**Introduzione**

Oltre a fornire astrazioni come processi e thread, spazi degli indirizzi e file, un sistema operativo controlla anche tutti i dispositivi di I/O del computer, inviando comandi ai dispositivi,

intercettando gli interrupt e gestendo gli errori. Dovrebbe anche fornire un’interfaccia fra i

dispositivi e il resto del sistema che sia semplice e facile da usare.

La parte di software responsabile della gestione dell’I/O rappresenta una frazione significativa dell’intero sistema operativo.

**Princìpi hardware dell’I/O**

**Dispositivi di I/O**

I dispositivi di I/O possono essere suddivisi in due categorie: dispositivi a blocchi e dispositivi a caratteri.

Un dispositivo a blocchi è quello che archivia informazioni in blocchi di dimensioni fisse, ognuno con il proprio indirizzo. Tutti i trasferimenti sono in unità di uno o più blocchi (consecutivi) interi. La caratteristica essenziale di un dispositivo a blocchi è che ciascun blocco può essere letto o scritto indipendentemente da tutti gli altri. Dischi fissi, dischi CD-ROM e penne USB sono classici dispositivi a blocchi.

Un dispositivo a caratteri gestisce un flusso di caratteri, senza alcuna struttura a blocchi. Non è indirizzabile e non ha alcuna operazione di ricerca. Stampanti, interfacce di rete e mouse possono essere visti come dispositivi a caratteri.

Questo schema di classificazione non è perfetto. Alcuni dispositivi rimangono esclusi.

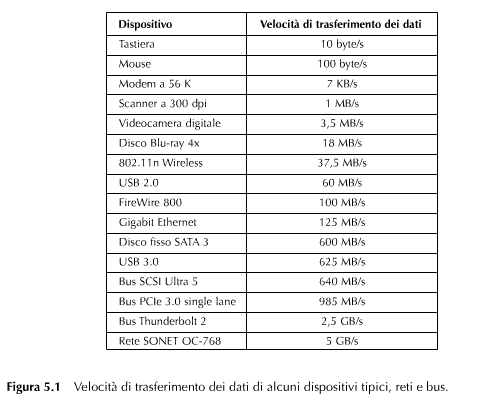
I clock, per esempio, non sono indirizzabili a blocchi e non generano né accettano flussi di

caratteri: tutto ciò che fanno è produrre interrupt a intervalli ben definiti. Analogamente i

video a mappatura di memoria non si adattano a questo modello e nemmeno i touch screen.

I dispositivi di I/O possono avere velocità molto diverse, il che mette una notevole pressione sul software affinché il trasferimento dati avvenga con buone prestazioni. La Figura 5.1 mostra le velocità di alcuni dispositivi classici. La maggior parte di questi tende a divenire

sempre più veloce con il passare degli anni.

****

**Controller dei dispositivi**

I dispositivi di I/O consistono tipicamente di una componente meccanica e di una elettronica.

La componente elettronica è detta controller del dispositivo (device controller) o adattatore (adapter). Sui personal computer è spesso presente nella forma di un chip sulla scheda madre o di una scheda a circuiti stampati inseribile in un alloggiamento di espansione (PCI).

La parte meccanica è il dispositivo stesso. Questa impostazione è mostrata nella

Figura 1.6.

Molti controller possono gestire più dispositivi identici.

Se l’interfaccia fra il controller e il dispositivo è un’interfaccia standard, possono esistere due mercati separati di produttori. Per esempio, quello che effettivamente esce dall’unità disco è uno stream seriale di bit, che parte con un preambolo (preamble), seguito dai 4096 bit in un settore e alla fine da un controllo numerico, chiamato anche codice di correzione degli errori (ECC - error correcting code). Il lavoro del controller è convertire il flusso seriale di bit in un blocco di byte ed eseguire ogni necessaria correzione di errore. Dopo che il codice di correzione è stato verificato e il blocco dichiarato libero da errori, esso può essere copiato nella memoria principale.

