

Trasparenze del Corso di *Sistemi Operativi*

Ivan Scagnetto

Università di Udine

Copyright © 2000-04 Marino Miculan (miculan@dimi.uniud.it)

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

1

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

428

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

429

Dispositivi a blocchi e a carattere

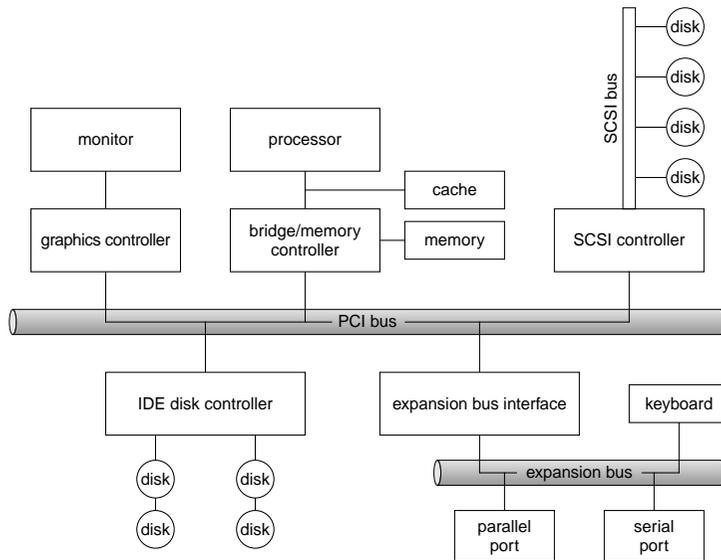
Suddivisione logica nel modo di accesso:

- 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).

430

Comunicazione CPU-I/O

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



431

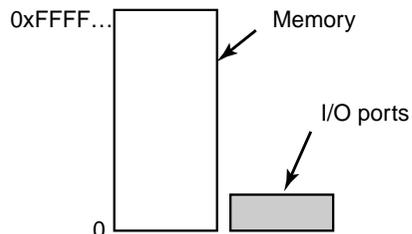
Comunicazione CPU-I/O (cont)

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.

432

Two address



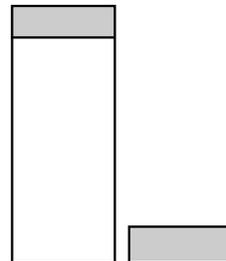
(a)

One address space



(b)

Two address spaces



(c)

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.

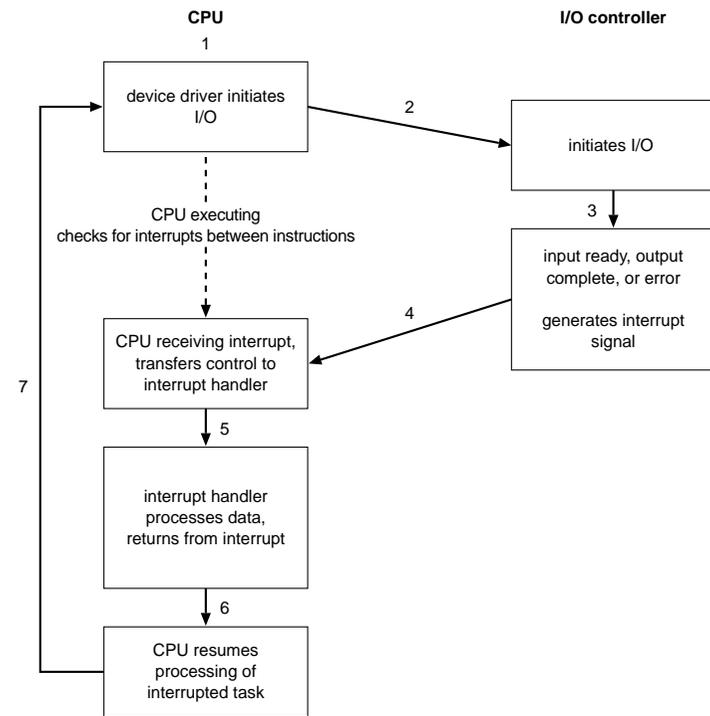
433

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.

