

CAPITOLO

13

OBIETTIVI DEL CAPITOLO

- Analisi della struttura del sottosistema di I/O di un sistema operativo.
- Elementi base dell'hardware di I/O e problematiche correlate.
- Aspetti relativi alle prestazioni dell'hardware e del software di I/O.

Sistemi di I/O

I due compiti principali di un calcolatore sono l'I/O e l'elaborazione. In molti casi il compito principale è costituito dall'I/O mentre l'elaborazione è semplicemente accessoria: quando si consulta una pagina web, o quando si modifica un file, ciò che più direttamente interessa l'utente è la lettura o l'immissione di informazioni, non l'elaborazione di qualche risposta.

Il ruolo di un sistema operativo nell'I/O di un calcolatore è quello di gestire e controllare le operazioni e i dispositivi di I/O. Sebbene in altri capitoli compaiano argomenti collegati, in questo capitolo sono raccolti tutti gli elementi utili alla composizione di un quadro d'insieme dell'I/O. Poiché il genere delle interfacce hardware stabilisce i requisiti che le funzioni interne del sistema operativo devono possedere, si descrivono innanzitutto i fondamenti dell'hardware di I/O. Quindi si discutono i servizi di I/O che il sistema operativo fornisce e come questi sono inclusi nell'interfaccia di I/O per le applicazioni; inoltre si spiega come il sistema operativo colmi il divario tra le interfacce hardware e quelle per le applicazioni. Si prende in esame anche il meccanismo STREAMS di UNIX System V, che consente a un'applicazione di comporre dinamicamente catene di codice di driver. Infine, si trattano gli aspetti riguardanti le prestazioni dell'I/O e i principi di progettazione dei sistemi operativi utili al miglioramento delle prestazioni dell'I/O.

13.1 Introduzione

Il controllo dei dispositivi connessi a un calcolatore è una delle questioni più importanti che riguardano i progettisti di sistemi operativi. Poiché i dispositivi di I/O sono così largamente diversi per funzioni e velocità (si considerino per esempio un mouse, un disco e un archivio di nastri), altrettanto diversi devono essere i metodi di controllo. Tali metodi costituiscono il *sottosistema di I/O* del kernel; questo sottosistema separa il resto del kernel dalla complessità di gestione dei dispositivi di I/O.

La tecnologia dei dispositivi di I/O mostra due tendenze tra loro in conflitto. Da una parte, si osserva la crescente uniformazione a standard delle interfacce fisiche e logiche, e ciò semplifica l'introduzione nei calcolatori e nei sistemi operativi già esistenti di più avanzate generazioni di dispositivi. D'altra parte, però, si assiste a una crescente varietà di dispositivi di I/O; alcuni di loro sono tanto diversi dai dispositivi precedenti da rendere molto difficile il compito di integrarli nei calcolatori e nei sistemi operativi esistenti. Questo problema si affronta con una combinazione di tecniche hardware e software. Gli elementi di base dell'hardware di I/O – porte, bus e controllori di dispositivi – si possono connettere con un'ampia varietà di dispositivi di I/O. Il kernel del sistema operativo è strutturato in moduli di driver dei dispositivi allo scopo di incapsulare i dettagli e le particolarità dei diversi dispositivi. I **driver dei dispositivi** offrono al sottosistema di I/O un'interfaccia uniforme per l'accesso ai dispositivi, così come le chiamate di sistema forniscono un'interfaccia uniforme tra le applicazioni e il sistema operativo.

13.2 Hardware di I/O

I calcolatori fanno funzionare un gran numero di tipi di dispositivi. La maggior parte rientra nella categoria dei dispositivi di memorizzazione (dischi, nastri), dispositivi di trasmissione (connessioni di rete, Bluetooth), interfacce uomo-macchina (schermi, tastiere, mouse, ingressi e uscite audio). Altri dispositivi sono più specializzati, come i dispositivi di pilotaggio di un caccia a reazione. In questi velivoli il pilota interagisce col calcolatore di bordo tramite la *cloche* e la pedaliera, il calcolatore invia comandi per l'attivazione dei motori che azionano timoni, *flap* e propulsori. Nonostante l'incredibile varietà dei dispositivi di I/O, bastano alcuni concetti per capire come siano connessi e come il sistema operativo li controlli.

Un dispositivo comunica con un sistema elaborativo inviando segnali attraverso un cavo o attraverso l'etere e comunica con il calcolatore tramite un punto di connessione (**porta**), per esempio una porta seriale. Se più dispositivi condividono un insieme di fili, la connessione è detta *bus*. Un **bus** è un insieme di fili e un protocollo rigorosamente definito che specifica l'insieme dei messaggi che si possono inviare attraverso i fili. In termini elettronici, i messaggi si inviano tramite configurazioni di livelli di tensione elettrica applicate ai fili con una definita scansione temporale. Quando un dispositivo A ha un cavo che si connette a un dispositivo B e il dispositivo B ha un cavo che si connette a un dispositivo C che a sua volta è collegato a una porta

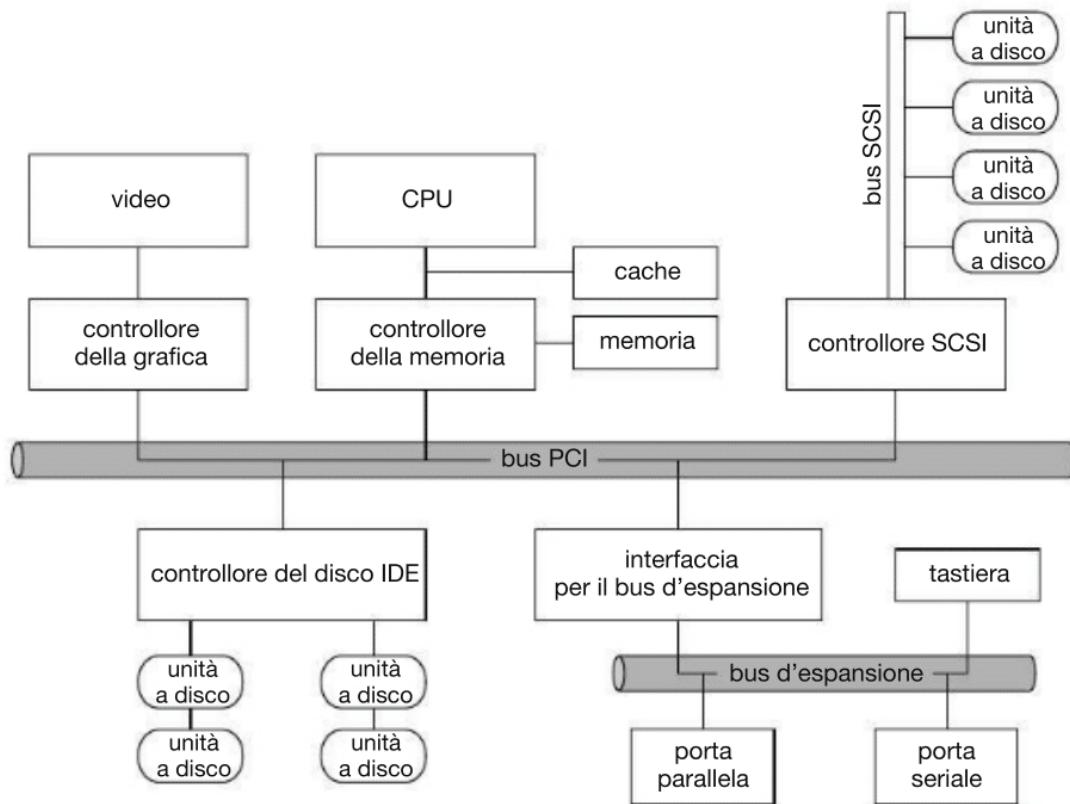


Figura 13.1 Tipica struttura del bus di un PC.

di un calcolatore, si ottiene il cosiddetto **collegamento in daisy chain**, che di solito funziona come un bus.

I bus sono ampiamente usati nell'architettura dei calcolatori e differiscono tra loro per formato dei segnali, velocità, throughput e metodo di connessione. La Figura 13.1 mostra una tipica struttura di bus di PC; si tratta di un **bus PCI** (il comune bus di sistema dei PC) che connette il sottosistema CPU-memoria ai dispositivi veloci, e di un **bus d'espansione** cui si connettono i dispositivi relativamente lenti come la tastiera e le porte seriali e USB. Nella parte superiore destra della figura quattro dischi sono collegati a un bus SCSI (*small computer system interface*) inserito nel relativo controllore. Altri tipi comuni di bus usati per connettere i principali componenti di un computer includono il **PCI Express** (PCIe), con throughput massimo di 16 GB al secondo e **HyperTransport**, con throughput che arriva fino a 25 GB al secondo.

Un **controllore** è un insieme di componenti elettronici che può far funzionare una porta, un bus o un dispositivo. Un controllore di porta seriale è un semplice controllore di dispositivo; si tratta di un singolo circuito integrato (o di una sua parte) nel calcolatore che controlla i segnali presenti nei fili della porta seriale. Per contro un controllore SCSI non è semplice; poiché il protocollo SCSI è complesso, il controllore del bus SCSI è spesso realizzato come una scheda hardware separata, un **adattatore**, che s'inserisce nel calcolatore. Esso contiene generalmente un'unità d'elaborazione, microcodice, e memoria privata che gli consentono di elaborare i messaggi del protocollo SCSI. Alcuni

dispositivi sono dotati di propri controllori incorporati. Osservando un’unità a disco si vede, da un lato, una scheda elettronica a essa agganciata; si tratta del controllore che attua la parte lato disco del protocollo di qualche tipo di connessione, per esempio SCSI o SATA (*serial advanced technology attachment*). Ha un’unità d’elaborazione e microcodice per l’esecuzione di molti compiti, come localizzazione dei settori difettosi, prelievo anticipato (*prefetching*), gestione del buffer e della cache.

L’unità d’elaborazione fornisce comandi e dati al controllore per portare a termine trasferimenti di I/O tramite uno o più registri per dati e segnali di controllo. La comunicazione con il controllore avviene attraverso la lettura e la scrittura, da parte dell’unità d’elaborazione, di configurazioni di bit in questi registri. Un modo in cui questa comunicazione può avvenire è tramite l’uso di speciali istruzioni di I/O che specificano il trasferimento di un byte o una parola a un indirizzo di porta di I/O. L’istruzione di I/O attiva le linee di bus per selezionare il giusto dispositivo e trasferire bit dentro o fuori dal registro di dispositivo. In alternativa, il controllore di dispositivo può supportare l’**I/O memory mapped** (*mappato in memoria*). In questo caso i registri di controllo del dispositivo sono mappati in un sottoinsieme dello spazio d’indirizzi della CPU, che esegue le richieste di I/O usando le ordinarie istruzioni di trasferimento di dati per leggere e scrivere i registri di controllo del dispositivo alle locazioni di memoria fisica in cui sono mappati.

Certi sistemi usano entrambe le tecniche. I PC per esempio usano istruzioni di I/O per controllare alcuni dispositivi e l’I/O memory mapped per controllarne altri. Nella Figura 13.2 sono riportati gli usuali indirizzi delle porte di I/O dei PC. Il controllore della grafica utilizza alcune porte di I/O per le operazioni di controllo di base, ma dispone di un’ampia regione mappata in memoria che serve a mantenere i contenuti dello schermo. Il processo scrive sullo schermo inserendo i dati nella regione mappata in memoria; il controllore genera l’immagine dello schermo sulla base del contenuto

indirizzi per l’I/O (in esadecimale)	dispositivo
000-00F	controllore DMA
020-021	controllore delle interruzioni
040-043	timer
200-20F	controllore dei giochi
2F8-2FF	porta seriale (secondaria)
320-32F	controllore del disco
378-37F	porta parallela
3D0-3DF	controllore della grafica
3F0-3F7	controllore dell’unità a dischetti
3F8-3FF	porta seriale (principale)

Figura 13.2 Indirizzi delle porte dei dispositivi di I/O nei PC (elenco parziale).

di questa regione di memoria. Questa tecnica è semplice da usare; inoltre la scrittura di milioni di byte nella memoria grafica è più veloce dell'invio di milioni di istruzioni di I/O. La facilità di scrittura in un controllore di I/O memory mapped è però controbilanciata da uno svantaggio: un comune errore di programmazione è la scrittura in una regione di memoria sbagliata causata da un puntatore errato. Ciò rende i registri dei dispositivi mappati in memoria vulnerabili a modifiche accidentali. Naturalmente, le tecniche di protezione della memoria aiutano a ridurre tale rischio.

Una porta di I/O consiste in genere di quattro registri: **status**, **control**, **data-in** e **data-out**.

- La CPU legge dal registro **data-in** per ricevere dati.
- La CPU scrive nel registro **data-out** per emettere dati.
- Il registro **status** contiene alcuni bit che possono essere letti dalla CPU e indicano lo stato della porta; per esempio indicano se è stata portata a termine l'esecuzione del comando corrente, se un byte è disponibile per essere letto dal registro **data-in**, se si è verificato un errore del dispositivo.
- Il registro **control** può essere scritto per attivare un comando o per cambiare il modo di funzionamento del dispositivo. Per esempio, un certo bit nel registro **control** della porta seriale determina il tipo di comunicazione tra *half-duplex* e *full-duplex*, un altro abilita il controllo di parità, un terzo imposta la lunghezza delle parole a 7 o 8 bit, altri selezionano una tra le velocità che la porta seriale può sostenere.

La tipica dimensione dei registri di dati varia tra 1 e 4 byte. Certi controllori hanno circuiti integrati FIFO che possono contenere parecchi byte per l'invio e la ricezione di dati, in modo da espandere la capacità del controllore oltre la dimensione del registro di dati. Un circuito integrato FIFO può contenere una piccola sequenza di dati finché il dispositivo o la CPU non sono in grado di riceverli.

13.2.1 Polling

Il protocollo completo per l'interazione fra la CPU e un controllore può essere intricato, ma la fondamentale nozione di **handshaking** (*negoziazione*) è semplice, ed è illustrata con un esempio. Si assuma l'uso di due bit per coordinare la relazione di tipo produttore-consamatore fra il controllore e la CPU. Il controllore specifica il suo stato per mezzo del bit **busy** del registro **status**; pone a 1 il bit **busy** quando è impegnato in un'operazione, e lo pone a 0 quando è pronto a eseguire il comando successivo. La CPU comunica le sue richieste tramite il bit **command-ready** nel registro **command**: pone questo bit a 1 quando il controllore deve eseguire un comando. In questo esempio, la CPU scrive in una porta coordinandosi con un controllore per mezzo del seguente handshaking.

1. La CPU legge ripetutamente il bit **busy** finché questo non vale 0.
2. La CPU pone a 1 il bit **write** del registro dei comandi e scrive un byte nel registro **data-out**.

3. La CPU pone a 1 il bit `command-ready`.
4. Quando il controllore si accorge che il bit `command-ready` è posto a 1, pone a 1 il bit `busy`.
5. Il controllore legge il registro dei comandi e trova il comando `write`; legge il registro `data-out` per ottenere il byte da scrivere, e compie l'operazione di I/O sul dispositivo.
6. Il controllore pone a 0 il bit `command-ready`, pone a 0 il bit `error` nel registro `status` per indicare che l'operazione di I/O ha avuto esito positivo, e pone a 0 il bit `busy` per indicare che l'operazione è terminata.

La sequenza appena descritta si ripete per ogni byte.

Durante l'esecuzione del passo 1, la CPU è in **attesa attiva** (*busy-waiting*) o in **interrogazione ciclica** (*polling*): itera la lettura del registro `status` finché il bit `busy` assume il valore 0. Se il controllore e il dispositivo sono veloci, questo metodo è ragionevole, ma se l'attesa rischia di prolungarsi, sarebbe probabilmente meglio se la CPU si dedicasse a un'altra operazione. In questo caso si pone il problema di come la CPU possa sapere quando il controllore è tornato libero. È necessario che la CPU serva certi tipi di dispositivi rapidamente, o si potrebbero perdere alcuni dati. Quando, per esempio, i dati affluiscono in una porta seriale o dalla tastiera, il piccolo buffer del controllore diverrà presto pieno, e se la CPU attende troppo a lungo prima di riprendere la lettura dei byte, si perderanno informazioni.

In molte architetture di calcolatori sono sufficienti tre istruzioni della CPU per effettuare il polling di un dispositivo: `read`, lettura di un registro del dispositivo; `logical-and`, usata per estrarre il valore di un bit di stato, e `branch`, salto a un altro punto del codice se l'argomento è diverso da zero. Chiaramente, il polling è in sé un'operazione efficiente; tale tecnica diviene però inefficiente se le ripetute interrogazioni trovano raramente un dispositivo pronto per il servizio, mentre altre utili elaborazioni attendono la CPU. In tali casi, anziché richiedere alla CPU di eseguire il polling, può essere più efficiente far sì che il controllore comunichi alla CPU che il dispositivo è pronto. Il meccanismo hardware che permette tale comunicazione si chiama **interruzione** (*interrupt*).

