Università di Udine

Laurea in Biotecnologie — A.A. 2006/07

Trasparenze del Corso di *Sistemi Operativi*

Marino Miculan

Università di Udine

Introduzione

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

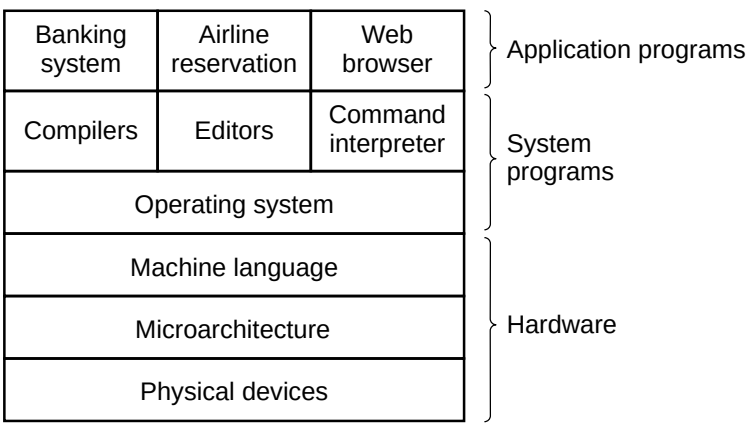
Cosa è un sistema operativo?

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, videgiochi, programmi di produttività personale,. . .)

4. Utenti (persone, macchine, altri calcolatori)

Cosa è un sistema operativo? (2)

Non c'è una definizione completa ed esauriente: dipende dai contesti.

- Un programma che agisce come intermediario tra l'utente/programmatore e l'hardware del calcolatore.

- Assegnatore di risorse
  Gestisce ed alloca efficientemente le risorse finite della macchina.

- Programma di controllo
  Controlla l'esecuzione dei programmi e le operazioni sulle risorse del sistema di calcolo.
  Condivisione corretta rispetto al tempo e rispetto allo spazio

Obiettivi di un sistema operativo

Realizzare una *macchina astratta*: implementare funzionalità di alto livello, nascondendo dettagli di basso livello.

- Eseguire programmi utente e rendere più facile la soluzione dei problemi dell'utente

- Rendere il sistema di calcolo più facile da utilizzare e programmare

- Utilizzare l'hardware del calcolatore in modo sicuro ed efficiente

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

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 (Non Uniform Memory Access) o NORMA (No Remote Memory Access) (migliaia di CPU)
- Job di calcolo intensivo (es. biologia)
- 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 per Personal Computer

- *Personal computers* – sistemi di calcolo dedicati ad un singolo utente, spesso non esperto
- Interfaccia utente evoluta (GUI)
- Grande varietà di dispositivi di I/O (tastiere, mouse, schermi, piccole stampanti) e molto variabile
- Prioritaria la facilità d'uso, reattività e flessibilità rispetto alle prestazioni e allo sfruttamento delle risorse
- Spesso non sono necessari avanzati sistemi di protezione.

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

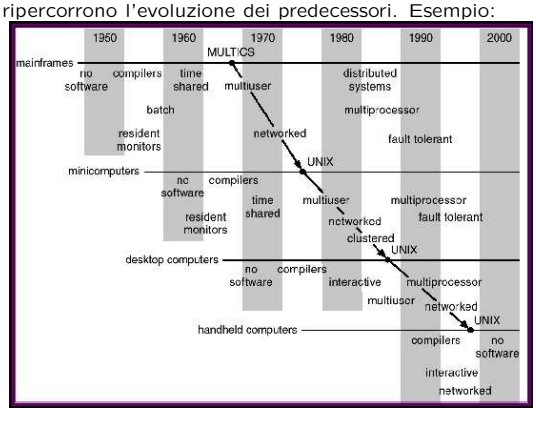
- Per calcolatori palmari (PDA), cellulari, ma anche televisori, forni a microonde, lavatrici, etc.
- Hanno spesso caratteristiche di real-time
- Limitate risorse hardware
- esempio: PalmOS, Epoc, PocketPC, QNX.

Sistemi operativi per smart card

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

"L'ontogenesi ricapitola la filogenesi"

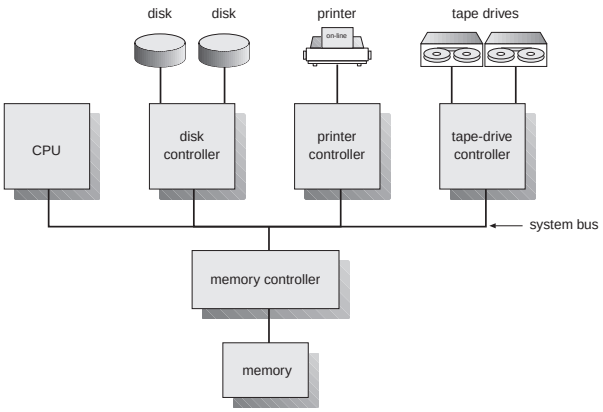
L'hardware e il software (tra cui i sistemi operativi) in ogni nuova classe di calcolatori ripercorrono l'evoluzione dei predecessori. Esempio:



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

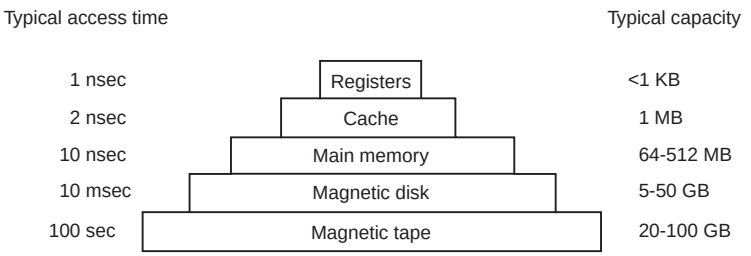


Struttura della Memoria

- 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)

I sistemi di memorizzazione sono organizzati gerarchicamente, secondo

- velocità
- costo
- volatilità



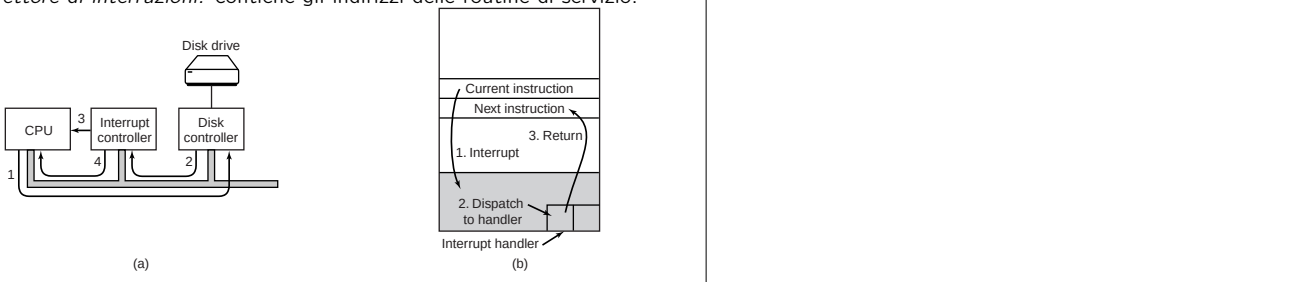
Caching – duplicare i dati più frequentemente usati di una memoria, in una memoria più veloce. La memoria principale può essere vista come una cache per la memoria secondaria.

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 muove 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*.

Schema comune degli Interrupt

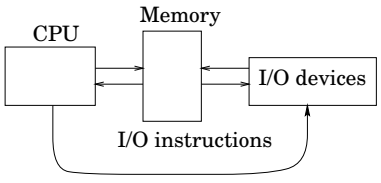
- Gli interrupt trasferiscono il controllo alla routine di servizio dell'interrupt. Due modi:
  - *polling*
  - *vettore di interruzioni:* contiene gli indirizzi delle routine di servizio.



Gestione dell'interrupt

- L'hardware salva l'indirizzo dell'istruzione interrotta (p.e., sullo stack).
- Il S.O. preserva lo stato della CPU salvando registri e program counter in apposite strutture dati.
- Per ogni tipo di interrupt, uno specifico segmento di codice determina cosa deve essere fatto.
- Terminata la gestione dell'interrupt, lo stato della CPU viene riesumato e l'esecuzione del codice interrotto viene ripresa.
- 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 (istruzioni `TRAP`, `SVC`).
- Un sistema operativo è *guidato da interrupt*

Direct Memory Access (DMA)



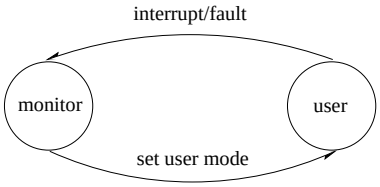
- Usata per dispositivi in grado di trasferire dati a velocità prossime a quelle della memoria
- I controller trasferiscono blocchi di dati dal buffer locale direttamente alla memoria, senza intervento della CPU.
- Viene generato un solo interrupt per blocco, invece di uno per ogni byte trasferito.

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

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"

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.

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

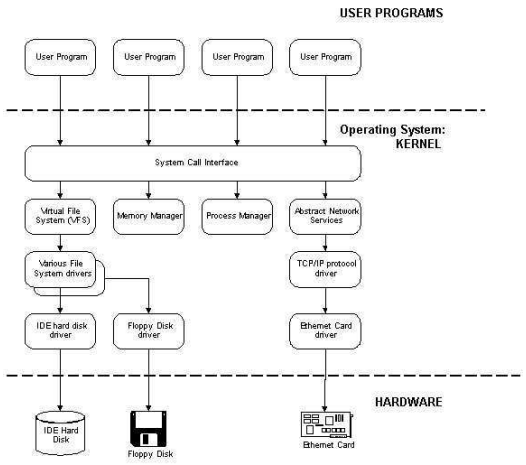
Programmi di sistema

- 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.

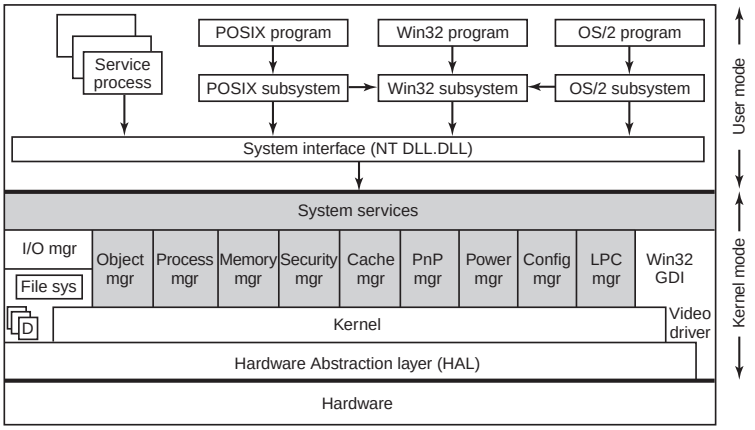
Componenti comuni dei sistemi operativi

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

Struttura dei Sistemi Operativi – Stratificazione di Linux



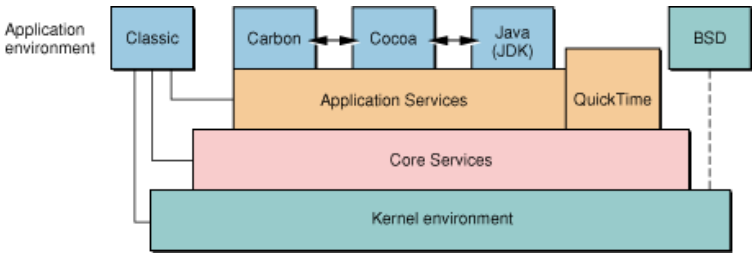
Stratificazione più microkernel: Windows 2000



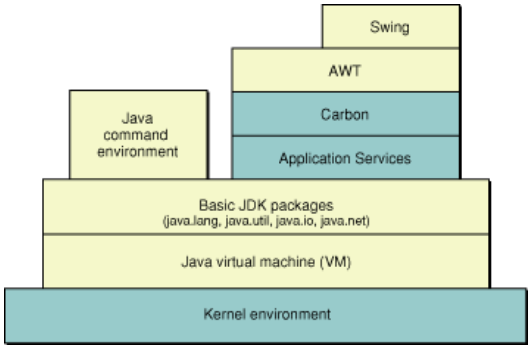
Soluzione stratificata/microkernel: Mac OS X

