Corso di Sistemi Operativi I

Laurea in Informatica

a.a. 2003/04

Docenti

- Teoria:
  - Giorgio Delzanno
  - ufficio 104
  - tel. 6638
  - e-mail: giorgio@disi.unige.it
- Laboratorio:
  - Stefano Bencetti
  - e-mail: bencetti@disi.unige.it

Informazioni

- Orario (da Ottobre-a Dicembre)
  - Lunedi 11-13 (?)
  - Martedi 14-16
  - Venerdi 9-11
- 56 ore di teoria
  - Introduzione ai Sistemi Operativi
  - Gestione processi, memoria, fyle system, device
- 16 di laboratorio
  - Gestione di una macchina come amministratore di sistema
  - Installazione e configurazione di hardware, applicazioni e servizi

Esame

- Il corso vale 9 crediti
- Scritto, probabilmente spezzato in compitini, con esercizi e domande di teoria
- Orale per la parte di laboratorio e per la verifica dello scritto

Testi, Appunti e Info sul corso

- Riferimenti
  - I moderni sistemi operativi - Seconda ed. - Andrew S. Tanenbaum - Jackson libri Università
- Letture integrative
  - Sistemi Operativi: Concetti ed esempi - Sesta edizione - Silberschatz, Galvin, Gagne - Addison Wesley
  - UNIX: Architettura di sistema - Maurice J. Bach - Jackson
- Pagina web:

http://www.disi.unige.it/person/DelzannoG/SO1

Trasparenze su Sistemi Operativi I

Copyright © 2000-03 Marino Miculan (miculan@dimi.uniud.it)

La copia letterale e la distribuzione di questa presentazione nella sua integrità sono permesse con qualsiasi mezzo, a condizione che questa nota sia riprodotta.

2003-04 Modificati ed integrati con altro materiale da Giorgio Delzanno su autorizzazione di Marino Miculan

Introduzione

- Cosa è un sistema operativo?
- Evoluzione dei sistemi operativi
- Tipi di sistemi operativi
- Concetti fondamentali
- Chiamate di sistema
- Struttura dei Sistemi Operativi

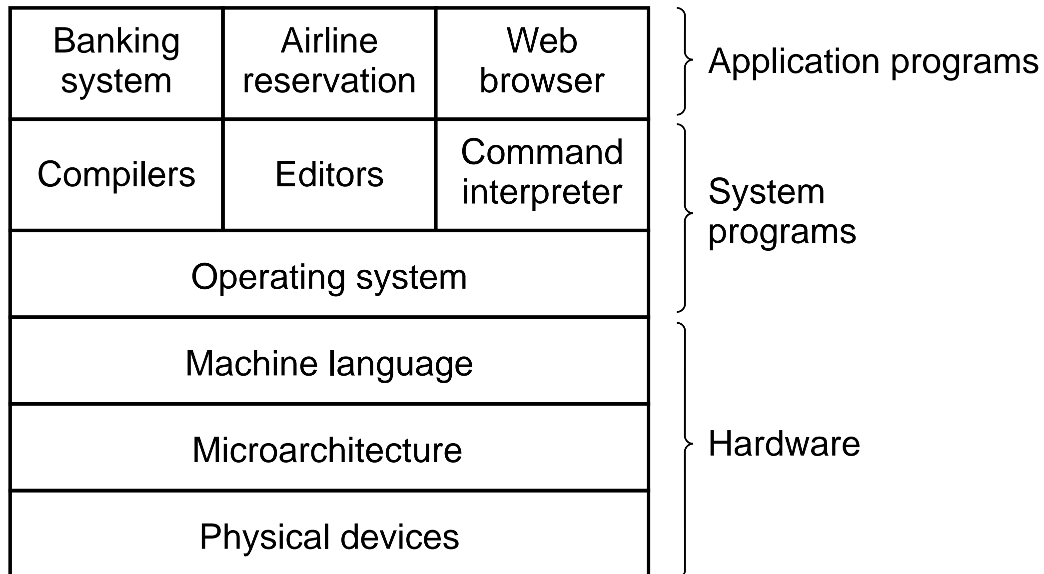
Cosa è un sistema operativo?

Possibili risposte:

- È un *programma di controllo*
- È un *gestore di risorse*
- È un *fornitore di servizi*
- . . .

Nessuna di queste definizioni è completa

Visione astratta delle componenti di un sistema di calcolo



Componenti di un sistema di calcolo

1. Hardware – fornisce le risorse computazionali di base: (CPU, memoria, dispositivi di I/O).
2. Sistema operativo – controlla e coordina l'uso dell'hardware tra i programmi applicativi per i diversi utenti
3. Programmi applicativi — definiscono il modo in cui le risorse del sistema sono usate per risolvere i problemi computazionali dell'utente (database, videgiochi, programmi di produttività personale,. . .)
4. Utenti (persone, macchine, altri calcolatori)

Cosa è un sistema operativo? (2)

- Un programma che agisce come intermediario tra l'utente di un calcolatore e l'hardware del calcolatore stesso.
- Obiettivi di un sistema operativo:
  - Eseguire programmi utente e rendere più facile la soluzione dei problemi dell'utente
  - Rendere il sistema di calcolo più facile da utilizzare
  - Utilizzare l'hardware del calcolatore in modo efficiente

  Questi obiettivi sono in contrapposizione. A quale obiettivo dare priorità dipende dal contesto.

Alcune definizioni di Sistema Operativo

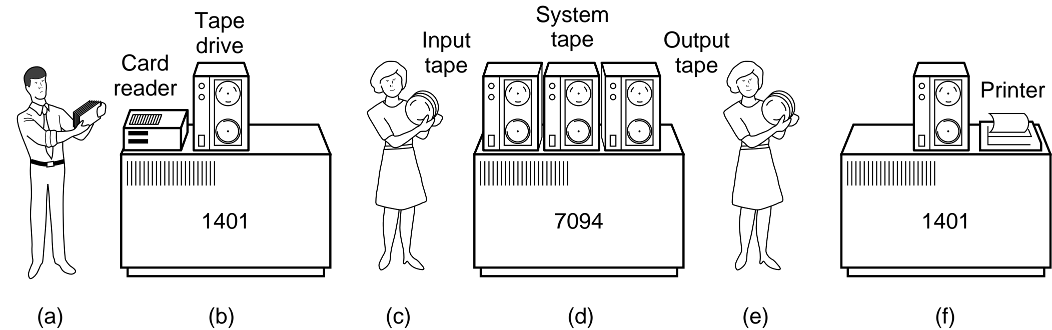
- Macchina astratta
  Implementa funzionalità di alto livello, nascondendo dettagli di basso livello.
- Allocatore di risorse
  Gestisce ed alloca le risorse finite della macchina.
- Programma di controllo
  Controlla l'esecuzione dei programmi e le operazioni sui dispositivi di I/O.
  Condivisione rispetto al tempo e rispetto allo spazio

Primi sistemi – Macchine nude e crude (primi anni '50)

- Struttura
  - Grossi calcolatori funzionanti solo da console
  - Sistemi single user; il programmatore era anche utente e operatore
  - I/O su nastro perforato o schede perforate
- Primi Software
  - Assemblatori, compilatori, linker, loader
  - Librerie di subroutine comuni
  - Driver di dispositivi
- Uso inefficiente di risorse assai costose
  - Bassa utilizzazione della CPU
  - Molto tempo impiegato nel setup dei programmi

Semplici Sistemi Batch

- Utente ≠ operatore
- Aggiungere un lettore di schede



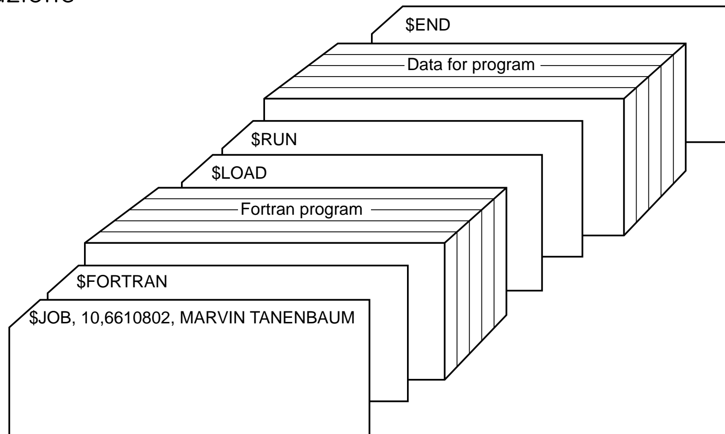
- Ridurre il tempo di setup raggruppando i job simili (*batch*)
- Sequenzializzazione automatica dei job – automaticamente, il controllo passa da un job al successivo. Primo rudimentale sistema operativo
- Monitor residente
  - inizialmente, il controllo è in monitor
  - poi viene trasferito al job
  - quando il job è completato, il controllo torna al monitor

Semplici Sistemi Batch (Cont.)

- Problemi
  1. Come fa il monitor a sapere la natura del job (e.g., Fortran o assembler?) o quale programma eseguire sui dati forniti?
  2. Come fa il monitor a distinguere
     (a) un job da un altro
     (b) dati dal programma
- Soluzione: schede di controllo

Schede di controllo

- Schede speciali che indicano al monitor residente quali programmi mandare in esecuzione

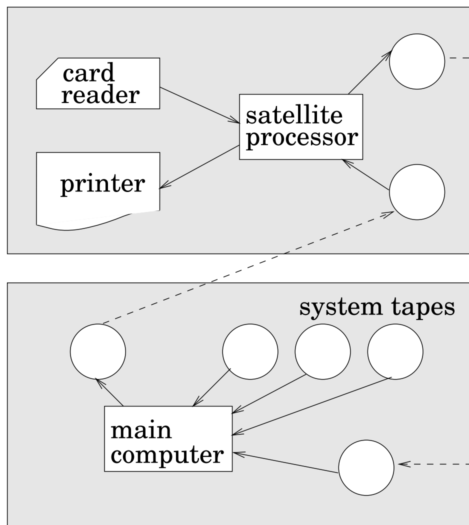


- Caratteri speciali distinguono le schede di controllo dalle schede di programma o di dati.

Schede di controllo (Cont.)

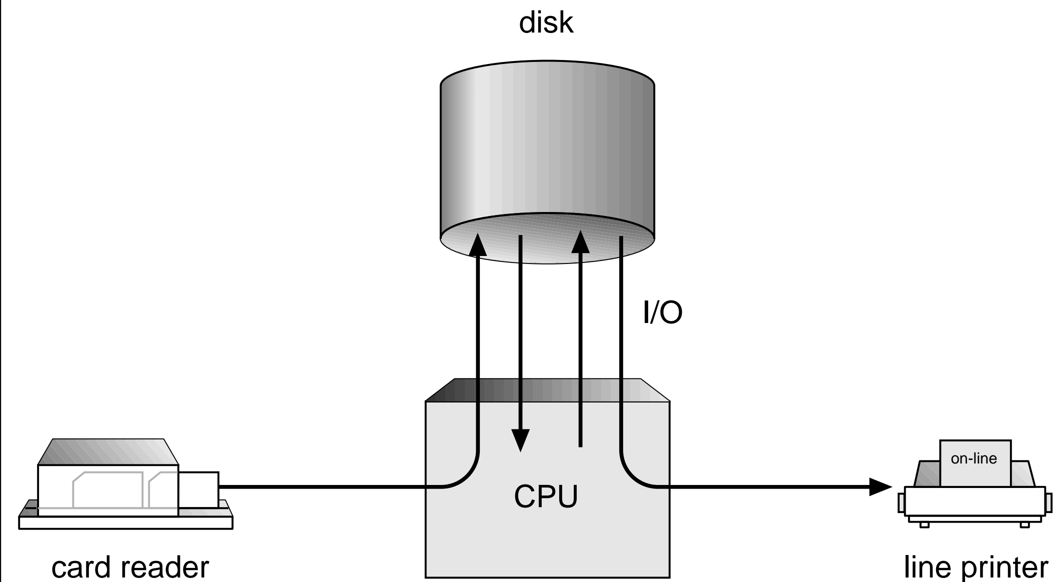
- Una parte del monitor residente è
  - Inteprete delle schede di controllo – responsabile della lettura e esecuzione delle istruzioni sulle schede di controllo
  - Loader – carica i programmi di sistema e applicativi in memoria
  - Driver dei dispositivi – conoscono le caratteristiche e le proprietà di ogni dispositivo di I/O.
- Problema: bassa performance – I/O e CPU non possono sovrapporsi; i lettore di schede sono molto lenti.
- Soluzione: operazioni off-line – velocizzare la computazione caricando i job in memoria da nastri, mentre la lettura e la stampa vengono eseguiti off-line

Funzionamento Off-Line



- Il computer principale non è limitato dalla velocità dei lettori di schede o stampanti, ma solo dalla velocità delle unità nastro.
- Non si devono fare modifiche nei programmi applicativi per passare dal funzionamento diretto a quello off-line
- Guadagno in efficienza: si può usare più lettori e più stampanti per una CPU.

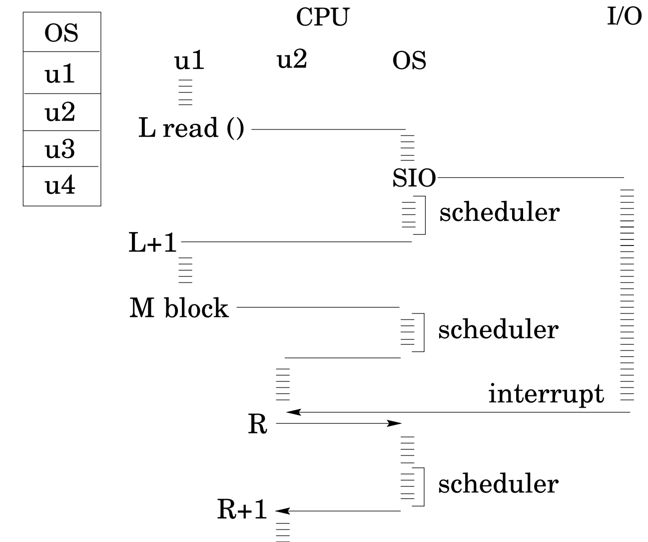
Spooling



- Spool = Simultaneous peripheral operation on-line
- Sovrapposizione dell'I/O di un job con la computazione di un altro job. Mentre un job è in esecuzione, il sistema operativo
  - legge il prossimo job dal lettore di schede in un'area su disco (coda dei job)
  - trasferisce l'output del job precedente dal disco alla stampante
- *Job pool* – struttura dati che permette al S.O. di scegliere quale job mandare in esecuzione al fine di aumentare l'utilizzazione della CPU.

Anni 60: Sistemi batch Multiprogrammati

Più job sono tenuti in memoria nello stesso momento, e la CPU fa a turno su tutti i job



Caratteristiche dell'OS richieste per la multiprogrammazione

- routine di I/O devono essere fornite dal sistema
- Gestione della Memoria – il sistema deve allocare memoria per più job
- Scheduling della CPU – il sistema deve scegliere tra più job pronti per l'esecuzione
- Allocazione dei dispositivi

Anni 70: Sistemi Time-Sharing – Computazione Interattiva

- La CPU è condivisa tra più job che sono tenuti in memoria e su disco (la CPU è allocata ad un job solo se questo si trova in memoria)
- Un job viene caricato dal disco alla memoria, e viceversa (*swapping*)
- Viene fornita una comunicazione on-line tra l'utente e il sistema; quando il sistema operativo termina l'esecuzione di un comando, attende il prossimo "statement di controllo" non dal lettore di schede bensí dalla tastiera dell'utente.
- Deve essere disponibile un file system on-line per poter accedere ai dati e al codice

Anni 80: Personal Computer

- *Personal computers* – sistemi di calcolo dedicati ad un singolo utente
- I/O devices – tastiere, mouse, schermi, piccole stampanti
- Comodità per l'utente e reattività
- Interfaccia utente evoluta (GUI)
- Spesso gli individui hanno un uso esclusivo del calcolatore, e non necessitano di avanzate tecniche di sfruttamento della CPU o sistemi di protezione.

Anni 90: Sistemi operativi di rete

- Distribuzione della computazione tra più processori
- Sistemi *debolmente accoppiati* – ogni processore ha la sua propria memoria; i processori comunicano tra loro attraverso linee di comunicazione (e.g., bus ad alta velocità, linee telefoniche, fibre ottiche,. . .)
- In un sistema operativi di rete, l'utente ha coscienza della differenza tra i singoli nodi.
  - Trasferimenti di dati e computazioni avvengono in modo esplicito
  - Poco tollerante ai guasti
  - Complesso per gli utenti

Il futuro: Sistemi operativi distribuiti

- In un sistema operativo distribuito, l'utente ha una visione *unitaria* del sistema di calcolo.
  - Condivisione delle risorse (dati e computazionali)
  - Aumento della velocità – bilanciamento del carico
  - Tolleranza ai guasti
- Un sistema operativo distribuito è molto più complesso di un SO di rete.
- Esempi di servizi (non sistemi) di rete: NFS, P2P (KaZaA, Gnutella, . . .), Grid computing. . .

Riepilogo

- I generazione ('45-'55): relè/valvole, no sistema operativo
- II generazione ('55-'65): transistor e schede performate
  - sistemi batch: IBM 1401 (scheda ⇔ nastro) e IBM 7094 (calcolo)
- III generazione ('65-'80): circuiti integrati
  - compatibilita' tra macchine IBM diverse (360,370, . . .)
  - OS/360 con spooling e multiprogrammazione
  - MULTICS: servizio centralizzato e time-sharing
  - PDP-1 . . . -11: minicalcolatori a 18bit
  - UNIX: Versione singolo utente di MULTICS per PDP-7

- IV Generazione ('80-oggi): circuiti integrati su larga scala
  - Personal Computer IBM e MS-DOS
  - MacIntosh di Apple con GUI (Graphical User Interface)
  - Sistema operativo Windows
    - * Windows costruito su DOS
    - * Windows 95 e Windows 98 (ancora con codice assembly a 16bit)
    - * Windows NT e Windows 2000 (a 32bit)
    - * Windows Me (update di Windows 98)
    - * Windows XP
  - Linux versione open source di Unix

Tipologie di Sistemi Operativi

Diversi obiettivi e requisiti a seconda delle situazioni

- Supercalcolatori
- Mainframe
- Server
- Multiprocessore
- Personal Computer
- Real Time
- Embedded

Sistemi operativi per mainframe

- Grandi quantità di dati ($> 1TB \simeq 10^{12}B$)
- Grande I/O
- Elaborazione "batch" non interattiva
- Assoluta stabilità (uptime $> 99,999\%$)
- Applicazioni: banche, amministrazioni, ricerca...
- Esempi: IBM OS/360, OS/390

Sistemi operativi per supercalcolatori

- Grandi quantità di dati ($> 1TB$)
- Enormi potenze di calcolo (es. NEC Earth-Simulator, 40 TFLOP)
- Architetture con migliaia di CPU
- Elaborazione "batch" non interattiva
- Esempi: Unix, o ad hoc

Sistemi per server

- Sistemi multiprocessore con spesso più di una CPU in comunicazione stretta.
- Rilevamento automatico dei guasti
- Elaborazione su richiesta (semi-interattiva)
- Applicazioni: server web, di posta, dati, etc.
- Esempi: Unix, Linux, Windows NT e derivati

Sistemi Real-Time

- Vincoli temporali fissati e ben definiti
- Sistemi *hard real-time*: i vincoli devono essere soddisfatti (es. fermare il braccio meccanico)
  - la memoria secondaria è limitata o assente; i dati sono memorizzati o in memoria volatile, o in ROM.
  - In confitto con i sistemi time-sharing; non sono supportati dai sistemi operativi d'uso generale
  - Usati in robotica, controlli industriali, software di bordo. . .
- Sistemi *soft real-time*: i vincoli possono anche non essere soddisfatti, ma il sistema operativo deve fare del suo meglio
  - Uso limitato nei controlli industriali o nella robotica

  - Utili in applicazioni (multimedia, virtual reality) che richiedono caratteristiche avanzate dei sistemi operativi

Sistemi operativi embedded

- Per calcolatori palmari (PDA), cellulari, ma anche televisori, forni a microonde, lavatrici, etc.
- Hanno spesso caratteristiche di real-time
- Limitate risorse hardware
- esempio: PalmOS, Epoc, PocketPC, QNX.

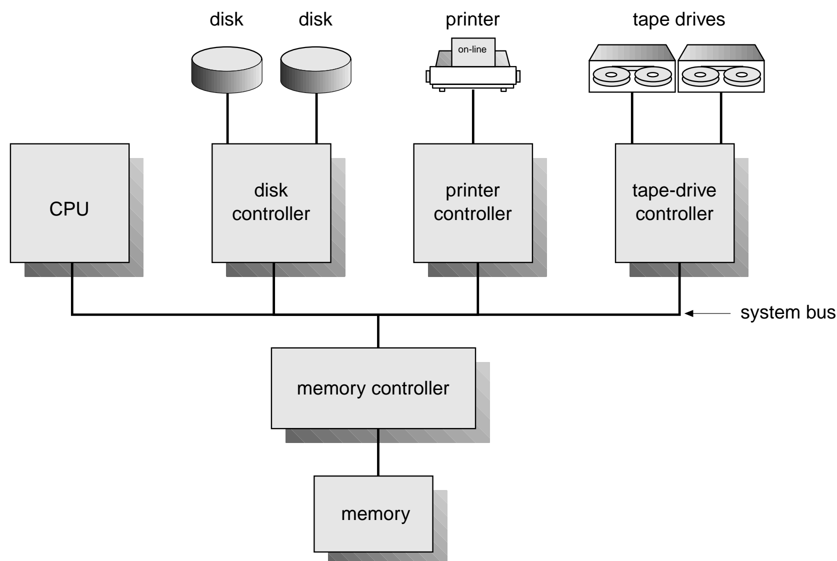
Sistemi operativi per smart card

- Girano sulla CPU delle smartcard
- Stretti vincoli sull'uso di memoria e alimentazione
- implementano funzioni minime
- Esempio: JavaCard

Struttura dei Sistemi di Calcolo

- Operazioni dei sistemi di calcolo
- Struttura dell'I/O
- Struttura della memoria
- Gerarchia delle memorie
- Protezione hardware
- Invocazione del Sistema Operativo

Architettura dei calcolatori



Operazioni dei sistemi di calcolo

- I dispositivi di I/O e la CPU possono funzionare concorrentemente
- Ogni controller di dispositivo gestisce un particolare tipo di dispositivo.
- Ogni controller ha un buffer locale
- La CPU muovi dati da/per la memoria principale per/da i buffer locali dei controller
- l'I/O avviente tra il dispositivo e il buffer locale del controller
- Il controller informa la CPU quando ha terminato la sua operazione, generando un *interrupt*.

Funzioni comuni degli Interrupt

- Gli interrupt trasferiscono il controllo alla routine di servizio dell'interrupt, generalmente attraverso il *vettore di interruzioni*, che contiene gli indirizzi di tutte le routine di servizio.
- L'hardware deve salvare l'indirizzo dell'istruzione interrotta.
- Interrupt in arrivo sono *disabilitati* mentre un altro interrupt viene gestito, per evitare che vadano perduti.
- Un *trap* è un interrupt generato da software, causato o da un errore o da una esplicita richiesta dell'utente.
- Un sistema operativo è *guidato da interrupt*

Gestione degli Interrupt

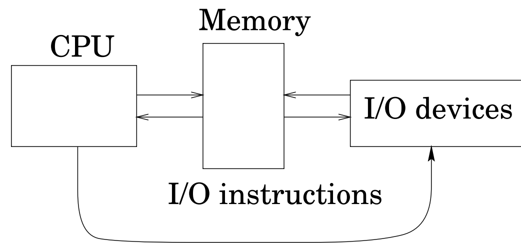
- Il sistema operativo preserva lo stato della CPU salvando registri e program counter.
- Determinazione di quale tipo di interrupt è avvenuto:
  - *polling*
  - vettore di interrupt
- Per ogni tipo di interrupt, uno specifico segmento di codice determina cosa deve essere fatto.

Struttura dell'I/O

- I/O sincrono: dopo che l'I/O è partito, il controllo ritorna al programma utente solo dopo che l'I/O è stato completato
  - l'istruzione **wait** blocca la CPU fino alla prossima interruzione
  - oppure, un tramite un ciclo di attesa (*busy wait*)
  - al più una richiesta di I/O è eseguita alla volta; non ci sono I/O paralleli

- I/O asincrono: dopo che l'I/O è partito, il controllo ritorna al programma utente senza aspettare che l'I/O venga completato
  - *chiamata di sistema* (*System call*) – richiede al sistema operativo di sospendere il processo in attesa del completamento dell'I/O.
  - Se non ci sono processi da eseguire la CPU esegue un'istruzione **wait**
  - una *tabella dei dispositivi* mantiene tipo, indirizzo e stato di ogni dispositivo di I/O.
  - Il sistema operativo accede alla tabella dei dispositivi per determinare lo stato, e per mantenere le informazioni relative agli interrupt.

Struttura del Direct Memory Access (DMA)



- Usata per dispositivi in grado di trasferire dati a velocità prossime a quelle della memoria

- I controller trasferiscono blocchi di dati dal buffer locale direttamente alla memoria, senza intervento della CPU.

- Viene generato un solo interrupt per blocco, invece di uno per ogni byte trasferito.

Struttura della Memoria

- Memoria principale (RAM) – la memoria che la CPU può accedere direttamente.

- Memoria secondaria (Dischi, floppy, CD, ...) – estensione della memoria principale che fornisce una memoria non volatile (e solitamente più grande)

Gerarchia della Memoria

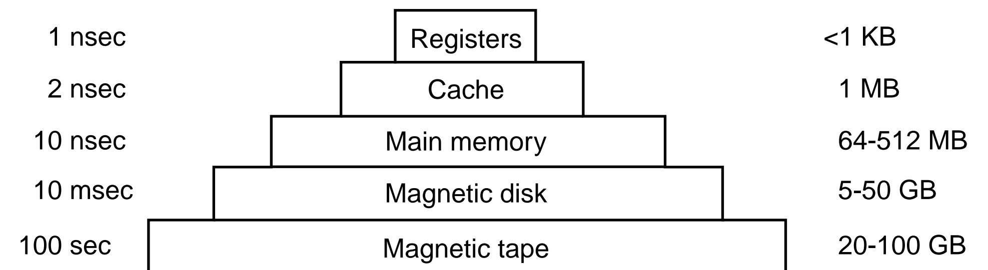
I sistemi di memorizzazione sono organizzati gerarchicamente, secondo

- velocità

- costo

- volatilità

Caching – duplicare i dati più frequentemente usati di una memoria, in una memoria più veloce. La memoria principale può essere vista come una cache per la memoria secondaria.



Protezione hardware

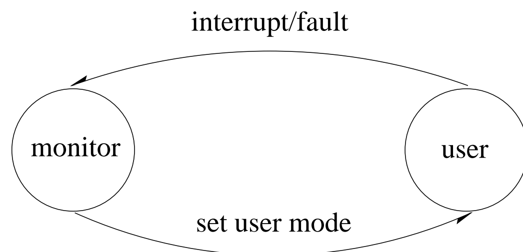
- Funzionamento in dual-mode
- Protezione dell'I/O
- Protezione della Memoria
- Protezione della CPU

Funzionamento Dual-Mode

- La condivisione di risorse di sistema richiede che il sistema operativo assicuri che un programma scorretto non possa portare altri programmi (corretti) a funzionare non correttamente.
- L'hardware deve fornire un supporto per differenziare almeno tra due modi di funzionamento
  1. *User mode* – la CPU sta eseguendo codice di un utente
  2. *Monitor mode* (anche *supervisor mode*, *system mode*, *kernel mode*) – la CPU sta eseguendo codice del sistema operativo

Funzionamento Dual-Mode (Cont.)

- La CPU ha un *Mode bit* che indica in quale modo si trova: supervisor (0) o user (1).
- Quando avviene un interrupt, l'hardware passa automaticamente in modo supervisore



- Le *istruzioni privilegiate* possono essere eseguite solamente in modo supervisore

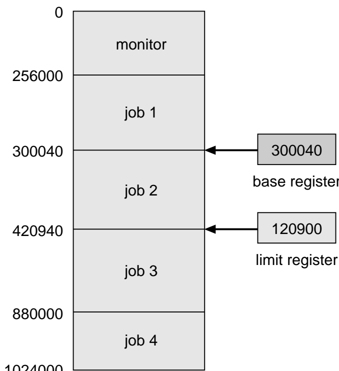
Protezione dell'I/O

- Tutte le istruzione di I/O sono privilegiate
- Si deve assicurare che un programma utente non possa mai passare in modo supervisore (per esempio, andando a scrivere nel vettore delle interruzioni)

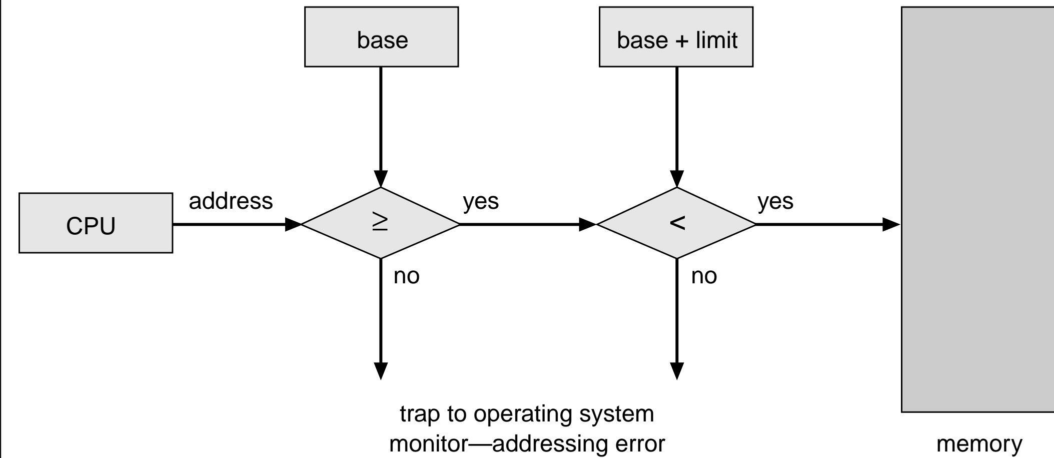
Protezione della Memoria

Si deve proteggere almeno il vettore delle interruzioni e le routine di gestione degli interrupt

- Per avere la protezione della memoria, si aggiungono due registri che determinano il range di indirizzi a cui un programma può accedere:
  registro base contiene il primo indirizzo fisico legale
  registro limite contiene la dimensione del range di memoria accessibile
- la memoria al di fuori di questo range è protetta



Protezione della Memoria (Cont.)



- Essendo eseguito in modo monitor, il sistema operativo ha libero accesso a tutta la memoria, sia di sistema sia utente
- Le istruzioni di caricamento dei registri base e limite sono privilegiate

Protezione della CPU

- il *Timer* interrompe la computazione dopo periodi prefissati, per assicurare che periodicamente il sistema operativo riprenda il controllo
  - Il timer viene decrementato ad ogni *tick* del clock (1/50 di secondo, tipicamente)
  - Quanto il timer va a 0, avviene l'interrupt
- Il timer viene usato comunemente per implementare il time sharing
- Serve anche per mantenere la data e l'ora
- Il caricamento del timer è una istruzione privilegiata

Invocazione del sistema operativo

- Dato che le istruzioni di I/O sono privilegiate, come può il programma utente eseguire dell'I/O?
- Attraverso le *system call* – il metodo con cui un processo richiede un'azione da parte del sistema operativo
  - Solitamente sono un interrupt software (**trap**)
  - Il controllo passa attraverso il vettore di interrupt alla routine di servizio della trap nel sistema operativo, e il mode bit viene impostato a "monitor".
  - Il sistema operativo verifica che i parametri siano legali e corretti, esegue la richiesta, e ritorna il controllo all'istruzione che segue la system call.
  - Con l'istruzione di ritorno, il mode bit viene impostato a "user"

Struttura dei Sistemi Operativi

- Componenti del sistema
- Servizi del Sistema Operativo
- Chiamate di sistema (*system calls*)
- Programmi di Sistema
- Struttura del Sistema
- Macchine Virtuali

Componenti comuni dei sistemi

1. Gestione dei processi
2. Gestione della Memoria Principale
3. Gestione della Memoria Secondaria
4. Gestione dell'I/O
5. Gestione dei file
6. Sistemi di protezione
7. Connessioni di rete (*networking*)
8. Sistema di interpretazione dei comandi

Gestione dei processi

- Un *processo* è un programma in esecuzione. Un processo necessita di certe risorse, tra cui tempo di CPU, memoria, file, dispositivi di I/O, per assolvere il suo compito.
- Il sistema operativo è responsabile delle seguenti attività, relative alla gestione dei processi:
  - creazione e cancellazione dei processi
  - sospensione e riesumazione dei processi
  - fornire meccanismi per
    - sincronizzazione dei processi
    - comunicazione tra processi
    - evitare, prevenire e risolvere i *deadlock*

Gestione della Memoria Principale

- La *memoria principale* è un (grande) array di parole (byte, words...), ognuna identificata da un preciso indirizzo. È un deposito di dati rapidamente accessibili dalla CPU e dai dispositivi di I/O.
- La memoria principale è *volatile*. Perde il suo contenuto in caso di *system failure*.
- Il sistema operativo è responsabile delle seguenti attività relative alla gestione della memoria:
  - Tener traccia di quali parti della memoria sono correntemente utilizzate, e da chi.
  - Decidere quale processo caricare in memoria, quando dello spazio si rende disponibile.
  - Allocare e deallocare spazio in memoria, su richiesta.

Gestione della memoria secondaria

- Dal momento che la memoria principale è volatile e troppo piccola per contenere tutti i dati e programmi permanentemente, il calcolatore deve prevedere anche una *memoria secondaria* di supporto a quella principale.
- La maggior parte dei calcolatori moderni utilizza *dischi* come principale supporto per la memoria secondaria, sia per i programmi che per i dati.
- Il sistema operativo è responsabile delle seguenti attività relative alla gestione dei dischi:
  - Gestione dello spazio libero
  - Allocazione dello spazio
  - Schedulazione dei dischi

Gestione del sistema di I/O

- Il sistema di I/O consiste in
  - un sistema di cache a buffer
  - una interfaccia generale ai gestori dei dispositivi (*device driver*)
  - i driver per ogni specifico dispositivo hardware (controller)

Gestione dei File

- Un *file* è una collezione di informazioni correlate, definite dal suo creatore. Comunemente, i file rappresentano programmi (sia sorgenti che eseguibili (*oggetti*)) e dati.
- Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei file:
  - Creazione e cancellazione dei file
  - Creazione e cancellazione delle directory
  - Supporto di primitive per la manipolazione di file e directory
  - Allocazione dei file nella memoria secondaria
  - Salvataggio dei dati su supporti non volatili

Sistemi di protezione

- Per *Protezione* si intende un meccanismo per controllare l'accesso da programmi, processi e utenti sia al sistema, sia alle risorse degli utenti.
- Il meccanismo di protezione deve:
  - distinguere tra uso autorizzato e non autorizzato.
  - fornire un modo per specificare i controlli da imporre
  - forzare gli utenti e i processi a sottostare ai controlli richiesti

Networking (Sistemi Distribuiti)

- Un *sistema distribuito* è una collezione di processori che non condividono memoria o clock. Ogni processore ha una memoria propria.
- I processori del sistema sono connessi attraverso una *rete di comunicazione.*
- Un sistema distribuito fornisce agli utenti l'accesso a diverse risorse di sistema.
- L'accesso ad una risorsa condivisa permette:
  - Aumento delle prestazioni computazionali
  - Incremento della quantità di dati disponibili
  - Aumento dell'affidabilità

Interprete dei comandi

- Molti comandi sono dati al sistema operativo attraverso *control statement* che servono per
  - creare e gestire i processi
  - gestione dell'I/O
  - gestione della memoria secondaria
  - gestione della memoria principale
  - accesso al file system
  - protezione
  - networking

Interprete dei comandi (Cont.)

- Il programma che legge e interpreta i comandi di controllo ha diversi nomi:
  - interprete delle schede di controllo (sistemi batch)
  - interprete della linea di comando (DOS, Windows)
  - shell (in UNIX)
  - interfaccia grafica: Finder in MacOS, Explorer in Windows, gnome-session in Unix. . .

  La sua funzione è di ricevere un comando, eseguirlo, e ripetere.

Servizi dei Sistemi Operativi

- Esecuzione dei programmi: caricamento dei programmi in memoria ed esecuzione.
- Operazioni di I/O: il sistema operativo deve fornire un modo per condurre le operazioni di I/O, dato che gli utenti non possono eseguirle direttamente,
- Manipolazione del file system: capacità di creare, cancellare, leggere, scrivere file e directory.
- Comunicazioni: scambio di informazioni tra processi in esecuzione sullo stesso computer o su sistemi diversi collegati da una rete. Implementati attraverso *memoria condivisa* o *passaggio di messaggi*.
- Individuazione di errori: garantire una computazione corretta individuando errori nell'hardware della CPU o della memoria, nei dispositivi di I/O, o nei programmi degli utenti.

Funzionalità addizionali dei sistemi operativi

Le funzionalità addizionali esistono per assicurare l'efficienza del sistema, piuttosto che per aiutare l'utente

- Allocazione delle risorse: allocare risorse a più utenti o processi, allo stesso momento
- Accounting: tener traccia di chi usa cosa, a scopi statistici o di rendicontazione
- Protezione: assicurare che tutti gli accessi alle risorse di sistema siano controllate

Chiamate di Sistema (System Calls)

- Le chiamate di sistema formano l'interfaccia tra un programma in esecuzione e il sistema operativo.
  - Generalmente, sono disponibili come speciali istruzioni assembler
  - Linguaggi pensati per programmazione di sistema permettono di eseguire system call direttamente (e.g., C, Bliss, PL/360).
- Tre metodi generali per passare parametri tra il programma e il sistema operativo:
  - Passare i parametri nei *registri*.
  - Memorizzare i parametri in una tabella in memoria, il cui indirizzo è passato come parametro in un registro.
  - Il programma mette i parametri sullo *stack*, da cui il sistema operativo li recupera.

Tipi di chiamate di sistema

Controllo dei processi: creazione/terminazione processi, esecuzione programmi, (de)allocazione memoria, attesa di eventi, impostazione degli attributi,...

Gestione dei file: creazione/cancellazione, apertura/chiusura, lettura/scrittura, impostazione degli attributi,...

Gestione dei dispositivi: allocazione/rilascio dispositivi, lettura/scrittura, collegamento logico dei dispositivi (e.g. mounting)...

Informazioni di sistema: leggere/scrivere data e ora del sistema, informazioni sull'hardware/software installato,...

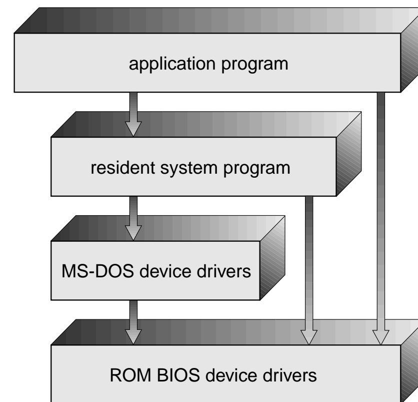
Comunicazioni: creare/cancellare connessioni, spedire/ricevere messaggi,...

Programmi di sistema

- I programmi di sistema forniscono un ambiente per lo sviluppo e l'esecuzione dei programmi. Si dividono in
  - Gestione dei file
  - Modifiche dei file
  - Informazioni sullo stato del sistema e dell'utente
  - Supporto dei linguaggi di programmazione
  - Caricamento ed esecuzione dei programmi
  - Comunicazioni
  - Programmi applicativi
- La maggior parte di ciò che un utente vede di un sistema operativo è definito dai programmi di sistema, non dalle reali chiamate di sistema.

Struttura dei Sistemi Operativi -Approccio semplice

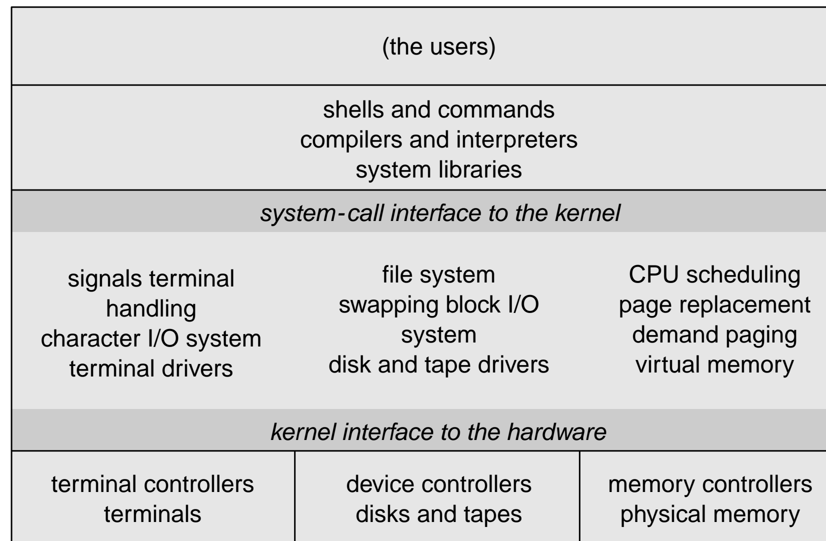
- MS-DOS – pensato per fornire le massime funzionalità nel minore spazio possibile.
  - non è diviso in moduli (è cresciuto oltre il previsto)
  - nonostante ci sia un po' di struttura, le sue interfacce e livelli funzionali non sono ben separati.



Struttura dei Sistemi Operativi - Approccio semplice

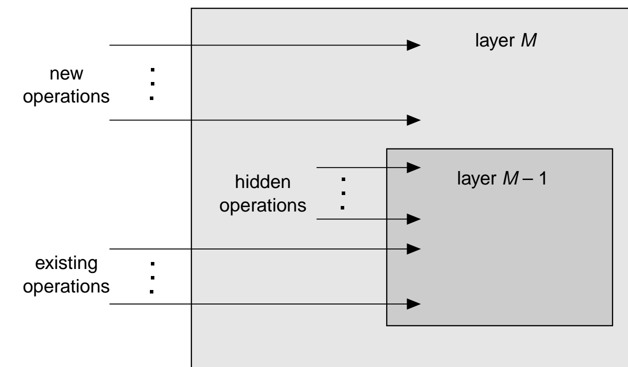
- UNIX – limitato dalle funzionalità hardware, lo UNIX originale aveva una debole strutturazione. Consiste almeno in due parti ben separate:
  - Programmi di sistema
  - Il kernel
    - ∗ consiste in tutto ciò che sta tra le system call e l'hardware
    - ∗ implementa il file system, lo scheduling della CPU, gestione della memoria e altre funzioni del sistema operativo: molte funzionalità in un solo livello.

Struttura dei Sistemi Operativi – Unix originale



Struttura dei sistemi operativi – Approccio stratificato

- Il sistema operativo è diviso in un certo numero di strati (livelli); ogni strato è costruito su quelli inferiori. Lo strato di base (livello 0) è l'hardware; il più alto è l'interfaccia utente.
- Secondo la modularità, gli strati sono pensati in modo tale che ognuno utilizza funzionalità (operazioni) e servizi solamente di strati inferiori.



Struttura dei sistemi operativi – Stratificazione di THE

- La prima stratificazione fu usata nel sistema operativo THE per un calcolatore olandese nel 1969 da Dijkstra e dai suoi studenti.

- THE consisteva dei seguenti sei strati:

layer 5:	user programs
layer 4:	buffering for input and output devices
layer 3:	operator-console device driver
layer 2:	memory management
layer 1:	CPU scheduling
layer 0:	hardware

Stratificazione

- Il sistema MULTICS era organizzato ad annelli concentrici (livelli)

- Per accedere ad un livello piú interno occorreva una chiamata di sistema che attivava una TRAP

- L'organizzazione ad anelli si poteva estendere anche a sottosistemi utente (studente lavora a livello $n + 1$, programma di correzioni lavora a livello n per evitare interferenze)

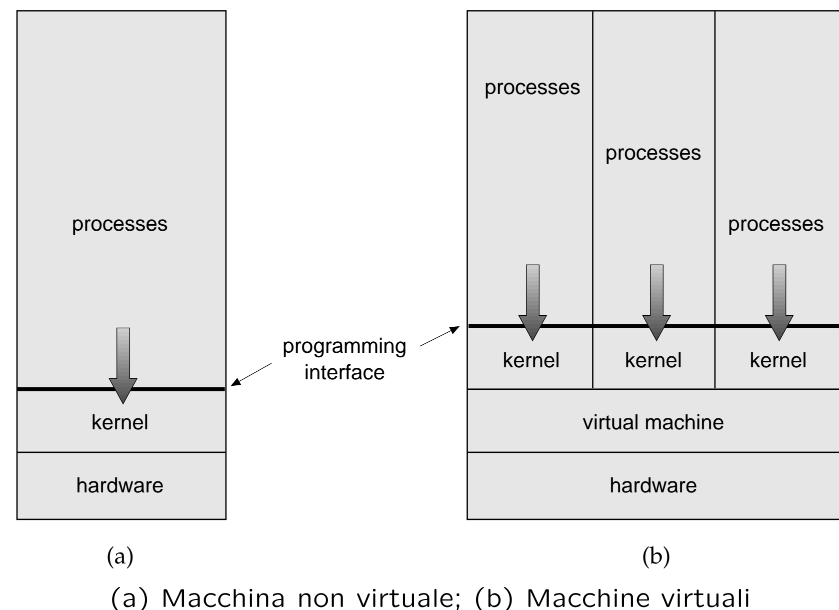
Macchine Virtuali

- Una *macchina virtuale* porta l'approccio stratificato all'estremo: tratta hardware e il sistema operativo come se fosse tutto hardware.

- Una macchina virtuale fornisce una interfaccia *identica* all'hardware nudo e crudo sottostante.

- Il sistema operativo impiega le risorse del calcolatore fisico per creare le macchine virtuali:
  - Lo scheduling della CPU crea l'illusione che ogni processo abbia il suo processore dedicato.
  - La gestione della memoria crea l'illusione di una memoria virtuale per ogni processo
  - Lo spooling può implementare delle stampanti virtuali
  - Spazio disco può essere impiegato per creare "dischi virtuali"

Macchine Virtuali (Cont.)



(a) Macchina non virtuale; (b) Macchine virtuali

Vantaggi/Svantaggi delle Macchine Virtuali

- Il concetto di macchina virtuale fornisce una protezione completa delle risorse di sistema, dal momento che ogni macchina virtuale è isolata dalle altre. Questo isolamento non permette però una condivisione diretta delle risorse.
- Un sistema a macchine virtuali è un mezzo perfetto per l'emulazione di altri sistemi operativi, o lo sviluppo di nuovi sistemi operativi: tutto si svolge sulla macchina virtuale, invece che su quella fisica, quindi non c'è pericolo di far danni.
- Implementare una macchina virtuale è complesso, in quanto si deve fornire un *perfetto* duplicato della macchina sottostante. Può essere necessario dover emulare ogni singola istruzione macchina.
- Approccio seguito in molti sistemi: Windows, Linux, MacOS, JVM,...

Exokernel

- Estensione dell'idea di macchina virtuale
- Ogni macchina virtuale di livello utente vede solo un *sottoinsieme* delle risorse dell'intera macchina
- Ogni macchina virtuale può eseguire il proprio sistema operativo
- Le risorse vengono richieste all'exokernel, che tiene traccia di quali risorse sono usate da chi
- Semplifica l'uso delle risorse allocate: l'exokernel deve solo tenere separati i domini di allocazione delle risorse

Meccanismi e Politiche

- I kernel tradizionali (monolitici) sono poco flessibili
- Distinguere tra *meccanismi* e *politiche*:
  - i meccanismi determinano *come* fare qualcosa;
  - le politiche determinano *cosa* deve essere fatto.

  Ad esempio: assegnare l'esecuzione ad un processo è un meccanismo; scegliere *quale* processo attivare è una politica.
- Questa separazione è un principio molto importante: permette la massima flessibilità, nel caso in cui le politiche debbano essere cambiate.
- Estremizzazione: il kernel fornisce solo i meccanismi, mentre le politiche vengono implementate in user space.

Sistemi con Microkernel

- *Microkernel*: il kernel è ridotto all'osso, fornisce soltanto i meccanismi:
  - Un meccanismo di comunicazione tra processi
  - Una minima gestione della memoria e dei processi
  - Gestione dell'hardware di basso livello (driver)
- Tutto il resto viene gestito da processi in spazio utente: ad esempio, tutte le politiche di gestione del file system, dello scheduling, della memoria sono implementate come processi.
- Meno efficiente del kernel monolitico
- Grande flessibilità; immediata scalabilità in ambiente di rete
- Sistemi operativi recenti sono basati, in diverse misure, su microkernel (AIX4, BeOS, GNU HURD, MacOS X, QNX, Tru64, Windows NT ...)

