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

Laurea in Informatica — A.A. 2003/04

Trasparenze del Corso di *Sistemi Operativi*

Marino Miculan

Università di Udine

Sistemi di I/O

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

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

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

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

Livelli di astrazione

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

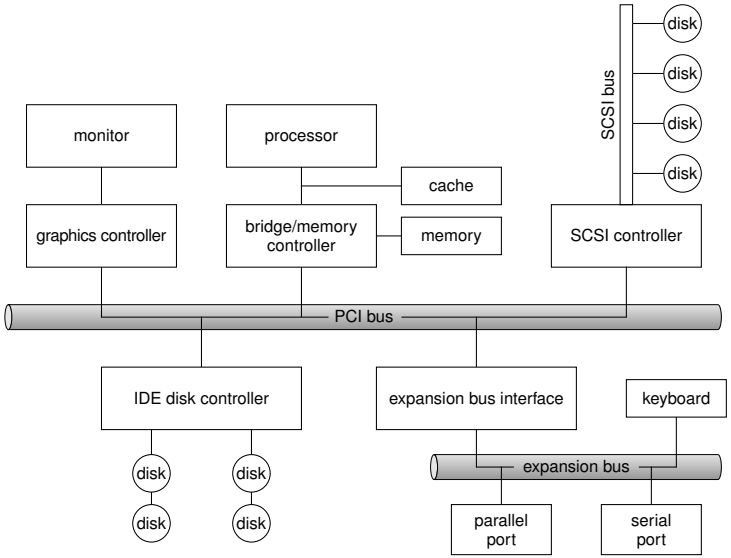
Dispositivi a blocchi e a carattere

Suddivisione logica nel modo di accesso:

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

Comunicazione CPU-I/O

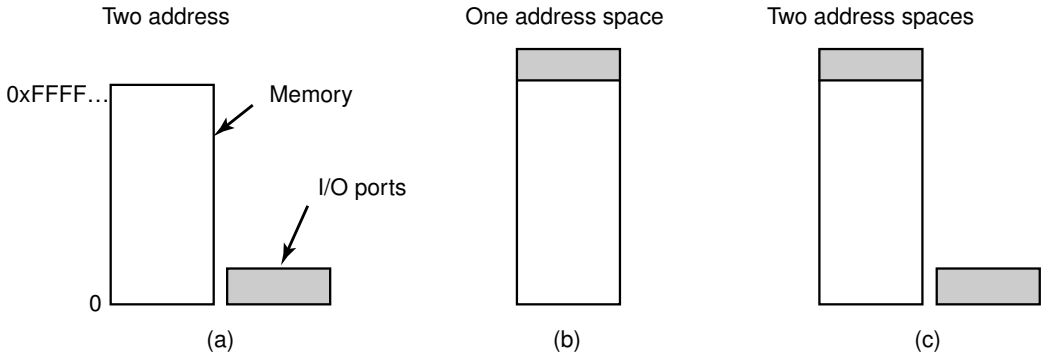
Concetti comuni: *porta*, *bus* (daisychain o accesso diretto condiviso), *controller*



Comunicazione CPU-I/O (cont)

Due modi per comunicare con il (controller del) dispositivo

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



Modi di I/O

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

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

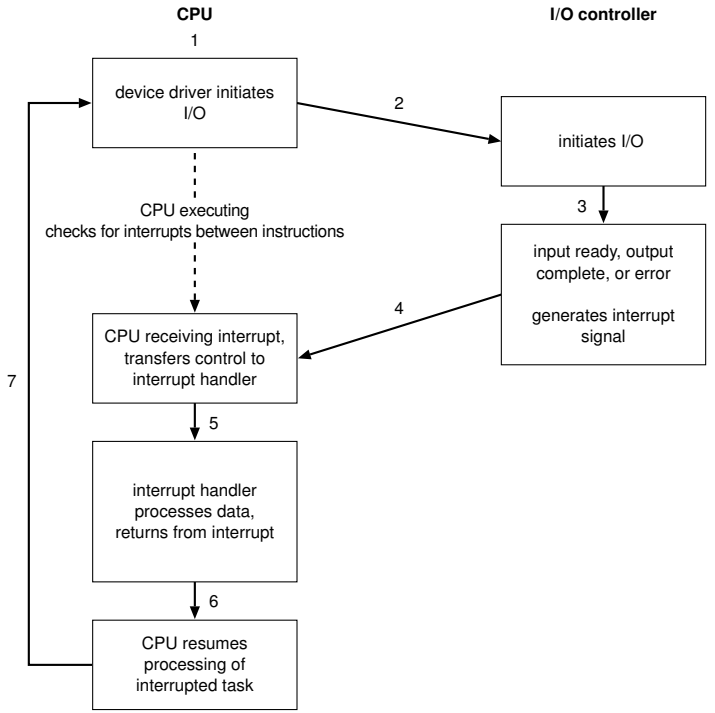
Efficiente solo se la velocità del dispositivo è paragonabile con quella della CPU.