- 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



Processi e Thread

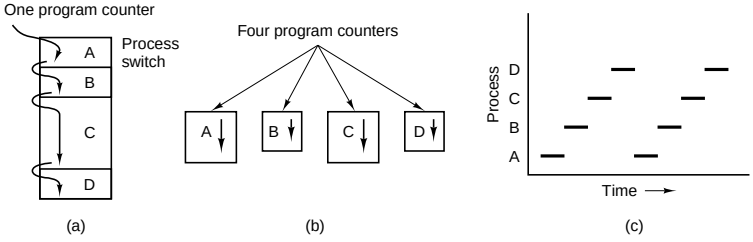
- Concetto di processo
- Operazioni sui processi
- Stati dei processi
- Threads
- Schedulazione dei processi

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

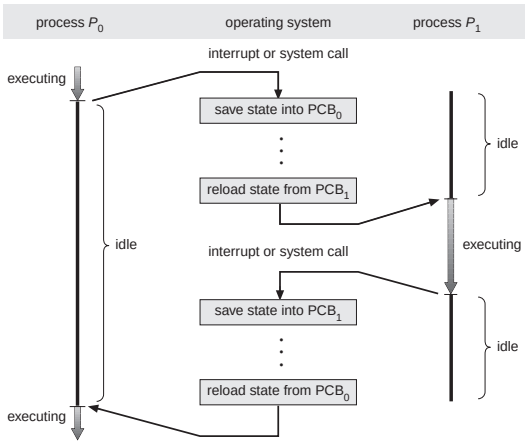
Multiprogrammazione

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



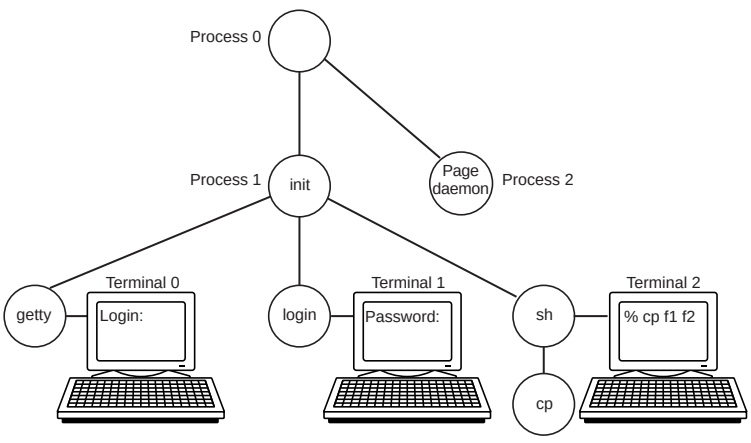
Switch di Contesto

- 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*.



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.

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

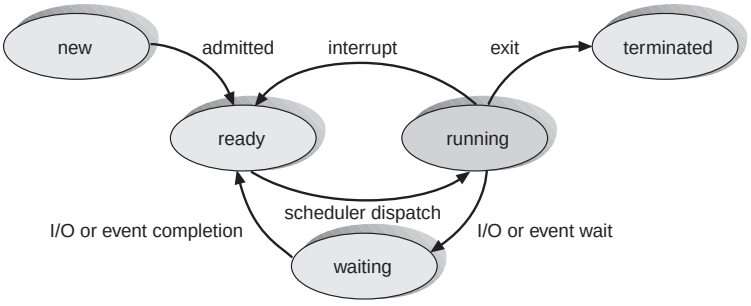
ready: il processo attende di essere assegnato ad un processore

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



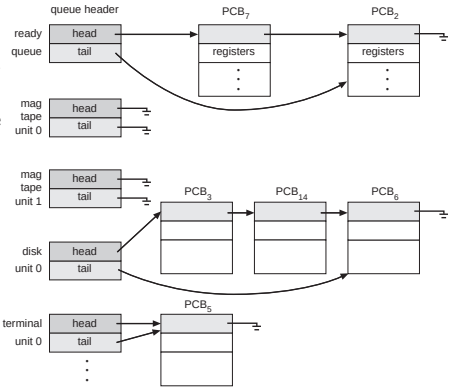
Process Control Block (PCB)

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

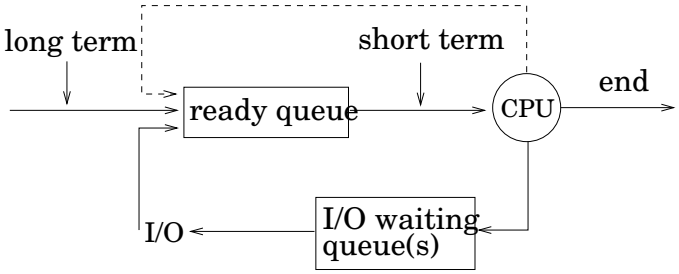
Code di scheduling dei processi

- 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.
- 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.

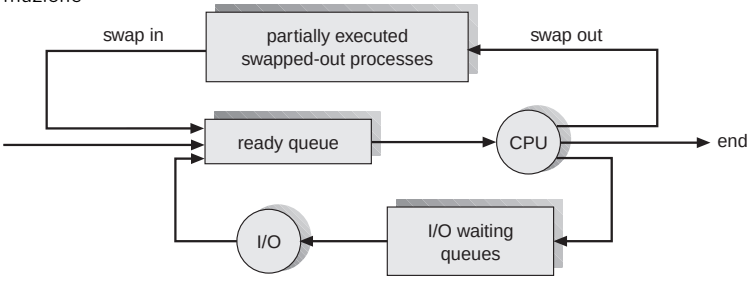


Gli Scheduler (Cont.)

- 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.)

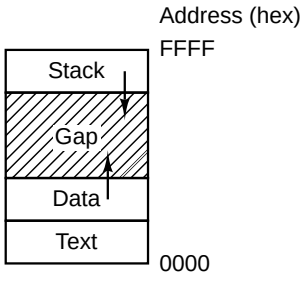
Alcuni sistemi hanno anche lo *scheduler di medio termine* (o *swap scheduler*) sospende temporaneamente i processi per abbassare il livello di multiprogrammazione



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.

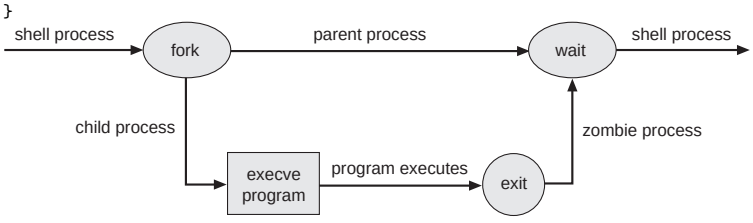


Creazione di un processo: la chiamata fork(3)

```
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);
 }
}
```



Dai processi...

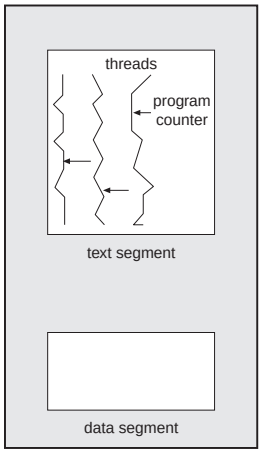
I processi finora studiati incorporano due caratteristiche:

- 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,...

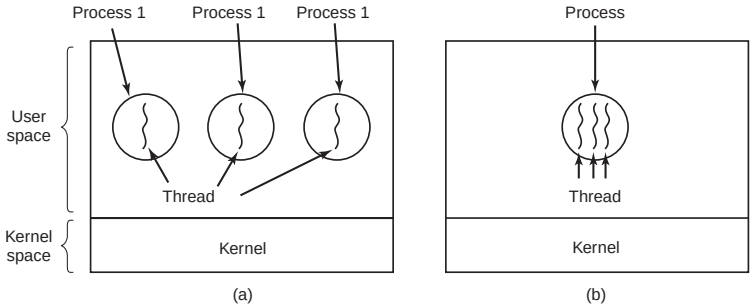
Queste due componenti sono in realtà *indipendenti*

. . . ai thread

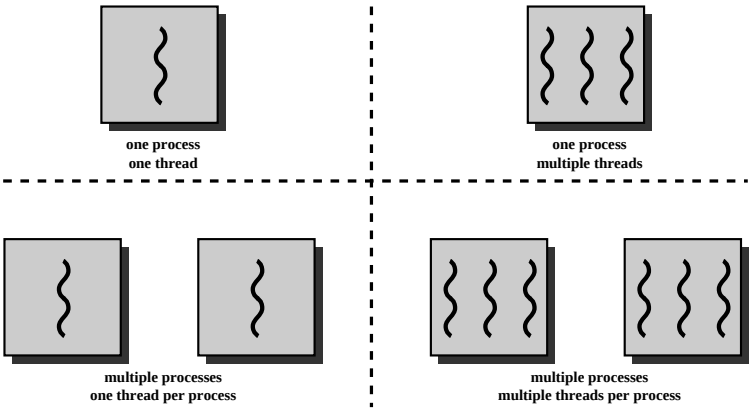
- Un *thread* (o *processo leggero*, *lightweight process*) è una unità di esecuzione:
  - program counter, insieme registri
  - stack del processore
  - stato di esecuzione
- Un thread condivide con i thread suoi pari una unità di allocazione risorse:
  - il codice eseguibile
  - i dati
  - le risorse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



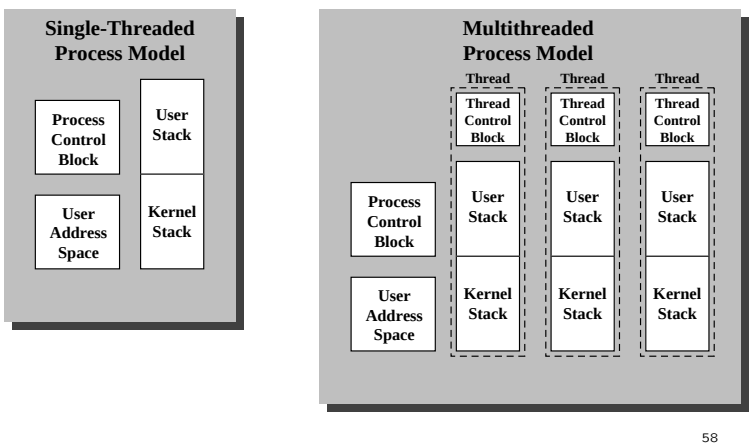
Esempi di thread



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)

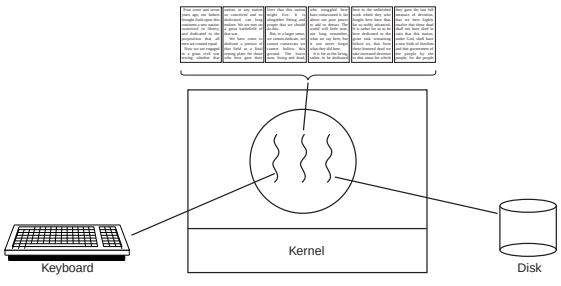
Condivisione di risorse tra thread (Cont.)

- 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

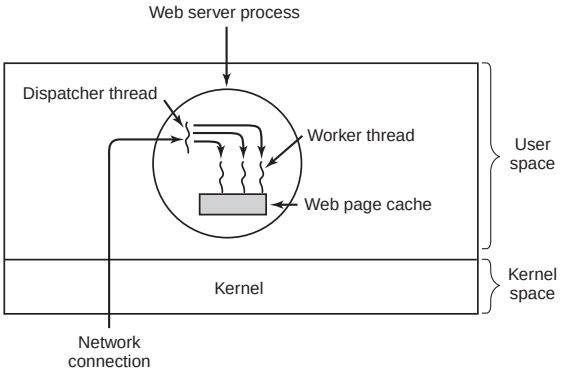
Lavoro foreground/background: mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background. Spreadsheets (ricalcolo automatico), word processor (reimpaginazione, controllo ortografico,. . .)

