Università di Udine — Facoltà di Scienze MM.FF.NN.

Laurea in Informatica — A.A. 2002/03

Trasparenze del Corso di *Sistemi Operativi*

Marino Miculan

Università di Udine

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.

Introduzione

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

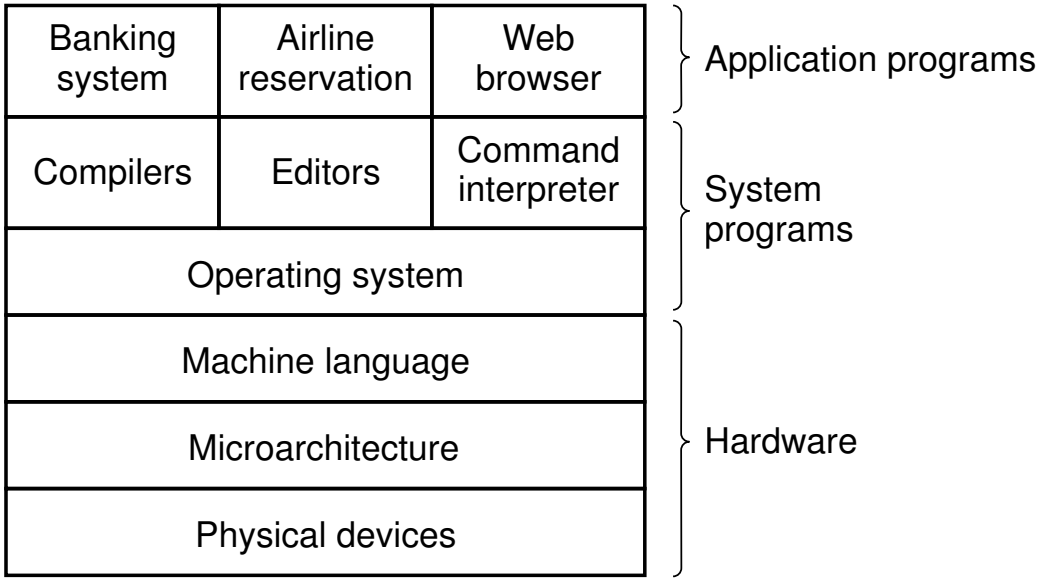
Cosa è un sistema operativo?

Possibili risposte:

- È un *programma di controllo*
- È un *gestore di risorse*
- È un *divoratore di risorse*
- È un *fornitore di servizi*
- È simile ad un *governo*: non fa niente, di per sé...
- ...

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 varî 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 (compilatori, database, videogiochi, 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.

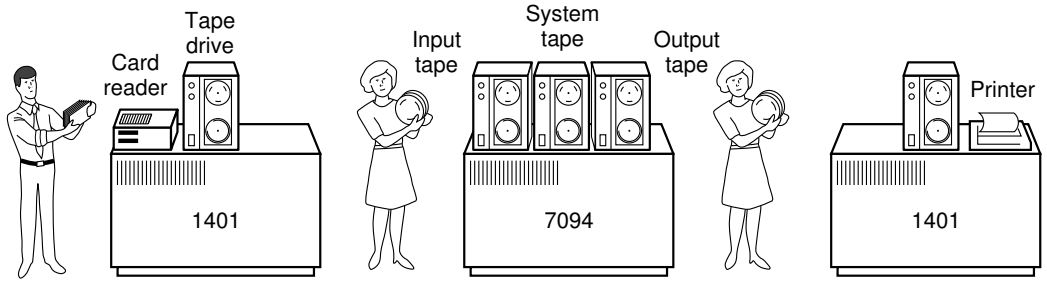
- Programma di controllo
  Controlla l'esecuzione dei programmi e le operazioni sui dispositivi di I/O. Condivisione rispetto al tempo e rispetto allo spazio

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
- Molto sicuri
- Uso inefficiente di risorse assai costose
  - Bassa utilizzazione della CPU
  - Molto tempo impiegato nel setup dei programmi

Semplici Sistemi Batch

- Assumere un operatore
- Utente $\neq$ operatore
- Aggiungere un lettore di schede



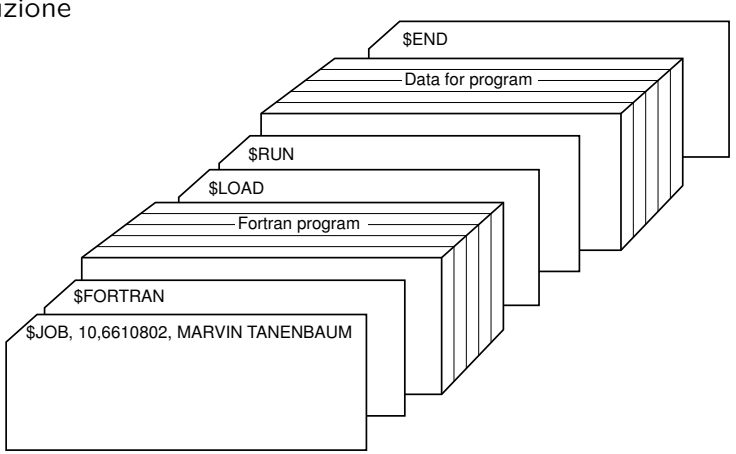
- Ridurre il tempo di setup raggruppando i job simili (*batch*)
- Sequenzializzazione automatica dei job – automaticamente, il controllo passa da un job al successivo. Primo rudimentale sistema operativo
- Monitor residente
  - inizialmente, il controllo è in monitor
  - poi viene trasferito al job
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

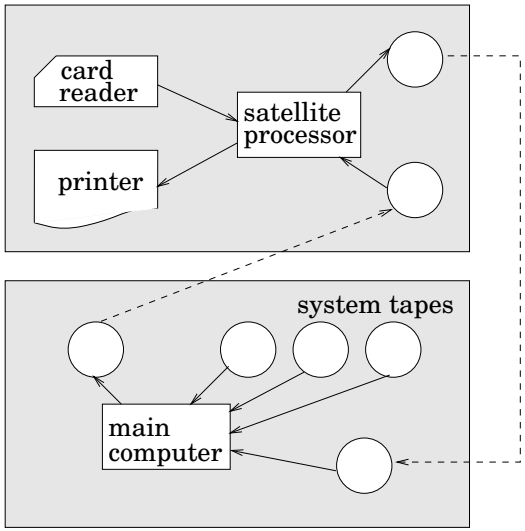


- Caratteri speciali distinguono le schede di controllo dalle schede di programma o di dati.

Schede di controllo (Cont.)

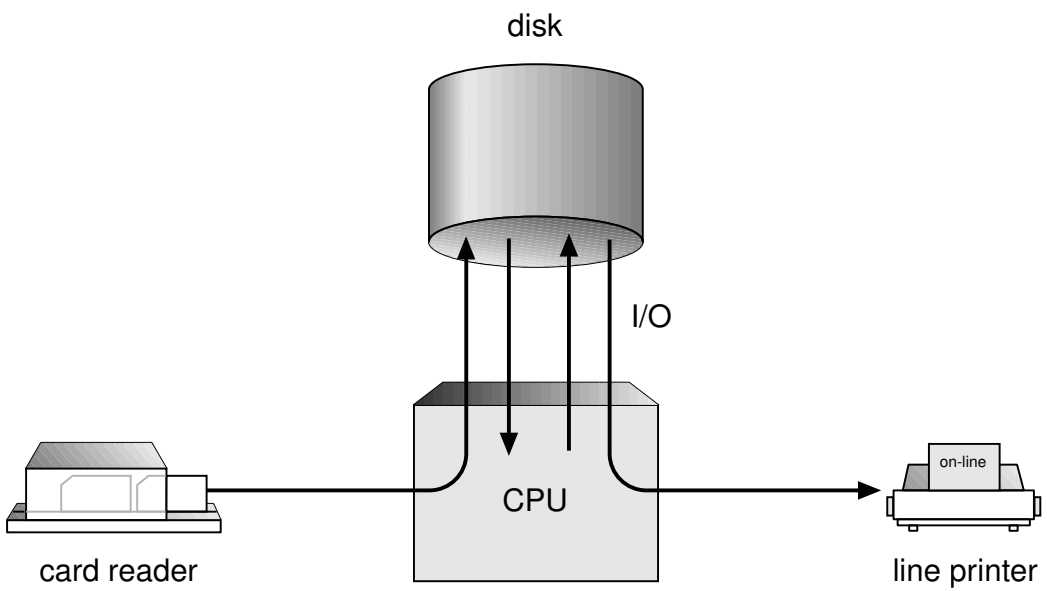
- Una parte del monitor residente è
  - Inteprete delle schede di controllo – responsabile della lettura e esecuzione delle istruzioni sulle schede di controllo
  - Loader – carica i programmi di sistema e applicativi in memoria
  - Driver dei dispositivi – conoscono le caratteristiche e le proprietà 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.

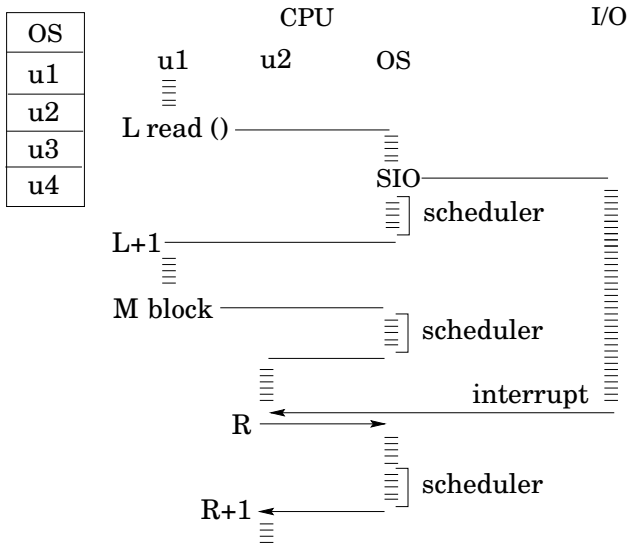
Spooling



- 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)
- Possono adottare tecnologie sviluppate per sistemi operativi più grandi; 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

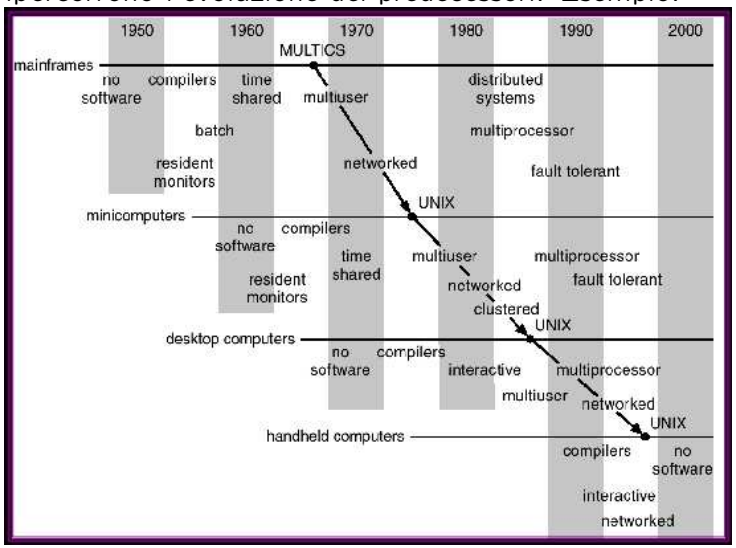
- Distribuzione della computazione tra più processori
- Sistemi *debolmente accoppiati* – ogni processore ha la sua propria memoria; i processori comunicano tra loro attraverso linee di comunicazione (e.g., bus ad alta velocità, linee telefoniche, fibre ottiche,. . .)
- In un sistema operativi di rete, 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. . .

L'ontogenesi riassume la filogenesi

L'hardware e il software (tra cui i sistemi operativi) in ogni nuova classe di calcolatori ripercorrono l'evoluzione dei predecessori. Esempio:



Lo zoo

Diversi obiettivi e requisiti a seconda delle situazioni

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

Sistemi operativi per mainframe

- Enormi quantità di dati ($> 1TB$)
- 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 NUMA o NORMA (migliaia di CPU)
- Job di calcolo intensivo
- Elaborazione "batch" non interattiva
- Esempi: Unix, o ad hoc

Sistemi per server

- Sistemi multiprocessore con spesso più di una CPU in comunicazione stretta.
- Degrado graduale delle prestazioni in caso di guasto (*fail-soft*)
- Riconfigurazione hardware a caldo
- 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
  - 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.

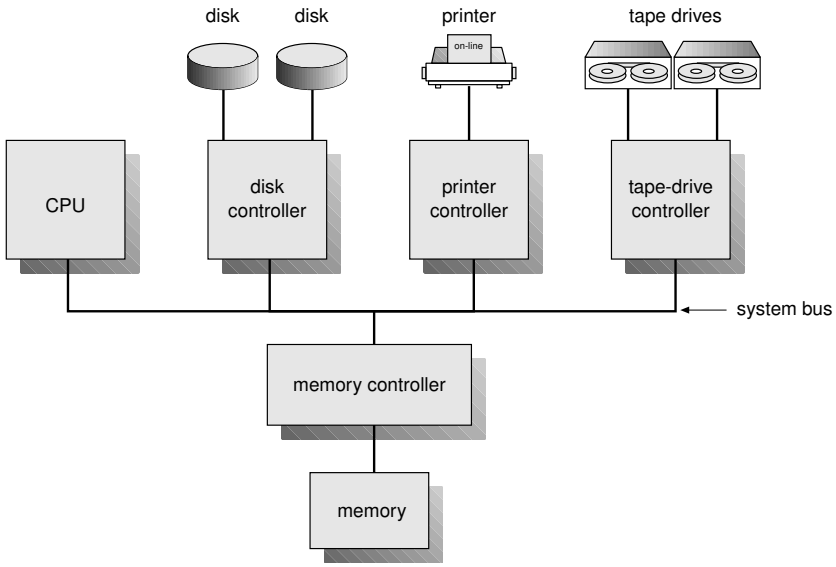
Sistemi operativi per smart card

- 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 avviene 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 ciclo di attesa (*busy wait*); accede alla memoria
  - al più una richiesta di I/O è eseguita alla volta; non ci sono I/O paralleli

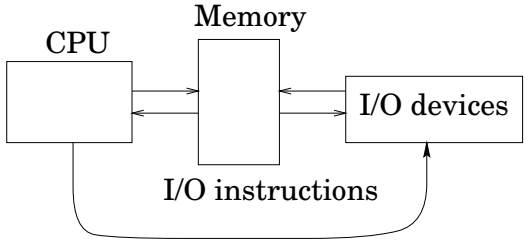
- I/O asincrono: dopo che l'I/O è partito, il controllo ritorna al programma utente senza aspettare che l'I/O venga completato

  – *chiamata di sistema* (*System call*) – richiede al sistema operativo di sospendere il processo in attesa del completamento dell'I/O.

  – 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

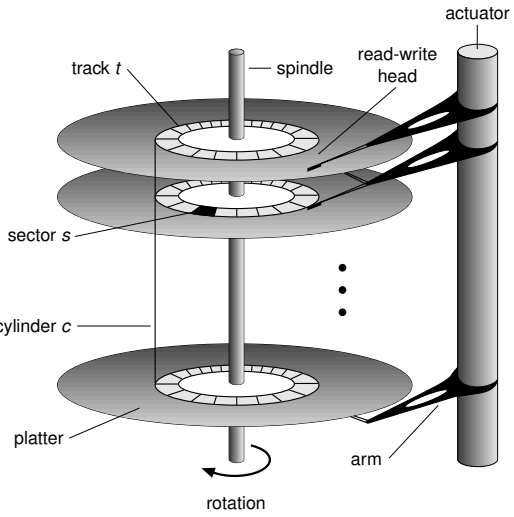
- Memoria principale – la memoria che la CPU può accedere direttamente.

- Memoria secondaria – estensione della memoria principale che fornisce una memoria non volatile (e solitamente più grande)

Dischi magnetici

piatti di metallo rigito ricoperti di materiale ferromagnetico, in rotazione

- la superfice del disco è logicamente divisa in *tracce*, che sono sottodivise in *settori*
- Il *controller dei dischi* determina l'interazione logica tra il dispositivo ed il calcolatore

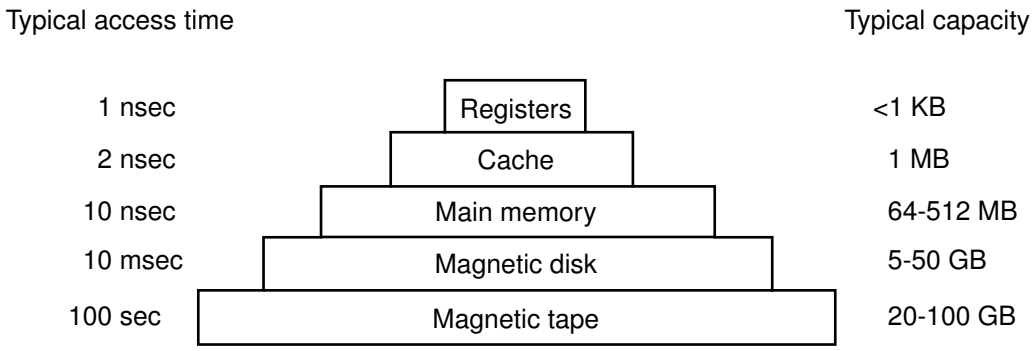


Gerarchia della Memoria

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

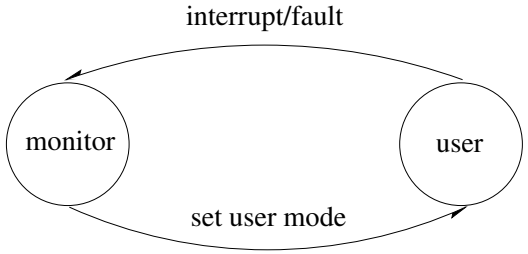
- Funzionamento in dual-mode
- Protezione dell'I/O
- Protezione della Memoria
- Protezione della CPU

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

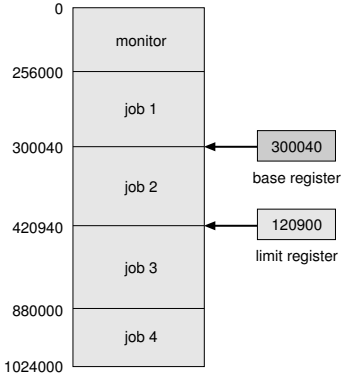
Protezione dell'I/O

- 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)

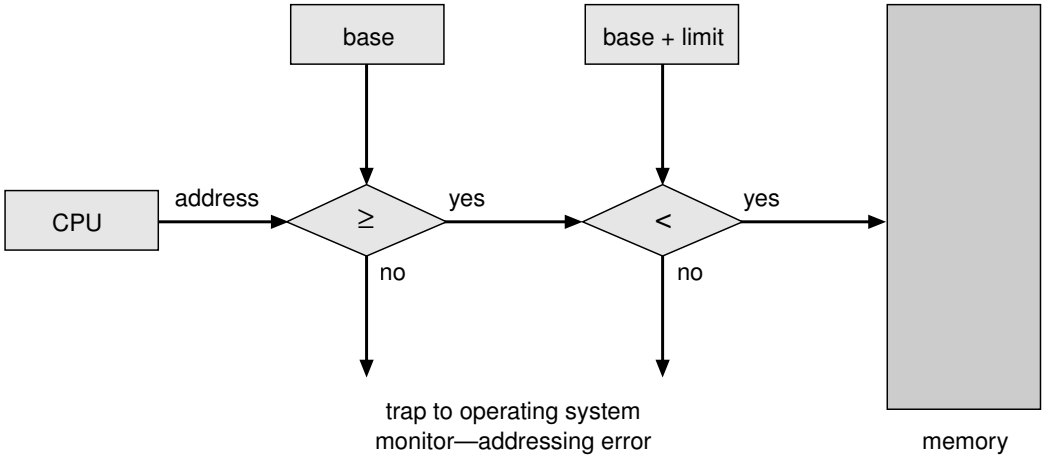
Protezione della Memoria

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
- *Progettazione ed Implementazione del Sistema*
- *Generazione del Sistema*

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 *push*a i parametri sullo *stack*, e il sistema operativo li *pop*pa.

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,...

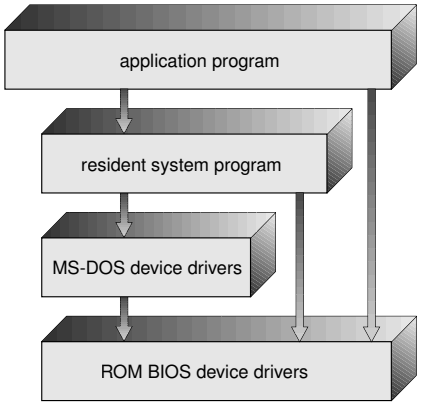
Comunicazioni: creare/cancellare connessioni, spedire/ricevere messaggi,...

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 – L'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 – L'approccio semplice (Cont

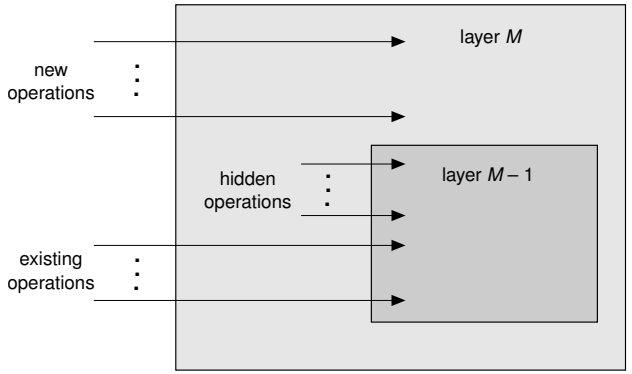
- UNIX – limitato dalle funzionalità hardware, lo UNIX originale aveva una debole strutturazione. Consiste almeno in due parti ben separate:
  - Programmi di sistema
  - Il kernel
    - consiste in tutto ciò che sta tra le system call e l'hardware
    - implementa il file system, lo scheduling della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

Struttura dei Sistemi Operativi – Unix originale

(the users)		
shells and commands compilers and interpreters system libraries		
system-call interface to the kernel		
signals terminal handling character I/O system terminal drivers	file system swapping block I/O system disk and tape drivers	CPU scheduling page replacement demand paging virtual memory
kernel interface to the hardware		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

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.

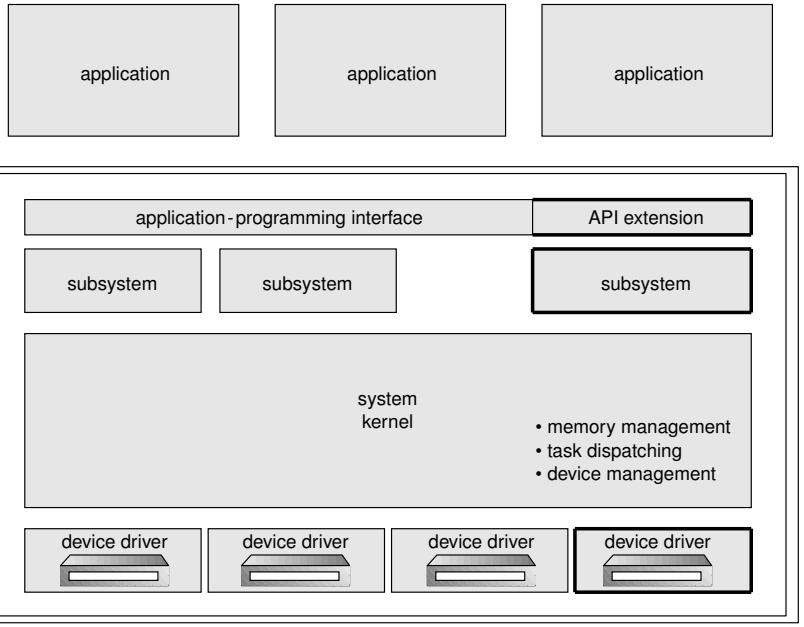


Struttura dei sistemi operativi – Stratificazione di THE OS

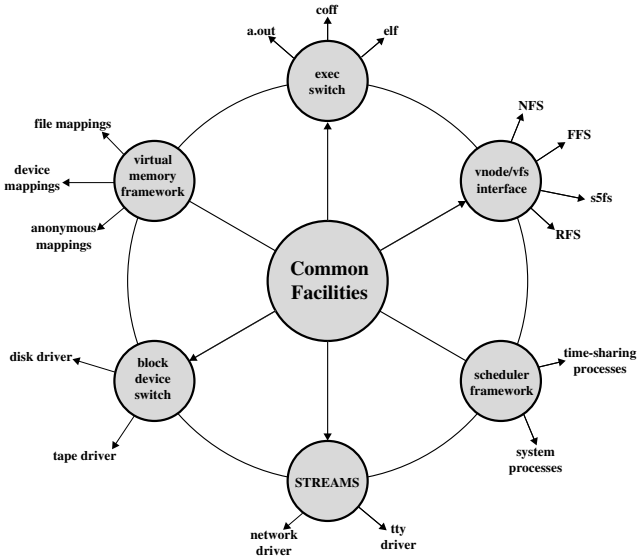
- La prima stratificazione fu usata nel SO THE. Sei strati come segue:

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

Struttura dei Sistemi Operativi – Stratificazione di OS/2



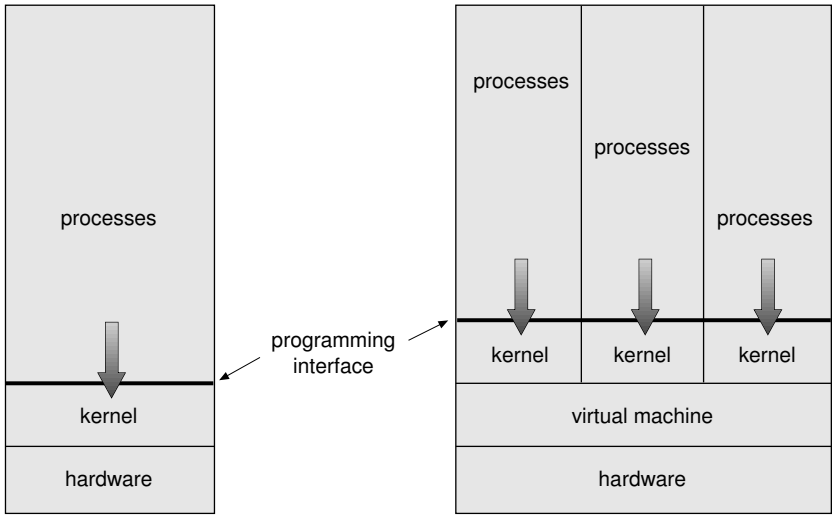
Struttura dei Sistemi Operativi – Solaris



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

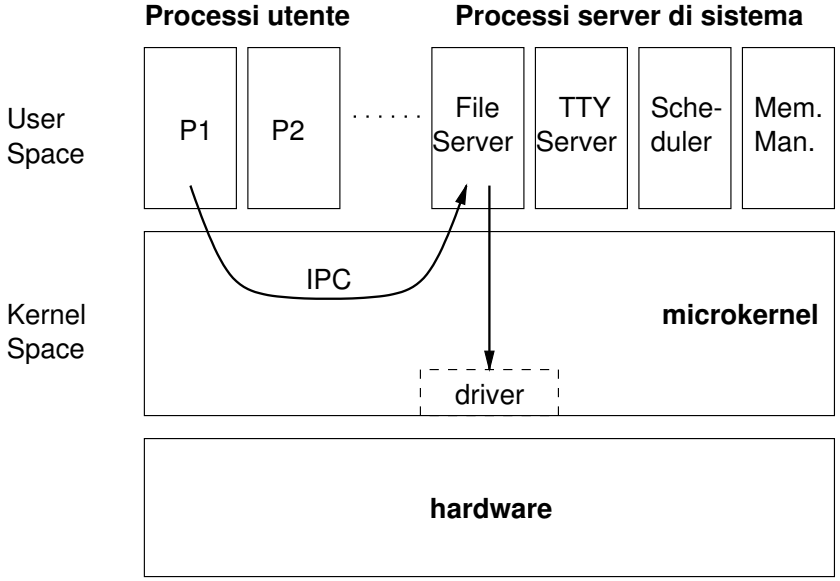
- I kernel tradizionali (monolitici) sono poco flessibili
- Distinguere tra *meccanismi* e *politiche*:
  - i meccanismi determinano *come* fare qualcosa;
  - le politiche determinano *cosa* deve essere fatto.

  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.

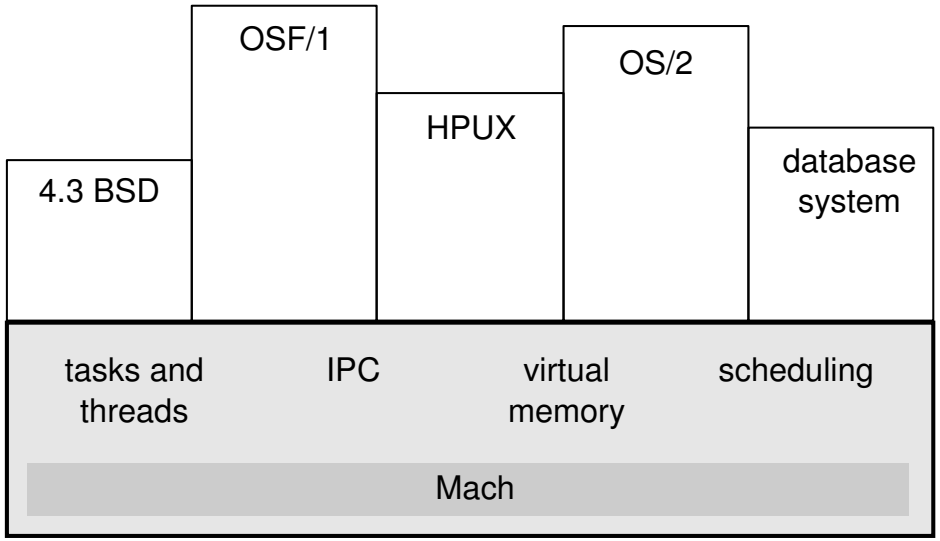
Sistemi con Microkernel

- *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 ...)

Microkernel: funzionamento di base



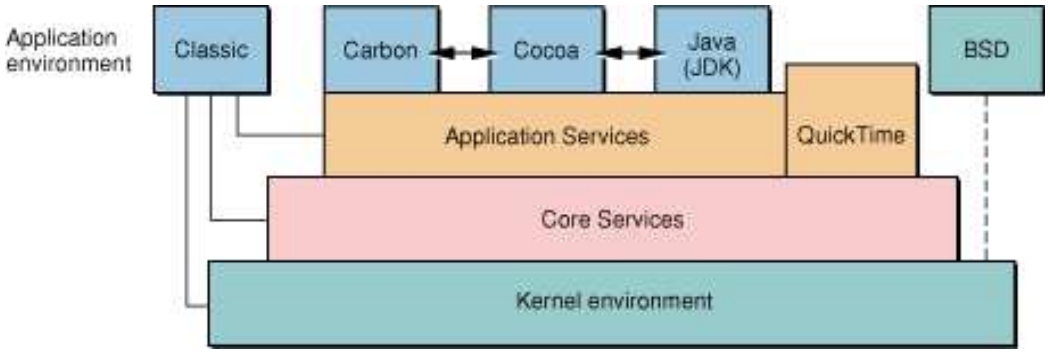
Microkernel: Mach



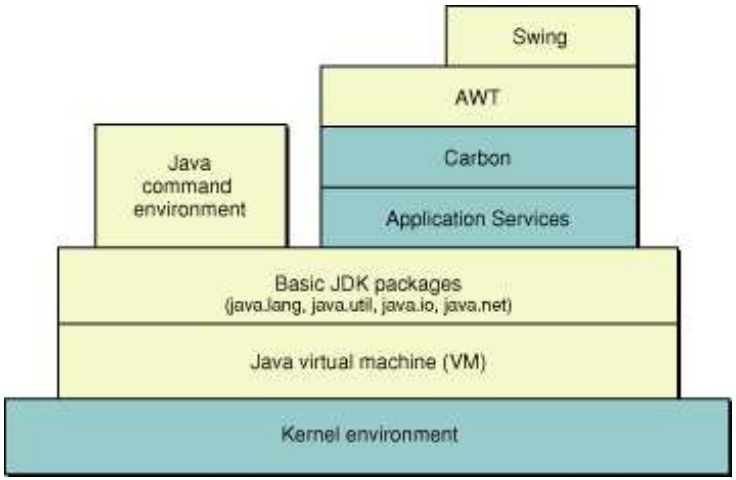
Soluzione stratificata/microkernel: Mac OS X

- Application (or execution) environments: Carbon, Cocoa, Java, Classic, and BSD Commands.
- Application Services. Servizi di sistema usati da tutti gli ambienti applicativi (collegati con la GUI). Quartz, QuickDraw, OpenGL. . . .
- Core Services. Servizi comuni, non collegati alla GUI. Networking, tasks, . . .
- Kernel environment. Mach 3.0 per task, thread, device drivers, gestione della memoria, + BSD che implementa rete, file system, thread POSIX.

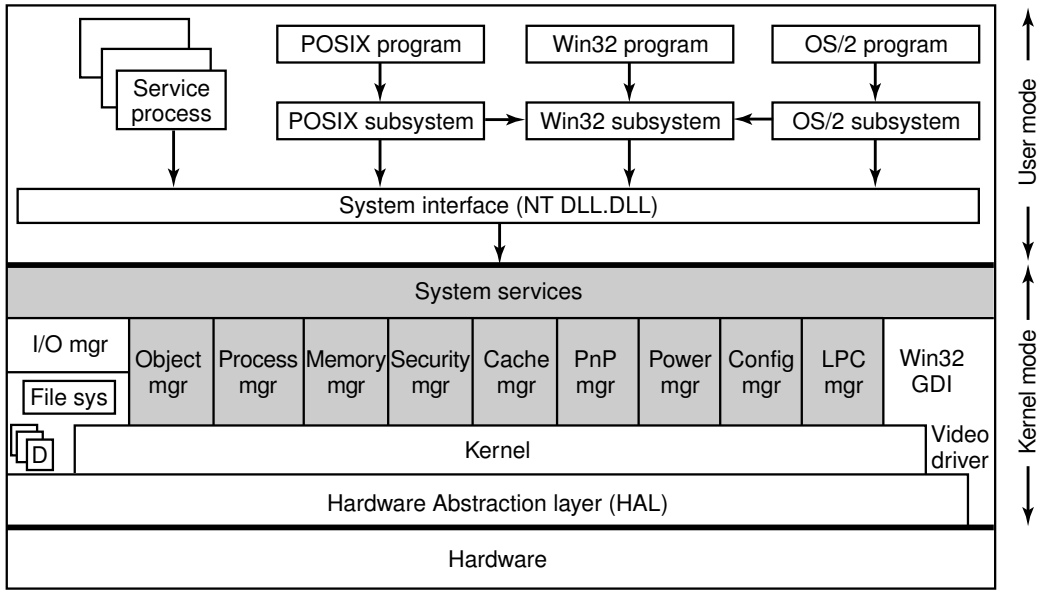
Soluzione stratificata/microkernel: Mac OS X



Soluzione stratificata/microkernel: Mac OS X



Soluzione stratificata non più microkernel: Windows 2000



Processi e Thread

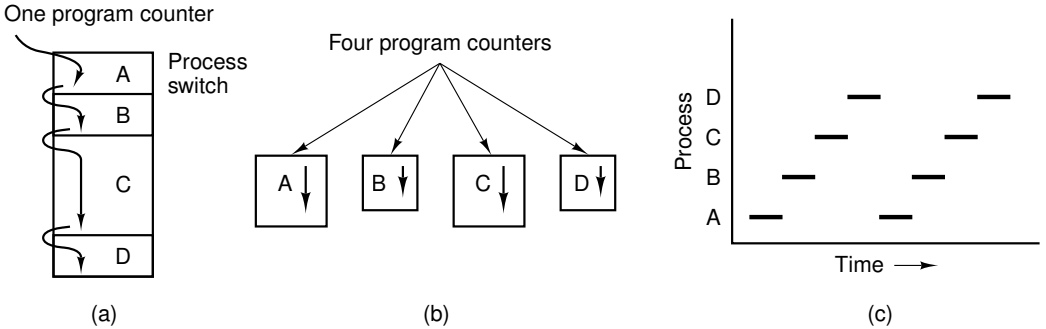
- Concetto di processo
- Operazioni sui processi
- Stati dei processi
- Threads
- Schedulazione dei processi

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

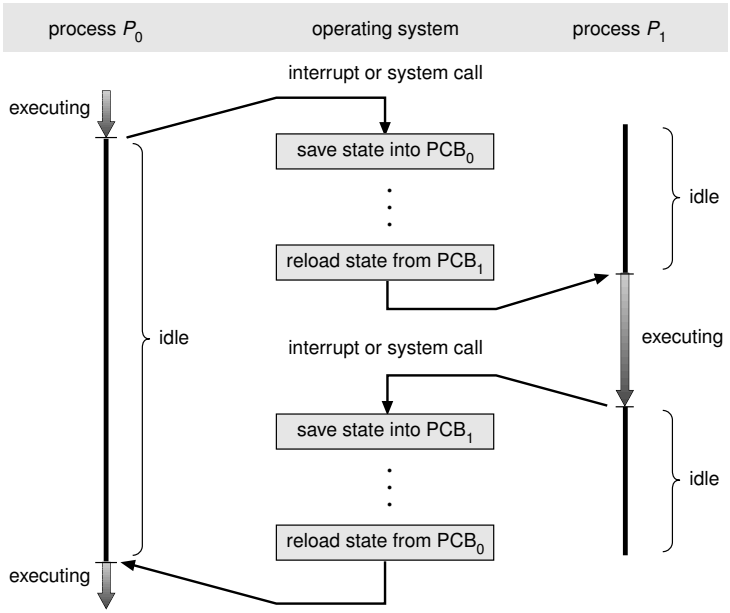
Multiprogrammazione

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



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.
- 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

Creazione dei processi

- Quando viene creato un processo
  - Al boot del sistema (intrinseci, daemon)
  - Su esecuzione di una system call apposita (es., `fork()`)
  - Su richiesta da parte dell'utente
  - Inizio di un job batch

  La generazione dei processi indica una naturale gerarchia, detta *albero di processi*.
- Esecuzione: alternative
  - Padre e figli sono in esecuzione concorrente
  - Il padre attende che i figli terminino per riprendere l'esecuzione

- Condivisione delle risorse:
  - 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: alternative
  - I figli duplicano quello del padre (es: `fork()`)
  - I figli caricano sempre un programma (es: `CreateProcess()`)

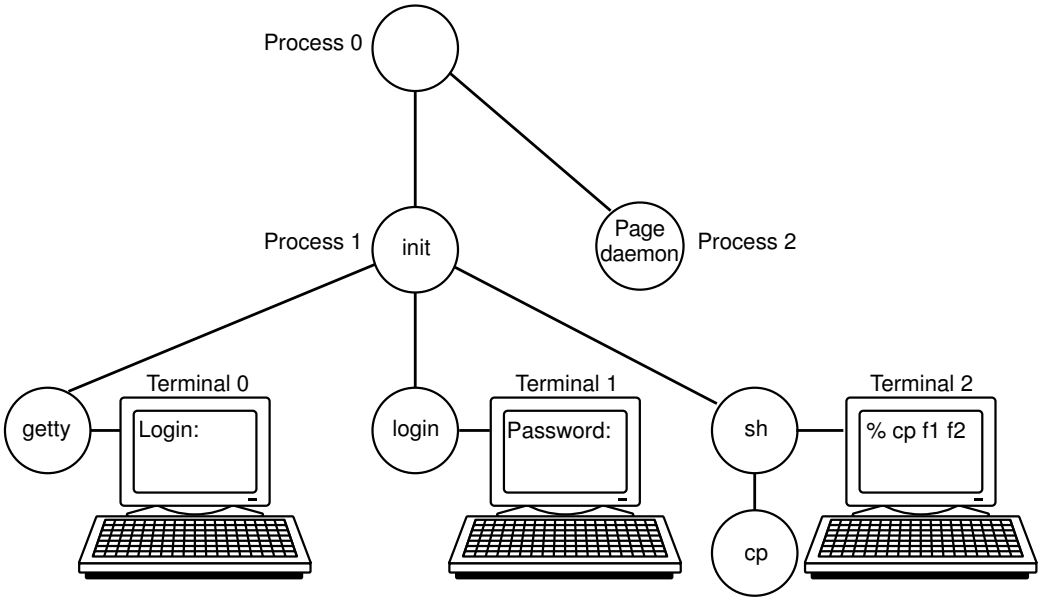
Terminazione dei Processi

- Terminazione volontaria—normale o con errore (**exit**). I dati di output vengono ricevuti dal processo padre (che li attendeva con un **wait**).
- 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 (p.e., segnali possono essere mandati solo all'interno di un gruppo, o ad un intero gruppo).
- In UNIX: tutti i processi discendono da `init` (PID=1). Se un parent 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 una *handle* del figlio, che comunque può essere passata.



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

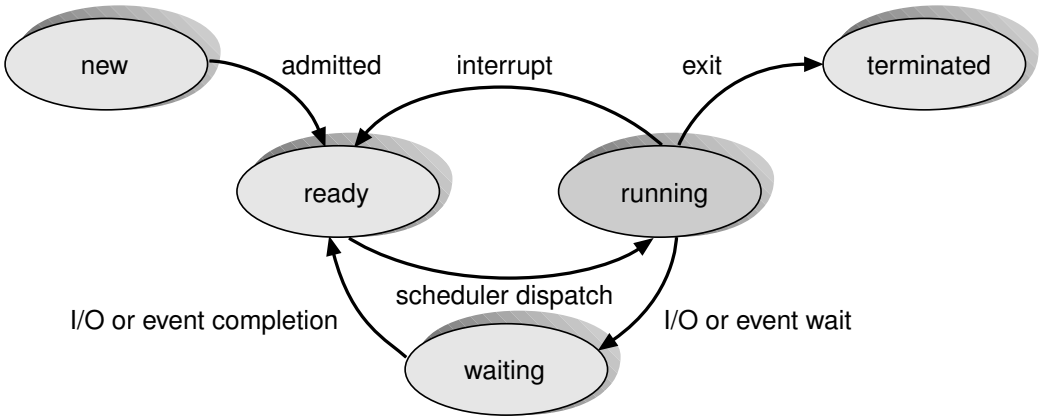
ready: il processo attende di essere assegnato ad un processore

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

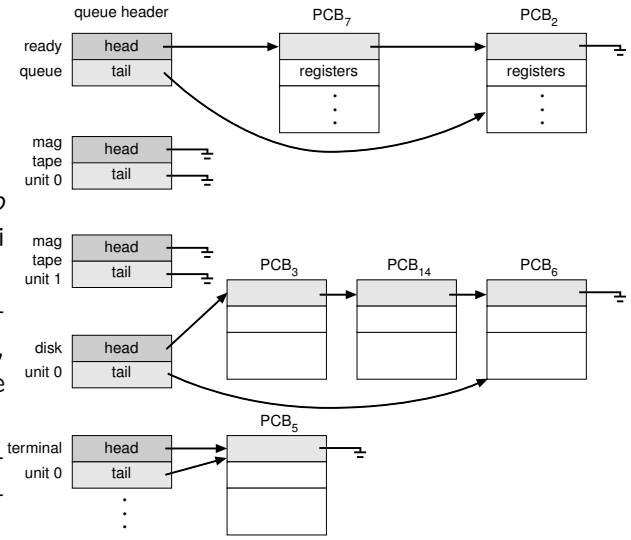
- Stato del processo
- Dati identificativi (del processo, dell'utente)
- Program counter
- Registri della CPU
- Informazioni per lo scheduling della CPU
- Informazioni per la gestione della memoria
- Informazioni di utilizzo risorse
  - tempo di CPU, memoria, file...
  - eventuali limiti (*quota*)
-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

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.

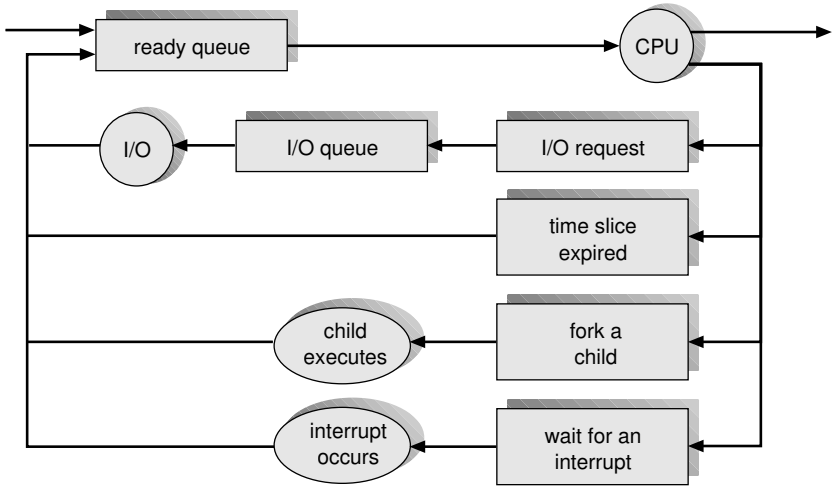
Code di scheduling dei processi



- 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

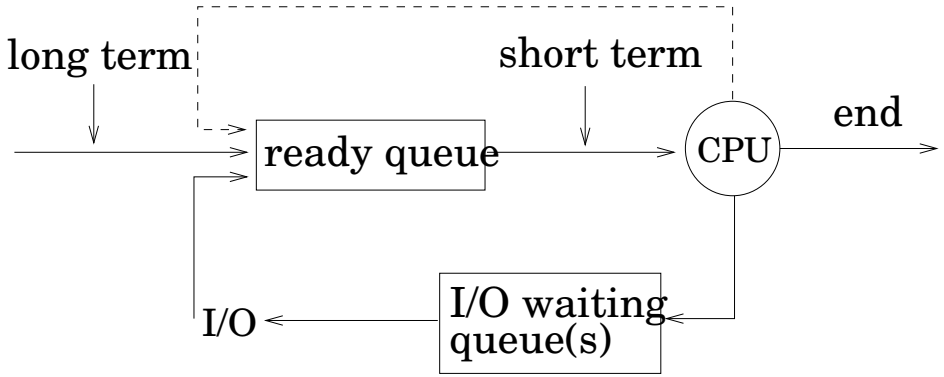
I processi, durante l'esecuzione, migrano da una coda all'altra



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.

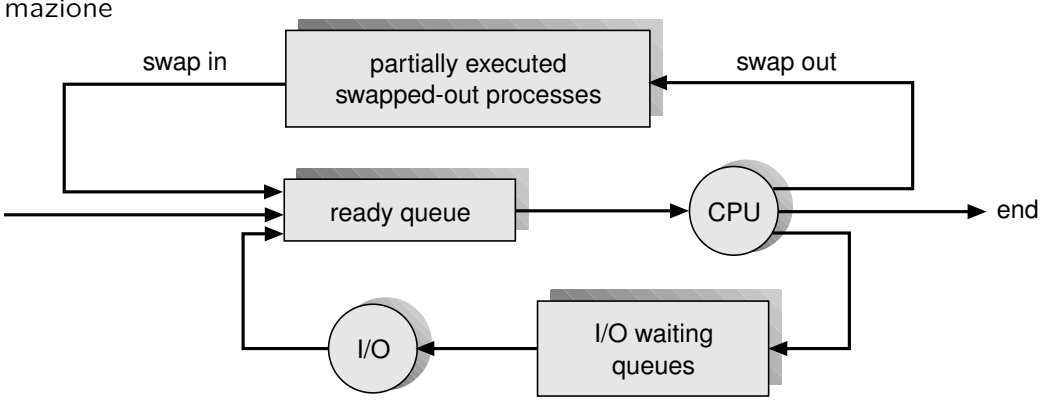


Gli Scheduler (Cont.)

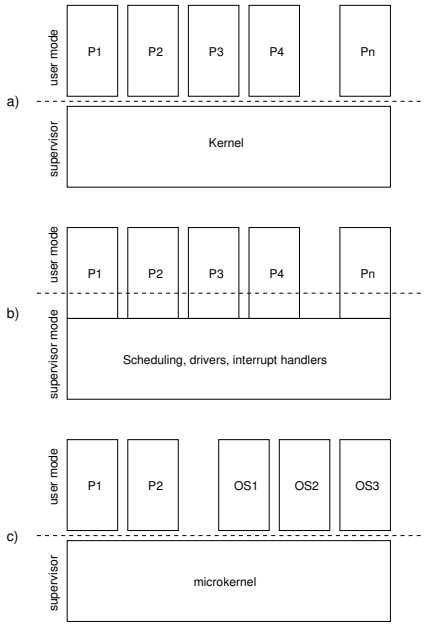
- Lo scheduler di breve termine è invocato molto frequentemente (decine di volte al secondo) ⇒ deve essere veloce
- Lo scheduler di lungo termine è invocato raramente (secondi, minuti) ⇒ può essere lento e sofisticato
- I processi possono essere descritti come
  - I/O-bound: lunghi periodi di I/O, brevi 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.

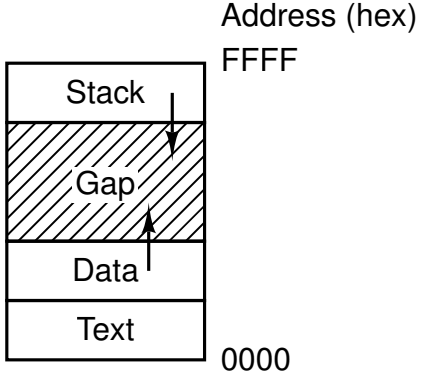
Esecuzione kernel all'interno dei processi

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.

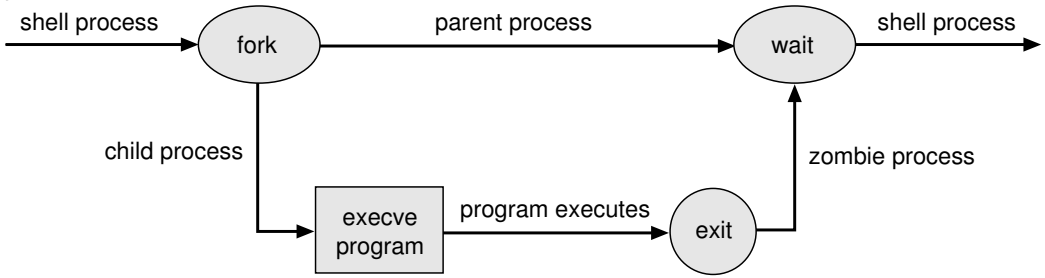


Creazione di un processo: la chiamata fork(3)

```
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);
 }
}
```



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

I segnali POSIX

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

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: PID, PPID
  - identificatori dell'utente: *real UID, effective UID*
  - 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
  - contiene dati relativi alla gestione della memoria virtuale per il text

Process Control Block (Cont.)

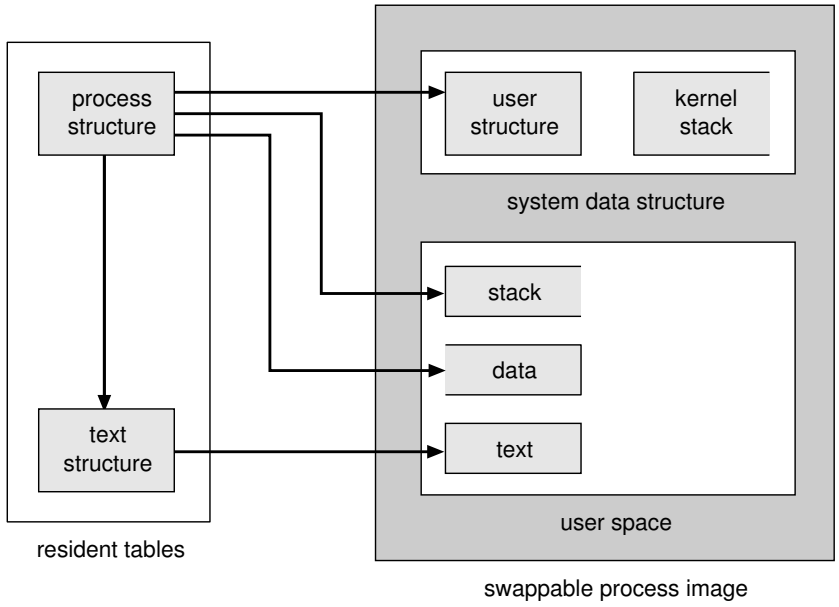
- Le informazioni sul processo che sono richieste solo quando il processo è residente sono mantenute nella *user structure* (o *u structure*).