I/O a interrupt

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

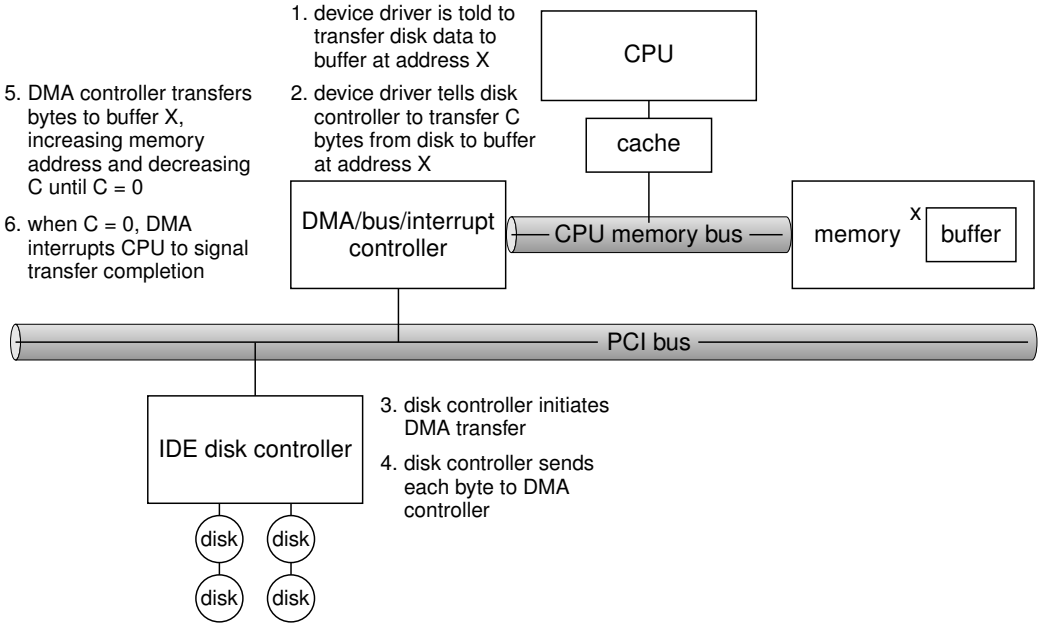
Vettore di interrupt: tabella che associa ad ogni interrupt l'indirizzo di una corrispondente routine di gestione.

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

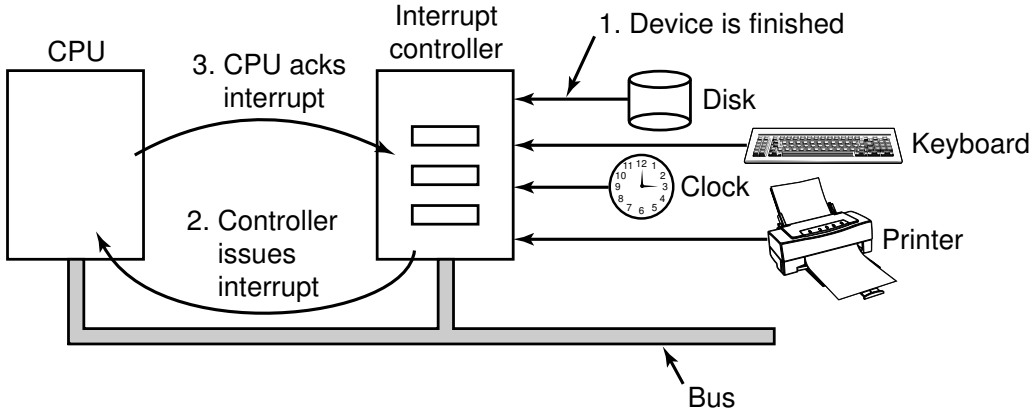


Direct Memory Access

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



Gestione degli interrupt

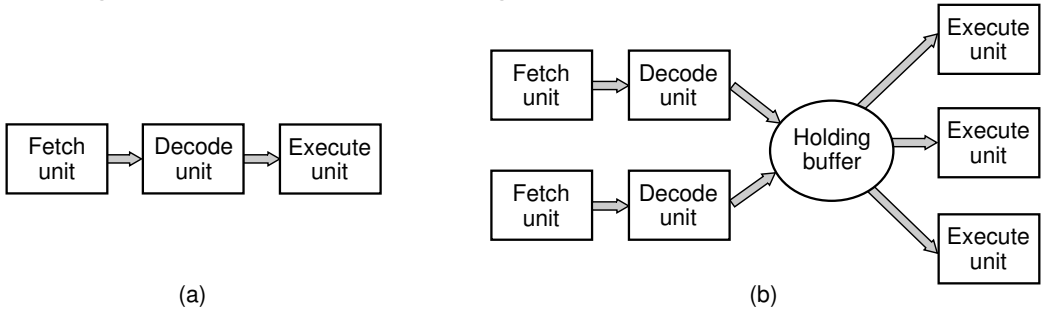


Gestione degli interrupt

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

Gestione degli interrupt e CPU avanzate

- Le CPU con pipeline hanno grossi problemi: il PC non identifica nettamente il punto in cui riprendere l'esecuzione — anzi, punta alla prossima istruzione da mettere nella pipeline.
- Ancora peggio per le superscalari: le istruzioni possono essere già state eseguite, ma fuori ordine! cosa significa il PC allora?