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

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()`.

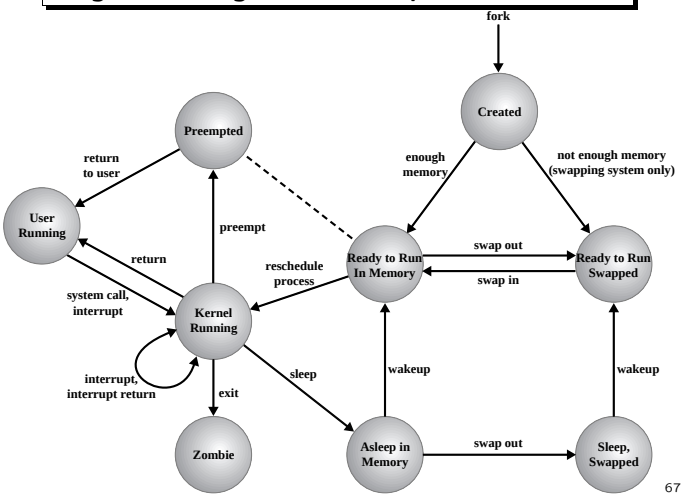
Permette di implementare i thread a livello kernel.

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

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

Processi e Thread di Windows 2000

Nel gergo Windows:

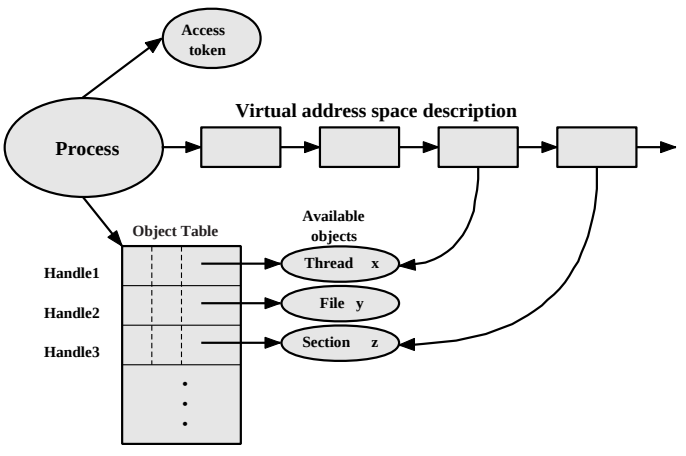
Job: collezione di processi che condividono quota e limiti

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.

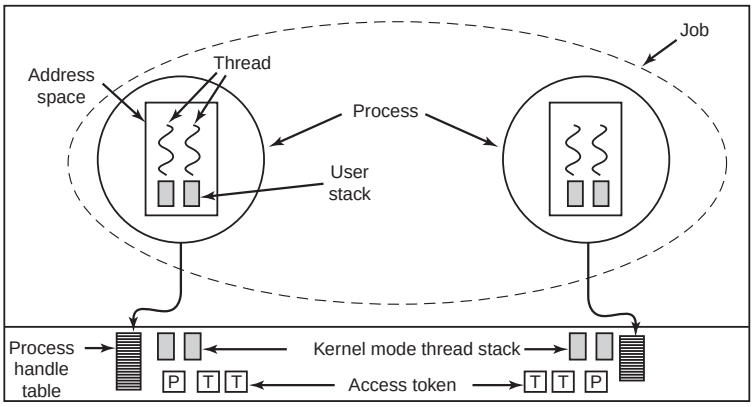
Thread: entità schedulata dal kernel. Alterna il modo user e modo kernel. Doppio stack. Creato con `CreateThread`.

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

Struttura di un processo in Windows 2000

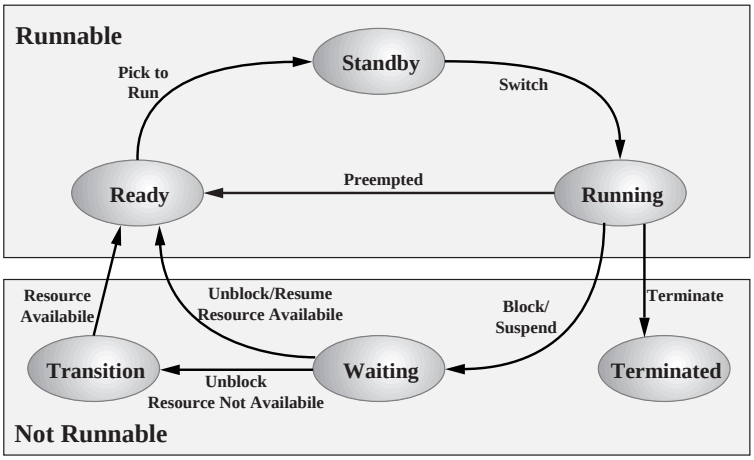


Job, processi e thread in Windows 2000



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 (analogo di "swapped, ready")
- Terminated: terminato, ma non ancora cancellabile (o riattivabile)



Cooperazione tra Processi

- Principi
- Il problema della sezione critica: le *race condition*
- Semafori
- Monitor
- Scambio di messaggi
- Barriere

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)

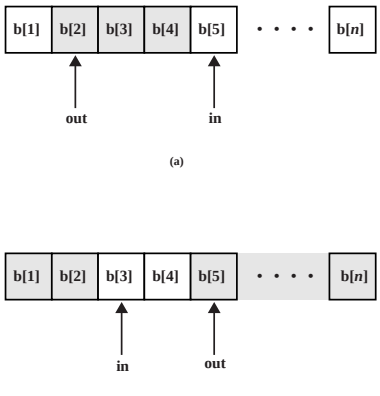
Questioni da considerare:

- 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.



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

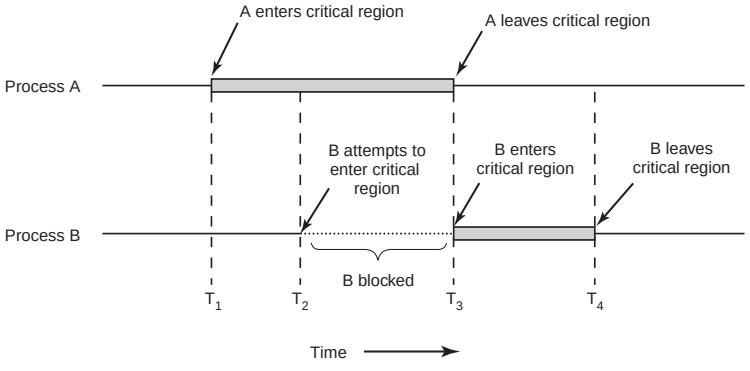
Race condition: più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi.

- 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 (1981)

```
#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

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 può essere bloccato all'infinito 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.

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

⋮	⋮
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:

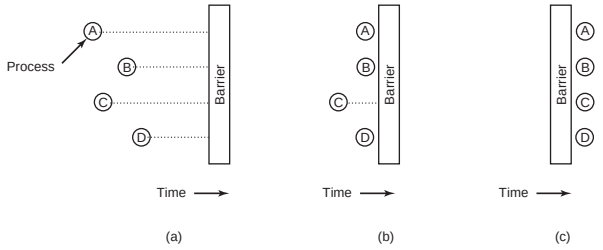
```
down(S): S.value := S.value − 1;
         if S.value < 0
           then begin
                    aggiungi questo processo a S.L;
                    sleep();
                end;
up(S):   S.value := S.value + 1;
         if S.value ≤ 0
           then begin
                    togli un processo 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)

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).



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;
```


Produttore-consumatore con monitor

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

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


Monitor (cont.)

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

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

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 */
     }
}
```

Esempio di problema: Lettori-Scrittori

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

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

Implementazione con i semafori:

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


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

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


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

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*

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*.