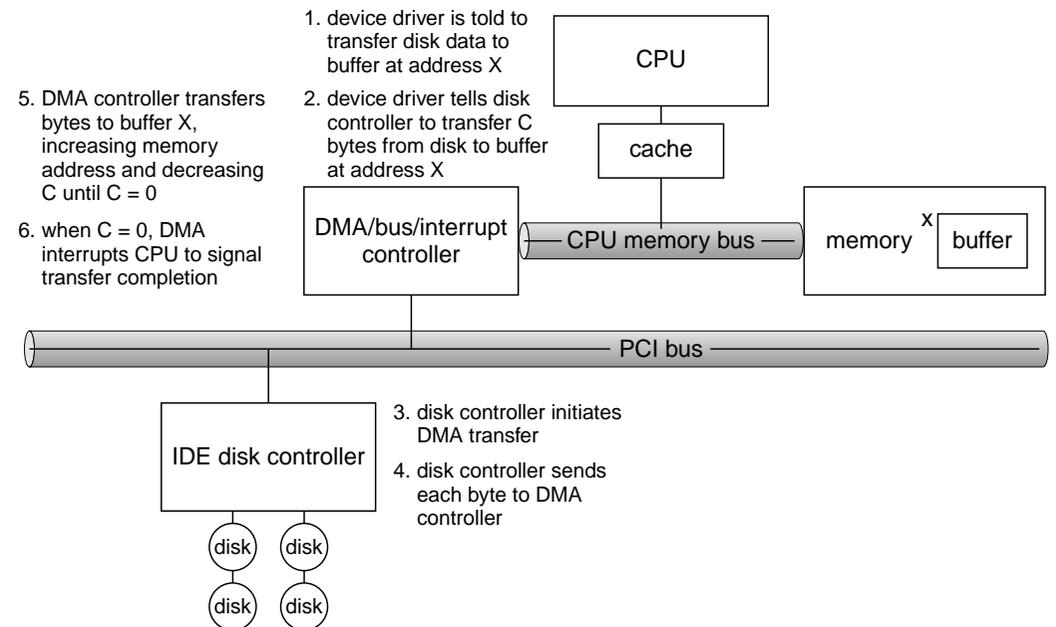
Vettore di interrupt: tabella che associa ad ogni interrupt l'indirizzo di una corrispondente routine di gestione.