Interruzioni precise

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

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

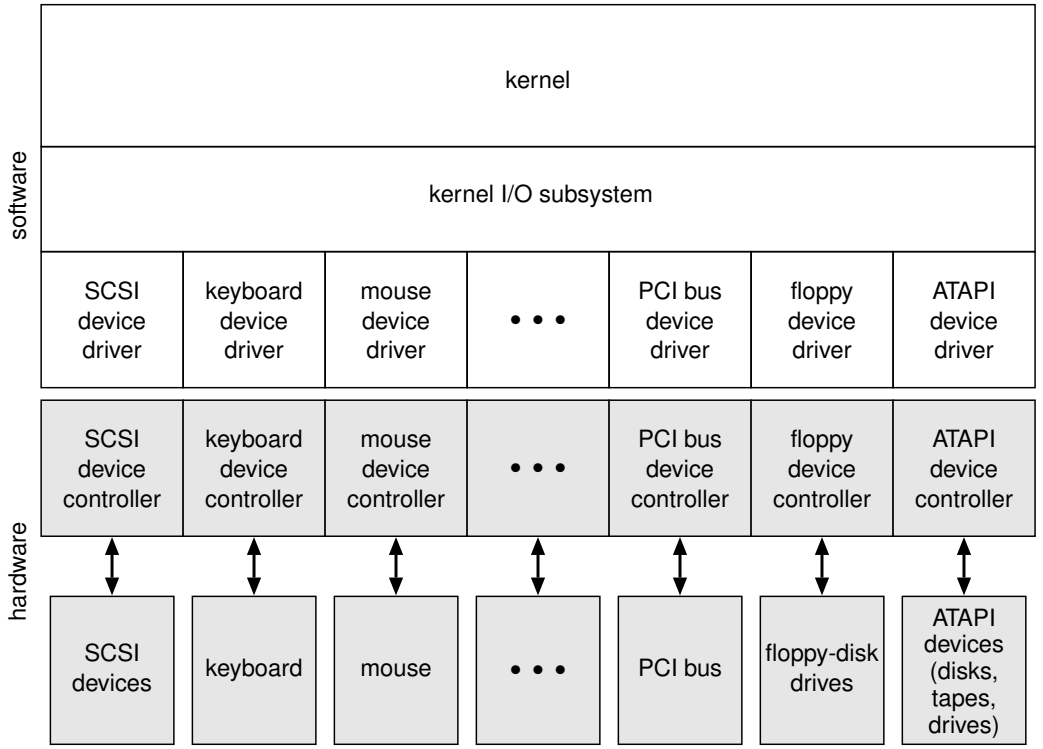
Evoluzione dell'I/O

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

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

Interfaccia di I/O per le applicazioni

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



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

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

Dispositivi a blocchi e a carattere

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

Dispositivi di rete

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

I/O bloccante, non bloccante, asincrono

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

Sottosistema di I/O del kernel

Deve fornire molte funzionalità

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

Sottosistema di I/O del kernel (cont.)

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

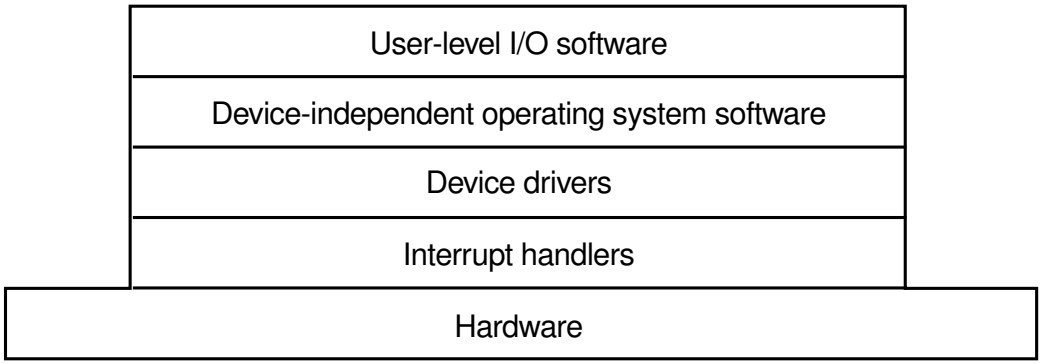
Sottosistema di I/O del kernel (cont.)

Gestione degli errori

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

I livelli del software di I/O

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



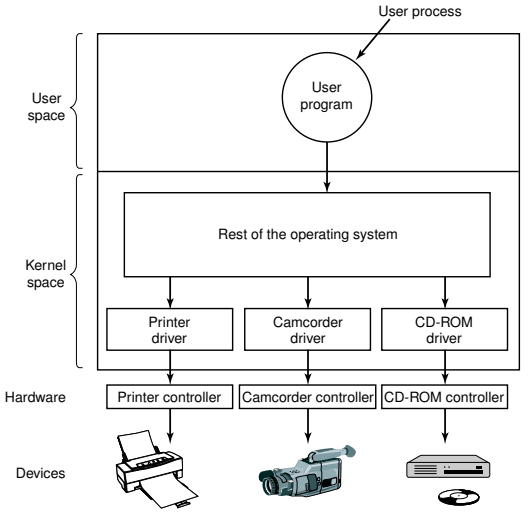
Driver delle interruzioni

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

Driver dei dispositivi

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

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



Passi eseguiti dai driver dei dispositivi

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

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

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

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

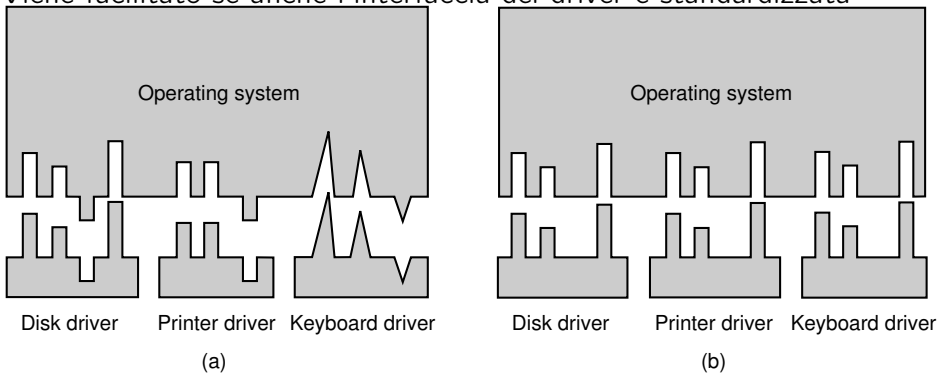
Software di I/O indipendente dai dispositivi