Quando la CPU vuole leggere una parola (dalla memoria o da una porta di I/O):

- pone l’indirizzo sull’address bus;

- asserisce il segnale READ sul bus di controllo;

- asserisce un segnale per dire da dove si svolge la lettura (I/O o memoria);

- attende la risposta della memoria o del dispositivo.

**Input/output mappato in memoria**

Ogni controller dispone di pochi registri usati per le comunicazioni con la CPU. Nello scrivere in questi registri il sistema operativo comanda al dispositivo di inviare dati, accettarli, porsi in stato di acceso o spento o eseguire qualche altra azione. Leggendo da questi registri, il sistema operativo apprende quale sia lo stato del dispositivo, se sia predisposto ad accettare un nuovo comando e così via.

Oltre ai registri di controllo, molti dispositivi hanno un buffer di dati su cui il sistema operativo può scrivere e leggere.

La questione che sorge è come la CPU comunichi con i registri di controllo e i buffer dei dati del dispositivo. Esistono tre alternative:

- port-mapped I/O o I/O isolato;

- memory-mapped I/O;

- hybrid-mapped I/O.

**Port-mapped I/O**

Comunica come con un vero e proprio dispositivo di I/O.

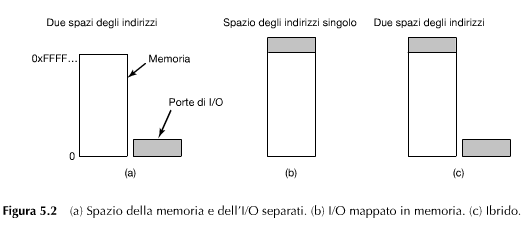
A ciascun registro di controllo è assegnato un numero di porta di I/O, un intero a 8 o a 16 bit. L’insieme di tutte le porte di I/O forma lo spazio delle porte di I/O ed è protetto in modo che i normali programmi utente non possano accedervi (lo può fare solo il sistema operativo).

Le istruzioni utilizzate dalla CPU per leggere/scrivere i registri di controllo all’indirizzo PORT sono:

- *IN REG,PORT*: la CPU legge il registro di controllo *PORT* e salva il risultato nel registro *REG*;

- *OUT PORT,REG*: la CPU scrive il contenuto di *REG* in un registro di controllo del dispositivo.

In tale schema, gli spazi degli indirizzi della memoria e dell’I/O sono diversi, come illustrato nella Figura 5.2(a).

****

Poiché le informazioni sono mappate su indirizzi separati dalla memoria la cache non viene influenzata dai dati che transitano verso/dal dispositivo (contrariamente l’effetto sarebbe disastroso) e inoltre i dispositivi non devono esaminare i riferimenti di memoria per capire quando rispondere.

D’altro canto questo schema presenta i seguenti svantaggi:

- Per leggere e scrivere i registri di controllo del dispositivo sono necessarie istruzioni dedicate assembly (IN e OUT) e non esistono istruzioni C o C++ che possano realizzarle. L’utilizzo di queste istruzioni aggiunge overhead al controllo dell’I/O.

- È necessario un meccanismo di protezione speciale per controllare lo svolgimento delle operazioni di I/O da parte dei processi utente.

- A differenza dell’I/O mappato in memoria che vede i registri di controllo del device come locazioni di memoria (e quindi variabili di un qualsiasi programma C/C++), occorre un passaggio ulteriore per spostare il contenuto dal registro della CPU alla memoria.

- I driver di controllo dei dispositivi non possono essere scritti utilizzando esclusivamente il linguaggio C (o C++).

- Il sistema operativo non riesce ad assegnare in modo semplice e dinamico i dispositivi ai processi utente.

**Memory-mapped I/O**

Il secondo approccio, introdotto con il PDP-11, consiste nel mappare tutti i registri di controllo nella memoria. A ciascun registro di controllo viene assegnato un indirizzo di memoria univoco non più utilizzabile come spazio di memoria principale.

Generalmente gli indirizzi assegnati sono quelli nella parte superiore dello spazio degli indirizzi, come mostrato nella Figura 5.2(b).