13.2.2 Interruzioni

Il meccanismo di base dell'interruzione funziona come segue. L'hardware della CPU ha un input, detto **linea di richiesta dell'interruzione**, del quale la CPU controlla lo stato dopo l'esecuzione di ogni istruzione. Quando rileva il segnale di un controllore sulla linea di richiesta dell'interruzione, la CPU salva lo stato corrente e salta alla **routine di gestione dell'interruzione** (*interrupt-handler routine*), che si trova a un indirizzo prefissato di memoria. Questa procedura determina le cause dell'interruzione, porta a termine l'elaborazione necessaria, ripristina lo stato ed esegue un'istruzione `return from interrupt` per far sì che la CPU ritorni nello stato in cui si trovava prima della sua interruzione. Il controllore del dispositivo *genera* un'interruzione della CPU sulla linea di richiesta delle interruzioni, che la CPU *rileva e recapita*

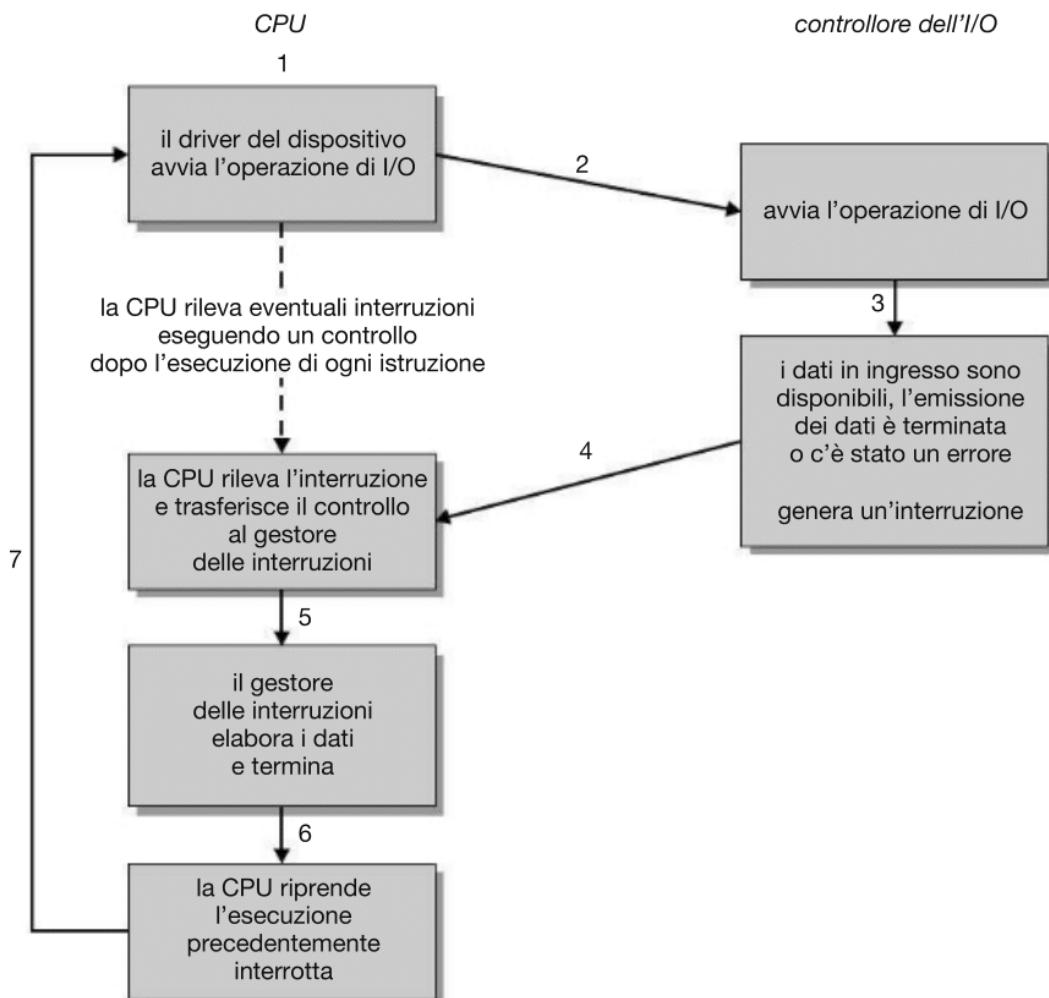


Figura 13.3 Ciclo di I/O basato sulle interruzioni.

al gestore delle interruzioni, che a sua volta *gestisce* l'interruzione corrispondente servendo il dispositivo. Nella Figura 13.3 è riassunto il ciclo di I/O causato da un'interruzione della CPU. In questo capitolo daremo molta importanza alla gestione delle interruzioni, perché persino un sistema a singola utenza ne gestisce centinaia al secondo, mentre un server ne può gestire centinaia di migliaia.

Il meccanismo di base delle interruzioni che abbiamo appena descritto permette alla CPU di rispondere a un evento asincrono, come quello di un controllore di un dispositivo che divenga pronto per essere servito. Nei sistemi operativi moderni sono necessarie capacità di gestione delle interruzioni più raffinate.

1. Si deve poter posporre la gestione delle interruzioni durante le fasi critiche dell'elaborazione.
2. Si deve disporre di un meccanismo efficiente per passare il controllo all'appropriato gestore delle interruzioni, senza dover esaminare ciclicamente tutti i dispositivi (*polling*) per determinare quale abbia generato l'interruzione.

3. Si deve disporre di più livelli d'interruzione, di modo che il sistema possa distinguere le interruzioni ad alta priorità da quelle a priorità inferiore, servendo le richieste con la celerità appropriata al caso.

In un calcolatore moderno queste tre caratteristiche sono fornite dalla CPU e dal **controllore hardware delle interruzioni**.

La maggior parte delle CPU ha due linee di richiesta delle interruzioni. Una è quella delle **interruzioni non mascherabili**, riservata a eventi quali gli errori di memoria irrecuperabili. La seconda linea è quella delle **interruzioni mascherabili**: può essere disattivata dalla CPU prima dell'esecuzione di una sequenza critica di istruzioni che non deve essere interrotta. L'interruzione mascherabile è usata dai controllori dei dispositivi per richiedere un servizio.

Il meccanismo delle interruzioni accetta un **indirizzo** – un numero che seleziona una specifica procedura di gestione delle interruzioni da un insieme ristretto. Nella maggior parte delle architetture questo indirizzo è uno scostamento relativo in una tabella detta **vettore delle interruzioni**, contenente gli indirizzi di memoria degli specifici gestori delle interruzioni. Lo scopo di un meccanismo vettorizzato di gestione delle interruzioni è di ridurre la necessità che un singolo gestore debba individuare tutte le possibili fonti d'interruzione per determinare quale di loro abbia richiesto un servizio. In pratica, tuttavia, i calcolatori hanno più dispositivi (e quindi, più gestori delle interruzioni) che elementi nel vettore delle interruzioni. Una maniera diffusa di risolvere questo problema consiste nel **concatenamento delle interruzioni** (*interrupt chaining*), in cui ogni elemento del vettore delle interruzioni punta alla testa di una lista di gestori delle interruzioni. Quando si verifica un'interruzione, si chiamano uno alla volta i gestori nella lista corrispondente finché non se ne trova uno che può soddisfare la richiesta. Questa struttura è un compromesso fra l'overhead di una tabella delle interruzioni enorme e l'inefficienza dell'uso di un solo gestore delle interruzioni.

Nella Figura 13.4 è descritto il vettore delle interruzioni della CPU Intel Pentium. Gli eventi da 0 a 31, non mascherabili, si usano per segnalare varie condizioni d'errore; quelli dal 32 al 255, mascherabili, si usano, per esempio, per le interruzioni generate dai dispositivi.

Il meccanismo delle interruzioni realizza anche un sistema di **livelli di priorità delle interruzioni**. Esso permette alla CPU di differire la gestione delle interruzioni di bassa priorità senza mascherare tutte le interruzioni, e permette a un'interruzione di priorità alta di sospendere l'esecuzione della procedura di servizio di un'interruzione di priorità bassa.

Un sistema operativo moderno interagisce con il meccanismo delle interruzioni in vari modi. All'accensione della macchina esamina i bus per determinare quali dispositivi siano presenti, e installa gli indirizzi dei corrispondenti gestori delle interruzioni nel vettore delle interruzioni. Durante l'I/O, i vari controllori di dispositivi generano le interruzioni della CPU quando sono pronti per un servizio. Queste interruzioni significano che è stato completato un output, o che sono disponibili dati in ingresso, o che un'operazione non è andata a buon fine. Il meccanismo delle interruzioni si usa anche per gestire un'ampia gamma di **eccezioni**, come la divisione

indice del vettore	descrizione
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19-31	(Intel reserved, do not use)
32-255	maskable interrupts

Figura 13.4 Vettore delle interruzioni della CPU Intel Pentium.

per 0, l'accesso a indirizzi di memoria protetti o inesistenti o il tentativo di eseguire un'istruzione privilegiata in modalità utente. Gli eventi che producono le interruzioni hanno una proprietà in comune: inducono il sistema operativo a eseguire urgentemente una procedura autonoma.

Un sistema operativo può fare altri usi proficui di un efficiente meccanismo hardware e software che memorizza una piccola quantità d'informazioni sullo stato della CPU e poi richiama una procedura del kernel. Per esempio, molti sistemi operativi usano il meccanismo delle interruzioni per la gestione della memoria virtuale. Un'eccezione di pagina mancante genera un'interruzione che sospende il processo corrente e trasferisce il controllo dell'esecuzione al relativo gestore nel kernel. Tale gestore memorizza le informazioni sullo stato del processo, lo sposta nella coda d'attesa, compie le necessarie operazioni di gestione della cache delle pagine, avvia un'operazione di I/O per prelevare la pagina giusta, schedula la ripresa dell'esecuzione di un altro processo e ritorna dall'interruzione.

Un altro esempio è dato dall'implementazione delle *chiamate di sistema*. Solitamente i programmi sfruttano routine di libreria per eseguire chiamate di sistema. La routine controlla i parametri passati dall'applicazione, li assembla in una struttura dati appropriata da passare al kernel, e infine esegue una particolare istruzione detta **interruzione software** o **trap**. Questa istruzione ha un operando che identifica il servizio del kernel desiderato. Quando un processo esegue l'istruzione di eccezione, l'hardware delle interruzioni salva lo stato del codice utente, passa al modo kernel e recapita l'interruzione alla procedura del kernel che realizza il servizio richiesto.

A una trap si assegna una priorità di interruzione relativamente bassa rispetto a quelle date alle interruzioni dei dispositivi – eseguire una chiamata di sistema per conto di un'applicazione è meno urgente di quanto non sia servire un controllore prima che la sua coda FIFO trabocchi causando la perdita di informazioni.

Le interruzioni si possono inoltre usare per gestire il flusso di controllo all'interno del kernel. Si consideri un esempio di elaborazione richiesta per completare una lettura da un disco. Un passo necessario è quello di copiare dati dallo spazio del kernel al buffer dell'utente. Questa azione richiede tempo, ma non è urgente, e non dovrebbe bloccare la gestione delle interruzioni con priorità più alta. Un altro passo è quello di avviare il successivo I/O in attesa relativo a quell'unità a disco. Questo passo ha priorità più alta: se le unità a disco si devono usare in modo efficiente, è necessario avviare l'evasione della successiva richiesta di I/O non appena la precedente sia stata soddisfatta. Di conseguenza una *coppia* di gestori delle interruzioni realizza il codice del kernel che compie le letture dai dischi. Il gestore ad alta priorità mantiene le informazioni sullo stato dell'I/O, risponde al segnale d'interruzione del dispositivo, avvia il prossimo I/O in attesa e genera un'interruzione a bassa priorità per completare il lavoro. Più tardi, in un momento in cui la CPU non è occupata in compiti ad alta priorità, si serve l'interruzione a bassa priorità. Il gestore corrispondente completa l'I/O a livello utente copiando i dati dal buffer del kernel a quello dell'applicazione, e richiamando poi lo scheduler per aggiungere l'applicazione alla coda dei processi pronti.

Un'architettura del kernel basata su thread è adatta alla realizzazione di più livelli di priorità delle interruzioni, e a dare la precedenza alla gestione delle interruzioni rispetto alle elaborazioni in background delle procedure del kernel e delle applicazioni. Questo concetto si può esemplificare considerando il kernel del sistema operativo Solaris. In questo sistema i gestori delle interruzioni si eseguono come thread del kernel cui si riservano valori elevati di priorità. Tali priorità garantiscono la precedenza dei gestori delle interruzioni rispetto al codice delle applicazioni e al lavoro ordinario del kernel, e inoltre realizzano le necessarie relazioni di priorità fra i diversi gestori delle interruzioni. Le priorità fanno sì che lo scheduler dei thread di Solaris sospenda i gestori delle interruzioni di bassa priorità a vantaggio di quelli di priorità più alta, e la realizzazione basata su thread permette alle architetture multiprocessore di eseguire parallelamente diversi gestori delle interruzioni. L'architettura delle interruzioni dei sistemi Windows è descritta nel Capitolo 19, presente sulla pagina web del volume.

Riassumendo, le interruzioni sono usate diffusamente dai sistemi operativi moderni per gestire eventi asincroni e per eseguire procedure in modalità supervisore nel kernel. Per far sì che i compiti più urgenti siano portati a termine per primi, i calcolatori moderni usano un sistema di priorità delle interruzioni. I controllori dei dispositivi, i guasti hardware e le chiamate di sistema generano interruzioni al fine di innescare l'esecuzione di procedure del kernel. Poiché le interruzioni sono usate in modo massiccio per affrontare situazioni in cui il tempo è un fattore critico, è necessario avere un'efficiente gestione delle interruzioni per ottenere buone prestazioni del sistema.

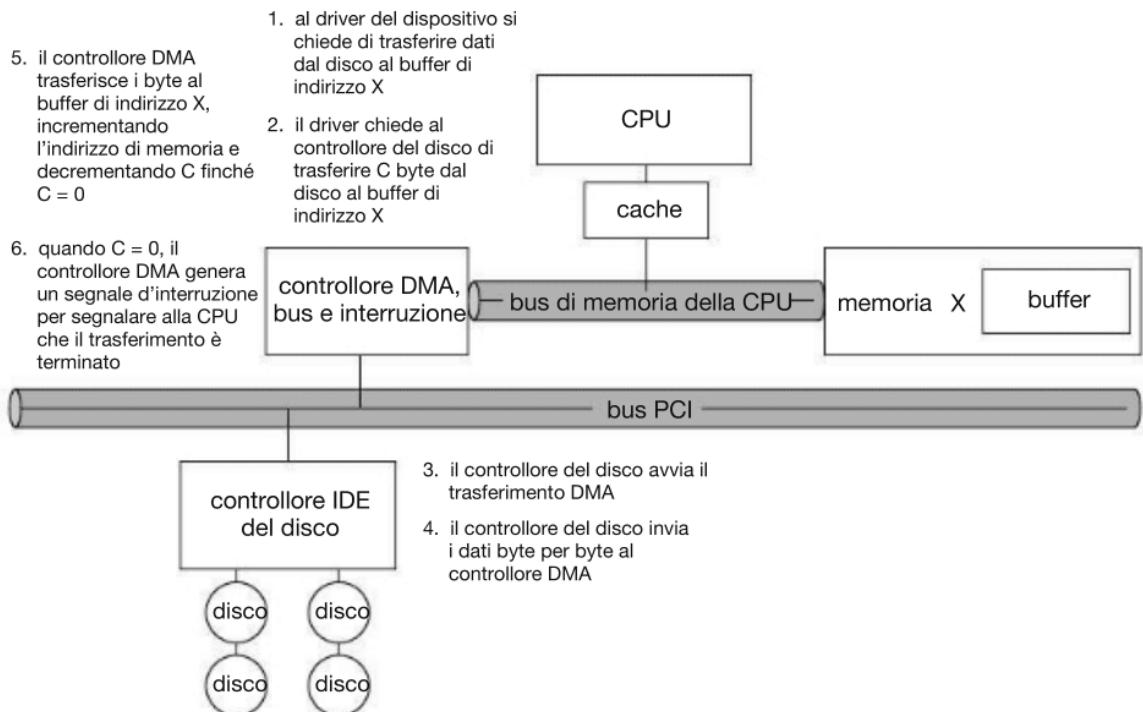
13.2.3 Accesso diretto alla memoria (DMA)

Quando un dispositivo compie trasferimenti di grandi quantità di dati, come nel caso di un'unità a disco, l'uso di una costosa CPU per il controllo dei bit di stato e per la scrittura di dati nel registro del controllore un byte alla volta, detto **I/O programmato** (*programmed I/O, PIO*), sembra essere uno spreco. In molti calcolatori si evita di sovraccaricare la CPU assegnando una parte di questi compiti a un processore specializzato, detto controllore dell'**accesso diretto alla memoria** (*direct memory-access, DMA*). Per dar avvio a un trasferimento DMA, la CPU scrive in memoria un blocco di comando per il DMA. Esso contiene un puntatore alla locazione dei dati da trasferire, un altro puntatore alla destinazione dei dati, e il numero dei byte da trasferire. La CPU scrive l'indirizzo di questo blocco di comando nel controllore del DMA, e prosegue con altre attività. Il controllore DMA agisce quindi direttamente sul bus della memoria, presentando al bus gli indirizzi di memoria necessari per eseguire il trasferimento senza l'aiuto della CPU. Un semplice controllore DMA è un componente standard in tutti i sistemi moderni, dagli smartphone ai mainframe.

