



Politecnico di Torino

Corso di Laurea Ingegneria Informatica **LM-32 (DM270)**
A.a. 2022/2023

Programmazione di sistema

Appunti presi dal corso del Professor Cabodi

Pierpaolo Bene (s319841)

Per info o segnalazioni: peppobene@gmail.com

Indice

Chapter 9: Main Memory	4
ADDRESS BINDING	5
ALLOCATION	6
Hierarchical Paging.....	9
Swapping	11
Chapter 10: Main Memory	13
VIRTUAL MEMORY	13
DEMANDING PAGING.....	14
FREE-FRAME LIST	15
DEMANDING PAGE OPTIMIZATION.....	16
COPY-ON-WRITE	16
BASIC PAGE REPLACEMENT	17
FIRST IN FIRST OUT (FIFO) ALGORITHM	18
OPTIMAL ALGORITHM.....	18
LEAST RECENTLY USED (LRU) ALGORITHM	18
PAGE BUFFERING ALGORITHMS	19
APPLICATIONS AND PAGE REPLACEMENT	20
ALLOCATION OF FRAMES.....	20
FIXED ALLOCATION.....	20
RECLAIMING PAGES.....	21
NON UNIFORM MEMORY ACCESS	21
THRASHING.....	21
WORKING SET MODEL.....	22
ALLOCATING KERNEL MEMORY	23
BUDDY SYSTEM.....	23
SLAB ALLOCATOR	23
PREPAGING	24
PAGE SIZE.....	24
PROGRAM STRUCTURE	24
Chapter 11: Mass – Storage Systems	25
DISK STRUCTURE.....	25
DISK STRUCTURE.....	26
Chapter 13: File System Interface.....	27
FILE CONCEPT.....	27
OPEN FILE	27

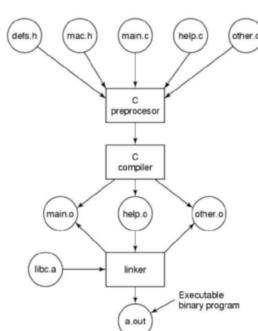
FILE STRUCTURE	27
ACCESS METHODS	28
DIRECTORY STRUCTURE	28
DISK STRUCTURE	28
DIRECTORY ORGANIZATION	28
SINGLE-LEVEL DIRECTORY	29
TWO-LEVEL DIRECTORY	29
MOUNTING FILE SYSTEM	29
<i>Chapter 14: File System Implementation</i>	30
FILE SYSTEM STRUCTURE	30
FILE SYSTEM LAYERS	30
FILE SYSTEM OPERATIONS	31
DIRECTORY IMPLEMENTATION	32
ALLOCAZIONE DEI FILE	32
PERFORMANCE	35
FREE-SPACE MANAGEMENT	35
PERFORMANCE	36
PAGE CACHE	36
RECOVERY	36
<i>Chapter 12: I/O Systems</i>	37
I/O Hardware	37
POLLING	38
INTERRUPT	38
APPLICATION I/O INTERFACE	39
BLOCK AND CHARACTER DEVICES	39
NETWORK DEVICES	39
CLOCKS AND TIMERS	39
NON BLOCKING AND ASYNCHRONOUS I/O	40
VECTORED I/O	40
KERNEL I/O SUBSYSTEM	40
Transforming I/O requests to Hardware Operations	41

Chapter 9: Main Memory

OBIETTIVO: analizzare il ruolo del sistema operativo nel fornire ai processi utente una porzione di memoria logica che viene mappata sulla memoria fisica.

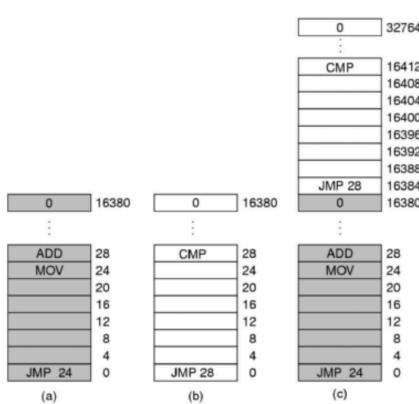
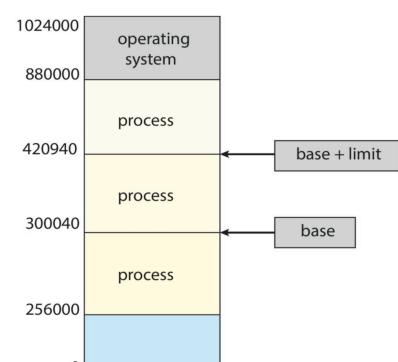
BACKGROUND: il sistema di elaborazione può avere più programmi attivi in contemporanea, perciò, ci troviamo in un contesto di **Multiprogrammazione**. Al doppio click dell'utente, l'eseguibile viene mandato alla memoria RAM, che insieme ai registri rappresenta l'unica memoria alla quale la CPU accede direttamente. Mentre l'accesso ai registri richiede un solo ciclo di clock, l'accesso in RAM ne richiede molteplici, esiste perciò un meccanismo noto come **Cache**, posto tra i registri e la memoria principale con l'obiettivo di velocizzare l'accesso a quest'ultima. Il sistema operativo deve garantire inoltre una qualche forma di protezione tale per cui un processo acceda a dei dati a cui non dovrebbe accedere, sia per errore che intenzionalmente, nel caso di processi malevoli.

The Model of Run Time



La figura a sinistra illustra il meccanismo di creazione di un programma eseguibile, a partire dai '.h' e '.c'. E' importante notare che a partire dai file sorgenti '.c' e dagli header '.h', è compito del compilatore quello di generare per ogni sorgente il file oggetto, che poi il linker unirà in un unico eseguibile 'a.out' a cui sono state anche aggiunte le funzioni di libreria 'libc.a'. Un eseguibile così generato verrà utilizzato per avviare un processo. In un sistema multiprogrammazione, ad un certo punto, saranno presenti in RAM, sia vari processi, sia il Kernel.

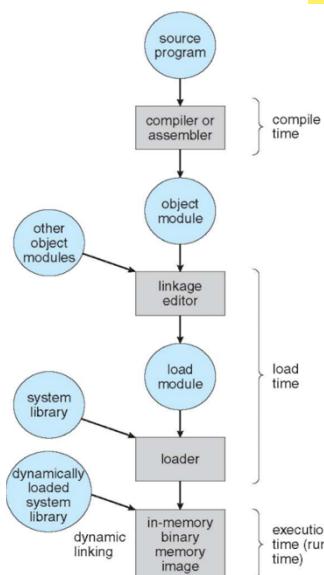
In questa situazione è fondamentale che il processo non esca dai "bordi" della memoria che gli viene assegnata, per fare ciò è fondamentale che nella CPU ci siano due indirizzi, quello di **base**, cioè l'indirizzo da cui parte il processo ed registro **limite**, che non è dove il processo arriva, ma è quanto bisogna aggiungere al base register per arrivare al confine della memoria ad esso assegnata. Nell'illustrazione sottostante si può



vedere come due processi a e b, credono di lavorare a partire dall'indirizzo 0. Effettivamente è così per il processo a ma non per b che parte da 16384. Per risolvere il problema noto come **Relocation**, il base register viene sommato all'instruction register tramite un sommatore e poi passato al program counter, in modo che programma virtualmente crede di essere all'indirizzo zero, ma viene rilocato automaticamente via Hardware.

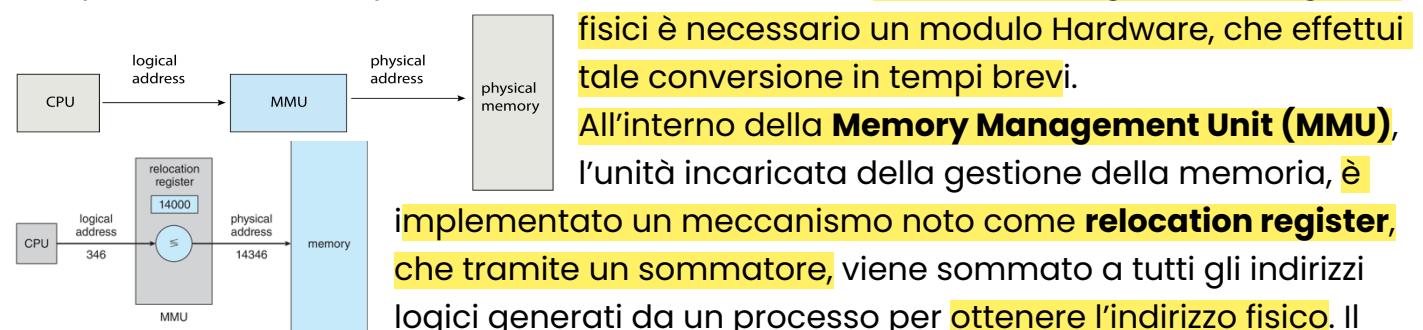
ADDRESS BINDING

Un grande problema nei sistemi a multiprogrammazione è il seguente: quando un eseguibile viene portato da disco in memoria, non ha un meccanismo di gestione automatizzata degli indirizzi, non conosce cioè a priori a quale indirizzo funzionerà. Gli indirizzi all'interno del codice sorgente sono perciò **simbolici**, il compilatore traduce il programma in codice assembler scrivendo al posto di tali simboli degli indirizzi rilocabili, ad esempio "14 bytes dall'inizio di questo modulo", che va completato poi dal linker o dal loader. Ci sono tre possibili fasi in cui il Binding avviene:



- **Compile time**: cioè la fase di generazione dell'eseguibile. Bisogna perciò conoscere a priori a quale indirizzo il processo deve andare in memoria. Tale meccanismo va bene per sistemi relativamente semplici.
- **Load Time**: cioè la fase iniziale dell'esecuzione, ovvero il caricamento dell'eseguibile in memoria. Fare il Binding in fase di load richiede che il codice eseguibile sia rilocabile.
- **Execution Time**: in questo caso il Binding è differito fino all'esecuzione in memoria di un determinato modulo o istruzione. Supponiamo che un programma chiama la funzione a e la funzione b, allora si farà il Binding degli indirizzi di tali funzioni solo quando vengono chiamate.

Il Binding in fase di load prevede che sia il linker che il loader collaborino per convertire gli indirizzi. La risoluzione del Binding in fase di esecuzione viene risolto tramite l'utilizzo di **indirizzi logici**, cioè ciò che vede il processore, e **indirizzi fisici**, cioè quello usato dalla CPU per accedere alla porzione desiderata della RAM. Per convertire gli indirizzi logici in



fisici è necessario un modulo Hardware, che effettui tale conversione in tempi brevi.

All'interno della **Memory Management Unit (MMU)**,

l'unità incaricata della gestione della memoria, è

implementato un meccanismo noto come **relocation register**, che tramite un sommatore, viene sommato a tutti gli indirizzi logici generati da un processo per ottenere l'indirizzo fisico. Il

Binding può essere utilizzato per realizzare quindi vari schemi di caricamento ed esecuzione del programma "dinamici", poiché libera il sistema dalla necessità di dover risolvere gli indirizzi durante la scrittura del programma.

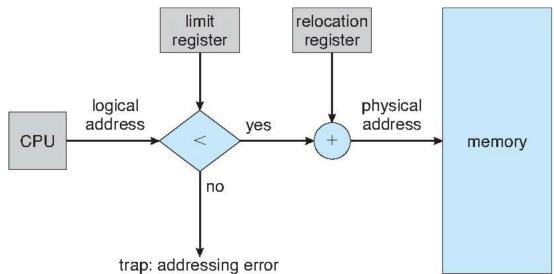
Dynamic Loading: un programma può essere caricato in memoria scomposto in pezzi più piccoli. Una funzione, ad esempio, non viene caricata fin quando non viene chiamata. Ciò ottimizza l'utilizzo della memoria, senza richiedere un supporto speciale dall'OS, anche se possono essere presenti librerie specifiche per il programmatore.

Dynamic Linking: il linking è la fase in cui il linker aggiunge al programma le funzioni dalle librerie, decidendone gli indirizzi. Farlo dinamicamente significa posporre questa operazione alla fase di esecuzione del programma. Per fare cioè al posto della funzione effettiva viene assegnato uno stub (funzione fittizia), aggiornato ad execution time.

ALLOCATION

Fino ad ora abbiamo dato per scontato che ai processi venga assegnata una fetta contigua di memoria, in realtà questo è solo lo schema più semplice ma in realtà non è sempre utilizzato perché come vedremo porta con sé dei problemi.

Contiguous Allocation: è il modo più semplice per far condividere la Ram a più processi, ad ognuno dei quali viene assegnato un intervallo di indirizzi che viene

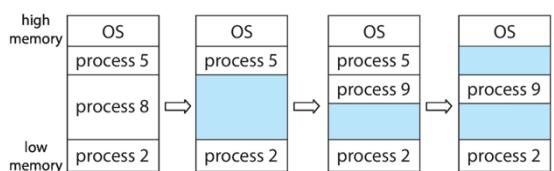


chiamato **partizione**. L'indirizzo logico viene convertito in fisico tramite la somma al **Relocation register**, come già visto. Viene inoltre eseguita un'operazione di controllo, verificando che l'indirizzo sia minore del **limit register**, in caso contrario si invoca una trap per gestire l'errore.

Il numero di processi collocabili in memoria dipende dal numero di partizioni allocabili, che sono di dimensioni variabili. Quando un processo termina lascia uno "spazio vuoto" nel punto in cui era stato allocato, bisogna perciò definire delle politiche che permettano di decidere a priori dove il processo viene allocato. Tre possibili soluzioni:

- **First – fit:** Alloco il processo nel primo buco abbastanza grande per ospitarlo. La premessa è che nel far partire un processo devo sapere quanto occuperà. Tale politica è ottimale in termini di velocità nell'allocare un processo;
- **Best – fit:** Alloco il processo nel buco più piccolo possibile, abbastanza grande da contenerlo, con l'obiettivo di lasciare meno "spazio vuoto" possibile;
- **Worst – fit:** Alloco il processo nel buco più grande possibile. Il ragionamento è che lascio abbastanza spazio residuo per far sì che ci entrino altri processi.

È chiaro che in base alle necessità del sistema verrà scelta una di queste politiche.

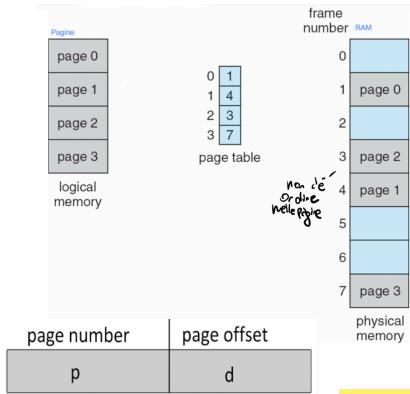


Un altro problema da affrontare nella fase di **allocazione** è la **Frammentazione**, cioè l'alternanza di "pieni e vuoti" in memoria dopo aver allocato un certo numero di processi.

Parliamo di **Frammentazione esterna** quando queste alternanze sono al di fuori dei processi, come nell'ultima parte della figura. Parliamo invece di **Frammentazione interna** se la memoria allocata ad un processo è più grande del processo è più grande del necessario, lasciando una parte di memoria inutilizzata. Statisticamente a causa della frammentazione si perde in un sistema 1/3 degli N blocchi allocati. Quando una memoria è troppo frammentata si attuano politiche di **deframmentazione**, cioè spostare i processi in modo da minimizzare il numero di "buchi", ciò è possibile anche dinamicamente, cioè mentre il processo è in stato di esecuzione cioè in stato 'ready'. Ciò non è privo di rischi, spostare un processo mentre è in esecuzione potrebbe generare il cosiddetto **problema di I/O**: un processo potrebbe essere in fase di "wait", è cioè sorgente o destinazione di un'operazione con delle periferiche. Due soluzioni:

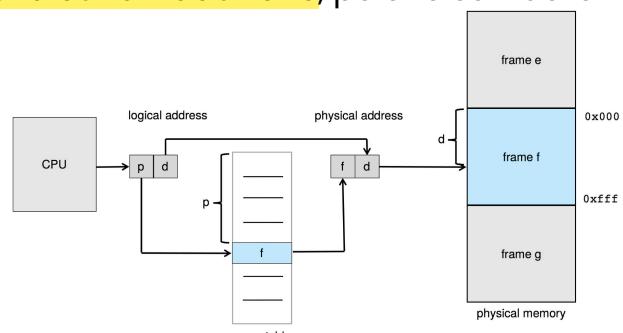
- Vietare la rilocazione in memoria di un processo che è in "wait" di un I/O;
- Creare una copia dei dati del processo necessari alle I/O da salvare su un 'buffer' di kernel, che farà da intermediario.

Torniamo a parlare di frammentazione. La vera soluzione che si usa per ridurla è il **Paging** (paginazione). L'idea è quella di non associare uno spazio variabile ai processi ma di spezzettarli in 'pagine', di dimensione fissa. In questo modo la frammentazione esterna viene risolta completamente. La memoria fisica viene divisa in blocchi di dimensione fissa (una potenza di due) detti **frames**, la memoria fisica diventa quindi un 'vettore di frames', così come la memoria logica diventa un 'vettore di pagine' contigue, ognuna delle quali è allocata in un frame della memoria fisica.



