SISTEMI OPERATIVI

by diverse fonti

1. **Introduzione 5**
   1. Revision History 5
2. **ELEMENTI DI UN CALCOLATORE 6**
   1. REGISTRI CPU 7
   2. ESECUZIONE ISTRUZIONI (fetch/execute: tipi di istruzioni) 8
      1. INTERAZIONE CPU-MEMORIA 8
      2. INTERAZIONE CPU-MODULI DI I/O 8
      3. CHIAMATA A PROCEDURE 9
         1. REGISTRI DI CPU PER LO STACK: 9
      4. I/O: 11
         1. I/O PROGRAMMATO 11
         2. INTERRUPT-DRIVEN 11
      5. INTERRUPT, HANDLER, E "NUOVO" CICLO DI ISTRUZIONE 12
         1. TIPI DI INTERRUPT 13
            1. INTERRUPT HANDLER: 13
            2. CICLO DELLE INTERRUPT: 13
            3. DIRECT MEMORY ACCESS (DMA) 16
   3. GERARCHIA DELLE MEMORIE 17
      1. MEMORIA SECONDARIA 17
      2. DISK CACHE 17
      3. RAM CACHE 18
3. **Panoramica sui sistemi operativi moderni 18**
   1. SERVIZI OFFERTI 19
   2. KERNEL 19
   3. MODALITÀ DI ESECUZIONE DI UN SISTEMA OPERATIVO (vedi sotto sotto dopo stati processi) 19
      1. EVOLUZIONE DEI SISTEMI OPERATIVI: 19
      2. I/O bound vs CPU bound 21
      3. TIME SHARING 21
      4. ASPETTI PRINCIPALI DI UN S.O. 21
      5. ARCHITETTURE MODERNE 23
4. **Processi: Dispatching, Stati, Descrizione e Controllo 23**
   1. S.O. e PROCESSI 23
   2. SYSTEM CALL 23
   3. QUANDO VIENE CREATO UN PROCESSO? 24
   4. QUANDO TERMINA UN PROCESSO? 25
   5. SCHEDULER E DISPATCHER 25
      1. IMPLEMENTAZIONE DI UN PROCESSO IN SISTEMI MULTIPROGRAMMATI 25
   6. MODELLI DEGLI STATI DI UN PROCESSO 25
      1. CASI IN CUI VA SOSPESO UN PROCESSO 28
   7. DESCRIZIONE DEI PROCESSI 29
      1. IMMAGINE DI UN PROCESSO 29
      2. P.C.B. (process control block) 29
      3. CREAZIONE DEI PROCESSI 29
      4. TABELLE 30
   8. CONTROLLO DEI PROCESSI 30
      1. MODE SWITCH 30
      2. PROCESS SWITCH (o Context Switch) 31
         1. Quando effettuare Process Switch? 31
         2. Come effettuare Process Switch? (Cosa richiede?) 31
      3. MODALITA’ DI ESECUZIONE DI UN SISTEMA OPERATIVO 31
5. **Tecniche di gestione della memoria centrale 33**
   1. GESTIONE DELLA MEMORIA 33
   2. TECNICHE DI PARTIZIONE DELLA MEMORIA: 33
      1. PARTIZIONAMENTO FISSO(SW): 33
      2. PARTIZIONAMENTO DINAMICO (SW): 34
      3. BUDDY SYSTEM(sw): 35

4. APPROCCIO LAZY CONTRO "MERGING APPENA DEALLOCHI" 36
5. PAGINAZIONE (hw) 37
1. COME METTO UN PROCESSO IN RAM: 37
6. SEGMENTAZIONE (hw) 37
1. Tavola dei segmenti e traduzione dell'indirizzo logico e fisico: 38
6. MEMORIA VIRTUALE 39
1. PAGINAZIONE + MEMORIA VIRTUALE 41
1. TECNICHE DI GESTIONE DELLA PT SE TROPPO GRANDE: 42
1. Tabella delle pagine a DUE LIVELLI gerarchiche: 42
2. Tabella di pagina invertita (IPT): 44
3. MAS E SAS: 46
4. Translation lookaside buffer (TLB): 47
1. TLB E MEMORIA CACHE DELLA RAM: 52
2. SEGMENTAZIONE + MEMORIA VIRTUALE 53
1. MEMORIA VIRTUALE con segmentazione paginata (SEGMENTAZIONE+PAGINAZIONE): 54
3. RUOLO DEL S.O. NELLA GESTIONE DELLA MEMORIA 56
1. POLITICHE DI FETCH: 56
2. POLITICHE DI POSIZIONAMENTO (PLACEMENT): 57
3. POLITICHE DI SOSTITUZIONE (REPLACEMENT): 57
1. ALGORITMI DI SOSTITUZIONE PAGINE: 57
1. Optimal Policy 57
2. Least Recently Used (LRU) 58
3. FIFO 58
4. CLOCK (second chance) 59
5. AGING (politica dell'età) 60
6. PAGE BUFFERING 60
4. GESTIONE DEL RESIDENT SET 61
1. RESIDENT SET: 61
2. WORKING SET 63
5. GESTIONE RESIDENT SET TRAMITE WORKING SET 63
1. STRATEGIA TEORICA (WS=RS): 63
2. PAGE FAULT FREQUENCY ALGORITHM: 63
6. 5.POLITICHE DI CLEANING: 65
7. 6.CONTROLLO DEL CARICO: 65
7. Scheduling a breve medio e lungo termine, algoritmi per cpu scheduling 65
1. ALGORITMI DI SCHEDULAZIONE (scheduling policies): 68
2. ALGORITMI/POLITICHE DI SCHEDULING: 68
1. FCFS (First Came First Served) 69
2. RR (Round Robin) 69
1. VRR (Virtual Round Robin): 71
3. SPN (Shortest Process Next) 71
4. SRT (Shortest Remaining Time) 72
5. HRRN (Highest Response Ratio Next) 72
6. FEEDBACK 73
3. SCHEDULING IN LINUX 74
8. SCHEDULAZIONE DEL DISCO 76
1. POLITICHE DI SCHEDULING DEL DISCO: 76
2. RAID (Redundant Array of Independent Disks) 79
1. TECNICHE UTILIZZATE: 80
2. TIPI DI RAID: 80
3. TIPI DI RICHIESTE: POSSIBILITA’ IN SCRITTURA/LETTURA: 80
4. RAID più USATI: 80
5. RAID NIDIFICATI (nested raid) =ARRAY NIDIFICATI: 82
6. RAID MENO USATI: 84
7. JBOD (just a bunch of disk): 86
8. IMPLEMENTAZIONI DEI RAID: 86

9. **CAPITOLO 12: PARTE NON FATTA 87**
10. **Unix File Management, inode, Linux VFS, ext2 .... 88**
    1. FILES MANAGEMENT 88
    2. FILE MANAGEMENT SYSTEM: 88
    3. GESTIONE DEI FILE SOTTO UNIX 89
        1. INODE 89
            1. STRUTTURA GERARCHICA 90
    4. TIPI DI FILE SYSTEM 91
11. **Makefile e debugger 93**
12. **Linking, compilazione e debugging 104**

Introduzione

Il seguente testo vuole unire tutti gli appunti degli studenti che hanno seguito il corso di Sistemi Operativi a partire dall'Anno Accademico 2009-2010 presso la Facoltà d'Ingegneria nel Corso di Studi d'Ingegneria Informatica dell'Università degli Studi di Roma Tre.
Lo spirito con il quale si vuole divulgare questo testo è quello di migliorare il suo contenuto con l'avanzare del tempo grazie al contributo che viene data da chi vuole partecipare alla realizzazione di questo testo divulgato nella rete sotto licenza Creative Commons tramite il sito http://www.danielepalladino.it/.

Revision History

Ogni volta che il file viene aggiornato (anche con un piccolo contributo) sarebbe cosa gradita inserire l'anno (inteso come Anno Accademico) e un nickname (facoltativo):

- 2009-2010: Palla, Akira, Cheruba

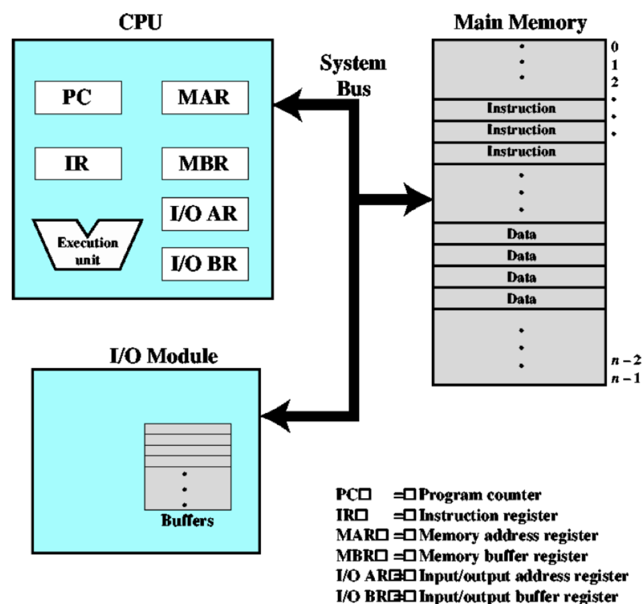
Lezione 06/10/2009

Sistema operativo: sfrutta le risorse hardware di uno o più processori per fornire servizi agli utenti, gestisce la memoria e i dispositivi di I/O.

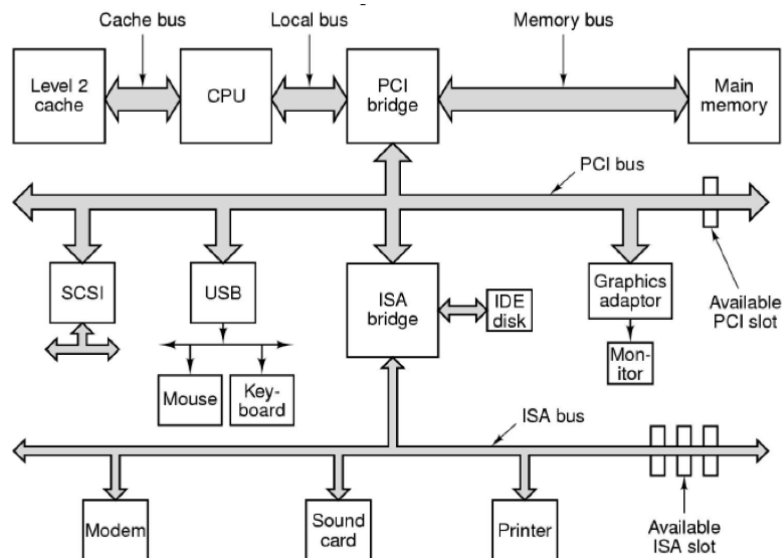
ELEMENTI DI UN CALCOLATORE

- **Processore**: elaborazione dati, controllo operazioni:
- **CPU**: effettua lo scambio di dati con la memoria, utilizzando 2 registri:
    - MAR - specifica l'indirizzo in memoria per la successiva operazione di lettura/scrittura
    - MBR - contiene i dati da scrivere in memoria o riceve i dati letti dalla memoria

  gestisce l'I/O con altri 2 registri:
    - I/O AR che specifica un particolare dispositivo di I/O
    - BR (buffer) per lo scambio dei dati tra processore e I/O
- **Memoria principale (la RAM)**: memorizza dati e programmi. Memoria primaria, volatile; modulo di memoria=locazioni definite da indirizzi numerati sequenzialmente, ciascuna contenente un # binario che può essere un dato o un istruzione
- **Moduli di I/O**: trasferimento dati tra CPU e dispositivi esterni, tra memoria e dispositivi esterni; contiene buffer temporanei
- **Interconnessione del sistema**: per comunicazione tra processori, memoria centrale, e I/O



ARCHITETTURA Pentium



REGISTRI CPU

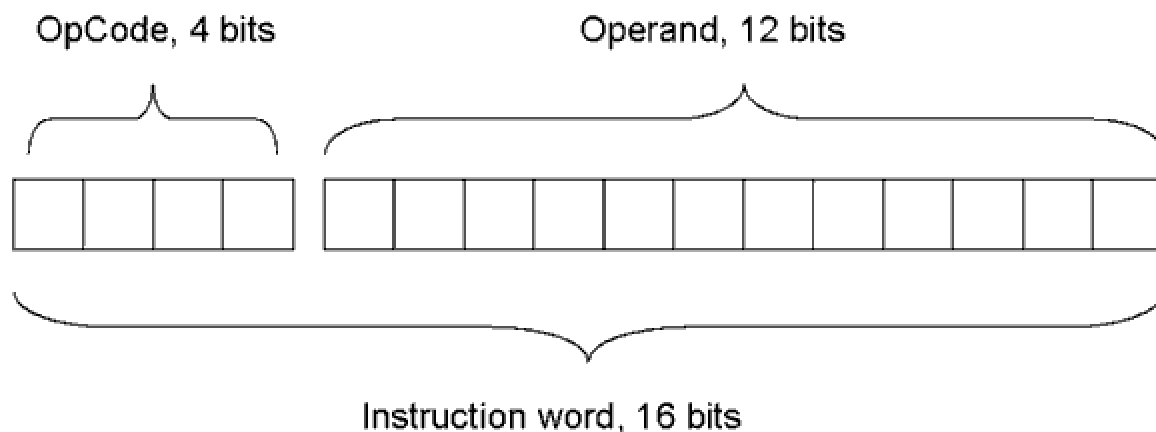
- **Registri visibili all'utente**: permettono all'utente di non dover usare memoria centrale per gli accessi che necessitano più velocità, in modo da massimizzare le prestazioni. Si dividono in:
    - **Registri dei dati**
    - **Registri degli indirizzi**: hanno gli indirizzi dei dati e delle istruzioni nella memoria principale: **index** che contiene dei valori che possono essere aggiunti ad indirizzi più specifici per ottenere l'indirizzo effettivo; **segment pointer** che riporta il numero di segmento quando la memoria è divisa in segmenti (blocchi di lunghezza variabile); **stack pointer**: registro puntatore allo stack che, nel caso di indirizzamento con stack, risiede in memoria principale e l'utente può puntarlo grazie a questo registro.
- **Registri di flag**: parzialmente visibili agli utenti, contengono bit settati dall'HW del processore in seguito al risultato di alcune operazioni.
- **Registri di controllo e di stato**: possono essere acceduti solamente dal processore per gestire l'esecuzione delle varie operazioni macchina, e dal sistema operativo che gestisce l'ordine delle operazioni che devono essere eseguite (salvataggio e ripristino dello stato della cpu passando da un processo ad un altro). Sono: MAR, MBR, I/O AR, I/O BR citati sopra, poi:
    - PC - contiene l'indirizzo della prossima istruzione da eseguire;
    - IR - contiene l'operazione attualmente in esecuzione;
    - PSW (parola di stato del programma) - registro o insieme di registri contenenti informazioni di stato, come i flag (i flags bit settati a 1 o 0 a seconda del tipo di risultato dell'operazione appena eseguita, per es: risultato positivo, negativo, zero e overflow), se le interrupt sono abilitate o meno e se sono in modalità utente o supervisore (bit di condizione);
- **Registri per la gestione delle interrupts**

ESECUZIONE ISTRUZIONI (fetch/execute: tipi di istruzioni)

INTERAZIONE CPU-MEMORIA

Eseguire un'istruzione significa eseguire il prelievo (fetch) dell'istruzione dalla memoria ed eseguirla per una volta (execute) (eseguire un programma equivale ad eseguire più operazioni ovvero eseguire più volte fetch/execute). Viene utilizzato un registro di appoggio detto accumulatore AC (registro accumulatore) dove posso caricare tutto ciò che ritengo necessario per eseguire l'operazione. Operazione salvata in memoria: 4 bit di opcode (es: MOV) e 12 bit di indirizzo (dove risiedono i parametri dell'istruzione).

Un programma che deve essere eseguito è rappresentato da una serie di istruzioni salvate in memoria che vengono eseguite una ad una. Le operazioni salvate in memoria hanno il formato della figura, dove i primi 4 bit (OPCODE) rappresentano il codice identificativo dell'operazione, mentre i restanti 12 (ADDRESS o OPERAND) rappresentano l'indirizzo dei parametri che servono all'esecuzione.



- **Fetch**: Nel PC è riportato l'indirizzo della prossima istruzione che và eseguita: questo registro viene incrementato ogni volta e permette di eseguire il flusso di operazioni del programma in ordine crescente. L'indirizzo viene acceduto e il contenuto della casella di memoria (l'istruzione) viene trasportata nel IR (=istruzione corrente, ciò che il processore deve eseguire).
- **Execute**: dai primi 4 bit dell'OPCODE il processore riconosce l'operazione da eseguire (avrà cablati i procedimenti da fare per le diverse istruzioni) mentre dai rimanenti bit di ADDRESS ricava l'indirizzo della casella dove sono contenuti i parametri che l'operazione/istruzione richiede, come ad esempio il secondo operatore di un'addizione (il primo di solito è contenuto nello stack).

INTERAZIONE CPU-MODULI DI I/O

Un modulo di I/O (es. un controller del disco) può scambiare direttamente dati con il processore, ad esempio quando un processore scrive/legge dati da esso.
DMA: I/O può scambiare dati direttamente con la memoria.
ELABORAZIONE DATI CPU: esegue operazioni aritmetiche logiche sui dati.
CONTROLLO: istruzioni che modificano le sequenza di esecuzione.

CHIAMATA A PROCEDURE

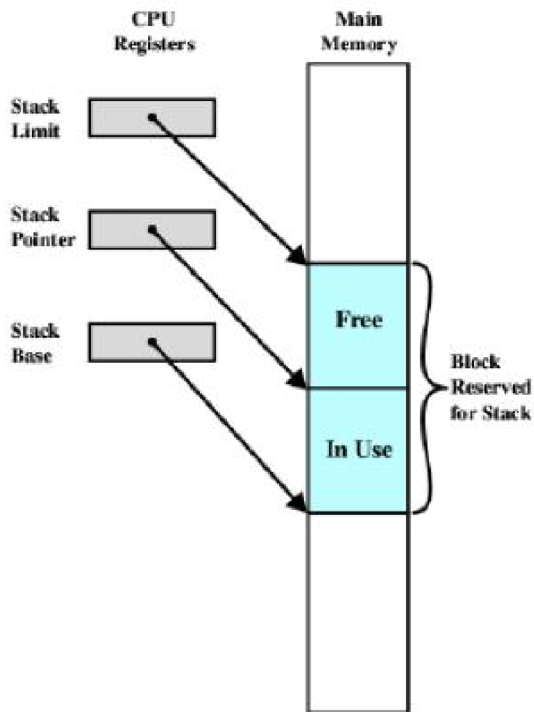
Una procedura (routine esterna) è una sequenza di istruzioni esterna al normale flusso del programma che il processore stà eseguendo in quel momento. Tutte le operazioni che vengono eseguite nel momento in cui, nel flusso normale, viene chiamata una routine esterna, permettono, una volta che questa è terminata, di riprendere dal punto in cui tutto si era interrotto.
In pratica, quando si ha una chiamata a procedura esterna, l'indirizzo dell'istruzione che stavo eseguendo al momento della chiamata deve essere salvato, deve essere eseguita la routine esterna e quindi il processore deve riprendere dall'operazione precedentemente salvata. La chiamata a procedura si differenzia dal JUMP perché al suo termine il processore continua ad eseguire l'operazione precedente.
STACK: La struttura che permette il ritorno è lo stack, una pila LIFO residente in memoria RAM, la cui testa viene sempre indicata dal registro SP (Stack Pointer) del processore. Esiste anche la possibilità di avere due registri nel processore che già contengono i primi due elementi dello stack per velocizzare le operazioni.

REGISTRI DI CPU PER LO STACK:

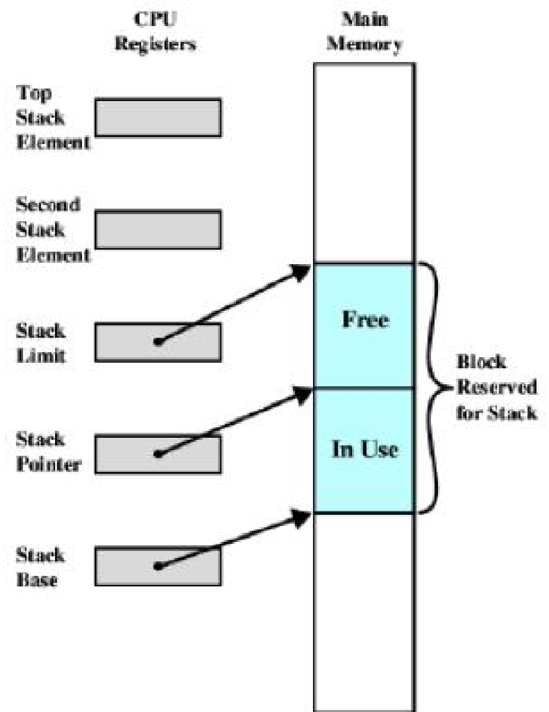
Nello heap sono presenti le strutture dati allocate durante il programma, mentre i dati contengono dati fissi. Il testo è la sequenza di istruzioni del programma. Per esempio la chiamata alla procedura print(5):

1. vedo che ho un parametro di valore 5 e lo salvo nello stack.
2. salvo nello stack l'indirizzo che tale operazione occupa nel testo del programma (return point);
3. se dichiaro variabili locali le salvo nello stack.

Nello stack infatti sono presenti: parametri, return pointer (indirizzo da cui si dovrà ripartire) e variabil locali. A fine procedura si ripulisce lo stack al contrario (variabil locali, poi return point e poi parametri).
Se durante la routine avrò un'ulteriore chiamata, ripeterò la stessa procedura impilando nello stack.



(a) All of stack in memory

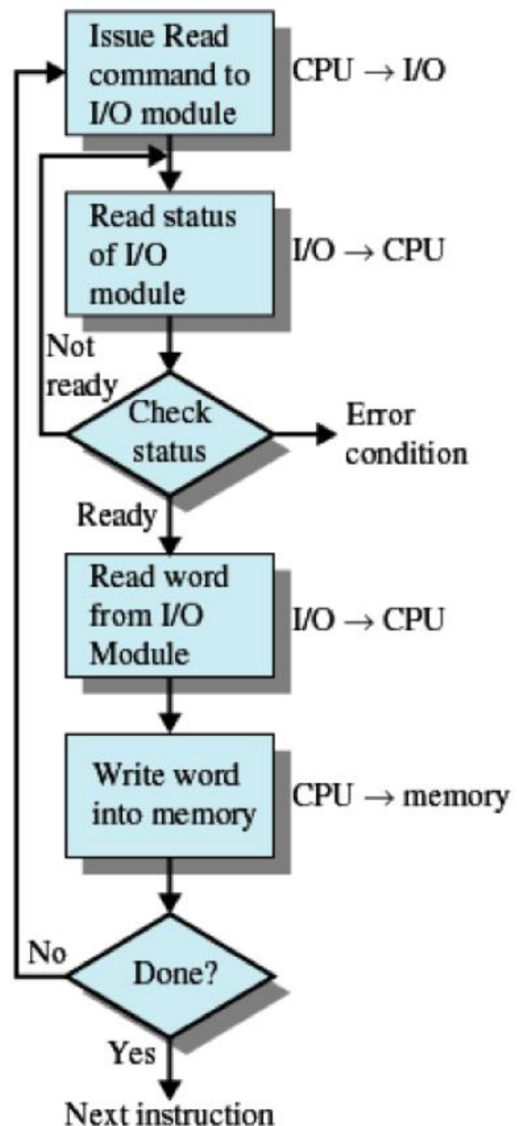


(b) Two top elements in registers

I/O:

Quando viene fatta una qualsiasi operazione di I/O, il processore comunica direttamente con la periferica con un set di operazioni del tutto simili a quelle usate per l'accesso alla memoria. La periferica può accedere direttamente alla memoria (DMA Direct Memory Access), sollevando quindi il processore dal farlo risparmiandone così il tempo operativo.Ci sono 2 approcci all'I/O:

1. **I/O PROGRAMMATO**
2. **INTERRUPT-DRIVEN**



I/O PROGRAMMATO

E' gestito con tecniche *BUSY WAITING*: il processore quando incontra un istruzione di I/O, la esegue inviando un comando all'apposito modulo di I/O e rimane in attesa che il modulo computi, facendo un ciclo che accede di continuo ad un bit di controllo sulla periferica, finché questo non riporterà che il processo è effettivamente terminato. Questo sistema è molto semplice ma tiene occupato il processore durante tutto il processo di I/O risultando quindi molto inefficiente e consigliato solamente per periferiche molto veloci che non richiedono un ciclo di attesa troppo lungo.
Le istruzioni di I/O che il processore deve eseguire sono:

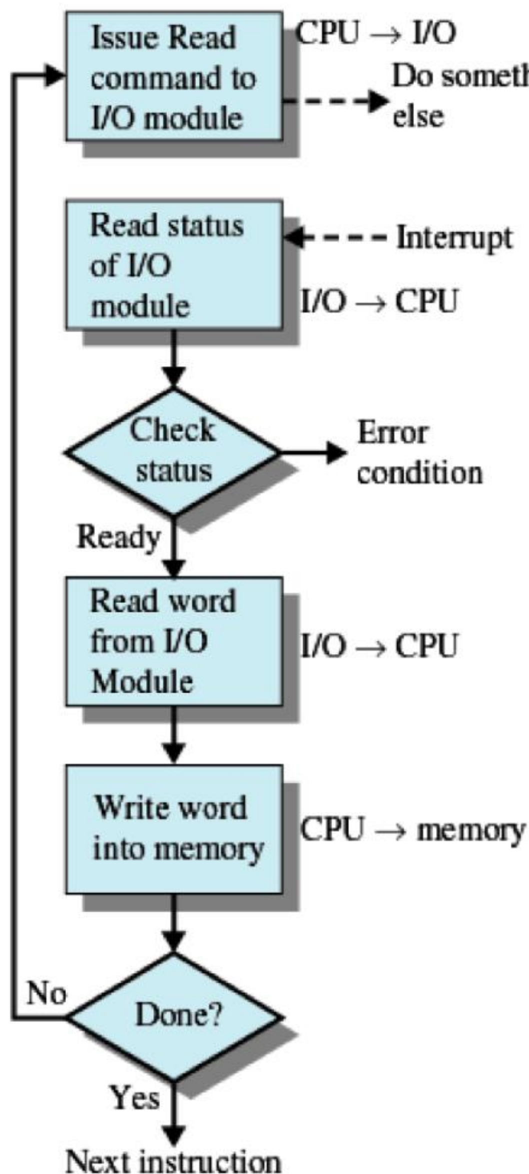
- di controllo per attivare un dispositivo esterno e determinare l'operazione da effettuare;
- di test sui moduli di I/O;
- di lettura/scrittura tra registri di CPU e dispositivi esterni;

Lezione 15/10/2009

INTERRUPT-DRIVEN

Con Busy Waiting le prestazioni decadono (la CPU attende troppo). In genere le periferiche vanno molto più lentamente del processore, quindi se il processore deve attendere che un dispositivo di I/O termini le sue operazioni prima di poter continuare ad eseguire un programma, le prestazioni ne risentono.

Le interruzioni si basano sulla filosofia che il tempo in cui il processore aspetta che la periferica termini, è meglio impiegarlo per eseguire nuovi programmi piùttosto che attendere e basta. Qui la CPU:



1. Invia comando al modulo di I/O, e non attende, bensì continua a fare altro;
2. I/O interromperà la CPU quando pronto (con un interrupt);
3. La CPU scambia dati con I/O
4. La CPU riprende il suo lavoro.

Infatti:

1. La CPU invia read al modulo;
2. La CPU salva il contesto (PC + stato registri CPU) del programma corrente che ha richiesto i dati di I/O);
3. La CPU esegue operazioni di altri processi;
4. Alla fine di ogni ciclo di istruzione la CPU controlla se sono presenti interrupt pendenti;
5. Se ci sono, la CPU salva il contesto del programma in esecuzione, esegue la routine per gestire l'interrupt (handler) e legge le parole dal modulo I/O salvandole in RAM;
6. Infine la CPU ripristina il contesto del programma che aveva chiesto tale parola (che ora è disponibile in RAM).

OSSERVAZIONI: più efficiente dello I/O programmato, ma ogni parola che va dalla RAM a I/O deve passare per la CPU; è necessaria la presenza di un buffer per le parole lette da I/O.

INTERRUPT, HANDLER, E "NUOVO" CICLO DI ISTRUZIONE

Una interrupt è una interruzione asincrona del normale flusso di esecuzione provocata da un evento esterno (timer, I/O, etc.). Sostanzialmente ogni periferica è abilitata ad interrompere il processore per qualsiasi motivo riguardante l'I/O.
L'interrupt provoca la chiamata di un "interrupt handler routine" che è parte del sistema operativo ed eseguito in modalità privilegiata. Ogni interrupt da luogo ad un mode switch, viene salvato il program counter e, poiché l'interruzione è asincrona, anche lo stato della CPU per essere ripristinato dopo l'esecuzione dell'interrupt handler routine.
Non è vero che ogni interrupt può essere associato ad un processo che ha richiesto una operazione di I/O.
ANALOGIE CON SYSTEM CALL: il flusso di esecuzione viene interrotto, si salva il program counter e almeno parte dello stato che poi verrà ripristinato; la routine chiamata viene eseguita in modalità privilegiata (mode switch).
DIFFERENZE CON SYSTEM CALL: in una system call la chiamata avviene in maniera sincrona, cioè prevista dal programmatore; lo stato della cpu può essere usato per passare i parametri alla system call e per ritornare risultati.
SYSTEM-CALL: Una system call dà sempre luogo ad un mode switch. Un processo per lanciare un nuovo processo deve eseguire una system call. Un processo per ottenere nuova memoria deve eseguire una system call. Una system call bloccante (il processo

esegue una system call e poi viene bloccato in attesa della "risposta" da parte del kernel) causa sempre un process switch se ci sono altri processi in attesa.

TIPI DI INTERRUPT

Le interruzioni possono avvenire per diversi motivi:

- I/O: generate da una periferica per segnalare il termine di un'operazione o un errore.
- Programma: un programma può generare un interrupt come risultato di un'operazione, come un'operazione macchina illegale, o un accesso ad aree di memoria per le quali non ha i permessi necessari, un overflow aritmetico o a una divisione per 0.
- Timer: generate da un timer del sistema, permettono di eseguire operazioni ripetute periodicamente.
- Errori (Hardware failure): l'hardware può lanciare un errore per cali di tensione o violazione di bit di parità.

Il TEMPO DELLA CPU quindi si divide nel seguente modo: se un processo è in esecuzione, questo viene eseguito finché non scade il tempo CPU a lui assegnato (il suo time slot) o non è presente al suo interno una richiesta di I/O. Automaticamente il sistema richiama una procedura le cui istruzioni preparano l'I/O: questa terminerà con l'attivazione della periferica che comincerà a lavorare, mentre il controllo torna al processore che intanto può eseguire altre istruzioni mentre la periferica lavora per l'I/O.

INTERRUPT HANDLER:

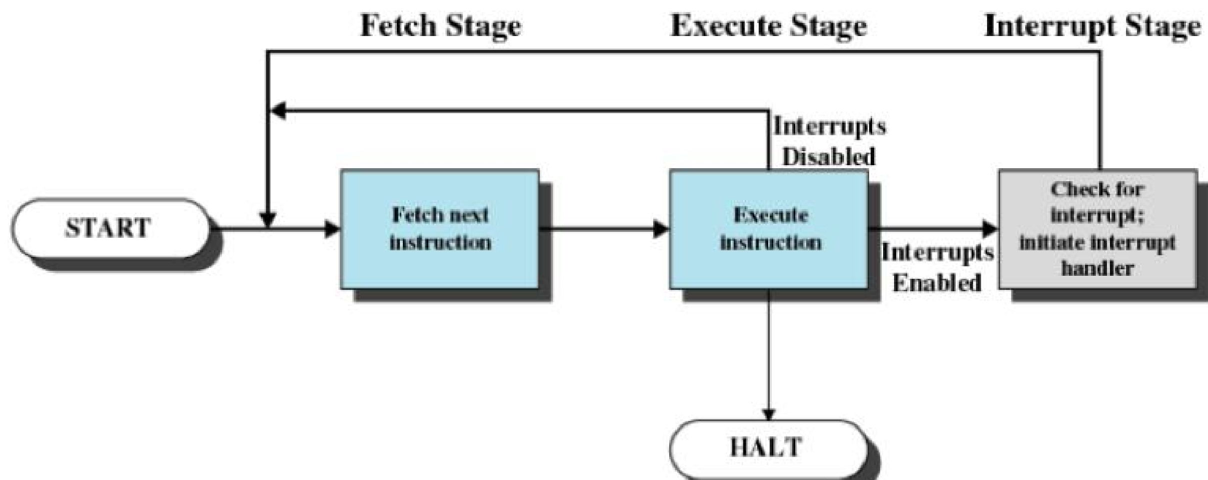
Ne può esistere uno singolo che riconosce la causa dell'interrupt, o diversi, ognuno associato ad un tipo di periferica.
Quando la periferica ha finito ed attende altri dati, emette una richiesta di interrupt, se questa viene accettata dal processore, esso sospende il programma corrente e salta ad un programma di servizio per quel particolare dispositivo di I/O, noto come interrupt handler, una serie di sotto-procedure atte a processare i dati inviati da chi ha interrotto e predisporre lo stesso dispositivo a riceverne di nuovi. Infine, dopo che il dispositivo è stato servito, riprende ad eseguire il programma originale. In tutto questo, il programma utente non deve contenere nessun codice speciale, poiché sono il processore e il s.o. ad occuparsi della gestione degli interrupts.

- UNICO HANDLER: In genere nel sistema operativo c'è un solo interrupt handler che riconosce la causa dell'interruzione (ci sono diverse linee da cui possono arrivare le interrupt e ogni linea corrisponde a un tipo di interrupt con una certa priorità, oppure c'è una sola linea con una sottolinea addizionale per determinare l'indirizzo del dispositivo) ed esegue una procedura specifica per gestirla.
- DIVERSI HANDLER: è anche possibile trovare sistemi operativi in cui ogni tipologia di interrupt è associata ad un tipo di periferica e ognuna ha un suo interrupt handler eseguito a seconda del tipo di tipologia di interrupt occorsa. In questo caso è il processore che riconosce l'interrupt e decide quale procedura di gestione eseguire.

L'interrupt Handler deve essere una procedura estremamente snella e rapida in quanto, verificandosi centinaia di interrupt al secondo deve poter essere eseguita velocemente.

CICLO DELLE INTERRUPT:

Aggiunto al ciclo dell'istruzione, permette al processore di controllare se sia avvenuta un'interruzione, controllando un apposito segnale.



Gli interrupt non possono interrompere l’esecuzione di un’istruzione del processore in quanto questa non può essere ripresa dal punto in cui era stata interrotta (quindi la CPU controlla la presenza di interrupt alla fine del normale ciclo fetch+execute dell’istruzione corrente). Il processore verifica la presenza di interrupt in coda alla fine di ogni esecuzione di istruzione prima di passare alla successiva.
ELABORAZIONE DELLE INTERRUPT: Quando un dispositivo completa un operazione di I/O:

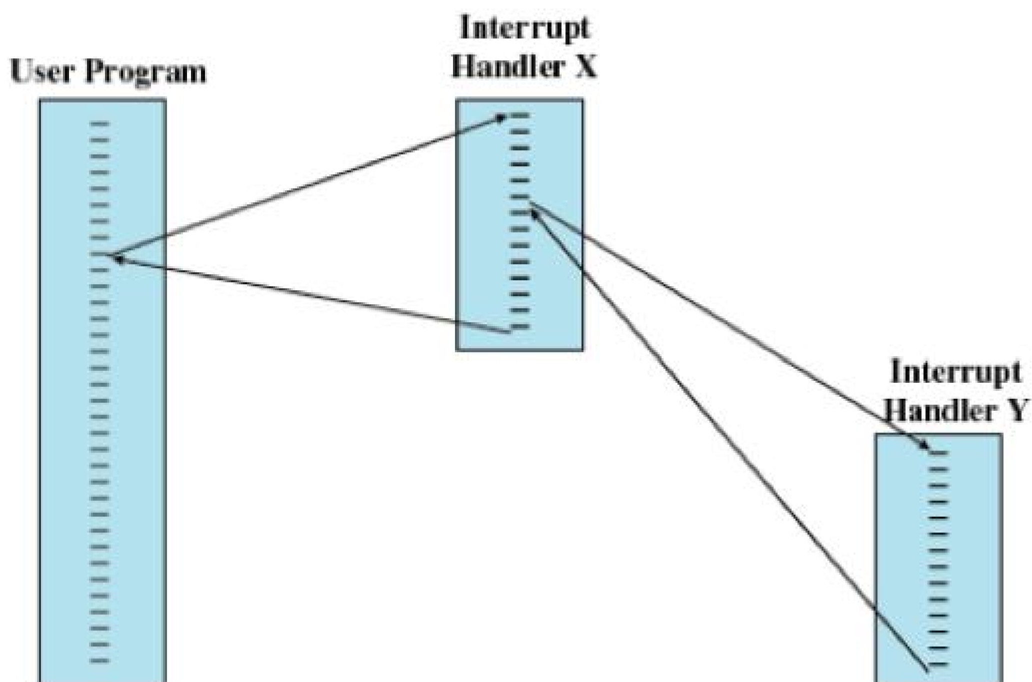
1. Il dispositivo invia il segnale di interrupt al processore;
2. Il processore termina l'istruzione corrente e poi risponde alla richiesta;
3. Il processore controlla che si sia verificata una interrupt e manda il segnale di ack al dispositivo che l'ha generata. L'ack permette al dispositivo di rimuovere il suo segnale di interrupt (sono presenti linee tra dispositivo e CPU su cui inviare segnali di interrupt).
4. Il processore si prepara a trasferire il controllo alla routine esterna degli interrupt (l'handler); salva nello stack le info del programma corrente per poterlo poi riprendere dopo (PSW,PC).
5. Il processore carica PC con l'indirizzo dell'handler, che risponderà a questa interruzione. Dopo il caricamento del PC, ricomincia il ciclo di fetch. Il controllo viene quindi trasferito all'handler.
6. PC e PSW sono salvati nello stack; l'handler salva anche i registri del processore che probabilmente cambierà per gestire l'interrupt;
7. L'handler elabora l'interrupt;
8. L'handler ripristina i registri;
9. L'handler ripristina PSW e PC dallo stack;
10. Il processore esegue l'istruzione successiva del programma che era stato interrotto dall'interrupt.

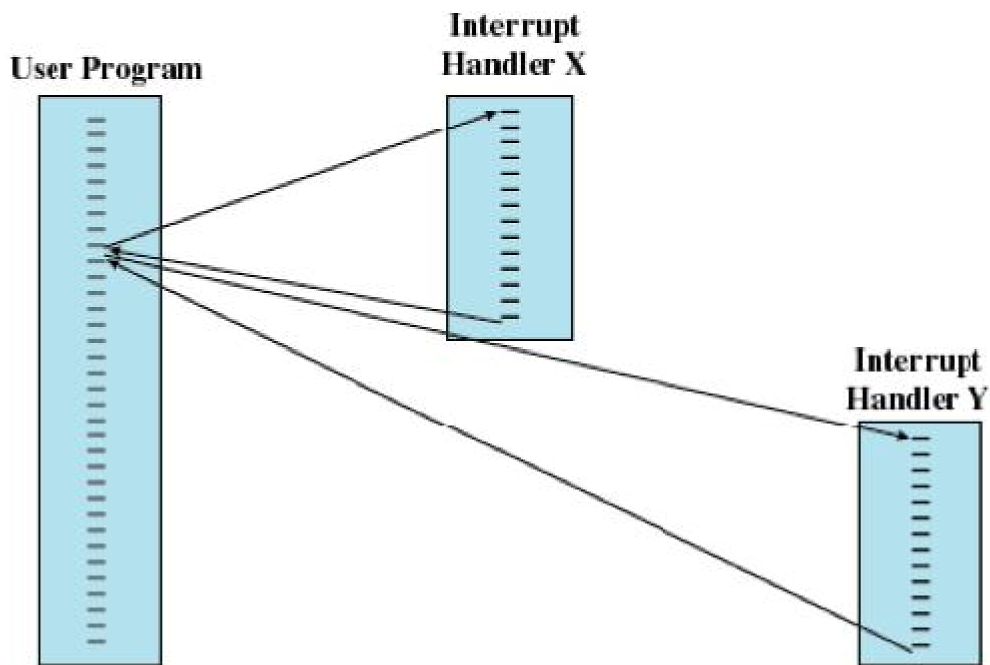
PROCESSAMENTO HW E SW DELLE INTERRUPT: nel processamento vengono coinvolti sia hw che sw: in hw, al processore arriva l'interrupt dalla periferica che l'ha chiamata: il processore finisce l'istruzione corrente, vede il tipo di interrupt, salva PC e PSW nello stack, carica nuovo PC con l'indirizzo della relativa routine da eseguire, e la CPU salta a tale IHR; in sw viene salvato lo stato del processo, viene gestito l'interrupt e viene ripristinato lo stato del processo. Non è possibile fare tutto in sw perchè la PSW (stato) cambia, potrei quindi salvare anche i registri ma "costa" meno fare delle cose in hw.
HARDWARE: il dispositivo di I/O manda un interrupt, la CPU finisce l'istruzione corrente quindi il processore manda una ack "interrupt ricevuta" al dispositivo. Il processore salva PSW e PC nello stack di controllo, la CPU carica indirizzo handler nel PC
SOFTWARE: l'handler salva lo stato processore (registri), poi elabora interrupt e alla fine ripristina lo stato della CPU. Infine ripristina anche PSW e PC.