L'handshaking tra il controllore del DMA e il controllore del dispositivo si svolge grazie a una coppia di fili detti **DMA-request** e **DMA-acknowledge**. Il controllore del dispositivo manda un segnale sulla linea **DMA-request** quando una word di dati è disponibile per il trasferimento. Questo segnale fa sì che il controllore DMA prenda possesso del bus di memoria, presenti l'indirizzo desiderato ai fili d'indirizzamento della memoria e mandi un segnale lungo la linea **DMA-acknowledge**. Quando il controllore del dispositivo riceve questo segnale, trasferisce in memoria la word di dati e rimuove il segnale dalla linea **DMA-request**.

Quando l'intero trasferimento termina, il controllore del DMA interrompe la CPU. Nella Figura 13.5 è rappresentato questo processo. Quando il controllore del DMA prende possesso del bus di memoria, la CPU è temporaneamente impossibilitata ad accedere alla memoria centrale, sebbene abbia accesso ai dati contenuti nella sua cache primaria e secondaria. Questo fenomeno, noto come **sottrazione di cicli**, può rallentare le computazioni della CPU; ciononostante l'assegnamento del lavoro di trasferimento di dati a un controllore DMA migliora in generale le prestazioni complessive del sistema. In alcune architetture per realizzare la tecnica DMA si usano gli indirizzi della memoria fisica, mentre in altre s'impiega l'**accesso diretto alla memoria virtuale** (*direct virtual-memory access, DVMA*): in questo caso si usano indirizzi virtuali che poi si traducono in indirizzi fisici. La tecnica DVMA permette di compiere i trasferimenti di dati tra due dispositivi che eseguono I/O memory mapped senza far intervenire la CPU o accedere alla memoria centrale.

Nei kernel che operano in modalità protetta, in genere il sistema operativo non permette ai processi di impartire direttamente comandi ai dispositivi. Ciò protegge i dati dalle violazioni dei controlli d'accesso e il sistema da un eventuale uso scorretto dei controllori dei dispositivi che potrebbe portare a una caduta del sistema stesso. Il sistema operativo invece esporta delle funzioni di I/O che possono essere eseguite da processi sufficientemente privilegiati per effettuare operazioni di basso livello sull'hardware sottostante. Quando invece il kernel non può garantire la protezione della

**Figura 13.5** Passi di un trasferimento DMA.

memoria, i processi hanno accesso diretto ai controllori dei dispositivi. Questo accesso diretto si può utilizzare in modo da ottenere buone prestazioni, perché evita la comunicazione col kernel, i cambi di contesto e l'interazione fra diversi livelli del kernel. Purtroppo interferisce con la stabilità e la sicurezza del sistema. La tendenza comune per i sistemi operativi d'uso generale è quella di proteggere la memoria e i dispositivi in modo da salvaguardarli da applicazioni accidentalmente o volutamente dannose.

13.2.4 Concetti principali dell'hardware di I/O

Sebbene gli aspetti dell'I/O che riguardano i dispositivi siano complessi se si analizzano tanto dettagliatamente quanto farebbe un progettista elettronico, i concetti appena descritti sono sufficienti per comprendere molti aspetti dell'I/O per ciò che concerne i sistemi operativi. Ecco un sommario dei concetti principali:

- bus;
- controllore;
- porta di I/O e suoi registri;
- procedura di handshaking tra la CPU e il controllore di un dispositivo;
- esecuzione dell'handshaking per mezzo del polling o delle interruzioni;
- delega dell'I/O a un controllore DMA nel caso di trasferimenti di grandi quantità di dati.

Precedentemente in questo paragrafo è stato fornito un esempio dell'*handshaking* che avviene tra un controllore di dispositivo e un host. In realtà, la grande varietà di dispositivi esistenti pone un problema a chi voglia realizzare concretamente un sistema operativo. Ogni tipo di dispositivo ha proprie funzionalità, proprie definizioni dei bit di controllo, e un proprio protocollo per l'interazione con la macchina – e tutto ciò varia da dispositivo a dispositivo. Come deve essere progettato un sistema operativo affinché sia possibile collegare al calcolatore nuovi dispositivi senza che sia necessario riscrivere il sistema operativo stesso? E inoltre, vista la grande varietà di dispositivi, come può il sistema operativo fornire alle applicazioni un'interfaccia per l'I/O uniforme ed efficace? Discuteremo questi aspetti nel seguito.

13.3 Interfaccia di I/O per le applicazioni

In questo paragrafo si discutono le tecniche e le interfacce di un sistema operativo che permettono un trattamento standardizzato e uniforme dei dispositivi di I/O. Si spiega, per esempio, come un'applicazione possa aprire un file residente in un disco senza sapere di che tipo di disco si tratti, e come si possano aggiungere al calcolatore nuove unità a disco e altri dispositivi senza che si debba modificare il sistema operativo.

I metodi qui esposti coinvolgono l'astrazione, l'incapsulamento e la stratificazione del software. In particolare, si può effettuare un'astrazione rispetto ai dettagli delle differenze tra i dispositivi per l'I/O identificandone alcuni tipi generali. A ognuno di questi tipi si accede per mezzo di un insieme standardizzato di funzioni – un'**interfaccia**. Le differenze sono incapsulate in moduli del kernel detti driver dei dispositivi, specializzati internamente per gli specifici dispositivi, ma che comunicano con l'esterno per mezzo delle interfacce uniformi. Nella Figura 13.6 è illustrata la divisione in strati software di quelle parti del kernel che riguardano la gestione dell'I/O.

Lo scopo dello strato dei driver dei dispositivi è di nascondere al sottosistema di I/O del kernel le differenze fra i controllori dei dispositivi, in modo simile a quello con cui le chiamate di sistema di I/O incapsulano il comportamento dei dispositivi in alcune classi generiche che nascondono le differenze hardware alle applicazioni. Il fatto che così il sottosistema di I/O sia reso indipendente dall'hardware semplifica il lavoro di chi sviluppa il sistema operativo, e va inoltre a vantaggio dei costruttori dei dispositivi. Questi, infatti, o progettano i nuovi dispositivi in modo tale che siano compatibili con un'interfaccia host-controllore già esistente (per esempio SATA), oppure scrivono driver che permettano ai nuovi dispositivi di essere gestiti dai sistemi operativi più diffusi. In questo modo, nuovi dispositivi sono utilizzabili da un calcolatore senza che occorra attendere lo sviluppo del codice di supporto da parte del produttore del sistema operativo.

Sfortunatamente per i produttori di dispositivi, ogni tipo di sistema operativo ha le sue convenzioni riguardanti l'interfaccia dei driver dei dispositivi. Così, un dato dispositivo potrà essere venduto con molti driver diversi – per esempio, driver per Windows, Linux, AIX e Mac OS X. I dispositivi (Figura 13.7) possono differire in molti aspetti.

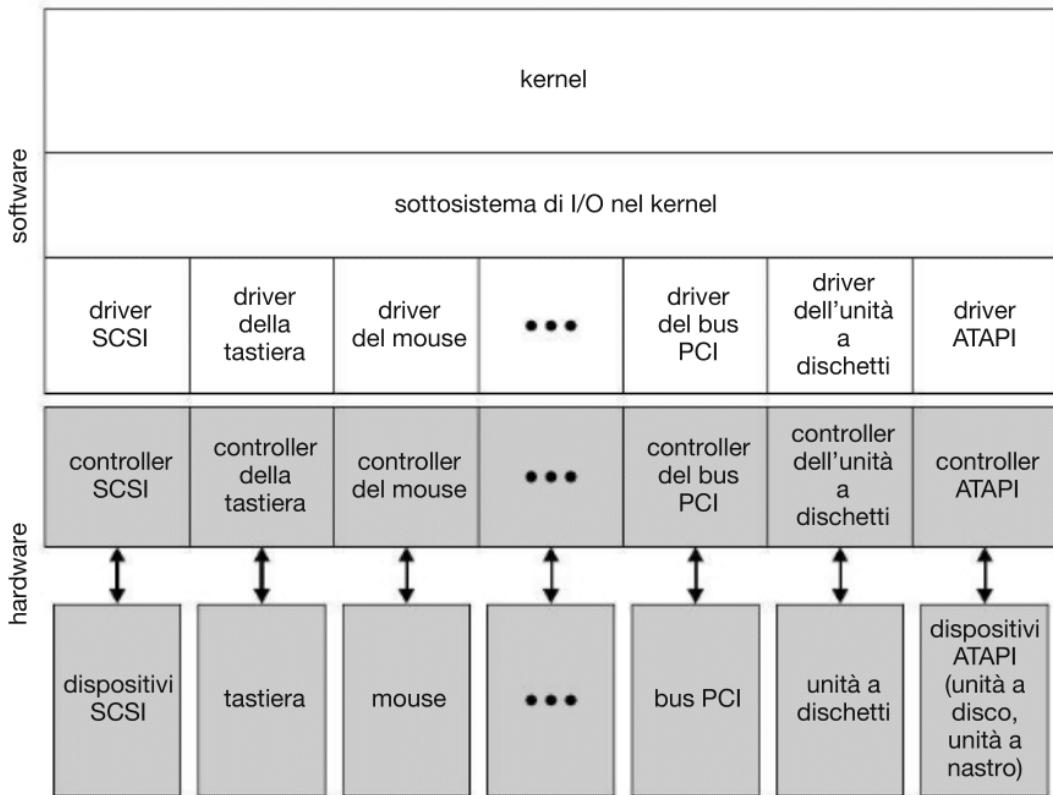


Figura 13.6 Struttura relativa all'I/O nel kernel.

- **Trasferimento a flusso di caratteri o a blocchi.** Un dispositivo a flusso di caratteri trasferisce dati un byte alla volta, mentre uno a blocchi trasferisce un blocco di byte in un'unica soluzione.
- **Sequenziale o accesso diretto.** Un dispositivo sequenziale trasferisce dati secondo un ordine fisso dipendente dal dispositivo, mentre l'utente di un dispositivo ad accesso diretto può richiedere l'accesso a una qualunque delle possibili locazioni di memorizzazione.
- **Dispositivi sincroni o asincroni.** Un dispositivo sincrono trasferisce dati con un tempo di risposta prevedibile, in maniera coordinata rispetto al resto del sistema. Un dispositivo asincrono ha tempi di risposta irregolari o non prevedibili, non coordinati con gli altri eventi del computer.
- **Condivisibili o dedicati.** Un dispositivo condivisibile può essere usato in modo concorrente da diversi processi o thread, mentre ciò è impossibile se il dispositivo è dedicato.
- **Velocità di funzionamento.** Può variare da alcuni byte al secondo fino a qualche gigabyte al secondo.
- **Lettura e scrittura, sola lettura o sola scrittura.** Alcuni dispositivi possono emettere e ricevere dati, ma altri possono trasferire dati in una sola direzione.

aspetto	variazione	esempio
modalità di trasferimento dei dati	a caratteri a blocchi	terminale unità a disco
modalità d'accesso	sequenziale casuale	modem lettore di CD-ROM
prevedibilità dell'I/O	sincrono asincrono	unità a nastro tastiera
condivisione	dedicato condiviso	unità a nastro tastiera
velocità	latenza tempo di ricerca velocità di trasferimento attesa fra le operazioni	
direzione dell'I/O	solo lettura solo scrittura lettura e scrittura	lettore di CD-ROM controllore della grafica unità a disco

Figura 13.7 Caratteristiche dei dispositivi per l'I/O.

Per ciò che riguarda l'accesso delle applicazioni ai dispositivi, molte di queste differenze sono nascoste dal sistema operativo, e i dispositivi sono raggruppati in poche classi convenzionali. Si è riscontrato che i modi d'accesso ai dispositivi che ne risultano sono utili e largamente applicabili. Anche se la forma precisa delle chiamate di sistema può variare nei diversi sistemi operativi, le classi di dispositivi sono abbastanza regolari. Le convenzioni d'accesso principali includono l'I/O a blocchi, l'I/O a flusso di caratteri, l'accesso ai file mappati in memoria, e le socket di rete. I sistemi operativi forniscono anche chiamate di sistema speciali per l'accesso a qualche dispositivo aggiuntivo, per esempio un orologio o un timer. Qualche sistema operativo mette a disposizione un insieme di chiamate di sistema per display grafici, video e audio.

La maggior parte dei sistemi ha anche una “via di fuga” (*escape*) o *back door* che permette il passaggio trasparente di comandi arbitrari da un'applicazione a un driver di dispositivo. In UNIX tale funzione è svolta dalla chiamata di sistema `ioctl()` (che sta per *I/O control*) e consente a un'applicazione di impiegare qualsiasi funzionalità fornita da qualsiasi driver di dispositivo, senza che per questo sia necessario creare nuove chiamate di sistema. Gli argomenti di `ioctl()` sono tre: il primo è un descrittore di file che collega l'applicazione al driver desiderato facendo riferimento a un dispositivo gestito da quel driver; il secondo è un numero intero che seleziona uno dei comandi forniti dal driver; il terzo argomento, infine, è un puntatore a un'arbitraria struttura dati in memoria, tramite la quale l'applicazione e il driver si scambiano le informazioni di controllo o i dati necessari.

13.3.1 Dispositivi con trasferimento a blocchi e a caratteri

L’interfaccia per i **dispositivi a blocchi** sintetizza tutti gli aspetti necessari per accedere alle unità a disco e ad altri dispositivi basati sul trasferimento di blocchi di dati. Ci si aspetta che il dispositivo comprenda istruzioni come `read()` e `write()` e, nel caso sia ad accesso casuale, anche un comando `seek()` per specificare il blocco successivo da trasferire. Di solito le applicazioni comunicano con questi dispositivi tramite un’interfaccia di file system. Si vede che `read()`, `write()` e `seek()` catturano l’essenza del comportamento dei dispositivi con trasferimento a blocchi, in modo che le applicazioni non vedano le differenze di basso livello fra questi dispositivi.

Il sistema operativo e certe applicazioni particolari come quelle per la gestione delle basi di dati possono trovare più conveniente trattare questi dispositivi come una semplice sequenza lineare di blocchi. In questo caso si parla di **I/O a basso livello** (*raw I/O*). L’uso del file system da parte delle applicazioni che gestiscono già in proprio un buffer per l’I/O comporta l’inutile intervento di un buffer aggiuntivo. Analogamente, l’uso dei lock da parte del sistema operativo nei confronti di applicazioni che già implementano meccanismi per la mutua esclusione relativi ai blocchi dei file risulta ridondante, o peggio contraddittorio. Per superare tali potenziali conflitti, l’I/O a basso livello passa il controllo del dispositivo direttamente all’applicazione, togliendo così di mezzo il sistema operativo. Purtroppo, ciò significa anche che non risulta disponibile sul dispositivo in questione alcun servizio del sistema operativo. Un compromesso che si sta diffondendo sempre più è fornire una modalità d’accesso ai file che disabiliti il buffer e i meccanismi di gestione dei lock; nel mondo UNIX si parla di **I/O diretto**.

L’accesso ai file mappato in memoria può costituire uno strato software sopra i driver dei dispositivi a blocchi. Piuttosto che offrire funzioni di lettura e scrittura, un’interfaccia di mappaggio in memoria fornisce la possibilità di accedere a un’unità a disco tramite un vettore di byte della memoria centrale. La chiamata di sistema che associa un file a una regione di memoria restituisce l’indirizzo di memoria virtuale di una copia del file. Gli effettivi trasferimenti di dati sono eseguiti solo quando necessari per soddisfare una richiesta d’accesso all’immagine in memoria. Poiché i trasferimenti si trattano nello stesso modo in cui si gestisce l’accesso su richiesta a una pagina di memoria virtuale, l’I/O memory mapped è efficiente. Esso è inoltre conveniente per i programmati perché l’accesso a un file memory mapped è semplice tanto quanto la lettura e la scrittura in memoria. I sistemi operativi che gestiscono la memoria virtuale utilizzano comunemente l’interfaccia di mappaggio in memoria per i servizi del kernel. Per esempio, quando il sistema operativo deve eseguire un programma, mappa l’eseguibile in memoria, quindi trasferisce il controllo all’indirizzo iniziale. Questo tipo d’interfaccia è spesso usato anche per l’accesso del kernel all’area di swapping nei dischi.