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

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[...];
}
```

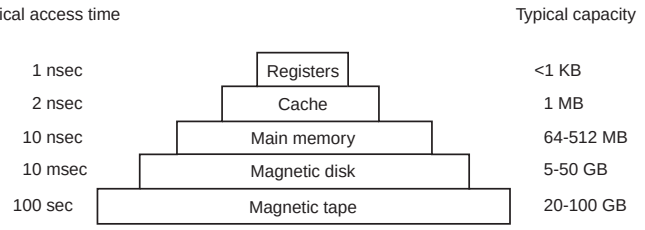
- `size`: dimensione del messaggio in `mtext`
- `type`: tipo di messaggio richiesto; 0 = qualsiasi tipo
- `flag`: modo di spedizione/ricezione: `0` = comportamento normale (bloccante); `IPC_NOWAIT` = non bloccante

Gestione della Memoria

- Fondamenti
- Spazio indirizzi logico vs. fisico
- Allocazione contigua
  - partizionamento fisso
  - partizionamento dinamico
- Allocazione non contigua
  - 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*

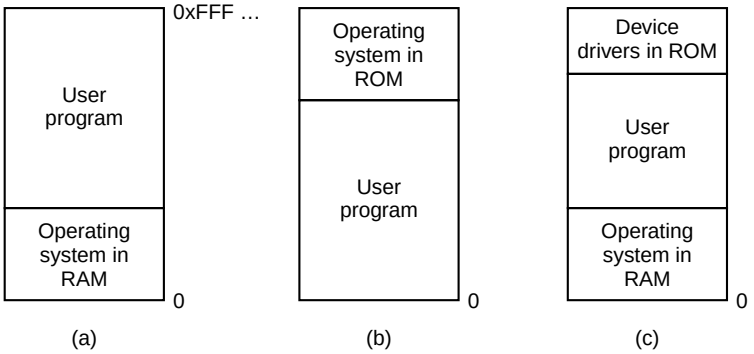


Gestione della memoria: Fondamenti

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



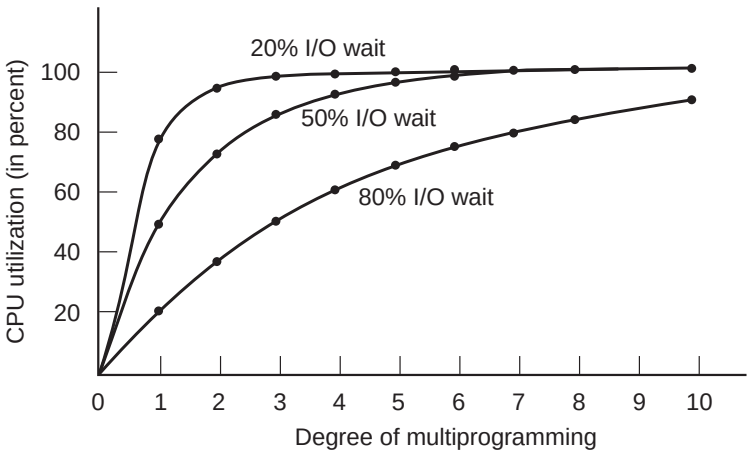
Un solo programma per volta (oltre al sistema operativo). (c) il caso del DOS.

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%)



Multiprogrammazione (cont)

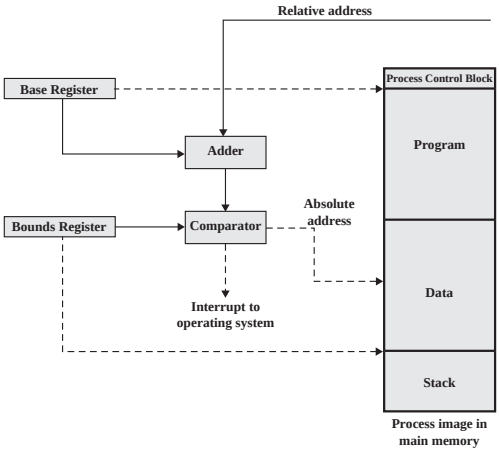
- Ogni programma deve essere portato in memoria e posto nello spazio indirizzi di un processo, per poter essere eseguito.
- *Coda in input:* l'insieme dei programmi su disco in attesa di essere portati in memoria per essere eseguiti.
- La selezione è fatta dallo scheduler di lungo termine (se c'è).
-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

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.

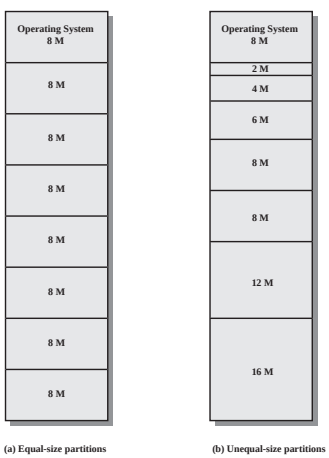


Allocazione contigua

- 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.

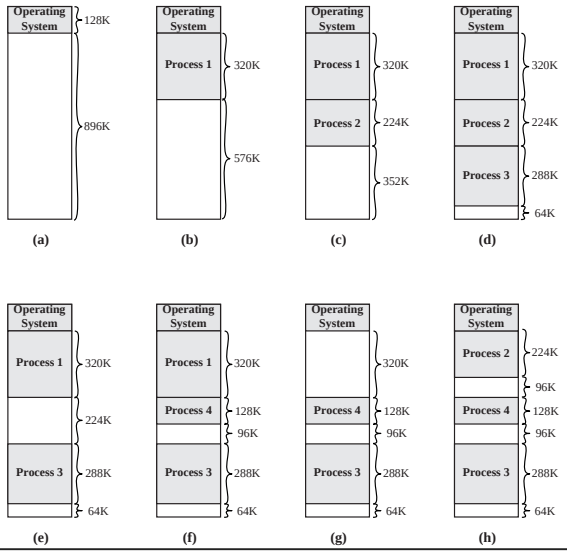
Allocazione contigua: partizionamento statico

- 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: 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.



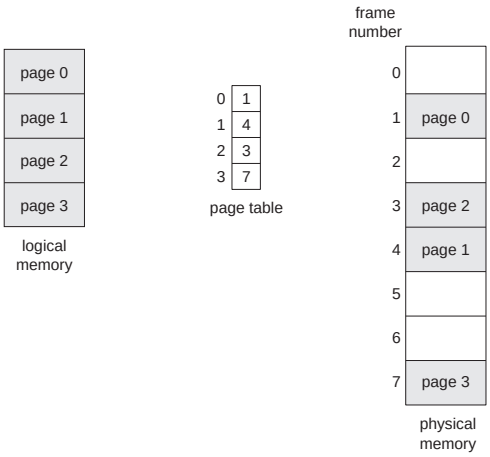
Allocazione contigua: partizionamento dinamico (cont.)

- 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 non contigua: Paginazione

- Lo spazio logico di un processo può essere allocato in modo 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

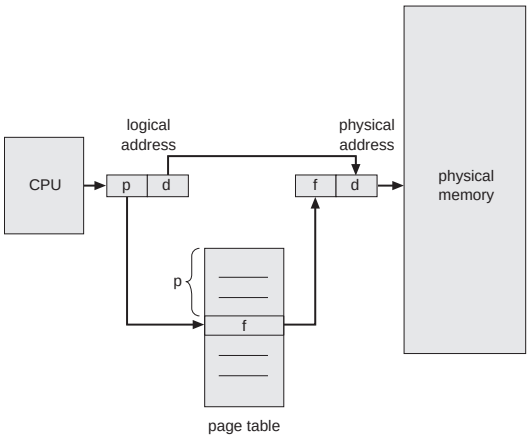
Esempio di paginazione



Schema di traduzione degli indirizzi

L'indirizzo generato dalla CPU viene diviso in

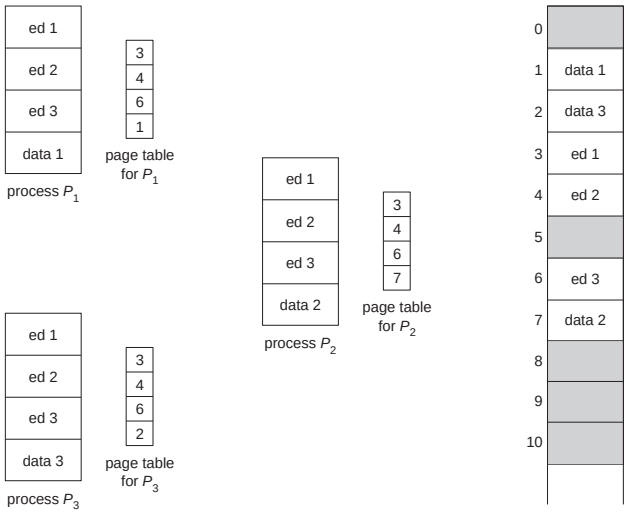
- *Numero di pagina* p*:* usato come indice in una *page table* che contiene il numero del frame contenente la pagina p.
- *Offset di pagina* d*:* combinato con il numero di frame fornisce l'indirizzo fisico da inviare alla memoria.



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



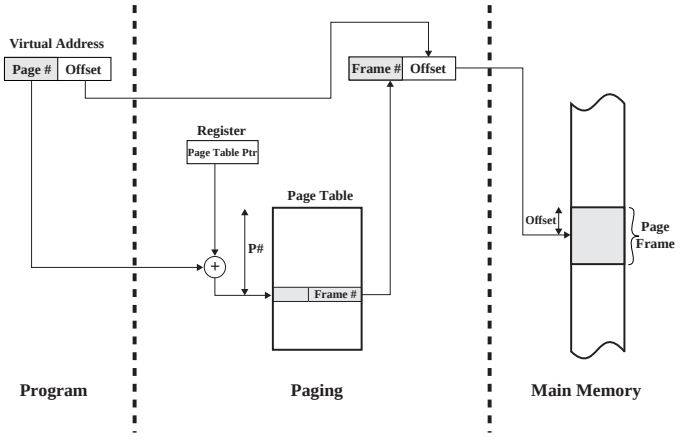
Paginazione: protezione

- 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)

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) ⇒ 2M in registri.
- La page table viene tenuta in memoria principale
  - *Page-table base register (PTBR)* punta all'inizio della page table
  - *Page-table length register (PTLR)* indica il numero di entry della page table

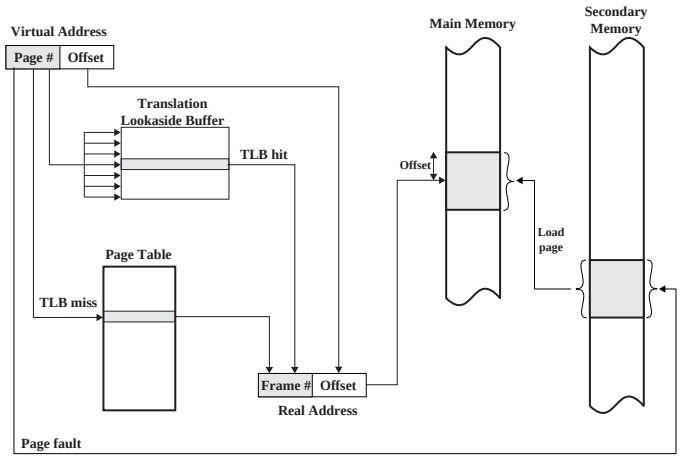
Paginazione con page table in memoria



Paginazione con page table in memoria (cont.)

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

Registri Associativi (TLB)



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

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

Tempo effettivo di accesso con TLB

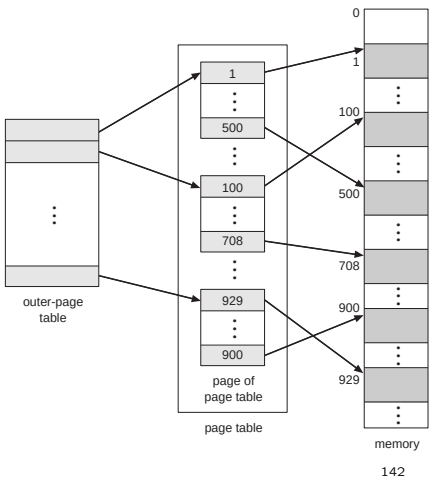
- ϵ = tempo del lookup associativo
- t = tempo della memoria
- α = *Hit ratio*: percentuale dei page # reperiti nel TLB (dipende dalla grandezza del TLB, dalla natura del programma. . .)

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

- 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

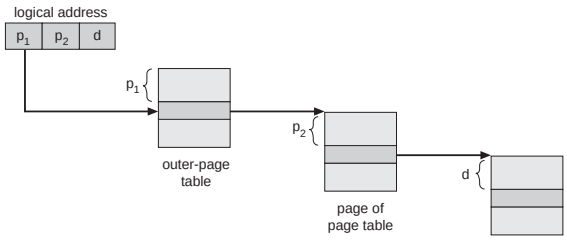
Per ridurre l'occupazione della page table, si pagina la page table stessa.
Solo le pagine effettivamente usate sono allocate in memoria RAM.



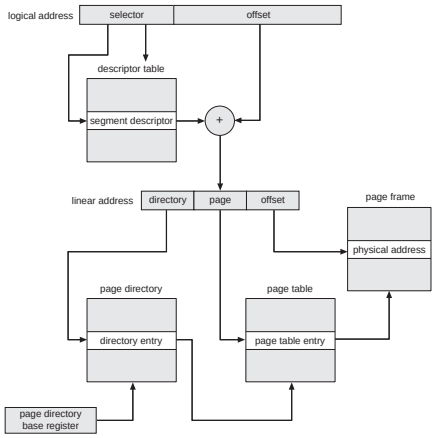
Esempio di paginazione a due livelli

Un indirizzo logico (a 32 bit con pagine da 4K) è diviso in

- un numero di pagina di 20 bit, diviso in
  - un *directory number* di 10 bit
  - un *page offset* di 10 bit.
- un offset di 12 bit



Segmentazione con paginazione a 2 livelli: la IA32



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*)

Molti vantaggi sia per gli utenti che per il sistema

- Lo spazio logico può essere molto più grande di quello fisico (astrazione).
- Meno consumo di memoria ⇒ più processi in esecuzione ⇒ maggiore multiprogrammazione ⇒ maggiore sfruttamento ed efficienza
- 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.

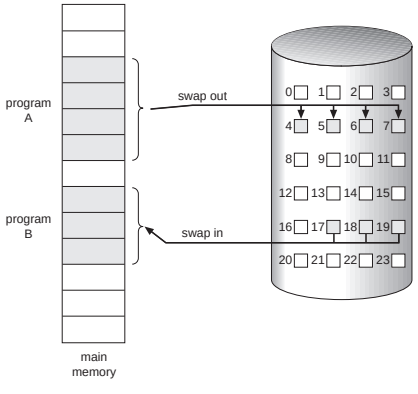
Paginazione su richiesta

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

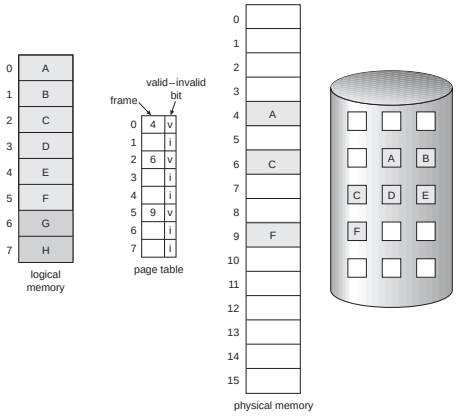
  Pager: processo che implementa una politica di gestione delle pagine dei processi (caricamento/scaricamento).

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*)

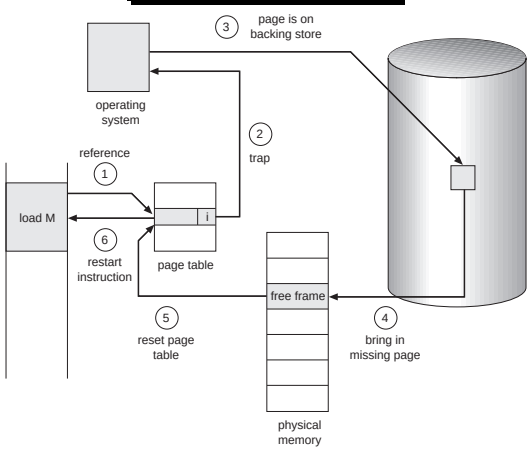
  – Caricare la pagina richiesta nel frame così liberato (*swap in*)

  – 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

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.

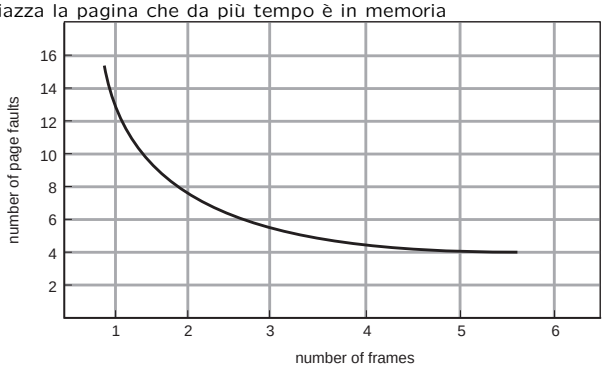
Algoritmi di rimpiazzamento delle pagine

- 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.

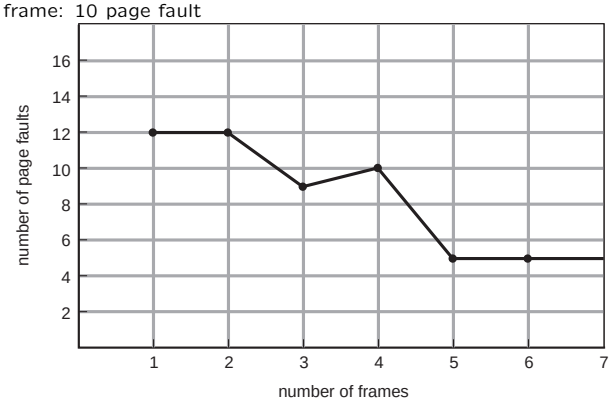
Algoritmo First-In-First-Out (FIFO)

- 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

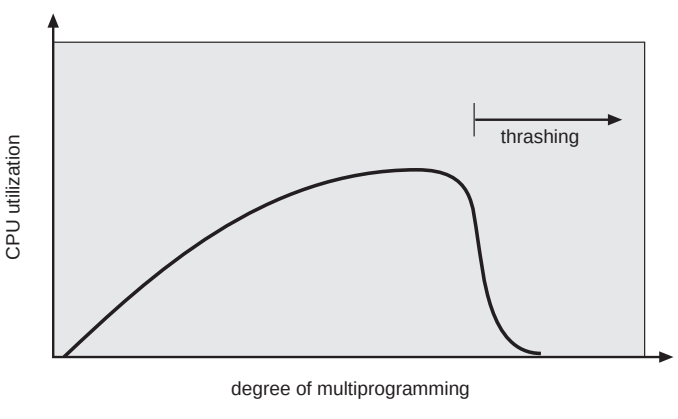
Algoritmo Least Recently Used (LRU)

- 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
- Implementato (con approssimazioni) nella maggior parte dei S.O.

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

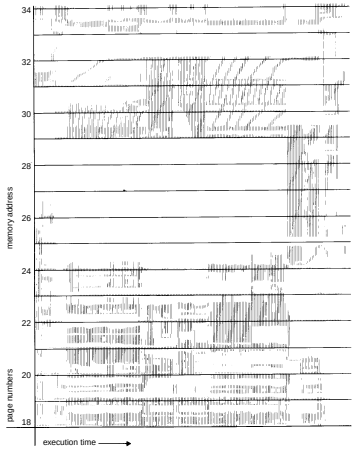
- Il thrashing del sistema avviene quando la memoria fisica è inferiore somma delle località dei processi in esecuzione. Può essere causato da un processo che si espande e in presenza di rimpiazzamento globale.



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
- Il sistema operativo "cerca" di mantenere in memoria la località di ogni processo



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*)

Sulla dimensione delle pagine

- La dimensione della pagina è solitamente imposta dall'architettura HW. 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

Al programmatore non far sapere...

- 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)

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 e sono accessibili unitariamente
- 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.
- 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.

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 info...

Molte di 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, con appositi file di configurazione, come 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
```