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

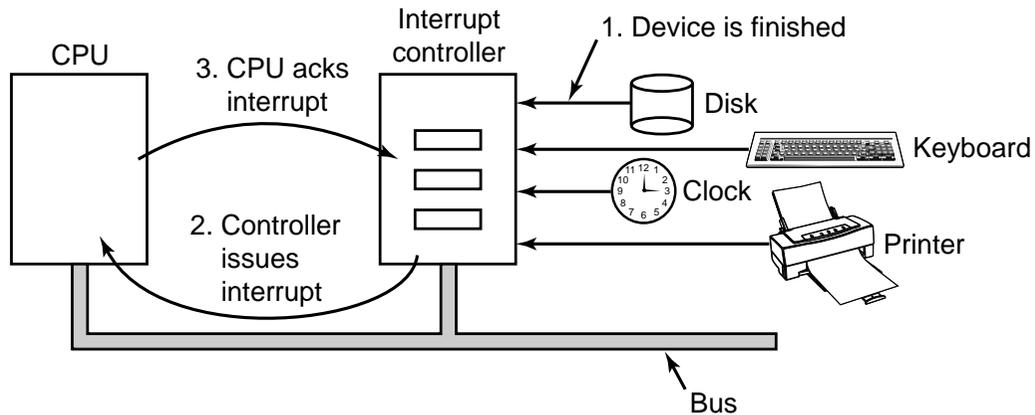


Direct Memory Access

- 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



436

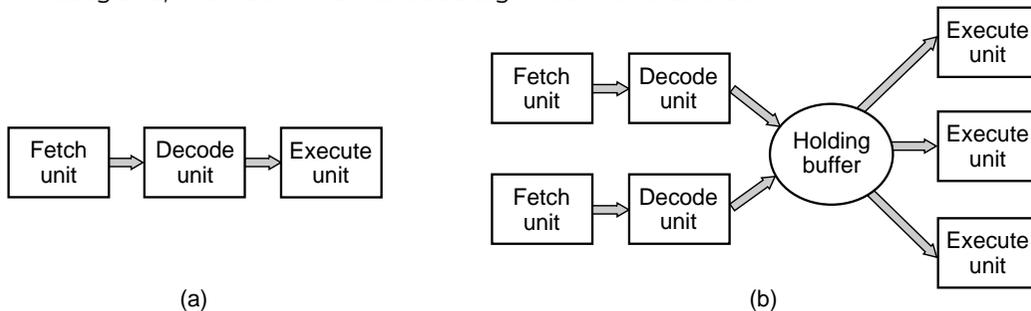
Gestione degli interrupt

- 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

437

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?



438

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
 - 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 ⇒ grandi latenze. . .

439

- 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, 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, ...

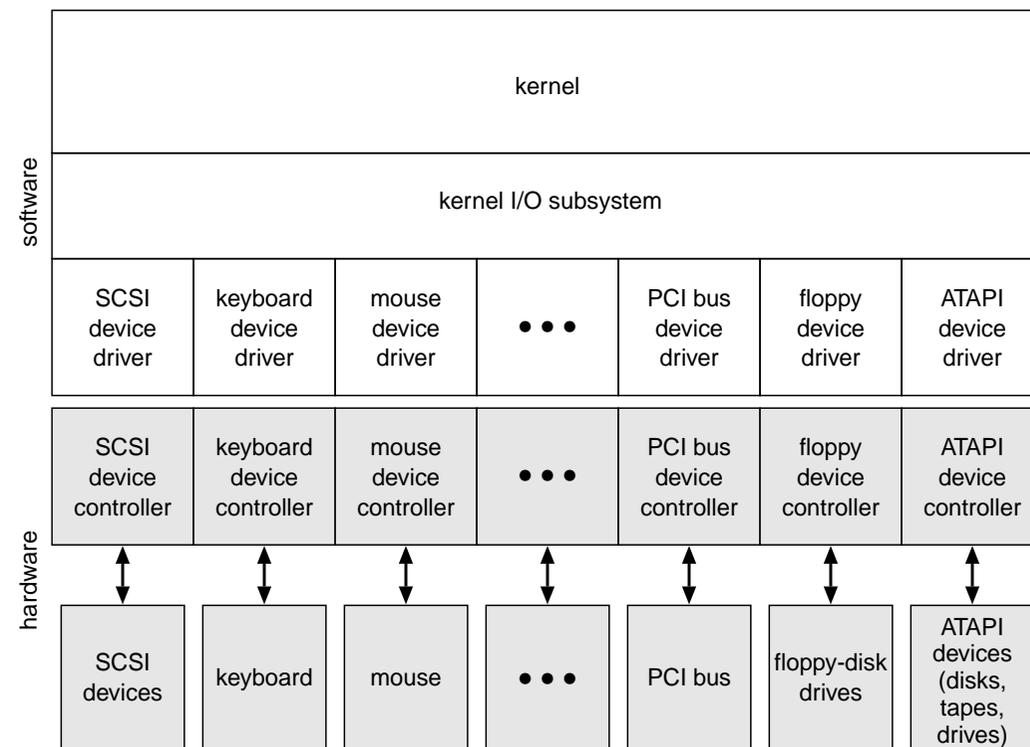
Tipicamente in un sistema di calcolo sono presenti più tipi di I/O.

440

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.

441



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*

442

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), 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.

443

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, ...)

444

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

445

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)

446

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!

447

Sottosistema di I/O del kernel (cont.)