Per allocare un programma composto da N pagine, occorre trovare N frames liberi in memoria. Il problema, arrivati a questo punto, è che gli indirizzi logici saranno contigui, mentre quelli fisici no. Per ovviare questa problematica si crea una tabella, chiamata **page table**, vediamo come funziona nello specifico: l'indirizzo logico viene diviso in due parti, **page number**, che di fatto è l'indice della pagina e l'**offset** (displacement), cioè quanto devo scorrere nella pagina per

trovare il dato cercato. Avere potenze di due come dimensione di frame, consente di dividere l'indirizzo efficientemente. Ad esempio, ho un indirizzo m bit dove m=32, di cui n=10, sono dedicati all'offset ed m-n = 22 bit, sono dedicati al page number. I frame e le pagine hanno la stessa dimensione, quindi se l'indirizzo è ad esempio pagina 7, offset 11, l'offset necessita di alcuna traduzione, poiché se il dato cercato si trova in una certa posizione nella pagina, sarà nella stessa posizione nel frame. Perciò, come mostrato nello schema, la CPU emette l'indirizzo logico che viene diviso in p e d, l'offset va direttamente alla memoria fisica, mentre p serve per trovare il numero del frame all'interno della page table. Abbiamo quindi risolto il problema della frammentazione esterna, resta quella interna, come la calcoliamo?



Esempio

Process size = 72.766 bytes. Page size = 2048 bytes. Quanti bytes di frammentazione interna?

$$\text{Numero di pagine necessarie: } \frac{\text{Process size}}{\text{Page size}} = \frac{72.766}{2048} = 35.53$$

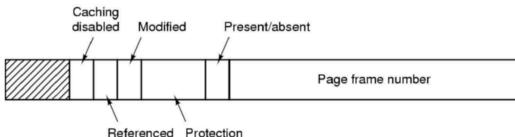
Assegno 36 pagine, approssimando per eccesso all'intero più vicino. Ora posso calcolare quanto dell'ultima pagina viene usato e quanti bytes rimangono inutilizzati:

$$\text{Numero di bytes utilizzati nell'ultima pagina: } 72.766 - 35 \times 2048 = 1086 \text{ bytes}$$

$$\text{Numero di bytes inutilizzati nell'ultima pagina: } 2048 - 1086 = 962 \text{ bytes}$$

La frammentazione interna ammonta perciò a 962 bytes.

Statisticamente la frammentazione interna è grande, in media, mezzo frame. E' meglio quindi avere frame piccoli o grandi? Dal punto di vista della frammentazione la situazione migliora, ma avere più grame significa avere una page table più grande. In generale nei sistemi reali le pagine vanno dagli 8 KB ai 4 MB.



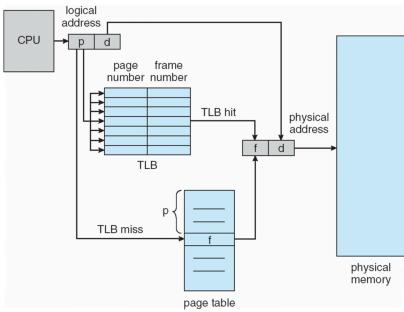
In realtà una riga della page table non contiene solamente il frame number, ma anche se tale frame è stato modificato o se è abilitato il caching.

La page table viene salvata in main memory, quindi nella RAM, essendo troppo grande per essere salvata nei registri della CPU. Per ottimizzarne l'utilizzo vengono tuttavia salvati due registri: il **Page-table Page Register (PTBR)**, che contiene l'indirizzo della RAM in cui la page table inizia, ed il **Page-table Length Register (PTLR)**, che indica la dimensione della page-table. Ciò significa che se per tradurre da logico a fisico con l'allocazione contigua bastava un passaggio interno alla CPU, quindi senza 'sprecare' colpi di clock, con la page table è necessario fare una lettura in RAM. Per ogni accesso in memoria devo fare quindi due accessi in memoria, uno per leggere l'indirizzo del frame dalla page table ed uno per leggere effettivamente il frame, ciò comporta un alto costo in termini di tempo. Per ottimizzare tale processo si fa uso di un meccanismo chiamato **Translation Lookaside Buffer (TLB)**, cioè una tabella, di dimensioni inferiori della page table, salvata nella CPU. Tipicamente ha dalle 64 alle 1024 righe, ogni riga

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

ha un'associazione pagina logica-frame fisico, ad esempio pagina 140 → frame 31, a differenza della page table che ha il frame 31 scritto all'indice 140 della page table. La TLB è quindi molto più compatta, ma prima di poter scrivere l'associazione 'pagina 140 frame 31' occorrerebbe una scansione lineare della

page table, viene perciò utilizzata una memoria associativa, che sfruttando dei comparatori hardware, in grado di completare tale operazione a costo zero.



Vado quindi a vedere se ho un dato nella TLB, se c'è allora ho un TLB hit, altrimenti ho una TLB miss. In quest'ultimo caso vado a copiare il dato dalla RAM alla TLB e riparto andando a cercare il dato nella TLB.

Occorre quindi poter stimare la probabilità di avere una hit. Supponiamo che un accesso in memoria impieghi 10 ns se ho il dato nella TLB, alternativamente impiego 20 ns perché dovrò fare due accessi in memoria. Supponiamo inoltre che nell' 80% dei casi ho una hit e nel 20% una miss. Il tempo effettivo di accesso in memoria, detto **Effective Access Time (EAT)** è: $0,8 \times 10 + 0,2 \times 20 = 12$ nanosecondi.

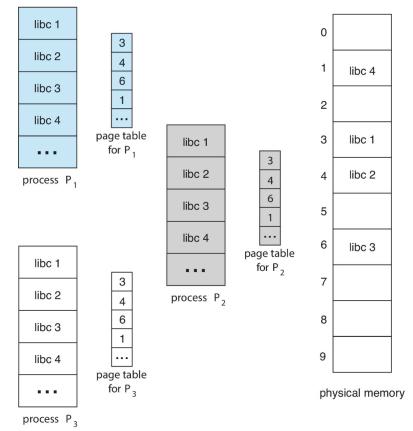
Un importante vantaggio della page table è quello della **memory protection**. Infatti la riga della page table non contiene solo il numero del frame, ma anche un bit che indica se la pagina è valida o invalida e prende il nome di "**bit di validità**".

Un ulteriore vantaggio è la possibilità di condividere delle pagine tra processi diversi, a patto che siano in sola lettura, occupando meno RAM. All'atto pratico ogni processo può vedere delle pagine come se fossero sue, ma in realtà sono mappate tutte sullo stesso frame. Tali pagine si dicono **reentrant** (rientranti), non hanno cioè uno stato che ricorda la precedente esecuzione, in pratica non usano variabili globali.

Ovviamente è importante che la condivisione sia limitata al codice rientrante e non sia estesa a tutti i dati di un processo.

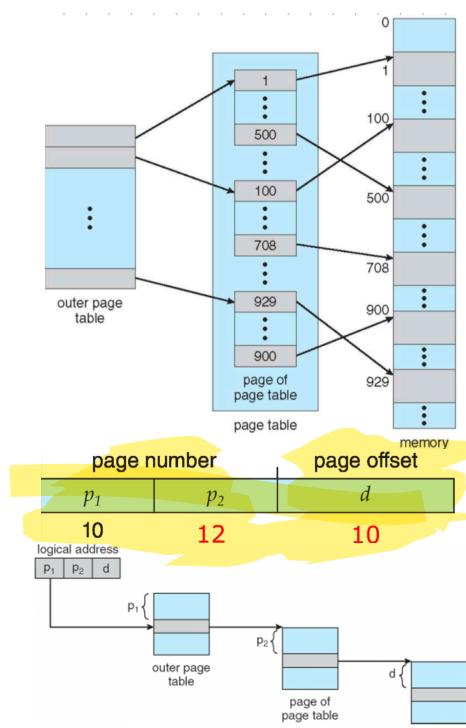
Nell'esempio a destra si vedono 3 processi che usano una libreria, rimappata per tutti i processi sullo stesso frame.

Rispetto alla versione di pagine copiate per ogni processo si passa da 12 frame usati a 4.



Ora vediamo nello specifico com'è fatta la tabella delle pagine. Prendiamo ad esempio uno spazio di indirizzamento di 32 bit, la dimensione delle pagine è di 4KB (2¹² Byte) quindi nell'indirizzo i 12 bit bassi rappresentano l'offset e i 20 bit alti il numero di pagina. Se abbiamo a disposizione una RAM dell'ordine dei Gbytes, riusciremmo a rappresentare il numero di frame con un numero di bit compreso fra 16 e 32, ma per motivi di uniformità se 16 bit non sono sufficienti, se ne utilizzano 32. Quindi 4 byte per riga, ottenendo una page table di 4 MBytes. In memoria la tabella deve essere contigua. Ciò è ancora fattibile se dobbiamo allocare solo 4 MB di page table. Ma considerando che n processi devono avere la propria page table, allora avere un'allocazione contigua in RAM può diventare complicato. Per ovviare a questo problema vedremo 3 possibili soluzioni: Paging gerarchico, page table con Hashing e inverted page table.

Hierarchical Paging



L'idea è se mi servono 4MB la tabella gerarchica richiede 4 MB partizionati anziché contigui. Quella che prima era una tabella delle pagine formata da un vettore contiguo, viene spezzato in sezioni e si pone davanti ad essa un ulteriore tabella di "primo livello" che serve solo ad individuare in quale tabella di "secondo livello" si va a cercare l'effettivo numero di frame. L'indirizzo logico di 32-bit viene quindi diviso non più in 2 parti, ma in 3. Nell'esempio a sinistra è rappresentato un indirizzo a 32 bit, diviso in 10 bit di offset (d) e 22 bit di indice di pagina. Una page table di 2²² righe avrebbe 4Mega Righe. Il page number viene quindi scomposto in p₁ di 10 bit e p₂ di 12 bit. La traduzione da logico a fisico viene fatta usando p₁ per selezionare una riga nella page table più esterna, contenente l'indirizzo di una page table di 2 livello. Si usa poi p₂ per individuare la

riga della seconda page table che contiene il numero di frame. Dentro al frame si usa poi il displacement per trovare l'indirizzo fisico.

Ciò che abbiamo visto può essere applicato anche su indirizzi a 64 bit.

Supponiamo di avere pagine da 4 KB (2^{12}). La page table ha quindi 54 righe, perché come nell'esempio precedente si prendono i 12 bit bassi come offset, i 52 alti come page number. Dividendo la tabella in due avrei comunque una tabella esterna ancora da 2^{42} righe e una interna di 2^{10} righe. Conviene quindi fare un ulteriore separazione generando una page table gerarchica a 3 livelli. In questo

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

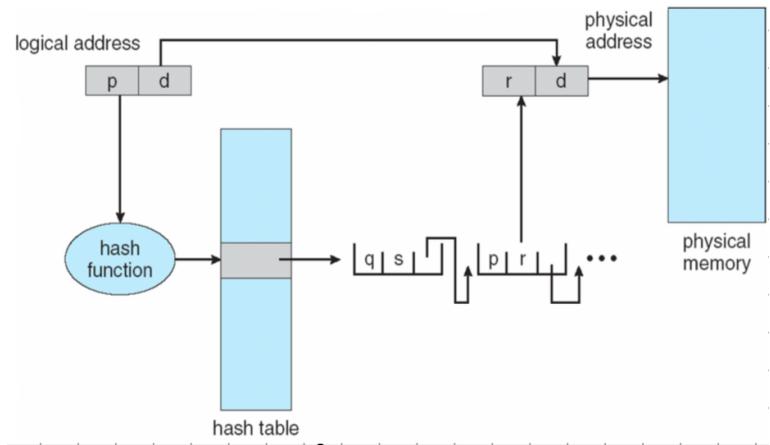
modo però devo tenere conto che per accedere ad un dato dovrà fare accesso prima alla 2nd outer page, poi alla outer page,

poi ancora alla inner page ed infine al frame. Per ogni dato da prelevare dalla memoria avrà quindi 4 accessi, è importante perciò poter fare affidamento sulla TLB, poiché un eventuale TLB miss sarebbe molto costosa.

Hashed Page Table

Un'alternativa alle tabelle delle pagine gerarchiche soprattutto per quanto riguarda il problema del tempo di accesso, è quello di usare l'**Hashing**, il quale offre tempi minori di accesso. La tabella di Hash può essere di due tipi:

- La prima, in cui ad ogni entry corrisponde un frame
- La seconda, in cui ad ogni entry corrisponde un "cluster"



Il vettore inoltre è generalmente più compatto, poiché con le liste di collisione si possono mettere più entries nella stessa riga.

Inverted Page Table

L'inverted page table, inverte la logica usata fino ad ora. Rappresenta una

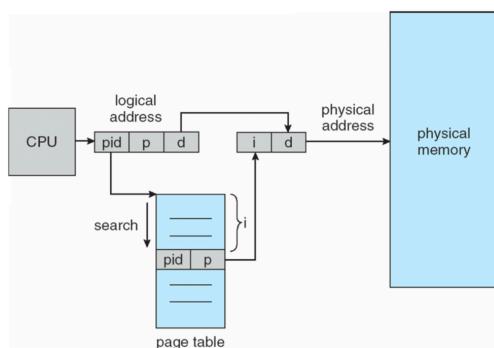


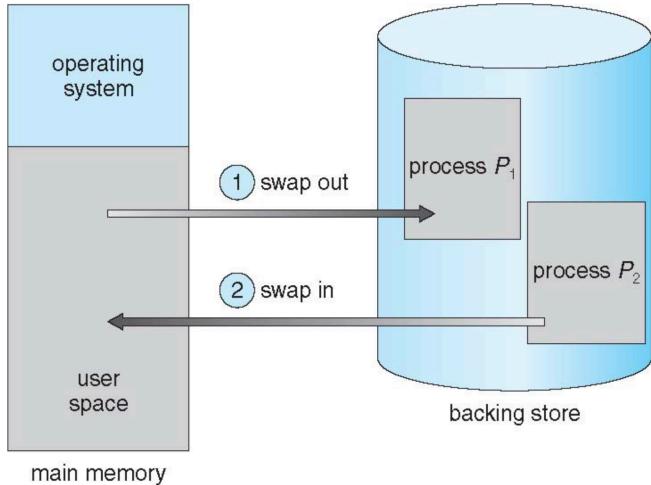
tabella in cui invece di usare il numero di pagina come indice ed il frame come contenuto, usa il numero di pagina come contenuto ed il frame come indice. Supponendo di avere la pagina 100 associata al frame 23, anziché usare l'indice 100 per trovare il numero di frame ($p[100] = 23$), cerco il valore 100 linearmente nella page table e

per avere il numero di frame vedo qual è l'indice della riga contenente il valore 100 ($100 = p[23]$). Ciò comporta che la dimensione della page table è proporzionale alla dimensione della Ram, ma ho lo svantaggio del dover

eseguire una ricerca lineare. Per risolvere questo problema posso utilizzare una tabella di hash, le cui entries sono poste in liste di adiacenza.

Swapping

Supponiamo di avere troppi processi in Ram e vogliamo avviare uno nuovo. È

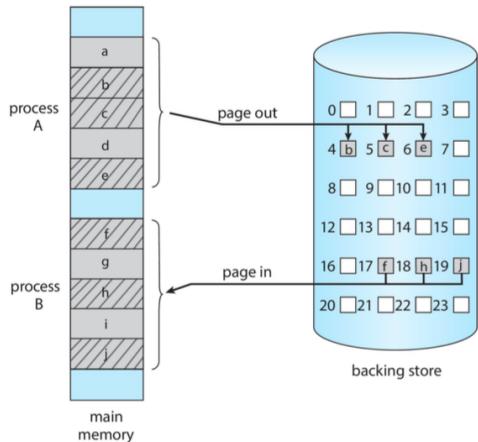


necessario avere una politica che ci indichi dove collocarlo e soprattutto chi buttare fuori. Fra le alternative abbiamo la politica di **swapping**, ovvero fare lo "**swap out**" per il processo che esce e lo "**swap in**" per il processo che entra. Dove va però a finire il processo uscente? I suoi frame verranno spostati in una porzione del disco adibita a questo lavoro chiamata "**Backing store**". Una volta che devo riportare i frame del processo nella Ram, li

rimetto esattamente dov'erano prima o no? Dipende da come viene realizzato il load e il link che abbiamo visto in precedenza, cioè se gli indirizzo sono rilocabili. In generale i sistemi operativi implementano varianti di queste tecniche. Lo swapping è una situazione estrema, implica che la memoria sia troppo carica. Ma quanto costa fare swapping?

Un "**context switch**", cioè cambiare processo in esecuzione, ha un costo non trascurabile, in quanto si tratta di processi che hanno bisogno di salvare i registri e poi verificare che i registri ripristinati siano validi per l'esecuzione. Aciò vanno aggiunti i tempi necessari per trasferire i frames da disco a Ram a viceversa. Pensando ad un processo di 100 MB ed un disco con transfer rate di 50 MB/sec, impiegerò due secondi di swap out e due di swap in. Un altro problema nel context switching può venir fuori con le I/O, simile a quanto già visto con la deframmentazione. Ovvero potrei spostare un processo che è in fase di wait di un I/O, le **due soluzioni** sono: una drastica, ovvero non consentire lo swap out di quel processo, oppure fare in modo che un processo che è in stato di wait non abbia mai dati di memoria user, coinvolti nell'I/O, ma ciò richiede un **double buffering**, cosa significa? Significa che copio i dati del processo (ad esempio per fare un output verso un dispositivo) prima dalla memoria user alla memoria kernel, che è un processo abbastanza veloce da fare rispetto al trasferimento verso una memoria esterna, e finita questa fase posso fare swap out del processo, dopodiché copio i dati dalla memoria kernel all'I/O, quando l'I/O è pronto. Come già accennato lo swapping oggi si usa poco, poiché le Ram sono abbastanza capienti, tranne nei sistemi mobile.

Un meccanismo alternativo di swapping è lo **swapping with paging**, sviluppato tenendo conto del fatto che un processo potrebbe salvare sul backing store,



non tutte, bensì alcune delle sue pagine, tendenzialmente quelle che non servono per far sì che il processo prosegua le sue operazioni. Ad esempio il processo A in figura lavora sulle pagine 'a' e 'd', quindi 'b','c' e 'd' vengono spostate sul backing store. Il processo b al contrario ha bisogno di 'f','h' e 'j' che quindi vengono swappate in dal backing store.