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

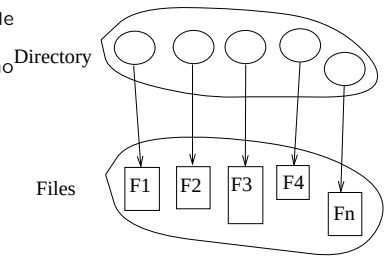
Troncamento: azzerare la lunghezza di un file, mantenendo tutti gli altri attributi

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

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

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

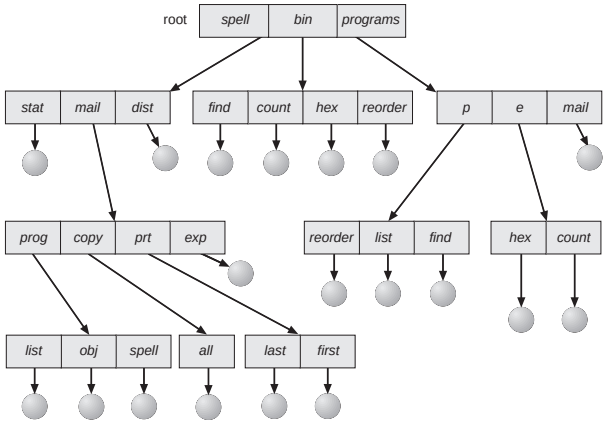


Organizzazione logica delle directory

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: ad albero



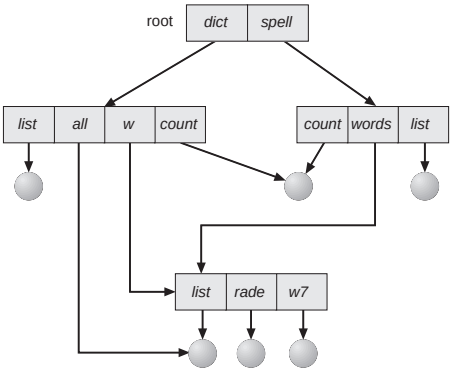
Directory ad albero (cont.)

- 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

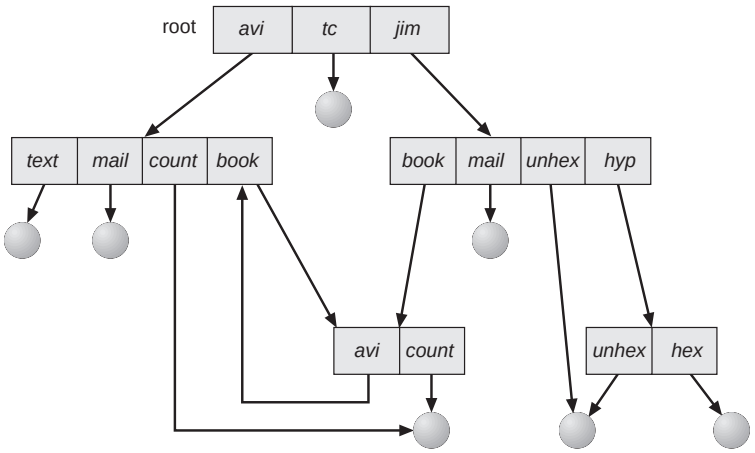
Directory a grafo aciclico (DAG)

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



Directory a grafo (cont.)

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

object / domain	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 (cioè quasi vuota): 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

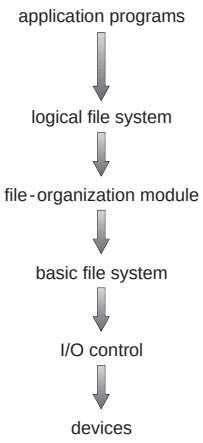
- Ogni processo possiede UID e GID, con i quali si verifica l'accesso

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.

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!)

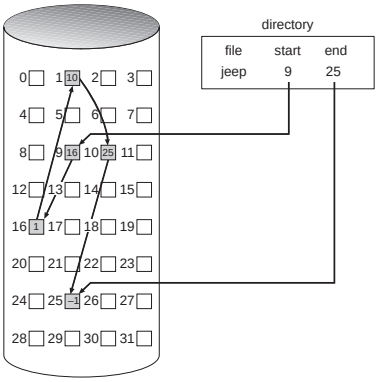
Allocazione contigua

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

Allocazione concatenata

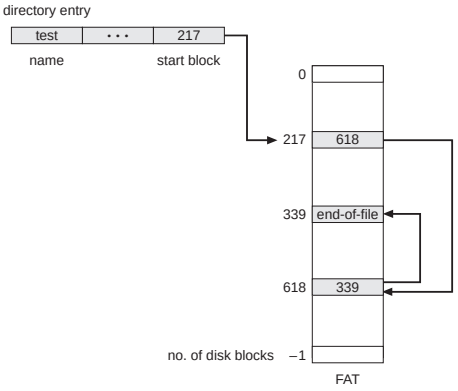
Ogni file è una linked list di blocchi, che possono essere collocati dovunque



- 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): bisogna scorrere la lista

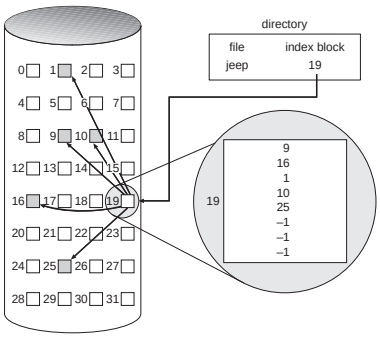
Allocazione concatenata (cont.)

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



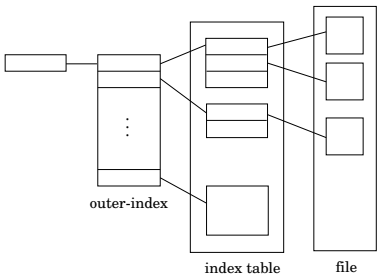
Allocazione indicizzata

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



- Supporta accesso diretto
- Allocazione dinamica senza frammentazione esterna
- Traduzione: serve una lettura in più per accedere all'indice

Allocazione indicizzata (cont.)

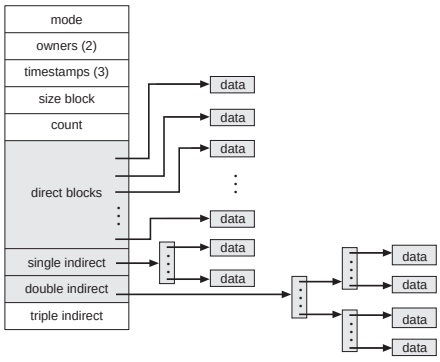


- Variante: indice a due (o più) livelli
- Dim. massima: con blocchi da 4K, si arriva a $(4096/4)^2 = 2^{20}$ blocchi $= 2^{32}B = 4GB$.
- Traduzione dellla posizione: servono due (o più) lettura in più per accedere ai vari livelli di indici

Unix: Inodes

Un file in Unix è rappresentato da un *inode* (nodo indice), che sono allocati in numero finito alla creazione del file system. Contiene:

modo di accesso e tipo del file
UID e GID del possessore
Dimensione del file in byte
Timestamp di ultimo accesso, di ultima modifica, di ultimo modifica dell'inode
Numero di link hard che puntano a questo inode
Blocchi diretti: puntatori ai primi 10–12 blocchi del file
Primo indiretto: indirizzo del blocco indice a 1 livello
Secondo indiretto: indirizzo del blocco indice a 2 livelli
Terzo indiretto: indirizzo del blocco indice a 3 livelli



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.

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!