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

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

Interfacciamento uniforme

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



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

Esempio di interfaccia per i driver

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

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

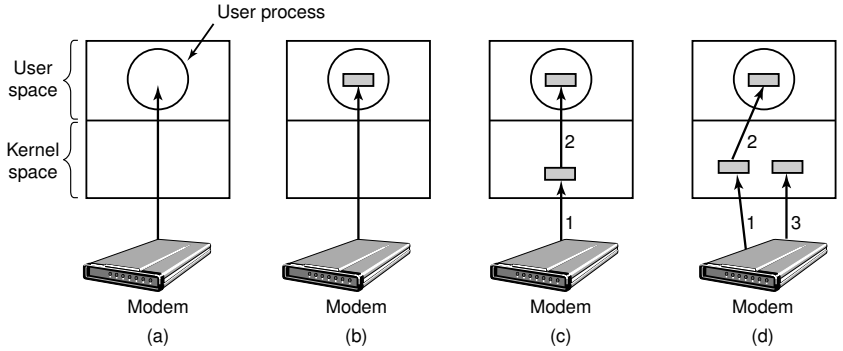

Esempio: le operazioni di un terminale (una seriale)

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

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

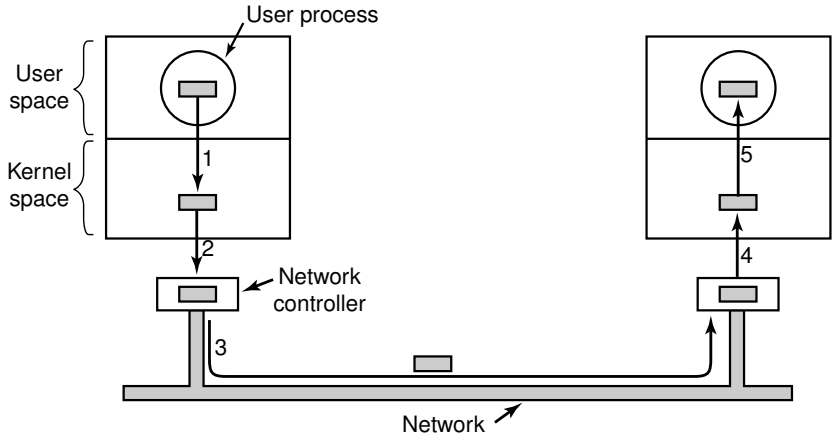
Bufferizzazione

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



Bufferizzazione (cont.)