  Fa parte dello spazio indirizzi modo user, read-only (ma scrivibile dal kernel) e contiene (tra l'altro)
  - real UID, effective UID, real GID, effective GID
  - gestione di ogni segnali (exit, ignore, esecuzione di una funzione)
  - terminale di controllo
  - risultati/errori delle system call
  - tabella dei file aperti
  - limiti del processo
  - mode mask (*umask*)

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



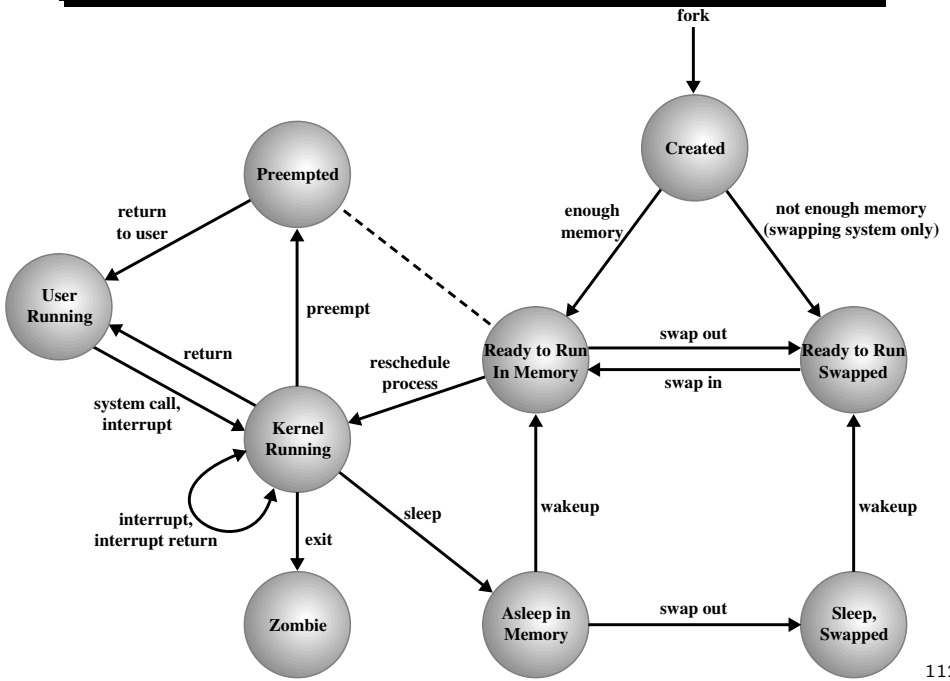
Creazione di un processo

- 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

Creazione di un processo (Cont.)

- La **vfork** *non* copia i segmenti data e stack; vengono condivisi
  - il system data segment e la process structure vengono creati
  - il processo padre rimane sospeso finché il figlio non termina o esegue una **execve**
  - il processo padre usa **vfork** per produrre il figlio, che usa **execve** per cambiare immediatamente lo spazio di indirizzamento virtuale — non è necessario copiare data e stack segments del padre
  - comunemente usata da una shell per eseguire un comando e attendere il suo completamento:
  - efficiente per processi grandi (risparmio di tempo di CPU), ma potenzialmente pericolosa delicata (le modifiche fatte dal processo figlio prima della **execve** si riflettono sullo spazio indirizzi del padre)

Diagramma degli stati di un processo in UNIX



Stati di un processo in UNIX (Cont.)

- User running: esecuzione in modo utente
- Kernel running: esecuzione in modo kernel
- Ready to run, in memory: pronto per andare in esecuzione
- Asleep in memory: in attesa di un evento; processo in memoria
- Ready to run, swapped: eseguibile, ma swappato su disco
- Sleeping, swapped: in attesa di un evento; processo swappato
- Preempted: il kernel lo blocca per mandare un altro processo
- Zombie: il processo non esiste più, si attende che il padre riceva l'informazione dello stato di ritorno

Dai processi...

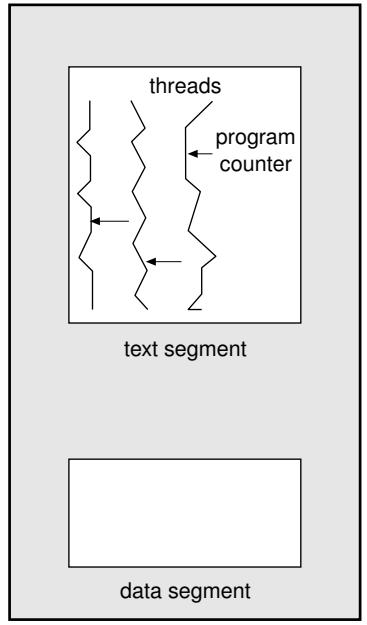
I processi finora studiati incorporano due caratteristiche:

- 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 (UID, GID)...
- 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,...

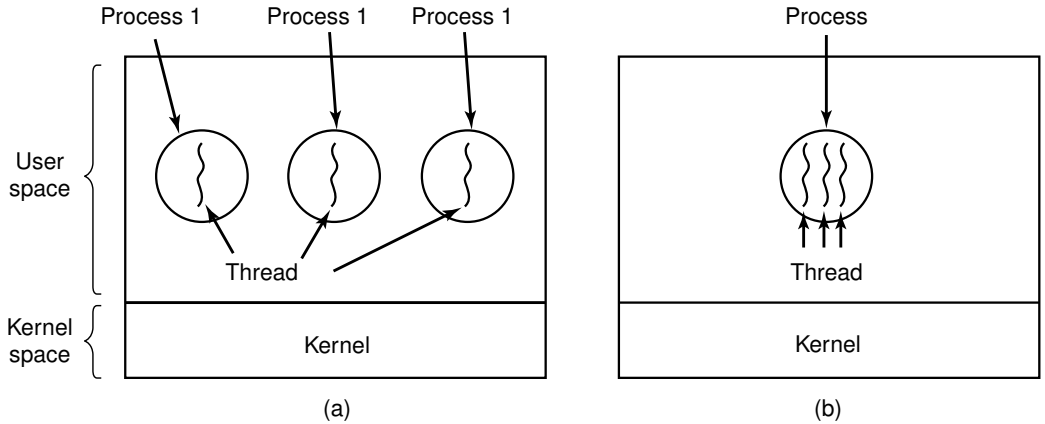
Queste due componenti sono in realtà *indipendenti*

...ai thread

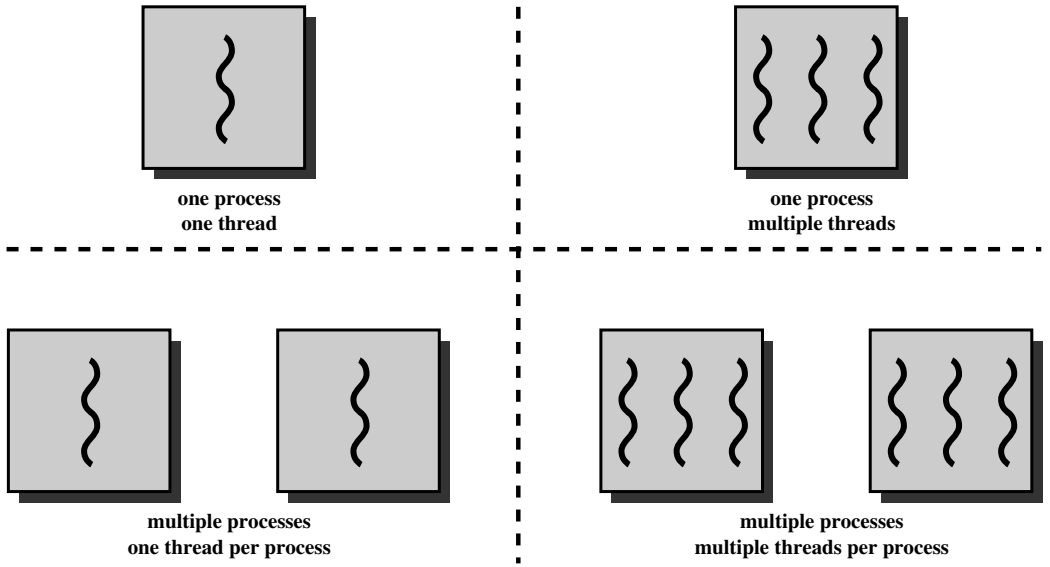
- Un *thread* (o *processo leggero*, *lightweight process*) è una unità di esecuzione:
  - program counter, insieme registri
  - stack del processore
  - stato di esecuzione
- Un thread condivide con i thread suoi pari una unità di allocazione risorse:
  - il codice eseguibile
  - i dati
  - le risorse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



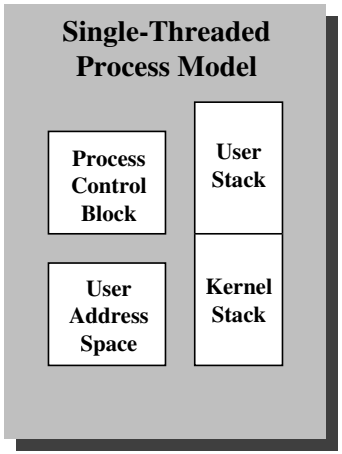
Esempi di thread

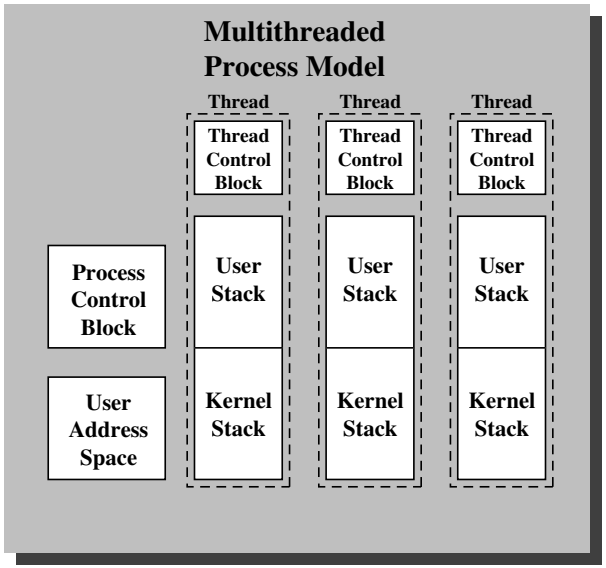


Processi e Thread: quattro possibili scenari



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)

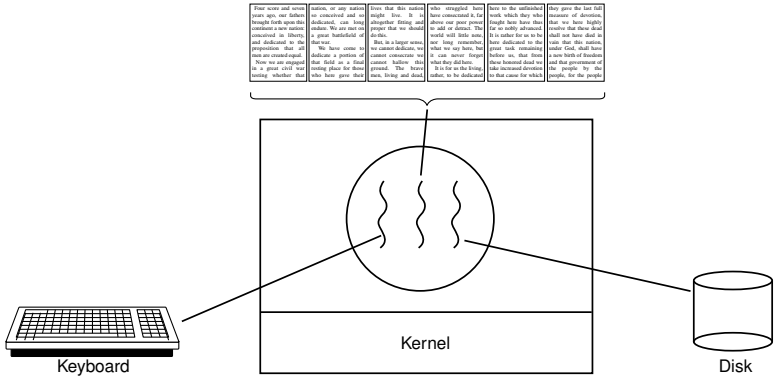
Condivisione di risorse tra thread (Cont.)

- 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

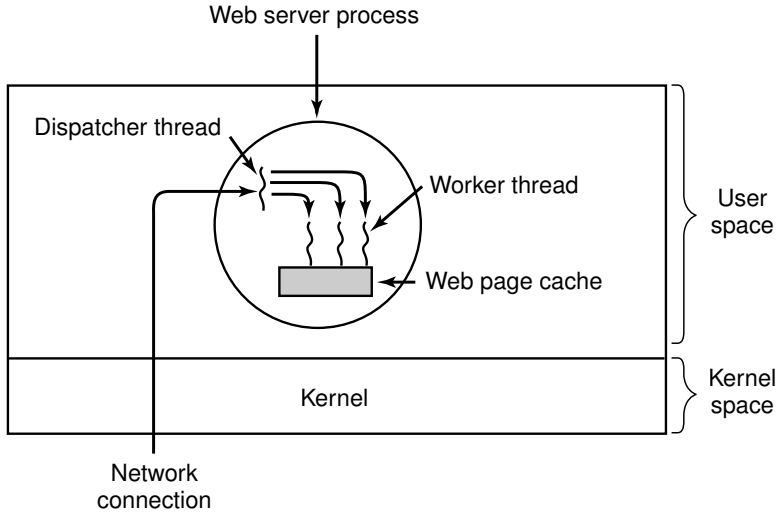
Lavoro foreground/background: mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background. Spreadsheets (ricalcolo automatico), word processor (reimpaginazione, controllo ortografico,. . .)

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, . . .

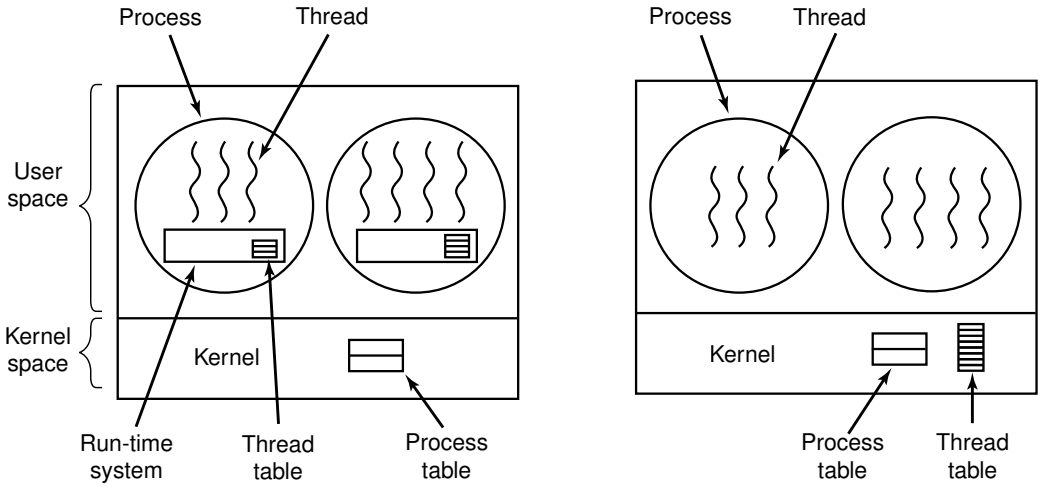
  blocco: un thread si ferma, e l'esecuzione passa ad un altro thread/processo. Può essere volontario (`thread_yield`) o su richiesta di un evento;

  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

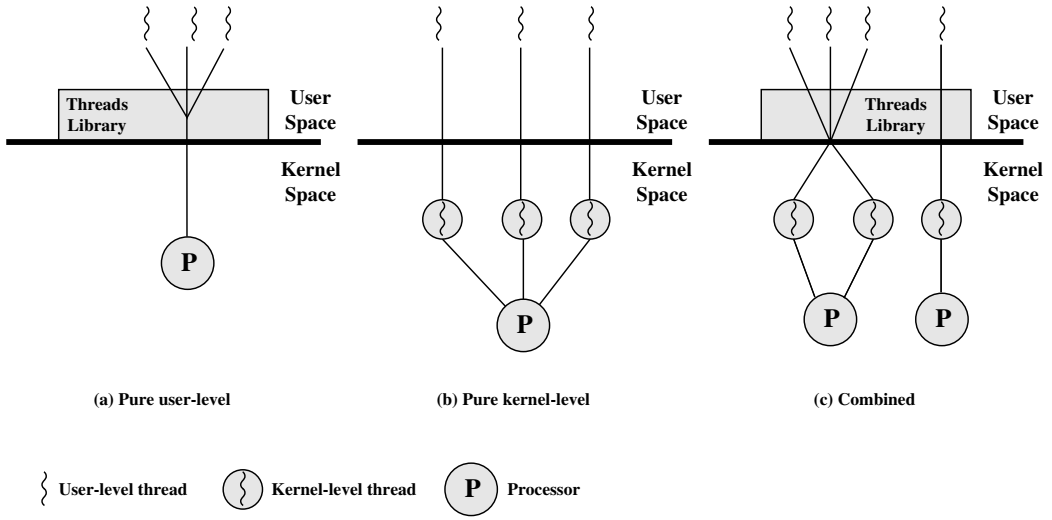
Svantaggi:

- 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:

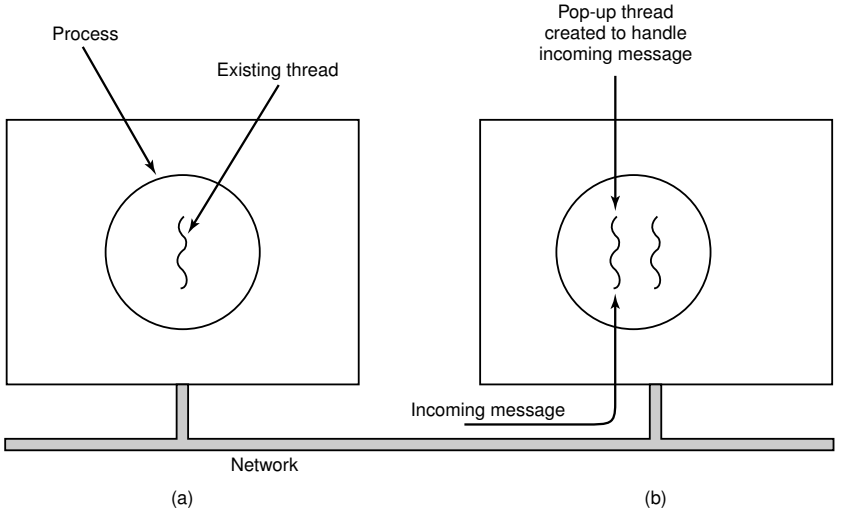
- tutti quelli dei ULT e KLT
- alta flessibilità: il programmatore può scegliere di volta in volta il tipo di thread che meglio si adatta

Svantaggio: portabilità

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

Thread pop-up

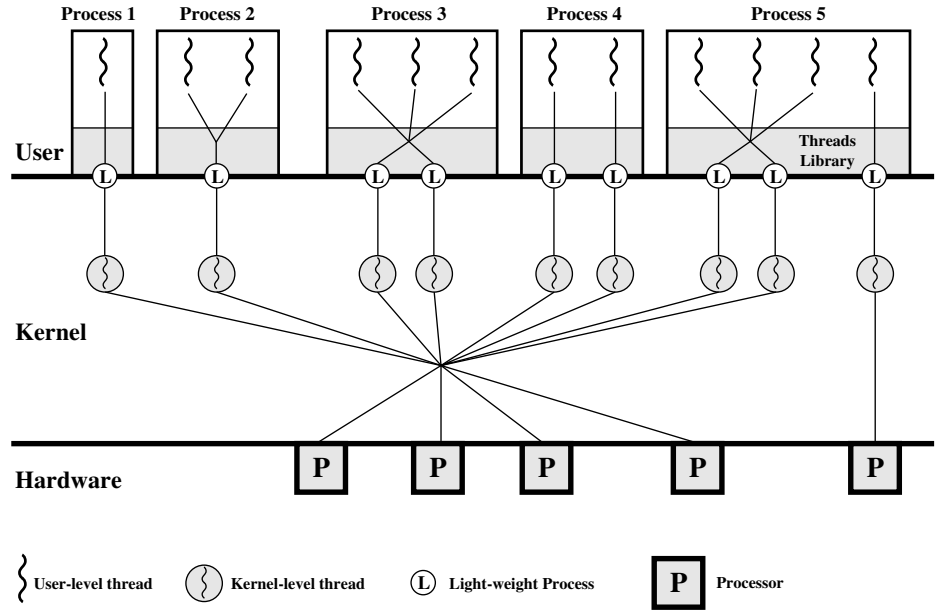
- I thread *pop-up* sono thread creati in modo asincrono da eventi esterni.



(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

Esempio: I Thread di Solaris



I thread di Solaris (cont.)

Nel gergo Solaris:

Processo: il normale processo UNIX (spazio indirizzi utente, stack, PCB, . . .)

User-level thread: implementato da una libreria a livello utente. Invisibili al kernel.

Lightweight process: assegnamento di ULT ad un thread in kernel. Ogni LWP supporta uno o più ULT, ed è gestito dal kernel.

Kernel thread: le entità gestite dallo scheduler.

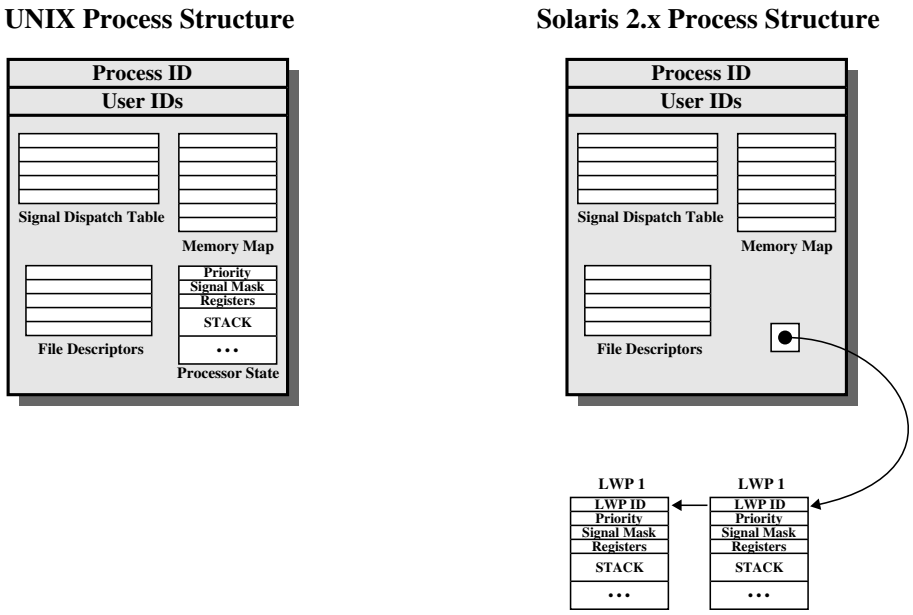
I thread in Solaris (Cont.)

È possibile specificare il grado di parallelismo *logico* e *fisico* del task

- task con parallelismo logico hanno più ULT su un solo LWP
  $\Rightarrow$ comodità di progetto, efficienza di switch
- task con parallelismo fisico hanno più ULT su più LWP
  $\Rightarrow$ efficienza di esecuzione
- task con necessità real-time possono fissare un ULT ad un LWP (*pinning*) (eventualmente, soggetto a politica `SCHED_RT`)

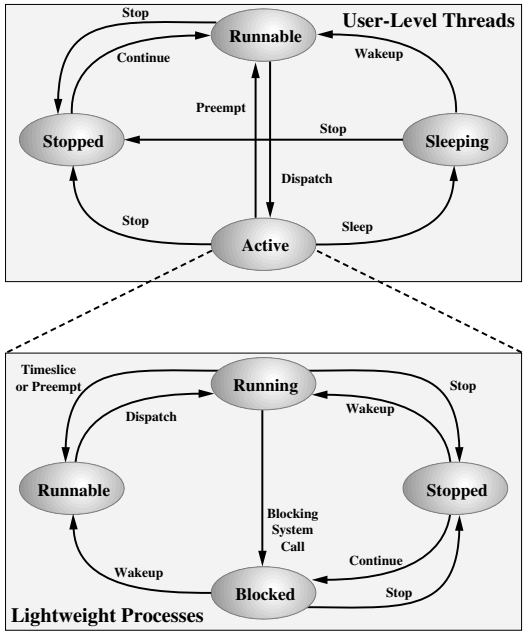
I task di sistema (swapper, gestione interrupt, . . .) vengono implementati come kernel thread (anche pop-up) non associati a LWP.

Strutture per processi e LWP in Solaris



Stati di ULT e LWP di Solaris

- Sleep: un ULT esecue una primitiva di sincronizzazione e si sospende
- Wakeup: la condizione viene soddisfatta
- Dispatch: un LWP è libero, il thread viene selezionato
- Preempt: si sblocca un ULT a priorità maggiore
- Yielding: un ULT rilascia il controllo eseguendo `thr_yield`



Processi e Thread di Linux

Linux fornisce una peculiare system call che generalizza la `fork()`:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

I flag descrivono cosa il thread/processo figlio deve condividere con il parent

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID

A seconda dei flag, permette di creare un nuovo thread nel processo corrente, o un processo del tutto nuovo. P.e.: se tutto a 0, corrisponde a `fork()`.

Permette di implementare i thread a livello kernel.

Stati dei processi/thread di Linux

In `include/linux/sched.h`:

- Ready: pronto per essere schedulato
- Running: in esecuzione
- Waiting: in attesa di un evento. Due sottocasi: interrompibile (segnali non mascherati), non interrompibile (segnali mascherati).
- Stopped: Esecuzione sospesa (p.e., da SIGSTOP)
- Zombie: terminato, ma non ancora cancellabile

Processi e Thread di Windows 2000

Nel gergo Windows:

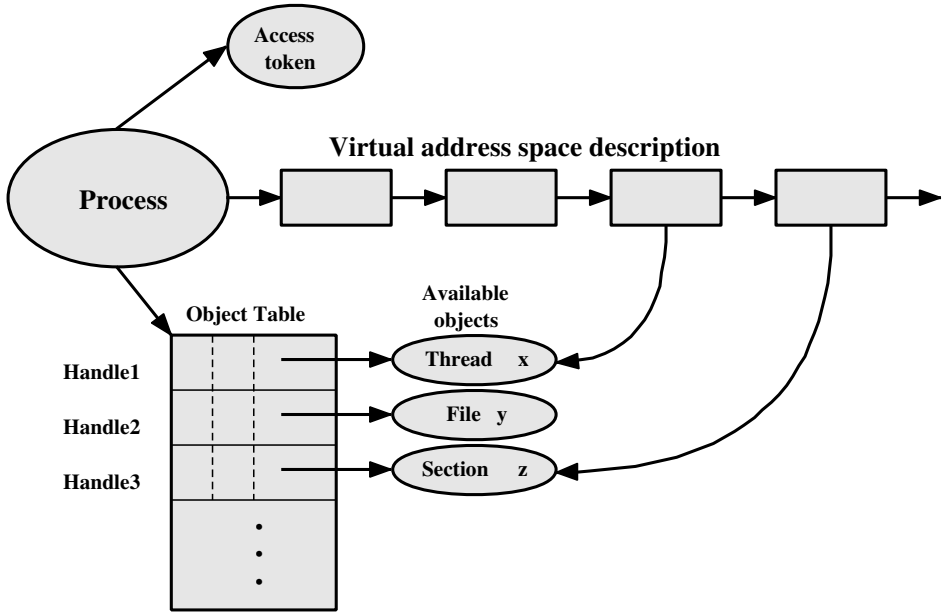
Job: collezione di processi che condividono quota e limiti

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.

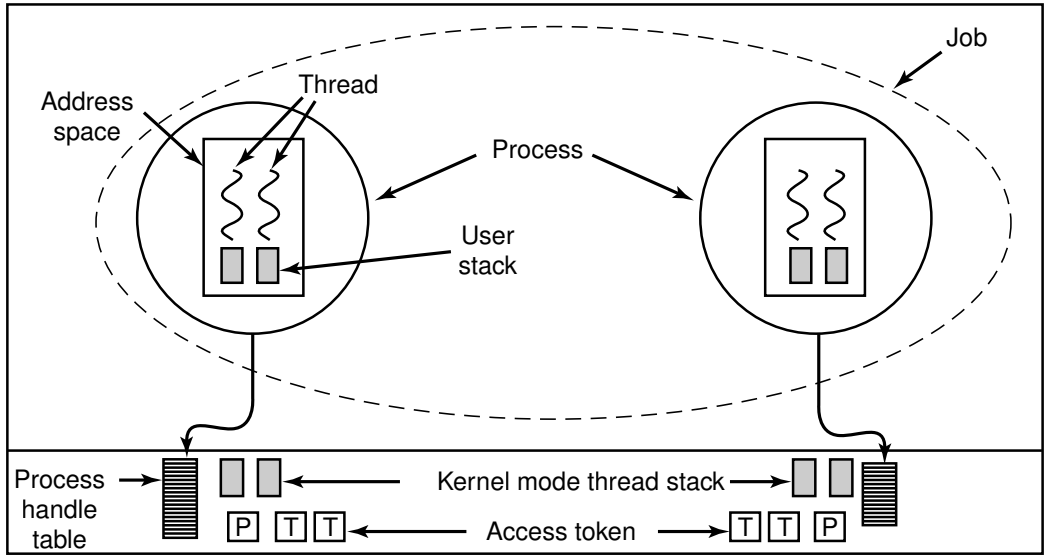
Thread: entità schedulata dal kernel. Alterna il modo user e modo kernel. Doppio stack. Creato con `CreateThread`.

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

Struttura di un processo in Windows 2000

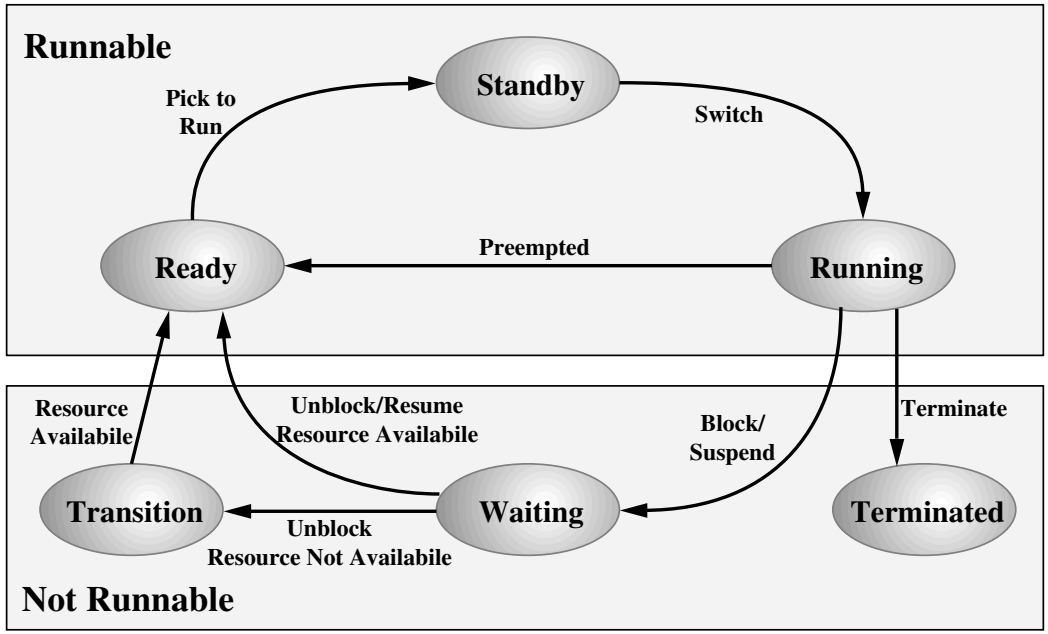


Job, processi e thread in Windows 2000



Stati dei thread di Windows

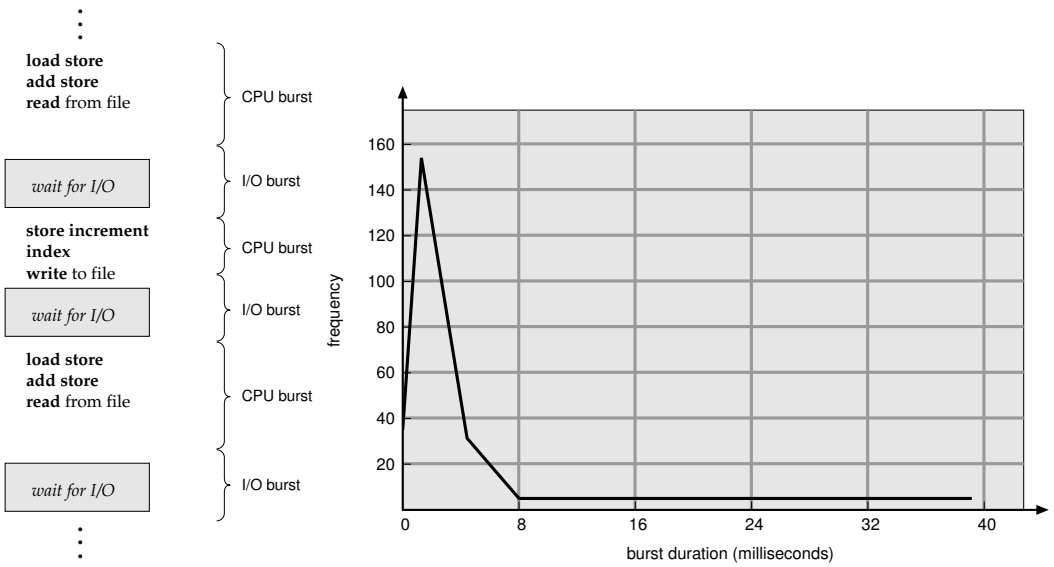
- Ready: pronto per essere schedulato
- Standby: selezionato per essere eseguito
- Running: in esecuzione
- Waiting: in attesa di un evento
- Transition: eseguibile, ma in attesa di una risorsa
- Terminated: terminato, ma non ancora cancellabile



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:* # di processi completati nell'unità di tempo
- *Tempo di turnaround:* 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

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

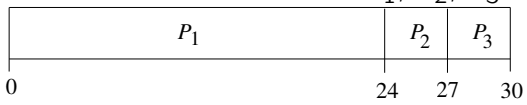
Nota: in generale, non esiste soluzione ottima sotto tutti gli aspetti

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

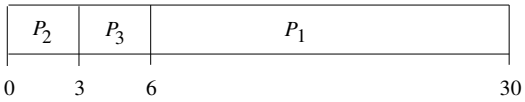
Diagramma di Gantt con l'ordine di arrivo P_1, P_2, P_3



- 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.

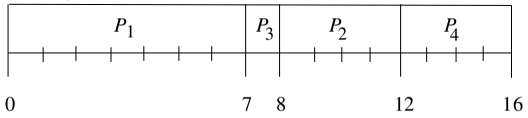
Scheduling Shortest-Job-First (SJF)

- 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)

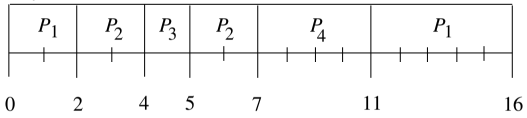


Tempo di attesa medio = (0 + 6 + 3 + 7)/4 = 4

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

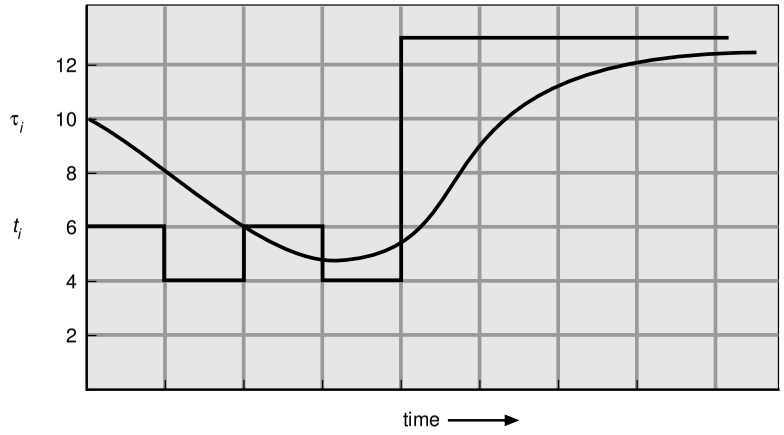
- Espandendo la formula:

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

- 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 $\equiv$ 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 (*aginng*) – con il passare del tempo, i processi non eseguiti aumentano la loro priorità

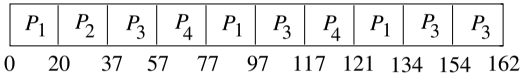
Round Robin (RR)

- 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

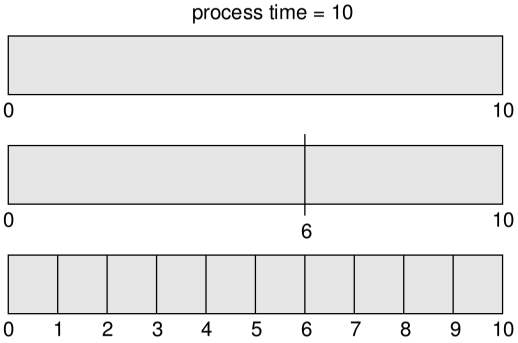
- Diagramma di Gantt



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

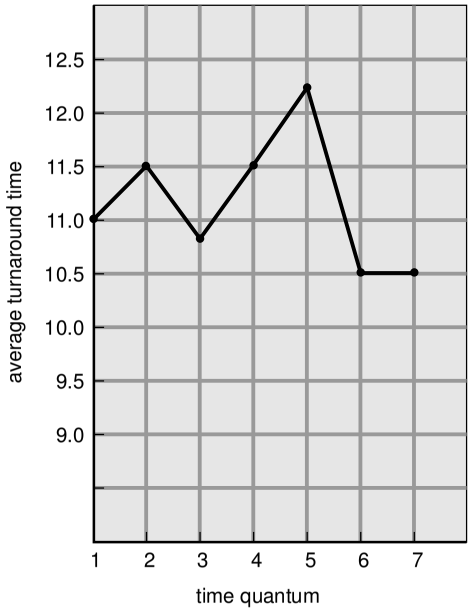
Prestazioni dello scheduling Round-Robin

- 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

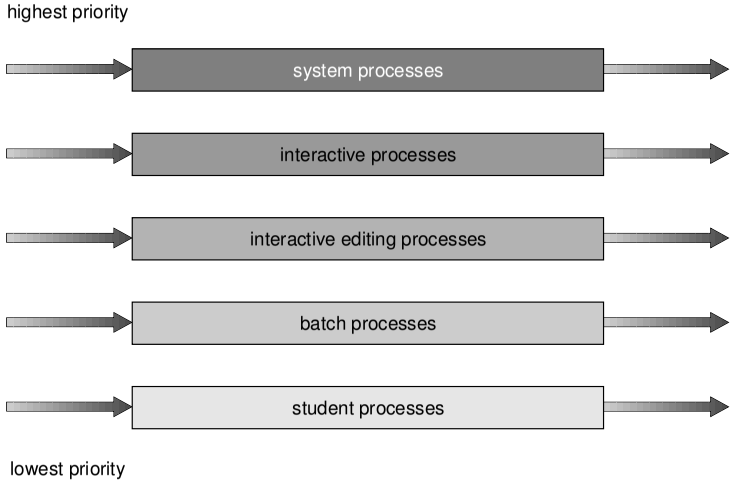
Prestazioni dello scheduling Round-Robin (Cont.)



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)



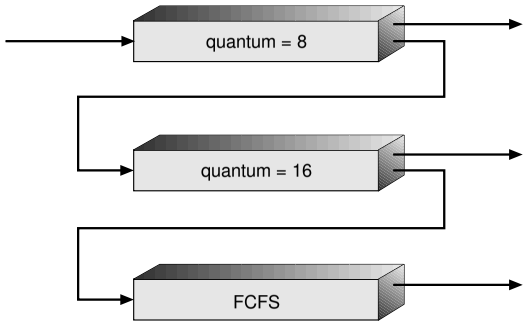
Scheduling con code multiple (Cont.)

- 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*) $\Rightarrow$ 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, ...

Scheduling Real-Time (cont.)

- Eventi *aperiodici*: imprevedibili (es: segnalazione da un sensore)
- Eventi *periodici*: avvengono ad intervalli di tempo regolari o prevedibili (es.: (de)codifica audio/video).

  Dati m eventi periodici, questi sono *schedulabili* se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

  dove
  - P_i = periodo dell'evento i
  - C_i = tempo di CPU necessario per gestire l'evento i

Scheduling RMS (Rate Monotonic Scheduling)

- a priorità statiche, proporzionali alla frequenza.
- Lo schedulatore esegue sempre il processo pronto con priorità maggiore, eventualmente prelazionando quello in esecuzione
- Solo per processi periodici, a costo costante.
- Garantisce il funzionamento se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

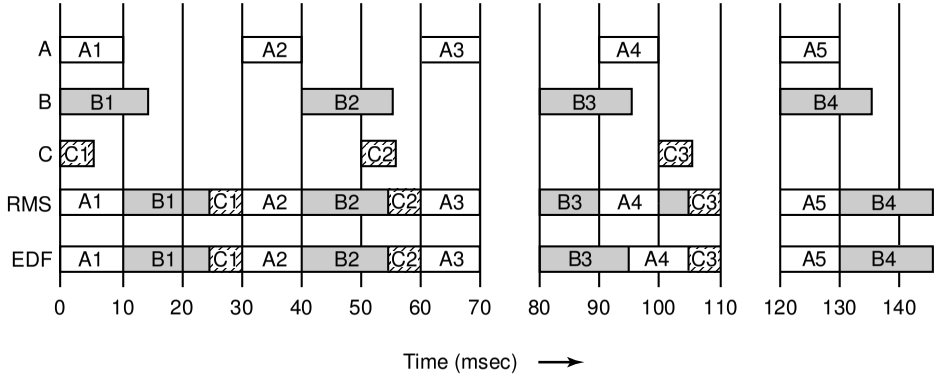
  Con $m = 3$, il limite è 0,780.

  Per $m \to \infty$, questo limite tende a $\log 2 = 0,693$.

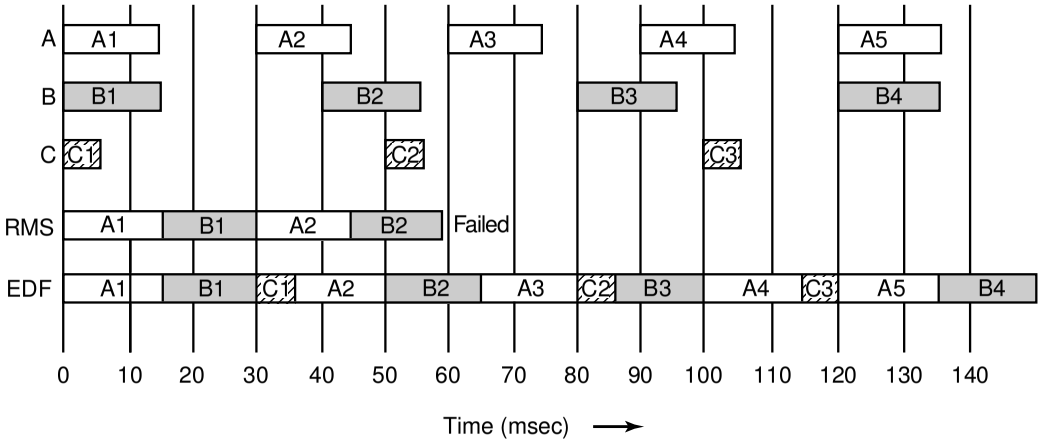
- Semplice da implementare

Scheduling EDF (Earlier Deadline First)

- a priorità dinamiche, in base a chi scade prima.
- Adatto anche per processi non periodici.
- Permette di raggiungere anche il 100% di utilizzo della CPU.
- Più complesso (e costoso) di RMS



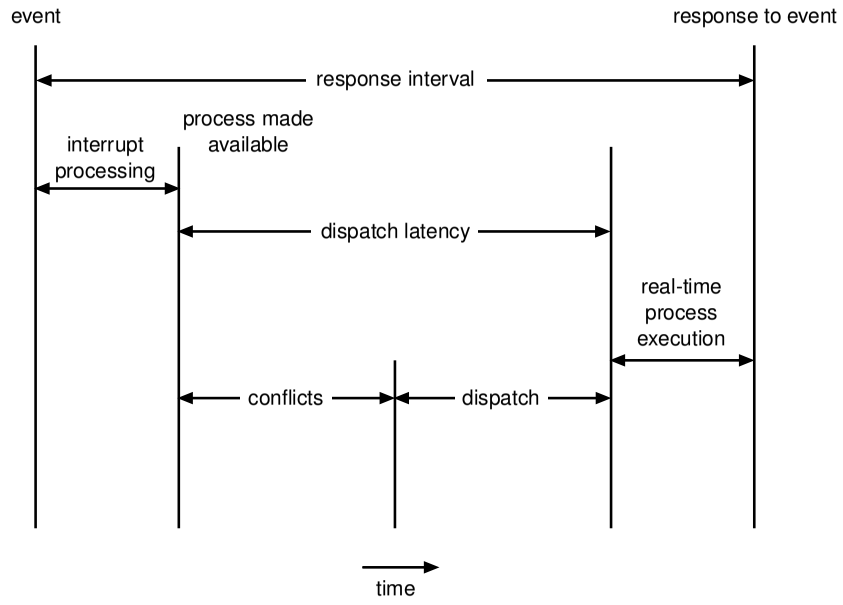
Esempio di fallimento di RMS



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.

Latenza di dispatch (cont.)

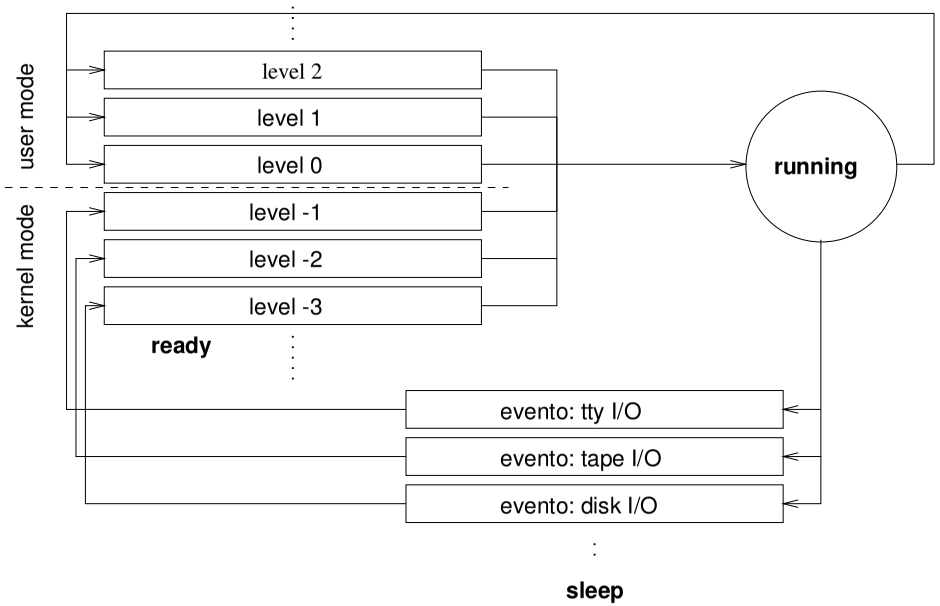


Scheduling di breve termine in Unix tradizionale

(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 *tutti* i processi che erano in *sleep* sull'evento vengono messi 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

Scheduling in Unix tradizionale (Cont.)



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

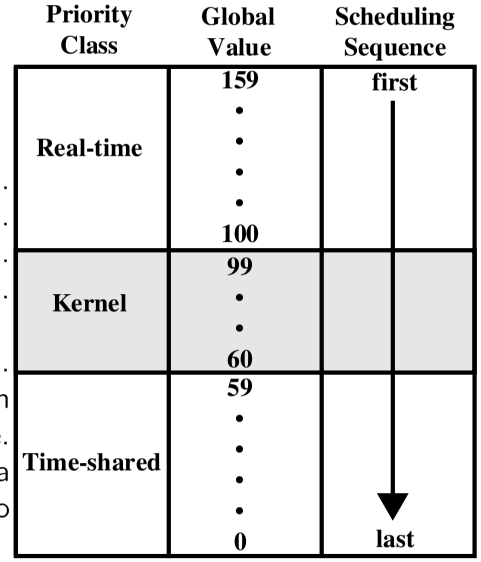
Scheduling in Unix moderno (4.4BSD, SVR4 e successivi)

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.



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, ...)

Classi di Scheduling: Solaris

```
miculan@maxi:miculan$ priocntl -l
CONFIGURED CLASSES
==================
SYS (System Class)
TS (Time Sharing)
    Configured TS User Priority Range: -60 through 60
RT (Real Time)
    Maximum Configured RT Priority: 59
IA (Interactive)
    Configured IA User Priority Range: -60 through 60
miculan@maxi:miculan$
```


Classi di Scheduling in Solaris: Real-Time

```
miculan@maxi:miculan$ dispadmin -c RT -g
# Real Time Dispatcher Configuration
RES=1000

# TIME QUANTUM                    PRIORITY
# (rt_quantum)                      LEVEL
      1000                #         0
       ...
      1000                #         9
       800                #        10
       ...
       800                #        19
       600                #        20
       ...
       600                #        29
       400                #        30
       ...
       400                #        39
       200                #        40
       ...
       200                #        49
       100                #        50
       ...
       100                #        59
miculan@maxi:miculan$
```


Classi di Scheduling in Solaris: Time-Sharing

```
miculan@maxi:miculan$ dispadmin -c TS -g
# Time Sharing Dispatcher Configuration
RES=1000

# ts_quantum ts_tqexp ts_slpret ts_maxwait ts_lwait PRI LEVEL
       200       0       50          0        50     #     0
       200       0       50          0        50     #     1
       200       0       50          0        50     #     2
       ...
       200       0       50          0        50     #     9
       160       0       51          0        51     #    10
       160       1       51          0        51     #    11
       160       2       51          0        51     #    12
       ...
       160       8       51          0        51     #    18
       160       9       51          0        51     #    19
       120      10       52          0        52     #    20
       ...
       120      19       52          0        52     #    29
        80      20       53          0        53     #    30
        80      21       53          0        53     #    31
       ...
        80      29       54          0        54     #    39
        40      30       55          0        55     #    40
        40      31       55          0        55     #    41
       ...
        40      48       58          0        59     #    58
        20      49       59      32000        59     #    59
miculan@maxi:miculan$
```

Classi di Scheduling in Solaris: Time-Sharing

RES: *resolution* della colonna ts_quantum (1000=millesimi di secondo)

ts_quantum: quanto di tempo

ts_tqexp: *time quantum expiration level*: livello a cui portare un processo che ha terminato il suo quanto

ts_slpret: *sleep priority return level*: livello a cui portare il processo dopo un wakeup

ts_maxwait, ts_lwait: se un processo non termina un quanto di tempo da più di ts_maxwait secondi, viene portato a ts_lwait (ogni secondo)

L'utente `root` può modificare run-time le tabelle di scheduling con il comando `dispadmin`. **ESTREMA CAUTELA!!**

Classi di Scheduling in Linux

Diverso da SVR4; aumentato il supporto real-time

- Ogni processo ha una priorità statica ed una dinamica, tra 0 e 99
- Lo scheduler seleziona sempre il processo con priorità statica più alta; a parità di statica, quello a priorità dinamica maggiore
- Prelezione del processo running se un processo a priorità statica maggiore passa in ready. Il processo prelazionato porta a termine l'esecuzione dello stesso quanto, quando non ci sono processi a priorità statica maggiore

Classi di Scheduling in Linux (Cont.)

Tre politiche di scheduling (`man sched_setscheduler`)

SCHED_FIFO: per processi real-time

- First-Come, First-Served
- priorità statica = priorità dinamica, compresa tra 1 e 99

SCHED_RR: per processi real-time, conforme POSIX.1b

- round-robin, quanto configurabile (sched_rr_get_interval)
- priorità statica = priorità dinamica, compresa tra 1 e 99

SCHED_OTHER: per i processi "normali"

- time sharing, round-robin
- priorità statica = 0, priorità dinamica = n. di quanti "saltati" − nice

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)

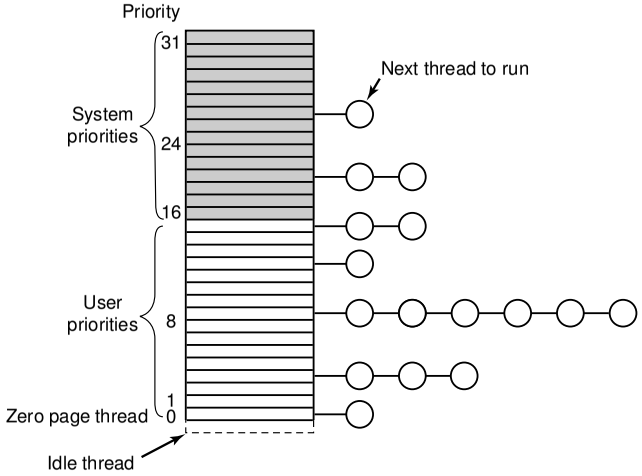
Scheduling di Windows 2000

- 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*.



- 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

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)

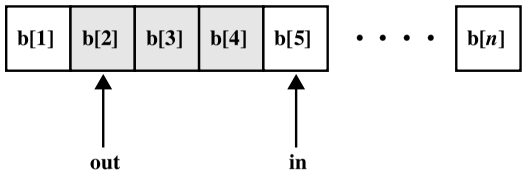
Questioni da considerare:

- 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?