INTERRUPT MULTIPLE: Potrebbe accadere che mentre viene eseguita una routine di gestione di un interrupt si presenti una nuova interrupt.
La situazione può essere affrontata in 2 modi:

1. disabling interrupts: Mentre eseguo una routine di interrupt le disabilito, e una volta terminata, prima di riprendere il vecchio processo, controllo che non ce ne siano di altre in coda. Questo metodo però non tiene conto delle priorità, infatti se ho una periferica che produce input a raffica, e non la servo immediatamente, rischio di perdere dei dati.
2. priority-based: questo metodo prevede che una interrupt A di una periferica con priorità più alta possa interrompere la routine B di una periferica meno prioritaria, questa verrà quindi ripresa al termine di A.

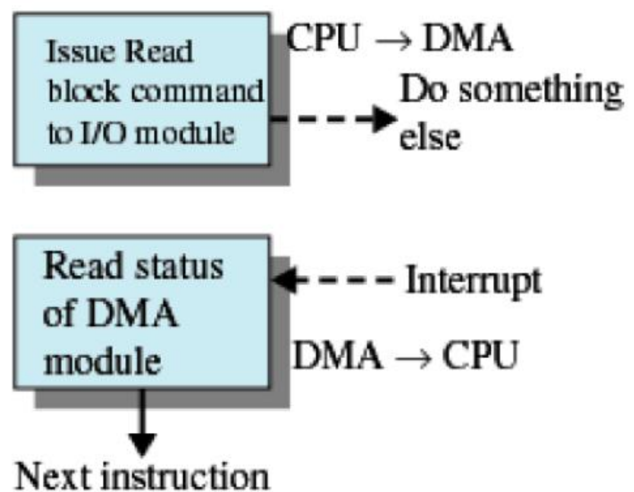
Se si deve eseguire una operazione di output su di una periferica momentaneamente occupata, allora la routine di gestione dell'output che invoco mi fa attendere che questa si liberi. Anche in questo caso però si guadagna rispetto al busy waiting grazie al tempo che le periferica passa ad elaborare i dati che non viene mai sprecato ad attendere.





MULTIPROGRAMMAZIONE: Il processore può rimanere inattivo quando il tempo richiesto per completare un operazione di I/O è maggiore di quello per eseguire il codice tra 2 successive chiamate di I/O. La multiprogrammazione risolve il problema in questo modo: se il processore esegue più programmi, rispetto a prima cambierà che nel momento in cui l'handler ripassa il controllo al processore, questo riprenderà l'esecuzione del programma interrotto dall'interrupt solo e soltanto se questo ha priorità maggiore rispetto agli altri programmi in coda.

DIRECT MEMORY ACCESS (DMA)



Nonostante l'efficienza del meccanismo di interrupt, in un computer sono presenti moltissimi dispositivi che possono generare altrettante interruzioni al microsecondo. Nel caso di dispositivi di lettura, questi generano caratteri in input a grande velocità accompagnate da interrupt e la richiesta alla CPU di trasferire tali caratteri in memoria. In questo modo il tempo perso per le operazioni di I/O da parte del processore diventa rilevante. Quindi spesso ogni dispositivo viene abilitato al DMA, in pratica un dispositivo unico collegato al bus, o una serie di dispositivi inclusi nei dispositivi, che permettono agli stessi dispositivi di effettuare accesso diretto alla memoria e gestire quindi il flusso di dati dalla memoria del computer al loro buffer di I/O, sollevandone quindi la CPU. Quando arriva un'interrupt per l'input, la CPU delegherà il tutto al modulo DMA (o al modulo del dispositivo se ne è provvisto) comunicandogli l'indirizzo del dispositivo da servire, la porzione di memoria RAM dedicata all'operazione (in cui DMA scriverà/leggerà parole scambiate col dispositivo di I/O) e il tipo di operazione da fare. Nel caso quindi del flusso di caratteri, il modulo DMA si occuperà di trasferire un

carattere alla volta nella memoria e lancerà un interrupt solo quando avrà finito. Il modulo DMA per il trasferimento dati deve usare il bus per trasferire i dati, entrando quindi in concorrenza con la CPU. La CPU viene rallentata in quanto se trova il bus occupato, deve attendere che la DMA dei dispositivi termini, questo però è nulla rispetto agli interrupt che dovrebbe sopportare in mancanza di DMA, inoltre la CPU accederà alla memoria il meno possibile, in quanto la memoria centrale è molto più lenta della CPU.

GERARCHIA DELLE MEMORIE

Scendendo nella gerarchia diminuisce il costo per bit, aumenta la capacità, aumenta tempo di accesso, diminuisce la frequenza di accesso alla memoria da parte del processore (piramide).

- INBOARD MEMORY: alta velocità di accesso ma costi elevati.
    - Registers: stessa tecnologia dei processori, molto veloce e si trovano nei processori.
    - Cache: memoria ad accesso rapido, molto efficiente per eseguire confronti (per trovare dati). Ne esistono anche diversi livelli.
    - Main memory: RAM.
- OUTBOARD STORAGE: grande capacità, basso costo per bit ma minore velocità di accesso. CD, Dvd, dischi magnetici. Questo tipo di storage può essere gestito dal software, in quanto la lettura da disco è molto più lenta rispetto alla lettura da RAM.
- OFF-LINE STORAGE: memorie su nastri magnetici, mo, worm basate su località spaziale (devo leggere cose a ciclo nell'arco di poco tempo, come le audiocassette) o temporale (devo leggere celle di memoria contigue come gli array).

MEMORIA SECONDARIA

Possono essere di diversi tipi e servono a stoccare i dati di natura statica, la loro caratteristica principale è di non essere di tipo volatile (non hanno bisogno di alimentazione per mantenere i dati) e di essere economiche, al contrario di memorie più veloci e costose. Possono essere: memoria ausiliaria, memoria di stoccaggio di massa o memoria esterna, dischi portatili, pen drives.

DISK CACHE

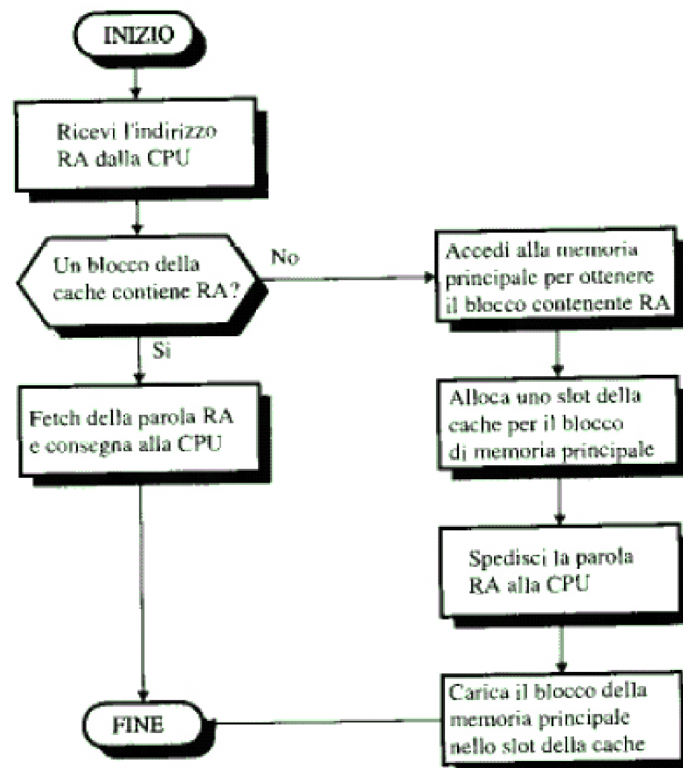
La Disk Cache è quella porzione di memoria RAM in cui il computer immagazzina le informazioni più frequentemente accedute su disco. Poichè l'accesso alla memoria primaria (RAM) è più rapido di quello ad un disco, utilizzando una cache (=porzione di ram) si sveltiscono le operazioni. La cache può essere di tipo hardware (presente sul controller del disco rigido) oppure software (cioè creata da un programma utilizzando parte della memoria principale).
Il driver della Disk Cache immagazzina i dati entrati più di recente nella RAM. Quando un programma ha bisogno di far entrare altri dati, il sistema operativo per prima cosa verifica se i dati si trovano già nella cache prima di leggerli direttamente dal disco. Poichè i computer possono accedere ai dati attraverso le RAM più velocemente rispetto a qualsiasi altro sistema di immagazzinamento, il disk caching è in grado di incrementare in maniera significativa le performance.
Un meccanismo analogo è quello della memoria virtuale, che però mette dati poco utilizzati della memoria centrale su disco.

RAM CACHE

Contribuisce alla gestione della memoria. Risiede tra memoria principale RAM e processore, è piccola e veloce e serve a limitare i tempi di accesso che di norma ci sono poiché il processore è veloce ad accedere ai suoi registri mentre la memoria principale richiede tempi di accesso maggiori. La cache contiene una copia di alcuni blocchi della RAM, e il processore prima di accedere alla RAM controlla che il dato che cerca non sia già in cache: se è presente lo prende con tempi inferiori altrimenti lo prende dalla memoria e viene copiato il blocco nella cache per possibili accessi futuri.

PRINCIPIO DI LOCALITA': Se il processore accede ad una certa informazione 'i', nelle prossime istruzioni o riaccede ad 'i' od a 'i+n' o 'i-n'. Nella cache viene quindi copiato il blocco (i-n, i+n). E' una memoria di tipo associativo, quindi veloce nei confronti, dove vengono salvate delle porzioni di disco accedute di recente. Essendo una memoria di tipo associativo, è composta da una serie di righe (come una tabella) e in cui la prima colonna è un identificatore della porzione di memoria conservata.

LETTURA DELLA CACHE:

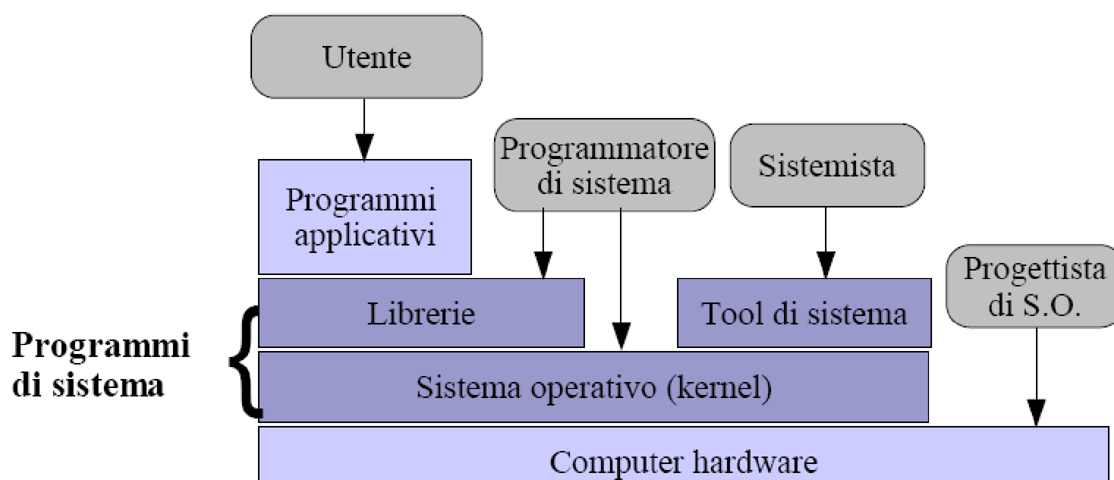


Panoramica sui sistemi operativi moderni

SISTEMA OPERATIVO: Programma che controlla l'esecuzione di programmi applicativi e agisce come interfaccia tra utente e hw del computer, avendo 3 obiettivi di fondo:

1. Convenienza
2. Efficienza nell'utilizzo delle risorse
3. Capacità di evolversi.

LIVELLI:



RISORSA: Tutto quello che serve per eseguire un programma:
- CPU time
- I/O devices
- Memory
- Executable code

SERVIZI OFFERTI

- Creazione dei programmi
- Esecuzione dei programmi
- Accesso a dispositivi di I/O
- Accesso controllato ai file
- Accesso al sistema
- Rilevazione di errori e risposta
- Contabilità
- Gestione risorse del computer e controllo delle funzioni di base
- Direzione del processore nell'utilizzo delle risorse di sistema nella temporizzazione dell'esecuzione dei programmi.

KERNEL

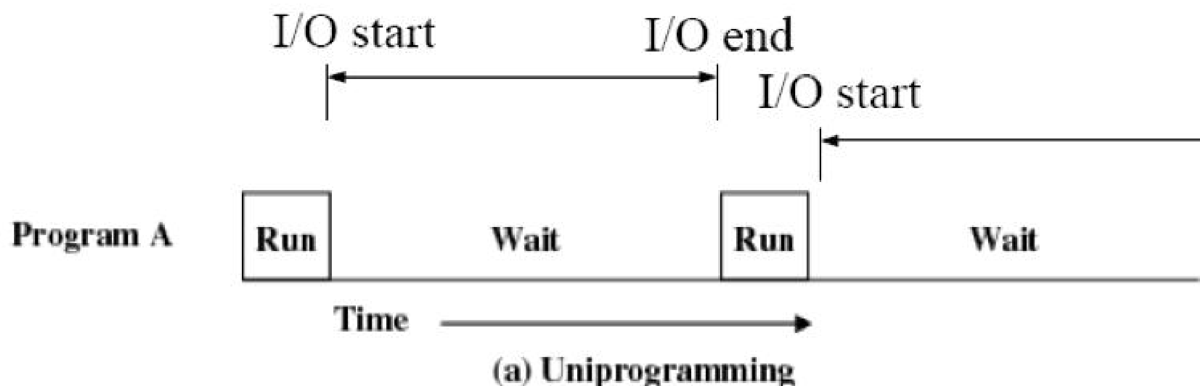
Porzione del sistema operativo che sta in memoria principale; contiene le più comuni funzioni usate.

MODALITÀ DI ESECUZIONE DI UN SISTEMA OPERATIVO (vedi sotto sotto dopo stati processi)

EVOLUZIONE DEI SISTEMI OPERATIVI:

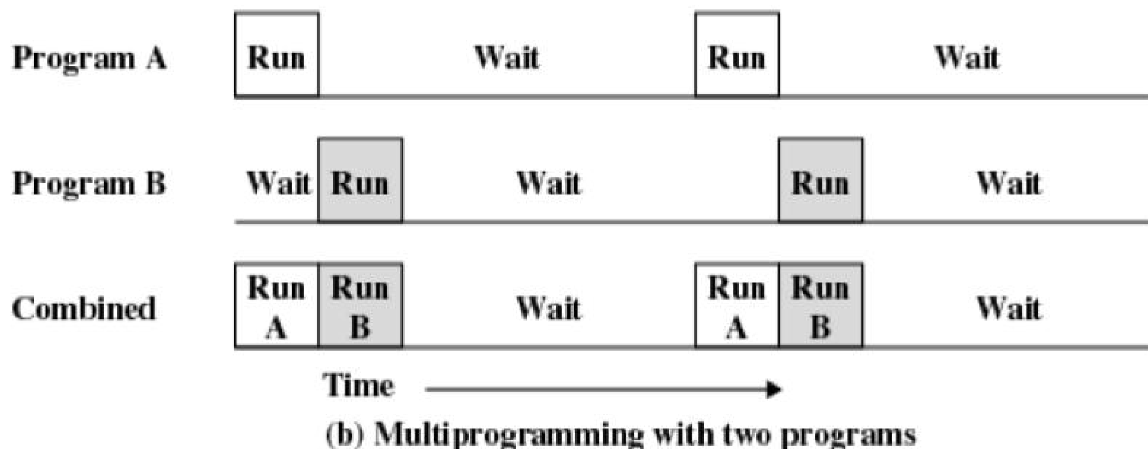
1. **batch monoprogrammati**: L'utente non ha più accesso diretto alla macchina (prima, con l'elaborazione seriale era così: ogni utente aveva un quanto di tempo prenotato di utilizzo della macchina, e gli utenti potevano lavorare uno dopo

l'altro). Il processore deve attendere tantissimo tempo per le operazioni di I/O, poichè i dispositivi di I/O sono molto più lenti di lui, e quando si eseguono i programmi (job) in modo sequenziale il processore rimane occupato in attesa, prima di poter eseguire il "prossimo" job.

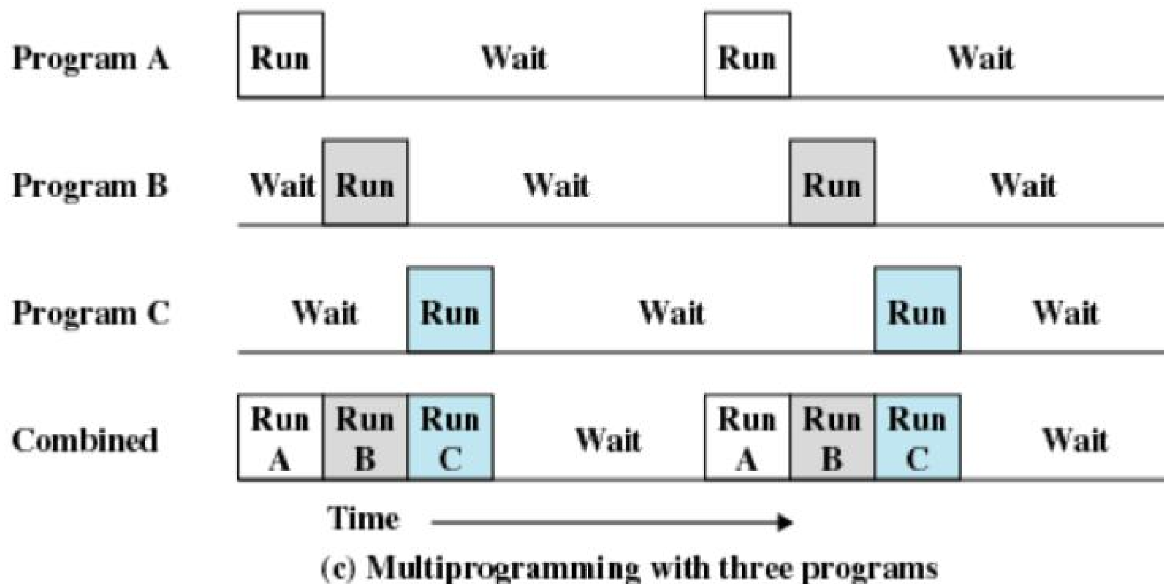


(a) Uniprogramming

1. **batch multiprogrammati**: Ha come obiettivo massimizzare l'uso del processore e i dispositivi di I/O mantendendoli il più possibile simultaneamente occupati; il sistema operativo preleva le istruzioni dal linguaggio di controllo dei job, ovvero sono fornite insieme ai job. In risposta a segnali di operazioni di I/O terminate, il processore passa alternativamente fra il processo in memoria principale. Noto il fatto che ci deve essere memoria sufficiente a contenere sistema operativo (monitor) e programmi, ora il processore, quando un job deve aspettare un operazione di I/O, può passare ad un altro job, che molto probabilmente non è anch'esso in attesa di I/O. Con due programmi in memoria, uno farà I/O burst, l'altro CPU burst ottimizzando quindi l'uso della CPU. In questi sistemi quindi, ci sono più processi in memoria, in attesa di essere eseguiti dal processore quando un altro processo finisce l'I/O: serve un meccanismo di GESTIONE DELLA MEMORIA e di SCHEDULAZIONE DEI PROCESSI visto che và deciso quale processo ,tra i tanti in attesa, debba essere eseguito prima o dopo.



(b) Multiprogramming with two programs



(c) Multiprogramming with three programs

I/O bound vs CPU bound

Tratto da Wikipedia:
Si definiscono cpu-bound i processi che sfruttano pesantemente le risorse computazionali del processore, ma non richiedono servizi di ingresso/uscita dati al sistema operativo in quantità rilevanti. È in contrapposizione a IO-bound.
Un classico esempio di tali processi sono i programmi di calcolo matematico, i quali necessitano spesso di un'enorme potenza di calcolo, ma sfruttano l'I/O solo all'inizio della loro vita (per caricare gli input) ed alla fine di essa (per produrre gli output).
La differenza tra CPU-bound e IO-bound è rilevante nell'ambito degli scheduler, in quanto diversi algoritmi di scheduling possono privilegiare oltremodo i programmi CPU-bound (specialmente quelli non-preemptive) portando a starvation di altri processi (per converso, scheduler che interrompono troppo frequentemente portano a privilegiare i task IO-bound, i quali sarebbero comunque interrotti per la maggior parte del ciclo di vita in attesa delle periferiche).

TIME SHARING

Ha come obiettivo minimizzare il tempo di risposta alle necessità dell'utente: più utenti usano il sistema contemporaneamente di modo da ridurre i costi. Il sistema operativo preleva le istruzioni dai comandi inseriti da terminale. Qui più utenti hanno accesso simultaneo attraverso dei terminali, mentre il s.o. alterna l'esecuzione di ciascun programma utente per un periodo breve, detto quanto di elaborazione. Entra in gioco il concetto di utente interattivo, ovvero che interagisce direttamente con la macchina.
PROBLEMI: non fare interferire job tra loro, autorizzazioni negli accessi al file system in caso di più utenti.

ASPETTI PRINCIPALI DI UN S.O.

1. GESTIONE DEI PROCESSI
2. GESTIONE DELLA MEMORIA (Virtuale): memoria virtuale, paging, segmentazione, rilocazione, allocazione.
3. PROTEZIONE DELL'INFO E SICUREZZA: sicurezza: controllo dell'accesso degli utenti al sistema; controllo del flusso di informazioni nel sistema verso gli utenti; certificazione delle politiche di sicurezza adottate; disponibilità del sistema dopo interruzioni; confidenzialità: accesso a risorse solo se l'utente è autorizzato;

integrità dei dati: protezione da modifice non autorizzate; autenticazione degli utenti.

4. SCHEDULING E GESTIONE RISORSE: fairness=equità tra processi che competono alla stessa risorsa; tempo di risposta differenziale. Differenziazione dei diversi tipi di job; efficienza nello sfruttare risorse: max thoroughpunt, minimo tempo di risposta. Elementi dello scheduling:
    1. Code gestite dal sistema operativo: almeno una per ogni risorsa;
    2. CPU: code a breve termine hanno processi che risiedono in memoria principale e che sono pronti per l'esecuzione; code a lungo termine con processi nuovi in attesa di utilizzare il sistema.
5. Struttura del sistema: (vedi immagine sottostante) Livelli: 1-4: hardware; 5-11: sistema operativo; 13: user interface.

Tabella 2.4 *Architettura di un sistema operativo gerarchico*

Livello	Nome	Oggetti	Operazioni di esempio
13	Shell	Ambiente di programmazione utente	Comandi nel linguaggio di shell
12	Processi utente	Processi utente	Quit, kill, suspend, resume
11	Directory	Directory	Create, destroy, attach, detach, search, list
10	Dispositivi	Dispositivi esterni, come stampanti, schermi e tastiere	Create, destroy, open, close, read, write
9	File system	File	Create, destroy, open, close, read, write
8	Comunicazioni	Pipe	Create, destroy, open, close, read, write
7	Memoria virtuale	Segmenti, pagine	Read, write, fetch
6	Memoria secondaria locale	Blocchi di dati, canali dei dispositivi	Read, write, allocate, free
5	Processi primitivi	Processi primitivi, semafori, lista dei processi pronti	Suspend, resume, wait signal
4	**Interruzioni**	**Programmi per la gestione delle interruzioni**	**Invoke, mask, unmask, retry**
3	**Procedure**	**Procedure, stack delle chiamate, display**	**Mark stack, call, return**
2	**Insieme delle istruzioni**	**Stack di valutazione, interprete di microprogramma, dati scalari e array**	**Load, store, add, subtract, branch**
1	**Circuiti elettronici**	**Registri, porte, bus, ecc.**	**Clear, transfer, ectivate, complement**

ARCHITETTURE MODERNE

- Microkernel: assegna al kernel solo poche funzioni essenziali, come gestione dei diversi spazi di indirizzamento, comunicazione tra processi (IPC) e schedulazione di base. gli altri servizi del s.o. sono forniti da processi (server) eseguiti in modalità utente.
- Multithread: tecnica con cui un processo, eseguendo un'applicazione, viene suddiviso in linee di esecuzione separate (thread), eseguibili simultaneamente. Thread: unità di allocazione del lavoro; processo: collezione di uno o più thread e delle risorse di sistema associate (memoria, file aperti e dispositivi) ovvero un programma in esecuzione.
- Multiprocessore simmetrico (SMP): massimo di efficienza e affidabilità (condivisione della RAM tra le CPU).

Processi: Dispatching, Stati, Descrizione e Controllo

PROCESSI=PROCESSO E PCB (Process Control Block)

Un processo (job) è:
- Un programma in esecuzione;
- Un'istanza di un programma in esecuzione;
- L'entità che può essere assegnata ed eseguita su un processore;
- Un'unità di attività caratterizzata dall'esecuzione di una sequenza di istruzioni e abbinata ad un insieme di risorse di sistema;

Un processo è caratterizzato dal suo CONTESTO DI ESECUZIONE (PCB):
- Identificativo;
- Stato;
- Priorità;
- Puntatori alla memoria;
- Informazioni di I/O;
- Informazioni di accounting;
- Dati presenti nei registri del processore (context data) e contesto di esecuzione del programma (stato CPU, registri).

S.O. e PROCESSI

Gestisce l'esecuzione dei processi massimizzando l'utilizzo della CPU, e minimizzando il tempo di risposta (molto importante), alloca risorse necessarie al processo e supporta la comunicazione tra processi e la creazione di processi utente.

SYSTEM CALL

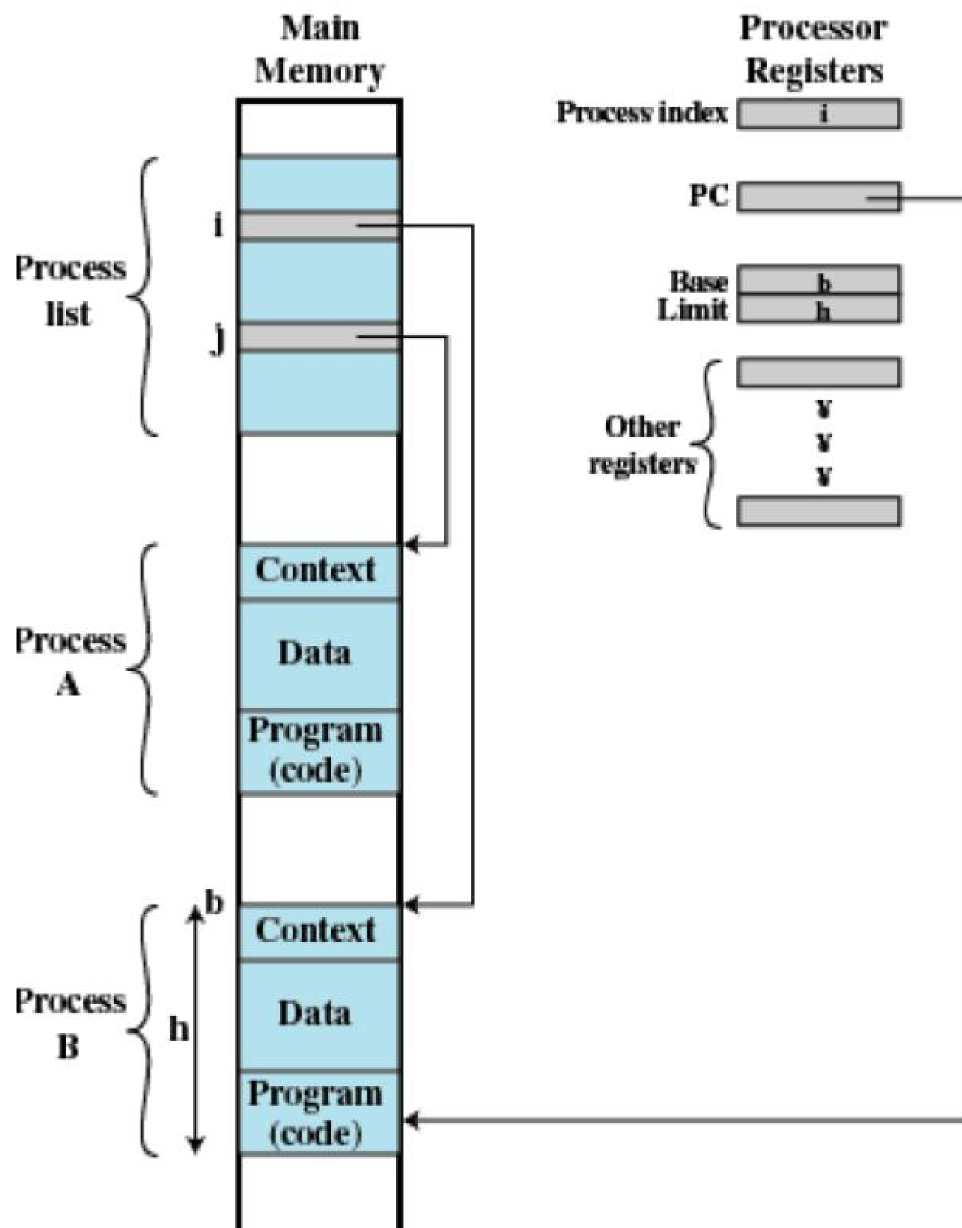
I processi fanno chiamate di sistema al kernel che, in kernel mode, comunicherà col dispositivo di I/O. Sono fatte dai processi alle CPU per chiedere ad esempio operazioni di I/O.
La CPU accede al buffer del dispositivo se e solo se non contiene già i dati allora contettarà il modulo del dispositivo. In questo caso per il tempo che il dispositivo lavora sull'I/O (lettura/scrittura), la CPU congela il processo e si dedica ad altri processi,

successivamente I/O eseguirà una interrupt.
Nota: Una system call non è un interrupt!

QUANDO VIENE CREATO UN PROCESSO?

I processi sono creati:

1. Dal sistema operativo per fornire un servizio: ad esempio la stampa (utente non aspetta la creazione del processo);
2. Nuovo processo batch: operazioni pianificate in windows. Processi periodici in linux; quando la CPU non ha niente da fare li esegue;
3. Logon: l'utente si collega al sistema (creazione explorer in windows e creazione shell in linux);
4. Dal processo padre: con la fork(..). In linux è fork (che crea un processo figlio con un suo PCB indipendente) + exec



QUANDO TERMINA UN PROCESSO?

Un processo termina:
1. Normalmente: quando termina da solo fa system call exit al sistema operativo;
2. Se supera il tempo totale di esecuzione;
3. Per memoria finita: processo vuole mememoria che non è disponibile;
4. Per Limiti di memoria violati: se accede ad aree di memoria di cui non ha i permessi;
5. Per errori aritmetici: divisione per zero;
6. Per tempo scaduto: se attende troppo un evento;
7. Per operazione I/O fallita: file non trovati;
8. Per istruzione non valida;
9. Se vuole eseguire istruzioni eseguibili solo dal sistema operativo;
10. Per uso improprio dei dati: dati sbagliati;
11. Per opera dell utente o del sistema operativo: ad esempio se ho stallo;
12. Su richiesta del genitore;
13. Se muore il genitore.

SCHEDULER E DISPATCHER

- **Scheduler**: parte del kernel che decide qual'è il prossimo processo che il processore deve eseguire;
- **Dispatcher**: viene sempre eseguito contestualmente ad un mode switch di tipo kernel->user. Il dispacher è quella parte del sistema operativo che ripristina lo stato del processo che deve essere eseguito di li a poco (il processo viene deciso dallo scheduler). Esso viene eseguito quando il processo che era in esecuzione deve rilasciare la CPU e cioè nelle seguenti situazioni:
    1. Ha eseguito una system call bloccante;
    2. E' scaduto il suo quanto di tempo;
    3. Un processo con priorità maggiore diviene pronto (per scheduling preemptive);
    4. Il processo termina la sua esecuzione.

  Quindi il dispatcher switcha da un processo ad un altro.

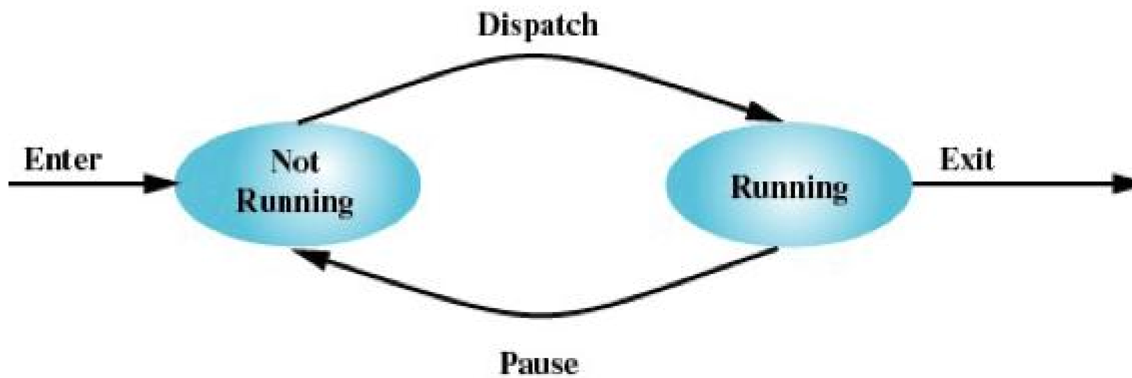
IMPLEMENTAZIONE DI UN PROCESSO IN SISTEMI MULTIPROGRAMMATI

I processi risiedono in memoria principale, in blocchi di memoria che contengono programma, dati, informazioni di contesto (PCP), e sono inseriti in una lista dei processi, la quale contiene una entry per ogni processo e comprende un puntatore alla locazione del blocco di memoria che lo contiene (in alcuni casi comprende pure le info di contesto). Il processore tiene traccia del processo attuale nel registro indice dei processi; il PC punta all'istruzione successiva del processo da eseguire, e i registri base e limite definiscono la porzione di memoria occupata dal processo.

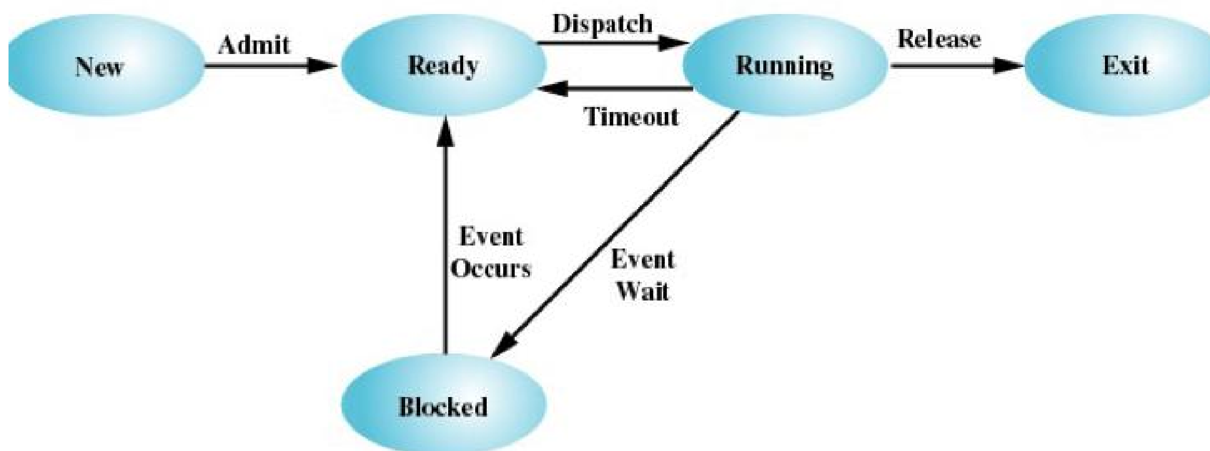
MODELLI DEGLI STATI DI UN PROCESSO

Un processo è caratterizzato da stati e gestito da un dispatcher, che passa al processore i processi selezionandoli opportunamente.
Esistono vari modelli per descrivere l'insieme degli stati di un generico processo:
- Modello base, a due stati:
    1. In esecuzione;

2. Non in esecuzione.



- Modello intermedio, a cinque stati:
    1. In esecuzione (running: il processo è eseguito dalla CPU)
    2. Bloccato/in attesa (blocked: processi in attesa di un evento)
    3. Pronto (ready: processi che possono essere eseguiti dalla CPU ma non sono al momento eseguiti)
    4. Nuovo
    5. Stato d'uscita.

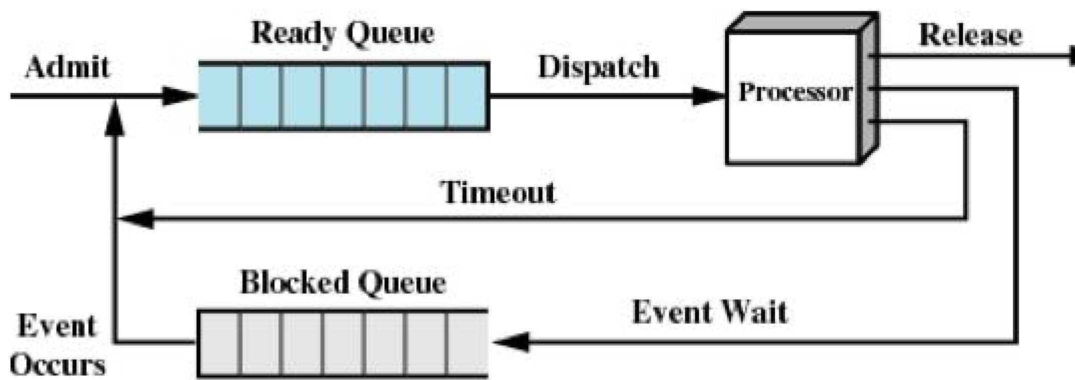


Lo stato bloccato/in attesa prevede, dal punto di vista gestionale, la presenza di una coda per ogni tipo di evento.

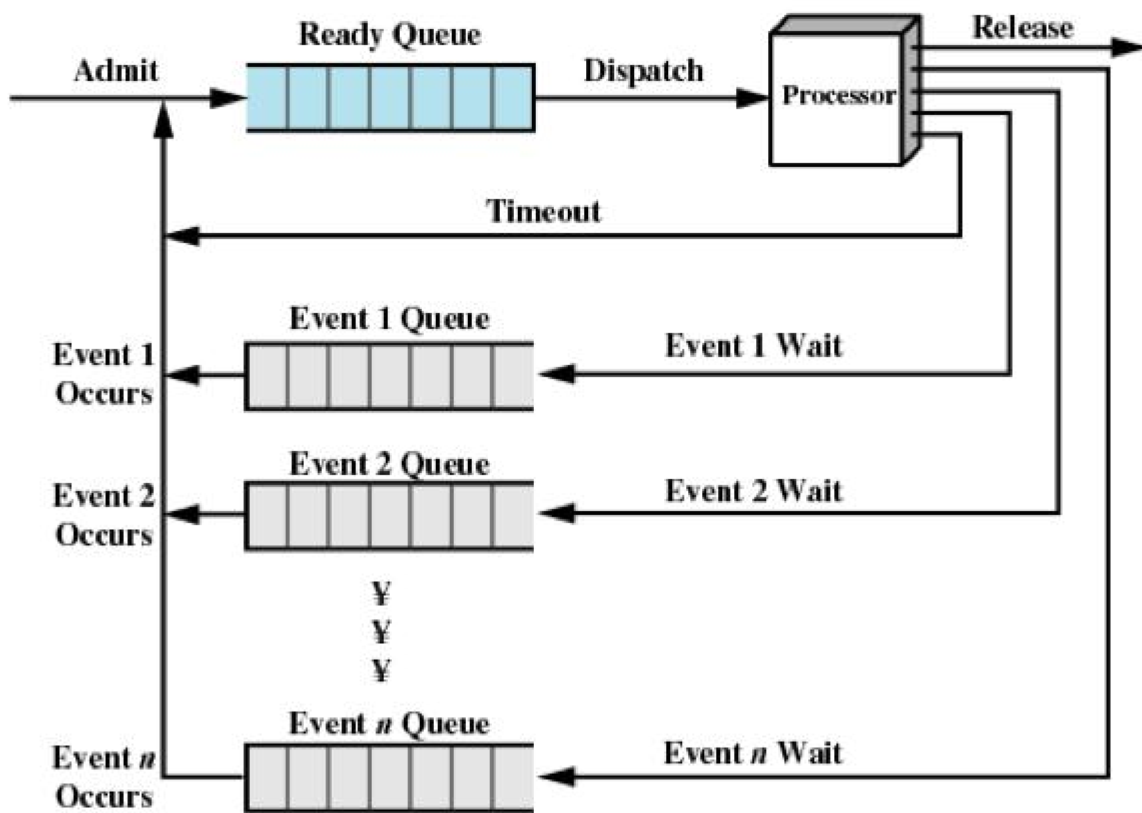
<u>TRANSIZIONI</u>:

1. running->ready: quando scade il quanto di tempo;
2. ready->running: lo scheduler ha scelto il processo che và in running;
3. running->blocked: processo ha eseguito system call bloccante;
4. blocked->ready: è accaudo l'evento per cui il processo era in attesa;
5. new->ready: il processo è stato lanciato;
6. running->exit: processo ha terminato l'esecuzione.