La tastiera è un esempio di dispositivo al quale si accede tramite un’interfaccia a **flusso di caratteri**. Le chiamate di sistema fondamentali per le interfacce di questo tipo permettono a un’applicazione di acquisire (`get()`) o inviare (`put()`) un carat-

tere. Basandosi su quest’interfaccia è possibile costruire librerie che offrono l’accesso riga per riga, con buffering ed editing (per esempio, quando l’utente preme il tasto backspace, si rimuove il carattere precedente dalla sequenza di caratteri da inserire). Questo tipo d’accesso è conveniente per dispositivi come tastiere, mouse, modem, che producono dati “spontaneamente”, cioè in momenti che non possono essere sempre previsti dalle applicazioni. Lo stesso tipo d’accesso è adatto anche ai dispositivi che emettono dati organizzati in modo naturale come sequenza lineare di byte, per esempio le stampanti o le schede audio.

13.3.2 Dispositivi di rete

Poiché i modi di indirizzamento e le prestazioni tipiche dell’I/O di rete sono notevolmente differenti da quelli dell’I/O delle unità a disco, la maggior parte dei sistemi operativi fornisce un’interfaccia per l’I/O di rete diversa da quella caratterizzata dalle operazioni `read()`, `write()` e `seek()` usata per i dischi. Un’interfaccia disponibile in molti sistemi operativi, tra i quali UNIX e Windows NT, è l’interfaccia di rete **socket** (*presa di corrente*).

Si pensi a una presa di corrente elettrica a muro: vi si può collegare qualunque apparecchiatura elettrica; per analogia, le chiamate di sistema di un’interfaccia socket permettono a un’applicazione di creare una socket, collegare una socket locale all’indirizzo di un altro punto della rete (ciò ha l’effetto di collegare questa applicazione alla socket creata da un’altra applicazione), controllare se un’applicazione si inserisce nella socket locale, e inviare o ricevere pacchetti di dati lungo la connessione. Per supportare lo sviluppo di server, l’interfaccia socket fornisce anche una funzione chiamata `select()` che gestisce un insieme di socket. Essa restituisce informazioni sulle socket per le quali sono presenti pacchetti che attendono d’essere ricevuti, e su quelle che hanno spazio per accettare un pacchetto da inviare. L’uso della funzione `select()` elimina il polling e il busy waiting altrimenti necessari per l’I/O di rete. Queste funzioni incapsulano il comportamento essenziale delle reti, facilitando notevolmente la creazione di applicazioni distribuite che possano sfruttare qualsiasi hardware di rete e stack di protocolli.

Sono stati sviluppati molti altri metodi per affrontare il problema della comunicazione di rete e fra i processi. Il sistema operativo Windows, per esempio, fornisce un’interfaccia per la scheda di rete e un’altra per i protocolli di rete. Il sistema operativo UNIX, banco di prova storico delle tecnologie di rete, offre pipe *half-duplex*, code FIFO *full-duplex*, STREAMS *full-duplex*, code di messaggi e socket.

13.3.3 Orologi e timer

La maggior parte dei calcolatori ha timer e orologi hardware che forniscono tre funzioni essenziali:

- segnare l’ora corrente;
- segnalare il tempo trascorso;
- regolare un timer in modo da avviare l’operazione x al tempo t .

Queste funzioni sono spesso usate sia dal sistema operativo sia da applicazioni per cui il tempo è un fattore importante. Purtroppo, le chiamate di sistema che realizzano queste funzioni non sono standardizzate da un sistema operativo all’altro.

Il dispositivo che misura la durata di un lasso di tempo e che può avviare un’operazione si chiama **timer programmabile**; si può regolare in modo da attendere un certo tempo e poi generare un’interruzione, e può anche ripetere questo processo continuamente, generando così interruzioni periodiche. Lo scheduler usa questo meccanismo per generare un’interruzione che sospende un processo quando il suo quanto di tempo è scaduto. Il sottosistema dell’I/O delle unità a disco lo usa per riversare periodicamente nei dischi il contenuto della *buffer cache*, e il sottosistema di rete lo usa per annullare operazioni che procedono troppo lentamente a causa di congestioni di rete o guasti. Il sistema operativo può inoltre fornire un’interfaccia per permettere ai processi utenti di usare i timer. Simulando orologi virtuali, il sistema operativo può anche gestire un numero di richieste d’uso dei timer maggiore del numero dei timer fisici. Per far ciò il kernel (o il driver del timer) mantiene una lista ordinata cronologicamente delle interruzioni richieste dagli utenti e dalle proprie procedure, e imposta il timer per la prima scadenza. Quando il timer genera l’interruzione, il kernel manda un segnale al richiedente, e reimposta il timer per la scadenza successiva.

In molti calcolatori, la frequenza delle interruzioni generate dall’orologio è fra le 18 e le 60 al secondo. Ciò costituisce un grado di precisione non sufficientemente fine per un calcolatore moderno che può eseguire centinaia di milioni di istruzioni al secondo. La precisione degli impulsi d’attivazione è limitata dalla bassa frequenza del timer, e dall’overhead aggiuntivo dato dal mantenimento di orologi virtuali. Inoltre, se lo stesso timer si usa per fornire l’ora corrente del sistema, questa potrà derivare. Nella maggior parte dei calcolatori, l’orologio hardware è costruito sulla base di un contatore ad alta frequenza. In alcuni casi, è possibile leggere da un registro del dispositivo il valore di questo contatore, cosicché esso può essere visto come un orologio ad alta precisione. Sebbene non sia in grado di generare interruzioni, offre una misura accurata degli intervalli di tempo.

13.3.4 I/O non bloccante e asincrono

Un altro aspetto dell’interfaccia delle chiamate di sistema è la scelta fra I/O bloccante e non bloccante. Quando un’applicazione impiega una chiamata di sistema **bloccante** si sospende l’esecuzione dell’applicazione, che passa dalla coda dei processi pronti per l’esecuzione a una coda d’attesa del sistema. Quando la chiamata di sistema termina, l’applicazione è posta nuovamente nella coda dei processi pronti in modo che possa riprendere l’esecuzione; Quando riprenderà l’esecuzione, riceverà i valori riportati dalla chiamata di sistema. Le operazioni fisiche compiute dai dispositivi di I/O sono in genere asincrone – richiedono un tempo variabile o non prevedibile. Cionondimeno, la maggior parte dei sistemi operativi impiega chiamate di sistema bloccanti come interfaccia per le applicazioni, perché in questo caso il codice delle applicazioni è più facilmente comprensibile del corrispondente codice non bloccante.

Alcuni processi a livello utente necessitano di una forma **non bloccante** di I/O. Un esempio è quello di un'interfaccia utente con cui s'interagisce col mouse e la tastiera mentre elabora dati e li mostra sullo schermo. Un altro esempio è un'applicazione video che legge frame da un file su disco e simultaneamente li decomprime e li mostra sullo schermo.

Uno dei modi in cui chi progetta un'applicazione può sovrapporre elaborazione e I/O è scrivere un'applicazione a più thread. Alcuni di loro eseguono chiamate di sistema bloccanti, mentre altri continuano l'elaborazione. Alcuni sistemi operativi forniscono chiamate di sistema non bloccanti per l'I/O. Una chiamata di questo tipo non arresta l'esecuzione dell'applicazione per un tempo significativo. Al contrario, essa restituisce rapidamente il controllo all'applicazione, fornendo un parametro che indica quanti byte di dati sono stati trasferiti.

Una possibile alternativa alle chiamate di sistema non bloccanti è costituita dalle chiamate di sistema asincrone. Esse restituiscono immediatamente il controllo al chiamante, senza attendere che l'I/O sia stato completato. L'applicazione continua a essere eseguita, e il completamento dell'I/O è successivamente comunicato all'applicazione per mezzo dell'impostazione del valore di una variabile nello spazio d'indirizzi dell'applicazione oppure tramite la generazione di un segnale o interrupt software, o ancora tramite una procedura di richiamo (*callback*) eseguita fuori del normale flusso lineare d'elaborazione dell'applicazione. La differenza fra chiamate di sistema non bloccanti e asincrone è che una `read()` non bloccante restituisce immediatamente il controllo, fornendo i dati che è stato possibile leggere (l'intero numero di byte richiesti, una parte, o anche nessun dato). Una chiamata `read()` asincrona richiede un trasferimento di cui il sistema garantisce il completamento, ma solo in un momento successivo e non prevedibile. Entrambi i metodi sono illustrati dalla Figura 13.8.

In tutti i moderni sistemi operativi si verificano attività asincrone. Spesso queste attività non sono gestite da utenti o applicazioni, ma fanno parte del funzionamento del sistema operativo. Due validi esempi sono l'I/O del disco e della rete. Di norma,

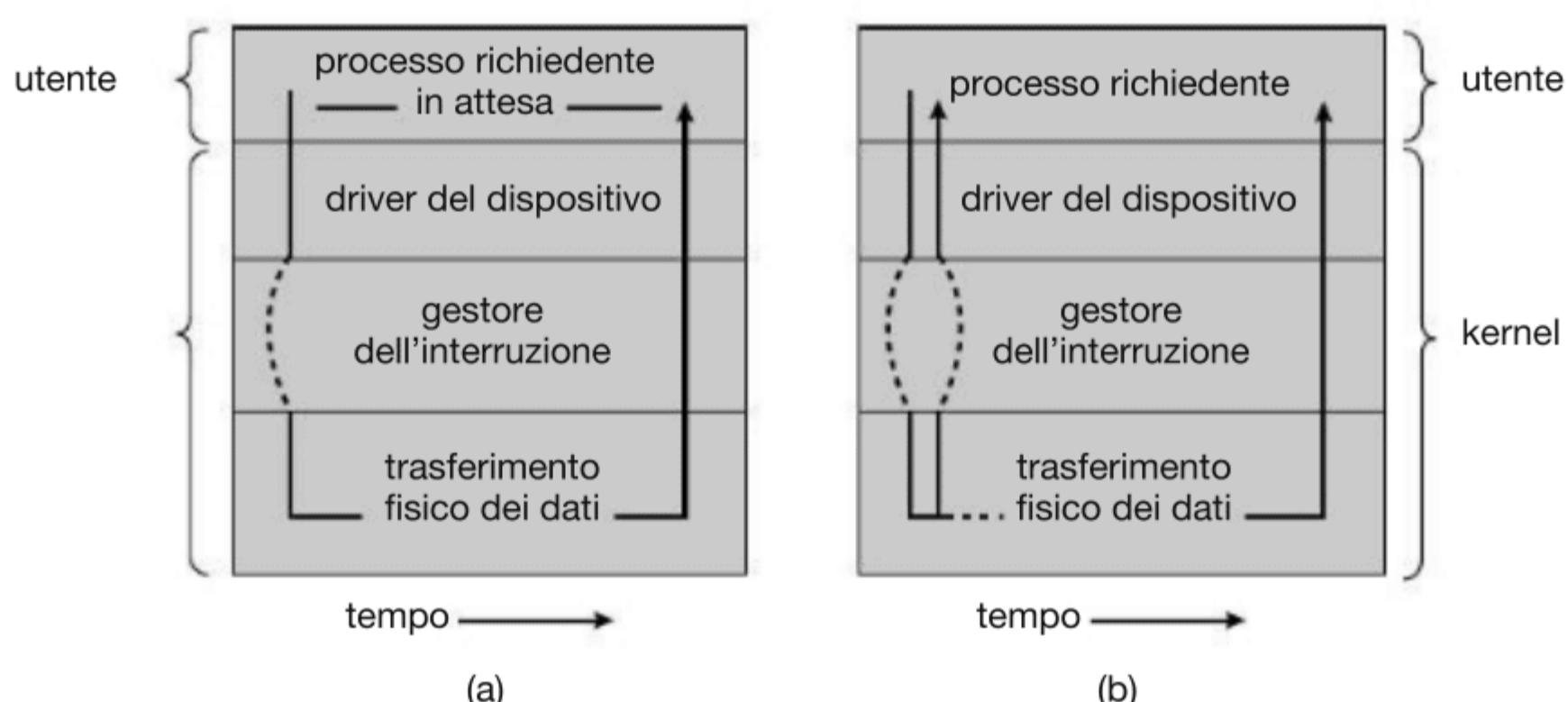


Figura 13.8 Due metodi per l'I/O; (a) sincrono e (b) asincrono.

quando un'applicazione emette una richiesta di invio sulla rete o una richiesta di scrittura su disco, il sistema operativo rileva la richiesta, inserisce l'I/O in un buffer e restituisce il controllo all'applicazione. Appena possibile, per ottimizzare le prestazioni complessive, il sistema operativo completa la richiesta. Se nel frattempo si verifica un errore di sistema, l'applicazione perderà tutte le richieste in atto. I sistemi operativi impongono pertanto, di solito, un limite sul tempo massimo per rispondere a una richiesta. Per esempio, alcune versioni di UNIX ripuliscono i propri buffer del disco ogni 30 secondi; in altre parole ogni richiesta deve essere evasa entro 30 secondi. La coerenza dei dati delle applicazioni viene mantenuta dal kernel, che legge i dati dai suoi buffer prima di inviare richieste di I/O ai dispositivi, assicurando che i dati non ancora scritti siano comunque restituiti al richiedente. Si noti che più thread che eseguono I/O sullo stesso file potrebbero non ricevere dati coerenti, a seconda di come il kernel implementa il suo I/O. In questa situazione, i thread potrebbero dover utilizzare protocolli di locking. Alcune richieste di I/O devono essere eseguite immediatamente, per cui le chiamate di sistema di I/O forniscono di solito un modo per indicare che una determinata richiesta, o l'I/O verso un particolare dispositivo, dovrà essere eseguita in maniera sincrona.

Un buon esempio di comportamento non bloccante è la chiamata di sistema `select()` per le socket di rete. Essa include un argomento che specifica il tempo d'attesa massimo. Se questo valore è 0, l'applicazione può rilevare attività di rete senza arrestarsi. Tuttavia, l'uso della `select()` introduce un overhead aggiuntivo, perché essa può stabilire soltanto se sia possibile compiere dell'I/O: per un effettivo trasferimento di dati, la `select()` deve essere seguita da istruzioni come `read()` o `write()`. Una variante di questo metodo, adottata per esempio dal sistema Mach, è l'impiego di una chiamata di sistema bloccante per la lettura multipla. Essa specifica con una singola chiamata di sistema le richieste di lettura desiderate per diversi dispositivi, e termina non appena una di loro sia stata soddisfatta.

13.3.5 I/O vettorizzato

Alcuni sistemi operativi forniscono un'altra importante variante di I/O tramite le loro interfacce delle applicazioni. L'I/O vettorizzato permette a una chiamata di sistema di eseguire più operazioni di I/O che coinvolgono più locazioni. Per esempio, la chiamata di sistema UNIX `readv` accetta un vettore di buffer e permette di inserire nel vettore i dati letti da una sorgente oppure di scrivere da quel vettore verso una destinazione. Lo stesso trasferimento potrebbe essere realizzato per mezzo di diverse singole invocazioni di chiamate di sistema, ma questo metodo, detto **scatter-gather**, risulta utile per una serie di motivi.

Il trasferimento del contenuto di più buffer distinti tramite un'unica chiamata di sistema permette di evitare cambi di contesto e l'overhead delle chiamate di sistema. In assenza di I/O vettorizzato i dati dovrebbero prima essere trasferiti in un unico buffer più grande, nel giusto ordine, e poi trasmessi. Quest'ultimo metodo risulta piuttosto inefficiente. Inoltre, alcune versioni di scatter-gather forniscono l'atomicità, assicurando così che tutti gli I/O vengano eseguiti senza interruzioni (evitando dunque

la corruzione di dati nel caso in cui altri thread stiano effettuando I/O verso gli stessi buffer). Quando possibile, i programmati sfruttano le potenzialità dell'I/O scatter-gather per aumentare la produttività e diminuire l'overhead del sistema.

13.4 Sottosistema di I/O del kernel

Il kernel fornisce molti servizi riguardanti l'I/O; vari servizi – scheduling, gestione del buffer, delle cache, delle code di spooling, riservazione dei dispositivi e gestione degli errori – sono offerti dal sottosistema di I/O del kernel, e sono realizzati a partire dai dispositivi e dai relativi driver. Il sottosistema di I/O è responsabile anche della propria salvaguardia contro processi malfunzionanti e utenti malintenzionati.

13.4.1 Scheduling dell'I/O

Fare lo scheduling di un insieme di richieste di I/O significa stabilirne un ordine d'esecuzione efficace; l'ordine in cui si verificano le chiamate di sistema da parte delle applicazioni è raramente la scelta migliore. Lo scheduling può migliorare le prestazioni complessive del sistema, distribuire equamente gli accessi dei processi ai dispositivi e ridurre il tempo d'attesa medio per il completamento di un'operazione di I/O. Ecco un semplice esempio che illustra queste potenzialità. Si supponga che la testina di lettura di un'unità a disco sia vicina alla parte iniziale del disco, e che tre applicazioni impartiscano comandi di lettura bloccanti per quest'unità. L'applicazione 1 richiede la lettura di un blocco che si trova vicino alla parte finale del disco, l'applicazione 2 quella di un blocco vicino alla parte iniziale e l'applicazione 3 quella di un blocco situato nella zona centrale. Il sistema operativo può ridurre la distanza percorsa dalla testina del disco servendo le richieste nell'ordine 2, 3, 1. Simili riordinamenti delle sequenze di servizio delle richieste sono l'essenza dello scheduling dell'I/O.