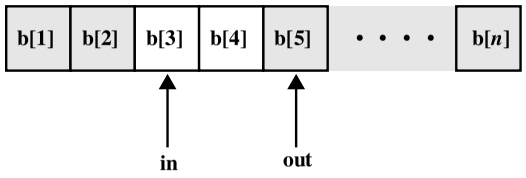
Mantenere la consistenza dei dati richiede dei meccanismi per assicurare l'esecuzione ordinata dei processi cooperanti.

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

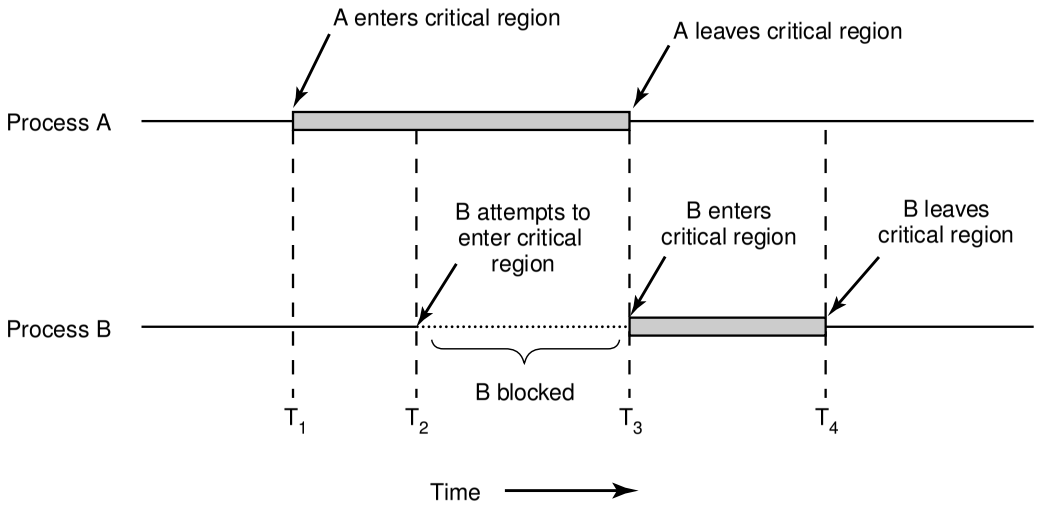
Race condition: più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi.

- 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)

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);    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, i.e., 1,2,3,3,3,3,4,

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.
- 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

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

- S_2 viene eseguito solo dopo S_1.

Implementazione dei semafori

- 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(S): $S.value := S.value - 1$;
if $S.value < 0$
then begin
aggiungi questo processo a $S.L$;
sleep();
end;

up(S): $S.value := S.value + 1$;
if $S.value \leq 0$
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 *wait* e *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
```


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 `synchronize` 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.

Passaggio 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.

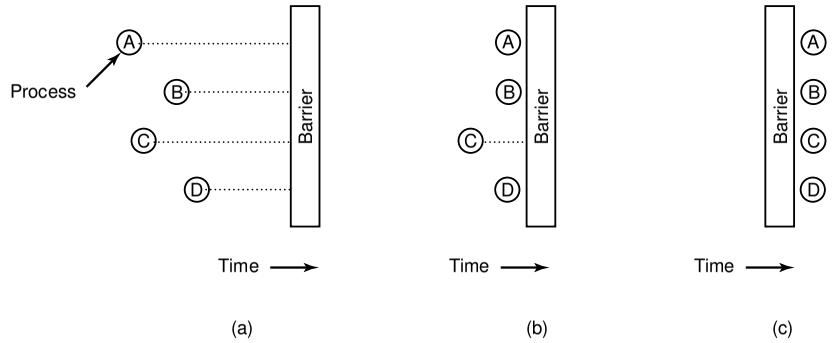
Produttore-consumatore con scambio di messaggi

- 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.
- Comunicazione *sincrona*
  - I messaggi vengono spediti direttamente al processo destinazione
  - L'oggetto delle `send` e `receive` sono i processi
  - Le `send` e `receive` si bloccano fino a che la controparte non esegue la chiamata duale (*rendez-vous*).

```
#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 */
     }
}
```

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

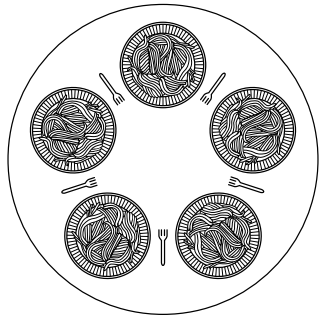
Esempi paradigmatici di programmazione concorrente. Presi come testbed per ogni primitiva di programmazione e comunicazione.

(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 bacchette più vicine, una alla volta.
- Quando ha due bacchette, un filosofo mangia senza fermarsi.
- Terminato il pasto, lascia le bacchette 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 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];                 /* 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( );               /* yum-yum, 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]);
     }
}
```

```
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: 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.

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
  - `mutex`: per mutua esclusione
- Ogni barbiere (uno) 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 */
    }
}
```

Primitive di comunicazione e sincronizzazione in UNIX

- Tradizionali:
  - pipe: canali monodirezionali per produttori/consumatori
  - segnali: interrupt software asincroni
- *InterProcess Communication di SysV:*
  - semafori
  - memoria condivisa
  - code di messaggi
- Per la rete: *socket*

Semafori in UNIX

Caratteristiche dei semafori di Unix:

- i semafori sono mantenuti dal kernel, e identificati con un numero
- possiedono modalità di accesso simili a quelli dei file
- si possono creare *array* di semafori con singole operazioni
- si può operare su più semafori (dello stesso array) simultaneamente

Creazione dei semafori: semget(2)

```
int semget ( key_t key, int nsems, int semflg )
```

- `key`: intero identificante l'array di semafori
- `nsems`: numero di semafori da allocare
- `semflg`: flag di creazione, con modalità di accesso. Es: `IPC_CREAT | 0644` per la creazione di un nuovo insieme, con modalità di accesso `rw-r--r--` `0` per usare un insieme già definito.

Il risultato è una "handle" al set di semafori, oppure `NULL` su fallimento.

Operazione sui semafori: semop(2)

```
int semop(int semid, struct sembuf *sops,
      unsigned nsops)
```

- `semid`: puntatore all'array di semafori
- `sops`: puntatore ad array di operazioni; ogni operazione è una `struct sembuf` come segue:

```
struct sembuf {
  short int sem_num; /* semaphore number */
  short int sem_op;  /* semaphore operation */
  short int sem_flg; /* operation flag */
};
```

- `nsops`: numero di operazioni

Il risultato è `NULL` se la chiamata ha successo, -1 se è fallita.

Operazione sui semafori: semop(2)

Ogni singola operazione:

- `sem_num` indica su quale semaforo operare
- `sem_flg` può essere 0 (bloccante), `IPC_NOWAIT` (non bloccante), ...
- `sem_op` è un intero da sommare algebricamente al semaforo indicato da `sem_num`.
  - se il valore del semaforo andrebbe negativo, e `sem_flg`=0, il processo viene sospeso finché il valore non è sufficiente
  - valori positivi incrementano il semaforo (non sono bloccanti)

Tutte le operazioni vengono svolte atomicamente.

Controllo dei semafori: semctl(2)

```
int semctl(int semid, int semnum, int cmd, union semun arg)
```

Alcune possibili operazioni sono:

- `cmd=GETALL`: leggi i valori dei semafori
- `cmd=SETALL`: imposta i valori dei semafori
- `cmd=IPC_RMID`: cancella il set di semafori
- `cmd=GETPID`: restituisce il PID del processo che ha fatto l'ultima `semop`

Tutte le operazioni vengono svolte atomicamente.

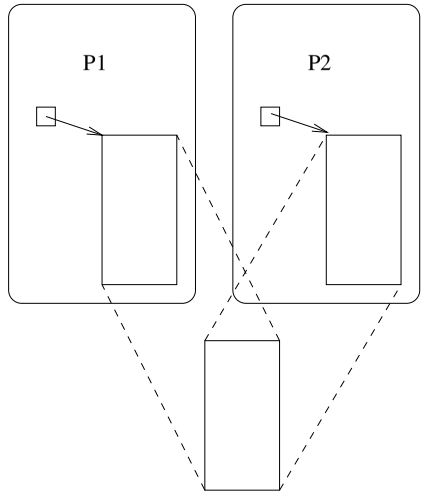
Queste operazioni di controllo non sono bloccanti.

Memoria condivisa

Due o più processi possono condividere parte dello spazio di indirizzamento.

Caratteristiche della memoria condivisa

- non è sincronizzante
- è allocata e deallocata dal kernel, ma accessibile direttamente da user space, senza necessità di system call.
- i segmenti condivisi possiedono modalità di accesso simili a quelli dei file
- è la più veloce forma di comunicazione tra processi



Creazione della memoria condivisa: shmget(2)

```
int shmget ( key_t key, int size, int shmflg )
```

- `key`: intero identificante il segmento di memoria
- `size`: dimensione (in byte)
- `shmflg`: flag di creazione, con modalità di accesso. Es: `IPC_CREAT | 0644` per la creazione di un nuovo segmento, con modalità di accesso `rw-r--r--` `0` per usare un segmento già definito.

Il risultato è una "handle" al segmento, oppure `NULL` su fallimento.

Operazioni sulla memoria condivisa

```
ptr = shmat(int shmid, const void *shmaddr, int shmflg)
```

- `shmid`: handle del segmento
- `shmaddr`: indirizzo dove appicicare il segmento condiviso. 0=lascia scegliere al sistema.
- `shmflg`: modo di attacco. Es: `SHM_RDONLY` = read-only. 0 = read-write.

Il risultato è il puntatore alla zona di memoria attaccata se la chiamata ha successo, 0 se è fallita.

```
int shmdt(const void *shmaddr)
```

Distacca la zona di memoria condivisa puntata da `shmaddr`

Esempio di uso della memoria condivisa

Problema: un processo "demone" che stampa una riga di caratteri ogni 4 secondi; un altro che controlla il carattere e la lunghezza della riga.

Soluzione: con memoria condivisa. File di definizione comuni:

```
/* line.h - definizioni comuni */
struct info {
  char c;
  int  length;
};

#define KEY      ((key_t)(1243))
#define SEGSIZE  sizeof(struct info)
```



```
/* pline.c - print a line of char - con shared memory */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

void main()
{
  int              i, id;
  struct info     *ctrl;

  /* Creazione dell'area condivisa */
  id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
  if (id < 0) {
    perror("pline: shmget failed");
    exit(1);
  }
  /* Attaccamento all'area */
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
    perror("pline: shmat failed");
    exit(2);
  }
  /* Impostazione valori di default */
  ctrl->c      = 'a';
  ctrl->length = 10;

  /* Loop principale */
  while (ctrl->length > 0) {
    for (i=0; i<ctrl->length; i++) {
      putchar(ctrl->c);
      sleep(1);
    }
    putchar('\n');
    sleep(4);
  }
  exit(0);
}
```



```
/* cline.c - set the line to print - con shared memory */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

void main(unsigned argc, char **argv)
{
  int              id;
  struct info     *ctrl;

  if (argc != 3) {
    fprintf(stderr, "usage: cline <char> <length>\n");
    exit(3);
  }

  /* Dichiarazione dell'area condivisa */
  id = shmget(KEY, SEGSIZE, 0);
  if (id < 0) {
    perror("cline: shmget failed");
    exit(1);
  }
  /* Attaccamento all'area */
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
    perror("cline: shmat failed");
    exit(2);
  }

  /* Copia dei valori nell'area condivis
  ctrl->c      = argv[1][0];
  ctrl->length = atoi(argv[2]);

  exit(0);
}
```


Esecuzione di pline/cline

```
miculan@coltrane:Shared_Memory$ pline
aaaaaaaaaa
aaaaaaaaaa
bbbbb
bbbbb
aaaaaaa
eeeeeeeeeeeeeeee
miculan@coltrane:Shared_Memory$
```

```
miculan@coltrane:Shared_Memory$ ./cline b 5
miculan@coltrane:Shared_Memory$ ./cline a 7
miculan@coltrane:Shared_Memory$ ./cline e 30
miculan@coltrane:Shared_Memory$ ./cline a 0
miculan@coltrane:Shared_Memory$
```

pline con semafori

Proteggiamo la sezione condivisa con un semaforo:

```
/* line.h - definizioni comuni */
struct info {
  char c;
  int  length;
};

#define KEY      ((key_t)(1243))
#define SEGSIZE  sizeof(struct info)
#define SKEY     ((key_t)(101))
```


pline con semaforo

```
/* sempline.c - print a line of char
 * con shared memory e semafori */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "line.h"

void main()
{
  int              i, j, id, sem;
  struct info     *ctrl;
  struct sembuf    lock, unlock;

  /* Creazione dell'area condivisa */
```



```
  id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
  if (id < 0) {
    perror("pline: shmget failed");
    exit(1);
  }

  /* Attaccamento all'area */
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
    perror("sempline: shmat failed");
    exit(2);
  }

  /* Creazione del semaforo, se non c'e' gia' */
  sem = semget(SKEY, 1, IPC_CREAT | 0666);
  if (sem < 0) {
    perror("sempline: semget failed");
    exit(1);
  }
```

```
  /* Preparazione delle due operazioni sul semaforo */
  lock.sem_num = unlock.sem_num = 0;
  lock.sem_op  = -1;
  lock.sem_flg = unlock.sem_flg = 0;
  unlock.sem_op  = 1;

  /* Impostazione valori di default: */
  ctrl->c      = 'a';
  ctrl->length = 10;

  /* Loop principale (il semaforo e' ancora a 0) */
  while (ctrl->length > 0) {
    for (i=0; i<ctrl->length; i++) {
      putchar(ctrl->c);
      /* ciclo di rallentamento */
      for (j=0; j<1000000; j++);
    }
    putchar('\n');
```

```
    /* liberiamo l'area condivisa durante lo sleep... */
    semop(sem, &unlock, 1);
    sleep(4);
    /* ...ma ora ce la riprendiamo */
    semop(sem, &lock, 1);
  }

  /* Cancellazione del semaforo */
  semctl(sem, 0, IPC_RMID);

  exit(0);
}
```

cline con semaforo

```
/* semcline.c - set the line to print
 * con shared memory e semafori */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "line.h"

void main(unsigned argc, char **argv)
{
  int              id, sem;
  struct info     *ctrl;
  struct sembuf    sop;

  if (argc != 3) {
```



```
    fprintf(stderr, "usage: semcline <char> <length>\n");
    exit(3);
  }

  /* Dichiarazione dell'area condivisa */
  id = shmget(KEY, SEGSIZE, 0);
  if (id < 0) {
    perror("semcline: shmget failed");
    exit(1);
  }

  /* Attaccamento all'area */
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
    perror("semcline: shmat failed");
    exit(2);
  }

  /* Dichiarazione del semaforo */
```

```
  sem = semget(SKEY, 1, 0);
  if (sem < 0) {
    perror("semcline: semget failed");
    exit(1);
  }

  /* Lock dell'area condivisa */
  sop.sem_num = 0;
  sop.sem_op  = -1;
  sop.sem_flg = 0;
  semop(sem, &sop, 1);

  /* Copia dei valori nell'area condivisa */
  ctrl->c      = argv[1][0];
  ctrl->length = atoi(argv[2]);

  /* Unlock dell'area condivisa */
  sop.sem_op  = 1;
  semop(sem, &sop, 1);

  exit(0);
}
```

Monitoraggio della memoria condivisa: ipcs e ipcrm

```
miculan@coltrane:Shared_Memory$ ipcs -m -s
------ Shared Memory Segments --------
key        shmid      owner     perms     bytes     nattch    status
0x00000000 923653     miculan   777       65536     2         dest
0x000004db 1017862    miculan   666       8         0

------ Semaphore Arrays --------
key        semid      owner     perms     nsems     status
0x00000065 0          miculan   666       1

miculan@coltrane:Shared_Memory$ ipcrm shm 1017862
resource deleted
miculan@coltrane:Shared_Memory$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner     perms     bytes     nattch    status
0x00000000 923653     miculan   777       65536     2         dest

miculan@coltrane:Shared_Memory$
```


Memoria condivisa con mmap(2)

Permette di accedere a file su disco come a zone di memoria.

```
ptr = mmap(id *start, size_t length, int prot, int flags, int fd, off_t offset)
```

- `start`: indirizzo suggerito per l'attacco. 0=lascia scegliere al sistema
- `length`: dimensione (in byte)
- `prot`: flag di protezione. Es: `PROT_READ | PROT_WRITE`
- `flags`: flag di mappatura. Esempi:
  `MAP_SHARED` : condiviso con altri processi; `MAP_PRIVATE` : mappatura privata
- `fd`: file descriptor del file da mappare in memoria
- `offset`: offset nel file da cui iniziare la mappatura

`ptr` punta all'inizio del segmento, oppure è `NULL` su fallimento.

Sincronizzazione di thread in Solaris 2

- Quattro primitive aggiuntive di sincronizzazione per supportare multitasking, multithreading (anche real-time) e macchine multiprocessore
- Vengono impiegate per i thread a livello kernel
- Accessibili anche per thread a livello utente.
- Quando un thread si blocca per una operazione su queste strutture, altri thread dello stesso processo possono procedere
- Sono implementate con operazioni di test-and-set

Sincronizzazione di thread in Solaris 2 (Cont.)

- **Semafori** di thread: singoli, incremento/decremento
- **Lock di mutua esclusione adattativi** (*adaptative mutex*), adatti per proteggere brevi sezioni di codice: sono spinlock che possono trasformarsi in veri block se il lock è tenuto da un processo in wait.
- **Variabili condition**, per segmenti di codice lunghi (associate a mutex)

```
mutex_enter(&m);
...
while(<condizione>) {
  cv_wait(&cv, &m);
}
...
mutex_exit(&m);
```

- **Lock lettura/scrittura**: più thread possono leggere contemporaneamente, ma solo un thread può avere accesso in scrittura.

Pipe e filtri

Le *pipe* sono la più comune forma di IPC

- canali unidirezionali FIFO, a buffer limitato, senza struttura di messaggio, tra due processi, accessibili attraverso dei file descriptor
- Le pipe sono al cuore della filosofia UNIX: le soluzioni a problemi complessi si ottengono componendo strumenti semplici ma generali ("distribuzione algoritmica a granularità grossa")
- Da shell, è possibile comporre singoli comandi in catene di pipe

```
$ ls | pr | lpr
```

- Classica soluzione per situazioni produttore/consumatore
- *Filtro:* un comando come *pr, awk, sed, sort*, che ricevono dati dallo standard input, lo processano e dànno il risultato sullo standard output.

Pipe: creazione, utilizzo

- Una pipe viene creata con *pipe(2)*:

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- Se ha successo (risultato = 0)
  - `filedes[0]` è la coda della pipe (l'output)
  - `filedes[1]` è la testa della pipe (l'input)
- I due file descriptor possono essere usati per letture/scritture con le syscall *read(2), write(2)*

Pipe: creazione, utilizzo (cont.)

La creazione di una pipe viene sempre fatta da un processo padre, prima di uno o più fork. Ad esempio come segue:

1. Processo A crea una pipe, ottenendo i due file descriptor
2. A si forka due volte, creando i processi B, C, che ereditano i file descriptor
3. A chiude entrambi i file descriptor, B chiude quello di output, C quello di input della pipe
4. Ora B è il produttore e C è il consumatore; possono comunicare attraverso la pipe con delle `read/write`.

Pipe: creazione, utilizzo (cont.)

- Non viene creato nessun file su disco: viene solo allocato un buffer di memoria (tipicamente, 1 pagina (tipicamente, 4K))
- sincronizzazione "lasca" tra produttore/consumatore
  - una `read` da una pipe vuota blocca il processo finché il produttore non vi scrive
  - una `write` su una pipe piena blocca il processo finché il consumatore non legge
- La `read` restuisce EOF se non c'è nessun processo che ha aperto l'input della pipe in scrittura.

Pipe: esempio: `who | sort`

```
/* File whosort.c */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main ()
{
  int fds[2];

  pipe(fds);    /* Crezione della pipe */

  /* Il 1o figlio attacca il suo stdin alla fine del pipe */
  if (fork() == 0) {
    dup2(fds[0], 0);


    close(fds[1]);   /* questo close e' essenziale */
    execlp("sort", "sort", 0);
  }

  /* Il 2o figlio attacca il suo stdout all'inizio del pipe */
  else if (fork() == 0) {
    dup2(fds[1], 1);
    close(fds[1]);
    execlp("who", "who", 0);
  }

  /* Il parent chiude la pipe, e aspetta i figli */
  else {
    close(fds[0]);
    close(fds[1]);  /* questo close e' essenziale */
    wait(0);
    wait(0);
  }

  exit(0);
}
```


Pipe: limiti

- Sono unidirezionali
- Non mantengono struttura del messaggio
- Un processo non può sapere chi è dall'altra parte del "tubo"
- Devono essere prearrangiate da un processo comune agli utilizzatori
- Non funzionano attraverso una rete (sono locali ad ogni macchina)
- Non sono permanenti

La soluzione a questi problemi saranno le *socket*.

Named pipe

- Sono pipe "residenti" su disco.
- Creazione: con la syscall *mknod(2)*, o con il comando `mknod`:

```
$ mknod tubo p
$ ls -l tubo
prw-r--r--  1 miculan ospiti  0 Jan 10 16:54 tubo
$
```

- Si usano come un normale file: si aprono con la *fopen*, e vi si legge/scrive come da qualunque file.
- In realtà i dati non vengono mai scritti su disco.
- Vantaggi: sono permanenti, hanno meccanismi di protezione, possono essere usate anche da processi non parenti
- Svantaggi: bisogna ricordarsi di "pulirle", alla fine.

Code di messaggi

Le *code di messaggi* sono una forma di IPC di SysV

- "mailbox" strutturate, in cui si possono riporre e ritirare messaggi
- preservano la struttura e il tipo dei messaggi
- accessibili da più processi
- non necessitano di un processo genitore comune per la creazione
- soggetti a controllo di accesso come i file, semafori,...
- sia le code, sia i singoli messaggi sono permanenti rispetto alla vita dei processi
- si possono monitorare e cancellare da shell con i comandi `ipcs`, `ipcrm`

Code di messaggi (cont.)

- consentono una sincronizzazione "lasca" tra processi
  - una lettura generica (senza tipo) da una queue vuota blocca il processo finché qualcuno non vi scrive un messaggio
  - una scrittura su una queue piena blocca il processo finché qualcuno non consuma un messaggio
- non sono permanenti su disco
- non permettono comunicazione tra macchine diverse

Adatte per situazioni produttore/consumatore locali, con messaggi strutturati.

Code di messaggi: creazione

```
id = msgget(key, flag)
```

- `key`: intero identificatore della coda di messaggi
- `flag`: modo di creazione:
  `IPC_CREAT | 0644` crea una nuova coda con modo `rw-r--r--`
  `0` si attacca ad una coda preesistente
- `id`: handle per successivo utilizzo della coda

Fallisce se si chiede di creare una coda che esiste già, o se si cerca di attaccarsi ad una coda per la quale non si hanno i permessi.

Code di messaggi: spedizione/ricezione

```
msgsnd(id, ptr, size, flag)
msgrcv(id, ptr, size, type, flag)
```

- `id`: handle restuita dalla `msgget` precedente
- `ptr`: puntatore ad una struct della forma

```
struct message {
   long mtype;
   char mtext[...];
}
```

- `size`: dimensione del messaggio in `mtext`
- `type`: tipo di messaggio richiesto; 0 = qualsiasi tipo
- `flag`: modo di spedizione/ricezione: `0` = comportamento normale (bloccante); `IPC_NOWAIT` = non bloccante

Segnali

- Strumenti per gestire eventi *asincroni* — sorta di interrupt software. *Asincroni*= non sono collegati o sincronizzati con una istruzione del processo che li riceve.
- Esempio: il segnale di interrupt SIGINT, è usato per fermare un comando prima della terminazione (CTRL-C).

  Un processo che esegue un'istruzione non valida riceve un SIGILL.
- I segnali sono impiegati anche per notificare eventi "normali"
  - iniziare/terminare dei sottoprocessi su richiesta
  - SIGWINCH informa il processo che la finestra in cui i dati sono mostrati è stata ridimensionata
- Ogni processo può cambiare la gestione di default dei segnali (tranne per SIGKILL)

Segnali: esempio

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int n = 0;

void dispatcher(int sig)
{
  printf("--> SEGNALE %d: ", sig);

  switch (sig) {
  case SIGHUP:
    printf("incremento il contatore\n");
    n++;
    break;
  case SIGQUIT:
    printf("decremento il contatore\n");
    n--;
```



```
    break;
  case SIGINT:
    printf("me ne faccio un baffo!\n");
    break;
  }

  return;
}

void main(unsigned argc, char **argv)
{
  sigset(SIGINT, &dispatcher);
  sigset(SIGHUP, &dispatcher);
  sigset(SIGQUIT, &dispatcher);

  while (n < 5) {
    pause();
  }
  exit(0);
}
```

Segnali: esempio (cont.)

```
> a.out >log &
[1] 18293
> kill -INT 18293
> kill -INT 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -QUIT 18293
> kill -QUIT 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -HUP 18293
18293: No such process
[1]    Done                    a.out > log
> more log
--> SEGNALE 2: me ne faccio un baffo!
--> SEGNALE 2: me ne faccio un baffo!
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 3: decremento il contatore
--> SEGNALE 3: decremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
>
```

Cosa fare quando arriva un segnale? Alcuni esempi

1. Ignorare il segnale!
2. Terminare il processo in modo "pulito"
3. Riconfigurare dinamicamente il processo
4. Dumping di tabelle e strutture dati interne
5. Attivare/Disattivare messaggi di debugging
6. Implementare un timeout sulle chiamate bloccanti

Deadlock

- Molte risorse dei sistemi di calcolo possono essere usate in modo esclusivo
- I sistemi operativi devono assicurare l'uso consistente di tali risorse
- Le risorse vengono allocate ai processi in modo esclusivo, per un certo periodo di tempo. Gli altri richiedenti vengono messi in attesa.
- Ma un processo può avere bisogno di molte risorse contemporaneamente.
- Questo può portare ad attese circolari $\Rightarrow$ il deadlock (stallo)
- Situazioni di stallo si possono verificare su risorse sia locali sia distribuite, sia software sia hardware. Sono sempre in agguato!
- È necessario avere dei metodi per preveniere, riconoscere o almeno curare i deadlock

Risorse

- Una *risorsa* è una componente del sistema di calcolo a cui i processi possono accedere in modo esclusivo, per un certo periodo di tempo.
- Risorse *prerilasciabili*: possono essere tolte al processo allocante, senza effetti dannosi. Esempio: memoria centrale.
- Risorse *non prerilasciabili*: non può essere ceduta dal processo allocante, pena il fallimento dell'esecuzione. Esempio: stampante.
- I deadlock si hanno con le risorse non prerilasciabili

Risorse

- Protocollo di uso di una risorsa:
  1. Richiedere la risorsa
  2. Usare la risorsa
  3. Rilasciare la risorsa
- Se al momento della richiesta la risorsa non è disponibile, ci sono diverse alternative (attesa, attesa limitata, fallimento, fallback...)

Allocazione di una risorsa

Con dei semafori (tipica soluzione user-space): associamo un mutex alla risorsa.

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
      down(&resource_1);
      use_resource_1( );
      up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources( );
      up(&resource_2);
      up(&resource_1);
}
```

(b)

Allocazione di più risorse

Più mutex, uno per ogni risorsa. Ma come allocarli?

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources( );
      up(&resource_2);
      up(&resource_1);
}

void process_B(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources( );
      up(&resource_2);
      up(&resource_1);
}
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources( );
      up(&resource_2);
      up(&resource_1);
}