Con un disp di I/O:



Con più disp di I/O:



(b) Multiple blocked queues

Figure 3.8 Queuing Model for Figure 3.6

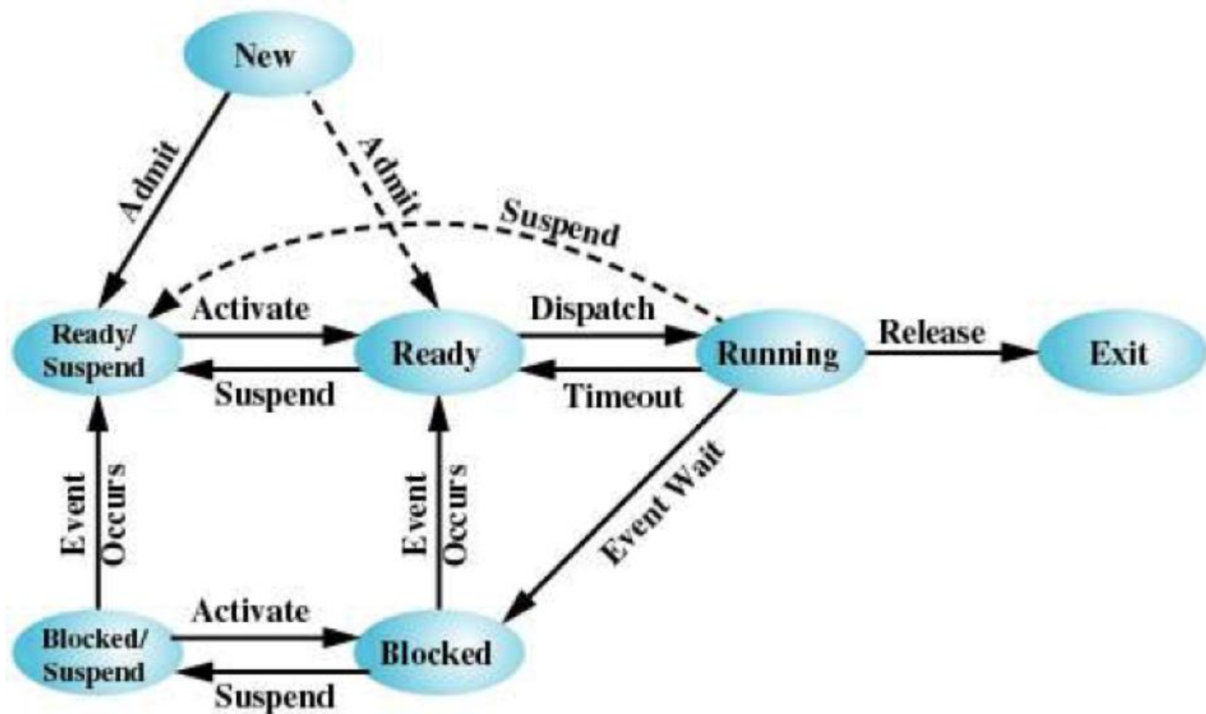
- Modello completo, a sette stati:
    1. In esecuzione;
    2. Pronto;
    3. Pronto e sospeso;
    4. Bloccato;

5. Bloccato e sospeso;
6. Nuovo;
7. Stato d'uscita.

NUOVI STATI: sono bloccato/sospeso e pronto/sospeso.
Il processore è più veloce dell'I/O quindi spesso molti processi restano in attesa occupando inutilmente la memoria:

- Tali processi vengono swappati su disco per liberarla
- Tali processi passano dallo stato bloccato a quello sospeso

L'introduzione degli stati sospesi (che lo differenziano dal modello precedente) si rende necessaria per descrivere quelle situazioni in cui è necessario liberare ulteriore memoria per qualche motivo.



(b) With Two Suspend States

CASI IN CUI VA SOSPESO UN PROCESSO

- Se un processo è in blocco da 10ms significa che 10ms fa ha eseguito una system-call:
- Swap (trasferimento su disco): il sistema operativo deve liberare sufficiente memoria per caricarci un processo ready;
- Altre cause per il sistema operativo: può sospendere un processo in background o di utilità, o un processo sospettato di dare problemi;
- Richiesta di un utente interattivo: per debug o per usare una risorsa;
- Temporizzazione: se un processo è eseguito periodicamente, nell'attesa lo sospendo;
- Su richiesta del processo genitore.

A questo modello si ispirano i moderni sistemi operativi.

Lezione 27/10/2009

DESCRIZIONE DEI PROCESSI

Dato che ai processi sono associate risorse di vario tipo è necessario che il sistema operativo conservi in memoria delle tabelle o strutture analoghe cross-referenziate, con informazioni su:

- Dispositivi di I/O;
- Memoria utilizzata; file utilizzati;
- Elenco dei processi.

L'elenco dei processi contiene solitamente puntatori alle cosiddette immagini dei processi.

IMMAGINE DI UN PROCESSO

Può stare in memoria virtuale e contenere la page table ed è costituita da:

- Process control block (l'insieme degli attributi del processo)
- Dati utilizzati (user data)
- Istruzioni da eseguire (user program)
- Stack LIFO a lui associato.

L'esecuzione di un processo comporta il caricamento in memoria centrale o virtuale dell'intera immagine del processo.

P.C.B. (process control block)

Il P.C.B., il cui ruolo centrale è la gestione dei processi da parte del sistema operativo, si compone di 3 parti:

1. Gli identificativi di processo, dell'eventuale processo padre e dell'utente;
2. Le informazioni sullo stato del processore, in particolare: registri visibili all'utente, registri di controllo e di stato, stack pointers;
3. Informazioni per il controllo del processo: privilegi, priorità, stato, risorse in uso, informazioni sulla gestione della memoria.

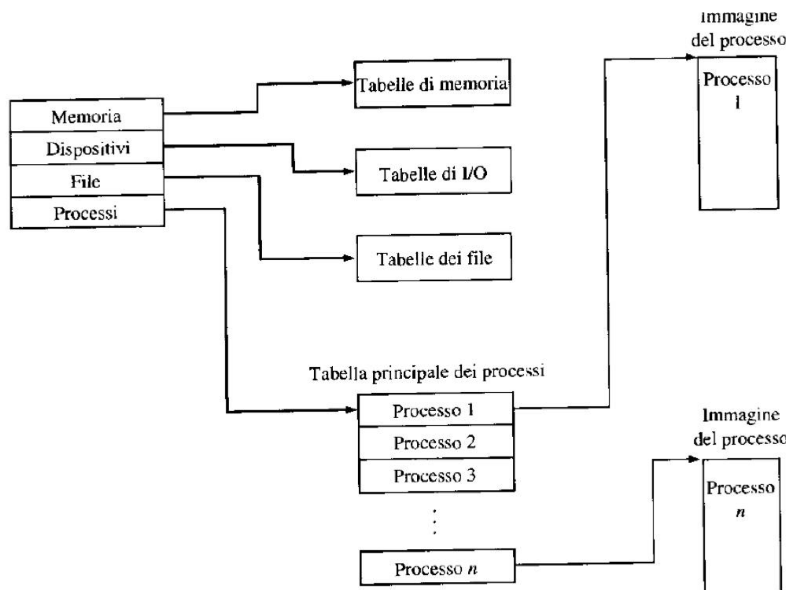
Solitamente, le operazioni di lettura, scrittura e modifica dei process control blocks sono effettuate esclusivamente da un'apposita routine del sistema operativo, detta handler routine, in modo da evitare il più possibile modifiche dannose, accidentali o volute che siano. In questo modo inoltre modifiche alla struttura di come è fatto un process control block coinvolgeranno solo la handler routine e non tutte le routine che utilizzano i process control block.

CREAZIONE DEI PROCESSI

Avviene in 5 fasi:

1. Assegnamento di un identificativo univoco al nuovo processo;
2. Allocazione delle risorse per il processo stesso;
3. Inizializzazione del process control block relativo al processo;
4. Inserimento del processo nell'opportuna coda, relativamente allo suo stato;
5. Creazione di eventuali strutture accessorie di supporto.

TABELLE



Sono strutture di supporto per PCB/processi:

- Tabella processi: 1 riga per ogni processo; per attivare il processo: puntatore al pcb; albero, hash,..;
- Tabella memoria: RAM allocata al processo; HD allocato al processo; attributi protetti per accedere a regioni RAM condivise; informazioni per gestire la memoria virtuale;
- Tabella I/O: dispositivi disponibili e occupati; informazioni su operazioni di I/O; locazione di memoria da cui leggere, e dove scrivere;
- Tabella file: file esistenti, locazione su HD, stato corrente, attributi.

STRUTTURE DI CONTROLLO DEI PROCESSI: sono per la locazione dell'immagine del processo, e per gli attributi del suo PCB.

CONTROLLO DEI PROCESSI

MODALITA' DI ESECUZIONE DEI PROCESSI: Un processo può essere eseguito in due modalità distinte, che si cambiano con un mode switch.

MODE SWITCH

Avviene quando si serve un interrupt o una system call (user -> kernel) o quando si fa il dispatching di un processo (kernel -> user).
Nel ciclo di istruzione la CPU fa un ciclo di interrupt per controllare la presenza di interrupt, se ci sono salva:

- il contesto (info che interrupt può cambiare e che saranno da ripristinare);
- parte del PCB: informazioni sullo stato del processore (registri, PC, stack), programmi in running, e mette nel PC l'indirizzo iniziale dell'handler (gestore dell'interrupt), poi fa un mode switch da user mode a kernel mode.

1. user mode: la modalità tradizionale utilizzata dalle applicazioni; non possiede tutti i privilegi e non può svolgere alcune operazioni; solo una parte della memoria può essere acceduta (lo user space);
2. kernel mode (o system mode o control mode): la modalità utilizzata dal sistema operativo e dalle relative operazioni (il kernel è eseguito in questa modalità).

Il fatto di avere due modalità è dettato dalla necessità di separare i ruoli, le competenze e le responsabilità tra i generici programmi e il sistema operativo. Quest'ultimo in particolare, lavorando a contatto con le risorse, si trova a svolgere operazioni spesso delicate e potenzialmente pericolose se non effettuate in maniera attenta e controllata, cioè protetta. Da ciò, la necessità della distinzione:

- Certe operazioni privilegiate possono essere eseguite;
- L'area di memoria protetta (kernel space) può essere acceduta;
- L'I/O và sempre gestito in kernel mode.

Tipiche operazioni effettuate dal sistema operativo in kernel mode sono:

- Creazione e terminazione di processi;
- Scheduling, dispatching e switching di processi;
- Gestione della memoria: indirizzamento, swapping...;
- Gestione dell'I/O;
- Monitoraggio e accounting.

PROCESS SWITCH (o Context Switch)

Ripristina il contesto del processo. Una system call bloccante causa sempre un process switch se ci sono altri processi. Un process switch può avvenire solo e soltanto in kernel-mode. Un process switch avviene sempre contestualmente a 2 mode switch.

Quando effettuare Process Switch?

Avviene in seguito a:

- Interruzioni di sistema esterne (interrupts): clock interrupt, I/O interrupt, memory fault;
- Interruzioni di sistema interne (traps: errori), dovute solitamente ad errori interni di qualche sorta;
- Chiamate ad una routine del sistema operativo (system call), come l'apertura di un file.

Come effettuare Process Switch? (Cosa richiede?)

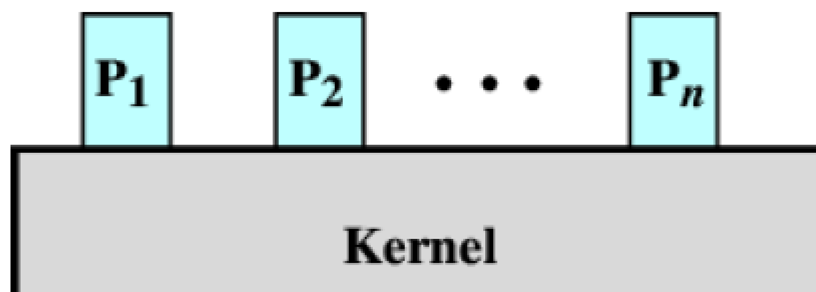
- Salvataggio del contesto del processore allo stato attuale;
- Aggiornamento del control block del processo corrente;
- Movimento del process control block nell'apposita coda;
- Selezione di un altro processo da parte dello scheduler;
- Aggiornamento del process control block del processo selezionato;
- Aggiornamento delle strutture dati di gestione della memoria, a seconda della tecnica usata;
- Ripristino del contesto del processore, relativo al nuovo processo, così com'era stato lasciato in precedenza.

Lezione 10/2009

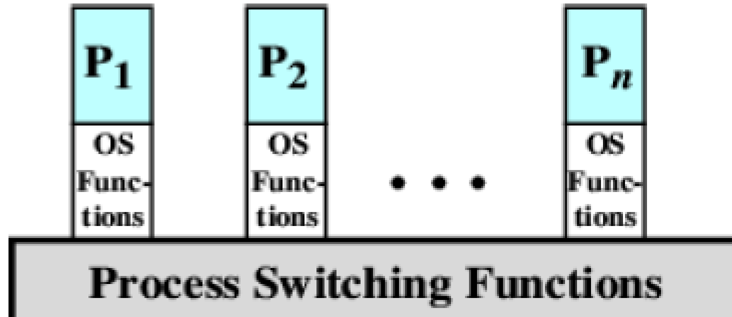
MODALITA' DI ESECUZIONE DI UN SISTEMA OPERATIVO

Storicamente, un sistema operativo può essere eseguito in 3 modi:

1. Nonprocess kernel ("non come un processo"), cioè esternamente rispetto ai processi



2. Within user processes ("dentro i processi degli utenti"), cioè solo il dispatcher rimane esterno mentre le funzioni di sistema sono accoppiate ai processi utente, attraverso solo un cambio di modalità (es.: Linux)



3. Process based operating system (sistema operativo basato sui processi): ad eccezione del dispatcher, tutte le funzioni del sistema operativo sono progettate in maniera modulare, ed eseguite come processi indipendenti. Richiede notevoli doti di design e severità di progettazione, rendendo il sistema molto elegante e funzionale, grazie alla obbligata modularità (es.: BeOS)

Process Identification
Processor State Information
Process Control Information
Process Co Block
User Stack
Private User Address Space (Programs, Data)
Kernel Stack
Shared Address Space
P1
P2
Pn
OS1
OSk
Process Switching Functions

Tecniche di gestione della memoria centrale

GESTIONE DELLA MEMORIA

Sono 5 i compiti principali del sistema operativo per la gestione della memoria:
1. Isolamento dei processi;
2. Allocazione e gestione automatica della memoria;
3. Supporto per la programmazione modulare;
4. Protezione e controllo dell’accesso;
5. Memorizzazione a lungo termine.

Per soddisfare tali necessità, i sistemi operativi usano la memoria virtuale.
MEMORIA VIRTUALE: permette ai programmi di indirizzare la memoria da un punto di vista logico, senza preoccuparsi della quantità di memoria fisicamente disponibile. I programmi quindi vedono più memoria di quella realmente disponibile.
La gestione della memoria intende soddisfare i seguenti 5 requisiti:
1. Riallocazione: capacità di allocare un processo in aree diverse ogni volta che sia necessario, come nello swapping-in: a tale scopo un processo non deve essere legato in maniera assoluta ad alcuni indirizzi di memoria;
2. Protezione: intesa come la possibilità di mantenere distinte zone di memoria appartenenti a processi diversi, in modo da evitare sia pericolose sovrapposizioni, casuali o intenzionali, che l'accesso da parte di processi (e utenti) non autorizzati;
3. Condivisione: complementare al precedente, nel caso in cui sia necessario che alcuni processi accedano a zone di memoria contenenti istruzioni o dati comuni;
4. Organizzazione logica: deve esserci una corrispondenza tra l'organizzazione logica della memoria, che è strutturata in maniera lineare, e l'organizzazione modulare delle applicazioni software;
5. Organizzazione fisica: la memoria deve essere organizzata in due livelli, la memoria centrale, volatile e veloce nell'accesso, e la memoria secondaria, più lenta ma capace di conservare dati nel tempo (e in maggior quantità). La relazione che intercorre tra le due memorie deve essere responsabilità del gestore della memoria, per evitare al programmatore compiti onerosi e quasi impossibili come l'overlaying o il calcolo esatto della memoria utilizzata dai processi generati da un suo programma.

TECNICHE DI PARTIZIONE DELLA MEMORIA:

PARTIZIONAMENTO FISSO(SW):

- A) Partizionamento a dimensione fissa: la memoria è suddivisa in un numero fisso di partizioni e ad ogni processo viene assegnata una partizione libera, finché tutte le partizioni non sono usate (ovvero fino a quando non ci sono partizioni libere).

  PROBLEMI: dimensione programma -> dimensione partizione -> va fatto overlay dividendo il programma in moduli caricabili un pezzo per volta; usa inefficientemente la RAM: qualsiasi programma occupa una partizione -> <u>frammentazione interna</u> ovvero spreco di spazio in ogni partizione (se ho una partizione da 8mb e se arriva un processo da 2mb questi la occupa sprecando gli altri 6mb che restano inutilizzabili).
  Inoltre il numero di partizioni limita il numero processi in memoria; di conseguenza il partizionamento fisso non viene mai usato.
  ALGORITMO DI ALLOCAZIONE:
  - Finchè ho partizioni libere alloco i processi;

- Se non si hanno partizioni libere il sistema operativo swappa fuori da una partizione un processo per mettercene un altro.

Per OVERLAYING si intende una tecnica di programmazione che consiste nell'organizzare istruzioni e dati in moduli che possono essere assegnati alla stessa porzione di memoria e che devono essere gestiti da uno switcher software (facente parte del programma stesso e realizzato dal programmatore) che carica i moduli necessari a runtime.

- B) Partizionamento a dimensione diverse (non dinamica): Parte di questi problemi può essere risolta usando partizioni di taglio diverso ed algoritmi per la selezione della partizione più adatta al processo. Diminuisce l'overlay; si rispettano maggiormente le dimensioni del processo rispetto a quelle delle partizioni (partizioni più appropriate per ogni processo). Comunque anche in questo caso sono presenti degli svantaggi, in quanto il numero di partizioni è fisso e se il sistema comprende molti piccoli processi, ci sarà sempre uno spreco di risorse.

ALGORITMO DI ALLOCAZIONE:

- Assegno il processo alla partizione più piccola che lo può contenere (contro: presupone la conoscenza di quanta memoria richiederà un processo);
- Serve una coda per ogni partizione per ricordare dove sono stati i processi così da poterceli riallocare (pro: minimizza frammentazione interna; contro: se non arriva un processo con dimensioni comprese in quella di una partizione, tale partizione resta inutilizzata);

Soluzione: utilizzo una sola coda per tutti i processi, e seleziono per ogni processo la partizione più piccola.

PRO:

- Flessibilità partizionamento fisico;
- Semplice;
- Poco sovraccarico del sistema operativo;

PARTIZIONAMENTO DINAMICO (SW):

Ad ogni processo viene assegnato man mano lo spazio che serve. Si usano partizioni in numero e lunghezza variabile; quando un processo viene caricato nella RAM gli viene allocata tanta memoria quanta richiesta e mai di più.

PROBLEMA: frammentazione esterna: si presenta quando lo spazio comincia a riempirsi ed è necessario selezionare il posto migliore dove mettere un processo, tra quelli disponibili. L'algoritmo inizia bene poi inizia a creare tutti piccoli buchi in RAM:

- La memoria diventa sempre più frammentata e l'utilizzo peggiora;
- La memoria esterna a tutte le partizioni diventano sempre più frammentate.

SOLUZIONE->COMPATTAZIONE DELLO SPAZIO LIBERO: di tanto in tanto si compattano i buchi per formare una nuova partizione (si rendono contigui), ma è complesso e costoso farlo poichè vanno risistemati i processi in RAM, controllando e risistemando tutti i puntatori.

ALGORITMO DI ALLOCAZIONE=QUALE BUCO LIBERO SCEGLIERE?

Sono necessari degli algoritmi di posizionamento (placement):

- <u>Firstfit</u>: è il più semplice ed efficace. Scandisce memoria dall'inizio e sceglie il primo blocco disponibile sufficientemente grande; è il più veloce, più semplice e il migliore. Tende però a riempire la parte iniziale della RAM di piccole partizioni libere da esaminare ad ogni passo. Tale tecnica di partizionamento causa un elevata frammentazione esterna.
- <u>Best-fit</u>: sceglie il blocco più vicino alla richiesta in termini di dimensione (sceglie blocco più piccolo); garantisce che il frammento lasciato sia il più piccolo possibile,

  CONTRO: la RAM è rapidamente inframezzata di blocchi troppo piccoli per soddisfare le richieste di allocazione e quindi la compattazione và fatta molto di frequente.

- Next-fit (il più usato): scandisce la memoria partendo dalla locazione dell’ultima allocazione; sceglie il successivo blocco disponibile a tale allocazione e sufficientemente grande.

  CONTRO: leggermente peggiore del Firstfit, ma alloca sempre nel blocco libero alla fine della RAM. Tale blocco, che di solito il più grande libero, viene smantellato in piccoli frammenti, quindi è spesso richiesta la compattazione.

ALGORITMO DI RIALLOCAZIONE
Quando la RAM è piena di processi e non c’è spazio nemmeno dopo la compattazione, non si aspetta che un processo attivo si sblocchi (si sprecherebbe tempo). Il sistema operativo swappa uno dei processi in RAM per fare spazio ad un altro: il sistema operativo deve scegliere quale processo rimpiazzare (riallocazione).

BUDDY SYSTEM(sw):

Usato da Linux. Finora si è visto che il partizionamento fisso limita il numero di processi attivi e usa inefficientemente la memoria (frammentazione interna). Il partizionamento dinamico è più complicato e comporta il costo della compattazione (oltre alla frammentazione esterna).
Il buddy system è un compromesso. L'idea è quella di dividere la memoria in blocchi di dimensione 2^k, con k compreso tra L ed U, dove 2^L è il più piccolo blocco allocabile e 2^U è il più grande, eventualmente pari a tutta la memoria disponibile.
ALGORITMO:
- Inizialmente tutto lo spazio è considerato come un unico blocco di 2^U.
  - Se la memoria s richiesta da un processo è $2^{U-1} < s <= 2^U$, tutto il blocco di dimensione 2^U è allocato.
  - Altrimenti ($s < 2^{U-1}$) il blocco 2^U è diviso in due buddies (blocchi) uguali da 2^{U-1} bytes.
- Se $2^{U-2} < s <= 2^{U-1}$:
  - Alloco uno dei 2 buddies;
  - Altrimenti ridivido uno dei 2 buddies;
  - Continuo cosi fino a che, fondendo e splittando blocchi più grandi, non trovo la partizione con dimensione pari ad una potenza di 2 che soddisfa le esigenze del processo (dimensione >=s).

Si mantiene una lista di buchi (blocchi liberi) L_i (i=0,1,..U) per ogni dimensione 2^i dei blocchi liberi (al massimo ho U liste):
- SPLITTING: un buco con dimensione 2^{i+1} (padre) lo posso rimuovere dividendolo in 2 buddy di dimensione 2^i da inserire nella lista i-esima L_i;
- COALESCING (merge): rimuove 2 buddy liberi dalla lista L_i e li mette nella lista L_{i+1} (crea il padre).

ALGORITMO:
- se ho richiesta di dimensione k t.c. $2^{i-1} < k <= 2^i$ cerco un blocco libero di dimensione 2^i così:

procedure get_hole:
  - input: i (precondition: i≤U);
  - output: a block c of size 2^i (postcondition: L_i does not contain c).

```
if ( Li is empty ) {
    b = get_hole(i+1);
    < split b into two buddies b1 and b2 >
    < put b1 and b2 into Li  >
}
c= < first hole in Li>
<remove c form Li>
return c.
```

Nota bene: due blocchi contigui non sono necessariamente due buddy.

Lezione 03/11/2009

COME SI COMPORTANO LA FRAMMENTAZIONE INTERNA E QUELLA ESTERNA:
La frammentazione esterna è limitata poiché la dimensione dei frammenti che non hanno buddy è sempre maggiore o uguale al blocco più piccolo allocato nel sistema.
La frammentazione interna: è presente, ma non si spreca più della metà della memoria allocata. Secondo l'ipotesi di richieste uniformemente distribuite se ne spreca mediamente 1/4.

APPROCCIO LAZY CONTRO "MERGING APPENA DEALLOCHI"

Le sequenze in cui ciascuna allocazione è seguita dalla rispettiva deallocazione sono inefficienti poiché il buddy system ad ogni deallocazione unisce ciò che aveva spezzato. Un approccio migliore è quello "lazy" in cui si và in cerca di blocchi da unire solo se, in fase di allocazione, non trovo un blocco sufficientemente grande per soddisfare la richiesta.
IN PRATICA:
Ogni livello (dimensione) ha una lista di nodi con stessa dimensione, ad ogni passo cerca un nodo con dimensione richiesta e semmai dividi nodi in 2 figli buddies di stessa dimensione. Quando deallochi memoria puoi unire nodi fratelli (stessa dimensione e stesso padre), le foglie sono il partizionamento corrente. Se 2 buddy sono foglie uno dei 2 deve essere allocato altrimenti li fondi (merge), quando splitto un blocco di dimensione *i* lo levo dalla lista *i* e aggiungo 2 blocchi alla lista *i*/2.
Inoltre se due allocazioni di memoria vicine, di pari dimensione, risultano vuote, vengono fuse in un unica allocazione di dimensione superiore.
L'insieme degli slot di memoria usati, creati e disponibili viene mantenuto in una serie di liste.
RILOCAZIONE:
La rilocazione è il problema è allocare processi in RAM. Può capitare che lo stesso processo và in RAM, poi esce, poi rientra, occupando partizioni diverse. Col partizionamento fisso ipotizzo che non accada e l'indirizzo assoluto RAM coincide con l'indirizzo relativo del processo più il suo indirizzo base, ma se ho partizioni di dimensioni uguali e una coda sola per ogni partizione diversa può acccadere.
Stessa cosa con il partizionamento dinamico. Con la compattazione sposto processi già in RAM, cambiando indirizzi assoluti (in RAM).

Nota: Le locazioni in RAM a cui un processo fa riferimento non sono fisse.

Soluzione:
1. Indirizzo logico: riferimento ad una locazione in ram indipendente dall'assegnazione corrente dei dati in RAM; va tradotto in indirizzo fisico prima di accedere alla RAM;
2. Indirizzo relativo: è un indirizzo logico espresso come locazione relativa ad un punto conosciuto (inizio programma);
3. Indirizzo assoluto (o fisico): locazione effettiva in RAM (dove risiede l'immagine del processo):
    - Traduzione indirizzo logico in indirizzo fisico va fatta via hw (la esegue MMU);
    - Alla partenza di un processo il sistema operativo aggiorna il registro base (l'indirizzo dell'inizio del programma) e il registro limite (l'indirizzo della fine del programma);
    - Si esegue il processo incontrando l'indirizzo relativo (IR, chiamata di sistema, dati): indirizzo fisico = indirizzo relativo + registro base;
    - Poi confronto l'indirizzo assoluto con il registro limite;

◦ Se l’assoluto è nei limiti accedo alla RAM, altrimenti genero interrupt al sistema operativo per comunicargli l’errore.

PAGINAZIONE (hw)

Senza memoria virtuale si divide la memoria in piccole parti uguali, dette frames e i processi in pagine. E' presente frammentazione interna nell’ultima pagina del processo ma nessuna frammentazione esterna.
Il sistema operativo mantiene una lista di frame liberi in RAM. Il massimo numero di pagine è 2^n con n = bit per rappresentare il numero di pagina nell’indirizzo logico.
Se ho 4gb di indirizzo virtuale ho un indirizzo di 32 bit (232); se ho 4kb a pagina ho 20 bit per indirizzarla (220=4096); 32-20=12 bit per l’offset; 220 righe della page table, ogni riga di 3 o 4 byte ->dim page table=3o 4*220.

COME METTO UN PROCESSO IN RAM:

Non basta più un registro base. Ogni processo possiede una tabella delle pagine che mette in corrispondenza le pagine del processo con i frames che le contengono in quel determinato istante.
Ha una riga per ogni pagina del processo (indicizzata proprio col numero di pagina). Ogni riga contiene il numero di frame in RAM corrispondente a quella pagina:

- Dimensione della pagina = dimensione del frame = potenze di 2;
- Indirizzo relativo (inizio programma) = indirizzo logico (numero di pagina, offset);
- Schema indirizzo logico è trasparente al programmatore, assemblatore, linker;
- Facile passare dall'indirizzo relativo all'indirizzo logico (sono uguali) e poi all'indirizzo fisico.

L'indirizzo logico è formato da due parti, una con il numero di pagina (n bit a sinistra) e una con l'offset all'interno della pagina (m bit a destra).
Con il numero di pagina accedo alla page table e ricavo il numero del frame corrispondente in RAM, e quindi:

- L'indirizzo fisico iniziale del frame k è $k*2^m$;
- L'indirizzo fisico del byte indirizzato è numero di frame concatenato all’offset.

In questo modo la frammentazione è solo quella interna presente nell'ultima pagina del processo.

RIASSUMENDO:

- La paginazione è simile al partizionamento fisso: memoria in frame, processo in pagine, tutti di dimensione fissa e 1frame=1pagina, con la differenza però che qui le partizioni sono piccole e un programma può occupare più partizioni in RAM e non per forza contigue (tanto ho la page table che associa pagine a frame e quindi è come se lo fossero);
- L'indirizzo relativo è uguale all'indirizzo logico;
- L'indirizzo fisico è uguale al numero del frame concatenato all'offset, dove il numero del frame si ricava dalla tabella delle pagine accedendovi con il numero della pagina ricavato dai bit a sinistra nell’indirizzo logico, mentre l'offset è rappresentato dai bit più a destra dell’indirizzo logico.

SEGMENTAZIONE (hw)

Simile alla paginazione. La segmentazione divide un programma in segmenti, non per forza della stessa dimensione, in modo da eliminare la frammentazione interna (ma creando la frammentazione esterna). La memoria però non viene partizionata, ogni

segmento ha comunque lunghezza massima.
Simile al partizionamento dinamico ma qui un programma può usare più di una partizione e le partizioni usate non devono per forza essere contigue. La segmentazione è visibile al programmatore (al contrario della paginazione). Il programmatore deve tener conto delle dimensioni massime dei segmenti.

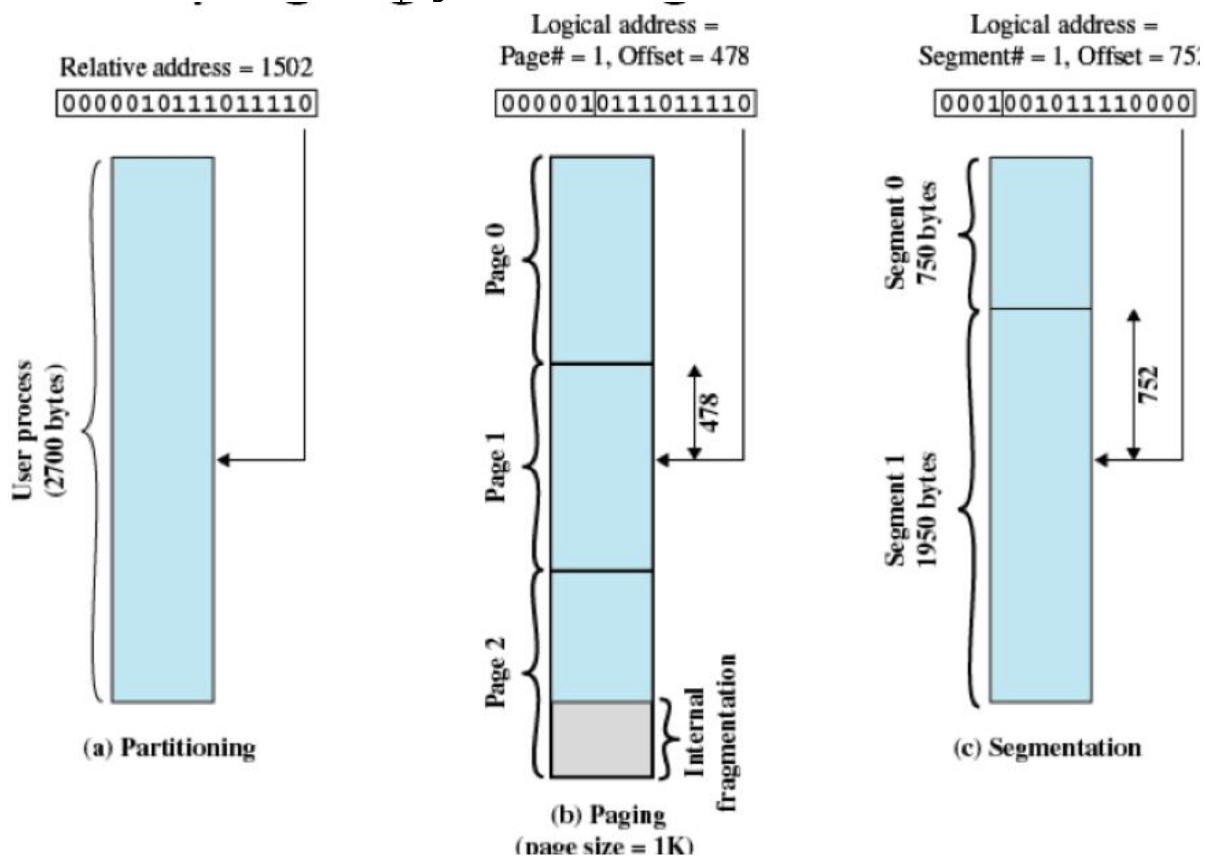
Tavola dei segmenti e traduzione dell'indirizzo logico e fisico:

Per ogni processo l'elenco dei segmenti è conservato in un apposita tabella dei segmenti, che contiene la lunghezza del segmento e l'indirizzo iniziale del segmento (base) da sommare all'offset.

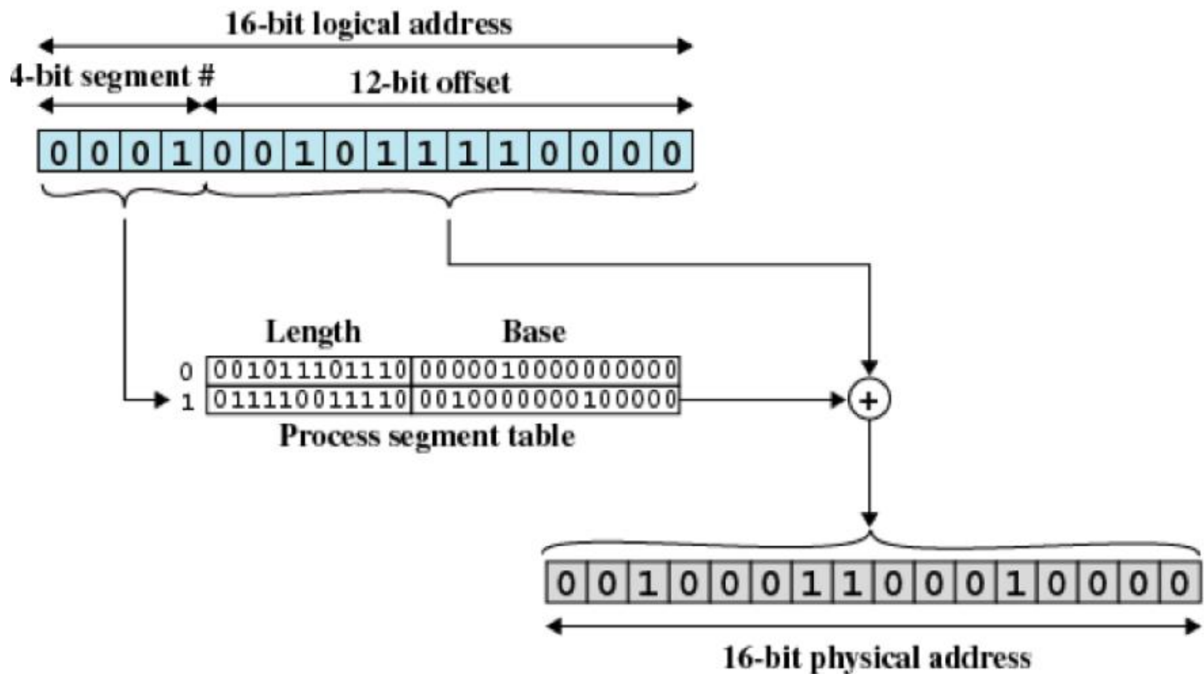
1. Un processo è un insieme di segmenti, un processo running va portato in RAM;
2. Il processo ha la sua tavola dei segmenti il cui indirizzo sta in un registro hw;
3. L'indirizzo logico di n+m bit (n bit più a sinistra sono il numero del segmento, e gli m più a destra sono l'offset all'interno del segmento in RAM); dimensione massima segmento = $2^m = 2^{offset}$.
4. Estraggo il numero del segmento dall'indirizzo logico e lo uso per accedere alla tavola dei segmenti per estrarre il relativo indirizzo fisico iniziale del segmento;
5. Confronto l'offset con la lunghezza del segmento (campo della tabella) e se l'offset è maggiore della lunghezza massima del segmento (length-offset < 0) l'indirizzo non è valido;
6. Se invece è valido: indirizzo fisico = indirizzo fisico iniziale segmento + offset (qui ho proprio la somma, mentre con la paginazione avevo una concatenazione).

Non è possibile pertanto creare una corrispondenza immediata tra indirizzo fisico e logico:

- memoria virtuale con paginazione:



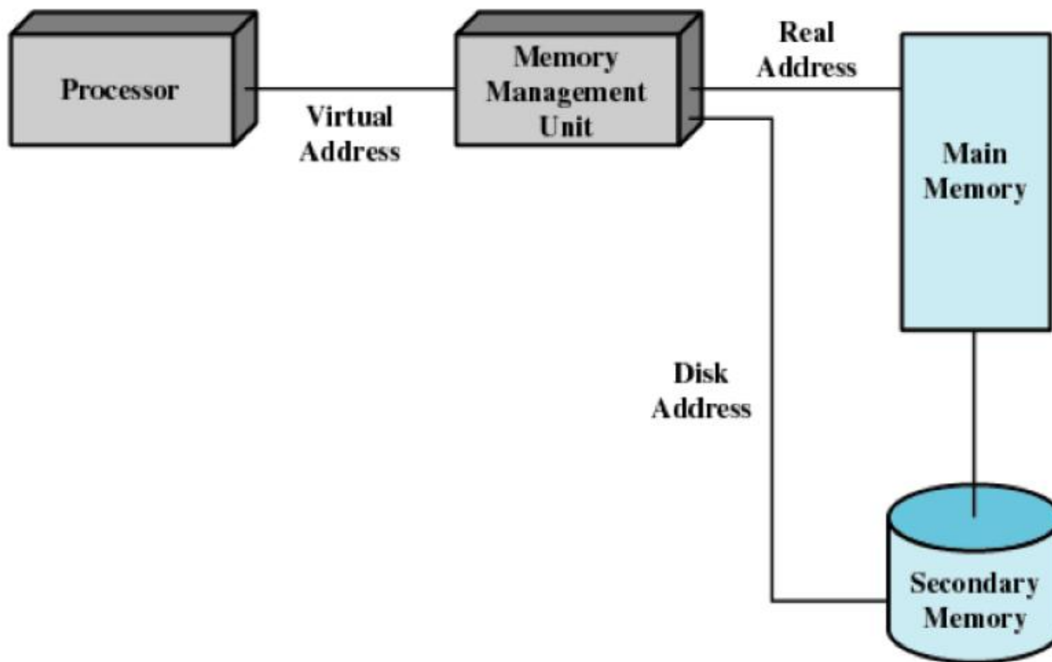
- memoria virtuale con segmentazione:



Lezione 07/11/2009

MEMORIA VIRTUALE

La memoria virtuale permette di indirizzare la memoria principale da un punto di vista logico, senza preoccuparsi della quantità di memoria fisicamente disponibile: è come se creassi un mondo virtuale in cui posso spaziare come voglio. Non è quindi più necessario che un processo, per essere eseguito, abbia tutte le sue pagine/segmenti in memoria: carico di volta in volta i pezzi che servono per andare avanti nell’esecuzione. Il mappaggio tra indirizzo logico e indirizzo fisico lo fa il processore attraverso l'MMU.