Concludiamo il discorso su CH9 vedendo alcuni esempi (non ci sono all'esame ☺)
L'intel IA-32 bit, supporta un concetto noto come segmentazione, che non approfondiamo. IL segmento è come una pagina ma di dimensione variabile, può arrivare fino a 4GB e il processo ha una tabella di segmenti che remanda a delle pagine che rimandano al frame. L'indirizzo logico ha un selettore ed un offset per generare un indirizzo lineare, che rintraccia poi la entry di una page table a 1 o 2 livelli.

L'architettura ARM sfrutta invece un microcontrollore (cioè un microprocessore specializzato in I/O), oggi spesso a 32 bit. Sfrutta 0 pagine da 4KB a 16KB o "sections", cioè pagine da 1 da 16 MB e ci sono due livelli di TLB per garantire pochi accessi alle pagine.

Chapter 10: Main Memory

OBIETTIVO: analizzare la memoria virtuale. Definiremo la paginazione a richiesta, utile per di definire uno spazio di indirizzamento virtuale realizzato in modo dinamico.

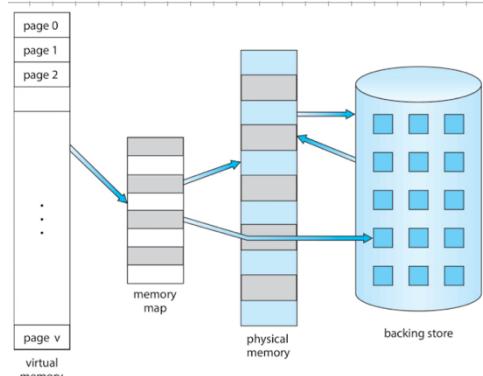
BACKGROUND: abbiamo già visto che un programma per essere eseguito non ha bisogno di tutte le sue parti. Una parte o non è mai usata durante l'esecuzione oppure potrebbe esserlo ma in tempi diversi rispetto ad altre. Quindi, potrebbe essere conveniente switchare le parti del programma. Serve quindi un supporto per far eseguire un programma solo parzialmente caricato in memoria così da avere più processi eseguibili in memoria e velocizzare anche l'esecuzione, avendo meno pagine da caricare.

VIRTUAL MEMORY

Chiamiamo **memoria virtuale** la netta separazione tra lo spazio di memoria di indirizzi logici e quelli fisici, che terrà sotto nascosto che una parte degli indirizzi logici sono mappati ad indirizzi fisici ed una parte no. Quindi virtualmente c'è tutto lo spazio logico ma realmente no. Come conseguenza la porzione di memoria che vede un processo può arrivare ad essere più grande della RAM stessa, consentendo di condividere alcune parti della memoria tra processi, di farne girare di più e di fare swap in maniera più efficiente.

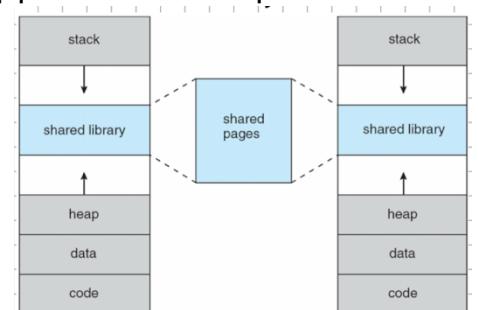
Definiamo **virtual address space** l'insieme degli indirizzi logici che vede un processo. Tale spazio solitamente parte dall'indirizzo 0 ed è contiguo. Mentre gli indirizzi fisici corrispondono a frame, non necessariamente tutte le pagine corrispondono ad un frame.

La traduzione logico-fisica è fatta dalla MMU. Tale meccanismo è implementato con la



"paginazione a richiesta". In questa figura a sinistra c'è lo spazio di indirizzamento virtuale, raffigurato più grande della Ram (physical memory) . La **memory map** è l'insieme delle tabelle delle pagine che traduce da logico a fisico. Alcune pagine possono essere direttamente in memoria fisica, quindi in un frame (blocchi blu), altri sono nel **Backing store** (blocchi in grigio). Nella figura a destra è rappresentato lo spazio di

indirizzamento virtuale. Dati e codice vengono messi 'in fondo' essendo a dimensione fissa, lo stack parte invece dal primo indirizzo e l'heap dall'ultimo disponibile, essendo parti che in un processo tendono a crescere/decrescere dinamicamente. Posso anche mettere le librerie condivise nel mezzo.



DEMANDING PAGING

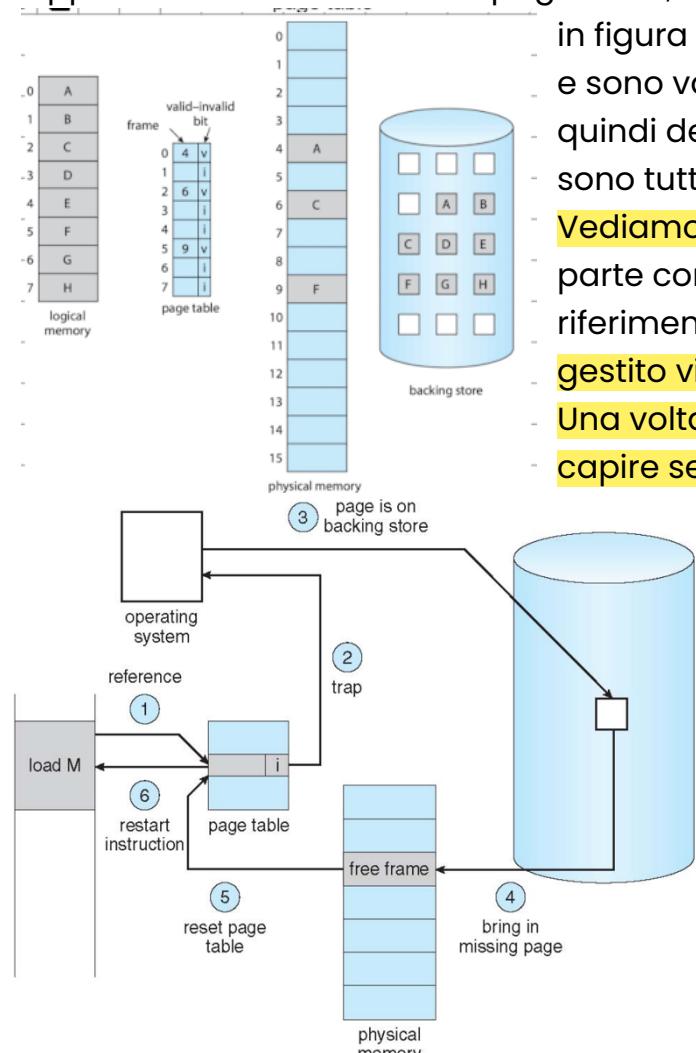
Il concetto di "paginazione a richiesta" consente di modellizzare l'idea di avere una pagina che entra in un frame, solo quando ce n'è effettivamente bisogno. I vantaggi li abbiamo già visti. Questo meccanismo è molto simile allo swapping with paging, ma si differenzia perché qui è centrale la singola pagina e non l'intero processo. Se una pagina è necessaria la "riferiamo", se questo riferimento è non valido significa che la pagina è ancora su disco, quindi la andiamo a copiare in memoria e la usiamo da lì. Avendo a disposizione questo meccanismo possiamo fare il "**lazy swapper**", cioè non porto in memoria una pagina di un processo finché quel processo non ne ha bisogno. Quando facciamo swapping, ci basiamo sulla capacità di indovinare in futuro quali saranno le pagine di cui avrò bisogno. Per farlo serve un hardware prestante, in particolare una MMU dedicata a tali operazioni. È possibile inoltre che un processo tenti di accedere ad una pagina che non è mappata su un suo frame ma è già in memoria, ad esempio può succedere per le pagine shared.

Per gestire questo meccanismo nelle page table e nelle TLB, di fianco al frame c'è un bit di validità, che indica se la pagina è in un frame, ed è quindi "**memory resident**" oppure no.

Frame #	valid-invalid bit
0	v
1	v
2	v
3	i
4	i
5	i
6	i
7	i
...	
9	i
10	
11	
12	
13	
14	
15	

page table

Supponiamo di accedere alla page table, ad esempio alle pagine 0,2 e 5. Nel processo



in figura queste pagine sono in frame, quindi in Ram e sono valide. Tutte le altre hanno "i" (invalid), ci sono quindi dei 'buchi'. Da notare anche che nel backing ci sono tutte le pagine, sia quelle in frame, che non.

Vediamo cosa significa gestire un **page fault**. Si parte con un riferimento a pagina. Il primo riferimento con bit di validità settato su invalid è gestito via hardware, la MMU scatenerà una trap. Una volta inviata la trap, il Kernel deve cercare di capire se il riferimento è sbagliato e quindi chiudere

il programma (es. puntatore Null) se invece è un puntatore valido a pagina che non è in Ram, fa questo: cerca un frame libero nella Ram, porta la pagina da disco a quel frame, mette apposto la page table e setta a v il bit di validità e fa ripartire l'istruzione di load. A cosa serve questo esempio? A capire che la page fault costa parecchio in termini di tempo. Come caso estremo di paginazione a richiesta parto da un processo che non ha alcuna pagina in memoria ed ogni nuova richiesta ad

una pagina genera un page fault, questo caso prende il nome di **pura paginazione a richiesta**.

Bisogna però anche considerare che una sola istruzione può richiedere più pagine. Ad esempio, una istruzione che somma due numeri presi dalla memoria e salva il risultato in memoria. Se scompongo tale operazione apparentemente banale, osservo che per esegirla devo prima leggere l'istruzione, le due variabili da leggere e una variabile da scrivere, ho quindi 4 accessi in memoria. Ciò per mettere in evidenza che potenzialmente un singolo page fault può portare più pagine in memoria. Va anche considerato che ci sono istruzioni che hanno indirizzi di partenza e arrivo sovrapposti o sono su più pagine. L'importante è tenere sempre a mente che il page fault può avere più pagine da gestire e per farlo serve un hardware prestante. Come posso gestirlo?

FREE-FRAME LIST

Trovare un frame libero, dopo un page fault, può essere complicato. Un meccanismo semplice che facilita tale operazione è la **free-frame list**, ovvero una lista di frame liberi. Ogni frame libero contiene un puntatore al frame libero successivo. Il Kernel ha il puntatore al primo frame libero. In alcuni casi il Kernel può mettere a disposizione dei processi dei frame liberi e "pre-azzerati". Inizialmente la lista contiene tutta la memoria. Alla luce di ciò rivediamo i passi necessari dopo un page fault: parte la trap, vengono salvati i registri del processo, viene capito che l'interrupt era dovuta al page fault, viene chiamato il gestore dei page fault che, dopo aver determinato che il page reference è legale, fa partire una lettura della pagina dal disco. Mentre viene copiata la pagina dal disco al frame trovato sulla frame-list, il kernel si mette in stato wait e da la CPU ad un altro processo. A questo punto riceve un "I/o completed", sempre tramite interrupt, dal disco, salva i registri dell'altro processo che era partito nell'attesa e si rende conto che l'interrupt ricevuto veniva dal disco. A questo punto sistema la page table e cerca di far ripartire il processo che aveva scatenato il page fault mettendolo in coda. Tutto ciò di nuovo mette in evidenza l'alto costo del page fault. Le tre attività principali sono quindi: l'interrupt, la lettura da disco ed il riavvio del processo. La lettura da disco è la parte più costosa in termini di tempo. Con **page fault rate** indichiamo la probabilità di page fault, compresa tra 0 e 1. Supponendo di fare un dato numero di accessi in memoria l'**effective access time (EAT)** è:

$$EAT = (1 - p) \times \text{memory access time} + p \times (\text{page fault overhead} + \text{swept page out} + \text{swept page in})$$

Esempio

Memory access time = 200 ns. Average page fault service time = 8 ms

$$EAT = (1 - p) \times 200\text{ns} + p \times (8\text{ ms}) = 200\text{ ns} + p \times 7.999.800\text{ ns} \rightarrow \text{varia al variare di } p$$

Ipotizzando un page fault ogni 1000 accessi: EAT = 8,2 μ s.

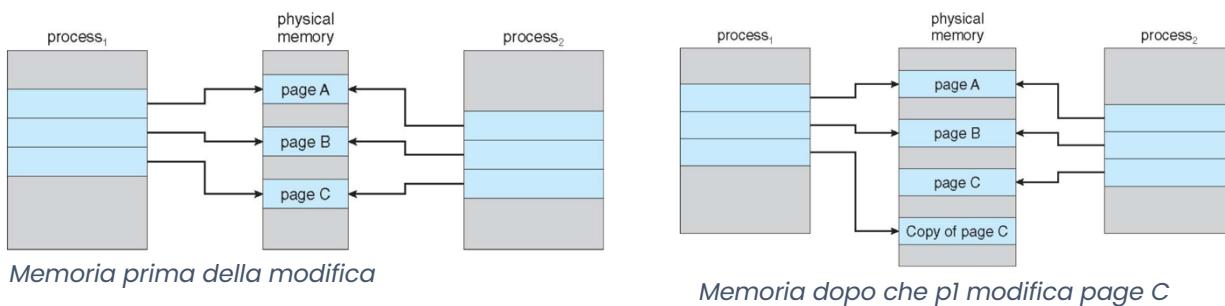
Ho quindi rallentato l'accesso di un fattore pari a 40. Il page fault incide altamente sul tempo effettivo di accesso in memoria.

DEMANDING PAGE OPTIMIZATION

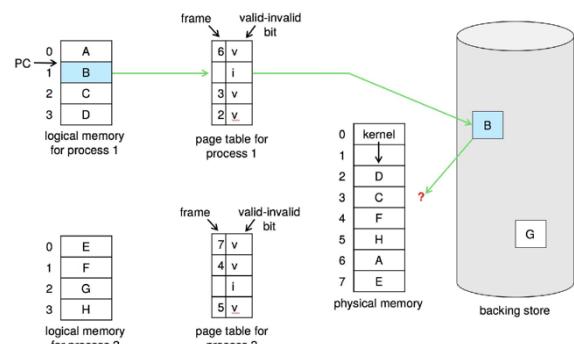
Per rendere più immediata la gestione del page-fault una possibilità è rendere più efficiente l'accesso al disco. L'altra possibilità rendere più veloce la partizione di swap e fare il demanding page fra quella partizione e Ram, anziché passare da disco a Ram. Ciò comporta che il tempo di avvio sarà più lento ma avrà meno problemi durante l'esecuzione. In Solaris si utilizza la strategia opposta: si fa paginazione sul disco ma quando si fa lo swap out le pagine si gettano via anziché salvarle nel disco, in caso vengono recuperate dall'eseguibile. Per fare ciò si usa l'anonymous memory, lo stack e heap partono puliti e vengono riempiti progressivamente, scrivo ad esempio nell'heap tramite una malloc molto grande ma poi ne uso solo una parte. Di fatto alcune di queste pagine non vengono mai usate e non c'è mai uno swap in da disco, perciò, non ci sarà neanche swap out. Ci sono inoltre pagine che non vengono modificate dal processo ma solo lette, quindi le butto senza swap out.

COPY-ON-WRITE

La **copy-on-write** è una strategia legata a processi che condividono le pagine. Se ho infatti un parent ed un child, appena dopo la fork, questi condivideranno le stesse pagine. Se uno dei due modifica una pagina questa viene copiata in memoria. In questo modo le pages vengono duplicate solo se il processo ci scrive sopra. In generale quando servono pagine libere vengono anche gestiti dai **pool** di pagine azzerate, da affiancare alla free-frame list.

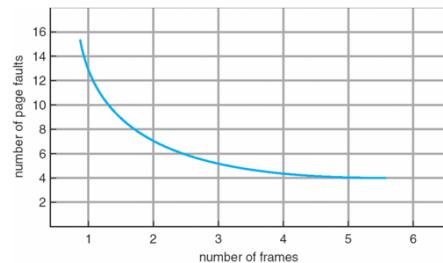


Cosa succede se non ci sono più frame liberi? Inevitabilmente dovremo far uscire una pagina per fare posto. Ma quale pagina scelgo? Per rispondere a questa domanda esistono degli algoritmi ben precisi, il cui unico obiettivo è evitare di cacciare fuori la pagina a cui un processo farà accesso subito dopo, sarebbe il caso peggiore. Tali algoritmi sono detti di **Page Replacement**. Per fare page replacement è fondamentale usare il **modify**(o **dirty**) **bit**, ovvero il bit associato ad ogni pagina che ci dice se il processo che ne ha fatto uso l'ha modificata oppure no, in caso negativo dopo averla cacciata dalla memoria è inutile farne una copia sul disco. Nell'esempio a destra il processo 1 richiede B, che non è presente in memoria e il sistema avendo Ram piena cerca di decidere chi buttare fuori. Scopriamo come può capirlo :O .



BASIC PAGE REPLACEMENT

Il problema si attiva: il page fault ha prelevato la pagina dal disco e non sa dove metterla, parte la ricerca del frame, vado a vedere sulla free-frame list ma è vuota. Il sistema deve trovare un **frame vittima**, da buttare fuori ed eventualmente, in caso di **dirty bit attivo, copiare sul disco**. Ovviamente è preferibile una pagina con dirty bit a 0. Copio a questo punto la pagina nel frame che ho liberato e continuo riavviando il processo. Fin qui niente di nuovo. Prima di vedere gli algoritmi promessi facciamo un'ulteriore osservazione. Quanti frame do ad un processo al momento dell'avvio? Se ne assegno tanti allora sarà più difficile, per tale processo, riempirli tutti e quindi sarà meno probabile dover fare page replacement. La Ram potrà però ospitare meno processi in totale. La scelta quindi dipende come al solito dal contesto in cui il mio sistema dovrà operare.