Il sistema operativo UNIX

- Storia
- Principi di progetto
- Interfaccia per il programmatore
- Interfaccia utente

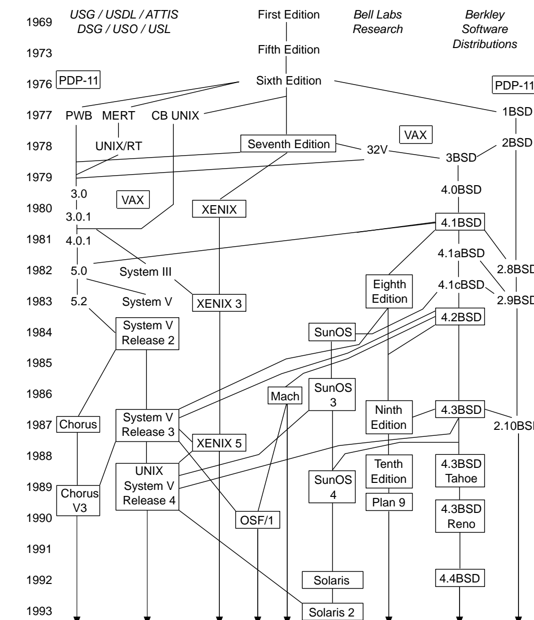
Storia

- Sviluppo originale nel 1969, di Ken Thompson e Dennis Ritchie del Research Group ai Bell Laboratories; incorpora caratteristiche di altri sistemi operativi, specialmente MULTICS.
- La terza versione fu scritta in C, sviluppato ai Bell Labs appositamente per supportare UNIX.
- Il più importante centro di ricerca su UNIX, non dell'AT&T: Università della California a Berkeley (*Berkeley Software Distributions*).
  - 4BSD UNIX risultò da fondi DARPA per lo sviluppo di un UNIX standard per uso governativo.
  - Sviluppato sul VAX, 4.3BSD fu una delle versioni più influenti sullo sviluppo dei seguenti S.O. Fu portata a molte altre piattaforme.

Storia (Cont.)

- Diversi progetti di standardizzazione cercano di consolidare le varianti di UNIX, per raggiungere una interfaccia di programmazione uniforme: ISO ha rilasciato POSIX; X/Open (ora Open Group), ha rilasciato XPG3 e XPG4, e le specifiche UNIX95 e UNIX98.
- Attualmente, la maggior parte degli UNIX commerciali rientra in XPG3 o XPG4. Tutti sono conformi a POSIX.
- Recentemente, c'è stato un ritorno al metodo di sviluppo originario, con il movimento Open Source (GNU/Linux, FreeBSD, OpenBSD, . . .). Open Source risponde alla necessità naturale dei programmatori di riusare il più possibile il codice ed il lavoro già fatto—anche (e specialmente) dagli altri.

Storia (schematica) delle versioni di UNIX



Vantaggi dei primi UNIX

- Scritto in un linguaggio ad alto livello: portabile
- Distribuito in sorgente: modificabile
- Forniva un insieme di primitive potenti su piattaforme poco costose
- Piccolo, modulare, progettazione pulita.

UNIX è facilmente estendibile, senza snaturarlo: nel corso degli anni, aggiunto supporto per

- rete e ambienti distribuiti (TCP/IP, DCE, NFS, CORBA)
- architetture parallele (SMP, NUMA, NORMA)

- interfacce grafiche
- multithreading
- microkernel, . . .

Principi di progetto di UNIX

"Il coltello svizzero del software" (Dennis Ritchie)

- Originalmente sviluppato da programmatori, per programmatori.
- Progettato per essere un sistema time-sharing multiutente.
- Separazione tra interfaccia e kernel; l'interfaccia (shell) può essere facilmente rimpiazzata.
- Il file system è ad albero; il meccanismo di controllo di accesso permette di controllare ogni file.
- Il kernel supporta i file come una sequenza non strutturata di byte.

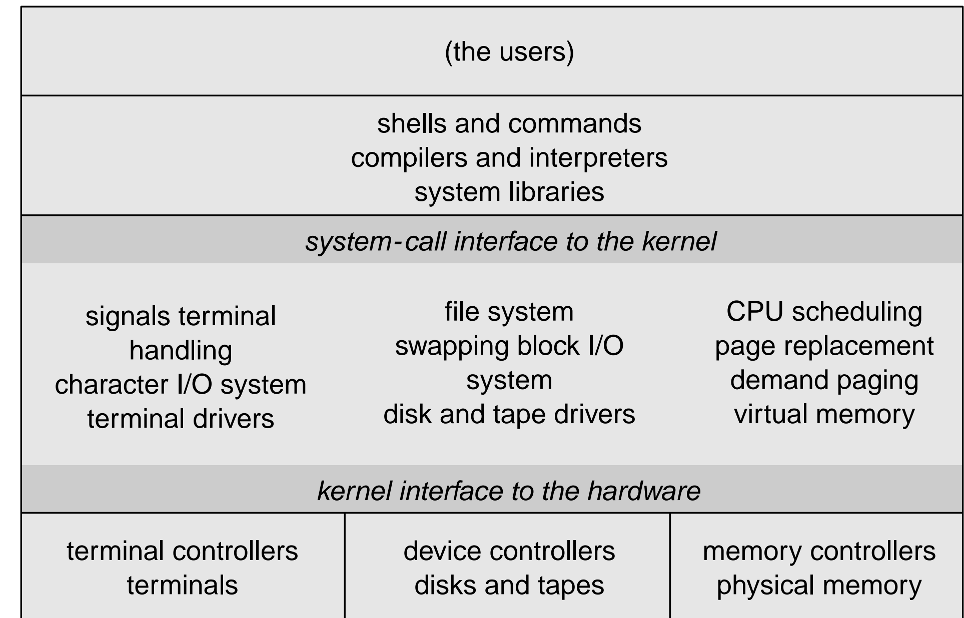
- Supporto per più processi; un processo può facilmente creare nuovi processi.
- L'ambiente è interattivo, e fornisce diverse utilità per lo sviluppo di programmi.

Interfaccia per il programmatore

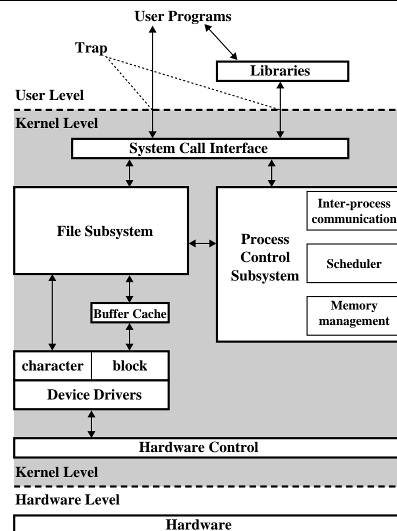
Come la maggior parte dei sistemi, UNIX consiste di due parti separate:

- Kernel: tutto ciò che sta tra l'interfaccia delle system call e l'hardware
  - Implementa il file system, scheduling della CPU, gestione della memoria, protezione e altre funzionalità attraverso le chiamate di sistema
- Programmi di sistema: usano le chiamate di sistema per fornire funzioni di utilità, e.g., compilatori, gestione file, . . .

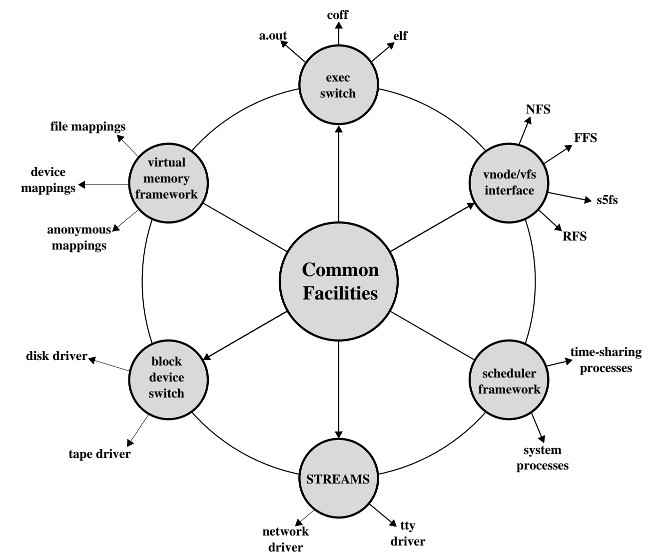
Struttura stratificata di 4.3BSD



Kernel UNIX tradizionale (fino a 4.3BSD)



Kernel UNIX moderno (modulare)



Interfacce al programmatore e all'utente

- Le chiamate di sistema definiscono l'*interfaccia al programmatore (API)* di UNIX. Affiancate da molte librerie standard.
- L'insieme dei programmi di sistema definiscono l'*interfaccia utente*; le stesse funzionalità sono disponibili anche attraverso *interfacce utente grafiche* (GUI).
- All'incirca, tre categorie di chiamate di sistema in UNIX
  - Gestione file
  - Controllo dei processi
  - Gestione delle informazioni

  I dispositivi vengono gestiti come i file (stesse system call).

Come funziona Unix?

- In fase di inizializzazione il processo *init* crea l'interprete dei comandi, chiamato *shell*
- Il processo *shell* legge un comando da tastiera (o dal file indicato dall'utente), lo interpreta, e provvede ad eseguirlo
- La shell accetta dall'utente dei comandi che devono essere scritti utilizzando un linguaggio specifico
- Esistono diverse versioni del programma shell: csh, tcsk, ksh, bash ecc. Volendo, si puo' scrivere la propria shell.
- Comando=programma che gira come utente superuser (con privilegi speciali)

Esempi di comandi

- Consultazione del manuale: man

```
elios> man passwd

PASSWD(1)           Unix Programmer's Manual           PASSWD(1)
NAME
        passwd - change password
SYNOPSIS
        passwd [ name ]

DESCRIPTION
        passwd  will  change  the specified user's password.  Only
        the superuser is allowed to change other user's passwords.
        If the user is not root, then the old password is prompted
        for and verified.

       ...
```


File in Unix

- Come in qualunque sistema operativo, anche in Unix dati e codice sono memorizzati su file (sequenza di byte).
- Ogni file ha un nome seguito eventualmente da un estensione, che viene normalmente utilizzata per indicare il tipo del file.
- I file sono raggruppati in directory.
- Esistono 3 tipi di file:
  - File normali (o flat) che contengono dati o codice,
  - File directory, che contengono altri file,
  - File speciali, dispositivi di I/O, trattati a tutti gli effetti come file.

Directory

- Sono utilizzati per raggruppare i file e sono organizzate ad albero.
- Un file e' individuato dal suo nome specifico, e dal cammino (path name) che bisogna fare nell'albero delle directory per arrivare ad esso.
- Il cammino e' indicato a partire dalla radice (indicata con il solo carattere /) con i vari nomi delle directory attraversate separate fra loro dallo stesso carattere /.
- Il path name specifica quindi la posizione del file nell'albero delle directory (attenzione: Unix distingue fra lettere minuscole e maiuscole).

Esempio di Directory

- Il path-name

  /user/SysOp/Esame

- specifica la posizione del file Esame a partire dalla radice (pathname assoluto).
- Per evitare di usare cammino completo si utilizza il concetto di Working Directory (WD):
- E' possibile posizionarsi ad un certo punto (la WD) dell'albero delle directory (con il comando "cd"), e da quel momento in poi i nomi dei file possono essere dati in modo relativo a quella directory.

- Il comando di shell "pwd" provoca la scrittura della working directory corrente
- Al momento del login viene automaticamente aparta una WD detta "home directory", che per il superuser e' la root /. La home directory e' decisa dall'amministratore del sistema per ogni utente.
- La shell riconosce alcuni caratteri a cui da' un significato preciso. Fra gli altri qui ricordiamo:
  - . indica la directory corrente
  - .. indica la directory genitore
  - ~ indica la home directory
- Con il comando "cd nomedirectory" si puo' cambiare working directory.

Esempio di Directory

```
[giorgio:etabeta:303:/usr] cd bin
[giorgio:etabeta:304:/usr/bin] pwd
/usr/bin
[giorgio:etabeta:305:/usr/bin] cd ..
[giorgio:etabeta:306:/usr] cd java
[giorgio:etabeta:307:/usr/java] pwd
/usr/java
[giorgio:etabeta:312:/usr/java] cd ../local/share
[giorgio:etabeta:313:/usr/local/share] pwd
/usr/local/share
```

Organizzazione del File System

```
/           Directory generale del sistema, detta "root"
/bin        Contiene i comandi piu' importanti per l'utente
/dev        Contiene i file di accesso ai dispositivi fisici del calcolatore
            (dischi, memoria, porte seriali e parallele, ... )
/lib        Contiene le librerie dinamiche necessarie al funzionamento
            dei programmi
/etc        Contiene dei file e le sottodirectory per l'amministrazione
            del sistema
/tmp        Contiene i file temporanei del sistema e degli utenti
/var        Contiene sottodirectory con file che tendono a crescere di
            dimensioni.
/var/spool  Contiene i file di spool temporanei di vari programmi:
            stampa, mail, ...
/var/adm    Contiene i file con messaggi del sistema
/home       Contiene le directory assegnate agli utenti
/sbin       Contiene i programmi di partenza del sistema
/usr        Contiene il grosso del sistema operativo. E' divisa a sua
            volta in sottodirectory
  /usr/bin     Contiene i comandi di base
  /usr/sbin    Contiene i comandi di amministrazione del sistema
  /usr/include Contiene gli header file per la programmazione C e quindi
               per la creazione del kernel
  /usr/man     Contiene i manuali
  /usr/lib     Contiene le librerie per la programmazione e file di
               supporto per molti programmi
```


Contenuto di una directory

```
Il comando ``ls'' lista i nomi dei file della WD

``ls -l'' da' una lista lunga dei file della WD,
          con indicazioni sul tipo e la lunghezza di ogni file

``ls -a'' lista anche i file nascosti, cioe' quelli che cominciano
          con il carattere "."

``ls -R'' elenca ricorsivamente i file della WD e le sue
          sottodirectory
```


Bit di protezione

- Utilizzando "ls -a" compare una lettera prima dei permessi:

```
d     directory
l     link simbolico
c     file speciale a caratteri
b     file speciale a blocchi
```


File speciali

- L'accesso ai dispositivi hardware avviene attraverso i device file.
- Essi sono quindi visibili attraverso le system call per la lettura e scrittura di file.
- Sono elencati nella directory /dev.

```
[giorgio:etabeta:298:~] cd /dev
[giorgio:etabeta:299:/dev] ls -al | more
total 284
drwxr-xr-x   18 root     root        86016 Sep 11 18:30 ./
drwxr-xr-x   20 root     root         4096 Sep 11 18:30 ../
crw-------    1 root     root      10,  10 Apr 11  2002 adbmouse
crw-r--r--    1 root     root      10, 175 Apr 11  2002 agpgart
crw-------    1 root     root      10,   4 Apr 11  2002 amigamouse
crw-------    1 root     root      10,   7 Apr 11  2002 amigamouse1
```

File speciali

- *Block file*: associati a dispositivi organizzati a blocchi ed accessibili in modo diretto (es. dischi)
- *Character file*: associati a dispositivi organizzati come sequenze di caratteri (es. stampanti)
- Tutti i file speciali hanno un
  - Major Device Number: specifica la classe del device (floppy, terminale)
  - Minor Device Number: identifica il numero dell'unita'
- Una tabella Unix associa ad ogni file speciale un codice che identifica il device driver del dispositivo in questione.
- I device con lo stesso Major D.N. condividono il codice dell'unico driver di quel tipo di dispositivo

File speciali

```
hda                Primo disco fisso IDE
hda1, hda2 ..      Partizioni disco fisso IDE
hdb                Secondo disco IDE
ttyS0, ttyS1 ...   Porte seriali input
cua0, cua1, ...    Porte seriali output (modem)
lp0, lp1, ...      Porte parallele
fd0, fd1, ...      Unita' dischetti
fd0H1440           Unita' dischetti formattata 1.44 MB
null               Nullo
```


Protezioni

- Ad ogni file (e ad ogni processo) e' associato
  - un proprietario, individuato dallo uid (user identifier) e il gruppo di appartenenza (gid, group identifier) del proprietario,
  - un insieme di permessi, ognuno rappresentato con un bit, che ne definiscono l'utilizzo.
- I permessi sono di tre tipi: lettura (R), scrittura (W), esecuzione (X).
- Con il permesso di lettura si puo' listare il file, con quello di scrittura si puo' modificarlo, anche azzerarlo, ma non cancellarlo come file.
- Il permesso di esecuzione permette di eseguirlo, purche' sia un binario eseguibile o uno script.

- Per una directory il permesso di lettura consente di listarne il contenuto, in scrittura indica la possibilita' di modificare una directory, infine il permesso di esecuzione indica la possibilita' di attraversarla per accedere a sue sottodirectory.
- I possibili utilizzatori sono di tre tipi: il possessore, il gruppo a cui appartiene, tutti gli altri utenti (other).
- Esempio:

```
R W X     R W X     R W X

owner     group     world
```

Esempio di permessi

- Una directory da' il diritto X e non quello W a other.
- Uno dei file listati nella directory da' il permesso W.
- Allora un qualunque utente puo' modificare, anche azzerare totalmente il file, ma non puo' cancellarne il nome dalla directory.
- Il permesso di scrittura per group (g) e per other (o) e' comunque protetto: si puo' cioe' scrivere sul file, ma non cambiarne gli attributi
- Solo il proprietario di un file puo' modificare gli attributi del file (ad esempio tramite il comando "chmod").
- I bit di protezione si applicano anche ai file speciali.

Comando chmod

- Si possono cambiare i permessi dei file con il comando chmod.

```
krypton> chmod o-w README
```

- elimina la possibilita' di scrittura all'owner.

Link tra file

- E' possibile indicare lo stesso file fisico con piu' di un nome, attraverso un link simbolico
- Questi file sono listati con il carattere l prima dei permessi, inoltre sulla destra compare il nome del file con il simbolo -> seguito dal nome del file a cui e' linkato

  Esempio: `pvm3 -> /home/elios/pvm/pvm3`

- Il comando usato per creare il link fra pvm3 e /home/elios/pvm/pvm3 (la directory esistente) e'

```
ln -s   /home/elios/pvm/pvm3  pvm3
```


- L'opzione -s specifica che si tratta di un link simbolico, cioe' solo fra i nomi dei file.
- I due file (la directory su cui si inserisce il link pvm3, e /home/elios/pvm/pvm3) possono anche essere su supporti fisici diversi

Cancellazione file

- Per la cancellazione di un file si usa il comando "rm nomefile".
- Le directory si cancellano con il comando "rmdir nomedir", solo se sono gia' vuote, oppure con il comando ricorsivo "rm -r nomedir"
- krypton > ls -li README PROVA

```
102  -rw-r--r--    2  giorgio  prof       1267 Apr 22 14:56 PROVA
102  -rw-r--r--    2  giorgio  prof       1267 Apr 22 14:56 PROVA1

krypton > rm PROVA1
krypton > ls -li PROVA*

102  -rw-r--r--    1  giorgio  prof       1267 Apr 22 14:56 PROVA
```


Gestione del file system

- Unix vede i file in modo indipendente dal supporto fisico su cui sono scritti.
- La struttura ad albero delle directory e' presente anche sul supporto su cui i file si trovano.
- Occorre allora agganciare la radice di questa struttura ad una foglia dell'albero che origina da root.
- Supponiamo di voler *montare* il file system contenuto in un floppy disc come sottodirectory di /user/SysOp, di nome AA1999
- Possiamo usare il comando "mount" come segue

```
mount    /dev/fd0   /user/SysOp/AA1999
```


- Il comando mount potrebbe essere disabilitato, cioe' utilizzabile solo dal superuser.
- Al momento dell'inizializzazione del sistema, sono eseguiti comandi di mount che provvedono a montare eventuali supporti necessari.
- Dopo aver montato un supporto, non e' piu' necessario riferirlo per accedere ai file.
- Ad esempio, per accedere al file "PROVA" nel floppy disc montato e' sufficiente il suo nome: /user/SysOp/AA1999/PROVA
- Il comando "umount" consente di *smontare* un FS, ma solo quando non ci sono attivita' ancora da completare che accedono a quei file
- Si deve solo specificare il device, non la directory:

```
krypton > umount /dev/fd0
```

Demoni e processi

- I demoni sono processi, lanciati per lo piu' all'avvio del sistema, che aspettano le richieste dell'utente (stile client-server).
- Ad esempio il demone lpd aspetta le richieste di stampa date con il comando lpr.
- Il comando per avere la lista dei processi e' ps. Senza opzioni vengono listati solo i processi di cui il richiedente e' proprietario, e che sono stati attivati durante la stessa sessione (fra lo stesso login - logout). L'opzione aux consente di avere il listato completo.

Monitoraggio processi: il comando ps(1)

```
disi > ps

PID     TTY   STAT      TIME       COMMAND

16200      p2   s      0:04.92       -tcsh
16330      p2   R      0:01.87       ps

krypton > ps -aux

USER       PID %CPU %MEM   VSZ  RSS TTY      S    STARTED      TIME      COMMAND
root       315  0,0  0,2 1,39M  48K ??       I      gui 19     0:29.26   /usr/sbin/inetd
root       320  0,0  0,0 1,34M   8K ??       I      gui 19     0:05.74   /usr/sbin/cron
root       351  0,0  0,4 1,38M 112K ??       I      gui 19     0:07.55   /usr/lbin/lpd
root       274  0,0  0,4 1,55M 128K ??       I      gui 19     3:22.66   /usr/sbin/snmpd
gianuzzi  5843 20,0  1,7 2,03M 520K ttyp4    S    14.42.18     0:01.48   - (tcsh)
root       376  7,0 10,2 14,1M 3,1M ??       S <    gui 19    04:21:18   /usr/bin/X11/X -nice -2 -auth /usr/var/
gianuzzi  4389  3,0  2,3 9,49M 696K ??       S    09.38.53     0:08.86   /usr/bin/X11/mwm

Dove ( con varie opzioni):

User          proprietario del processo
PID           identificatore del processo
%CPU, %MEM    Percentuali utilizzo CPU e memoria
              nell'ultimo minuto
SIZE          Dimensione del processo
RSS           KB di memoria occupata
TTY           Porta seriale associata al programma
STAT          Stato del processo
              S=sleeping, R=running, I=idle, Z=zombie
START         Orario di avvio del processo
TIME          Tempo utilizzato effettivamente
COMMAND       Comando con cui e' stato lanciato il processo
```


Alcuni Processi

- init e' l'unico processo lanciato dal kernel, essenziale in quanto i l sistema operativo esegue solo funzioni e non comandi.
- I processi della lista il cui nome finisce con d sono demoni, mentre getty e' il programma di gestione di accesso al sistema
- Il comando per interrompere un processo in background e' kill pid dove pid e' il process identifier.
- Ad esempio kill 315 causa, se si hanno i necessari privilegi, la morte del processo 315 (demone gestione rete).

Generalita' sulla Shell

- E' un programma che fa da intermediario fra l'utente e il kernel.
- Legge una linea di caratteri e la interpreta attivando i processi come richiesti dall'utente.
- Esistono diverse shell (sh, csh, tcsh, bash ecc.). La bash (FSF) e' quella normalmente usata in Linux, la shell bash.

Funzionamento della Shell

- La shell accetta comandi per Unix dall'utente e li manda in esecuzione, secondo le direttive date.
- Assegna automaticamente ad ogni programma mandato in esecuzione: lo standard input, cioe' da dove riceve i dati (tastiera), lo standard output, dove deve stampare i risultati (schermo), lo standard error, dove scrive i messaggi d'errore (schermo).
- Un comando normalmente invia l'output su monitor (ad esempio ls). Come fare se invece si vuole la directory listata su file per poterla ad esempio stampare?
- Si usa il linguaggio delle shell: ">", "<" e "2>" seguiti da un nome di file, *ridirigono* l'output, l'input e l'error al file specificato

Ridirezionamento

```
krypton > cat README
........ < listato del file README >

krypton > cat > file2
linea1                --> scritti dall'utente
linea2
^D

krypton > cat README file2 > file1
```

- Con il primo comando si lista su video il contenuto del file README
- Con il secondo si crea il file file2, che conterra' le due righe scritte da tastiera dall'utente
- Con il terzo i file README e file2 sono listati in sequenza sul file file1

Problema

- Vogliamo sapere quanti processi attivi ci sono appartenenti all'utente gianuzzi.
- Il comando "ps -aux" ci consente di conoscere tutti i processi (la lista potrebbe essere lunga).
- Il comando "grep stringa-caratteri lista-file" ci consente di cercare le occorrenze della stringa all'interno dei file indicati.
- Posso usare i comandi

```
krypton > ps aux > proc
krypton > grep gianuzzi proc
  gianuzzi  5843 520K ttyp4 S 14.42.18 0:01.48 - (tcsh)
  gianuzzi  4389 696K ??    S 09.38.53 0:08.86 /usr/bin/X11/mwm
```

- Il file "proc" va tuttavia cancellato. Si può fare in altri modi?

Pipe

- In Unix c'e' la possibilita' di definire un pipe (tubo) di comandi, in cui l'output di un comando e' usato come input del successivo.
```
krypton > ps aux | grep gianuzzi

  gianuzzi  5843 520K ttyp4 S 14.42.18 0:01.48 - (tcsh)
  gianuzzi  4389 696K ??    S 09.38.53 0:08.86 /usr/bin/X11/mwm
```

- Il carattere | (che e' un altro dei caratteri gestiti dalla shell) separa due comandi: l'output del primo e' l'input del secondo.
- Si puo' fare una sequenza di piu' di due comandi.

Espressioni regolari

- Le espressioni regolari sono espressioni costruite su operatori interpretati su insiemi di stringhe.
- Tutte le shell supportano linguaggio per comandi con espressioni regolari
- Ad esempio
  - l'operatore “*” sta per qualsiasi numero (anche zero) di caratteri diversi da “/”
  - “?” puo' essere sostituito da un singolo carattere.
  - la stringa racchiusa fra doppi apici “...” e' presa cosi' com'e'

Interpretazione espressioni regolari

- La shell legge il comando
- Se trova caratteri speciali li trasforma nella lista di nomi di file presenti nel sistema che possono essere costruiti da quell'argomento
- Manda in esecuzione il comando sulla lista di file

Esempio di espressioni regolari

```
krypton > ls
f1.c      f2.c       f3.txt       m.tex

krypton > ls *
f1.c      f2.c       f3.txt       m.tex

krypton > ls f*
f1.c      f2.c       f3.txt

krypton > ls *.c
f1.c      f2.c

krypton > ls f*.?
f1.c      f2.c
```

- Ci sono altre possibilita', ad esempio /usr/[a-f]* indica tutti i file della directory /usr il cui nome inizia con una lettera minuscola fra a e f comprese.

Processi e Thread

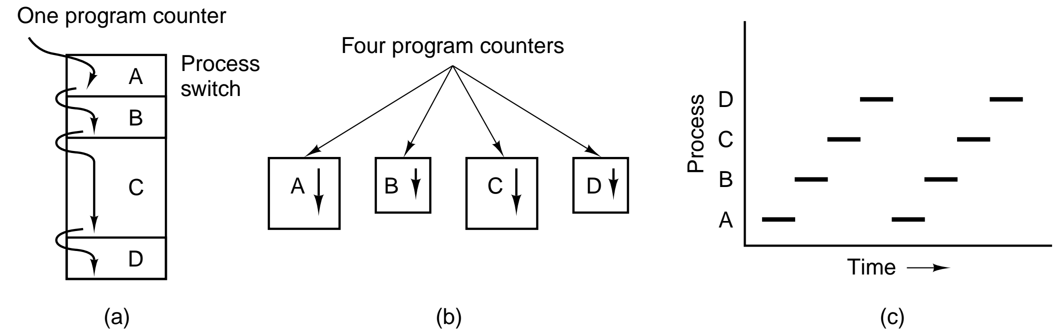
- Concetto di processo
- Operazioni sui processi
- Stati dei processi
- Threads
- Schedulazione dei processi

Il Concetto di Processo

- Un sistema operativo esegue diversi programmi
  - nei sistemi batch – “jobs”
  - nei sistemi time-shared – “programmi utente” o “task”
- I libri usano i termini *job* e *processo* quasi come sinonimi
- Processo: programma in esecuzione. L’esecuzione è sequenziale.
- Un processo comprende anche tutte le risorse di cui necessita, tra cui:
  - programma
  - program counter
  - stack
  - sezione dati
  - dispositivi

Multiprogrammazione

Multiprogrammazione: più processi in memoria, per tenere occupate le CPU.
Time-sharing: le CPU vengono “multiplexate” tra più processi

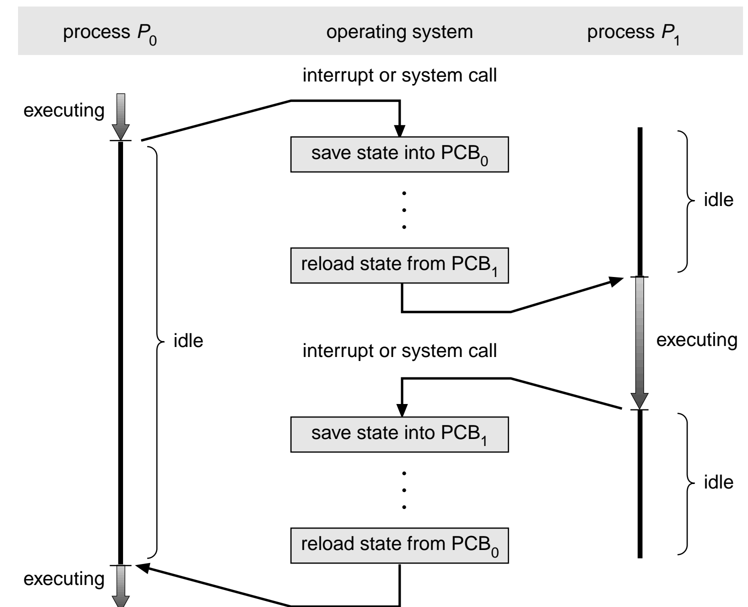


Switch causato da terminazione, prelazione, system-call bloccante.

Switch di Contesto

- Quando la CPU passa ad un altro processo, il sistema deve salvare lo stato del vecchio processo e caricare quello del nuovo processo.
- Lo stato del processo viene salvato nel Process Control Block (PCB)
- Il tempo di *context-switch* porta un certo overhead; il sistema non fa un lavoro utile mentre passa di contesto
- Può essere un collo di bottiglia per sistemi operativi ad alto parallelismo (migliaia - decine di migliaia di thread).
- Il tempo impiegato per lo switch dipende dal supporto hardware

Switch di contesto



Creazione dei processi

- Quando viene creato un processo
  - Al boot del sistema (es. demoni di stampa, di rete, ecc.)
  - Su esecuzione di una system call apposita (es. tramite la system call `fork` di Unix)
  - Su richiesta da parte dell'utente
  - Inizio di un job batch

  La generazione dei processi indica una naturale gerarchia, detta *albero di processi*.

- Esecuzione: Possibili alternative
  - Padre e figli sono in esecuzione concorrente
  - Il padre attende che i figli terminino per riprendere l'esecuzione

- Condivisione delle risorse: Possibili alternative
  - Padre e figli condividono le stesse risorse
  - I figli condividono un sottoinsieme delle risorse del padre
  - Padre e figli non condividono nessuna risorsa
- Spazio indirizzi: Possibili alternative
  - I figli duplicano quello del padre
  - I figli caricano sempre un programma

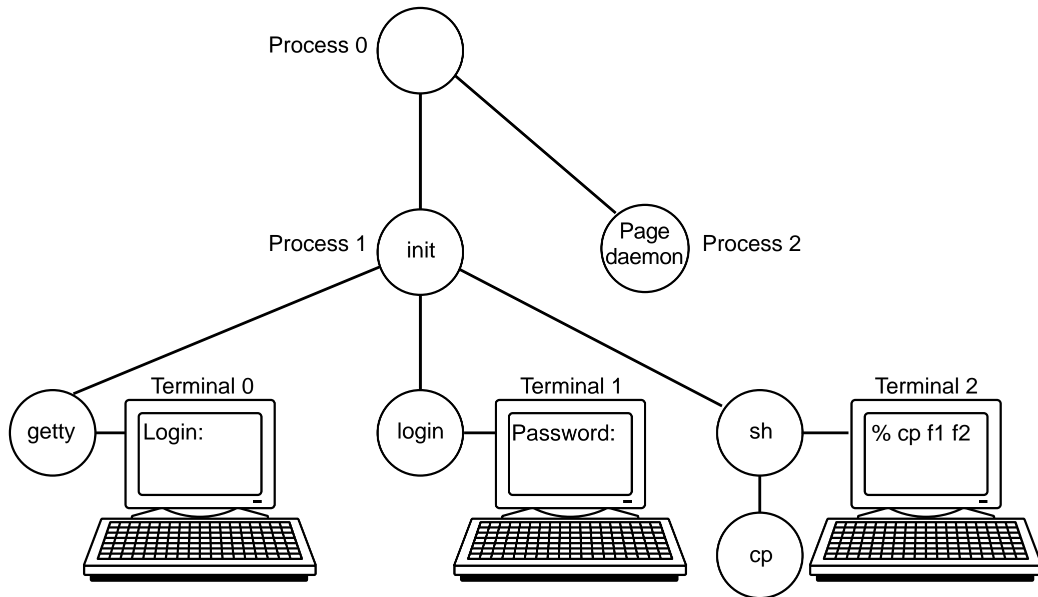
Terminazione dei Processi

- Terminazione volontaria—normale o con errore. I dati di output vengono ricevuti dal processo padre (che li attendeva ad esempio invocando la system call **wait** in Unix).
- Terminazione involontaria: errore fatale (superamento limiti, operazioni illegali, . . .)
- Terminazione da parte di un altro processo (uccisione)
- Terminazione da parte del kernel (es.: il padre termina, e quindi vengono terminati tutti i discendenti: terminazione *a cascata*)

Le risorse del processo sono deallocate dal sistema operativo.

Gerarchia dei processi

- In alcuni sistemi, i processi generati (*figli*) rimangono collegati al processo generatore (*parent*, *genitore*).
- Si formano "famiglie" di processi (*gruppi*)
- Utili per la comunicazione tra processi: e.g. i **segnali** possono essere mandati solo all'interno di un gruppo, o ad un intero gruppo.
- In UNIX: tutti i processi discendono da `init` (processo con identificatore PID=1). Se un processo genitore muore, il figlio viene ereditato da `init`. Un processo non può diseredare il figlio.
- In Windows non c'è gerarchia di processi; il task creatore ha un puntatore al figlio, che comunque può essere passato ad un altro processo.



Stato del processo

Durante l'esecuzione, un processo cambia *stato*.

In *generale* si possono individuare i seguenti stati:

new: il processo è appena creato

running: istruzioni del programma vengono eseguite da una CPU.

waiting: il processo attende qualche evento

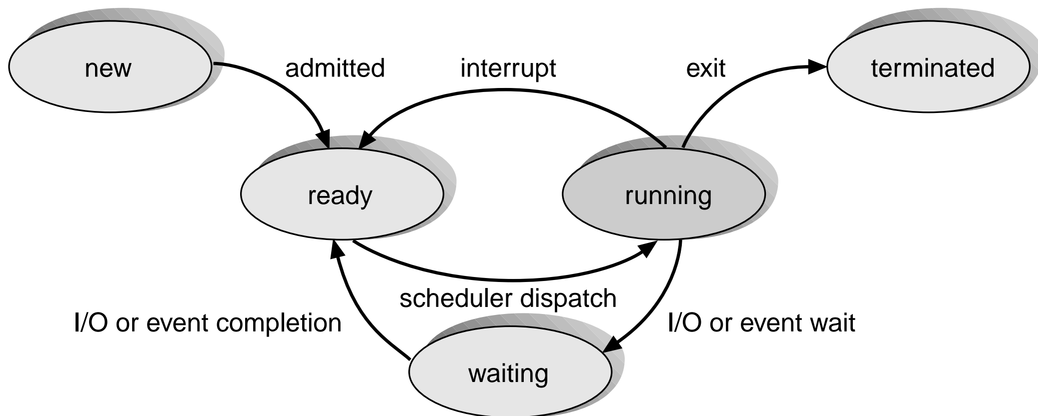
ready: il processo attende di essere assegnato ad un processore

terminated: il processo ha completato la sua esecuzione

Il passaggio da uno stato all'altro avviene in seguito a interruzioni, richieste di risorse non disponibili, selezione da parte dello scheduler, ...

$0 \leq$ n. processi in running $\leq$ n. di processori nel sistema

Diagramma degli stati



Process Control Block (PCB)

Contiene le informazioni associate ad un processo

- Stato del processo
- Dati identificativi (del processo, dell'utente)
- Program counter
- Registri della CPU
- Informazioni per lo scheduling della CPU
- Informazioni per la gestione della memoria
- Informazioni di utilizzo risorse
  - tempo di CPU, memoria, file...
  - eventuali limiti (*quota*)
- Stato dei segnali (per la comunicazione interprocess)

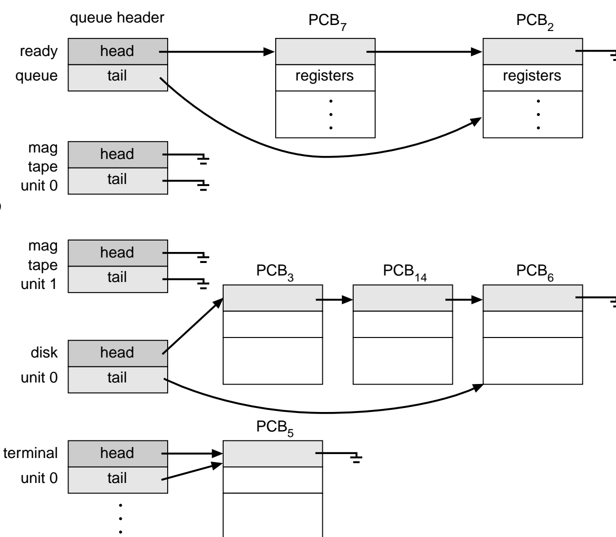
Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Gestione di una interruzione hardware

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

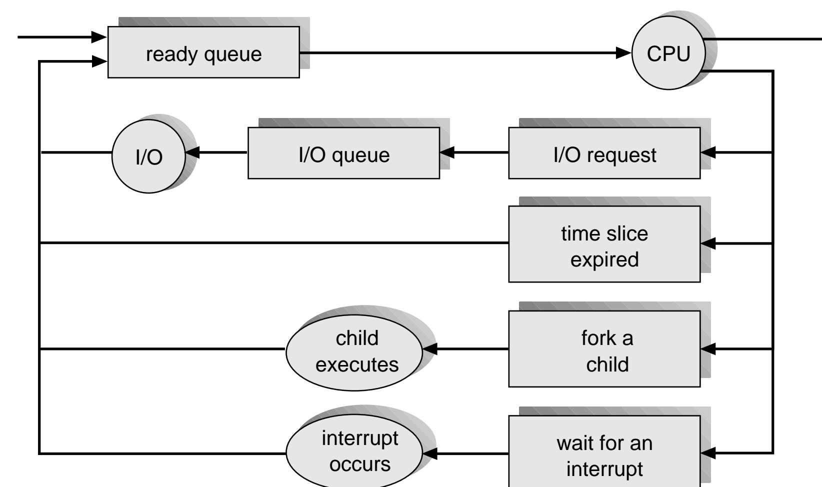
Code di scheduling dei processi

- Coda dei processi (*Job queue*) – insieme di tutti i processi nel sistema
- *Ready queue* – processi residenti in memoria principale, pronti e in attesa di essere messi in esecuzione
- *Code dei dispositivi* – processi in attesa di un dispositivo di I/O.



Migrazione dei processi tra le code

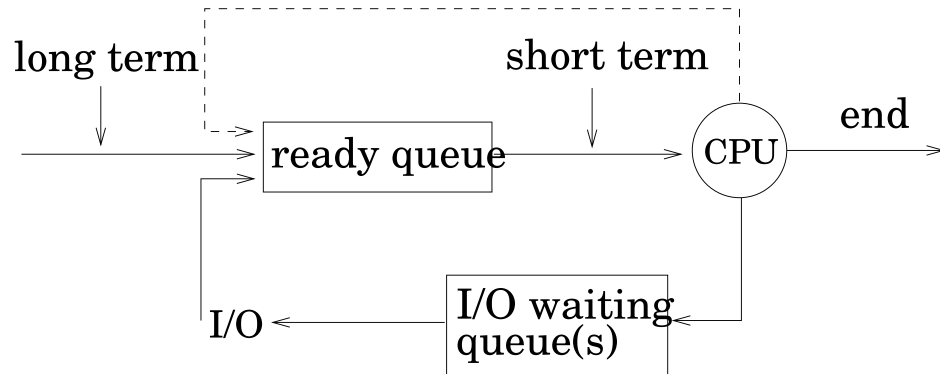
I processi, durante l'esecuzione, migrano da una coda all'altra



Gli *scheduler* scelgono quali processi passano da una coda all'altra.

Gli Scheduler

- Lo *scheduler di lungo termine* (o *job scheduler*) seleziona i processi da portare nella ready queue.
- Lo *scheduler di breve termine* (o *CPU scheduler*) seleziona quali processi ready devono essere eseguiti, e quindi assegna la CPU.

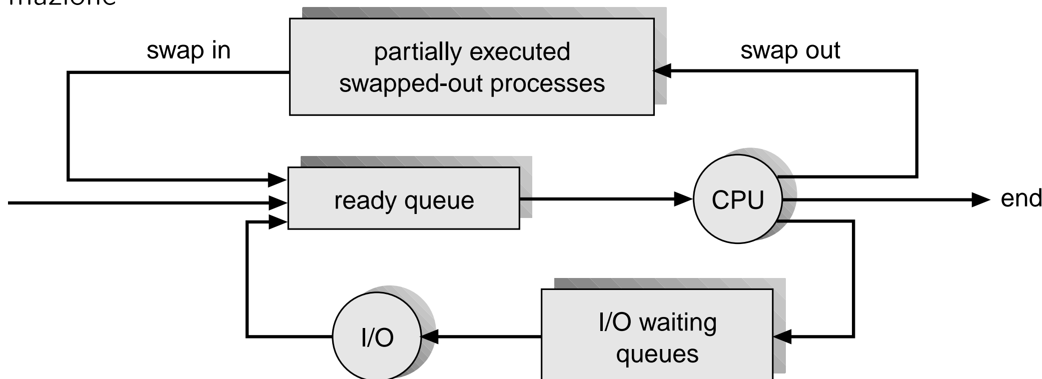


Gli Scheduler (Cont.)

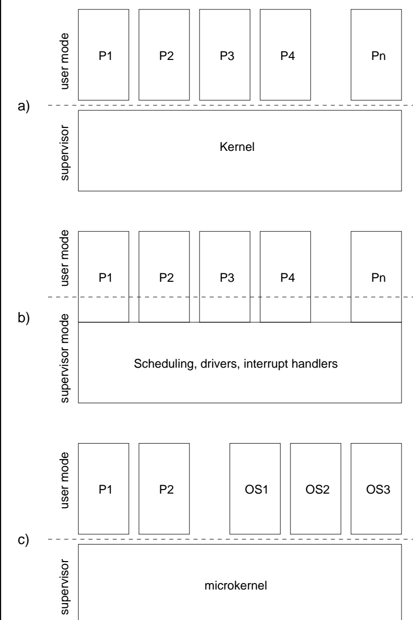
- Lo scheduler di breve termine è invocato molto frequentemente (decine di volte al secondo) ⇒ deve essere veloce
- Lo scheduler di lungo termine è invocato raramente (secondi, minuti) ⇒ può essere lento e sofisticato
- I processi possono essere descritti come
  - I/O-bound: lunghi periodi di I/O, brevi periodi di calcolo.
  - CPU-bound: lunghi periodi di intensiva computazione, pochi (possibilmente lunghi) cicli di I/O.
- Lo scheduler di lungo termine controlla il grado di multiprogrammazione e il *job mix*: un giusto equilibrio tra processi I/O e CPU bound.

Gli Schedulers (Cont.)

Alcuni sistemi hanno anche lo *scheduler di medio termine* (o *swap scheduler*) sospende temporaneamente i processi per abbassare il livello di multiprogrammazione



Modelli di esecuzione dei processi



Esecuzione kernel separata dai processi utente.

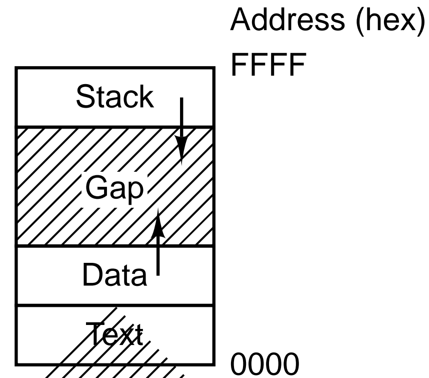
Esecuzione kernel all'interno dei processi

Stretto necessario all'interno del kernel; le decisioni vengono prese da processi di sistema.

Esempio esteso: Processi in UNIX tradizionale

- Un *processo* è un programma in esecuzione + le sue risorse
- Identificato dal *process identifier (PID)*, un numero assegnato dal sistema.
- Ogni processo UNIX ha uno spazio indirizzi separato. Non vede le zone di memoria dedicati agli altri processi.

  Un processo UNIX ha tre segmenti:
  - Stack: Stack di attivazione delle subroutine. Cambia dinamicamente.
  - Data: Contiene lo heap e i dati inizializzati al caricamento del programma. Cambia dinamicamente su richiesta esplicita del programma (es., con la **malloc**).
  - Text: codice eseguibile. Non modificabile, protetto in scrittura.



Alcune chiamate di sistema per gestione dei processi

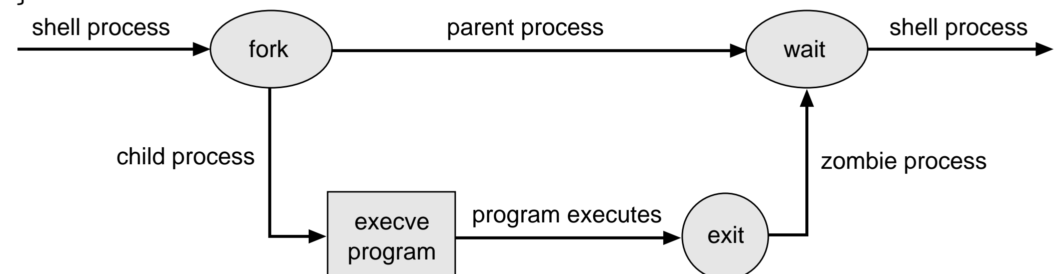
System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Creazione di un processo: la chiamata fork

```
pid = fork();
if (pid < 0) {
   /* fork fallito */
} else if (pid > 0) {
   /* codice eseguito solo dal padre */
} else {
   /* codice eseguito solo dal figlio */
}
/* codice eseguito da entrambi */
```


Esempio: ciclo fork/wait di una shell

```
while (1) {
 read_command(commands, parameters);
 if (fork() != 0) {    /* parent code */
   waitpid(-1, &status, 0);
 } else {              /* child code */
   execve(command, parameters, NULL);
 }
}
```



Gestione e implementazione dei processi in UNIX

- In UNIX, l'utente può creare e manipolare direttamente più processi
- I processi sono rappresentati da *process control block*
  - Il PCB di ogni processo è memorizzato in parte nel kernel (*process structure, text structure*), in parte nello spazio di memoria del processo (*user structure*)
  - L'informazione in questi blocchi di controllo è usata dal kernel per il controllo dei processi e per lo scheduling.

Process Control Blocks

- La struttura base più importante è la *process structure*: contiene
  - stato del processo
  - puntatori alla memoria (segmenti, u-structure, text structure)
  - identificatori del processo
  - identificatori dell'utente
  - informazioni di scheduling (e.g., priorità)
  - segnali non gestiti
- La *text structure*
  - è sempre residente in memoria
  - memorizza quanti processi stanno usando il segmento text (permette quindi condivisioni del codice)
  - contiene dati relativi alla gestione della memoria virtuale per il text

Process Control Block (Cont.)

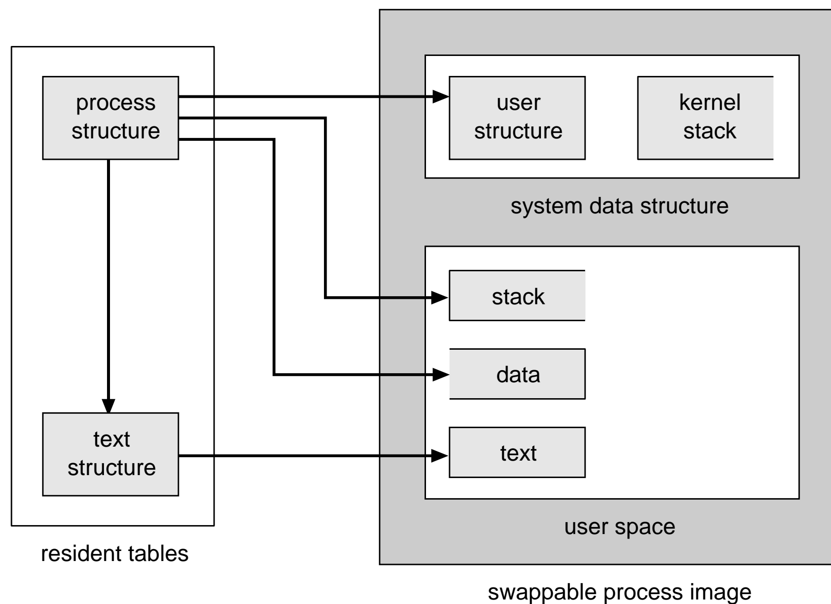
- Le informazioni sul processo che sono richieste solo quando il processo è residente sono mantenute nella *user structure* (o *u structure*).

  Fa parte dello spazio indirizzi modo user, read-only (ma scrivibile dal kernel) e contiene (tra l'altro)
  - identificatore utente e gruppo
  - risultati/errori delle system call
  - tabella dei file aperti
  - limiti del processo

Segmenti dei dati di sistema

- La maggior parte della computazione viene eseguita in user mode; le system call vengono eseguite in modo di sistema (o supervisore).
- Le due fasi di un processo non si sovrappongono mai: un processo si trova sempre in una o l'altra fase.
- Per l'esecuzione in modo kernel, il processo usa uno stack separato (*kernel stack*), invece di quello del modo utente.
- Kernel stack + u structure = *system data segment* del processo

Parti e strutture di un processo



Creazione di un processo

- La **fork** alloca una nuova process structure per il processo figlio
  - nuove tabelle per la gestione della memoria virtuale
  - nuova memoria viene allocata per i segmenti dati e stack
  - i segmenti dati e stack e la user structure vengono copiati ⇒ vengono preservati i file aperti, UID e GID, gestione segnali, etc.
  - il text segment viene condiviso, puntando alla stessa text structure
- La **execve** non crea nessun nuovo processo: semplicemente, i segment dati e stack vengono rimpiazzati

Threads

Dai processi. . .