VANTAGGI:

1. Ci sono più processi in RAM quindi con più possibilità di avere processi in stato ready;
2. La dimensione del processo può essere maggiore della dimensione della RAM: il programmatore non deve più fare overlay (dividetare in moduli un programma grande) perchè lo fa il sistema operativo insieme alla CPU caricando alcune parti di programma alla volta.
3. Se deve eseguire una piccola routine di un programma carico solo quella in ram (t risparmiato)-->no i/o ram<->disco di pezzi inutilizzati .
4. avere una gestione stratificata e distinta degli indirizzi di memoria, tramite l'indirizzamento logico e quello fisico;
5. separare un processo in pezzi, segmenti o pagine che siano; fa si che non sia necessario che tutti i segmenti o le pagine di un processo siano presenti in memoria, ma che ci siano solo quelle effettivamente utilizzate (o che saranno utilizzate a breve) in un determinato istante. Tale insieme di pagine è chiamato RESIDENT SET (porzione di un processo che e' attualmente caricato in memoria principale). Se il processo richiede una porzione di codice o dati che si trovano su una pagina/segmento non presente in memoria centrale, viene generato un errore di page/segment fault: il processo viene posto nello stato di "bloccato" e la pagina/segmento richiesta caricata; contemporaneamente, un altro processo viene eseguito. È bene inoltre notare che in questo modo possono essere mantenuti e usati in memoria centrale (detta real memory) più processi, anche di dimensioni maggiori della memoria stessa, in quanto le pagine/segmenti non necessari si troveranno nella memoria centrale ma in quella virtuale. Molti page fault su un processo modificano le prestazioni degli altri processi. Un processo per ottenere nuova memoria deve fare una system call.

TRASHING

Fenomeno per il quale, quando la ram è satura di processi e devono entrarne altri, il processore impiega la maggior parte del suo tempo a gestire l'I/O da disco per caricare/scaricare le pagine dei processi invece che eseguire istruzioni. Si cerca di evitare fenomeni di questo tipo pianificando la strategia di sostituzione delle pagine in memoria principale quando questa risulta al limite della capacità.
Ci viene in aiuto il **principio di località**, il principio base seguito nella pianificazione di strategie di sostituzione di pagine della memoria principale. Afferma che i riferimenti al

programma o ai dati all'interno di un processo tendono a raggruparsi.
Abbiamo due tipi di località, quella temporale (dati e istruzioni di un ciclo for) e quella spaziale (es.: accesso ad un'insieme di dati strutturati e sequenziali).
In conclusione per ottenere un buona gestione della memoria VIRTUALE è necessario:

- L'uso della paginazione o della segmentazione, a livello hardware;
- L'uso del sistema operativo per la scelta e lo spostamento di pagine e segmenti dalla memoria centrale a quella virtuale.

Nota: MEMORIA VIRTUALE E DISK CACHING SONO 2 COSE DIVERSE
Analogie: entrambe le tecniche mirano a tenere in memoria solo dati e programmi acceduti di frequente.
Differenze: le tecniche agiscono in domini differenti: virtual memory (processi, pagine, segmenti); disk caching (files).

PAGINAZIONE + MEMORIA VIRTUALE

Permette ai processi di essere compresi in blocchi di dimensioni fissate, chiamati pagine. L'indirizzo virtuale è un numero di una pagina e un offset. Ogni pagina può stare ovunque nella memoria principale (non per forza in ordine con le altre pagine dello stesso processo). Indirizzo fisico (reale) nella memoria principale è gestito unicamente dal kernel. L'uso della paginazione in abbinamento alla memoria virtuale richiede diversi accorgimenti. Innanzitutto è necessario predisporre all'interno delle TABELLE DELLE PAGINE di un bit di presenza e uno di modifica, per sapere rispettivamente se la pagina in questione è presente in memoria centrale o meno, e se è stata modificata, in modo da comportarsi di conseguenza. La page table deve stare in ram, ma dato che un processo potrebbe avere un numero altissimo di pagine e una corrispondente tabella delle pagine molto grande,è necessaria una gestione particolare di tale tabella, mettendola in memoria virtuale (insieme all'immagine del processo) --> PT impaginate come processi e solo la parte di essa necessaria per eseguire un P in un certo momento verrà messa in ram.

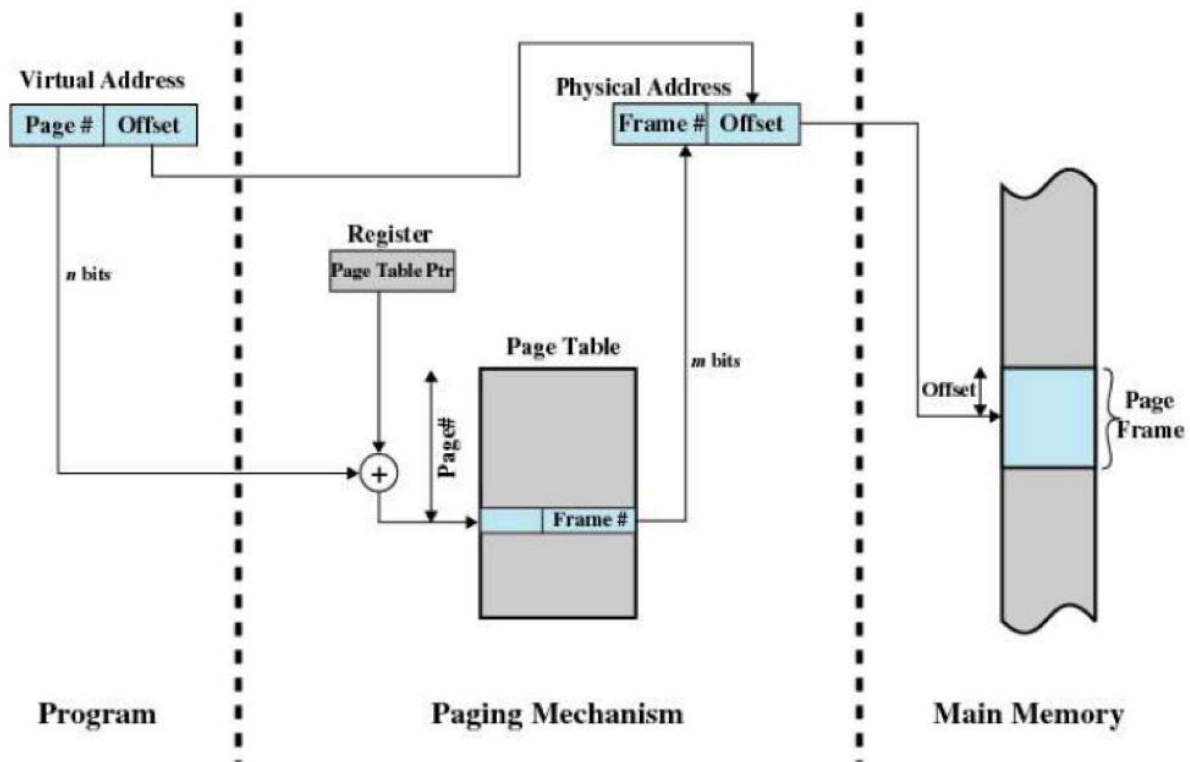
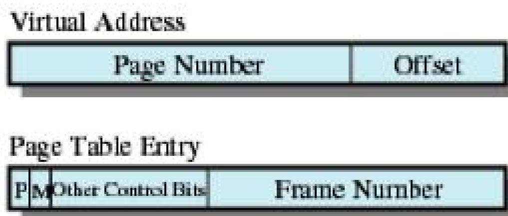


Figure 8.3 Address Translation in a Paging System



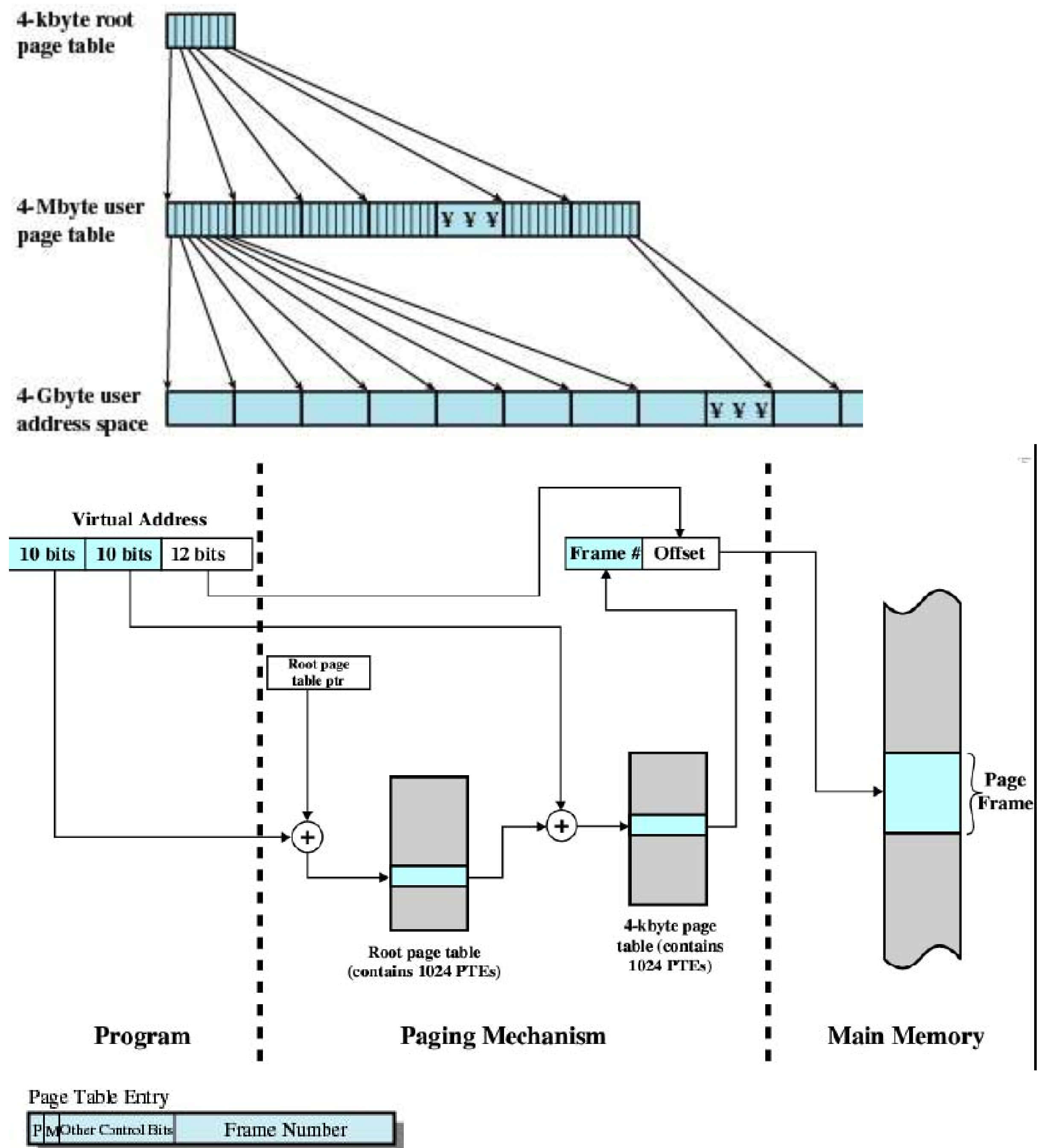
TECNICHE DI GESTIONE DELLA PT SE TROPPO GRANDE:

Tabella delle pagine a DUE LIVELLI gerarchiche:

- Il primo livello, detto tabella delle pagine radice, indirizza le pagine del secondo livello;
- il secondo livello indirizza invece la memoria effettivamente usata.

Il succo è: tabelle delle pagine al secondo livello indirizzano le pagine della memoria principale (se ho una PT questa ha una riga per ogni pagina della memoria principale); poi posso mettere tali PT in pagine, e mappare tali pagine in una tabella principale (la root), che avrà 1 riga per ogni pagina occupata dalla/dalle PT a livello sottostante. Se la root table ha lunghezza X e la max lunghezza della pt è y, un processo può essere composto di X*Y pagine. La max lunghezza di una tabella delle pagine di solito è una pagina!!
Pro: permette di non avere in memoria l'intera tabella.
Contro: per una singola lettura si possono avere molti page fault (al massimo pari al numero dei livelli) e che comunque si devono fare un numero di accessi a memoria in più

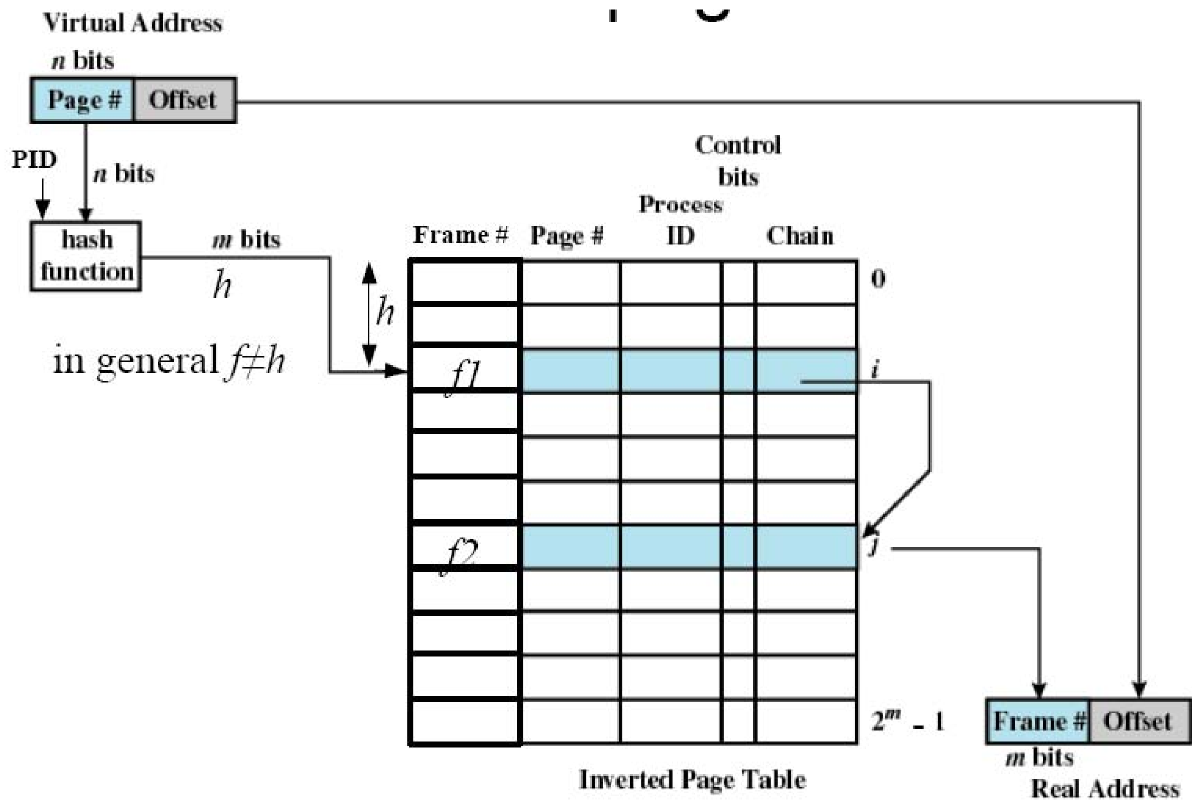
pari al numero dei livelli. Richiede un doppio accesso alla memoria (uno per portare in memoria la porzione di PT che ci serve, e 1 per accedere alla pagina (ai dati)).



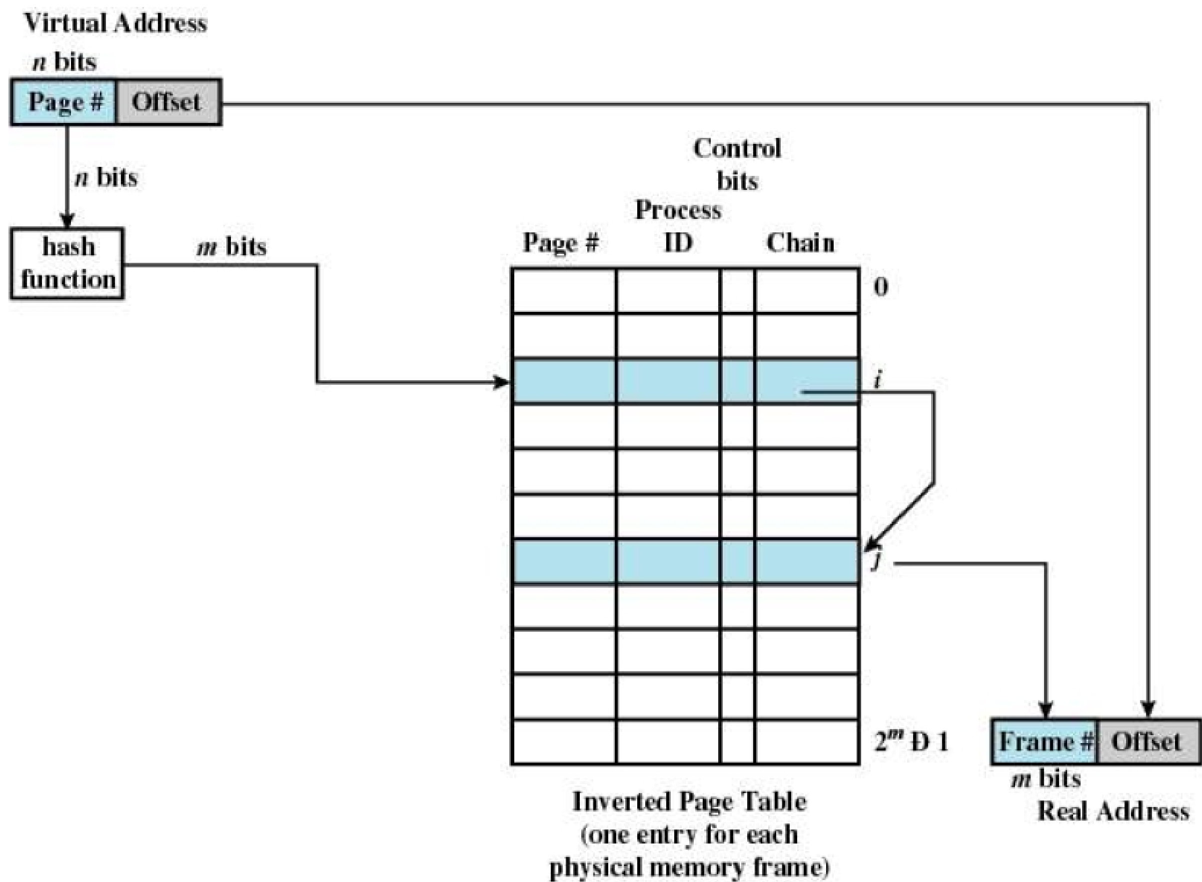
Es.: con indirizzi a 32 bit (spazio ind virtuali di 2^32=4GB) e pagine da 4 kbyte (4096byte=2^12 -->12 bit di offset nell'indirizzo quindi), abbiamo 2^20 pagine (poiché 32 bit è ind totale, 12 bit offset: 32-12=20 bit per indirizzare le pagine) , cioè 2^20 righe della tabella delle pagine: la PT ha 1 riga per ogni pagina della ram. Per mappare ciascuna pagina (pagetable -->frame ram) abbiamo bisogno di 4 (2^2) byte (i campi di ogni riga della page table), quindi per tutte le pagine avremo bisogno di 4x2^20 byte = 4 Mbyte, cioè: dimRecordPT*NumeroRecordPT=dimPageTable. Questa tabella di

4MB=4096byte occupa 4MB/4KB (cioè dimTabella/dimPagina) = 1024 pagine= 2^10 pagine può essere a sua volta. Posso mappare questa tabella in una tabella principale (root page table) che avrà una riga per ogni sua pagina, e quindi occupa (considerando ogni riga di 4 byte) 2^10x4 byte = 4KB;

Tabella di pagina invertita (IPT):



Anche questa tecnica serve a mitigare la dimensione delle Page Table, ma nelle ipt reali esiste un unica tabella delle pagine, alla quale si accede utilizzano il risultato di una funzione hash applicata al numero di pagina presente nell'indirizzo logico dato.



La tabella ha 4 campi:

1. numero di pagina
2. id del processo
3. i bit di controllo
4. campo di concatenazione.

Il numero del frame è il valore hash f calcolato. Tale informazione non è quindi tra i campi della tabella.
L'accesso alla inverted page table viene fatto per verificare che f sia un valore corretto.
Per ogni riga f della IPT (e quindi per ogni frame f) i campi sono *process id* del processo p a cui il frame f è assegnato, numero della pagina di p contenuta in frame f, chain (in caso di collisioni si scorre una lista di frames, control bits).
Difatti, la tabella funziona come una tabella hash concatenata. Nel caso in cui una riga è già utilizzata, l'ultimo campo rimanda ad un altra riga e cosi via, finché non si raggiunge quella cercata (che pertanto avrà il campo di concatenazione vuoto o nullo).

PRO: mantiene in memoria un numero di entry pari al numero di frame della memoria fisica (le entry contengono l'id della pagina). Si applica una funzione hash al numero della pagina per calcolare il numero del frame. L'accesso alla tabella può confermare che tale frame contiene la pagina cercata o meno. Le COLLISIONI sono gestite mediante la tecnica del chaining. Spesso è sufficiente un solo accesso a memoria.
CONTRO: in caso di collisioni si deve percorrere la catena. Il numero di accessi a memoria non è costante.

Aggiornare la IPT:
Lo fa il s.o.:

- calcolo frame h1 tramite hash leggendo riga ipt per h1;

- se h1 è libero aggiorno riga h1 nell'ipt: #page, #frame, pid, e setto chain=0, altrimenti scelgo un altra riga h2 (riapplicando hash) e vedo la stessa cosa: se h2 libero aggiorno riga h2 in ipt com sopra, ma poi setto chain =1 nella riga h1 di ipt, altrimenti continuo iterativamente (posso avere liste di collegamenti lunghe).

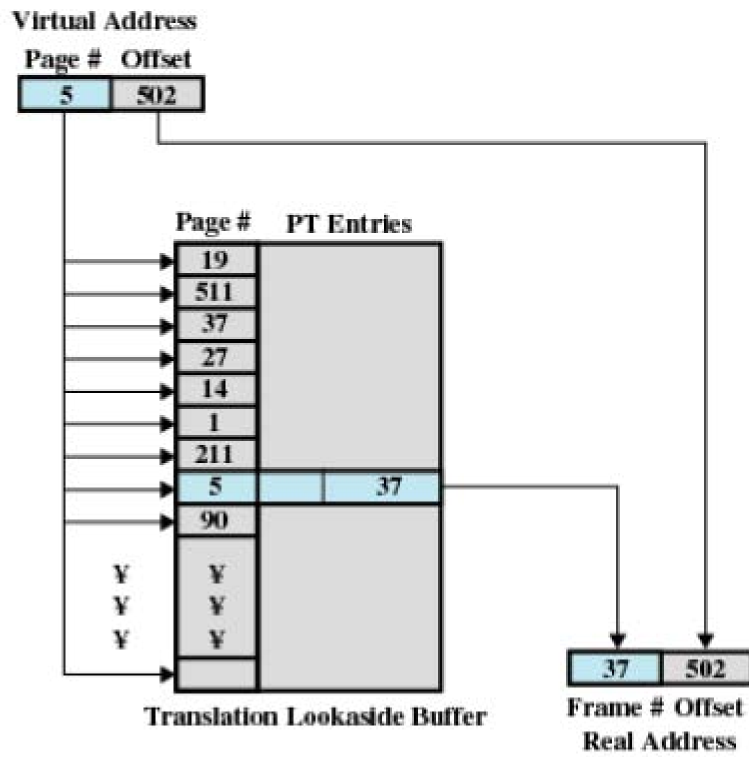
Leggere IPT:
Lo fa il s.o.:

- calcolo hash per h1;
- vedo riga h1 nell'ipt: se #page e pid sono corretti allora leggo il #frame e accedo a ram, altrimenti seguo il campo chain fino a trovare pid e #page corretti; se non li trovo vuol dire che la page non è in ram e va caricata -->faccio PAGE FAULT al so per comunicarglielo.

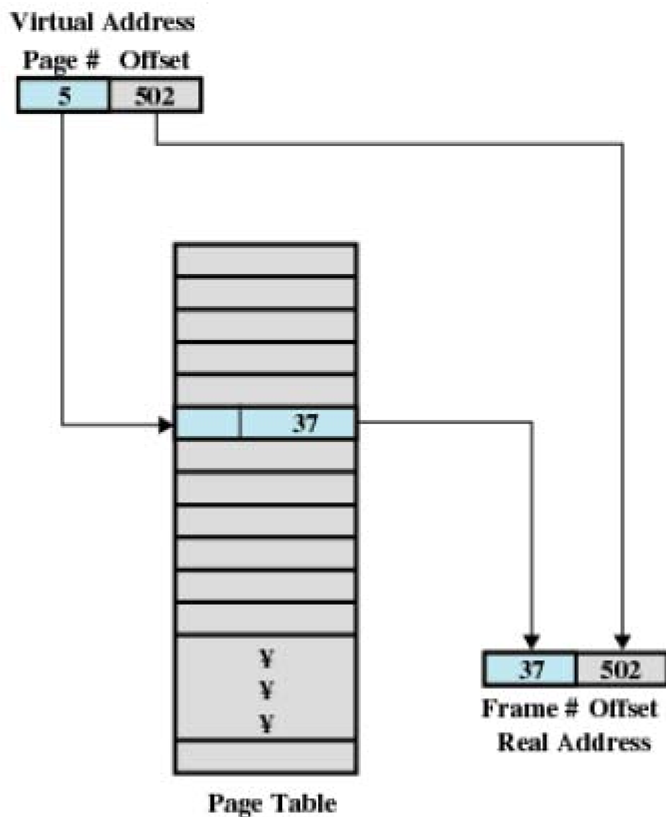
MAS E SAS:

Multiple addressing space e single addressing space. Nel modello MAS ciascun processo ha un proprio spazio di indirizzamento mentre nel modello SAS lo spazio di indirizzamento è uno solo. In altre parole, nel modello MAS lo stesso indirizzo virtuale ha un significato che dipende dal processo (spazi di indirizzamento separato) mentre in SAS lo stesso indirizzo ha sempre lo stesso significato, è quindi possibile accedere alla memoria degli altri processi con un semplice accesso semplificando la condivisione dei dati.
Le inverted page tables sono usate sopratutto in sistemi SAS.

Translation lookaside buffer (TLB):



(b) Associative mapping

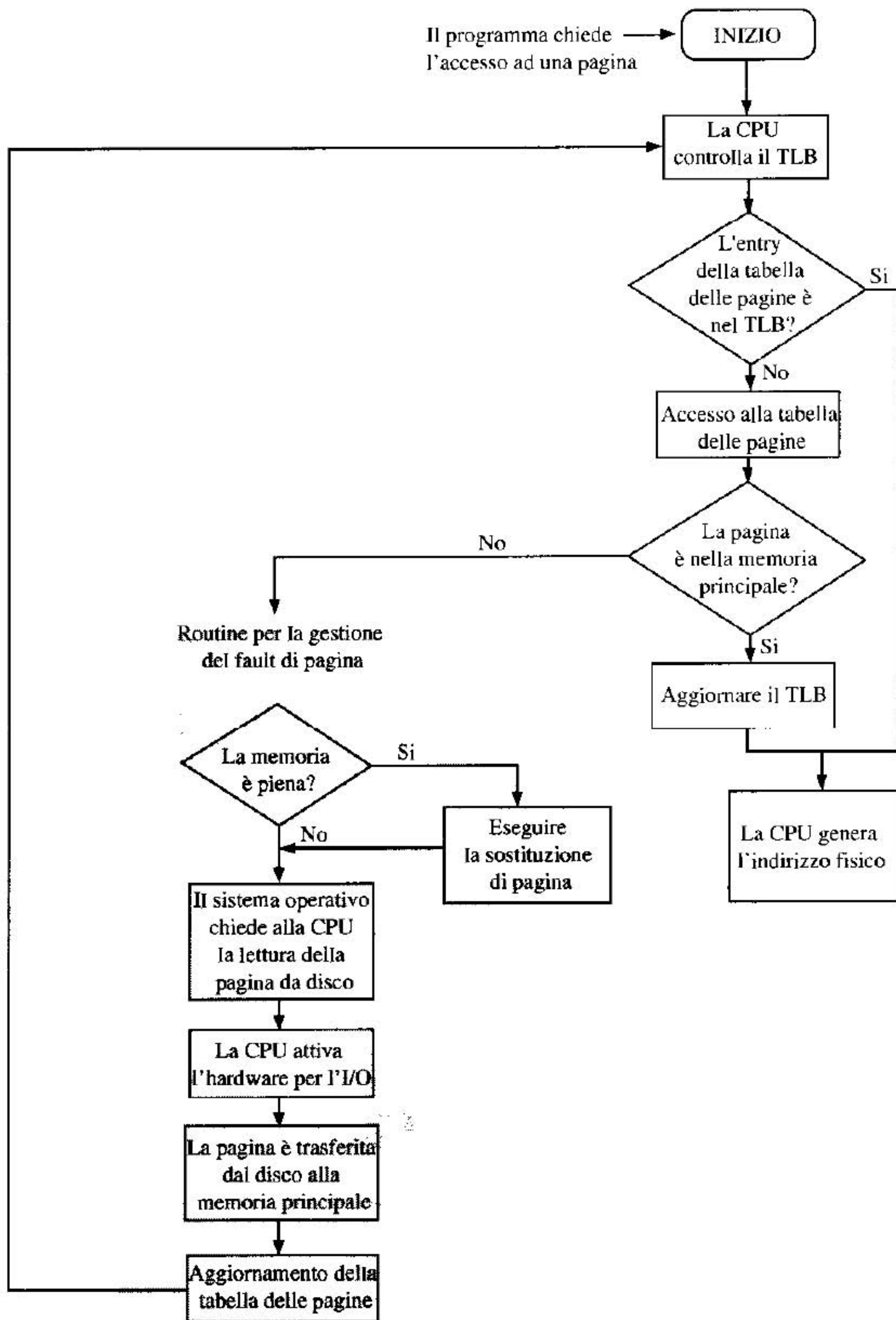


(a) Direct mapping

Si utilizza un buffer, per la precisione una cache associativa (cioè ogni entry nella tabella contiene sia il numero di pagina che l'entry completo della page table, in quanto non sono ovviamente presenti numeri di pagina disposti in maniera sequenziale o comunque ordinata in qualche modo) con le pagine più utilizzate di recente.

TRADUZIONE VIRTUALE->FISICO:

1. Dato ind virtuale(logico) cpu esamina tlb cache;
2. Se ce la riga pt ricavo il #frame e calcolo indirizzo fisico (reale) ram;
3. Se non c'è uso il #page nell'ind logico come indice nella ipt: se nella ipt la pagina risulta già in ram aggiorno tlb; altrimenti pagefault e so fa i/o per caricare pagina da disco (cpu nel frattempo dell'i/o può fare altro); se ram è piena si attiva alg di sostituzione pagina altrimenti aggiorno pt e ricomincio controllando tlb. Data la velocità delle memoria cache e la validità del principio di località, tale tecnica può determinare un aumento delle prestazioni.


Il programma chiede
l'accesso ad una pagina
INIZIO
La CPU
controlla il TLB
L'entry
della tabella
delle pagine è
nel TLB?
Si
No
Accesso alla tabella
delle pagine
La pagina
è nella memoria
principale?
No
Si
Routine per la gestione
del fault di pagina
Aggiornare il TLB
La memoria
è piena?
Si
No
Eseguire
la sostituzione
di pagina
La CPU genera
l'indirizzo fisico
Il sistema operativo
chiede alla CPU
la lettura della
pagina da disco
La CPU attiva
l'hardware per l'I/O
La pagina è trasferita
dal disco alla
memoria principale
Aggiornamento della
tabella delle pagine

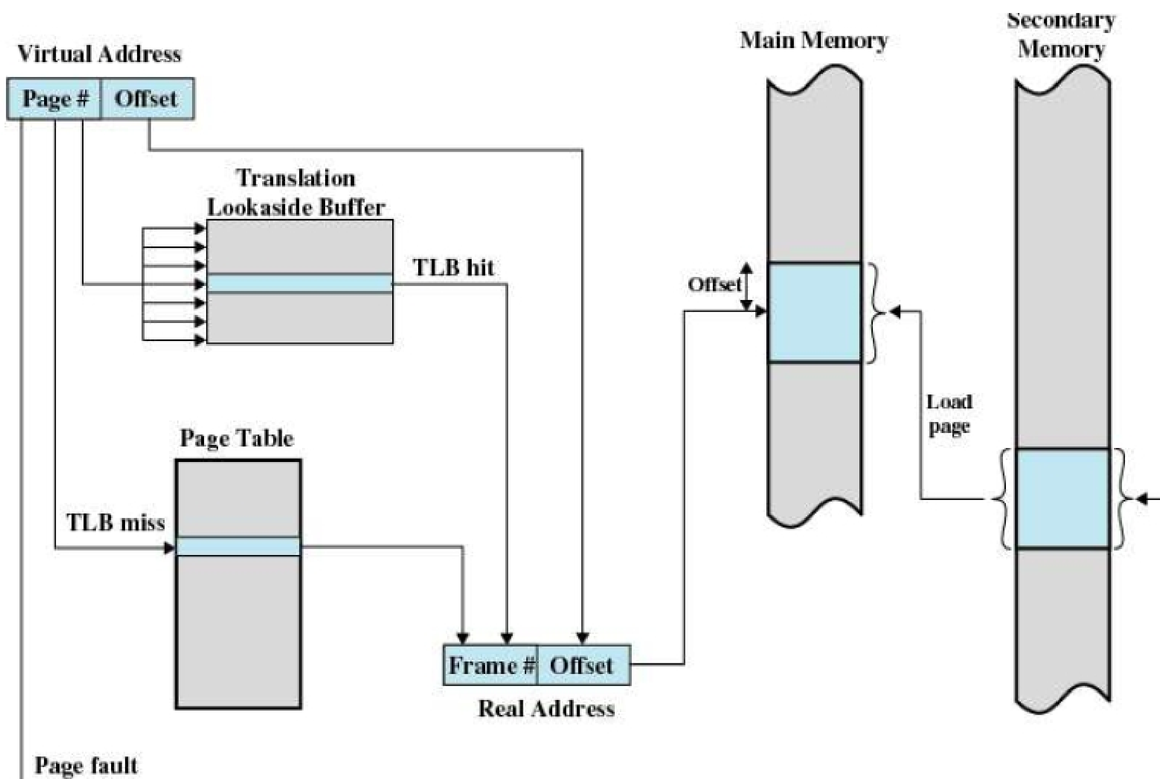


Figure 8.7 Use of a Translation Lookaside Buffer

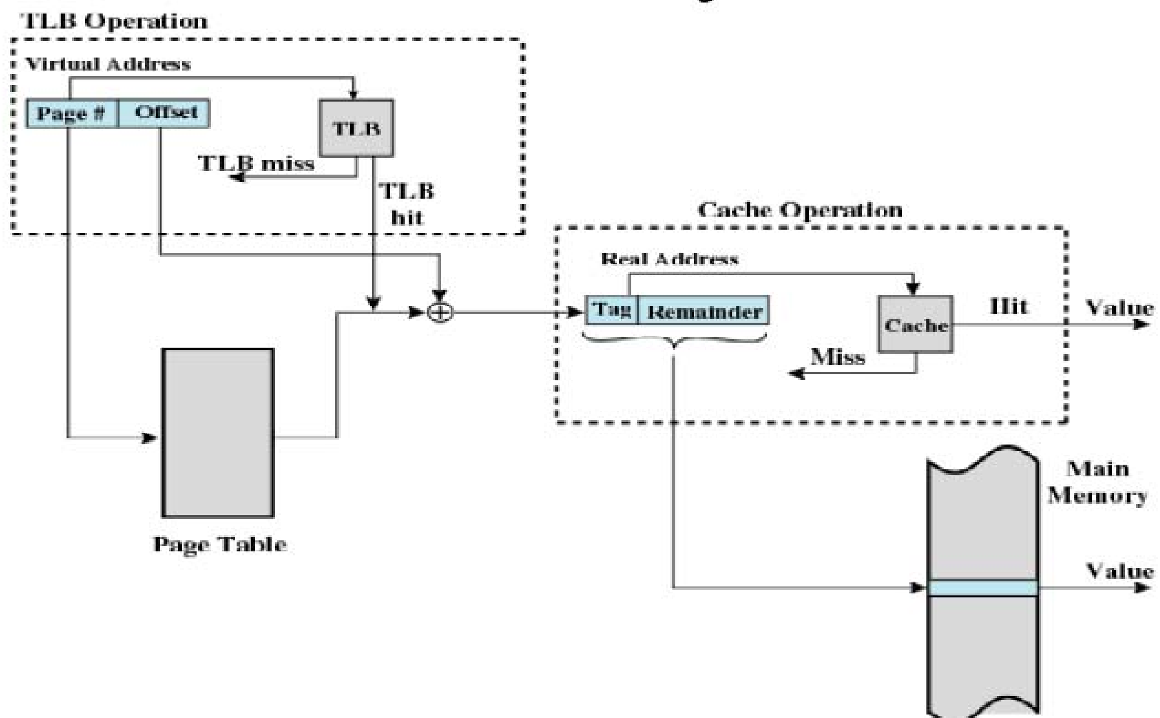
Indirizzo virtuale

Numero di pagina	Offset

Entry della tabella delle pagine

P	M	Altri bit di controllo	Numero di frame

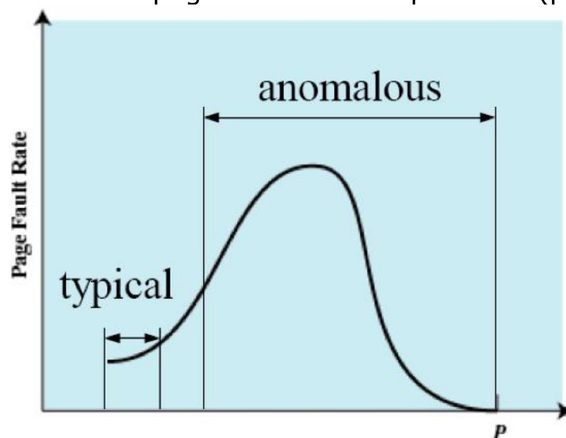
TLB E MEMORIA CACHE DELLA RAM:



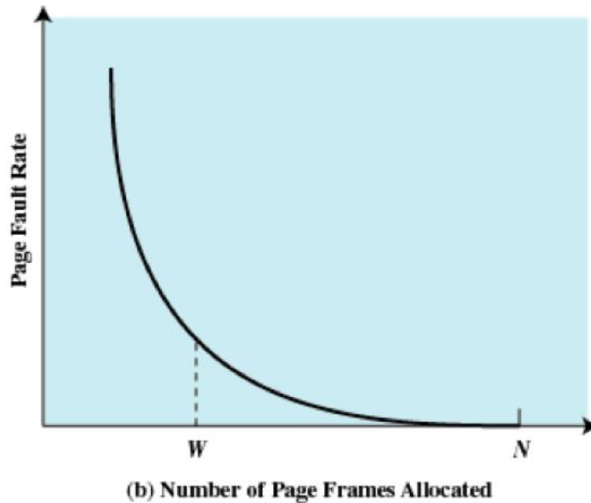
Quando si controlla nella pt se la pagina e già in ram (non era nella tlb) genero indirizzo fisico fatto da tag e resto e accedo alla cache ram per vedere se il blocco ram è già presente (accesso più veloce), altrimenti faccio come detto sopra (routine per la gestione page fault che carica pagina in memoria).