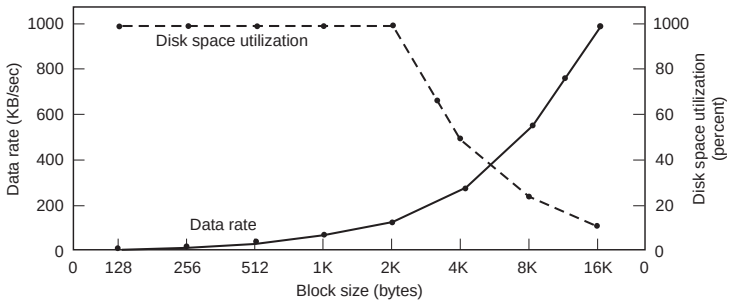
Sulla dimensione dei blocchi

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, NTFS); sotto Windows/DOS 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

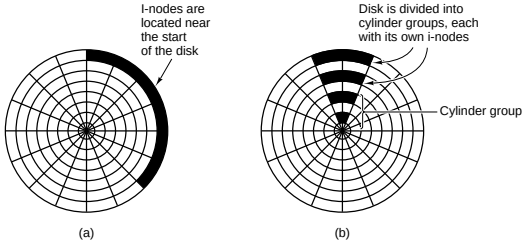
- sul controller: usato come *buffer di traccia* per ridurre la latenza a 0 (quasi)
- (gran) parte della memoria principale, prelevando pagine dalla free list. Può arrivare a riempire tutta la memoria RAM: "un byte non usato è un byte sprecato".

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).

Migliorare le performance: ridurre il movimento della testina

- 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

- I dispositivi di memoria di massa hanno un MTBF relativamente breve
- Inoltre i crash di sistema possono essere causa di perdita di informazioni in cache non ancora trasferite al supporto magnetico.
- Due tipi di affidabilità:
  - *Affidabilità dei dati:* avere la certezza che i dati salvati possano venir recuperati.
  - *Affidabilità dei metadati:* garantire che i metadati non vadano perduti/alterati (struttura del file system, bitmap dei blocchi liberi, directory, inodes. . .).
- Perdere dei dati è costoso; perdere dei metadati è critico: può comportare la perdita della *consistenza del file system* (spesso irreparabile e molto costoso).

Affidabilità del dati

Possibili soluzioni per aumentare l'affidabilità dei dati

- 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.

- Aumentare l'affidabilità dei dispositivi (es. RAID).

RAID

- RAID = Redundant Array of Inexpensive/Independent Disks: implementa affidabilità del sistema memorizzando informazione ridondante.
- La ridondanza viene gestita dal controller (RAID *hardware*) — o molto spesso dal driver (RAID *software*).
- Diversi *livelli* (organizzazioni), a seconda del tipo di ridondanza

  0: *striping*: i dati vengono "affettati" e parallelizzati. Alte performance, ma non c'è ridondanza.

  1: *Mirroring* o *shadowing*: duplicato di interi dischi. Eccellente resistenza ai crash, basse performance in scrittura

  5: *Block interleaved parity:* come lo striping, ma un disco a turno per ogni stripe viene dedicato a contenere codici Hamming del resto della stripe. Alta resistenza, discrete performance (in caso di aggiornamento di un settore, bisogna ricalcolare la parità)

Consistenza del file system

- Alcuni blocchi contengono informazioni critiche sul file system (specialmente quelli contenenti metadati)
- Per motivi di efficienza, questi blocchi critici non sono sempre sincronizzati (a causa delle cache)
- Consistenza del file system: in seguito ad un crash, blocchi critici possono contenere informazioni incoerenti, sbagliate e contradditorie.
- Due approcci al problema della consistenza del file system:

  curare le inconsistenze dopo che si sono verificate, con programmi di controllo della consistenza (*scandisk*, *fsck*): usano la ridondanza dei metadati, cercando di risolvere le inconsistenze. Lenti, e non sempre funzionano.

  prevenire l'inconsistenze: i journalled file system.

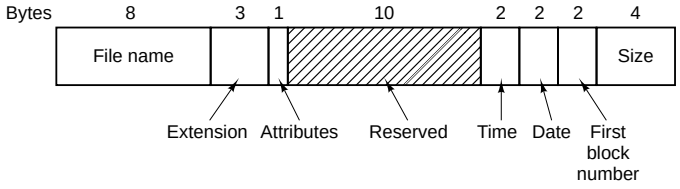
Journalled File System

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

- 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

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

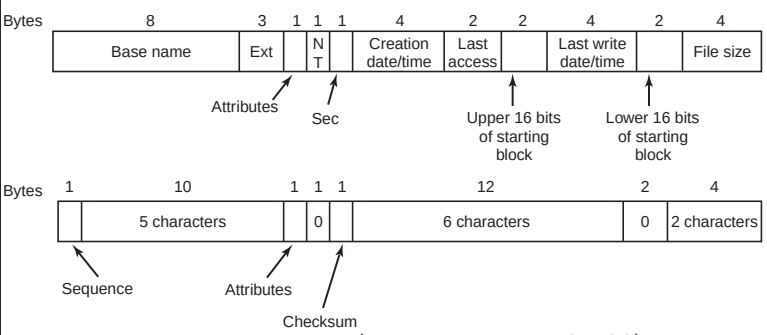
In MS-DOS, tutta la FAT viene caricata in memoria.

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



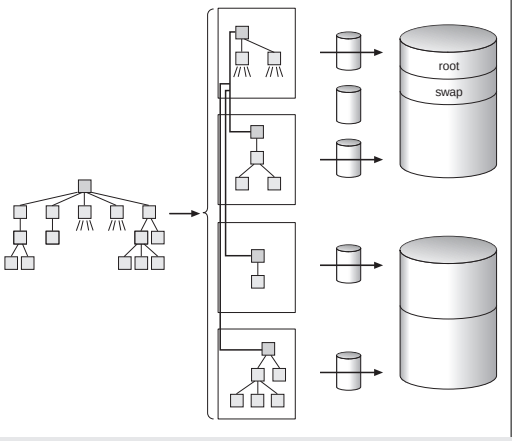
Aove attributes=0x0F (valore invalido per MS-DOS)

Esempio

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

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

UNIX: Il Virtual File System



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

Il Virtual File System (cont.)

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

Il Virtual File System (cont.)

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

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

I File System Fisici di UNIX

- UNIX (Linux in particolare) supporta molti tipi di file system fisici: SYSV, 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 inode 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)

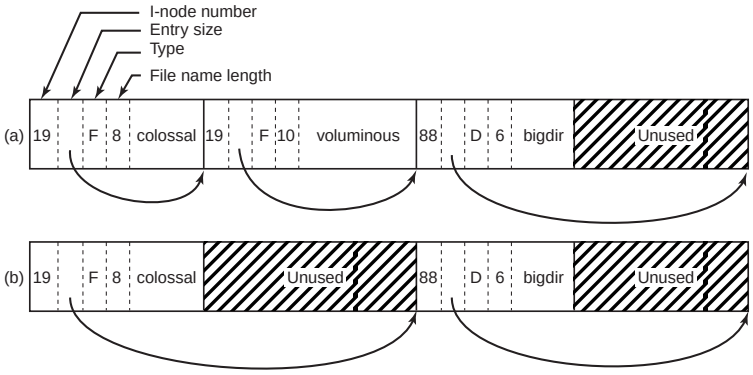
- I timestamp sono in POSIX "epoch": n. di secondi dal 01/01/1970, UTC. (Quindi l'epoca degli Unix a 32 bit dura 2^{31} secondi, ossia fino alle 3:14:07 UTC di martedì 19 gennaio 2038).
- Gli indici indiretti vengono allocati su richiesta
- Accesso più veloce per file piccoli
- N. massimo di blocchi indirizzabile: con blocchi da 4K, puntatori da 4byte

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

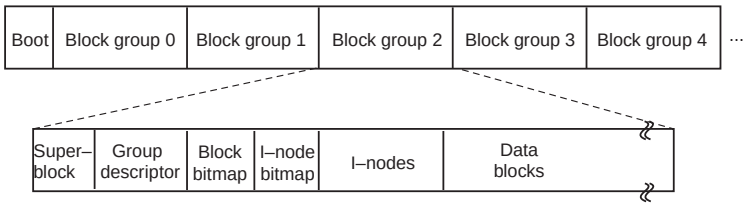
  molto oltre le capacità dei sistemi a 32 bit.

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*)



Esempio di file system fisico: EXT2



- Derivato da UFS, ma con 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.

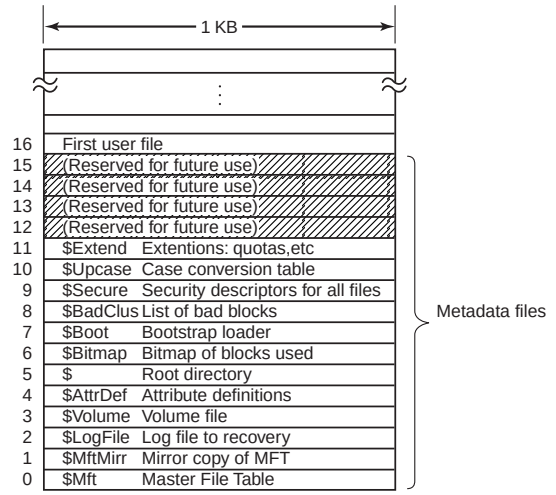
Struttura di NTFS

- 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)

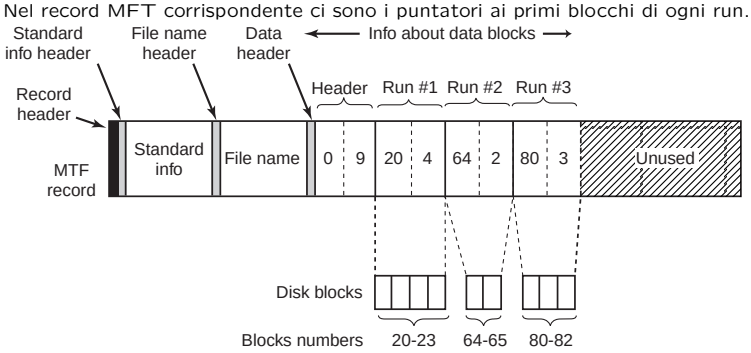
- È un file di record di 1K, ognuno dei quali descrive un file o una directory
- Può essere collocato ovunque, sul disco,e crescere secondo necessità
- Il primo blocco è indicato nel boot block.
- Le prime 16 entry descrivono l'MFT stesso e il volume (analogo al superblock di Unix). Indicano la posizione 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*)



File NTFS non residenti

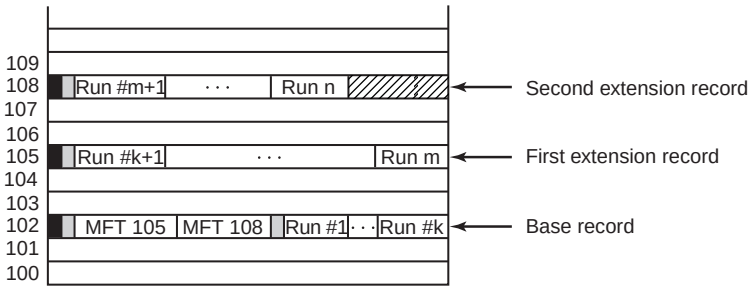
I file non immediati si memorizzano a "run": sequenze di blocchi consecutivi. Nel record MFT corrispondente ci sono i puntatori ai primi blocchi di ogni run.



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.



I passi per usare un disco sotto Unix

La prima volta:	1. Collegarlo alla macchina :–) 2. Partizionamento: *fdisk, cfdisk, druid, . . .* 3. Creazione dei file system: *mkfs, newfs*
A regime:	4. Mounting dei file system: *mount* 5. Buon Lavoro! 6. Unmounting: *umount*
Manutenzione:	7. Verifica e correzione dei problemi: *fsck* 8. Ottimizzazione (molto rara): *tunefs*. La deframmentazione non è necessaria!

Perché partizionare un disco

Gli HD vengono partizionati per diversi motivi:

Usi differenti: diversi file system, area di swap, altri S.O.. . .