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



Eccessivo uso della bufferizzazione incide sulle prestazioni

Gestione degli errori

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

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

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

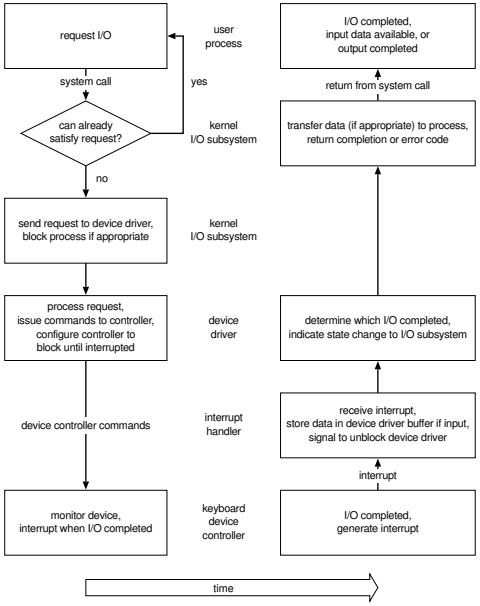
Software di I/O a livello utente

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

Traduzioni delle richieste di I/O in operazioni hardware

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

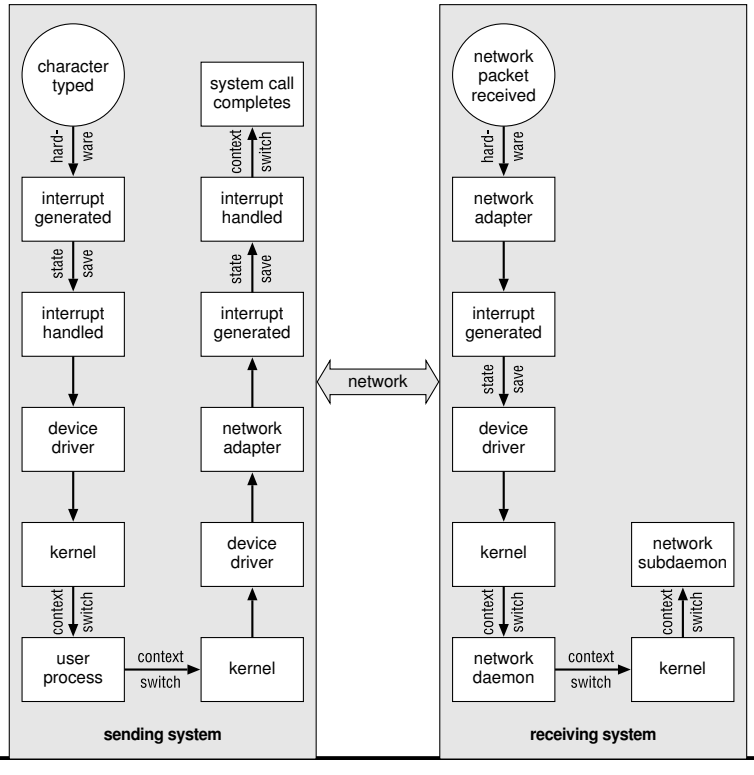
Esecuzione di una richiesta di I/O



Performance

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

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

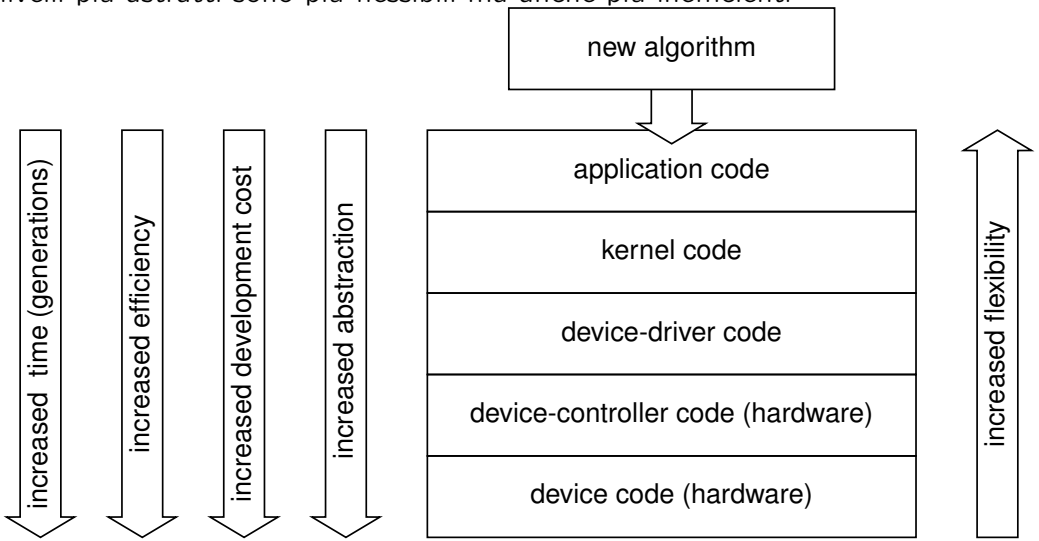


Migliorare le performance

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

Livello di implementazione

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



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

Struttura dei dischi

- La gestione dei dischi riveste particolare importanza.
- I dischi sono indirizzati come dei grandi array monodimensionali di *blocchi logici*, dove il blocco logico è la più piccola unità di trasferimento con il controller.
- L'array monodimensionale è mappato sui settori del disco in modo sequenziale.
  - Settore 0 = primo settore della prima traccia del cilindro più esterno
  - la mappatura procede in ordine sulla traccia, poi sulle rimanenti traccie dello stesso cilindro, poi attraverso i rimanenti cilindri dal più esterno verso il più interno.

Schedulazione dei dischi

- Il sistema operativo è responsabile dell'uso efficiente dell'hardware. Per i dischi: bassi *tempi di accesso* e alta *banda di utilizzo*.
- Il tempo di accesso ha 2 componenti principali, dati dall'hardware:
  - *Seek time* = il tempo (medio) per spostare le testine sul cilindro contenente il settore richiesto.
  - *Latenza rotazionale* = il tempo aggiuntivo necessario affinché il settore richiesto passi sotto la testina.
- Tenere traccia della posizione angolare dei dischi è difficile, mentre si sa bene su quale cilindro si trova la testina
- Obiettivo: mimimizzare il tempo speso in seek
- Tempo di seek $\approx$ distanza di seek; quindi: mimimizzare la distanza di seek
- Banda di disco = il numero totale di byte trasferiti, diviso il tempo totale dalla prima richiesta di servizio e il completamento dell'ultimo trasferimento.

Schedulazione dei dischi (Cont.)

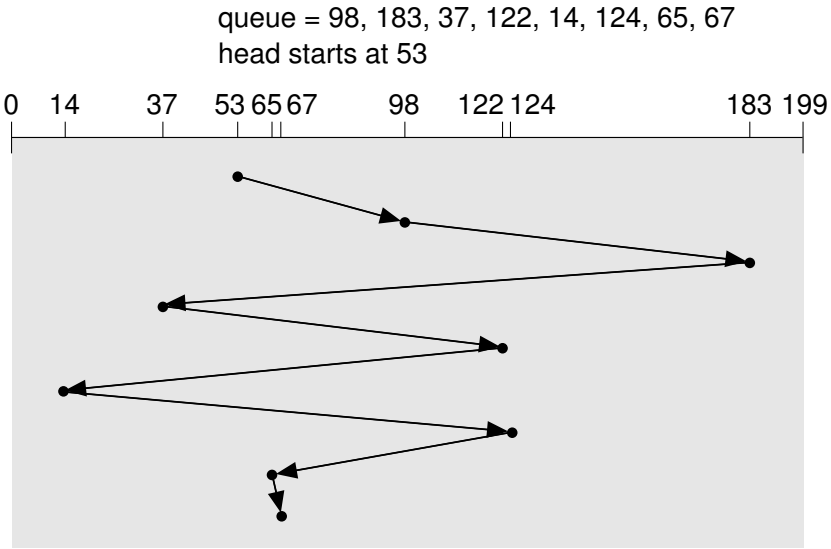
- Ci sono molti algoritmi per schedulare le richieste di I/O di disco.
- Al solito, una trattazione formale esula dal corso
- Illustreremo con una coda di richieste d'esempio, su un range di cilindri 0–199:

  98, 183, 37, 122, 14, 124, 65, 67

- Supponiamo che la posizione attuale della testina sia 53

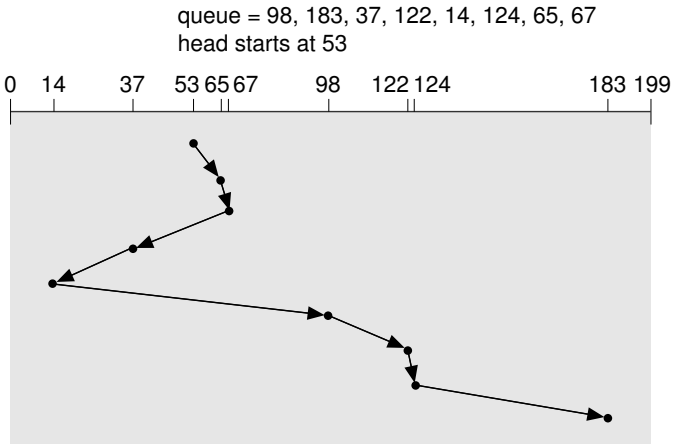
FCFS

Sull'esempio: distanza totale di 640 cilindri



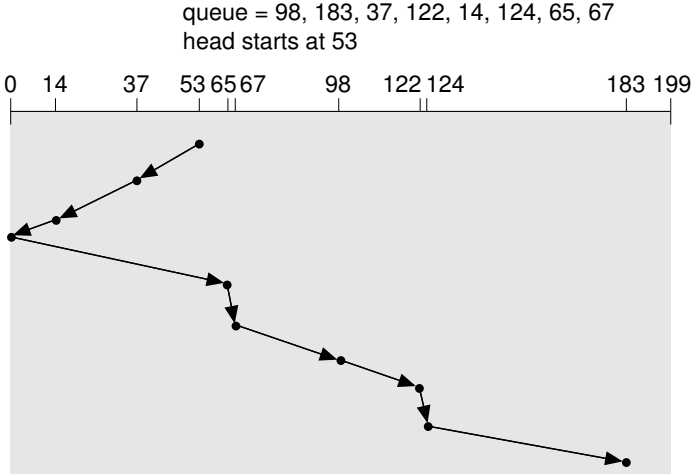
Shortest Seek Time First (SSTF)

- Si seleziona la richiesta con il minor tempo di seek dalla posizion corrente
- SSTF è una forma di scheduling SJF; può causare *starvation*.
- Sulla nostra coda di esempio: distanza totale di 236 cilindri (36% di FCFS).



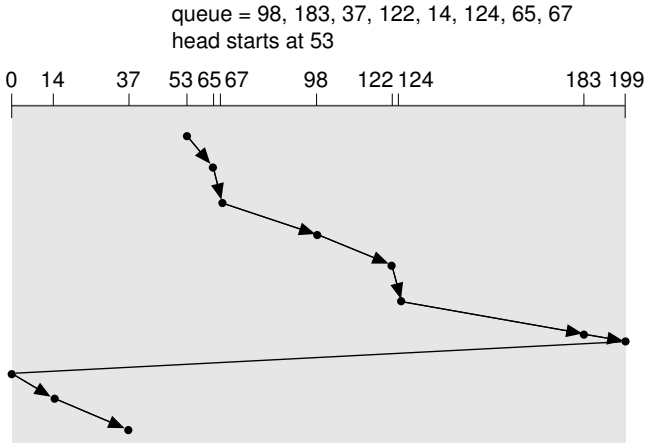
SCAN (o "dell'ascensore")

- Il braccio scandisce l'intera superficie del disco, da un estremo all'altro, servendo le richieste man mano. Agli estreme si inverte la direzione.
- Sulla nostra coda di esempio: distanza totale di 208 cilindri



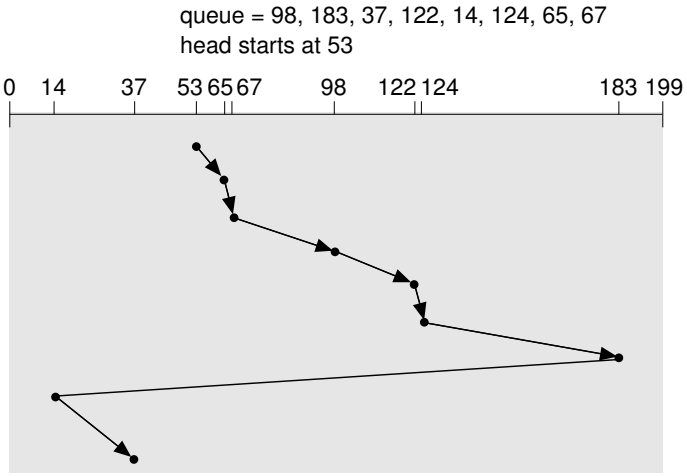
C-SCAN

- Garantisce un tempo di attesa più uniforme e equo di SCAN
- Tratta i cilindri come in lista circolare, scandita in rotazione dalla testina si muove da un estremo all'altro del disco. Quando arriva alla fine, ritorna immediatamente all'inizio del disco senza servire niente durante il rientro.



C-LOOK

- Miglioramento del C-SCAN (esiste anche il semplice LOOK)
- Il braccio si sposta solo fino alla richiesta attualmente più estrema, non fino alla fine del disco, e poi inverte direzione immediatamente.



Quale algoritmo per lo scheduling dei dischi?

- SSTF è molto comune e semplice da implementare, e abbastanza efficiente
- SCAN e C-SCAN sono migliori per i sistemi con un grande carico di I/O con i dischi (si evita starvation)
- Le performance dipendono dal numero e tipi di richieste
- Le richieste ai dischi dipendono molto da come vengono allocati i file, ossia da come è implementato il file system.
- L'algoritmo di scheduling dei dischi dovrebbe essere un modulo separato dal resto del kernel, facilmente rimpiazzabile se necessario.

  (Es: in questi giorni, si discute di cambiare lo scheduler di Linux: anticipatory, deadline I/O, Stochastic Fair Queuing, Complete Fair Queuing. . .)
- Sia SSTF che LOOK (e varianti circolari) sono scelte ragionevoli come algoritmi di default.

Gestione dell'area di swap

- L'area di swap è parte di disco usata dal gestore della memoria come estensione della memoria principale.
- Può essere ricavata dal file system normale o (meglio) in una partizione separata.
- Gestione dell'area di swap
  - 4.3BSD: alloca lo spazio appena parte il processo per i segmenti text e data. Lo stack, man mano che cresce.
  - Solaris 2: si alloca una pagina sullo stack solo quando si deve fare un page-out, non alla creazione della pagina virtuale.
  - Windows 2000: Viene allocato spazio sul file di swap per ogni pagina virtuale non corrispondente a nessun file sul file system (es: DLL).