Dimensione delle pagine: fattori in gioco:

- frammentazione interna: se dim pagina troppo piccola ho meno framm interna ma un processo richiederà più pagine -->pt più grandi -->pt divise tra mem virtuale e ram -->doppio fault di pagina per un solo riferimento a memoria (caricamento riga pt e caricamento pagina).
- fault di pagina (si cerca pagina non ancora in ram): se pagina piccola ho molte pagine in ram quindi pochi fault, ma se pagina è grande ho poche pagine in ram e quindi molti fault. <u>Fault aumentano ad aumentare delle dim delle pagine</u> ma quando dim pagina=dim intero processo (punto p) pf diminuiscono.



Se invece ho dim pagine fisse i pf diminuiscono quando #pagine in ram cresce (w=dim working set):



- dim ram: se cresce ram cresce spazio indirizzi;
- tecnica di progr usata: tende a diminuire la località dei riferimenti a ram in un processo (obj oriented ha più moduli, + ogg con riferimenti sparsi, mentre multithread cambiano il flusso spesso e hanno rif sparsi);
- dim tlb: >>dim procesi --> <<località--> <<successo tlb --> tlb diventa collo di bottiglia.
    - soluz: usare tlb + grande ma non si può visto che dim ram cresce + in fretta;
    - potrei usare dim pagina più grandi, ma per i motivi suddetti non si può (+ fault..).

SOLUZIONE: uso + dimensioni di pagina: mappo thread in pagine piccole e mappo regioni contigue processo in poche pagine.

SEGMENTAZIONE + MEMORIA VIRTUALE

La segmentazione con supporto alla memoria virtuale presenta una combinazione dei vantaggi, in particolare:
- gestione semplificata di dati strutturati crescenti;
- ricompilazione e modifica dei programmi in maniera indipendente;
- facilitazioni nella protezione e nella condivisione;
    - No framm interna;
    - Ma frammentaz esterna si.

In maniera analoga alla paginazione, il sistema operativo usa per ogni processo (insieme di segmenti) la tabella dei segmenti che possiede lunghezza segmento, indirizzo iniziale segmento in ram (indirizzo fisico iniziale) e dei flags, quali bit di presenza di un segmento in memoria centrale, di modifica, di protezione o condivisione, di locking. La tabella deve stare in ram.

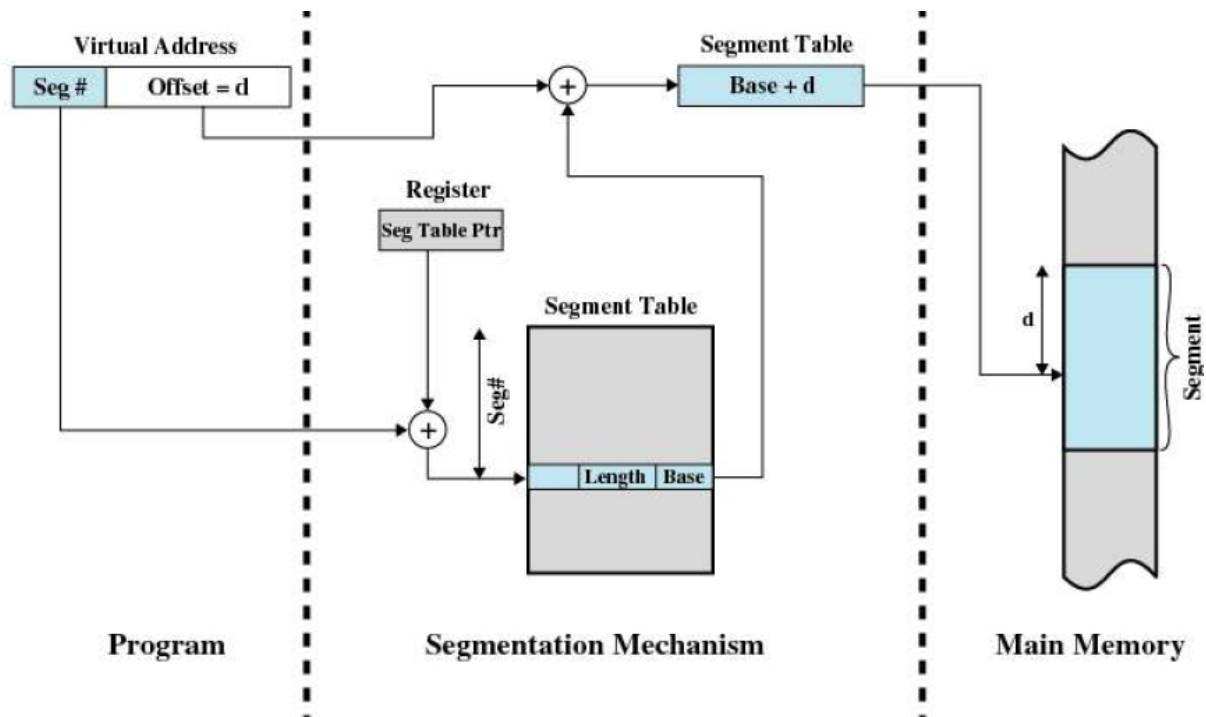


Figure 8.12 Address Translation in a Segmentation System

Indirizzo virtuale

Numero di segmento	Offset

Entry della tabella di segmenti

P	M	Altri bit di controllo	Lunghezza	Base del segmento

La ricostruzione dell'indirizzo fisico a partire da quello virtuale si ottiene anche in questo caso, come il precedente, sommando il numero di segmento virtuale con il valore del puntatore alla tabella dei segmenti, memorizzato solitamente in un registro, quindi utilizzando il resto dell'indirizzo virtuale come per la segmentazione classica (ind iniziale segmento + offset=ind fisico).

MEMORIA VIRTUALE con segmentazione paginata (SEGMENTAZIONE+PAGINAZIONE):

La segmentazione con memoria virtuale non è molto usata, mentre è più diffusa la combinazione di segmentazione, paginazione e memoria virtuale.
La paginazione permette:

- trasparenza al programmatore
- no problemi di frammentazione interna
- stesse dimensioni tra frame e pagine->si possono sviluppare algoritmi evoluti per la gestione della memoria.

La segmentazione permette:

- una sua visibilità al programmatore;
- gestione di strutture dinamiche;
- modularità;
- supporto per la condivisione e protezione.

In pratica i segmenti sono a loro volta divise in pagine, mentre per passare da un indirizzo virtuale ad uno fisico è necessario sfruttare le tabelle di segmentazione e di paginazione del processo nel seguente modo:
Una prima porzione dell'indirizzo virtuale fornisce il numero di segmento, da sommare al puntatore, presente in un registro, alla tabella dei segmenti; si ottiene l'ind iniziale della tabella delle pagine di quel segmento; sommo tale indirizzo al #di pagina e ottengo la riga della pt che mi dà il frame corrispondente; prendo il #frame dalla pt, lo concateno (paginazione) con l'offset dell'indirizzo logico e ottengo indirizzo fisico del frame in ram.

- tab segmenti: ogni riga ha length segmento, ind iniziale tab pagine, ma niente bit modifica (M) e presenza (P), che staranno a livello di pagina;
- tab pagine: ogni riga ha #pagina, #frame corrispondente, bit presenza, bit modifica.

PERMETTE:

- condivisione: con più riferimenti allo stesso segmento nella tab dei segm di processi diversi;
- protezione: con ind iniz segmento + offset vedo dim max segmento e controllo validità indirizzo.

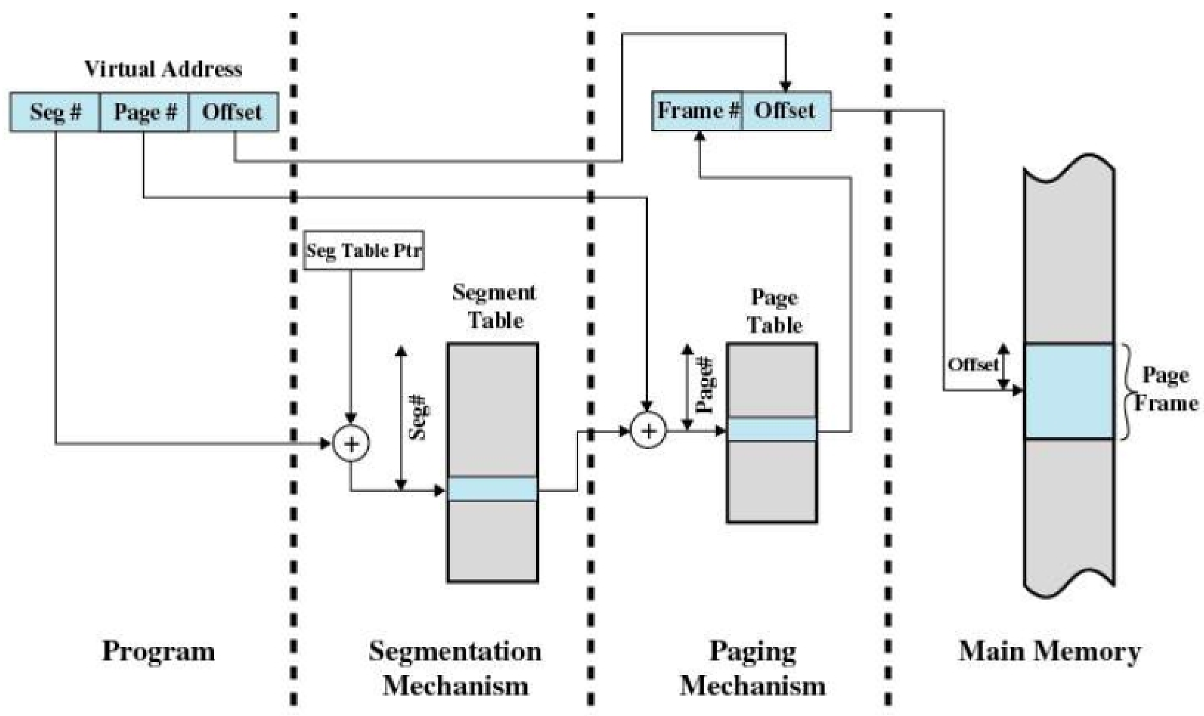


Figure 8.13 Address Translation in a Segmentation/Paging System

Indirizzo virtuale

Numero di segmento	Numero di pagina	Offset

Entry della tabella dei segmenti

Altri bit di controllo	Lunghezza	Base del segmento

Entry della tabella delle pagine

P	M	Altri bit di controllo	Numero di frame

(c) Segmentazione e paginazione combinate

RUOLO DEL S.O. NELLA GESTIONE DELLA MEMORIA

Il s.o. deve assolvere 5 compiti principali inerenti la gestione della memoria:

1. isolamento dei processi, per evitare interferenze tra più procesi;
2. allocazione e gestione automatica della memoria;
3. supporto per la programmazione modulare;
4. protezione e controllo dell'accesso;
5. memorizzazione a lungo termine.

Come vengono assolti questi compiti?
Bisogna valutare 3 fattori:

1. Possibilità di usare la memoria virtuale (ormai comune in tutti i sistemi);
2. Scelta della tecnica di gestione, paginazione, segmentazione o entrambe: la segmentazione pura è molto rara, pertanto in seguito parleremo di pagine, o intese come pagine di un sistemazione di paginazione o come pagine di un sistema misto segmentazione/paginazione;
3. Algoritmi utilizzati per la scelta delle pagine da sostituire. Compito del sistema operativo è quello di ottimizzare le prestazioni e ridurre al minimo gli errori di pagina.

Ovviamente, i fattori che influiscono sulle prestazioni sono numerosi, e non è possibile dare una risposta definitiva, ma solo analizzare le soluzioni proposte e le relative problematiche.

POLITICHE DI FETCH:

Determinano quando una pagina deve essere caricata in memoria.
Sono principalmente due:

- pagine su richiesta (demand paging) con la quale carico in ram solo quando si fa un riferimento a tale pagina (avvio processo e avrò tanti fault, carico pagine appena le chiedono (dopo ogni fault) e poi ho sempre meno fault;
- precaricamento delle pagine, cioè quando un insieme di pagine vicine a quella di interesse vengono caricate subito, anche se non sono utilizzate immediatamente, sperando che per il principio di località serviranno in seguito (il vantaggio di questa tecnica non è stato dimostrato).
    - carico pagina che ha generato fault + altre pagine: se pagine in memoria in aree contigue carico le pagine contigue insieme invece che 1 per volta 1 dopo l’altra;

  - prepagino all’avvio del processo scegliendo le pagine desiderate oppure prepagino ad ogni fault (meglio perché non è trasparente al programmatore).

POLITICHE DI POSIZIONAMENTO (PLACEMENT):

Determina dove deve risiedere un pezzo di processo nella memoria reale. Per un sist che usa la paginazione pura o la paginazione combinata a segmentazione, generalmente questo è un fattore rilevante. Sono quelle viste in precedenza, bestfit e firstfit in particolare. Laddove si usi la segmentazione, da sola o combinata, la scelta dell'algoritmo è pressoché indifferente, tranne che nei sistemi multiprocessore, dove tali politiche sono influenzate da fattori come la distanza fisica tra memoria centrale e processore.

POLITICHE DI SOSTITUZIONE (REPLACEMENT):

È necessario innanzitutto dire che molti fattori intervengono nell'argomento:
- quante pagine sono attive per un processo (dim resident set);
- quale insieme di pagine considerare per la sostituzione: solo quelle del processo o tutti i frames (ambito locale/globale del resident set).
- quale particolare pagina sostituire (la trova un qualche algoritmo tra quelli che vedremo.

Questi elementi suddetti, solitamente, riguardano la gestione del cosiddetto resident set, cioè l'insieme delle pagine di un processo attualmente presenti in memoria centrale.
ALGORITMI:
Tutte le strategie di sostituzione hanno come obiettivo la sostituzione della pagina che verrà richiesta il più lontano possibile nel tempo (ovvero la pagina a cui ci si riferirà di meno nel prossimo futuro, la cui rimozione provocherebbe quindi un minor numero di page fault). La maggior parte delle strategie cerca di prevedere l'andamento futuro delle richieste di pagina sulla base degli andamenti passati. La scelta dell'algoritmo tra quelli che vedremo di seguito è molto importante, perché un buon algoritmo, cioè che seleziona la pagina effettivamente migliore, può richiedere d'altra parte un carico di lavoro eccessivo per il processore. FRAME IN BLOCCO: frame che sono bloccati in mem principale e che non possono essere sostituiti. Ad es quelli destinati ai buffer dell'I/O in memoria, a buona parte del kernel, alle strutture di controllo.

ALGORITMI DI SOSTITUZIONE PAGINE:

Optimal Policy

Quello ideale, da usare come metro di paragone: seleziona per la sostituzione la pagina alla quale ci si riferirà dopo il tempo più lungo (vedi, tra quelle nei frame, quella a cui ti riferirai più avanti (a dx nella lista riferimenti)).
Ovviamente irrealizzabile (anticausale), può essere usato solo come metro di paragone tra le strategie realizzabili che seguono;
Optimal: fault ad ogni accesso? Una tale stringa non esiste. Considera una stringa di riferimenti che inizia con tutte pagine diverse fino alla richiesta i-1. La richiesta i-sima è la prima richiesta che viene fatta ad una pagina già richiesta in precedenza. Chiamiamo p tale pagina. La richiesta precedente a p è stata fatta all'istante j<i. Vogliamo dimostrare che all'istante i-esimo non c'è mai page fault, cioé che all'istante i-esimo p è già in memoria, cioè che dall'istante j+1 in poi p rimane in memoria, cioè che optimal sceglie sempre una pagina diversa da p negli istanti j+1...i-1.
Consideriamo le scelte che optimal fa su sulle richieste j+1...i-1. Optimal sostituisce la pagina per cui il prossimo istante in cui sarà referenziata è il più lontano nel tempo (le pagine non più referenziate sono considerate come infinitamente distanti nel tempo e quindi vengono sostuite prima delle altre). Nota che le pagine in memoria subito prima

del riferimento i-esimo sono un sottoinsieme delle pagine riferite agli istanti 1...i-1. Per tali pagine, tranne per p, il successivo istante di riferimento è sempre strettamente maggiore di i poiché la pagina p è la prima ad essere referenziata per la seconda volta, quindi optimal preferisce sempre una pagina che è diversa da p e p rimarrà in memoria.
fine lezione - Daniele Palladino 30/11/2009 11:14

Least Recently Used (LRU)

Least recently used (osserva il passato e prevede il futuro) approssima opt.
Sostituisce la pagina scegliendo quella non referenziata da più tempo, la più lontana "verso sinistra" nella lista dei riferimenti. Ad ogni riferimento ordina il resident set mettendo a sinistra l'ultima acceduta e a destra la più vecchia (uscirà la più vecchia a destra quando hai page fault).
Questa politica risulta molto vicina a quella ottima, ma non va bene quando ho intervalli di spazi di indirizzo percorsi ripetutamente in sequenza (percorro memoria più volte in una stesa direzione, avanti e indietro)
-->lru scarica le pagine nello stesso ordine in cui sono richieste -->infiniti page fault.
Richiede un enorme carico di lavoro al sistema, in quanto deve sia avere un modo (timestamp) per stabilire quando una pagina è stata referenziata, sia confrontare le pagine tra loro per trovare quella non referenziata da più tempo.
SOLUZIONE:
Variante di MRU (most recently used). Cerca l'occorrenza di lunghe sequenza di riferimento a pagine;
Regole:

1. Sequenza=serie di fault ad indirizzo virtuale consecutivi crescente in una direzione, l'ultima pagina aggiunta è la testa di tale sequenza;
2. Per sostituire una pagina cerco seq lunga >L; esamino tempo degli N fault+recenti in ogni sequenza; scelgo sequenza con N-esime fault più recente;
3. Scelgo la prima pagina in ram che sia M o + pagine distante dalla testa; vanno bene L=20, N=5, M=20. LRU=WS? stringhe di riferimenti su cui LRU con F frames da gli stessi page fault di WS con finestra delta=F: "le strighe in cui |W(t,delta)| non varia ed è sempre pari a F al variare di t". F=delta =3 1 2 3 1 2 3 è nella classe 1 2 1 3 1 2 1 3 1 non è nella classe; il secondo accesso alla pagina 2 genera un fault con WS e non lo genera con LRU (sottolineati gli elementi che ricadono nella finestra temporale, di lunghezza delta, nel momento del fault).

FIFO

Tratta le pagine come se appartenessero ad un semplice buffer circolare basato sulla ben nota politica del FIFO (prima-entri-prima-esci).
Si toglie la pagina che è da più tempo in ram (osservi i resident set precedenti e non la lista dei riferimenti, togliendo quella che è presente in più RS di fila (a sinistra)).
Serve un puntatore che va in cerchio lungo i frame del processo. Questa politica, molto semplice da gestire e da realizzare, ha comunque una sua logica, in quanto si può pensare che una pagina che è entrata da molto nel buffer man mano che passa il tempo non serva più.
Non tiene conto della storia passata delle pagine. **FIFO = CLOCK?**

- 3 frame e 4 pagine;
- Dai una classe di stringhe di riferimenti su cui FIFO da gli stessi page fault di CLOCK:
    - Dai una classe che genera un fault ad ogni accesso;
    - Prova quindi a dare una classe che NON genera un fault ad ogni accesso;
    - Una stringa con un fault ad ogni accesso sia per FIFO che per CLOCK è 1 2 3 4 1 2 3 4 .... Per non avere un accesso senza fault bisogna far riferimento ad una pagina residente ma grazie all'uso dello "use bit" questo normalmente fa sì che CLOCK si comporti meglio di FIFO. La

classe può essere la seguente "le stringhe in cui dopo un fault si referenziano tutte le pagine residenti " Infatti se tutti i frame hanno use=1 CLOCK sostituisce la pagina nel frame attualmente puntato dalla lancetta e muove la lancetta al rossimo frame, cioè si comporta come FIFO. Nella seguente espressione regolare gli accessi sottolineati sono quelli che mettono use=1. 1 2 3 (4 2 3 4 1 3 4 1 2 4 1 2 3 1 2 3)* F F F F1 F2 F3 F4 Per tale classe di stringhe di riferimenti FIFO e CLOCK sostituiscono le stesse pagine..

FIFO dà stessi fault di CLOCK (senza dare un fault ad ogni richiesta, e considerando 4 frame e 5 pagine) se uso una sequenza del tipo: varie soluzioni possibili ad esempio 1 2 3 4 (5esce 1 2 3 4 5 1esce 2 3 4 5 1 2esce 3 4 5 1 2 3esce 4 1 2 3 5 4esce 5 1 2 3 4)* . in alternativa (1 2 3 4 5 4 3)*.

FIFO=LRU? (CON 4 PAGINE E 3 FRAME):

Stringhe di riferimenti che danno gli stessi page fault sia per FIFO che per LRU sostituendo le stesse pagine:

- Dai una classe che genera un fault ad ogni accesso: "la classe delle stringhe che non fanno mai un accesso ad una pagina che è già residente". Cioè ciascuna pagina è acceduta solo quando abbiamo il page fault che la ha caricata. Infatti in questo caso LRU mantiene una coda identica a FIFO. esempio 1 2 3 4 1 2 3 4 ... F F F F1 F2 F3 F4 F1 .... (Fn indica che è stata sostituita la pagina n) che può essere espressa come (1 2 3 4)*
- Prova quindi a dare una classe che NON genera un fault ad ogni accesso: possiamo inserire delle sottostringhe tra un fault e l'altro che non cambino l'ordine, ad esempio 1 2 3 2 3 1 2 3 1 2 3 4 1 2 3 4 ... F F F F1 F2 F3 F4 F1 .... Un esempio della classe richiesta è (1 2 3 ((1|2|3)* 2 3) 4)* dove ((1|2|3)* 2 3) lascia invariato l'ordine all'interno della coda LRU e non genera page fault.

STRINGA DI RIF CHE GENERA FAULT AD OGNI PAGINA: 1 2 3 4 1 2 3 4... da 4 in poi ciascun riferimento genera un page fault che sostituisce un'altra pagina. Viene sostituita sempre la prossima pagina che verrà referenziata. NUMERO MINIMO DI PAGINE: Almeno devo avere 4 pagine (con 3 frame) altrimenti avrei solo i primi 3 fault.

CLOCK (second chance)

Tenta di emulare la politica LRU,approssimandolo.
Utilizza un ulteriore bit per frame, detto bit d'uso. Una volta che una pagina è caricata in memoria, il bit viene posto pari ad 1, quando poi è necessario trovare la pagina da sostituire, l'algoritmo cerca tra le pagine sostituibili una pagina avente bit pari a 0 e, contemporaneamente, pone a zero tutte le pagine con bit pari ad 1 che incontra.
In questo modo, se nel primo giro non trova nessuna pagina con bit pari a zero, ne troverà sicuramente una nel giro successivo. Il puntatore punta sempre al frame successivo, ma se la pagina è già in ram, setti il suo usebit=1 (se era a 0) e il PUNTATORE RESTA FERMO dove era alla posizione precedente.
Le prestazioni di questo algoritmo sono intermedie tra quelle dell'LRU e del FIFO.
Cioè':

- -se devi sostituire:
    - scorri e cerchi usebit=0 settando quelli =1 in =0;
    - sostituisci e al nuovo metti bit=1 e PUNTATORE va nella posizione successiva a quella dove inserisci;
- -se la pagina già è in ram:
    - setti suo usebit=1 se è =0, ma puntatore resta sempre fermo alla posizione precedente!!

VARIANTE CON BIT DI MODIFICA

È possibile aggiungere un ulteriore bit di modifica, in modo da avere le 4 combinazioni (bit uso-bit modifica):

1. non usato di recente e non modificato;

2. usato di recente e non modificato;
3. non usato di recente e modificato;
4. usato di recente e modificato.

A questo punto l'algoritmo cerca prima una pagina non usata e non modificata, non trovandola, ne cerca un'altra non usata e modificata. La prima che trova la seleziona e contemporaneamente imposta a 0 i bit d'uso dei frame che incontra, se anche in questo caso non trova nulla, riparte, avendo nel passaggio precedente impostato tutti i bit d'uso a 0.
Al più si compiono 4 giri del buffer:

1. cerca frame con u=m=0; non modifica i bit;
2. in caso di fallimento cerca frame con u=0,m=1;
    - durante la scansione pone gli use bit a 0;
3. in caso di fallimento si ripete il passo 1.

CLOCK=FIFO?
Se non usi bit hai proprio FIFO. Una stringa con un fault ad ogni accesso sia per FIFO che per CLOCK è 1 2 3 4 1 2 3 4 ....
Per non avere un accesso senza fault bisogna far riferimento ad una pagina residente, ma grazie all'uso dello "use bit" questo normalmente fa sì che CLOCK si comporti meglio di FIFO.

AGING (politica dell'età)

Basato sulla politica LRU, mantiene uno stimatore/contatore di età per ogni processo: minore è il valore di tale contatore, maggiore è l'età del processo (cioè il tempo che risiede in memoria).
Periodicamente tutti gli use bit delle pagine sono visitati e i relativi stimatori aggiornati (o incrementando i contatori per le pagine usate e decrementandoli per quelle non usate, oppure facendo uno shift a destra di ogni stimatore e sommando al bit più significativo il bit di uso). L'età dei processi e il tempo risultano essere in questo modo quantizzati. Inoltre, anche se una pagina è referenziata molte volte in un periodo, essa risulterà usata sempre una volta.
DIFFERENZA CON LRU: I riferimenti ai processi molto "vecchi" sono dimenticati (LRU tiene più informazione), perché una volta raggiunto lo zero il contatore non può essere cambiato ulteriormente;
FUNZIONAMENTO: ogni quanto di tempo (sweep) controlla lo stimatore associato ad ogni pagina, si rimuove la pagina con stimatore più basso (maggiore è l'età=t in memoria nell'RS).
COME AGGIORNO STIMATORE:

- Uno stimatore per ogni pagina;
- In ogni sweep cerco quello con valore più basso;
- Controllo use bit (cioè se nello sweep precedente ho usato quali pagine) e se pagina i è usata (usebit=1) metto a sx del suo stimatore un 1 (es 1000-->1100) shiftando;
    - se invece non ho usato quella pagina nello sweep precedente (usetbit=0) metto uno 0 a sx del suo stimatore (1000-->0100);
- selezione la pagina da rimuovere: con stimatore più basso.

ALL'ENTRATA DI UNA NUOVA PAGINA: non so la storia di tale pagina,quindi setto il suo stimatore a un valore alto (es 11111) cosi rimane in ram per molto tempo e se non la uso ci metterà poco a scendere di valore (aggiungo zeri).

PAGE BUFFERING

Variante della politica FIFO.
Una pagina da sostituire non è persa ma è inserita opportunamente in coda (a seconda che sia stata modificata o meno) ad una delle due liste di pagine presenti in memoria centrale, la lista delle pagine libere e la lista delle pagine modificate. In pratica quindi la

pagina suddetta non è spostata fisicamente: la sua entry nella tabella delle pagine è trasferita nella lista delle pagine libere o nella lista di quelle modificate. Per migliorare le prestazioni, il sistema cerca di mantenere sempre un piccolo numero di pagine libere.
Nel momento in cui è necessario caricare in memoria centrale una pagina:

- si effettua la ricerca di una pagina da sostituire; la pagina individuata viene inserita a seconda che sia stata modificata o meno in coda alla lista opportuna;
- viene rimossa la pagina in testa alla lista usata e utilizzato il frame libero così ottenuto.

In questo modo le liste di pagine costituiscono una sorta di buffer dove si trovano le pagine che possono essere sostituite e dove tali pagine sostituibili stazionano (senza quindi essere rimosse immediatamente) per il tempo necessario ad arrivare in testa alla lista. Nel caso in cui una pagina presente in una lista sia di nuovo necessaria, questa può essere richiamata immediatamente, incrementando le prestazioni.
Quando una pagina non modificata sta per essere sostituita in realtà rimane in memoria. Quindi se il processo fa di nuovo riferimento alla pagina in questione, questa pagina in realtà è ancora in memoria, ed il suo caricamento è in pratica immediato. Inoltre la lista delle pagine modificate è scritta su disco tutta insieme: in tal modo si riduce il numero di accessi al disco.
I vantaggi rispetto all'accesso al disco in lettura:

- quando avviene un page fault se la pagina e' nel page buffer il page fault non provoca accesso al disco (minor page fault)

I vantaggi rispetto all'accesso al disco in scrittura:

- Le pagine modificate (dirty) quando vengono tolte dal resident set devono essere scritte su disco. Tale scrittura puo' essere rimandata se il frame non deve essere usato immediatamente ed eseguita dal disco come una attivita' a bassa priorita', ad esempio nei momenti in cui il disco non ha altre richieste da servire e ottimizzando il movimento della testina poiche' sappiamo gia' le zone di disco che dobbiamo scrivere. Inoltre potrebbe accadare che la pagina venga richiesta prima della sua scrittura.

fine lezione -Daniele Palladino 09/12/2009 11:13

GESTIONE DEL RESIDENT SET

Insieme delle pagine di un processo residenti in memoria centrale
L'uso della memoria virtuale permette di eseguire processi senza che siano presenti in memoria centrale tutte le pagine di ciascun processo. Il sistema operativo può scegliere quante pagine assegnare ad un processo e come. Intervengono diversi fattori:

- meno pagine per processo --> più processi in memoria >meno swapping per caricare nuovi processi;
- meno pagine per processo --> maggiore probabilità di page fault ->maggiore swapping per caricare le pagine utilizzate da ciascun processo;
- oltre un certo valore, il numero di pagine di un processo presenti in memoria centrale non ha particolari effetti sulla frequenza dei page fault.

RESIDENT SET:

insieme delle pagine di un processo che sono in ram in un certo momento. La sua dimensione implica il numero di page fault che ci sono e dipende dalla dimensione delle pagine.
I sistemi operativi adottano pertanto DUE POLITICHE:

1. allocazione fissa: la quantità di pagine da assegnare ad un processo viene stabilita quando il processo viene creato e rimane costante per tutta la vita del processo stesso; ad ogni fault una pagina deve lasciar posto a quella mancante.

2. allocazione variabile o dinamica: il numero di pagine di un processo presenti in memoria varia nel tempo a seconda delle esigenze del processo stesso.

CONTRO: Presenta un maggior overhead per il sistema che deve sempre monitorare i processi attivi, inoltre richiede specializzazione dell'hardware. Queste politiche sono inoltre ambo influenzate dal tipo di strategia delle pagine da sostituire (replacement strategy), che può essere locale o globale, a seconda che le pagine da sostituire siano selezionate tra quelle del processo stesso o tra tutte quelle presenti in memoria centrale (escluse quelle in stato di "locked" per qualche motivo, come l'uso da parte del kernel).

TRE TIPI DI STRATEGIE (e non 4, in quanto l'allocazione fissa con strategia di sostituzione globale è impossibile)

IMPORTANTE LA NOZIONE DI AMBITO DI SOSTITUZIONE che può essere:

- locale: sceglie la pagina da sostituire tra quelle residenti in memoria del processo che ha causato il fault, è ovviamente l'unica scelta per l'allocazione fissa, ma può essere adottata anche da quella variabile;
- globale: considera tutte le pagine non bloccate in memora principale come candidate alla sostituzione; può sicuramente essere adottata da una politica di allocazione variabile.

1. allocazione fissa ambito locale: in questo caso il processo ha a disposizione un numero fisso di frame allocabili (dim rs) stabilito in base a vari fattori come il tipo di applicazione; si sostituisce una pagina dello STESSO processo tramite gli algoritmi visti prima.
    - **SVANTAGGI**:
    - se l'allocazione è troppo piccola, allora ci saranno molti fault di pagina, con conseguente rallentamento dovuto all'I/O su disco;
    - se troppo grande ci saranno pochi processi in memoria, con un utililzzo quindi meno efficiente del processore.

- allocazione variabile ambito globale: questa politica è la più semplice da realizzare, sebbene presenti lo svantaggio che, nella scelta delle pagine da sostituire, opera su tutta la memoria (ad eccezione dei frame ram locked: es kernel), con il rischio non calcolabile di selezionare una pagina molto importante per un processo che sarà eseguito subito dopo;
    - **FUNZIONAMENTO**: Il s.o. tiene una lista di frame liberi. Quando avviene un page fault:
        - un frame libero viene aggiunto al RS del processo causante il fault, e la pagina richiesta viene caricata;
        - se non ci sono frame liberi, ne viene scelto uno globalmente tra quelli non bloccati, sulla base delle precedenti considerazioni.
    - **Conseguenza**: il RS di un processo sarà diminuito, ma non c'è un criterio per stabilire quale processo perderà una pagina. Si limitano gli svantaggi introducendo il page buffering.
- allocazione variabile ambito locale: questa tecnica è simile alla numero 1, con la variante che, di tanto in tanto, il sistema valuta in qualche modo se il numero di pagine assegnato al processo è scarso, sufficiente o eccessivo, agendo di conseguenza nel caso di uno sbilanciamento per difetto o per eccesso: la dim del rs è variabile.
    - **PASSI:**
        - a) quando si carica in memoria un processo,gli vengono allocati alcuni frame, il cui numero dipende da vari fattori(tipo processo,config della macchina,richieste del programma,carico corrente..);
        - b) quando avviene un fault si seleziona la pagina da sostituire tra quelle del RS del processo che ha causato il fault.
        - c) periodicamente si rivaluta l'allocazione fornita al processo e la si aumenta o decrementa per migliorare le prestazioni globali. I suoi punti cardine sono quindi la rivalutazione dell'allocazione per ogni processo, e la periodicità della rivalutazione. Fondamentale, per quest'ultima politica (che risulta essere la più utilizzata), è il concetto di working set.

WORKING SET

L'insieme delle pagine in memoria centrale di un processo che, all'istante t, sono state referenziate in un precedente intervallo delta (che rappresenta una sorta di finestra temporale).
T virtuale: si consideri una seq di riferimenti a memoria generati da un processo P. Siano r(1), r(2).. tali accessi. r(i) è la pagina contenente l'i-esimo indi referenziato da P. t=1,2,3... è chiamato tempo virtuale per il processo P. Maggiore è la finestra delta,maggior o uguale è il WS--> È utilizzata una funzione in due variabili, W(t,delta). Osserviamo che il tempo è inteso come numero di indirizzi virtuali referenziati sequenzialmente: 1,2,3... indirizzi referenziati rappresentano 1,2,3...unità di tempo (non il valore dell'indirizzo, ma il fatto di essere un indirizzo: cioè, due indirizzi referenziati uno dopo l'altro, rappresentano due unità di tempo...).
La RELAZIONE TRA WS E DELTA: ws è legato alla finestra delta, con la relazione:

$$W(t,\Delta) \subseteq W(t,\Delta+1):$$

+ grande è delta + grande è ws.
La DIMENSIONE DEL WS (CARDINALITA'): invece è compresa nell'intervallo: [1,min(delta,N)] con N numero di pagine totali del processo. Si osserva che, per una finestra fissata, la dimensione del working set varia a seconda delle esigenze, mentre che all'aumentare della finestra, la dimensione del working set cresce tendendo a una soglia pari a N.

GESTIONE RESIDENT SET TRAMITE WORKING SET

STRATEGIA TEORICA (WS=RS):

A livello ideale sarebbe conveniente porre il resident set pari al working set, quindi il working set può essere utilizzato come strategia per gestire la dimensione del resident set nel seguente modo:

1. monitoraggio del working set di ogni processo;
2. rimozione periodica dal resident set delle pagine che non si trovano nel working set. Questa è una politica LRU;
3. eseguire un processo solo se il suo working set è in memoria centrale.

PROBLEMI CORRELATI: Questa strategia è comunque non applicabile direttamente perché presenta problemi di varia natura, dovuti al fatto che non è possibile fare previsioni efficaci per il futuro basandosi sul passato, oltre alle problematiche tipiche delle politiche LRU (overhead).
SOLUZIONE: In alternativa, invece di monitorare direttamente il working set, si lavora con la frequenza dei page fault, con algoritmi simili al

PAGE FAULT FREQUENCY ALGORITHM:

In breve PFF si basa sull'ipotesi che la frequenza di page faults varia in modo inversamente proporzionale al variare della dimensione del resident set. Si fissa un valore "ottimale" f della frequenza di page faults (che assumiamo corrispondere alla situazione |RS|=|WS|). Ad ogni page fault si stima la PFF, se PFF>f si aumenta |RS| se PFF<f si diminuisce |RS|. Il pregio di PFF è che è facilmente implementabile.
Svantaggio: PFF richiede molta memoria durante i transitori tra vari "regimi di località".
DETTAGLIO PFF: Come detto, se il resident set di un processo è > del suo working set, si avrà una bassa percentuale di faults, e viceversa.
Si può sfruttare questa idea tramite l'uso di una frequenza di soglia P:

- se PFF(page faults frequency) è > di P, si aumenta il resident set;
- -viceversa si diminuisce il RS.

Il PFF viene stimato nel modo seguente:

1. si mantiene un contatore t dei riferimenti a memoria;
2. ad ogni page faulti si stima il nuovo PFF
   - $PFF_{now} = \alpha + \frac{1}{t - t_1} + (1 - \alpha) \cdot PFF_{precedente}$
     $\alpha \in (0,1]$
3. si decide l'azione da intraprendere.

Questa strategia migliora con l'adozione del PAGE BUFFERING, d'altro canto essa peggiora durante i periodi di transizione di località: in questa fase avvengono numerosi page fault che portano il processo ad aumentare il proprio RS a scapito di altri procesi, che possono anche venire sospesi e spostati su disco.

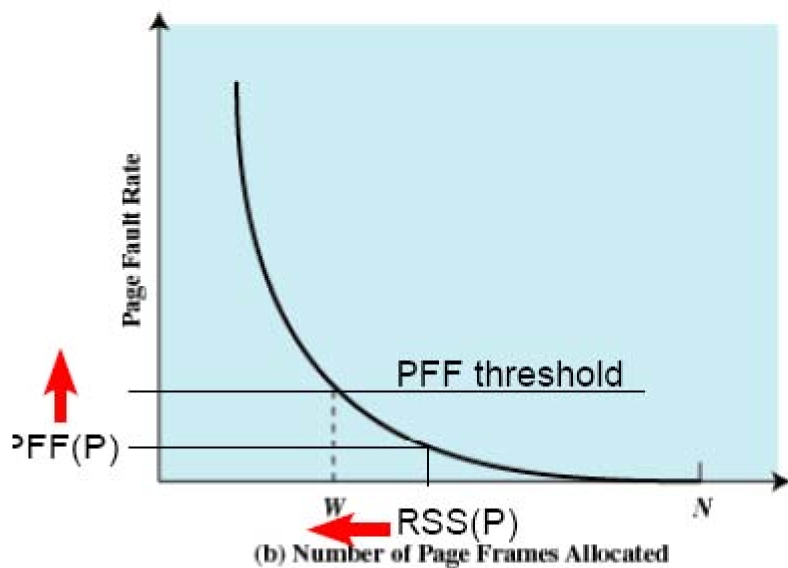
In pratica, a parole:

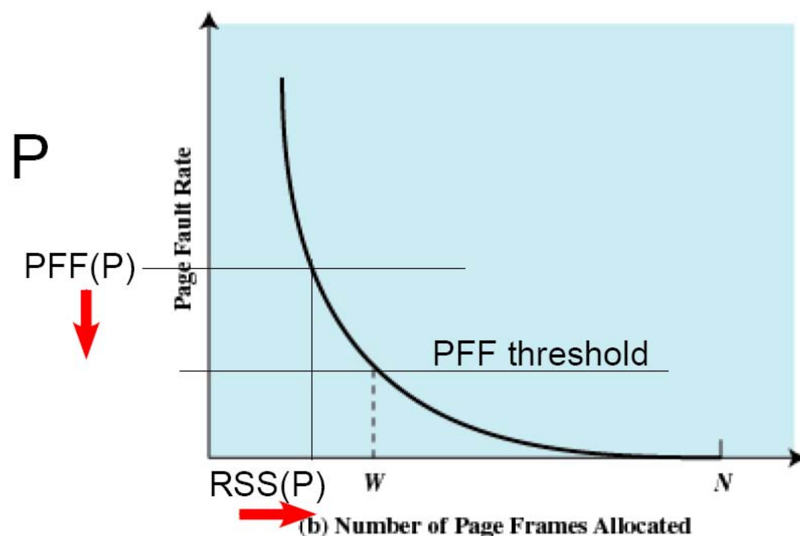
Questo algoritmo prevede la presenza di un use bit associato ad ogni pagina in memoria, che viene posto ad 1 ogni volta che una pagina è referenziata.

Quando avviene un page fault, un contatore del tempo virtuale viene incrementato:

- Se il valore di tale contatore è inferiore ad una certa soglia F, allora una nuova pagina è aggiunta al resident set del processo.
- Altrimenti tutte le pagine con use bit pari a zero sono rilasciate.

Nota: Un raffinamento di tale algoritmo è possibile con l'uso di due soglie.