void process_B(void) {
      down(&resource_2);
      down(&resource_1);
      use_both_resources( );
      up(&resource_1);
      up(&resource_2);
}
```

(b)

Allocazione di più risorse (cont.)

- La soluzione (a) è sicura: non può portare a deadlock
- La soluzione (b) non è sicura: può portare a deadlock
- Non è detto neanche che i due programmi siano scritti dallo stesso utente: come coordinarsi?
- Con decine, centinaia di risorse (come quelle che deve gestire il kernel stesso), determinare se una sequenza di allocazioni è sicura non è semplice
- Sono necessari dei metodi per
  - riconoscere la possibilità di deadlock (prevenzione)
  - riconoscere un deadlock
  - risoluzione di un deadlock

Il problema del Deadlock

- Definizione di deadlock:

  Un insieme di processi si trova in deadlock (stallo) se ogni processo dell'insieme è in attesa di un evento che solo un altro processo dell'insieme può provocare.

- Tipicamente, l'evento atteso è proprio il rilascio di risorse non prerilasciabili.
- Il numero dei processi e il genere delle risorse e delle richieste non è influente.

Condizioni necessarie per il deadlock

Quattro condizioni necessarie (ma non sufficienti!) perché si possa verificare un deadlock [Coffman et al, 1971]:

1. **Mutua esclusione:** ogni risorsa è assegnata ad un solo processo, oppure è disponibile
2. **Hold&Wait:** i processi che hanno richiesto ed ottenuto delle risorse, ne possono richiedere altre
3. **Mancanza di prerilascio:** le risorse che un processo detiene possono essere rilasciate dal processo solo volontariamente.
4. **Catena di attesa circolare di processi**: esiste un sottoinsieme di processi $\{P_0, P_1, ..., P_n\}$ tali che P_i è in attesa di una risorsa che è assegnata a P_{i+1} mod n

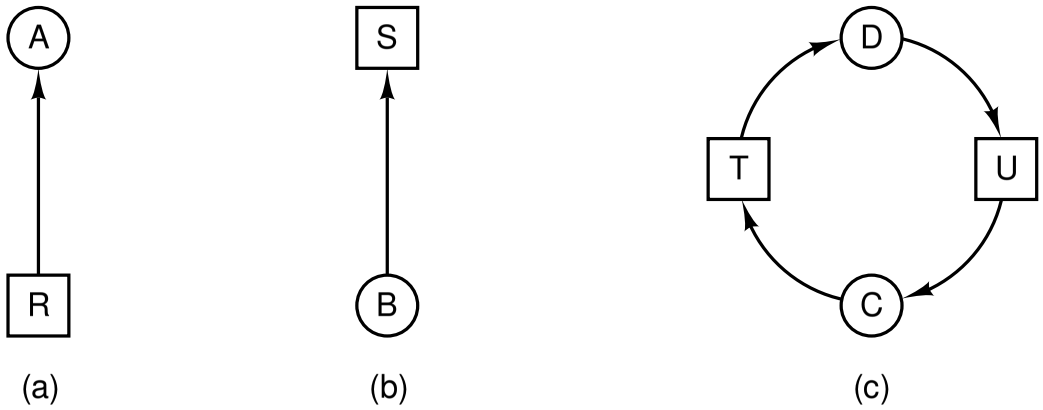
Se anche solo una di queste condizioni manca, il deadlock NON può verificarsi. Ad ogni condizione corrisponde una politica che il sistema può adottare o no.

Grafo di allocazione risorse

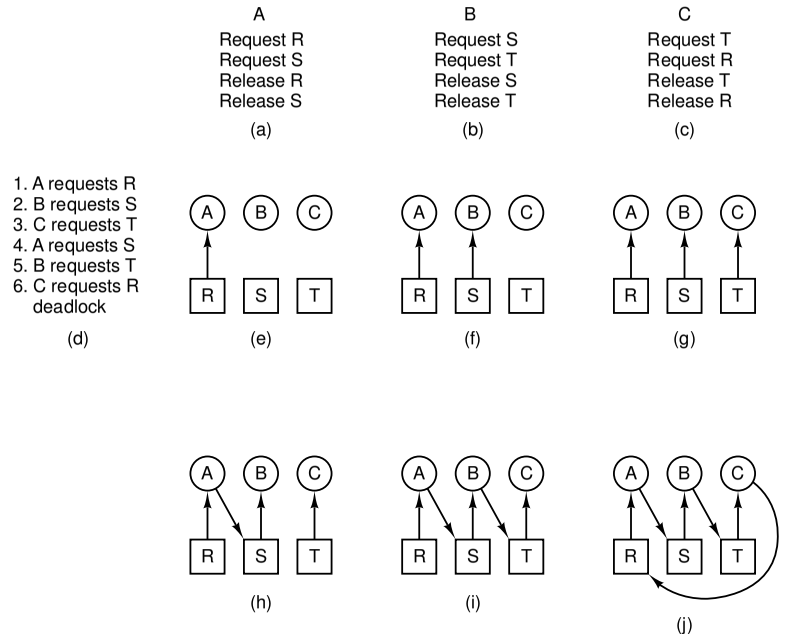
Le quattro condizioni si modellano con un grafo orientato, detto *grafo di allocazione delle risorse*: Un insieme di vertici V e un insieme di archi E

- V è partizionato in due tipi:
  - $P = \{P_1, P_2, \ldots, P_n\}$, l'insieme di tutti i processi del sistema.
  - $R = \{R_1, R_2, \ldots, R_m\}$, l'insieme di tutte le risorse del sistema.
- *archi di richiesta*: archi orientati $P_i \longrightarrow R_j$
- *archi di assegnamento (acquisizione)*: archi orientati $R_j \longrightarrow P_i$

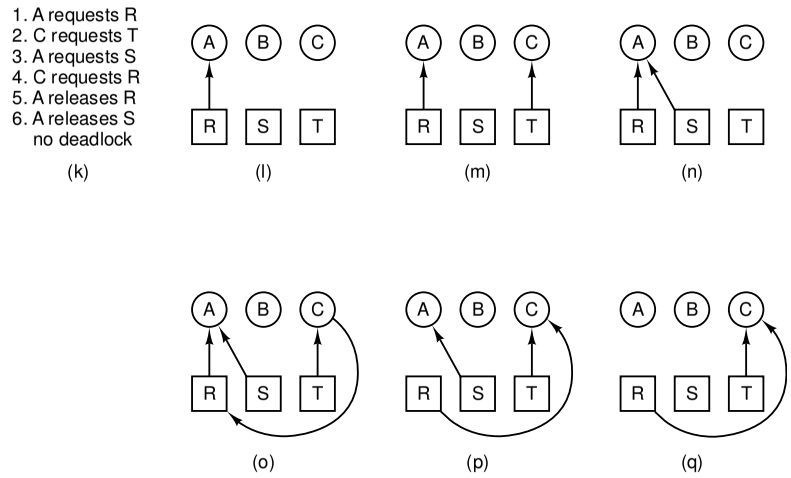
Uno *stallo* è un ciclo nel grafo di allocazione risorse.



Grafo di allocazione risorse (cont.)



Grafo di allocazione risorse (cont.)



Principali fatti

- Se il grafo non contiene cicli $\Rightarrow$ nessun deadlock.
- Se il grafo contiene un ciclo $\Rightarrow$
  - se c'è solo una istanza per tipo di risorsa, allora deadlock
  - se ci sono più istanze per tipo di risorsa, allora c'è la possibilità di deadlock

Uso dei grafi di allocazione risorse

I grafi di allocazione risorse sono uno strumento per verificare se una sequenza di allocazione porta ad un deadlock.

- Il sistema operativo ha a disposizione molte sequenze di scheduling dei processi
- per ogni sequenza, può "simulare" la successione di allocazione sul grafo
- e scegliere una successione che non porta al deadlock.

Il FCFS è una politica "safe", ma insoddisfacente per altri motivi.

Il round-robin in generale non è safe.

Gestione dei Deadlock

- Ignorare il problema, fingendo che non esista (Molto usato).
- Permettere che il sistema entri in un deadlock, riconoscerlo e quindi risolverlo.
- Cercare di evitare dinamicamente le situazioni di stallo, con una accorta gestione delle risorse.
- Assicurare che il sistema non possa mai entrare **mai** in uno stato di deadlock, negando una delle quattro condizioni necessarie.

Primo approccio: Ignorare il problema

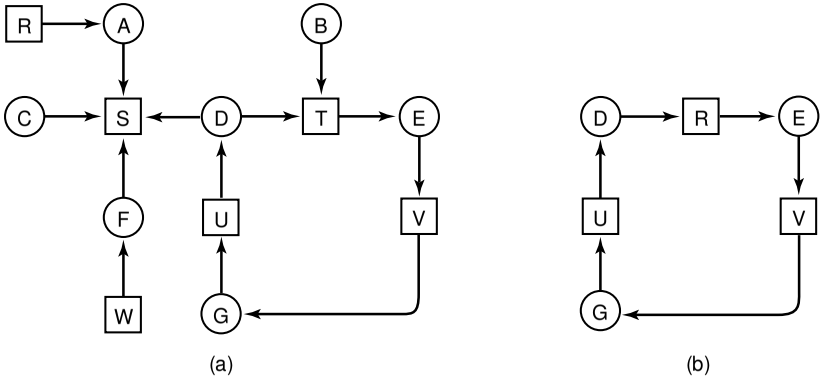
- Assicurare l'assenza di deadlock impone costi (in prestazioni, funzionalità) molto alti.
- Costi necessari per alcuni, ma insopportabili per altri.
- Si considera il rapporto costo/benefici: se la probabilità che accada un deadlock è sufficientemente bassa, non giustifica il costo per evitarlo
- Esempi: il `fork` di Unix, la rete Ethernet, ...
- Approccio adottato dalla maggior parte dei sistemi (Unix e Windows compresi): ignorare il problema.
  - L'utente preferisce qualche stallo occasionale (da risolvere "a mano"), piuttosto che eccessive restrizioni.

Secondo approccio: Identificazione e risoluzione del Deadlock

- Lasciare che il sistema entri in un deadlock
- Riconoscere l'esistenza del deadlock con opportuni algoritmi di identificazione
- Avere una politica di risoluzione (recovery) del deadlock

Algoritmo di identificazione: una risorsa per classe

- Esiste una sola istanza per ogni classe
- Si mantiene un grafo di allocazione delle risorse



- Si usa un algoritmo di ricerca cicli per grafi orientati (v. ASD).
- Costo di ogni chiamata: $O(n^2)$, dove n = numero nodi (= processi+risorse)

Algoritmo di identificazione: più risorse per classe

Strutture dati:

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Invariante: per ogni $j = 1, \ldots, m$: $\sum_{i=1}^{n} C_{ij} + A_j = E_j$

Algoritmo di identificazione

1. $Finish[i] = false$ per ogni $i = 1, \ldots, n$
2. Cerca un i tale che $R[i] \leq A$, ossia $\forall j : R_{ij} \leq A_j$
3. Se esiste tale i:
   - $Finish[i] = true$
   - $A = A + R[i]$ (cioè $A_j = R_{ij}$ per ogni j)
   - Vai a 2.
4. Altrimenti, se esiste i tale che $Finish[i] = false$, allora P_i è in stallo.

L'algoritmo richiede $O(m \times n^2)$ operazioni per decidere se il sistema è in deadlock.

Uso degli algoritmi di identificazione

- Gli algoritmi di identificazione dei deadlock sono costosi
- Quando e quanto invocare l'algoritmo di identificazione? Dipende:
  - Quanto frequentemente può occorrere un deadlock?
  - Quanti processi andremo a "sanare" (almeno uno per ogni ciclo disgiunto)
- Diverse possibilità:
  - Ad ogni richiesta di risorse: riduce il numero di processi da bloccare, ma è molto costoso
  - Ogni k minuti, o quando l'uso della CPU scende sotto una certa soglia: il numero di processi in deadlock può essere alto, e non si può sapere chi ha causato il deadlock

Risoluzione dei deadlock: Prerilascio

- In alcuni casi è possibile togliere una risorsa allocata ad uno dei processi in deadlock, per permettere agli altri di continuare
  - Cercare di scegliere la risorsa più facilmente "interrompibile" (cioè restituibile successivamente al processo, senza dover ricominciare daccapo)
  - Intervento manuale (sospensione/continuazione della stampa)
- Raramente praticabile

Risoluzione dei deadlock: Rollback

- Inserire nei programmi dei *check-point*, in cui *tutto* lo stato dei processi (memoria, dispositivi e risorse comprese) vengono salvati (accumulati) su un file.
- Quando si scopre un deadlock, si conoscono le risorse e i processi coinvolti
- Uno o più processi coinvolti vengono riportati ad uno dei checkpoint salvati, con conseguente rilascio delle risorse allocate da allora in poi (*rollback*)
- Gli altri processi possono continuare
- Il lavoro svolto dopo quel checkpoint è perso e deve essere rifatto.
  - Cercare di scegliere i processi meno distanti dal checkpoint utile.
- Non sempre praticabile. Esempio: ingorgo traffico.

Risoluzione dei deadlock: Terminazione

- Terminare uno (o tutti, per non far torto a nessuno) i processi in stallo
- Equivale a un rollback iniziale.
- Se ne terminiamo uno alla volta, in che ordine?
  - Nel ciclo o fuori dal ciclo?
  - Priorità dei processi
  - Tempo di CPU consumata dal processo, e quanto manca per il completamento
  - Risorse usate dal processo, o ancora richieste per completare
  - Quanti processi si deve terminare per sbloccare lo stallo
  - Prima i processi batch o interattivi?
  - Si può ricominciare daccapo senza problemi?

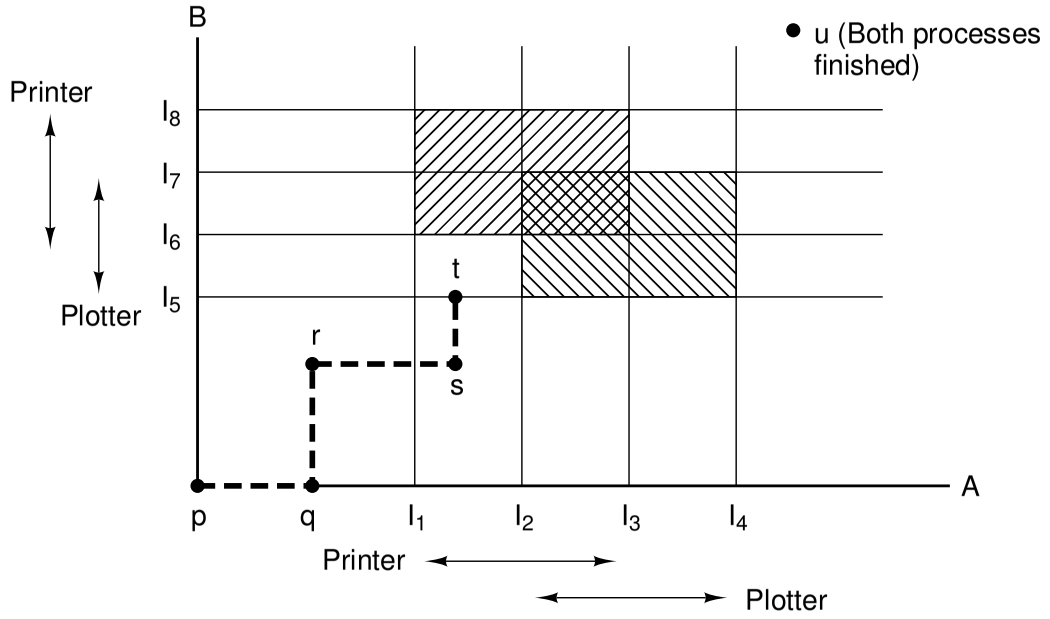
Terzo approccio: Evitare dinamicamente i deadlock

Domanda: è possibile decidere al volo se assegnare una risorsa, evitando di cadere in un deadlock?

Risposta: sì, a patto di conoscere *a priori* alcune informazioni aggiuntive.

- Il modello più semplice ed utile richiede che ogni processo dichiari *fin dall'inizio* il numero *massimo* di risorse di ogni tipo di cui avrà bisogno nel corso della computazione.
- L'algorito di deadlock-avoidance esamina dinamicamente lo stato di allocazione delle risorse per assicurare che non ci siano mai code circolari.
- Lo *stato* di allocazione delle risorse è definito dal numero di risorse allocate, disponibili e dalle richieste massime dei processi.

Traiettorie di risorse



Stati sicuri

- Quando un processi richiede una risorsa, deve decidere se l'allocazione lascia il sistema in uno *stato sicuro*
- Lo stato è *sicuro* se esiste una *sequenza sicura* per tutti i processi.
- La sequenza $\langle P_1, P_2, \ldots, P_n \rangle$ è sicura se per ogni P_i, la risorsa che P_i può ancora richiedere può essere soddisfatta dalle risorse disponibili correntemente più tutte le risorse mantenute dai processi $P_1, \ldots, P_{i-1}$.
  - Se le risorse necessarie a P_i non sono immediatamente disponibili, può aspettare che i precedenti finiscano.
  - Quando i precedenti hanno liberato le risorse, P_i può allocarle, eseguire fino alla terminazione, e rilasciare le risorse allocate.
  - Quando P_i termina, P_{i+1} può ottenere le sue risorse, e così via.

Esempio

Sequenza sicura:

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

Sequenza non sicura (lo stato (b) non è sicuro).

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Osservazioni

- Se il sistema è in uno stato sicuro $\Rightarrow$ non ci sono deadlocks.
- Se il sistema è in uno stato NON sicuro $\Rightarrow$ possibilità di deadlocks.
- Deadlock avoidance: assicurare che il sistema non entri mai in uno stato non sicuro.

Algoritmo del Banchiere (Dijkstra, '65)

Controlla se una richiesta può portare ad uno stato non sicuro; in tal caso, la richiesta non è accettata.

Ad ogni richiesta, l'algoritmo controlla se le risorse rimanenti sono sufficienti per soddisfare la massima richiesta di almeno un processo; in tal caso l'allocazione viene accordata, altrimenti viene negata.

Funziona sia con istanze multiple che con risorse multiple.

- Ogni processo deve dichiarare *a priori* l'uso massimo di ogni risorsa.
- Quando un processo richiede una risorsa, può essere messo in attesa.
- Quando un processo ottiene tutte le risorse che vuole, deve restituirle in un tempo finito.

Esempio dell'algoritmo del banchiere per risorsa singola

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10
(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2
(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1
(c)

Cosa succede se in (b), si allocano 2 istanze a B?

Algoritmo del Banchiere (Cont.)

- Soluzione molto studiata, in molte varianti
- Di scarsa utilità pratica, però.
- È molto raro che i processi possano dichiarare fin dall'inizio tutte le risorse di cui avranno bisogno.
- Il numero dei processi e delle risorse varia dinamicamente
- Di fatto, quasi nessun sistema usa questo algoritmo

Quarto approccio: prevenzione dei Deadlock

Negare una delle quattro condizioni necessarie (Coffman et al, '71)

- Mutua Esclusione
  - Le risorse condivisibili non hanno questo problema
  - Per alcune risorse non condivisibili, si può usare lo *spooling* (che comunque introduce competizione per lo spazio disco)
  - Regola di buona programmazione: allocare le risorse per il minor tempo possibile.

Prevenzione dei Deadlock (cont)

- Hold and Wait: garantire che quando un processo richiede un insieme di risorse, non ne richiede nessun'altra prima di rilasciare quelle che ha.
  - Richiede che i processi richiedano e ricevano tutte le risorse necessarie all'inizio, o che rilascino tutte le risorse prima di chiederne altre
  - Se l'insieme di risorse non può essere allocato in toto, il processo aspetta (metodo transazionale).
  - Basso utilizzo delle risorse
  - Possibilità di starvation
- Negare la mancanza di prerilascio: impraticabile per molte risorse

Prevenzione dei Deadlock (cont)

- Impedire l'attesa circolare.
  - permettere che un processo allochi al più 1 risorsa: molto restrittivo
  - *Ordinamento delle risorse*
    - si impone un ordine totale su tutte le classi di risorse
    - si richiede che ogni processo richieda le risorse nell'ordine fissato
    - un processo che detiene la risorsa j non può mai chiedere una risorsa $i < j$, e quindi non si possono creare dei cicli.
  - Teoricamente fattibile, ma difficile da implementare:
    - l'ordinamento può non andare bene per tutti
    - ogni volta che le risorse cambiano, l'ordinamento deve essere aggiornato

Approccio combinato alla gestione del Deadlock

- I tre approcci di gestione non sono esclusivi, possono essere combinati:
  - rilevamento
  - elusione (avoidance)
  - prevenzione

  si può così scegliere l'approccio ottimale per ogni classe di risorse del sistema.
- Le risorse vengono partizionati in classi ordinate gerarchicamente
- In ogni classe possiamo scegliere la tecnica di gestione più opportuna.

Blocco a due fasi (two-phase locking)

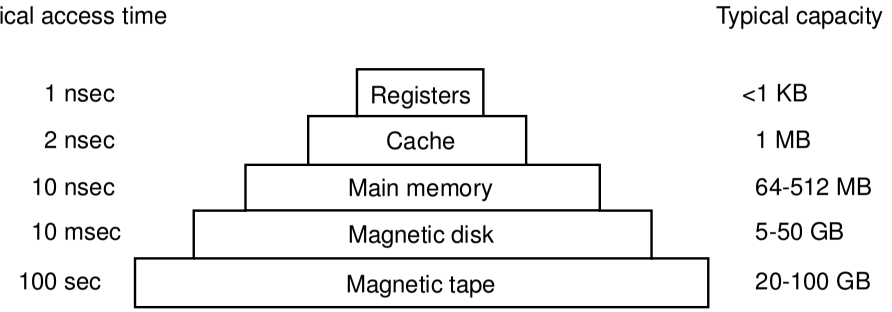
- Protocollo in due passi, molto usato nei database:
  1. Prima il processo prova ad allocare tutte le risorse di cui ha bisogno per la transazione.
  2. Se non ha successo, rilascia tutte le risorse e riprova. Se ha successo, completa la transazione usando le risorse.
- È un modo per evitare l'hold&wait.
- Non applicabile a sistemi real-time (hard o soft), dove non si può far ripartire il processo dall'inizio
- Richiede che il programma sia scritto in modo da poter essere "rieseguito" daccapo (non sempre possibile)

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*

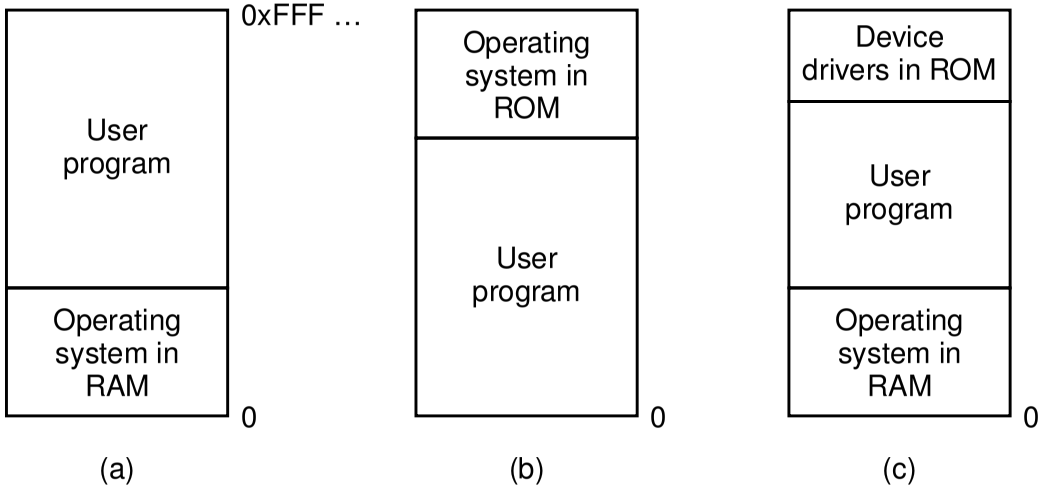


Gestione della memoria: Fondamenti

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



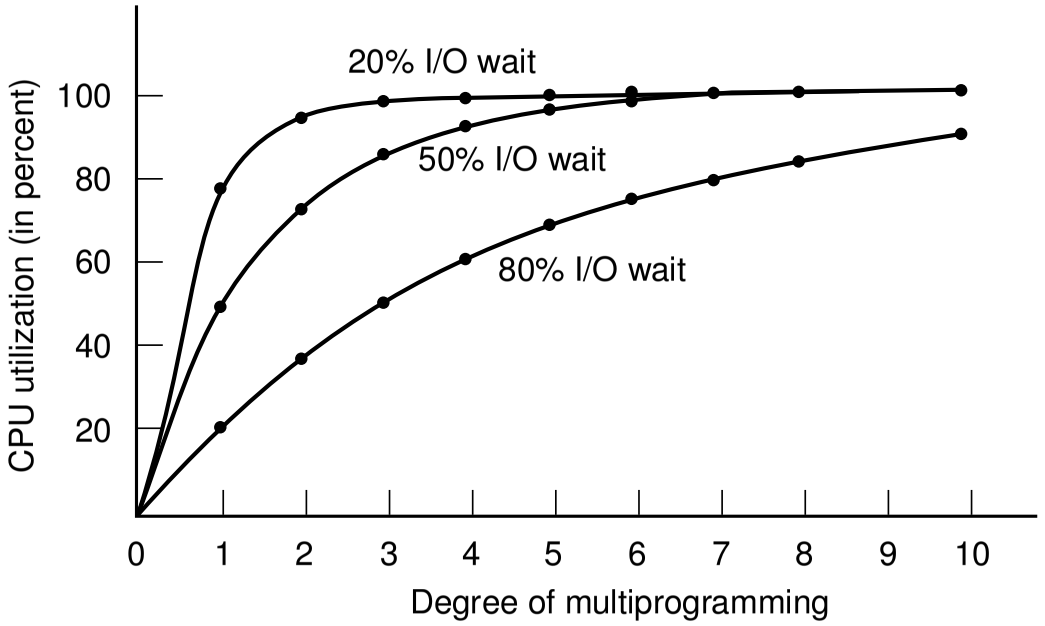
Un solo programma per volta (oltre al sistema operativo). (c) il caso del DOS.

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. Con n processi:

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

- Maggiore il *grado di multiprogrammazione*, maggiore l'utilizzo della CPU
- Il modello è ancora impreciso (i processi non sono indipendenti); un modello più accurato si basa sulla teoria delle code.
- Può essere utile per stimare l'opportunità di upgrade. Esempio:
  - Memoria = $16MB$: grado = 4, utilizzo CPU =60%
  - Memoria = $32MB$: grado = 8, utilizzo CPU =83% (+38%)
  - Memoria = $48MB$: grado = 12, utilizzo CPU =93% (+12%)



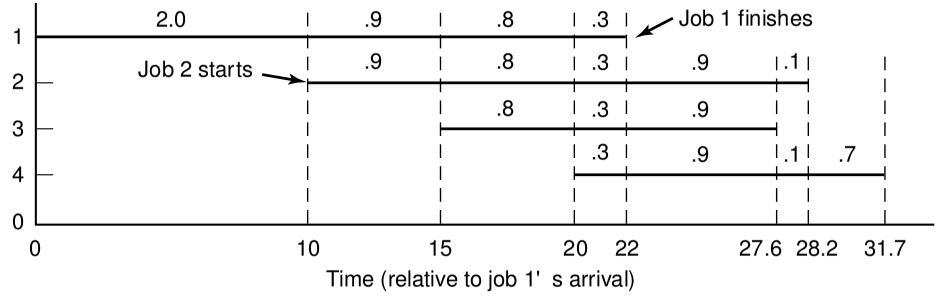
Analisi delle prestazioni: esempio

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

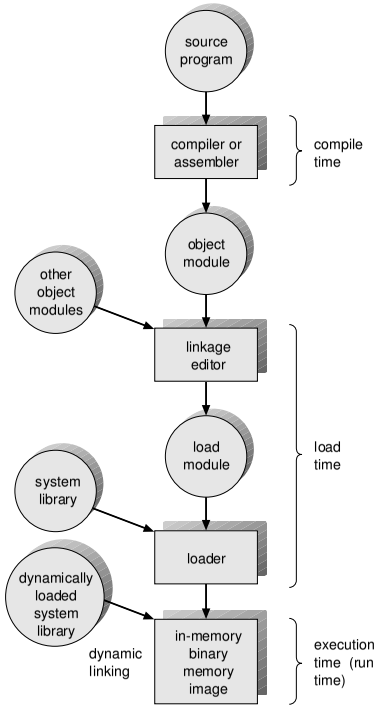
Multiprogrammazione (cont)

- 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

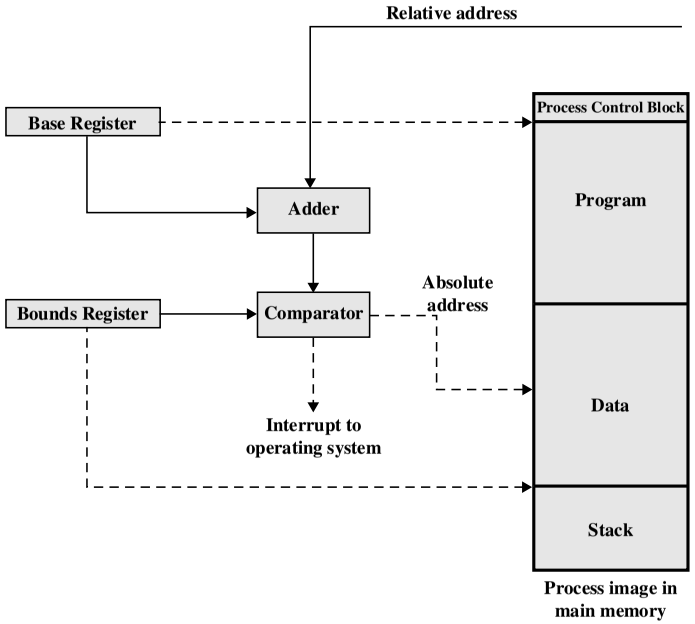
- **Linking dinamico:** le librerie vengono collegate all'esecuzione.
  Esempi: le .so su Unix, le .DLL su Windows.
- Nell'eseguibile si inseriscono piccole porzioni di codice, dette *stub*, che servono per localizzare la routine.
- Alla prima esecuzione, si carica il segmento se non è presente in memoria, 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.

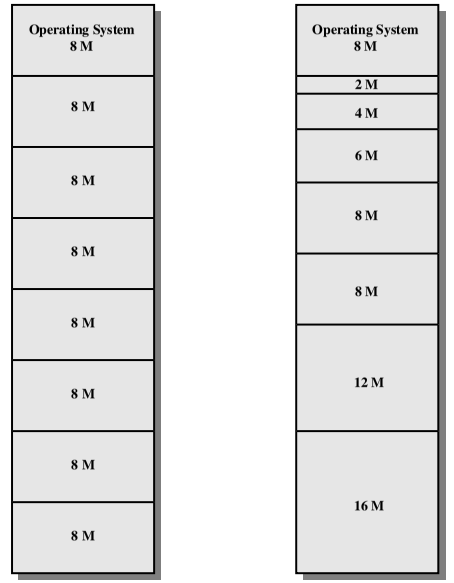


Allocazione contigua

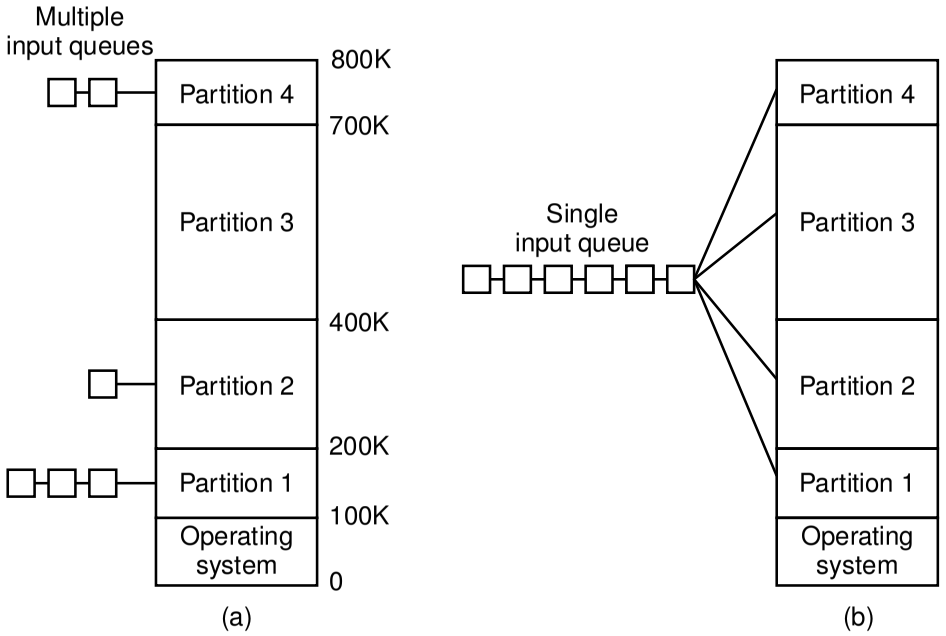
- La memoria è divisa in (almeno) due partizioni:
  - Sistema operativo residente, normalmente nella zona bassa degli indirizzi assieme al vettore delle interruzioni.
  - Spazio per i processi utente — tutta la memoria rimanente.
- Allocazione a partizione singola
  - Un processo è contenuto tutto in 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



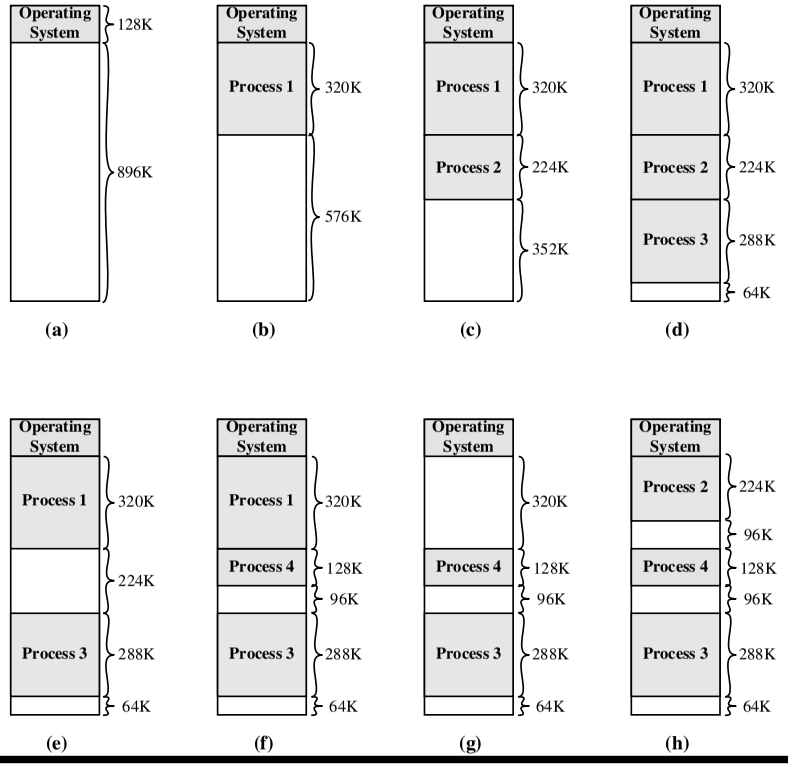
Allocazione contigua: code di input



- 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)

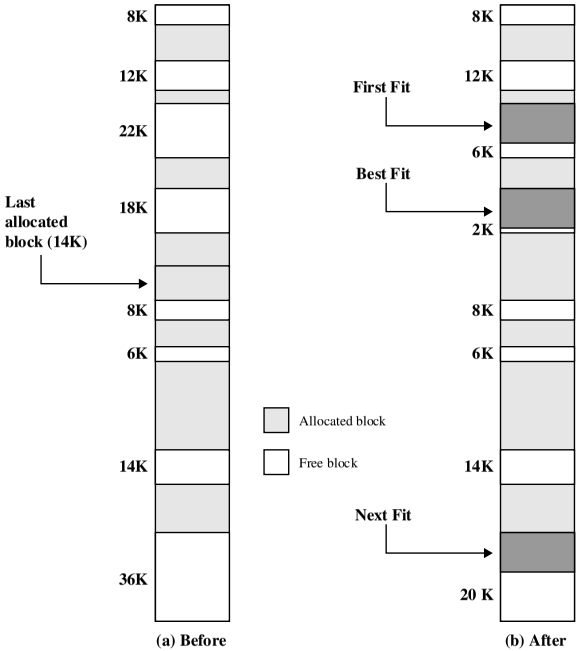
Allocazione contigua: partizionamento dinamico (cont.)

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.

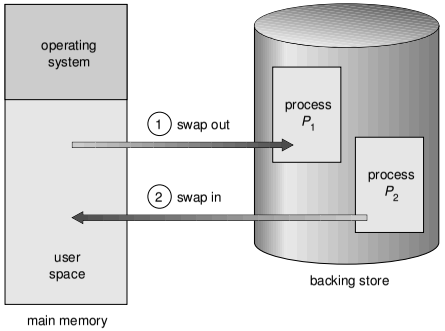
In generale, gli algoritmi migliori sono il first-fit e il next-fit. Best-fit tende a frammentare molto. Worst-fit è più lento.

Allocazione contigua: esempi di allocazione



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.



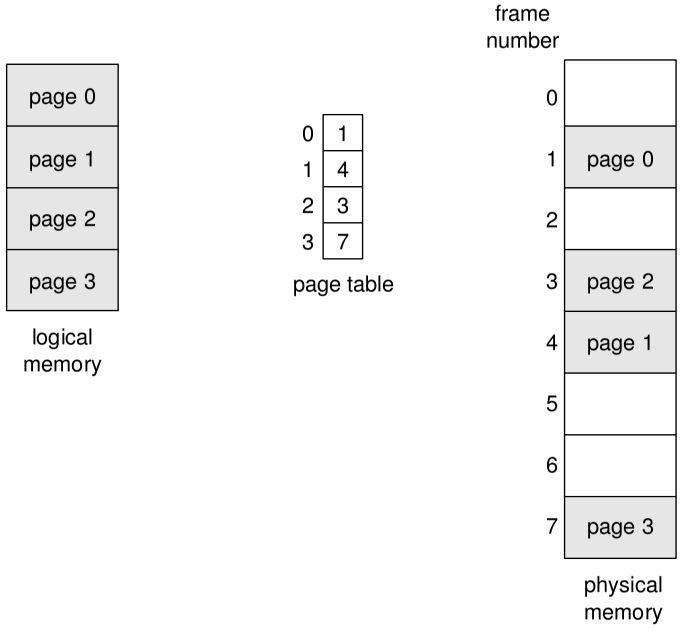
Swapping (Cont.)

- 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.

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

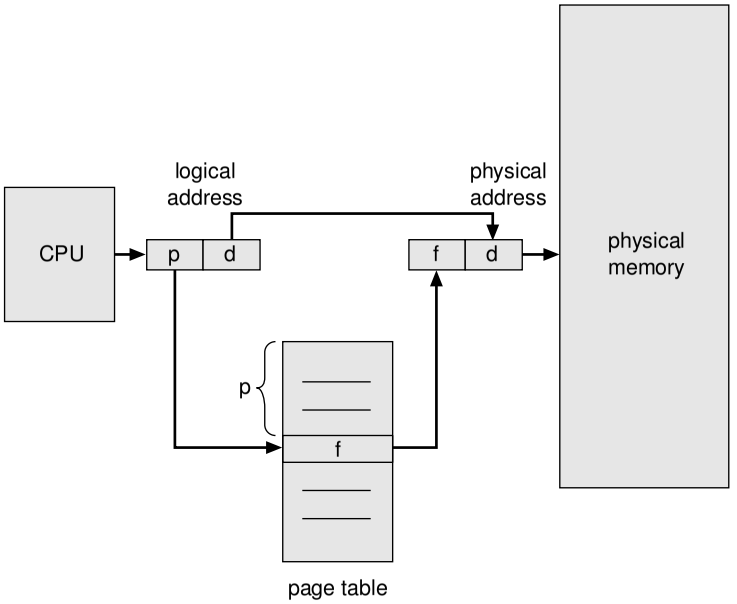
Esempio di paginazione



Schema di traduzione degli indirizzi

L'indirizzo generato dalla CPU viene diviso in

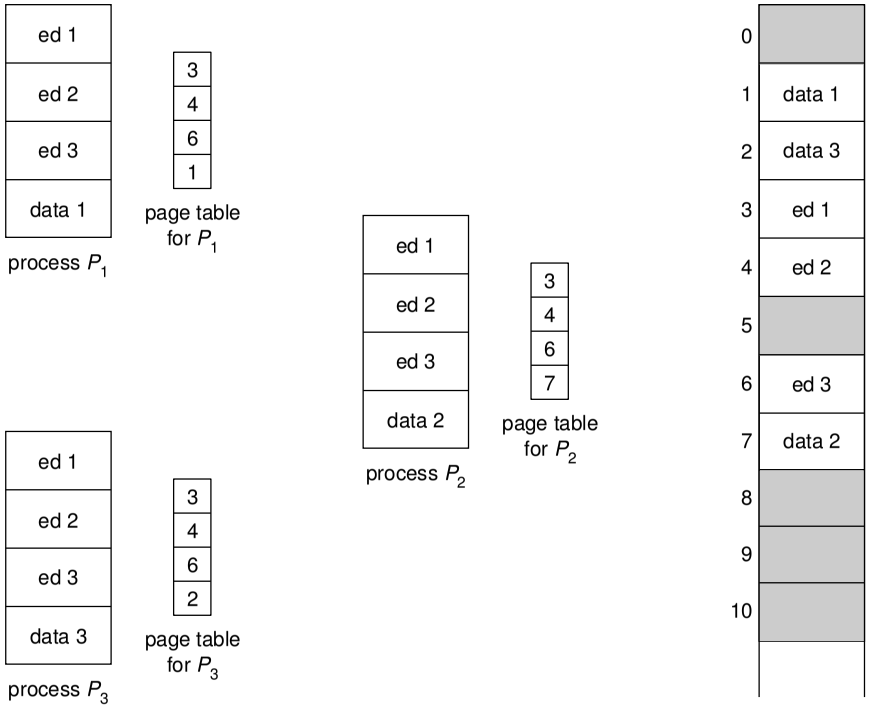
- *Numero di pagina* p*:* usato come indice in una *page table* che contiene il numero del frame contenente la pagina p.
- *Offset di pagina* d*:* combinato con il numero di frame fornisce l'indirizzo fisico da inviare alla memoria.



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

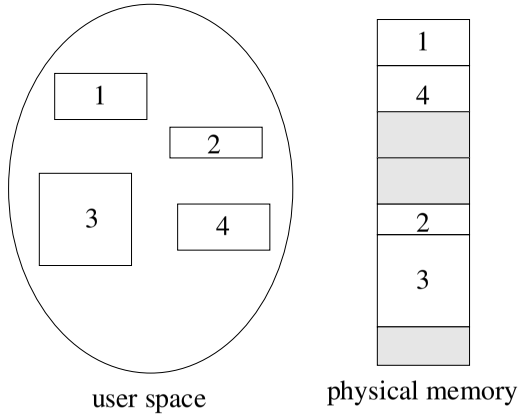


Paginazione: protezione

- 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 ⇒ violazione di indirizzi (Segment violation)

Allocazione non contigua: Segmentazione

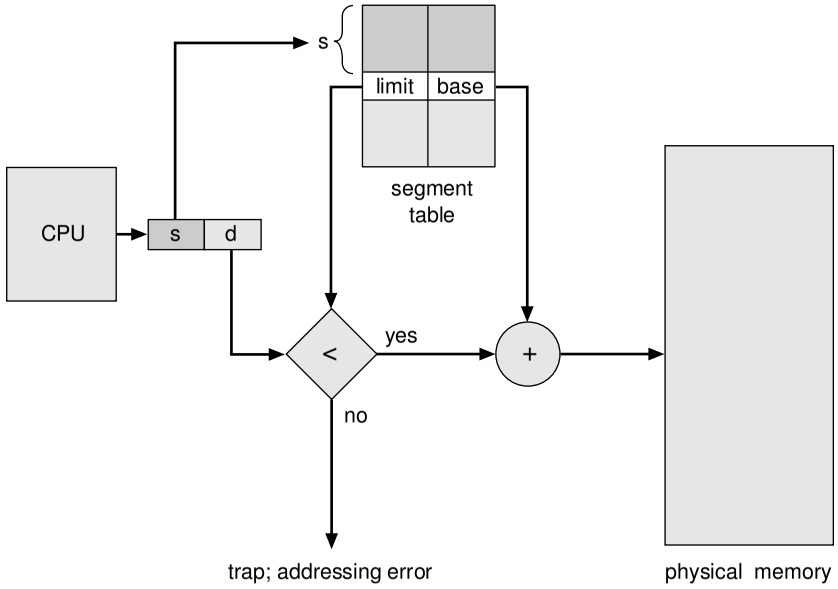
- È uno schema di MM che supporta la visione *utente* della memoria
- Un programma è una collezione di segmenti. Un segmento è una unità logica di memoria; ad esempio: programma principale, procedure, funzioni, variabili locali, variabili globali stack, tabella dei simboli memoria condivisa, ...



Architettura della Segmentazione

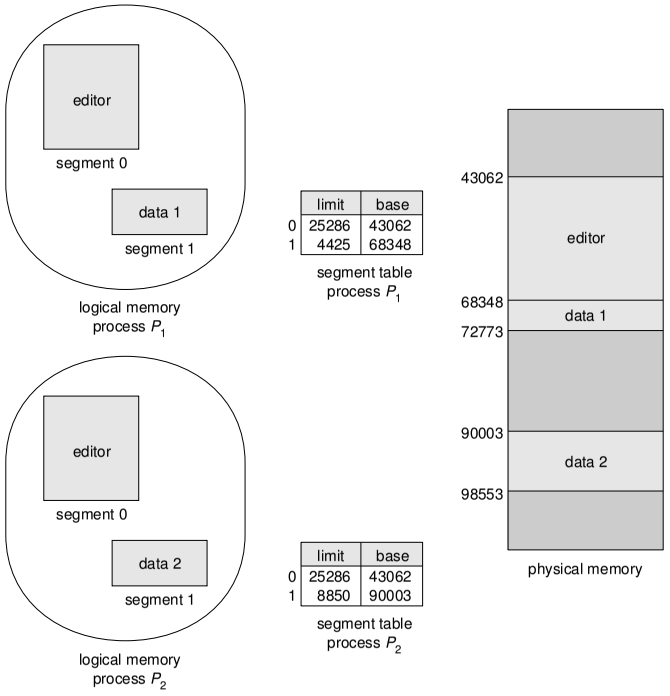
- L'indirizzo logico consiste in un coppia
  <segment-number, offset>.
- La *segment table* mappa gli indirizzi bidimensionali dell'utente negli indirizzi fisici unidimensionali. Ogni entry ha
  - *base:* indirizzo fisico di inizio del segmento
  - *limit:* lunghezza del segmento
- *Segment-table base 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.

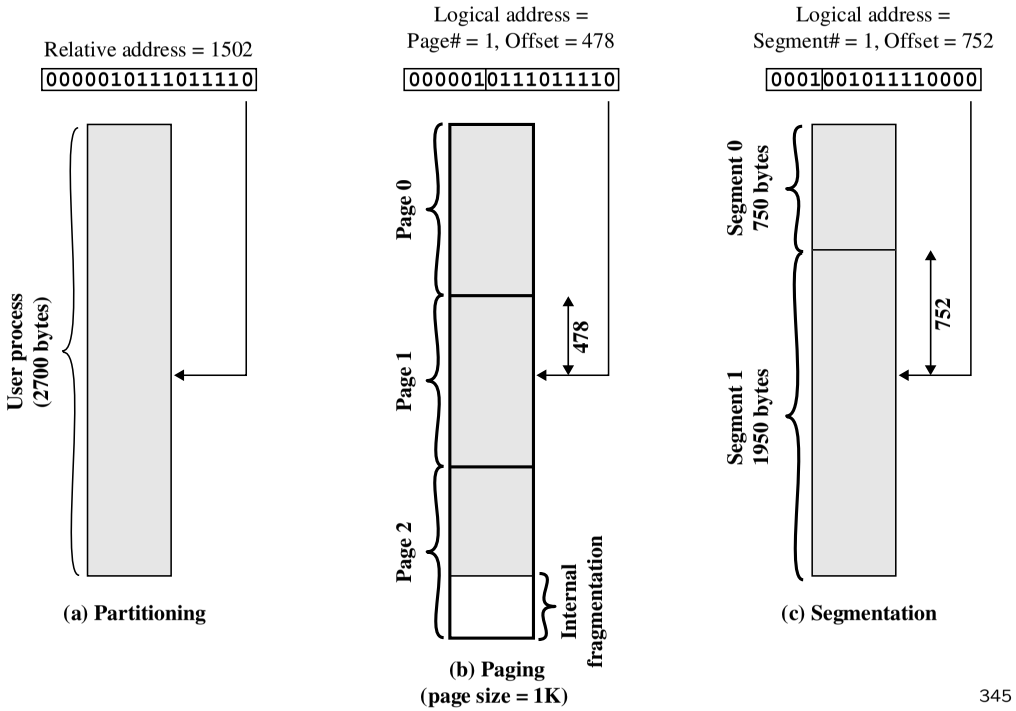
Hardware per la segmentazione

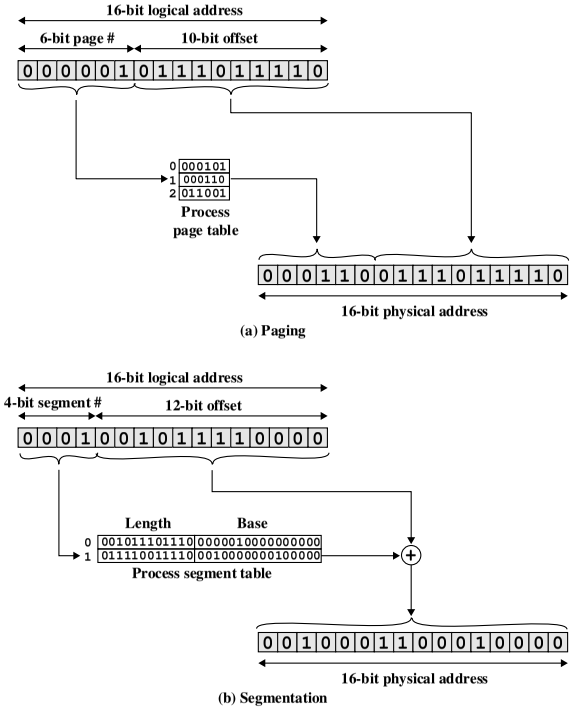


Architettura della Segmentazione (cont.)

- 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.



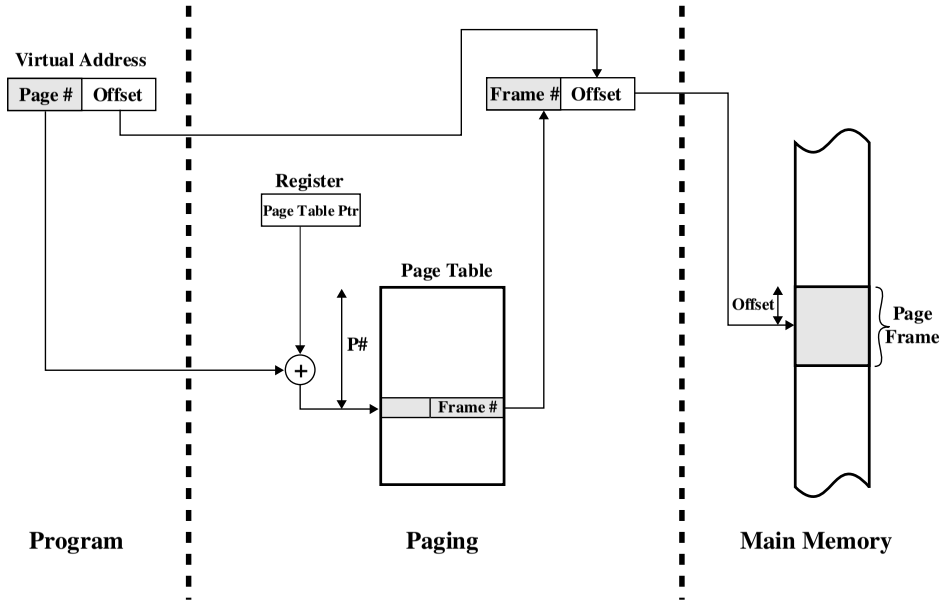




Implementazione della Page Table

- 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: ci sono $2^{20} > 10^6$ entry. A 16 bit l'una (max RAM = 256M) $\Rightarrow$ 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

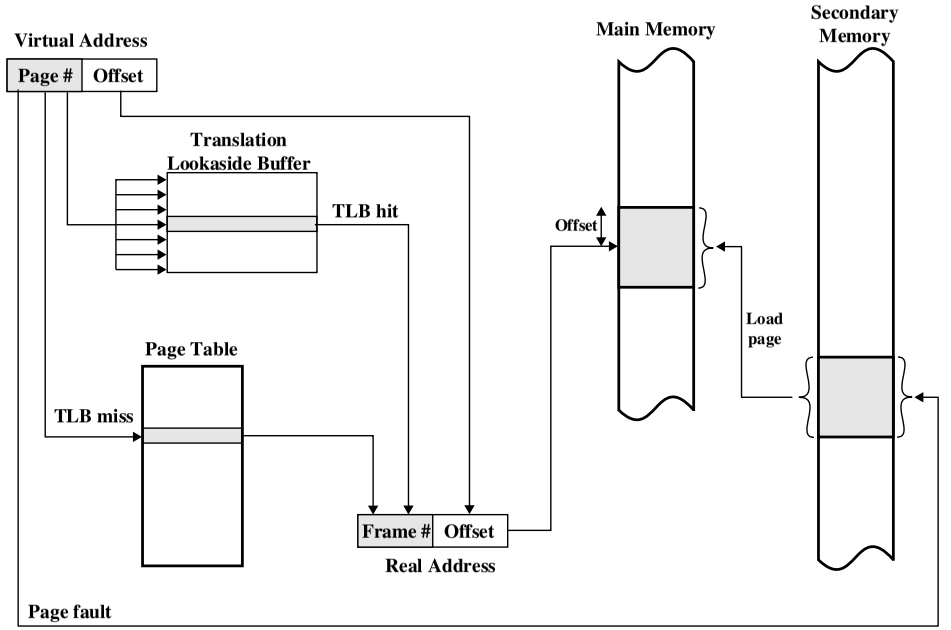
Paginazione con page table in memoria



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 $\Rightarrow$ 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 $\Rightarrow$ 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

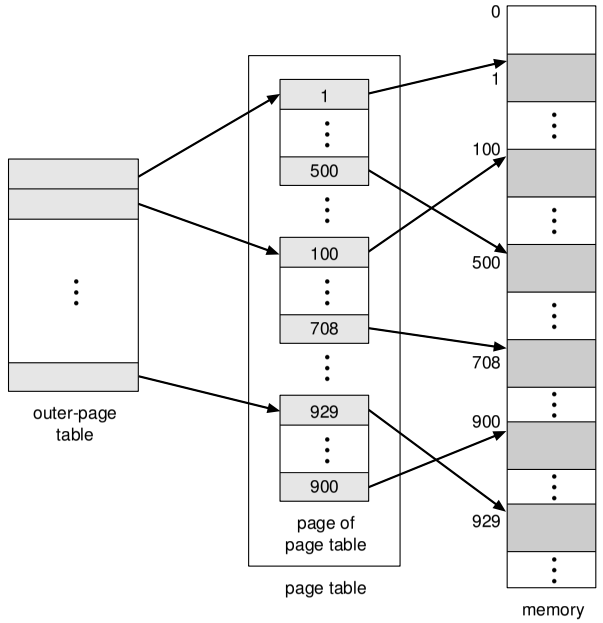
- ϵ = tempo del lookup associativo
- t = tempo della memoria
- α = *Hit ratio*: percentuale dei page # reperiti nel TLB (dipende dalla grandezza del TLB, dalla natura del programma...)

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

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

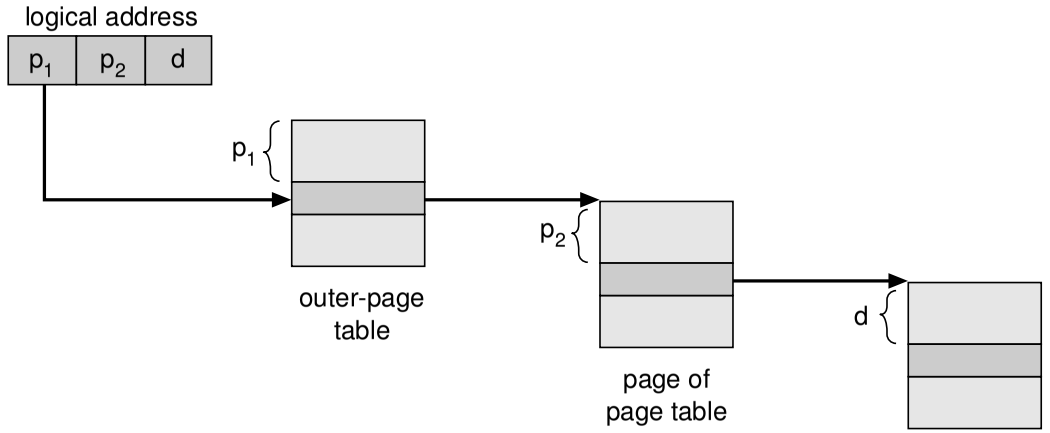
Paginazione a più livelli

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.
- Il caching degli indirizzi di pagina permette di ridurre drasticamente l'impatto degli accessi multipli

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

- 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)

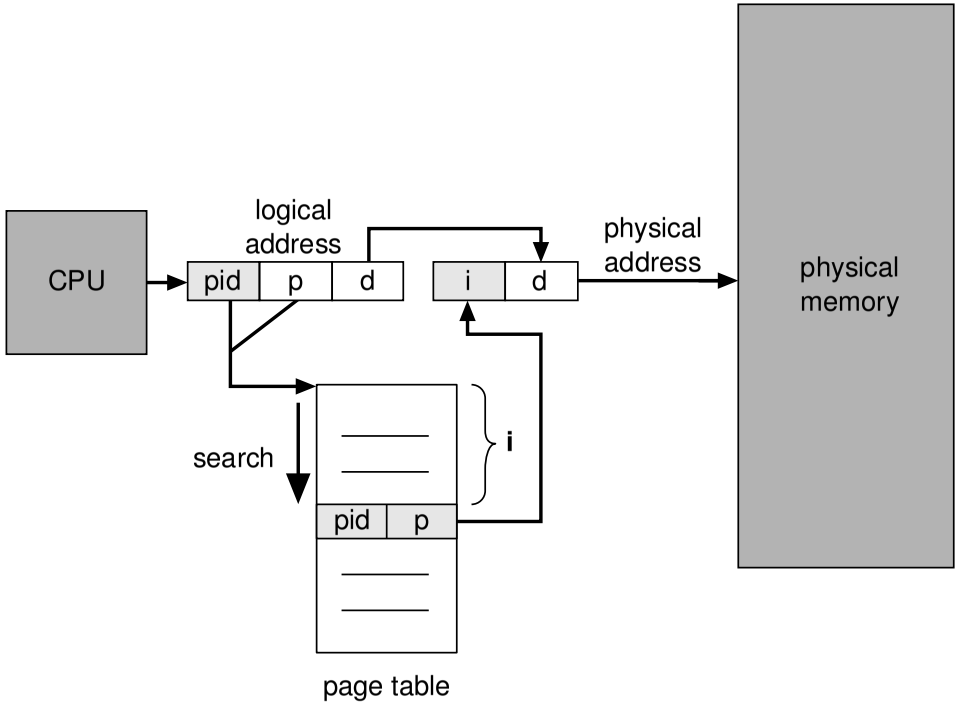
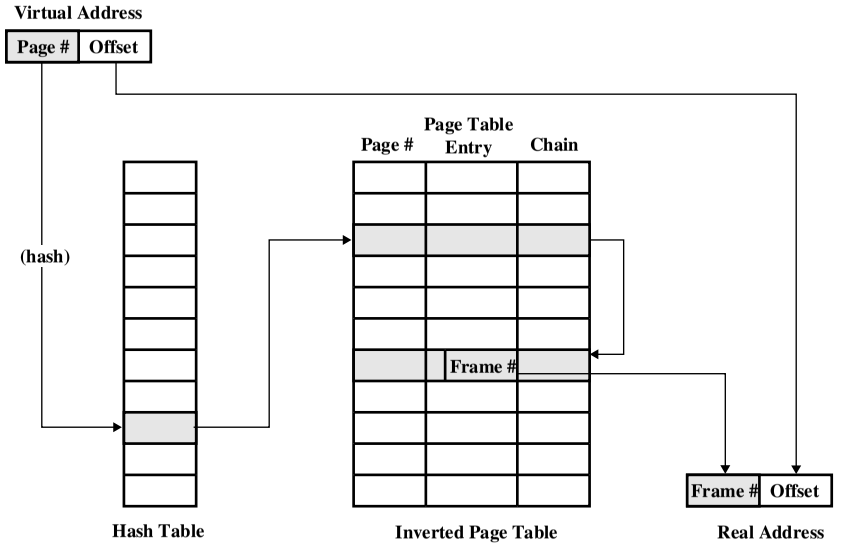


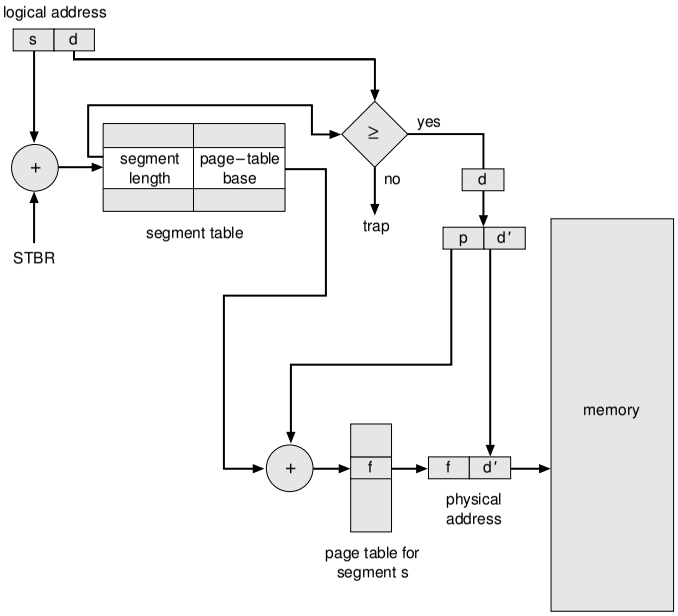
Tabella delle pagine invertita con hashing

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).

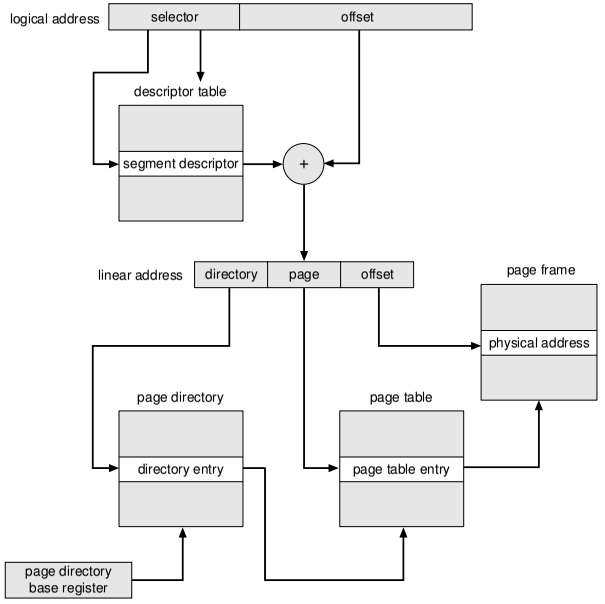


Segmentazione con paginazione: MULTICS

- 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

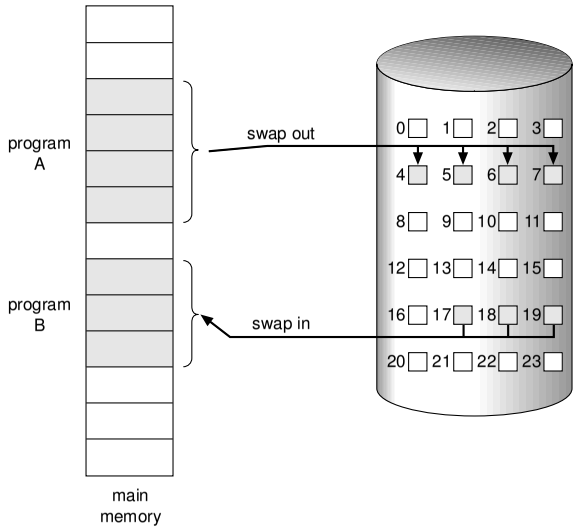
Paginazione su richiesta

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

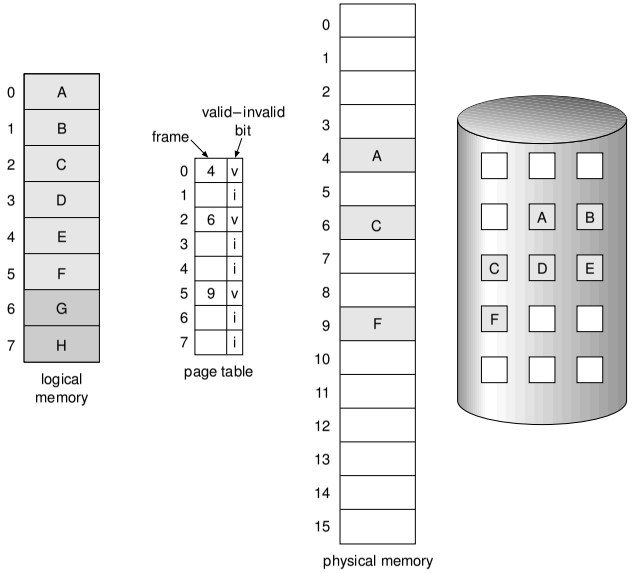
  Pager: processo che implementa una politica di gestione delle pagine dei processi (caricamento/scaricamento).

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 guarda in un'altra tabella se è stata 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:

  – trovare qualche pagina in memoria, ma in realtà non usata, e scaricarla su disco (*swap out*)

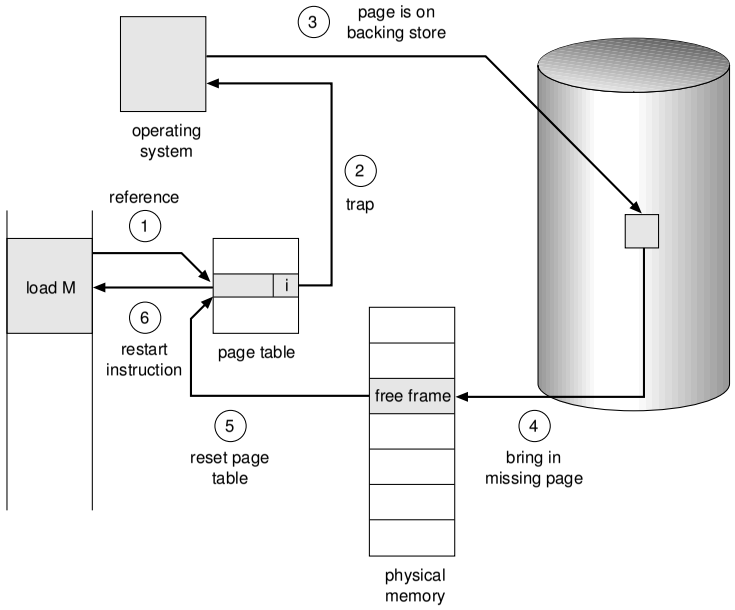
  – Caricare la pagina richiesta nel frame così liberato (*swap in*)

  – 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)

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

Esempio di Demand Paging

- Tempo di accesso alla memoria (comprensivo del tempo di traduzione): 60 nsec
- Assumiamo che 50% delle volte che una pagina deve essere rimpiazzata, è stata modificata e quindi deve essere scaricata su disco.
- Swap Page Time = 5 msec = $5e6$ nsec (disco molto veloce!)
- $EAT = 60(1-p) + 5e6 * 1.5 * p = 60 + (7.5e6 - 60)p$ in nsec
- Si ha un degrado del 10% quando $p = 6/(7.5e6 - 60) = 1/1250000$

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

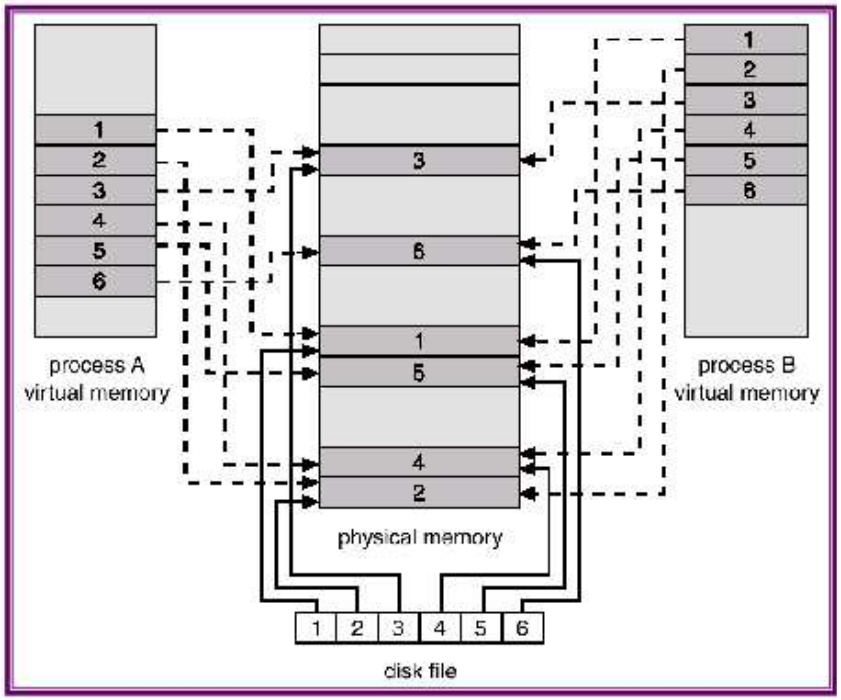
Creazione dei processi: Copy on Write

- 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



Creazione dei processi: 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

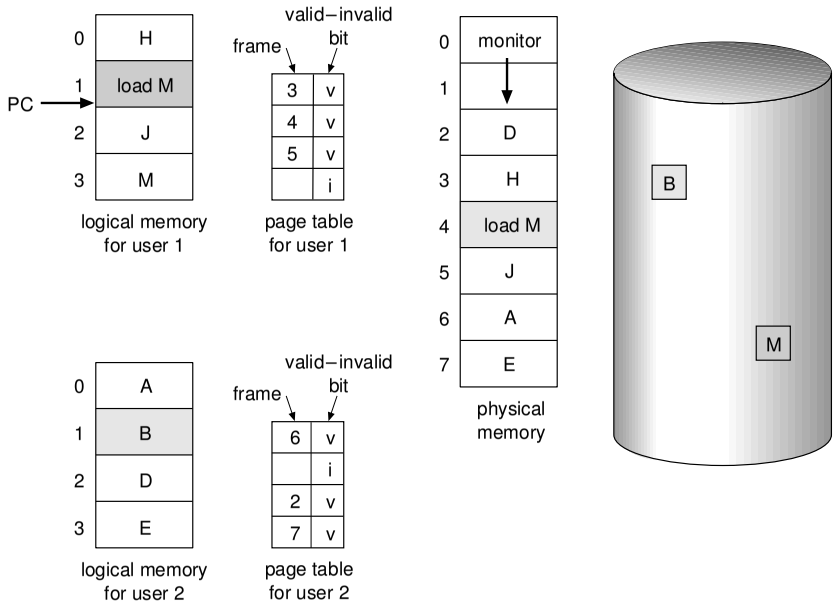
- Ad un page fault, può succedere che non esistono frame liberi

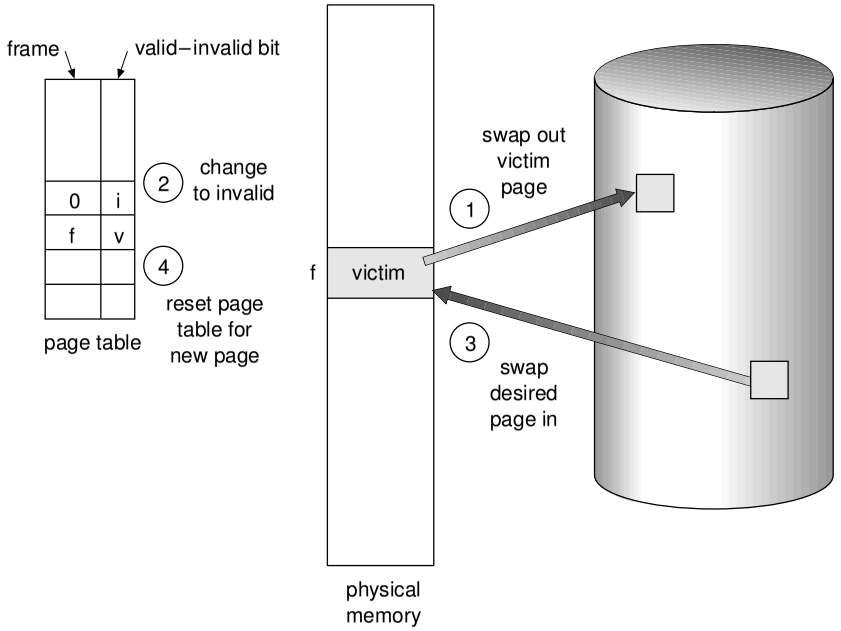
- Si modifica la routine di gestione del page fault aggiungendo la *sostituzione delle pagine* che libera un frame occupato (*vittima*)

- 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.)





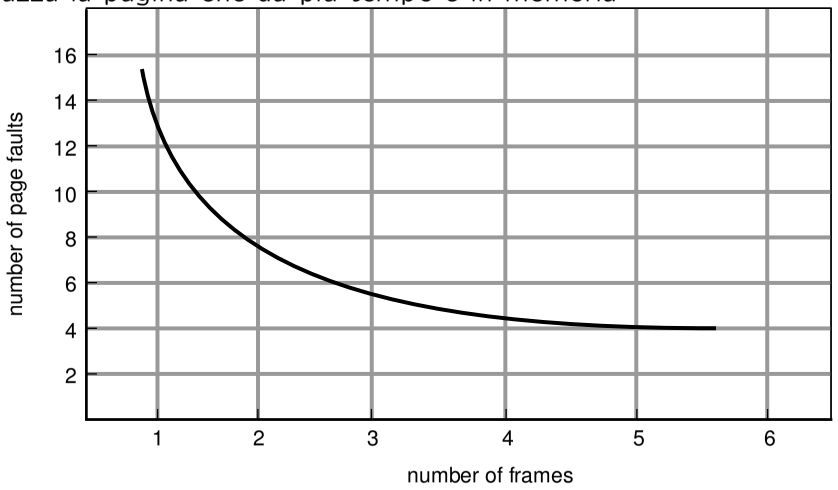
Algoritmi di rimpiazzamento delle pagine

- 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.
- 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.

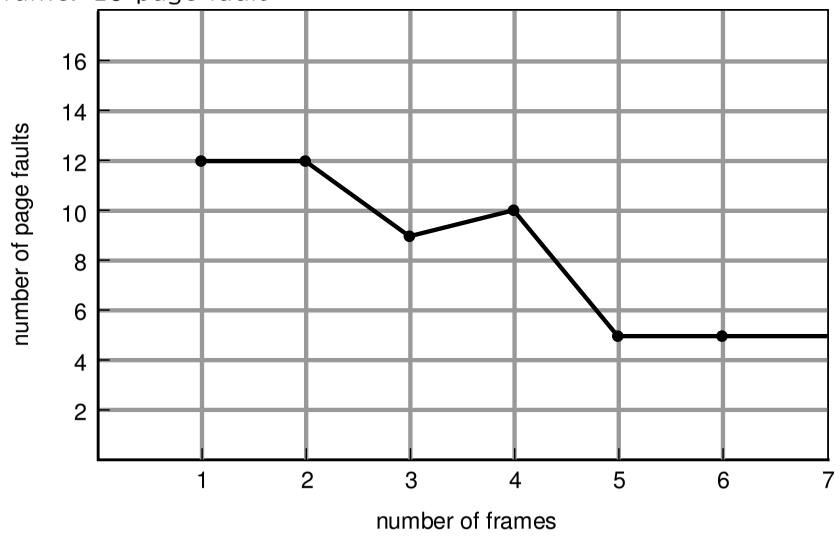
Algoritmo First-In-First-Out (FIFO)

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



- 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!

Algoritmo ottimale (OPT o MIN)

- Si rimpiazza la pagina che non verrà riusata per il periodo più lungo
- 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
- 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 shifta gli array di tutte le pagine immettendovi i bit di riferimento, che vengono settato a 0
- Si rimpiazza la pagina che ha il numero più basso nell'array

Differenze con LRU:

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

In genere comunque è una buona approssimazione.

Page	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

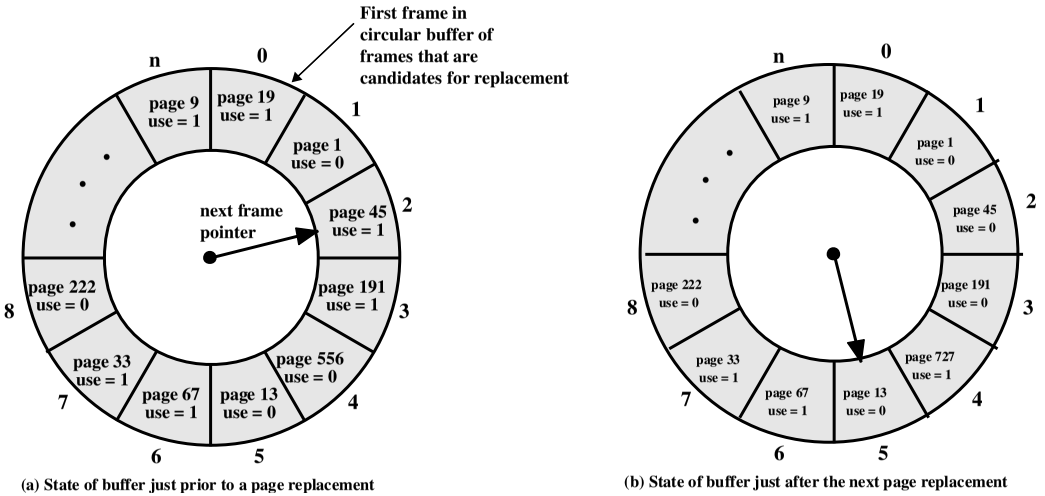
Approssimazioni di LRU: CLOCK (o "Second chance")

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

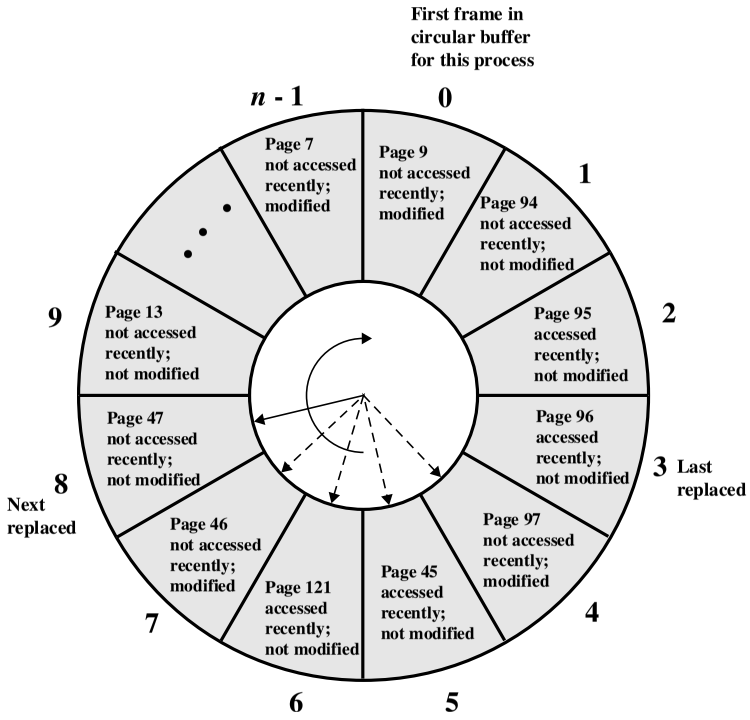


(a) State of buffer just prior to a page replacement

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

Approssimazioni di LRU: CLOCK migliorato

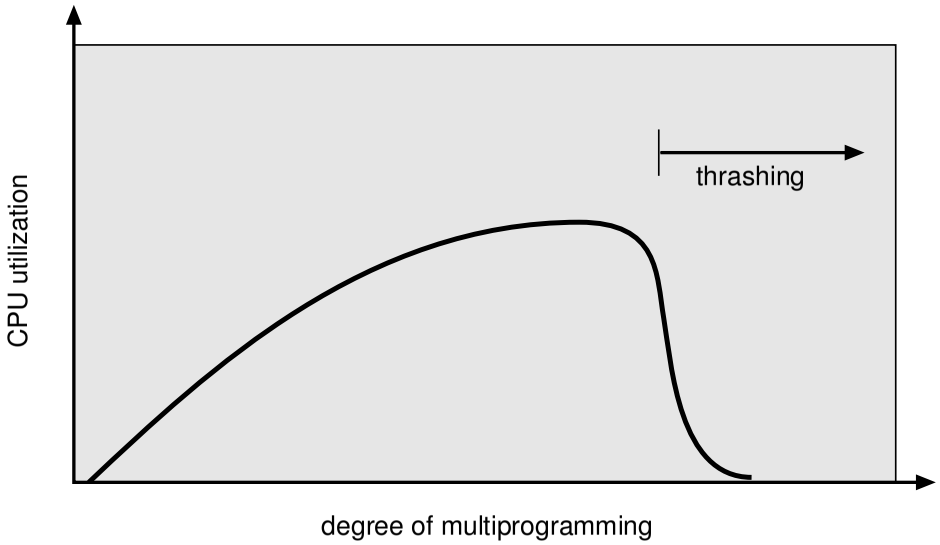
- Usare due bit per pagina: il reference (r) e il dirty (d) bit
  - non usata recentemente, non modificata ($r = 0, d = 0$): buona
  - non usata recentemente, ma modificata ($r = 0, d = 1$): meno buona
  - usata recentemente, non modificata ($r = 1, d = 0$): probabilmente verrà riusata
  - usata recentemente 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

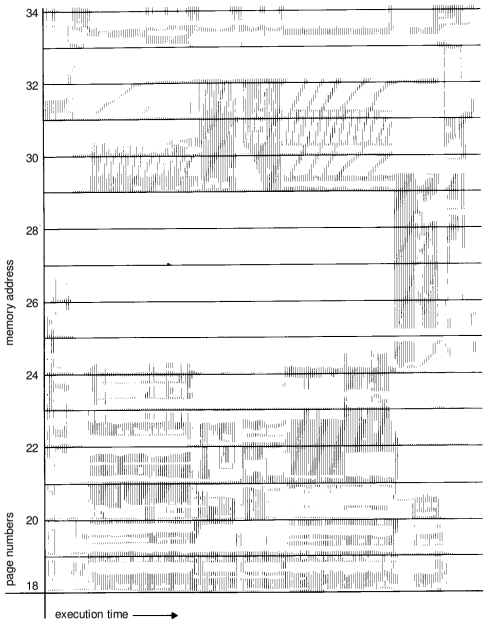
- Il thrashing del sistema avviene quando la memoria fisica è inferiore somma delle località dei processi in esecuzione. Può essere causato da un processo che si espande e in presenza di rimpiazzamento globale.



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)

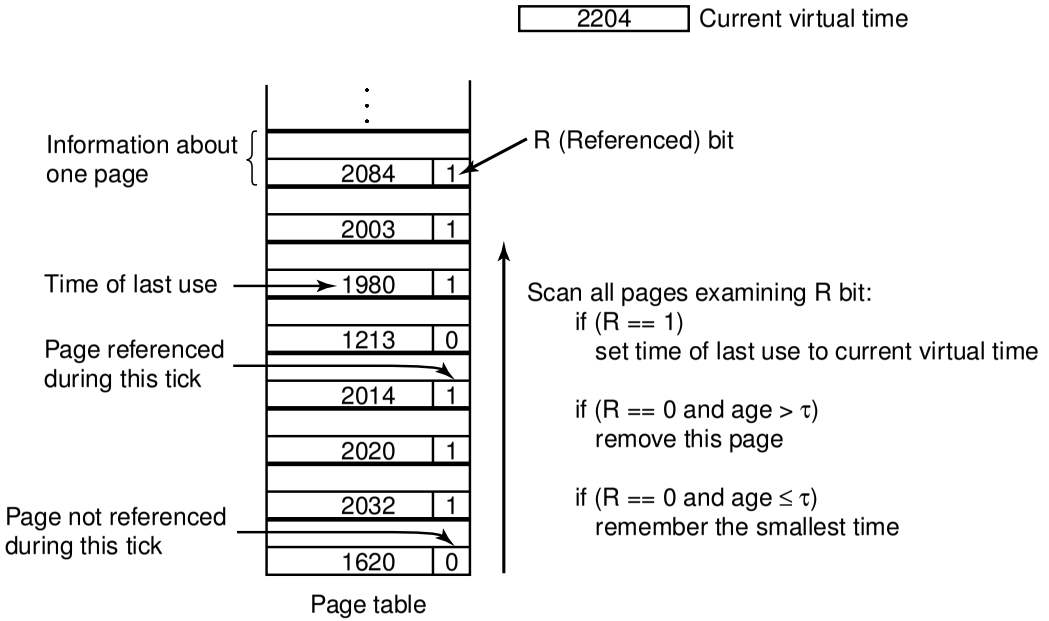
Si impedisce il thrashing, massimizzando nel contempo l'uso della CPU.

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.



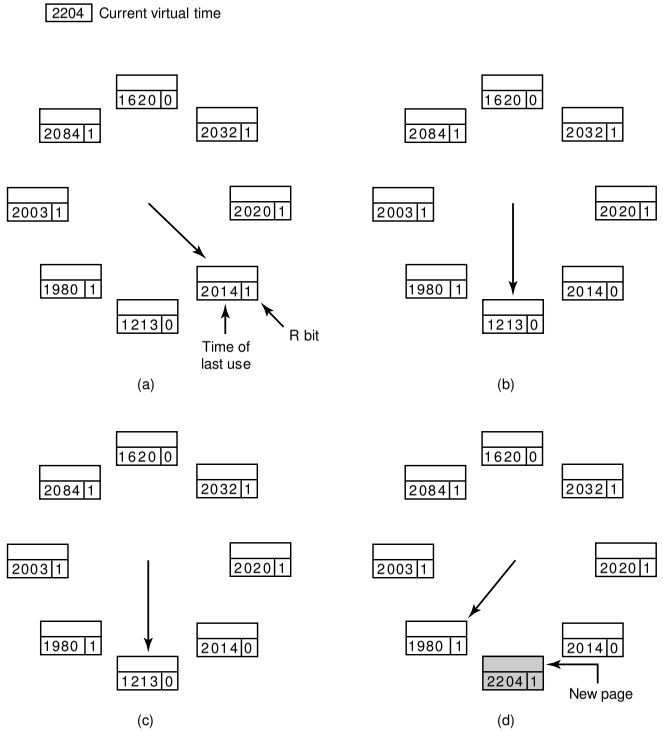
Algoritmo di rimpiazzamento WSClock

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)

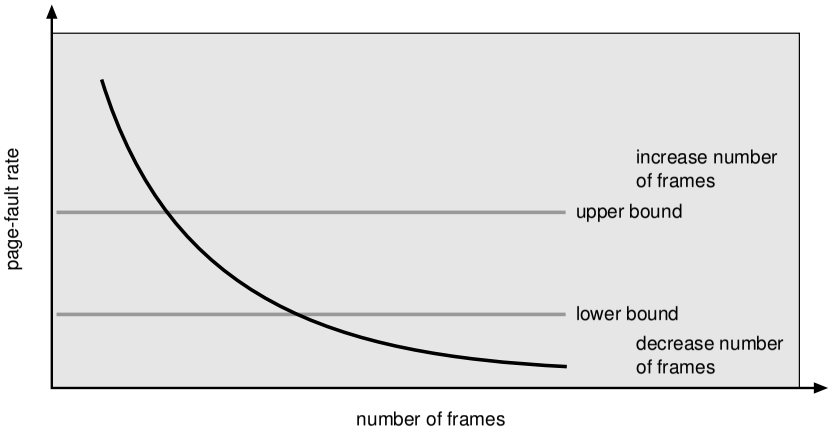
- si mantiene un contatore del tempo di CPU impiegato da ogni processo
- le pagine sono organizzate ad orologio; inizialmente, lista vuota
- ogni entry contiene i reference e dirty bit R, M, e un registro *Time of last use*, che viene copiato dal contatore ad ogni reference. La differenza tra questo 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

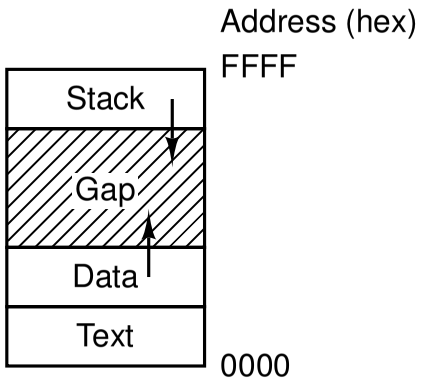
- Durante I/O, i frame contenenti i buffer non possono essere swappati
  - I/O solo in memoria di sistema ⇒ costoso
  - Lockare in memoria i frame contenenti buffer di I/O (*I/O interlock*) ⇒ delicato (un frame lockato potrebbe non essere più rilasciato)

Modello della memoria in Unix

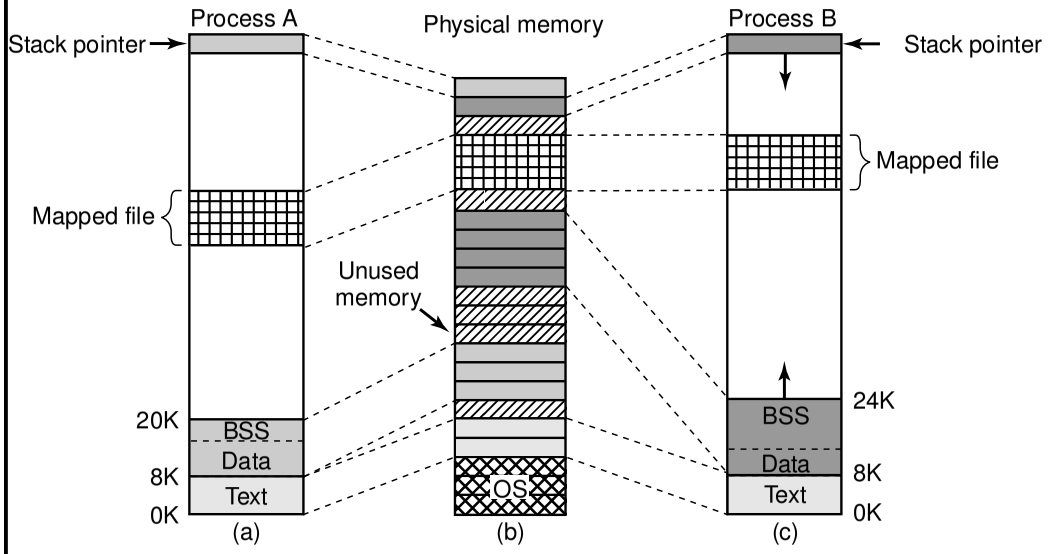
Ogni processo UNIX ha uno spazio indirizzi separato. Non vede le zone di memoria dedicati agli altri processi.

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.



Mappatura dei segmenti in memoria reale



Gestione della memoria in UNIX

- Dipende dall'hardware sottostante
  - Fino a 3BSD (1978): solo segmentazione con swapping.
  - Dal 3BSD: paginazione; da 4.2BSD (1983): paginazione on demand
  - Esistono anche UNIX senza memoria virtuale
- Viene mantenuta una free list di frame con buffering e prepaging
- Le pagine vengono allocate dalla lista libera dal kernel, su base libera, con *copy-on-write*
- La free list viene mantenuta entro un certo livello dal processo *pagedaemon*. Applica varianti del CLOCK
- Lo swapping rimane come soluzione contro il thrashing: uno *scheduler a medio termine* decide il grado di multiprogrammazione. Si attiva automaticamente, in situazioni estreme

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.

Quando si alloca la memoria (cont.)

- *minfree* = parametro fissato al boot, in proporzione alla memoria fisica. Tipicamente, 100K–5M.
- se la free list scende sotto *minfree*, il kernel si rifiuta di allocare nuove pagine di memoria.
- Le pagine vengono cercate sulla free list, prima di caricarle da disco
- 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 lista libera
- I segmenti di codice condiviso non vengono swappati (solitamente) $\Rightarrow$ meno I/O e meno memoria usata
- Le pagine vengono lockate in memoria per I/O asincrono

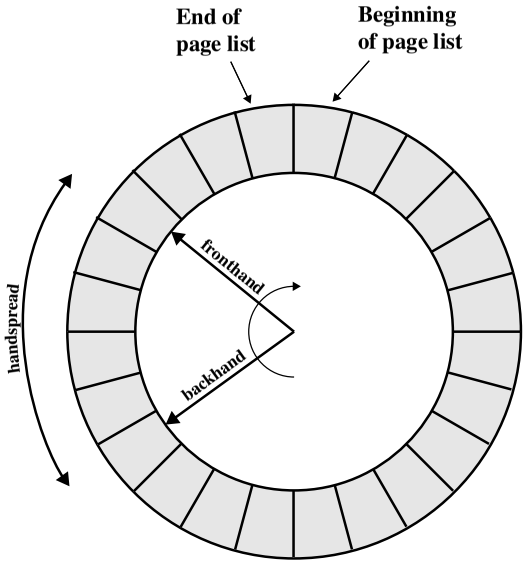
Quando si libera memoria

- La sostituzione di pagina viene implementata da un processo, il *pagedaemon*, noto anche come *pageout* (PID=2) su Solaris, *kswapd* (PID=5) su Linux, . . . Spesso è un thread di kernel
- Viene lanciato al boot e si attiva ad intervalli regolari (tip. 2–4 volte al secondo) o su richiesta del kernel
- Parametro: *lotsfree* = n. limite di frame liberi. Fissato al boot; 5–10% dei frame totali (1M–64M)
- Il pagedaemon interviene quando # frame liberi $<$ *lotsfree*
- Applica un rimpiazzamento globale
- In alcuni UNIX (specie quelli più vecchi): algoritmo dell'orologio. Inadeguato per grandi quantità di memoria: libera pagine troppo lentamente

CLOCK a due lancette

Negli UNIX moderni: algoritmo dell'orologio *a 2 lancette*.
Due puntatori scorrono la lista delle pagine allocate

- il primo azzera il reference bit
- il secondo sceglie la pagina vittima: se trova $r = 0$ (la pagina non è stata usata), il frame viene salvato e posto sulla lista libera
- si fanno avanzare i due puntatori
- si ripete finché # frame liberi $\geq$ *lotsfree*



CLOCK a due lancette (cont.)

- La distanza tra i puntatori (*handspread*) viene decisa al boot, per liberare frame abbastanza rapidamente
- La velocità di scan dipende da quanto manca a *lotsfree*: meno memoria libera c'è, più velocemente si muovono i puntatori
- Variante: "isteresi"
  - ulteriore parametro *maxfree*; *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) decide quale processo deve essere swappato su disco
- Viene lanciato al boot; spesso è un thread di kernel
- Parametro: *desfree*, impostato al boot. 100K–10M.
  $$minfree < desfree < lotsfree$$
- Lo swapper si sveglia ogni 1–2 secondi, e interviene solo se
  $$\#\text{ frame liberi} < minfree \quad \text{e}$$
  $$\#\text{ frame liberi} < desfree \text{ nella storia recente}$$

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 Priority+Residence Time più alto.
  - 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 le page tables (*easy swap*) oppure no (*hard swap*);
3. se è un hard swap, libera memoria swappando qualche altro processo;
4. carica le page table in memoria e mette il processo in "ready, in memory"
5. Solitamente, si provvede anche a prepaginare le pagine swappate

Si ripete finché non ci sono processi da caricare.

Considerazioni

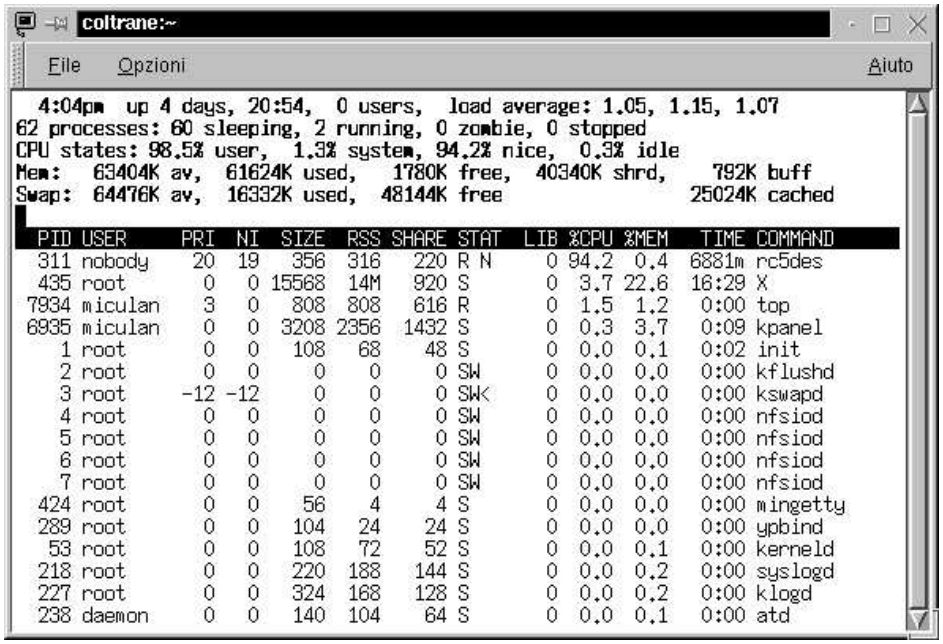
Interazione tra scheduling a breve termine, a medio termine e paginazione

- 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
- (comunque, i processi non dovrebbero mai essere swappati!)
- processi real-time (i.e., in classi SCHED_FIFO e SCHED_RR) non vengono mai swappati

Monitorare i processi e la memoria: il comando ps lx