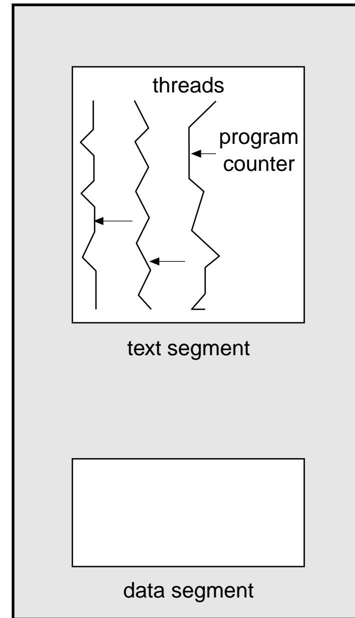
I processi finora studiati incorporano due caratteristiche:

- Unità di allocazione risorse: codice eseguibile, dati allocati staticamente (variabili globali) ed esplicitamente (heap), risorse mantenute dal kernel (file, I/O, workind dir), controlli di accesso . . .
- Unità di esecuzione: un percorso di esecuzione attraverso uno o più programmi: stack di attivazione (variabili locali), stato (running, ready, waiting,. . .), priorità, parametri di scheduling,. . .

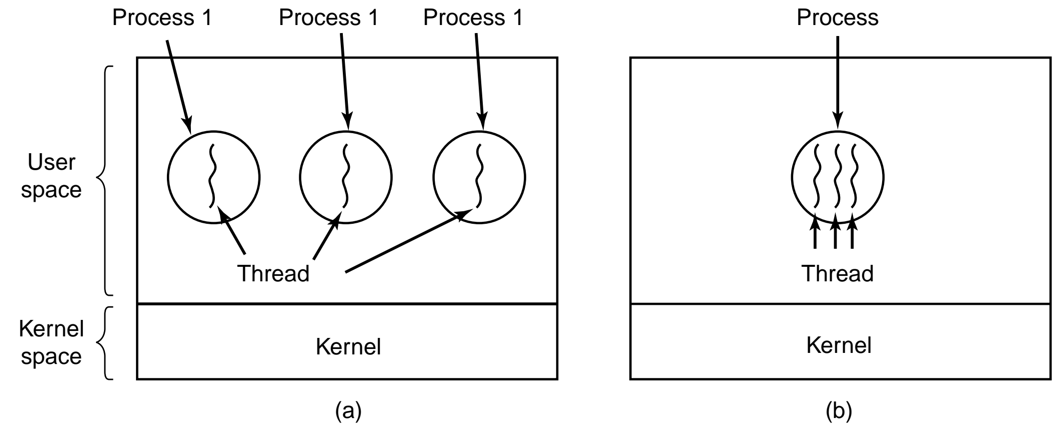
Queste due componenti sono in realtà *indipendenti*

. . . ai thread

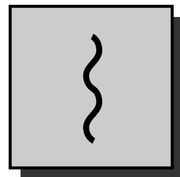
- Un *thread* (o *processo leggero*, *lightweight process*) è una unità di esecuzione:
  - program counter, insieme registri
  - stack del processore
  - stato di esecuzione
- Un thread condivide con i thread suoi pari una unità di allocazione risorse:
  - il codice eseguibile
  - i dati
  - le risorse richieste al sistema operativo
- un *task* = una unità di risorse + i thread che vi accedono



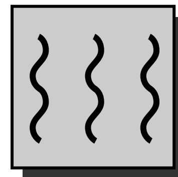
Esempi di thread



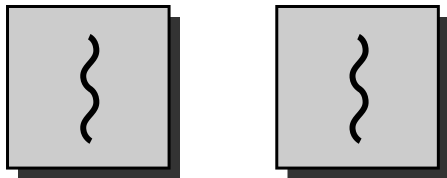
Processi e Thread: quattro possibili scenari



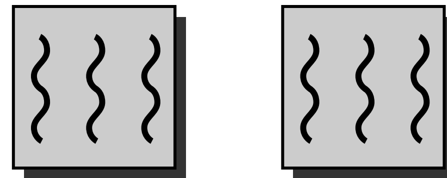
one process
one thread



one process
multiple threads

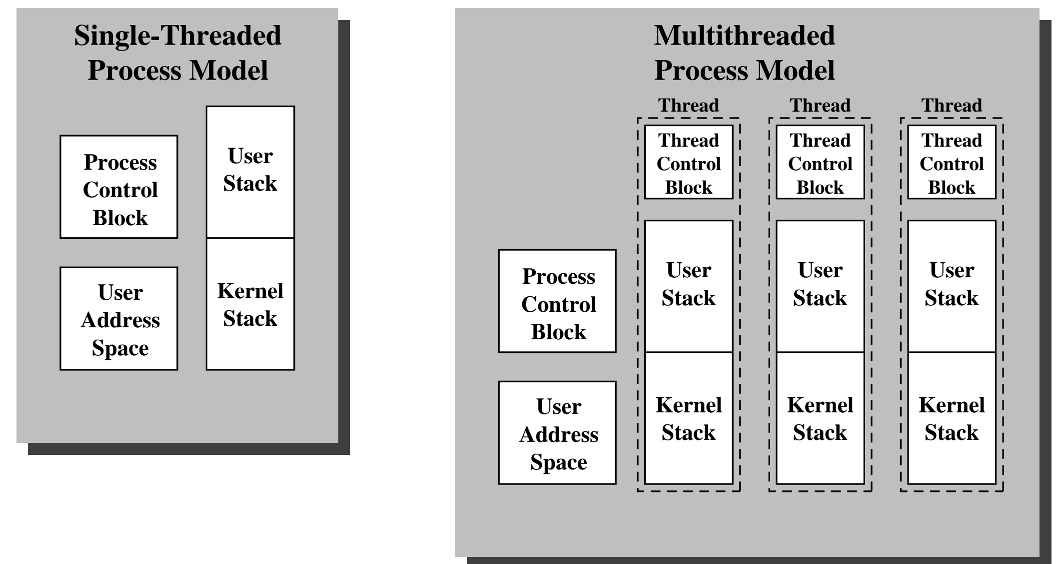


multiple processes
one thread per process



multiple processes
multiple threads per process

Modello multithread dei processi



Risorse condivise e private dei thread

Tutti i thread di un processo accedono alle stesse risorse condivise

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Condivisione di risorse tra i thread

- Vantaggi: maggiore efficienza
  - Creare e cancellare thread è più veloce (100–1000 volte): meno informazione da duplicare/creare/cancellare (e a volte non serve la system call)
  - Lo scheduling tra thread dello stesso processo è molto più veloce che tra processi
  - Cooperazione di più thread nello stesso task porta maggiore throughput e performance
    (es: in un file server multithread, mentre un thread è bloccato in attesa di I/O, un secondo thread può essere in esecuzione e servire un altro client)

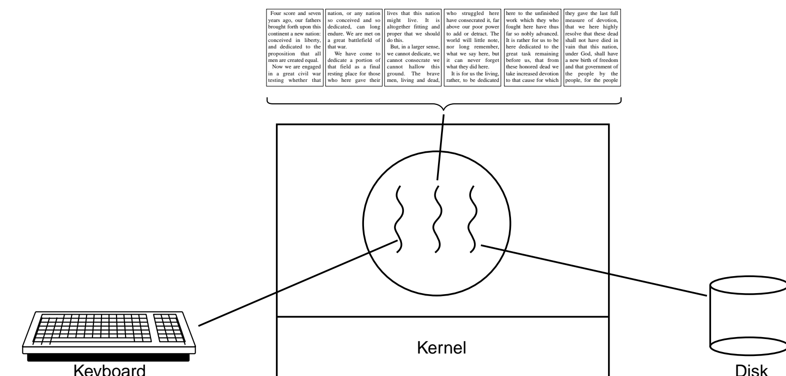
Condivisione di risorse tra thread (Cont.)

- Svantaggi:
  - Maggiore complessità di progettazione e programmazione
    - ∗ i processi devono essere "pensati" paralleli
    - ∗ minore information hiding
    - ∗ sincronizzazione tra i thread
    - ∗ gestione dello scheduling tra i thread può essere demandato all'utente
  - Inadatto per situazioni in cui i dati devono essere protetti
- Ottimi per processi cooperanti che devono condividere strutture dati o comunicare (e.g., produttore–consumatore, server, . . .): la comunicazione non coinvolge il kernel

Esempi di applicazioni multithread

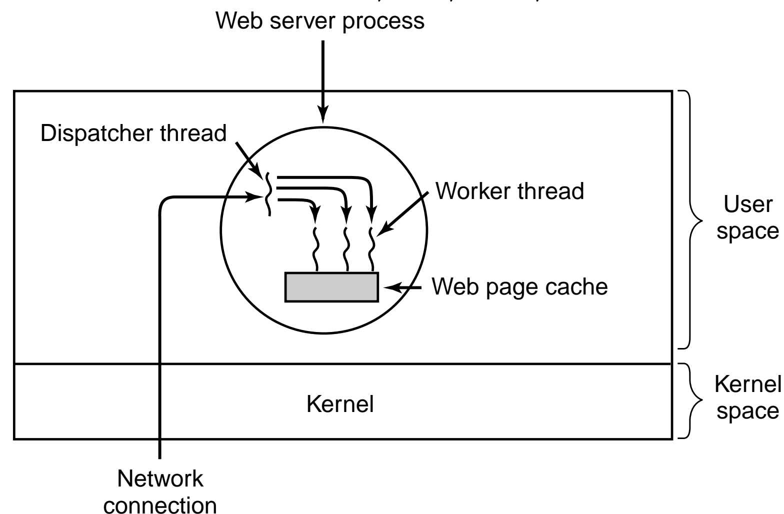
Lavoro foreground/background: mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background. Spreadsheets (ricalcolo automatico), word processor (reimpaginazione, controllo ortografico,. . .)

Elaborazione asincrona: operazioni asincrone possono essere implementate come thread. Es: salvataggio automatico.



Esempi di applicazioni multithread (cont.)

Task intrinsecamente paralleli: vengono implementati ed eseguiti più efficientemente con i thread. Es: file/http/dbms/ftp server, ...



Stati e operazioni sui thread

- Stati: *running*, *ready*, *blocked*. Non ha senso "swapped" o "suspended"
- Operazioni sui thread:

  creazione (spawn): un nuovo thread viene creato all'interno di un processo (`thread_create`), con un proprio punto d'inizio, stack, ...

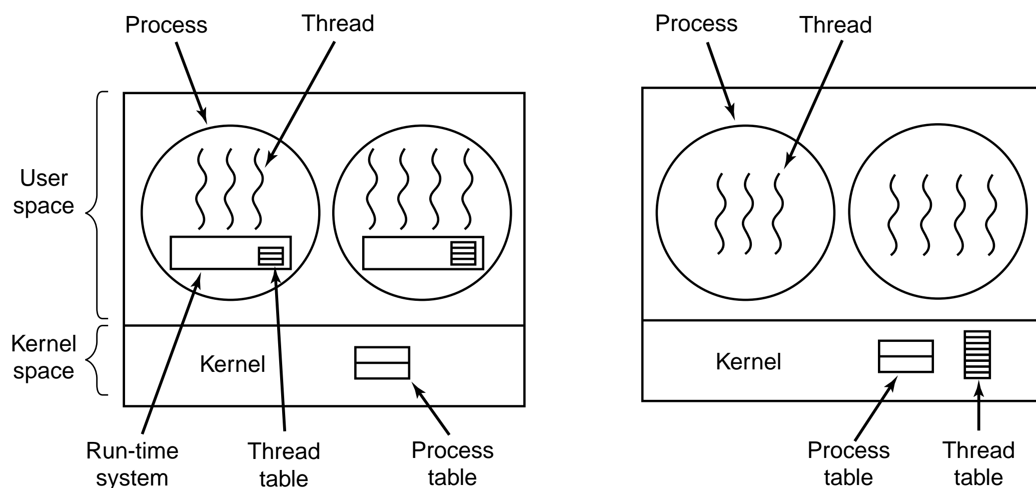
  blocco: un thread si ferma, e l'esecuzione passa ad un altro thread/processo. Può essere volontario (`thread_yield`) o su richiesta di un evento;

  sblocco: quando avviene l'evento, il thread passa dallo stato "blocked" al "ready"

  cancellazione: il thread chiede di essere cancellato (`thread_exit`); il suo stack e le copie dei registri vengono deallocati.

- Meccanismi per la sincronizzazione tra i thread (semafori, `thread_wait`): indispensabili per l'accesso concorrente ai dati in comune

Implementazioni dei thread: Livello utente vs Livello Kernel



User Level Thread

User-level thread (ULT): stack, program counter, e operazioni su thread sono implementati in librerie a livello utente.

Vantaggi:

- efficiente: non c'è il costo della system call
- semplici da implementare su sistemi preesistenti
- portabile: possono soddisfare lo standard POSIX 1003.1c (pthread)
- lo scheduling può essere studiato specificatamente per l'applicazione

User Level Thread (Cont.)

Svantaggi:

- non c'è scheduling automatico tra i thread
  - non c'è prelazione dei thread: se un thread non passa il controllo esplicitamente monopolizza la CPU (all'interno del processo)
  - system call bloccanti bloccano tutti i thread del processo: devono essere sostituite con delle routine di libreria, che blocchino solo il thread se i dati non sono pronti (*jacketing*).
- L'accesso al kernel è sequenziale
- Non sfrutta sistemi multiprocessore
- Poco utile per processi I/O bound, come file server

Esempi: thread CMU, Mac OS ≤ 9, alcune implementazioni dei thread POSIX

Kernel Level Thread

Kernel-level thread (KLT): il kernel gestisce direttamente i thread. Le operazioni sono ottenute attraverso system call. Vantaggi:

- lo scheduling del kernel è per thread, non per processo $\Rightarrow$ un thread che si blocca non blocca l'intero processo
- Utile per i processi I/O bound e sistemi multiprocessor

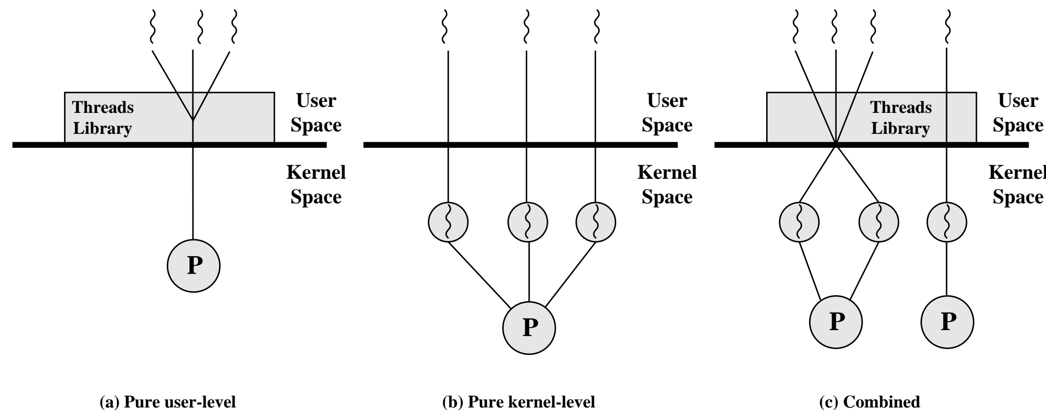
Svantaggi:

- meno efficiente: costo della system call per ogni operazione sui thread
- necessita l'aggiunta e la riscrittura di system call dei kernel preesistenti
- meno portabile
- la politica di scheduling è fissata dal kernel e non può essere modificata

Esempi: molti Unix moderni, OS/2, Mach.

Implementazioni ibride ULT/KLT

Sistemi ibridi: permettono sia thread livello utente che kernel.



Implementazioni ibride (cont.)

Vantaggi:

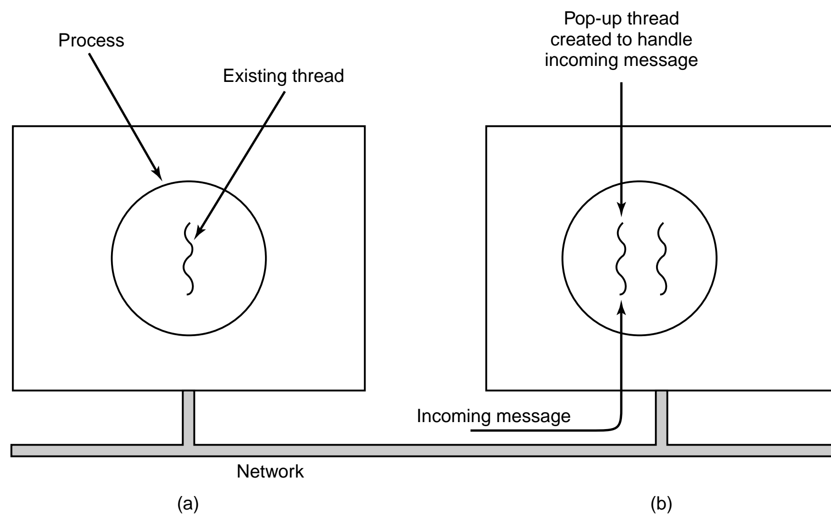
- tutti quelli dei ULT e KLT
- alta flessibilità: il programmatore può scegliere di volta in volta il tipo di thread che meglio si adatta

Svantaggio: portabilità

Es: Solaris 2 (thread/pthread e LWP), Linux (pthread e cloni), Mac OS X, Windows NT, ...

Thread pop-up

- I thread *pop-up* sono thread creati in modo asincrono da eventi esterni.



(a) prima; (b) dopo aver ricevuto un messaggio esterno da gestire

- Molto utili in contesti distribuiti, e per servizio a eventi esterni
- Bassi tempi di latenza (creazione rapida)
- Complicazioni: dove eseguirli?
  - in user space: safe, ma in quale processo? uno nuovo? crearlo costa...
  - in kernel space: veloce, semplice, ma delicato (thread bacati possono fare grossi danni)
- Implementato in Solaris

Processi e Thread di Windows 2000

Nel gergo Windows:

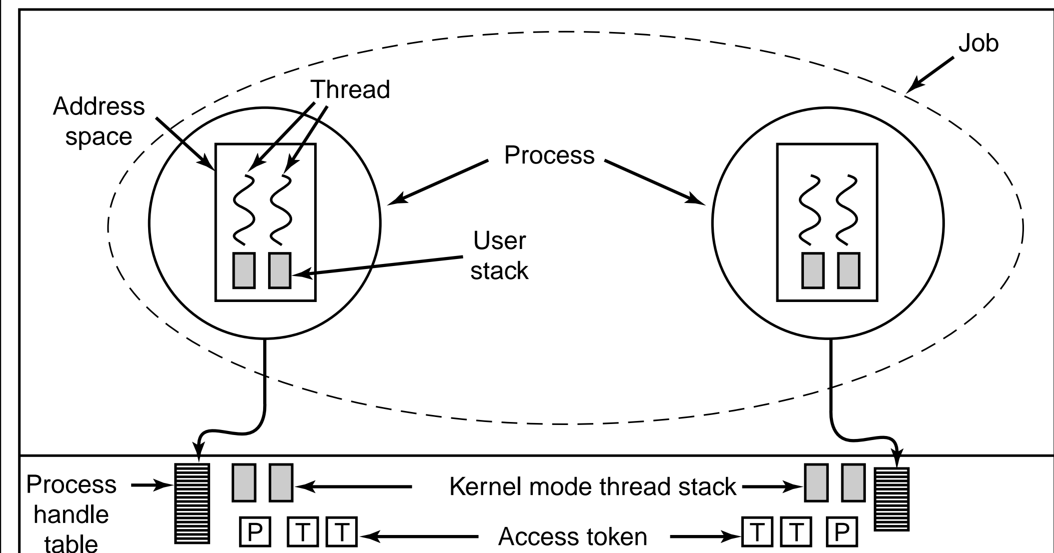
Job: collezione di processi che condividono quota e limiti

Processo: Dominio di allocazione risorse (ID di processo, token di accesso, handle per gli oggetti che usa). Creato con `CreateProcess` con un thread, poi ne può allocare altri.

Thread: entità schedulata dal kernel. Alterna il modo user e modo kernel. Doppio stack. Creato con `CreateThread`.

Fibra (thread leggero): thread a livello utente. Invisibili al kernel.

Job, processi e thread in Windows 2000



Cooperazione tra Processi

- Principi
- Il problema della sezione critica: le *race condition*
- Supporto hardware
- Semafori
- Monitor
- Scambio di messaggi
- Barriere
- Problemi classici di sincronizzazione

Processi (e Thread) Cooperanti

- Processi *indipendenti* non possono modificare o essere modificati dall'esecuzione di un altro processo.
- I processi *cooperanti* possono modificare o essere modificati dall'esecuzione di altri processi.
- Vantaggi della cooperazione tra processi:
  - Condivisione delle informazioni
  - Aumento della computazione (parallelismo)
  - Modularità
  - Praticità implementativa/di utilizzo

IPC: InterProcess Communication

Meccanismi di comunicazione e interazione tra processi (e thread)

Questioni da considerare:

- Come può un processo passare informazioni ad un altro?
- Come evitare accessi inconsistenti a risorse condivise?
- Come sequenzializzare gli accessi alle risorse secondo la causalità?

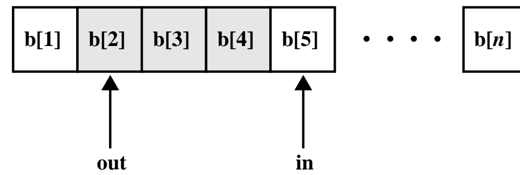
Mantenere la consistenza dei dati richiede dei meccanismi per assicurare l'esecuzione ordinata dei processi cooperanti.

Tipologia di comunicazione

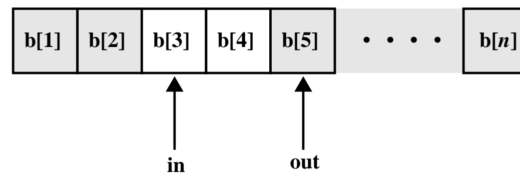
- Scambio di messaggi
  - comunicazione diretta/indiretta (tramite una mailbox)
  - code di messagi con capacità zero/limitata/illimitata
  - comunicazione sincrona/asincrona
- Memoria condivisa (=canale)

Esempio: Problema del produttore-consumatore

- Tipico paradigma dei processi cooperanti: il processo *produttore* produce informazione che viene consumata da un processo *consumatore*
- Soluzione a memoria condivisa: tra i due processi si pone un buffer di comunicazione di dimensione fissata.



(a)



(b)

Produttore-consumatore con buffer limitato

- Dati condivisi tra i processi

type *item* = ... ;
var *buffer*: **array** [0..*n*-1] **of** *item*;
in, *out*: 0..*n*-1;
counter: 0..*n*;
in, *out*, *counter* := 0;

Processo produttore

```
repeat
      ...
    produce un item in nextp
      ...
    while counter = n do no-op;
    buffer[in] := nextp;
    in := in + 1 mod n;
    counter := counter + 1;
until false;
```

Processo consumatore

```
repeat
    while counter = 0 do no-op;
    nextc := buffer[out];
    out := out + 1 mod n;
    counter := counter − 1;
      ...
    consuma l'item in nextc
      ...
until false;
```

- Le istruzioni
  - *counter* := *counter* + 1;
  - *counter* := *counter* − 1;

devono essere eseguite *atomicamente*: se eseguite in parallelo non atomicamente, possono portare ad inconsistenze.

Race conditions

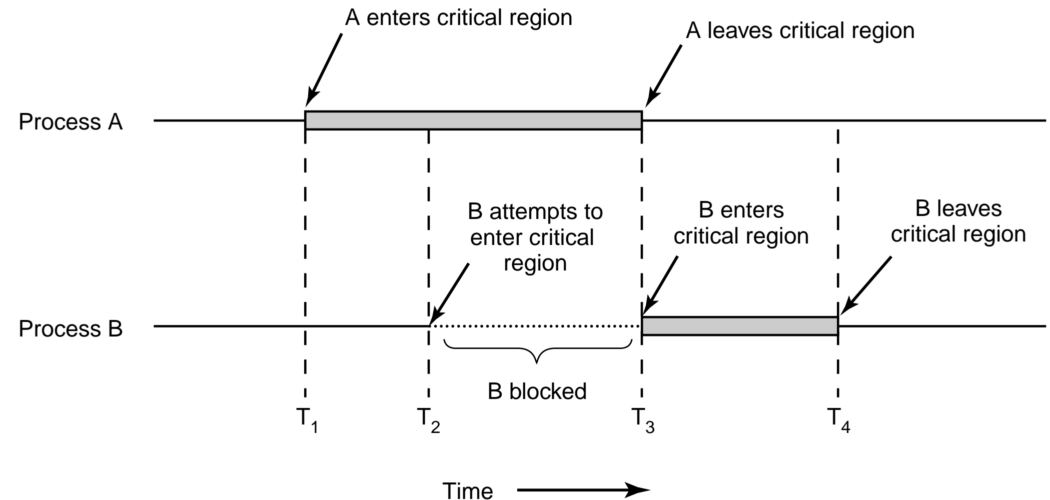
Race condition: più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi.

- Frequenti nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
- Estremamente pericolose: portano al malfunzionamento dei processi cooperanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema
- difficili da individuare e riprodurre: dipendono da informazioni astratte dai processi (decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, . . .)

Problema della Sezione Critica

- n processi che competono per usare dati condivisi
- Ogni processo ha un segmento di codice, detto *sezione critica* in cui si accede ai dati condivisi.
- Problema: assicurare che quando un processo esegue la sua sezione critica, nessun altro processo possa entrare nella propria sezione critica.
- Bisogna proteggere la sezione critica con apposito *codice di controllo*

```
while (TRUE) {
    entry section
        sezione critica
    exit section
        sezione non critica
};
```

Criteri per una Soluzione del Problema della Sezione Critica

1. **Mutua esclusione:** se il processo P_i sta eseguendo la sua sezione critica, allora nessun altro processo può eseguire la propria sezione critica.
2. **Progresso:** se nessun processo è nella sezione critica e esiste un processo che desidera entrare nella propria sezione critica, allora l'esecuzione di tale processo non può essere posposta indefinitamente.
3. **Attesa limitata:** se un processo P ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri processi di accedere alla propria sezione critica prima del processo P deve essere limitato.

- Si suppone che ogni processo venga eseguito ad una velocità non nulla.
- Non si suppone niente sulla velocità *relativa* dei processi (e quindi sul numero e tipo di CPU)

Progresso: Definizioni alternative

- Progresso: varie definizioni

  1. Se nessun processo nella sezione critica ed esiste un processo che desidera entrare nella propria sezione critica, allora l'esecuzione di tale processo non può essere posposta indefinitamente.
  2. Se nessun processo in esecuzione nella sua sezione critica. e qualche processo desidera entrare nella propria sez.crit., solo i processi che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non pu essere rimandata indefinitamente
  3. Un processo al di fuori della sua sezione critica non puo prevenire altri processi dall'entrare la propria; i processi che cercano simultaneamente di accedere alla sezione critica devono decidere quale processo entra.

- Nota: *deadlock* (i processi sono bloccati) → violazione di progresso

Bounded waiting

- Bounded waiting
  1. Se un processo ha gi richiesto l'ingresso nella sua sezione critica, esiste un limite superiore al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche priam che si accordi la richiesta al primo processo
- Nota: *starvation* (un processo non entra mai) → violazione di bounded waiting

Bounded waiting ≠ Progresso

- **Controesempio 1**
  Entry nella sez. critica controllata con una variabile condivisa turn: i processi entrano a turno stretto (P_1, P_2, $\ldots P_k$, P_1,...,P_k,P_1,...,P_k,...) cioè quando P_i esce setta $turn = (i + 1 mod k) + 1$.
  Se ad es. P_i nella sez non critica esegue un loop infinito (tutti i P_j sono bloccati per sempre) allora non vale progresso (deadlock) .
  Vale bounded waiting (i processi aspettano al piu' $k - 1$ turni.
  Se si verifica un deadlock : nessun P_i entra mai in s.c. quindi il bound e' 0).
- **Controesempio 2**
  Scegliamo in modo casuale il processo che entra dall'insieme dei processi che simultaneamente cercano di entrare in s.c.
  Vale progresso (nessun processo al di fuori della reg. critica puo influenzare la scelta di quale processo puo' entrare).
  Non vale bounded waiting (nel caso peggiore P_i non entra mai: starvation).

Soluzioni hardware: controllo degli interrupt

- Il processo può disabilitare TUTTI gli interrupt hw all'ingresso della sezione critica, e riabilitarli all'uscita
  - Soluzione semplice; garantisce la mutua esclusione
  - ma pericolosa: il processo può non riabilitare più gli interrupt, acquisendosi la macchina
  - può allungare di molto i tempi di latenza
  - non scala a macchine multiprocessore (a meno di non bloccare tutte le altre CPU)
- Inadatto come meccanismo di mutua esclusione tra processi utente
- Adatto per brevi(ssimi) segmenti di codice affidabile (es: in kernel, quando si accede a strutture condivise)

Soluzioni software

- Supponiamo che ci siano solo 2 processi, P_0 e P_1
- Struttura del processo P_i (l'altro sia P_j)

```
while (TRUE) {
    entry section
        sezione critica
    exit section
        sezione non critica
}
```

- Supponiamo che i processi possano condividere alcune variabili (dette *di lock*) per sincronizzare le proprie azioni

Tentativo sbagliato

- Variabili condivise
  - **var** *occupato*: (0..1);
    inizialmente *occupato* = 0
  - *occupato* = 0 $\Rightarrow$ un processo può entrare nella propria sezione critica
- Processo P_i

```
while (TRUE) {
    while (occupato ≠ 0) no-op;  ⇓  occupato := 1;
        sezione critica
    occupato := 0;
        sezione non critica
};
```

- Non funziona: lo scheduler può agire dopo il ciclo, nel punto indicato.

Alternanza stretta

- Variabili condivise
  - **var** *turn*: (0..1);
    inizialmente *turn* = 0
  - *turn* = $i \Rightarrow P_i$ può entrare nella propria sezione critica
- Processo P_i

```
while (TRUE) {
    while (turn ≠ i) no-op;
        sezione critica
    turn := j;
        sezione non critica
};
```


Alternanza stretta (cont.)

- Soddisfa il requisito di mutua esclusione, ma non di progresso (richiede l'alternanza stretta) $\Rightarrow$ inadatto per processi con differenze di velocità
- È un esempio di *busy wait*: attesa *attiva* di un evento (es: testare il valore di una variabile).
  - Semplice da implementare
  - Porta a consumi inaccettabili di CPU
  - In genere, da evitare, ma a volte è preferibile (es. in caso di attese molto brevi)
- Un processo che attende attivamente su una variabile esegue uno *spin lock*.

Algoritmo di Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Algoritmo di Peterson (cont)

- Basato su una combinazione di *richiesta* e *accesso*
- Soddisfa tutti i requisiti; risolve il problema della sezione critica per 2 processi
- Si può generalizzare a N processi
- È ancora basato su spinlock

Algoritmo del Fornaio

Risolve la sezione critica per n processi, generalizzando l'idea vista precedentemente.

- Prima di entrare nella sezione critica, ogni processo riceve un numero. Chi ha il numero più basso entra nella sezione critica.
- Se i processi P_i and P_j ricevono lo stesso numero: se $i < j$, allora P_i è servito per primo; altrimenti P_j è servito per primo.
- Lo schema di numerazione genera numeri in ordine crescente

Algoritmo del Fornaio

Process P**i**

```
var choosing: array[1,...,N] of boolean;
var number: array[1,...,N] of integer;

while TRUE do
  choosing[i] = TRUE;
  number[i] = max{number[1],...,number[N]} + 1;
  choosing[i] = FALSE;
  for k : 1 to N do
    while choosing[k] do no-op;
    while (number[k] ≠ 0 and (⟨number[k], k⟩ ≪ ⟨number[i], i⟩)) do no-op;

  - critical section -

  number[i] = 0;
end.
```

Dove $\langle a, b\rangle \ll \langle c, d\rangle$ sse $a < c$ oppure $a = c$ e $b < d$

Istruzioni di Test&Set

- Istruzioni di Test-and-Set-Lock: testano e modificano il contenuto di una parola atomicamente

```
function Test-and-Set (var target: boolean): boolean;
  begin
    Test-and-Set := target;
    target := true;
  end;
```

- Questi due passi devono essere implementati come atomici in assembler. (Es: le istruzioni BTC, BTR, BTS su Intel). Ipoteticamente:

  `TSL RX,LOCK`

  Copia il contenuto della cella `LOCK` nel registro `RX`, e poi imposta la cella `LOCK` ad un valore $\neq 0$. Il tutto atomicamente (viene bloccato il bus di memoria).

Istruzioni di Test&Set (cont.)

```
enter_region:
      TSL REGISTER,LOCK            | copy lock to register and set lock to 1
      CMP REGISTER,#0              | was lock zero?
      JNE enter_region             | if it was non zero, lock was set, so loop
      RET | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0                 | store a 0 in lock
      RET | return to caller
```

- corretto e semplice
- è uno spinlock — quindi busy wait
- Problematico per macchine parallele

Evitare il busy wait

- Le soluzioni basate su spinlock portano a
  - busy wait: alto consumo di CPU
  - inversione di priorità: un processo a bassa priorità che blocca una risorsa viene ostacolato nella sua esecuzione da un processo ad alta priorità in busy wait sulla stessa risorsa.
- Idea migliore: quando un processo deve attendere un evento, che venga posto in *wait*; quando l'evento avviene, che venga posto in *ready*
- Servono specifiche syscall o funzioni di kernel. Esempio:
  - *sleep()*: il processo si autosospende (si mette in *wait*)
  - *wakeup(pid)*: il processo *pid* viene posto in *ready*, se era in *wait*.

  Ci sono molte varianti. Molto comune: con *evento* esplicito.

Produttore-consumatore con sleep e wakeup

```
#define N 100                              /* number of slots in the buffer */
int count = 0;                             /* number of items in the buffer */

void producer(void)
{
      int item;

      while (TRUE) {                       /* repeat forever */
            item = produce_item( );        /* generate next item */
            if (count == N) sleep( );      /* if buffer is full, go to sleep */
            insert_item(item);             /* put item in buffer */
            count = count + 1;             /* increment count of items in buffer */
            if (count == 1) wakeup(consumer);    /* was buffer empty? */
      }
}

void consumer(void)
{
      int item;

      while (TRUE) {                       /* repeat forever */
            if (count == 0) sleep( );      /* if buffer is empty, got to sleep */
            item = remove_item( );         /* take item out of buffer */
            count = count – 1;             /* decrement count of items in buffer */
            if (count == N – 1) wakeup(producer);  /* was buffer full? */
            consume_item(item);            /* print item */
      }
}
```


Produttore-consumatore con sleep e wakeup (cont.)

- Risolve il problema del busy wait
- Non risolve la corsa critica sulla variabile *count*
- I segnali possono andare *perduti*, con conseguenti *deadlock*
- Soluzione: salvare i segnali "in attesa" in un contatore

Semafori

Strumento di sincronizzazione generale (Dijkstra '65)

- Semaforo *S*: variabile intera inizializzata con un valore non-negativo (se S inizializzato a $N > 0$ allora N=numero massimo di processi che possono essere nella sezione critica simultaneamente)
- Vi si può accedere solo attraverso 2 operazioni **atomiche:**
  - *up(S)*: incrementa *S*
  - *down(S)*: attendi finché *S* è maggiore di 0; quindi decrementa *S*
- Normalmente, l'attesa è implementata spostando il processo in stato di *wait*, mentre la *up(S)* mette uno dei processi eventualmente in attesa nello stato di *ready*.
- I nomi originali erano P (*proberen*, testare) e V (*verhogen*, incrementare)

Esempio: Sezione Critica per n processi

- Variabili condivise:
  - **var** *mutex* : *semaphore*
  - inizialmente *mutex* = 1
- Processo P_i

```
while (TRUE) {
    down(mutex);
        sezione critica
    up(mutex);
        sezione non critica
}
```


Esempio: Produttore-Consumatore con semafori

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                     /* TRUE is the constant 1 */
        item = produce_item( );        /* generate something to put in buffer */
        down(&empty);                  /* decrement empty count */
        down(&mutex);                  /* enter critical region */
        insert_item(item);             /* put new item in buffer */
        up(&mutex);                    /* leave critical region */
        up(&full);                     /* increment count of full slots */
    }
}
```



```
void consumer(void)
{
    int item;

    while (TRUE) {                     /* infinite loop */
        down(&full);                   /* decrement full count */
        down(&mutex);                  /* enter critical region */
        item = remove_item( );         /* take item from buffer */
        up(&mutex);                    /* leave critical region */
        up(&empty);                    /* increment count of empty slots */
        consume_item(item);            /* do something with the item */
    }
}
```

Esempio: Sincronizzazione tra due processi

- Variabili condivise:
  - **var** *sync* : *semaphore*
  - inizialmente $sync = 0$
- Processo P_1 Processo P_2

Processo P_1	Processo P_2
⋮	⋮
S_1;	down(sync);
up(sync);	S_2;
⋮	⋮

- S_2 viene eseguito solo dopo S_1.

Implementazione dei semafori a livello Kernel

- La definizione classica usava uno *spinlock* per la *down:* facile implementazione (specialmente su macchine parallele), ma inefficiente
- Alternativa: il processo in attesa viene messo in stato di *wait*
- In generale, un semaforo è un record

```
type semaphore = record
        value: integer;
        L: list of process;
    end;
```

- Assumiamo due operazioni fornite dal sistema operativo:
  - *sleep()*: sospende il processo che la chiama (rilascia la CPU)
  - *wakeup*(*P*): pone in stato di *ready* il processo *P*.

Implementazione dei semafori (Cont.)

- Le operazioni sui semafori sono definite come segue:

```
down/wait(S): S.value := S.value - 1;
              if S.value < 0
                then begin
                        aggiungi questo processo a S.L;
                        sleep();
                     end;
up/signal(S): S.value := S.value + 1;
              if S.value ≤ 0
                then begin
                        togli un processo P da S.L;
                        wakeup(P);
                     end;
```


Implementazione dei semafori (Cont.)

- *value* può avere valori negativi: indica quanti processi sono in attesa su quel semaforo
- le due operazioni *down (wait)* e *up (signal)* devono essere *atomiche* fino a prima della *sleep* e *wakeup:* problema di sezione critica, da risolvere come visto prima:
  - disabilitazione degli interrupt: semplice, ma inadatto a sistemi con molti processori
  - uso di istruzioni speciali (test-and-set)
  - ciclo busy-wait (spinlock): generale, e sufficientemente efficiente (le due sezioni critiche sono molto brevi)

Mutex

- I mutex sono semafori con due soli possibili valori: *bloccato* o *non bloccato*
- Utili per implementare mutua esclusione, sincronizzazione, ...
- due primitive: *mutex_lock* e *mutex_unlock*.
- Semplici da implementare, anche in user space (p.e. per thread). Esempio:

```
mutex_lock:
      TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
      CMP REGISTER,#0             | was mutex zero?
      JZE ok                      | if it was zero, mutex was unlocked, so return
      CALL thread_yield           | mutex is busy; schedule another thread
      JMP mutex_lock              | try again later
ok:   RET | return to caller; critical region entered

mutex_unlock:
      MOVE MUTEX,#0               | store a 0 in mutex
      RET | return to caller
```


Osservazione: Memoria condivisa?

Implementare queste funzioni richiede una qualche memoria condivisa.

- A livello kernel: strutture come quelle usate dai semafori possono essere mantenute nel kernel, e quindi accessibili da tutti i processi (via le apposite system call)
- A livello utente:
  - all'interno dello stesso processo: adatto per i thread
  - tra processi diversi: spesso i S.O. offrono la possibilità di condividere segmenti di memoria tra processi diversi (*shared memory*)
  - alla peggio: file su disco

Deadlock con Semafori

- **Deadlock (stallo):** due o più processi sono in attesa indefinita di eventi che possono essere causati solo dai processi stessi in attesa.
- L'uso dei semafori può portare a deadlock. Esempio: siano S e Q due semafori inizializzati a 1

P_0	P_1
down(S);	*down*(Q);
down(Q);	*down*(S);
⋮	⋮
up(S);	*up*(Q);
up(Q);	*up*(S);

- Programmare con i semafori è molto delicato e prono ad errori, difficilissimi da debuggare. Come in assembler, solo peggio, perché qui gli errori sono race condition e malfunzionamenti non riproducibili.

Monitor

- Un *monitor* è un tipo di dato astratto che fornisce funzionalità di mutua esclusione
  - collezione di dati privati e funzioni/procedure per accedervi.
  - i processi possono chiamare le procedure ma non accedere alle variabili locali.
  - *un solo* processo alla volta può eseguire codice di un monitor
- Il programmatore raccoglie quindi i dati condivisi *e tutte le sezioni critiche relative* in un monitor; questo risolve il problema della mutua esclusione
- Implementati dal compilatore con dei costrutti per mutua esclusione (p.e.: inserisce automaticamente `lock_mutex` e `unlock_mutex` all'inizio e fine di ogni procedura)

```
monitor example
      integer i;
      condition c;

      procedure producer( );
      .
      .
      .
      end;

      procedure consumer( );
      .
      .
      .
      end;
end monitor;
```

Monitor: Controllo del flusso di controllo

Per sospendere e riprendere i processi, ci sono le variabili *condition*, simili agli eventi, con le operazioni

- *wait(c):* il processo che la esegue si blocca sulla condizione c.
- *signal(c):* uno dei processi in attesa su c viene risvegliato.

  A questo punto, chi va in esecuzione nel monitor? Due varianti:

  - chi esegue la *signal(c)* si sospende automaticamente (*monitor di Hoare*)
  - la *signal(c)* deve essere l'ultima istruzione di una procedura (così il processo lascia il monitor) (*monitor di Brinch-Hansen*)
  - i processi risvegliati possono provare ad entrare nel monitor solo dopo che il processo attuale lo ha lasciato

  Il successivo processo ad entrare viene scelto dallo scheduler di sistema
- i *signal* su una condizione senza processi in attesa vengono persi

Produttore-consumatore con monitor

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```


Monitor (cont.)

- I monitor semplificano molto la gestione della mutua esclusione (meno possibilità di errori)
- Veri costrutti, non funzioni di libreria ⇒ bisogna modificare i compilatori.
- Implementati (in certe misure) in veri linguaggi.
  Esempio: i metodi `synchronized` di Java.
  - solo un metodo `synchronized` di una classe può essere eseguito alla volta.
  - Java non ha variabili *condition*, ma ha *wait* and *notify* (+ o − come *sleep* e *wakeup*).
- Un problema che rimane (sia con i monitor che con i semafori): è necessario avere *memoria condivisa* ⇒ questi costrutti non sono applicabili a sistemi distribuiti (reti di calcolatori) senza memoria fisica condivisa.

Scambio di messaggi

- Comunicazione non basato su memoria condivisa con controllo di accesso.
- Basato su due primitive (chiamate di sistema o funzioni di libreria)
  - `send(destinazione, messaggio)`: spedisce un messaggio ad una certa destinazione; solitamente non bloccante.
  - `receive(sorgente, &messaggio)`: riceve un messaggio da una sorgente; solitamente bloccante (fino a che il messaggio non è disponibile).
- Meccanismo più astratto e generale della memoria condivisa e semafori
- Si presta ad una implementazione su macchine distribuite

Problematiche dello scambio di messaggi

- Affidabilità: i canali possono essere inaffidabili (es: reti). Bisogna implementare appositi protocolli fault-tolerant (basati su acknowledgment e timestamping).
- Autenticazione: come autenticare i due partner?
- Sicurezza: i canali utilizzati possono essere intercettati
- Efficienza: se prende luogo sulla stessa macchina, il passaggio di messaggi è sempre più lento della memoria condivisa e semafori.

Produttore-consumatore con scambio di messaggi

- Comunicazione *asincrona*
  - I messaggi spediti ma non ancora consumati vengono automaticamente bufferizzati in una *mailbox* (mantenuto in kernel o dalle librerie)
  - L'oggetto delle `send` e `receive` sono le mailbox
  - La `send` si blocca se la mailbox è piena
  - La `receive` si blocca se la mailbox è vuota.

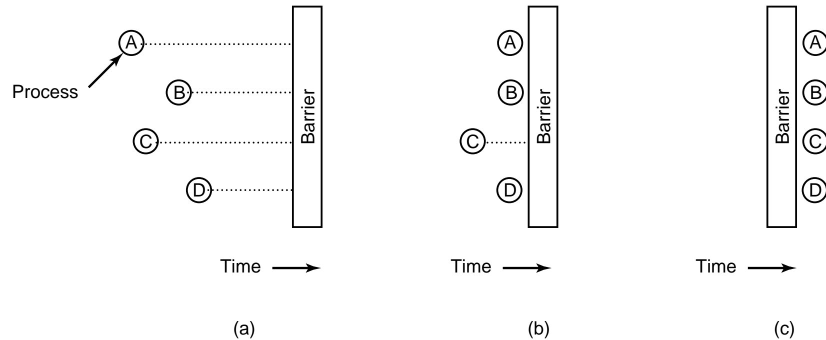

```
#define N 100                               /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                              /* message buffer */

    while (TRUE) {
        item = produce_item( );             /* generate something to put in buffer */
        receive(consumer, &m);              /* wait for an empty to arrive */
        build_message(&m, item);            /* construct a message to send */
        send(consumer, &m);                 /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);              /* get message containing item */
        item = extract_item(&m);            /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```

Prod.-cons. con scambio di messaggi asincrono

- Il consumatore manda N messaggi vuoti (che verrrano allocati in un buffer di sistema)
- Ogni qualvolta il produttore produce un item, prende un messaggio vuoto spedito dal consumatore, lo riempie e lo manda indietro
- Il numero di elementi del "buffer" rimane costante (=N)
- Il produttore si blocca (assumiamo che la receive sia bloccante) se non ci sono messaggi vuoti (il buffer per la comunicazione Prod-Cons è pieno)
- Il consumatore si blocca se non arrivano messaggi dal consumatore (il buffer per la comunicazione è vuoto)

Barriere

- Meccanismo di sincronizzazione per *gruppi* di processi, specialmente per calcolo parallelo a memoria condivisa (es. SMP, NUMA)
  - Ogni processo alla fine della sua computazione, chiama la funzione `barrier` e si sospende.
  - Quando tutti i processi hanno raggiunto la barriera, la superano *tutti assieme* (si sbloccano).



I Grandi Classici

Esempi paradigmatici di programmazione concorrente. Presi come testbed per ogni primitiva di programmazione e comunicazione.

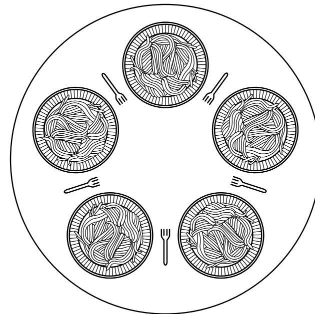
(E buoni esempi didattici!)

- Produttore-Consumatore a buffer limitato (già visto)
- I Filosofi a Cena
- Lettori-Scrittori
- Il Barbiere che Dorme

I Classici: I Filosofi a Cena (Dijkstra, 1965)

n filosofi seduti attorno ad un tavolo rotondo con n piatti di spaghetti e n forchette (bastoncini). (nell'esempio, $n = 5$)

- Mentre pensa, un filosofo non interagisce con nessuno
- Quando gli viene fame, cerca di prendere le forchette più vicine, una alla volta.
- Quando ha due forchette, un filosofo mangia senza fermarsi.
- Terminato il pasto, lascia le forchette e torna a pensare.



Problema: programmare i filosofi in modo da garantire

- assenza di deadlock: non si verificano mai blocchi
- assenza di starvation: un filosofo che vuole mangiare, prima o poi mangia.

I Filosofi a Cena—Una non-soluzione

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                  /* philosopher is thinking */
          take_fork(i);              /* take left fork */
          take_fork((i+1) % N);      /* take right fork; % is modulo operator */
          eat( );                    /* yum-yum, spaghetti */
          put_fork(i);               /* put left fork back on the table */
          put_fork((i+1) % N);       /* put right fork back on the table */
     }
}
```

Possibilità di deadlock: se tutti i processi prendono contemporaneamente la forchetta alla loro sinistra. . .

I Filosofi a Cena—Tentativi di correzione

- Come prima, ma controllare se la forchetta dx è disponibile prima di prelevarla, altrimenti rilasciare la forchetta sx e riprovare daccapo.
  - Non c'è deadlock, ma possibilità di starvation.
- Come sopra, ma introdurre un ritardo casuale prima della ripetizione del tentativo.
  - Non c'è deadlock, la possibilità di starvation viene ridotta ma non azzerata. Applicato in molti protocolli di accesso (CSMA/CD, es. Ethernet). Inadatto in situazione mission-critical o real-time.

I Filosofi a Cena—Soluzioni

- Introdurre un semaforo `mutex` per proteggere la sezione critica (dalla prima `take_fork` all'ultima `put_fork`):
  - Funziona, ma solo un filosofo per volta può mangiare, mentre in teoria $\lfloor n/2 \rfloor$ possono mangiare contemporaneamente.
- Tenere traccia dell'*intenzione* di un filosofo di mangiare. Un filosofo ha tre stati (THINKING, HUNGRY, EATING), mantenuto in un vettore `state`. Un filosofo può entrare nello stato EATING solo se è HUNGRY e i vicini non sono EATING.
  - Funziona, e consente il massimo parallelismo.