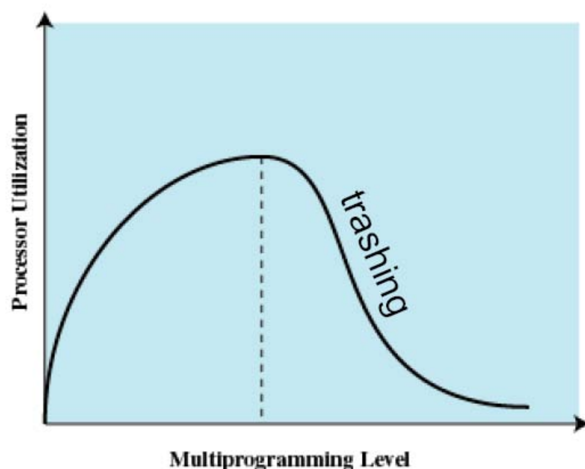


5.POLITICHE DI CLEANING:

Riguardano il come determinare quando una pagina dovrebbe essere scritta in memoria secondaria (su hd e quindi tolta dalla ram). Le tecniche principali sono 3: la demanding cleaning(pagina scritta su disco solo quando deve essere restituita); il precleaning (pagine scritte a gruppi prima della loro selezione per la sostituzione). Tali tecniche presentano gli stessi vantaggi/svantaggi delle rispettive politiche di fetching; L'approccio migliore usa il page buffering: cleaning soloper pagine sostituibili. Le pagine vengono divise in 2 gruppi, modificate e non modificate. Quelle modificate vengono scritte in batch (insieme), quelle non modificate vengono richiamate se accedute,perse se sovrascritte.

6.CONTROLLO DEL CARICO:

Il controllo del carico riguarda il numero di processi residenti in memoria centrale -> livello di multiprogrammazione: >>#processi in ram --><<dim RS --> >>#page fault -->thrashing. Tale gestione riguarda la gestione della memoria e va effettuata in maniera accorta perché come, come visto, un numero eccessivo o scarso di processi influisce pesantemente sulle prestazioni. Con pochi processi aumenta la probabilità che siano in blocco, e si perde molto tempo a trasferirli su disco. Inoltre l uso del processore diventa inefficiente. Con troppi processi si rischia il trashing. Se va ridotto il livello di multiprogrammazione, uno o più processi residenti devono essere trasferiti su disco-> SOSPENSIONE di un processo: se necessario per ridurre il carico, deve essere attuata. 6 possibili modalità di scelta del processo da sospendere: processo a priorità più bassa; processo con più page fault; ultimo processo attivo; processo più grande; processo con il più piccolo resident set; processo con la finestra attiva più grande.



Scheduling a breve medio e lungo termine, algoritmi per cpu scheduling

Con il termine schedulazione, si intende il modo di gestire le code in cui i processi attendono in maniera da minimizzare l'attesa e ottimizzare le prestazioni nel sistema. Nel mondo dei sistemi operativi esistono 4 tipi di schedulazione:

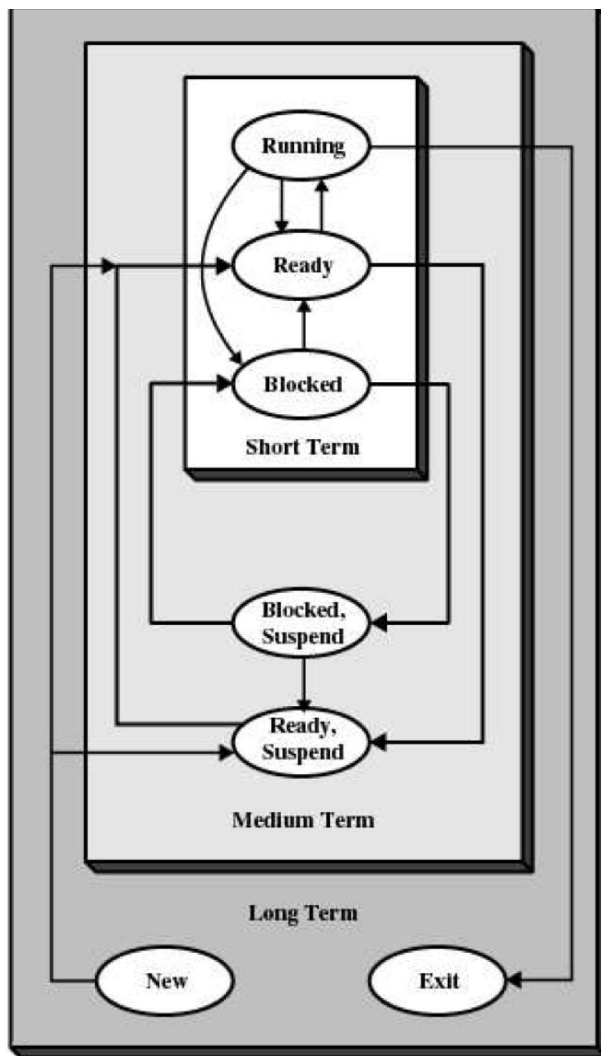


Figure 9.2 Levels of Scheduling

1. **A breve termine** (VEDI SOTTO): gestito solitamente da dal dispatcher, seleziona il processo da mandare in esecuzione. E' lo scheduler eseguito più spesso. Alcuni eventi che invocano tale scheduler sono i clock o gli I/O interrupts, le chiamate di sistema e i segnali;
2. **A medio termine**: riguarda il problema dello swapping (dei processi da Ram a disco-->processi bloccati), e le relative decisioni connesse allo spostamento di un file dalla/in memoria centrale alla/dalla memoria secondaria; è eseguito meno volte di 1. ma più di 3.
3. **A lungo termine**: determina quali tipi di processi devono essere ammessi nel sistema per essere eseguiti, cioè controlla il grado di multiprogrammazione. Una volta ammesso, un processo può essere trasferito nella coda della schedulazione a breve termine o in quella della schedulazione a medio termine (dipende dal metodo di creazione di un processo: se alla creazione va messo su disco, lo mette in quella a medio termine ad esempio). Lo scheduler responsabile di questo tipo di schedulazione deve affrontare due problematiche, una relativa alla scelta di quando ammettere un processo, l'altra relativa alla quantità di processi ammissibili. Il primo punto è correlato dal fatto che più processi sono creati/ ammessi e quindi presenti in memoria, minore sarà il tempo disponibile per ogni

processo. Il secondo punto si basa su diversi principi, come la priorità, il tempo di esecuzione previsto, le risorse richieste dal processo;
4. **Relativa all'I/O** (VEDI CAP 11).

I criteri per la schedulazione a breve termine sono divisibili in due dimensioni:
- orientati all'utente/orientati al sistema
- orientati alle prestazioni/non orientati alle prestazioni.

Utente	Sistema
Prestazioni	~~Prestazioni~~

1. **orientati all'utente (user – oriented)**: cercano di agire in modo tale da migliorare la percezioni che l'utente ha della reattività del sistema o del singolo processo. Questo è l'elemento difatti più importante per i sistemi interattivi e per gli interessi degli utenti che ne fanno uso;
2. **orientati al sistema (system – oriented)**: focalizzano sull'efficacia e l'efficienza dell'uso del processore, come ad esempio il throughput. Sono di minor importanza per l'utente e in genere per i sistemi ad utente singolo;
3. **orientati alle prestazioni**: sono solitamente di tipo quantitativo e possono essere misurati, come il tempo di risposta e il throughput;
4. **non orientati alle prestazioni**: non direttamente collegati con le prestazioni .

ESEMPI DI QUESTI CRITERI PER SCHEDULING A BREVE TERMINE: 1.1(orientati all'utente,relativi alle prestazioni): -t di risposta: per un processo interattivo, è il t che trascorre dall'invio di una richiesta fino a quando la risp non comincia ad essere ricevuta. E' una misura migliore del t di turnaround; lo scheduling deve minimzzare questo tempo e massimizzare il numero di utenti interattivi che hanno un t di risp accettabile; -t di turnaround: intervallo di t tra invio di un processo e il suo completamento.Comprende t di esecuzione e t speso in attesa di risorse (compreso il processore); è una misura per i job batch; -scadenze: specificato il termine di completamento di un processo, lo scheduling può subordinare altri obiettivi oltre a quello di massimizzare la percentuale di scadenze raggiunta. 1.2(orientati all'utente, non orientati alle prestazioni): -prevedibilità: posso eseguire un processo nellllo stesso tempo e allo stesso costo, indipend dal carico del sistema; variazioni di t di risp e ti di turnaround sono fastidiose per l'utenza. 2.1(orientati al sistema,relativi alle prestazioni): -throughput: scheduling massimizza il #di processi completati per unità di tempo (misura quantità di lavoro eseguita, che dipende dalla lunghezza media di un processo, influenzata cmq dalla strategiad di scheduling; -utilizzazione cpu: percent di t in cui la cpu è occupata(importante in sistemi costosi e condivisi, meno import per sistemi mono-utente e t reale). 2.2(orientati al sistema, non orientati alle prestazioni): -fairness (equità): i processi in generale devono essere trattati allo stesso modo, e nessun processo deve subire starvation. -priorità: scheduling favorisce processi a priorità più alta; -bilanciamento risorse: scheduling deve tenere impegnate le risorse, favorendo processi che le sottoutilizzano quelle più impegnate (coinvolge anche scheduling a medio e lungo termine).
PRIORITA': In molti sistemi, ad ogni processo è assegnata una priorità, e il processo con priorità più alta viene selezionato dal dispatcher. In genere, dati due numeri i e j tale che i<j, allora la priorità[i]>priorità[j], cioè ad un indice di priorità minore, corrisponde una priorità maggiore. È bene ricordare, come visto nella gestione dei processi, che l'uso di questo meccanismo porta ad una stagnazione per i processi a priorità bassa, che vengono selezionati solo quando la coda delle priorità "alta" è vuota: in questo caso è necessario usare una politica che aumenti la priorità dei processi in maniera opportuna.

ALGORITMI DI SCHEDULAZIONE (scheduling policies):

Le politiche di schedulazione sono tutte caratterizzate da una funzione di selezione, che stabilisce quale processo tra quelli presenti nella coda ready debba essere selezionato per essere eseguito. Tale funzione si basa su priorità, risorse consumate e/o utilizzate, caratteristiche di esecuzione del processo. Di particolare interesse in questo caso sono il tempo totale speso (attesa + esecuzione), il tempo di esecuzione e il tempo totale di servizio s (compreso e).
È inoltre importante la modalità di decisione, di tipo preemptive (interrompe processo in esecuzione per sostituirlo con un altro) o non-preemptive (lasciano terminare il processo in esecuzione). Le prime, in genere, necessitano di un lavoro più oneroso rispetto alle altre (overhead), ma ottengono migliori risultati, evitando la monopolizzazione del processore da parte di un particolare processo.
Infine è utilizzato il turnaround time (TAT), cioè il tempo totale speso nel sistema (in genere, tempo di attesa w + tempo di servizio s), o meglio ancora il normalized turnaround time, cioè il rapporto tra il turnaround time e il tempo di servizio ((w+s)/s), che rappresenta un valore indicativo del ritardo con il quale il processo è "servito" dal sistema.
SCHEDULAZIONE=GESTIONE CODE.

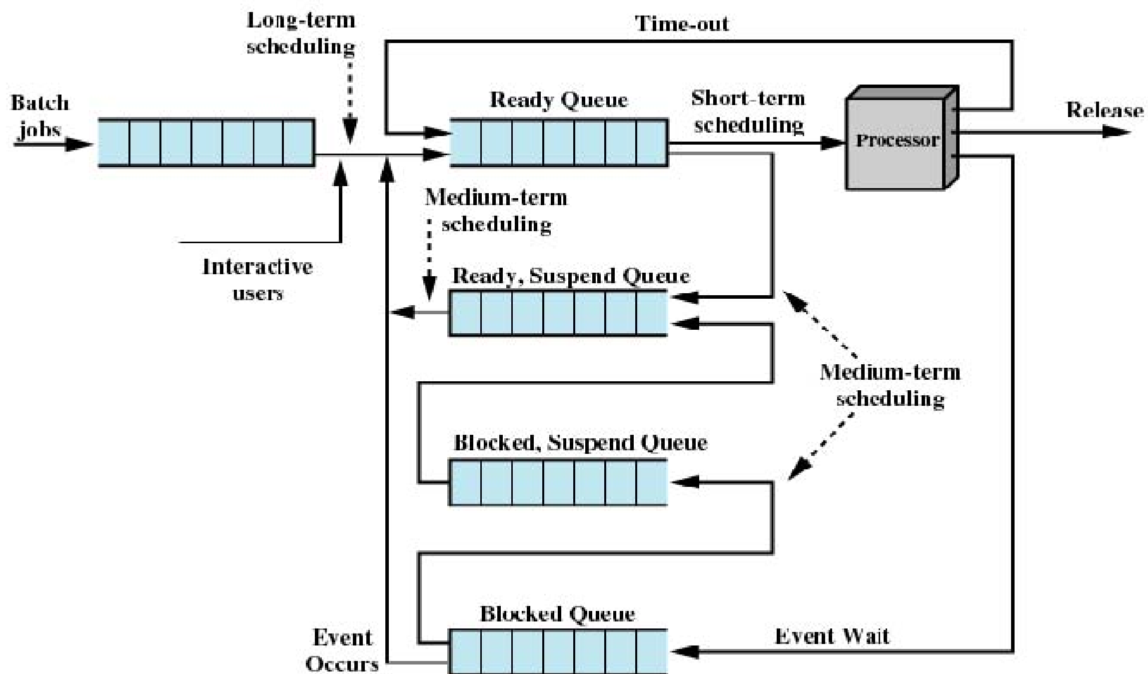


Figure 9.3 Queuing Diagram for Scheduling

ALGORITMI/POLITICHE DI SCHEDULING:

Riportiamo di seguito gli algoritmi/politiche di scheduling. Nello scheduling si vogliono favorire i processi i/o bound (es: processi interattivi tipo word) che usano poco la cpu.

- **PROCESSI I/O BOUND**: sprecano più tempo a usare dispositivi da I/O e poco la CPU (nello scheduling si vogliono favorire i processi i/o bound es: processi interattivi tipo word).
- **PROCESSI CPU BOUND**: processi che usano molto la cpu e pochissimo i disp di I/O.

- **BURST DI PROCESSO**: tempo speso in esecuzione tra due operazioni di I/O.

FCFS (First Came First Served)

- Ok solo processi lunghi
- no starvation
- max[w]
- non-preemptive,
- t risposta alto se varia t esecuzione processo

E' basato sulla tecnica FIFO, funziona meglio con processi lunghi, piuttosto che con quelli brevi, e ha il problema che favorisce i processi dipendenti dal processore (cpu bound) rispetto a quelli I/O bound: se un cpu-bound è in esecuzione, tutti gli i/o bound devono attendere o nelle code di I/O (blocked) o nelle Ready -->molti disp di I/O sono inutilizzati.
Quando cpu-bound esce dalla cpu (da running) quelli I/O diventano running ma si bloccano sugli eventi di I/O. Se viene bloccato il processo cpu bound, il processore rimane in ozio.
FUNZIONAMENTO: criterio di selezione=max[w]=max attesa in coda: primo arrivato primo servito.
Appena un processo diventa Ready lo metto in coda Ready; quando viene sospeso un processo, il più vecchio in coda Ready è selezionato per essere eseguito.

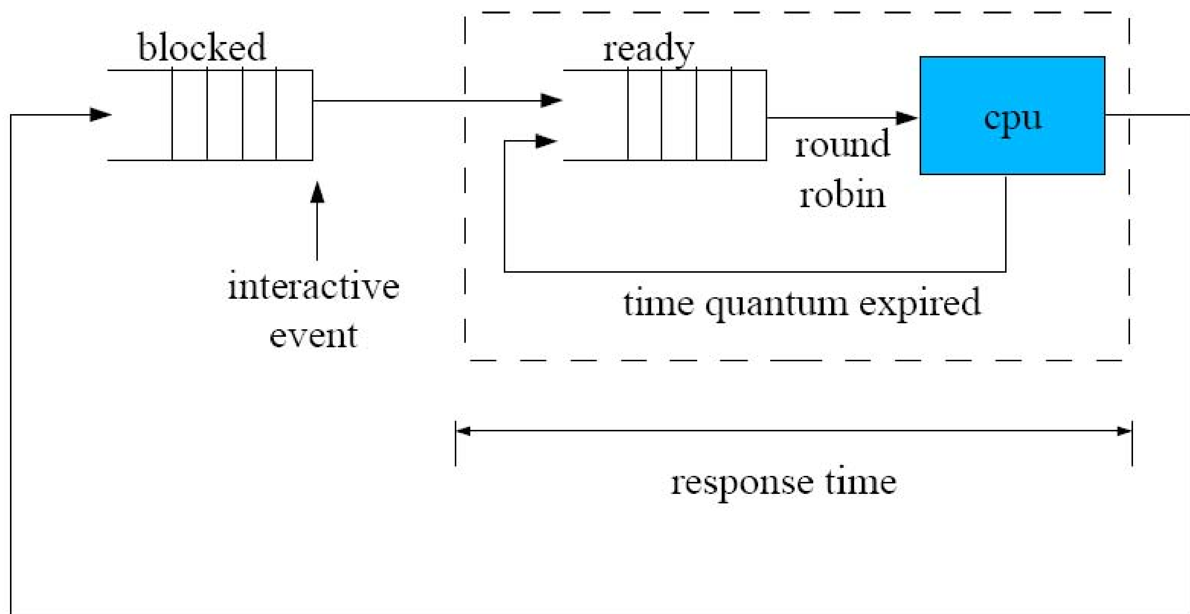
1. Determino il t in cui si sospende il processo.
2. Da esso calcolo t di turnaround=t che il processo trascorre in coda (wait) o il tempo totale (attesa w + tempo di servizio s della CPU).
3. Posso osservare il turnaround normalizzato che è il rapporto tra turnaround e t di servizio (cioè (w+s)/s): ha valore minimo 1. Più lungo è il t di esecuzione 'e' di un processo maggiore è il ritardo assoluto totale che può essere tollerato.

MIGLIORAMENTI: lo combino con le priorità, lo scheduler ha più code con diversa priorità (vedi feedback).

RR (Round Robin)

- Conosciuto anche come *time slicing*;
- è preemptive;
- overhead per interrupt,unfair.

Si vuole ridurre la penalità che i processi brevi subiscono con FCFS. Questa politica utilizza il prerilascio basato sul clock: degli slot di tempo fissati, detti spesso time slice, sfruttando dei clock interrupt.
FUNZIONAMENTO: nuovi processi arrivano e sono inseriti in coda Ready, gestita con FCFS. Allo scadere dello slot, viene sollevato un interrupt e il dispatcher seleziona il processo successivo nella coda, con una tecnica FCFS, se un processo viene bloccato per l'I/O viene messo nella coda I/O (quindi se ho tutti processi i/o bound nessuno attende in ready ma vanno tutti li).



COME SCEGLIERE IL TIME SLOT: deve essere tarato affinchè il cpu-burst (t in cui un processo usa la cpu tra 2 richieste di i/o) dei processi interattivi sia < del quanto di tempo, quindi scegliere un quanto di tempo > del t richiesto per un interazione con la cpu (se fosse minore i processi richiederebbero almeno 2 time slot). Quanto di tempo più lungo permette di avere t di risposta che è la metà (o meglio).
Il consiglio è quello di evitare slot troppo piccoli, ma di usare come dimensione quella appena più grande del tempo richiesto da una interazione o funzione di processo. Se ho solo processi I/O bound con cpu burst trascurabile, i processi non aspetterebbero manco un secondo in coda ready. Se invece hai n processi CPU bound, questi in coda ready aspetterebbero un tempo pari a (n-1)/n.
EFFETTI DEL QUANTO DI TEMPO (con molti utenti):

- Se ho un cpu burst < quanto di tempo , il t di risposta è =t per eseguire cpu-burst.
- Se ho cpu-burst>quanto di tempo-> poichè quanto scade in mezzo al cpu burst, il t di risposta si allunga in base ai processi che ho sulla macchina (se ho 10 processi, e il primo è interattivo ma ha un cpu-burst che è > quanto di tempo, quando s.o. toglie la cpu allo scadere del quanto, devo cmq attendere i quanti di tempo degli altri 9 processi).

QUANDO ROUND ROBIN=FCFS: quando scelgo quanto di tempo = al più lungo dei processi in running.
EFFICIENTE: nei sistemi time sharing, o sist orientati alle transazioni.
INEFFICIENZA: interrupt generano overhead.
INCONVENIENTE-> QUANDO È UNFAIR?: processi i/o bound sono svantaggiati anche qui (come in FIFO),poichè anche se non finiscono il loro quanto di tempo se vanno in blocco ritornano in coda ad attendere (perdendo il resto del quanto di tempo che non hanno utilizzato), mentre i processi cpu-bound si prendono sempre tutto il quanto di tempo (non li interrompe nessuno per tutto il loro cpu burst).
Round robin è unfair verso i processi che vanno in blocco prima dello scadere del loro quanto di tempo (quelli I/O bound: usano CPU per poco tempo (burst piccolo) e poi si bloccano in attesa dell'operazione di I/O; quelli CPU bound invece usano la cpu per tutto il loro quanto di tempo e non attendono nulla). Lo schema che affronta questo problema è quello del VRR.
FRAZIONE DI TEMPO DI CPU UTILIZZATA CON n PROCESSI I/O BURST:
$f(n,b,c)=1$ se $b \le (n-1)c$; $f(n,b,c)=nc/(b+c)$ se $b>(n-1)c$.

VRR (Virtual Round Robin):

Rende fair Round Robin. Funziona come round robin ma usa una coda ausiliaria FCFS con i processi rilasciati da un blocco I/O (cioè i processi che sono in attesa in code di i/o e che sono stati messi li interrompendo la loro esecuzione senza che gli scadesse il q di tempo, vengono messi in un'altra coda diversa da quella ready e schedulati con fcfs (il più vecchio,che attende di più, sarà il prossimo)). Quando si deve decidere il prossimo processo, questa coda ha priorità più alta rispetto a quelli in coda Ready; si preleva un processo da tale coda ausiliaria e viene eseguito per un quanto di tempo di base meno il t totale trascorso in esecuzione dall'ultima volta che è stato selezionato dalla coda Ready (in pratica lo si esegue per il quanto di tempo che gli è rimasto dalla precedente esecuzione). Infatti chi non finisce il suo quanto di tempo "guadagna un credito"-> se un processo i/o bound va in blocco(non completando quindi il suo quanto di tempo), la parte di quanto che rimane viene messa da parte, e il processo va in una coda con più priorità, che viene schedulata in modo tale da dargli il quanto di tempo residuo prima di essere messo di nuovo in circolo come processo normale.

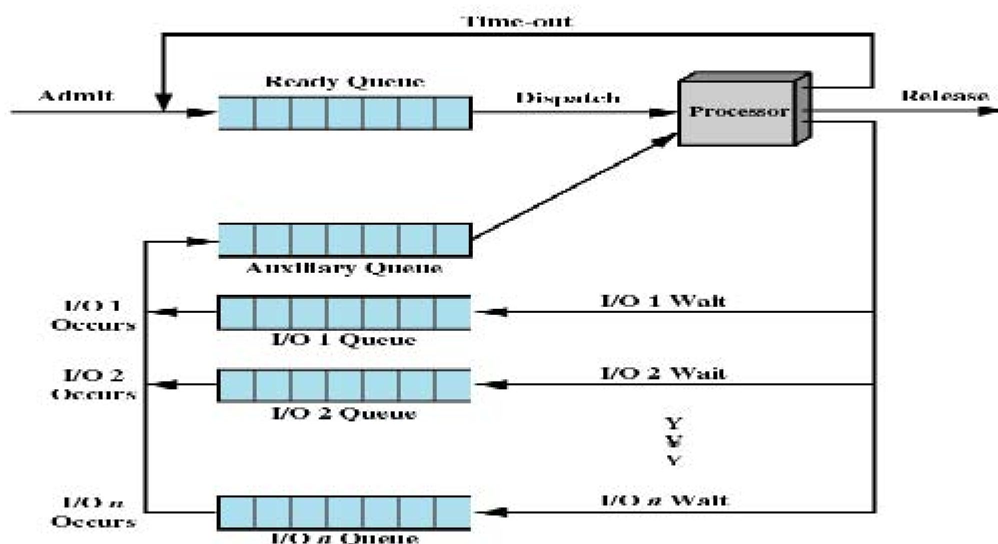


Figure 9.7 Queuing Diagram for Virtual Round-Robin Scheduler

SPN (Shortest Process Next)

- SPN, SJF;
- non-preemptive;
- stima;
- ok processi corti;
- starvation proc lunghi.

Il processo selezionato è quello con il minor tempo di esecuzione previsto, cioè il prossimo processo sarà quello più breve tra tutti (cerca di limitare il danno di FCFS che preferisce processi lunghi).
In pratica però è come predire il futuro! I processi con t esecuzione più breve scavalcano quelli più lunghi, ma la difficoltà sta nello stabilire quanto sia tale tempo, che può essere fatto ad esempio in maniera empirica o con statistiche. Ad esempio con la media esponenziale si ha:

$$S_{n+1} = \alpha T_n + (1-\alpha) S_n \rightarrow \alpha \in (0,1]$$

- a è un peso costante che determina il peso relativo dato dalle osservazioni più e meno recenti;
- T_n è il t di esecuzione del procesore per la n-esima istanza di questo processo (t tot di esecuzione per processi batch, o t di burst per processi interattivi);
- S_n è il valore previsto per la n-esima istanza.

Più vecchia è l'osservazione meno conta nella media, Più grande è "a" maggiore è il peso dato alle osservazioni più recenti (a=0.8 dò peso alle ultime 4 osservazioni, a=0.2 alle ultime 8).
PROBLEMI: i processi più lunghi rischiano la STARVATION: se c'è un arrivo stabile di processi più brevi. Infatti SPN toglie la supremazia ai processi più lunghi, ma non va bene in sistemi time sharing o orientati alle transazioni poichè non è preemptive.
SE STO ESEGUENDO UN PROCESSO E INTANTO MI ARRIVANO PROCESSI DA FARE + LUNGHI DI CIO CHE RIMANE DI QUELLO CHE STO ESEGUENDO, NON MI INTERESSA POICHè QUESTO APPROCCIO NON E' PREEMPTIVE

SRT (Shortest Remaining Time)

- Come SPN ma preemptive
- stima
- ok processi corti
- overhead
- starvation processi lunghi.

Lo scheduler seleziona sempre il processo con la minor stima del tempo atteso rimanente di esecuzione. Anche in questo caso è necessario possedere un metodo per effettuare la stima; non si aspetta che la cpu finisca di processare, infatti se mentre sto processando arriva un processo più corto (in termini di service time) quello in esecuzione viene stoppato e si inizia a processare quello più corto. Quando un nuovo processo va in coda Ready, può avere un quanto di tempo rimanente più breve di quello del processo in esecuzione al momento, quindi lo scheduler deve potere prerilasciare ogni volta che un nuovo processo diventa ready.
CONFRONTO CON I PRECEDENTI: SRT non avvantaggia i processi più lunghi come FCFS e a differenza di round robin non ha l'overhead dovuto agli interrupt, ma ha cmq overhead dovuto alla memorizzazione dei t di servizio trascorsi. Può essere meglio di SPN poichè dà una preferenza immediata ai processi più brevi a discapito di quelli più lunghi.
PROBLEMI: rischio di starvation per i processi più lunghi.

HRRN (Highest Response Ratio Next)

Pure lui stima. In generale, con una politica vogliamo minimizzare il normalized turnaround time. Per far ciò dobbiamo utilizzare delle stime, dato che non conosciamo a priori il tempo di servizio (necessario per il calcolo del turnaround time). Ora, indichiamo con R la seguente quantità: $R=(w+s)/s$ con R = tempo di risposta, w = tempo speso in attesa del processore, s = valore atteso del tempo di servizio.
Osserviamo che se il processo è selezionato immediatamente per l'esecuzione, R è uguale al normalized turnaround time. Pertanto, possiamo pensare ad una strategia che, nel caso in cui un processo sia nello stato bloccato, sceglie un altro processo con valore di R più grande (quindi questa politica seleziona di volta in volta usando il criterio max[R]: il più alto rapporto è il prossimo). Tale approccio è molto interessante in quanto sono favoriti sia i processi molto veloci (denominatore << numeratore) che quelli che attendono da molto nel sistema (anche in questo caso, den < num.).

FEEDBACK

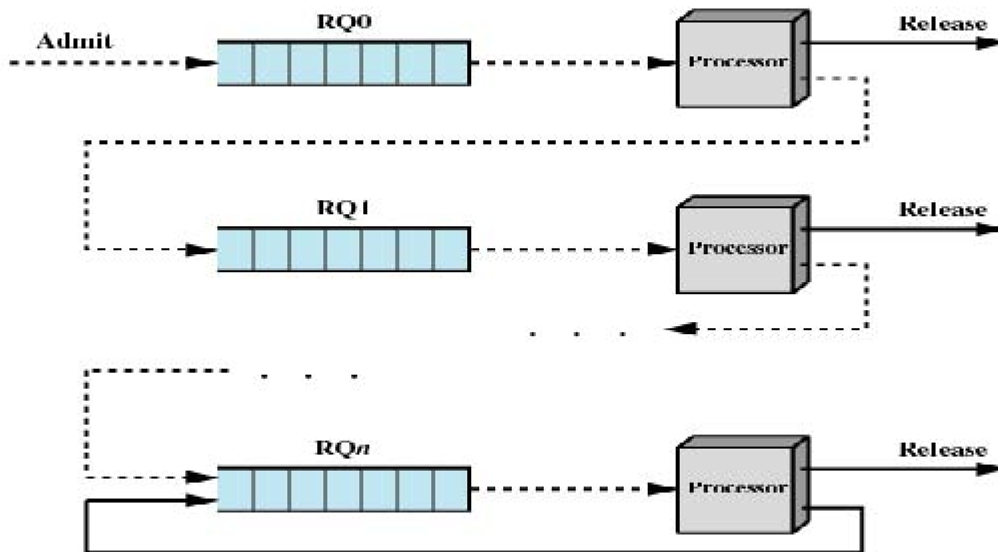


Figure 9.10 Feedback Scheduling

- preemptive
- usa FCFS in tutte le code e RR nell'ultima coda a priorità più bassa;
- priorità;

Da usare se non si hanno indicazioni sul tempo richiesto dai processi. Si usa allora uno scheduling di tipo preemptive, con in aggiunta la seguente tecnica: penalizzo i processi che sono stati più a lungo in esecuzione (quelli che usano di più la cpu: i cpu bound). Si usa il prerilascio e priorità dinamiche. Nel momento in cui un processo entra nel sistema, viene messo nella coda R0. In seguito, dopo essere stato interrotto da un altro processo, invece di tornare nella coda R0, va nella coda R1 (a priorità più bassa), e cosi via per tutti i processi, fino a raggiungere l'ultima coda Rn-esima del sistema (quella a priorità più bassa), dove rimane finché non termina. Un processo più breve completerà prima la sua esecuzione (t esecuzione minore) quindi non scenderà molto nella classifica delle code e sarà cmq avvantaggiato; i processi più lunghi invece tenderanno a finire in code sempre a priorità più bassa. Tute le code tranne l'ultima sono gestite con FCFS e l'ultima è gestita con round robin->chi arriva in quest'ultima coda non può andare più in basso, quindi lo si reinserisce in questa coda finchè non ha terminato la sua esecuzione (finchè non finisce il quanto di tempo).
PROCESSI FAVORITI: Sono avvantaggiati i processi veloci e brevi. feedback è in grado di dare più cpu ai processi I/O BOUND (che risalgono nella gradutaria delle priorità), poichè: più un processo usa la cpu più questo ha priorità svantaggiosa (cpu bound usano solo cpu, quindi prendono priorità più bassa quando escono dal running) ->i processi nuovi saranno avvantaggiati rispetto agli altri.
VARIANTI:

1. un processo scala di priorita'... sempre quando scade il suo quanto di tempo oppure quando scade il quanto e c'è almeno una altro processo nel sistema;
2. un processo aumenta di priorita' quando va in blocco... aumento fisso ogni volta che va in blocco oppure aumento dipende dal tempo speso in blocco.

FUNZIONI PER LE CODE DI FEEDBACK CHE NON VANNO BENE: Nessuna delle due funzioni è pienamente soddisfacente. Tra le due possiamo scegliere f2 poiché il suo comportamente è sempre coerente con gli obbiettivi dello scheduling anche se presenta ampi intervalli di variabilità.

$$f_1(c,b)=\left\lfloor n\cdot\frac{c}{b+1}\right\rfloor$$

$$f_2(c,b)=\left\lfloor \frac{n}{2}(c-b+1)\right\rfloor$$

SCHEDULING IN LINUX

Linux ha uno scheduling per i processi normali che è un feedback con preemption; i processi in kernel-mode possono anch'essi essere interrotti. Kernel 2.6 è un kernel preemptable: fa una stima del cpu-burst; le priorità più alte ce l hanno i processi con cpu-burst più piccoli; il comando nice dice quanto devo essere gentile e carino rispetto gli altri processi(nice 20 comando, quel comando girerà con priorità data dall'utente). Processi real-time: si ha scheduling FCFS o RoundRobin con quanto di tempo definito dall'utente; sempre con priorità e preemption ma tutto fissato a priori (non si va su e giu con le priorità come in feedback).

ESERCIZI: turnaround: t di attesa + t di esecuzione (t attesa=t effettivo inizio-t arrivo).
T di risposta: turnaround/t esecuzione (processing time).
Segue un esempio riassuntivo. Nota bene: i processi vanno considerati presenti in coda (e quindi a disposizione dello scheduling) solo nell'istante successivo al momento in cui arrivano, non tutti insieme da subito. Quindi all'istante 5 saranno presenti in coda (sempre che non siano già stati eseguiti completamente) solo i processi arrivati precedentemente, cioè nel caso specifico A,B e C.

---CAPITOLO 11:PARTE NON FATTA---

GESTIONE DELL' I/O E SCHEDULAZIONE DEL DISCO=I sistemi esterni che impegnano l'I/O di un computer possono essere divisi in 3 macro categorie: 1. interagenti con le persone (human readable), adatti cioè alla relazione con l'utente, quali display, tastiere e mouse; 2. interagenti con altre macchine (machine readable), cioè adatti alla relazione con dispositivi elettronici, come i dischi; 3. comunicativi, come i modems. Ci sono, in ogni caso, grosse differenze all'interno di ciascuna classe, principalmente dovute a: • applicazioni con le quali interagiscono; • complessità nell'essere controllate; • tipologia del trasferimento dati (blocco, flusso); • rappresentazione dei dati; • gestione degli errori; • modalità e velocità trasferimento dati (data rate).
TECNICHE DI GESTIONE DELL'I/O:In generale, 3 tecniche sono utilizzate per la gestione dell'I/O:
1. programmata: il processore richiede un comando di I/O e attende (busy waiting) che sia eseguito; 2. guidata dagli interrupts (interrupts driven): il processore, dopo aver richiesto un comando I/O, esegue altre istruzioni, in "attesa" di un interrupt che lo avvisi che il suo comando di I/O è stato eseguito; 3. accesso diretto alla memoria (DMA – Direct Memory Access): su richiesta del processore, un apposito modulo si occupa degli scambi di dati tra il dispositivo di I/O e la memoria centrale;successivamente, quando tutte le operazioni di I/O sono eseguite, il modulo DMA solleva un interrupt d'avviso (VEDI SOTTO).
EVOLUZIONE FUNZIONI I/O: 1. Il processore controlla direttamente le periferiche;

2. il processore si avvale di un modulo di controllo separato;3. il modulo di controllo viene dotato dell'uso degli interrupts;4. il modulo di controllo è fornito della tecnica DMA suddetta;5. il modulo di controllo è realizzato come processore indipendente con un proprio set di istruzioni;6. il modulo di controllo è dotato di memoria locale, diventando un vero e proprio sistema indipendente (detto I/O processor).

ACCESSO DIRETTO ALLA MEMORIA (DMA)= Considerata una delle conquiste più importanti nella gestione dell'I/O insieme agli interrupts. Quando il processore effettua una richiesta di lettura o scrittura al modulo DMA, gli invia le seguenti informazioni: tipo operazione; dispositivo coinvolto e linee dati e di comunicazione da utilizzare; quindi, attraverso la linea dati, gli indirizzi di memoria centrale dove leggere o scrivere le info richieste; il numero di parole (words) da leggere o scrivere. A questo punto, il processore continua il suo lavoro e il modulo DMA effettua completamente quello assegnatogli, sollevando un interrupt quando termina. L'uso del DMA e il rapporto che questi ha con le periferiche di I/O, può essere implementato in vari modi,che elenchiamo in ordine di efficienza (minore -> maggiore):➢ singlebus: tutti i moduli, DMA compreso, condividono lo stesso bus;➢ singlebus con DMA parzialmente integrato nei moduli di I/O;➢ I/O bus: un unico modulo DMA che gestisce tramite un bus I/O tutti i moduli di I/O, in una struttura gerarchica a due livelli.

OBIETTIVI DELLA PROGETTAZIONE NELLA GESTIONE DELL'I/O: • efficienza: le operazioni di I/O hanno sempre rappresentato un collo di bottiglia per i sistemi, data la loro molto minore velocità in confronto ai processori e le memorie centrali; • generalità, cioè il progettare i dispositivi e i moduli i più uniformi e standard possibili. Data la loro moltitudine e differenza, questi sono solitamente gestiti con un approccio gerarchico, in modo da nascondere le specifiche del singolo elemento lavorando ad alto livello con operazioni comuni e tipiche quali write/read, open/close, lock/unlock.
STRUTTURA LOGICA DELLE FUNZIONI DI I/O: Come spesso avviene in informatica, l'I/O è gestito e realizzato con un architettura a strati, che presenta, oltre ai vantaggi noti, la possibilità di modellare e controllare adeguatamente l'interazione tra sistemi (tipicamente: l'operatore e il dispositivo) che agiscono con velocità molto diverse tra loro. In genere abbiamo 3 livelli: 1. logico (logical I/O): la periferica viene vista come una risorsa logica, senza particolari riferimenti alla sue funzioni e strutture specifiche; 2. dispositivo (device I/O): le operazioni richieste sono convertite in istruzioni comprensibili al dispositivo in questione; 3. schedulazione e controllo: a questo livello vengono gestite le code del dispositivo e il controllo, come quello degli interrupts. Questo è lo strato che interagisce con l'hardware. In particolare, per la gestione delle periferiche di archiviazione con supporto al file system, sono presenti anche i seguenti 3 livelli (al posto di quello logico):1. gestore delle directory: i nome simbolici dei files sono convertiti in riferimenti estratti da apposite tabelle;2. file system: si occupa della struttura logica dei files e dei permessi;3. organizzazione fisica: converte gli indirizzi logici in indirizzi fisici.
BUFFERING:Le operazioni di I/O, in cui si spostano dati da/alla memoria centrale, richiedono che le porzioni di memoria interessate rimangano occupate fino al completamento delle operazioni. Ciò però risulta molto scomodo, sia perché pagine intere di un processo sono bloccate (e quindi non sostituibili), pur se utilizzate in piccola parte, sia perché c'è il rischio di deadlock (Il deadlock (letteralmente, "blocco mortale") è dovuto al fatto che se la memoria è piena, il sistema operativo potrebbe sospendere e bloccare il processo in attesa dell'operazione di I/O; l'operazione di I/O, parallelamente è bloccata perché aspetta che il processo sia swapped out dalla memoria... Per evitare ciò è necessario allocare la memoria per l'operazione di I/O prima che la richiesta di I/O sia sollevata). È pertanto necessario l'uso di un buffer. A seconda del tipo di periferiche, il buffer può essere blockoriented (es.: dischi), cioè gestisce le informazioni in blocchi di dimensioni fisse, oppure streamoriented (es.: stampante), privo di qualsiasi struttura.
GESTIONE DEI BUFFER: I buffer sono gestiti in vario modo: ➢ single buffer: quando un processo deve effettuare un'operazione di I/O, il sistema operativo gli assegna un buffer; per le periferiche blockoriented, i dati sono trasferiti prima nel buffer; quando il

trasferimento del primo blocco è finito, il processo muove il blocco nello spazio apposito e richiede immediatamente un altro blocco: tale tecnica è detta reading ahead o anticipated input. Ovviamente ciò complica l'architettura del sistema operativo. Per le periferiche streamoriented, si usano le tecniche lineattime (usata per terminali) o byteattime, cioè si cerca di strutturare in qualche maniera fittizia le informazioni in modo da lavorare poi in maniera simile alla blockoriented; ➢ double buffer: permette di incrementare le prestazioni del precedente, utilizzando un buffer per l'input e l'altro per l'output; ➢ circular buffer: nel caso in cui si usino più di due buffers, l'insieme dei buffers è definito buffer circolare, in cui ogni buffer costituisce una sua unità base.
---FINE CAPITOLO 11: PARTE NON FATTA--------------------------------

--CAPITOLO 11: PARTE DA SAPERE--------------------------------

SCHEDULAZIONE DEL DISCO

Lo scheduling riguardante la gestione del disco (inteso come la scelta di quali porzioni del disco leggere a seconda delle richieste del sistema) dipende in primo luogo dai parametri legati alle prestazioni del disco stesso nell'accesso ai dati in esso memorizzati, e cioè:

- seek time=t di ricerca= tempo necessario alla testina per posizionarsi sulla traccia= m*n+s con m=costante drive disco, n=#tracce, s=t avvio; varia tra 1 e 20ms (mediamente 8ms), si può controllare
- rotational delay=ritardo di rotazione=rotational latency: tempo necessario affinchè il settore giusto della traccia transiti sotto la testina; la somma di questi due tempi e detta tempo d'accesso; 7200rpm (giri al minuto) ho 8.3ms per una rotazione completa e 4ms di media; posso leggere insieme i settori contigui per ottimizzare (prendo richieste e faccio batching,considerandole come una sola richiesta);
- transfer time: tempo necessario al trasferimento dei dati,oltre che all'attesa per l'uso del dispositivo e del canale, nel caso questi risultino in altro modo occupati (dati vanno da disco a buffer interno, poi da buffer interno a buffer controller, poi da buffer controller a memoria). influenzato cmq dalla velocità del disco ed è un pò < del t di rotation; buffer disco->buffer controller->memoria, il tutto in un tempo < del t di accesso. ---> IL TEMPO CRITICO E' IL T DI SEEK: non si deve perdere tempo a posizionare la testina.