Vediamo come si può valutare l'efficacia di un algoritmo: utilizzeremo una **reference string** e 'computeremo' il numero di page fault.

Reference string usata: **7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 0 1 7 0 1**.

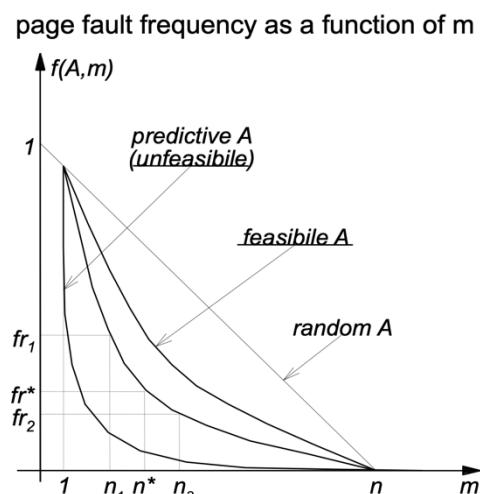
La **Page Fault Frequency** ci aiuterà a valutare tali algoritmi:

$$\text{Page Fault Frequency} = f(A, m) = \sum_w p(w) \frac{F(A, m, w)}{\text{len}(w)}$$

- A page replacement algorithm under evaluation
- w a given reference string
- $p(w)$ probability of reference string w (Se ho più stringhe di riferimento, ad ognuna assegno una probabilità, tale per cui la somma di tutte faccia 1. Questa probabilità dipende dal tipo di programma.)
- $\text{len}(w)$ length of reference string w
- m number of available page frames
- $F(A, m, w)$ number of page faults generated with the given reference string (w) using algorithm A on a system with m page frames.

Un altro fattore da considerare è il **reference bit**, simile al modify bit ma mi dice se un processo ha mai fatto accesso alla pagina, anche in sola lettura.

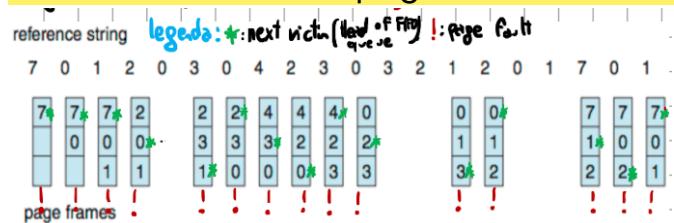
Nel grafico a sinistra si mettono sulle ascisse la page fault frequency. Il caso ideale (in alto) è l'algoritmo predittivo, ovvero in grado di prevedere il futuro e capire esattamente di quanti frame avrà bisogno il processo, senza scatenare page fault.



L'algoritmo centrale (feasible A) è un algoritmo furbo, in cui il numero di page fault contro numero di frame disponibili va diminuendo. Se avessimo un algoritmo che prende casualmente, ci potremmo aspettare che con m sufficientemente grande, non abbiamo page fault. Questo perché ho assegnato abbastanza frame al processo da far sì che non vada mai in page fault. Il numero di page fault è quindi strettamente collegato con il numero di frame assegnati al processo.

FIRST IN FIRST OUT (FIFO) ALGORITHM

La prima pagina entrata in un frame sarà anche la prima ad uscire. Non ottimale ma semplice da implementare. Suppongo di avere 3 Frames. I primi tre inserimenti sono necessariamente dei page fault, non avendo frame disponibili. Pagina 7 nel primo

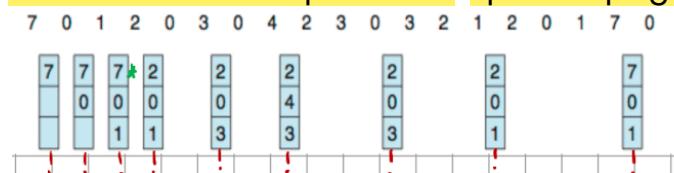


frame, 0 nel secondo, 1 nel terzo. Aggiungo la 2, ma tutti i frame sono pieni. L'head, ovvero l'ultima pagina aggiunta è la 7, quindi la caccio fuori ed inserisco il 2. Lo 0 sarà la nuova head della coda FIFO. Vado

avanti e la pagina 0 non mi da page fault essendo già in frame, poi devo inserire pagina 3 quindi tolgo lo 0 che era la head precedente e così via. In totale ho 15 page faults! La FIFO si basa sull'idea che se una pagina è in memoria da molto tempo, sarà probabile che non mi servirà più. Ovviamente il risultato varia anche molto in base alla stringa di riferimento. Alcune stringhe portano alla '**Anomalia di Belady**', cioè il numero di page fault aumenta aumentando il numero di frame disponibili. Una stringa che risponde a tale anomalia è la stringa : 1 2 3 4 1 2 5 1 2 3 4 5.

OPTIMAL ALGORITHM

Non realizzabile nella realtà, perché non so di quali pagine avrà bisogno. Questo algoritmo conosce già il futuro. Leggiamo cioè tutta la reference string prima di eseguire l'algoritmo e sceglio come vittima quella che per più tempo in futuro non mi servirà o non sarà più usata. I primi 3 page fault sono inevitabili of course. L'algoritmo

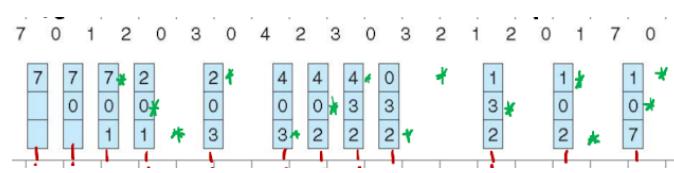


al momento di inserire il 2 vede che lo 0 sarà usato nella prossima iterazione, l' 1 tra 10 iterazioni e il 7 tra 14, quindi butto fuori il 7. Mentre con la fifo il 3 buttava fuori lo 0,

adesso il 3 butta fuori l'1 e così via. Siamo passati da 15 a 9 page fault. È ottimale nel senso che preso il numero di frame e la stringa di riferimento, non posso avere meno page fault con nessun'altro algoritmo. Si potrebbe dimostrare ma non lo faremo ☺.

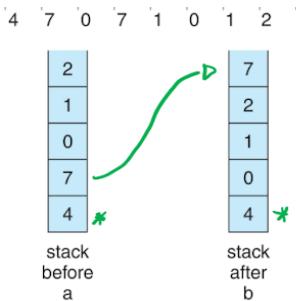
LEAST RECENTLY USED (LRU) ALGORITHM

Un modo per provare a riprodurre un optimal algorithm nella realtà è dire: è probabile che la pagina a cui farò accesso più in là nel tempo, sia quella a cui ho fatto un accesso più lontano nel passato. Tale idea si basa sul fatto che spesso si accede alla stessa pagina in accessi vicini. Si usa la storia passata nel tentativo di prevedere il futuro. Di nuovo i primi tre page fault sono inevitabili, nel momento di inserire il 2 vedo



che il 7 è quello dall'accesso meno recente e lo scelgo come vittima. Al momento di inserire il 3 vedo che l'accesso più lontano appartiene all'1 e lo

rimpiazza e così via. In questo caso ho 12 page fault. C'è da considerare che questo algoritmo ha un costo molto elevato, poiché ogni pagina ha un contatore in cui viene inserito l'ultimo accesso fatto, al page fault si fa una ricerca del minimo contatore.



- Si può pensare di implementare il meccanismo del contatore con una lista doppio linkata sullo stack. In figura prima dell'istante 'a', le pagine ordinate in base all'ultimo accesso sia recente è : 2, 1, 0, 7, 4. Cioè la più recente è 2, la meno recente è la 4, che è la vittima.
- Viene realizzata come lista doppio linkata perché consente di far slittare i valori a differenza di un vettore.

Gli algoritmi LRU sono ancora troppo costosi, si tende quindi ad utilizzare algoritmi detti **LRU approssimati**, che si accontentano di fare una scelta di una pagina che è stata utilizzata abbastanza meno recentemente rispetto a cercare per forza quella con contatore minimo. Per fare ciò si utilizza il reference bit, precedentemente accennato, implementato via hardware. Se siamo in grado di azzerare in certi istanti tutti i reference bit, allora esiste un modo per determinare se una pagina è abbastanza recente perché ha il reference bit a 1, a 0 se non lo è. Il **Second – chance Algorithm** è un algoritmo FIFO (che non è da buttare essendo $O(1)$ in certi casi), che fa un ulteriore check: se la testa della FIFO ha reference bit = 0 allora è la vittima, altrimenti passo al successivo fin quando non trovo una pagina con reference bit a 0. Di questo algoritmo esistono numerose varianti (che vedremo negli esercizi) tenendo conto non solo del reference bit, ma anche del modify bit. In sostanza cerco una pagina con entrambi i bit a 0, così da non doverla copiare sul disco, poi una con ref=0 e modify=1, ma dovrò scriverla su disco. Poi una ref = 1, mod = 0 e l'ultima ref=1, mod =1.

Ci sono inoltre algoritmi che guardano la frequenza di accessi in un certo intervallo, prendono il nome di **Counting Algorithm**. L'idea è che una pagina mi serve se nell'ultimo "tempo di riferimento" l'ho usata tante volte, non mi serve se l'ho usata poche volte. Ce ne sono di due tipi:

- **Least frequently used (LFU) Algorithm:** se l'ho usata poco mi servirà lontano nel futuro, quindi è la vittima;
- **Most frequently used (MFU) Algorithm:** se l'ho usata di più, mi servirà poco, quindi è la vittima.

Sono diametralmente opposti e quale conviene utilizzare dipende dal contesto di utilizzo.

Ora esaminiamo problematiche legate nell'esplorare tentativi di ottimizzare l'efficienza della paginazione, che non siano legate alla scelta della vittima.

PAGE BUFFERING ALGORITHMS

L'obiettivo degli algoritmi di **Page buffering** è quello di ovviare al problema "ho sbagliato la scelta della vittima", ho scelto cioè una pagina che servirà tra poco. Come funziona? Ovviamente mi trovo nella situazione in cui ho bisogno di un frame libero ma sono tutti pieni. Tengo un pool (un insieme) di frame liberi, quando parte il page fault seleziono una vittima, che viene aggiunta al free-frame pool. Tutte le vittime che aggiungo alla pool verranno copiate su disco in un secondo momento. Quando e come questo salvataggio su disco viene fatto, è la parte difficile dell'algoritmo. Una volta inserita nel pool, se faccio riferimento alla pagina la recupero.

APPLICATIONS AND PAGE REPLACEMENT

Se nel determinare la politica di allocazione/sostituzione pagine, faccio decidere anche al programma utente oltre che al kernel, potrei avere dei vantaggi.

L'applicazione potrebbe infatti avere più chiaro cosa fa adesso e cosa farà in futuro, un esempio sono i database, programmi che fanno tante visite in memoria. Essi potrebbero avere delle strategie nei loro algoritmi che aiutano o sostituiscono il S.O. nel decidere cosa sta in memoria e cosa no. Non entriamo nei dettagli, ma citiamo questa possibilità per capire che esiste un potenziale conflitto nel creare duplicati di pagine quando sia il kernel sia il processo utente tentano di fare lo stesso lavoro.

ALLOCATION OF FRAMES

Un altro aspetto da valutare per ottimizzare l'efficienza del paging è capire qual è il numero migliore di frames da assegnare. Per ora chiediamoci: Qual è il minimo?

Un frame non è sufficiente mai, perché esistono istruzioni Assembler che richiedono più di una pagina in parallelo. Ovviamente dipende molto dal contesto e non è possibile definirlo a priori, ma esploriamo in generale le possibilità a disposizione.

FIXED ALLOCATION

- **Equal Allocation:** alloco a tutti i processi un numero fisso di frame, una volta deciso quanti frame assegnare. Avendo ad esempio 100 frames a disposizione e 5 processi, alloco 20 frames ciascuno. Questo tipo di allocazione è molto equilibrata perché so a priori quanta Ram dare ad un processo, ma potrei sprecare dei frames perché un processo potrebbe non usarli tutti
- **Proportional Allocation:** se conosciamo la dimensione di un processo, cioè la dimensione del suo "address space" possiamo attribuirgli Ram in maniera proporzionale. Vediamo come

$$S_i = \text{dimensione del processo } p_i \quad S = \text{dimensione totale processi} = \sum s_i$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m \text{ dove } m = \text{numero totale di frames}$$

Esempio

$M = 64 \text{ frames totali}$. $S_1 = 10 \text{ pagine}$. $S_2 = 127 \text{ pagine}$.

$$a_1 = \frac{10}{137} \times 64 \cong 4 \text{ . } a_2 = \frac{127}{137} \times 64 \cong 57$$

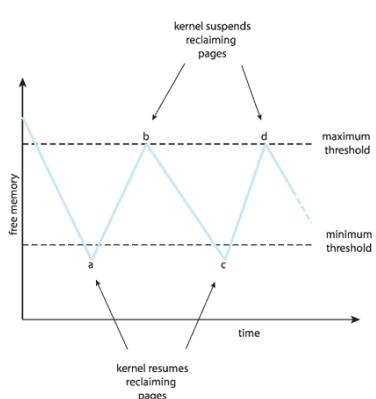
Un'ulteriore problematica si presenta quando cerchiamo la vittima. La cerchiamo solo fra le pagine del processo attuale o globalmente fra tutti? Entrambe sono valide:

- **Global replacement:** si cerca la vittima fra tutti i frames, di fatto un processo può rubare frames, quindi memoria ad un altro. Questo può però impattare negativamente sulla loro esecuzione, poiché ogni processo non è più indipendente dagli altri.
- **Local replacement:** L'alternativa è che un processo cerchi solo fra le sue pagine. Ciò può comportare un sottoutilizzo della memoria. Infatti un processo che non usa tutti i suoi frames non li cederà mai a chi ne ha bisogno.

RECLAIMING PAGES

Un altro concetto fondamentale : un processo può accedere ad una lista globale di free frame. Mi pongo come obiettivo il fatto che questa lista abbia sempre un

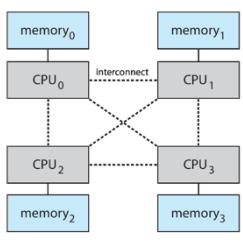
certo numero di frame liberi disponibili, attivo perciò il page replacement non quando la lista è vuota ma quando il numero di free-frames scende sotto una certa soglia. **Tale strategia è detta Reclaiming Pages.**



Supponiamo di porre sulle ascisse il tempo e sulle ordinate il numero di frame liberi. Come detto scesi sotto una certa soglia attiviamo il Reclaiming pages, quando la free list diventa sufficientemente piena disattiviamo il Reclaiming pages(istante b e d).

NON UNIFORM MEMORY ACCESS

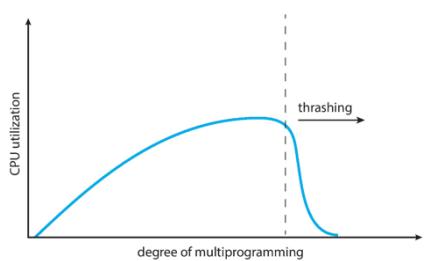
Con **non uniform memory access (NUMA)** si intende il fatto che nei sistemi moderni multicore, più processori hanno una memoria condivisa.



Concettualmente è come se la Ram fosse divisa in partizioni di Ram locali ad ogni processore. Ad esempio, la CPU 0 accede più velocemente alla memory 0, che alla 1. È opportuno quindi che un processo su una data CPU usi i free-frame sulla memoria locale a quella CPU.

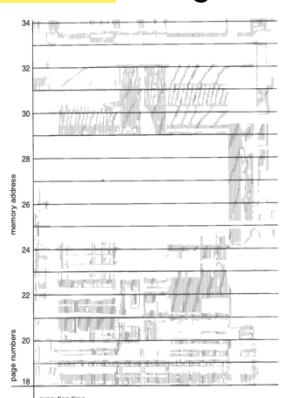
THRASHING

Quando un sistema va in crisi? Se il processo non ha abbastanza frame, allora continuerà a generare molti page fault, che, come abbiamo detto, son costosi. Questo significa innescare il threading, avere un degrado di prestazioni molto forte. Significa che stiamo salvando memoria per far girare quanti più processi , ma



Nessuno di questi processi viene effettivamente eseguito. Paradossalmente stiamo cercando di non avere buchi nell'uso della CPU, ma superata una certa soglia la CPU rimane inutilizzata. Non perché i processi non ne hanno bisogno ma perché non riesco più ad assegnare le pagine. Questo è il **thrashing**. Cioè ci

porta ad osservare che i processi lavorano con un **modello di località**. Nel grafico sulle ascisse ho il tempo di esecuzione, quindi più avanza più istruzioni saranno eseguite. Sulle ordinate ci sono le pagine, quindi lo spazio di indirizzamento virtuale. Il programma parte e lavora sulle pagine tra 22 e 26 e qualcuna sul 31-32. Dopo un po' lavora sulle pagine dalla 18 alla 24 e 29-30. Il programma lavora per strutture dati e in thrashing ci andiamo quando la sommatoria delle località dei processi è maggiore della memoria disponibile.



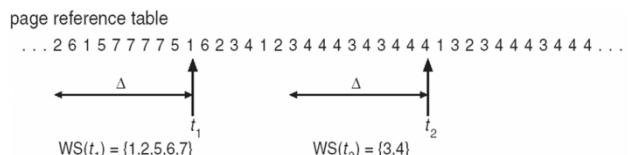
WORKING SET MODEL

Un primo modello che usa il concetto di località degli accessi è il **Working Set Model**. La **località degli accessi** è l'insieme delle pagine su cui un processo lavora in un dato intervallo, nel presente e nel futuro. Un modo di decidere qual è la località è di vedere su quali pagine ho lavorato negli ultimi istanti. Chiamiamo Δ il **working set window**, WSS_i il **Working Set Size** del processo p_i, cioè il numero di pagine su cui lavorato. Il WSS varia quindi da 1 a Δ nel tempo. Se la finestra temporale fosse troppo piccola il WSS potrebbe non comprendere la località, viceversa se fosse troppo grande potrebbe comprendere tutto il programma. Cerchiamo quindi di prendere una finestra temporale adeguata e chiediamo al sistema $D = \sum_i WSS_i$ prendendo i frame di tutto il processo.