```
#define N            5             /* number of philosophers */
#define LEFT         (i+N–1)%N     /* number of i's left neighbor */
#define RIGHT        (i+1)%N       /* number of i's right neighbor */
#define THINKING     0             /* philosopher is thinking */
#define HUNGRY       1             /* philosopher is trying to get forks */
#define EATING       2             /* philosopher is eating */
typedef int semaphore;             /* semaphores are a special kind of int */
int state[N];                      /* array to keep track of everyone's state */
semaphore mutex = 1;               /* mutual exclusion for critical regions */
semaphore s[N] = {0,...,0};        /* one semaphore per philosopher */

void philosopher(int i)            /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                 /* repeat forever */
        think( );                  /* philosopher is thinking */
        take_forks(i);             /* acquire two forks or block */
        eat( );                    /* eat your   spaghetti */
        put_forks(i);              /* put both forks back on table */
    }
}
```

```
void take_forks(int i)             /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                  /* enter critical region */
    state[i] = HUNGRY;             /* record fact that philosopher i is hungry */
    test(i);                       /* try to acquire 2 forks */
    up(&mutex);                    /* exit critical region */
    down(&s[i]);                   /* block if forks were not acquired */
}
void put_forks(i)                  /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                  /* enter critical region */
    state[i] = THINKING;           /* philosopher has finished eating */
    test(LEFT);                    /* see if left neighbor can now eat */
    test(RIGHT);                   /* see if right neighbor can now eat */
    up(&mutex);                    /* exit critical region */
}

void test(i)                       /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

I Classici: Lettori-Scrittori

Un insieme di dati (es. un file, un database, dei record), deve essere condiviso da processi *lettori* e *scrittori*

- Due o più lettori possono accedere contemporaneamente ai dati
- Ogni scrittore deve accedere ai dati in modo esclusivo.

Implementazione con i semafori:

- Tenere conto dei lettori in una variabile condivisa, e fino a che ci sono lettori, gli scrittori non possono accedere.
- Dà maggiore priorità ai lettori che agli scrittori.


```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                  /* repeat forever */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc + 1;                /* one reader more now */
        if (rc == 1) down(&db);     /* if this is the first reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        read_data_base( );          /* access the data */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc – 1;                /* one reader fewer now */
        if (rc == 0) up(&db);       /* if this is the last reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        use_data_read( );           /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                  /* repeat forever */
        think_up_data( );           /* noncritical region */
        down(&db);                  /* get exclusive access */
        write_data_base( );         /* update the data */
        up(&db);                    /* release exclusive access */
    }
```

I Classici: Il Barbiere che Dorme

In un negozio c'è un solo barbiere, una sedia da barbiere e n sedie per l'attesa.

- Quando non ci sono clienti, il barbiere dorme sulla sedia.
- Quando arriva un cliente, questo sveglia il barbiere se sta dormendo.
- Se la sedia è libera e ci sono clienti, il barbiere fa sedere un cliente e lo serve.
- Se un cliente arriva e il barbiere sta già servendo un cliente, si siede su una sedia di attesa se ce ne sono di libere, altrimenti se ne va.

Problema: programmare il barbiere e i clienti filosofi in modo da garantire assenza di deadlock e di starvation.

Il Barbiere—Soluzione

- Tre semafori:
  - `customers`: i clienti in attesa (contati anche da una variabile `waiting`)
  - `barbers`: conta i barbieri in attesa (0 o 1)
  - `mutex`: per mutua esclusione
- Il barbiere esegue una procedura che lo blocca se non ci sono clienti; quando si sveglia, serve un cliente e ripete.
- Ogni cliente prima di entrare nel negozio controlla se ci sono sedie libere; altrimenti se ne va.
- Un cliente, quando entra nel negozio, sveglia il barbiere se sta dormendo.