POLITICHE DI SCHEDULING DEL DISCO:

ALGORITMI DI DISK SCHEDULING:

- **INPUT**: richieste (numeri di traccia); locazione corrente testina; altri stati dell'algoritmo.
- **OUTPUT**: prossima richiesta da servire?
- **OBIETTIVI**: massimizzare il throughput (# richieste servite nell'unità di tempo); essere fair rispetto i vari richiedenti (i processi)->non ci piace se certe richieste vengono svantaggiate; si vuole evitare il problema della starvation (starvation: 2 utenti, uno va avanti e uno no (t di attesa))->starvation <> fairness: starvation posso creare sequenza infinita di richieste dove c è una che non verrà mai seguita (è in starvation); fairness ci sono certe categorie di richieste che vengono seguite in maniera non fair (non uniforme) rispetto alle altre (certi

processi sono avvantaggiati..). ***NEGLI ESERCIZI****: "supponi un alg di disk scheduling...si verificano casi di starvation?" NON FARE ESEMPI DI FAIRNESS.*

Si ha una coda per ogni dispositivo, con le richieste di read/write di quel dispositivo. Va deciso come schedulare queste richieste, poichè questo influenza come si muove la testina (ogni richiesta richiede dati che stanno in posti magari diversi, la testina deve spostarsi-->ordinando i punti richiesti faccio muovere la testina nel migliore dei modi-->risparmio sul t di seek-->quello che voglio!) .

1. **POLITICA RANDOM=SCHEDULAZIONE CASUALE**: Le politiche per lo scheduling del disco sono solitamente confrontate con la politica random, che consiste nel selezionare casualmente le richieste di accesso al disco dalla relativa coda. Tale politica può essere vista come una media politica pessima e pertanto utilizzata come metro di paragone per le seguenti.
2. **FIFO**: non fornisce particolari benefici;
    - è FAIR: tutte le req sono servite;
    - le req sono eseguite sequenzialmente secondo l'ordine di arrivo;
    - **PRO:** ok per pochi processi in coda e se le req richiedono settori vicini;
    - **CONTRO:** se ho tanti processi e req di tracce a caso-->diventa simile a random.
        - non raggruppa richieste di tracce vicine
        - priorità: ha il compito di ottimizzare gli obiettivi del sistema operativo, piuttosto che gli accessi al disco; valida per sistemi con lavori brevi, sconsigliata per l'utilizzo con basi di dati;
3. **LIFO**: esaudisce la richiesta dell'ultimo arrivato.
    - è UNFAIR: preferisce i processi nuovi appena arrivati.
    - dà STARVATION: le richieste più vecchie vengono soddisfatte solo se la coda viene svuotata completamente.
    - come random;
    - **PRO**: può essere buono (buon throughput) se il disco è libero.
4. **SSTF**=shortest service time first: (il tempo di servizio più corto per primo): sceglie sempre dalla coda la request che richiede il t di ricerca (seek) minore.
    - **PRO:** meglio di FIFO. Throughput ottimo; raggruppa di suo richieste di tracce contigue (più richieste le racchiude in una richiesta).
    - **CONTRO:**
        - dà STARVATION: quando ci sono req consecutive per tracce vicine (un processo ottiene il monopolio del disco se fa tutte richieste a tracce vicinissime: è sempre scelto poichè a t di seek bassissimo, e gli altri processi mai eseguiti).
        - è UNFAIR: se ne frega di req lontane dalla testina (forse quando passerà li vicino le fa visto che solo in quel caso gli costano poco (seek basso)).
5. **SCAN=**alg ASCENSORE (ELEVATOR)=LOOK: dato che tutte le politiche suddette (tranne FIFO) hanno il rischio di non soddisfare tutte le richieste più vecchie fino allo svuotamento della coda (tale situazione è indicata con il termine starvation: in ogni caso, anche questa politica soffre di starvation nel caso molto particolare in cui legge sempre dalla stessa traccia), è stato introdotto lo scan.
    - consiste nel soddisfare tutte le richieste presenti man mano che la testina si muove in una stessa direzione. Poi arrivato all'ultima traccia (o se non ha più richieste in quella direzione) torna indietro ripercorrendo la stessa strada e fa le richieste che incontra.
    - STARVATION SOLO NEL CASO in cui ho letture su stessa traccia (fa solo quelle e continua in quella direzione). PER ELIMINARLA potrei imporre il vincolo che si può fare una sola richiesta per ogni traccia.
    - NUOVE RICHIESTE: non sfrutta la località: è sfavorevole all'area attraversata per ultima: quando va in una direzione se arriva una nuova richiesta di una traccia "davanti" la testina (nella direzione in cui sta andando) la fa, altrimenti se la richiesta richiede tracce "dietro" la testina, queste saranno eseguite al ritorno (prima va in fondo poi quando torna indietro le fa-->t attesa elevato).

        - è UNFAIR: favorisce processi che richiedono tracce + esterne o interne (ci passa sopra 2 volte in breve tempo); problema risolto da C-scan. Inoltre favorisce i nuovi processi in coda (lo risolve N-step-scan). Le nuove richieste sono eseguite dopo molto tempo (arriva in fondo e quelle prima le fa solo al ritorno).
        - t di seek non ottimizzato ma buon throughput (meglio cmq SSTF). QUANDO è = a SSTF: quando braccio si muove verso #traccia minori fanno stesso cammino. apposite varianti:
        - **C-Scan(scan circolare)** : simile a SCAN ma limita lo scanning in una sola direzione. Esegue le req che incontra, poi arrivato in fondo ritorna all'inizio senza eseguire ulteriori richieste al ritorno, e poi riparte, riducendo il ritardo massimo possibile per le nuove richieste.
            - è FAIR: rispetto a scan è fair poichè non favorisce nessuno (torna indietro subito e riparte da li, le nuove richieste non aspettano tantissimo come in scan).
            - DA' STARVATION: come in scan, quando cioè si hanno letture su stessa traccia.
            - ha un buon throughput;
            - meglio scan.
            - usato in linux.
            - SE VOGLIO OTTIMIZZARE LA LETTURA DI FILE MOLTO LUNGHI COME SISTEMA I BLOCCHI (CHE CONTENGONO più DI UNA TRACCIA)?? Il modo migliore di sistemare i blocchi è in settori consecutivi all'interno di una stessa traccia (questo è vero per qualsiasi algoritmo di disk-scheduling) e in tracce consecutive nel senso di movimento della testina durante la lettura.
        - **NStepScan**: (per evitare la starvation) divide la coda in più sottocode, lavorando su una alla volta come se fosse una coda singola: per N=1, si comporta come una politica fifo; per N molto grande, si comporta come una scan tradizionale;
        - **FScan**: lavora con due code. Quando lo scan inizia, tutte le richieste si trovano in una coda, e le ulteriori richieste che arrivano vengono inserite nell'altra. La seconda coda viene servita allo svuotarsi della prima, quindi la seconda è come se diventasse la prima e viceversa...
- **RAGGRUPPARE LE RICHIESTE**: se in coda ho richieste a tracce vicine tra loro, lo scheduler può esaudire tutte queste richieste come fosse una richiesta sola; SSTF comprende questa tecnica di suo; FIFO invece non lo fa ma gli può essere aggiunta.
- **PROBLEMA "WRITE-STARVING-READ":** tutti gli algoritmi visti non risolvono tale problema in cui:
    - **SCRITTURE:** un processo può mandare tutte le scritture che vuole disinteressandosene, poichè queste non sono bloccanti e vengono accodate al dispositivo di i/o finchè c'è un buffer per farlo-->le write arrivano sempre tutte insieme.
    - **LETTURE:** sono invece di solito bloccanti, cioè chiedo lettura e devo attenderne i risultati, poi richiedo lettura e riattendo i risultati...per continuare il lavoro un processo deve quindi attendere-->chi legge va più lento di chi scrive se ho 2 processi in cui 1 scrive e l'altro legge.
    - è un problema più di UNFAIRNESS che di STARVATION: scritture sono favorite.

LINUX: dentro i suoi sorgenti ci sono 4 disk scheduler di cui 1 solo è attivo su un certo dispositivo (Su un disco posso averne uno e su un altro un altro). Per poterli usare, in fase di compilazione del kernel devono essere selezionati.

1. **NOOP**: come FIFO ma fa request merging; non si usa praticamente mai.
2. **DEADLINE:** scheduler a scadenza; variante del one-way elevator, con l'unica differenza che le richieste sono marcate con il tempo di arrivo (c'è una struct dati per gestire tale funzionalità: 3 code). Inoltre ogni richiesta ha una scadenza: mezzo secondo per le letture e 5 secondi per le scritture->"una cosa in coda non dovrebbe aspettare più di tali timeout"; se aspetta di più di tali timeout allora ha

precendenza. Elimina problemi di starvation per problemi tipo write-starving-reads.

- **PROBLEMA**: l'unico suo problema è che, ad un certo stiamo facendo scrittura tra le tante, poi ho lettura da altra parte del disco: vado a leggere quando scade, risolvo lettura e torno indietro, e questo può accadere più volte->faccio avanti e indietro con la testina rendendo inefficiente tutto il processo.

3. **ANTICIPATORY:** scheduler di default di qualche tempo fa, che "guarda avanti" anche se ovviamente non può prevedere il futuro; si ferma un attimo aspettando che tutto vada avanti, e vedendo ciò che succede, ma di quanto? 6ms. Cioè: prima di fare la successiva operazione dopo una lettura (ad es. una scrittura) aspetto poichè potrebbe arrivare di li a poco una lettura; applicazione prende i dati, se li copia da una parte e manda un altra lettura: se arriva entro i primi 6ms la servo poichè molto probabilmente starà li vicino.
    - deadline approach: one-way elevetor+scadenze;
    - prima di eseguire qualcosa di diverso da una read dopo una read attende 6ms->non li perdi anche se non fai nulla, poichè evita write-starving-reads e non faccio avanti indietro come deadline. L'attesa poi non è sempre fatta, ma dipende dal processo: processo che manda read in sequenza vengono riconosciuti e si attendono i 6ms, mentre se non sono processi di questo tipo non si attendo niente. Efficiente per file molto grandi ma non efficiente per dbms (letture di pezzi qua e là)->ok per streaming ma non per dbms.
4. **COMPLETE FAIR QUEUEING** (CFQ): è quello di default oggi. Ha preso le cose migliori da tutti gli altri: request merging+one way elevator+anticipatory+permette le priorità (approccio round robin sui processi: tende a schedulare ogni processo per un pò di millisecondi); un processo può ovviamente non impegnare tutto il suo tempo che l'alg gli ha dedicato, ma prima di passare ad un altro processo si attende un pò; -supporta priorità di I/O: si può cambiare priorità di I/O del processo;
    - **PRO**: ottimo per sistemi multiutente con i processi gestiti in modo fair (se lancio piccola lettura e grossa scrittura, la lettura finisce prima poichè ha i suoi quanti di tempo e nessuno glieli tocca).

DA SHELL:

- per cambiare algoritmo di scheduling: si può selezionarlo a runtime (senza fare reboot) e per dispositivo (uno scheduling diverso per ogni disco);
- file system virtuali: /proc e /sys. Lì dentro ci sono file virtuali il cui contenuto è creato al volo dal kernel quando leggi il file. Dentro ci trovi tutti gli alg di scheduling disponibili nel sistema con tra [ ] quello selezionato; con echo e poi cat lo selezioni.
- comando time: dà il t di esecuzione di un processo: time cat .... dice il tempo che ci impiega lo scheduler. –per modificare lo scheduler usato vai su .../hda/queue/scheduler e lo modifichi scrivendoci dentro il nome_scheduler che vuoi usare (echo nome_scheduler > ../hda/...).

RAID (Redundant Array of Independent Disks)

Idea: avere un insieme di dischi fisici visti dal sistema operativo come un singolo disco (dispositivo logico (fisicamente sono distinti)), le prestazioni dei dischi risultano costituire un vero collo di bottiglia per i sistemi. Sono pertanto state ideate diverse tecniche per l'incremento complessivo delle prestazioni, basate sull'uso in parallelo di più componenti.
pro:

- migliora tolleranza ai guasti (quando uno o più hd si rompono, e non guasti intesi come errori) e prestazioni;
- non perdi dati;

- disponibilità del servizio: se disco fisico s è rotto, quello logico (virtuale) continua a funzionare;
- no ridondanze;
- hot-swapping: sfilo un disco e ne metto uno uguale senza spegnere la macchina->farà REBUILDING per ripristinare lo stato normale;
- hot spare: dischi pronti per la sostituzione già montati nella macchina->fa automaticamente rebuilding quando sostituisce il disco da solo.

TECNICHE UTILIZZATE:

- **mirroring:** su 2 dischi scrivo stessa cosa->RIDONDANZA->su un disco copio tutti i dati che stanno su un altro disco->ho 2 dischi gemelli come caratteristiche e dati che contengono. Si può fare anche su più dischi, ma è costoso (non aumenta la capacità anche se aumenta tolleranza ai guasti), ma tipicamente si fa con 2 dischi.
- **duplexing**: variante di mirroring, che duplica dischi e controller.
- **parity**: bit ridondanti per ricreare ciò che si è perso;
- **humming error correction:** se c è un errore su disco fisico ce l'ho su tutto il disco non su un settore->errori sul disco sono rarissimi (o si rompe o funziona bene); i codici a correzione di errore stanno proprio dentro: quando scrivi un byte ne vengono scritti di più.
- **striping**: no ridondanze, ma migliora solo le prestazioni. Scrivere dati consecutivi sul disco logico su dischi fisici separati.
    - **vantaggio:** se faccio striping a livello di blocco; leggo in parallelo (vantaggio prestazionale); si può fare a vari livello (bit, byte, blocco (settore o gruppi di settori)).

TIPI DI RAID:

Si possono nidificare i raid uno dentro l'altro (nested=raid annidati: raid su dispositivi raid(es:raid10)). Uso un raid piùttosto che un altro in base al tipo di operazione che devo fare.

TIPI DI RICHIESTE: POSSIBILITA' IN SCRITTURA/LETTURA:

- I/O sequenziale: richieste di file molto lunghi(streaming) e blocchi grandi;
- I/O random: grossa frequenza di richieste per piccola quantità di dati (sistemi OLTP: database con sopra un application server).
- read/write.

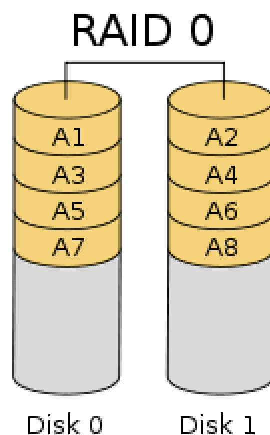
RAID più USATI:

- **RAID0 (non ridondante)**: solo striping: dati divisi in strip consecutivi sugli N dischi. I dati di un disco sono divisi in strips, che possono essere viste come partizioni in cui il disco stesso è diviso. I dati sono scritti sequenzialmente sulle strips, pertanto la rottura di un disco comporta la perdita di strips appartenenti a strisce diverse e la perdita di tutto l'array. La stripe 0 va su disco 1, ..,la stripe n va su disco n. Se si rompe un disco perdo le stripe che memorizza,quindi i pezzi di file in esse contenuti->BOOM!!!Se scrivo un file lungo 4 stripe me lo ritrovo su 4 dischi; le letture andranno in parallelo->t di lettura è quello di una sola stripe; velocità vale per grossi file e per richieste di tipo random purchè abbastanza random per cui tutti i dischi sono abbastanza coinvolti.
    - **PRO**: i/o sempre buono: speedup*n. Lettura e scrittura bulk (sequenziali) sono estremamente più veloci perché si riesce a leggere o scrivere N strip contigui contemporaneamente. Lettura e scrittura

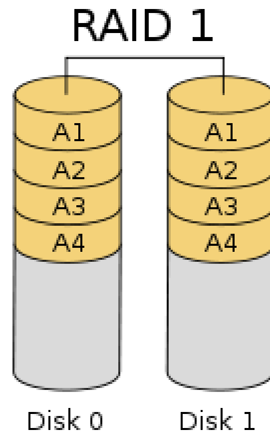
casuale sono estremamente più veloci poiché se gli accessi sono distribuiti uniformemente sugli N dischi riesco a servire N richieste contemporaneamente.

  - **CONTRO:** è molto peggiore poiché basta che uno dei dischi si danneggi perché tutto il sistema sia non funzionante. se si rompe un disco perdo tutto; se quel disco avrà pezzi di tutti i file sarà difficile ricreare anche un solo file.
  - **MEAN TIME BETWEEN FAILURES** *(MTBF: TEMPO MEDIO TRA 2 GUASTI SUCCESSIVI): MTBF=1/lambda; viene diviso per il numero di dischi->lambda è la frequenza dei guasti; lambda è l'inverso di questo tempo; se ho 4 dischi uguali e tutti hanno stesso t tra 2 guasti e li guardi tutti insieme, le frequenze si sommano. Inverso della somma degli inversi di ciò, per lambda tutte uguali viene fuori diviso n. Se un sistema A fallisce quando uno dei suoi disci D1,...,Dn fallisce, si ha che la freq totale dei guasti del sistema è: ; per lambda tutte uguali per ogni disco ho lambda(sistema)=Ndischi*lambda(undisco)--> MTBF(totale)=MTBF(undisco)/N con N=numero dischi. N=numero dischi in striping. Prestazioni: moltiplicate per N; x/N + epsylon.*

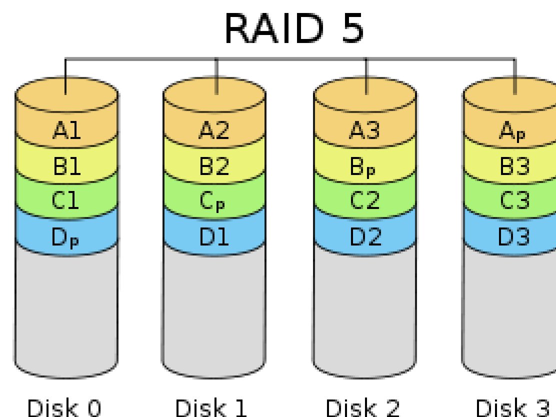
MTBF è uguale a 1/somma delle frequenze dei singoli guasti. Se la frequenze sono uguali per tutti i dischi si ha: 1/N*frequenza guasto con N = # dischi



- **RAID1:**solo mirroring (duplexing) cioè copio i dischi: tutti i dati sono scritti contemporaneamente in tutti i dischi.
  - **CONTRO:** ho 2 dischi ma dal punto di vista della spazio è come se ne ho uno solo; scrittura più lenta: "ho scritto quando entrambi i dischi hanno scritto"->t di scrittura: il peggiore dei due. Per file grossi dipende dal controller/software; inoltre il disco è sprecato, 50% efficienza rispetto alla capacità totale dei dischi. PRO: letture in parallelo sui 2 dischi->lettura il doppio più veloce; tolleranza ai guasti(se muore uno, l'altro contiene tutto); in stato degradato non ho più ridondaza; lettura si può fare in parallelo (prestazioni raddoppiate, in base cmq al controller o al sw che gestisce il raid) e scrittura va fatta su entrambi i dischi (no aumento prestazioni). USO: per i backup, ottimo per letture random e di piccole dimensioni.



- **RAID5:** il più usato; problema in raid3 e raid4: tutti scrivono la propria parità sul disco di parità->collo di bottiglia.
  - **SOLUZIONE A RAID3 E RAID4:** usa block level striping (divido in blocchi) con parità su più dischi. RAID 5 prevede che i dati siano suddivisi in blocchi sistemati consecutivamente su N+1 dischi. Inoltre un blocco di parità è inserito per ciascuna .riga. come in figura. Lettura bulk(sequenziale) e transazionale sono estremamente più veloci (bulk xN, transazionale x(N+1)). La scrittura risente del fatto che il blocco di parità va aggiornato e quindi questo è un collo di bottiglia. L'affidabilita è buona, si può sopportare il guasto di un disco senza perdere dati.Qui distribuisco la parità su tutti i dischi->quando prendo 2 stripe a caso da scrivere, la possibilità che le relative stripe di parità siano su stesso disco è molto maggiore che non in raid3 e raid4->scritture molto migliorate. Raid5 scrive ogni volta su due dischi. Se la distribuzione delle richieste è uniforme e le richieste sono tutte note in anticipo può scegliere sempre richieste con coppie di dischi non sovrapposte e quindi parallelizzabili.CONTRO: Si richiede sempre la conoscenza delle altre strip che stanno intorno; ridondanza. PRO: più prestazioni in lettura; collo di bottiglia distribuito sui vari dischi; tolleranza al fallimento di 1 disco (rebuild con xor).

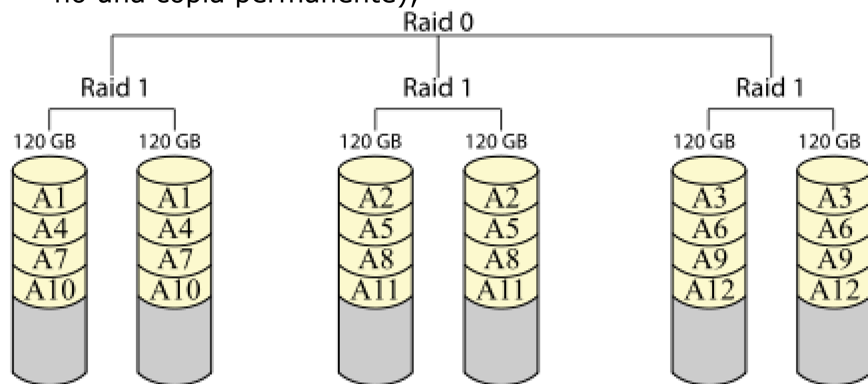


RAID NIDIFICATI (nested raid) =ARRAY NIDIFICATI:

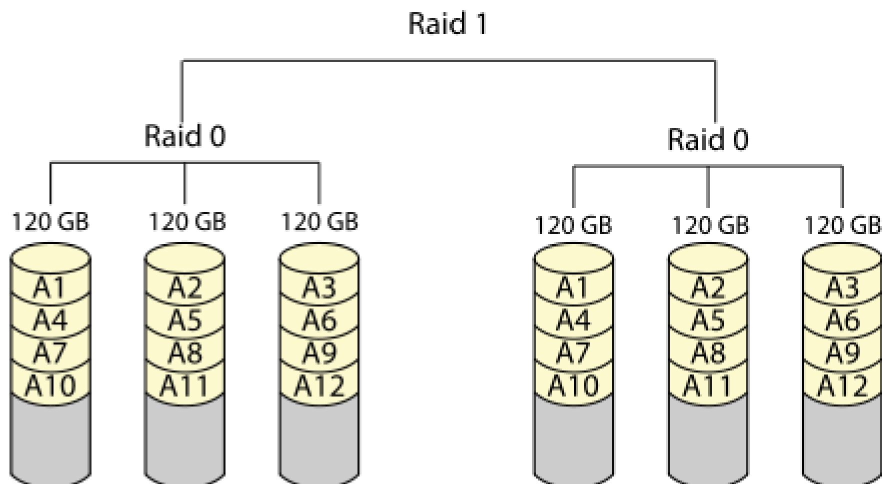
Utilizzare come mattone per costruire un array degli array. RAID XY significa che prima ho un certo numero di array di tipo RAID X e poi si applicano i RAID Y (si beccano i difetti di X).

- **RAID10**: prima raid 1 (tanti mirroring cioè backup) poi sui vari mirroring fai striping (raid0).

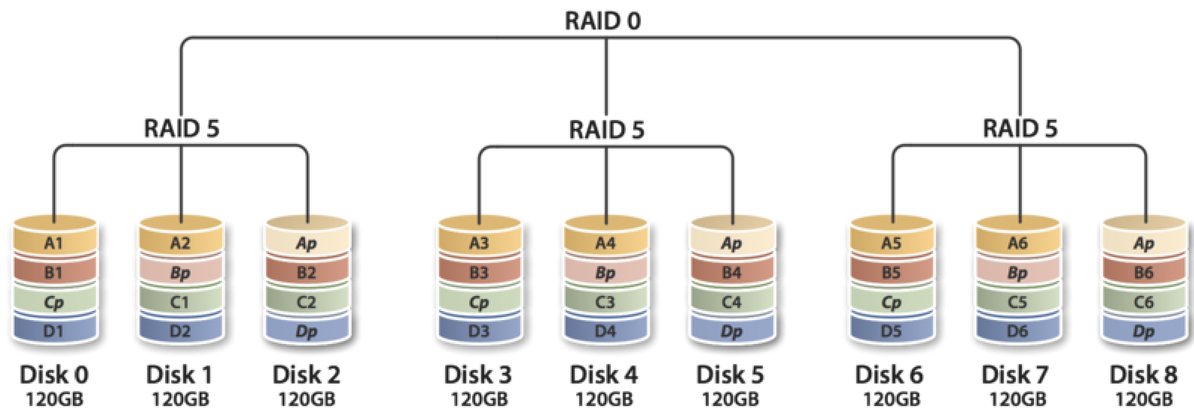
- **PRO**: performance molto migliorate rispetto i precedenti; parallelismo letture; ogni array di livello basso può sopportare una rottura: stripe sopravvive. Buone prestazioni in rebuilding: tutti gli array non coinvolti nel problema (tutti tranne 1) non si accorgono di nulla, e solo l'array che ha subito il danno fa rebuilding->stato degradato abbastanza buono. Permette "quasi" 2 fault.
- **CONTRO**: devi avere almeno 4-6 dischi. Se si rompe un disco resto ridondante sugli altri. In pratica copi disco su un altro col mirroring (raid1,ridondanza), poi dividi in strip su più dischi-->ottieni info divisa su più dischi (ok letture parallele) ma con backup di ogni disco (ok contro fail). Raid0 volumi logici 01,cioé: 2 dischi sono usati per fare striping, e gli altri 2 sono usati per fare mirroring di ognuno di quei 2 (con striping se se ne rompe uno sono fottuto, mentre con gli altri 2 in mirroring ne ho una copia permanente);



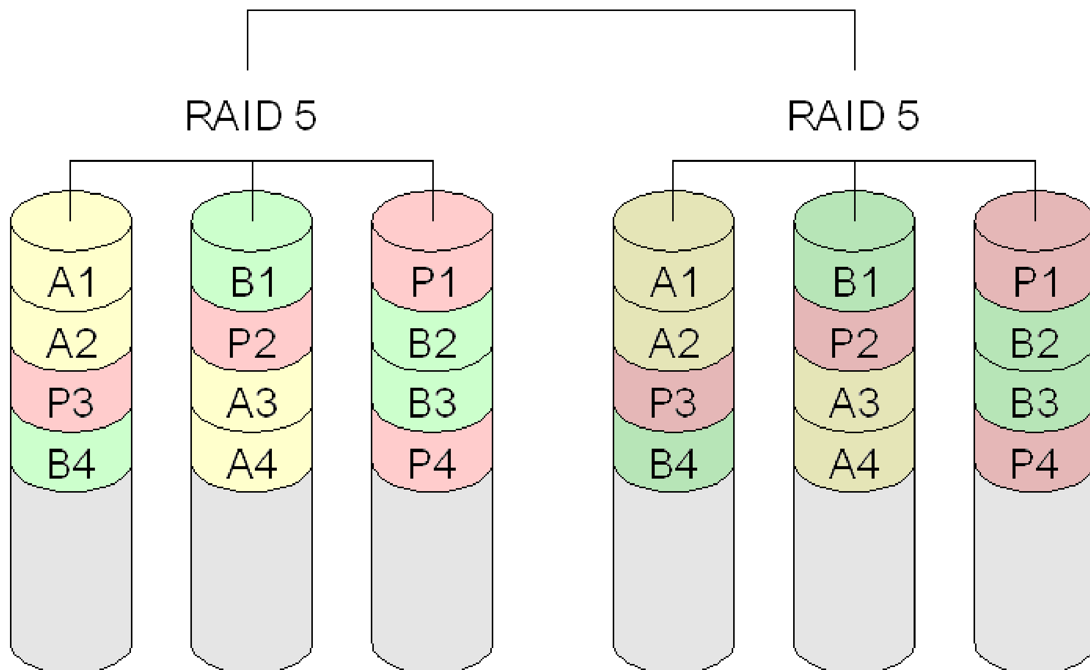
- **RAID01**: simile a raid10. Ma in uno stato degradato: dipende dal controller/sw che gestisce la cosa.
  - **CONTRO:** tolleranza ai guasti. Se si rompe un hard disk, tutto l'array è considerato non utilizzabile e non hai più ridondanza. Se si rompe un disco non ho più mezzo mirroring->no più ridondanza->permette un solo fault. RAID10 O RAID01? Costo e prestazioni simili, ma alcuni controller possono supportare lo 01 che è un pò peggio.



- **RAID 50:** Un insieme di array raid5 vengono messi in raid 0. Le parita' sono indipendenti su ciascun array raid 5.



- **RAID 51:** Un insieme di array raid5 vengono messi in raid 1.

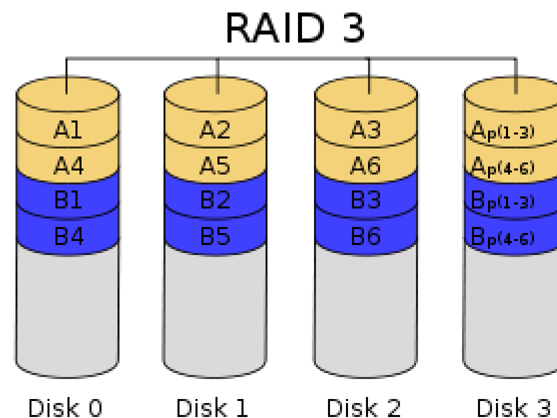


RAID MENO USATI:

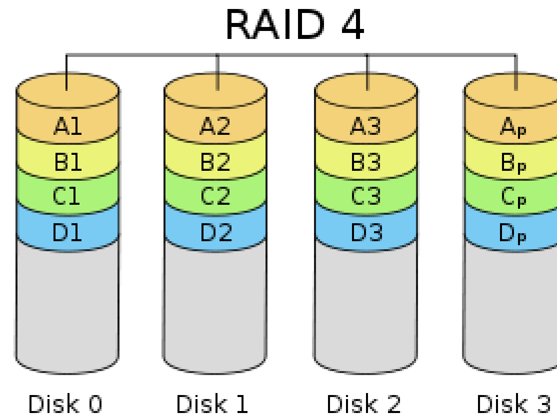
- **RAID2**: non più usato poichè usa codici ridondanti. Ha bit level striping (strip di bit e non di blocchi o byte): per 4 bit consecutivi metti un bit in ogni disco; con 3 bit di ridondanza negli altri tre dischi (codici di hamming: correggono errori di lettura).
  - **PRO:** utile per correggere errori su bit->cmq questo è ora incluso nei dischi stessi: inutile!
  - **CONTRO:** dischi devono essere sincronizzati tra loro (girare a stessa velocità e settori devono stare nella stessa pos nello stesso istante in tutti i dischi)->hw costoso: correggo errori tramite cpu e sequenze di

hamming. Mai usato poichè ora i dischi ce l hanno già nell'hw la correzione degli errori di lettura.

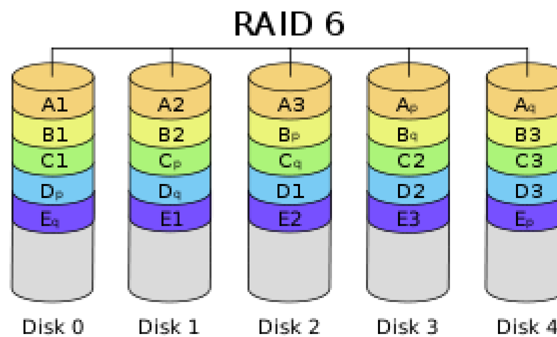
- **RAID3**: striping a livello di byte, e un disco con bit di parità. Le stripe sono tipicamente minori del settore (1024byte);
    - **PRO:** lettura molto buona (raid0 + la parità); REBUILDING: tramite xor.
    - **CONTRO:** scrittura scarsa, poichè la parità quando scrivo va scritta->parità non dipende solo dal settore che scrivo ma anche da ciò che c è intorno: stripe di parità si calcola facendo lo XOR->una scrittura di una stripe in uno dei dischi può comportare delle letture: la parità non sta sullo stesso disco delle stripe(vedi raid5 che invece lo fa), quindi devi scrivere la strip su un disco e poi la parità su un altro-->scrittura in più!!!. Se scrivo b1 e ho info per scrivere la parità, e sto scrivendo b2 e ho in cache le altre strip bi per la parità, ho come collo di bottiglia il disco il parità (una parità sola può scrivere);
        - **PROBLEMI CONSEGUENTI CIO'**: conoscenza di ciò che ci serve per scrivere la parità e la possibilità di scrivere la parità per una stripe alla volta.
        - **CALCOLO PARITA'**: può essere cpu intensive (cioè occupare la cpu). Non si tratta di settare registro su DMA ma è un calcolo vero->implementazione in hw (s.o. non vede niente meno di un settore, non può avere stripe cosi piccole). ES: ho 3 stripes (sequenze di bit) A, B,C e la stripe di parità P=AxorBxorC. Se voglio scrivere A in A', poichè A B C stanno tutti sulla stessa rega, devo ricalcolare la parità: posso farlo in 2 modi: la nuova parità è P'=A'xorBxorC (BeC vecchi necessari->letture da fare: O leggiamo B e C o uso la cache (non li leggo veramente cosi)), oppure P'=A'xor A xor P (con A e P che avevo prima se li avevo letti).



- **RAID4**: come raid3 ma si può fare in sw, lo troviamo di più di raid3. stessi vantaggi e svantaggi di raid3: devo conoscere altri settori in scrittura/collo di bottiglia su disco di parità; tolleranza ai guasti.
    - **SCRITTURE:** Raid4 deve scrivere ogni volta sul disco di parità, non può parallelizzare.



- **RAID6:** ancor più ridondante di raid5; supporta 2 guasti (più tutela); potrei avere un disco in più non utilizzato. Possibilità per usare tutti i dischi per la lettura. Scrittura di 2 parità. Costoso.



JBOD (just a bunch of disk):

spanning! un insieme di dischi, anche detto spanning, non è proprio un raid. é come se hai i dischi uno di seguito all altro: scrivo prima su uno, poi in ordine. Non ci sono aumenti di prestazioni a meno che il disco (l'array) non sia usato in modo omogeneo e leggi in parallelo. Se se ne rompe uno butto uno, ma si recupera prima: un utility permetterebbe di recuperare i file. Con raid0 è molto più difficile (non esistono i file di QUEL disco, poichè i file sono distribuiti).

- **VANTAGGIO:** con dischi di dim diversa non perdi spazio.
- **COMPARAZIONE:** scegli raid in base all'operazione che devi fare.

IMPLEMENTAZIONI DEI RAID:

Sono divise in sw, hw, ibride.

- **IMPL SW:** non hanno bisogno di altro che non del s.o. Possono essere inefficienti nel calcolo della parità (cpu impegnata nel calcolo della parità ogni volta che scrivi su disco, mentre vorresti farci altro: cmq la cpu potrebbe essere grossa e manco sarebbe il collo di bottiglia)->diventano sempre più convenienti. linux ha raid0,1,4,5 e posso fare nesting come voglio poichè Linux può fare raid su qualsiasi dispositivo a blocchi (su 2 pennette, sui 2 file trattati come disp a blocchi mediante il device loop, su una penna e un disco, array di array, array misti (striping di una cosa di cui solo un pezzo è mirrored e l'altro non è mirrored (flessibilità))..). Windows: striping (raid0), mirroring(raid1), raid5 e spanning (JBOD)

- **IMPL HW**: cioè nei controller, che hanno necessità di un driver speciale di configurazione per il raid. Danno cmq la poss di calcolare loro la parità avendo loro l'hw necessario.
- **IMPL IBRIDE**: bios sono controller per il raid, ma a basso costo. Necessità di drive speciali (sono estensioni del bios); poca efficienza nel calcolo della parità (richiedono supporto al sw).

CAPITOLO 12: PARTE NON FATTA

ORGANIZZAZIONE DEI FILE E LORO ACCESSO: L'organizzazione dei files dipendente fortemente dai seguenti criteri:• tempo di accesso;• facilità di aggiornamento;•costo della memorizzazione (e dei dispositivi, quindi);• semplicità di mantenimento;• affidabilità (reliability);
Come si può vedere, questi criteri risultano essere in contrasto tra loro, ed è pertanto necessario fare delle scelte a vantaggio di alcuni e a discapito di altri. Sono comunque state ideate i 5 tipi di organizzazione che seguono:1. la pila;
2. il file sequenziale;3. il file sequenziale indicizzato;4. il file indicizzato;5. il file ad accesso diretto o tramite tabella hash.
Per semplicità di trattazione, non entriamo nel merito delle singole organizzazioni.

RECORD BLOCKING (SUDDIVISIONE DEI RECORDS IN BLOCCHI):Come detto, il record rappresenta l'unità logica di base del file. Dato che le periferiche di massa sono organizzate in blocchi, ne consegue che la relazione tra i records e i blocchi risulta essere molto importante per le prestazioni del sistema.
Ci sono diverse considerazioni da fare:➢ dimensione dei blocchi (grandi vs piccole);➢ tipo di blocchi (fissi vs dinamici). In genere, blocchi grandi contengono più records è quindi sono vantaggiosi, ma non sempre, come nel caso in cui i records siano letti in maniera molto casuale. TECNICHE DI BLOCKING:
Una volta selezionata la dimensione, possono essere utilizzate tre tecniche di blocking:1. blocchi fissi: contengono più record possibili, ma lo spazio che avanza nel blocco, va sprecato (una sorta di frammentazione interna;2. blocchi a lunghezza variabile divisi: records di lunghezza variabile sono messi in sequenza ed eventualmente tagliati (distribuiti..) su due blocchi: tecnica buona ma difficilmente implementabile in maniera efficiente; 3. blocchi a lunghezza variabile nondivisi: i records hanno lunghezza variabile e sono messi in sequenza come nel caso precedente però, nel caso in cui l'ultimo record non entri nell'ultimo blocco, non viene "tagliato" e distribuito ma ne viene usato un altro e lo spazio rimasto libero nel penultimo e nell'ultimo va sprecato.