```
  F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
100   121   632   622   0   0   1696     0 wait4  SW   ?          0:00 [Default]
000   121   643   632   0   0   5564  1024 do_pol S    ?          0:01 gnome-sessi
000   121   660     1   1   0   5588   520 do_sel S    ?          0:06 gnome-smpro
000   121   664     1   3   0   4840  2208 do_sel S    ?          1:06 enlightenme
000   121   680     1   0   0   2704   316 do_pol S    ?          0:00 gnome-name-
000   121   683     1   0   0   7852  4052 do_pol S    ?          0:04 panel --sm-
000   121   685     1   0   0   3024  1464 do_sel S    ?          0:10 xscreensave
000   121   687     1   0   0   7836   852 do_pol S    ?          0:04 gmc --sm-co
000   121   689     1   0   0   3592   884 do_sel S    ?          0:01 Eterm
000   121   697   689   0   0   1748     0 wait4  SW   ttyp1      0:00 [bash]
000   121   727     1  14   5   6424  1008 do_pol SN   ?         13:23 cpumemusage
000   121   729     1   0   0   6484   556 do_pol S    ?          0:08 gnomexmms -
000   121   731     1   3   0   6436  1584 do_pol S    ?          1:07 gnomepager_
000   121   733     1   0   0   6472   804 do_pol S    ?          0:04 clockmail_a
000   121   755   697   0   0   5180  2368 do_sel S    ttyp1      0:02 pine -i
000   121  1020     1   0   0   4796  3392 do_sel S    ?          0:00 Eterm
000   121  1023  1020   0   0   1752  1048 read_c S    ttyp0      0:00 -bash
000   121  1034  1023   4   0   7420  6136 do_sel S    ttyp0      1:19 sdr
604   121  1054  1034  10  10      0     0 do_exi ZN   ttyp0      0:02 [vic <defun
000   121  1062     1   0   0   4576  3172 do_sel S    ?          0:02 Eterm
000   121  1065  1062   0   0   1776  1092 read_c S    ttyp2      0:00 -bash
```


Monitorare i processi e la memoria: il comando top



Monitorare i processi e la memoria: il comando vmstat

```
miculan@ten:~> vmstat 5
 procs      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s0 s1 s2 s3   in   sy   cs us sy id
 0 0 0 102232  7264   0 137  5  0  0  0  0  8  0  0  0  442 24730 365 26 13 61
 0 0 0  99260  6164   0 468 44  0  0 1440 0 14 1  0  0  479 21848 358 30 36 34
 0 0 0  96324  3964   0 146  4  0  0 808 0  1  0  0  0  213 24221 186 38  9 53
 0 0 0  95148  2740   0 100  4  0  0 208 0  0  0  0  0  364 25165 331 30 10 60
 0 0 0  95008  2560   0  79  0  4  4  0  0  0  0  0  0  472 25218 400 28 10 62
 0 0 0  94592  2136   0  11  1  3  3  0  0  6  0  0  0  324 25114 224 24 10 66
 0 0 0  93192  2040   0  61 76 24 45  0 17  4  0  0  0  464 27347 382 38 17 45
 0 0 0  92420  1984   0  51  0 32 81  0 44  4  0  0  0  291 26620 246 35 14 52
 0 0 0  93268  2084   3  69  7 32 56  0 23  6  0  0  0  292 23713 272 24 10 66
 0 0 0  93572  2192   2  63 10 12 12  0  0  3  0  0  0  309 23867 263 25 10 65
 0 0 0  94216  2540   0   1  5  7  7  0  0  1  0  0  0  257 23890 215 23  7 69
 0 0 0  94128  2428   0  36  0  4  4  0  0  0  0  0  0  445 25718 391 30 11 59
 0 0 0  93784  2224   2  44  1 12 13  0  0 11  1  0  0  299 23573 223 21  9 69
 0 1 0  93600  2224   4  96  1 93 100 0  7  6  0  0  0  530 23410 443 23 14 63
 0 0 0  94052  2472   0   4  0  2  2  0  0  0  0  0  0  328 24105 322 26  9 65
 0 0 0  93996  2416   0   6  4  5  5  0  0  1  0  0  0  370 24347 343 23 11 66
 0 0 0  94216  2488   0   3  3  2  2  0  0  1  0  0  0  218 24154 196 21  9 70
 0 0 0  94344  2580   2   0  4 31 31  0  0  0  0  0  0  219 23826 176 21  8 72
 0 0 0  94288  2364   0   0 124 1  1  0  0  0  0  0  0  366 23770 271 20 10 70
 0 0 0  94708  2504   0  74  0  1  1  0  0  8  1  0  0  343 24442 261 24 10 66
 0 0 0  94516  2336   2  51 34 21 28  0  6  2  0  0  0  359 24339 333 22 12 66
 0 0 0  92708  5660   0  16 36 12 73  0 60  1  0  0  0  350 23529 361 24 11 65
 0 0 0  94720  6460   0  25  4  0  0  0  0  0  0  0  0  330 23820 292 35 10 55
```

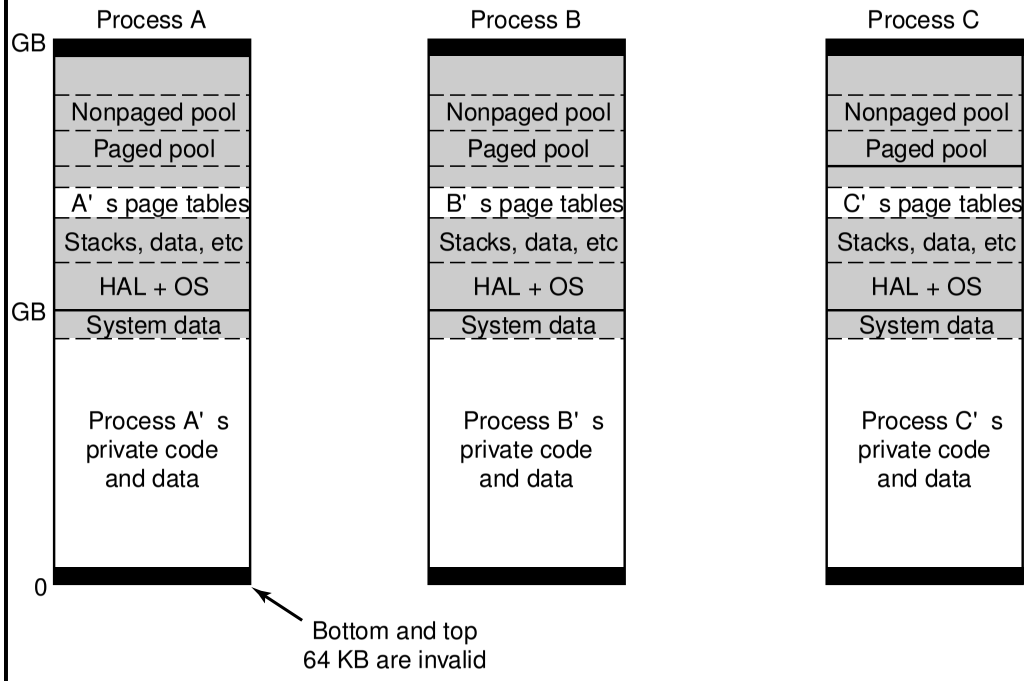

Modello della memoria in Windows 2000

Ogni task di Windows riceve uno spazio indirizzi di 4G, diviso in due parti

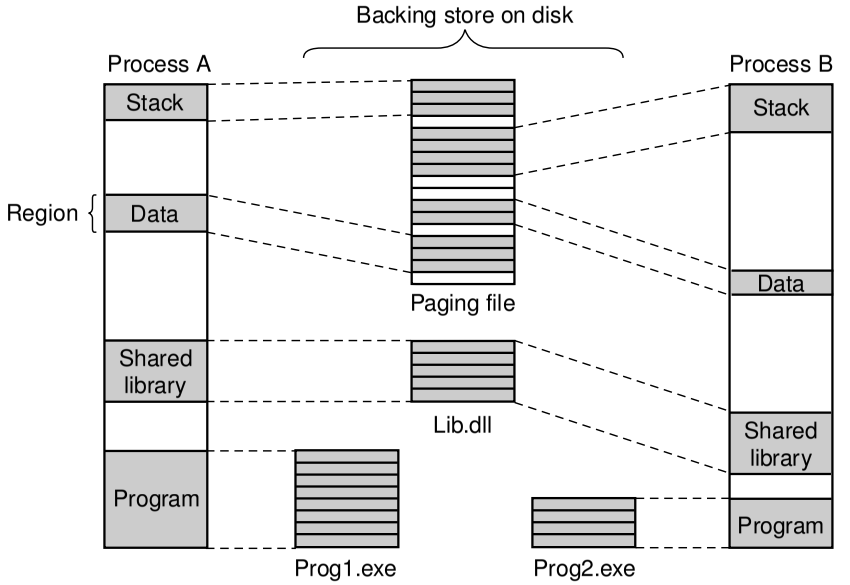
- codice e dati, nella parte bassa (< 2G) Liberamente accessibile.
- kernel e strutture di sistema (comprese le page tables) nella parte alta La maggior parte di questo spazio non è accessibile, neanche in lettura.
- i primi ed ultimi 64kb sono invalidi per individuare rapidamente errori di programmazione (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.



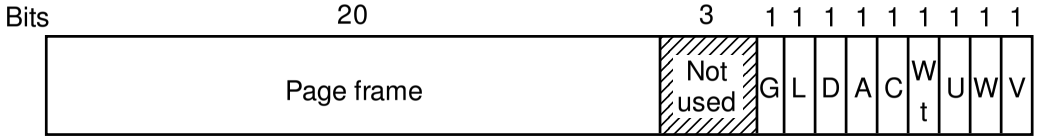
Mappatura dei segmenti in memoria reale e su file



Gestione dei page fault

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

Page table entry per il Pentium:



G: Page is global to all processes
L: Large (4-MB) page
D: Page is dirty
A: Page has been accessed
C: Caching enabled/disabled

Wt: Write through (no caching)
U: Page is accessible in user mode
W: Writing to the page permitted
V: Valid page table entry

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
- *Reserved*: riservata da un processo ma non ancora usata. Non fa parte del ws fino a che non viene veramente usata.
- *Committed*: usata da un processo e associata ad un blocco su disco

Casi di page fault

- La pagina riferita non è committed
  ⇒ Terminazione del processo
- Violazione di protezione
  ⇒ Terminazione del processo
- Scrittura su una pagina condivisa
  ⇒ Copy-on-write su una pagina reserved
- Crescita dello stack
  ⇒ Allocazione di una pagina azzerata
- La pagina riferita è salvata ma non attualmente caricata in memoria
  ⇒ il vero page fault: pagein della pagina mancante

Algoritmo di rimpiazzamento di pagina

La paginazione è basata sul modello del Working Set

- ogni processo ha una dim. minima e massima (non sono limiti hard)
- Tutti i processi iniziano con lo stesso min e max (risp. 20-50 e 45-345, in proporzione alla RAM; modificabile dall'admin)
- Ad un page fault:
  - 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

Sistemi di I/O

- Incredibile varietà di dispositivi di I/O
- Grossolanamente, tre categorie

  Human readable: orientate all'interazione con l'utente. Es.: terminale, mouse

  Machine readable: adatte alla comunicazione con la macchina. Es.: disco, nastro

  Comunicazione: adatte alla comunicazione tra calcolatori. Es.: modem, schede di rete

Livelli di astrazione

- Per un ingegnere, un dispositivo è un insieme di circuiteria elettronica, meccanica, temporizzazioni, controlli, campi magnetici, onde, ...
- Il programmatore ha una visione *funzionale:* vuole sapere *cosa* fa un dispositivo, e come farglielo fare, ma non gli interessa sapere *come* lo fa.
- Vero in parte anche per il sistema operativo: spesso i dettagli di più basso livello vengono nascosti dal *controller*.
- (Anche se ultimamente si vedono sempre più dispositivi a controllo software...)
- Tuttavia nella progettazione del software di I/O è necessario tenere presente dei principi generali di I/O

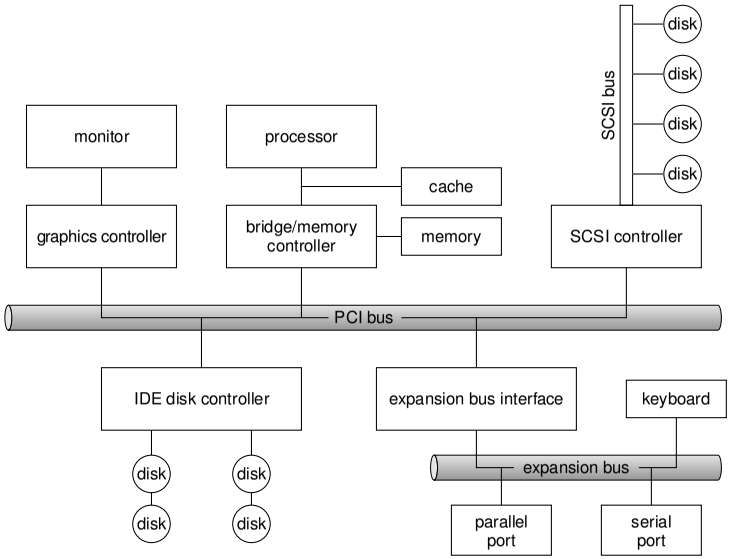
Dispositivi a blocchi e a carattere

Suddivisione logica nel modo di accesso:

- Dispositivi a blocchi: permettono l'accesso diretto ad un insieme finito di blocchi di dimensione costante. Il trasferimento è strutturato a blocchi. Esempio: dischi.
- Dispositivi a carattere: generano o accettano uno stream di dati, non strutturati. Non permettono indirizzamento. Esempio: tastiera
- Ci sono dispositivi che esulano da queste categorie (es. timer), o che sono difficili da classificare (nastri).

Comunicazione CPU-I/O

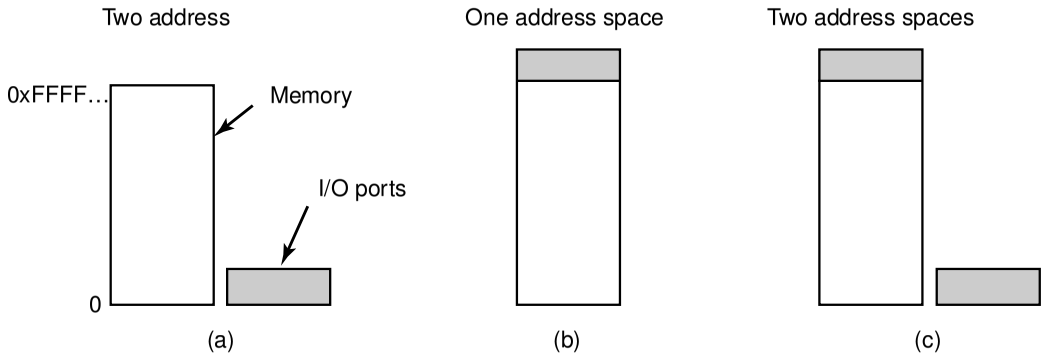
Concetti comuni: *porta*, *bus* (daisychain o accesso diretto condiviso), *controller*



Comunicazione CPU-I/O (cont)

Due modi per comunicare con il (controller del) dispositivo

- insieme di istruzioni di I/O dedicate: facili da controllare, ma impossibili da usare a livello utente (si deve passare sempre per il kernel)
- I/O mappato in memoria: una parte dello spazio indirizzi è collegato ai registri del controller. Più efficiente e flessibile. Il controllo è delegato alle tecniche di gestione della memoria (se esiste), es: paginazione.
- I/O separato in memoria: un segmento a parte distinto dallo spazio indirizzi è collegato ai registri del controller.



Modi di I/O

	Senza interrupt	Con interrupt
trasferimento attraverso il processore	Programmed I/O	Interrupt-driven I/O
trasferimento diretto I/O-memoria		DMA, DVMA

Programmed I/O (I/O a interrogazione ciclica): Il processore manda un comando di I/O, e poi attende che l'operazione sia terminata, testando lo stato del dispositivo con un loop busy-wait (*polling*).

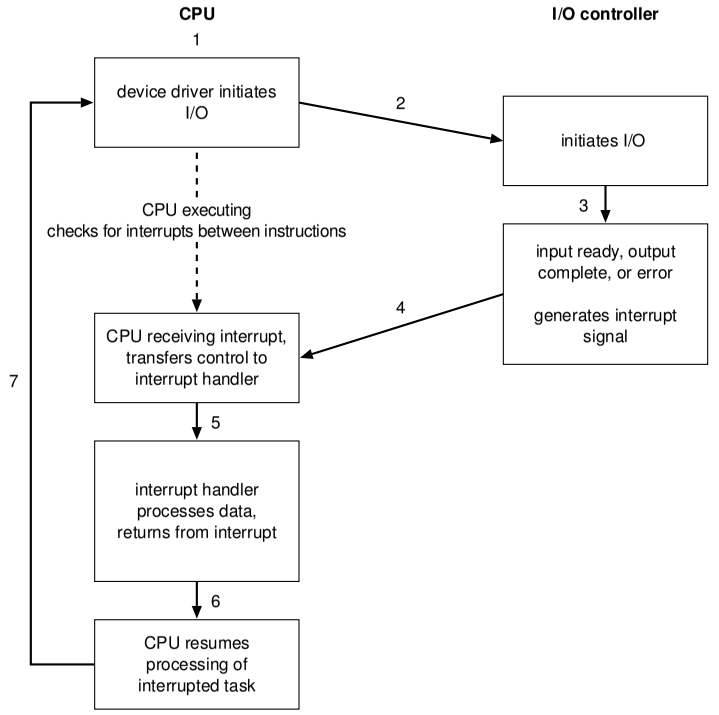
Efficiente solo se la velocità del dispositivo è paragonabile con quella della CPU.

I/O a interrupt

Il processore manda un comando di I/O; il processo viene sospeso. Quando l'I/O è terminato, un interrupt segnala che i dati sono pronti e il processo può essere ripreso. Nel frattempo, la CPU può mandare in esecuzione altri processi o altri thread dello stesso processo.

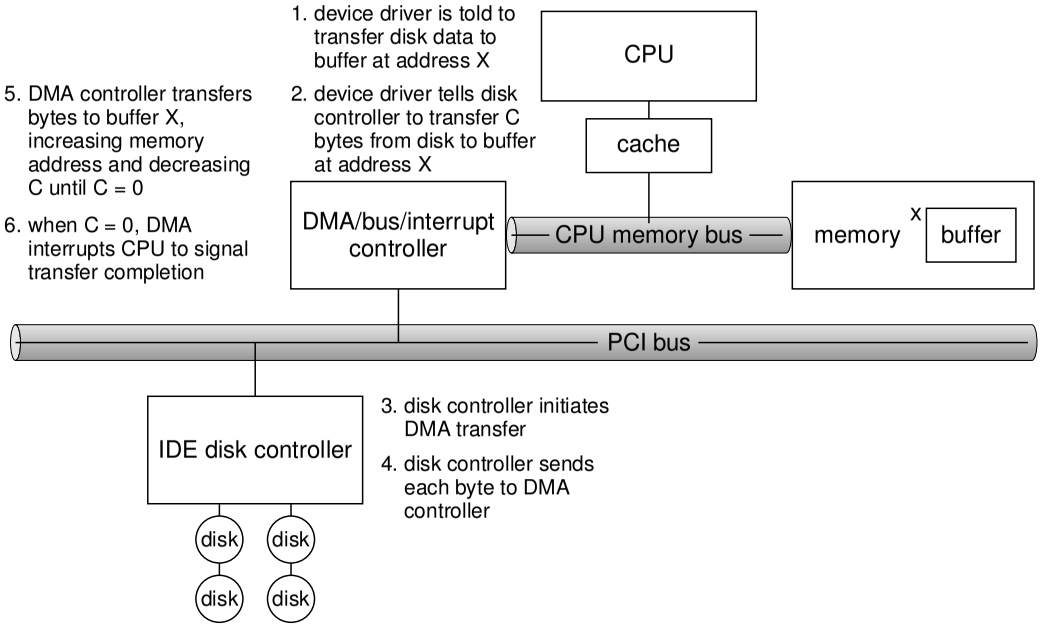
Vettore di interrupt: tabella che associa ad ogni interrupt l'indirizzo di una corrispondente routine di gestione.

Gli interrupt vengono usati anche per indicare eccezioni (e.g., divisione per zero)

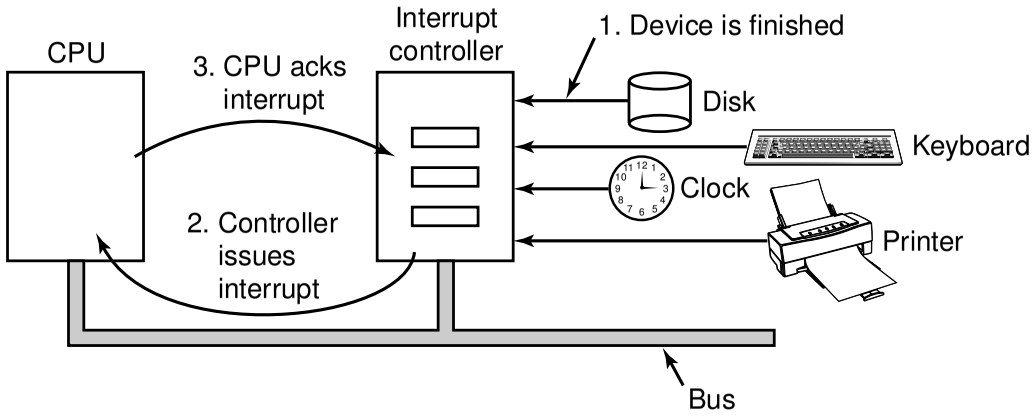


Direct Memory Access

- Richiede un controller DMA
- Il trasferimento avviene direttamente tra il dispositivo di I/O e la memoria *fisica*, bypassando la CPU.
- Il canale di DMA contende alla CPU l'accesso al bus di memoria: sottrazione di cicli (cycle stealing).
- Variante: Direct *Virtual* Memory Access: l'accesso diretto avviene allo spazio indirizzi virtuale del processo, e non a quello fisico. Esempio simile: AGP (mappatura attraverso la GART, *Graphic Address Relocation Table*)



Gestione degli interrupt



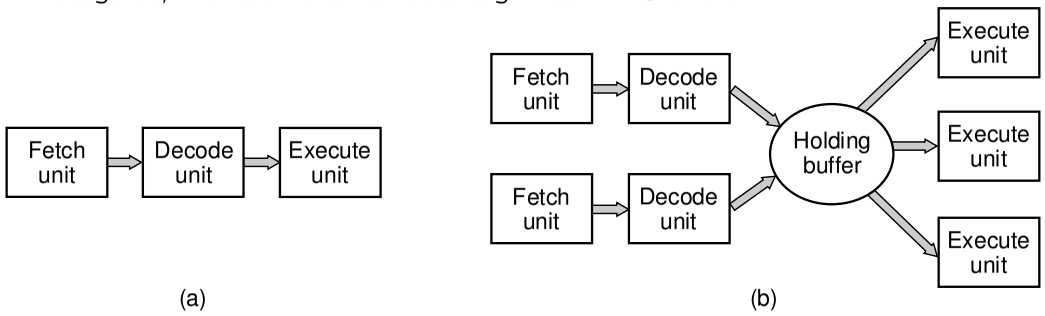
Gestione degli interrupt

- Quando arriva un interrupt, bisogna salvare lo stato della CPU:
  - Su una copia dei registri: gli interrupt non possono essere annidati, neanche per quelli a priorità maggiore
  - Su uno stack:
    - quello in spazio utente porta problemi di sicurezza e page fault
    - quello del kernel può portare overhead per la MMU e la cache

Gestione degli interrupt e CPU avanzate

- Le CPU con pipeline hanno grossi problemi: il PC non identifica nettamente il punto in cui riprendere l'esecuzione — anzi, punta alla prossima istruzione da mettere nella pipeline.

- Ancora peggio per le superscalari: le istruzioni possono essere già state eseguite, ma fuori ordine! cosa significa il PC allora?



Interruzioni precise

- Una interruzione è *precisa* se:
  - Il PC è salvato in un posto noto
  - TUTTE le istruzioni precedenti a quella puntata dal PC sono state eseguite COMPLETAMENTE
  - NESSUNA istruzione successiva a quella puntata dal PC è stata eseguita (ma possono essere state iniziate)
  - Lo stato dell'esecuzione dell'istruzione puntata dal PC è noto
- Se una macchina ha interruzioni imprecise:
  - è difficile riprendere esattamente l'esecuzione in hardware.
  - la CPU riversa tutto lo stato interno sullo stack e lascia che sia il SO a capire cosa deve essere fatto ancora
  - Rallenta la ricezione dell'interrupt e il ripristino dell'esecuzione $\Rightarrow$ grandi latenze...

- Avere interruzioni precise è complesso
  - la CPU deve tenere traccia dello stato interno: hardware complesso, meno spazio per cache e registri
  - "svuotare" le pipeline prima di servire l'interrupt: aumenta la latenza, entrano bolle (meglio avere pipeline corte).
- Pentium Pro e successivi, PowerPC, AMD K6-II, UltraSPARC, Alpha hanno interrupt precisi (ma non tutti), mentre IBM 360 ha interrupt imprecisi

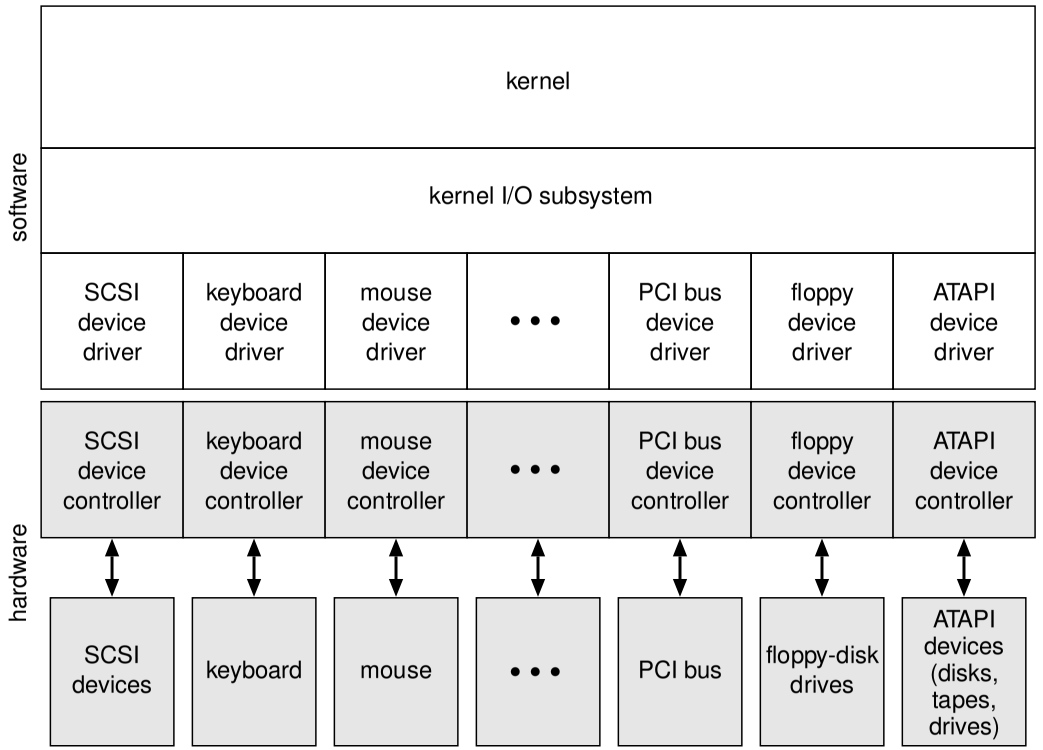
Evoluzione dell'I/O

1. Il processore controlla direttamente l'hardware del dispositivo
2. Si aggiunge un controller, che viene guidato dal processore con PIO
3. Il controller viene dotato di linee di interrupt; I/O interrupt driven
4. Il controller viene dotato di DMA
5. Il controller diventa un processore a sé stante, con un set dedicato di istruzioni. Il processore inizializza il PC del processore di I/O ad un indirizzo in memoria, e avvia la computazione. Il processore può così *programmare* le operazioni di I/O. Es: schede grafiche
6. Il controller ha una CPU e una propria memoria — è un calcolatore completo. Es: terminal controller, scheda grafica accelerata, ...

Tipicamente in un sistema di calcolo sono presenti più tipi di I/O.

Interfaccia di I/O per le applicazioni

- È necessario avere un trattamento uniforme dei dispositivi di I/O
- Le chiamate di sistema di I/O incapsulano il comportamento dei dispositivi in alcuni tipi generali
- Le effettive differenze tra i dispositivi sono contenute nei *driver*, moduli del kernel dedicati a controllare ogni diverso dispositivo.



Interfaccia di I/O per le applicazioni (cont.)

- Le chiamate di sistema raggruppano tutti i dispositivi in poche classi generali, uniformando i modi di accesso. Solitamente sono:
  - I/O a blocchi
  - I/O a carattere
  - accesso mappato in memoria
  - socket di rete
- Spesso è disponibile una syscall "scappatoia", dove si fa rientrare tutto ciò che non entra nei casi precedenti (es.: *ioctl* di UNIX)
- Esempio: i timer e orologi hardware esulano dalle categorie precedenti
  - Fornire tempo corrente, tempo trascorso
  - un timer *programmabile* si usa per temporizzazioni, timeout, interrupt
  - in UNIX, queste particolarità vengono gestite con la *ioctl*

Dispositivi a blocchi e a carattere

- I dispositivi a blocchi comprendono i dischi.
  - Comandi tipo *read*, *write*, *seek*
  - I/O attraverso il file system e cache, oppure direttamente al dispositivo (*crudo*) per applicazioni particolari
  - I file possono essere *mappati in memoria*: si fa coincidere una parte dello spazio indirizzi virtuale di un processo con il contenuto di un file
- I dispositivi a carattere comprendono la maggior parte dei dispositivi. Sono i dispositivi che generano o accettano uno stream di dati. Es: tastiera, mouse (per l'utente), ratti (per gli esperimenti), porte seriali, schede audio...
  - Comandi tipo *get, put* di singoli caratteri o parole. Non è possibile la *seek*
  - Spesso si stratificano delle librerie per filtrare l'accesso agli stream.

Dispositivi di rete

- Sono abbastanza diverse sia da device a carattere che a blocchi, per modo di accesso e velocità, da avere una interfaccia separata
- Unix e Windows/NT le gestiscono con le *socket*
  - Permettono la creazione di un collegamento tra due applicazioni separate da una rete
  - Le socket permettono di astrarre le operazioni di rete dai protocolli
  - Si aggiunge la syscall *select* per rimanere in attesa di traffico sulle socket
- solitamente sono supportati almeno i collegamenti *connection-oriented* e *connectionless*
- Le implementazioni variano parecchio (pipe half-duplex, code FIFO full-duplex, code di messaggi, mailboxes di messaggi,...)

I/O bloccante, non bloccante, asincrono

- Bloccante: il processo si sospende finché l'I/O non è completato
  - Semplice da usare e capire
  - Insufficiente, per certi aspetti ed utilizzi
- Non bloccante: la chiamata ritorna non appena possibile, anche se l'I/O non è ancora terminato
  - Esempio: interfaccia utente (attendere il movimento del mouse)
  - Facile da implementare in sistemi multi-thread con chiamate bloccanti
  - Ritorna rapidamente, con i dati che è riuscito a leggere/scrivere
- Asincrono: il processo continua mentre l'I/O viene eseguito
  - Difficile da usare (non si sa se l'I/O è avvenuto o no)
  - Il sistema di I/O segnala al processo quando l'I/O è terminato

Sottosistema di I/O del kernel

Deve fornire molte funzionalità

- Scheduling: in che ordine le system call devono essere esaudite
  - solitamente, il first-come, first-served non è molto efficiente
  - è necessario adottare qualche politica per ogni dispositivo, per aumentare l'efficienza
  - Qualche sistema operativo mira anche alla fairness
- Buffering: mantenere i dati in memoria mentre sono in transito, per gestire
  - differenti velocità (es. modem→disco)
  - differenti dimensioni dei blocchi di trasferimento (es. nastro→disco)

Sottosistema di I/O del kernel (cont.)

- Caching: mantenere una copia dei dati più usati in una memoria più veloce
  - Una cache è sempre una copia di dati esistenti altrove
  - È fondamentale per aumentare le performance
- Spooling: buffer per dispositivi che non supportano I/O interleaved (es. stampanti)
- Accesso esclusivo: alcuni dispositivi possono essere usati solo da un processo alla volta
  - System call per l'allocazione/deallocazione del dispositivo
  - Attenzione ai deadlock!

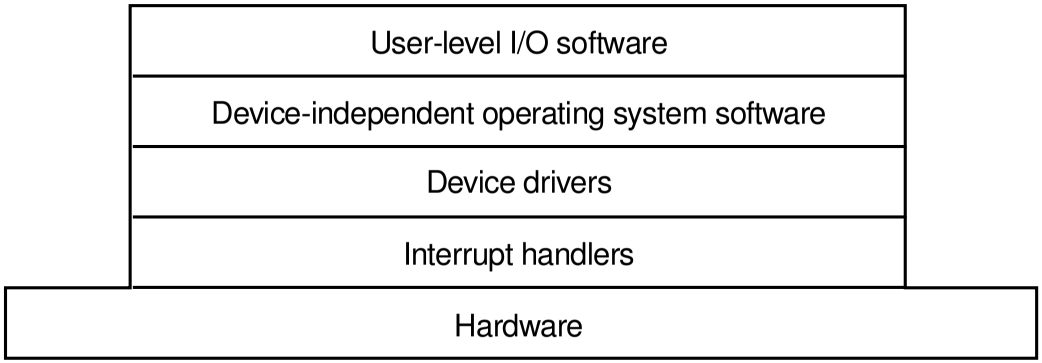
Sottosistema di I/O del kernel (cont.)

Gestione degli errori

- Un S.O. deve proteggersi dal malfunzionamento dei dispositivi
- Gli errori possono essere transitori (es: rete sovraccarica) o permanenti (disco rotto)
- Nel caso di situazioni transitorie, solitamente il S.O. può (tentare di) recuperare la situazione (es: richiede di nuovo l'operazione di I/O)
- Le chiamate di sistema segnalano un errore, quando non vanno a buon fine neanche dopo ripetuti tentativi
- Spesso i dispositivi di I/O sono in grado di fornire dettagliate spiegazioni di cosa è successo (es: controller SCSI).
- Il kernel può registrare queste diagnostiche in appositi *log di sistema*

I livelli del software di I/O

Per raggiungere gli obiettivi precedenti, si *stratifica* il software di I/O, con interfacce ben chiare (maggiore modularità)



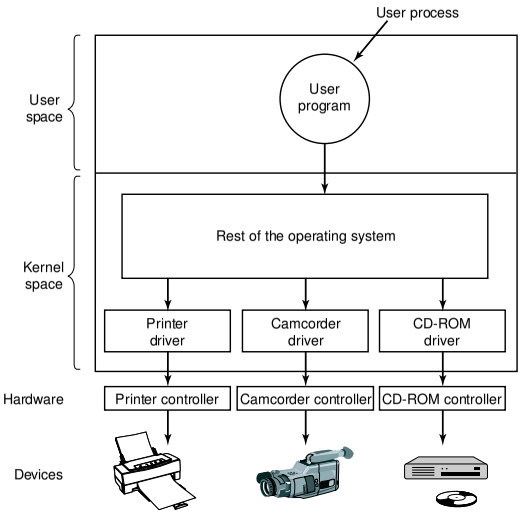
Driver delle interruzioni

- Fondamentale nei sistemi time-sharing e con I/O interrupt driven
- Passi principali da eseguire:
  1. Salvare i registri della CPU
  2. Impostare un contesto per la procedura di servizio (TLB, MMU, stack. . .)
  3. Ack al controllore degli interrupt (per avere interrupt annidati)
  4. Copiare la copia dei registri nel PCB
  5. Eseguire la procedura di servizio (che accede al dispositivo; proprio di ogni dispositivo)
  6. Eventualmente, cambiare lo stato a un processo in attesa (e chiamare lo scheduler di breve termine)
  7. Organizzare un contesto (MMU e TLB) per il processo successivo
  8. Caricare i registri del nuovo processo dal suo PCB
  9. Continuare il processo selezionato.

Driver dei dispositivi

Software (spesso di terze parti) che accede al controller dei device

- Hanno la vera conoscenza di come far funzionare il dispositivo
- Implementano le funziolità standardizzate, secondo poche classi (ad esempio: carattere/blocchi)
- Vengono eseguito in spazio kernel
- Per includere un driver, può essere necessario ricompilare o rilinkare il kernel.
- Attualmente si usa un meccanismo di caricamento run-time



Passi eseguiti dai driver dei dispositivi

1. Controllare i parametri passati
2. Accodare le richieste di una coda di operazioni (soggette a scheduling!)
3. Eseguire le operazioni, accedendo al controller
4. Passare il processo in modo *wait* (I/O interrupt-driven), o attendere la fine dell'operazione in busy-wait.
5. Controllare lo stato dell'operazione nel controller
6. Restituire il risultato.

I driver devono essere *rientranti*: a metà di una esecuzione, può essere lanciata una nuova esecuzione.

I driver non possono eseguire system call (sono sotto), ma possono accedere ad alcune funzionalità del kernel (es: allocazione memoria per buffer di I/O)

Nel caso di dispositivi "hot plug": gestire l'inserimento/disinserimento a caldo.

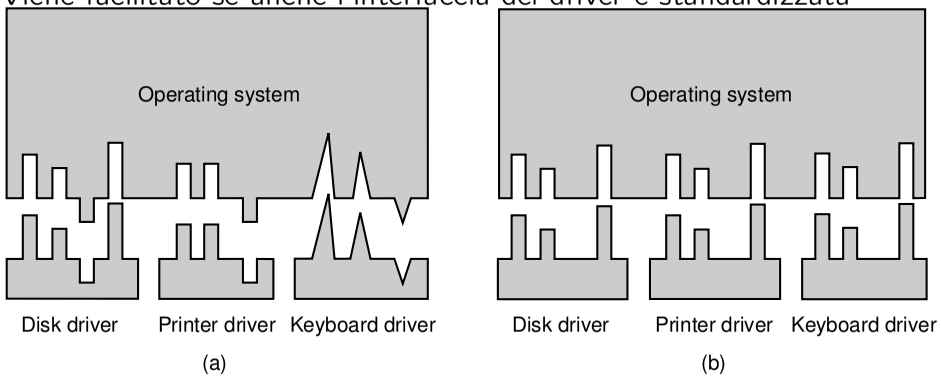
Software di I/O indipendente dai dispositivi

Implementa le funzionalità comuni a tutti i dispositivi (di una certa classe):

- fornire un'interfaccia uniforme per i driver ai livelli superiori (file system, software a livello utente)
- Bufferizzazione dell'I/O
- Segnalazione degli errori
- Allocazione e rilascio di dispositivi ad accesso dedicato
- Uniformizzazione della dimensione dei blocchi (blocco logico)

Interfacciamento uniforme

- Viene facilitato se anche l'interfaccia dei driver è standardizzata



- Gli scrittori dei driver hanno una specifica di cosa devono implementare
- Deve offrire anche un modo di *denominazione uniforme*, flessibile e generale
- Implementare un meccanismo di protezione per gli strati utente (strettamente legato al meccanismo di denominazione)

Esempio di interfaccia per i driver

In Linux un driver implementa (alcune delle) funzioni specificate dalla struttura `file_operations`

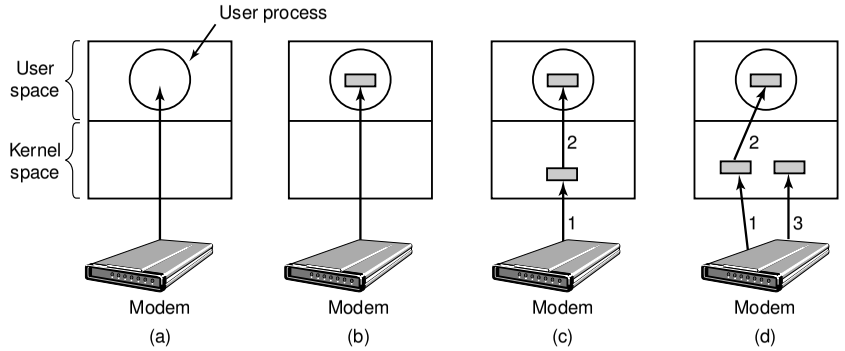
```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsign
};
```

Esempio: le operazioni di un terminale (una seriale)