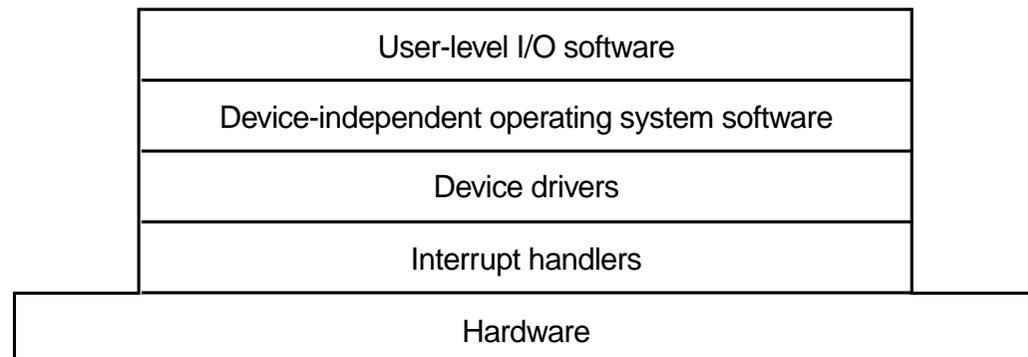
Gestione degli errori

- 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*

448

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)



449

Driver delle interruzioni

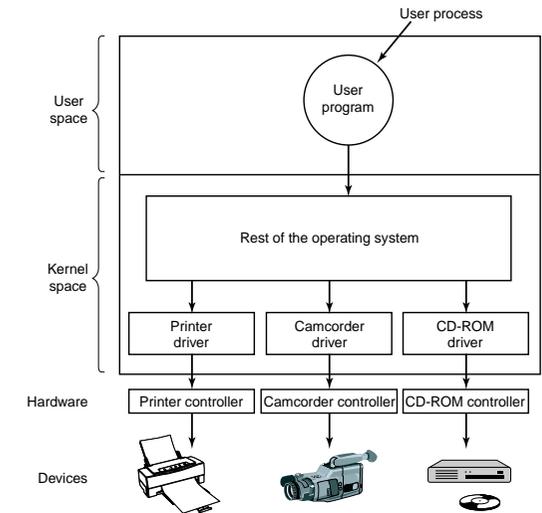
- Fondamentale nei sistemi time-sharing e con I/O interrupt driven
- Passi principali da eseguire:
 1. Salvare i registri della CPU
 2. Impostare un contesto per la procedura di servizio (TLB, MMU, stack. . .)
 3. Ack al controllore degli interrupt (per avere interrupt annidati)
 4. Copiare la copia 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.

450

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 eseguiti in spazio kernel
- Per includere un driver, può essere necessario ricompilare o ralinkare il kernel.
- Attualmente si usa un meccanismo di caricamento run-time



451

Passi eseguiti dai driver dei dispositivi

1. Controllare i parametri passati
2. Accodare le richieste in 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 in uno strato sottostante), 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.

452

Software di I/O indipendente dai dispositivi

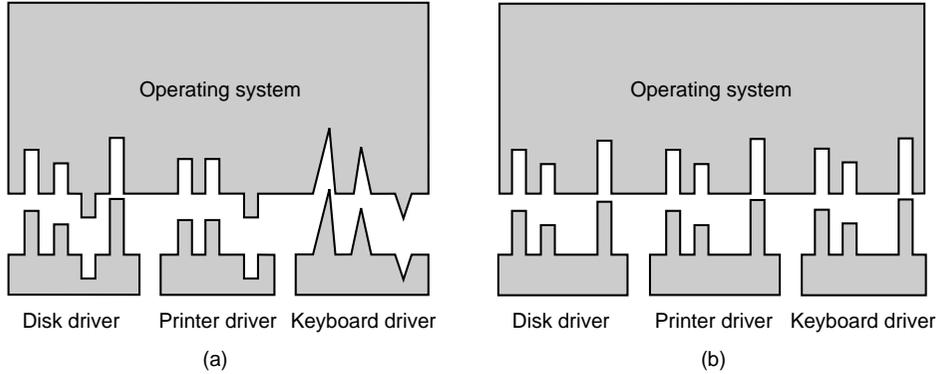
Implementa le funzionalità comuni a tutti i dispositivi (di una certa classe):