GESTIONE DELLA MEMORIA SECONDARIA=nella memoria secondaria, un file consiste in un insieme di blocchi. Il sistema operativo o il gestore del file system è responsabile per l'allocazione di tali blocchi. Ciò solleva due questioni riguardanti la gestione: 1. come allocare i files in memoria secondaria; 2. necessità di tenere traccia dello spazio libero. L'allocazione dei files comprende tre problematiche: a. quanto spazio assegnare al file quando viene creato. Per risolvere il problema è possibile utilizzare le tecniche della preallocazione o dell'allocazione dinamica, a seconda che la dimensione di un particolare file sia risettivamente conosciuta/calcolabile a priori o meno. In genere ciò non è possibile e si usa pertanto l'allocazione dinamica; b. quanti blocchi assegnare a ciascuna porzione del file: una porzione è una delle unità contigue in cui viene diviso lo spazio allocato per il file. In pratica l'uso di porzioni contigue, variabili e grandi nelle dimensione, formate cioè da più blocchi, presenta il vantaggio di migliorare le prestazioni, grazie a tabelle di allocazione ridotte, sebbene la difficoltà di riutilizzare lo spazio sia alta. Inversamente, porzioni piccoli e fissate nelle dimensioni diminuiscono la frammentazione esterna dello spazio ma richiedono tabelle più complesse; c. il tipo di struttura dati da utilizzare per tenere traccia delle porzioni assegnate ad un file. Per

affrontare questa problematiche sono state ideate tre tecniche: allocazione contigua, in cui al momento della creazione del file viene allocato un insieme fisso di blocchi; allocazione a catena, in cui ciascun blocco allocato contiene un puntatore al successivo ed pertanto sufficiente memorizzare nella tabella di allocazione solo il puntatore al primo blocco del file: di tanto in tanto inoltre viene effettuata della compattazione per recuperare spazio; allocazione indicizzata, caratterizzata dall'avere il primo blocco costituente una sorta di indice agli altri blocchi: ciò inserisce un ulteriore livello di indicizzazione, peggiorando le prestazioni, ma permette l'accesso diretto ad ogni blocco e l'uso sia della preallocazione che della allocazione dinamica. È in genere il più usato per la sua flessibilità e le sue prestazioni.

Unix File Management, inode, Linux VFS, ext2

FILES MANAGEMENT

In genere la memoria secondaria è associata ad un file system, cioè ad un modello logico che la descrive e la rende utilizzabile. Un file system permette agli utenti di creare collezioni di dati, dette files, che hanno il compito di:
- archiviare i dati in maniera persistente;
- essere condivisi tra i processi;
- avere eventualmente una struttura interna utile per un'applicazione. Le operazioni riguardanti i files, fornite da un file system: create; delete; open; close; read; write. Infine, un file system mantiene informazioni sui files gestiti, come permessi, data creazione, ultimo accesso eseguito...

Tradizionalmente, la trattazione dei files prevede l'introduzione di diverse strutture per la gestione dei dati:
- il field (campo), che rappresenta l'unità base;
- il record, formato da più campi;
- il file, costituito da un insieme di records;
- il database, che contiene dati strutturati in maniera relazionale.

È comunque possibile che un file sia solamente un flusso di bytes senza una particolare organizzazione interna.
Le operazioni eseguibili su di un file sono, tipicamente:
- retrieve all / one / next / previous (recupera tutti / uno / prossimo / precedente);
- insert all / one / next / previous (inserisci tutti / uno / prossimo / precedente);
- delete all / one / next / previous (cancella tutti / uno / prossimo / precedente);
- update all / one / next / previous (aggiorna tutti / uno / prossimo / precedente).

FILE MANAGEMENT SYSTEM:

Un file management system è quella porzione di software che implementa e gestisce un file system nella pratica, fornendo le suddette operazioni e caratteristiche. In generale, gli obiettivi di file management system sono:
- garantire la validità dei dati;
- ottimizzare le prestazioni;
- fornire un supporto I/O ad una gran varietà di dispositivi di memorizzazione diversi;
- permettere all'utente di effettuare le operazioni suddette;
- minimizzare o eliminare i rischi di potenziali perdite o danneggiamenti dei dati;

- fornire un'interfaccia comune ai processi;
- permettere una gestione multiutente.

Per raggiungere questi obiettivi è pertanto necessario che un file management system soddisfi i seguenti requisiti:

- possibilità per ogni utente autorizzato di creare, distruggere, leggere, scrivere, modificare files;
- gestione dei files con nomi simbolici facilmente utilizzabili dall'utente umano;
- possibilità di scambiare dati tra i files;
- possibilità di strutturare porzioni del file system secondo le esigenze specifiche di ciascun utente;
- possibilità di gestire permessi sulle operazioni effettuate.

GESTIONE DEI FILE SOTTO UNIX

Nei sistemi Unix sono presenti 6 tipologie di file:

1. File regolare o ordinario: contiene informazioni di qualsiasi tipo, strutturate in qualsiasi maniera, solitamente stabilita dall'applicazione che lo ha creato o dall'utente. Viene trattato in genere come un flusso (stream) di bit;
2. Directory: contiene una lista di nomi di files e relativi puntatori per accedervi, tramite INODE (vedi sotto). Le directory sono organizzate gerarchicamente, leggibili da qualsiasi utente ma modificabili solo tramite apposite funzioni o sottoprogrammi del sistema operativo (si pensi, in Linux, ai comandi rmdir, chmod...);
3. File speciale: non contiene dati ma permette di mappare un dispositivo fisico con un nome di file, in modo che successivamente sia possibile gestire tale dispositivo come un file;
4. Named pipes: hanno il compito di facilitare la comunicazione interprocesso. Un pipe file buffer riceve dati in input in modo che un processo che legge dall'output della pipe riceve i dati secondo una modalità fifo. Questa tipologia e la precedente sono caratteristiche quasi esclusive dei sistemi Unix-like.
5. File di collegamento forte (hard linking file): un collegamento è essenzialmente un nome associato ad un file esistente: un inode può avere più hard link; cancello inode sse non ha più hlink associati (inode è tutto tranne i nomi; hlink sono i nomi agli inode); puntano proprio al file fisico; non possono essere invalidi.
6. File di collegamento simbolico (soft link): questo è un file che contiene un percorso relativo o assoluto; può essere non valido; è associato ad un hard link e non punta al file fisico.

INODE

Chiariamo il concetto di inode introdotto per le directory. Un inode descrive vari attributi di un file (utente, gruppo, permessi, data/ora di modifica e accesso, ecc) e quali blocchi del disco contengono il file. Se le informazioni sui blocchi del file non entrano in un singolo inode se ne usano altri organizzati in una struttura ad albero. L'inode contiene anche il numero di riferimenti al file. Infatti un file può avere più nomi (hard link). Una directory in unix contiene coppie (nome, inode id).Un inode è una struttura di controllo per mezzo della quale il sistema operativo gestisce i file. Grazie ad un file di collegamento forte (HARD LINK), più nomi di file possono essere assegnati ad un singolo inode, ma un inode è associato esclusivamente ad un file fisico e viceversa.
L'allocazione dei files è gestita dinamicamente e in maniera indicizzata. L'inode contiene 39 bytes di informazioni di indirizzamento, divisi in 13 indirizzi da 3 byte.

STRUTTURA GERARCHICA

La struttura gerarchica che i file system dei sistemi UNIX utilizzano per tenere traccia dei blocchi di un file che ha la sua radice nell'Inode. L'inode è piccolo e può stare in ram, ed ha diversi puntatori ai blocchi del file a cui è associato (inode sta 1 a 1 con un file fisico; 1 a molti con un hlink):

- i primi 10 (09) indirizzi dell'inode puntano direttamente ai primi 10 blocchi del file su disco;
- l'undicesimo punta ad un blocco contenente le successiva porzione dell'indice: tale organizzazione è detta single indirect;
- il dodicesimo punta ad un blocco contenente una lista di ulteriori blocchi single indirect: tale struttura è detta double indirect;
- il tredicesimo punta ad un blocco contenente un terzo livello di indicizzazione, detto triple indirect.

Il blocco punta cioè ad una serie di blocchi indicizzati come al punto precedente, con la doble indirect. Pertanto, è possibile utilizzare file di dimensione massima pari a circa 16 gigabytes (10 blocchi diretti + 256 blocchi single indirect + 256 x 256 = 65 blocchi double indirect + 256 x 65k = 16 M blocchi , ogni blocco 1kbyte, totale 16 gigabyte).Con file brevi non ho blocchi (grigi nella figura) in più da mantenere per tener traccia della struttura e accedo direttamente al blocco. Per file lunghi i primi byte li leggo subito,poi leggo il blocco grigio, poi leggo i blocchi del file. Se il file è ancora più lungo leggo un blocco grigio, poi paso a altri blocchi grigi puntati da questo, e cosi via fino ad arrivari ai blocchi "veri"...COMPROMESSO tra efficienza di allocazione nella struttura e velocità di accesso, ma cmq non sono limitato a tenere file per forza piccoli, anzi.

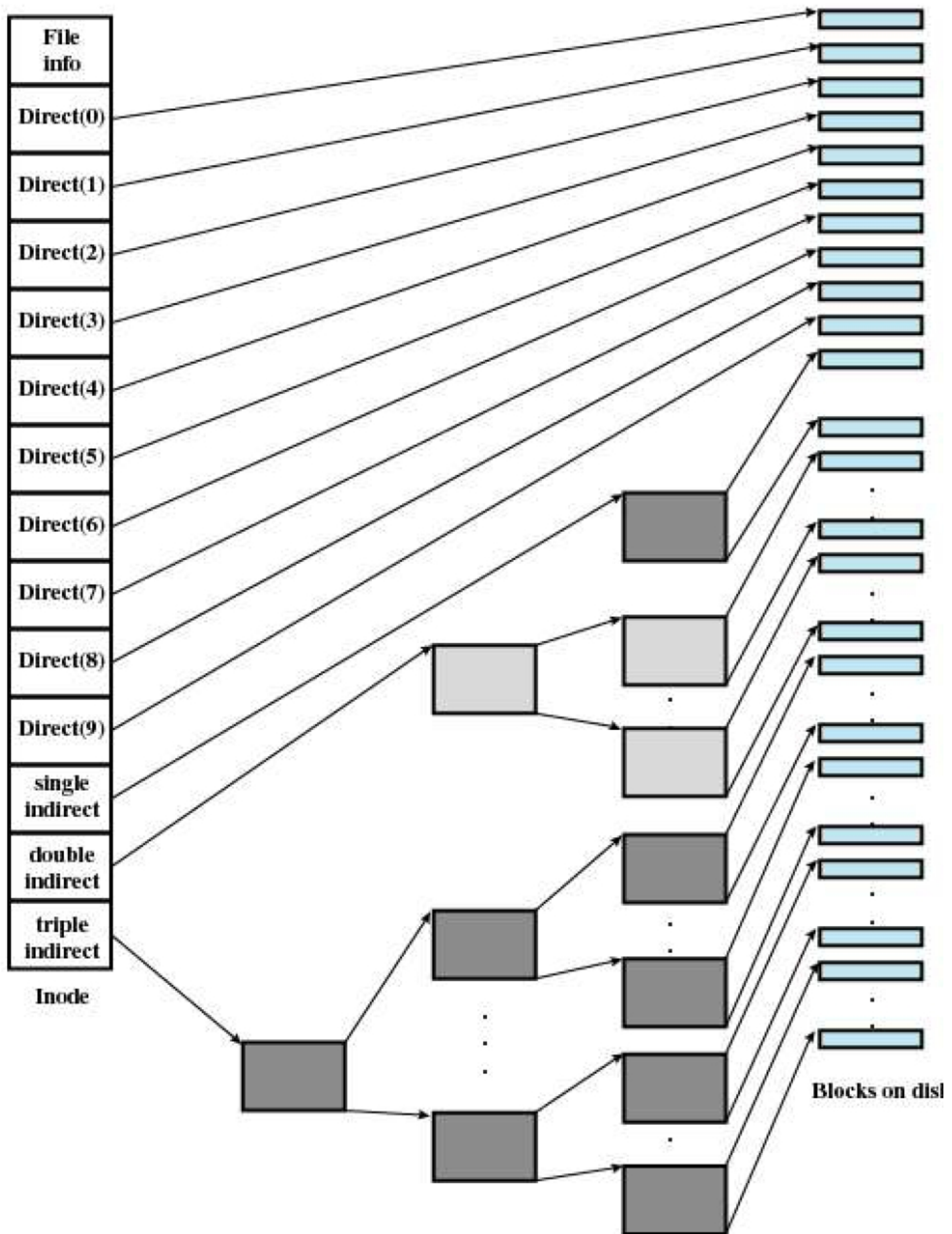


Figure 12.13 Layout of a UNIX File on Disk

TIPI DI FILE SYSTEM

- **FAT**: non ha verifiche di inconsistenza;

- **NTFS**: un pò meglio di FAT;
- **EXT3/EXT2**: in linux. EXT2 (vedi sotto) è come ntfs e contiene info ridondanti: quando spegni il pc a urto, il sist si accorge che tale fs non è stato smontato e quindi va verificato prima del suo riutilizzo-> check all avvio->problema;
- **FS JOURNALED:** meno efficienti ma hanno una sorta di "diario": quando devono fare operazioni, prima scrivono su un file (il file journal), poi lo fanno, poi dicono che l 'hanno fatto; se mentre fanno operazioni va via la corrente non si fa tutto il check, ma si rifà solo ciò che non è stato fatto (sovrascrivendo ciò che è stato già scritto->inefficienza: scrive 2 volte, una su journal e una vera));
- **EXT3:** versione journaled di ext2.

LINUX: pemette di avere tanti tipi di file system grazie al linux virtual file system. nello schema la system call va su una interfaccia del VFS la quale smista la chiamata all'implementazione del fs giusto.

1. **LVFS** (LINUX Virtual File System): Linux si avvale di un file system virtuale con il compito di fornire un interfaccia tra i file system effettivi e il sistema operativo. Il VFS rappresenta, con un modello molto simile a quello ad oggetti, tutti i concetti riguardanti un file system, sia dal punto di vista delle sue caratteristiche che delle sue funzionalità. In pratica presuppone che i files siano oggetti che condividono proprietà generiche indipendentemente dal file system sottostante effettivamente usato. Per ottenere ciò utilizza apposite tabelle che convertono le generiche operazioni del file system virtuale in quelle specifiche del file system che si sta considerando. Alcuni oggetti che hanno un ruolo primario nel VFS sono:
    - i superblock objects, che rappresentano uno specifico file system;
    - gli inode objects, che rappresentano un file specifico;
    - i dentry objects, che modellano le info di una specifica directory;
    - i file objects, che rappresentano invece un file aperto associato con un processo.
2. **FILE SYSTEM EXT2** (Linux):Il file system di Linux è il cosiddetto ext2 (ispirato alla struttura di quello BSD), attualmente utilizzato nella sua più recente versione journalized (cioè è dotato di una sorta di tabella con le ultime modifiche effettuate, utilizzata per il ripristino in caso di crash o spegnimento accidentale), nota come ext3. Ext2 supporta tutte le caratteristiche di un filesystem standard Unix, è in grado di gestire nomi di file lunghi (256 caratteri, estensibili a 1012) con una dimensione massima di 4 Tb.

Oltre alle caratteristiche standard, ext2 fornisce alcune estensioni che non sono presenti sugli altri filesystem Unix.

- Descrittori gruppi: dice dove stanno tutti i gruppi (cio è ridondato su tutti i blocchi: se se ne rompe uno riesco cmq a leggere il file system).
- bitmap: un bit per ciascun blocco di dati (se è =0 è libero, se è =1 è occupato); dice se gli inode(struttura dati fondamentale associata a un file,escluso il nome) sono liberi o meno BITMAP UNA PER GLI INODE E UNA PER I DATI;
- inode sono numerati,identificati da un numero per farne rifermento da una directory(record variabile lunghezza nome file qualsiasi).

Dato il numero dell'inode risalgo al blocco del disco in cui questo inode è memorizzato cosi: gruppi sono tutti stessa lunghezza e contengono stesso numero di inode(inode distribuiti equamente tra i vari gruppi): capisco in che gruppo sto (i/inodeCertoGruppo e vedi il resto della divisione).

DA SHELL: dumpe2fs nomeDispositivo (es: dumpe2fs /dev/hda1)per ottenere info sulla struttura: "quanti inode ha un gruppo,quanti inode totali,quanti sono liberi…".

Le principali sono le seguenti:

- i file attributes consentono di modificare il comportamento del kernel quando agisce su gruppi di file. Possono essere impostati su file e directory e in quest'ultimo caso i nuovi file creati nella directory ereditano i suoi attributi.
- sono supportate entrambe le semantiche di BSD e SVr4 come opzioni di montaggio. La semantica BSD comporta che i file in una directory sono creati con lo stesso identificatore di gruppo della directory che li contiene.

- l'amministratore può scegliere la dimensione dei blocchi del filesystem in fase di creazione, a seconda delle sue esigenze (blocchi più grandi permettono un accesso più veloce, ma sprecano più spazio disco).
- l filesystem implementa link simbolici veloci, in cui il nome del file non è salvato su un blocco, ma tenuto all'interno dell'inode (evitando letture multiple e spreco di spazio), non tutti i nomi però possono essere gestiti così per limiti di spazio (il limite è 60 caratteri).
- vengono supportati i file immutabili (che possono solo essere letti) per la protezione di file di configurazione sensibili, o file appendonly che possono essere aperti in scrittura solo per aggiungere dati (caratteristica utilizzabile per la protezione dei file di log).

Ciascun gruppo di blocchi contiene una copia delle informazioni essenziali del filesystem (superblock e descrittore del filesystem sono quindi ridondati) per una maggiore affidabilità e possibilità di recupero in caso di corruzione del superblock principale. L'utilizzo di raggruppamenti di blocchi ha inoltre degli effetti positivi nelle prestazioni dato che viene ridotta la distanza fra i dati e la tabella degli inode. Le directory sono implementate come una linked list con record di dimensione variabile. Ciascuna voce della lista contiene il numero di inode, la sua lunghezza, il nome del file e la sua lunghezza, secondo lo schema in figura; in questo modo è possibile implementare nomi per i file anche molto lunghi (fino a 1024 caratteri) senza sprecare spazio disco.

Makefile e debugger

Il parametro 0 è il nome dell'eseguibile
$? per la shell è il valore di ritorno dell'ultimo comando eseguito.

Entriamo nella cartella params2. Per compilare diamo un

*gcc -o params *.c*

Questo prende ciascun file .c e lo compila separatamente, prende i .o che escono fuori e li linka insieme per formare l'eseguibile *params.* Questo funziona bene perchè ho pochi files, ma se fossero migliaia? Ogni volta che cambio una virgola ricompilo tutto? No, vorremmo ricompilare solo le parti modificate, ma ricompilare ogni file modificato in modo manuale è un lavoro rognoso.

Vediamo un semplice Makefile. Due regole fondamentali:

- Il nome di file standard si chiama Makefile o makefile (con o senza maiuscola, potremmo forzare il make ad usare altri nomi, ma la pratica è assodata e standardizzata)
- In assenza di specifiche, il primo target è quello eseguito di default (nel nostro caso *params*)

Vediamo riga per riga il Makefile della cartella params2.

params: params_main.o params_print.o

La semantica di questa riga è “raggiungi il target chiamato *params*, a partire dalle dipendenze, che sono *params_main.o* e *params_print.o*”.
Da queste **dipendenze** (params_main, params_print) parto per creare il mio **target**, che si chiama *params*. Potrebbero non esistere i files da cui partiamo, in tal caso dobbiamo occuparci della loro creazione nei passi successivi. Si noti che sono i files.o
. Ora sappiamo qual'è il target e quali sono le dipendenze, ma come creiamo il target? Con la riga successiva, che specifica il comando necessario:

cc -o params params_main.o params_print.o

Nel mondo Unix *cc* vuol dire “cerca il compilatore di default”, nello specifico questa riga vuol dire “cerca il compilatore e compila questa lista di files con queste opzioni”. Attenzione, lo spazio dopo i due punti deve essere una tabulazione, attenzione alla configurazione dell'editor.
Un target rivisto in modo rozzo:

target: **dipendenze**
comando per creare l'oggetto

Torniamo alla creazione del mio *params*: noi creiamo *params* a partire da files .o, ma io ho in mano dei files .c! Con la riga successiva costruisco *params_main.o* a partire dal relativo .c

params_main.o: params_main.c

cc -c -o params_main.o params_main.c

che vuol dire “crea il file params_main.o a partire da params_main.c tramite il comando cc -c -o ecc...”. Analoga è la creazione di *params_print*

params_print.o: params_print.c

cc -c -o params_print.o params_print.c

clean:

*rm params *.o *~*

Clean: usato per fare pulizia: per eseguire il target clean notiamo che:

- clean **non dipende da niente** (e meno male!)
- Fa cancellazione di tutti i files .o, e inoltre cancella i files che finiscono in tilde; questo poichè molti editor creano dei files di backup che terminano in tilde, e li eliminiamo. Occhio a non cancellare i files .c, sennò buttiamo il nostro progetto.

Nota che per ogni target che definiamo nel make, possiamo usarlo:

make params
make clean

Eseguono compiti diversi! Uno costruisce il progetto, l'altro fa pulizia. Usare make liscio (senza parametri) fa eseguire il compito di default (il primo target)

Ora vediamo un Makefile un po' più furbo (preso dalla cartella params3), mica posso scrivere delle liste di migliaia di files...

La prima parte è uguale a quella vista prima:

params: params_main.o params_print.o

cc -o params params_main.o params_print.o

La seguente riga vuol dire che: “Tutti i files .o (target) vengono creati a partire dal relativo file .c (dipendenza)”

%.o: %.c

con che comando? Deve essere un comando parametrico, ovviamente:

cc -c -o $@ $<

Notazioni:

- $@ vuol dire “il target di questa regola”. Se sto creando il file pippo.o allora $@ vuol dire “pippo.o”
- $< sta per “la prima dipendenza”. Se stiamo costruendo “pippo.o” allora %.c e $< contengono “pippo.c”. Vedremo il comando per “tutte le dipendenze”.

In sintesi, creo un target per ogni file .o e lo costruisco a partire dal relativo file .c con il solito comando di compilazione, la cui notazione è spiegata sopra.

clean:

*rm -f params *.o *~*

make clean come sopra: il “-f” sta per cancellazione forzata, vedere *man rm* per maggiori informazioni.

Passiamo alla cartella params4

params: params_main.o params_print.o

*cc -o $@ $^*

$^: **tutte le dipendenze (** magari un file .o dipende da più di un file .c....)

#%.o: %.c

*# cc -c -o $@ $^*

Questa regola è commentata, ma viene eseguita lo stesso. Magia? No, il make ha delle regole predefinite per i "lavori banali". Il comando

Make -np | less

mostra "ciò che il make sa già fare"

clean:

*rm -f params *.o *~*

Questo clean non contiene nulla di nuovo.

Passiamo a Params5, e vediamo come si usano le variabili

esempio: creo una variabile (da usare con **$(nomevariabile)**)che contiene il nome dell'eseguibile, nel nostro caso EXECNAME, che come osserviamo contiene la stringa *params*:

OBJS*= params_main.o params_print.o*

EXECNAME*= params*

$(EXECNAME)*:* ***$(OBJS)***

*cc -o $@ $^*

Solita semantica
: crea l'eseguibile a partire da tutta la lista di oggetti (params_main.o, params_print.o), tramite il comando solito. Si noti che ora è stato usato il simbolo per “tutte le dipendenze”.

%.o: %.c

cc -c -o $@ $<

clean:

*rm -f **$(EXECNAME)** *.o *~*

Stesso lavoro è stato fatto per OBJS, che sta per Objects. OBJS è un nome assegnato da noi. Se passiamo alla cartella params7 notiamo l'introduzione di **CFLAGS.**
CFLAGS aggiunge dei flag di compilazione, per ottimizzare la compilazione in base alle mie necessità. Es.

CFLAGS= -g

l'opzione -g nello specifico significa “metti i simboli di debug”. A questo punto il debugger potrà dare parecchie informazioni quando sarà eseguito questo file. Piccola nota: l'ultimo esercizio d'esame si fa con un debugger, quindi ricordarsi di mettere i simboli di debug...
Se non vogliamo i simboli di debug possiamo levarli con il comando *strip*. Di solito questo alleggerisce molto l'eseguibile compilato.

Debugger: GDB

Possiamo debuggare il programma con

gdb nomeprogramma

Ottenere informazioni: *help nomecomando*(anche usabile con “l”)
list (scritto anche “l”): Lista un pezzo del programma. Il kernel ignora le informazioni di debug nella fase di esecuzione, ma il debugger ovviamente le usa..

l params_print.c:5

sta per “mostrami la quinta linea di codice del file params_print.c”

Nota: dare l'invio a vuoto è rieseguire l'ultimo comando dato. Vediamo ora vari comandi che possono essere usati all'interno di gdb.

B (minuscolo) inserisce un breakpoint. Es:

b main

Inserisce un breakpoint al main

R (minuscolo) fa partire il programma, sempre con gli stessi parametri. Sta per **run.** Se rieseguo **run** mi chiede se voglio far ripartire tutto da capo.

vediamo che si blocca sulla riga 7. Ovvio, è la prima vera istruzione che fa qualche cosa, non si mettono breakpoints su dichiarazioni di variabili.

Il comando **next (scritto anche "n")** passa alla prossima riga.

print (scritto anche "p"): stampa quello che gli viene passato come parametro. Es

(gdb) print argv[1]

$1 = 0x0

step (scritto anche "s"): entra nella funzione.

display i: come la print, ma lo stampa in continuazione, ossia ad ogni **next** che eseguo.

backtrace (scritto anche "bt"): dà informazioni sullo stack della funzione. **Up** mi fa salire al frame della funzione che ha chiamato quella corrente. Ad esempio, se ci trovavamo in params_print dopo una up andremo a vedere in main. Dualmente, **down** mi porta al frame sottostante. (che in realtà è quello della funzione chiamata).

Per inserire un breackpoint per fermarsi all'i-esimo passaggio:

- inserisci il BP nel punto desiderato;
- comando: ig 4 2 → ignora il quarto BP per due volte

es: main---> down--->mi trovo in print_params.
print_params-->up-->mi trovo in main.

Continue: esegue fino alla fine.
info bp (**info breakpoints)** dà informazioni sui breakpoints. Posso disabilitarli con **dis** o eliminarli. Tecniche interessanti prevedono i breakpoint disabilitati *finchè una certa condizione non è verificata*

cond 3 i==3 ignora il bp3 finchè i non vale 3.

Inoltre **ignore (ig)** seguito da due interi:

ig 3 9999

Sta per "ignora il breakpoint 3 per 9999 volte in cui ci si passa sopra, ma alla decimillesima controlla"

Shell Scripting

E' uso comune scrivere l'estensione .sh, ma non è strettamente necessario; inoltre bisogna ricordarsi di assegnare i permessi di esecuzione al file:

chmod +x programma.sh

(+x---> assegna i permessi di eXecute; -x li rimuove). Il comando chmod assegna e rimuove permessi.

Per eseguire il programma bisogna usare ./

./programma.sh

Prima riga del file:
#!/bin/sh

sh indica "shell", ma può essere usato anche altro. La semantica è "esegui il comando /bin/sh ed esegui lo script, il file "pippo"" (se il nome dello script è pippo)
Usare altro: un comando utile per eseguire pippo (pippo è il nome del file che stiamo scrivendo)

es:

#!/usr/bin/awk -f (pippo)

Uso l'eseguibile di awk, con parametro -f legge il programma dal file pippo. Ovviamente in pippo devo mettere il mio programma in awk.

Nota: il carattere # introduce un commento, ma usato in prima riga con il carattere ! notifica al kernel che si tratta di uno script.
Tipici interpreti che si usano nel mondo linux sono:

/bin/sh
/bin/bash (quasi la stessa cosa, spesso sono lo stesso eseguibile; sh è una versione più vecchia della bash)
...altre shell, come cshell, zshell: stessa filosofia della bash
/usr/bin/perl
/usr/bin/python
/usr/bin/awk -f (il -f ci vuole per forza, per passare ad awk un file di input)

Comandi interni: comando "help" dalla shell

Controllo di flusso: for, if, while (raramente usato, tipicamente si usa il for), case (che equivale allo switch di java e c, usato per fare start e stop dei servizi)

Parametri dello script: la variabile $

$1= il primo parametro
$2= il secondo parametro
....
$0= contiene il nome del programma stesso. Questa è una caratteristica standard posix.
Esempio: il seguente script:

```
#! /bin/sh

echo hello world
echo $1 $2 $3
echo $0
```

la sua esecuzione:

```
brian:/Sistemi Operativi/provascript$ ./programma.sh  a w e r t
hello world
a w e
./programma.sh
```

$*= è la concatenazione di tutti i parametri, ma non stampa $0
$#= il numero dei parametri
$$ = pid della shell
$! = pid dell'ultimo processo lanciato.
$?= introduce un altro concetto interessante: il valore di ritorno (l'exit status) dei comandi. Tutti i processi quando ritornano restituiscono un numero; 0 se tutto ok, diverso da 0 se c'è qualche problema; ogni valore è determinato da un certo tipo di errore. Posso usarlo nel caso in cui mi devo chiedere "ma il processo lanciato alla riga precedente è terminato con successo?"

Esempio: il seguente script

```
#! /bin/sh

echo hello world

ps
echo $?
```

Stampa a video:

```
brian:/Sistemi Operativi/provascript$ ./programma.sh  a w e r t
hello world
  PID TTY          TIME CMD
9096 pts/0    00:00:00 bash
9673 pts/0    00:00:01 rmiregistry
9792 pts/0    00:00:01 java
11278 pts/0    00:00:00 programma.sh
11279 pts/0    00:00:00 ps
0<-----------------------------------ecco il risultato di $?
```

Domanda tipica: "esiste in esecuzione un processo con questo nome? Voglio accertarmene prima di startarne uno nuovo". Posso usare ps | grep...

CONTROLLO DI FLUSSO

```
if [ $1 = "ciao" ] ; then
  echo vero
else
  echo falso
fi
```

inizia con IF e finisce con FI. Attenzione agli spazi. Le parentesi quadre sono un'abbreviazione del comando test

Il comando test
prende in input stringhe, dà il risultato di un test (in termini di exit status di un programma). Posso fare controlli come verificare se un nome è un file o una cartella, ecc...

Esempio:

```
brian:/Sistemi Operativi/provascript$ test a = a; echo $?
0
```

I test di tipo aritmetico si fanno con operatori diversi come -gt (greter than) -lt (less than) ecc...

Usare test in una if

```
if test "5" -gt "6" ; then
  echo vero
else
echo falso
fi
```

E' come scrivere, con le quadre,

```
if [ "5" -gt "6" ]; then
  echo vero
else
echo falso
fi
```

Nella if si può usare qualunque programma con un exit status significativo.

IL FOR

```
for i in a b c d e ; do
echo $i
done
```

note:
la prima volta che dichiaro i NON devo usare il dollaro. Dopo si, per richiamarne il valore. La parola chiave è in, che introduce il campo di azione del for. La variabile di ambiente i viene cambiata di volta in volta.La parola chiave che chiude il for è done E' un modo per fare iterazione su stringhe, non su numeri come siamo abituati a fare. Ricordiamo che la shell è pensata per lavorare sui files. Per avere sequenze di numeri possiamo usare il comando seq

esempio: ottenere 10 numeri

```
brian:/Sistemi Operativi/provascript$ seq 1 10
1
2
3
4
5
6
7
8
9
10
```

Come posso usare il comando seq in un for? Con il backquote...

```
for i in `seq 1 10` ; do
echo pippo$i.txt
done
```

(Per l'esecuzione vedere programma1.sh)

```
brian:.../Sistemi Operativi/provascript$ ./programma1.sh ciao
vero
pippo1.txt
pippo2.txt
pippo3.txt
pippo4.txt
pippo5.txt
pippo6.txt
pippo7.txt
pippo8.txt
pippo9.txt
pippo10.txt
```

Uso simile: cat

for... `cat ....` separo il file passato come parametro a cat, splitto in parole e OGNI parola viene processata dal for.

semplificare la lettura: ficcalo in una variabile d'ambiente
x = `cat ....`

Attenzione, la variabile d'ambiente ha dei limiti di lunghezza...

Il costrutto . (punto)

. /ciccio/pollo/file1

vuol dire "leggi il file file1 e includilo"

Si possono usare le funzioni.

Il costrutto CASE

```
case $1 in        //si legga: "se la variabile $1, ossia il primo parametro è..."
        start)    //sintassi fissa, valore e parentesi.
            ... do some commands
      ;;        //indica la fine della gestione del caso "start"

      *)        // valore di default, se nessuno dei precedenti ha matchato

esac            //parola chiave che termina il case
```

~~~~~~~~~~~~~~ altri argomenti trattati nellla stessa lezione~~~~~~~~~~~~~~~~~~~~

cat /proc/cpuinfo----------> dà informazioni sulla CPU

cat /proc/devices----------> dispositivi di cui abbiamo già i driver avviati

Questi non sono files reali, ma sono costruiti dal kernel on the fly. Non ho effettuato letture da disco per ottenere le informazioni per CPU o devices; /proc/ tira fuori delle informazioni di sistema e le rende disponibili alle applicazioni che ne fanno uso.
Esempio: ps legge da /proc
Se guardiamo in /proc ci sono delle cartelle con dei numeri, che corrispondono ai PID dei processi.

Comandi vari

hd = HexDump: fa un dump esadecimale del parametro passato.

strace = System Trace.

Uso:
strace comando

Mostra una riga per ciascuna system call eseguita dal comando comando
Esempio:
strace ls

vediamo le scritture su 1 (identificatore di standard output)

Linking, compilazione e debugging

I programmi sono molto grandi, se in Java siamo abituati a separare in classi in C avviene la stessa cosa, con la separazione del codice in vari files.
Un programma in c sta in un file .c. Questo passa per un precompilatore, che in output dà un precompilato. Questa operazione è fatto dal C pre compiler. Si fa uso di librerie, con la direttiva include. Esistono altre direttive:

#define
~~~~~~~~~~~~~~

#if(n)def
#endif

Le ultime due vanno usate per la compilazione condizionale, ossia non è detto che tutto quello che mettiamo nel file .c viene compilato. Dopo si passa per il compilatore (GCC), e successivamente l'assemblatore tira fuori un semilavorato, cioè dei file oggetto, in formato noto come ELF. Non è il programma completo però, perchè ricordiamo che dobbiamo usare le librerie. Questo file oggetto insieme ad altri file oggetto passa per un linker, e ne viene fuori un eseguibile, sempre in formato ELF, che il kernel è in grado di leggere.

.c

precompiler

GCC

.o (in ELF)

assemblatore

Eseguibile (in ELF)

Linker

precompilato

Altri files .o , che magari sono necessari per l'esecuzione del progetto.

Di fatto eseguire gcc ciccio.c fa tutti questi passaggi.
Si usano infine altri files oggetto che magari sono necessari all'esecuzione del progetto. E' possibile usare i files oggetto anche in mancanza del sorgente (che può essere chiuso o commercialmente protetto), e inoltre aumenta la velocità della compilazione (non vogliamo compilarci la libreria che fa printf ogni volta...).

Il semilavorato non contiene le procedure, ma solo le chiamate, e dei simboli che hanno la semantica: “questo file fornisce queste funzioni (simboli) a,b,c, e necessita di queste funzioni: x,y,z (simboli)”. E' un problema di dipendenze, i files oggetto quindi *forniscono e necessitano* di simboli.

Il ***linker statico*** cerca i simboli, iterativamente cerca le dipendenze finchè tutti i simboli non sono stati risolti. Sostituisce le chiamate a procedura, di fatto sostituisce delle chiamate con degli indirizzi. E' il compilatore che si occupa dei parametri. Il linker da solo non è conscio dei tipi o dei parametri. Prendiamo una funzione di esempio:

int open (char*,...)

tipo di ritorno, nome funzione e parametri. Il compilatore, per usare tale funzione necessita solo di queste informazioni (note come ***prototipo***), non vuole conoscerne tutto il codice. Se la funzione è scritta da noi conosciamo il prototipo (ce l'abbiamo davanti agli occhi), ma in caso di inclusione di librerie non conosciamo il codice che un prototipo nasconde (e meno male). Diventa quindi necessario fornire all'utilizzatore (il programmatore che usa una libreria) il prototipo. Sono stati inventati per questo gli include. ***Con una include si mette nel nostro codice C il prototipo delle funzioni che sono nella libreria.***

Riprendiamo le direttive ifdef, ifndef, ecc..

Define: permette di definire dei simboli (non i simboli del linker, ma vivono nel precompilatore). Questi vivono all'interno del nostro programma.

#Define PI 3,14

ogni volta che nel programma scrivo PI viene considerato come 3.14.
Uno degli usi è per esempio la definizione delle costanti.Possono essere usati per prendere delle decisioni.
Esempio: dobbiamo compilare il sorgente sia su win che su linux, per fare la stessa cosa abbiamo una libreria che gira su win e una sotto linux.

#ifdef //sono sotto linux...
fai qualcosa

#else (allora sono sotto windows!)
do something

(dovrebbe esistere anche else if)

#endif termina il costrutto

Immaginiamo il kernel linux che gira su molte architetture... deve contenere molte possibilità del genere. Quando si precompila si valutano le condizioni delle ifdef, e si inseriscono o meno dei pezzi di codice. Esempio

#ifdef linux

Il precompilatore si chiede se esiste il simbolo linux, e se esiste prende il blocco ad esso collegato e lo include nell'output (quello che andrà in input a gcc), e non inserisce i blocchi di codice collegati a condizioni di ifdef non soddisfatte.

```
#ifdef linux

esegui A

#ifdef win

esegui B
```

se linux esiste, A viene incluso e B no.

Questi costrutti di ifdef li useremo con le include.

Consideriamo il file A.c

```
#include <B.h>
```

e nel file B.c, come nel file B.h c'è scritto che B fa uso di D.

```
#include <D.h>
```

Il programmatore di A conosce solo l'interfaccia che B gli propone, A non sa che B fa uso di D. Il problema è che il programmatore di A.c potebbe importare inconsciamente D.h, quindi inserire nel suo file A.c la seguente riga.

```
#include <D.h>
```

Questo fa incazzare un po' il compilatore... come si risolve? Compilazione condizionale! Ogni volta che usiamo dei file .h lo ficchiamo in una condizione.

Proviamo ad importare pippo.h, vedendone il codice.

```
#ifndef  __pippo_h__
#define  __pippo_h__
...tutto il codice che ci va nel file pippo.h
#endif
```

Nota: gli underscore sono una prassi.

Questo vuol dire che ogni volta che scrivo la include, il precompilatore si chiede: “Ma ho già incluso pippo.h? Se non l'ho incluso ifNdef (**ifNotDefined**) allora mettici tutto il codice che ti scrivo qui sotto”
Semantica terra terra: “Io sono pippo. Sono stato incluso? Se si, skippo la mia inclusione, altrimenti mi autoincludo”.

Vediamo un esempio reale, reperibile in /usr/include/stdio.h:

```
/*
*        ISO C99 Standard: 7.19 Input/output        <stdio.h>
*/
```

```
#ifndef _STDIO_H

#if !defined __need_FILE && !defined __need___FILE
# define _STDIO_H          1
# include <features.h>

...altra roba...
```

Compilazione

gcc hello.c

fornisce il file a.out (nome di default). Per assegnare un nome usiamo l'opzione -o

gcc hello.c -o hello

fornisce il file eseguibile
hello

proviamo a linkare staticamente, con l'opzione --static. Da 6 Kb (linking dinamico) di prima siamo passati a 500 kb, con uno stupido programma HelloWorld.

Se esploriamo il file hello così creato (con comando "file hello") notiamo un "not striped", il che vuol dire che il file si porta appresso un sacco di informazioni aggiuntive, visualizzabili con Objdump.
Objdump file
Nella sezione .text c'è il codice prodotto dal linker ed eseguito dal kernel.

Volendo si può vedere la tabella dei simboli precedentemente citata, (vedere man per dettagli)
per vedere le librerie dinamiche che abbiamo linkato possiamo usare il comando

ldd

esempio:

```
brian:~.../Sistemi Operativi$ ldd /bin/ls
        linux-gate.so.1 =>  (0xb80d7000)
        librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb809b000)
        libselinux.so.1 => /lib/libselinux.so.1 (0xb8081000)
        libacl.so.1 => /lib/libacl.so.1 (0xb8078000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7f1a000)
        libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7f01000)
        /lib/ld-linux.so.2 (0xb80bd000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7efd000)
        libattr.so.1 => /lib/libattr.so.1 (0xb7ef8000)
```

Se usiamo Objectdump con opzione -D vediamo il disassemblato:

vediamo una start (punto di ingresso della funzione), un po' di push e di preparazione ambiente, una chiamata al main.
Se vediamo il disassemblato del file linkato staticamente troviamo una marea di roba, cioè anche le lbibrerie che sono state incluse.

Gcc -c opzione per dire di compilare e fermarsi alla fase di creazione dei .o
gcc -c hello.c -o hello.o

ora con objdump vediamo che c'è una strana chiamata al main. Ricordiamo che questo è un semilavorato, la semantica è “chiama la funzione 19, chiamata main” (nell'esempio visto a lezione c'era 19).

Comando **ltrace:** mostra le chiamate alle librerie.

Domande d'esame

* spazio di indirizzamento di un processo
  - gli indirizzi logici che un processo può usare con il relativo spazio di memoria virtuale

* regione allocata
  - parte contigua dello spazio di indirizzamento che e' legalemente accessibile al processo

* memory mapping
  - tecnica che permette di accedere e modificare byte di un file mediante accessi ad una regione dello spazio di indirizzamento associata a tale file