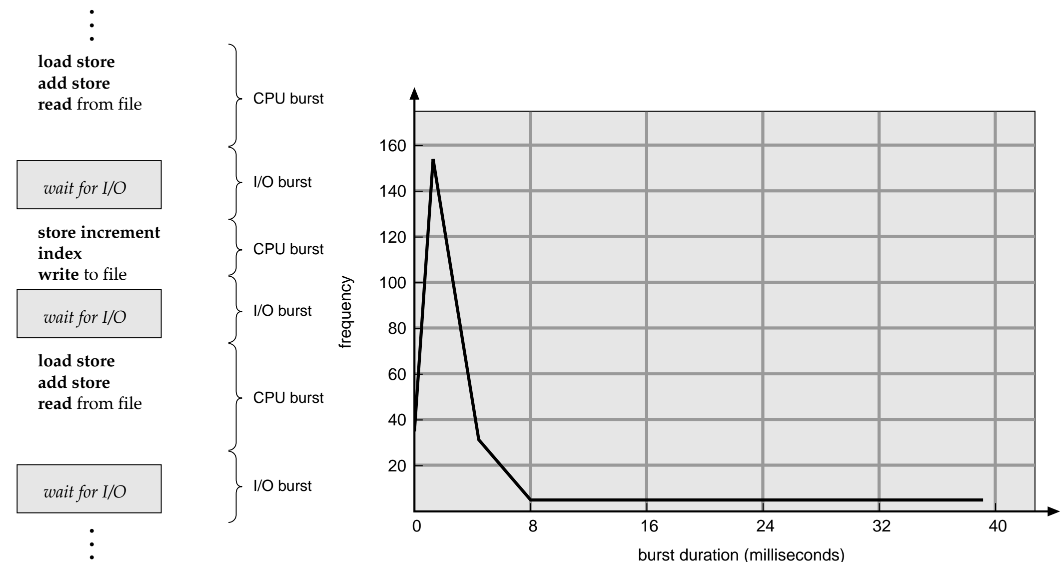
```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}
```

```
void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Scheduling della CPU

- Concetti base
  - Massimizzazione dell'uso della CPU attraverso multiprogrammazione
  - Ciclo Burst CPU–I/O: l'esecuzione del processo consiste in un ciclo di periodi di esecuzione di CPU e di attesa di I/O.
- Criteri di Scheduling
- Algoritmi di Scheduling

I/O e CPU burst



Scheduler a breve termine

- Seleziona tra i processi in memoria e pronti per l'esecuzione, quello a cui allocare la CPU.
- La decisione dello scheduling può avere luogo quando un processo
  1. passa da running a waiting
  2. passa da running a ready
  3. passa da waiting a ready
  4. termina.
- Scheduling nei casi 1 e 4 è *nonpreemptive* (senza prelazione)
- Gli altri scheduling sono *preemptive*.
- L'uso della prelazione ha effetti sulla progettazione del kernel (accesso condiviso alle stesse strutture dati)

Dispatcher

- Il *dispatcher* è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler di breve termine. Questo comporta
  - switch di contesto
  - passaggio della CPU da modo supervisore a modo user
  - salto alla locazione del programma utente per riprendere il processo
- È essenziale che sia veloce
- La *latenza di dispatch* è il tempo necessario per fermare un processo e riprenderne un altro

Criteri di Valutazione dello Scheduling

- *Utilizzo della CPU:* mantenere la CPU più carica possibile.
- *Throughput (produttività):* # di processi completati nell'unità di tempo
- *Tempo di turnaround (completamento):* tempo totale impiegato per l'esecuzione di un processo
- *Tempo di attesa:* quanto tempo un processo ha atteso in ready
- *Tempo di risposta:* quanto tempo si impiega da quando una richiesta viene inviata a quando si ottiene la prima risposta (**not** l'output — è pensato per sistemi time-sharing).
- *Varianza del tempo di risposta:* quanto il tempo di risposta è variabile

Mememto: Se media $\mu = \frac{1}{n}\Sigma_{i=1}^{n} v_i$ allora varianza $\sigma^2 = \frac{1}{n}\Sigma_{i=1}^{n}(v_i - \mu)^2$

Obiettivi generali di un algoritmo di scheduling

All systems
Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems
Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems
Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

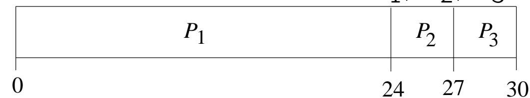
Nota: in generale, non esiste soluzione ottima sotto tutti gli aspetti

Scheduling First-Come, First-Served (FCFS)

- Senza prelazione — inadatto per time-sharing
- Equo: non c'è pericolo di starvation.
- Esempio:

Processo	Burst Time
P_1	24
P_2	3
P_3	3

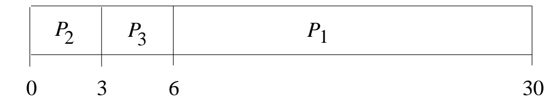
Diagramma di Gantt con l'ordine di arrivo P_1, P_2, P_3



- Tempi di attesa: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Tempo di attesa medio: $(0 + 24 + 27)/3 = 17$

Scheduling FCFS (Cont.)

- Supponiamo che i processi arrivino invece nell'ordine P_2, P_3, P_1. Diagramma di Gantt:



- Tempi di attesa: $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Tempo di attesa medio: $(6 + 0 + 3)/3 = 3$
- molto meglio del caso precedente
- *Effetto convoglio*: i processi I/O-bound si accodano dietro un processo CPU-bound.

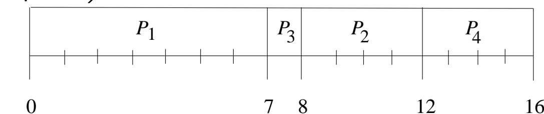
Scheduling Shortest-Job-First (SJF)

- Si associa ad ogni processo la lunghezza del suo prossimo burst di CPU. I processi vengono ordinati e schedulati per tempi crescenti.
- Due schemi possibili:
  - nonpreemptive: quando la CPU viene assegnata ad un processo, questo la mantiene finché non termina il suo burst.
  - preemptive: se nella ready queue arriva un nuovo processo il cui prossimo burst è minore del tempo rimanente per il processo attualmente in esecuzione, quest'ultimo viene prelazionato. (Scheduling *Shortest-Remaining-Time-First*, SRTF).
- SJF è ottimale: fornisce il minimo tempo di attesa per un dato insieme di processi.
- Si rischia la *starvation*

Esempio di SJF Non-Preemptive

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

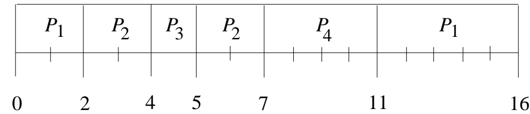


Tempo di attesa medio = $(0 + 6 + 3 + 7)/4 = 4$

Esempio di SJF Preemptive

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SRTF (preemptive)



Tempo di attesa medio = $(9 + 1 + 0 + 2)/4 = 3$

Come determinare la lunghezza del prossimo ciclo di burst?

- Si può solo dare una *stima*
- Nei sistemi batch, il tempo viene stimato dagli utenti
- Nei sistemi time sharing, possono essere usati i valori dei burst precedenti, con una media pesata esponenziale

  1. t_n = tempo dell'n-esimo burst di CPU
  2. τ_{n+1} = valore previsto per il prossimo burst di CPU
  3. α parametro, $0 \leq \alpha \leq 1$
  4. Calcolo:

$$\tau_{n+1} := \alpha t_n + (1 - \alpha)\tau_n$$

Esempi di media esponenziale

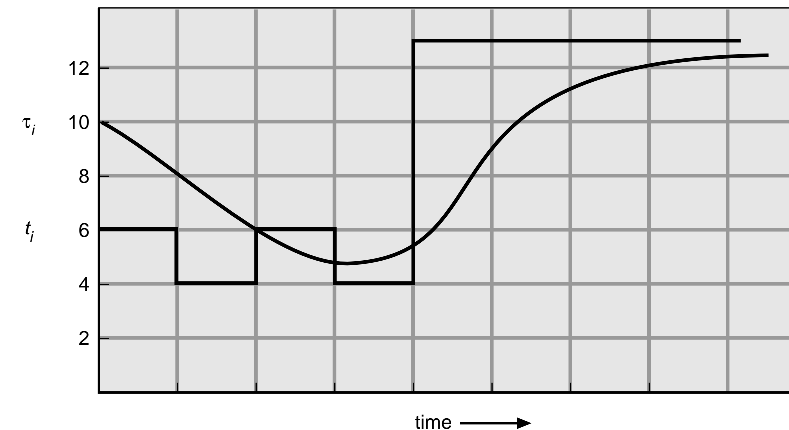
- Espandendo la formula:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0$$

- Se $\alpha = 0$: $\tau_{n+1} = \tau_0$
  - la storia recente non conta
- Se $\alpha = 1$: $\tau_{n+1} = t_n$
  - Solo l'ultimo burst conta
- Valore tipico per α: 0.5; in tal caso la formula diventa

$$\tau_{n+1} = \frac{t_n + \tau_n}{2}$$

Predizione con media esponenziale



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Scheduling a priorità

- Un numero (intero) di priorità è associato ad ogni processo
- La CPU viene allocata al processo con la priorità più alta (intero più piccolo ≡ priorità più grande)
- Le priorità possono essere definite
  - internamente: in base a parametri misurati dal sistema sul processo (tempo di CPU impiegato, file aperti, memoria, interattività, uso di I/O...)
  - esternamente: importanza del processo, dell'utente proprietario, dei soldi pagati, ...
- Gli scheduling con priorità possono essere preemptive o nonpreemptive
- SJF è uno scheduling a priorità, dove la priorità è il prossimo burst di CPU previsto

Scheduling con priorità (cont.)

- Problema: *starvation* – i processi a bassa priorità possono venire bloccati da un flusso continuo di processi a priorità maggiore
  - vengono eseguiti quando la macchina è molto scarica
  - oppure possono non venire mai eseguiti
- Soluzione: invecchiamento (*aging*) – con il passare del tempo, i processi non eseguiti aumentano la loro priorità

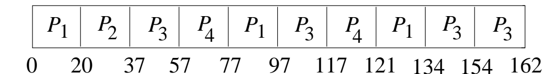
Round Robin (RR)

- Algoritmo con prelazione specifico dei sistemi time-sharing: simile a FCFS ma con prelazione quantizzata.
- Ogni processo riceve una piccola unità di tempo di CPU — il *quanto* — tipicamente 10-100 millisecondi. Dopo questo periodo, il processo viene prelazionato e rimesso nella coda di ready.
- Se ci sono n processi in ready, e il quanto è q, allora ogni processo riceve $1/n$ del tempo di CPU in periodi di durata massima q. Nessun processo attende più di $(n-1)q$

Esempio: RR con quanto = 20

Processo	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

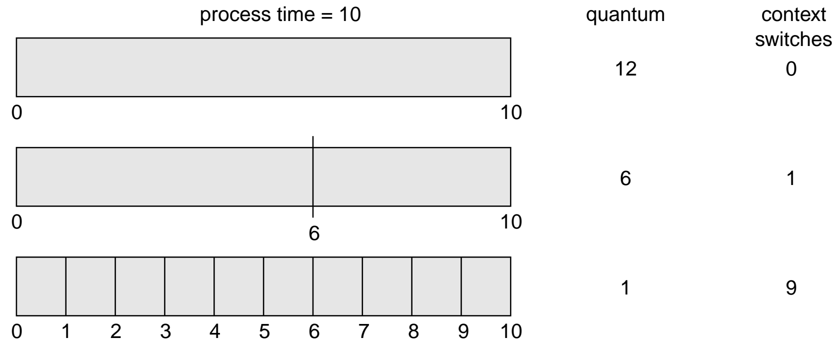
- Diagramma di Gantt



- Tipicamente, si ha un tempo di turnaround medio maggiore, ma minore tempo di risposta

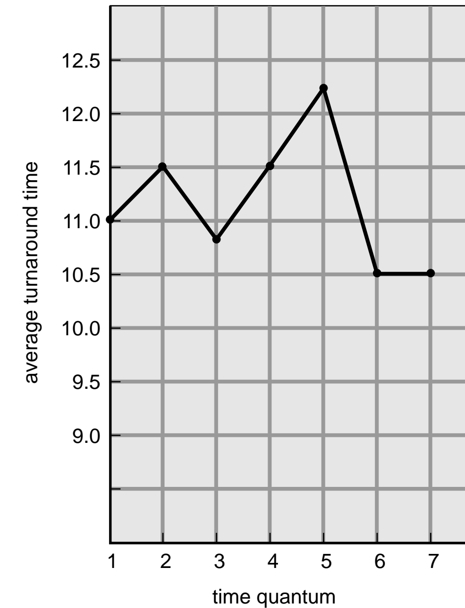
Prestazioni dello scheduling Round-Robin

- q grande $\Rightarrow$ degenera nell'FCFS
- q piccolo $\Rightarrow$ q deve comunque essere grande rispetto al tempo di context switch, altrimenti l'overhead è elevato



- L'80% dei CPU burst dovrebbero essere inferiori a q

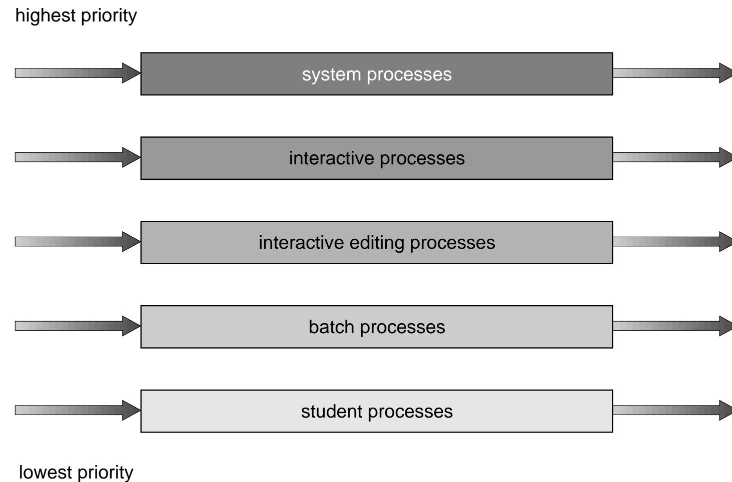
Prestazioni dello scheduling Round-Robin (Cont.)



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Scheduling con code multiple

- La coda di ready è partizionata in più code separate: ad esempio, processi "foreground" (interattivi), processi "background" (batch)



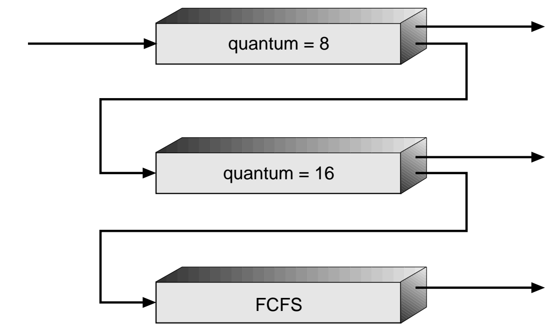
Scheduling con code multiple (Cont.)

- Ogni coda ha un suo algoritmo di scheduling; ad esempio, RR per i foreground, FCFS o SJF per i background
- Lo scheduling deve avvenire tra tutte le code: alternative
  - Scheduling a priorità fissa: eseguire i processi di una coda solo se le code di priorità superiore sono vuote.
    $\Rightarrow$ possibilità di starvation.
  - Quanti di tempo per code: ogni coda riceve un certo ammontare di tempo di CPU per i suoi processi; ad es., 80% ai foreground in RR, 20% ai background in FCFS

Scheduling a code multiple con feedback

- I processi vengono spostati da una coda all'altra, dinamicamente. P.e.: per implementare l'aging: se un processo ha usato recentemente
  - molta CPU, viene spostato in una coda a minore priorità
  - poca CPU, viene spostato in una coda a maggiore priorità
- Uno scheduler a code multiple con feedback viene definito dai seguenti parametri:
  - numero di code
  - algoritmo di scheduling per ogni coda
  - come determinare quando promuovere un processo
  - come determinare quando degradare un processo
  - come determinare la coda in cui mettere un processo che entra nello stato di ready

Esempio di code multiple con feedback



Tre code:

- Q_0 – quanto di 8 msec
- Q_1 – quanto di 16 msec
- Q_2 – FCFS

Scheduling:

- Un nuovo job entra in Q_0, dove viene servito FCFS con prelazione. Se non termina nei suoi 8 millisecondi, viene spostato in Q_1.
- Nella coda Q_1, ogni job è servito FCFS con prelazione, quando Q_0 è vuota. Se non termina in 16 millisecondi, viene spostato in Q_2.
- Nella coda Q_2, ogni job è servito FCFS senza prelazione, quando Q_0 e Q_1 sono vuote.

Schedulazione garantita

- Si promette all'utente un certo quality of service (che poi deve essere mantenuto)
- Esempio: se ci sono n utenti, ad ogni utente si promette $1/n$ della CPU.
- Implementazione:
  - per ogni processo T_p si tiene un contatore del tempo di CPU utilizzato da quando è stato lanciato.
  - il tempo di cui avrebbe diritto è $t_p = T/n$, dove $T =$ tempo trascorso dall'inizio del processo.
  - priorità di $P = T_p/t_p$ — più è bassa, maggiore è la priorità

Schedulazione a lotteria

- Semplice implementazione di una schedulazione "garantita"
  - Esistono un certo numero di "biglietti" per ogni risorsa
  - Ogni utente (processo) acquisisce un sottoinsieme di tali biglietti
  - Viene estratto casualmente un biglietto, e la risorsa viene assegnata al vincitore
- Per la legge dei grandi numeri, alla lunga l'accesso alla risorsa è proporzionale al numero di biglietti
- I biglietti possono essere passati da un processo all'altro per cambiare la priorità (esempio: client/server)

Scheduling multi-processore (cenni)

- Lo scheduling diventa più compesso quando più CPU sono disponibili
- Sistemi *omogenei:* è indiff. su quale processore esegue il prossimo task
- Può comunque essere richiesto che un certo task venga eseguito su un preciso processore (*pinning*)
- Bilanciare il carico (*load sharing*) ⇒ tutti i processori selezionano i processi dalla stessa ready queue
- problema di accesso condiviso alle strutture del kernel
  - *Asymmetric multiprocessing (AMP):* solo un processore per volta può accedere alle strutture dati del kernel — semplifica il problema, ma diminuisce le prestazioni (carico non bilanciato)
  - *Symmetric multiprocessing (SMP):* condivisione delle strutture dati. Serve hardware particolare e di controlli di sincronizzazione in kernel

Scheduling Real-Time

- *Hard real-time:* si richiede che un task critico venga completato entro un tempo ben preciso e garantito.
  - prenotazione delle risorse
  - determinazione di tutti i tempi di risposta: non si possono usare memorie virtuali, connessioni di rete, ...
  - solitamente ristretti ad hardware dedicati
- *Soft real-time:* i processi critici sono prioritari rispetto agli altri
  - possono coesistere con i normali processi time-sharing
  - lo scheduler deve mantenere i processi real-time prioritari
  - la latenza di dispatch deve essere la più bassa possibile
  - adatto per piattaforme general-purpose, per trattamento di audio-video, interfacce real-time, ...

Minimizzare il tempo di latenza

- Un kernel *non prelazionabile* è inadatto per sistemi real-time: un processo non può essere prelazionato durante una system call
  - *Punti di prelazionabilità (preemption points):* in punti "sicuri" delle system call di durata lunga, si salta allo scheduler per verificare se ci sono processi a priorità maggiore
  - *Kernel prelazionabile:* tutte le strutture dati del kernel vengono protette con metodologie di sincronizzazione (semafori). In tal caso un processo può essere sempre interrotto.
- *Inversione delle priorità:* un processo ad alta priorità deve accedere a risorse attualmente allocate da un processo a priorità inferiore.
  - *protocollo di ereditarietà delle priorità:* il processo meno prioritario eredita la priorità superiore finché non rilascia le risorse.

Scheduling di breve termine in Unix tradizionale

(fino a 4.3BSD e SVR3)

- a code multiple, round-robin
- ogni processo ha una priorità di scheduling;
  numeri più grandi indicano priorità minore
- Feedback negativo sul tempo di CPU impiegato
- Invecchiamento dei processi per prevenire la starvation
- Quando un processo rilascia la CPU, va in *sleep* in attesa di un *event*
- Quando l'evento occorre, il kernel esegue un *wakeup* con l'indirizzo dell'evento e il processo *asleep* in testa alla coda sull'evento viene messo nella coda di ready (*)
- I processi che erano in attesa di un evento in modo kernel rientrano con priorità *negativa* e non soggetta a invecchiamento

- (*) Nota:
  - In Unix tradizionale gli eventi sono mappati in indirizzi del kernel
  - Eventi diversi possono essere mappati nello stesso indirizzo kernel (ad es. attesa su un buffer e attesa di completamento di I/O vengono mappati sull'indirizzo del buffer)
  - Più processi possono essere asleep sullo stesso indirizzo kernel
  - Il kernel non tiene traccia di quanti processi sono in attesa
  - A seguito di una wakeup *tutti i processi in attesa* su un evento vengono risvegliati e spostati nello stato Ready
  - Tuttavia molti di essi torneranno subito asleep

Scheduling in Unix tradizionale (Cont.)

- 1 quanto = 5 o 6 tick = 100 msec
- alla fine di un quanto, il processo viene prelazionato
- quando il processo j rilascia la CPU
  - viene incrementato il suo contatore CPU_j di uso CPU
  - viene messo in fondo alla stessa coda di priorità
  - riparte lo scheduler su tutte le code
- 1 volta al secondo, vengono ricalcolate tutte le priorità dei processi in user mode (dove $nice_j$ è un parametro fornito dall'utente):
  $CPU_j = CPU_j/2$ (*fading esponenziale*)
  $P_j = CPU_j + nice_j$

  I processi in kernel mode non cambiano priorità.

Scheduling in Unix tradizionale (Cont.)

In questo esempio, 1 secondo = 4 quanti = 20 tick

Tempo	Processo A Pr_A	CPU_A	Processo B Pr_B	CPU_B	Processo C Pr_C	CPU_C
0	0	0	0	0	0	0
	0	5	0	0	0	0
	0	5	0	5	0	0
	0	5	0	5	0	5
	0	10	0	5	0	5
1	5	5	2	2	2	2
	5	5	2	7	2	2
	5	5	2	7	2	7
	5	5	2	12	2	7
	5	5	2	12	2	12
2	2	2	6	6	6	6
	2	7	6	6	6	6
	2	12	6	6	6	6
	2	17	6	6	6	6
	2	22	6	6	6	6
3	11	11	3	3	3	3
	11	11	3	8	3	3
	11	11	3	8	3	8
	11	11	3	13	3	8

...

Scheduling in Unix tradizionale (Cont.)

Considerazioni

- Adatto per time sharing generale
- Privilegiati i processi I/O bound - tra cui i processi interattivi
- Garantisce assenza di starvation per CPU-bound e batch
- Quanto di tempo indipendente dalla priorità dei processi
- Non adatto per real time
- Non modulare, estendibile

Inoltre il kernel 4.3BSD e SVR3 non era prelazionabile e poco adatto ad architetture parallele.

Scheduling in Unix moderno (4.4BSD, SVR4 e successivi)

Applicazione del principio di separazione tra il meccanismo e le politiche

- Meccanismo generale
  - 160 livelli di priorità (numero maggiore ≡ priorità maggiore)
  - ogni livello è gestito separatamente, event. con politiche differenti
- *classi di scheduling:* per ognuna si può definire una politica diversa
  - intervallo delle priorità che definisce la classe
  - algoritmo per il calcolo delle priorità
  - assegnazione dei quanti di tempo ai varî livelli
  - migrazione dei processi da un livello ad un altro
- Limitazione dei tempi di latenza per il supporto real-time
  - inserimento di punti di prelazionabilità del kernel con check del flag `kprunrun`, settato dalle routine di gestione eventi

Scheduling in Unix moderno (4.4BSD, SVR4 e successivi)

Assegnazione di default: 3 classi

Real time: possono prelazionare il kernel. Hanno priorità e quanto di tempo fisso.

Kernel: prioritari su processi time shared. Hanno priorità e quanto di tempo fisso. Ogni coda è gestita FCFS.

Time shared: per i processi "normali". Ogni coda è gestita round-robin, con quanto minore per priorità maggiore. Priorità variabile secondo una tabella fissa: se un processo termina il suo quanto, scende di priorità.

Priority Class	Global Value	Scheduling Sequence
Real-time	159 • • • • 100	first ↓
Kernel	99 • • 60	↓
Time-shared	59 • • • • 0	↓ last

Considerazioni sullo scheduling SVR4

- Flessibile: configurabile per situazioni particolari
- Modulare: si possono aggiungere altre politiche (p.e., batch)
- Le politiche di default sono adatte ad un sistema time-sharing generale
- manca(va) uno scheduling real-time FIFO (aggiunto in Solaris, Linux, ...)

Scheduling di Windows 2000

Un thread esegue lo scheduler quando

- esegue una chiamata bloccante
- comunica con un oggetto (per vedere se si sono liberati thread a priorità maggiore)
- alla scadenza del quanto di thread

Inoltre si esegue lo scheduler in modo asincrono:

- Al completamento di un I/O
- allo scadere di un timer (per chiamate bloccanti con timeout)

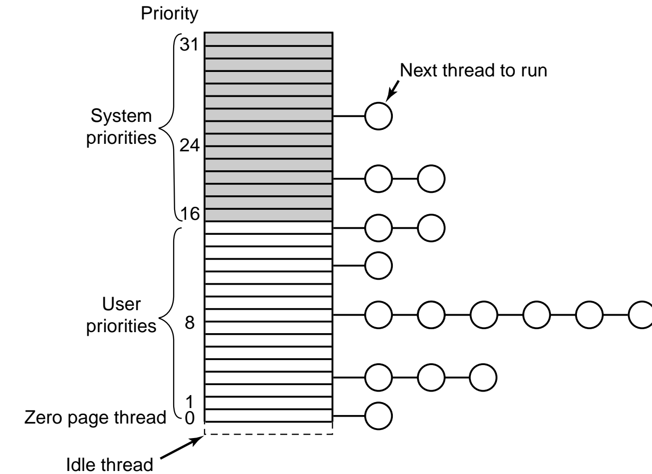
Scheduling di Windows 2000

- I processi possono settare la classe priorità di *processo* (`SetPriorityClass`)
- I singoli thread possono settare la priorità di *thread* (`SetThreadPriority`)
- Queste determinano la *priorità di base* dei thread come segue:

		Win32 process class priorities					
		Realtime	**High**	**Above Normal**	**Normal**	**Below Normal**	**Idle**
Win32 thread priorities	**Time critical**	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Scheduling di Windows 2000

- I thread (NON i processi) vengono raccolti in code ordinate per priorità, ognuna gestita round robin. Quattro classi: *system* ("real time", ma non è vero), *utente*, *zero*, *idle*.

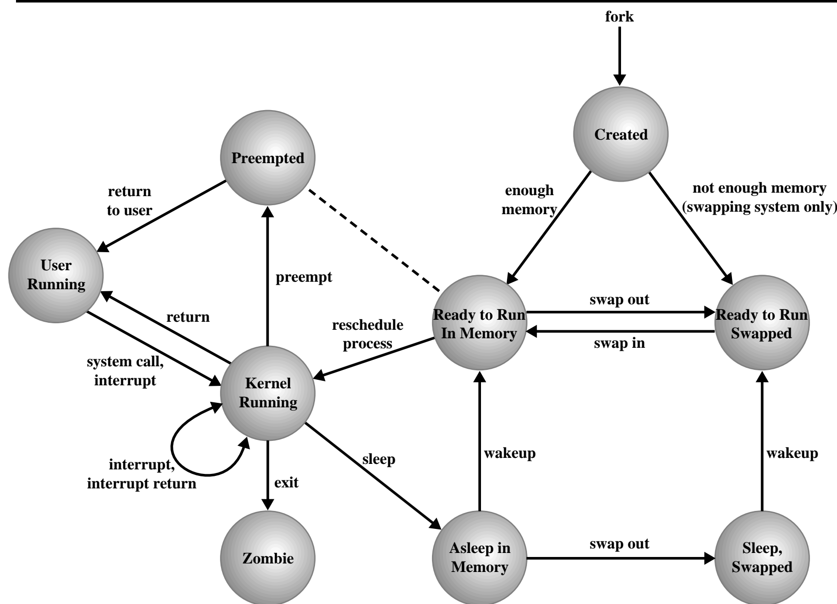


- Solo lo *zero page thread* assume priorità zero (è un thread del kernel che si occupa di ripulire le pagine di memoria quando sono rilasciate da altri thread)
- Lo scheduler sceglie sempre dalla coda a priorità maggiore
- La priorità di un thread utente può essere temporaneamente maggiore di quella base (*spinte*)
  - per thread che attendevano dati di I/O (spinte fino a +8)
  - per dare maggiore reattività a processi interattivi (+2)
  - per risolvere inversioni di priorità

profondimento su Gestione dei processi in U

- Diagramma di stati di un processo Unix
- Esempio di ciclo di vita
- Algoritmo di gestione delle interruzioni
- User e Kernel mode
- Livelli di contesto, trap ed interrupt

Diagramma degli stati di un processo in UNIX



Ciclo di vita di un processo Unix

- Il processo entra nel sistema nello stato *Created* quando il padre crea il processo tramite la chiamata a `fork`
- Il processo muove in uno stato in cui è pronto a partire, ad es., *Ready to run in memory*
- Se viene selezionato dallo scheduler muove nello stato *Kernel running* (esecuzione in Kernel mode) dove termina la sua parte di *fork*.
- Quando termina l'esecuzione della chiamata di sistema può passare allo stato *User running* (esecuzione in modo User) oppure potrebbe passare nello stato *Preempted* (cioè lo scheduler ha selezionato un'altro processo)
- Dallo stato *User running* può passare allo stato *Kernel running* a seguito di un'altra chiamata di sistema oppure di un'interrupt, ad es., di clock

- Dopo aver eseguito la routine di gestione dell'interrupt di clock il Kernel potrebbe schedulare un altro processo da mandare in esecuzione e quindi il processo in questione passerebbe allo stato *Preempted*
- Lo stato *Preempted* enfatizza il fatto che i processi Unix possono essere prelazionati solo quando tornano da modo Kernel a modo utente
- Se durante l'esecuzione in modo Kernel (ad es. di una system call) il processo deve eseguire operazioni di I/O passa allo stato *Asleep in memory* per essere risvegliato successivamente e passare nello stato *Ready to run*
- Gli stati con etichetta *Swapped* corrispondono a situazioni nelle quali il processo non è più fisicamente in memoria centrale (ad es. non c'è abbastanza memoria per gestire i processi in multitasking)

User and Kernel Mode

- I processi Unix possono operare in modo user e kernel: cioè il kernel esegue nel contesto di un processo le operazioni per gestire chiamate di sistema e interrupt
  - Alla partenza del sistema il codice del kernel viene caricato in memoria principale (con strutture dati (tabelle) necessarie per mappare indirizzi virtuali kernel in indirizzi fisici)
  - Un processo in esecuzione in modo user non può accedere allo spazio di indirizzi del kernel
  - Quando un processo passa ad eseguire in modo kernel tale vincolo viene rilasciato: in questo modo si può eseguire codice del kernel (routine di gestione di interrupt/codice di una chiamata di sistema) nel contesto del processo utente
- Il contesto di un processo: contesto utente (codice, dati, stack), registri, e contesto kernel (entry nella tabella dei processi, u-area, stack kernel)

Livelli di contesto

- La parte dinamica del contesto di un processo (kernel stack, registri salvati) è organizzata a sua volta come stack con un numero di posizioni che dipende dai livelli di interrupt diversi ammessi nel sistema

- Ad esempio se il sistema gestisce interrupt software, interrupt di terminali, di dischi, di tutte le altre periferiche, e di clock: avremo al più sette livelli di contesto
  - Livello 0: User
  - Livello 1: Chiamate di sistema
  - Livelli 2-6: Interrupt (l'ordine dipende dalla priorità associata alle interrupt)

Esempio di esecuzione nel contesto di un processo

- Il processo esegue una chiamata di sistema: il kernel salva il suo contesto (registri, program e stack pointer) nel livello 0 e crea il contesto di livello 1

- La CPU riceve e processa un interrupt di disco (il controllo viene fatto prima dell'esecuzione della prossima istruzione): il kernel salva il contesto di livello 1 (registri, stack kernel) e crea il livello 2 nel quale si esegue la routine di gestione dell'interrupt di disco

- La CPU riceve un interrupt di clock: il kernel salva il contesto di livello 2 (registri, stack kernel per la routine di gestione dell'interrupt disco) e crea il livello 3 nel quale si esegue la routine di gestione dell'interrupt di clock

- La routine termina l'esecuzione: il kernel recupera il livello di contesto 2 e così via

- Tutti questi passi vengono fatti sempre all'*interno dello stesso processo*: cambia solo la sua parte di contesto dinamica

Algoritmo di gestione delle interruzioni

- L'algoritmo del kernel per la gestione di un'interrupt consiste dei seguenti passi:
  - salva il contesto del processo corrente
  - determina fonte dell'interrupt (trap/interrupt I/O/ ecc)
  - recupera l'indirizzo di partenza della routine digestione delle interrupt (dal vettore delle interrupt)
  - invoca la routine di gestione dell'interrupt
  - recupera il livello di contesto precedente

- Per motivi di efficienza parte della gestione di interruzioni e trap viene eseguita direttamente dalla CPU (dopo aver seguito un'istruzione): il kernel dipende quindi dal processore

Trap/Interrupt vs Context switch

- Il modo user/kernel permette al kernel di lavorare nel contesto di un altro processo senza dover creare nuovi processi kernel

- Con questo meccanismo un processo in modo kernel può svolgere funzioni logicamente collegate ad altre processi (ad es. la gestione dei dati restituiti da un lettore di disco) e non necessariamente collegate al processo che *ospita* momentaneamente il kernel

- Nota: La gestione di trap/interrupt si basa su una sorta di context switch all'interno di un processo: il controllo non passa ad un'altro processo ma è necessario salvare la parte corrente del contesto dinamico del processo all'interno dello stesso processo

Ma allora quando avviene un context switch tra processi?

- Il kernel vieta context switch arbitrari per mantenere la consistenza delle sue strutture dati
- Il controllo può passare da un processo all'altro in quattro possibili scenari:
  - Quando un processo si sospende
  - Quando termina
  - Quando torna a modo user da una chiamata di sistema ma non è più il processo a più alta priorità
  - Quando torna a modo user dopo che il kernel ha terminato la gestione di un'interrupt a non è più il processo a più alta priorità
- In tutti questi casi il kernel lascia la decisione di quale processo da eseguire allo scheduler

Gestione della Memoria

- Fondamenti
- Associazione degli indirizzi alla memoria fisica
- Spazio indirizzi logico vs. fisico
- Allocazione contigua
  - partizionamento fisso
  - partizionamento dinamico
- Allocazione non contigua
  - Paginazione
  - Segmentazione
  - Segmentazione con paginazione
- Implementazione

Gestione della Memoria

- La memoria è una risorsa importante, e limitata.
- "I programmi sono come i gas reali: si espandono fino a riempire la memoria disponibile"
- Memoria illimitata, infinitamente veloce, economica: non esiste.
- Esiste la *gerarchia della memoria*, gestita dal *gestore della memoria*

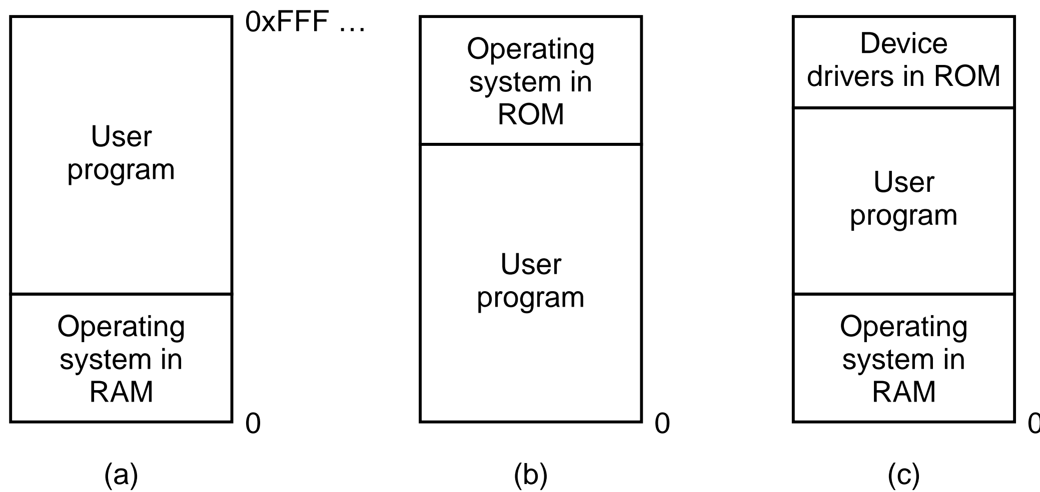
Typical access time		Typical capacity
1 nsec	Registers	<1 KB
2 nsec	Cache	1 MB
10 nsec	Main memory	64-512 MB
10 msec	Magnetic disk	5-50 GB
100 sec	Magnetic tape	20-100 GB

Gestione della memoria: Fondamenti

La gestione della memoria mira a soddisfare questi requisiti:

- Organizzazione logica: offrire una visione astratta della gerarchia della memoria: allocare e deallocare memoria ai processi su richiesta
- Organizzazione fisica: tener conto a chi è allocato cosa, e effettuare gli scambi con il disco.
- Rilocazione
- Protezione: tra i processi, e per il sistema operativo
- Condivisione: aumentare l'efficienza

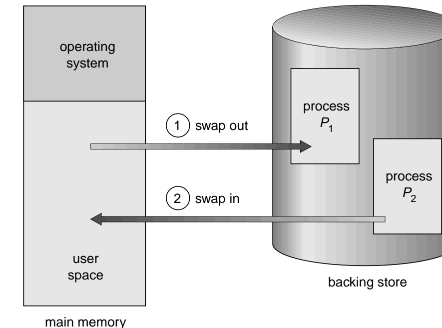
Monoprogrammazione



Un solo programma per volta (oltre al sistema operativo). (a) vecchi mainframe e minicomputer; (b) palmari e sistemi embedded; (c) BIOS (Basic Input Output System) in MS-DOS.

Swapping

- Un processo in esecuzione può essere temporaneamente rimosso dalla memoria e riversato (*swapped*) in una memoria secondaria (detta *backing store* o *swap area*); in seguito può essere riportato in memoria per continuare l'esecuzione.
- Lo spazio indirizzi di interi processi viene spostato
- *Backing store:* dischi veloci e abbastanza larghi da tenere copia delle immagini delle memorie dei processi che si intende swappare.



Swapping (Cont.)

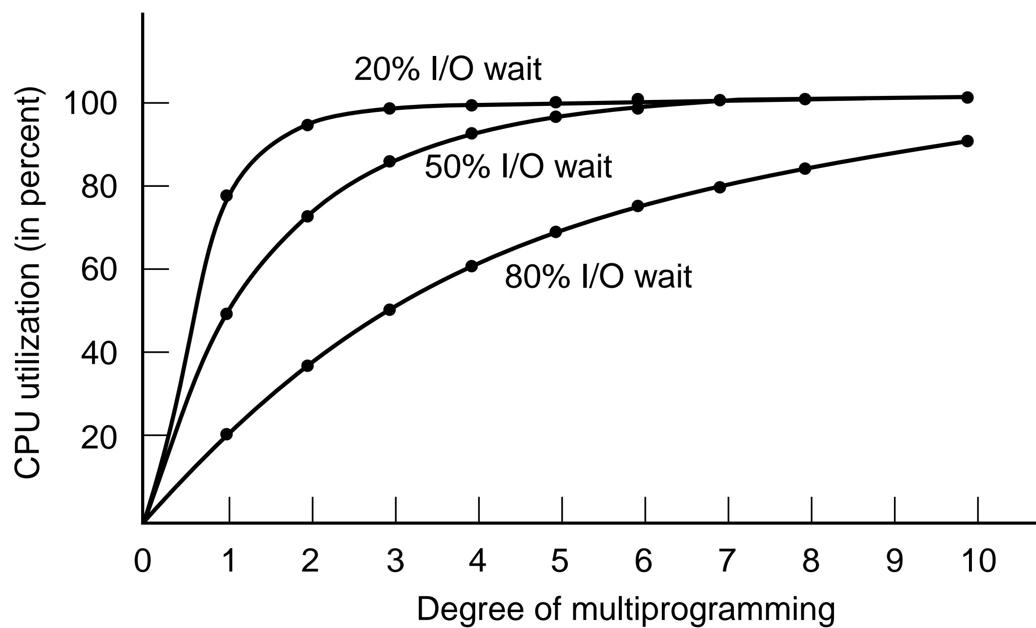
- È gestito dallo scheduler di medio termine
- Allo swap-in, il processo deve essere ricaricato esattamente nelle stesse regioni di memoria, a meno che non ci sia un binding dinamico
- *Roll out, roll in:* variante dello swapping usata per algoritmi di scheduling (a medio termine) a priorità: processi a bassa priorità venogno riversati per permettere il ripristino dei processi a priorità maggiore.
- La maggior parte del tempo di swap è nel trasferimento da/per il disco, che è proporzionale alla dimensione della memoria swappata.
- Per essere swappabile, un processo deve essere "inattivo": buffer di I/O asincrono devono rimanere in memoria, strutture in kernel devono essere rilasciate, etc.
- Attualmente, lo swapping standard non viene impiegato–troppo costoso.
- Versioni modificate di swapping erano implementate in molti sistemi, es. primi Unix, Windows 3.x.

Multiprogrammazione

- La monoprogrammazione non sfrutta la CPU
- Idea: se un processo usa la CPU al 20%, 5 processi la usano al 100%
- Più precisamente, sia p la percentuale di tempo in attesa di I/O di un processo (probabilità che il processo sia in attesa). Con n processi (indipendenti):

$$\text{utilizzo CPU } = 1 - p^n$$

- Maggiore il *grado di multiprogrammazione*, maggiore l'utilizzo della CPU



Se attesa=80%, almeno 10 processi per ottenere il 90% di uso della CPU

Uso del modello

- Il modello precedente è impreciso (i processi non sono indipendenti)
- Può essere utile tuttavia per stimare l'opportunità di upgrade.
- Esempio: Memoria=32MB, sistema operativo=16MB, processi utenti=4MB, attesa media=80
  - Memoria per utenti = 16MB: grado = 4, utilizzo CPU =60%
  - Se aggiungo 16MB agli utenti = 32MB: grado = 8, utilizzo CPU =83% (+38%)
  - Se aggiungo altri 12MB= 48MB: grado = 12, utilizzo CPU =93% (+12%)
  - La seconda upgrade non conviene

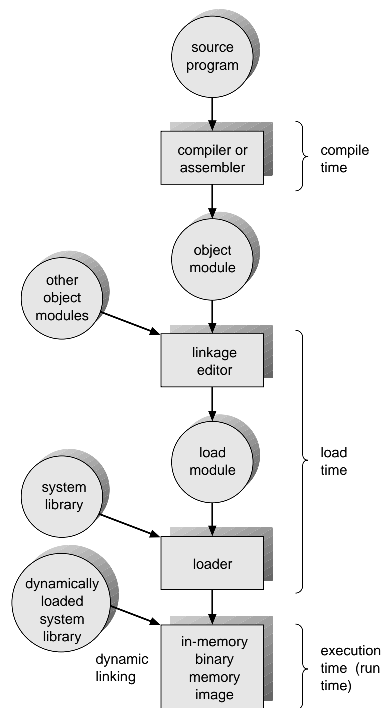
Multiprogrammazione (cont)

- Ogni programma deve essere portato in memoria e posto nello spazio indirizzi di un processo, per poter essere eseguito.
- *Coda in input:* l'insieme dei programmi su disco in attesa di essere portati in memoria per essere eseguiti.
- La selezione è fatta dallo scheduler di lungo termine (se c'è).
- Sorgono problemi di *rilocazione* e *protezione*

Binding degli indirizzi

L'associazione di istruzioni e dati a indirizzi di memoria può avvenire al

- **Compile time:** Se le locazioni di memoria sono note a priori, si può produrre del codice *assoluto*. Deve essere ricompilato ogni volta che si cambia locazione di esecuzione.
- **Load time:** La locazione di esecuzione non è nota a priori; il compilatore genera codice *rilocabile* la cui posizione in memoria viene decisa al momento del caricamento. Non può essere cambiata durante l'esecuzione.
- **Execution time:** L'associazione è fatta durante l'esecuzione. Il programma può essere spostato da una zona all'altra durante l'esecuzione. Necessita di un supporto hardware speciale per gestire questa rilocazione (es. registri *base* e *limite*).



Caricamento dinamico

- Un segmento di codice (eg. routine) non viene caricato finché non serve (la routine viene chiamata).
- Migliore utilizzo della memoria: il codice mai usato non viene caricato.
- Vantaggioso quando grosse parti di codice servono per gestire casi infrequenti (e.g., errori)
- Non serve un supporto specifico dal sistema operativo: può essere realizzato completamente a livello di linguaggio o di programma.
- Il sistema operativo può tuttavia fornire delle librerie per facilitare il caricamento dinamico.

Collegamento dinamico

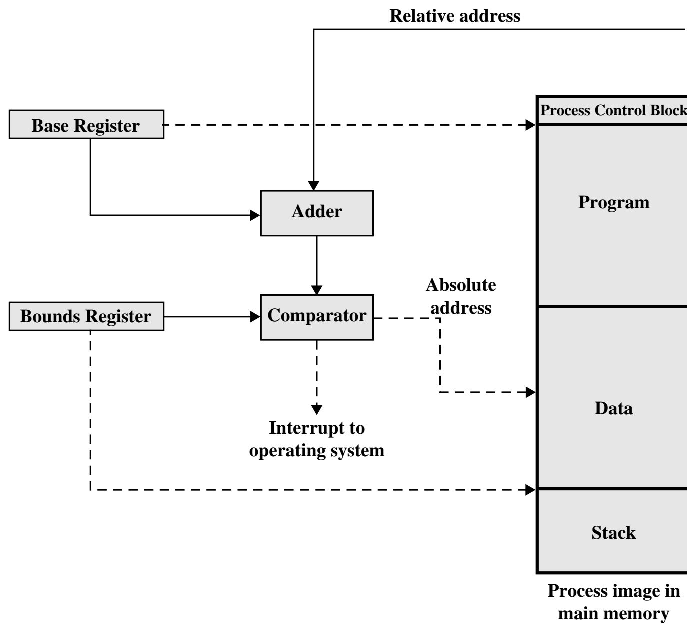
- **Linking dinamico:** le librerie vengono collegate all'esecuzione. Esempi: le .so su Unix, le .DLL su Windows.
- Nell'eseguibile si inseriscono piccole porzioni di codice, dette *stub*, che servono per localizzare la routine.
- Alla prima esecuzione, si carica il segmento se non è presente in memoria, e lo stub viene rimpiazzato dall'indirizzo della routine e si salta alla routine stessa.
- Migliore sfruttamento della memoria: il segmento di una libreria può essere condiviso tra più processi.
- Utili negli aggiornamenti delle librerie (ma bisogna fare attenzione a tener traccia delle versioni!)
- Richiede un supporto da parte del sistema operativo per far condividere segmenti di codice tra più processi.

Spazi di indirizzi logici e fisici

- Il concetto di *spazio indirizzi logico* che viene legato ad uno *spazio indirizzi fisico* diverso e separato è fondamentale nella gestione della memoria.
  - *Indirizzo logico:* generato dalla CPU. Detto anche *indirizzo virtuale*.
  - *Indirizzo fisico:* indirizzo visto dalla memoria.
- Indirizzi logici e fisici coincidono nel caso di binding al compile time o load time
- Possono essere differenti nel caso di binding al tempo di esecuzione. Necessita di un hardware di traduzione.

Memory-Management Unit (MMU)

- È un dispositivo hardware che associa al run time gli indirizzi logici a quelli fisici.
- Nel caso più semplice, il valore del registro di rilocazione viene sommato ad ogni indirizzo richiesto da un processo.
- Il programma utente vede solamente gli indirizzi logici; non vede *mai* gli indirizzi reali, fisici.

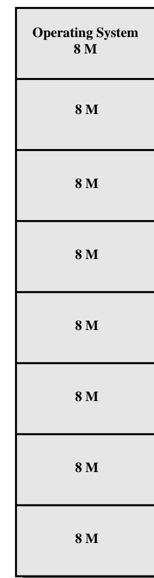


Allocazione contigua

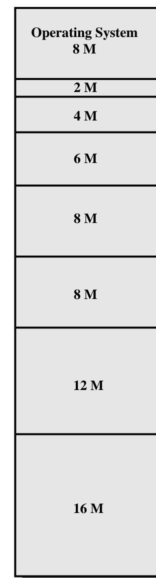
- La memoria è divisa in (almeno) due partizioni:
  - Sistema operativo residente, normalmente nella zona bassa degli indirizzi assieme al vettore delle interruzioni.
  - Spazio per i processi utente — tutta la memoria rimanente.
- Allocazione a partizione singola
  - Un processo è contenuto tutto in una sola partizione
  - Schema di protezione con *registri di rilocazione e limite*, per proteggere i processi l'uno dall'altro e il kernel da tutti.
  - Il registro di rilocazione contiene il valore del primo indirizzo fisico del processo; il registro limite contiene il range degli indirizzi logici.
  - Questi registri sono contenuti nella MMU e vengono caricati dal kernel ad ogni context-switch.

Allocazione contigua: partizionamento statico

- La memoria disponibile è divisa in partizioni fisse (uguali o diverse)
- Il sistema operativo mantiene informazioni sulle partizioni allocate e quelle libere
- Quando arriva un processo, viene scelta una partizione tra quelle libere e completamente allocatagli
- Porta a **frammentazione interna**: la memoria allocata ad un processo è superiore a quella necessaria, e quindi parte non è usata.
- Oggi usato solo su hardware povero

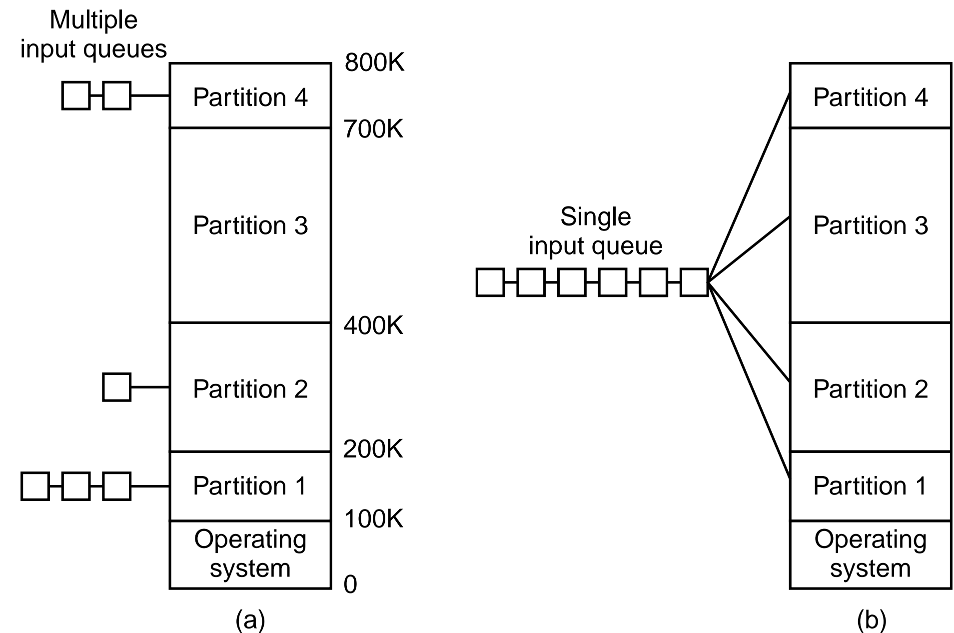


(a) Equal-size partitions



(b) Unequal-size partitions

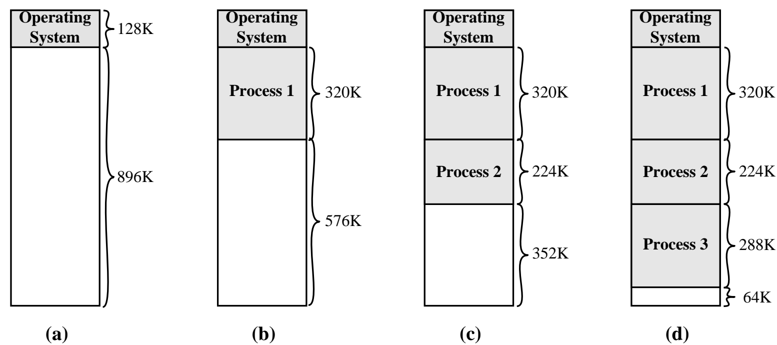
Allocazione contigua: code di input

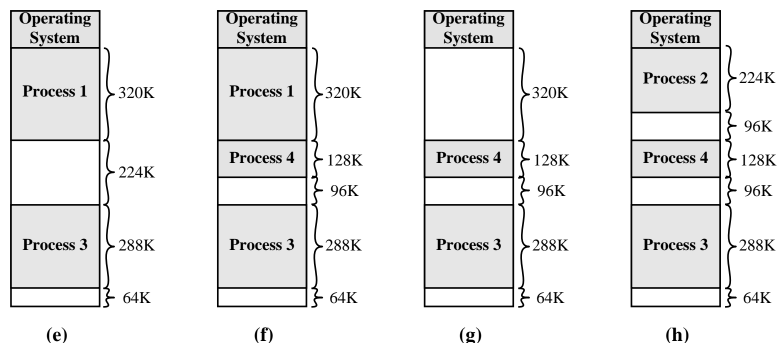


- Quando arriva un processo, viene scelta una partizione tra quelle libere e completamente allocatagli
- Una coda per ogni partizione: possibilità di inutilizzo di memoria
- Una coda per tutte le partizioni: come scegliere il job da allocare?
  - first-fit: per ogni buco, il primo che ci entra
  - best-fit: il più grande che ci entra. Penalizza i job piccoli (che magari sono interattivi. . .)

Allocazione contigua: partizionamento dinamico

- Le partizioni vengono decise al runtime
- *Hole:* blocco di memoria libera. Buchi di dimensione variabile sono sparpagliati lungo la memoria.
- Il sistema operativo mantiene informazioni sulle partizioni allocate e i buchi
- Quando arriva un processo, gli viene allocato una partizione all'interno di un buco sufficientemente largo.





Allocazione contigua: partizionamento dinamico (cont.)

- Hardware necessario: niente se la rilocazione non è dinamica; base-register se la rilocazione è dinamica.
- Non c'è frammentazione interna
- Porta a **frammentazione esterna**: può darsi che ci sia memoria libera sufficiente per un processo, ma non è contigua.
- La frammentazione esterna si riduce con la *compattazione*
  - riordinare la memoria per agglomerare tutti i buchi in un unico buco
  - la compattazione è possibile solo se la rilocazione è dinamica
  - Problemi con I/O: non si possono spostare i buffer durante operazioni di DMA. Due possibilità:
    - ∗ Mantenere fissi i processi coinvolti in I/O
    - ∗ Eseguire I/O solo in buffer del kernel (che non si sposta mai)

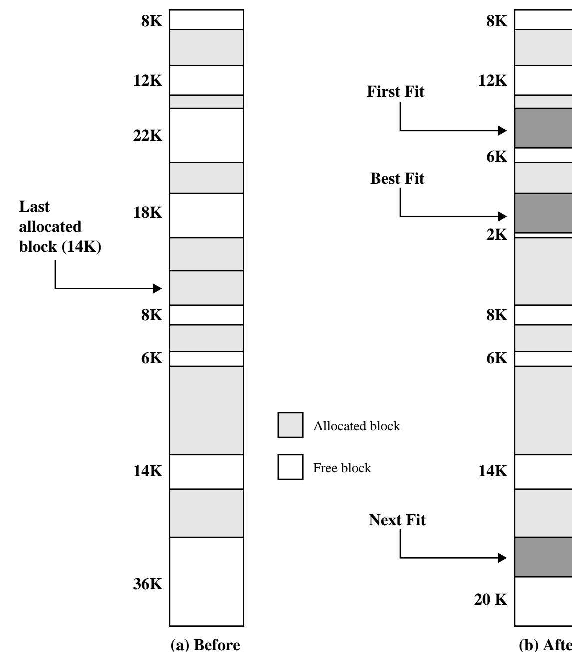
Allocazione contigua: partizionamento dinamico (cont.)

Come soddisfare una richiesta di dimensione n?

- **First-fit:** Alloca il *primo* buco sufficientemente grande
- **Next-fit:** Alloca il *primo* buco sufficientemente grande a partire dall'ultimo usato.
- **Best-fit:** Alloca il *più piccolo* buco sufficientemente grande. Deve scandire l'intera lista (a meno che non sia ordinata). Produce il più piccolo buco di scarto.
- **Worst-fit:** Alloca il *più grande* buco sufficientemente grande. Deve scandire l'indera lista (a meno che non sia ordinata). Produce il più grande buco di scarto.

In generale, gli algoritmi migliori sono il first-fit e il next-fit. Best-fit tende a frammentare molto. Worst-fit è più lento.

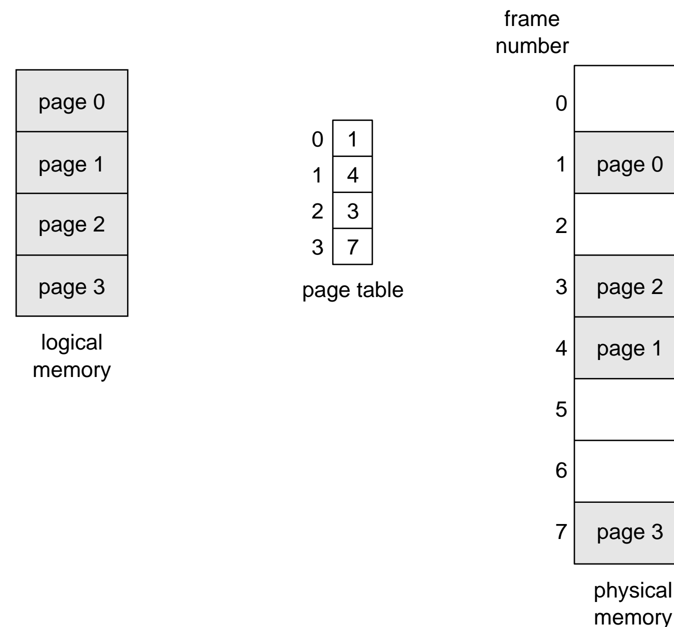
Allocazione contigua: esempi di allocazione



Allocazione non contigua: Paginazione

- Lo spazio logico di un processo può essere non contiguo: ad un processo viene allocata memoria fisica dovunque essa si trovi.
- Si divide la memoria fisica in *frame*, blocchi di dimensione fissa (una potenza di 2, tra 512 e 8192 byte)
- Si divide la memoria logica in *pagine*, della stessa dimensione
- Il sistema operativo tiene traccia dei frame liberi
- Per eseguire un programma di n pagine, servono n frame liberi in cui caricare il programma.
- Si imposta una *page table* per tradurre indirizzi logici in indirizzi fisici.
- Non esiste frammentazione esterna
- Ridotta frammentazione interna

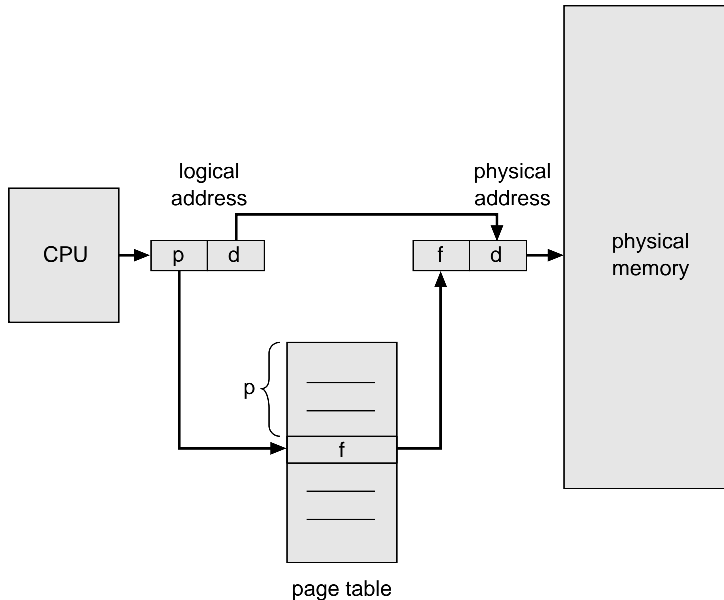
Esempio di paginazione



Schema di traduzione degli indirizzi

L'indirizzo generato dalla CPU viene diviso in

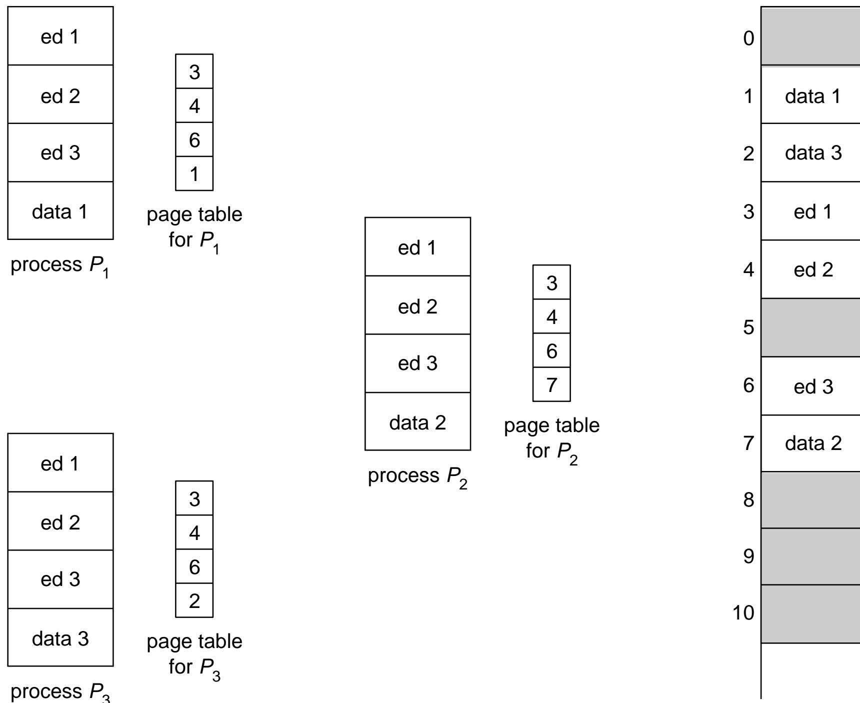
- *Numero di pagina* p*:* usato come indice in una *page table* che contiene il numero del frame contenente la pagina p.
- *Offset di pagina* d*:* combinato con il numero di frame fornisce l'indirizzo fisico da inviare alla memoria.



Paginazione: condivisione

La paginazione permette la condivisione del codice

- Una sola copia di codice read-only può essere condivisa tra più processi. Il codice deve essere *rientrante* (separare codice eseguibile da record di attivazione). Es.: editors, shell, compilatori, ...
- Il codice condiviso appare nelle stesse locazioni logiche per tutti i processi che vi accedono
- Ogni processo mantiene una copia separata dei proprî dati

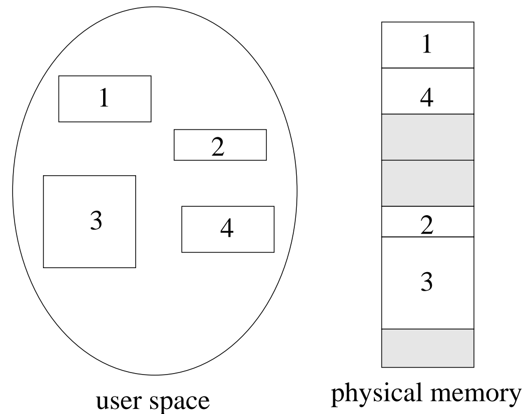


Paginazione: protezione

- La protezione della memoria è implementata associando bit di protezione ad ogni frame.
- *Valid* bit collegato ad ogni entry nella page table
  - "valid" = indica che la pagina associata è nello spazio logico del processo, e quindi è legale accedervi
  - "invalid" = indica che la pagina non è nello spazio logico del processo $\Rightarrow$ violazione di indirizzi (Segment violation)

Allocazione non contigua: Segmentazione

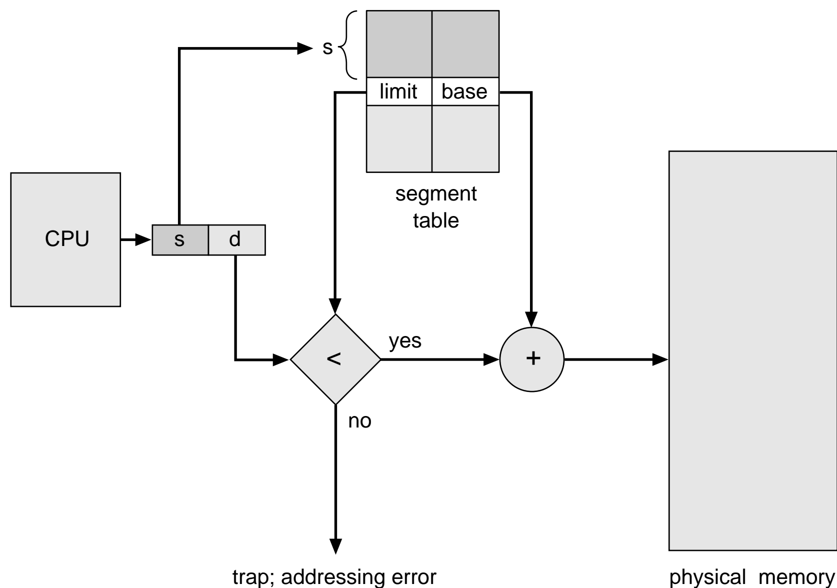
- È uno schema di MM che supporta la visione *utente* della memoria
- Un programma è una collezione di segmenti. Un segmento è una unità logica di memoria; ad esempio: programma principale, procedure, funzioni, variabili locali, variabili globali stack, tabella dei simboli memoria condivisa, ...



Architettura della Segmentazione

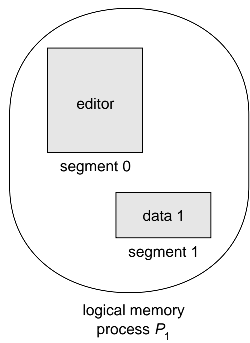
- L'indirizzo logico consiste in un coppia
  <segment-number, offset>.
- La *segment table* mappa gli indirizzi bidimensionali dell'utente negli indirizzi fisici unidimensionali. Ogni entry ha
  - *base:* indirizzo fisico di inizio del segmento
  - *limit:* lunghezza del segmento
- *Segment-table base register (STBR)* punta all'inizio della tabella dei segmenti
- *Segment-table length register (STLR)* indica il numero di segmenti usati dal programma
  segment number s è legale se $s <$ STLR.

Hardware per la segmentazione



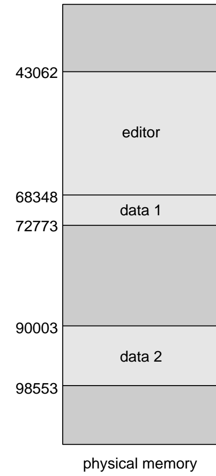
Architettura della Segmentazione (cont.)

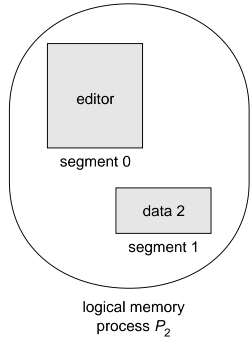
- Rilocazione
  - dinamica, attraverso tabella dei segmenti
- Condivisione
  - interi segmenti possono essere condivisi
- Allocazione
  - gli stessi algoritmi dell'allocazione contigua
  - frammentazione esterna; non c'è frammentazione interna
- Protezione: ad ogni entry nella segment table si associa
  - bit di validità: 0 $\Rightarrow$ segmento illegale
  - privilegi di read/write/execute
- I segmenti possono cambiare di lunghezza durante l'esecuzione (es. lo stack): problema di allocazione dinamica di memoria.



	limit	base
0	25286	43062
1	4425	68348

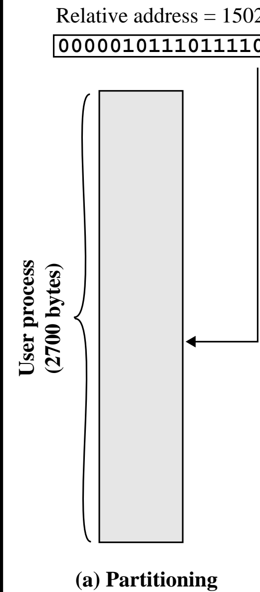
segment table
process P_1

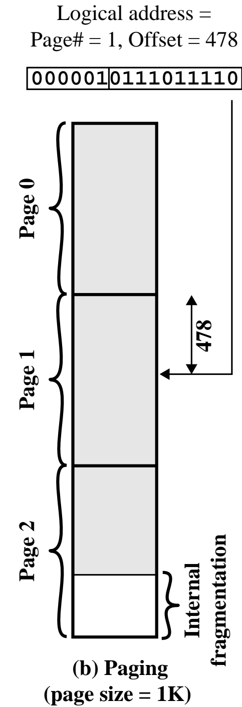


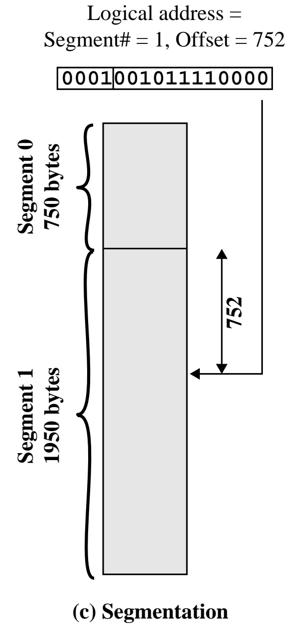


	limit	base
0	25286	43062
1	8850	90003

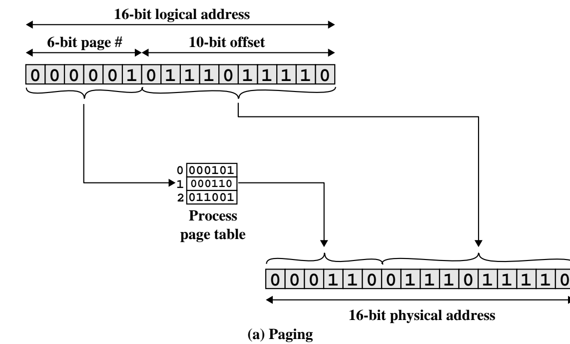
segment table
process P_2

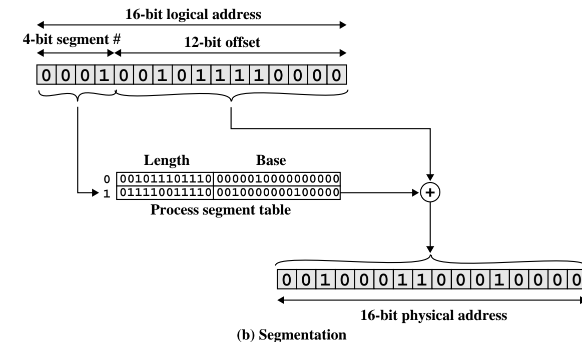






- Indirizzi usati nella paginazione
  - Spazio di indirizzi logici = 2^m
  - Dimensione di pagina = 2^n
  - Numero di pagine = 2^{m-n}
  - $m - n$ bit più significativi di un indirizzo logico per il numero di pagina
  - n bit meno significativi per offset

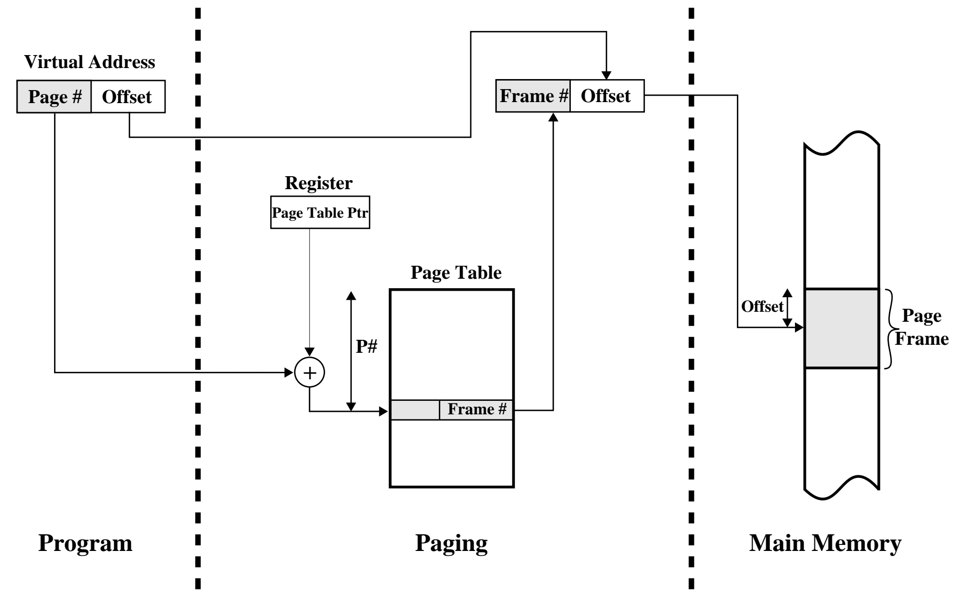




Implementazione della Page Table

- Idealmente, la page table dovrebbe stare in registri veloci della MMU.
  - Costoso al context switch (carico/ricarico di tutta la tabella)
  - Improponibile se il numero delle pagine è elevato. Es: indirizzi virtuali a 32 bit, pagine di 4K (2^{12}): ci sono $2^{20} > 10^6$ entry.
  - Se usiamo 2byte per ogni entry (max RAM = 256M) abbiamo bisogno di $2^{21} = 2M$ in registri.
- La page table viene tenuta in memoria principale
  - *Page-table base register (PTBR)* punta all'inizio della page table
  - *Page-table length register (PTLR)* indica il numero di entry della page table

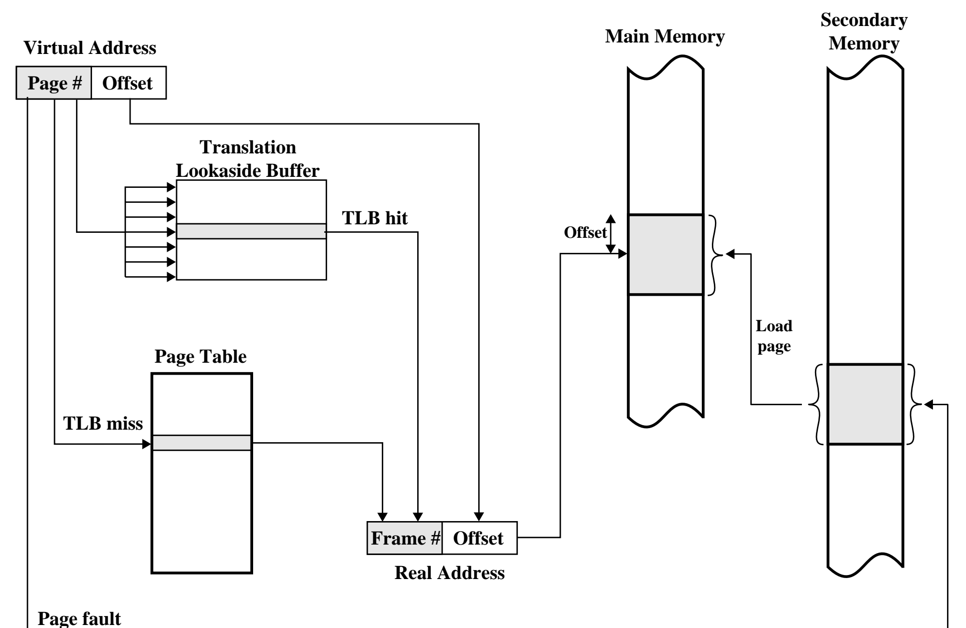
Paginazione con page table in memoria



Paginazione con page table in memoria (cont.)

- Rimane comunque un grande consumo di memoria (1 page table per ogni processo). Nell'es. di prima: 100 processi ⇒ 200M in page tables (su 256MB RAM complessivi).
- Ogni accesso a dati/istruzioni richiede 2 accessi alla memoria: uno per la page table e uno per i dati/istruzioni ⇒ degrado del 100%.
- Il doppio accesso alla memoria si riduce con una cache dedicata per le entry delle page tables: *registri associativi* detti anche *translation look-aside buffer (TLB)*.

Registri Associativi (TLB)



Traduzione indirizzo logico (A', A'') con TLB

- Il virtual page number A' viene confrontato con tutte le entry contemporaneamente.
- Se A' è nel TLB (TLB hit), si usa il frame # nel TLB
- Altrimenti, la MMU esegue un normale lookup nelle page table in memoria, e sostituisce una entry della TLB con quella appena trovata
- Il S.O. viene informato solo nel caso di un page fault

Variante: software TLB

I TLB miss vengono gestiti direttamente dal S.O.

- nel caso di una TLB miss, la MMU manda un interrupt al processore (*TLB fault*)
- si attiva una apposita routine del S.O., che gestisce le page table e la TLB esplicitamente

Abbastanza efficiente con TLB suff. grandi ($\geq$ 64 entries)

MMU estremamente semplice $\Rightarrow$ lascia spazio sul chip per ulteriori cache

Molto usato (SPARC, MIPS, Alpha, PowerPC, HP-PA, Itanium...)

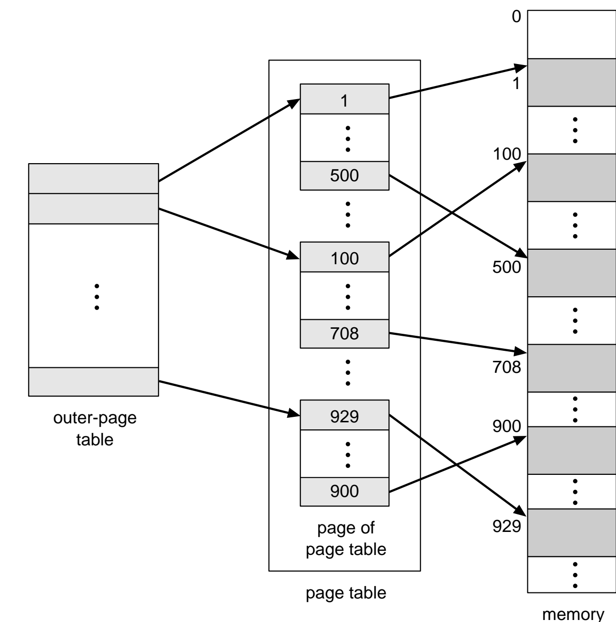
Tempo effettivo di accesso con TLB

- ϵ = tempo del lookup associativo
- t = tempo della memoria
- α = *Hit ratio*: percentuale dei page # reperiti nel TLB (dipende dalla grandezza del TLB, dalla natura del programma...)

$$EAT = (t+\epsilon)\alpha + (2t+\epsilon)(1-\alpha) = (2-\alpha)t + \epsilon$$

- In virtù del *principio di località*, l'hit ratio è solitamente alto
- Con $t = 50ns, \epsilon = 1ns, \alpha = 0.98$ si ha $EAT/t = 1.04$

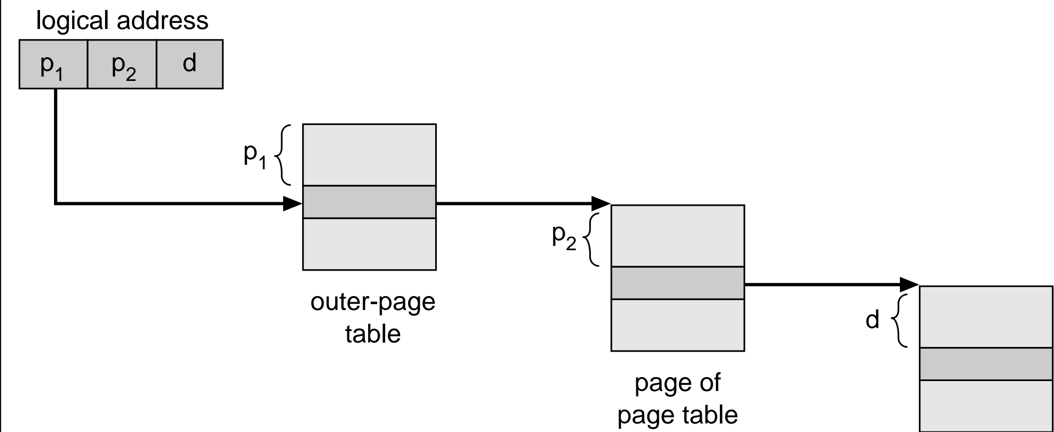
Paginazione a più livelli



Per ridurre l'occupazione della page table, si pagina la page table stessa.
Solo le pagine effettivamente usate sono allocate in memoria RAM.

Esempio di paginazione a due livelli

- Un indirizzo logico (a 32 bit con pagine da 4K) è diviso in
  - un numero di pagina consistente in 20 bit
  - un offset di 12 bit
- La page table è paginata, quindi il numero di pagina è diviso in
  - un *directory number* di 10 bit
  - un *page offset* di 10 bit.



Performance della paginazione a più livelli

- Dato che ogni livello è memorizzato in RAM, la conversione dell'indirizzo logico in indirizzo fisico può necessitare di 4 accessi alla memoria (in caso di TLB miss: lookup in TLB, 2 accessi alla RAM per la rilocazione, 1 accesso per leggere la casella di memoria)
- Il caching degli indirizzi di pagina permette di ridurre drasticamente l'impatto degli accessi multipli. Ad esempio se in caso di TLB miss servono 5 accessi alla memoria (ad es. 4 livelli di page tabling)

$$EAT = \alpha(t + \epsilon) + (1 - \alpha)(5t + \epsilon) = \epsilon + (5 - 4\alpha)t$$

- Nell'esempio di prima, con un hit rate del 98%: $EAT/t = 1.1$: 10% di degrado
- Schema molto adottato da CPU a 32 bit (IA32 (Pentium), 68000, SPARC a 32 bit, ...)

Tabella delle pagine invertita

- Una tabella con una entry per ogni *frame*, non per ogni page.
- Ogni entry consiste nel numero della pagina (virtuale) memorizzata in quel frame, con informazioni riguardo il processo che possiede la pagina.
- Diminuisce la memoria necessaria per memorizzare le page table, ma aumenta il tempo di accesso alla tabella.
- Questo schema è usato su diversi RISC a 32 bit (PowerPC), e tutti quelli a 64 bit (UltraSPARC, Alpha, HPPA, ...), ove una page table occuperebbe petabytes (es: a pagine da 4k: $8 \times 2^{52} = 32$PB per ogni page table)

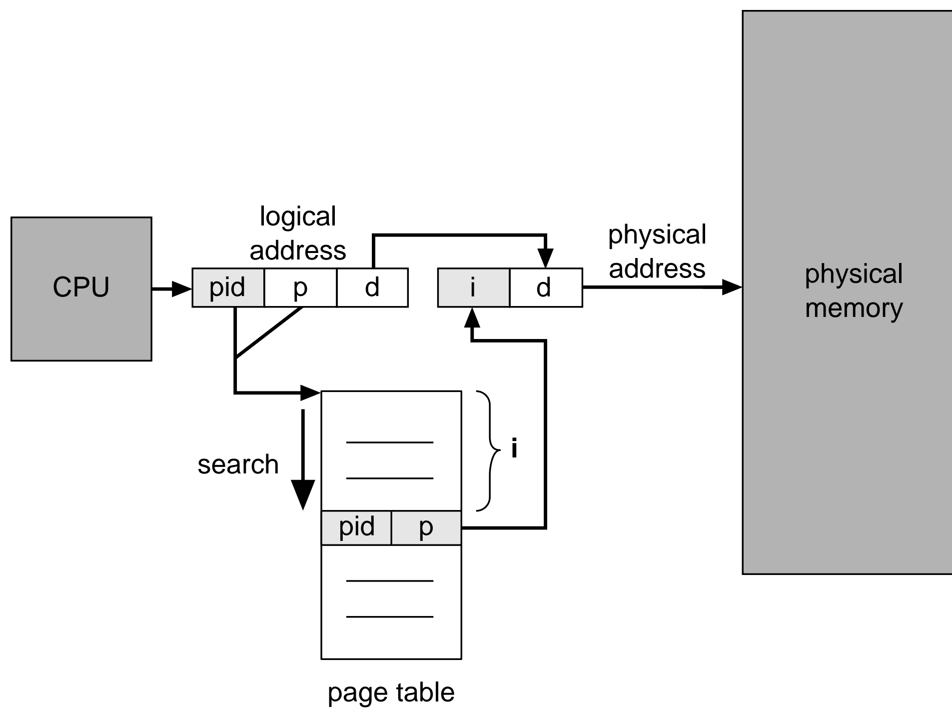
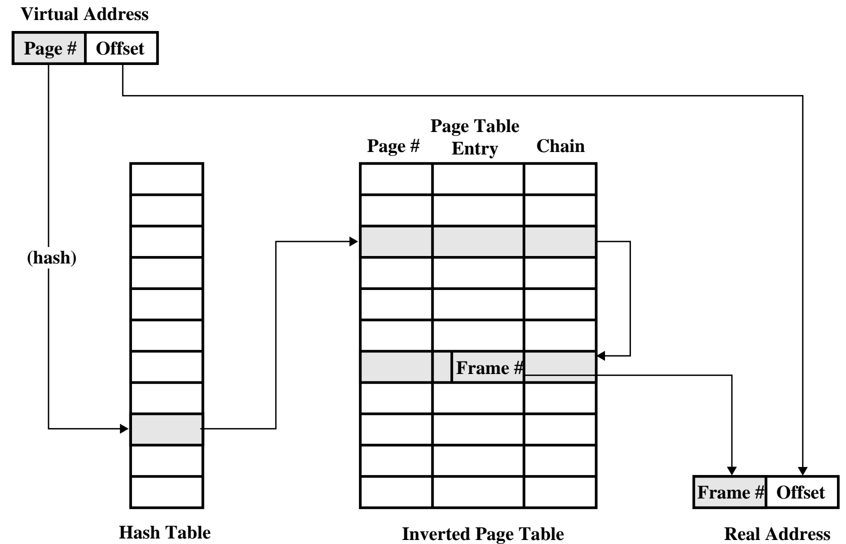


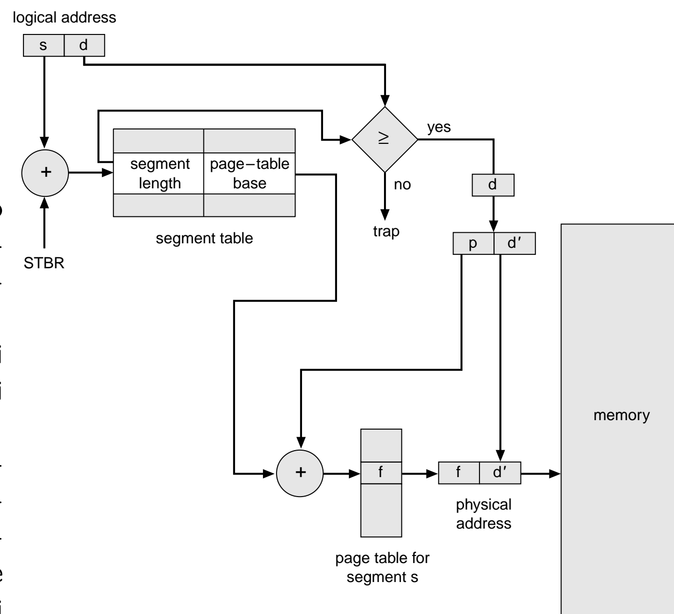
Tabella delle pagine invertita con hashing

Per ridurre i tempi di ricerca nella tabella invertita, si usa una funzione di hash (hash table) per limitare l'accesso a poche entry (1 o 2, solitamente).

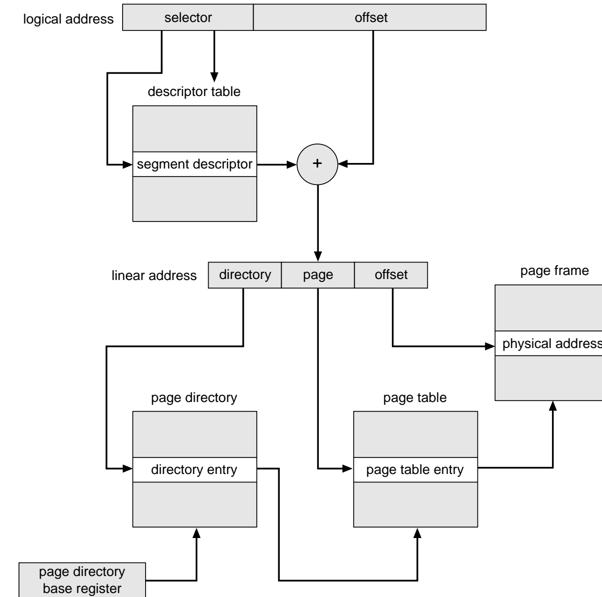


Segmentazione con paginazione: MULTICS

- Il MULTICS ha risolto il problema della frammentazione esterna paginando i segmenti
- Permette di combinare i vantaggi di entrambi gli approcci
- A differenza della pura segmentazione, nella segment table ci sono gli indirizzi base delle page table dei segmenti



Segmentazione con paginazione a 2 livelli: la IA32



Sommario sulle strategie della Gestione della Memoria

- Supporto Hardware: da registri per base-limite a tabelle di mappatura per segmentazione e paginazione
- Performance: maggiore compessità del sistema, maggiore tempo di traduzione. Un TLB può ridurre sensibilmente l'overhead.
- Frammentazione: la multiprogrammazione aumenta l'efficienza temporale. Massimizzare il num. di processi in memoria richiede ridurre spreco di memoria non allocabile. Due tipi di frammentazione.
- Rilocazione: la compattazione è impossibile con binding statico/al load time; serve la rilocazione dinamica.

Sommario sulle strategie della Gestione della Memoria (Cont)

- Swapping: applicabile a qualsiasi algoritmo. Legato alla politica di scheduling a medio termine della CPU.
- Condivisione: permette di ridurre lo spreco di memoria e quindi aumentare la multiprogrammazione. Generalmente, richiede paginazione e/o segmentazione. Altamente efficiente, ma complesso da gestire (dipendenze sulle versioni).
- Protezione: modalità di accesso associate a singole sezioni dello spazio del processo, in caso di segmentazione/paginazione. Permette la condivisione e l'identificazione di errori di programmazione.

Memoria Virtuale

Memoria virtuale: separazione della memoria logica vista dall'utente (programmatore) dalla memoria fisica

Solo *parte* del programma e dei dati devono stare in memoria affinché il processo possa essere eseguito (*resident set*)

Memoria Virtuale: perché

Molti vantaggi sia per gli utenti che per il sistema

- Lo spazio logico può essere molto più grande di quello fisico.
- Meno consumo di memoria $\Rightarrow$ più processi in esecuzione $\Rightarrow$ maggiore multiprogrammazione
- Meno I/O per caricare i programmi

Porta alla necessità di caricare e salvare parti di memoria dei processi da/per il disco al runtime.

La memoria virtuale può essere implementata come
paginazione su richiesta oppure
segmentazione su richiesta

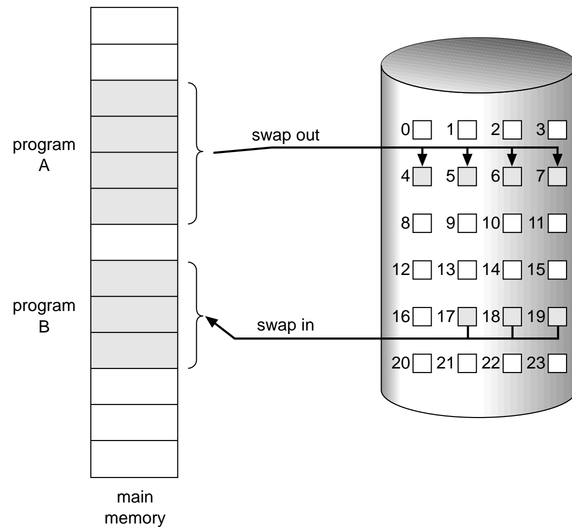
Paginazione su richiesta

Schema a paginazione, ma in cui si carica una pagina in memoria solo quando è necessario

- Meno I/O
- Meno memoria occupata
- Maggiore velocità
- Più utenti/processi

Una pagina è richiesta quando vi si fa riferimento

- viene segnalato dalla MMU
- se l'accesso non è valido ⇒ abortisci il processo
- se la pagina non è in memoria ⇒ caricala dal disco



Swapping vs. Paging

Spesso si confonde *swapping* con *paging*

- Swapping: scambio di interi processi da/per il backing store

  Swapper: processo che implementa una politica di swapping (scheduling di medio termine)

- Paging: scambio di gruppi di pagine (sottoinsiemi di processi) da/per il backing store

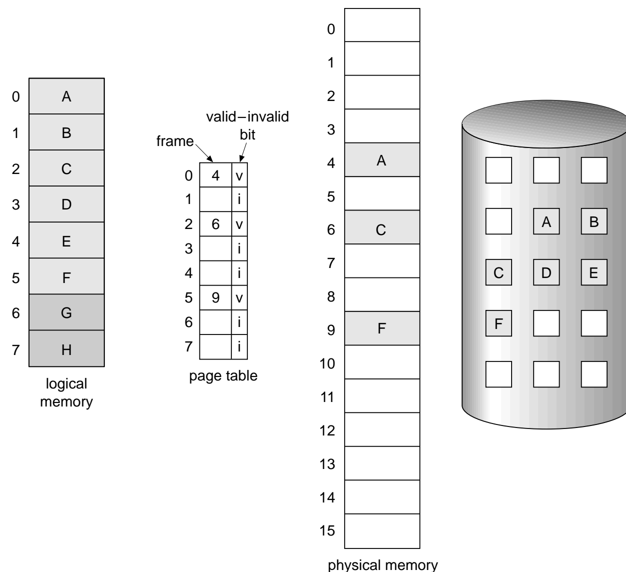
  Pager: processo che implementa una politica di gestione delle pagine dei processi (caricamento/scaricamento).

Sono concetti molto diversi, e non esclusivi!

Purtroppo, in alcuni S.O. il pager viene chiamato "swapper" (es.: Linux: `kswapd`)

Valid–Invalid Bit

- Ad ogni entry nella page table, si associa un bit di validità. (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Inizialmente, il bit di validità è settato a 0 per tutte le pagine.
- La prima volta che si fa riferimento ad una pagina (non presente in memoria), la MMU invia un interrupt alla CPU: *page fault.*

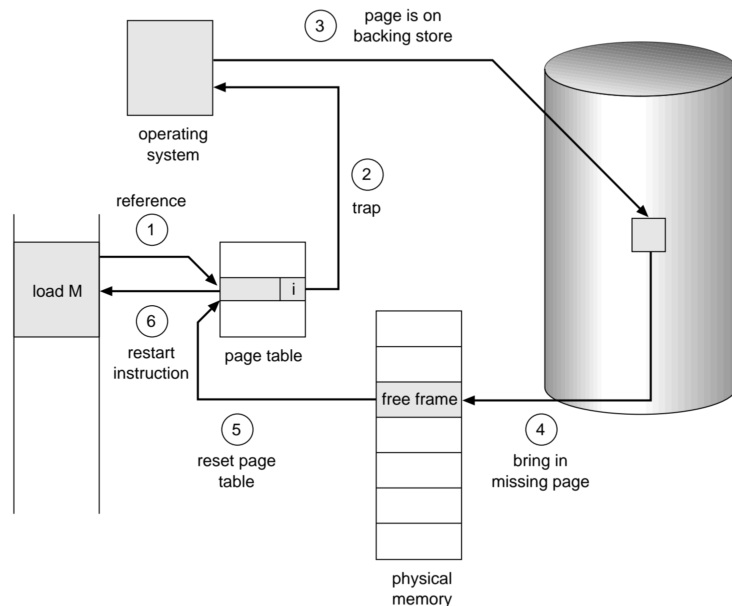


Routine di gestione del Page Fault

- il S.O. controlla in un'altra tabella se è un accesso non valido (fuori dallo spazio indirizzi virtuali assegnati al processo) ⇒ abort del processo ("segmentation fault")
- Se l'accesso è valido, ma la pagina non è in memoria bisogna:
  - Trovare qualche pagina in memoria, che non sia usata, e scaricarla su disco (*swap out*)
  - Caricare la pagina richiesta nel frame così liberato (*swap in*)
  - Aggiornare le tabelle delle pagine
- L'istruzione che ha causato il page fault deve essere rieseguita in modo consistente

  ⇒ vincoli sull'architettura della macchina. Es: la MVC dell'IBM 360.

Page Fault: gestione



Performance del paging on-demand

- p = Page fault rate; $0 \leq p \leq 1$
  - $p = 0 \Rightarrow$ nessun page fault
  - $p = 1 \Rightarrow$ ogni riferimento in memoria porta ad un page fault
- Tempo effettivo di accesso (EAT)

$$
\begin{aligned}
EAT &= (1-p) \times \text{accesso alla memoria} \\
&\quad +p(\text{overhead di page fault} \\
&\qquad [+\text{swap page out}] \\
&\qquad +\text{swap page in} \\
&\qquad +\text{overhead di restart})
\end{aligned}
$$

Esempio di Demand Paging

- Tempo di accesso alla memoria (comprensivo del tempo di traduzione) (TAM): 60 nsec
- Assumiamo che 50% delle volte che una pagina deve essere rimpiazzata, è stata modificata e quindi deve essere scaricata su disco.
- Tralasciamo overhead di page fault e restart
- Swap Page Time (SPT) = 5 msec = $5 * 10^6$ nsec (disco molto veloce!)
- $EAT = 60(1-p) + (2 * SPT)p/2 + (SPT)p/2 =$
  $= 60(1-p) + 5 * 10^6 * 1.5 * p + = 60 + (7.5 * 10^6 - 60)p$ in nsec
- Si quindi ha un degrado del 10% (rispetto al tempo di accesso alla memoria TAM) quando $66 = 60 + (7.5 * 10^6 - 60)p$ cioè

$$p = 6/(7.5 * 10^6 - 60) = 1/1250000$$

Considerazioni sul Demand Paging

- problema di performance: si vuole un algoritmo di rimpiazzamento che porti al minor numero di page fault possibile
- l'area di swap deve essere il più veloce possibile $\Rightarrow$ meglio tenerla separata dal file system (possibilmente anche su un device dedicato) ed accedervi direttamente (senza passare per il file system). Blocchi fisici = frame in memoria.
- La memoria virtuale con demand paging ha benefici anche alla creazione dei processi

Creazione dei processi: Copy on Write

- Il Copy-on-Write permette al padre e al figlio di condividere inizialmente le stesse pagine in memoria.

  Una pagina viene copiata *se* e *quando* viene acceduta in scrittura.

- COW permette una creazione più veloce dei processi

- Le pagine libere devono essere allocate da un set di pagine azzerate

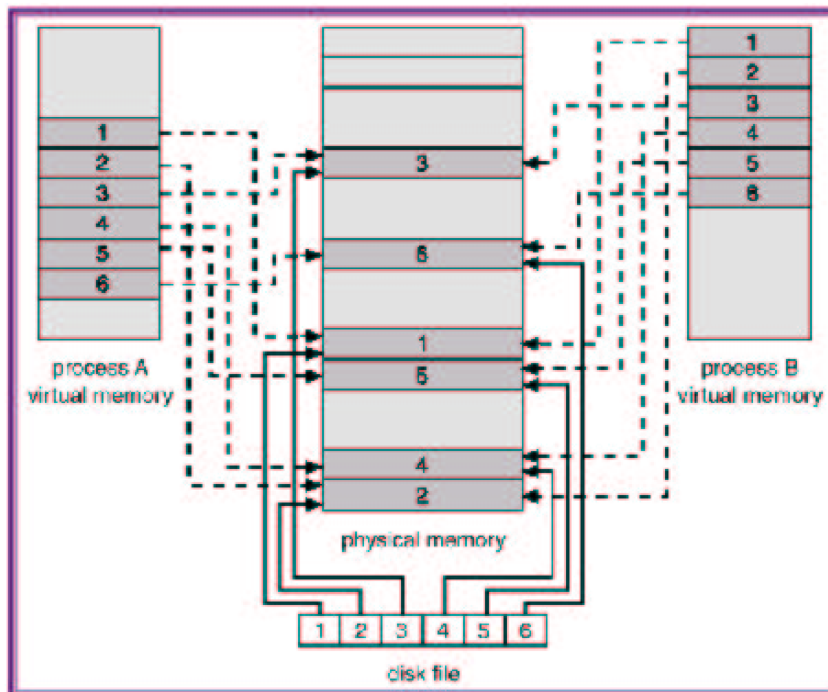
Memory-Mapped I/O

- Memory-mapped file I/O permette di gestire l'I/O di file come accessi in memoria: ogni blocco di un file viene *mappato* su una pagina di memoria virtuale

- Un file (es. DLL, .so) può essere così letto come se fosse in memoria, con demand paging. Dopo che un blocco è stato letto una volta, rimane caricato in memoria senza doverlo rileggere.

- La gestione dell'I/O è molto semplificata

- Più processi possono condividere lo stesso file, condividendo gli stessi frame in cui viene caricato.



Sostituzione delle pagine

- Aumentando il grado di multiprogrammazione, la memoria viene *sovrallocata:* la somma degli spazi logici dei processi in esecuzione è superiore alla dimensione della memoria fisica

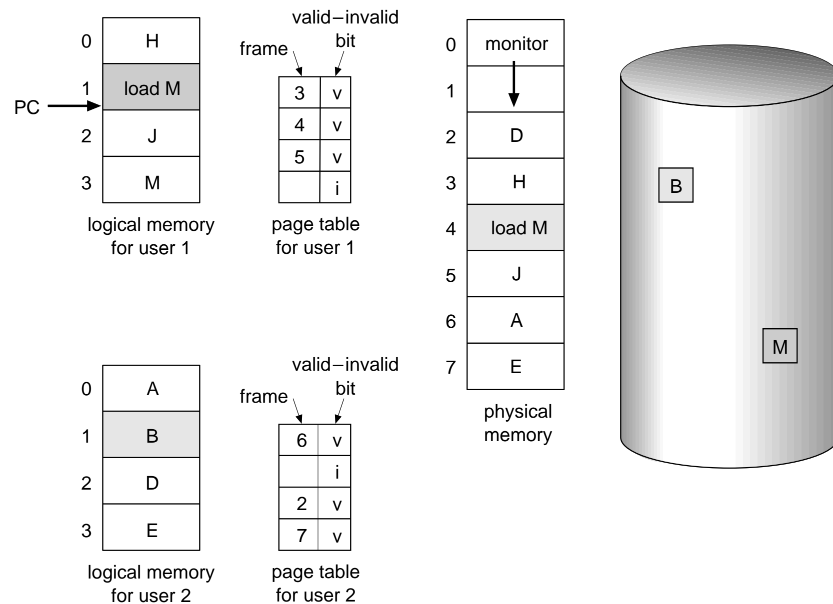
- Ad un page fault, può succedere che non esistono frame liberi

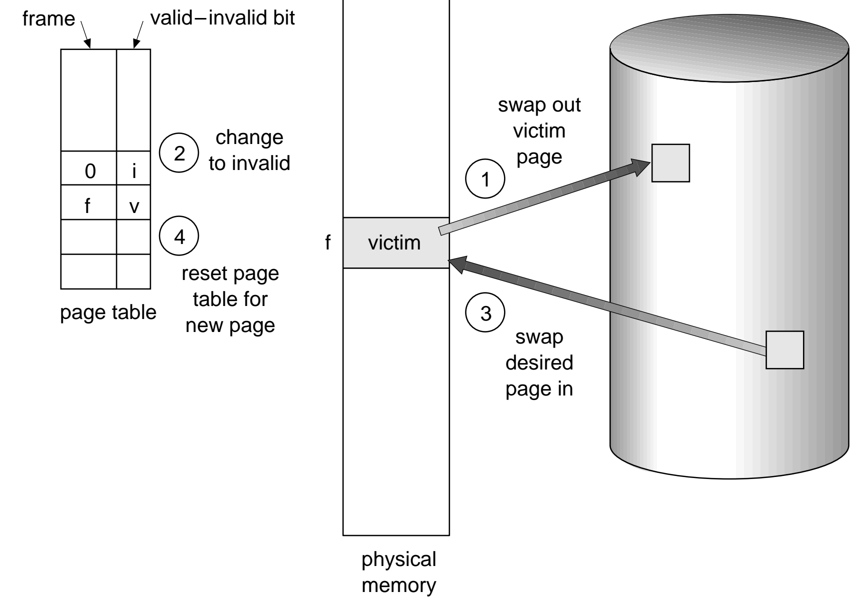
- Si modifica la routine di gestione del page fault aggiungendo la *sostituzione delle pagine* che libera un frame occupato (*vittima*)

- Bit di modifica (*dirty bit*): segnala quali pagine sono state modificate, e quindi devono essere salvate su disco. Riduce l'overhead.

- Il rimpiazzamento di pagina completa la separazione tra memoria logica e memoria fisica: una memoria logica di grandi dimensioni può essere implementata con una piccola memoria fisica.

Sostituzione delle pagine (cont.)





Algoritmi di rimpiazzamento delle pagine

- È un problema molto comune, non solo nella gestione della memoria (es: cache di CPU, di disco, di web server. . .)
- Si mira a minimizzare il page-fault rate.
- In generale (ma non sempre) maggiore memoria implica minor tasso di page fault
- Un modo per valutare questi algoritmi: provarli su una sequenza prefissata di accessi alla memoria, e contare il numero di page fault.
- In tutti i nostri esempi, la sequenza sarà di 5 pagine in questo ordine

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

Algoritmo First-In-First-Out (FIFO)

- Si rimpiazza la pagina che da più tempo è in memoria
- Data la sequenza:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

  - Con 3 frame (3 pagine per volta possono essere in memoria): 9 page fault
  - Con 4 frame: 10 page fault
- Il rimpiazzamento FIFO soffre dell'*anomalia di Belady:* + memoria fisica

$\not\Rightarrow$ – page fault!

number of page faults: 16, 14, 12, 10, 8, 6, 4, 2

number of frames: 1, 2, 3, 4, 5, 6, 7

Algoritmo ottimale (OPT o MIN)

- Si rimpiazza la pagina che non verrà riusata per il periodo più lungo
- Data la sequenza

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

  con 4 frame: 6 page fault
- Tra tutti gli algoritmi, è quello che porta al minore numero di page fault e non soffre dell'anomalia di Belady
- Ma come si può prevedere quando verrà riusata una pagina?
- Algoritmo usato in confronti con altri algoritmi

Algoritmo Least Recently Used (LRU)

- Approssimazione di OPT: studiare il passato per prevedere il futuro
- Si rimpiazza la pagina che da più tempo non viene usata
- Con 4 frame: 8 page fault
- È la soluzione ottima con ricerca *all'indietro* nel tempo: LRU su una stringa di riferimenti r è OPT sulla stringa $reverse(r)$
- Quindi la frequenza di page fault per la LRU è la stessa di OPT su stringhe invertite.
- Non soffre dell'anomalia di Belady (è un *algoritmo di stack*)
- Generalmente è una buona soluzione
- Problema: LRU necessita di notevole assistenza hardware

Matrice di memoria

Dato un algoritmo di rimpiazzamento, e una reference string, si definisce la **matrice di memoria**: $M(m, r)$ è l'insieme delle pagine caricate all'istante r avendo m frames a disposizione

Esempio di matrice di memoria per LRU:

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults	P	P	P	P	P	P	P		P					P			P						P	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

Algoritmi di Stack

Un algoritmo di rimpiazzamento si dice *di stack* se per ogni reference string r, per ogni memoria m:

$$M(m, r) \subseteq M(m + 1, r)$$

Ad esempio, OPT e LRU sono algoritmi di stack. FIFO non è di stack

Fatto: Gli algoritmi di stack non soffrono dell'anomalia di Belady.

Implementazioni di LRU

Implementazione a contatori

- La MMU ha un contatore (32-64 bit) che viene automaticamente incrementato dopo ogni accesso in memoria.
- Ogni entry nella page table ha un registro (*reference time*)
- ogni volta che si riferisce ad una pagina, si copia il contatore nel registro della entry corrispondente
- Quando si deve liberare un frame, si cerca la pagina con il registro più basso

Molto dispendioso, se la ricerca viene parallelizzata in hardware.

Implementazioni di LRU (Cont.)

Implementazione a stack

- si tiene uno stack di numeri di pagina in un lista double-linked (puntatori next, previous, head, tail)
- Quando si riferisce ad una pagina, la si sposta sul top dello stack (Richiede la modifica di 6 puntatori).
- Quando si deve liberare un frame, la pagina da swappare è quella in fondo allo stack: non serve fare una ricerca

Implementabile in software (microcodice). Costoso in termini di tempo.

Approssimazioni di LRU: reference bit e NFU

Bit di riferimento (*reference bit*)

- Associare ad ogni pagina un bit R, inizialmente =0
- Quando si riferisce alla pagina, R viene settato a 1
- Si rimpiazza la pagina che ha $R = 0$ (se esiste).
- Non si può conoscere l'ordine: impreciso.

Variante: **Not Frequently Used** (NFU)

- Ad ogni pagina si associa un contatore
- Ad intervalli regolari (*tick*, tip. 10-20ms), per ogni entry si somma il reference bit al contatore.
- Problema: pagine usate molto tempo fa contano come quelle recenti

Approssimazioni di LRU: aging

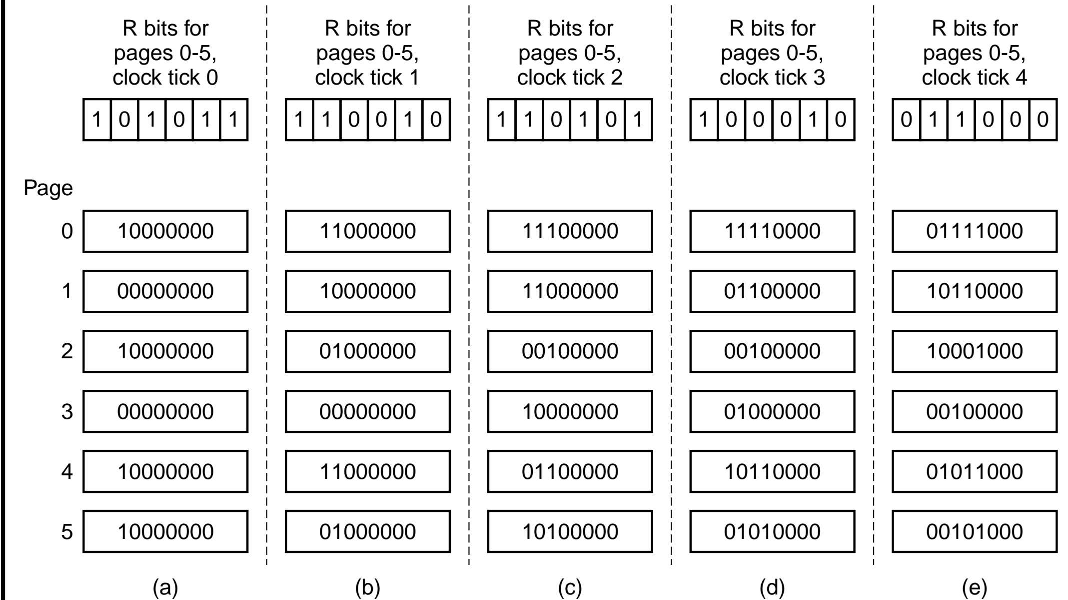
Aggiungere bit supplementari di riferimento, con peso diverso.

- Ad ogni pagina si associa un array di bit, inizialmente =0
- Ad intervalli regolari, un interrupt del timer fa partire una routine che sposta di un bit a destra (right-shift) gli array di tutte le pagine immettendo nel bit più significativo di ogni array il corrispondente bit di riferimento, che poi viene posto a 0
- Si rimpiazza la pagina che ha il numero binario più piccolo nell'array

Differenze con LRU:

- Non può distingure tra pagine accedute nello stesso tick.
- Il numero di bit è finito $\Rightarrow$ la memoria è limitata

In genere comunque è una buona approssimazione.



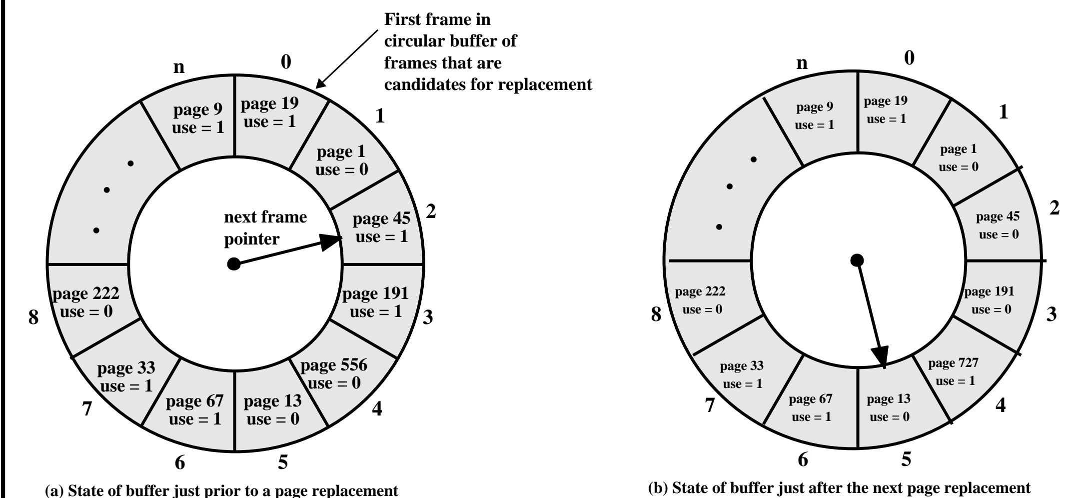
Approssimazioni di LRU: CLOCK (o "Second chance")

Idea di base: se una pagina è stata usata pesantemente di recente, allora probabilmente verrà usata pesantemente anche prossimamente.

- Utilizza il reference bit.
- Si segue un ordine "ad orologio"
- Se la pagina candidato ha il reference bit = 0, rimpiazzala
- se ha il bit = 1, allora
  - imposta il reference bit 0.
  - lascia la pagina in memoria
  - passa alla prossima pagina, seguendo le stesse regole

Nota: se tutti i bit=1, degenera in un FIFO

Buona approssimazione di LRU; usato (con varianti) in molti sistemi

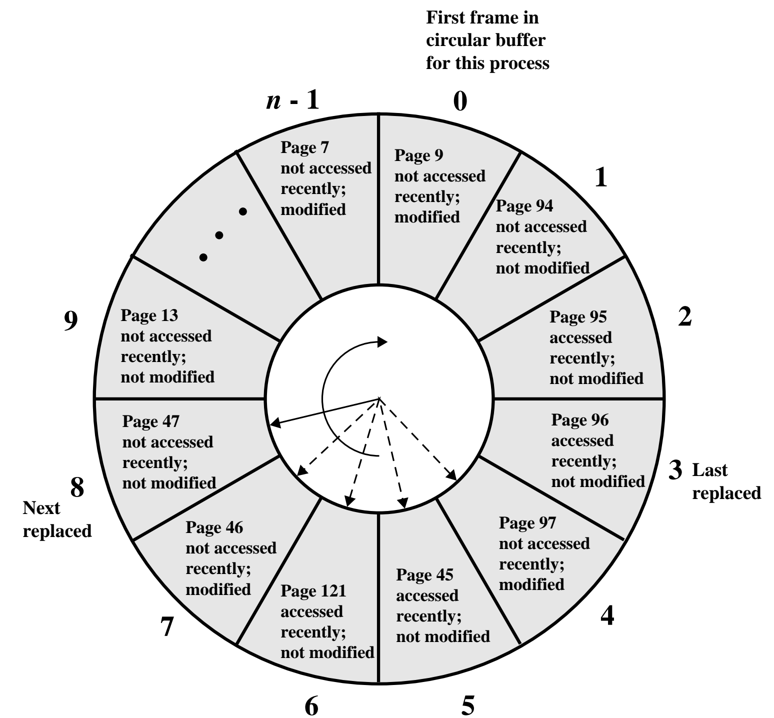


(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

Approssimazioni di LRU: CLOCK migliorato

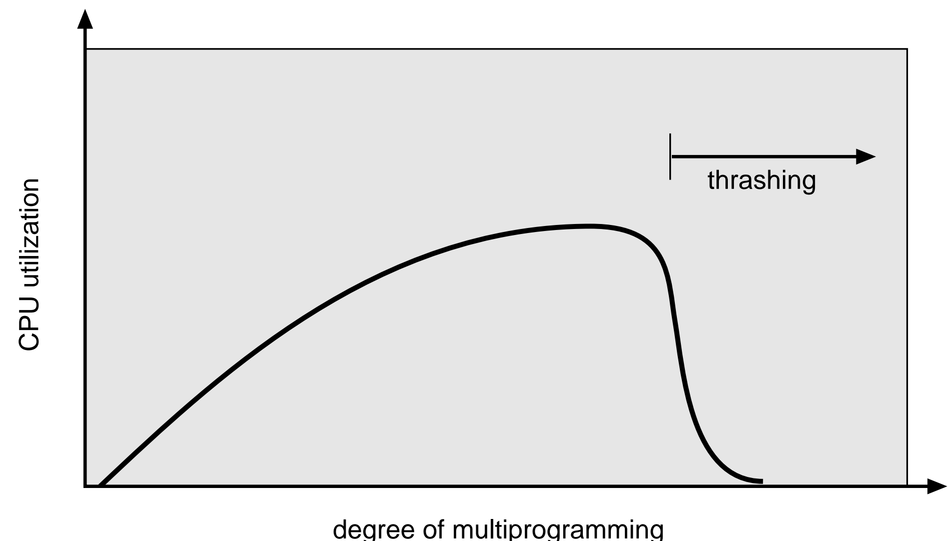
- Usare due bit per pagina: il reference (r) e il dirty (d) bit
  - non usata recentemente, non modificata ($r = 0, d = 0$): buona
  - non usata recentemente, ma modificata ($r = 0, d = 1$): meno buona
  - usata recentemente, non modificata ($r = 1, d = 0$): probabilmente verrà riusata
  - usata recentemente e modificata ($r = 1, d = 1$): molto usata
- si scandisce la coda dei frame più volte
  1. cerca una pagina con (0,0) senza modificare i bit; fine se trovata
  2. cerca una pagina con (0,1) azzerando i reference bit; fine se trovata
  3. vai a 1.
- Usato nel MacOS tradizionale (fino a 9.x)



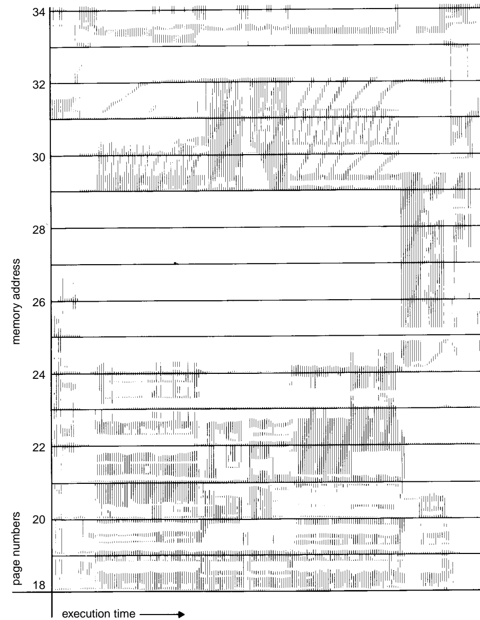
Thrashing

- Se un processo non ha "abbastanza" pagine, il page-fault rate è molto alto. Questo porta a
  - basso utilizzo della CPU (i processi sono impegnati in I/O)
  - il S.O. potrebbe pensare che deve aumentare il grado di multiprogrammazione (errore!)
  - un altro processo viene caricato in memoria
- *Thrashing:* uno o più processi spendono la maggior parte del loro tempo a swappare pagine dentro e fuori
- Il thrashing di un processo avviene quando la memoria assegnatagli è inferiore a quella richiesta dalla sua località

- Il thrashing del sistema avviene quando la memoria fisica è inferiore somma delle località dei processi in esecuzione. Può essere causato da un processo che si espande e in presenza di rimpiazzamento globale.



Principio di località



Ma allora, perché la paginazione funziona? Per il principio di località

- Una *località* è un insieme di pagine che vengono utilizzate attivamente assieme dal processo.
- Il processo, durante l'esecuzione, migra da una località all'altra
- Le località si possono sovrapporre

Impedire il thrashing: modello del working-set

- $\Delta \equiv$ working-set window $\equiv$ un numero fisso di riferimenti a pagine

  Esempio: le pagine a cui hanno fatto riferimento le ultime 10,000 instruzioni

- WSS_i (working set del processo P_i) $=$ numero totale di pagine riferite nell'ultimo periodo Δ. Varia nel tempo.
  - Se Δ è troppo piccolo, il WS non copre l'intera località
  - Se Δ è troppo grande, copre più località
  - Se $\Delta = \infty \Rightarrow$ copre l'intero programma e dati

- $D = \Sigma\ WSS_i \equiv$ totale frame richiesti.

- Sia $m =$ n. di frame fisici disponibile. Se $D > m \Rightarrow$ thrashing.

Algoritmo di allocazione basato sul working set

- il sistema monitorizza il ws di ogni processo, allocandogli frame sufficienti per coprire il suo ws

- alla creazione di un nuovo processo, questo viene ammesso nella coda ready solo se ci sono frame liberi sufficienti per coprire il suo ws

- se $D > m$, allora si sospende uno dei processi per liberare la sua memoria per gli altri (diminuire il grado di multiprogrammazione — scheduling di medio termine)

Si impedisce il thrashing, massimizzando nel contempo l'uso della CPU.

Approssimazione del working set: registri a scorrimento

- Si approssima con un timer e il bit di riferimento

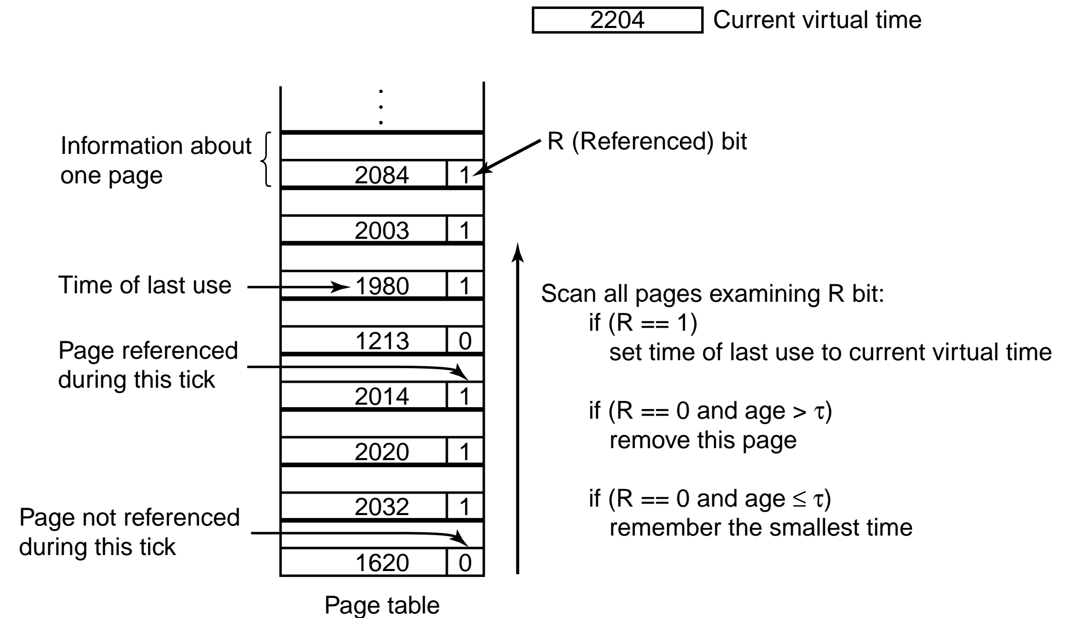
- Esempio: $\Delta = 10000$
  - Si mantengono due bit per ogni pagina (oltre al reference bit)
  - il timer manda un interrupt ogni 5000 unità di tempo
  - Quando arriva l'interrupt, si shifta il reference bit di ogni pagina nei due bit in memoria, e lo si cancella
  - Quando si deve scegliere una vittima: se uno dei tre bit è a 1, allora la pagina è nel working set

- Implementazione non completamente accurata (scarto di 5000 accessi)

- Miglioramento: 10 bit e interrupt ogni 1000 unità di tempo $\Rightarrow$ più preciso ma anche più costoso da gestire

Approssimazione del working set: tempo virtuale

- Si mantiene un *tempo virtuale corrente* del processo (=n, di tick consumati da processo)
- Si eliminano pagine più vecchie di τ tick
- Ad ogni pagina, viene associato un registro contenente il tempo di ultimo riferimento
- Ad un page fault, si controlla la tabella alla ricerca di una vittima.
  - se il reference bit è a 1, si copia il TVC nel registro corrispondente, il reference viene azzerato e la pagina viene saltata
  - se il reference è a 0 e l'età $> \tau$, la pagina viene rimossa
  - se il reference è a 0 e l'età $\leq \tau$, segnati quella più vecchia (con minore tempo di ultimo riferimento). Alla peggio, questa viene cancellata.



Algoritmo di rimpiazzamento WSClock

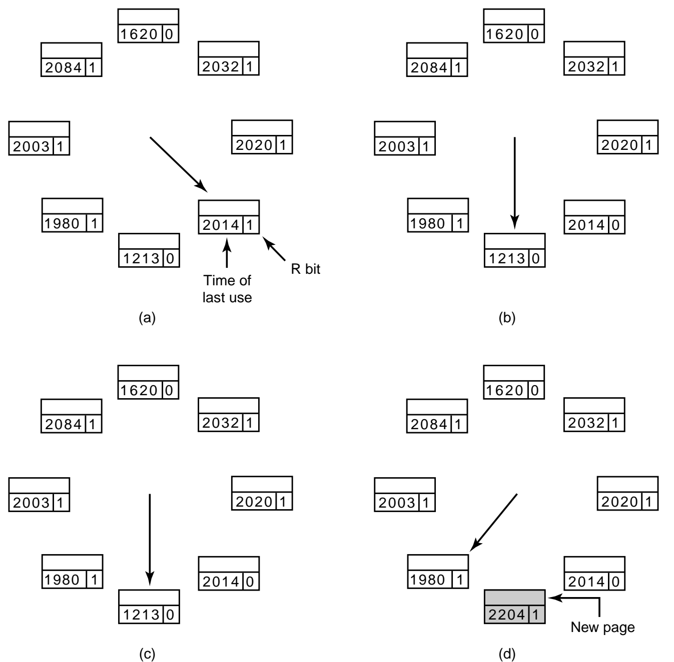
Variante del Clock che tiene conto del Working Set. Invece di contare i riferimenti, si tiene conto di una finestra temporale τ fissata (es. 100ms)

- si mantiene un contatore del tempo di CPU impiegato da ogni processo
- le pagine sono organizzate ad orologio; inizialmente, lista vuota
- ogni entry contiene i reference e dirty bit R, M, e un registro *Time of last use*, che viene copiato dal contatore ad ogni reference. La differenza tra questo registro e il contatore si chiama *età* della pagina.
- ad un page fault, si guarda prima la pagina indicata dal puntatore
  - se $R = 1$, si mette $R = 0$ e si passa avanti
  - se $R = 0$ e età $\leq \tau$: è nel working set: si passa avanti

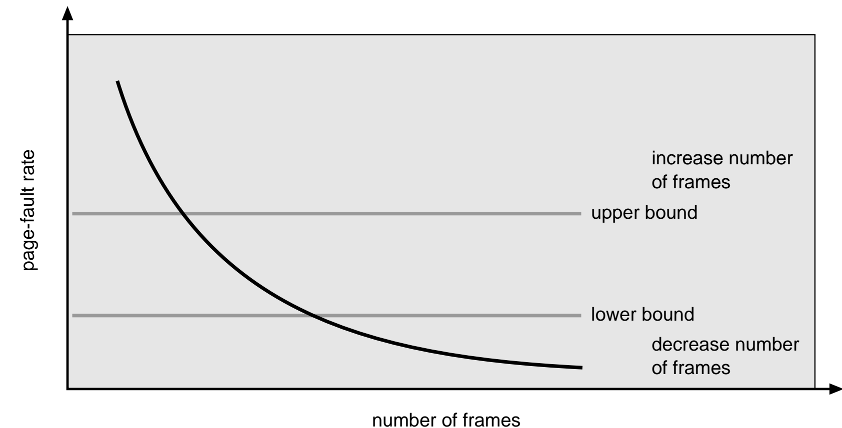
  - se $R = 0$ e età $> \tau$: se $M = 0$ allora si libera la pagina, altrimenti si schedula un pageout e si passa avanti

- Cosa succede se si fa un giro completo?
  - se almeno un pageout è stato schedulato, si continua a girare (aspettando che le pagine schedulate vengano salvate)
  - altrimenti, significa che tutte le pagine sono nel working set. Soluzione semplice: si rimpiazza una quasisiasi pagina pulita.

    Se non ci sono neanche pagine pulite, si rimpiazza la pagina corrente.



Impedire il thrashing: frequenza di page-fault



Si stabilisce un page-fault rate "accettabile"

- Se quello attuale è troppo basso, il processo perde un frame
- Se quello attuale è troppo alto, il processo guadagna un frame

Nota: si controlla solo il n. di frame assegnati, non quali pagine sono caricate.

Sostituzione globale vs. locale

- *Sostituzione locale:* ogni processo può rimpiazzare solo i proprî frame.
  - Mantiene fisso il numero di frame allocati ad un processo (anche se ci sono frame liberi)
  - Il comportamento di un processo non è influenzato da quello degli altri processi
- *Sostituzione globale:* un processo sceglie un frame tra tutti i frame del sistema
  - Un processo può "rubare" un frame ad un altro
  - Sfrutta meglio la memoria fisica
  - il comportamento di un processo dipende da quello degli altri
- Dipende dall'algoritmo di rimpiazzamento scelto: se è basato su un modello di ws, si usa una sostituzione locale, altrimenti globale.

Algoritmi di allocazione dei frame

- Ogni processo necessita di un numero minimo di pagine imposto dall'architettura (Es.: su IBM 370, possono essere necessarie 6 pagine per poter eseguire l'istruzione MOV)

Diversi modi di assegnare i frame ai varî processi

- Allocazione libera: dare a qualsiasi processo quante frame desidera.

  Funziona solo se ci sono sufficienti frame liberi.

- Allocazione equa: stesso numero di frame ad ogni processo

  Porta a sprechi (non tutti i processi hanno le stesse necessità)

- Allocazione proporzionale: un numero di frame in proporzione a
  - dimensione del processo
  - sua priorità (Solitamente, ai page fault si prendono frame ai processi a priorità inferiore)

Esempio: due processi da 10 e 127 pagine, su 62 frame:

$$\frac{10}{127+10} * 62 \cong 4 \qquad \frac{127}{127+10} * 62 \cong 57$$

L'allocazione varia al variare del livello di multiprogrammazione: se arriva un terzo processo da 23 frame:

$$\frac{10}{127+10+23} * 62 \cong 3 \qquad \frac{127}{127+10+23} * 62 \cong 49 \qquad \frac{23}{127+10+23} * 62 \cong 8$$

Buffering di pagine

Aggiungere un insieme (*free list*) di frame liberi agli schemi visti

- il sistema cerca di mantenere sempre un po' di frame sulla free list
- quando si libera un frame,
  - se è stato modificato lo si salva su disco
  - si mette il suo dirty bit a 0
  - si sposta il frame sulla free list *senza cancellarne il contenuto*
- quando un processo produce un page fault
  - si vede se la pagina è per caso ancora sulla free list (*soft page fault*)
  - altrimenti, si prende dalla free list un frame, e vi si carica la pagina richiesta dal disco (*hard page fault*)

Altre considerazioni

- Prepaging: caricare in anticipo le pagine che "probabilmente" verranno usate
  - applicato al lancio dei programmi e al ripristino di processi sottoposti a swapout di medio termine
- Selezione della dimensione della pagina: solitamente imposta dall'architettura. Dimensione tipica: 4K-8K. Influenza
  - frammentazione: meglio piccola
  - dimensioni della page table: meglio grande
  - quantità di I/O: meglio piccola
  - tempo di I/O: meglio grande
  - località: meglio piccola
  - n. di page fault: meglio grande

Altre considerazioni (cont.)

- La struttura del programma può influenzare il page-fault rate
  - Array A[1024,1024] **of** integer
  - Ogni riga è memorizzata in una pagina
  - Un frame a disposizione

Programma 1	Programma 2
for j := 1 to 1024 **do**	**for** i := 1 to 1024 **do**
for i := 1 to 1024 **do**	**for** j := 1 to 1024 **do**
A[i, j] := 0;	A[i, j] := 0;
1024 × 1024 page faults	1024 page faults

- Durante I/O, i frame contenenti i buffer non possono essere swappati
  - I/O solo in memoria di sistema ⇒ costoso
  - Lockare in memoria i frame contenenti buffer di I/O (*I/O interlock*) ⇒ delicato (un frame lockato potrebbe non essere più rilasciato)

Modello della memoria in Unix

- Ogni processo UNIX ha uno spazio indirizzi separato. Non vede le zone di memoria dedicate agli altri processi.
- Come detto più volte un processo UNIX ha tre *segmenti:*
  - **Stack** Stack di attivazione delle subroutine. Cambia dinamicamente.
  - **Data** Contiene i dati inizializzati al caricamento del programma e lo *heap* (memoria allocata dinamicante). Cambia dinamicamente su richiesta esplicita del programma (es., con la **malloc**).
  - **Text** codice eseguibile. In generale: Non modificabile, protetto in scrittura.
- I processi lavorano sullo spazio di indirizzamento *virtuale* (es. $0, \ldots, 2^{32} - 1$ su indirizzi a 32bit)

  - testo e dati: crescono a partire dalla bassa degli indirizzi *virtuali* ($0 \rightarrow$)
  - stack: cresce a partire dalla parte alta degli indirizzi *virtuali* ($\rightarrow 2^{32}$)
- Unix supporta inoltre
  - Condivisione dei segmenti di testo e dati (es dopo una `fork`) con copy-on-write
  - Hardware con spazio indirizzi separato per codice e dati
  - File mappati in memoria principale (ad es. per librerie condivise)

Gestione della memoria in UNIX

- Fino a 3BSD (1978): solo segmentazione con swapping;
- Da 3BSD: segmentazione + paginazione;
- Da 4BSD (1983): segmentazione + paginazione on demand
- Idea principale
  - Un processo in memoria (ready-to-run) ha solo bisogno della u-area e della tabella delle pagine;
  - le pagine di codice, dati e stack sono portate in memoria dinamicamente

Quando si alloca la memoria

- Ulteriore memoria può essere richiesta da un processo in corrispondenza ad uno dei seguenti eventi:
  1. Una fork, che crea un nuovo processo (allocazione di memoria per i segmenti data e stack);
  2. Una brk (e.g., in una `malloc`) che estende un segmento data;
  3. Uno stack che cresce oltre le dimensioni prefissate.
  4. Un accesso in scrittura ad una pagina condivisa tra due processi (*copy-on-write*)

  Oppure, per un processo che era swapped da troppo tempo e che deve essere caricato in memoria.

Gestione della memoria in UNIX Moderni (4BSD, ...)

- Il sistema operativo mantiene residente in memoria principale una tabella con informazioni sul contenuto dei frame della memoria chiamata *core map*
- Attraverso la core map si può gestire una lista di frame liberi (free list)
- Per ogni frame la corrispondente entry della core map mantiene informazioni quali:
  - se il frame è libero: puntatori a prec/succ. frame libero (cella della free list)
  - se il frame è occupato:
    - indirizzo su disco della pagina
    - indice per la tabella dei processi (processo che usa la pagina fisica)
    - puntatore all'inizio del segmento testo/dati/stack e relativo offset
    - informazioni per la paginazione

Gestione della memoria in UNIX

- Le pagine vengono allocate dalla lista libera dal kernel, su base libera, con strategia *copy-on-write* (ad es. per la fork)
- La free list viene mantenuta "piena" entro un certo livello dal demone *pagedaemon* (PID=2) che applica varianti dell'algoritmo dell'orologio
- Lo swapping rimane come soluzione contro il thrashing: uno *scheduler a medio termine* decide il grado di multiprogrammazione.
- Lo swapper (PID=0) si attiva automaticamente, in situazioni estreme

Allocazione di memoria

- Quando un processo viene lanciato, molte pagine vengono precaricate e poste sulla free list (prepaging)
- Quando un processo termina, le sue pagine vengono messe sulla free list
- Il sistema usa allocazione libera per gestire la richiesta di pagine da parte di un processo
- Tuttavia se la free list scende sotto un certo livello *minfree* fissato dal sistema, il kernel si rifiuta di allocare nuove pagine di memoria. Tipicamente,*minfree* è un valore tra 100K e 5M.
- La free list viene anche utilizzata come *memoria cache*: le pagine richieste da un processo che risultano *invalide* vengono cercate sulla free list, prima di essere caricate da disco
- Quando un processo termina, le sue pagine vengono messe sulla free list

Pagedaemon

- *Pagedaemon* viene lanciato al boot e si attiva ad intervalli regolari (tip. 2–4 volte al secondo) o su richiesta del kernel
- Il pagedaemon interviene quando il numero di frame liberi (NFL) è minore di *lotsfree* un parametro che denota il numero minimo ammessibile di frame liberi (solitamente 1/4 dei frame totali), altrimenti torna inattivo
- Nel primo caso ($NFL < lostfree$) inizia il trasferimento di pagine fino a che non ci sono *lotsfree* pagine libere
- Pagedaemon utilizza una politica di rimpiazzamento *globale* (non si guarda il processo a cui appartiene la pagina) e una variante dell'algoritmo dell'orologio: l'orologio (CLOCK) a 2 lancette

CLOCK a due lancette

- L'algoritmo usa due puntatori (lancette) per scorrere la lista delle pagine allocate nella *core map*.
- Fino a che $NFL <$ *lotsfree* si eseguono i seguenti passi:
  - la prima lancetta azzera il reference bit della pagina a cui punta correntemente
  - la seconda sceglie la pagina vittima:
    * se trova $r = 0$ (i.e. la pagina non è stata usata nel periodo trascorso tra il passaggio delle due lancette):
      - se dirty-bit=1 il frame viene salvato su disco
      - il frame viene aggiunto alla free-list
    * poi si fanno avanzare i due puntatori

CLOCK a due lancette (cont.)

- La distanza tra i puntatori (*handspread*) viene decisa al boot, per liberare frame abbastanza rapidamente
- Se le lancette sono vicine: solo le pagine realmente usate rimarranno in memoria
- Se le lancette sono distanti 359° = algoritmo dell'orologio (la seconda passa dopo un giro)
- Possibile variante:
  - ulteriore parametro *maxfree* > *lotsfree*
  - quando il livello di pagine scende sotto *lotsfree*, il pagedaemon libera pagine fino a raggiungere *maxfree*

  Permette di evitare una potenziale instabilità del CLOCK a due lancette.

Swapping di processi

- Interi processi possono essere sospesi (*disattivati*) temporaneamente, per abbassare la multiprogrammazione
- Il processo *swapper* o *sched* (PID=0, lanciato al boot) decide quale processo deve essere swappato su disco
- Parametro: *desfree*, impostato al boot. 100K–10M.
  minfree < *desfree* < *lotsfree*
- Lo swapper si sveglia ogni 1–2 secondi, e interviene solo se
  NFL < *minfree* e
  NFL < *desfree* nella storia recente

Swapping: chi esce...

- Regole di scelta del processo vittima:
  - si cerca tra i processi in "wait", senza considerare quelli in memoria da meno di 2 secondi
  - se ce ne sono, si prende quello con il valore *priorità+tempo di residenza in memoria* più alto (nota: tempo CPU≠ tempo in memoria)
  - Altrimenti, si cerca tra quelli in "ready", con lo stesso criterio
- Per il processo selezionato:
  - i suoi segmenti data e stack (non il text) vengono scaricati sul device di swap; i frame vengono aggiunti alla lista dei frame liberi
  - Nel PCB, viene messo lo stato "swapped" e agganciato alla lista dei processi swappati
- Si ripete fino a che sufficiente memoria viene liberata.

Swapping: . . . e chi entra

Quando *swapper* si sveglia da sé:

1. cerca nella lista dei PCB dei processi swappati e ready, il processo swappato da più tempo, ma almeno 2 secondi (per evitare thrashing);
2. se lo trova, determina se c'è sufficiente memoria libera per la page table e l'area-u (*easy swap*) oppure no (*hard swap*);
3. se è un hard swap, libera memoria swappando qualche altro processo;
4. carica le page table e l'area-u in memoria e mette il processo in "Ready-to-run, in memory"

Si ripete finché non ci sono processi da caricare.

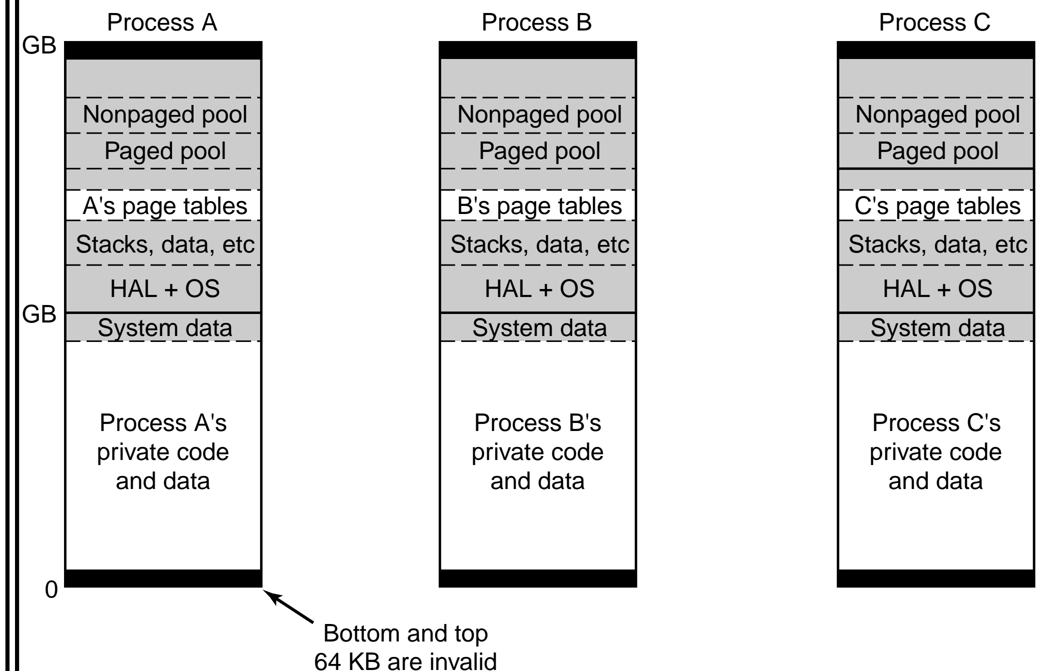
Considerazioni

Interazione tra scheduling a breve termine, a medio termine e paginazione

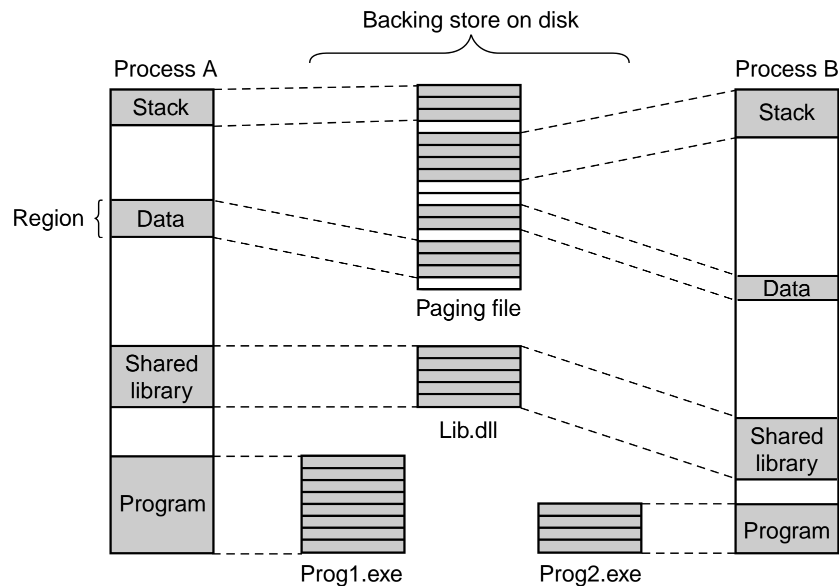
- minore è la priorità, maggiore è la probabilità che il processo venga swappato
- per ogni processo in esecuzione, la paginazione tende a mantenere in memoria il suo working set
- quindi, processi che non sono idle tendono a stare in memoria, mentre si tende a swappare solo processi idle da molto tempo
- nel complesso, il sistema massimizza l'utilizzo della memoria e la multiprogrammazione, limitando il thrashing e garantendo l'assenza di starvation per i processi swappati

Modello della memoria in Windows 2000

- Ogni processo utente di Windows riceve uno spazio indirizzi virtuali di 4G, diviso in due parti
  - codice e dati, nella parte bassa (<2G) Liberamente accessibile.
  - kernel, HAL (strato di astrazione hardware) e strutture di sistema (comprese le page tables) nella parte alta La maggior parte di questo spazio non è accessibile, neanche in lettura.
  - i primi ed ultimi 64kb sono invalidi per individuare rapidamente errori di programmazione (puntatori a 0 e -1).
- La memoria è puramente paginata (senza prepaging), con copy-on-write.
- Il caricamento dei segmenti è basato fortemente sul memory mapping.
- Mappare il kernel nello spazio dei processi utenti aumenta l'efficienza delle chiamate di sistema: un thread che passa in modo kernel non deve cambiare tabelle per gestire la rilocazione



Mappatura dei segmenti in memoria reale e su file



Gestione della paginazione

Nessuna forma di prepaging: tutte le pagine vengono caricate su page fault.

Le pagine possono essere in diversi *stati* dai quali dipendono le sorti di un page fault

Windows utilizza la strategia del working set per gestire la paginazione

Stati di una pagina

- *Available*: pagina non usata da nessun processo. Si divide in tre possibilità:
  - Free: riusabile
  - Standby: rimossa da un ws ma richiamabile (buffering)
  - Zeroed: riusabile e in più tutta azzerata (si cancellano i dati per ragioni di sicurezza)
- *Reserved*: riservata da un processo ma non ancora usata (ad es. spazio per lo stack di un thread). Non fa parte del ws fino a che non viene veramente usata.
- *Committed*: usata da un processo e associata ad un blocco su disco (*mappata*): la pagina fisica potrebbe non essere in memoria

Casi di page fault

- La pagina riferita non è committed
  $\Rightarrow$ Terminazione del processo
- Violazione di protezione
  $\Rightarrow$ Terminazione del processo
- Scrittura su una pagina condivisa
  $\Rightarrow$ Copy-on-write su una pagina reserved
- Crescita dello stack
  $\Rightarrow$ Allocazione di una pagina azzerata
- La pagina riferita è committed (ha un indirizzo fisico associato) ma non attualmente caricata in memoria
  $\Rightarrow$ il vero page fault: pagein della pagina mancante

Algoritmo di rimpiazzamento di pagina

La paginazione è basata sul modello del Working Set per i *processi*

- ogni processo ha un working set (insieme delle pagine mappate in memoria) e una dim. minima min e massima max (si può scendere sotto min e salire sopra max)
- Tutti i processi iniziano con lo stesso min e max (risp. 20-50 e 45-345, in proporzione alla RAM; modificabile dall'amministratore del sistema)
- Ad un page fault (ws=dimensione del working set)
  - se $ws < max$, la pagina viene allocata ed aggiunta al working set.
  - se $ws > max$, una pagina vittima viene scelta nel working set (politica di rimpiazzamento *locale*)
- I limiti possono cambiare nel tempo: se un processo sta paginando troppo (thrashing locale), il suo max viene aumentato
- vengono mantenute sempre libere almeno 512 pagine

Le pagine allocate vengono prelevate dalla *free list*:

- ogni secondo parte un thread del kernel (*balance set manager*)
- se la free list è troppo corta, parte il *working set manager* che esamina i working sets per liberare pagine
  - prima i processi più grandi e idle da più tempo; il processo in foreground è considerato per ultimo
  - se un processo ha $ws < min$ o ha avuto molti page fault recentemente, viene saltato
  - altrimenti una o più pagine vengono rimosse
- si ripete sempre più aggressivamente finché la free list ritorna accettabile
- anche parte del kernel può essere paginata
- Eventualmente un ws può scendere sotto il min
- non esiste completo swapout di processi

Gestione della memoria fisica

- I frame liberi sono organizzati in diverse liste mantenute dal sistema operativo
- Windows 2000 utilizza strategie ed euristiche complesse (che non vedremo) per gestire la scelta della pagina fisica da allocare per una richiesta
- Le liste sono 5:
  1. Pagine modificate: pagine eliminate da un ws, associate ad un processo, ma che devono essere copiate su disco
  2. Pagine in attesa (standby): eliminate da un ws, assoc. ad un processo, consistenti con immagine su disco
  3. Pagine libere: non sono più asociate ad alcun processo

  4. Azzerate (zeroed): libere e con contenuto azzerato

- le pagine nelle liste 1 e 2 si possono recuperare quando il processo corrispondente le richiede
- la lista 3 contiene pagine di processi che hanno terminato l'esecuzione
- Le pagine possono cambiare lista:
  - dei demoni di scrittura di pagine mappate/modificate spostano pagine da 1 a 2
  - come effetto di una deallocazion una pagina può andare da 2 a 3
  - il demone *zero page thread* (gira a priorità più bassa) sposta pagine da 3 a 4: se la CPU è inattiva vengono azzerate delle pagine (più utili di pagine sporche)

Il File System

Alcune necessità dei processi:

- Memorizzare e trattare grandi quantità di informazioni (> memoria principale)
- Più processi devono avere la possibilità di accedere alle informazioni in modo concorrente e coerente, nello spazio e nel tempo
- Si deve garantire integrità, indipendenza, persistenza e protezione dei dati

L'accesso diretto ai dispositivi di memorizzazione di massa (come visto nella gestione dell'I/O) non è sufficiente.

I File

La soluzione sono i *file* (archivi):

- File = insieme di informazioni correlate a cui è stato assegnato un nome
- Un file è la più piccola porzione unitaria di memoria logica secondaria allocabile dall'utente o dai processi di sistema.
- La parte del S.O. che realizza questa astrazione, nascondendo i dettagli implementativi legati ai dispositivi sottostanti, è il *file system*.
- Internamente, il file system si appoggia alla gestione dell'I/O per implementare ulteriori funzionalità.
- Esternamente, il file system è spesso l'aspetto più visibile di un S.O. (S.O. *documentocentrici*): come si denominano, manipolano, accedono, quali sono le loro strutture, i loro attributi, etc.

Attributi dei file (metadata)

Nome identificatore del file. L'unica informazione umanamente leggibile

Tipo nei sistemi che supportano più tipi di file. Può far parte del nome

Locazione puntatore alla posizione del file sui dispositivi di memorizzazione

Dimensioni attuale, ed eventualmente massima consentita

Protezioni controllano chi può leggere, modificare, creare, eseguire il file

Identificatori dell'utente che ha creato/possiede il file

Varie date e timestamp di creazione, modifica, aggiornamento informazioni. . .

Queste informazioni (*metadati*: dati sui dati) sono solitamente mantenute in apposite strutture (*directory*) residenti in memoria secondaria.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Denominazione dei file

- I file sono un meccanismo di astrazione, quindi ogni oggetto deve essere denominato.
- Il *nome* viene associato al file dall'utente, ed è solitamente necessario (ma non sufficiente) per accedere ai dati del file
- Le regole per denominare i file sono fissate dal file system, e sono molto variabili
  - lunghezza: fino a 8, a 32, a 255 caratteri
  - tipo di caratteri: solo alfanumerici o anche speciali; e da quale set? ASCII, ISO-qualcosa, Unicode?
  - case sensitive, insensitive
  - contengono altri metadati? ad esempio, il tipo?

Tipi dei file — FAT: name.extension

Tipo	Estensione	Funzione
Eseguibile	exe, com, bin o nessuno	programma pronto da eseguire, in liguaggio macchina
Oggetto	obj, o	compilato, in linguaggio macchina, non linkato
Codice sorgente	c, p, pas, f77, asm, java	codice sorgente in diversi linguaggi
Batch	bat, sh	script per l'interprete comandi
Testo	txt, doc	documenti, testo
Word processor	wp, tex, doc	svariati formati
Librarie	lib, a, so, dll	librerie di routine
Grafica	ps, dvi, gif	FILE ASCII o binari
Archivi	arc, zip, tar	file correlati, raggruppati in un file, a volte compressi

Tipi dei file — Unix: nessuna assunzione

Unix non forza nessun tipo di file a livello di sistema operativo: non ci sono metadati che mantengono questa informazione.

Tipo e contenuto di un file slegati dal nome o dai permessi.

Sono le applicazioni a sapere di cosa fare per ogni file (ad esempio, i client di posta usano i MIME-TYPES).

È possibile spesso "indovinare" il tipo ispezionando il contenuto (e.g. i magic numbers: informazioni memorizzate all'inizio di un file) attraverso programmi di sistema come `file`