```
static struct file_operations tty_fops = {
        llseek:         no_llseek,
        read:           tty_read,
        write:          tty_write,
        poll:           tty_poll,
        ioctl:          tty_ioctl,
        open:           tty_open,
        release:        tty_release,
        fasync:         tty_fasync,
};

static ssize_t tty_read(struct file * file, char * buf, size_t count,
                        loff_t *ppos)
{
...
        return i;
}
```

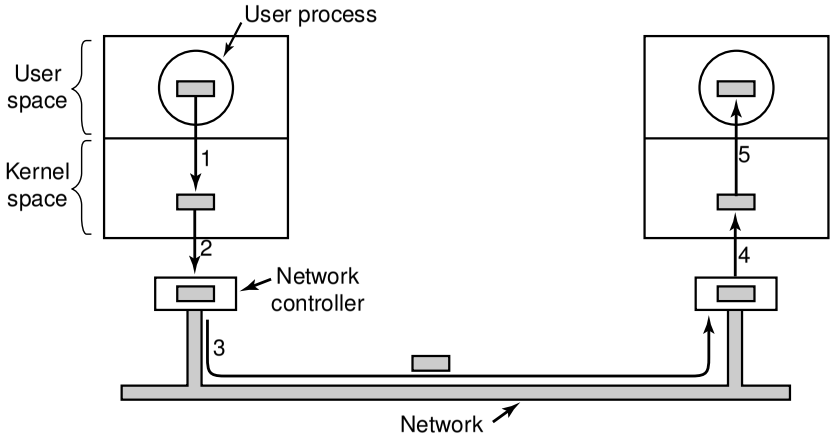
Bufferizzazione

- Non bufferizzato: inefficiente
- Bufferizzazione in spazio utente: problemi con la memoria virtuale
- Bufferizzazione in kernel: bisogna copiare i dati, con blocco dell'I/O nel frattempo.
- Doppia bufferizzazione



Bufferizzazione (cont.)

Permette di disaccoppiare la chiamata di sistema di scrittura con l'istante di effettiva uscita dei dati (output *asincrono*).



Eccessivo uso della bufferizzazione incide sulle prestazioni

Gestione degli errori

- Errori di programmazione: il programmatore chiede qualcosa di impossibile/inconsistente (scrivere su un CD-ROM, seekare una seriale, accedere ad un dispositivo non installato, etc.)

  Azione: abortire la chiamata, segnalando l'errore al chiamante.

- Errori del dispositivo. Dipende dal dispositivo
  - Se transitori: cercare di ripetere le operazioni fino a che l'errore viene superato (rete congestionata)
  - Abortire la chiamata: adatto per situazioni non interattive, o per errori non recuperabili. Importante la diagnostica.
  - Far intervenire l'utente/operatore: adatto per situazioni riparabili da intervento esterno (es.: manca la carta).

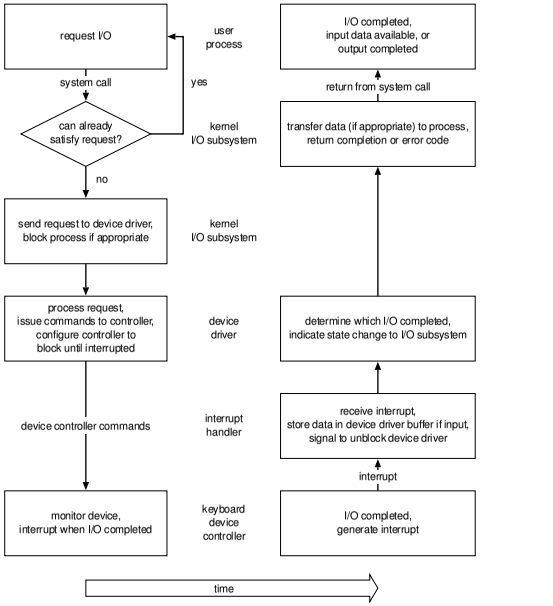
Software di I/O a livello utente

- Non gestisce direttamente l'I/O; si occupano soprattutto di formattazione, gestione degli errori, localizzazione...
- Dipendono spesso dal *linguaggio* di programmazione, e non dal sistema operativo
- Esempio: la `printf`, la `System.out.println`, etc.
- Realizzato anche da *processi di sistema*, come i demoni di spooling...

Traduzioni delle richieste di I/O in operazioni hardware

- Esempio: leggere dati da un file su disco
  - Determinare quale dispositivo contiene il file
  - Tradurre il nome del file nella rappresentazione del dispositivo
  - Leggere fisicamente i dati dal disco in un buffer
  - Rendere i dati disponibile per il processo
  - Ritornare il controllo al processo
- Parte di questa traduzione avviene nel file system, il resto nel sistema di I/O. Es.: Unix rappresenta i dispositivi con dei file "speciali" (in /dev) e coppie di numeri *(major,minor)*
- Alcuni sistemi (eg. Unix moderni) permettono anche la creazione di linee di dati customizzate tra il processo e i dispositivi hardware (*STREAMS*).

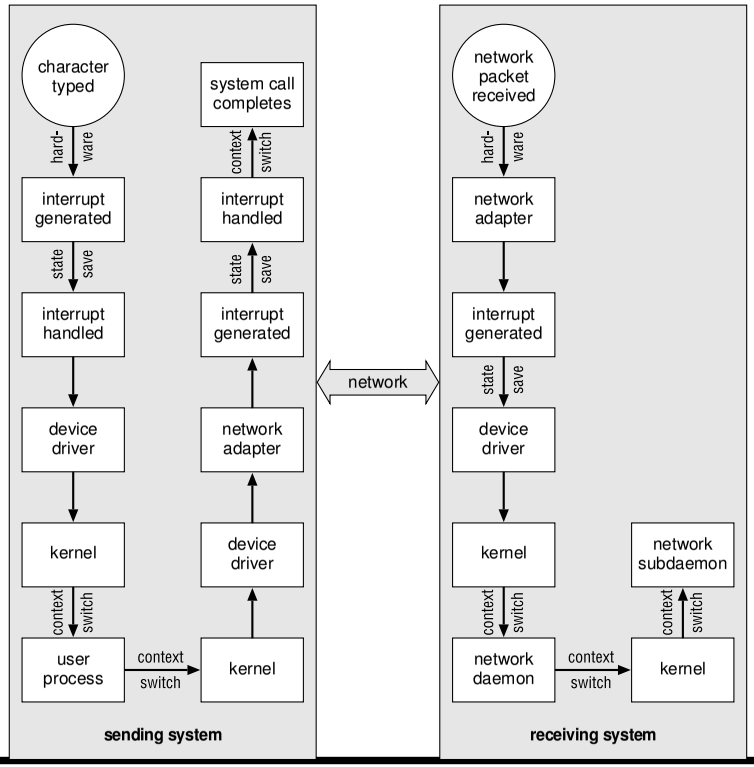
Esecuzione di una richiesta di I/O



Performance

L'I/O è un fattore predominante nelle performance di un sistema

- Consuma tempo di CPU per eseguire i driver e il codice kernel di I/O
- Continui cambi di contesto all'avvio dell'I/O e alla gestione degli interrupt
- Trasferimenti dati da/per i buffer consumano cicli di clock e spazio in memoria
- Il traffico di rete è particolarmente pesante (es.: telnet)

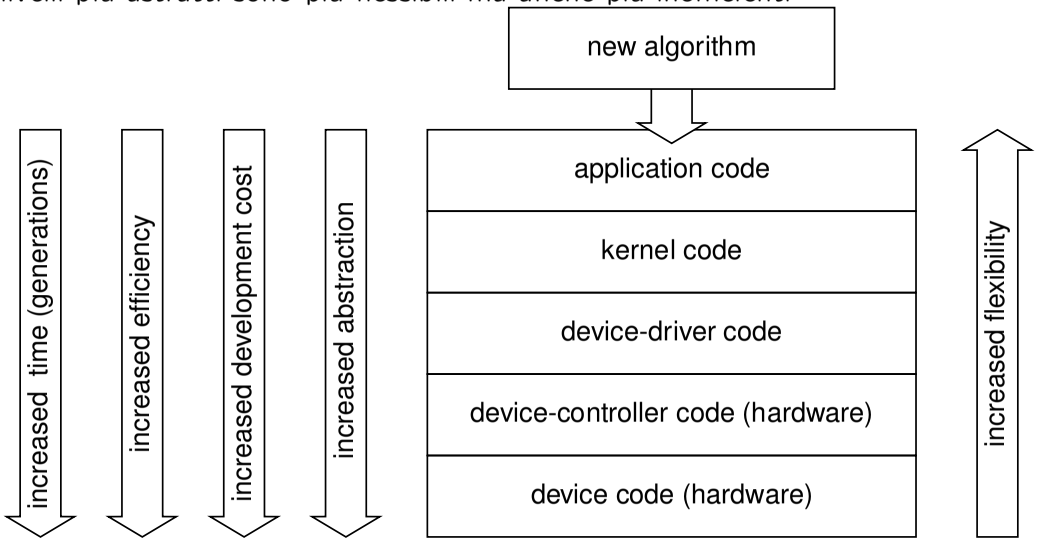


Migliorare le performance

- Ridurre il numero di context switch (es.: il telnet di Solaris è un thread di kernel)
- Ridurre spostamenti di dati tra dispositivi e memoria, e tra memoria e memoria
- Ridurre gli interrupt preferendo grossi trasferimenti, controller intelligenti, interrogazione ciclica (se i busy wait possono essere minimizzati)
- Usare canali di DMA, o bus dedicati
- Implementare le primitive in hardware, dove possibile, per aumentare il parallelismo
- Bilanciare le performance della CPU, memoria, bus e dispositivi di I/O: il sovraccarico di un elemento comporta l'inutilizzo degli altri

Livello di implementazione

A che livello devono essere implementate le funzionalità di I/O? In generale, i livelli più astratti sono più flessibili ma anche più inefficienti



- Inizialmente, gli algoritmi vengono implementati ad alto livello. Inefficiente ma sicuro.
- Quando l'algoritmo è testato e messo a punto, viene spostato al livello del kernel. Questo migliora le prestazioni ma è molto più delicato: un driver bacato può piantare tutto il sistema
- Per avere le massime performance, l'algoritmo può essere spostato nel firmware o microcodice del controller. Complesso, costoso.

Il sistema di I/O in Unix

- Il sistema di I/O nasconde le peculiarità dei dispositivi di I/O dal resto del kernel.
- Principali obiettivi:
  - uniformità dei dispositivi
  - denominazione uniforme
  - gestione dispositivi ad accesso dedicato
  - protezione
  - gestione errori
  - sincronizzazione
  - caching

Denominazione dei device: tipo, major e minor number

- UNIX riconosce tre grandi famiglie di dispositivi:
  - dispositivi a blocchi
  - dispositivi a carattere
  - interfacce di rete (*socket* e *door*) (v. gestione rete)
- All'interno di ogni famiglia, un numero intero (*major number*) identifica il tipo di device.
- Una specifica istanza di un dispositivo di un certo tipo viene identificato da un altro numero, il *minor number*.
- Quindi, tutti i dispositivi vengono identificati da una tripla $\langle tipo, major, minor \rangle$. Es: in Linux, primary slave IDE disk = $\langle c, 3, 64 \rangle$

I device sono file!

Tutti i dispositivi sono accessibili come file *speciali*

- nomi associati a inode che non allocano nessun blocco, ma contengono tipo, major e minor
- vi si accede usando le stesse chiamate di sistema dei file
- i controlli di accesso e protezione seguono le stesse regole dei file
- risiedono normalmente in `/dev`, ma non è obbligatorio

La directory /dev

```
$ ls -l /dev
total 17
-rwxr-xr-x   1 root     root        15693 Aug 13  1998 MAKEDEV*
crw-rw-r--   1 root     root       10,   3 May  5  1998 atibm
crw-rw-rw-   1 root     sys        14,   4 May  5  1998 audio
crw-rw--w-   1 root     sys        14,  20 May  5  1998 audio1
lrwxrwxrwx   1 root     root            3 Jun 19  1998 cdrom -> hdb
crw--w--w-   1 miculan  root        4,   0 Mar 19 11:29 console
crw-rw----   1 root     uucp        5,  64 Mar 19 08:29 cua0
crw-rw----   1 root     uucp        5,  65 May  5  1998 cua1
crw-rw----   1 root     uucp        5,  66 May  5  1998 cua2
crw-rw----   1 root     uucp        5,  67 May  5  1998 cua3
[...]
crw-rw-rw-   1 root     sys        14,   3 May  5  1998 dsp
crw-rw--w-   1 root     sys        14,  19 May  5  1998 dsp1
lrwxrwxrwx   1 root     root           15 Jun 19  1998 fd -> ../proc/self/fd/
brw-rw-r--   1 root     floppy      2,   0 May  5  1998 fd0
brw-rw-r--   1 root     floppy      2,  12 May  5  1998 fd0D360
brw-rw-r--   1 root     floppy      2,  16 May  5  1998 fd0D720
brw-rw-r--   1 root     floppy      2,  28 May  5  1998 fd0H1440
[...]
brw-rw----   1 root     disk        3,   0 May  5  1998 hda
brw-rw----   1 root     disk        3,   1 May  5  1998 hda1
brw-rw----   1 root     disk        3,   2 May  5  1998 hda2
brw-rw----   1 root     disk        3,   3 May  5  1998 hda3
brw-rw----   1 root     disk        3,   4 May  5  1998 hda4
[...]
brw-rw-rw-   1 root     disk        3,  64 May  5  1998 hdb
brw-rw----   1 root     disk        3,  65 May  5  1998 hdb1
brw-rw----   1 root     disk        3,  66 May  5  1998 hdb2
[...]
lrwxrwxrwx   1 root     root            9 Jun 19  1998 mouse -> /dev/cua0
[...]
```

La directory /dev (cont.)

```
crw-rw-rw-   1 root     root       1,   3 May  5  1998 null
[...]
crw-rw-rw-   1 root     tty        2, 176 May  5  1998 ptya0
crw-rw-rw-   1 root     tty        2, 177 May  5  1998 ptya1
crw-rw-rw-   1 root     tty        2, 178 May  5  1998 ptya2
crw-rw-rw-   1 root     tty        2, 179 May  5  1998 ptya3
[...]
brw-rw----   1 root     disk       8,   0 May  5  1998 sda
brw-rw----   1 root     disk       8,   1 May  5  1998 sda1
brw-rw----   1 root     disk       8,  10 May  5  1998 sda10
[...]
crw-rw-rw-   1 root     root       5,   0 Mar 19 12:07 tty
crw-------   1 root     root       4,   0 May  5  1998 tty0
crw-------   1 root     root       4,   1 Mar 19 08:29 tty1
crw-------   1 root     root       4,   2 Mar 18 17:52 tty2
[...]
crw-rw-rw-   1 root     root       1,   5 May  5  1998 zero
```

- Major, minor e nomi sono fissati dal vendor. Ci sono regole de facto (es: /dev/ttyN: vere linee seriali della macchina, ptyXY pseudoterminali, device `zero`, `null`, `random`).
- (Gli inode de)I file speciali vengono creati con il comando *mknod(1)*, eseguibile solo da root. Es:

  ```
  # mknod /tmp/miodisco b 3 0
  ```

  crea un device a blocchi, con major 3, minor 0 (equivalente a /dev/hda)

System call per I/O di Unix

Sono le stesse dei file, solo che sono usate su device. Le principali sono

- `open(2)`: aprire un file/dispositivo
- `read(2)`, `readdir(2)`, `write(2)`: leggi/scrivi da/su un dispositivo
- `close(2)`: chiudi (rilascia) un dispositivo
- `lseek(2)`: posiziona il puntatore di lettura/scrittura
- `ioctl(2)`: generica chiamata di controllo I/O

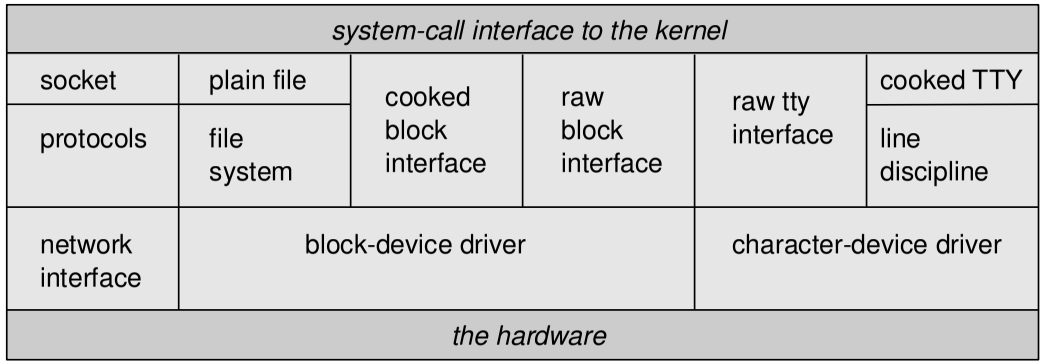
  ```
  int ioctl(int fd, int request, ...)
  ```

- `fsync(2)`: sincronizza lo stato su disco di un file con quello in memoria
- `mmap(2)`: mappa un file o device in memoria virtuale di un processo

```
void *mmap(void *start, size_t length,
           int prot, int flags,
           int fd, off_t offset)
```

I device a blocchi vengono utilizzati dal file system, ma possono essere usati anche direttamente.

Struttura di I/O in Unix

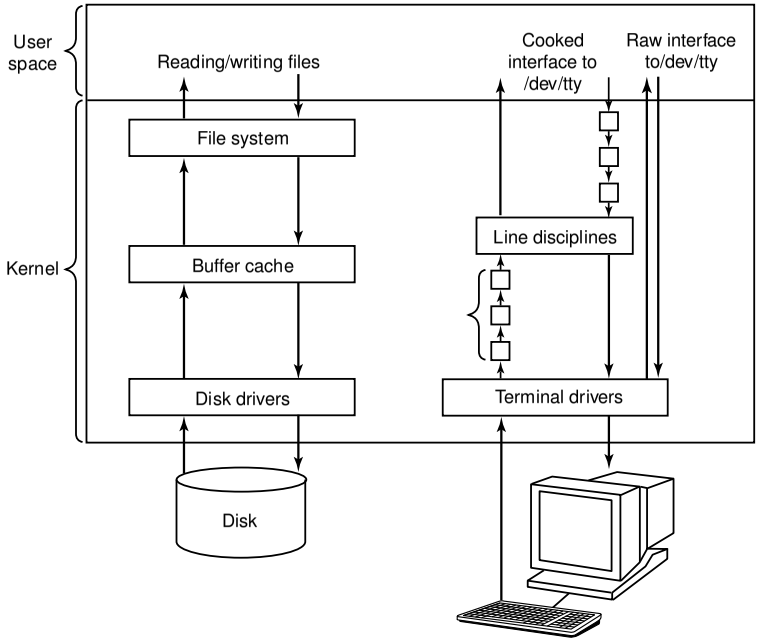


Il sistema di I/O consiste in

- parte indipendente dal device, che implementa anche cache per i dispositivi a blocchi, buffering per i dispositivi a carattere
- device driver specifici per ogni tipo di dispositivo

Uniformità: i driver hanno una interfaccia uniforme
Modularità: possono essere aggiunti/tolti dal kernel dinamicamente

Cache e buffering



Le Device Table e la struct file_operations

- Il kernel mantiene due *device table*, una per famiglia
- ogni entry corrisponde ad un major number e contiene l'indirizzo di una struttura `file_operations` come la seguente

```
struct file_operations miodevice_fops = {
  miodevice_seek,
  miodevice_read,     /* alcune di queste funzioni possono essere    */
  miodevice_write,    /* NULL, se non sono supportate o implementate */
  miodevice_readdir,
  miodevice_select,
  miodevice_ioctl,
  miodevice_mmap,
  miodevice_open,
  miodevice_flush,
  miodevice_release  /* a.k.a. close */
};
```


Le Device Table e la struct file_operations (cont.)

- queste strutture vengono create dai device driver e contengono le funzioni da chiamare quando un processo esegue una operazione su un device
- lo strato device-independent entra nella device table con il major per risalire alla funzione da chiamare realmente
- la struttura file_operations del device viene iscritta nell'appropriata *device table*. in corrispondenza al major number (**registrazione del device**). Avviene al momento del caricamento del driver (boot o installazione del modulo)

Dispositivi a blocchi: Cache

- Consiste in records (*header*) ognuno dei quale contiene un puntatore ad un'area di memoria fisica, un device number e un block number
- Gli header di blocchi non in uso sono tenuti in liste:
  - Buffer usati recentemente, in ordine LRU
  - Buffer non usati recentemente, o senza un contenuto valido (*AGE* list)
  - Buffer vuoti (*empty*), non associati a memoria fisica
- Quando un blocco viene richiesto ad un device, si cerca nella cache
- Se il blocco viene trovato, viene usato senza incorrere in I/O
- Se non viene trovato, si sceglie un buffer dalla AGE list, o dalla lista LRU se la AGE è vuota.

Dispositivi a blocchi: Cache (cont.)

- La dimensione della cache influenza le performance: se è suff. grande, il cache hit rate può essere alto e il numero di I/O effettivi basso.
- Solitamente, la cache viene estesa dinamicamente ad occupare tutta la memoria fisica lasciata dal memory management
- I dati scritti su un file sono bufferizzati in cache, e il driver del disco riordina la coda di output secondo le posizioni — questo permette di minimizzare i seek delle testine e di salvare i dati nel momento più opportuno.

Code di operazioni pendenti e dispositivi "crudi"

- Ogni driver tiene una coda di operazioni pendenti
- Ogni record di questa coda specifica
  - se è una operazione di lettura o scrittura
  - un indirizzo in memoria e uno nel dispositivo per il trasferimento
  - la dimensione del trasferimento
- Normalmente, la traduzione da block buffer a operazione di I/O è realizzata nello strato device-independent (accesso *cooked*)
- È possibile accedere direttamente alla coda delle operazioni di I/O, attraverso le interfacce *crude (raw)*.

Code di operazioni pendenti e dispositivi "crudi" (cont.)

- Le interfacce raw sono dispositivi a carattere, usati per impartire direttamente le operazioni di I/O al driver Es.: Solaris

```
$ ls -lL /dev/dsk/c0t0d0s0 /dev/rdsk/c0t0d0s0
brw-r----- 1 root  sys  32, 0 May  5  1998 /dev/dsk/c0t0d0s0
crw-r----- 1 root  sys  32, 0 May  5  1998 /dev/rdsk/c0t0d0s0
$
```

- Le interfacce crude non usano la cache
- Utili per operazioni particolari (formattazione, partizionamento), o per implementare proprie politiche di caching o proprî file system (es. database).

Dispositivi a caratteri: C-Lists

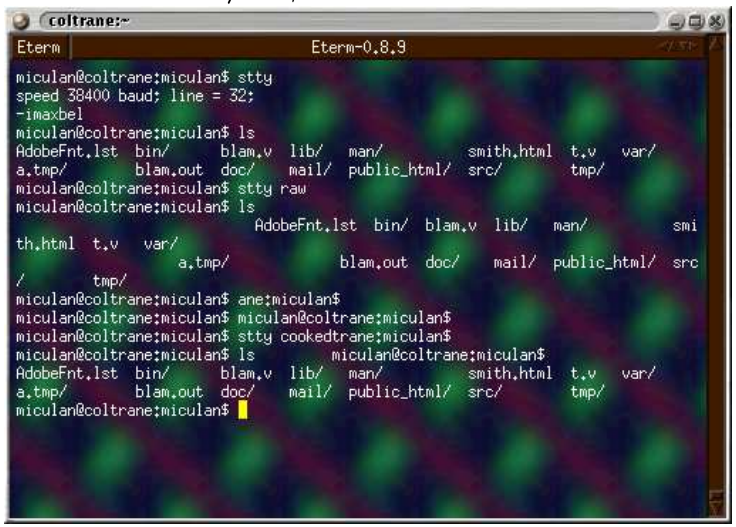
- *C-list:* sistema di buffering che mantiene piccoli blocchi di caratteri di dimensione variabile in liste linkate
- Implementato nello strato device independent dei dispositivi a carattere
- Trasferimento in output:
  1. Una *write* a un terminale accoda il blocco di caratteri da scrivere nella C-list di output per il device
  2. Il driver inizia il trasferimento del primo blocco sulla lista, o con DMA o con PIO sul chip di I/O.
  3. A trasferimento avvenuto, il chip di I/O solleva un interrupt; la routine di gestione verifica lo stato, e se ha avuto successo il blocco viene tolto dalla C-list
  4. salta a 2.

Dispositivi a caratteri: C-Lists (cont.)

- L'input avviene analogamente, guidato dagli interrupt.
- Accesso *cooked* (POSIX: *canonical*): normalmente, i caratteri in entrata e in uscita sono filtrati dalla *disciplina di linea*. Es: ripulitura dei caratteri di controllo in input, traduzione dei carateri di controllo in output.
- Accesso *crudo* (POSIX: *noncanonical*): permette di bypassare la C-list e la disciplina di linea. Ogni carattere viene passato/prelevato direttamente al driver senza elaborazione. Usato da programmi che devono reagire ad ogni tasto (eg., vi, emacs, server X, videogiochi)

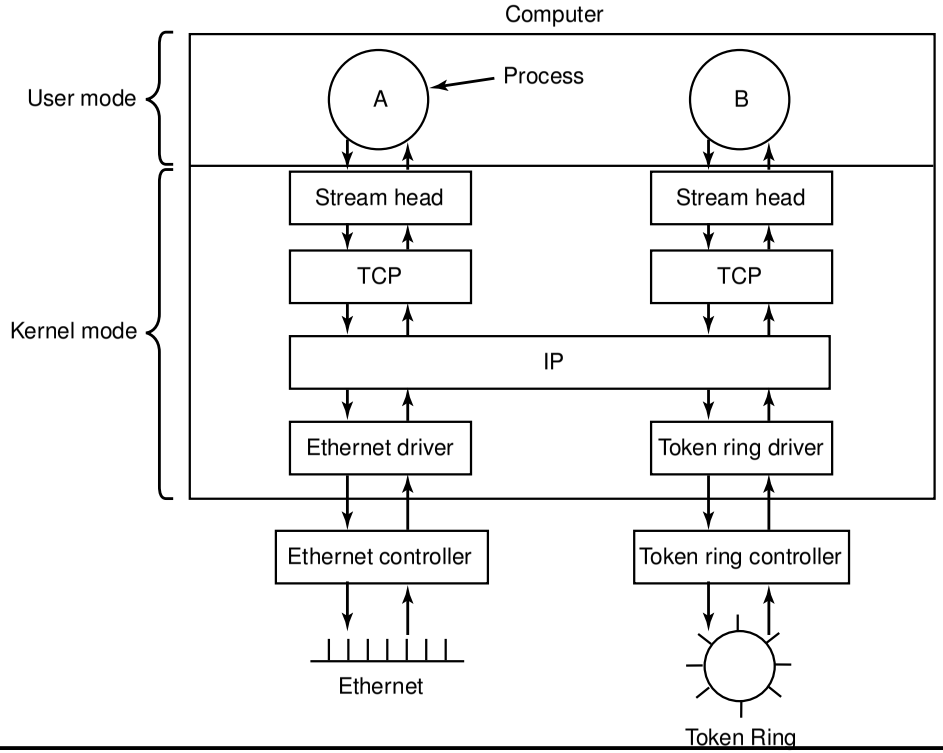
Controllo di linea: stty(2)

stty permette di impostare la disciplina di linea, velocità di trasferimento del dispositivo, modalità cooked/raw, ...



STREAM

- Generalizzazione della disciplina di linea; usato da System V e derivati
- Analogo delle pipe a livello utente (ma bidirezionali):
  - il kernel dispone di molti moduli per il trattamento di stream di byte, caricabili anche dinamicamente. Anche i driver sono moduli STREAM.
  - Questi possono essere collegati (impilati) dinamicamente con delle `ioctl` (es: `ioctl(fd, I_PUSH, "kb")`)
  - Ogni modulo mantiene una coda di lettura e scrittura
  - Quando un processo scrive sullo stream
    - il codice di testa interpreta la chiamata di sistema
    - i dati vengono messi in un buffer, che viene passato sulla coda di input del primo modulo
    - il primo modulo produce un output che viene messo sulla coda di input del secondo, etc. fino ad arrivare al driver



Monitoraggio I/O: iostat(1M)

```
miculan@maxi:miculan$ iostat 5
      tty           fd0           sd0           sd1           sd21          cpu
 tin tout kps tps serv  kps tps serv  kps tps serv  kps tps serv  us sy wt id
   1  182   0   0    0    6   1   87    5   1   45    0   0    0   3  1  0 95
   0   47   0   0    0   74  10  211    8   1   33    0   0    0   0  1  9 90
   0   16   0   0    0    0   0    0    0   0    0    0   0    0   2  2  0 96
   0   16   0   0    0    0   0    0    0   0    0    0   0    0   3  2  0 95
C
miculan@maxi:miculan$ iostat -xtc 5
                    extended device statistics       tty          cpu
device    r/s   w/s   kr/s   kw/s wait actv  svc_t  %w  %b  tin tout us sy wt id
fd0       0.0   0.0    0.0    0.0  0.0  0.0    0.0   0   0    1  182  3  1  0 95
sd0       0.1   0.5    1.2    4.8  0.0  0.1   86.7   0   1
sd1       0.5   0.2    3.7    1.3  0.0  0.0   45.1   0   0
sd21      0.0   0.0    0.0    0.0  0.0  0.0    0.0   0   0
nfs4      0.0   0.0    0.0    0.0  0.0  0.0    3.8   0   0
nfs47     0.0   0.0    0.0    0.0  0.0  0.0   29.3   0   0
nfs57     0.0   0.0    0.0    0.0  0.0  0.0   25.8   0   0
nfs69     0.0   0.0    0.0    0.0  0.0  0.0   20.1   0   0
nfs97     0.0   0.0    0.0    0.1  0.0  0.0   24.1   0   0
nfs171    0.0   0.0    0.0    0.0  0.0  0.0   19.4   0   0
nfs205    0.0   0.0    0.0    0.0  0.0  0.0   20.1   0   0
```

Sistema di I/O di Windows 2000

- Pensato per essere molto flessibile e modulare (Oltre 100 API distinte!)
- Prevede la riconfigurazione dinamica: i vari bus vengono scanditi al boot (SCSI) o al runtime (USB, IEEE1394).
- Gestisce anche l'alimentazione e la gestione dell'energia
- Tutti i file system sono driver di I/O.
- Consente l'I/O *asincrono*: un thread inizia un I/O e fissa un comportamento per la terminazione asincrona:
  - attendere su un oggetto evento per la terminazione
  - specificare una coda in cui verrà inserito un evento al completamento
  - specificare una funzione da eseguirsi al completamento (callback)

Implementazione dell'I/O in Windows 2000

- Insieme di procedure comuni, indipendenti dal dispositivo, più un insieme di driver caricabili dinamicamente.
- *Microsoft Driver Model*: definizione dell'interfaccia e di altre caratteristiche dei driver:
  1. Gestire le richieste di I/O in un formato standard (*I/O Request Packet*)
  2. Basarsi su oggetti
  3. Implementare il plug-and-play dinamico
  4. Implementare la gestione dell'alimentazione
  5. Configurabilità dell'utilizzo delle risorse (es. interrupt, porte di I/O,...)
  6. Rientranti (per supporto per elaborazione SMP)
  7. Portabili su Windows 98.

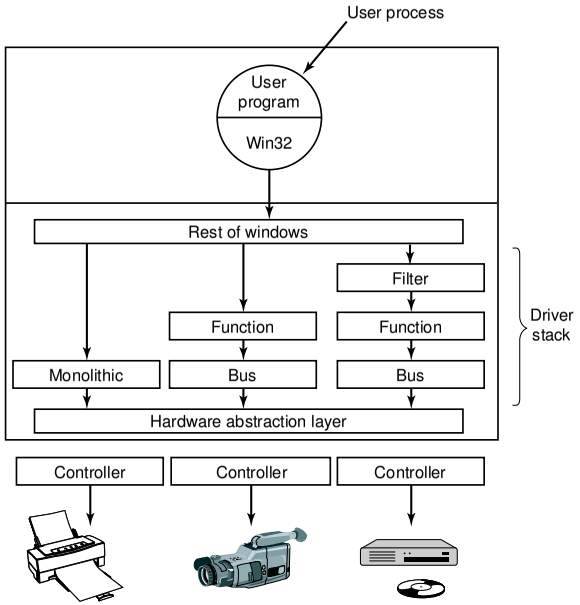
Installazione dei driver e dispositivi

1. All'avvio, o all'inserimento di un dispositivo, si esegue il PnP Manager.
2. Il PnP Manager chiede al dispositivo di identificarsi (produttore/modello)
3. Il PnP Manager controlla se il driver corrispondente è stato caricato
4. Se non lo è, controlla se è presente su disco
5. Se non lo è, chiede all'utente di cercarlo/caricarlo da supporto esterno.
6. Il driver viene caricato e la sua funzione *DriverEntry* viene eseguita:
   - inizializzazione di strutture dati interne
   - inizializzazione dell'oggetto *driver* creato da PnP Manager (che contiene puntatori a tutte le procedure fornite dal driver)
   - creazione degli oggetti *device* per ogni dispositivo gestito dal driver (attraverso la funzione *AddDevice*; richiamata anche da PnP Manager)
7. Gli oggetti device così creati vengono messi nella directory `\device` e appaiono come file (anche ai fini della protezione).

Stack di driver in Windows 2000

I driver possono essere autonomi, oppure possono essere *impilati*.

- simile agli STREAM di Unix
- ogni driver riceve un I/O Request Packet e passa un IRP al driver successivo
- ci sono anche driver "filtro"



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 alla ricerca dei *magic numbers*: utility `file`

```
$ file iptables.sh risultati Lucidi
iptables.sh: Bourne shell script text executable
risultati:   ASCII text
Lucidi:      PDF document, version 1.2
```


Tipi dei file — MacOS tradizionale: molti metadati

Nel MacOS tradizionale ogni file è composto da 3 componenti:

- *data fork*: sequenza non strutturata, simile a quelli Unix o DOS
- *resource fork*: strutturato, contiene codice, icone, immagini, etichette, dialoghi, ...
- *info*: metadati sul file stesso, tra cui *applicativo creatore* e *tipo*

L'operazione di apertura (*doppio click*) esegue l'applicativo indicato nelle info.

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, ...), eseguibili rilocabili (ELF, COFF)
  - 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.
- Operazioni:

  read next
  write next
  reset
  no *read* dopo l'ultimo *write*
  (*rewrite*)

- Adatto a dispositivi intrinsecamente sequenziali (p.e., nastri)

Metodi di accesso: accesso diretto

- Il puntatore può essere spostato in qualunque punto del file
- Operazioni:

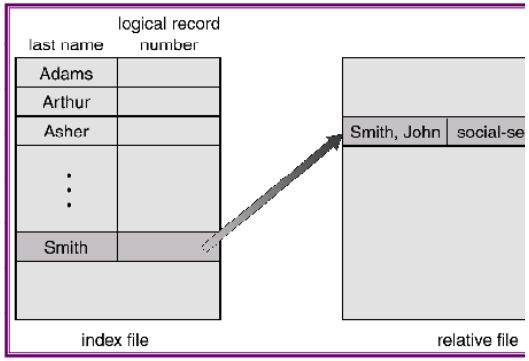
  read n
  write n
  seek n
  read next
  write next
  rewrite n

  n = posizione relativa a quella attuale

- 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

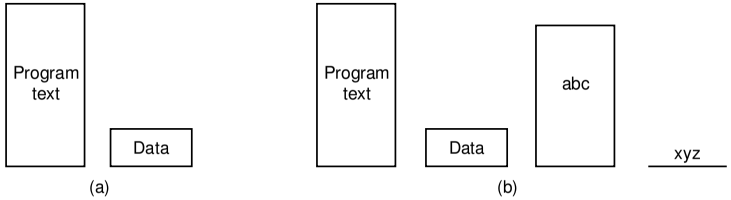


- Implementabile a livello applicazione in termini di file ad accesso diretto
- Usuale su mainframe (IBM, VMS), databases...


```
/* 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

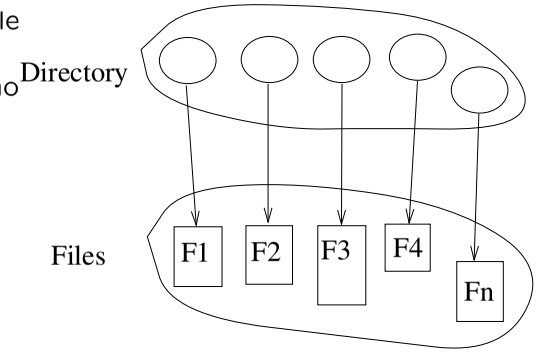
- Semplificano l'accesso ai file, rendendoli simili alla gestione della memoria.



- 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



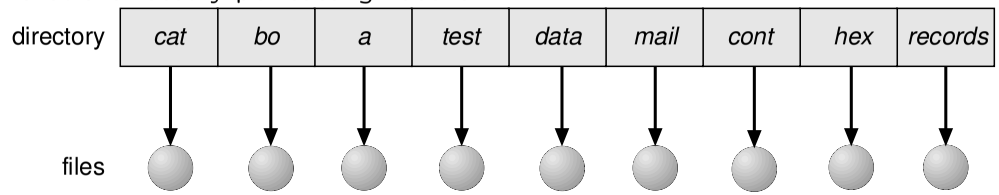
Organizzazione logica delle directory

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, . . .)

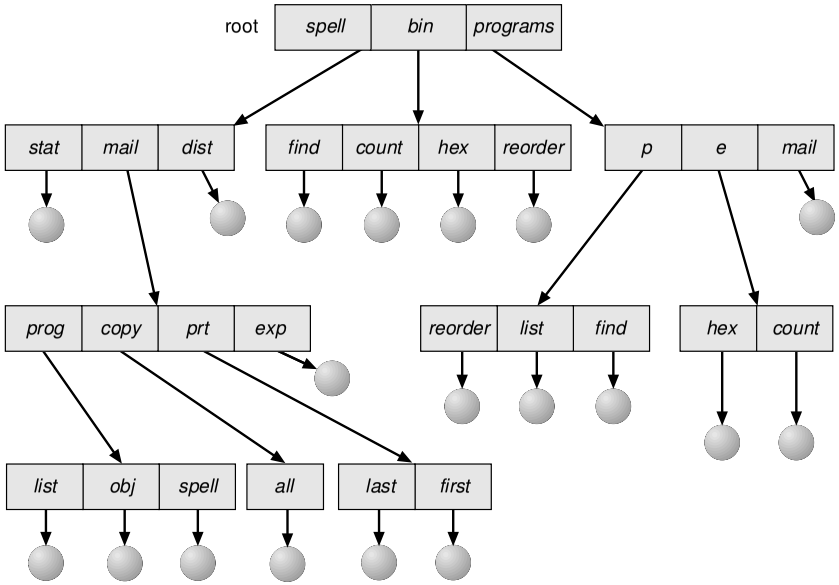
Tipi di directory: unica ("flat")

- 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



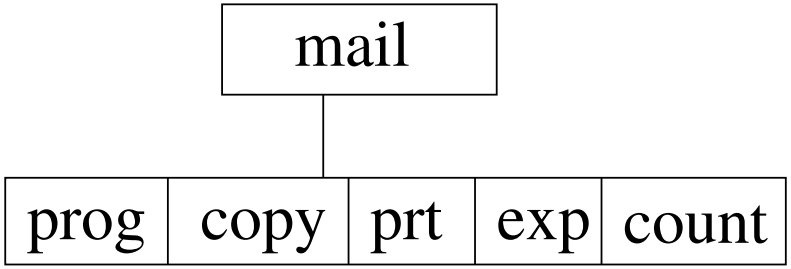
Directory ad albero (cont.)

- 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

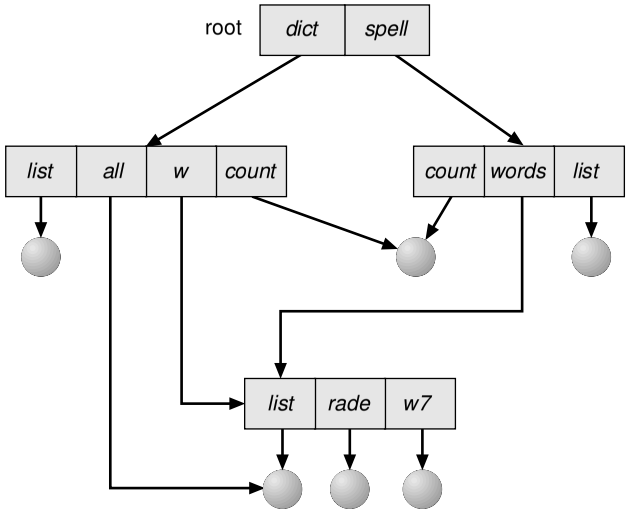


- Cancellando `mail` si cancella l'intero sottoalbero

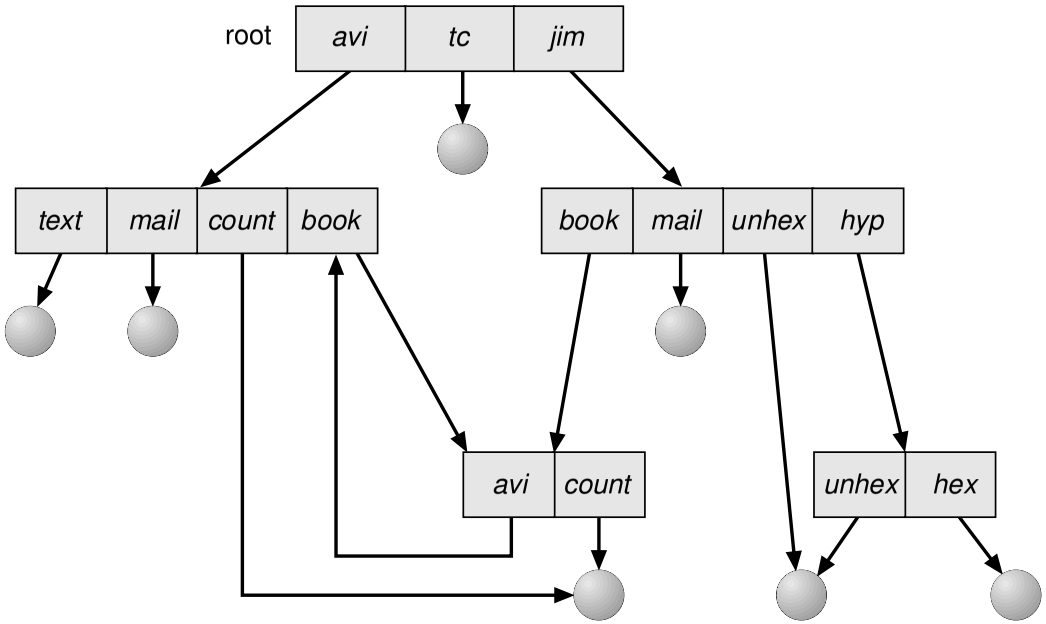
Directory a grafo aciclico (DAG)

File e sottodirectory possono essere condivise da più directory
Due nomi differenti per lo stesso file (aliasing)
Possibilità di puntatori "dangling". Soluzioni

- Puntatori all'indietro, per cancellare tutti i puntatori. Problematici perché la dimensione dei record nelle directory è variabile.
- Puntatori a daisy chain
- Contatori di puntatori 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. Algoritmi 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.

    Sono implementate da alcuni UNIX (e.g., *getfacl(1)* e *setfacl(1)* su Solaris)
  - *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 ACL.

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

				RWX
a)	owner access	7	⇒	1 1 1
b)	groups access	6	⇒	1 1 0
c)	public access	1	⇒	0 0 1

- Ogni processo possiede UID e 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

```
miculan@coltrane:Lucidi$ ls -l /usr/bin/lpr
-r-sr-sr-x   1 root     lp          15608 Oct 23 07:51 /usr/bin/lpr*
miculan@coltrane:Lucidi$
```

- 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

Setuid/setgid bit (cont.)