I progettisti di sistemi operativi realizzano lo scheduling mantenendo una coda di richieste per ogni dispositivo. Quando un'applicazione richiede l'esecuzione di una chiamata di sistema di I/O bloccante, si aggiunge la richiesta alla coda relativa al dispositivo. Lo scheduler dell'I/O riorganizza l'ordine della coda per migliorare l'efficienza globale del sistema e il tempo medio d'attesa cui sono sottoposte le applicazioni. Il sistema operativo può anche tentare di essere equo, in modo che nessuna applicazione riceva un servizio carente, o può dare priorità alle richieste sensibili al ritardo. Per esempio, le richieste del sottosistema per la memoria virtuale potrebbero avere priorità su quelle delle applicazioni. Parecchi algoritmi di scheduling per l'I/O delle unità a disco sono descritti dettagliatamente nel Paragrafo 10.4.

I kernel che mettono a disposizione l'I/O asincrono devono essere in grado di tener traccia di più richieste di I/O contemporaneamente. A questo fine, alcuni sistemi associano una **tavola dello stato dei dispositivi** alla coda dei processi in attesa. Gli elementi della tabella – uno per ogni dispositivo di I/O – indicano il tipo, l'indirizzo e lo stato del dispositivo: non funzionante, inattivo o occupato. Se il dispositivo è im-

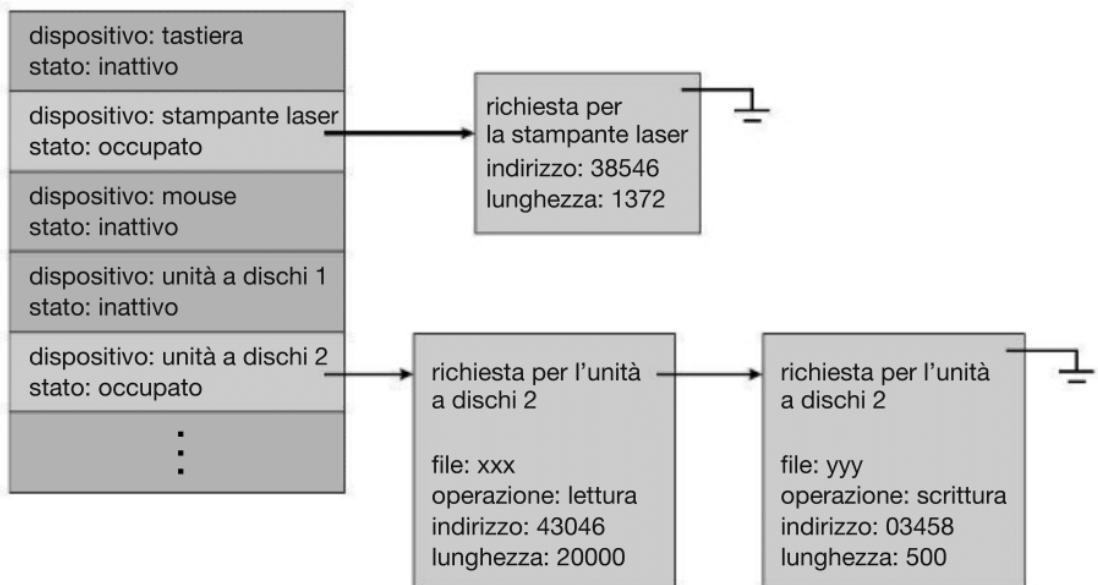


Figura 13.9 Tabella dello stato dei dispositivi.

pegnato nel servire una richiesta, il corrispondente elemento della tabella riporterà il tipo della richiesta e altri parametri a essa relativi. Si veda la Figura 13.9.

Lo scheduling dell’I/O è uno dei modi in cui il sottosistema di I/O migliora l’efficienza di un calcolatore; un altro è l’uso di spazio di memorizzazione nella memoria centrale o nei dischi, per tecniche di buffering, caching e spooling.

13.4.2 Gestione dei buffer

Un **buffer** è un’area di memoria che contiene dati durante il trasferimento fra due dispositivi o tra un’applicazione e un dispositivo. Si ricorre ai buffer per tre ragioni. La prima è la necessità di gestire la differenza di velocità fra il produttore e il consumatore di un flusso di dati. Si supponga, per esempio, di ricevere un file attraverso un modem e di volerlo memorizzare in un’unità a disco: il modem è circa mille volte più lento del disco, perciò conviene creare un buffer nella memoria principale per accumulare i byte che giungono dal modem. Quando tale buffer è pieno, si trasferisce il suo contenuto nel disco con un’unica operazione. Poiché quest’operazione di scrittura non è istantanea e il modem ha bisogno di ulteriore spazio per memorizzare i dati in arrivo, è necessario impiegare due buffer di questo tipo: quando il primo è pieno, si richiede la scrittura nel disco del suo contenuto e il modem comincia a scrivere nel secondo buffer mentre il primo viene scritto su disco. La scrittura nel disco dovrebbe terminare prima che il modem possa riempirlo, cosicché il modem potrà ricominciare a usare il primo buffer, mentre si trasferisce nel disco il contenuto del secondo. Questa **doppia bufferizzazione** svincola il produttore dal consumatore, rendendo così meno critico il problema della loro sincronizzazione. La necessità di questo disaccoppiamento è illustrata dalla Figura 13.10, che mostra le enormi differenze di velocità tra i dispositivi tipici di un calcolatore.

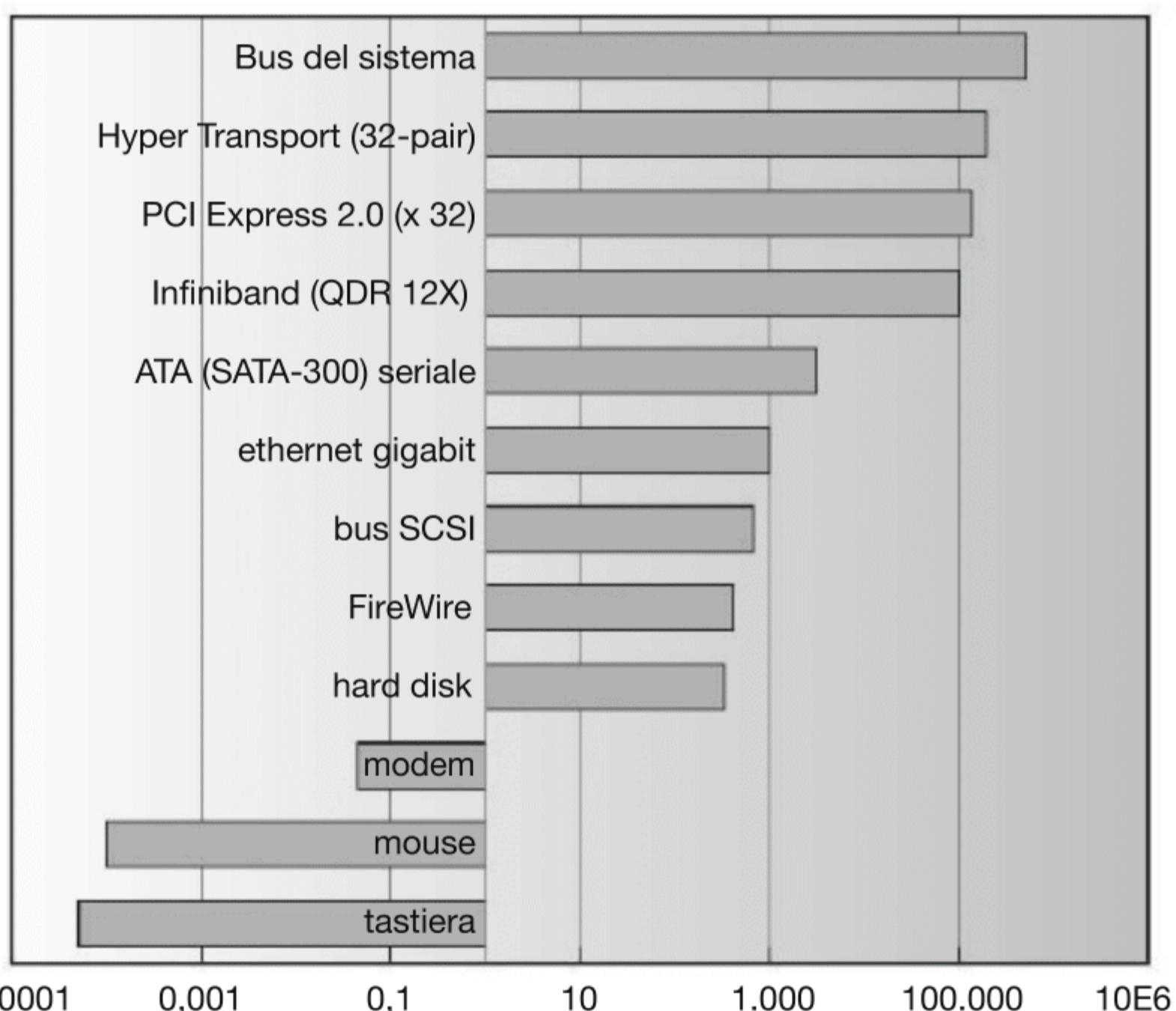


Figura 13.10 Velocità di trasferimento dei dispositivi di un Sun Enterprise 6000 (scala logaritmica).

Un secondo uso della bufferizzazione riguarda la gestione dei dispositivi che trasferiscono dati in blocchi di dimensioni diverse. Queste disparità sono particolarmente comuni nelle reti di calcolatori, dove spesso i buffer sono usati per frammentare e ricomporre messaggi. Quando un mittente spedisce un messaggio molto lungo, esso è spezzato in piccoli pacchetti che si spediscono attraverso la rete; il sistema destinatario provvede a ricostituire in un apposito buffer l'intero messaggio originario.

Il terzo modo in cui si può impiegare un buffer è per la realizzazione della *semantica della copia* nell'ambito dell'I/O delle applicazioni. Un esempio chiarirà il significato di semantica della copia. Si supponga che un'applicazione disponga di un buffer contenente dati da trasferire in un disco: esso richiederà l'esecuzione della chiamata di sistema `write()`, fornendo un puntatore al buffer e un numero intero che specifica il numero di byte da trasferire. Ci si può chiedere che cosa succede se, dopo che la chiamata di sistema restituisce il controllo all'applicazione, quest'ultima modifica il contenuto del buffer. Ebbene, la **semantica della copia** garantisce che la versione dei dati scritta nel disco sia conforme a quella contenuta nel buffer al momento della chiamata di sistema, indipendentemente da ogni successiva modifica. Una semplice maniera di realizzare questa semantica consiste nel far sì che la chiamata di sistema `write()` copi i dati forniti dall'applicazione in un buffer del kernel prima di restituire il controllo all'applicazione stessa. La scrittura nel disco si compie dalla memoria del

kernel, cosicché ogni successivo cambiamento nel buffer dell'applicazione non avrà effetti. In molti sistemi operativi si usa il metodo appena descritto: nonostante implichi una diminuzione dell'efficienza di certe operazioni di I/O, la sua semantica è chiara. Lo stesso effetto, tuttavia, si può ottenere più efficientemente tramite un uso intelligente della memoria virtuale e della protezione data dal copy-on-write delle pagine.

13.4.3 Cache

Una **cache** è una regione di memoria veloce che serve per mantenere copie di dati: l'accesso a queste copie è più rapido dell'accesso agli originali. Per esempio, le istruzioni di un processo correntemente in esecuzione sono memorizzate in un disco, copiate nella memoria fisica (che assume il ruolo di cache rispetto al disco) e copiate ulteriormente nelle cache primaria e secondaria della CPU. La differenza fra un buffer e una cache consiste nel fatto che il primo può contenere dati di cui non esiste altra copia, mentre una cache, per definizione, mantiene su un mezzo più efficiente una copia di informazioni memorizzate altrove.

L'uso delle cache e l'uso dei buffer sono due funzioni distinte, anche se a volte una stessa regione di memoria si può usare per entrambi gli scopi. Per esempio, per realizzare la semantica della copia e permettere uno scheduling efficiente dell'I/O su disco, il sistema operativo impiega dei buffer in memoria centrale per i dati dei dischi. Questi buffer sono anche usati come cache per migliorare l'efficienza delle operazioni di I/O che coinvolgono file condivisi da più applicazioni o file per i quali gli accessi per lettura e scrittura si susseguono rapidamente. Quando riceve una richiesta di I/O relativa a un file, il kernel controlla se la parte interessata del file è già presente nella cache: in questo caso è possibile evitare o differire l'accesso fisico al disco. Inoltre, i dati da scrivere nel disco sono depositati nella cache per diversi secondi, cosicché si accumulano grandi quantità di dati da trasferire: ciò permette uno scheduling efficiente. La strategia consistente nel differire le scritture per migliorare l'efficienza dell'I/O è illustrata nel Paragrafo 17.9.2, nel contesto dell'accesso ai file remoti.

13.4.4 Code di spooling e riservazione dei dispositivi

Una **coda di spooling** è un buffer contenente dati da inviare ad un dispositivo che non può accettare flussi di dati intercalati, per esempio una stampante. Sebbene una stampante possa servire una sola richiesta alla volta, diverse applicazioni devono poter richiedere simultaneamente la stampa di dati, senza che le stampe si mischino. Il sistema operativo risolve questo problema filtrando tutti i dati per la stampante: i dati da stampare provenienti da ogni singola applicazione si registrano in uno specifico spool file su disco; quando un'applicazione termina di stampare, il sistema di spooling aggiunge tale file alla coda di stampa; quest'ultima viene copiata sulla stampante, un file per volta. In certi sistemi operativi questa funzione viene gestita da un processo di sistema specializzato (demone), in altri da un thread del kernel. In entrambi i casi il sistema operativo fornisce un'interfaccia di controllo che permette agli utenti e agli amministratori del sistema di esaminare la coda, eliminare elementi della coda prima che siano stampati, sospendere il servizio di stampa per attività di manutenzione, e così via.

Alcuni dispositivi, come le unità a nastro e le stampanti, non possono alternare più richieste concorrenti di I/O da parte di diverse applicazioni. Lo spooling è uno dei modi in cui il sistema operativo può coordinare output concorrenti; un altro è quello di fornire esplicite funzioni di coordinamento. Alcuni sistemi operativi, fra i quali il VMS, permettono di accedere a un dispositivo in modo esclusivo: un processo può accedere a un dispositivo che non utilizzato, riservandosene l'uso, e restituirlo al sistema quando non ne ha più bisogno. Altri sistemi operativi impediscono l'apertura di più di un handle di file per un dato dispositivo. Molti sistemi operativi forniscono funzioni che permettono ai processi stessi di coordinare l'uso esclusivo dei dispositivi: il sistema Windows, per esempio, mette a disposizione chiamate di sistema che permettono a un'applicazione di aspettare finché un certo dispositivo si liberi. Inoltre la sua chiamata di sistema `OpenFile()` accetta un parametro che specifica il tipo d'accesso concesso ad altri thread concorrenti. In questi sistemi le applicazioni hanno la responsabilità di evitare le situazioni di stallo.

13.4.5 Gestione degli errori

Un sistema operativo che usa la protezione della memoria può difendersi da molti tipi di errori dovuti all'hardware o alle applicazioni, cosicché il blocco completo del sistema non è la necessaria conseguenza di ogni piccolo malfunzionamento. I dispositivi e i trasferimenti di I/O possono essere soggetti ad errori in molti modi, sia per motivi contingenti, come il sovraccarico di una rete di comunicazione, sia per ragioni "permanenti", come nel caso in cui il controllore dell'unità a disco si guasti. I sistemi operativi sono spesso capaci di compensare efficacemente le conseguenze negative dovute a errori temporanei: se per esempio una chiamata di sistema `read()` non ha successo, il sistema ritenterà la lettura; se una chiamata `send()` provoca un errore, il protocollo di rete può richiedere una `resend()`. Purtroppo, però, è difficile che il sistema operativo riesca a compensare gli effetti di errori dovuti a guasti permanenti di qualche componente importante.

Di norma, una chiamata di sistema di I/O riporta un bit d'informazione sullo stato d'esecuzione della chiamata, che indica la riuscita o l'insuccesso dell'operazione richiesta. Il sistema operativo UNIX usa inoltre una variabile intera detta `errno` per codificare piuttosto genericamente il tipo d'errore avvenuto; i valori possibili sono un centinaio e denotano errori dovuti per esempio a puntatori non validi, file non aperti o argomenti oltre i limiti ammessi. Per contro, alcuni tipi di dispositivi possono fornire informazioni assai dettagliate sugli errori, sebbene molti sistemi operativi attuali non siano progettati per passare questi dati alle applicazioni. Per esempio, il malfunzionamento di un dispositivo SCSI è riportato dal protocollo SCSI a tre livelli di dettaglio: usando un **codice di rilevazione** (*sense key*) che identifica la natura generale dell'errore (per esempio: errore hardware, o richiesta illegale); un **codice di rilevazione addizionale** che descrive la categoria cui appartiene il malfunzionamento (parametro del comando errato, o insuccesso del self-test del dispositivo); e infine un **qualificatore del codice di rilevazione addizionale** che fornisce informazioni ancora più dettagliate (quale parametro è errato, o quale componente del dispositivo non ha

superato il test). Inoltre, molti dispositivi SCSI mantengono internamente pagine di log degli errori avvenuti; queste pagine possono essere richieste dalla macchina, ma ciò accade raramente.