Abbiamo un $\Delta = 10$, ad un certo istante t₁

il Working set = {1 2 5 6 7}. WSS = 5.

Nell'istante t₂ WS = {3 4} e WSS = 2. Se



riusciamo a fare in modo che le pagine che stanno nei frame, quindi D, coincida con il working set abbiamo implementato con successo un Working Set Model. Se la memoria però è minore di D, allora si verifica Thrashing. Questa politica ha purtroppo un difetto: dopo ogni istante il Working Set potrebbe cambiare aumentando o diminuendo. Ad esempio, da t₁ all'istante successivo in cui facciamo accesso a pagina 6, non avrei page fault perché 6 era già in memoria; tuttavia, se voglio mantenere i frame allineati al working set devo buttare la pagina 2, perché esce fuori dalla working set window, il working set diventa {1 5 6 7} ho quindi buttato via una pagina quando non ho effettivamente bisogno di fare spazio. Perciò non faccio più swap out ad ogni page fault, ma ad ogni accesso ad una nuova pagina. Nella realtà si tende ad approssimare tale strategia e non applicarla alla lettera. Un modo per farlo è quello di usare il reference Bit. Supponiamo di avere $\Delta = 10.000$, invece di guardare tutta la window ad ogni accesso decidiamo di ricontrillare ed aggiornare il Working Set ogni 5000 istanti, attivando un interrupt che per ogni pagina azzera un reference bit, ma prima salva il reference bit precedente. Durante i prossimi 5000 istanti ogni accesso a pagina mette il reference bit a 1. Guardando il reference bit posso vedere le pagine a cui ho fatto o non fatto accesso negli ultimi 5000 istanti. Posso poi utilizzare tale informazione per i page fault dei prossimi 5000 istanti. Tale strategia si chiama **Campionamento fisso**. Un'altra strategia è la **Page Fault Frequency**, cioè se ho troppi page fault aumento il numero di frames dedicati al processo, se i page fault sono pochi li diminuisco.

Un'implementazione semplificata può utilizzare una soglia anziché due. Anziché ad intervalli di tempo fissi, lavoriamo solo quando c'è un page fault, visto che il page fault di per sé blocca già il processo. Misuriamo quindi la distanza di tempo dall'ultimo page fault, chiamata τ . Definiamo una costante c, definita sperimentalmente. Se $\tau < c$, allora aggiungo un frame e ci mettiamo la nuova pagina. Se $\tau \geq c$ allora non aggiungo frame, ma faccio page replacement.

Scelgo come vittima una delle pagine a cui non ho fatto accesso dall'ultimo page fault. Per tener conto di questo uso il caro vecchio reference bit. Sto in sostanza usando i page fault come intervallo di tempo. Vediamo un esempio.

Tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Stringa	6	4	4	3	2	4	4	4	1	3	3	2	4	4	5	1	2	2	2	6	1	2	5	4
Frame 0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	5	5	5	5	6	6	6	6	6
Frame 1	-	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	2	2	2	2	2	2	2	2
Frame 2	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1
Frame 3	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-	-	5	5	5
Frame 4	-	-	-	-	-	-	-	-	1	1	1	1	1	1	1	1	1	1	1	-	-	-	-	4
Fault	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Ref Bit									0															

vede = ref bit = 1
rosso = victim

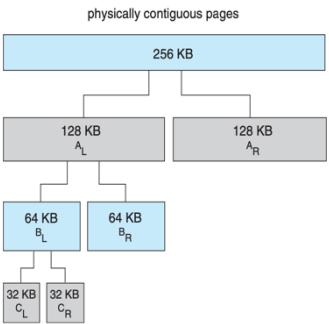
quindi aggiungo un frame e dentro ci metto pagina 1, azzero poi i reference bit. All'istante 15 ho un page fault, $\tau > 3$, vedo che 6 e 1 hanno reference bit a 0, vengono quindi rimpiazzate, metto pagina 5 e così via.

ALLOCATING KERNEL MEMORY

Finora ci siamo occupati della memoria da allocare a processi utente, con l'obiettivo di metterne quanti più possibile, senza andare in trashing. Viene gestito in modo diverso il kernel, poiché ha bisogno di alcune strutture dati allocate in modo contiguo, come la page table. Esiste quindi un memory pool dedicato solo al kernel. Vediamo delle alternative di allocazione

BUDDY SYSTEM

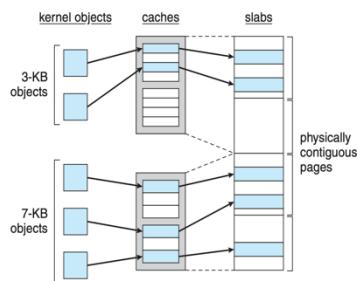
Il buddy system è un allocatore per il kernel, che alloca in modo contiguo, ma cercando di ridurre la frammentazione esterna. L'idea è di non fare una paginazione pura, ma di allocare in modo contiguo dei blocchi di memoria di dimensione fissa. Il buddy alloggia per potenze di 2, arrotondando alla potenza di 2 più grande di ciò che serve. Ad esempio, abbiamo una partizione libera di 256 KB ed il kernel richiede 21 KB. La partizione sarà divisa in due da 128 KB e una delle due in due da 64 KB, ulteriormente da 32 KB. Poi in una delle due da 32 viene allocato il blocco richiesto dal kernel.



SLAB ALLOCATOR

Lo slab allocator è un allocatore alternativo al buddy che cerca di avere un doppio livello di allocazione, ma senza usare le potenze di 2. Abbiamo lo slab, cioè una o più

pagine contigue, e la Cache, fatta da uno o più slab. Si cerca di allocare per un tipo di struttura dati del kernel una singola cache fatta di slab. Quando si crea una cache la si riempie con oggetti liberi, che vengono allocati quando vengono usati per una struttura dati del sistema. Ad esempio il kernel ha bisogno di struct da 3KB e da 7KB.



Quelle da 3 vengono allocate in una cache che fa riferimento alle pagine in memoria, quella da 7 KB ad un'altra cache con altri slab.

PREPAGING

Per diminuire il numero di page fault dei primissimi accessi in memoria, possiamo prepaginare alcune delle pagine iniziali del processo, cioè lo portiamo in frame prima che venga fatto il riferimento. Può essere sconveniente perché non è detto che poi siano effettivamente usate. S: numero di pagine prepaginate , a: numero di pagine effettivamente usate . S x a: numero pagine usate effettivamente. S x (1-a) : pagine non necessarie (overhead).

PAGE SIZE

Abbiamo visto che dal punto di vista della frammentazione interna (esterna viene risolta dal paging) è meglio avere pagine piccole, ma più sono piccole, più pagine avrò, più la tabella delle pagine sarà grande. Un altro aspetto di cui tener conto è L'I/O overhead: dal punto di vista di costo della copia di pagine da disco a memoria è meglio avere pagine grandi, poiché più sono grandi, meno volte dovrò accedere al disco. Il numero di page fault diminuisce infatti con pagine di dimensioni maggiori. Un altro concetto solo accennato è la **TLB reach**: quanto copre la TLB è dato da
TLB Reach = TLB size * Page size.

PROGRAM STRUCTURE

Cerchiamo di capire come la struttura di un programma può avere un impatto sulla località. Ho due programmi, entrambi azzerano una matrice, ma uno procede per righe ed uno per colonne. Ogni riga è salvata su una pagina, quindi il programma 1 in totale da $128 \times 128 = 16.384$ page fault, mentre il programma 2 da 128 page fault.

```
for (i = 0; i < 128; i++)      for (j = 0; j < 128; j++)
    for (j = 0; j < 128; j++)    for (i = 0; i < 128; i++)
        data[i,j] = 0;           data[i,j] = 0;
```

Chapter 11: Mass – Storage Systems

OBIETTIVO In questo capitolo descriveremo le memorie di massa e di schedulazione, gestione del dispositivo e delle strutture RAID.

OVERVIEW Cos'è una memoria di massa? Per memorie di massa intendiamo gli HDD

- Access Latency = Average access time = average seek time + average latency
 - For fastest disk 3ms + 2ms = 5ms
 - For slow disk 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- For example to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead =
 - 5ms + 4.17ms + 0.1ms + transfer time =
 - Transfer time = $4\text{KB} / 1\text{Gb/s} * 8\text{Gb / GB} * 1\text{GB} / 1024^2\text{KB} = 32 / (1024^2) = 0.031 \text{ ms}$
 - Average I/O time for 4KB block = $9.27\text{ms} + .031\text{ms} = 9.301\text{ms}$

e le NVM (non-volatile Memory).

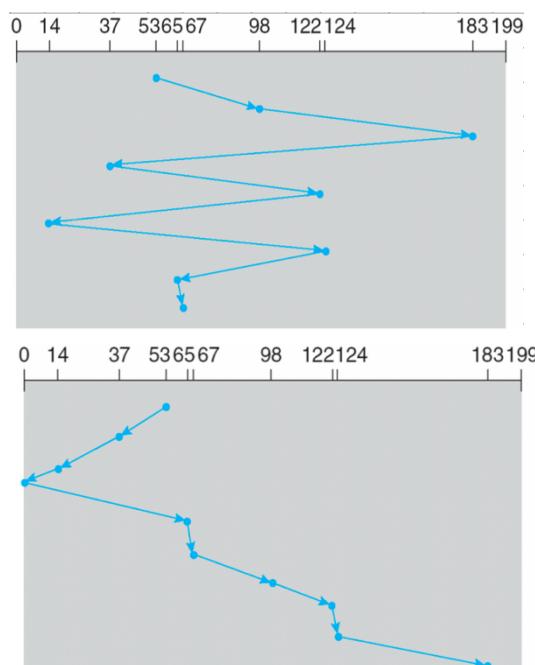
- Gli HDD sono molto più lenti della RAM, vanno da qualche Gigabytes a molti Terabytes, hanno un Transfer Rate molto lento, circa 1 Gbit/sec (occhio, gigabit non gigabyte).
- Le memorie NVM come i Solid-State Disks (SSD) sono molto più veloci ma hanno un costo maggiore.

Una memoria non volatile assomiglia più ad una Ram che ad un HDD, viene letta non per bit, ma per pagine. IL contro è che hanno un limite al numero di riscrittura, quindi negli anni sono state sviluppate strategie per far sì che il sistema scriva sulle varie porzioni della memoria in modo equilibrato, in modo da evitare il deperimento di alcune parti della memoria prima di altre. La vita di tali memorie viene misurata in drive writes per day (DWPD). Vediamo come possiamo gestire questi dischi.

Nand Flash Controller Algorithms: Se non si fanno riscritture le pagine avranno un po' di dati validi ed alcuni non validi. Il gestore del dispositivo deve quindi avere una tabella per tenere traccia di tali dati.

Volatile Memory: Talvolta anche le memorie volatili, come la RAM, possono essere usate per complementare la memoria di massa, a patto che ci ricordiamo che alla fine del lavoro mi ricordo di salvare i dati che mi servono, se non perdo tutto. Entriamo ora un pochino nello specifico e vediamo qual è la struttura dei dischi magnetici, che ad oggi con leggere modifiche si usa per gli SSD.

DISK STRUCTURE



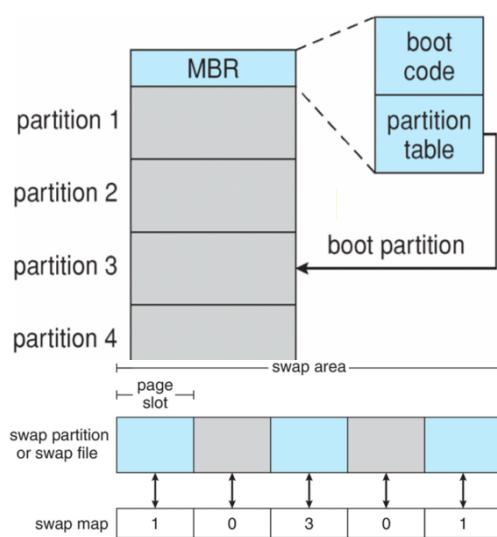
Un disco magnetico è visto come un **vettore di blocchi logici**, ogni blocco è un'unità di informazione che è utile leggere in un colpo solo. Questo vettore di blocchi logici deve essere rimappato in settori sequenziali sul disco. Il settore 0 è il primo settore della traccia più esterna. Si chiama **Disk Attachment** l'interfaccia con cui il disco interagisce con il resto del sistema, come SATA o USB, che contengono i bus di I/O. Il termine **Scheduling** indica una componente del Sistema Operativo che decide dove scrivere e leggere sul disco per minimizzare i tempi di tali operazioni. Bisogna infatti considerare che sul disco arrivano richieste dal Sistema Operativo, dai processi di sistema e quelli utente.

Conviene quindi riordinarle e ottimizzarle. Supponiamo che i controller del disco possono tenere una serie di operazioni in parallelo in sospeso. Su un disco, con tracce da 0-199, arrivano delle richieste: 98, 183, 37, 122, 14, 124, 65, 67. La testina è sulla traccia n° 53, usando una strategia First come First served (FCFS) che mostra come non ha senso andare da una parte all'altra del disco senza fermarmi sui blocchi intermedi, come un ascensore che sale dal piano 53 al 183, per poi tornare al 67. Per questo motivo si usa la politica **SCAN**, che fa esattamente come un ascensore. Una variazione è il **C-SCAN** che prevede un'unica salita, una volta arrivato in alto torna indietro.

Qual è il miglior algoritmo di disk scheduling?

Lo **shortest seek time first (SSTF)**, cioè cerco il dato più vicino alla testina e vado avanti così. SCAN, C-SCAN vanno bene per i dischi molto usati, ma possono causare starvation. Linux per evitarlo implementa un deadline scheduler, dove le operazioni di I/O vengono ordinate in base alla loro 'scadenza'. Il Discorso fatto fino ad ora ha senso solo su memorie magnetiche, poiché nelle memorie non volatili non c'è una testina che si muove. Sugli NVM c'è il cosiddetto NOOP (no scheduling). Nel momento in cui sul disco ci sono delle informazioni errate, sarebbe opportuno individuare l'errore e nel migliore dei casi riuscire a correggerlo. Dal punto di vista della gestione del disco, esso va formattato, cioè diviso in settori che il controller legge o scrive. Per arrivare ad un file system, cioè una formattazione logica serve quantomeno partizionare il disco in uno o più gruppi di cilindri. Questo per evidenziare che fra la visione logica di un disco a quella fisica ci sono dei livelli intermedi. Una delle partizioni è la root partition che contiene il sistema operativo, **mounted**, cioè agganciarla ad un OS in esecuzione, nella fase di boot.

Quando si fa un mount si verifica che il filesystem è corretto. A scopo esplicativo vediamo che il sistema Windows.



La ROM contiene il basic I/O system (BIOS), che legge e carica in RAM il Master Boot Record (MBR) che ha le info su come il disco è partizionato e dove trovare il kernel per caricarlo in RAM. Questo è il bootstrap.

Sul disco c'è anche la **partizione di swap**, visto come un vettore di blocchi, mappato con una 'swap map', praticamente è il backing store della RAM.

La disponibilità della rete fa sì che si possano avere dischi forniti da server e non fisicamente presenti sul dispositivo in uso. Distinguiamo quindi lo storage in host-attached, network-attached e cloud. Uno **storage array** è un insieme di dispositivi di storage messi insieme per fornire un servizio.

DISK STRUCTURE

La struttura RAID aumenta l'affidabilità di un disco aggiungendo ridondanza. Le strategie usate sono molteplici, l'obiettivo è aumentare il tempo medio tra due guasti. Ogni dato è copiato due volte, perciò per perderlo si dovranno perdere entrambe le copie. Consideriamo un disco con 100.000 ore di **mean time to failure**. Ci vogliono 10 ore per ripararlo (va ricoperto il dato manualmente). IL tempo medio per perdere un dato è $100.000^2 / 2 \times 10 = 500 \times 10^6$ ore, cioè 57.000 anni. Ciò significa che se un solo disco fallisce ogni 100.000 ore, statisticamente lo stesso dato fallisce su entrambi i dischi in contemporanea ogni 57.000 anni.

Chapter 13: File System Interface

OBIETTIVO Il capitolo 13, potrebbe essere titolato “specifiche per il File System per il sistema operativo”, cioè vedremo come vogliamo che il file system sia fatto, nel capitolo 14 vedremo com’è fatto il file system. Vedremo inoltre cosa sono i file, i direttori e che tipo di accessi vogliamo ai dischi, la protezione e tanto altro ancora ☺

FILE CONCEPT

Partiamo dalla cosa più semplice: i file. Di fatto un file è una sequenza di dati che può essere vista, ad un basso livello di astrazione, come uno spazio di indirizzamento contiguo. Oltre ad essere una generica sequenza di byte, possiamo associare un ‘tipo di dato’, cioè specificare se il file contiene dati o istruzioni, cioè codice. I file di dati possono poi essere numerici, caratteri o binari. Tale classificazione non è ‘rigida’ ma

varia in base al tipo di utilizzo. Esistono poi file di testo, sorgente ed eseguibili. Oltre a ciò, un file contiene, interessa quali operazioni posso fare su di esso come: creazione, scrittura, lettura, riposizionamento nel file, troncare un file, aprirlo e chiuderlo.