- Si impostano con il chmod

```
miculan@coltrane:C$ ls -l a.out
-rwxr-xr-x   1 miculan  ricerca     12045 Feb 28 12:11 a.out*
miculan@coltrane:C$ chmod 2755 a.out
miculan@coltrane:C$ ls -l a.out
-rwxr-sr-x   1 miculan  ricerca     12045 Feb 28 12:11 a.out*
miculan@coltrane:C$ chmod 4755 a.out
miculan@coltrane:C$ ls -l a.out
-rwsr-xr-x   1 miculan  ricerca     12045 Feb 28 12:11 a.out*
miculan@coltrane:C$
```


Implementazione del File System

I dispositivi tipici per realizzare file system: *dischi*

- trasferimento *a blocchi* (tip. 512 byte, ma variabile)
- accesso *diretto* a tutta la superficie, sia in lettura che in scrittura
- dimensione *finita*

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.

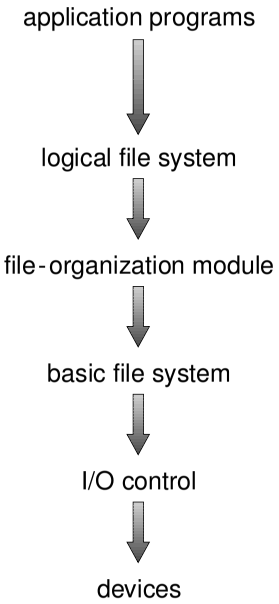


Tabella dei file aperti

- 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, sulla scrivania sotto MacOS)
  - 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

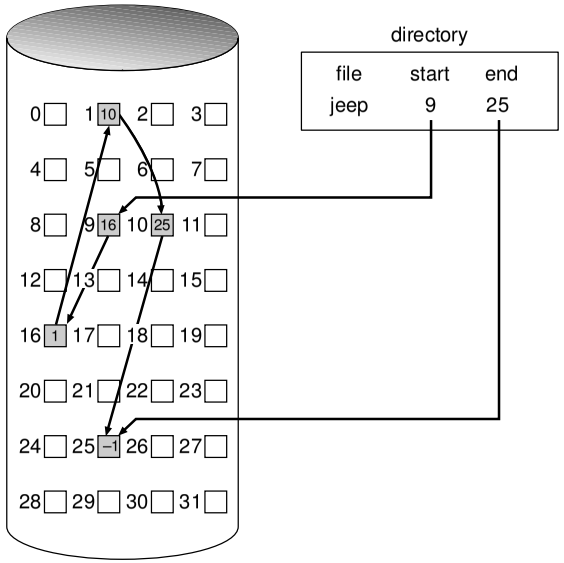
- Traduzione dall'indirizzo logico a quello fisico (per blocchi da 512 byte):

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

  - 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



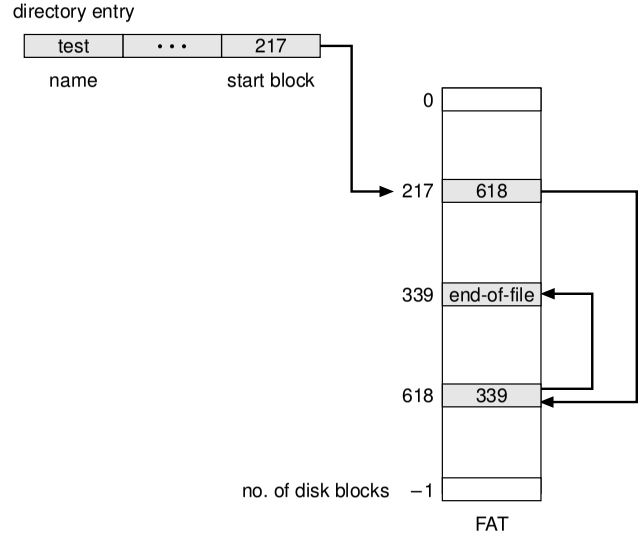
- Allocazione su richiesta; i blocchi vengono semplicemente collegati alla fine del file
- Semplice: basta sapere l'indirizzo del primo blocco
- Non c'è frammentazione esterna
- Bisogna gestire i blocchi liberi
- Non supporta l'accesso diretto (seek)
- Traduzione indirizzo logico:

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

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

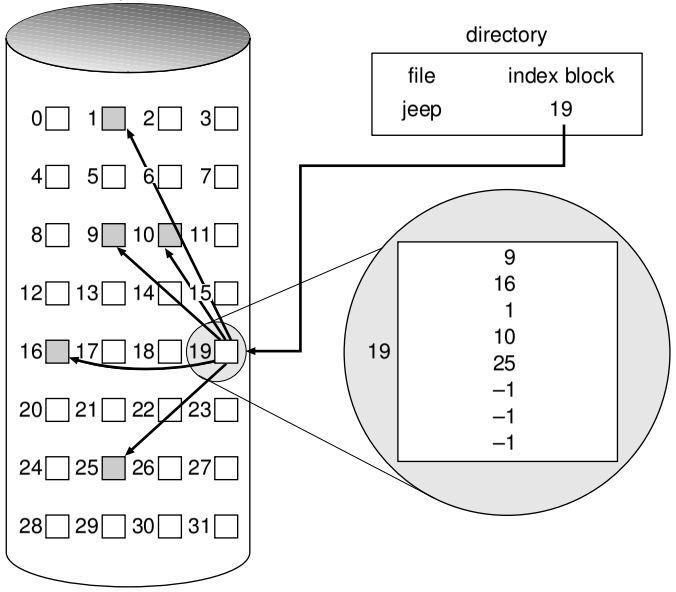
Allocazione concatenata (cont.)

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

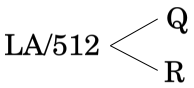


Allocazione indicizzata

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



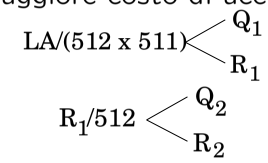
- Supporta accesso random
- Allocazione dinamica senza frammentazione esterna
- Traduzione: file di max 256K word e blocchi di 512 word: serve 1 blocco per l'indice



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

Allocazione indicizzata (cont.)

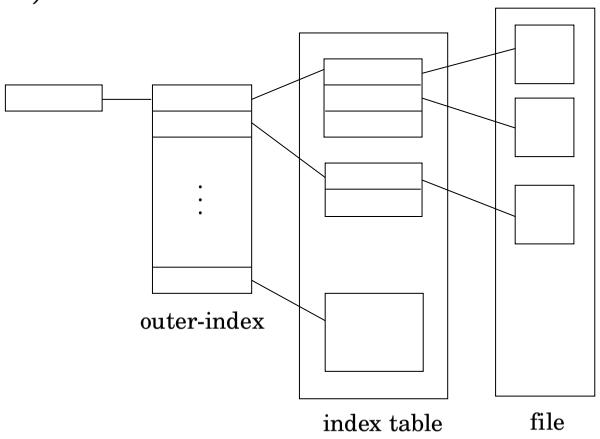
- Problema: come implementare il blocco indice
  - è una struttura supplementare: overhead ⇒ meglio piccolo
  - dobbiamo supportare anche file di grandi dimensioni ⇒ meglio grande
- Indice concatenato: l'indice è composto da blocchi concatenati. Nessun limite sulla lunghezza, maggiore costo di accesso.



  - 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

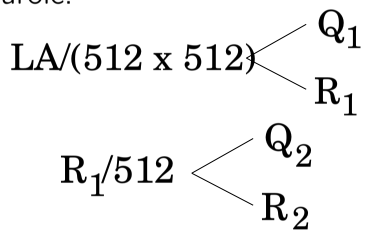
Allocazione indicizzata (cont.)

Indice a due (o più) livelli.



Dim. massima: con blocchi da 4K, si arriva a $(4096/4)^2 = 2^{20}$ blocchi = $2^{32}B = 4GB$.

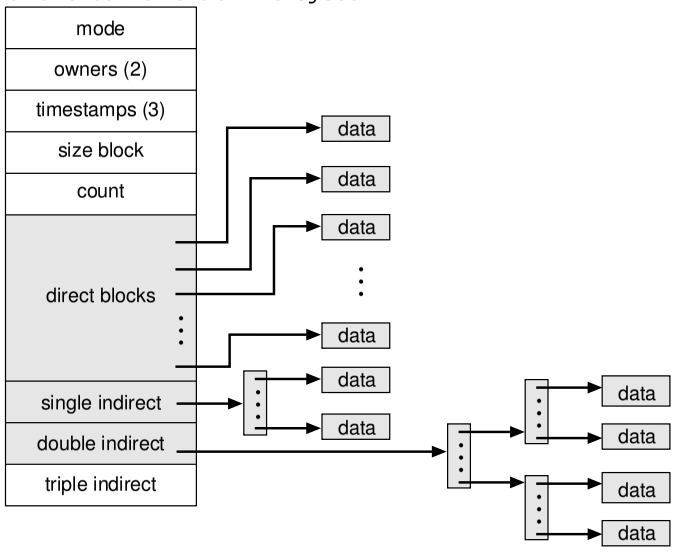
- Con blocchi da 512 parole:



  - 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 sono allocati in numero finito alla creazione del file system



Ogni inode 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 (*atime*), di ultima modifica (*mtime*), di ultimo cambiamento dell'inode (*ctime*)

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

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

  molto oltre le capacità dei sistemi a 32 bit.

Gestione dello spazio libero

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

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

  0101110101010111111011000000101000000010110101010111100

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

- Comodo per operazioni assembler di manipolazione dei bit
- Calcolo del numero del blocco

  (numero di bit per parola) $*$
  (numero di parole di valore 0) $+$
  offset del primo bit a 1

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

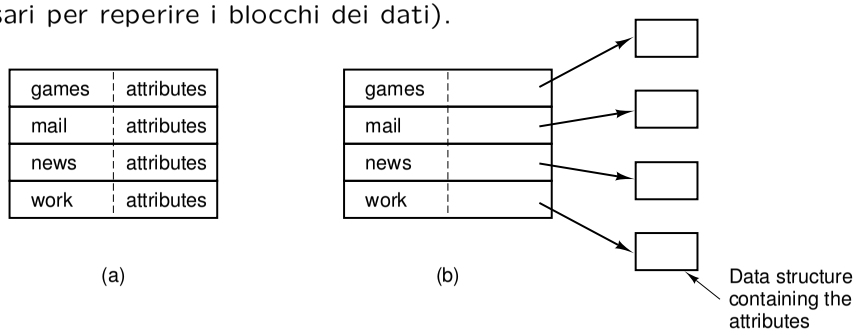
- Alternativa: *Linked list (free list)*

  – Inefficiente - non facile trovare blocchi liberi contigui

  – Non c'è spreco di spazio.

Implementazione delle directory

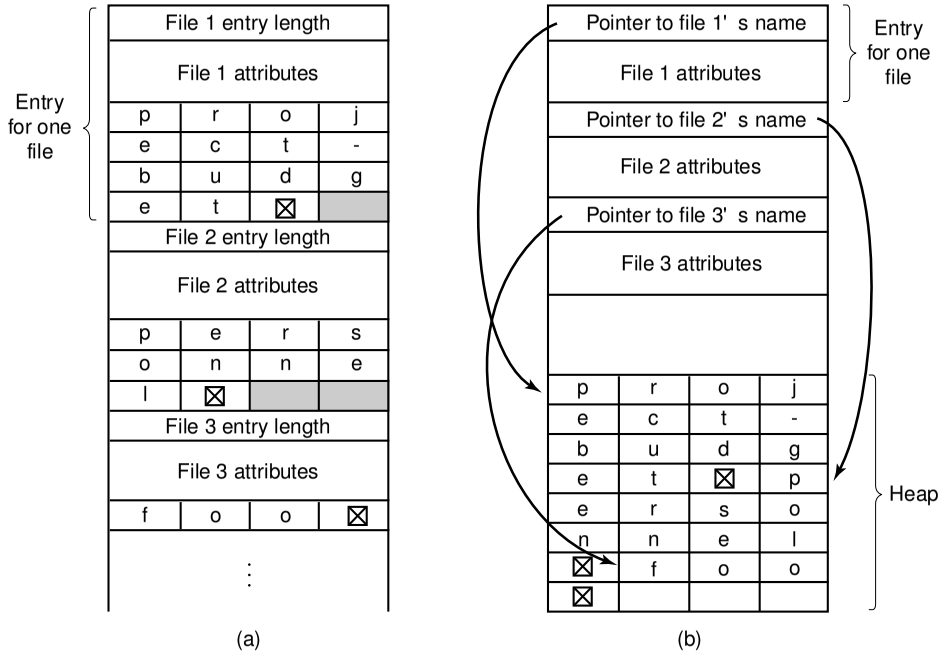
Le directory sono essenziali per passare dal nome del file ai suoi attributi (anche necessari per reperire i blocchi dei dati).



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

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

Directory: a dimensioni fisse, o a 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

Efficienza e performance

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!

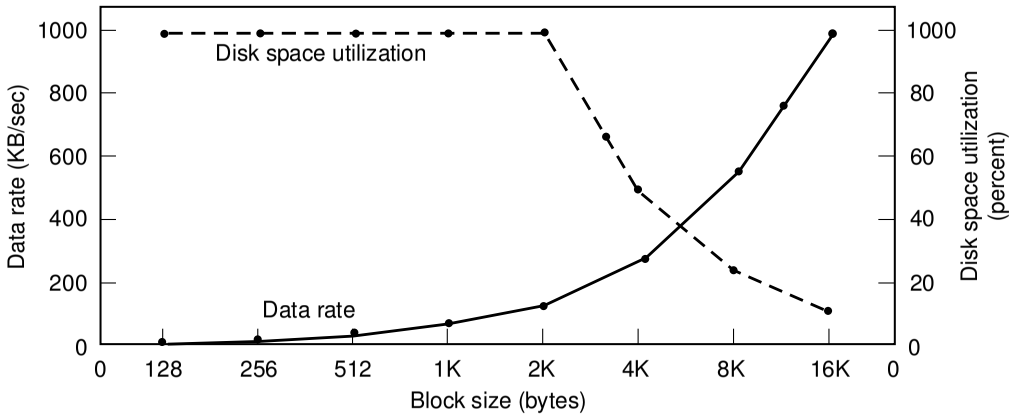
Sulla dimensione dei blocchi

Consideriamo questo caso: un disco con

- 131072 byte per traccia
- tempo di rotazione = 8.33 msec
- seek time = 10 msec

Il tempo per leggere un blocco di k byte è allora

$$10 + 4.165 + (k/131072) * 8.33$$



La mediana della lunghezza di un file Unix è circa 2KB.

Tipiche misure: 1K-4K (Linux, Unix); sotto Windows il cluster size spesso è imposto dalla FAT (anche se l'accesso ai file è assai più complicato).

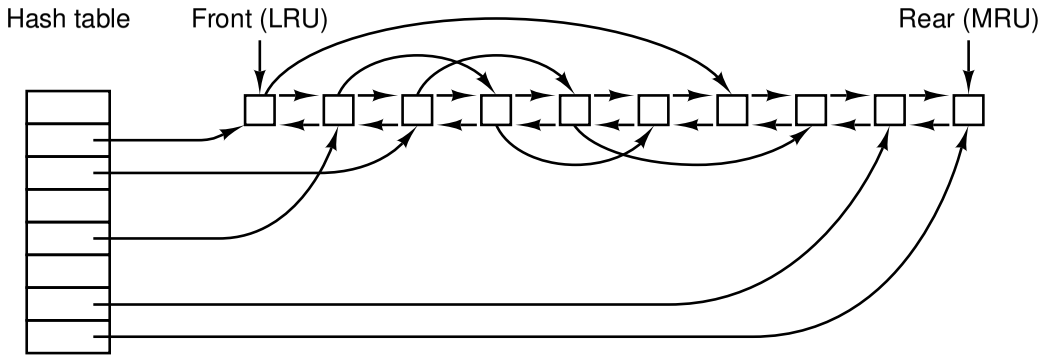
UFS e derivati ammettono anche il *fragment* (tipicamente 1/4 del block size).

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 la latenza a 0 (quasi)
- (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 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.

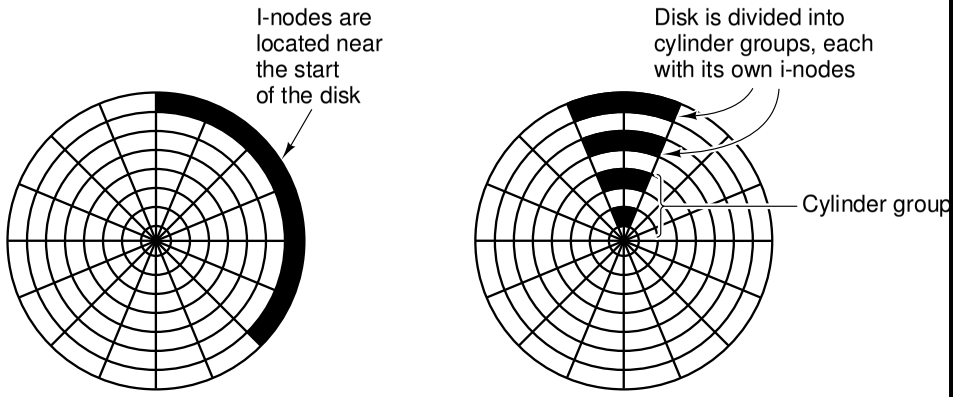
Anche le modifiche ai blocchi dati vengono trasferite prima della deallocazione:

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

Altri 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



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.

- Consistenza del file system: in seguito ad un crash, blocchi critici possono contenere informazioni incoerenti, sbagliate e contradditorie.
- Programmi di controllo della consistenza (*scandisk*, *fsck*): usano la ridondanza dei metadati, cercando di risolvere le inconsistenze. Lenti, e non sempre funzionano.

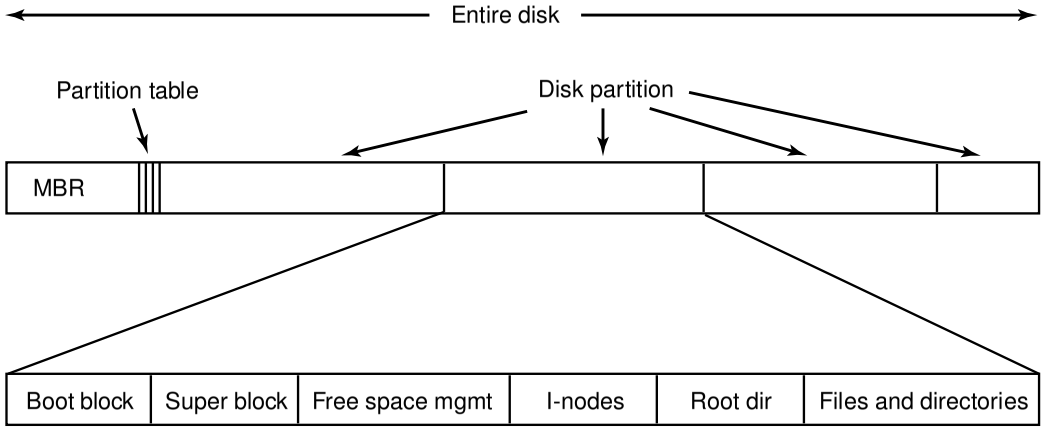
Journalled File System

Nei file system *journalled* (o *journaling*) si usano strutture e tecniche da DBMS (B+tree e "transazioni") per aumentare affidabilità e velocità

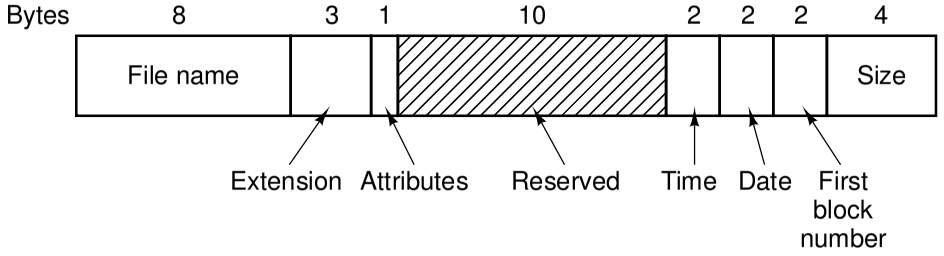
- Variazioni dei metadati (inodes, directories, bitmap, . . .) sono scritti immediatamente in un'area a parte, il *log* o *giornale*, prima di essere effettuate.
- Dopo un crash, per ripristinare la consistenza dei metadati è sufficiente ripercorrere il log $\Rightarrow$ non serve il *fsck*!
- Adatti a situazioni mission-critical (alta affidabilità, minimi tempi di recovery) e grandi quantità di dati.
- Esempi di file system journalled:

  XFS (SGI, su IRIX e Linux): fino a $2^{64} = 16$ exabytes > 16 milioni TB
  JFS (IBM, su AIX e Linux): fino a $2^{55} = 32$ petabyte > 32 mila TB
  ReiserFS e EXT3 (su Linux): fino a 16TB e 4TB, rispettivamente

Esempio di layout di un disco fisico



Directory in MS-DOS



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

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

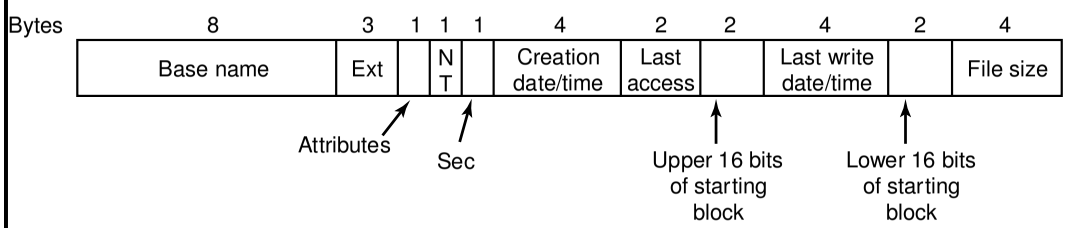
In MS-DOS, tutta la FAT viene caricata in memoria.

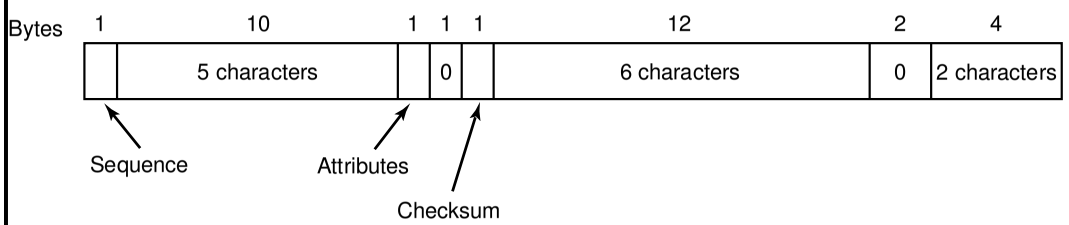
Il block size è chiamato da Microsoft *cluster size*

Limite superiore: 2^32 blocchi da 512 byte = 2TB

Directory in Windows 98

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

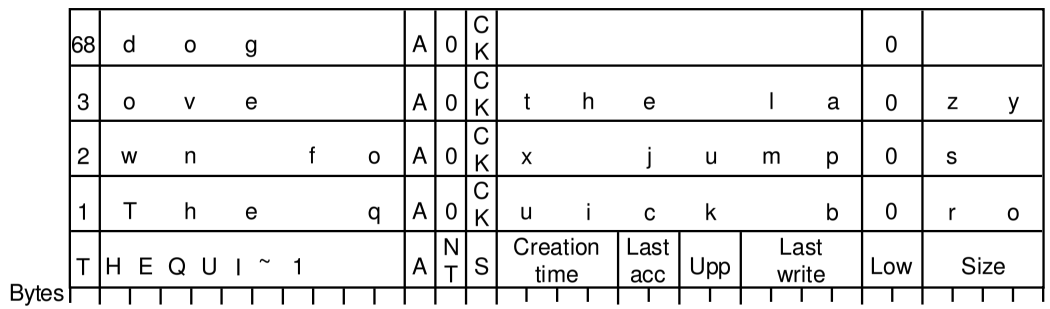




Aove attributes=0x0F (valore invalido per MS-DOS)

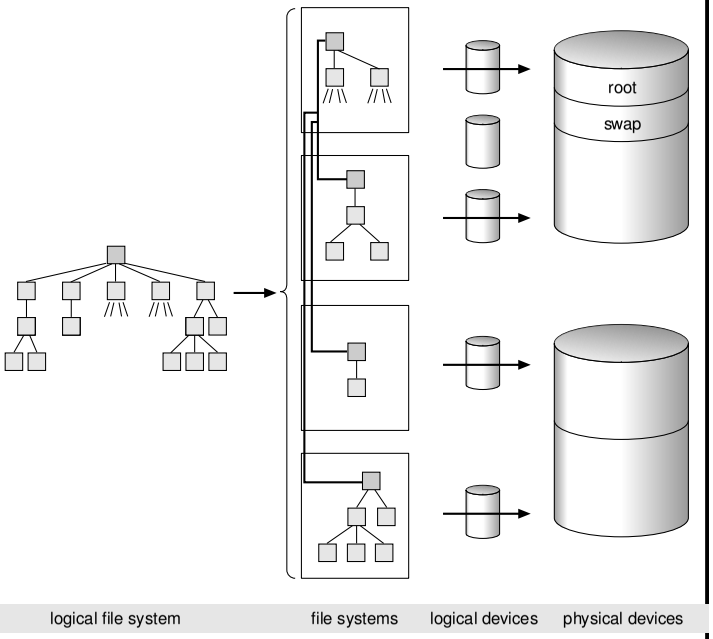
Esempio

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



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, UFS, EFS, EXT2, MSDOS, VFAT, ISO9660, HPFS, HFS, NTFS, ...
- Quelli preferiti sono UFS (Unix File System, aka BSD Fast File System), EXT2 (Extended 2), EFS (Extent File System) e i *journalled file systems* (JFS, XFS, EXT3, ...)
- 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)
  - in SYSV: 2K-8K (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
- N. massimo di blocchi indirizzabile: con blocchi da 4K, puntatori da 4byte

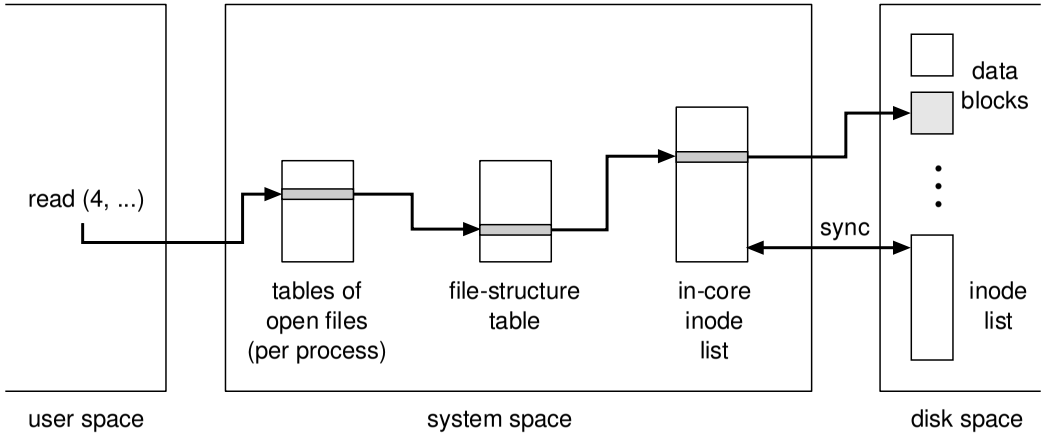
$$\begin{aligned} L_{max} &= 10 + 1024 + 1024^2 + 1024^3 \\ &> 1024^3 = 2^{30} blk \\ &= 2^{42} byte = 4\,TB \end{aligned}$$

molto oltre le capacità dei sistemi a 32 bit.

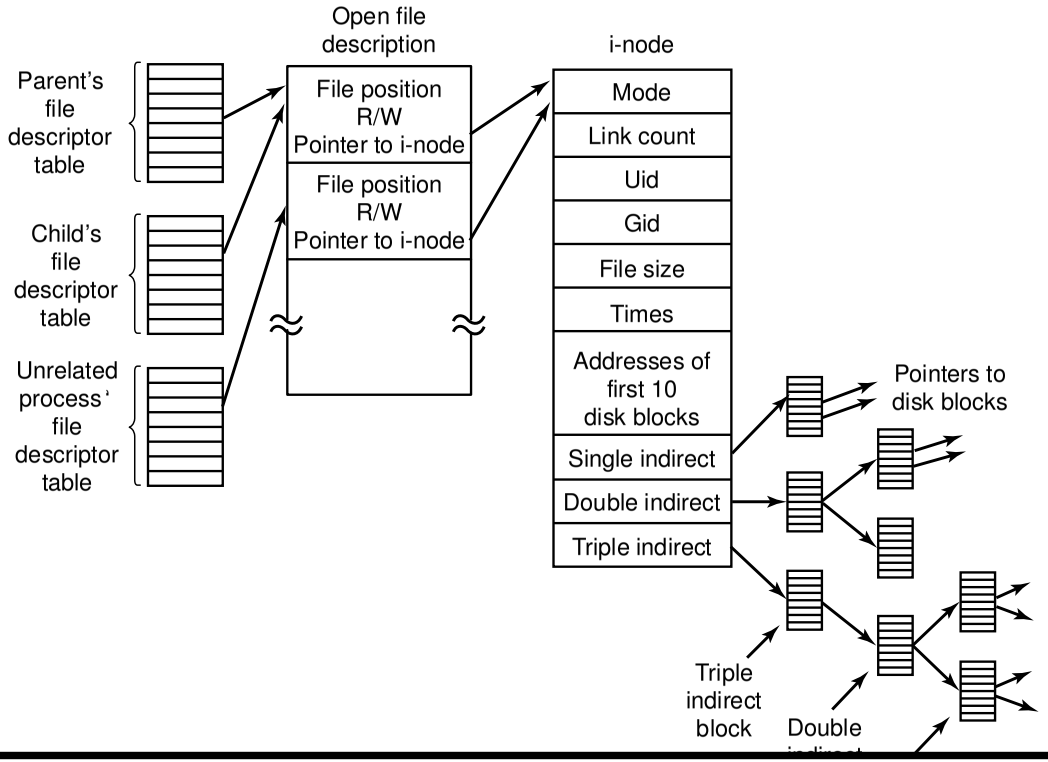
Traduzione da file descriptor a inode

- 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



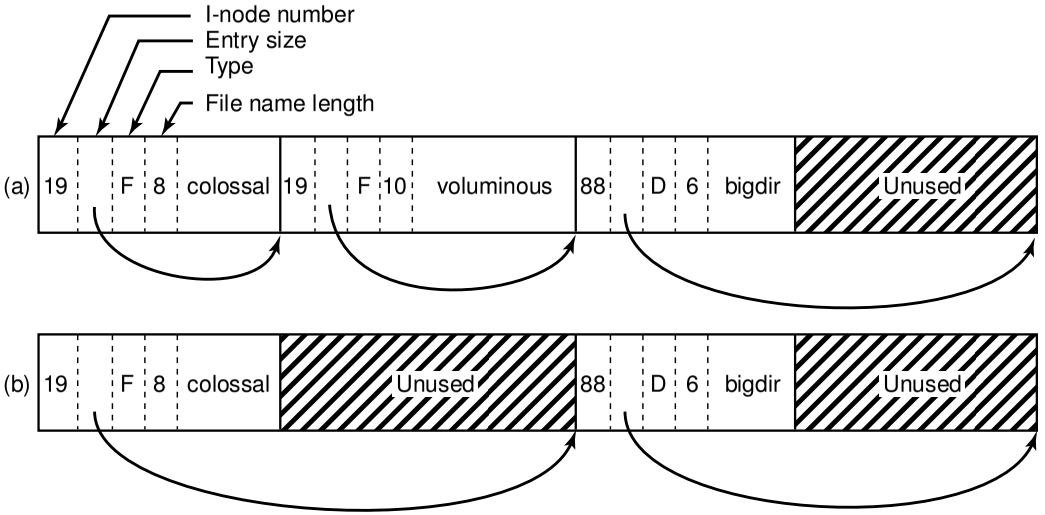
La tabella intermedia è necessaria per la semantica della condivisione dei file tra processi



- 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 $\Rightarrow$ 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

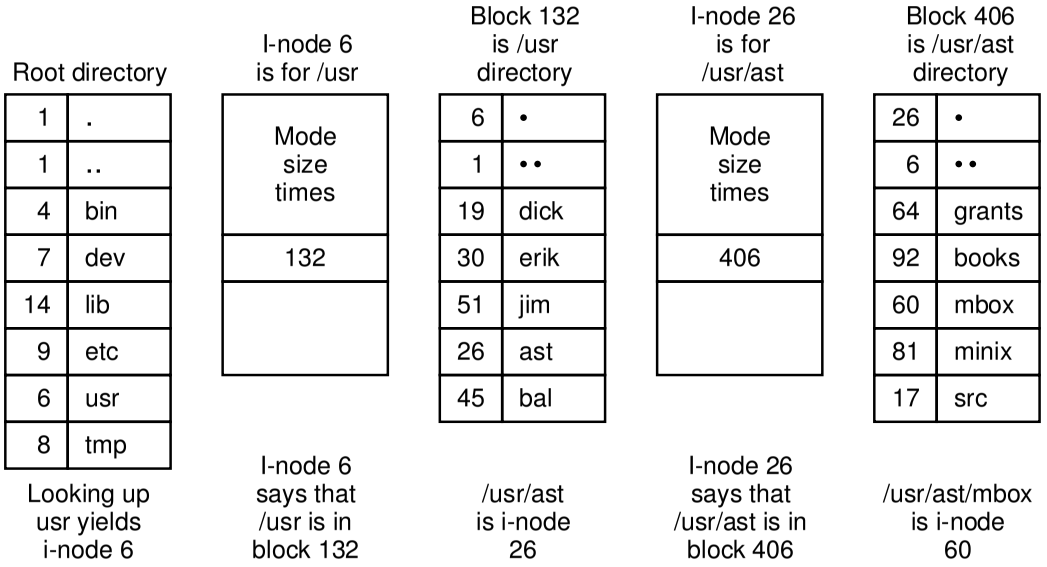
- Il tipo all'interno di un inode distingue tra file semplici e directory
- Una directory è un file con entry di lunghezza variabile. Ogni entry contiene
  - puntatore all'inode del file
  - posizione dell'entry successiva
  - lunghezza del nome del file (1 byte)
  - nome del file (max 255 byte)
- entry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

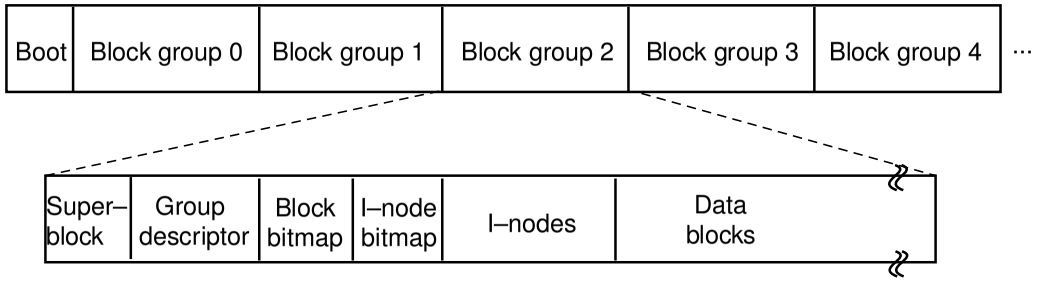
- In UFS (detto anche Berkley Fast File System), i blocchi hanno due dimensioni: il *blocco* (4-8K) e il *frammento* (0.5-1K)
  - Tutti i blocchi di un file sono blocchi tranne l'ultimo
  - L'ultima parte del file è tenuta in frammenti, q.b.
  - Es: un file da 18000 byte occupa 2 blocchi da 8K e 1 da 2K (non pieno)
- 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 Windows NT/2K/XP

- Un file è un oggetto strutturato costituito da *attributi*.
- Ogni attributo è una sequenza di byte distinta (*stream*), come in MacOS. Ogni stream è in pratica un file a se stante (con nome, dimensioni, puntatori, etc.).
- L'indirizzamento è a 64 bit.
- Tipicamente, ci sono brevi stream per i metadati (nome, attributi, Object ID) e un lungo stream per i veri dati, ma nulla vieta avere più stream di dati (es. file server per MacOS)
- I nomi sono lunghi fino a 255 caratteri Unicode.

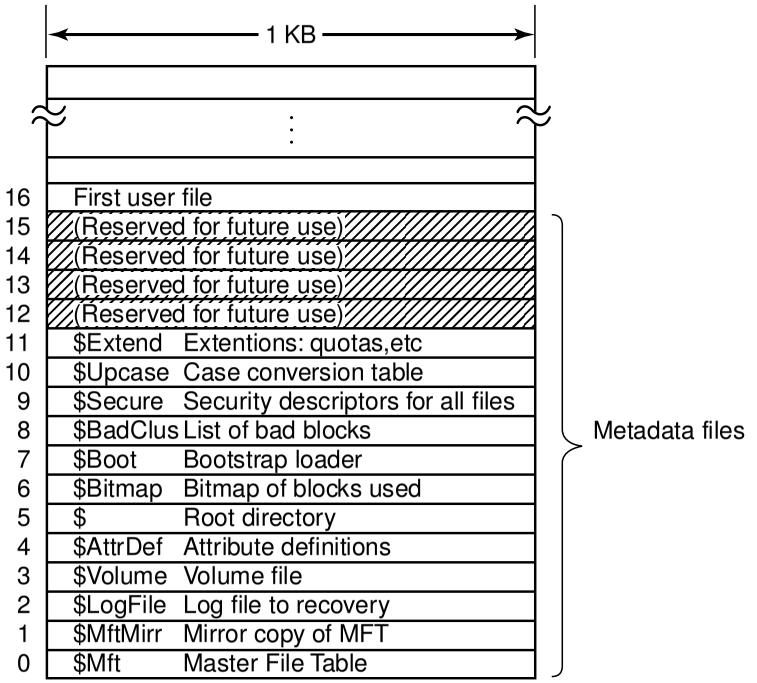
Struttura di NTFS

- Creato da zero, incompatibile all'indietro.
- Diverse partizioni possono essere unite a formare un *volume logico*
- Ha un meccanismo transazionale per i metadati (logging)
- Lo spazio viene allocato a *cluster:* potenze di 2 dipendenti dalla dimensione del disco (tra 512byte e 4K). All'interno di un volume, ogni cluster ha un *logical cluster number*.

Master File Table (MFT)

- È un file di record di 1K, ognuno dei quali descrive un file o una directory
- Può essere collocato ovunque, sul disco,e crescere secondo necessità
- Il primo blocco è indicato nel boot block.
- Le prime 16 entry descrivono l'MFT stesso e il volume (analogo al superblock di Unix). Indicano la posizione della root dir, il bootstrap loader, il logfile, spazio libero (gestito con una bitmap)...
- Ogni record successivo è uno header seguito da una sequenza di coppie (attributo header,valore). Ogni header contiene il tipo dell'attributo, dove trovare il valore, e vari flag.

  I valori possono seguire il proprio header (*resident attribute*) o essere memorizzati in un blocco separato (*nonresident attribute*)



Attributi dei file NTFS

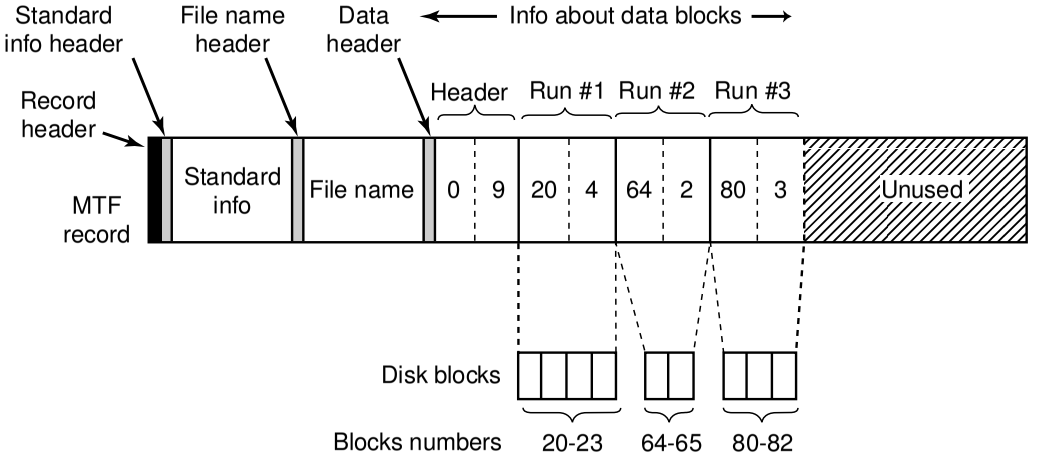
NTFS definisce 13 attributi standard

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

I Data contenono i veri dati; se sono residenti, il file si dice "immediate".

File NTFS non residenti

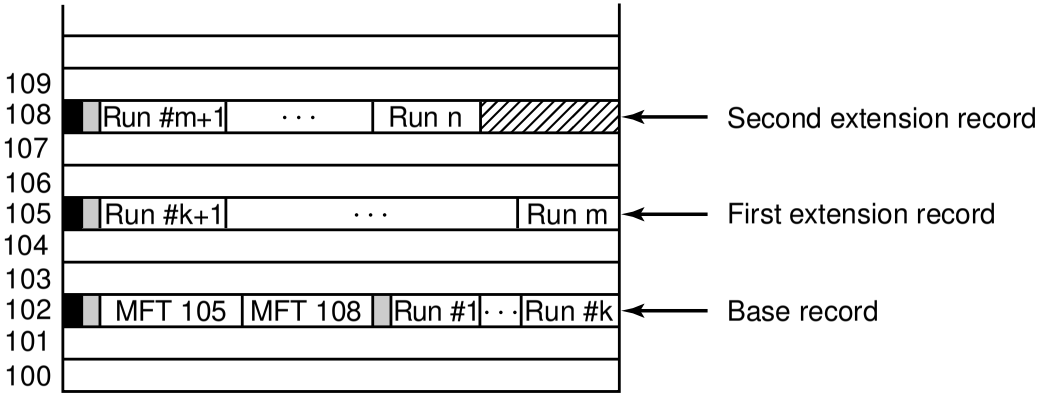
I file non immediati si memorizzano a "run": sequenze di blocchi consecutivi. Nel record MFT corrispondente ci sono i puntatori ai primi blocchi di ogni run.



Un file descritto da un solo MFT record si dice *short* (ma potrebbe non essere corto per niente!)

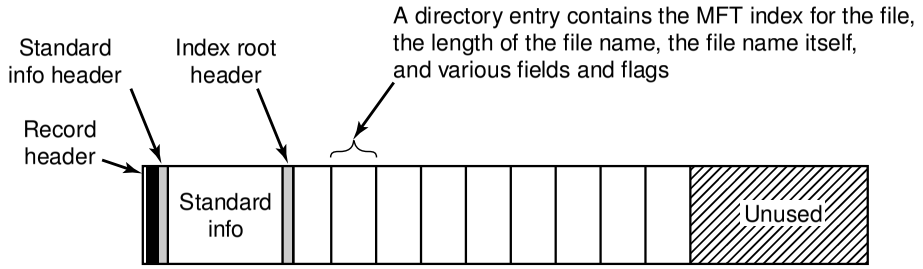
File "long"

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.



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.

Sistemi a più processori

Aumento della necessità di potenza di calcolo.

Velocità di propagazione del segnale (20 cm/ns) impone limiti strutturali all'incremento della velocità dei processori (es: 10GHz ⇒ max 2 cm)

Tendenza attuale: distribuire il calcolo tra più processori.

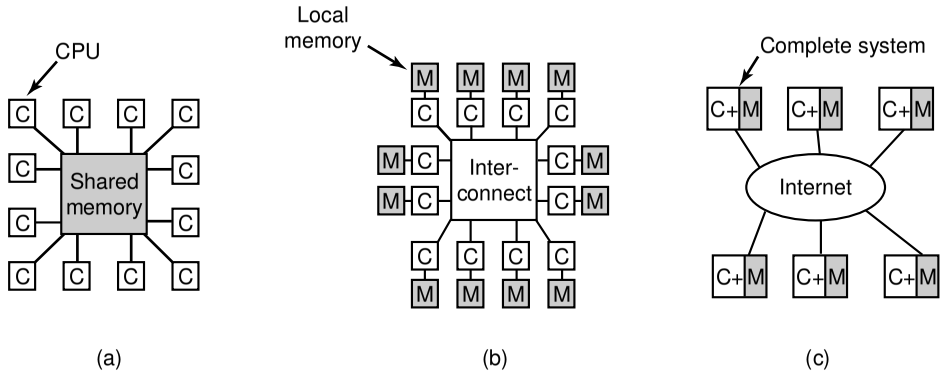
processori strettamente accoppiati :

- *sistemi multiprocessore:* condividono clock e memoria; comunicazione attraverso memoria condivisa: UMA (SMP, crossbar, . . .), NUMA (CC-NUMA, NC-NUMA);
- *multicomputer:* comunicazione con message passing: cluster, COWS.

processori debolmente accoppiati : non condividono clock e/o memoria; comunicano attraverso canali asincroni molto più lenti

- *Sistemi distibuiti:* reti.

Multiprocessore, multicomputer, sistema distribuito

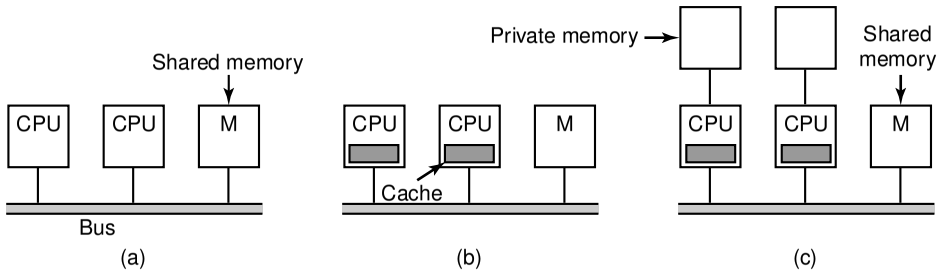


Differenze:

- costo
- scalabilità
- complessità di programmazione/utilizzo

Hardware multiprocessore: SMP UMA

I processori condividono il bus di accesso alla memoria.



Uso di cache per diminuire la contesa per la memoria.

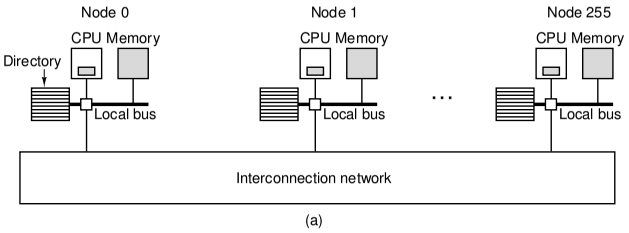
Problemi di coerenza $\Rightarrow$ uso di cache (bus snooping) e memorie private (necessitano di adeguata gestione da parte del compilatore)

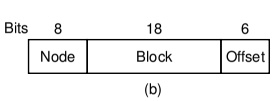
Limitata scalabilità (max 16 CPU, solitamente meno di 8). Oltre servono reti crossbar o omega, o si passa a NUMA.

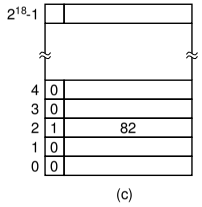
Hardware multiprocessore: NUMA

Applicabili a 100+ processori. Due o più tipi di memorie:

- locale: privata ad ogni CPU
- remota: condivisa tra le CPU; tempo di accesso > di quella locale.

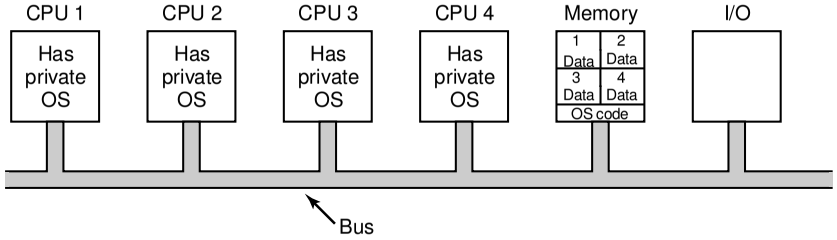






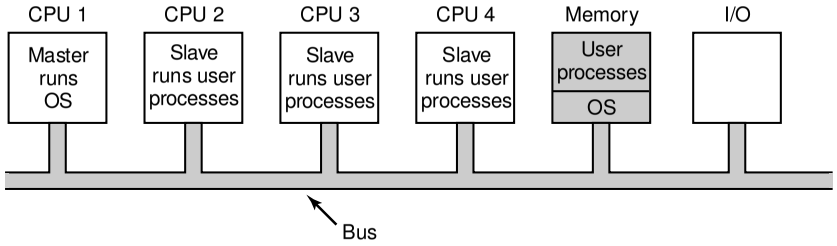
Sistemi Operativi per Multiprocessori

Ciascuna CPU ha il proprio sistema operativo



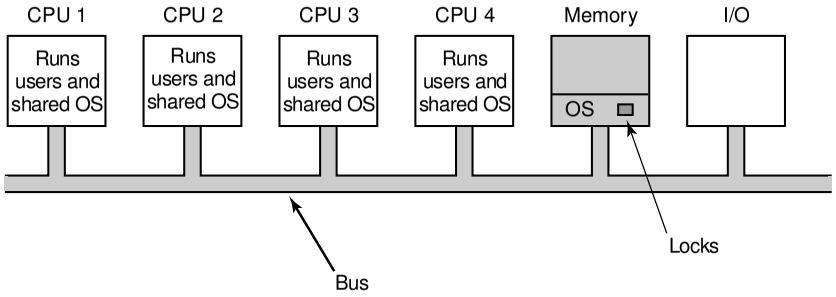
- Ogni CPU esegue privatamente il proprio SO, con i propri dati
- Non esiste distribuzione del carico (solo schedulazione locale), né condivisione della memoria (solo allocazione locale)
- problemi di coerenza tra i vari SO (es: cache di disco). Obsoleto

SO Multiprocessori Master-Slave (o AMP)



- Processi e thread di sistema e in spazio kernel vengono eseguiti solo da un processore; gli altri eseguono solo processi utente
- Il master può allocare i processi tra i vari slave, bilanciando il carico
- La memoria viene allocata centralmente
- Poco scalabile: il master è collo di bottiglia

SO Multiprocessori Simmetrici (SMP)



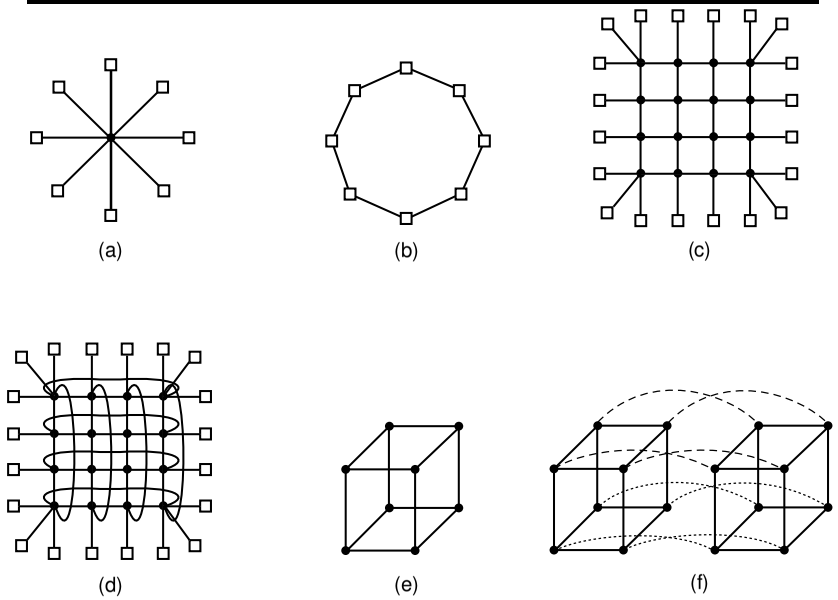
- Completa simmetria fra tutti i processori
- Il S.O. è sempre un collo di bottiglia
  - sia p_{sys} il tempo consumato in modo kernel da un processore
  - la probabilità che almeno 1 su n processori stia eseguendo codice kernel è $p = 1 - (1 - p_{sys})^n$.
  - Per $p_{sys} = 10\%$, $n = 10$: $p = 65\%$. Per $n = 16$: $p = 81.5\%$.