Affidabilità e performance dei dischi

- aumenta la differenza di velocità tra applicazioni e dischi
- le cache non sempre sono efficaci (es. transazioni, dati da esperimenti)
- Suddividere il carico tra più dischi che cooperano per offrire l'immagine di un disco unitario virtuale
- Problema di affidabilità:

$$MTBF_{array} = \frac{MTBF_{disco}}{\#dischi}$$

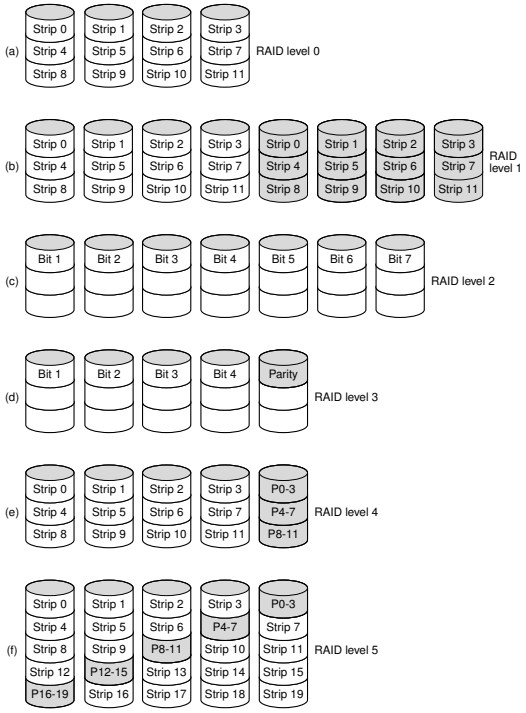
RAID

- 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 "affettate" e parallelizzate. Altissima performance, 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)



Il sistema di I/O in Unix

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

Denominazione dei device: tipo, major e minor number

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

I device sono file!

Tutti i dispositivi sono accessibili come file *speciali*

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

La directory /dev

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


La directory /dev (cont.)

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

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

  `# mknod /tmp/miodisco b 3 0`

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

System call per I/O di Unix

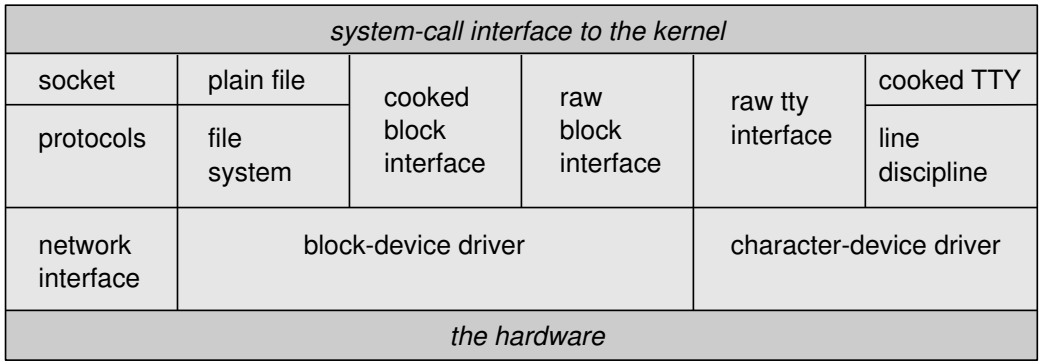
Sono le stesse dei file, solo che sono usate su device. Le principali sono

- `open(2)`: aprire un file/dispositivo
- `read(2)`, `readdir(2)`, `write(2)`: leggi/scrivi da/su un dispositivo
- `close(2)`: chiudi (rilascia) un dispositivo
- `lseek(2)`: posiziona il puntatore di lettura/scrittura
- `ioctl(2)`: generica chiamata di controllo I/O
  `int ioctl(int fd, int request, ...)`
- `fsync(2)`: sincronizza lo stato su disco di un file con quello in memoria
- `mmap(2)`: mappa un file o device in memoria virtuale di un processo

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

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

Struttura di I/O in Unix

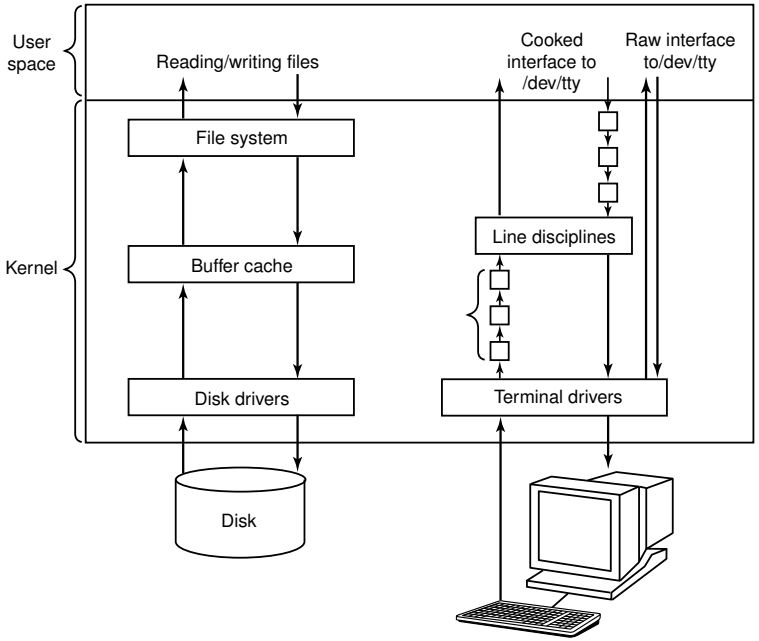


Il sistema di I/O consiste in

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

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

Cache e buffering



Le Device Table e la struct file_operations

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

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

Le Device Table e la struct file_operations (cont.)