```
$ file iptables.sh risultati Lucidi
iptables.sh: Bourne shell script text executable
risultati:   ASCII text
Lucidi:      PDF document, version 1.2
p.dvi:       TeX DVI file (TeX output 2003.09.30:1337)
```


Struttura dei file

- In genere, un file è una sequenza di bit, byte, linee o record il cui significato è assegnato dal creatore.
- A seconda del tipo, i file possono avere struttura
  - nessuna: sequenza di parole, byte
  - sequenza di record: linee, blocchi di lunghezza fissa/variabile
  - strutture più complesse: documenti formattati, archivi (ad albero, con chiavi, ...)
  - I file strutturati possono essere implementati con quelli non strutturati, inserendo appropriati caratteri di controllo
- Chi impone la struttura: due possibilità
  - il sistema operativo: specificato il tipo, viene imposta la struttura e modalità di accesso. Più astratto.
  - l'utente: tipo e struttura sono delegati al programma, il sistema operativo implementa solo file non strutturati. Più flessibile.

Operazioni sui file

Creazione: due passaggi: allocazione dello spazio sul dispositivo, e collegamento di tale spazio al file system

Cancellazione: staccare il file dal file system e deallocare lo spazio assegnato al file

Apertura: caricare alcuni metadati dal disco nella memoria pricipale, per velocizzare le chiamate seguenti

Chiusura: deallocare le strutture allocate nell'apertura

Lettura: dato un file e un *puntatore di posizione*, i dati da leggere vengono trasferiti dal *media* in un buffer in memoria

Scrittura: dato un file e un *puntatore di posizione*, i dati da scrivere vengono trasferiti sul *media*

Append: versione particolare di scrittura

Riposizionamento (seek): non comporta operazioni di I/O

Troncamento: azzerare la lunghezza di un file, mantenendo tutti gli altri attributi

Lettura dei metadati: leggere le informazioni come nome, timestamp, etc.

Scrittura dei metadati: modificare informazioni come nome, timestamps, protezione, etc.

Tabella dei file aperti

Queste operazioni richiedono la conoscenza delle informazioni contenute nelle directory. Per evitare di accedere continuamente alle dir, si mantiene in memoria una *tabella dei file aperti*. Due nuove operazioni sui file:

- Apertura: allocazione di una struttura in memoria (*file descriptor* o *file control block*) contenente le informazioni riguardo un file
- Chiusura: trasferimento di ogni dato in memoria al dispositivo, e deallocazione del file descriptor

A ciascun file aperto si associa

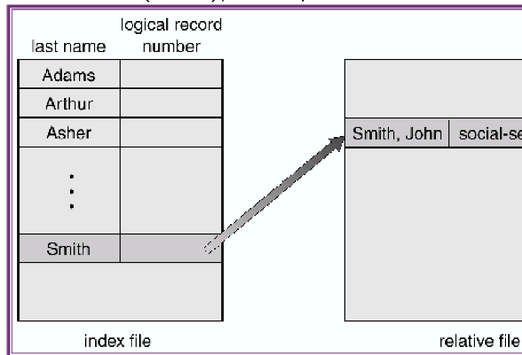
- Puntatore al file: posizione raggiunta durante la lettura/scrittura
- Contatore dei file aperti: quanti processi stanno utilizzando il file
- Posizione sul disco

Metodi di accesso

- Accesso sequenziale
  - Un puntatore mantiene la posizione corrente di lettura/scrittura
  - Si può accedere solo progressivamente, o riportare il puntatore all'inizio del file.
  - Adatto a dispositivi intrinsecamente sequenziali (p.e., nastri)
- Accesso diretto
  - Il puntatore può essere spostato in qualunque punto del file
  - L'accesso sequenziale viene simulato con l'accesso diretto
  - Usuale per i file residenti su device a blocchi (p.e., dischi)

Metodi di accesso: accesso indicizzato

- Un secondo file contiene solo parte dei dati, e puntatori ai blocchi (record) del vero file
- La ricerca avviene prima sull'indice (corto), e da qui si risale al blocco

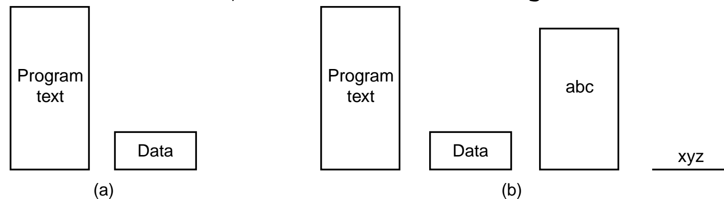


- Implementabile a livello applicazione in termini di file ad accesso diretto
- Usuale su mainframe (IBM, VMS), databases. . .