- Name – only information kept in human-readable form
- Identifier – unique tag (number) identifies file within file system
- Type – needed for systems that support different types
- Location – pointer to file location on device
- Size – current file size
- Protection – controls who can do reading, writing, executing
- Time, date, and user identification – data for protection, security, and usage monitoring

Attributi di un File

OPEN FILE

Per agire su un file è necessario aprirlo, per fare una open serve una tabella che tenga traccia dei file aperti, tale tabella prende il nome di **Open File Table**. All’interno di un file aperto ci sarà un puntatore all’ultima posizione scritta/letta, detto **file pointer**. Un file può essere inoltre condiviso tra più processi, per tenere traccia di quanti stanno accedendo in parallelo al file si utilizza un contatore detto **file - open count**. Altra info importante da salvare è la posizione del file sul disco e chi a tale file può accedervi. Per far sì che il file sia aperto in parallelo è poi necessario gestire una politica di accessi che gestisca eventuali sincronizzazioni. Il **file locking**, supportato da molti os, è implementato fornendo dei ‘lock’ associati automaticamente ad un file. Tali lock sono di due tipi: **shared lock**, il lock è condiviso, ma chi accede al file può farlo in sola lettura, **exclusive lock**, voglio modificare il file quindi posso accedervi solo io. Un’altra caratteristica dei lock è che possono essere obbligatori o lock advisory, cioè non sono obbligatori ma servono solo per dichiarare esplicitamente cosa sto facendo. Sulle slide è presente un esempio java per far capire che un lock può essere fatto anche solo su una porzione del file anziché su tutto.

FILE STRUCTURE

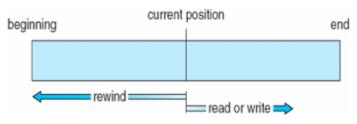
Come sono organizzati i dati in un file?

- **non ha una struttura**, ma è un semplice vettore di byte che va intrepretato da chi scrive/legge;
- In **Record**, dove il file è diviso in un record, cioè righe di testo, a lunghezza fissa o variabile;
- In **strutture complesse**, formattate e rilocabili come gli eseguibili che hanno sezioni diverse e vanno tutte interpretate in modo opportuno.

Queste non sono classificazioni rigide, ma spetta all’OS decidere come interpretare il file.

ACCESS METHODS

Concentriamoci ora sulle tipologie di accesso al file: sequenziale o diretto.

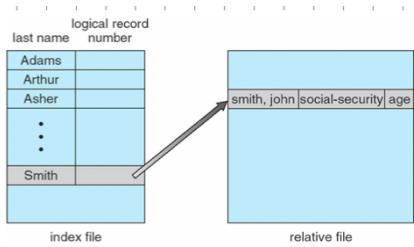


sequential access	implementation for direct access
reset	$cp = 0;$
read next	$read cp;$ $cp = cp + 1;$
write next	$write cp;$ $cp = cp + 1;$

Un File ad **accesso sequenziale** ha generalmente una posizione iniziale, finale e corrente. Quindi le uniche operazioni possibili saranno 'read next', 'write next', 'reset'.

Nei File ad **accesso diretto**, il file è fixed in 'logical records' che possono essere parole, blocchi etc. ed essendo n: numero di blocco, le operazioni consentite sono 'read n', 'write n', 'position to n', e una volta posizionati si può poi accedere

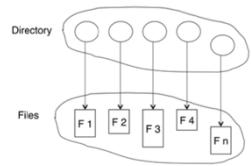
sequenzialmente tramite 'read next', 'write next'. A sinistra è mostrato come si può simulare l'accesso sequenziale nei file ad accesso diretto tramite cp (current pointer).



A partire da questi due metodi di accesso si possono realizzare tipologie più complicate basate ad esempio sulla creazione di un indice per il file. Nell'esempio c'è un file che fa da indice ed uno che contiene i dati. Il file indice contiene cognome ed indirizzo logico che consente di raggiungere per accesso diretto il secondo file.

DIRECTORY STRUCTURE

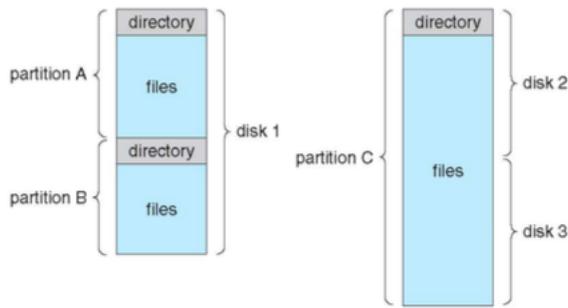
Con struttura direttori si intende una modalità per poter collezionare più file in una struttura gerarchica che può essere organizzata su più livelli. In un disco ci sono dei nodi e ciascuno corrisponde ad un file. Ne vedremo di altri tipi non temete.



DISK STRUCTURE

Un disco può essere suddiviso in partizioni e può includere una politica RAID. Una partizione può essere usata in modalità **raw**, cioè andando a sistemare manualmente i blocchi in memoria, oppure può essere **formattata** con un file system.

Ogni **volume** può contenere più partizioni ed un solo un file system, caratterizzato da



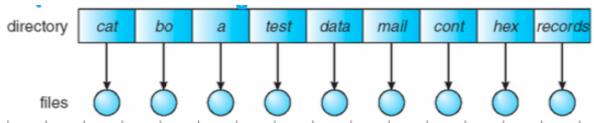
direttori, tabelle e info varie. Per rendere il concetto più chiaro a sinistra viene rappresentata la struttura di un disco. Il disco 1 è stato diviso in due partizioni ognuna con il suo File System. I dischi 2 e 3 sono invece uniti insieme in un'unica partizione.

Operations Performed on Directory: L'operazione principale che si fa su un direttorio è cercare un file, si possono inoltre creare, eliminare, o elencare i files.

DIRECTORY ORGANIZATION

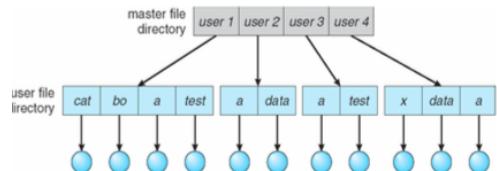
I Direttori sono organizzati in modo logico per trovare un file in maniera efficiente. Si usano quindi strategie per nominare i files, poiché è più comodo per l'utente, tenendo conto sempre che **due file possono avere lo stesso nome**, sia che **un file può avere più nomi diversi**. Inoltre può essere utile raggruppare i file per categoria, tipo ecc.

SINGLE-LEVEL DIRECTORY



I nodi sono i file, i rettangoli i direttori. Con un single level directory è difficile sia dare nomi, sia fare raggruppamenti.

TWO-LEVEL DIRECTORY



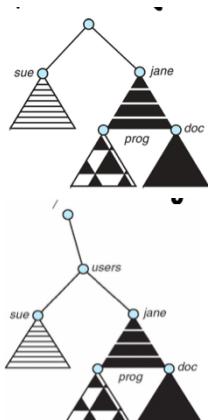
A più livelli è molto più semplice raggruppare file e posso anche dare lo stesso nome a utenti differenti, risolvendo i problemi di naming e grouping, devo però riferirmi al path name, cioè il nome composto.

Si possono usare poi directory a 3 o 4 livelli ma ciò complica la ricerca del file. Il current directory è utile per capire in quale directory ci troviamo, il path name può essere quindi relativo o assoluto. L'assoluto parte dalla radice, il relativo parte dal cd. E' importante sapere che i direttori non sono necessariamente un albero, perché i filesystem permettono di condividere i nodi, ciò significa che file e directory possono appartenere a più directory. Perciò la struttura utilizzata è un **Grafo-Aciclico**. I file possono avere quindi path name diversi, questo fenomeno si chiama 'aliasing'.

Il problema nasce nel momento in cui voglio cancellare un file condiviso, perché eliminandolo da un direttorio, devo eliminarlo automaticamente dagli altri, generando il **'dangling pointer'** cioè un puntatore a null. Ci sono due soluzioni possibili: i **back pointers**, letteralmente puntatori all'indietro, perché puntano anche i direttori che contengono il file, di modo che quando viene cancellato, anche gli altri puntatori al file possono vedere la modifica. Tale backpointer può essere implementato come vettore o come lista daisychain. Un'altra possibilità è tenere il file vivo finché non viene cancellato da tutti i direttori.

Un altro problema è che se non possiamo vedere globalmente il grafo dei direttori possiamo generare dei cicli. Per evitarlo possiamo adottare più strategie: una è evitare i cicli per costruzione, cioè i direttori non si condividono. Un altro è implementare una garbage collection, cioè ogni tanto si fa un check e se si trovano cicli si rimuovono. L'ultima soluzione prevede un algoritmo di individuazione dei cicli ogni volta che creo un nuovo link tra i direttori.

MOUNTING FILE SYSTEM



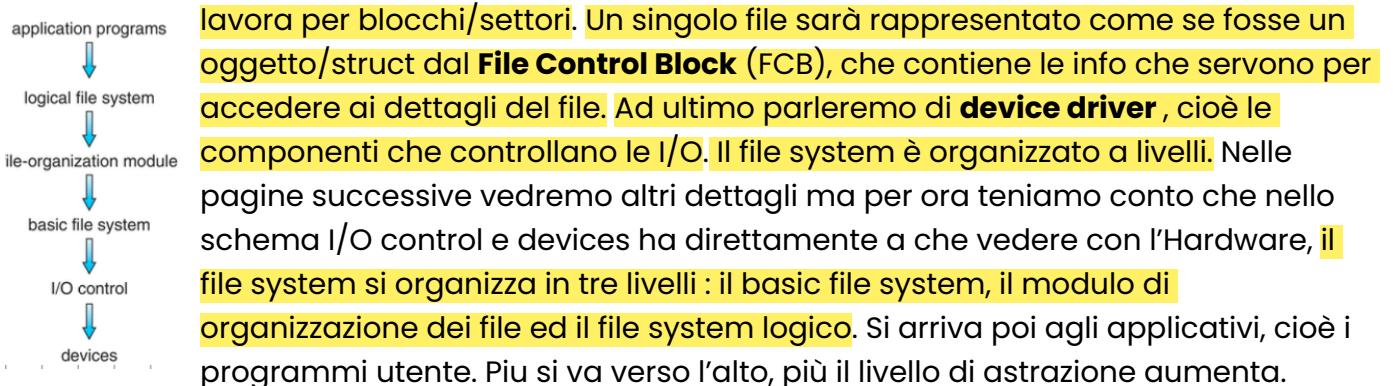
Fare mounting significa agganciare quelle info che abbiamo descritto al sistema operativo. L'idea è di poter vedere in un unico grafo più dischi. E' possibile fare file sharing tra più utenti, servono quindi meccanismi di protezione per decidere chi può scrivere, leggere ecc. In un sistema multi-user esiste una gerarchia in cui un utente è identificato sia da uno **user Id** che da un **group Id**. Per quanto riguarda il file sharing a livello di rete ci sono protocolli come FTP o file system distribuiti o quasi in automatico nel world wide web. In generale non è comunque argomento di questo corso. Nel momento in cui c'è di mezzo la rete è importantissimo tenere conte e saper gestire gli errori. Mentre se non trovo un dato sul disco ho un errore sul disco, sulla rete il più delle volte è semplicemente caduta la connessione. Occorre inoltre specificare come sincronizzare i file, visti i numerosi accessi multipli. Se più processi user accedono ad un file il puntatore è condiviso o ognuno ha il suo? As usual dipende dal sistema. I tipi di accesso in generale sono lettura, scrittura, esecuzione, append. In unix ad esempio ci sono l'owner, group e public che hanno ognuno privilegi diversi.

Chapter 14: File System Implementation

OBIETTIVO in questo capitolo, come già accennato, vedremo com'è strutturato il file system.

FILE SYSTEM STRUCTURE

Per comprendere la struttura di un file dobbiamo vedere come da un punto di vista logico si organizza un file che dal punto di vista fisico è un insieme di blocchi su un disco. Per quanto riguarda il file system, parliamo di informazioni che risiedono in memoria secondaria, quindi sul disco. Perciò non si cancellano quando spegniamo il sistema. Il File system fornisce un'interfaccia verso tutte le informazioni in memoria di massa (secondaria). E fornisce una traduzione da logico a fisico, oltre che un accesso efficiente al disco. Il disco sappiamo che



FILE SYSTEM LAYERS

Vediamo nei dettagli. I **device drivers** sono le componenti software che gestiscono l'accesso ai **dispositivi di I/O** e dal punto di vista del File System, il device driver non riceve comandi del tipo "read a .txt" ma "read drive 1, cylinder 72, track 2, sector 10" e porta ciò che leggi in memoria all'indirizzo 1060, siamo cioè vicini alle informazioni **livello hardware**. Il **basic file system** riesce già a vedere informazioni indipendenti dal dispositivo fisico, ma connesse al dispositivo logico, riceve comandi del tipo "prendi il blocco 123", vede quindi il file come vettore di blocchi. A questo livello cercano di velocizzare le cose, gestendo dei buffer che mantengono i dati in transito, evitando letture o scritture multiple dal disco, mantenendo ciò che serve in memoria. Il livello successivo verso l'alto è il **File organization module**, il modulo di organizzazione dei file, è il modulo che capisce e gestisce la collocazione dei file su blocchi logici su disco, traduce il blocco logico in blocco fisico, gestisce lo spazio libero e l'allocazione sul disco. Qui il livello di astrazione non è più il singolo blocco, ma un gruppo di blocchi.

Il **Logical file system** capisce i metadati e traduce ad esempio il nome del file nel numero del file o il puntatore al file, nei sistemi UNIX-like i file sono identificati con un numero, mentre nei sistemi tipo Windows si usano gli handle, che sono dei puntatori. A questo livello si mantengono, in UNIX, degli inodes, di cui parleremo. In sostanza è il livello più alto a cui si gestiscono le informazioni che sono più vicine a ciò che vuole l'utente.

La gestione a strati serve per ridurre la complessità delle operazioni e fornire interfacce chiare.

Di file system inoltre ce ne possono essere vari che differiscono per come codificano il file e per i dispositivi supportati: Unix ha UFS, Windows ha FAT, FAT32, NTFS, Linux addirittura più di 130. Noi in generale vedremo, come fatto per la memoria, strategie generiche e qualcosa di Unix vedendo gli Inodes e qualche dettaglio su Os161 nella parte internals.

FILE SYSTEM OPERATIONS

Diamo un'occhiata a quali sono le operazioni che si fanno su un file system, che saranno ciò che deve fornire l'API di un file system, cioè le funzioni che vanno supportate. Per effettuare tali operazioni sono necessarie delle strutture dati, gestite dal Kernel, sia su disco che in memoria. Per prima cosa vanno gestite le informazioni che dovranno essere gestite sono le info che permettono di fare bootstrap, abbiamo due strutture utili per tale operazione: **Boot Control Block**, sul disco, che contiene le informazioni da leggere subito per far partire il sistema dal calcolatore ed un **Volume Control Block**, che contiene il superblock e la master file table, che in sostanza contengono informazioni sulla descrizione delle partizioni su disco. Non approfondiamo per motivi di tempo. Sul disco ci sono poi le strutture a direttori, che permettono di collezionare i file e organizzarli per agevolare le ricerche.

L'informazione principale sul singolo file è il **File Control Block (FCB)**, contenente dettagli sul file alcuni relativi ai permessi sul file, alle date di creazione/modifica, il numero di inodes e altre informazioni a seconda della tipologia di file.

Tutto ciò che abbiamo visto sinora è salvato principalmente sul disco, in memoria invece quando l'OS fa il mount di un filesystem, lo aggancia cioè al disco, ci saranno strutture dati che replicano o completano le informazioni che stanno sul disco. Tutte le volte che ad esempio abbiamo gestito un file in C, abbiamo in realtà replicato in memoria delle informazioni che erano sul file. Tipicamente infatti ciò che si fa è: prendi i dati, copiali in una struttura dati in Ram, lavoraci e ricopia tutto sul file, poiché lavorare in Ram è tipicamente più comodo. Questo succede anche nella gestione del file system, cioè: le strutture dati in memoria che il Kernel usa per lavorare sui file, sono una 'ridondanza' cioè una copia di quelle sul disco, ma sono necessarie per poter lavorare in modo efficace. La **mount - table** non lo è, esiste cioè solo in memoria e su di essa sono memorizzate le informazioni sui file-system che sono agganciati e dove sono agganciati. Poi ci sono le cosiddette **system-wide open-file table** e **per-process open-file table** la prima tiene conto di tutti i file aperti nel sistema operativo, la seconda tiene conto dei file aperti da ogni singolo processo.

La (a) corrisponde ad una open di un file, la (b) corrisponde alla read di un file. Alla open

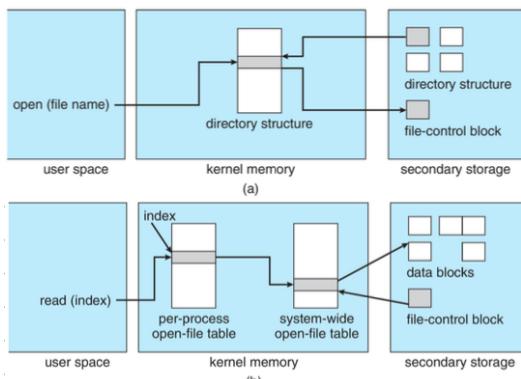


Figura importante per esame

possiamo associare una close, alla read una write, nel senso che si svolgono allo stesso modo di quello qui illustrato. La colonna di sinistra rappresenta il processo utente, quella centrale le strutture dati gestite dal kernel, quella di destra rappresenta i dischi. Bisogna inoltre tenere conto che la open viene fatta una tantum sul file, mentre la read viene ripetuta più volte. Perciò, la open può essere vista come una specie di inizializzazione, mentre la read lavora sul risultato prodotto dalla open.