13.4.6 Protezione dell'I/O

Gli errori sono strettamente connessi alla tematica della protezione. Un processo utente che, intenzionalmente o accidentalmente, cerchi di impartire istruzioni di I/O illegali può danneggiare il funzionamento normale di un sistema. Per impedire che tali danneggiamenti abbiano luogo, si possono opporre diverse contromisure.

Onde evitare che gli utenti effettuino operazioni di I/O illegali, si definiscono come privilegiate tutte le istruzioni relative all'I/O. Ne consegue che gli utenti non potranno impartire in modo diretto alcuna istruzione, ma dovranno farlo attraverso il sistema operativo. Un programma utente, per eseguire l'I/O, invoca una chiamata di sistema per chiedere al sistema operativo di svolgere una data operazione nel suo interesse (Figura 13.11). Il sistema, passando alla modalità privilegiata, verifica che la richiesta sia valida e, in tal caso, esegue l'operazione; esso trasferisce quindi il controllo all'utente.

Inoltre, il sistema di protezione della memoria deve tutelare dall'accesso degli utenti tutti gli indirizzi mappati in memoria e gli indirizzi delle porte di I/O. Il kernel, tuttavia, non può semplicemente negare qualunque tentativo di accesso da parte degli utenti: quasi tutti i videogiochi, nonché i programmi per il montaggio e la riproduzione di video, per esempio, per ottimizzare le prestazioni della grafica necessitano

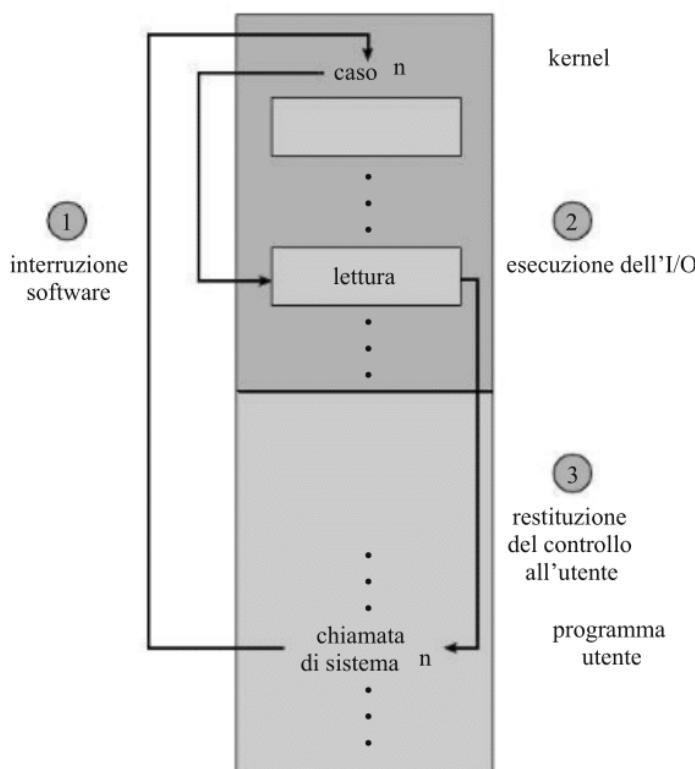


Figura 13.11 Uso delle chiamate di sistema per eseguire I/O.

dell'accesso diretto alla memoria del controllore della grafica, che è gestita in modalità memory mapped. In questi casi, il kernel potrebbe applicare dei lock per assegnare a un solo processo per volta la porzione della memoria grafica che rappresenta una finestra sullo schermo.

13.4.7 Strutture dati del kernel

Il kernel ha bisogno di mantenere informazioni sullo stato dei componenti di I/O, e usa a questo fine diverse strutture dati interne, un esempio delle quali è la tabella dei file aperti descritta nel Paragrafo 12.1. Il kernel usa molte strutture di questo tipo per tener traccia dei collegamenti di rete, delle comunicazioni con i dispositivi a caratteri e di altre attività di I/O.

Il sistema operativo UNIX permette l'accesso, con le modalità tipiche del file system, a diversi oggetti: file degli utenti, dispositivi, spazio d'indirizzi dei processi, e altri ancora. Sebbene ognuno di questi oggetti supporti una chiamata `read()`, le semantiche sono diverse secondo i casi. Quando il kernel, per esempio, deve leggere un file utente, ha bisogno di controllare la *buffer cache* prima di decidere l'effettiva esecuzione di un'operazione di I/O su un disco. Per leggere un disco privo di struttura logica (*raw disk*), il kernel deve accertarsi del fatto che la dimensione dell'insieme dei dati di cui è stato richiesto il trasferimento sia un multiplo della dimensione dei settori del disco e sia allineato con il settore interessato. Per leggere l'immagine di un processo, tutto ciò che occorre è copiare dati dalla memoria. UNIX incapsula queste differenze in una struttura uniforme usando una tecnica orientata agli oggetti. Il record di un file aperto, mostrato nella Figura 13.12, contiene una tabella di puntatori alle procedure appropriate secondo il tipo di file in questione.

Alcuni sistemi operativi applicano metodi orientati agli oggetti in misura più rilevante: il sistema Windows, per esempio, usa per l'I/O un sistema basato sullo scambio di messaggi. Una richiesta di I/O si converte in un messaggio che s'invia tramite il kernel al sottosistema per la gestione dell'I/O, quindi al driver del dispositivo; i contenuti del messaggio possono essere modificati a ogni passaggio intermedio. Quando l'operazione richiesta è di output, il messaggio contiene i dati da scrivere; quando invece l'operazione richiesta è di input, il messaggio contiene un buffer che si usa per ricevere i dati. Questo metodo può comportare una minore efficienza rispetto alle tecniche procedurali basate sulla condivisione delle strutture dati, ma semplifica la progettazione e la struttura del sistema di I/O e permette una maggiore flessibilità.

13.4.8 Concetti principali del sottosistema di I/O del kernel

Riassumendo, il sistema per l'I/O coordina un'ampia raccolta di servizi disponibili per le applicazioni e per altre parti del kernel; in generale sovrintende alle seguenti funzioni:

- gestione dello spazio dei nomi per file e dispositivi;
- controllo dell'accesso ai file e ai dispositivi;
- controllo delle operazioni (per esempio, un modem non può effettuare un `seek()`);

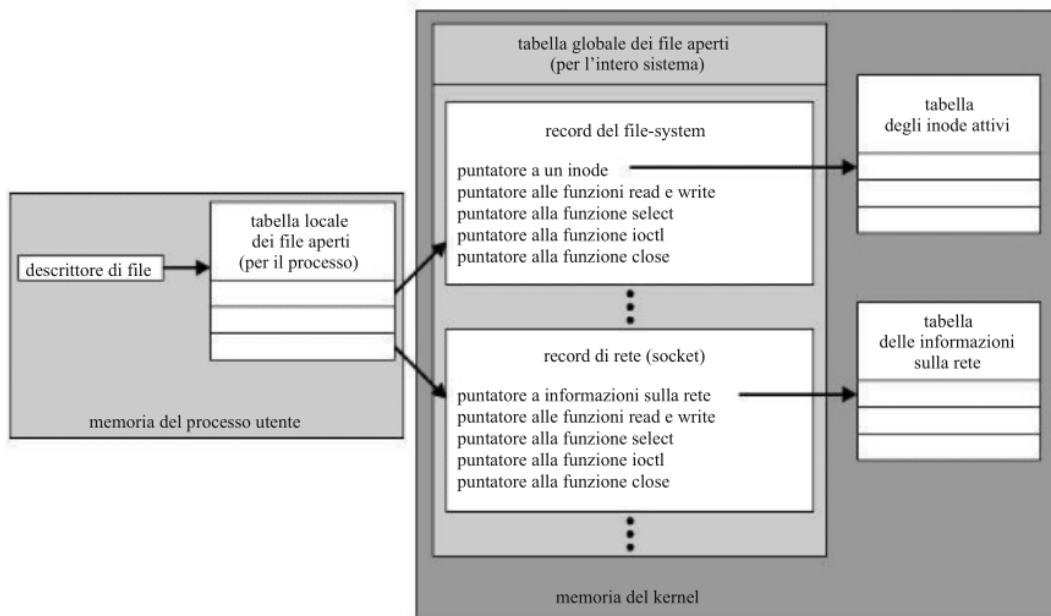


Figura 13.12 Struttura dell'I/O nel kernel di UNIX.

- allocazione dello spazio per il file system;
- allocazione dei dispositivi;
- gestione dei buffer, delle cache e delle code di spooling;
- scheduling dell'I/O;
- controllo dello stato dei dispositivi, gestione degli errori e procedure di ripristino;
- configurazione e inizializzazione dei driver dei dispositivi.

I livelli superiori del sottosistema per la gestione dell'I/O accedono ai dispositivi per mezzo dell'interfaccia uniforme fornita dai driver.

13.5 Trasformazione delle richieste di I/O in operazioni hardware

Il meccanismo di handshaking tra un driver e un controllore di dispositivo è già stato illustrato; tuttavia, non si è ancora spiegato come il sistema operativo associa alla richiesta di un'applicazione un insieme di fili di rete o uno specifico settore di disco. Si consideri per esempio la lettura di un file da un'unità a disco. L'applicazione fa riferimento ai dati per mezzo del nome del file: è compito del file system fornire il modo di giungere, attraverso la struttura delle directory, alla regione del disco appropriata, cioè quella dove i dati del file sono fisicamente residenti. Nell'MS-DOS, per esempio, il nome del file è associato a un numero che individua un elemento della tabella d'accesso ai file; tale elemento identifica i blocchi del disco assegnati al file. In UNIX il nome è associato a un numero di *inode*; l'*inode* corrispondente contiene le

informazioni necessarie per individuare lo spazio allocato. Ma come viene effettuato il collegamento fra il nome del file e il controller del disco (indirizzo hardware della porta o registri memory mapped del controller)?

Un metodo è quello utilizzato da un sistema relativamente semplice come l'MS-DOS. La prima parte di un nome di file dell'MS-DOS, precisamente la parte che precede i due punti, identifica uno specifico dispositivo. Per esempio, **c:** è la parte iniziale di ogni nome di file residente nell'unità a disco principale. Questa convenzione è codificata all'interno del sistema operativo: **c:** è associato a uno specifico indirizzo di porta per mezzo di una tabella dei dispositivi. Grazie all'uso dei due punti come separatore, lo spazio dei nomi dei dispositivi è distinto dallo spazio dei nomi del file system, ciò semplifica al sistema operativo l'associazione di funzioni aggiuntive ai dispositivi. Per esempio, è facile attivare lo spooling per i file di cui è stata richiesta la stampa.

Se, invece, i nomi dei dispositivi sono inclusi nell'ordinario spazio dei nomi del file system, come in UNIX, sono automaticamente disponibili i servizi legati ai nomi dei file. Per esempio, se il file system associa dei possessori ai nomi dei file e fornisce il controllo degli accessi a ogni nome di file, si potrà controllare anche l'accesso ai dispositivi, ed essi avranno un possessore. Visto che i file risiedono nei dispositivi, una tale interfaccia fornisce due livelli d'accesso al sistema d'I/O: i nomi si possono usare per accedere ai dispositivi stessi o ai file in essi contenuti.

UNIX rappresenta i nomi dei dispositivi all'interno dell'ordinario spazio dei nomi del file system. A differenza di un nome di file dell'MS-DOS, che include i due punti come separatore, in un nome di percorso di UNIX il nome del dispositivo non è esplicitamente separato. In effetti, nessuna parte del nome di percorso di un file è il nome di un dispositivo; UNIX impiega una **tabella di montaggio** (*mount table*), per associare i prefissi dei nomi di percorso ai corrispondenti nomi di dispositivi. Quando deve risolvere un nome di percorso, il sistema esamina la tabella per trovare il più lungo prefisso corrispondente: questo elemento della tabella indica il nome del dispositivo voluto. Anche questo nome è rappresentato come un oggetto del file system: tuttavia, quando UNIX cerca questo nome nelle strutture delle directory del file system, non trova il numero di un *inode*, ma una coppia di numeri <principale , secondario> (<**major** , **minor**>) che identifica un dispositivo. Il numero principale individua il driver che si deve usare per gestire l'I/O nel dispositivo in questione, mentre il numero secondario deve essere passato a questo driver affinché esso possa determinare, per mezzo di un'altra tabella, l'indirizzo della porta o l'indirizzo memory mapped del controllore del dispositivo interessato. I moderni sistemi operativi riescono a ottenere un notevole grado di flessibilità grazie all'uso di tabelle di lookup a vari livelli durante il processo che porta da una richiesta al controllore del dispositivo; questo processo, inoltre, è del tutto generale, cosicché non è necessario ricompilare il kernel ogni volta che si aggiungono al calcolatore nuovi dispositivi e nuovi driver. In effetti, alcuni sistemi operativi hanno la capacità di caricare driver di dispositivi su richiesta: all'avviamento, il sistema sonda i bus per determinare quali dispositivi siano presenti; quindi carica i necessari driver, operazione che può anche essere rinviata fino alla prima richiesta di I/O.

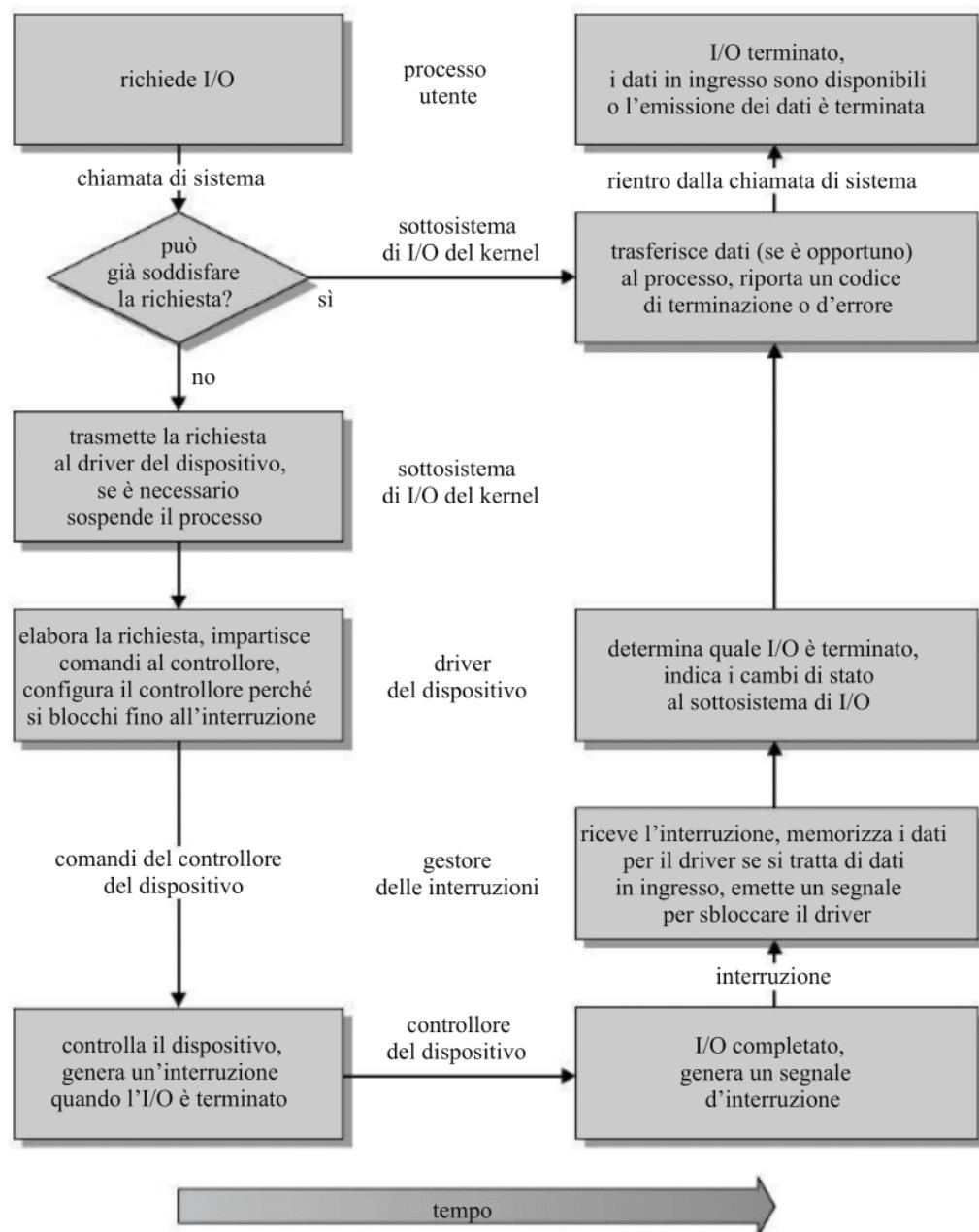


Figura 13.13 Schema d'esecuzione di una richiesta di I/O.