Questo schema presenta i seguenti vantaggi:

- I registri di controllo dei dispositivi sono delle variabili in memoria e possono essere gestiti da un programma C/C++.

- Il sistema operativo deve semplicemente evitare di mettere lo spazio di indirizzo contenente i registri di controllo nello spazio di indirizzamento virtuale di qualsiasi utente.

- Ogni dispositivo ha i suoi registri di controllo su una pagina diversa dello spazio di indirizzamento, in questo modo il sistema operativo può assegnare dinamicamente i dispositivi includendo le pagine desiderate.

- È sufficiente una sola istruzione di test in memoria per controllare lo stato del dispositivo.

- I driver di controllo possono essere scritti in C/C++.

**Hybrid**

Il terzo approccio è uno schema ibrido, mostrato nella Figura 5.2(c).

Questo schema presenta un buffer dei dati dei dispositivi di I/O mappati in memoria e porte di I/O separate per i registri di controllo. Il Pentium utilizza questa architettura, con gli indirizzi da 640K a 1 M riservati per i buffer dei dati dei dispositivi, oltre alle porte di I/O da 0 a 64K-1.

**Direct memory access (DMA)**

A prescindere se una CPU abbia o meno dispositivi mappati in memoria, ha bisogno di accedere ai controller dei dispositivi per scambiare dati con loro. La CPU può richiedere dati da un controller di I/O un byte alla volta, ma in questo modo spreca il tempo della CPU, quindi spesso si usa uno schema diverso, chiamato DMA (direct memory access - accesso diretto alla memoria).

Diamo per scontato che la CPU acceda a tutti i dispositivi e alla memoria con un singolo bus, come illustrato nella Figura 5.4.

Il sistema operativo può usare il DMA solo se l’hardware ha un controller DMA, cosa che la maggior parte dei sistemi possiede.

Il controller DMA ha accesso al bus di sistema indipendentemente dalla CPU, come mostrato nella Figura 5.4, non importa dove sia posto fisicamente. Esso contiene molti registri che possono essere scritti e letti dalla CPU, tra cui un registro degli indirizzi di memoria, un registro dei conteggi dei byte e uno o più registri di controllo. I registri di controllo specificano le porte di I/O da usare, la direzione del trasferimento (lettura dal dispositivo di I/O o scrittura verso il dispositivo di I/O), l’unità di trasferimento (un byte o una parola alla volta) e il numero di byte da trasferire alla volta.

Quando si usa il DMA la procedura di lettura dal disco è la seguente.

Per prima cosa, la CPU programma il controller DMA impostando i suoi registri in modo che conosca che cosa trasferire e dove (passo 1 della Figura 5.4). Invia anche un comando al controller del disco richiedendogli di leggere i dati dal disco nel suo buffer interno e di verificare il codice di controllo. Quando nel buffer del controller del disco vi sono dati validi, il DMA può partire.

Il controller DMA inizia il trasferimento inviando sul bus una richiesta di lettura al controller del disco (passo 2). Questa richiesta di lettura si presenta come qualunque altra richiesta e il controller del disco non sa, o non è interessato a sapere, se arriva dalla CPU o dal controller DMA.

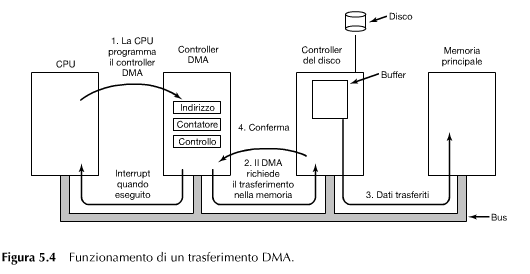
L’operazione di scrittura in memoria è un altro ciclo di bus standard (passo 3).

Quando l’operazione di scrittura è completata, il controller del disco manda un segnale di conferma sul controller DMA, sempre tramite il bus (passo 4).