- 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)

453

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)

454

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, unsigned
};
```

455

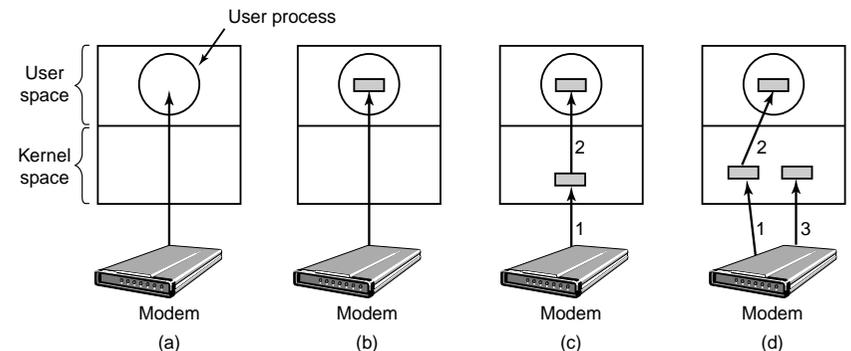
Esempio: le operazioni di un terminale (una seriale)

```
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

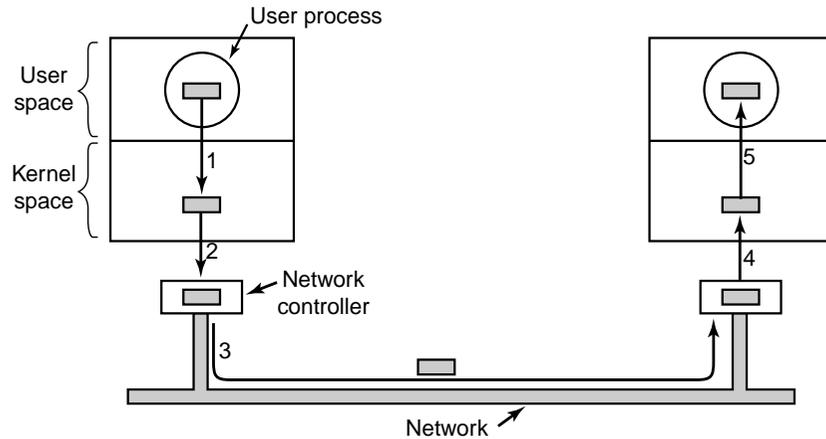
- Non bufferizzato: inefficiente
- Bufferizzazione in spazio utente: problemi con la memoria virtuale
- Bufferizzazione in kernel: bisogna copiare i dati, con blocco dell'I/O nel frattempo.
- Doppia bufferizzazione



456

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

457

Gestione degli errori

- Errori di programmazione: il programmatore chiede qualcosa di impossibile/inconsistente (scrivere su un CD-ROM, eseguire un'operazione di seek su 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).

458

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. . .

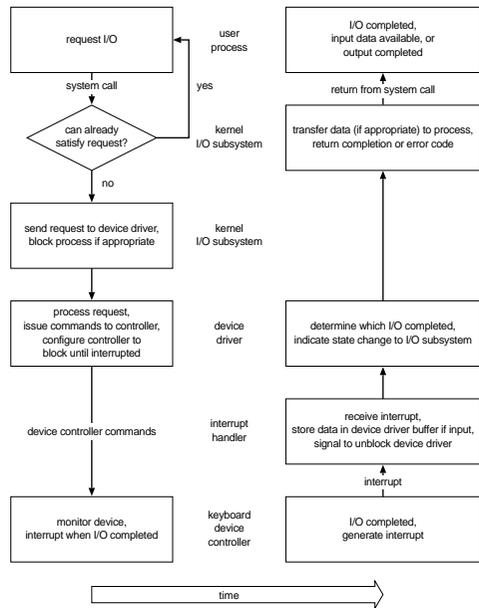
459

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 disponibili al 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*).