La seguente descrizione del tipico svolgimento di una richiesta di lettura bloccante (Figura 13.13) indica che l'esecuzione di un'operazione di I/O richiede una gran quantità di passi; ciò implica l'uso di un enorme numero di cicli di CPU.

1. Un processo esegue una chiamata di sistema `read()` bloccante relativa a un descrittore di file di un file precedentemente aperto.
2. Il codice della chiamata di sistema all'interno del kernel controlla la correttezza dei parametri. Nel caso di input, se i dati sono già presenti nella *buffer cache*, si passano al processo chiamante e l'operazione è conclusa.

3. Altrimenti, è necessario eseguire un'operazione di I/O fisico. Si rimuove il processo dalla coda dei processi pronti per l'esecuzione per inserirlo nella coda d'attesa relativa al dispositivo interessato e si effettua lo scheduling della richiesta di I/O. Infine il sottosistema di I/O invia la richiesta al driver del dispositivo; a seconda del sistema operativo, ciò avviene tramite la chiamata di una procedura, o per mezzo dell'invio di un messaggio interno al kernel.
4. Il driver del dispositivo assegna un buffer nello spazio d'indirizzi del kernel che serve per ricevere i dati immessi, ed esegue lo scheduling dell'I/O. Infine il driver impedisce comandi al controllore del dispositivo scrivendo nei suoi registri.
5. Il controllore aziona il dispositivo hardware per compiere il trasferimento dei dati.
6. Il driver può operare in polling, o può aver predisposto un trasferimento DMA nella memoria del kernel. Supponiamo che il trasferimento sia gestito dal controllore DMA, il quale genera un'interruzione al termine dell'operazione.
7. Tramite il vettore delle interruzioni, si attiva l'appropriato gestore dell'interruzione che, dopo aver memorizzato i dati necessari, avverte con un segnale il driver del dispositivo ed effettua il ritorno dall'interruzione.
8. Il driver riceve il segnale, individua la richiesta di I/O che è stata completata, si accerta della riuscita o del fallimento dell'operazione e segnala al sottosistema di I/O del kernel il completamento dell'operazione.
9. Il kernel trasferisce dati e/o codici di stato nello spazio degli indirizzi del processo chiamante, e sposta tale processo dalla coda d'attesa alla coda dei processi pronti per l'esecuzione.
10. Nel momento in cui è posto nella coda dei processi pronti per l'esecuzione, il processo non è più bloccato: quando lo scheduler gli assegnerà la CPU, esso riprenderà l'elaborazione. L'esecuzione della chiamata di sistema è completata.

13.6 STREAMS

Il sistema operativo UNIX System V offre un interessante meccanismo, chiamato **STREAMS**, che permette a un'applicazione di comporre dinamicamente catene di codice di driver. Uno *stream* è una connessione *full-duplex* fra un driver di dispositivo e un processo utente, e consiste di un elemento di interfaccia per il processo utente (*stream head*), un elemento che controlla il dispositivo (*driver end*), ed eventualmente un certo numero di moduli intermedi fra questi due elementi (*stream modules*). Tutti questi componenti contengono una coppia di code, una di lettura e una di scrittura; per il trasferimento dei dati tra le due code s'impiega uno schema a scambio di messaggi. La Figura 13.14 mostra la struttura del meccanismo di STREAMS.

Le funzioni d'elaborazione di STREAMS sono fornite da moduli che si inseriscono nello stream attraverso la chiamata di sistema `ioctl()`. Un processo può, per esempio, aprire tramite uno stream una porta seriale, e può inserire un modulo per editare

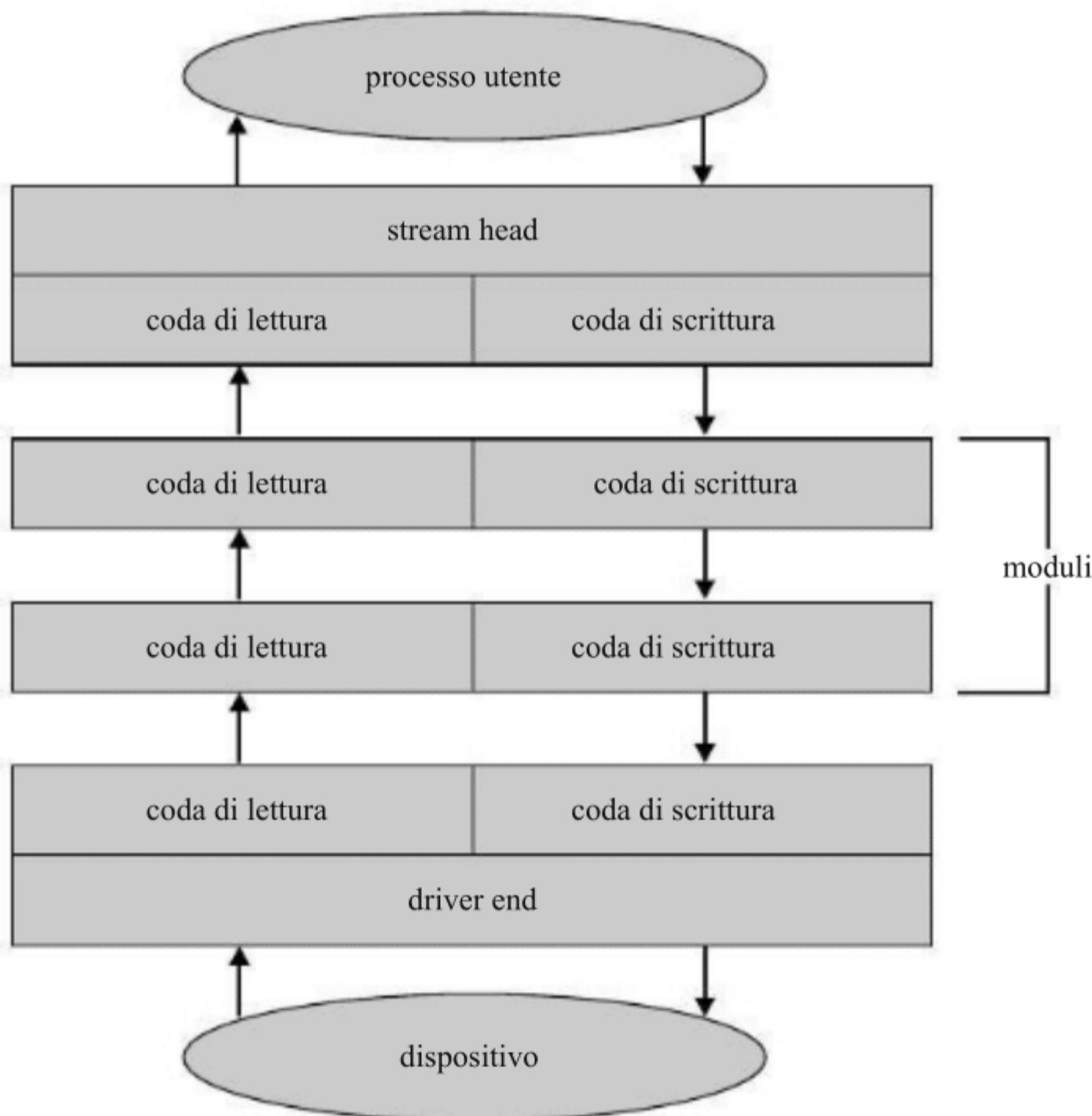


Figura 13.14 Struttura di STREAMS.

i dati immessi. Poiché i messaggi sono scambiati tra code di moduli adiacenti, la coda di un modulo potrebbe mandare in overflow la coda di un modulo adiacente. Per prevenire questo problema, una coda può disporre di un meccanismo di **controllo di flusso**. Senza controllo di flusso, una coda accetta tutti i messaggi e li trasferisce immediatamente alla coda del modulo adiacente, senza buffering. Una coda che invece impiega il controllo di flusso memorizza i messaggi in un buffer; se lo spazio disponibile in esso non è sufficiente, non accetta messaggi. Questo meccanismo richiede scambi di messaggi di controllo tra code in moduli adiacenti.

Un processo utente usa le chiamate di sistema `write()` o `putmsg()` per scrivere dati in un dispositivo; la chiamata di sistema `write()` scrive semplicemente dati non strutturati nello *stream*, mentre `putmsg()` permette al processo utente di specificare un messaggio. Indipendentemente dalla chiamata di sistema adoperata dal processo utente, lo stream head copia i dati in un messaggio e li recapita alla coda del modulo successivo. Questa copiatura dei messaggi continua finché il messaggio non giunge al driver end e quindi al dispositivo. Analogamente, il processo utente legge i dati dallo stream head usando la chiamata di sistema `read()` oppure `getmsg()`. Se si usa la `read()` lo stream head preleva un messaggio dalla coda del modulo adiacente e riporta al processo dati ordinari (una sequenza non strutturata di byte). Se si usa la `getmsg()` viene inviato un messaggio al processo utente.

L'I/O per mezzo di STREAMS è asincrono (o non bloccante), con l'eccezione di quando il processo utente comunica con lo stream head. Mentre scrive nello *stream*, il processo utente si blocca, se la coda successiva impiega il controllo di flusso, finché non ci sia spazio sufficiente per copiarvi il messaggio. Analogamente, il processo utente si blocca durante la lettura dallo *stream* finché non ci sono dati disponibili.

Come si è detto, il driver end, come lo stream head e i moduli intermedi, ha una coda di lettura e una di scrittura. Tuttavia deve poter rispondere a interruzioni come quelle generate quando un pacchetto di dati è pronto per essere letto da una rete. A differenza dello stream head che si può bloccare se non è possibile copiare un messaggio nella coda del modulo successivo, il driver end deve gestire tutti i dati in arrivo. I driver devono anche supportare il controllo di flusso. Tuttavia, se il buffer di un dispositivo è pieno, di solito ignora i messaggi in entrata. Si consideri una scheda di rete il cui buffer d'ingresso sia pieno; la scheda di rete deve semplicemente ignorare gli ulteriori messaggi fintantoché non vi sia sufficiente spazio per memorizzare i messaggi in arrivo.

Il vantaggio nell'utilizzo di STREAMS consiste nel disporre di un ambiente che permette uno sviluppo modulare e incrementale di driver di dispositivi e protocolli di rete. I moduli possono essere usati da diversi stream e quindi da diversi dispositivi. Per esempio, un modulo di rete potrebbe essere adoperato sia da una scheda di rete Ethernet sia da una scheda di rete wireless 802.11. Inoltre, invece di trattare l'I/O di dispositivi a caratteri come una sequenza non strutturata di byte, STREAMS permette la gestione dei limiti dei messaggi e delle informazioni di controllo tra i diversi moduli. L'impiego di STREAMS è assai diffuso nella maggior parte dei sistemi UNIX, ed è il metodo più usato per la scrittura di protocolli e driver di dispositivi. In UNIX System V e in Solaris, per esempio, il meccanismo delle socket è realizzato tramite STREAMS.

13.7 Prestazioni

L'I/O è uno tra i principali fattori che influiscono sulle prestazioni di un sistema: richiede un notevole impegno della CPU per l'esecuzione del codice dei driver e per uno scheduling equo ed efficiente dei processi quando essi sono bloccati e sbloccati. I risultanti cambi di contesto sfruttano fino in fondo la CPU e le sue memorie cache. L'I/O, inoltre, rivela le eventuali inefficienze dei meccanismi del kernel per la gestione delle interruzioni, e impegna il bus della memoria durante i trasferimenti di dati tra i controllori dei dispositivi e la memoria fisica, e ancora tra i buffer del kernel e lo spazio d'indirizzi delle applicazioni. Soddisfare in modo elegante tutte queste esigenze è una delle principali questioni che riguardano i progettisti di un calcolatore.

Sebbene i calcolatori moderni siano capaci di gestire molte migliaia di interruzioni al secondo, la loro gestione è un processo relativamente oneroso. Ogni interruzione fa sì che il sistema cambi stato, esegua il gestore delle interruzioni, e infine ripristini lo stato originario. Se il numero di cicli impiegato nel busy waiting non è eccessivo,

l’I/O programmato può essere più efficiente di quello basato sulle interruzioni. Il completamento di un’operazione di I/O in genere implica lo sblocco di un processo, comportando così l’overhead dovuto a un cambio di contesto.

Anche il traffico di una rete può portare a un alto numero di cambi di contesto; si consideri, per esempio, il login remoto da un calcolatore ad un’altro. Ciascun carattere inserito in un calcolatore deve essere comunicato all’altro: quando si inserisce un carattere nel primo calcolatore, la tastiera produce un segnale d’interruzione; il carattere arriva tramite il gestore delle interruzioni al driver del dispositivo, da questo al kernel, e quindi al processo utente. Il processo utente esegue una chiamata di sistema di I/O di rete al fine di inviare il carattere al secondo calcolatore. All’interno del kernel del primo calcolatore, il carattere attraversa gli strati di protocollo necessari per la costruzione di un pacchetto di rete e giunge al driver di rete, il quale trasferisce il pacchetto al controllore della interfaccia di rete. Quest’ultimo invia il carattere e genera un’interruzione che segnala al kernel il completamento della chiamata di sistema di I/O di rete.

A questo punto, il dispositivo di rete del secondo calcolatore riceve il pacchetto, e genera un’interruzione. I protocolli di rete estraggono il carattere dal pacchetto e lo consegnano all’appropriato demone di rete. Il demone di rete individua a quale sessione di login remoto appartiene il carattere e lo passa al sottodemone che gestisce la specifica sessione. Durante questo processo avvengono cambi di contesto e di stato (Figura 13.15). Di solito il destinatario rispedisce al mittente, sotto forma di eco, una copia del carattere originario, e ciò raddoppia il lavoro necessario.

Per eliminare i cambi di contesto implicati dal trasferimento di ciascun carattere dal demone al kernel, gli sviluppatori di Solaris hanno implementato nuovamente il demone `telnet` tramite thread interni al kernel. Secondo stime della Sun, queste migliorie hanno portato il massimo numero di connessioni contemporanee sostenibili da un grande server da qualche centinaio a qualche migliaio.

Altri sistemi usano unità d’elaborazione specifiche per la gestione dell’I/O dei terminali, riducendo così il carico delle gestione delle interruzioni gravante sulla CPU. Per esempio, i **concentratori di terminali** convogliano il traffico proveniente da centinaia di terminali su un’unica porta di un grande calcolatore. I **canali di I/O** sono unità d’elaborazione specializzate presenti nei mainframe e altri sistemi di alto profilo; il loro compito è sollevare la CPU di una parte del peso della gestione dell’I/O. L’idea di base è che i canali di I/O mantengano il flusso dei dati costante e uniforme, mentre la CPU rimane libera di elaborare i dati acquisiti. Come i controller dei dispositivi e i controllori DMA che si trovano nei calcolatori di minori dimensioni, un canale può eseguire programmi più raffinati e generali, e può quindi essere tarato per specifici carichi di lavoro.

Per migliorare l’efficienza dell’I/O si possono applicare diversi principi.

- Ridurre il numero dei cambi di contesto.
- Ridurre il numero di copiature dei dati in memoria durante i trasferimenti fra dispositivi e applicazioni.

