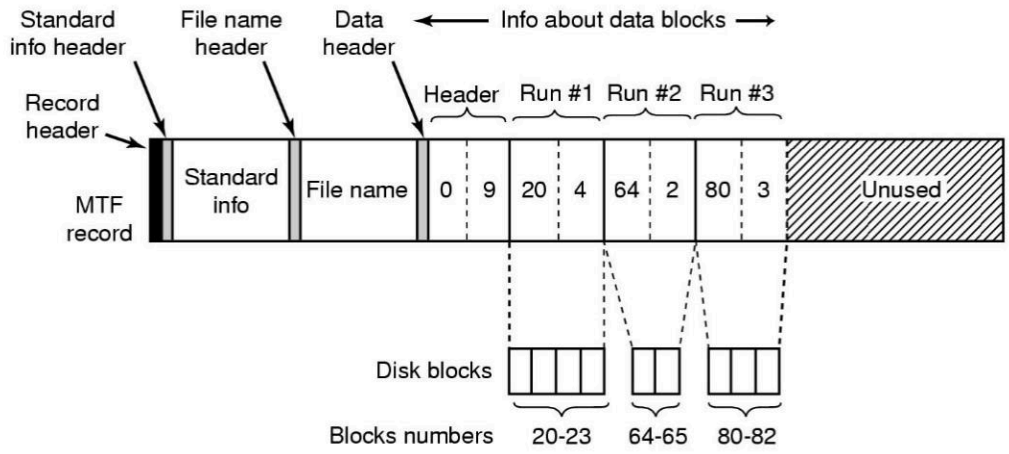
In FFS avevamo detto che il nome **NON** stava nei metadati del file ma era registrato esternamente nelle directory.
Qua invece il discorso cambia per motivi di design, la decisione punta a **mantenere più informazioni possibili** del file nella MFT nel caso il file **venga messo nella cache** (della MFT)**,** inoltre tecnicamente le **identità** dei file rese possibili dagli hard link sono considerate attributi, infatti NTFS offre la possibilità ad un file di avere più attributi File Name al suo interno.

Se il file é piccolo i metadati hanno il contenuto dei file stessi.

Se il file è invece più grande si utilizzano gli **extent**, attributi che contengono l'indirizzo di memoria dove inizia il file & la sua lunghezza IN BLOCCHI.
In realtà un **extent** può essere formato da più segnalatori che registrano i diversi segmenti di blocchi contigui che formano il file.
Questi segnalatori si chiamano **run** ed esistono per accomodare la dinamicitò dei file che possono cambiare di grandezza.

Nella figura in basso si ha la struttura di un master file record e l'attributo dati è formato dall'header e da **3 run.**

- L'header contiene 2 numeri, il primo indica il blocco logico[12] del file che dice al SO da quale posizione deve iniziare a riempire del file; il secondo mi dice da quanti blocchi è formato l'extent.
- Il primo run mi dice che la prossima porzione del file inizia dal blocco fisico 20 ed è lunga 4 blocchi.
- Il secondo run mi dice che i prossimi 2 blocchi iniziano dal blocco 64
- Infine l'ultimo run mi dice che dal blocco 80 i prossimi 3 blocchi sono del file.



[12]È registrato il blocco logico **del file** perchè il SO deve sapere quale parte del file deve costruire.
Inoltre non ho garanzia che gli extent partono sempre dal blocco 0.
Infatti potrei avere **più Master File Record** per lo stesso file e che hanno **extent** che partono da blocchi logici diversi perchè il file è molto grande e necessita di MFR aggiuntivi.
L' extent che rappresenta l'inizio del file parte sempre dal blocco 0 , ma magari il prossimo extent partirà per esempio dal blocco 67.
Se c'è 0 allora l'extent vuol dire che i prossimi **run** devono essere usati per costruire il file da appunto il blocco 0 (si fà SEEK_SET per capirci) .
Se c'è 67 allora vuol dire che i prossimi **run** devono essere usati per costruire il file dal blocco 67.

14) NTFS come gestisce i file piccoli? I dati del file stanno mei metadati se piccolo.

(LA MASTER FILE TABLE STA SUL DISCO)

15) NTFS frammentazione del file.

In NTFS quando un file necessita di tanti extent sparpagliati in diversi MFR allora è considerato frammentato.
Vuol dire che i blocchi del file non sono stati allocati in posizioni contigue ma in tante posizioni diverse e per registrarle tutte mi servono svariati MFR, quindi ho frammentazione pure nella Master File Table.
Questo accade quando un file cresce e il SO non trova abbastanza blocchi liberi, quindi crea più extent che puntano in posizioni diverse.
Per deframmentare il File System è necessario un tool apposito per la deframmentazione appunto.

16) FFS vs NTFS. cosa cambia. Pro e contro di entrambi.

FFS ha problemi con file piccoli, spreco un Inode e almeno un blocco, in NTFS i byte del file vengono direttamente salvati nei metadati.
FFS non deve caricare tutto l' Inode Array in memoria, tengo solo quelli aperti.
In NTFS ho una master file table e ogni File è rappresentato da almeno una entry, idealmente sarebbe una entry per ogni fil ma i file frammentati potrebbero usare più entry).
In ffs I blocchi nell'hard disk li raggruppo insieme. In NTFS i file con blocchi contigui.