- Un mutex generale su tutto il kernel riduce di molto il parallelismo
- bisogna parallelizzare l'accesso al kernel
- Si può suddividere il kernel in sezioni indipendenti, ognuna ad accesso esclusivo
- Ogni struttura dati del kernel deve essere protetta da mutex
- Richiede una accurata rilettura e "decorazione" del codice del sistema operativo con mutex
- Molto complesso e delicato: errate sequenze di mutex possono portare a deadlock di interi processori in modalità kernel, ossia di congelamenti del sistema

Multicomputer

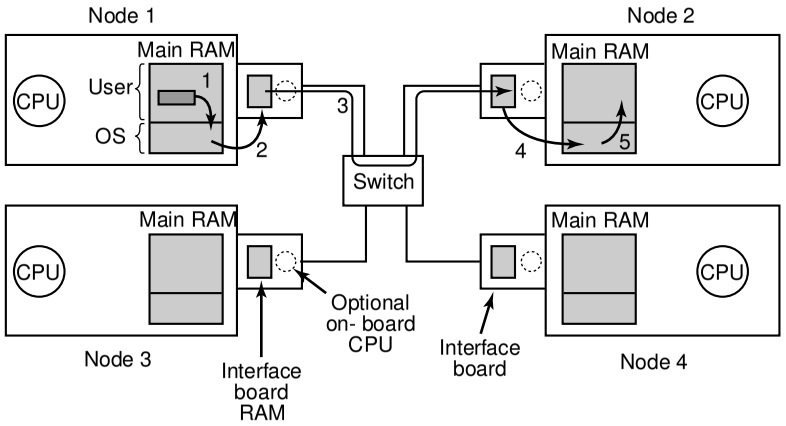
- I multiprocessori sono comodi perché offrono un modello di comunicazione simile a quello tradizionale
- Problemi di costo e scalabilità
- Alternativa: *multicomputer*: calcolatori strettamente accoppiati ma senza memoria condivisa, solo passaggio di messaggi
  - esempio: rete di PC con buone schede di rete (*Clusters of Workstations*)

Topologie di connessione per multicomputer



Gli ipercubi sono molto usati perché diametro = $\log_2$ nodi

Interfacce di rete



- Le interfacce di rete hanno proprie memorie, e spesso CPU dedicate
- Il trasferimento dei dati da nodo a nodo richiede spesso molte copie di dati e quindi rallentamento
- Può essere ridotto usando DMA, memory mapped I/O e lockando in memoria le pagine dei processi utente contenenti i buffer di I/O

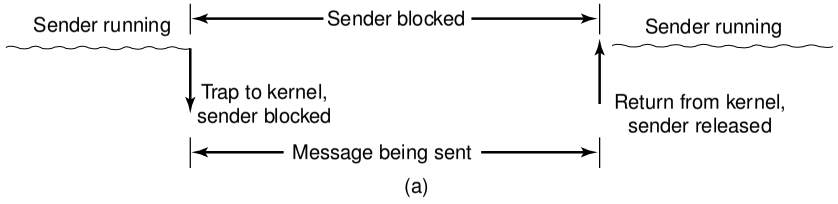
Programmazione in multicomputer: passaggio di messaggi

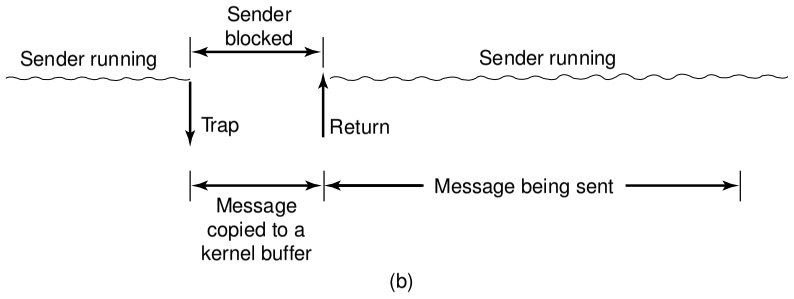
- Il modello base usa comunicazione con passaggio di messaggi
- Il S.O. offre primitive per la comunicazione tra i nodi

```
send(dest, &mptr);
receive(addr, &mptr);
```

- Il programmatore deve approntare appositi processi client/server tra i nodi, con protocolli di comunicazione
- Assomiglia più ad un sistema di rete che ad un unico sistema di calcolo: il programmatore vede la delocalizzazione del calcolo;
- Adottato in sistemi per calcolo scientifico (librerie PVM e MPI per Beowulf, . . . , dove il compilatore parallelizza e distribuisce il codice.)

Send bloccanti/non bloccanti





Quattro possibilità:

1. `send` bloccante: CPU inattiva durante la trasmissione del messaggio
2. `send` non bloccante, con copia su un buffer di sistema: spreco di tempo di CPU per la copia
3. `send` non bloccante, con interruzione di conferma: molto difficile da programmare e debuggare
4. Copia su scrittura: il buffer viene copiato quando viene modificato

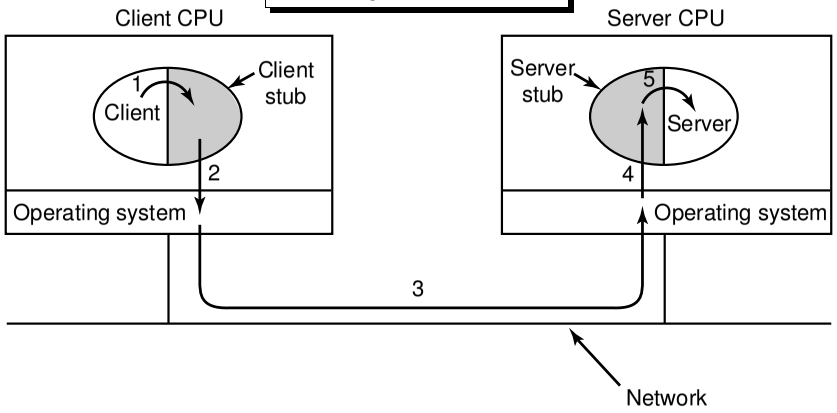
Tutto sommato, la (1) è la migliore (soprattutto se abbiamo a disposizione i thread).

Programmazione in multicomputer: RPC

Chiamate di procedure remote: un modello di computazione distribuita più astratto

- Idea di base: un processo su una macchina può eseguire codice su una CPU remota.
- L'esecuzione di procedure remote deve apparire simile a quelle locali.
- Nasconde (in parte) all'utente la delocalizzazione del calcolo: l'utente non esegue mai send/receive, deve solo scrivere ed invocare procedure come al solito.
- Versione a oggetti: RMI (Remote Method Invocation)

Esempio di RPC



1. Chiamata del client alla procedura locale (allo *stub*)
2. Impacchettamento (*marshaling*) dei dati e parametri
3. Invio dei dati al server RPC
4. Spacchettamento dei dati e parametri
5. Esecuzione della procedura remota sul server.

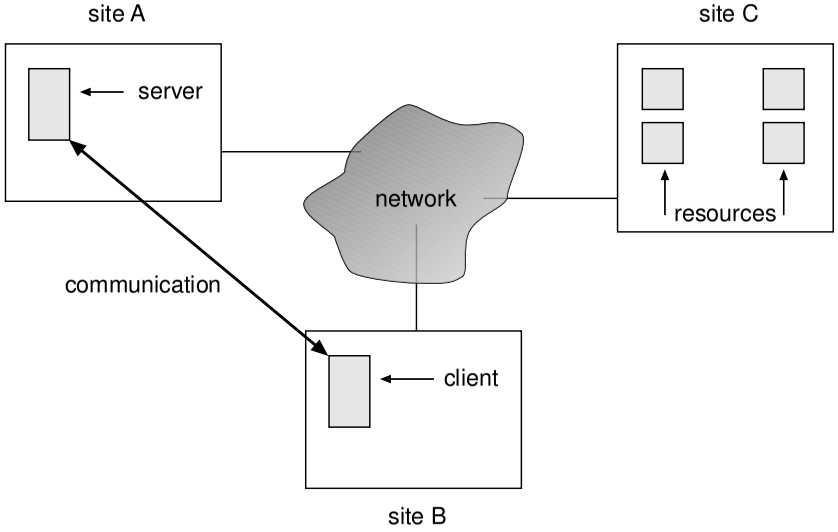
Problemi con RPC

- Come passare i puntatori? A volte si può fare il marshaling dei dati puntati e ricostruirli dall'altra parte (call-by-reference viene sostituito da call-by-copy-and-restore), ma non sempre. . .
- Procedure polimorfe, con tipo e numero degli argomenti deciso al runtime: difficile da fare uno stub polimorfo
- Accesso alle variabili globali: globali a cosa?

Ciò nonostante, le RPC sono state implementate ed usate molto diffusamente, con alcune restrizioni.

Sistemi Distribuiti

Sono sistemi lascamente accoppiati, normalmente più orientati verso la comunicazione (=accesso a risorse remote), che al calcolo intensivo.



server: processo fornitore un servizio; rende disponibile una risorsa ad altri processi (locali o remoti), di cui ha accesso esclusivo (di solito). Attende *passivamente* le richieste dai client.

client processo fruitore di un servizio: richiede un servizio al server (accesso alla risorsa). Inizia la comunicazione (attivo).

protocollo: insieme di regole che descrive le interazioni tra client e server.

Un processo può essere server e client.

Motivazioni per i sistemi distribuiti

- Condivisione delle risorse
  - condividere e stampare file su siti remoti
  - elaborazione di informazioni in un database distribuito
  - utilizzo di dispositivi hardware specializzati remoti
- Accelerazione dei calcoli: bilanciamento del carico
- Affidabilità: individuare e recuperare i fallimenti di singoli nodi, sostituire nodi difettosi
- Comunicazione: passaggio di messaggi tra processi su macchine diverse, similarmente a quanto succede localmente.

Confronto tra i vari modelli paralleli

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

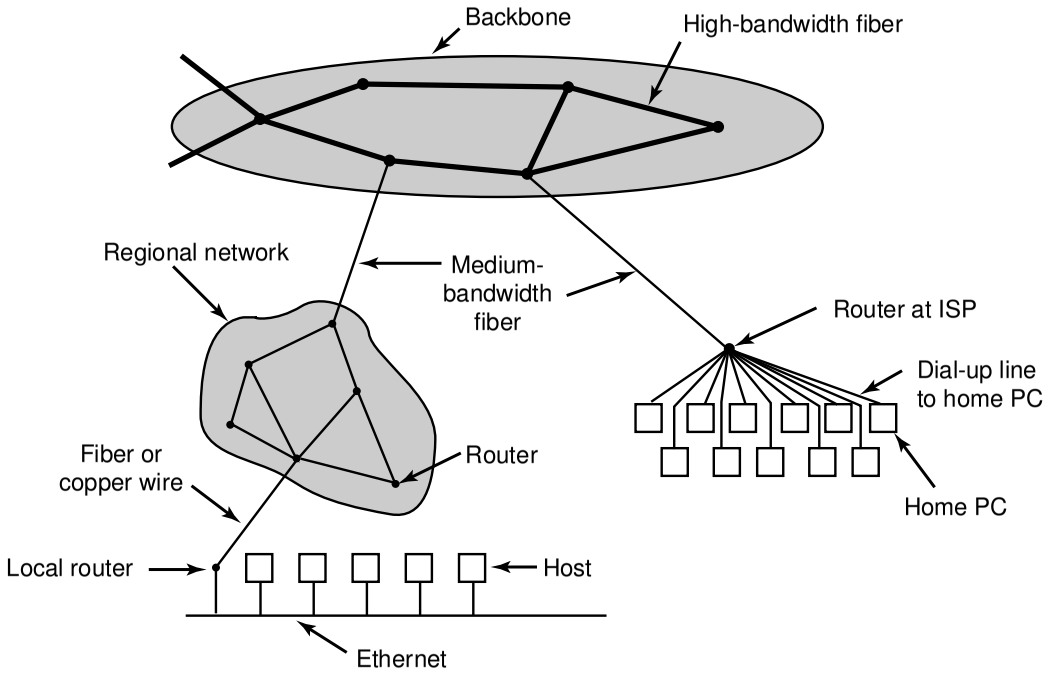
Hardware dei sistemi distribuiti: LAN e WAN

LAN: copre piccole aree geografiche

- struttura regolare: ad anello, a stella o a bus
- Velocità: $\approx$ 100 Mb/sec, o maggiore.
- Broadcast è fattibile ed economico
- Nodi: workstation e/o personal computer; qualche server e/o mainframe

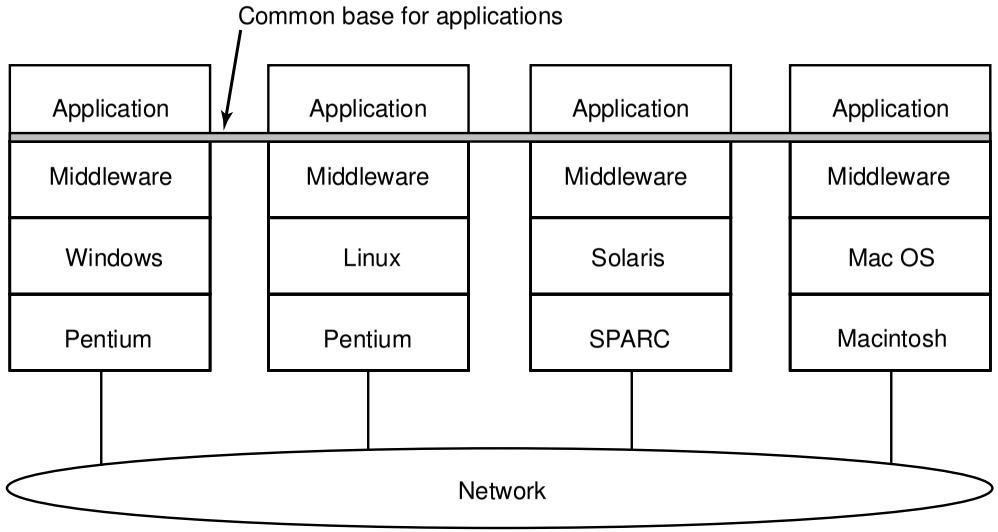
WAN: collega siti geograficamente distanti

- connessioni punto-punto su linee a lunga distanza (p.e. cavi telefonici)
- Velocità $\approx$ 100kbit/sec $-$ 34Mb/sec
- Il broadcast richiede solitamente più messaggi



Programmazione nei sistemi distribuiti

- L'apertura e l'accoppiamento lasco dei sistemi distribuiti aggiunge flessibilità
- Mancando un modello uniforme, la programmazione è complessa
- Spesso le applicazioni di rete devono reimplementare le stesse funzionalità (es: confrontate FTP, IMAP, HTTP, ...)
- Un sistema (operativo) distribuito aggiunge un paradigma (modello comune) alla rete sottostante, per uniformare la visione dell'intero sistema
- Spesso può essere realizzato da uno strato *al di sopra* del sistema operativo, ma *al di sotto* delle singole applicazioni: il *middleware*



Servizi di rete e servizi distribuiti

Un sistema operativo con supporto per la rete può implementare due livelli di servizi:

- *servizi di rete:* offre ai processi le funzionalità necessarie per per stabilire e gestire le comunicazioni tra i nodi del sistema distribuito (es.: socket)
- *servizi distribuiti:* sono modelli comuni (paradigmi di comunicazione) trasparenti che offrono ai processi una visione uniforme, unitaria del sistema distribuito. (es: file system remoto).

Tutti i S.O. moderni offrono servizi di rete; pochi offrono servizi distribuiti, e per modelli limitati.

Servizi e Sistemi operativi di rete

gli utenti sono coscienti delle diverse macchine in rete, e devono passare esplicitamente da una macchina all'altra

- Login su macchine remote (telnet)
- Trasferimento dati tra macchine (FTP)
- Posta elettronica, HTTP,...

Esempio di servizio di rete

```
ten$ ftp maxi
ftp> cd pippo/pluto
ftp> get paperino
ftp> quit
ten$
```

- Locazione non trasparente all'utente
- altro ambiente da imparare
- Manca vera condivisione della risorsa (duplicazione, possibili incoerenze)

Servizi di rete

I servizi sono funzionalità offerte a host e processi.

servizio orientato alla connessione "come un tubo", o una telefonata: si stabilisce una connessione, che viene usata per un certo periodo di tempo trasferendo una sequenza di bit, quindi si chiude la connessione

servizio senza connessione "come cartoline": si trasferiscono singoli messaggi, senza stabilire e mantenere una vera connessione

Ogni servizio ha anche un'*affidabilità:*

- *affidabile:* i dati arrivano sempre, non vengono persi né duplicati.
- *non affidabile:* i dati possono non arrivare, o arrivare duplicati, o in ordine diverso da quello originale

I servizi affidabili = servizi non affidabili + opportune regole di controllo (protocolli), come *pacchetti di riscontro*.

Introducono overhead (di traffico, di calcolo, di tempo) spesso inutile o dannoso. Es: voce/musica/video digitale

	Service	Example
Connection-oriented	Reliable message stream	Sequence of pages of a book
	Reliable byte stream	Remote login
	Unreliable connection	Digitized voice
Connectionless	Unreliable datagram	Network test packets
	Acknowledged datagram	Registered mail
	Request-reply	Database query

Protocolli di rete

- L'implementazione dei servizi di rete avviene stabilendo delle regole di comunicazione tra i vari nodi: i *protocolli*
- Un protocollo stabilisce le regole per attivare, mantenere, utilizzare e terminare un servizio di rete.
- L'implementazione dei protocolli per servizi evoluti è complesso; viene semplificato se i protocolli vengono *stratificati*
  - ogni strato dello *stack* (o *suite*) implementa nuove funzionalità in base a quelli sottostanti
  - in questo modo, si ottiene maggiore modularità e semplicità di progettazione e realizzazione.
- Soprattutto nei sistemi distribuiti, è fondamentale seguire protocolli standardizzati.

Modello ISO/OSI

Fisico: gestisce i dettagli fisici della trasmissione dello stream di bit.

Strato data-link: gestisce i *frame*, parti di messaggi di dimensione fissa, nonché gli errori e recuperi dello strato fisico

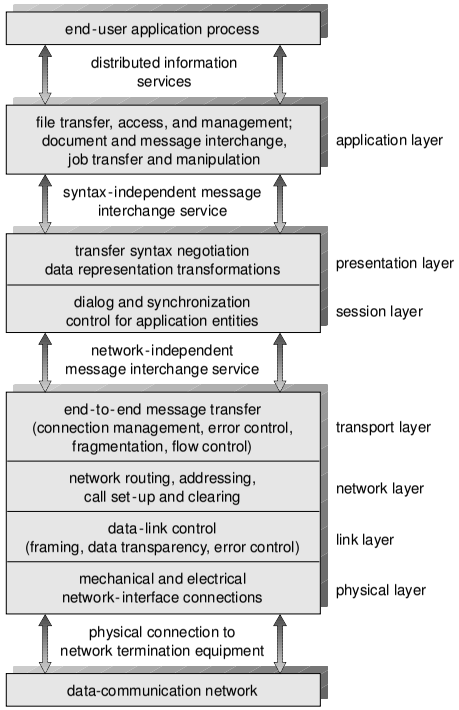
Strato di rete: fornisce le connessioni e instrada i pacchetti nella rete. Include la gestione degli indirizzi degli host e dei percorsi di instradamento.

Trasporto: responsabile dell'accesso di basso livello alla rete e per il trasferimento dei messaggi tra gli host. Include il partizionamento di messaggi in pacchetti, riordino di pacchetti, generazione di indirizzi fisici.

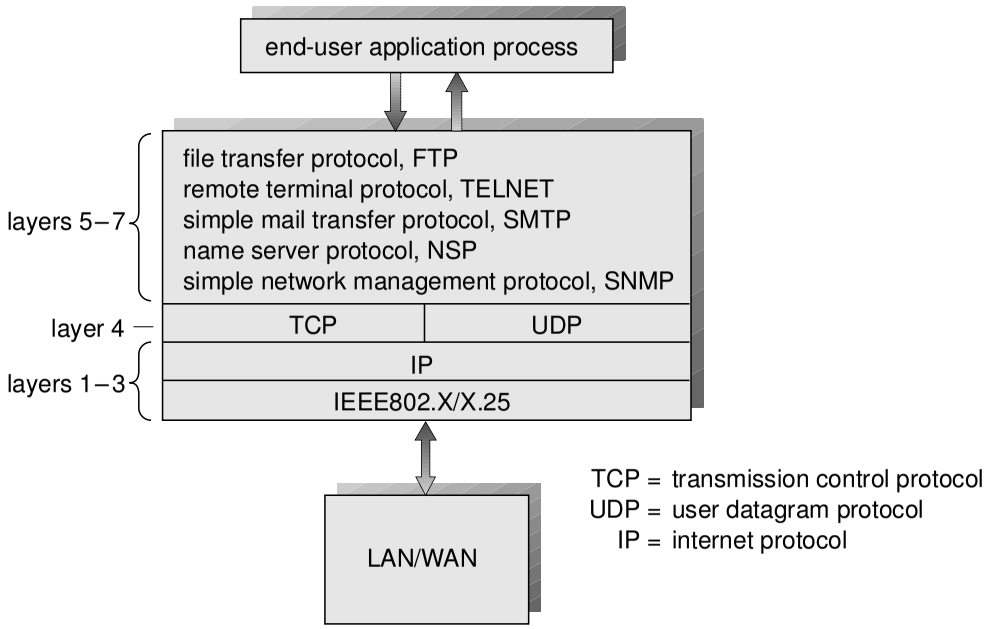
Sessione: implementa sessioni, ossia comunicazioni tra processi

Presentazione: risolve le differenze tra i varî formati dei diversi siti (p.e., conversione di caratteri)

Applicazione: interagisce con l'utente: trasferimento di file, protocolli di login remoto, trasferimento di pagine web, e-mail, database distribuiti, ...



Stack TCP/IP



Modelli di riferimento di rete e loro stratificazione

<table>
<tr><th>ISO
reference
model</th><th>ARPANET
reference
model</th><th>4.2BSD
layers</th><th>example
layering</th></tr>
<tr><td>application</td><td rowspan="3">process
applications</td><td rowspan="2">user programs
and libraries</td><td rowspan="2">telnet</td></tr>
<tr><td>presentation</td></tr>
<tr><td>session</td><td>sockets</td><td>sock_stream</td></tr>
<tr><td>transport</td><td rowspan="2">host–host</td><td rowspan="2">protocol</td><td>TCP</td></tr>
<tr><td rowspan="2">network
data link</td><td>IP</td></tr>
<tr><td rowspan="2">network
interface</td><td rowspan="2">network
interfaces</td><td rowspan="2">Ethernet
driver</td></tr>
<tr><td>hardware</td></tr>
<tr><td></td><td>network
hardware</td><td>network
hardware</td><td>interlan
controller</td></tr>
</table>

Socket

- Una socket ("presa, spinotto") è un'estremità di comunicazione tra processi
- Una socket in uso è solitamente legata (*bound*) ad un indirizzo. La natura dell'indirizzo dipende dal *dominio di comunicazione* del socket.
- Processi comunicanti nello stesso dominio usano lo stesso formato di indirizzi
- Una socket comunica in un solo dominio. I dominî implementati sono descritti in `<sys/socket.h>`. I principali sono
  - il dominio UNIX (AF_UNIX)
  - il dominio Internet (AF_INET, AF_INET6)
  - il dominio Novell (AF_IPX)
  - il dominio AppleTalk (AF_APPLETALK)

Tipi di Socket

- **Stream socket** forniscono stream di dati affidabili, duplex, ordinati. Nel dominio Internet sono supportati dal protocollo TCP.
- **socket per pacchetti in sequenza** forniscono stream di dati, ma i confini dei pacchetti sono preservati. Supportato nel dominio AF_NS.
- **socket a datagrammi** trasferiscono messaggi di dimensione variabile, preservando i confini ma senza garantire ordine o arrivo dei pacchetti. Supportate nel dominio Internet dal protocollo UDP.
- **socket per datagrammi affidabili** come quelle a datagrammi, ma l'arrivo è garantito. Attualmente non supportate.
- **socket raw** permettono di accedere direttamente ai protocolli che supportano gli altri tipi di socket; p.e., accedere TCP, IP o direttamente Ethernet. Utili per sviluppare nuovi protocolli.

Strutture dati per le socket

Struttura "generica" (fa da jolly)

```
struct sockaddr {
    short sa_family;   /* Address family */
    char sa_data[14];  /* Address data.  */
  };
```

Per indicare una socket nel dominio AF_UNIX

```
struct sockaddr_un {
    short sa_family;      /* Flag AF_UNIX */
    char  sun_path[108];  /* Path name  */
  };
```

Per indicare una socket nel dominio AF_INET

```
struct sockaddr_in {
    short sa_family;      /* Flag AF_INET */
    short sin_port;       /* Numero di porta  */
    struct in_addr sin_addr;  /* indir. IP */
    char  sin_zero[8];    /* riempimento */
  };
```

dove `in_addr` rappresenta un indirizzo IP

```
struct in_addr {
  u_long s_addr;          /* 4 byte */
};
```

Chiamate di sistema per le socket

```
s = socket(int domain, int type, int protocol)
```

crea una socket. Se il protocollo è 0, il kernel sceglie il protocollo più adatto per supportare il tipo specificato nel dominio indicato.

```
int bind(int sockfd, struct sockaddr *my_addr,
         int addrlen)
```

Lega un nome ad una socket. Il dominio deve corrispondere e il nome non deve essere utilizzato.

```
int connect(int sockfd,
            struct sockaddr *serv_addr,
            int addrlen)
```

Stabilisce la connessione tra `sockfd` e la socket indicata da `serv_addr`

Chiamate di sistema per le socket (cont.)

- Un processo server usa `listen` per fissare la coda di clienti e `accept` per mettersi in attesa delle connessioni. Solitamente, per ogni client viene creato un nuovo processo (fork) o thread.
- Una socket viene distrutta chiudendo il file descriptor associato alla connessione o con la `shutdown`.
- Con la `select` si possono controllare trasferimenti di dati da più file descriptors/socket descriptor
- Le socket a messaggi si usano con le system call `sendto` e `recvfrom`

```
int sendto(int s, const void *msg, int len, unsigned int
           flags, const struct sockaddr *to, int tolen);
int recvfrom(int s, void *buf, int len, unsigned int flags
             struct sockaddr *from, int *fromlen);
```


Servire più client contemporaneamente

- Assegnare un thread/processo ad ogni client. Esempio:

```
sock = socket(...);
bind(sock, ...);
listen(sock, 2);
while(1) {
  fd = accept(sock,...);
  if (fork() == 0) {
     /* gestione del client */
     ...
     exit(0);
  } else  /* padre */
    close(fd);
}
```

- Si possono usare i processi se i varî server sono scorrelati e per sicurezza; meglio i thread se i server devono accedere a strutture comuni e per maggiore efficienza.
- Se il sistema non supporta i thread ed è necessario avere uno stato comune, meglio usare la `select(2)`

Monitoraggio socket: netstat

```
miculan@coltrane:$ netstat --program
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program na
tcp        0      0 coltrane:34179          popmail.inwind.it:pop   ESTABLISHED 1741/fetchmail
tcp        0      0 coltrane:34178          farfarello:http         TIME_WAIT   -
tcp        0      0 coltrane:34174          maxi:ssh                ESTABLISHED -
tcp        0      0 coltrane:34184          lacerta.cc.uni:webcache ESTABLISHED 1267/opera
tcp        0      0 coltrane:34185          lacerta.cc.uni:webcache ESTABLISHED 1267/opera
tcp        0      0 coltrane:34180          lacerta.cc.uni:webcache ESTABLISHED 1267/opera
tcp        0      0 coltrane:34181          lacerta.cc.uni:webcache TIME_WAIT   -
tcp        0      0 coltrane:34182          lacerta.cc.uni:webcache TIME_WAIT   -
tcp        0      0 coltrane:34183          lacerta.cc.uni:webcache TIME_WAIT   -
tcp        0      0 coltrane:34177          lacerta.cc.uni:webcache ESTABLISHED 1267/opera
tcp        0      0 coltrane:ipp            ten.dimi.uniud.it:791   TIME_WAIT   -
tcp        0      0 coltrane:34176          ten.dimi.uniud.it:791   TIME_WAIT   -
tcp        0      0 coltrane:34186          ten.dimi.uniud.it:791   TIME_WAIT   -
tcp        0      0 coltrane:34175          ten.dimi.uniud.it:791   TIME_WAIT   -
tcp        0      0 coltrane:32772          ten.dimi.uniud.it:imap  ESTABLISHED 1233/pine
tcp        0      0 coltrane:32772          ten.dimi.uniud.it:imap  ESTABLISHED 1233/pine
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node PID/Program name    Path
unix  11     [ ]         DGRAM                    815    -                   /dev/log
unix  3      [ ]         STREAM     CONNECTED     5852   -                   /tmp/.X11-unix/X0
```

```
unix  3      [ ]         STREAM     CONNECTED     5851   1267/opera
unix  3      [ ]         STREAM     CONNECTED     5462   -                    /tmp/.X11-unix/X0
unix  3      [ ]         STREAM     CONNECTED     5461   1259/xdvi.bin
unix  3      [ ]         STREAM     CONNECTED     2314   -                    /tmp/.X11-unix/X0
unix  3      [ ]         STREAM     CONNECTED     2313   1145/xemacs
unix  3      [ ]         STREAM     CONNECTED     2194   -
unix  3      [ ]         STREAM     CONNECTED     2193   1144/gnome-terminal
unix  3      [ ]         STREAM     CONNECTED     2192   -
unix  3      [ ]         STREAM     CONNECTED     2191   1144/gnome-terminal
unix  3      [ ]         STREAM     CONNECTED     1948   -                    /tmp/.X11-unix/X0
unix  2      [ ]         DGRAM                    1683   -
[...]
unix  2      [ ]         DGRAM                    1619   -
unix  2      [ ]         DGRAM                    1540   -
unix  2      [ ]         DGRAM                    1489   -
unix  2      [ ]         DGRAM                    1273   -
unix  2      [ ]         DGRAM                    1165   -
unix  2      [ ]         DGRAM                    1107   -
unix  2      [ ]         DGRAM                    881    -
unix  2      [ ]         DGRAM                    824    -
miculan@coltrane:$
```

I daemon e i runlevel di Unix

- *daemon* = un processo di Unix che è sempre in esecuzione: viene lanciato al boot time e terminato allo shutdown da appositi script. Esempio, il *sendmail* (che gestisce l'email):
  lanciato da `/etc/rc.d/rc3.d/S16sendmail`
  terminato da `/etc/rc.d/rc0.d/K10sendmail`.

- Questi processi offrono *servizi* di sistema (login remoto, email, server di stampa, oraesatta. . .) o di "applicazione" (http, database server. . .).

- Tradizionalmente, il nome termina per "d" (kpiod, kerneld, crond, inetd. . .).

- Un insieme di daemon in esecuzione forma un *runlevel*.

I daemon e i runlevel di Unix (cont.)

Convenzionalmente, ci sono 7 runlevel:

0 = shutdown: terminazione di tutti i processi

1 = single user: no rete, no login multiutente

2 = multiuser, senza supporto di rete

3 = multiuser, supporto di rete

4 = non usato, sperimentale

5 = multiuser, supporto di rete, login grafico

6 = terminazione di tutti i processi e reboot

I daemon e i runlevel di Unix (cont.)

Ogni runlevel viene definito

- nei BSD-like: da uno script (es. `/etc/rc3`) che viene eseguito dal processo 1 (`init`) al boot

- nei SVR4-like: da una directory di script (es. `/etc/rc.d/rc3.d`), uno per servizio. All'ingresso nel runlevel, quelli che iniziano per `S` vengono eseguiti da `init` con l'argomento "start" (*sequenza di startup*)

I daemon e i runlevel di Unix (cont.)

- Es, su Linux (RedHat, SysV-like):

```
miculan@coltrane:~$ ls /etc/rc.d/rc3.d
K20nfs         K73ypbind     S10network   S25netfs        S80sendmail
K20rwhod       K74nscd       S12syslog    S28autofs       S85gpm
K46radvd       K74ntpd       S13portmap   S55sshd         S90crond
K50snmpd       K92ipchains   S14nfslock   S56rawdevices   S90xfs
K50snmptrapd   K92iptables   S17keytable  S56xinetd       S95anacron
K65identd      S05kudzu      S20random    S60lpd          S95atd
miculan@coltrane:~$
```

- Si sceglie il runlevel
  - di default: è scritto nella tabella `/etc/inittab`
  - al boot: argomento del kernel (e.g., `linux 1`)
  - al runtime: con il comando `telinit <n>`

I servizi e le porte standard di Unix

- Ad ogni servizio viene assegnata una *porta*
- I servizi più comuni utilizzano porte standard, sia TCP che UDP
- Le porte < 1024 possono essere utilizzate solo da processi con UID=0, per "garantire" la controparte della liceità del demone
- Chiunque può impiegare porte ≥ 1024, se non sono usati da altri processi
- Si può far girare i servizi su porte non standard (es.: alcuni httpd sono su porte diverse da 80)

Il "super server" `inetd`

- I daemon che vengono lanciati dagli script di runlevel sono "standalone"
- Viceversa, molti demoni standard vengono eseguiti sotto il "super server" *inetd* (o *xinetd*): il processo viene creato solo al momento della richiesta
- meno memoria consumata, ma più lento nello startup
- *inetd* si configura da `/etc/inetd.conf`

```
...
# These are standard services.
#
ftp    stream tcp   nowait  root   /usr/sbin/tcpd in.ftpd -l -a
telnet stream tcp   nowait  root   /usr/sbin/tcpd in.telnetd
#
# Shell, login, exec, comsat and talk are BSD protocols.
#
shell  stream tcp   nowait  root   /usr/sbin/tcpd in.rshd
login  stream tcp   nowait  root   /usr/sbin/tcpd in.rlogind
...
```



```
miculan@coltrane:miculan$ telnet ten 25
Trying 158.110.144.132...
Connected to ten.
Escape character is '^]'.
220 ten.dimi.uniud.it 5.67a/IDA-1.5 Sendmail is ready at Tue, 20 Apr 1999 12:55:12 +0200
vrfy miculan
250 Marino Miculan <miculan>
mail from: me
250 me... Sender ok
rcpt to: miculan
250 miculan... Recipient ok
data
354 Enter mail, end with "." on a line by itself
questo e' un messaggio di esempio
.
250 Ok
quit
221 ten.dimi.uniud.it closing connection
Connection closed by foreign host.
miculan@coltrane:miculan$
```

Servizi e Sistemi operativi distribuiti

Problemi di progetto:

Trasparenza e località: i sistemi distribuiti dovrebbero apparire come sistemi convenzionali e non distinguere tra risorse locali e remote

Mobilità dell'utente: presentare all'utente lo stesso ambiente (i.e., home dir, applicativi, preferenze,...) ovunque esso si colleghi

Tolleranza ai guasti: i sistemi dovrebbero continuare a funzionare, eventualmente con qualche degrado, anche in seguito a guasti

Scalabilità: aggiungendo nuovi nodi, il sistema dovrebbe essere in grado di sopportare carichi proporzionalmente maggiori

Sistemi su larga scala: il carico per ogni componente del sistema deve essere limitato da una costante indipendente dal numero di nodi, altrimenti non si può scalare oltre un certo limite

struttura dei processi server: devono essere efficienti nei periodi di punta, quindi è meglio usare processi multipli o thread per servire in parallello

Tolleranza ai guasti

Per assicurare che il sistema sia robusto, si deve

- *Individuare i fallimenti* di link e di siti
- *Riconfigurare il sistema* in modo che la computazione possa proseguire
- *Recuperare lo stato precedente* quando un sito/collegamento viene riparato

Rilevamento dei guasti: Handshaking

- Ad intervalli fissati, i siti A e B si spediscono dei messaggi *I-am-up*. Se il messaggio non perviene ad A entro un certo tempo, si assume che B è guasto, o il link è guasto, o il messaggio da B è andato perduto
- Quando A spedisce la richiesta *Are-you-up?*, specifica anche un tempo massimo per la risposta. Passato tale periodo, A conclude che almeno una delle seguenti situazioni si è verificata:
  - B è spento
  - Il link diretto (se esiste) da A a B è guasto
  - Il percorso alternativo da A a B è guasto
  - Il messaggio è andato perduto

  Non si può sapere con certezza quale evento si è verificato

Riconfigurazione

- È una procedura che permette al sistema di riconfigurarsi e continuare il funzionamento
- Se il link tra A e B si è guastato, questa informazione deve essere diramata agli altri siti del sistema, in modo che le tabelle di routing vengano aggiornate
- Se si ritiene che un sito si è guastato, allora ogni sito ne viene notificato affincé non cerchi di usare i servizi del sito guasto

Ripristino dopo un guasto

- Quando un collegamento o sito guasto viene recuperato, deve essere reintegrato nel sistema in modo semplice e lineare
- Ad esempio, se il collegamento tra A e B era guasto, quando viene riparato sia A che B devono essere notificati. Si può implementare ripetendo la procedura di handshaking
- Se il sito B era guasto, quando viene ripristinato deve notificare gli altri siti del sistema che è di nuovo in piedi. Il sito B può quindi ricevere dati dagli altri sistemi per aggiornare le tabelle locali

Modelli dei servizi distribuiti

Modi essenziali per implementare un modello omogeneo per un servizio distribuito: il middleware può implementare

Migrazione di dati: offre un modello di dati omogeneo tra i nodi (non distinguo "dove stanno i dati")

Migrazione delle computazioni: offre un modello uniforme di calcolo distribuito (non distinguo "dove viene eseguita la prossima istruzione")

Migrazione dei processi: offre un modello uniforme di schedulazione (non distinguo "dove viene eseguito un processo")

Coordinazione distribuita: offre un modello uniforme di memoria associativa distribuita (non distinguo "dove stanno i dati consumabili")

Migrazione di dati

Quando un processo deve accedere ad un dato, si procede al trasferimento dei dati dal server al client.

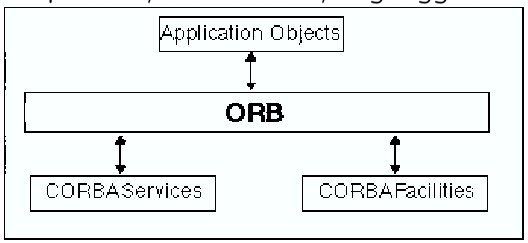
- Middleware basato su documenti: un modo uniforme per raccogliere ed organizzare documenti distribuiti eterogenei. Es: WWW, Lotus Notes.
- Middleware basato su *file system distribuiti:* decentralizzazione dei dati.
  - upload/download: copie di interi file (AFS, Coda) $\Rightarrow$ caching su disco locale, adeguato per accessi a tutto il file (i.e., connectionless)
  - accesso remoto: copie di parti di file (NFS, SMB) $\Rightarrow$ efficiente per pochi accessi, adeguato su reti locali (affidabili)

Migrazione di computazioni

- Quando un processo deve accedere ad un dato, si procede al trasferimento della *computazione* dal client al server. Il calcolo avviene sul server e solo il risultato viene restituito al client.
- Efficiente se trasferire la computazione costa meno dei dati
- client e server rimangono processi *separati*
- Implementazioni tipiche:
  - RPC (Remote Procedure Calls), RMI (Remote Method Invocation): il server implementa un *tipo di dato astratto* (o un oggetto) le cui funzioni (metodi) sono accessibili dai programmi client.
  - CORBA, Globe: il middleware (es. i server Object Request Brokers) instrada chiamate a metodi di oggetti remoti. L'utente vede un insieme di oggetti condivisi, senza sapere dove sono realmente localizzati.

Migrazione trasparente di computazioni: CORBA in OMA

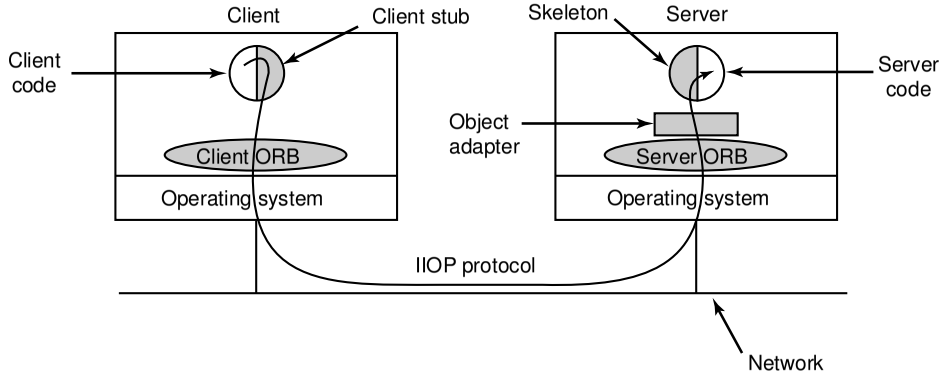
- Parte della *Object Management Architecture*, pensata per esser cross-platform (sistemi operativi, architetture, linguaggi differenti)



- Un oggetto CORBA esporta un insieme di metodi specificati da una interfaccia descritta in un formato standard (Interface Definition Language)
- La creazione di un oggetto CORBA restituisce un riferimento che può essere passato ad altri oggetti, o inserito in *directories*.
- Un processo client può chiamare i metodi di qualsiasi oggetto di cui ha il riferimento (che può ottenere da una directory)

CORBA (Cont.)

- La chiamata è mediata dal middleware, i server ORB. Simile a RMI.



- Una chiamata può attraversare più ORB. Protocollo standard (IIOP) per la comunicazione client-server e server-server.
- Limitazione: ogni oggetto è localizzato su un solo server $\Rightarrow$ scalabilità limitata. Risolto in Globe.

Migrazione di processi/thread

- Interi processi (o thread) vengono spostati da una macchina all'altra. L'utente non sa dove effettivamente viene eseguito un suo processo/thread.
- Gli scheduler dei singoli sistemi operativi comunicano per mantenere un carico omogeneo tra le macchine.
- Vantaggi:
  - Bilanciamento di carico
  - Aumento delle prestazioni (parallelismo)
  - Utilizzo di software/hardware specializzati
  - Accesso ai dati

  Svantaggio: spostare un processo è costoso e complesso
- Esempio: MOSIX, OSF DCE, Solaris Full Moon, Microsoft Wolfpack.

Coordinazione su memoria associativa distribuita

- Modo moderno per unire coordinazione e condivisione di dati
- Viene mantenuto uno spazio di memoria associativa distribuita ("tuplespace") ove possono essere caricate delle *tuple* (record, struct). Non sono oggetti, solo liste di scalari:

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
```

- Operazioni sullo spazio condiviso:
  - `out("abc",2,5)`: inserimento (non bloccante)
  - `in("abc", 2, ?i)`: rimozione bloccante con pattern matching
  - `read("abc", 2, ?i)`: lettura non distruttiva con pattern matching
  - `eval("worker", worker())`: creazione di un processo nel TupleSpace

Coordinazione distribuita

- Esempio: implementazione di un semaforo condiviso:

```
up(S) = Out("semaforo", S);
down(S) = In("semaforo", S);
```

- Pro: flessibili, generali

- Cons: difficili da implementare in modo realmente distribuito (spesso si usa un server centrale)

- Implementato in Linda, `publish/subscribe`, Jini, Klaim

Servizi remoti: RPC

- Richieste di accesso ad un file remoto: la richiesta è tradotta in messaggi al server, che esegue l'accesso al file, impacchetta i dati in un messaggio e lo rispedisce indietro al client.

- Un modo comune per implementare questo meccanismo è con le *Remote Procedure Call* (RPC), concettualmente simili a chiamate locali

  – I messaggi inviati ad un demone RPC in ascolto su una *porta* indicano quale procedura eseguire e con quali parametri. La procedura viene eseguita, e i risultati sono rimandati indietro con un altro messaggio

  – Una *porta* è un numero incluso all'inizio del messaggio. Un sistema può avere molte porte su un solo indirizzo di rete, per distinguere i servizi supportati.

- Maggiore differenza rispetto a chiamate locali: errori di comunicazione ne cambiano la semantica. Il S.O. deve cercare di ottenere questa astrazione.

Associazione porta/servizio per RPC

- Le informazioni di collegamento possono essere fissati (porta fissata a priori)

  – Al compile time, ogni chiamata RPC ha un numero di porta fissato

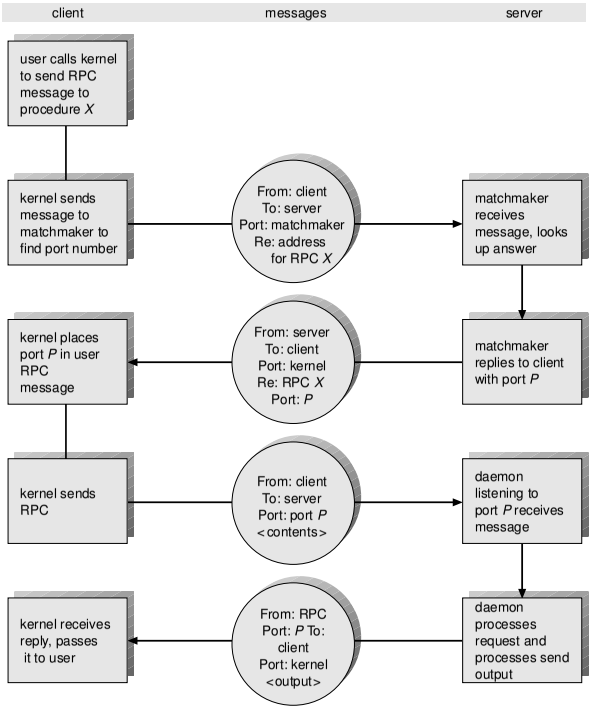
  – Il server non può cambiare la porta senza ricompilare il client

  – Molte porte/servizi sono standardizzati (eg: telnet=23, SMTP=25, HTTP=80 (ma questi non sono servizi RPC))

- L'associazione può essere stabilita dinamicamente con un meccanismo di rendezvous

  – Il S.O. implementa un demone di rendezvous su una porta fissata (111: *portmapper* o *matchmaker*)

  – I client, prima di eseguire la vera RPC, chiedono qual'è la porta da utilizzare al demone di rendezvous



Esempio di schema RPC

Un file system distribuito può essere implementato come un insieme di chiamate RPC

- I messaggi sono inviati alla porta associata al demone file server, sulla macchina su cui risiedono fisicamente i dati.
- I messaggi contengono l'indicazione dell'operazione da eseguirsi (i.e., **read**, **write**, **rename**, **delete**, o **status**).
- Il messaggio di risposta contiene i dati risultati dall'esecuzione della procedura, che è eseguta dal demone per conto del client

Il Sun Network File System (NFS)

- Una implementazione e specifica di un sistema software per accedere file remoti attraverso LAN (o WAN, se proprio uno deve...)
- Le workstation in rete sono viste come macchine indipendenti con file system indipendenti, che permettono di condividere in maniera trasparente
  - Una dir remota viene montata su una dir locale, come un file system locale.
  - La specifica della dir remota e della macchina non è trasparente: deve essere fornita. I file nella dir remota sono accessibili in modo trasparente
  - A patto di averne i diritti, qualsiasi file system o dir entro un file system può essere montato in remoto

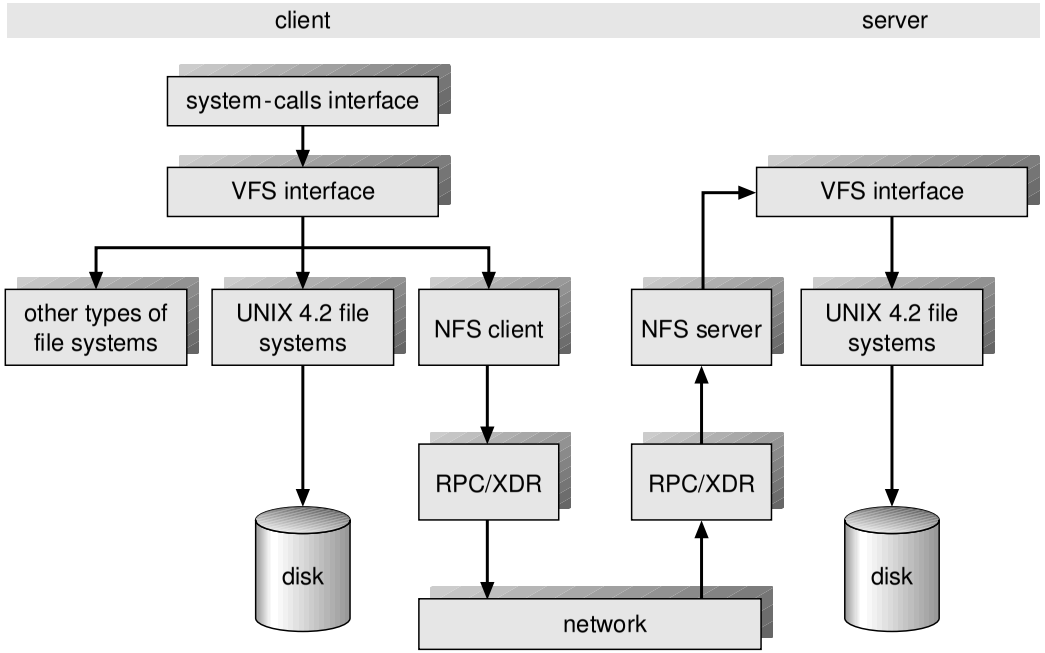
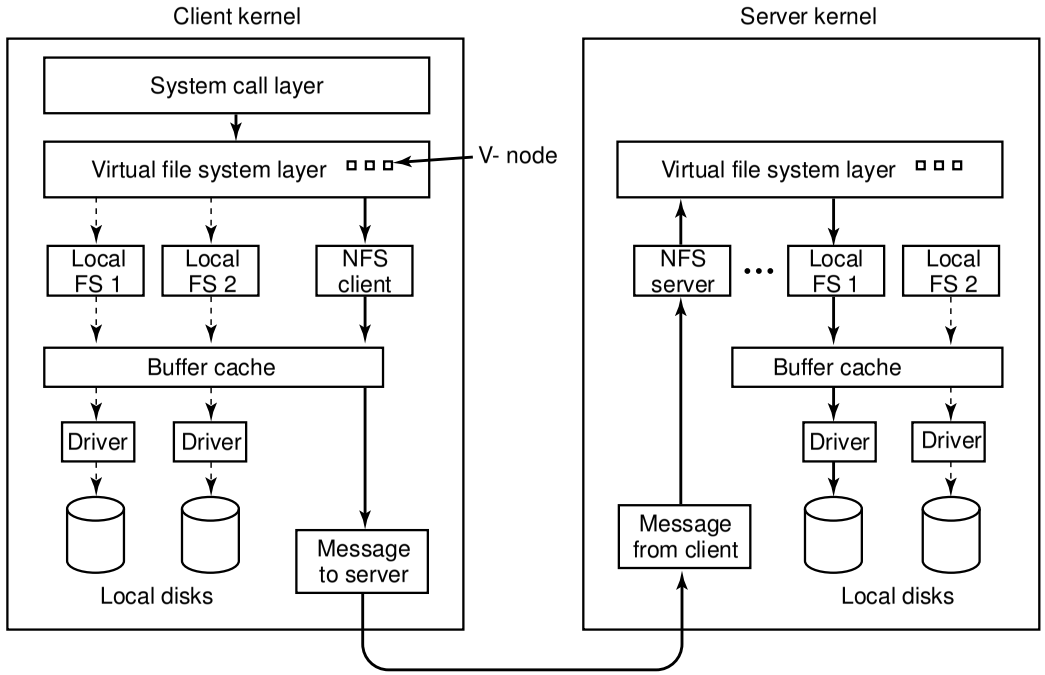
NFS (Cont.)

- NFS è progettato per funzionare in un ambiente eterogeneo, di macchine e S.O. differenti; la specifica è indipendente dagli strati sottostanti
- Questa indipendenza si ottiene attraverso chiamate RPC costruite sopra il protocollo eXternal Data Representation (XDR)
- La specifica NFS distingue tra i servizi del protocollo di mount e quello di vero accesso ai file remoti

Protocollo NFS Mount

- Stabilisce la connessione logica iniziale tra server e client
- L'operazione di mount include il nome della dir remota e della macchina server che lo contiene
  - La richiesta di mount è tradotta in una chiamata RPC e inoltrata al mount daemon sulla macchina server
  - *Export list:* specifica quali file system locali sono esportati per il mounting, con il nome delle macchine che possono montarli
- Il risultato dell'operazione di mount è un *file handle*, una chiave per le richieste successive.
- File handle: file-system identifier, e un numero di inode che identifica la directory montata nel file system esportato.
- L'operazione di mount cambia solo lo stato del client ma non modifica lo stato del server

Protocollo NFS

- Fornisce un insieme di RPC per le operazioni remote su file.
  - ricerca di un file in una directory
  - lettura di un insieme di entries in una dir
  - manipolazione di link e dir
  - accesso agli attributi dei file
  - lettura e scrittura dei file
- I server NFS sono *stateless*: ogni richiesta è a se stante, e deve fornire tutti gli argomenti che servono
- Per aumentare l'efficienza, NFS usa cache di inode e di blocchi sia sul client che sul server, con read-ahead (lettura dei blocchi anticipata) e write-behind (scrittura asincrona posticipata) $\Rightarrow$ problemi di consistenza
- Il protocollo NFS non fornisce controlli di concorrenza (accesso esclusivo, locking, ...). Vengono implementati da un altro server (lockd)

Architettura NFS