460

Esecuzione di una richiesta di I/O



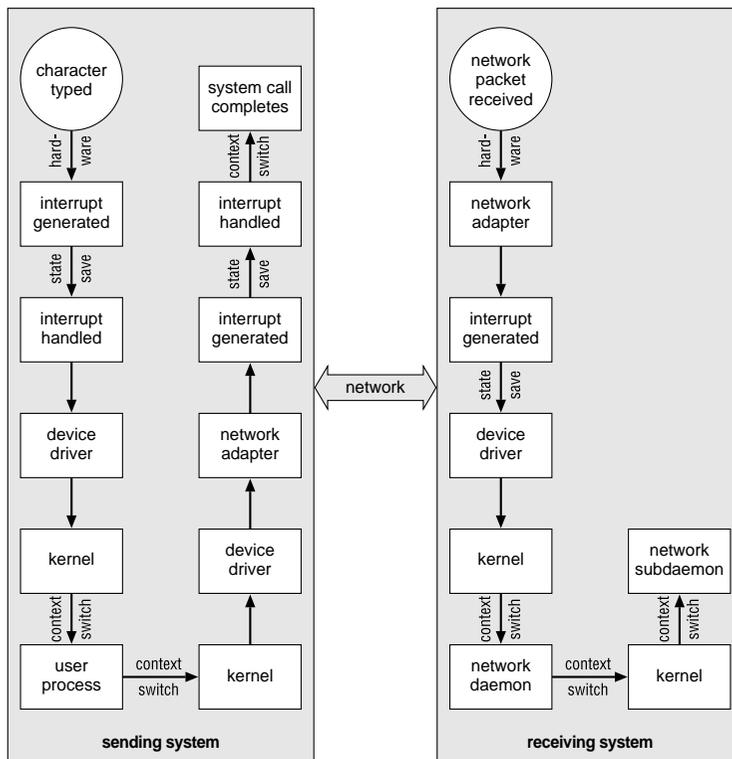
461

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)

462



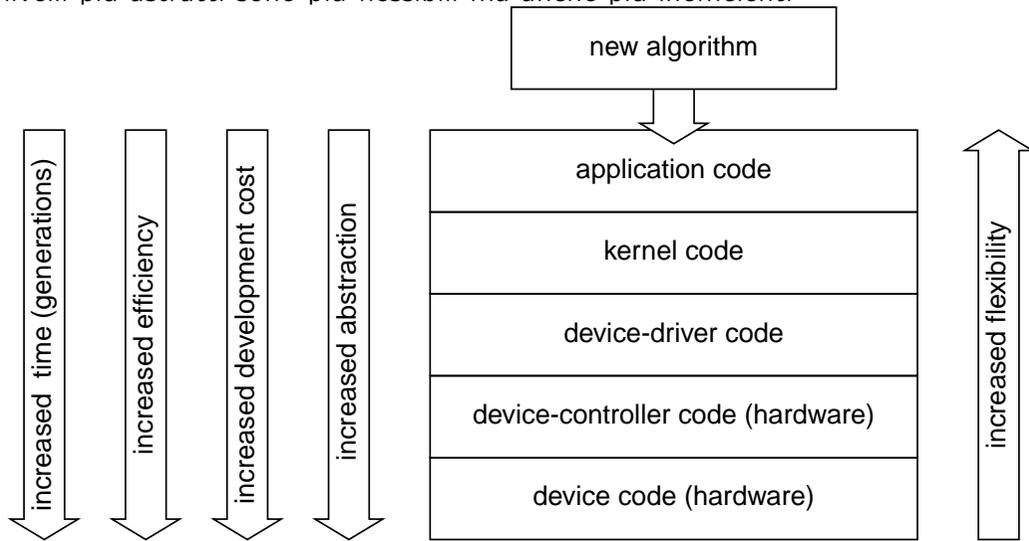
Migliorare le performance

- 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

463

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



464

- 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.