17) FAT: 16mb disco, blocchi da 512b, quanti blocchi mi servono per memorizzare la Fat.

- 16 MB = 2^{4+20}
- Blocchi che la FAT deve memorizzare = $2^{24} / 2^{9} = 2^{15}$ entry
- Per raggiungere tutti i 16 MB ho bisogno che l'indirizzo sia a 24 bit = 3 byte , quindi ogni entry è 3 byte
- Lunghezza in byte delle entry in FAT = $3 * 2^{15}$
- Blocchi che la FAT occupa = $3*2^{15} / 2^{9} = 3*2^{6} = 3 * 64 = 192$ blocchi

18) FileSystem da 16MB , 32 bit, blocchi da 1Kb, quanto é grande in blocchi la filetable?

Il mio spazio di indirizzamento mi permette di indicizzare un massimo di 2^{32} byte che sono 4 gigabyte, però io ho solo 16 megabyte che sono 2^{24} .
In realtà è un bene perchè vuol dire che la mia FAT dovrà tenere conto solo di quei 24 bit e non di tutti e 32, avrò una FileTable più piccola. (A meno che non sia un disco mutante che ha capacità variabile e domani mi ritrovo che ha 1 terabyte anziché 16 megabyte).
Poi ogni blocco è 2^{10}, quindi avrò 2^{14} entry nella mia fat.
2^{14} * 4 byte (parola di sistema) avrò 2^{16} byte di fat che in blocchi sono $2^{16 - 10} = 2^{6}$ = 64 blocchi di fat.

$2^{24} / 2^{10} = 2^{14}$
$2^{14} * 2^{2} = 2^{16}$ byte
$2^{16} / 2^{10} = 2^{6}$ blocchi

19) Fat con disco piccolo da 10mb , indirizzi a 32 bit, blocco 1k. Quanti blocchi occupa la fat? Quanti blocchi occupa ?

Con questa configurazione ho $10 * 2^{20} / 2^{10} = 10 * 2^{10}$ blocchi totali.
La fat avrà $10*2^{10}$ entry. Ogni entry è 2^{2} byte perchè ho 32 bit di indirizzo.
La fat pesa $10*2^{12}$ byte, cioè $10*2^{12} / 2^{10} = 10*2^{2}$ blocchi cioè 40 blocchi.

20) Se io ho un file 12kb , ho FFS dove ho 2 puntatori diretti poi uno diretto singolo e uno indiretto doppio. Blocchi da 512 byte.Word 32 bit.

21) FFS Blocchi da 4k file da 4MB, ho 10 blocchi diretti e 3 blocchi indiretti. Quanti uso?

Ho in totale $10 + 3(2^{12} / 2^{2})$ blocchi = 3082 blocchi al massimo
Il file occupa $2^{22} / 2^{12} = 2^{10}$ = 1024 blocchi
Se alloco i primi 10 blocchi diretti mi basta un blocco indiretto per i rimanenti 1014 blocchi.
Uso tutti i blocchi diretti & solo un blocco indiretto.

22) Fat32 indirizzi 32 bit, capacità 64 Mb, blocchi 512 byte , quanto è grossa la fat in blocchi?

64 Mb = 2^{26} capacità in byte

512 = 2^9 byte per blocco

2^{26} / 2^9 = 2^{17} blocchi totali = numero entry FAT

Assumo che ogni entry della FAT è un intero a 4 byte senza segno

2^{17} * 2^2 = 2^{19} byte totali FAT

2^{19} / 2^9 = 2^{10} **numero blocchi che la FAT occupa.**

(2^26 / 2^9 = 2^17 entry; 2^17 * 4 byte = 2^19 byte; 2^19 / 2^9 = 2^10 = 1024 blocchi)

23) FFS, 512 byte blocco, 3 diretti 1 indiretto. Capacità max

3 blocchi diretti mi danno già 1536 byte .
Un blocco di puntatori ha esattamente 512/4 = 2^{9-2} = 2^7 = 128 puntatori
3*512 + 128 * 512 = 3*512 + 2^7 * 2^9 = 3*512 + 2^{16} = 512(131) = 512 +2^{17} = 128.5 kb

24) FAT: Calcolo generale

Data una capacità di disco 2^C e una dimensione di blocco 2^k con un numero di bit di indirizzo divisibile per 8 [ADDRESS] allora per capire quanta è grande la FAT devo:

1. Capire quanti blocchi ho per coprire tutta la capacità , NBlocchiTot = 2^C / 2^k
2. NBlocchiTot == EntriesInFAT , NByteFat = EntriesInFat * ByteIndirizzo
3. NBlocchiPerAllocareFAT = NByteFAT / 2^k
4. Dato un numero di bit divisibile per 8 allora CapacitàMassimaFAT = ADDRESS* 2^k

1) Calcolo I-nodes FFS:
Data una lunghezza blocco 2^k , un numero di byte di indirizzo B e un numero di nodi diretti D0, indiretti primo lvl D1 , indiretti sec. Liv. D2 e D3 allora:

1. NumBlocchi = 2^(B*8)
2. CapacitàMax = NumBlocchi * 2^k
3. DimensioneFileMaxInBlocchi=
    a. D0 * + D1(2^k / B) + D2(2^k*2^k/B) + D3(2^k*2^k*2^k/B)