```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                    /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);         /* ANSI prototype */

#define BUF_SIZE 4096                     /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                  /* protection bits for output file */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);              /* syntax error if argc is not 3 */

     /* Open the input file and create the output file */
     in_fd = open(argv[1], O_RDONLY);     /* open the source file */
     if (in_fd < 0) exit(2);              /* if it cannot be opened, exit */
     out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
     if (out_fd < 0) exit(3);             /* if it cannot be created, exit */

     /* Copy loop */
     while (TRUE) {
          rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
     if (rd_count <= 0) break;            /* if end of file or error, exit loop */
          wt_count = write(out_fd, buffer, rd_count); /* write data */
          if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
     }

     /* Close the files */
     close(in_fd);
     close(out_fd);
     if (rd_count == 0)                   /* no error on last read */
          exit(0);
     else
          exit(5);                        /* error on last read */
}
```


File mappati in memoria

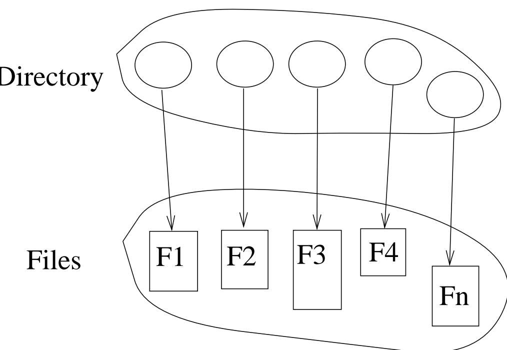
- Semplificano l'accesso ai file, rendendoli simili alla gestione della memoria.



- Relativamente semplice da implementare in sistemi segmentati (con o senza paginazione): il file viene visto come area di swap per il segmento mappato
- Non servono chiamate di sistema `read` e `write`, solo una `mmap`
- Problemi
  - lunghezza del file non nota al sistema operativo
  - accesso condiviso con modalità diverse
  - lunghezza del file maggiore rispetto alla dimensione massima dei segmenti.

Directory

- Una directory è una collezione di nodi contenente informazioni sui file (*metadati*)
- Sia la directory che i file risiedono su disco
- Operazioni su una directory
  - Ricerca di un file
  - Creazione di un file
  - Cancellazione di un file
  - Listing
  - Rinomina di un file
  - Navigazione del file system



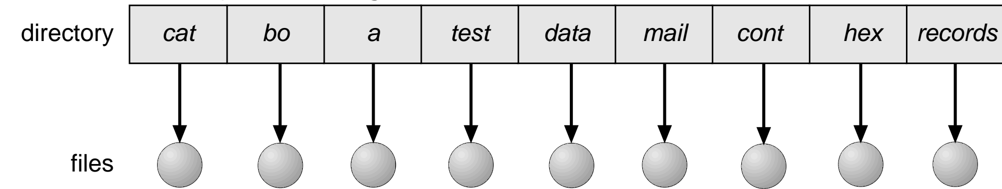
Organizzazione logica delle directory

Le directory devono essere organizzate per ottenere

- efficienza: localizzare rapidamente i file
- nomi mnemonici: comodi per l'utente
  - file differenti possono avere lo stesso nome
  - più nomi possono essere dati allo stesso file
- Raggruppamento: file logicamente collegati devono essere raccolti assieme (e.g., i programmi in C, i giochi, i file di un database, ...)

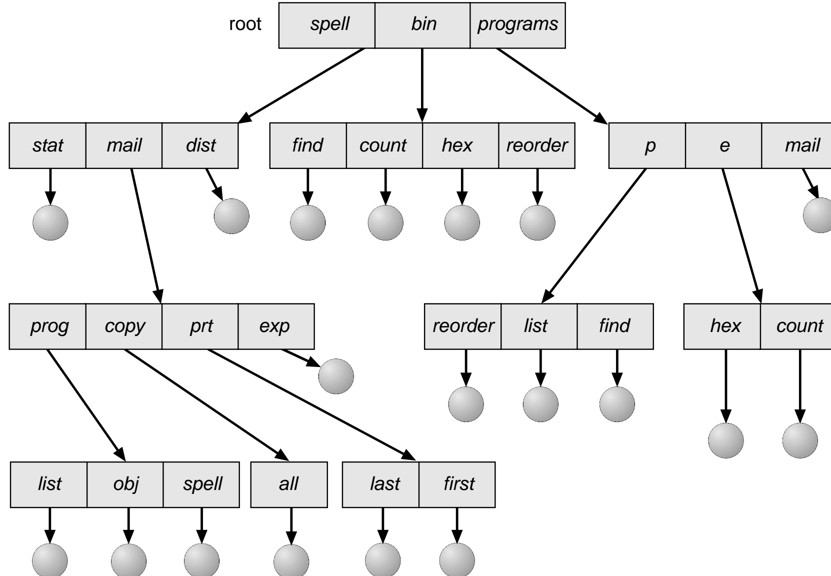
Tipi di directory: unica ("flat")

- Una sola directory per tutti gli utenti



- Problema di raggruppamento e denominazione
- Obsoleta
- Variante: a due livelli (una directory per ogni utente)

Tipi di directory: ad albero



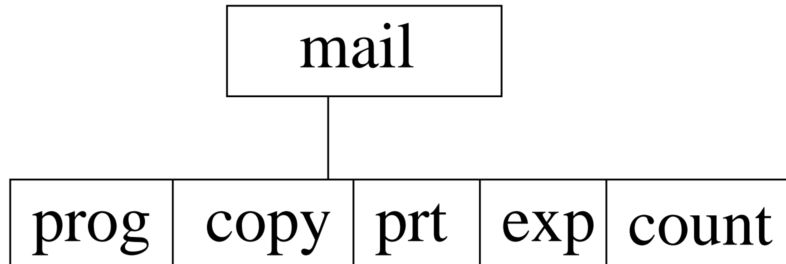
Directory ad albero (cont.)

- Ricerca efficiente
- Raggruppamento
- Directory corrente (working directory): proprietà del processo
  - **cd** /home/miculan/src/C
  - **cat** hw.c
- Nomi assoluti o relativi
- Le operazioni su file e directory (lettura, creazione, cancellazione, ...) sono relative alla directory corrente

Esempio: se la dir corrente è `/spell/mail`

mkdir count

crea la situazione corrente

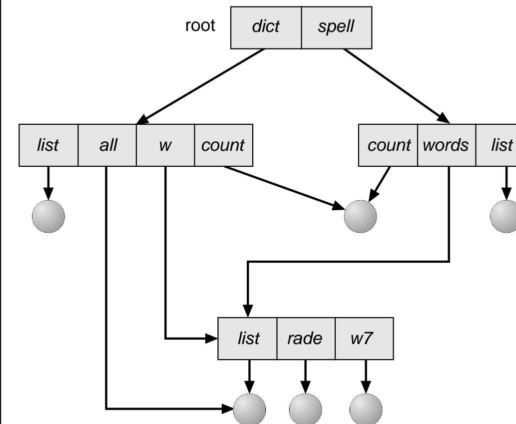


- Cancellando `mail` si cancella l'intero sottoalbero

Directory a grafo aciclico (DAG)

File e sottodirectory possono essere condivise da più directory

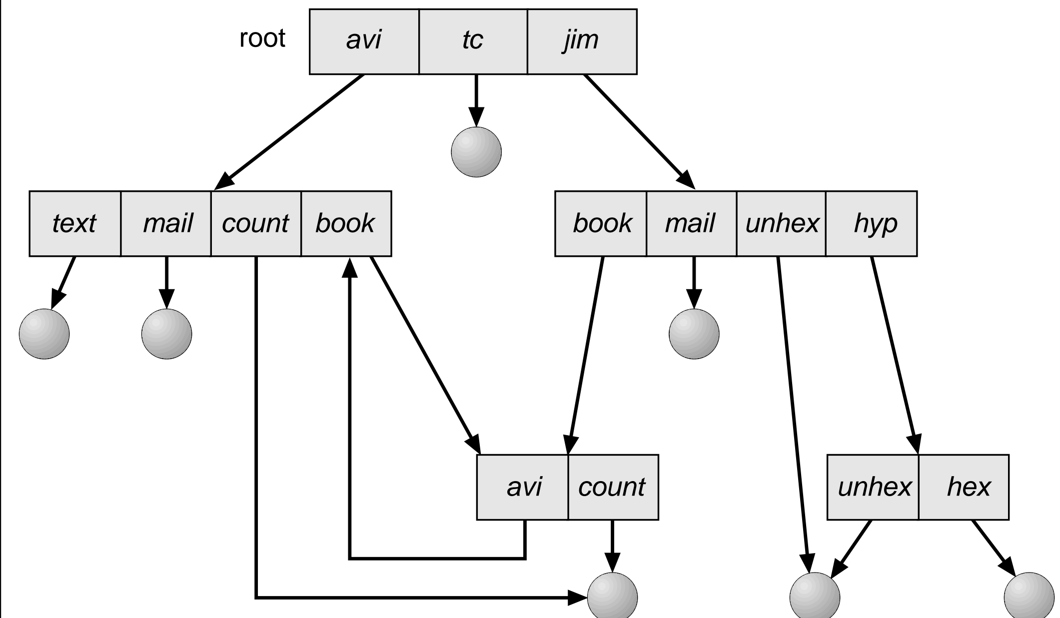
Due nomi differenti per lo stesso file (aliasing)



Problemi con directory DAG

- Possibilità di puntatori "dangling".
- Soluzioni
  - Puntatori all'indietro, per cancellare tutti i riferimenti ad un file da rimuovere. Problematici perché la dimensione dei record nelle directory diventa variabile (dipende dal numero di riferimenti).
  - Contatori di riferimenti per ogni file (UNIX)

Directory a grafo



I cicli sono problematici per la

- Visita: algoritmi costosi per evitare loop infiniti
- Cancellazione: creazione di *garbage*

Soluzioni:

- Permettere solo link a file (UNIX per i link hard)
- Durante la navigazione, limitare il numero di link attraversabili (UNIX per i simbolici)
- Garbage collection (costosa!)
- Ogni volta che un link viene aggiunto, si verifica l'assenza di cicli (Costoso).

Protezione

- Importante in ambienti multiuser dove si vuole condividere file
- Il creatore/possessore (non sempre coincidono) deve essere in grado di controllare
  - cosa può essere fatto
  - e da chi (in un sistema multiutente)
- Tipi di accesso soggetti a controllo (non sempre tutti supportati):
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

Matrice di accesso

Sono il metodo di protezione più generale

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Matrice di accesso (cont.)

- per ogni coppia (processo,oggetto), associa le operazioni permesse
- matrice molto sparsa: si implementa come
  - *access control list:* ad ogni oggetto, si associa chi può fare cosa.
  - *capability tickets:* ad ogni processo, si associa un insieme di tokens che indicano cosa può fare

Modi di accesso e gruppi in UNIX

Versione semplificata di access control list.

- Tre modi di accesso: **r**ead, **w**rite, **e**xecute
- Tre classi di utenti, per ogni file

				RWX
a)	owner access	7	$\Rightarrow$	1 1 1
b)	groups access	6	$\Rightarrow$	1 1 0
c)	public access	1	$\Rightarrow$	0 0 1

- Ogni processo possiede user identifier (UID) e group identifier (GID), con i quali si verifica l'accesso

Modi di accesso e gruppi in UNIX

- Per limitare l'accesso ad un gruppo di utenti, si chiede al sistemista di creare un gruppo apposito, sia G, e di aggiungervi gli utenti.
- Si definisce il modo di accesso al file o directory
- Si assegna il gruppo al file:
  chgrp G *game*

Effective User e Group ID

- In UNIX, il dominio di protezione di un processo viene ereditato dai suoi figli, e viene impostato al login
- In questo modo, tutti i processi di un utente girano con il suo UID e GID.
- Può essere necessario, a volte, concedere temporaneamente privilegi speciali ad un utente (es: *ps*, *lpr*, ...)
  - *Effective* UID e GID (EUID, EGID): due proprietà extra di tutti i processi (stanno nella U-structure).
  - Tutti i controlli vengono fatti rispetto a EUID e EGID
  - Normalmente, EUID=UID e EGID=GID
  - L'utente `root` può cambiare questi parametri con le system call *setuid(2), setgid(2), seteuid(2), setegid(2)*

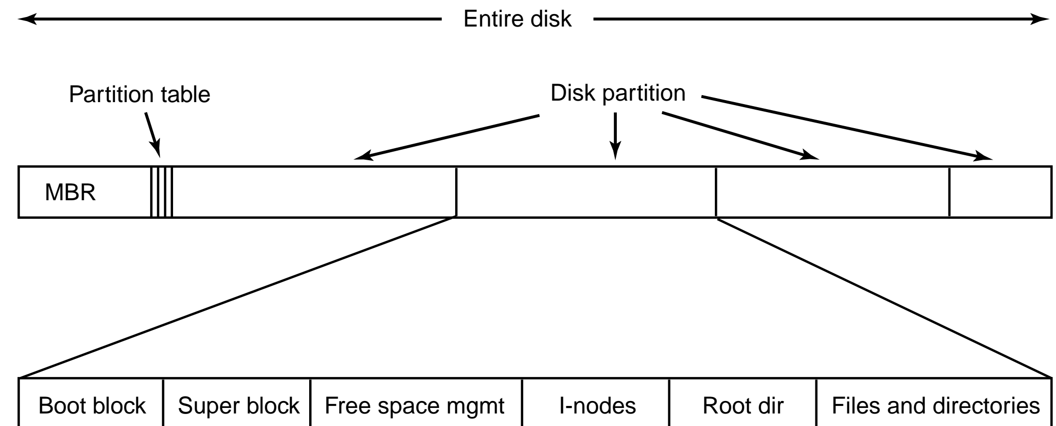
Setuid/setgid bit

- l'Effective UID e GID di un processo possono essere cambiati per la durata della sua esecuzione attraverso i bit **setuid** e **setgid**
- Sono dei bit supplementari dei file *eseguibili* di UNIX
- Se **setuid** bit è attivo, l'EUID di un processo che esegue tale programma diventa lo stesso del possessore del file
- Se **setgid** bit è attivo, l'EGID di un processo che esegue tale programma diventa lo stesso del possessore del file
- I real UID e GID rimangono inalterati

Implementazione del File System

- Il supporto utilizzato più frequentemente per la memorizzazione persistente di dati e' il disco
- Lo spazio disco viene solitamente suddiviso in *partizioni* e *blocchi* (tipicamente 512 byte)
- L'implementazione del file system deve preoccuparsi di come allocare i blocchi del disco, come organizzare i dati riepilogativi delle directory, e cosìvia.

Esempio di layout di un disco fisico



Struttura dei file system

programmi di applicazioni: applicativi ma anche comandi *ls*, *dir*, . . .

file system logico: presenta i diversi file system come un unica struttura; implementa i controlli di protezione

organizzazione dei file: controlla l'allocazione dei blocchi fisici e la loro corrispondenza con quelli logici. Effettua la traduzione da indirizzi logici a fisici.

file system di base: usa i driver per accedere ai blocchi fisici sull'appropriato dispositivo.

controllo dell'I/O: i driver dei dispositivi

dispositivi: i controller hardware dei dischi, nastri, etc.

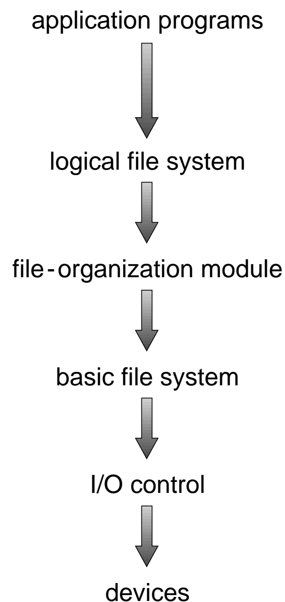


Tabella dei file aperti

- Per accedere ad un file è necessario conoscere informazioni riguardo la sua posizione, protezione, . . .
- questi dati sono accessibili attraverso le directory
- per evitare continui accessi al disco, si mantiene in memoria una *tabella dei file aperti*. Ogni elemento descrive un file aperto (*file control block*)
  - Alla prima open, si caricano in memoria i metadati relativi al file aperto
  - Ogni operazione viene effettuata riferendosi al file control block in memoria
  - Quando il file viene chiuso da tutti i processi che vi accedevano, le informazioni vengono copiate su disco e il blocco deallocato
- Problemi di affidabilità (e.g., se manca la corrente. . .)

Mounting dei file system

- Ogni file system fisico, prima di essere utilizzabile, deve essere *montato* nel file system logico
- Il montaggio può avvenire
  - al boot, secondo regole implicite o configurabili
  - dinamicamente: supporti rimovibili, remoti, ...
- Il punto di montaggio può essere
  - fissato (A:, C:, ... sotto Windows)
  - configurabile in qualsiasi punto del file system logico (Unix)
- Il kernel esamina il file system fisico per riconoscerne la struttura e tipo
- Prima di spegnere o rimuovere il media, il file system deve essere *smontato* (pena gravi inconsistenze!)

Allocazione contigua

Ogni file occupa un insieme di blocchi contigui sul disco

- Semplice: basta conoscere il blocco iniziale e la lunghezza
- L'accesso random è facile da implementare
- Frammentazione esterna. Problema di allocazione dinamica.
- I file non possono crescere (a meno di deframmentazione)
- Frammentazione interna se i file devono allocare tutto lo spazio che gli può servire a priori

- Traduzione dall'indirizzo logico a quello fisico (per blocchi da 512 byte):
- Se LA=indirizzo logico

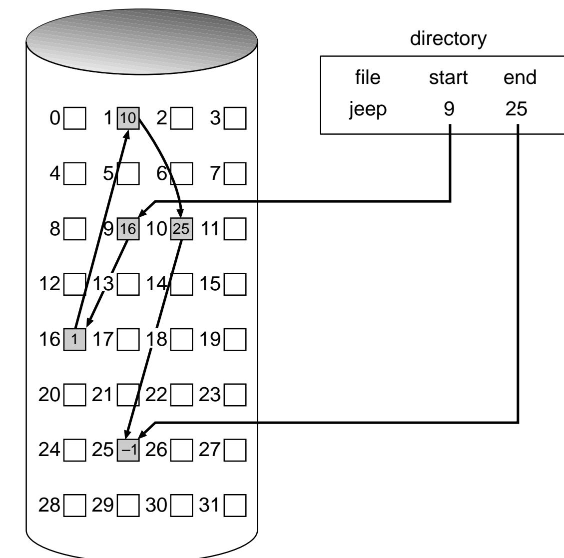
$$LA/512 \begin{cases} Q \\ R \end{cases}$$

e Q=quoziente, R=resto allora

- Il blocco da accedere = Q + blocco di partenza
- Offset all'interno del blocco = R

Allocazione concatenata

Ogni file è una linked list di blocchi, che possono essere sparpagliati ovunque sul disco



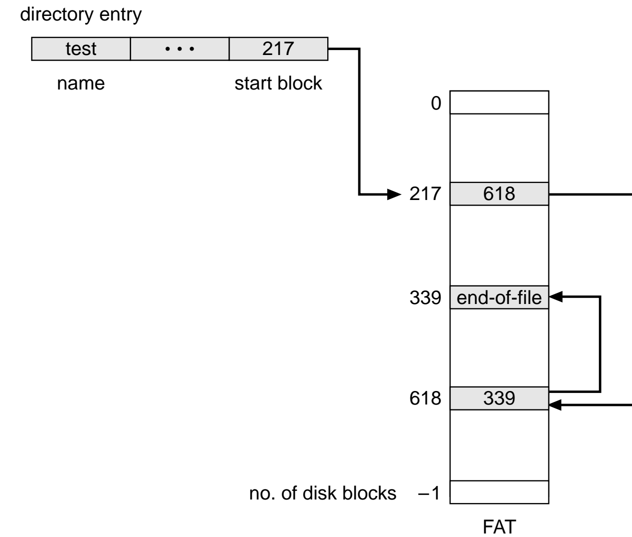
- Allocazione su richiesta; i blocchi vengono semplicemente collegati alla fine del file
- Semplice: basta sapere l'indirizzo del primo blocco
- Non c'è frammentazione esterna
- Bisogna gestire i blocchi liberi
- Non supporta l'accesso diretto
- Traduzione indirizzo logico (1 byte per il puntatore):

$$\text{LA}/511 \begin{cases} Q \\ R \end{cases}$$

  - Il blocco da accedere è il Q-esimo della lista
  - Offset nel blocco = R + 1

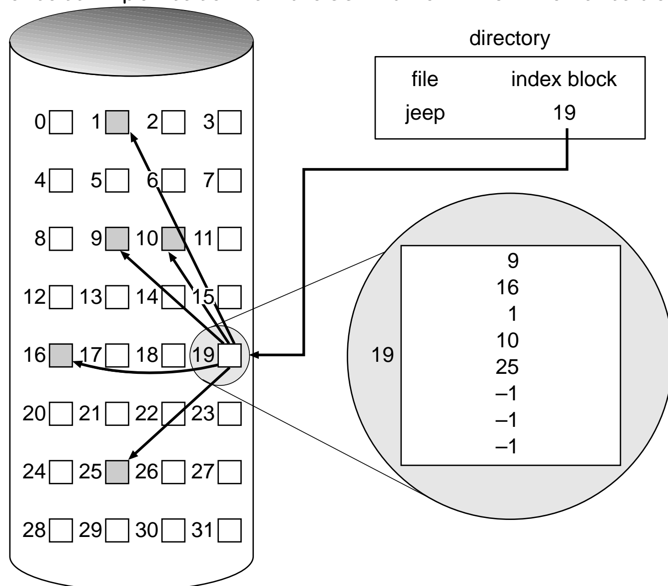
Allocazione concatenata (cont.)

Variante: *File-allocation table (FAT)* di MS-DOS e Windows. Mantiene la linked list in una struttura dedicata, all'inizio di ogni *partizione* del disco



Allocazione indicizzata

Si mantengono tutti i puntatori ai blocchi di un file in una *tabella indice*.



- Supporta accesso random
- Allocazione dinamica senza frammentazione esterna
- Traduzione: file di max 256K word e blocchi di 512 word: serve 1 blocco per l'indice (512x512=262144=256K)

$$\text{LA}/512 \begin{cases} Q \\ R \end{cases}$$

  - Q = offset nell'indice
  - R = offset nel blocco indicato dall'indice

Allocazione indicizzata (cont.)

- Problema: come implementare il blocco indice
  - è una struttura supplementare: overhead ⇒ meglio piccolo
  - dobbiamo supportare anche file di grandi dimensioni ⇒ meglio grande
- Indice concatenato: l'indice è composto da blocchi concatenati. Nessun limite sulla lunghezza, maggiore costo di accesso.

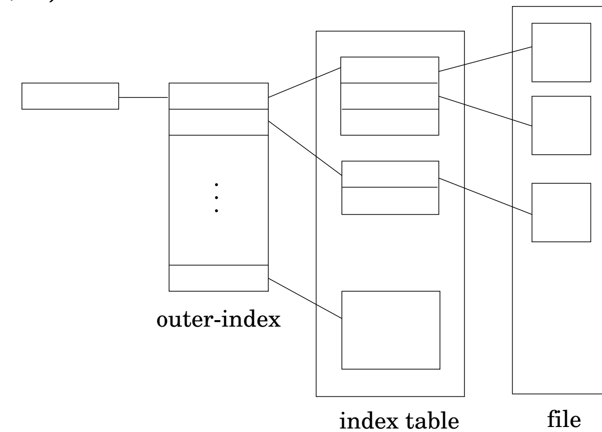
$$LA/(512 \times 511) \rightarrow Q_1, R_1$$

$$R_1/512 \rightarrow Q_2, R_2$$

  - Q_1 = blocco dell'indice da accedere
  - Q_2 = offset all'interno del blocco dell'indice
  - R_2 = offset all'interno del blocco del file

Allocazione indicizzata (cont.)

Indice a due (o più) livelli.



- Con blocchi da 512 parole:

$$LA/(512 \times 512) \rightarrow Q_1, R_1$$

$$R_1/512 \rightarrow Q_2, R_2$$

  - Q_1 = offset nell'indice esterno
  - Q_2 = offset nel blocco della tabella indice
  - R_2 = offset nel blocco del file

Unix: Inodes

- Un file in Unix è rappresentato da un *inode* (nodo indice) che contiene:

modo bit di accesso, di tipo e speciali del file

UID e GID del possessore

Dimensione del file in byte

Timestamp di ultimo accesso, modifica e mod. dell'inode

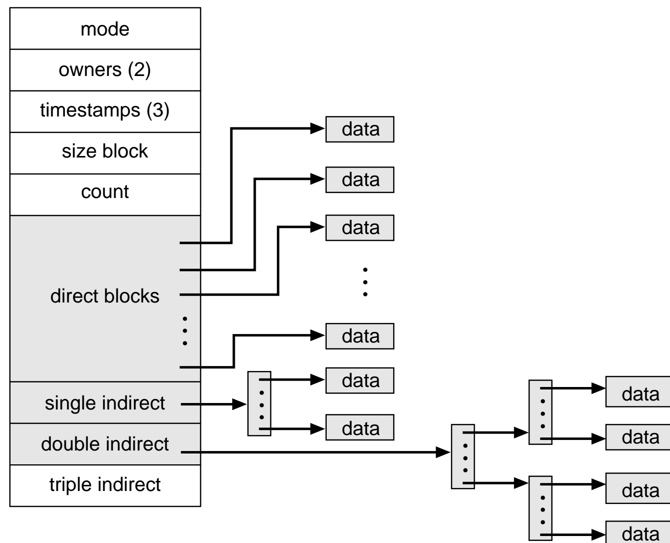
Numero di link hard che puntano a questo inode

Blocchi diretti: puntatori ai primi 12 blocchi del file

Primo indiretto: indirizzo del blocco indice dei primi indiretti

Secondo indiretto: indirizzo del blocco indice dei secondi indiretti

Terzo indiretto: indirizzo del blocco indice dei terzi indiretti (mai usato!)



Inodes (cont.)

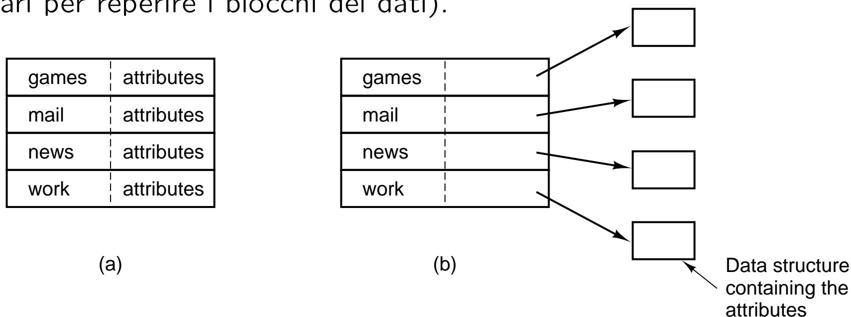
- Gli indici indiretti vengono allocati su richiesta
- Accesso più veloce per file piccoli
- N. massimo di blocchi indirizzabile: con blocchi da 4K, puntatori da 4byte (cioè indice=1 blocco ha 1024 riferimenti a blocchi)

$$\begin{aligned} L_{max} &= 12 + 1024 + 1024^2 + 1024^3 \\ &> 1024^3 = 2^{30} blk \\ &= 2^{42} byte = 4\,TB \end{aligned}$$

molto oltre le capacità dei sistemi a 32 bit.

Implementazione delle directory

Le directory sono essenziali per passare dal nome del file ai suoi attributi (anche necessari per reperire i blocchi dei dati).



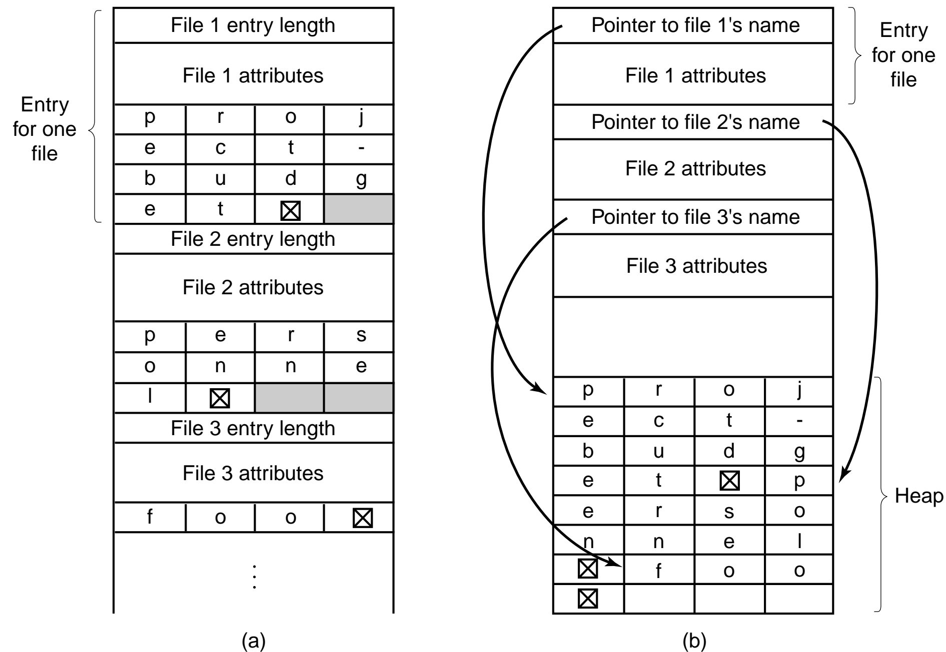
a) Gli attributi risiedono nelle entry stesse della directory (MS-DOS, Windows)

b) Gli attributi risiedono in strutture esterne (eg. inode dei file), e nelle directory ci sono solo i puntatori a tali strutture (UNIX)

Gestione dei nomi

- Fino ad ora abbiamo supposto che i nomi dei file siano brevi e di lughezza fissata
- In effetti in MS-DOS i nomi erano di al più 8 caratteri + 3 di estensione
- I file system moderni supportano tuttavia nomi di file di lunghezza variabile
- Come si puo implementare?
  - Si fissa una lunghezza massima (e.g. 255) riservando sempre tutto lo spazio (elementi di una directory sempre uguali)
  - Ogni elemento di una directory contiene la sua lunghezza, attributi, e il nome del file: problema della frammentazione
  - Ogni elemento di una directory contiene un puntatore al nome del file e gli atttributi del file; tutti i nomi vengono memorizzati insieme in un *heap* alla fine della directory

Dimensioni variabili e heap



Directory: liste, hash, B-tree

- Lista lineare di file names con puntatori ai blocchi dati
  - semplice da implementare
  - lenta nella ricerca, inserimento e cancellazione di file
  - può essere migliorata mettendo le directory in cache in memoria
- Tabella hash: lista lineare con una struttura hash per l'accesso veloce
  - si entra nella hash con il nome del file
  - abbassa i tempi di accesso
  - bisogna gestire le *collisioni*: ad es., ogni entry è una lista
- B-tree: albero binario bilanciato
  - ricerca binaria
  - abbassa i tempi di accesso
  - bisogna mantenere il bilanciamento

File condivisi

- In file system tipo Unix due directory diverse (e quindi due utenti diversi) possono puntare allo stesso file
  - Collegamento (hard link): i file mantengono un contatore per i riferimenti multipli (ad es. nell'i-node) in modo da evitare puntatori *dangling* (cioe' puntatori a file che non esistono più)
  - Collegamento simbolico (symbolic link): si crea un nuovo file di tipo speciale che contiene il path name del file da collegare; non crea problemi in cancellazione
    * Se si cancella il link simbolico non viene modificato il file
    * Se si cancella il file, chi prova ad usare il link otterrà un errore

Gestione dello spazio libero

I blocchi non utilizzati sono indicati da una *lista di blocchi liberi* — che spesso non è una vera lista

- Vettore di bit (*block map*): 1 bit per ogni blocco

  01011101010101111110110000001010000000101101010111100

$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ libero} \\ 1 \Rightarrow \text{block}[i] \text{ occupato} \end{cases}$$

- Comodo per operazioni assembler di manipolazione dei bit

Gestione dello spazio libero (Cont.)

- La bit map consuma spazio. Esempio:

  block size = 2^{12} bytes
  disk size = 2^{35} bytes (32 gigabyte)
  $n = 2^{35}/2^{12} = 2^{23}$ bits = 2^{20} byte = 1M byte

- Facile trovare blocchi liberi contigui

- Alternativa: *Linked list (free list)*
  - Inefficiente - non facile trovare blocchi liberi contigui
  - Non c'è spreco di spazio.

Affidabilità del file system

Perdere un file system, o parte di esso, è spesso un danno irreparabile e molto costoso.

- Backup (automatico o manuale) dei dati dal disco ad altro supporto (altro disco, nastri, ...)
  - dump *fisico*: direttamente i blocchi del file system (veloce, ma difficilmente incrementale e non selettivo)
  - dump *logico*: porzioni del virtual file system (più selettivo, ma a volte troppo astratto (link, file con buchi...))

  Recupero dei file perduti (o interi file system) dal backup: dall'amministratore, o direttamente dall'utente.

Come funziona il dump logico

- Il dump logico riguarda una particolare parte di file system (un suo sottoalbero) lo scopo e' riversare ad es. su nastro tutti i file modificati dall'ultimo dump. Occorre tuttavia mantenere su nastro tutte le informazioni sulla struttura dell'albero (cioe' anche directory e file non modificati)

- L'algoritmo utilizzato ad. es. in Unix effettua un *dump incrementale*

- Inizialmente si effettua un dump completo di tutto il file system

- Successivamente si salvano su nastro solo le modifiche dall'ultimo dump logico

Dump incrementale in Unix

- L'algoritmo per dump incrementale consiste di 4 fasi e utilizza una mappa di bit indicizzata sui numeri di i-node del file system considerato
  - I fase: si visita l'albero e si marcano tutti gli i-node associati a file modificati e si marcano anche tutte le directory;
  - II fase: si visita nuovamente l'albero e si smarcano tutte le directory che *non* contengono (ad una qualsiasi profondità nel corrispondente sottoalbero) file modificati

    Nota: i-node marcato $\rightarrow$ da salvare su nastro
  - III fase: si analizzano tutti gli i-node in ordine di numero e si scaricano su nastro tutte le directory *marcate* insieme ai loro attributi (proprietario, ecc)
  - IV fase: si analizzano tutti gli i-node in ordine di numero e si scaricano su nastro tutte i file *marcati* insieme ai loro attributi.

Dump incrementale in Unix

- Ad es. FS: /home/giorgio/Lucidi /home/giorgio/Articoli e Lucidi contiene un file slide modificato dall'ultimo dump
- Nella I fase: marchiamo /home /home/giorgio /home/giorgio/Lucidi /home/giorgio/Articoli e /home/giorgio/Lucidi/slide
- Nella II fase: smarchiamo solo /home/giorgio/Articoli: /home /home/giorgio /home/giorgio/Lucidi contengono file modificati
- Nella III fase salviamo su nastro le informazioni su /home /home/giorgio /home/giorgio/Lucidi
- Nella IV fase salviamo su nastro /home/giorgio/Lucidi/slide

Ripristino da dump su nastro

- Per ripristinare un file system da nastro di dump:
  - Si crea un file system vuoto
  - Si ripristina il dump *completo* più recente: le directory compaiono prima su nastro: vengono utilizzare per creare lo scheletro del file system e poi vengono riempite con i file
  - Si procede allo stesso modo con tutti i dump incrementali fatti dopo il dump completo

Consistenza del file system

- In seguito ad un crash, blocchi critici possono contenere informazioni incoerenti, sbagliate e contradditorie.
- Si utilizzano dei programmi di controllo della consistenza (*scandisk*, *fsck*): usano la ridondanza dei metadati, cercando di risolvere le inconsistenze.
- Programmi come *fsck* effettuano controlli sia per *blocchi* che per *file*

fsck: controllo per blocchi

- Nel controllo per *blocchi* si costruiscono due tabelle indicizzate sui blocco fisici
- La prima tabella tiene traccia di quante volte un blocco è presente in un file
- La seconda tabella tiene traccia di quante spesso un blocco è presente nella lista (o nella mappa di bit) dei blocchi liberi (o nella mappa di bit)
- *fsck* esegue due passi:
  - prima scandisce tutti gli i-node e recupera i numeri dei suoi blocchi: per ogni blocco incrementa il contatore nella prima tabella;
  - poi scandisce la lista dei blocchi liberi: per ogni blocco incrementa il contatore nella seconda tabella.

fsck: recovery di blocchi

- Se il file system è coerente ogni blocco avrà un 1 nella prima o nella seconda tabella
- Se un blocco non compare in nessuna delle due tabelle (ha contatore $= 0$ in entrambe): viene aggiunto alla lista libera
- Se un blocco compare più volte nella lista libera (ha contatore > 1 nella seconda tabella): si ricostruisce la lista
- Se un blocco compare più volte nello stesso file o in file diversi (ha contatore > 1 nella prima tabella): si allocano dei nuovi blocchi si fa una copia del contenuto del blocco inconsistente e si associano tale copie ai file a cui apparteneve il blocco in questione; inoltre si manda un messaggio di errore all'utente
- Se un blocco compare sia in un file che nella lista libera: si ricostruisce la lista rimuovendo il blocco dalla lista

fsck: controllo per file

- Nel controllo per *file* si utilizza una tabella indicizzata sui file che tiene traccia del numero di riferimenti al file
- *fsck* visita il file system e incrementa il contatore di ogni file che incontra in una directory durante la visita
  (nota: un file con link fisico puo' appartenere a più directory)
- Al termine della visita si confronta il contatore C associate ad un file nella tabella con il numero di riferimenti R contenuto nel suo i-node

fsck: recovery per file

- $C =$ valore del contatore associato al file f nella tabella costruita da *fsck*
- $R =$ valore del contatore di riferimenti nell'i-node del file f (modificato ogni qualvolta si crea un link fisico)
  - Se $C = R$ il file system è coerente
  - Se $R > C$ si ha un errore non grave ma un potenziale spreco di spazio: se cancello i file dalle directory in questione, R rimane > 0 e quindi l'i-node non viene rimosso.
    Recovery: si assegna R a C in modo che $C = R$.
  - Se $R < C$ si ha un errore grave. Se cancello un file da una delle directory in questione, R potrebbe diventare $= 0$ e quindi il suo i-node rimosso e i blocchi relativi rilasciati anche se esistono altre directory che puntano allo stesso file.
    Recovery: si assegna C ad R in modo che $C = R$.

Efficienza e performance di un file system

Dipende da

- algoritmi di allocazione spazio disco e gestione directory
- tipo di dati contenuti nelle directory
- grandezza dei blocchi
  - blocchi piccoli per aumentare l'efficienza (meno frammentazione interna)
  - blocchi grandi per aumentare le performance
  - e bisogna tenere conto anche della paginazione!

Accorgimenti

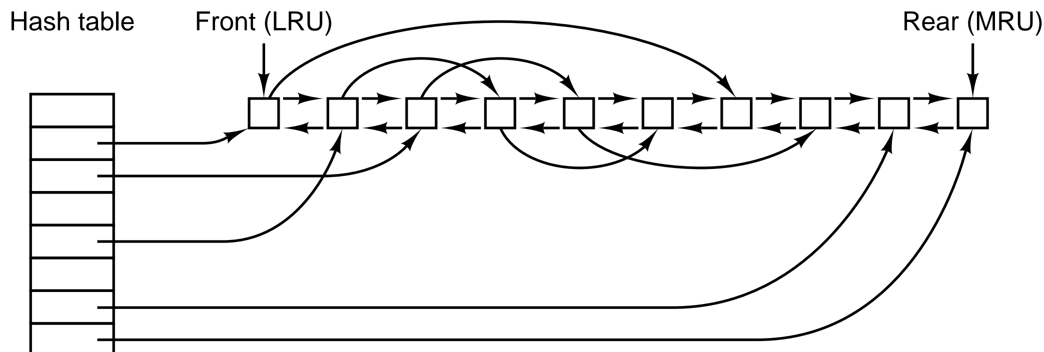
- *read-ahead*: leggere blocchi in cache *prima* che siano realmente richiesti.
  - Aumenta il throughput del device
  - Molto adatto a file che vengono letti in modo sequenziale; Inadatto per file ad accesso casuale (es. librerie)
  - Il file system può tenere traccia del modo di accesso dei file per migliorare le scelte.
- Ridurre il movimento del disco
  - durante la scrittura del file, sistemare vicini i blocchi a cui si accede di seguito (facile con bitmap per i blocchi liberi, meno facile con liste)
  - raggruppare (e leggere) i blocchi in gruppi (cluster)
  - collocare i blocchi con i metadati (inode, p.e.) presso i rispettivi dati

Migliorare le performance: caching

disk cache – usare memoria RAM per bufferizzare i blocchi più usati. Può essere

- sul controller: usato come *buffer di traccia* per ridurre il tempo di latenza nell'accesso al disco
- (gran) parte della memoria principale, prelevando pagine dalla free list. Può arrivare a riempire tutta la memoria RAM: "un byte non usato è un byte sprecato".

I buffer sono organizzati in una coda (ordinata a seconda del tempo di accesso, primo blocco=usato meno recentemente) con accesso hash



- La coda può essere gestia LRU, o CLOCK, ...
- Un blocco viene salvato su disco quando deve essere liberato dalla coda.
- Se blocchi critici vengono modificati ma non salvati mai (perché molto acceduti), si rischia l'inconsistenza in seguito ai crash.

- Variante di LRU: dividere i blocchi in categorie a seconda se
  - il blocco verrà riusato a breve? in tal caso, viene messo in fondo alla lista.
  - il blocco è critico per la consistenza del file system? (tutti i blocchi tranne quelli dati) allora ogni modifica viene immediatamente trasferita al disco.

Anche le modifiche ai blocchi dati vengono trasferite prima della deallocazione:

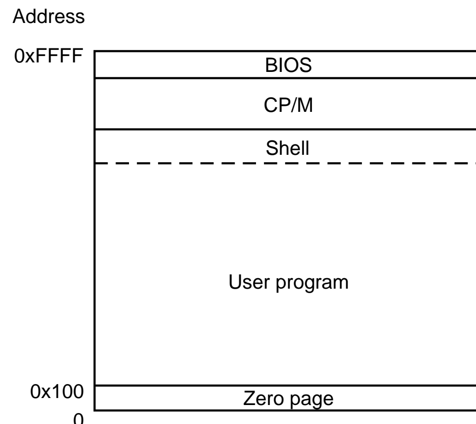
- asincrono: ogni 20-30 secondi (Unix, Windows)
- sincrono: ogni scrittura viene immediatamente trasferita anche al disco (*write-through cache*, DOS).

Esempi di File System

CP/M

- Il sistema CP/M può essere considerato l'antenato di MS-DOS
- CP/M era un sistema operativo per macchine con processori a 8 bit e 4KB di RAM e un singolo floppy disk di 8 pollici con capacità di 180 KB.
- CP/M era scarno e molto compatto e rappresenta un interessante esempio di sistema embedded
- Quando caricato nella RAM:
  - Il BIOS contiene una libreria di 17 chiamate di sistema (interfaccia harware)
  - Il sistema operativo occupa meno di 3584 byte (< 4KB!); la shell occupa 2 KB

  - Ultimi 256 byte: vettore interruzioni, buffer per la linea di comando

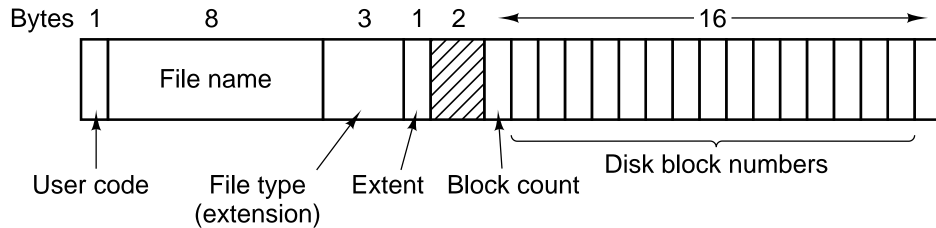


- Comandi vengono copiati nel buffer, poi CP/M cerca il programma da eseguire, lo scrive a partire dall'indirizzo 256 e gli passa il controllo
- Il programma può scrivere sopra la shell se necessario

File System in CP/M

- CP/M ha una sola directory (flat)
- Gli utenti si collegavano uno alla volta: i file avevano informazioni sul nome del proprietario
- Dopo l'avvio del sistema CP/M legge la directory e calcola una mappa di bit (di 23 byte per un disco da 180KB) per i blocchi liberi
- La mappa viene tenuta in RAM e buttata via quando si spegne la macchina

Elementi di directory in CP/M

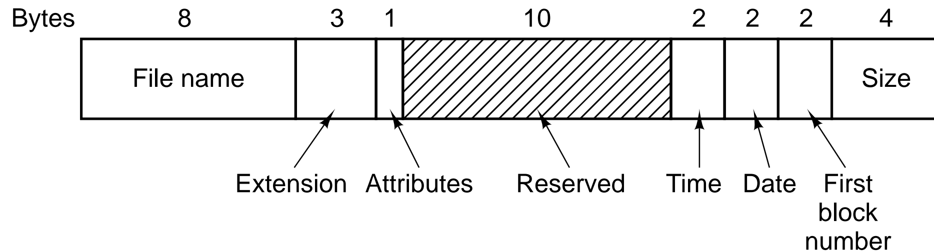


- Lunghezza del nome fissa: 8 caratteri + 3 di estensione
- Extent: serve per file con più di 16 blocchii: si possono usare più elementi di directory per lo stesso file, extent mantiene l'ordine con cui leggere i blocchi
- Contatore blocchi: numero complessivo di blocchi (non dimensione del file: serve EOF nell'ultimo blocco)
- Blocchi: dimensione da 1KB

File System in MS-DOS

- MS-DOS è una evoluzione di CP/M
- Le directory hanno struttura ad albero e non flat
- Si usa la File Allocation Table (FAT) per la mappa file blocchi e la gestione dei blocchi liberi (sono marcati in maniera speciale)
- Non è tuttavia multi utente

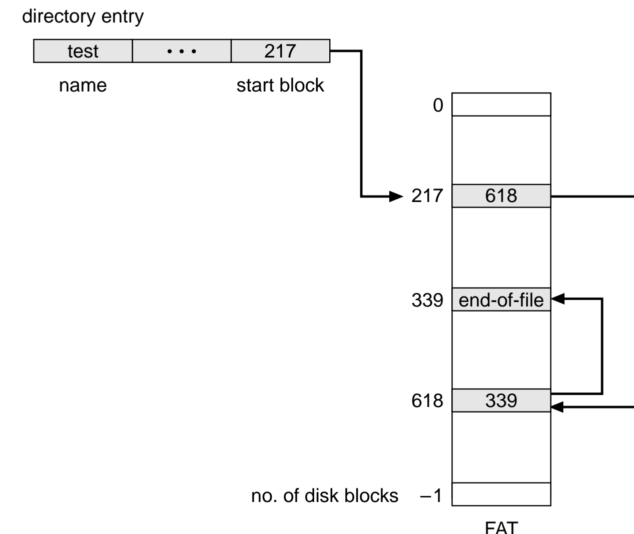
Directory in MS-DOS



- Lunghezza del nome fissa
- Attributi: read-only, system, archived, hidden
- Time: ore (5bit), min (6bit), sec (5bit)
- Date: giorno (5bit), mese (4bit), anno-1980 (7bit) (Y2108 BUG!)

File-allocation table (FAT)

Mantiene la linked list in una struttura dedicata, all'inizio di ogni *partizione* del disco



FAT12

- La prima versione di DOS usava una FAT-12 (cioé con indirizzi di disco di 12 bit) con blocchi da 512 byte
- Dimensione max di una partizione: $2^{12}x512$ byte ($2MB$)
- Dimensione della FAT: 4096 elementi da 2 byte
- Ok per floppy ma non per hard disk
- Con blocchi da 4KB si ottenevano partizioni da 16MB (con 4 partizioni: dischi da 64MB)

FAT16

- FAT-16 utilizza indirizzi di disco da 16 bit con blocchi da 2KB a 32KB
- Dimensione max di una partizione: $2GB$
- Dimensione della FAT: 128KB

FAT32

- A partire dalla seconda versione di Windows 95 si utilizzano indirizzi di 28 bit (non 32)
- Partizioni max: $2^{28} \times 32KB (= 2^{15})$ in realtà limitate a $2TB$ ($= 2048TB$)
- Vantaggi rispetto a FAT-16: un disco da 8GB può avere una sola partizione (su FAT 16 deve stare su 4 partizioni)
- Può usare blocchi di dimensione più piccola per coprire uguale spazio disco ($2GB$ FAT-16 deve usare blocchi da $32KB$ mentre FAT-32 può usare blocchi da $4KB$

FAT12, FAT16, FAT32

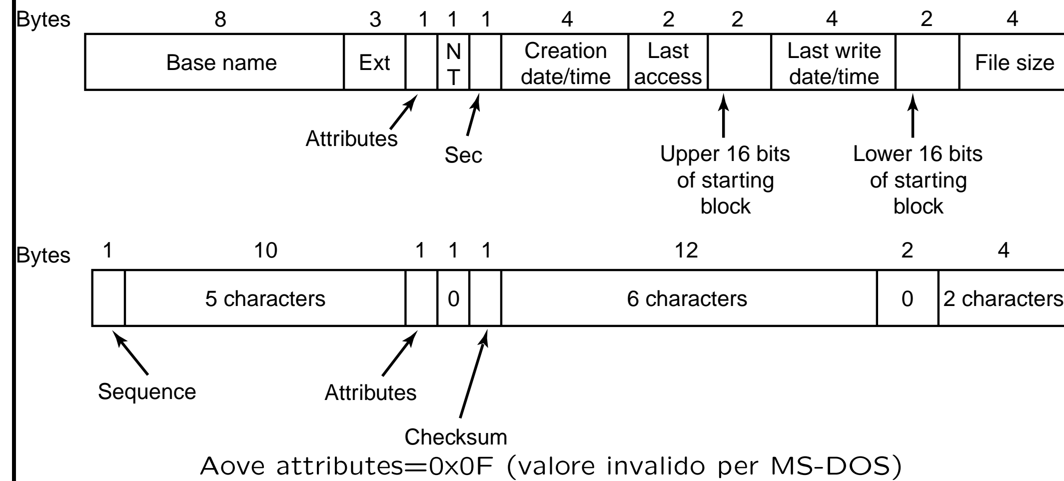
Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

In MS-DOS, tutta la FAT viene caricata in memoria.

Il block size è chiamato da Microsoft *cluster size*

Directory in Windows 98

Nomi lunghi ma compatibilità all'indietro con MS-DOS e Windows 3



Aove attributes=0x0F (valore invalido per MS-DOS)

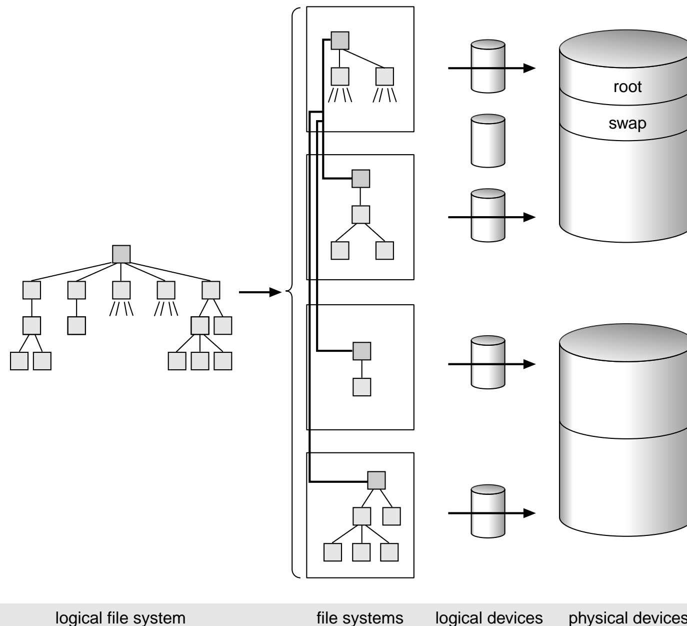
Esempio