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

Dispositivi a blocchi: Cache

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

Dispositivi a blocchi: Cache (cont.)

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

Code di operazioni pendenti e dispositivi "crudi"

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

Code di operazioni pendenti e dispositivi “crudi” (cont.)

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

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

- Le interfacce crude non usano la cache

- Utili per operazioni particolari (formattazione, partizionamento), o per implementare proprie politiche di caching o proprî file system (es. database).

Dispositivi a caratteri: C-Lists

- *C-list:* sistema di buffering che mantiene piccoli blocchi di caratteri di dimensione variabile in liste linkate

- Implementato nello strato device independent dei dispositivi a carattere

- Trasferimento in output:

  1. Una *write* a un terminale accoda il blocco di caratteri da scrivere nella C-list di output per il device

  2. Il driver inizia il trasferimento del primo blocco sulla lista, o con DMA o con PIO sul chip di I/O.

  3. A trasferimento avvenuto, il chip di I/O solleva un interrupt; la routine di gestione verifica lo stato, e se ha avuto successo il blocco viene tolto dalla C-list

  4. salta a 2.

Dispositivi a caratteri: C-Lists (cont.)

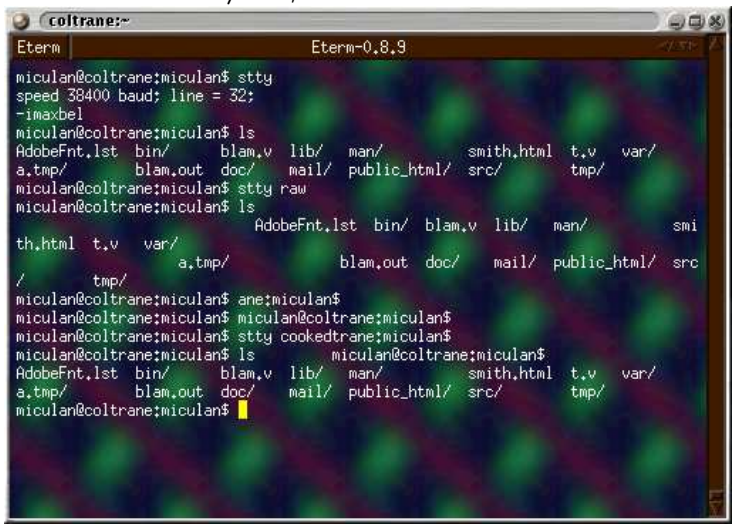
- L’input avviene analogamente, guidato dagli interrupt.

- Accesso *cooked* (POSIX: *canonical*): normalmente, i caratteri in entrata e in uscita sono filtrati dalla *disciplina di linea*. Es: ripulitura dei caratteri di controllo in input, traduzione dei carateri di controllo in output.

- Accesso *crudo* (POSIX: *noncanonical*): permette di bypassare la C-list e la disciplina di linea. Ogni carattere viene passato/prelevato direttamente al driver senza elaborazione. Usato da programmi che devono reagire ad ogni tasto (eg., vi, emacs, server X, videogiochi)

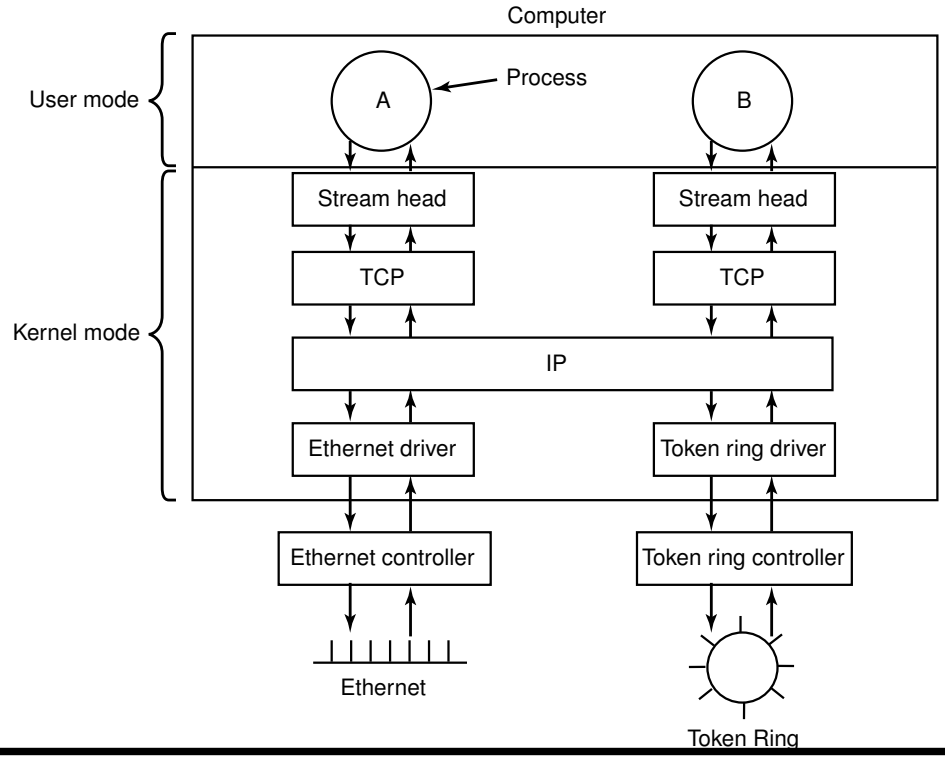
Controllo di linea: stty(2)

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



STREAM

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



Monitoraggio I/O: iostat(1M)

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


Sistema di I/O di Windows 2000

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

Implementazione dell'I/O in Windows 2000

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

Installazione dei driver e dispositivi

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

Stack di driver in Windows 2000

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

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