Robustezza: se una partizione si danneggia, può essere riformattata senza modificare le altre

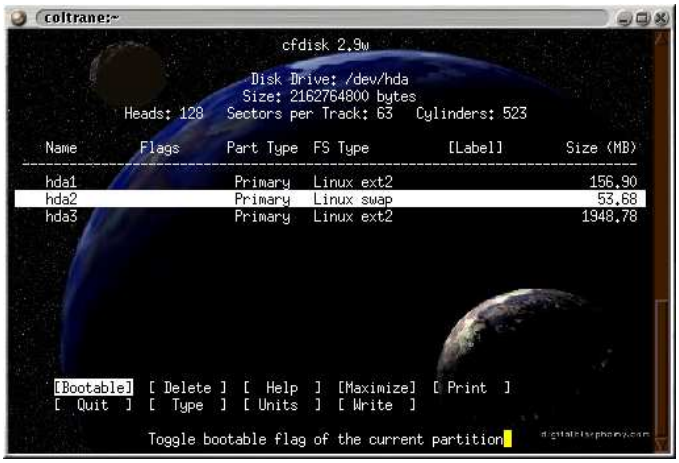
Confinamento: i file non possono superare i confini della partizione

Maggiore velocità al reboot e al backup

Partizionare un disco: fdisk

```
[root@coltrane miculan]# fdisk /dev/hda

Command (m for help): p

Disk /dev/hda: 128 heads, 63 sectors, 523 cylinders
Units = cylinders of 8064 * 512 bytes

   Device Boot    Start       End    Blocks   Id  System
/dev/hda1             1        38    153184+  83  Linux
/dev/hda2            39        51     52416   82  Linux swap
/dev/hda3            52       523   1903104   83  Linux

Command (m for help):
```

Partizionare un disco: cfdisk



Creazione di un file system

```
mkfs [-t fstype] [fs-options] device [size]
```

Diversi file system supportati: UFS, EXT2, SYSV, VFAT, . . .

Esempi di opzioni (per EXT2 e UFS)

-b block-size dimensione del bocco

-f fragment-size dimensione del fragment (solo UFS)

-i bytes-per-inode densità degli inode (default: 4K/inode)

-m reserved-percentage riserva per il sistemista (default: 5%)

-R raid-opts supporto per RAID

Sotto Solaris, si può usare anche *newfs* (più semplice).


```
miculan@coltrane:miculan$ mkfs /dev/fd0
mke2fs 1.12, 9-Jul-98 for EXT2 FS 0.5b, 95/08/09
Linux ext2 filesystem format
Filesystem label=
360 inodes, 1440 blocks
72 blocks (5.00%) reserved for the super user
First data block=1
Block size=1024 (log=0)
Fragment size=1024 (log=0)
1 block group
8192 blocks per group, 8192 fragments per group
360 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```


Montare il file system: mount(1m)

```
mount [-t fstype] [-o opzione] device mountpoint
```

device è un dispositivo a blocchi, *mountpoint* è il path assoluto di una directory (già esistente).
Senza argomenti: mostra i file system montati correntemente:

```
miculan@coltrane:Slides$ mount
/dev/hda1 on / type ext2 (rw)
none on /proc type proc (rw)
/dev/hda5 on /usr type ext2 (rw)
ten:/var/spool/mail on /var/spool/mail
       type nfs (rw,bg,actimeo=0,soft,addr=158.110.144.132)
ten:/user/ospiti on /user/ospiti
       type nfs (rw,addr=158.110.144.132)
```

Il VFS: la tabella /etc/fstab (o /etc/vfstab)

```
#device      mount         FS      mount       dump? fsck
#to mount    point         type    options           pass
#
/dev/hda1    /             ext2    defaults        1 1
/dev/hda3    /usr          ext2    defaults        1 2
/dev/hda2    swap          swap    defaults        0 0
/dev/hdc     /home         ext2    defaults        1 2
/dev/fd0     /mnt/floppy   vfat    noauto,user     0 0
/dev/sda4    /mnt/zip      vfat    noauto,user     0 0
/dev/cdrom   /mnt/cdrom    iso9660 noauto,ro,user  0 0
none         /proc         proc    defaults        0 0
none         /dev/pts      devpts  gid=5,mode=620  0 0
ten:/var/mail /var/spool/mail nfs  defaults        0 0
```

(defaults = rw,suid,dev,exec,auto,nouser,async)

Al boot, viene eseguito *mount -a*: tutti i fs "auto" sono montati.

Mounting (2)

In generale, solo root può montare file system.

Il file system da montare deve essere in uno stato consistente (ossia, deve essere stato smontato correttamente—altrimenti, si veda *fsck*)

La directory coperta diventa inaccessibile (*overlay*).

```
[root@coltrane /mnt]# mount
/dev/hda1 on / type ext2 (rw)
none on /proc type proc (rw)
/dev/hda5 on /usr type ext2 (rw)
[root@coltrane /mnt]# ls -al /mnt/floppy
drwxrwxr-x   2 root     root     1024 Feb  6  1996 .
drwxr-xr-x   6 root     root     1024 Oct  9  1998 ..
-rw-r--r--   1 root     root       15 Apr 13 14:50 pippo
[root@coltrane /mnt]#
```


Mounting (3)

```
[root@coltrane /mnt]# mount /dev/fd0 /mnt/floppy
[root@coltrane /mnt]# mount
/dev/hda1 on / type ext2 (rw)
none on /proc type proc (rw)
/dev/hda5 on /usr type ext2 (rw)
/dev/fd0 on /mnt/floppy type ext2 (rw)
[root@coltrane /mnt]# ls -al floppy
total 14
drwxr-xr-x   3 miculan  ospiti   1024 Apr 13 12:49 .
drwxr-xr-x   6 root     root     1024 Oct  9  1998 ..
drwxr-xr-x   2 root     root    12288 Apr 13 12:49 lost+found
[root@coltrane /mnt]#
```

Ora il file system del disco è integrato nel file system logico.

Smontare un file system

Prima di spegnere la macchina, o di togliere il dispositivo (floppy, zip,...), il suo fs deve essere staccato dal file system logico (pena la perdita di dati!):

`umount <device>|<mountpoint>`

Il file system viene sincronizzato e marcato "consistente", prima di essere staccato. Allo shutdown, viene automaticamente eseguito `umount -a`.

Attenzione: un file system non può essere smontato se è "busy" (attivo), ossia se nel kernel sono presenti inode di tale file system sono. Ciò succede quando

- sono ancora aperti dei file di tale fs, oppure
- uno o più processi hanno la CWD in tale fs.

Controllare il file system: df(1m)

df dà informazioni sull'uso dei file system montati:

```
miculan@coltrane:hfs$ df -k
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hda1              46815   36664     7734     83%   /
/dev/hda5            1930659  834155   996714     46%   /usr
ten:/var/spool/mail   567583  153597   357236     30%   /var/spool/mail
ten:/user/ospiti      962983  699454   215384     76%   /user/ospiti
/dev/fd0                1390      13     1305      1%   /mnt/floppy
miculan@coltrane:hfs$ df -i
Filesystem           Inodes   IUsed   IFree  %IUsed Mounted on
/dev/hda1             12096    3039    9057    25%  /
/dev/hda5            499712   52532  447180    11%  /usr
ten:/var/spool/mail       0       0       0     0%  /var/spool/mail
ten:/user/ospiti          0       0       0     0%  /user/ospiti
/dev/fd0                360      11     349     3%  /mnt/floppy
miculan@coltrane:hfs$
```


Controllare e riparare un file system: fsck

fsck permette di controllare e riparare i file system *non* montati

- a mano, con il comando `fsck [opzioni] <device>`
- automaticamente al boot, se indicato da /etc/fstab o se il fs non è stato smontato correttamente (e.g., crash della macchina).

I principali controlli che fsck esegue sono:

- blocchi allocati da più inodes
- blocchi allocati ma oltre i limiti del file system
- blocchi non allocati né presenti sulla lista libera
- blocchi allocati ma presenti sulla lista libera
- numero errato di link negli inode
- mancanza di "." e ".." nelle directory
- checksum del superblock
- numero di blocchi per gli inode errato
- numero di blocchi/inode liberi errato

Esempi di fsck

```
miculan@coltrane:miculan$ fsck /dev/fd0                ( Linux )
e2fsck 1.12, 9-Jul-98 for EXT2 FS 0.5b, 95/08/09
/dev/fd0: clean, 11/360 files, 63/1440 blocks



root@bodoni$ fsck /dev/rdsk/c0t0d0s3                  ( Solaris )
** /dev/rdsk/c0t0d0s3
** Last Mounted on /export
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
9447 files, 2760313 used, 665249 free (1313 frags, 82992 blocks,
                                    0.0% fragmentation)
```


Riparare file system: fsck

Quando *fsck* trova dei problemi, chiede se deve ripararli. Esempi:

I=1892371: Incorrect link count (fixed) l'inode era puntato da un numero di directory diverso da quello segnato

I=128731: unreferenced inode l'inode non era puntato da nessuna directory, e quindi è stato liberato

Succedono quando il file system è stato smontato malamente (e.g., crash della macchina): lo stato degli inode in memoria non è stato `sync`-ronizzato con quello su disco, e quindi il file viene perduto.

Se il superblock è danneggiato, fsck può sostituirlo con una sua copia:

```
$ fsck -o b=32  /dev/hda
```

Sistemi a più processori

Fatto: 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.

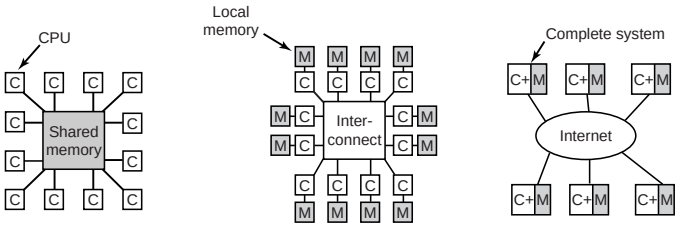
processori strettamente accoppiati :

- *sistemi multiprocessore:* condividono clock e/o memoria; comunicazione attraverso memoria condivisa: UMA (SMP, crossbar, . . .), NUMA (CC-NUMA, NC-NUMA), COMA;
- *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

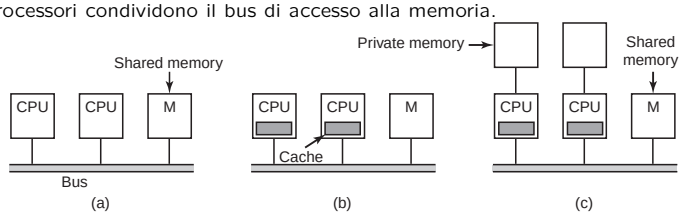


Differenze:

- 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.

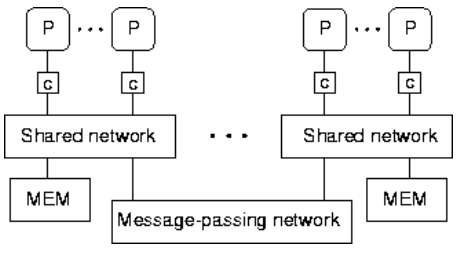
Problemi di coerenza ⇒ uso di cache (bus snooping) e memorie private (necessitano di adeguata gestione da parte del compilatore)

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

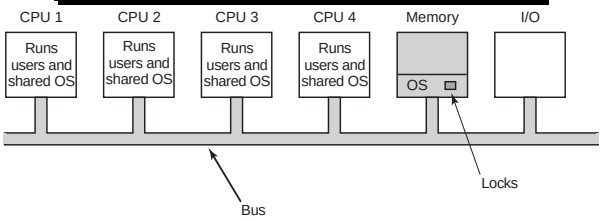
Hardware multiprocessore: NUMA

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

- locale: privata ad ogni CPU/gruppo SMP di CPU
- remota: condivisa tra le CPU; tempo di accesso $2-15$ volte quello locale.
- Reindirizzamento via rete/bus, risolto in hardware (MMU)
- Eventualmente, una cache locale per la memoria remota (CC-NUMA)



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 $n = 16$: $p = 81.5\%$.

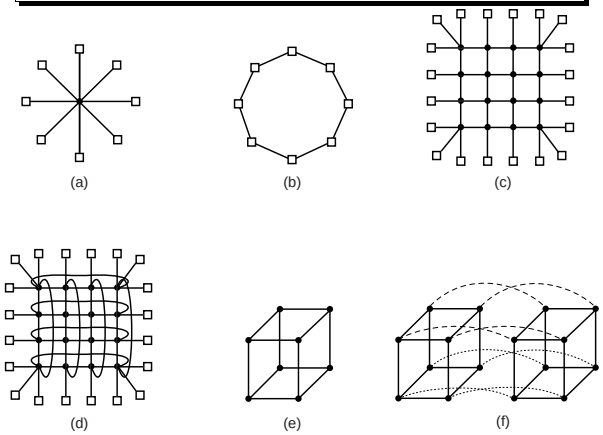
SO Multiprocessori Simmetrici (SMP) (cont.)

- 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

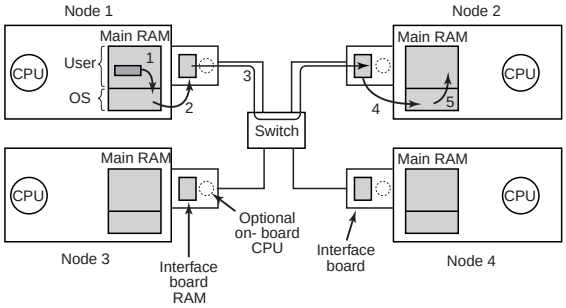
- I multiprocessori sono comodi perché offrono un modello di comunicazione simile a quello tradizionale
- Problemi di costo e scalabilità
- Alternativa: *multicomputer*: calcolatori strettamente accoppiati ma senza memoria condivisa, solo passaggio di messaggi
  - esempio: rete di PC con buone schede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

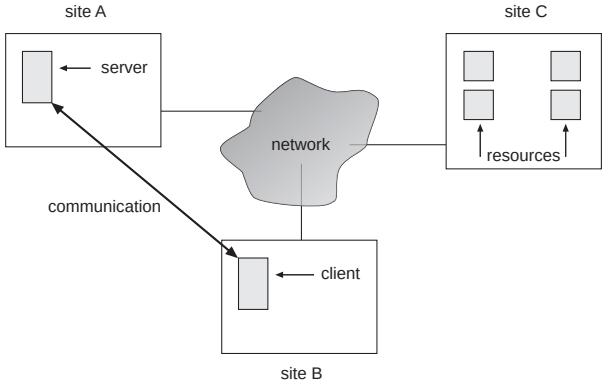
- Il modello base usa comunicazione con passaggio di messaggi
- Il S.O. offre primitive per la comunicazione tra i nodi