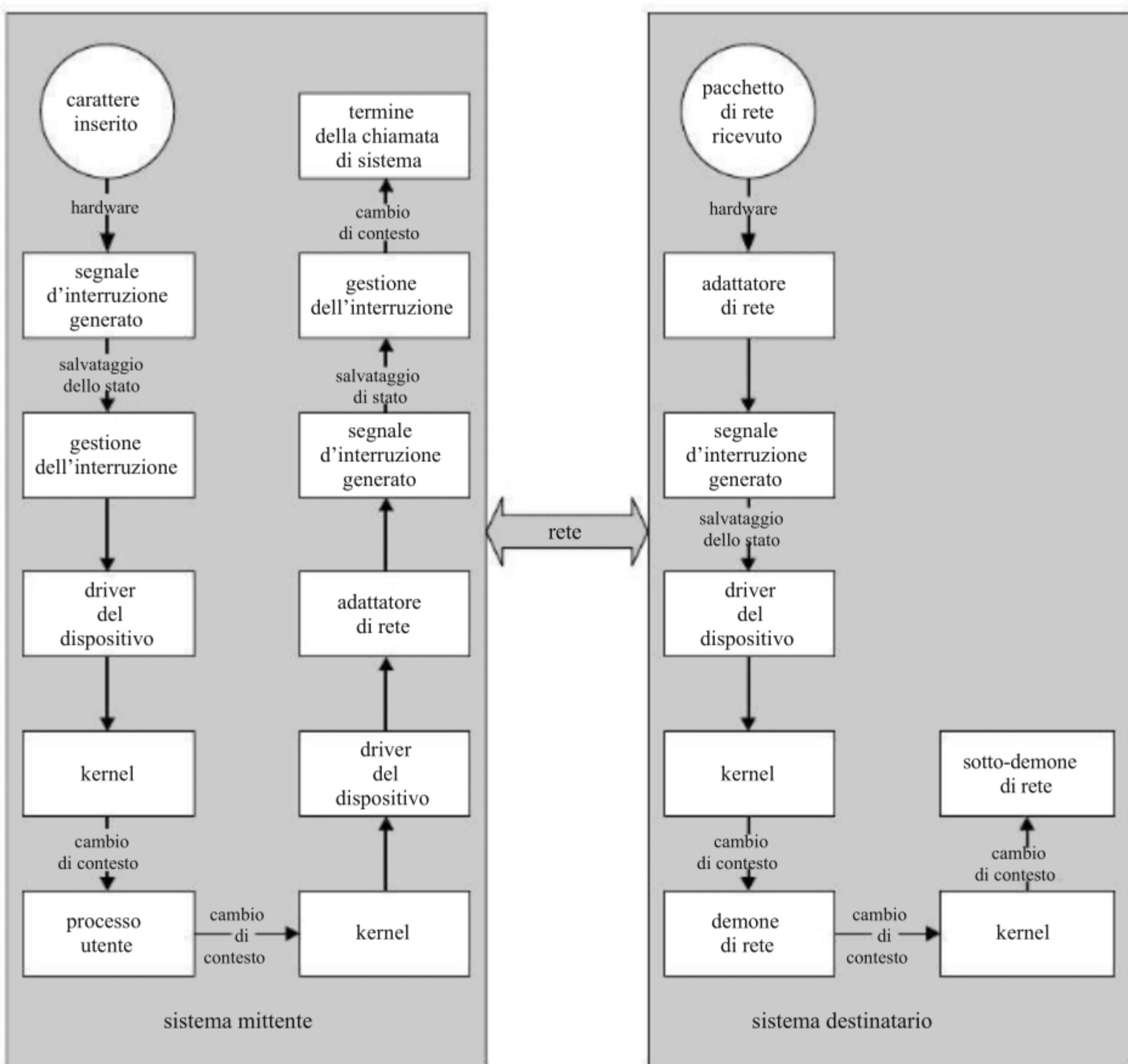


Figura 13.15 Comunicazione tra calcolatori.

- Ridurre la frequenza delle interruzioni utilizzando il trasferimento in blocco di grandi quantità di dati, controllori intelligenti e il polling (nel caso in cui si possa minimizzare il busy waiting).
- Aumentare il tasso di concorrenza usando controllori DMA intelligenti o canali di I/O per sollevare la CPU dalle semplici operazioni di copiatura di dati.
- Realizzare le primitive di elaborazione direttamente nell'hardware, così da permettere che la loro esecuzione sia simultanea alle operazioni di bus e di CPU.
- Equilibrare le prestazioni della CPU, del sottosistema di memoria, del bus e dell'I/O, giacché il sovraccarico di uno qualunque di questi settori provoca l'inutilizzo degli altri.

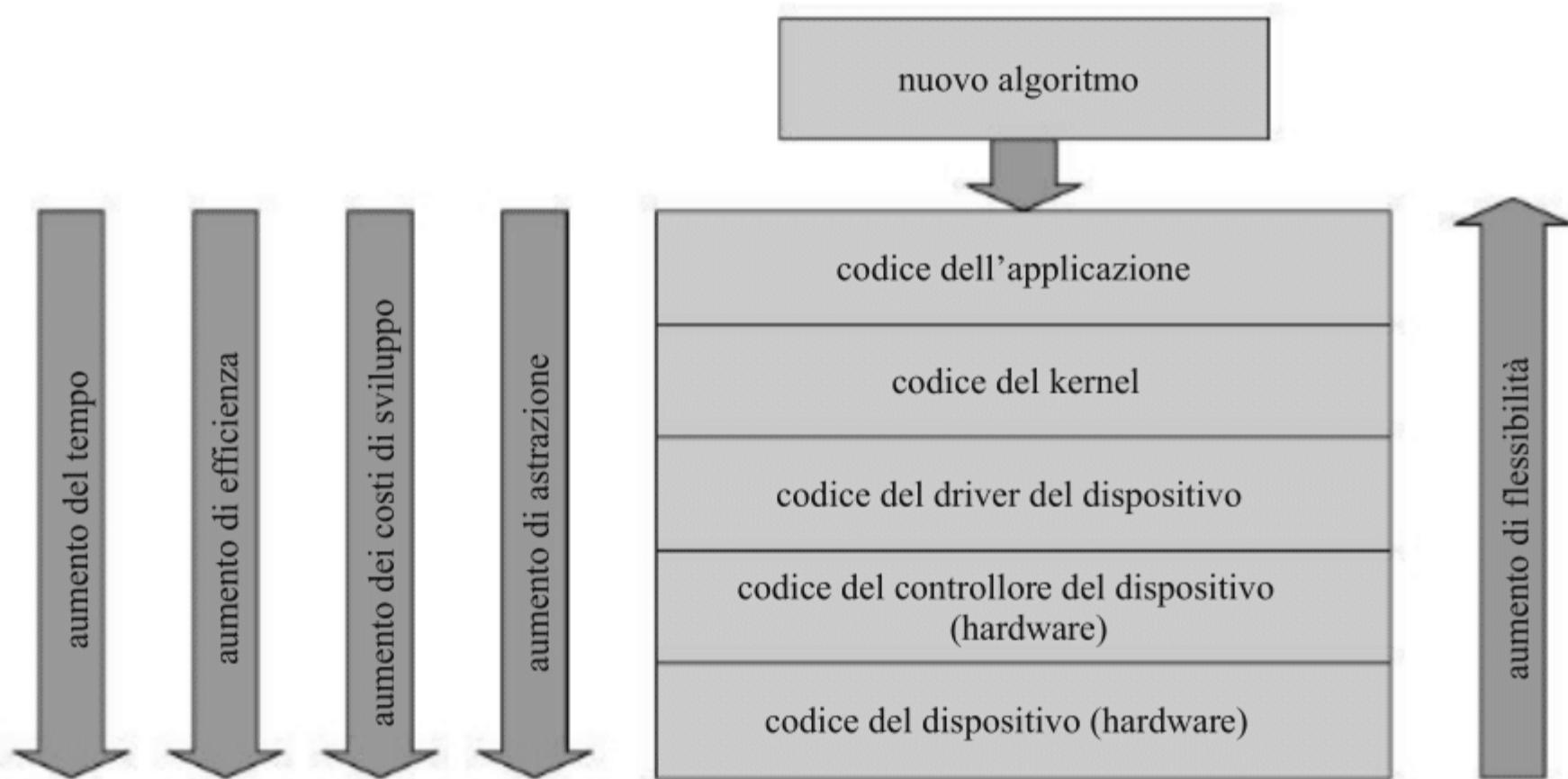


Figura 13.16 Successione delle funzionalità dei servizi di I/O.

La complessità dei dispositivi è assai variabile: un mouse per esempio è piuttosto semplice; i suoi movimenti e la pressione sui pulsanti sono convertiti in valori numerici, passati attraverso il driver del mouse, all'applicazione. Per contro, i servizi forniti dal driver delle unità a disco del sistema operativo Windows sono assai complessi: non solo il driver gestisce singole unità a disco, ma costruisce anche array RAID (si veda il Paragrafo 10.7) convertendo le richieste di lettura o scrittura di un'applicazione in un insieme coordinato di operazioni di I/O sui dischi. Il driver, inoltre, esegue sofisticati algoritmi di gestione degli errori e di recupero dei dati, e svolge diverse funzioni di ottimizzazione delle prestazioni dei dischi.

Ci si può chiedere se i servizi di I/O si debbano implementare nei dispositivi hardware, nei loro driver, o nelle applicazioni. Talvolta si può osservare (Figura 13.16) la seguente successione.

- Inizialmente, gli algoritmi sperimentali per l'I/O si codificano a livello dell'applicazione, dato che il codice dell'applicazione è flessibile ed è difficile che errori di programmazione causino l'arresto completo del sistema. In questo modo si evita inoltre di dover riavviare o ricaricare i driver dei dispositivi ogni volta che si modifica il codice degli algoritmi. Tuttavia, questi algoritmi sono spesso inefficienti a causa dell'overhead dovuto ai cambi di contesto necessari e dell'impossibilità di sfruttare le strutture dati del kernel e i suoi meccanismi interni (per esempio, la gestione dei messaggi, l'uso dei thread, i lock).
- Quando uno di questi algoritmi è stato messo a punto, è possibile ricodificarlo all'interno del kernel. Ciò può portare a un miglioramento delle prestazioni, ma la stesura del codice è più impegnativa, perché il kernel è un ambiente vasto e complesso. È inoltre necessario verificare accuratamente la correttezza del codice al fine di evitare alterazioni dei dati e l'arresto del sistema.

- Le prestazioni migliori si ottengono con l'integrazione delle funzioni di tali algoritmi in hardware, nei dispositivi o nei controllori. Gli svantaggi di questa tecnica comprendono la difficoltà e il costo di successive migliorie o dell'eliminazione di eventuali errori, il maggior tempo richiesto per portarne a termine la realizzazione (mesi invece di giorni), e la minore flessibilità. Per esempio, un controllore RAID può essere privo di funzioni che permettono al kernel di modificare la locazione o l'ordine di lettura o scrittura di singoli blocchi, anche se il kernel potrebbe possedere informazioni particolari sul carico di lavoro che gli permetterebbero di migliorare le prestazioni dell'I/O.

13.8 Sommario

I principali elementi hardware di un calcolatore coinvolti nell'esecuzione dell'I/O sono i bus, i controllori dei dispositivi e i dispositivi stessi. I trasferimenti dei dati tra i dispositivi e la memoria centrale sono controllati dalla CPU nel caso di I/O programmato, altrimenti, sono demandati al controllore DMA. Un modulo del kernel che controlla un dispositivo è detto driver. L'interfaccia fornita alle applicazioni, costituita dalle chiamate di sistema, è progettata per gestire diverse categorie fondamentali di dispositivi hardware, come i dispositivi a blocchi e a caratteri, i timer programmabili, i file mappati in memoria, le socket di rete. Di solito le chiamate di sistema bloccano il processo che le ha invocate; il kernel e le applicazioni che non devono attendere il completamento di un'operazione di I/O impiegano chiamate di sistema non bloccanti o asincrone.

Il sottosistema di I/O del kernel fornisce numerosi servizi: fra gli altri, lo scheduling dell'I/O, il buffering, il caching, le code di spooling, la gestione degli errori e la riservazione dei dispositivi. Un altro servizio è la traduzione dei nomi, che permette di associare ai nomi simbolici di file usati dalle applicazioni i dispositivi hardware corrispondenti. Si tratta di un processo che passa attraverso diversi stadi: dalla sequenza di caratteri che rappresenta il nome a un driver specifico e all'indirizzo di un dispositivo, e da qui all'indirizzo fisico delle porte di I/O o dei controllori di bus. Tale interpretazione può avvenire nell'ambito dello spazio dei nomi del file system (come in UNIX), o in uno specifico spazio di nomi dei dispositivi (come nell'MS-DOS).

STREAMS è una realizzazione e una metodologia che permettono di sviluppare in modo modulare ed incrementale i driver e i protocolli di rete. Utilizzando gli stream, i driver possono essere organizzati in una catena, attraverso cui passano i dati in maniera sequenziale e bidirezionale per l'elaborazione.

A causa dei molti strati di software presenti fra un dispositivo fisico e l'applicazione, le chiamate di sistema per l'I/O sono onerose in termini di utilizzazione della CPU. Questa struttura a strati comporta diversi overhead: per realizzare i cambi di contesto, gestire le interruzioni e i segnali usati per la comunicazione con i dispositivi, e copiare dati fra le aree di memoria per l'I/O del kernel e lo spazio d'indirizzi delle applicazioni.

Esercizi di ripasso

- 13.1** Individuate tre vantaggi che si ottengono dal collocare funzionalità all'interno del controllore di un dispositivo piuttosto che nel kernel. Individuate poi tre svantaggi.
- 13.2** L'esempio di handshaking del Paragrafo 13.2 utilizza 2 bit: un bit busy e un bit command-ready. È possibile implementare la stessa negoziazione con un solo bit? Se è possibile, descrivete il protocollo. Se non lo è, spiegate perché un bit non è sufficiente.
- 13.3** Perché un sistema potrebbe utilizzare I/O guidato dalle interruzioni per gestire una porta seriale singola e il polling per gestire un processore di front-end come un terminal concentrator?
- 13.4** Il polling per il completamento di I/O può sprecare un gran numero di cicli di CPU se il processore itera un ciclo busy-waiting molte volte prima che l'I/O sia terminato. D'altro canto, se il dispositivo di I/O è pronto per il servizio, l'interrogazione ciclica può essere molto più efficiente rispetto a rilevare e gestire un'interruzione. Descrivete una strategia ibrida per un dispositivo di I/O che combini polling, sleeping e interruzioni. Per ognuna di queste tre strategie (polling puro, interruzioni pure e ibrida) descrivete un contesto in cui quella strategia sia più efficiente delle altre.
- 13.5** In che modo il DMA aumenta la concorrenza? In che senso complica il progetto dell'hardware?
- 13.6** Perché è importante aumentare la velocità del bus di sistema e delle periferiche all'aumentare della velocità della CPU?
- 13.7** Fate una distinzione tra un driver STREAMS e un modulo STREAMS.

Esercizi

- 13.8** Nel caso di interruzioni multiple, inviate da dispositivi diversi approssimativamente nello stesso istante, si può applicare un criterio di priorità per stabilire l'ordine di precedenza da dare alle interruzioni. Valutate gli elementi che è necessario considerare per assegnare priorità alle interruzioni.
- 13.9** Valutate gli aspetti positivi e negativi causati della mappatura in memoria dei registri di controllo dei dispositivi per l'I/O.
- 13.10** Considerate le seguenti situazioni riguardanti l'I/O in un PC monoutente:
- un mouse usato insieme con un'interfaccia utente grafica;
 - un lettore di nastri in un sistema operativo multitasking (assumete l'impossibilità di preassegnare i dispositivi);
 - un'unità a disco contenente i file dell'utente;

- d. una scheda grafica con connessione diretta tramite bus, accessibile per mezzo di I/O memory mapped.

Per ognuna di queste situazioni, dite se è opportuno progettare il sistema operativo in modo che possa impiegare la gestione del buffer, lo spooling, il caching, o una loro combinazione. Dite inoltre se è opportuno usare il polling o le interruzioni. Argomentate le risposte.

- 13.11** In molti sistemi multiprogrammati, i programmi utenti accedono alla memoria per mezzo degli indirizzi virtuali, mentre il sistema operativo utilizza direttamente gli indirizzi fisici. Come si riflette questo fatto sulle operazioni di I/O, nella fase di avvio da parte del programma utente e, in seguito, sulla loro esecuzione da parte del sistema?
- 13.12** Quali sono le diverse forme di overhead di gestione causate dal servire un'interruzione?
- 13.13** Descrivete tre circostanze in cui sarebbe opportuno usare l'I/O bloccante, e altre tre in cui dovrebbe essere usato l'I/O non bloccante. Riflettete sulla possibilità di realizzare semplicemente l'I/O non bloccante e lasciare che i processi interroghino ciclicamente i dispositivi richiesti finché essi sono pronti.
- 13.14** Di norma, quando un dispositivo completa il proprio I/O, si genera una singola interruzione, gestita dal processore nel modo appropriato. In alcuni frangenti, tuttavia, il codice da eseguire al termine del lavoro di I/O può essere suddiviso in due segmenti: il primo, eseguito subito dopo che l'I/O è completato, pianifica anche una seconda interruzione per innescare l'esecuzione del secondo segmento di codice, in un momento successivo. A quale scopo si adotta questa strategia nel progettare i gestori delle interruzioni?
- 13.15** Alcuni controllori DMA forniscono l'accesso diretto alla memoria virtuale. In questo caso, i destinatari delle operazioni di I/O sono indirizzi virtuali, tradotti poi in indirizzi fisici durante l'accesso diretto alla memoria. Per quali versi tale funzionalità complica la progettazione del controllore DMA? Quali sono i suoi vantaggi?
- 13.16** Il sistema UNIX coordina le attività dei componenti per l'I/O del kernel manipolando le strutture dati condivise interne al kernel; invece il sistema Windows utilizza lo scambio di messaggi tra i componenti del kernel, con tecniche orientate agli oggetti. Valutate tre elementi a favore e tre a sfavore di ciascuna strategia.
- 13.17** Scrivete in uno pseudocodice una procedura che realizzi un orologio virtuale che comprenda l'accodamento e la gestione delle richieste del timer per il kernel e le applicazioni. Assumete che l'hardware fornisca tre canali di temporizzazione.
- 13.18** Considerate vantaggi e svantaggi che derivano dal garantire un trasferimento dei dati affidabile tra i moduli di un'applicazione STREAMS.