```
$ dir
THEQUI~1        749 03-08-2000  15:38 The quick brown fox jumps over the...
```

68	d o g	A	0	CK					0	
3	o v e	A	0	CK	t h e l a				0	z y
2	w n f o	A	0	CK	x j u m p				0	s
1	T h e q	A	0	CK	u i c k b				0	r o
T	H E Q U I ~ 1	A	NT	S	Creation time	Last acc	Upp	Last write	Low	Size

Bytes

UNIX: Il Virtual File System



Il file system *virtuale* che un utente vede può essere composto in realtà da diversi file system *fisici*, ognuno su un diverso dispositivo logico

Il Virtual File System (cont.)

- Il Virtual File System è composto da più file system fisici, che risiedono in dispositivi logici (*partizioni*), che compongono i dispositivi fisici (dischi)
- Il file system / viene montato al boot dal kernel
- gli altri file system vengono montati secondo la configurazione impostata
- ogni file system fisico può essere diverso o avere parametri diversi
- Il kernel usa una coppia <*logical device number, inode number*> per identificare un file
  - Il logical device number indica su quale file system fisico risiede il file
  - Gli inode di ogni file system sono numerati progressivamente

Il Virtual File System (cont.)

Il kernel si incarica di implementare una visione uniforme tra tutti i file system montati: operare su un file significa

- determinare su quale file system fisico risiede il file
- determinare a quale inode, su tale file system corrisponde il file
- determinare a quale dispositivo appartiene il file system fisico
- richiedere l'operazione di I/O al dispositivo

I File System Fisici di UNIX

- UNIX (Linux in particolare) supporta molti tipi di file system fisici (SysV, EXT2, EXT3 e anche MSDOS); quelli preferiti sono UFS (Unix File System, aka BSD Fast File System), EXT2 (Extended 2), EFS (Extent File System)
- Il file system fisico di UNIX supporta due oggetti:
  - file "semplici" (plain file) (senza struttura)
  - directory (che sono semplicemente file con un formato speciale)
- La maggior parte di un file system è composta da blocchi dati
  - in EXT2: 1K-4K (configurabile alla creazione)

Inodes

- Un file in Unix è rappresentato da un *inode* (nodo indice).
- Gli inodes sono allocati in numero finito alla creazione del file system
- Struttura di un inote in System V:

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Inodes (cont)

- Gli indici indiretti vengono allocati su richiesta
- Accesso più veloce per file piccoli
- L'inode contiene puntatori diretti e indiretti (a 1, 2, e 3 livelli)
- N. massimo di blocchi indirizzabile: con blocchi da 4K, puntatori da 4byte

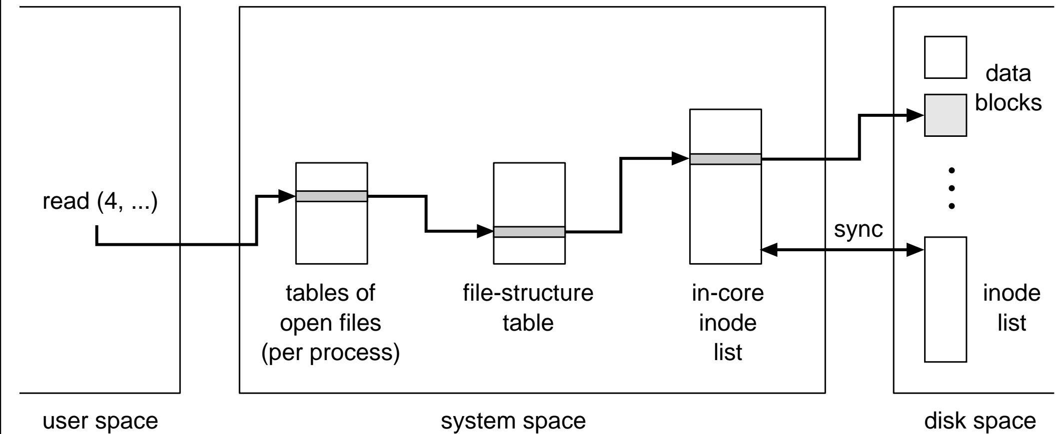
$$\begin{aligned} L_{max} &= 10 + 1024 + 1024^2 + 1024^3 \\ &> 1024^3 = 2^{30} blk \\ &= 2^{42} byte = 4\,TB \end{aligned}$$

molto oltre le capacità dei sistemi a 32 bit.

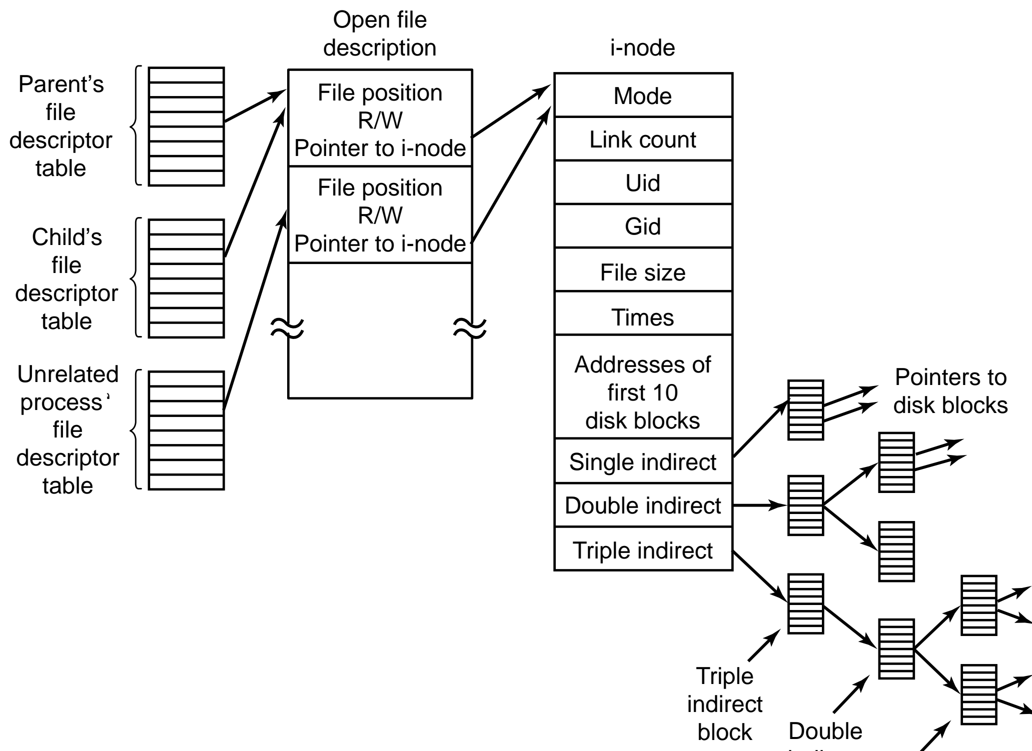
Traduzione da file descriptor a inode

- Le system calls che si riferiscono a file aperti (read, write, close, ...) prendono un *file descriptor* come argomento
- Il file descriptor viene usato dal kernel per entrare in una tabella di file aperti del processo. Risiede nella U-structure.
- Ogni entry della tabella contiene un puntatore ad una *file structure*, di sistema. Ogni file structure punta ad un inode (in un'altra lista), e contiene la posizione nel file.
- Ogni entry nelle tabelle contiene un contatore di utilizzo: quando va a 0, il record viene deallocato

File Descriptor, File Structure e Inode



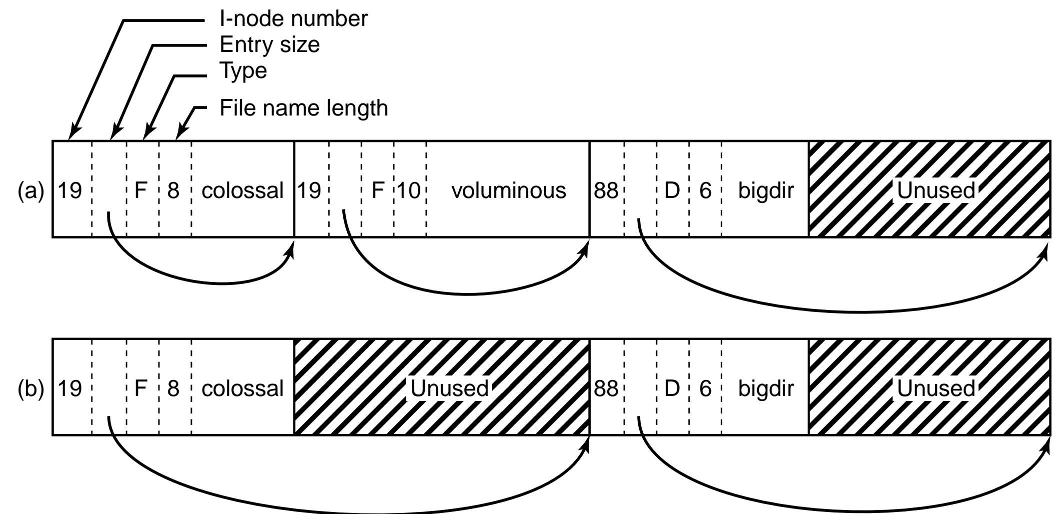
La tabella intermedia è necessaria per la semantica della condivisione dei file tra processi



- Le chiamate di lettura/scrittura e la seek cambiano la posizione nel file
- Ad una *fork*, i figli ereditano (una copia de) la tabella dei file aperti dal padre ⇒ condividono la stessa file structure e quindi la posizione nel file
- Processi che hanno aperto indipendentemente lo stesso file hanno copie private di file structure

Directory in UNIX

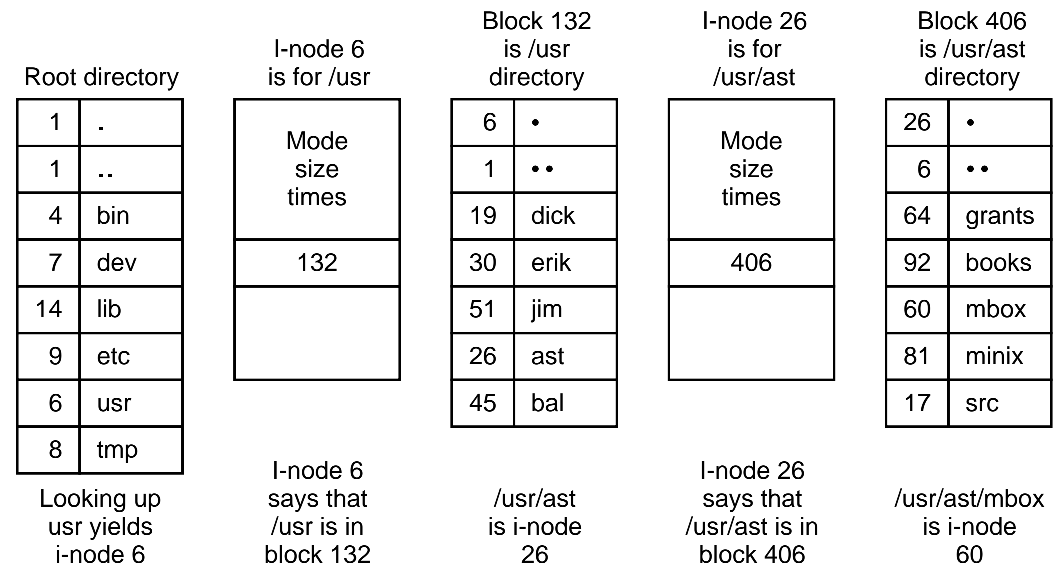
- Il tipo all'interno di un inode distingue tra file semplici e directory
- Una directory è un file con entry di lunghezza variabile. Ogni entry contiene
  - puntatore all'inode del file
  - posizione dell'entry successiva
  - lunghezza del nome del file (1 byte)
  - nome del file (max 255 byte)
- entry differenti possono puntare allo stesso inode (*hard link*)



Traduzione da nome a inode

L'utente usa i nomi (o path), mentre il file system impiega gli inode ⇒ il kernel deve risolvere ogni nome in un inode, usando le directory

- Prima si determina la directory di partenza: se il primo carattere è "/", è la root dir (sempre montata); altrimenti, è la current working dir del processo in esecuzione
- Ogni sezione del path viene risolta leggendo l'inode relativo
- Si ripete finché non si termina il path, o la entry cercate non c'è
- Link simbolici vengono letti e il ciclo di decodifica riparte con le stesse regole. Il numero massimo di link simbolici attraversabili è limitato (8)
- Quando l'inode del file viene trovato, si alloca una *file structure* in memoria, a cui punta il *file descriptor* restituito dalla *open(2)*



Esempio di file system fisico: Unix File System

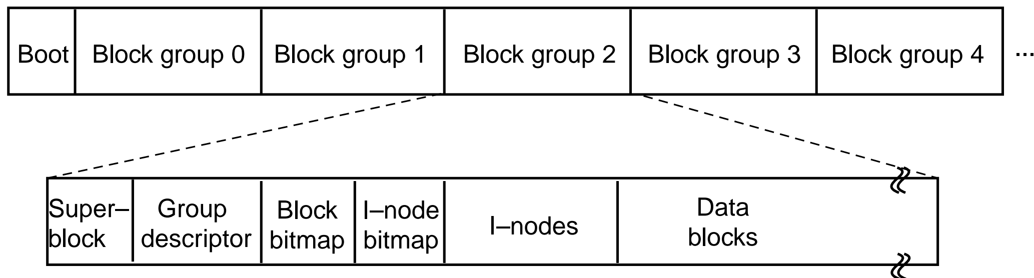
- In UFS (detto anche Berkley Fast File System), i blocchi hanno due dimensioni: il *blocco* (4-8K) e il *frammento* (0.5-1K)
  - Tutti i blocchi di un file sono blocchi tranne l'ultimo
  - L'ultima parte del file è tenuta in frammenti
- Riduce la frammentazione interna e aumenta la velocità di I/O
- La dimensione del blocco e del frammento sono impostati alla creazione del file system:
  - se ci saranno molti file piccoli, meglio un fragment piccolo
  - se ci saranno grossi file da trasferire spesso, meglio un blocco grande
  - il rapporto max è 8:1. Tipicamente, 4K:512 oppure 8K:1K.

Esempio di file system fisico: Unix File System (Cont)

- Si introduce una cache di directory per aumentare l'efficienza di traduzione
- Suddivisione del disco in *cilindri*, ognuno dei quali con il proprio superblock, tabella degli inode, dati. Quando possibile, si allocano i blocchi nello stesso gruppo dell'inode.

  In questo modo si riduce il tempo di seek dai metadati ai dati.

Esempio di file system fisico: EXT2



- Derivato da Berkley Fast File System (UFS)
- Blocchi tutti della stessa dimensione (1K-4K)
- Suddivisione del disco in *gruppi* di 8192 blocchi, ma non secondo la geometria fisica del disco
- Il *superblock* (blocco 0) contiene informazioni vitali sul file system (tipo di file system, primo inode, numero di gruppi, numero di blocchi liberi e inodes liberi,. . .)

- Ogni gruppo ha una copia del superblock, la propria tabella di inode e tabelle di allocazione blocchi e inode
- Per minimizzare gli spostamenti della testina, si cerca di allocare ad un file blocchi dello stesso gruppo

NTFS: File System di 2K, NT, XP

- Il file system NTFS è stato sviluppato *from scratch* per Windows NT
- Windows 2000 supporta sia FAT (-16 e -32) che NTFS
- NTFS supporta indirizzi di disco a 64 bit (ricordate che FAT-16 supporta indirizzi a 16 bit (max 2GB), e FAT-32 indirizzi a 28 bit (max 2TB))
- NTFS è un sistema molto più complesso del file system per MS-DOS

Caratteristiche di NTFS

- I nomi sono lunghi fino a 255 caratteri Unicode.
- I caratteri utilizzati sono in Unicode (2 byte per carattere)
- Si distinguono maiuscole e minuscole (ma le API Win32 no)
- A differenza di Unix e FAT-32, un file non è una sequenza lineare di file ma bensì si compone di attributi multipli (ad es. nome, flag, dati), ognuno rappresentato da una sequenza di byte
- L'idea di file come sequenza di attributi è stata introdotta dall'MacIntosh (MacOS)
- La lunghezza massima di una sequenza è 2^{64} byte (i.e. $18 exabyte \circ 18^{18}$)

Chiamate API Win32 del file system

- Le funzioni per il file system di API Win32 sono simili a quelle di Unix
- Ad esempio la chiamata `CreateFile` utilizzata per la creazione e apertura di un file restituisce un *gestore di file* (handle) come per il *file descriptor* di Unix


```
/* Open files for input and output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);

/* Copy the file. */
do {
    s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
    if (s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Implementazione di NTFS

- Diverse partizioni possono essere unite a formare un *volume logico*
- Lo spazio viene allocato in *cluster* (sequenze lineari di blocchi)
- Un blocco a dimensione variabile da 512byte e 64KB.
- La maggior parte dei dischi NTFS usa blocchi a 4KB
- La struttura principale in ogni volume è la Master File Table (MFT) che descrive file e directory

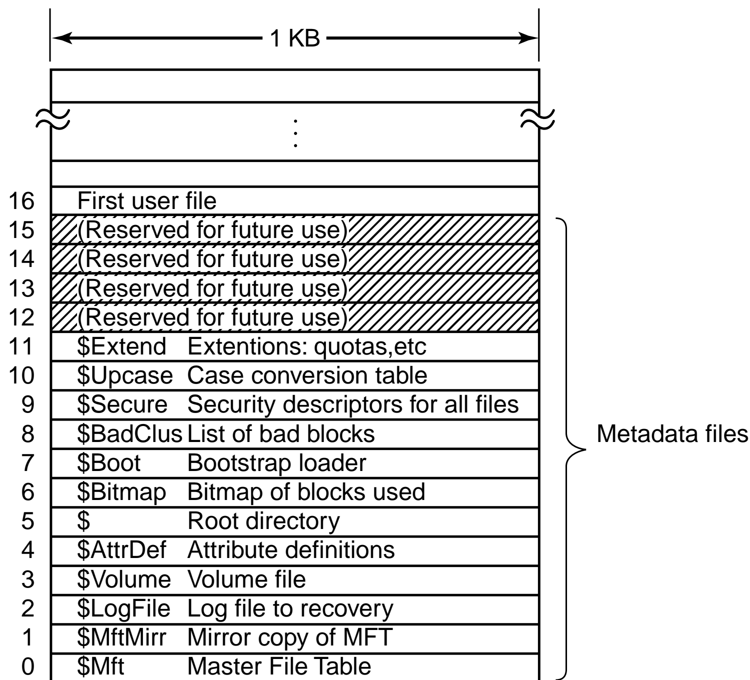
Master File Table (MFT)

- È una sequenza lineare di record di 1KB
- Ogni record della MFT descrive un file o una directory e contiene attributi e la lista di indirizzi disco per quel file
- Per file grandi si possono usare più record per la lista dei blocchi (record base con puntatore ad altri record)
- Una mappa di bit tiene traccia dei record MFT liberi
- Una MFT è a sua volta vista come un file: può essere collocata ovunque su disco (evita il problema di settori imperfetti nella prima traccia), e può crescere fino a 2^{48} record

Struttura della MFT

- I primi 16 record descrivono l'MFT stesso e il volume (analogo al superblock di Unix).
  - Descrizione della MFT (primo record) e copia ridondante (secondo record): es. posizione dei blocchi della MFT
  - File di log: registra operazioni di aggiunta/rimozone/modifica al file system
  - Informazioni sul volume (etichetta, dimensione)
  - Informazioni sugli attributi dei file
  - Posizione della root dir (file anch'essa)
  - Attributi ed indirizzi di disco della bitmap per gestire lo spazio libero
  - Lista di blocchi malfunzionanti
  - Informazioni di sicurezza

  - Mappa dei caratteri maiuscoli (non sempre ovvia)
  - Informazioni su quote disco
  - Gli ultimi 4 record sono riservati ad usi futuri
- L'indirizzo del primo blocco della MFT viene memorizzato nel blocco d'avvio



Record della MFT

- Le informazioni sui file utente vengono memorizzate a partire dal record 16 della MFT
- Ogni record ha un'intestazione (header) seguita da una sequenza di coppie (intestazione di attributo e valore).
- Il record header contiene ad esempio uncontatore di riferimenti al file (come per i-node), ed il numero effettivo di byte usato nel record
- Ogni intestazione di attributo contiene il tipo dell'attributo e la locazione e la lunghezza del corrispondente valore
- I valori possono seguire il proprio header (*resident attribute*) o essere memorizzati in un record separato (*nonresident attribute*)

Attributi dei file NTFS

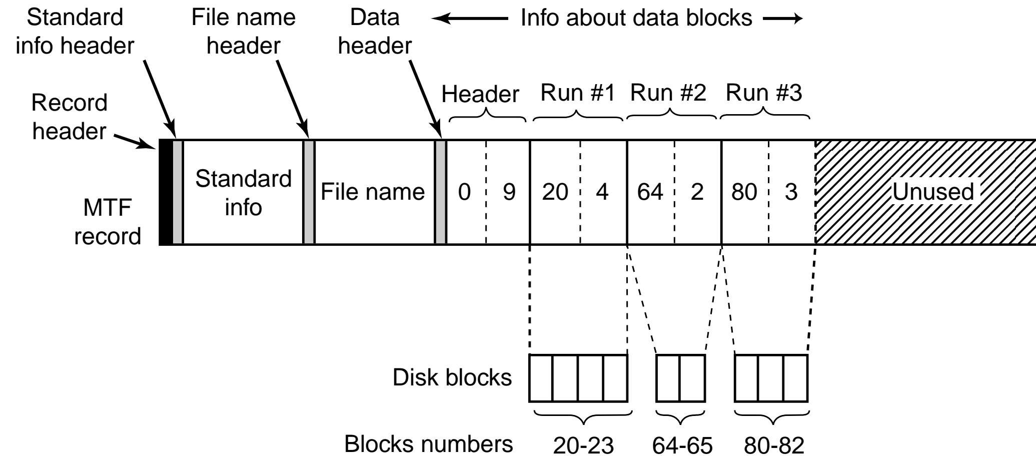
NTFS definisce 13 attributi standard quali

- Informazioni standard: proprietario, diritti, time-stamps, contatore di link fisici
- Nome del file (lunghezza variabile, Unicode)
- Identificatore di oggetto: nome unico per il file nel sistema
- Punti di analisi: per operazioni speciali durante il parsing del nome (ad es. montaggio)
- Dati: contiene gli indirizzi disco dei veri dati; se sono residenti, il file si dice "immediate"

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

File NTFS non residenti

I file non immediati si memorizzano a "run": sequenze di blocchi consecutivi
Nel record MFT corrispondente ci sono i puntatori ai primi blocchi di ogni run



- L'header del file contiene l'offset del primo blocco (e.g. 0) e il numero di blocchi totale (e.g. 9)
- Gli indirizzi dei run sono memorizzati a coppie (numero del primo blocco, numero dei blocchi del run)
- Nota: un file con n blocchi può essere memorizzato in numero di range che varia da 1 a n
- Un file descritto da un solo MFT record si dice *short* (ma potrebbe non essere corto per niente!)

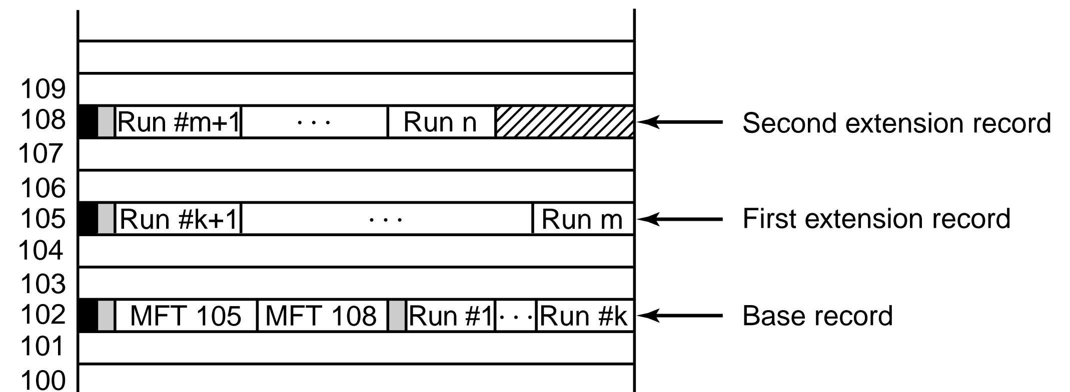
Dimensione File NTFS

- Non esiste un limite superiore alla dimensione di un file
- Ogni coppia richiede due numeri a 64 bit: 16 byte
- Una coppia può rappresentare più di un milione di blocchi disco consecutivi (ad es. 20 sequenze separate da 1 milione di blocchi da $1KB = 20KB$)
- Si usano metodi di compressione per memorizzate le coppie in meno di 16byte (si arriva fino a 4byte)

File "long"

Se il file è lungo o molto frammentato (es. disco frammentato), possono servire più di un record nell'MFT.

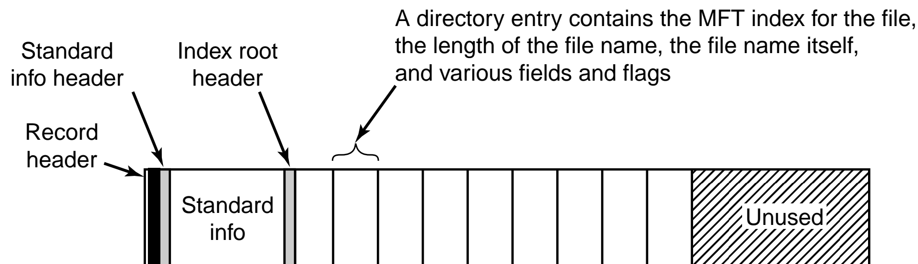
Prima si elencano tutti i record aggiuntivi, e poi seguono i puntatori ai run.



Se i puntatori ai record aggiuntivi non stanno su un solo MFT si memorizza la lista dei record con i blocchi su disco invece che nel MFT

Directory in NTFS

Le directory corte vengono implementate come semplici liste direttamente nel record MFT.



Directory più lunghe sono implementate come file nonresident strutturati a B+tree.

Navigazione in NTFS

- Consideriamo l'indirizzo

  C:\maria\web.html

- La directory radice \ contiene un puntatore alla lista di nomi di volumi logici (C, E, D, ecc)

- Il nome C: è un collegamento simbolico con la partizione del disco

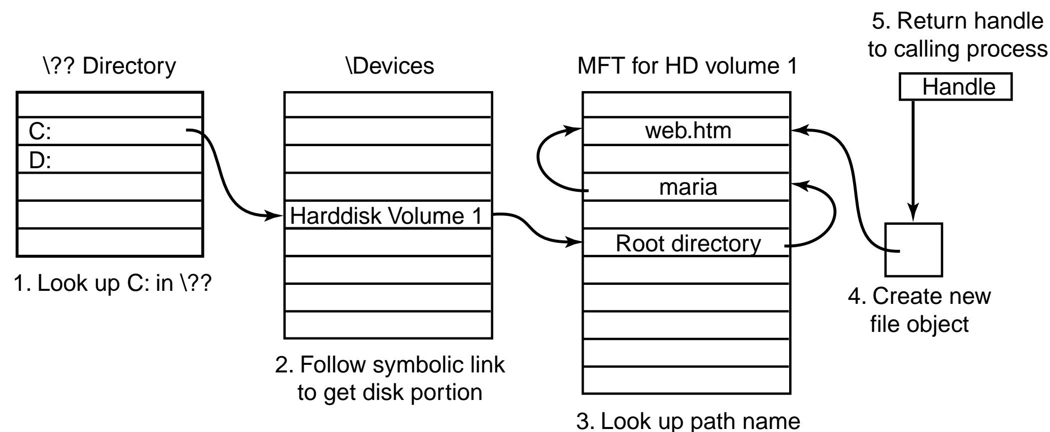
- Una volta identificata la partizione di può recuperare la corrispondente MFT

- Nel record 5 della MFT troviamo informazioni sulla directory radice del disco C

- La stringa *maria* viene cercata all'interno del record della directory radice, da tale ricerca otteniamo un indice nel MFT per la directory maria

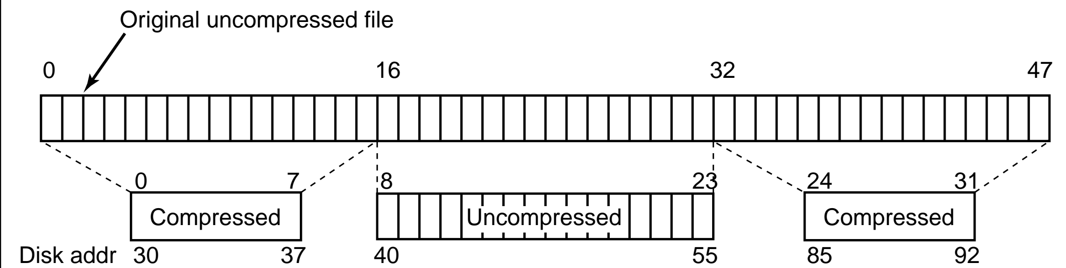
- Quindi esaminiamo il record alla ricerca di *web.html*

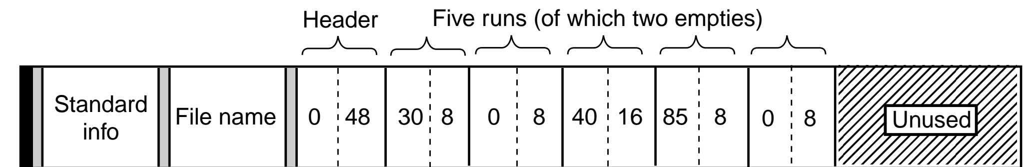
- Se la ricerca ha successo si crea un nuovo *handle* (oggetto) che contiene l'indice del file nel MFT



Compressione file

- NTFS supporta la compressione trasparente dei file (cioè i file vengono compressi quando creati e decompressi in lettura)
- L'algoritmo di compressione lavora su gruppi di 16 blocchi: se si riescono a comprimere si scrivono i blocchi compressi e si memorizzano nel record MFT dei blocchi virtuali (con indirizzo di disco 0) per i blocchi mancanti;
- poi si prosegue con i successivi 16 blocchi





- Quando NFTS legge un file e trova due coppie consecutive $(n, m)(0, k)$ capisce che $m + k$ sono stati compressi in m blocchi e applica l'algorimo di decompressione a quella sotto-sequenza

File System CDRom

- File system particolarmente semplici in quanto progettati per supporti di sola lettura
- Ad esempio, non viene tenuta traccia dei blocchi liberi: i CDROM non vengono modificati
- Esistono vari tipi di file system: lo standard ISO 9660 specifica il tipo comunemente adottato dai supporti di lettura per CDROM

Organizzazione dei CDROM

- I CDROM hanno una spirale continua che contiene i bit in una sequenza lineare
- I bit lungo la spirale sono divisi in blocchi di 2532 byte:
  - 2048 byte sono di dati,
  - I byte rimanenti contengono header e codici di correzione

Standard ISO 9660

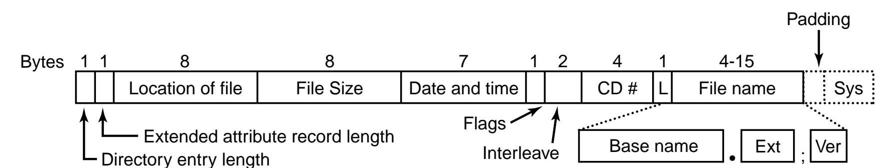
- File system leggibile da tutti i lettori e supportato dai principali sistemi operativi (anche MS-DOS)
- Lo standard definisce un file system per un'insieme di CDROM (fino a $2^{16} - 1$ CDROM), ognuno partizionato in volumi logici
- Ogni CDRom inzia con 16 blocchi la cui funzione non è definita dallo standard (vengono usati ad esempio per programmi di avvio)
- Di seguito si trova il descrittore di volume primario che contiene informazioni generali sul CDROM quali
  - identificatore di sistema (32 byte)
  - identificatore di volume (32 byte)

  - identificatore del distributore (128 byte)
  - identificatore del preparatore dei dati (128 byte)
  - nomi di tre file (riassunto, avviso di copyright ed informazioni bibliografiche)
- Il descrittore contiene anche i seguenti numeri chiave
  - dimensione del blocco logico (normalmente 2048, in alcuni casi 4096, 8192)
  - numero di blocchi
  - date di creazione e scadenza
- Infine contiene un elemento di directory per la directory radice dei dati memorizzati

Directory in un CDROM

- Le directory sono composte di un numero variabile di elementi
- Gli elementi sono di dimensione variabile (tra 10 e 12 campi)
- La profondit'a di una directory è al più di 8 livelli, mentre non c'è limite al numero di elementi in una directory
- I primi due elementi indicano la directory corrente e la directory genitore

Elementi di Directory in un CDROM



- Il primo byte identifica la lunghezza dell'elemento di directory
- Il secondo byte la lunghezza del record di eventuali attributi estesi
- Poi seguono
  - Posizione del file: i file sono identificati tramite la coppia posizione del primo blocco e lunghezza (i file sono memorizzati come blocchi contigui)
  - Data di registrazione (range 1900-2155)

- Flag: contiene bit per distinguere file da directory, bit per nascondere file, bit per abilitare uso di attributi estesi, bit per marcare l'ultimo elemento di directory
- Interlacciamento: usato solo in versioni avanzate
- Numero del CDROM sul quale è posizionato il file (l'elemento potrebbe far riferimento ad un file su un'altro CD dell'insieme)
- Dimensione del nome del file in byte
- Nome del file: nome (8 caratteri), punto, estensione (3 caratteri)
- Padding: usato per far si che ogni elemento di directory sia formato da un numero pari di byte (per allineamento)
- System use: utilizzato in modo speciale dai diversi sistemi operativi

Livelli in ISO 9660

- Lo standard definisce tre livelli
  - Livello 1: nomi con 8+3 caratteri e file memorizzati in blocchi contigui
  - Livello 2: nomi fino a 31 caratteri
  - Livello 3: nomi come nel livello 2, file formati da diverse sezioni di blocchi contigui; le sezioni possono essere condivise tra diversi file, o comparire più volte nello stesso file

Estensione Rock Ridge

- L'estensione Rock Ridge permette di rappresentare file system Unix in un CDROM

- Le informazioni aggiuntive vengono memorizzate nel campo *system use* e sono ad esempio:
  - Bit Unix per i diritti sui file, bit SETUID e SETGID
  - Nome alternativo: nome UNIX per il file
  - Campi per la rilocazione di una directory (per superare il limite di 8 livelli)
  - Time stamp contenuti negli inode di un file Unix (creazione, ultima modifica, ultimo accesso)

Estensione Joliet

- L'estensione Joliet permette di rappresentare file system Windows in un CDROM

- Le informazioni aggiuntive vengono memorizzate nel campo *system use* e specificano ad esempio:
  - Nome di file lungo (fino a 64 caratteri)
  - Insiemi di caratteri Unicode per i nomi (caratteri unicode occupano 2 byte, quindi nomi di al più 128 byte)
  - Profondità della struttura della directory maggiore di 8 livelli
  - Nomi di directory con estensione (non usato per ora)