```
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 programmatore o il compilatore parallelizza e distribuisce il codice.

Sistemi Distribuiti

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



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

Client, server, protocollo

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, contemporaneamente o in tempi successivi.

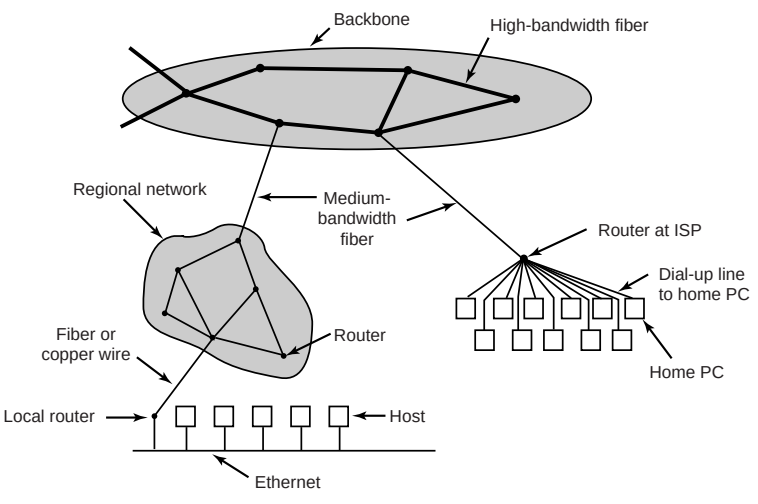
Hardware dei sistemi distribuiti: LAN e WAN

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



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 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

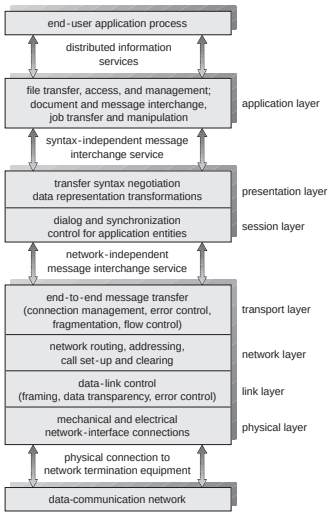
Strato di rete: fornisce le connessioni e instrada i pacchetti nella rete. Include la gestione degli indirizzi degli host e dei percorsi di instradamento.

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.

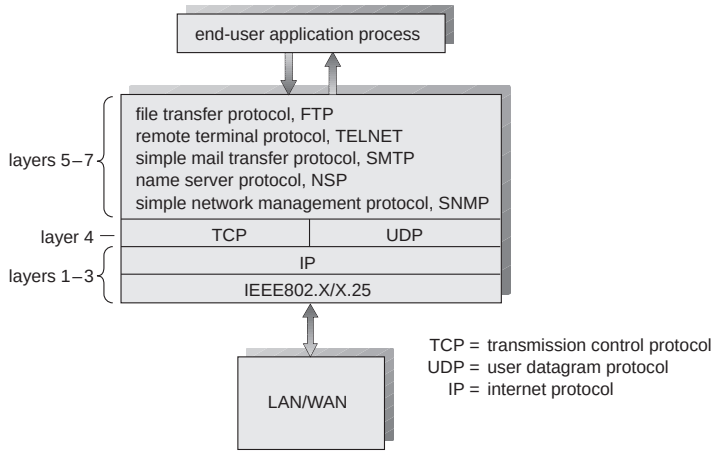
Sessione: implementa sessioni, ossia comunicazioni tra processi

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

ISO reference model	ARPANET reference model	4.2BSD layers	example layering
application	process applications	user programs and libraries	telnet
presentation			
session		sockets	sock_stream
transport	host–host	protocol	TCP
network			IP
data link	network interface	network interfaces	Ethernet driver
hardware			
	network hardware	network hardware	interlan controller

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`.
- 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);
```


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 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)

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

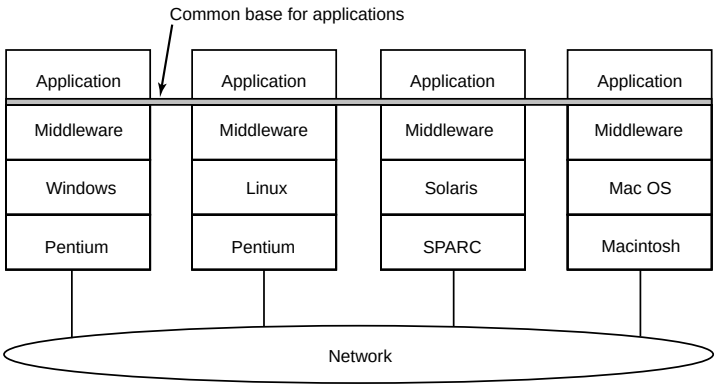
- È una procedura che permette al sistema di riconfigurarsi e continuare il funzionamento
- Se il link tra A e B si è guastato, questa informazione deve essere diramata agli altri siti del sistema, in modo che le tabelle di routing vengano aggiornate
- Se si ritiene che un sito si è guastato, allora 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

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*



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) ⇒ caching su disco locale, adeguato per accessi a tutto il file (i.e., connectionless)
  - accesso remoto: copie di parti di file (NFS, SMB) ⇒ efficiente per pochi accessi, adeguato su reti locali (affidabili)

Migrazione di computazioni

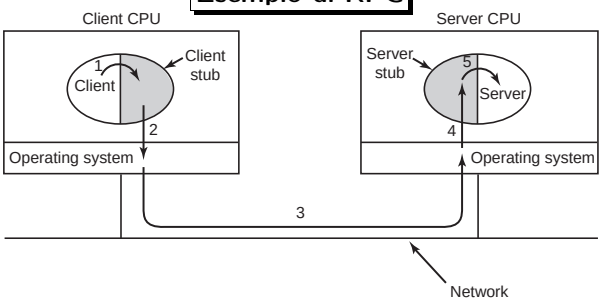
- Quando un processo deve accedere ad un dato, si procede al trasferimento della *computazione* dal client al server. Il calcolo avviene sul server e solo il risultato viene restituito al client.
- Efficiente se trasferire la computazione costa meno dei dati
- client e server rimangono processi *separati*
- 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.

Programmazione con migrazione di computazione: 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.

Servizi distribuiti su 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.

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 eseguita dal demone per conto del client

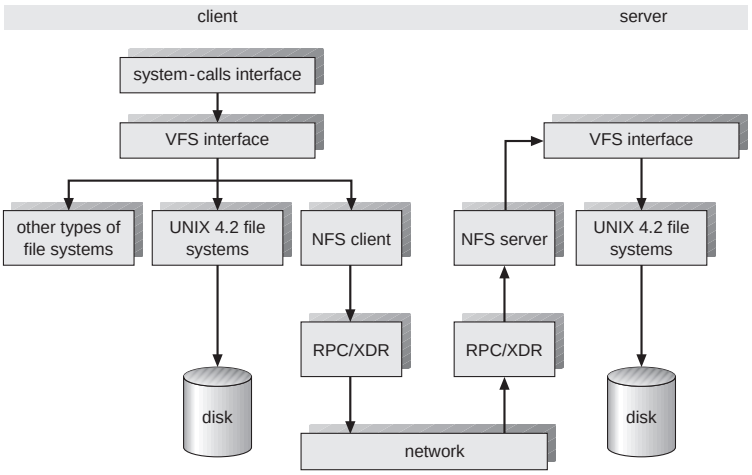
Il Sun Network File System (NFS)

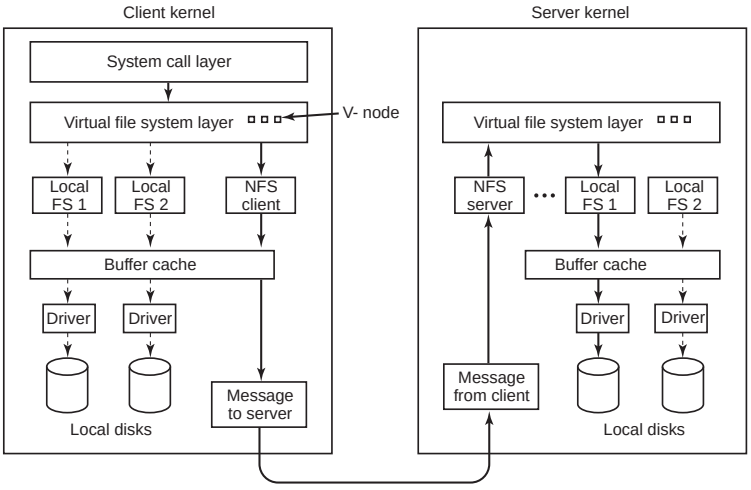
- Una implementazione e specifica di un sistema software per accedere file remoti attraverso LAN (o WAN, se proprio uno deve...)
- Le workstation in rete sono viste come macchine indipendenti con file system indipendenti, che permettono di condividere in maniera trasparente
  - Una dir remota viene montata 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

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



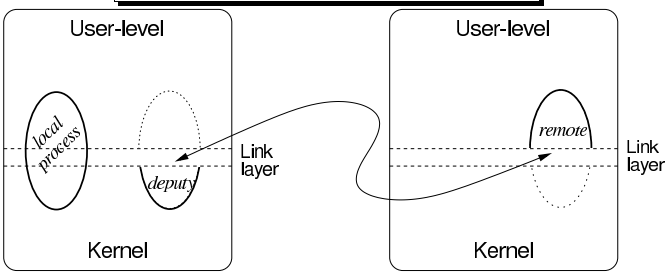


Migrazione di processi/thread

- 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, Apple Xgrid.

Migrazione di processi in MOSIX



- Solo la parte in user space (stack, dati, eseguibile) di un processo migra da un calcolatore all'altro. La parte in kernel space (*deputy*) non migra.
- L'esecuzione in user space continua sul nuovo nodo, ma le system call vengono intercettate dal link layer (MOSIX) e trasmesse al deputy sul nodo originale. Quindi ogni interazione con l'ambiente (I/O, IPC, socket...) vengono eseguite sul nodo originale.