Nella (a) a destra vediamo che ci sono dei blocchi che rappresentano la struttura a direttori ad un blocco che rappresenta il FCB. La open parte del nome del file (come in C) e localizza sul disco il File-Control Block, per fare ciò serve avere una copia della directory structure, cioè una tabella di ricerca nella memoria Kernel. In questo modo si ottiene il puntatore al file, come quando facciamo fp = fopen('a.txt0') in C. Di fatto accade che il file-control block viene copiato in una tabella, la system wide open file. Quindi la system-wide table contiene una copia del file, ed il processo che ha fatto la open troverà una entri nella sua 'per-process open-file table'. In questo modo se un altro processo fa open dello stesso file, esso non viene copiato nuovamente in memoria ma semplicemente la sua per-

process table avrà una riga che punterà alla stessa copia del file già fatta nella system-wide table. Queste due tabelle devono inoltre permettere di poter avere a processi diversi, puntatori al file diversi, cioè uno legge in testa, l'altro in coda ecc. oppure condividere i puntatori.

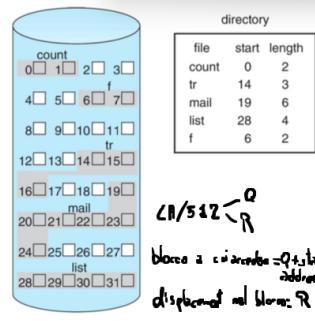
DIRECTORY IMPLEMENTATION

Cerchiamo ora di capire come si possono realizzare i direttori. Sostanzialmente i direttori sono una tabella di simboli speciali, poiché non si cerca in una sola lista/settore ma si cerca un nome fatto da 'pezzi' tipo 'home/peppo/video/Ue_trmon.mov' quindi come visto in ch13, si cerca in un grafo aciclico. Si possono visualizzare i direttori come:

- **Liste Lineari** di nomi di file, in cui ognuno punta al FCB. Sono più semplici da gestire, ma costano di più in termini di tempo, poiché richiedono ricerche lineari.
- **Hash Table** sono una valida alternativa, perché permettono un accesso diretto evitando la ricerca lineare. Il problema è che potrei dover cercare un file che è in una sottocartella, di una sottocartella, ..., della root. Ciò vuol dire che devo cercare il primo direttorio del 'path name' in una lista/tabella. Il secondo in un'altra lista/tabella e così via. Oppure bisogna gestire tabelle di hash in grado di gestire tabelle multiple. Ciò per capire che realizzare strutture dati non è banale.

ALLOCAZIONE DEI FILE

Allocazione Contigua Un file occupa blocchi contigui, il cui numero varia in base alle dimensioni del file. In questa figura si cerca di rappresentare la trasformazione da indirizzo logico a fisico. Essendo il disco suddiviso in una sequenza numerata di blocchi logici. Preso un



indirizzo 'logical address', la directory è una tabella che in sostanza dice che il file 'count' parte dal blocco 0 e ne occupa 2, il file tr parte da 14 ed è lungo 3 ecc. Supponendo di fare una read di un dato logical address. Q e R sono il numero del blocco (Q) e la posizione all'interno del blocco (R). Avendo blocchi grandi 512 bytes, divido l'indirizzo logico (LA) per la dimensione del bocco e in questo modo ottengo come quoziente il numero (indice) del blocco e come resto il displacement, da qui il nome Q e R. Questo tipo di allocazione (cioè quella contigua) è semplice da implementare e alle volte va anche bene, ma come sappiamo l'uso di blocchi contigui è causa di frammentazione esterna.

Extended Based System: uno schema di allocazione contigua, usato nei sistemi moderni. Un'estent rappresenta un insieme di blocchi contigui, ed un file occupa più extent. L'obbiettivo è quello di ridurre i problemi dell'allocazione contigua allocando non un unico grande intervallo di blocchi contigui, ma tanti intervalli contigui più piccoli. Non entriamo nei dettagli.

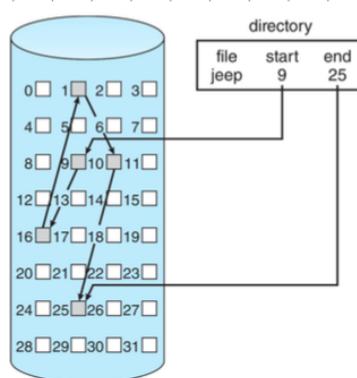
LINKED ALLOCATION Spesso i file sono letti in modo sequenziale e non ad accesso diretto. Nasce quindi l'idea di avere una lista linkata. Un file diventa quindi una lista di blocchi, che permette di non usare la contiguità. Perciò eliminiamo il problema della frammentazione esterna. Era una cosa che con i processi non potevamo fare perché l'accesso ai dati di un

block =

block	=	pointer

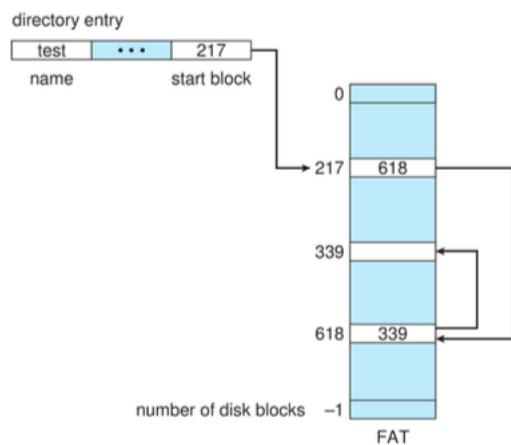
 processo non è sempre sequenziale, come succede nei file. Ogni blocco contiene quindi il puntatore al successivo ed il file termina quando in tale puntatore ho NULL. Esistono tuttavia dei problemi di affidabilità: se per un qualche problema perdo un blocco, avrò perso anche il suo puntatore al next e quindi tutti i blocchi successivi della lista. Le liste sono inoltre strutture dati lineari, quindi se per qualche motivo mi dovesse trovare a dover fare un accesso sequenziale mi ritroverei a dover accedere

al blocco n-esimo, dovrò scandire tutti i blocchi precedenti. Un altro appunto di cui tener conto



è che una parte della memoria del blocco viene mangiata dal puntatore. Perciò nel fare la traduzione da indirizzo logico, con la tecnica vista prima, bisogna tener conto che una volta ottenuti Q ed R, il displacement è R, cioè il resto, +1 ovvero la dimensione del puntatore (a patto che il puntatore sia all'inizio di ogni blocco) al blocco successivo. I dati non partono cioè dall'inizio del blocco ma sono shiftati della dimensione del puntatore e devo tenerne conto nel tradurre gli indirizzi. A questo punto, avendo una struttura a direttori, riga di un file nella struttura non contiene più indirizzo

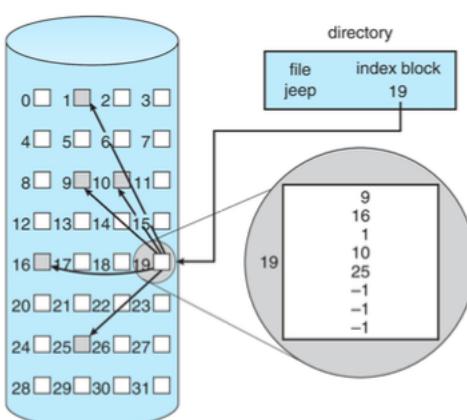
partenza e dimensione ma come blocco di partenza e blocco finale, di cui non ho necessariamente bisogno. Come già detto se in questa lista 'JEEP' perdo il blocco 1 perdo anche 10 e 25. Per questo motivo è stata inventata la **File Allocation Table (FAT)**. Il ragionamento è: se perdendo il blocco perdo non solo i dati contenuti ma soprattutto il puntatore al resto della lista allora perché non scorporare i puntatori dai blocchi di dato? Quindi rappresento i puntatori in un vettore chiamato FAT parallelo ai dati. Chiaramente non c'è alcuna magia, la FAT avrà bisogno di essere allocata e porterà via dei blocchi, se ad esempio ho un milione di blocchi da 1 KB e quindi 1 GB di blocchi di dato. Ogni blocco avrà il suo



puntatore nella FAT, che avrà quindi un milione di puntatori. Ipotizzando che un puntatore sia di 4 byte (in genere un puntatore è di 4-8 byte) la FAT peserà 4 MB, molto meno del peso dei dati. Nella FAT il puntatore 0 corrisponderà al blocco e così via. Nella directory (in figura) dico che il file 'test' inizia al blocco 217. Tale blocco non contiene il puntatore al next, ma il puntatore è nella FAT nella riga 218, che punta al blocco 618. Quindi nel vettore dei blocchi di dato (che non c'è in figura) prendo il blocco 618 e poi nella riga 618 della FAT vedo qual è il blocco successivo. Un altro vantaggio

è che essendo la FAT molto leggera, posso farne una copia e risolvere eventuali errori.

INDEXED ALLOCATION L'allocazione indicizzata tenta di sviluppare un approccio ad accesso diretto, nel tentativo di risolvere il problema delle linked list dall'alto costo di ricerca di un dato. Come accennato infatti se voglio il 100esimo blocco dovrò scandire tutti i 99 elementi precedenti e ciò costa tempo. L'allocazione mediante indici fornisce per ogni file il suo '**blocco di indici**'.

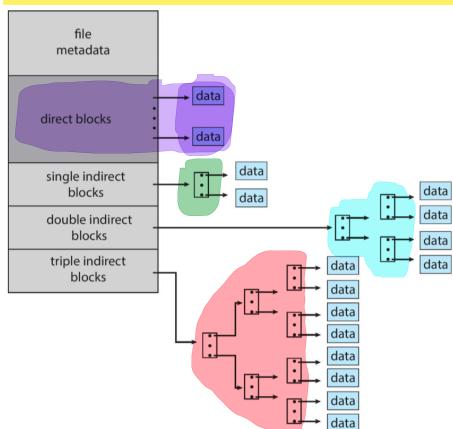


Vediamo un esempio. Supponiamo di avere il file "Jeep" sempre contenuto nei blocchi 1-9-10-16-26, che però anziché linkati internamente hanno dei puntatori in un ulteriore blocco, in questo caso il 19. Il blocco 19 infatti funziona come una tabella, se voglio il valore 0 allora sarà nel blocco 9, il primo valore nel 16 e così via. In questo modo garantisco accesso diretto.

La traduzione da indirizzo logico a fisico si fa in maniera simile all'allocazione contigua . $LA / 512 = Q$ e R , però una volta trovato l'indice Q ci posso arrivare in modo diretto. Poniamoci una domanda: e se il numero di indici che possono stare in un blocco non bastano? Nell'immagine di sopra il blocco 19 poteva contenere, ad esempio, 8 indici ed il file Jeep ne usava solo 5. E se ne servissero 10 o più? Le due soluzioni sono:

- Faccio una lista linkata di blocchi di indici e pagherò per fare una ricerca sequenziale ma solo sui blocchi di indici, che sono meno rispetto ai blocchi di dati. Ad esempio ogni blocco della tabella di indici contiene 511 indici e un puntatore alla prossima tabella di indici ed ognuno di questi indici contiene un puntatore ad un blocco di 512 (dati,bytes,parole non importa). Per convertire da indirizzo logico devo fare $LA / (511 * 512) = Q_1$ e R_1 dove Q_1 è il blocco di indici in cui andare a cercare l'indice. Per ottenere la posizione dell'indice nel blocco faccio $R_1 / 512 = Q_2$ e R_2 dove Q è la posizione dell'indice e R il displacement nel blocco.
- L'altra soluzione mira a garantire sempre un accesso diretto. Si creano quindi dei blocchi di indici a due livelli : ho quindi un blocco che contiene gli indici per arrivare ad uno fra i blocchi di indici da cui prendo il puntatore al blocco di dato. Un po' come visto con le tabelle gerarchiche nella page table. La traduzione sarà quindi $LA / (512 * 512) = Q_1$ e R_1 dove Q_1 è la riga dell'indice interno , poi $R_1 / 512 = Q_2$ e R_2 dove Q è la riga dell'indice del blocco, mentre R è il displacement.

Nei sistemi Unix e Linux si usa uno schema basato su **I-node** che è una tabella di indici



gerarchica ma sbilanciata. Vediamo meglio. La tabella è come un albero ruotato di 90°, la cui radice è il File Control Block, chiamato i-node. Se il file è piccolo ci saranno pochi livelli di indici, viceversa se è grande. Anziché quindi avere un File System dove tutti i file avranno n-livelli di indici, uso meno indici se il file è piccolo mentre di più se è grande. L'i-node, contiene oltre ai metadati del file anche i puntatori ad alcuni blocchi diretti. Ad esempio i primi 10 blocchi di dato

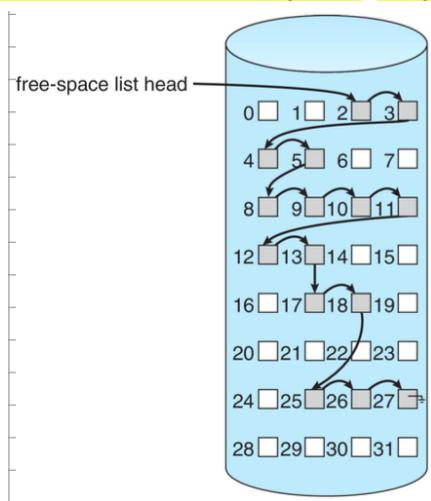
sono direttamente nel file control block. Se il file è piccolo non serve nient'altro. Se il file necessita di più blocchi allora avrà anche dall'undicesimo blocco in poi sono puntati da un blocco di indici contenuto sempre nell'i-node. Se il file consuma questo primo blocco di indici, avrà un altro blocco di indici a due livelli. Se anche questo è pieno si usa un ulteriore blocco a 3 livelli. È chiaro che la maggior parte della capienza dei blocchi di dato dal triplo livello. Se ad esempio un unico blocco può contenere 512 indici triple indirect block potrebbe indirizzare 512^3 blocchi di dato. Globalmente avrei un numero di blocchi di dato totale : $512^3 + 512^2 + 512 + 10$.

PERFORMANCE

Volendo fare un confronto di prestazioni non è così facile perché molto dipende dal tipo di accesso che si fa al file. L'allocazione contigua va bene per accessi sequenziali e accesso diretto (cioè casuale), la linked list è ottima per accesso sequenziale (detto 1000 volte). Una possibilità potrebbe essere lasciar decidere a chi crea il file in base all'uso che ne farà. Chiaro che un'allocazione indicizzata è ottima per l'accesso diretto e senza il problema della frammentazione esterna, ma è più complicata da implementare, inoltre un accesso un blocco di dato può richiedere un numero di accessi in lettura elevato, poiché dovrà leggere sia l'indice che il dato (come nella page table gerarchica). Una via di mezzo sarebbe allocare i blocchi di dato vicini in modo da entrare in uno e andare agli altri senza passare di nuovo per il blocco degli indici. Un mix contiguo e indicizzato. Un'alternativa è paralellizzare le esecuzioni.

FREE-SPACE MANAGEMENT

Abbiamo visto come si allocano i dati, vediamo come si gestiscono i blocchi o anche le pagine libere, in sostanza la **Gestione dello spazio libero**. In generale il file system mantiene una **free – space list** per tenere traccia dei blocchi/clusters disponibili o in alternativa una **bit vector** o **bit map** parallela ai blocchi che mette un 1 se il blocco è libero, 0 se è occupato. A questo punto servono algoritmi efficaci per gestirla, nello specifiche vedremo un esempio di bitmap su OS161.



Tuttavia richiede spazio aggiuntivo, ad ogni blocco serve infatti un bit aggiuntivo, che tuttavia è accettabile. Avendo ad esempio blocchi di 4 KB, cioè 2^{12} bytes ed un disco da 2^{40} bytes cioè 1 TB. Il numero di blocchi sarà $2^{40}/2^{12} = 2^{28}$ blocchi. Quindi una bitmap di 32 MegaBytes. Ottimizzando ancora le cose, compattando 4 blocchi contigui in un cluster otterò una bitmap di 8Mb. Alternativamente si può usare la **free-list**, implementata come una linked list di cui salvo solo in testa il primo blocco libero, poi ogni blocco punta il next.

Il vantaggio è che non occupa nessuna memoria aggiuntiva, lo svantaggio è che se voglio trovare un blocco libero in una posizione specifica, ad esempio voglio allocare dei dati in maniera contigua, dovrò fare la classica ricerca lineare che scandisce tutta la lista. Per ottimizzare la free list posso fare **Grouping**, cioè prendo un blocco libero e metto non solo il puntatore a next ma n puntatori ai blocchi successivi della lista. Si può fare **Counting** cioè tengo il puntatore al prossimo blocco libero e conto i blocchi liberi contigui successivi.

TRIMMING UNUSUED BLOCKS

Esiste un altro problema evidenziato dalla differenza tra gli HDD e le NVM. Mentre gli HDD consentono di sovrascrivere un blocco, gli SSD hanno bisogno di cancellare il blocco e poi riscriverlo, lo accenniamo e basta per capire che alle volte free-list e bitmap non bastano.

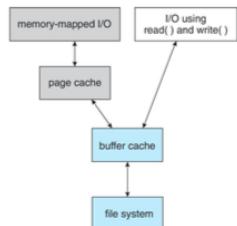
EFFICIENCY l'efficienza del file system dipende da strategie di allocazione scelte, quanto sono efficienti le ricerche sui direttori, le tabelle usate dai direttori stessi, se i dati sono a dimensioni fissa o variabile. Di nuovo lo accenniamo e basta.

PERFORMANCE

Una delle strategie che influenzano pesantemente le prestazioni è mantenere dati **vicini**, poichè più è largo ciò che si legge in un unico accesso che non implica spostamento di testine ecc. Un altro meccanismo già visto e che incide molto sulle prestazioni è il **buffer cache**, metto cioè i blocchi più usati in Ram, dal disco. Bisogna anche decidere se usare un meccanismo **Sincrono** con le I/O, cioè scrivo su I/O e attendo che finisca o **Asincrono**, cioè mando la scrittura e nel mentre faccio altro, che è più veloce ma più complicata. Come visto nel "problema di I/O". Ci sono inoltre free-behind e read-ahead, cioè letto un blocco ne leggo anche i successivi, perché nelle letture sequenziali spesso poi si accede ai blocchi successivi a quello richiesto.

PAGE CACHE

Come Un ultimo aspetto legato alle prestazioni è la **page Cache**, usata per il cosiddetto **memory mapped I/O**, si virtualizzano cioè i blocchi del disco sulle pagine di memoria.



In pratica si crea una corrispondenza tra pagine in memoria e blocchi sul disco e si crea una cache che tiene traccia di questa corrispondenza. Essendo che anche il file system ha il suo buffer cache per risolvere il problema delle I/O allora rischio di avere un doppio livello di caching, inefficiente. Nei sistemi Unix-like si realizza

una cache unificata. Questo per dire: " Attenti che in un'ottica architetturale duplicando le strategie di ottimizzazione si sputtana tutto".

RECOVERY

Nell'ottica di evitare errori che ci fanno perdere informazioni, le strategie consistono nel fare **check** di dischi per evitare eventuali errori, ma l'unico metodo è avere un **back-up** e delle **politiche di restore**. Per tenere traccia degli errori si creano dei **log structured**, cioè si salva ad ogni iterazione lo stato attuale, ciò che sto per fare e in questo modo se c'è un errore non perdo nulla, anche qui tutto solo accennato. Fine ☺

Chapter 12: I/O Systems

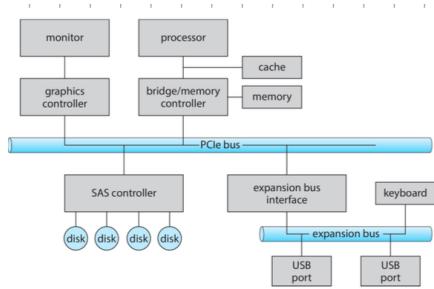
OBIETTIVO In questo capitolo vedremo come il kernel gestisce l'I/O, tenendo sempre conto che il mondo delle periferiche e spesso serve software scritto da produttori di terze parti. Vedremo l'interfaccia verso le applicazioni e le procedure per implementare in hardware ciò che il software richiede. Infine, come per gli altri capitoli, vedremo alcuni esempi di implementazioni e alcuni aspetti che influenzano le performance.

OVERVIEW L'I/O è importante in un OS perché lo fanno praticamente tutti i programmi, che sia hardware elettronico/meccanico/di rete, cito testualmente "il Pc non fa il caffè, ma la macchinetta deve poterla controllare". Oggi infatti i dispositivi di I/O sono tantissimi e vengono gestiti con operazioni diverse. Inoltre, ogni mese escono I/O più disparati. Dal punto di vista della terminologia parleremo di porte, bus e controllori mentre dal punto di vista hardware **il livello più basso è il device drivers**, cioè una componente del sistema operativo che pilota direttamente l'hardware.

I/O Hardware

Tendenzialmente distinguiamo le I/O in hardware di **Storage**, quindi memoria, **trasmissione**, come le schede di rete e **Human interface**, come mouse, tastiere ecc. Ora, ci sono alcuni concetti comuni che si riferiscono a come questi dispositivi si interfacciano con il computer:

- **Porta**: il punto fisico in cui il dispositivo si connette alla CPU;
- **Bus**: intendo il mezzo di trasporto dalla/alla periferica al sistema di elaborazione, ne esistono di vari tipi, in base alla velocità e alla loro interfaccia. I **Daisy Chain** sono meccanismi a catena in cui se dispositivo A parla a B c'è una specie di passaparola. Fra i bus ci sono i **PCI, expansion bus, SCSI**;
- **Controller (host adapter)**: è un entità attiva, cioè un microcontrollore che è sul dispositivo I/O e si interfaccia con i bus per parlare con la CPU.



Come interagiscono i dispositivi I/O con la CPU? Tali dispositivi sono visti dalla CPU come se fossero delle locazioni di memoria, ma in realtà i device driver piazzano comandi/indirizzi/dati su dei registri posti proprio dentro la periferica. Normalmente un dispositivo di I/O avrà uno **status register**, letto dalla CPU per capire lo stato del dispositivo, un **control register**, su cui viene scritto il comando da eseguire e un **data-in register** e **data-out register**.

Normalmente sono registri molto piccoli da 1-4 bytes. Tali registri vengono mappati in modo che la CPU li veda come indirizzi in memoria; quindi, i dispositivi hanno degli indirizzi e dal punto di vista assembler le CPU possono avere istruzioni specifiche per il dispositivo, quindi Load IO e Store IO o semplicemente la CPU manda una

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

load/store senza sapere se sta scrivendo/leggendo da IO. Questa è una possibile configurazione di quali sono i dispositivi che rispondono a indirizzi in I/O. Normalmente tali indirizzi hanno bisogno di meno bit rispetto alla RAM. Nell'esempio a sinistra bastano 3 cifre esadecimali quindi da 0 a 4 KB.

POLLING

Come si interagisce con un dispositivo di I/O? si fa il **polling**, un meccanismo basato su **busy-waiting**, che abbiamo già accennato. Cosa vuol dire fare polling?

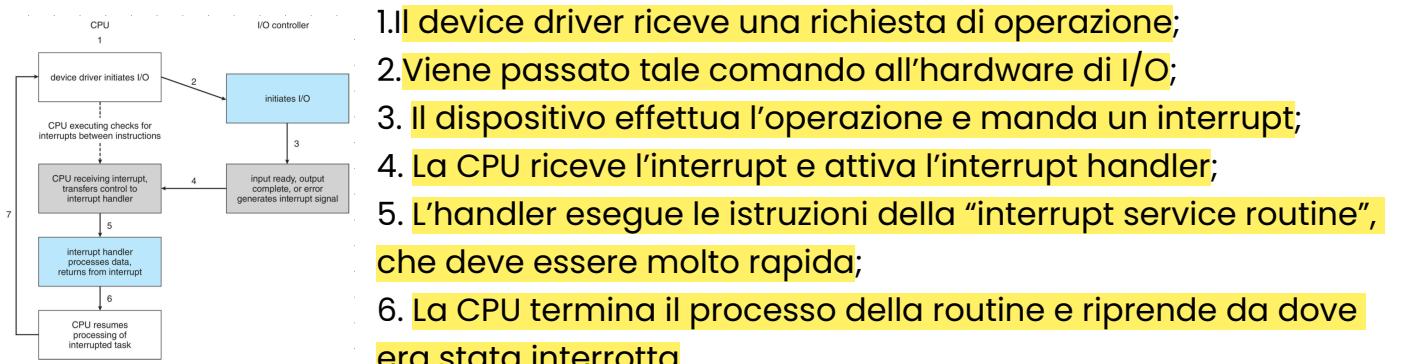
Vuol dire che per ogni dato da leggere o scrivere su I/O, in estrema sintesi, dobbiamo per la scrittura, aspettare che il dispositivo sia pronto a ricevere un dato, per la lettura, aspettare che il dato sia pronto. Per fare questo:

1. Leggo il busy bit dallo status register, fin quando non va a 0;
2. L'host mette a 1 il read o write bit (il control register) e prendiamo/scriviamo sul data in register o data out register;
3. L'host (cioè il microcontrollore) imposta l'operazione, ad esempio per una write dice "ho scritto il dato";
4. Il Controller dice sono occupato (busy bit = 1) e fa il trasferimento;
5. Il Controller pulisce il busy bit.

Tutto ciò ha una parte critica, cioè il punto 1. Poiché l'attesa potrebbe essere lunga, se la CPU decide di fare altro mentre è in busy wait potrebbe anche perdere il momento in cui il busy bit va a 0 e praticamente non farei mai ciò che deve fare sull'I/O.

INTERRUPT

La vera alternativa al polling è l'**Interrupt**. Fare polling significa volendosi svegliare alle 8 di mattina, svegliarsi ogni minuto e controllare l'ora, l'interrupt significa impostare una sveglia all'orario desiderato. Ad un certo punto la CPU riceve su un filo di ingresso detto "**Interrupt – request line**", una richiesta di interruzione attivando un protocollo specifico. L'**interrupt handler** si occupa di ricevere tali richieste che possono essere disattivate o ritardate, l'**interrupt vector**, semplicemente associa dei numeri interi a delle azioni precise per dire cosa bisogna fare arrivata un interrupt, in base a chi l'abbia mandata.



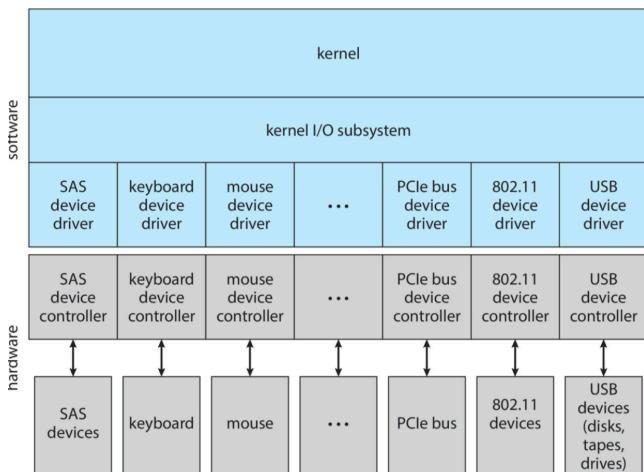
È bene tenere a mente che gli interrupt sono una forma di eccezione, usata per gestire gli errori software/hardware.

DIRECT MEMORY ACCESS: Bisogna considerare che non tutto può essere svolto dalla CPU, alcuni trasferimenti sono demandati direttamente al device di I/O. Per fare ciò si utilizza il **Direct Memory Access (DMA)**. Tale meccanismo richiede un **DMA controller**, sul dispositivo di I/O, ovvero un modulo hardware, che pilota i bus per trasferire dalla memoria dei dati. Per 'rubare' il controllo del bus il Sistema Operativo

deve fornire gli indirizzi ed il tipo di operazione e la size, esso opera un 'cycle stealing', la CPU non potrà fare uso del bus. Rimane comunque un meccanismo molto efficace.

APPLICATION I/O INTERFACE

A questo punto saliamo di un livello e dal device driver, vediamo che le applicazioni usano le system call per richiamare queste operazioni fatte dal device driver. I dispositivi possono variare in molti modi, possono essere: a flusso di caratteri singoli come la tastiera; a blocchi, come i dischi; sequenziale o ad accesso diretto; sincrono o asincrono; condivisibile o dedicato; veloce, come la rete o lento; scrittura-lettura, solo lettura o solo scrittura.



Questa classificazione viene ritrovata in quella che è l'architettura del sottosistema di I/O del kernel. In grigio ci sono le parti hardware, come il dispositivo di I/O e il device controller. Per ogni dispositivo c'è un device driver, quindi software, per gestirlo. Mettendo in evidenza alcune delle caratteristiche citate prima i dispositivi di I/O vengono raggruppati come Block I/O, character I/O, Memory – mapped file access, Network sockets.

BLOCK AND CHARACTER DEVICES

I dispositivi a blocchi sono normalmente i dischi, che supportano operazioni di read, write o seek (cioè spostarsi in quel settore). Possiamo andare in modalità Raw I/O, cioè niente file system quindi si va direttamente ai settori sul disco, o in modalità Direct I/O, cioè usando il file-system. È possibile un accesso ai file 'memory-mapped', cioè i file sul dispositivo di I/O sono mappati virtualmente come se fossero delle struct in memoria. Se invece pensiamo ai dispositivi a caratteri, quali tastiera o stampa a video, normalmente sono a disposizione delle get() o delle put(), tipicamente sequenziali. I dispositivi a blocchi usano il DMA, mentre quelli a stream no.

NETWORK DEVICES

Nei sistemi operativi sono largamente usati i dispositivi di rete. In linux, unix e Windows l'interfaccia usata per le reti è il socket, il comando più usato è il select(). Essendo oggetto di altri corsi, non le vediamo meglio 😊

CLOCKS AND TIMERS

Non tutti gli I/O sono uguali per scambiare dati, ma i programmable internal timer servono per generare segnali temporizzati.

NON BLOCKING AND ASYNCHRONOUS I/O

Introduciamo queste tipologie:

- **Bloccante**: un processo si blocca fin quando l'I/O viene completato. È molto facile da implementare, ma spesso non è sufficiente (è l'I/O sincrono).
- **Non bloccante**: se il processo fa partire l'I/O, senza fermarsi ad aspettare. L'I/O ritorna subito un risultato parziale. Può essere implementato o in automatico usando dei buffer, parcheggiando cioè i dati in una zona di memoria e faccio altro, o manualmente con i multi-thread, uno dedicato all'I/O e gli altri fanno altre operazioni. Questo tipo di I/O può tornare un conteggio dei byte scritti o letti. La 'select()' può dire se il dato è stato scritto o letto.
- **Asincrono**: di fatto è un I/O non bloccante, che prevede anche un meccanismo ben strutturato per far sì che il programma possa capire se l'I/O è terminato. Le strategie usate sono due: 1. Faccio partire l'I/O, faccio altro e quando ho finito mi metto in wait del dato dell'I/O, 2. Specifico al sistema una callback che verrà eseguita al termine dell'I/O.

VECTORED I/O

È una variante di quanto visto fino ad ora che permette di gestire un vettore di richieste di I/O anziché una singola richiesta.

KERNEL I/O SUBSYSTEM

Dal punto di vista del kernel esiste un problema di schedulazione, cioè avere più richieste di operazioni su dispositivi di I/O. Un modo per gestirle è usare una coda FIFO, altri os usano il 'fainess', cioè molto vagamente avere un criterio di suddivisione dei servizi tra i vari richiedenti. Un altro aspetto molto importante è il **buffering**, cioè quando devo trasferire dati in output da memoria a un I/O, salvo questi dati in memoria kernel prima di trasferirli al dispositivo, perché? I motivi sono come sempre variegati: ci potrebbero essere differenze di velocità tra i dispositivi A e B. Quindi anziché trasferire da A a B, trasferisco da A alla memoria, A è libero, poi da memoria a B. Così A non deve aspettare B. Un altro problema risolto tramite buffering è la differenza tra le dimensioni dei dati trasferiti da A a B, supponiamo che A scrive a blocchi di 1KB e B legge a blocchi di 4 KB, allora facciamo quattro trasferimenti da A al buffer e poi uno dal buffer a B. l'ultimo motivo è che è possibile che un processo voglia scrivere dati su disco e dopo aver fatto la richiesta di scrittura, voglia modificare i dati senza aspettare la fine della scrittura. Il buffer aiuta a garantire che se ho un duplicato capisco chi lo ha fatto e quando. Bufferizzare significa quindi ceo una copia da qualche parte del dato. Se da A copio sul buffer, la prossima volta che A vuole copiare sul buffer prima deve aspettare che il buffer si sia svuotato. Per risolvere questo problema si può usare il **double buffering**, cioè ci sono due buffer, uno su cui lo user scrive i dati, l'altro su cui il kernel processa i dati e poi si scambiano. Il **caching** è una copia ridondante di un'informazione su un dispositivo più veloce, sempre allineata rispetto all'originale. In pratica è la cache che abbiamo visto mille volte.

Lo spooling è una forma di buffering ma usato solo per dispositivi di uscita, come le stampanti. Non è nient'altro che una coda per gestire le richieste per quei dispositivi, che poi vengono eseguite una alla volta.

La **device reservation** è una gestione del dispositivo in mutua esclusione.

Gestione degli errori Il sistema operativo può cercare più volte per vedere se le cose vanno a posto da sole (spesso nelle reti). In generale c'è la possibilità di ritornare dei codici di errore quando l'I/O fallisce.

I/O protection Un processo user potrebbe involontariamente o intenzionalmente svolgere azioni 'illegali', cioè errate, e tenendo conto che le operazioni di I/O sono privilegiate perché passano dal Kernel c'è bisogno di effettuare tali operazioni mediante system calls.

Kernel data structure In pratica il kernel salva per tutti i dispositivi delle informazioni di stato in delle tabelle. Ci sono molte strutture complicate per gestire i buffer, le allocazioni e le azioni dei dispositivi, sono generalmente strutture dinamiche o in alcuni os 'object-oriented'.

Power Management Anche la gestione dell'alimentazione viene delegata al kernel, così come gestione delle temperature ecc. Android ad esempio cerca di capire come sono connessi i componenti tra loro e sono in grado di dire "non sono utilizzato spegnimi" per questioni di risparmio energetico.

- Management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot seek())
- File-system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization
- Power management of I/O devices

Sommario di ciò che gestisce il sottosistema I/O kernel

Transforming I/O requests to Hardware Operations

Vediamo quali sono i passi per eseguire un'operazione a partire dalla richiesta di I/O. Supponiamo una lettura di un file da disco per un dato processo: bisogna determinare in primis qual è il dispositivo che contiene il file, tradurre poi dal nome al dispositivo, leggere i dati dal disco in un buffer e poi darli dal buffer al processo, infine faccio return.

Streams è un esempio di comunicazione "full duplex", cioè bidirezionale. Consiste di una head che sarebbe il processo ed un driver dal lato del dispositivo. Prevede delle code di lettura e di scrittura.

Performance L'I/O impatta moltissimo sulle prestazioni di un sistema. Per migliorare tali prestazioni sarà importante ridurre il numero di context switch, il data copying e gli interrupt. Le ultime slide vengono solo commentate e non mi sembra nulla di importante. Fine.