Il controller DMA incrementa poi l’indirizzo di memoria da usare e diminuisce il conteggio dei

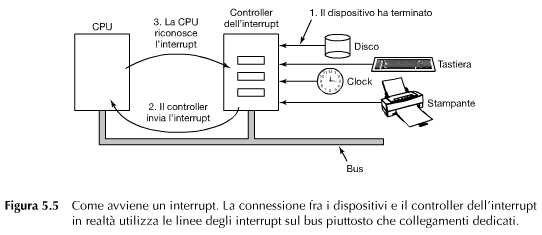
byte. Se questo conteggio è ancora maggiore di 0, allora i passi da 2 a 4 vengono ripetuti, finché il conteggio arriva a 0.

A quel punto il controller DMA manda un interrupt alla CPU per avvisarla che il trasferimento è completato. Quando l’esecuzione passa al sistema operativo, non deve copiare il blocco del disco in memoria, dato che è già lì.

****

**Ancora sugli interrupt**

In un personal computer tipico la struttura degli interrupt è quella illustrata nella Figura 5.5.

****

Quando un dispositivo di I/O finisce il lavoro che gli è assegnato, causa un interrupt, inviando un segnale su una linea del bus che gli è assegnata. Questo segnale è rilevato da un chip di un controller degli interrupt della scheda madre, che decide poi che cosa fare.

Se non ci sono altri interrupt in attesa, il controller degli interrupt lo processa immediatamente. Se ce n’è un altro in corso o un altro dispositivo ha fatto una richiesta simultanea sul bus a una linea di richiesta interrupt a maggiore priorità, il dispositivo è solo momentaneamente ignorato.

Per gestire l’interrupt, il controller assegna un numero alle linee degli indirizzi il cui dispositivo vuole attenzione e sottopone un segnale per interrompere la CPU.

Il segnale di interrupt fa sì che la CPU interrompa quanto stava facendo per iniziare a fare qualcos’altro. Il numero sulle linee degli indirizzi è usato come indice all’interno di una tabella chiamata vettore degli interrupt per prelevare un nuovo valore per il program counter, che punterà l’inizio della routine di servizio dell’interrupt (ISR).

Poco dopo il suo avvio, la procedura di servizio degli interrupt risponde all’interrupt, andando a scrivere un certo valore in una delle porte di I/O del controller degli interrupt. Questa risposta avvisa il controller che può mandarne un altro.

L’hardware salva sempre determinate informazioni prima di avviare la procedura di servizio; quali informazioni e dove vengono salvate varia decisamente da CPU a CPU. Di solito vengono salvati il program counter, i registri visibili, lo stack, …

Uno dei problemi è dove salvare le informazioni.

Una possibilità è metterle nei registri interni dove il sistema operativo può leggerle come e quando vuole. Un problema relativo a questo approccio è che poi il controller degli interrupt non può essere avvisato finché tutte le informazioni potenzialmente rilevanti non siano state lette, per timore che un secondo interrupt sovrascriva i registri interni salvando lo stato. Questa strategia porta a lunghi tempi morti quando gli interrupt sono disabilitati e ad una possibile perdita di interrupt e di dati.

Di conseguenza, la maggior parte delle CPU salva le informazioni nello stack.

Anche questo approccio presenta tuttavia dei problemi. Innanzitutto: in quale stack? Quello del processo utente o uno nel kernel? L’uso dello stack nel kernel è migliore di quello dei processi utente dal punto di vista dell’affidabilità dei riferimenti alle pagine (SP). Tuttavia, il passaggio alla modalità kernel può richiedere tempo di CPU a causa del cambio di contesto nella MMU.

**Interrupt precisi e imprecisi**

Un altro problema è causato dal fatto che la maggior parte delle moderne CPU usano frequentemente le pipeline e sono spesso superscalari. Nei sistemi più vecchi, al termine dell’esecuzione di ogni istruzione l’hardware controllava se ci fosse un interrupt in sospeso. Se era così, il contatore di programma e il PSW venivano inseriti nello stack e cominciava la sequenza di interrupt. Dopo l’esecuzione del gestore dell’interrupt aveva luogo il processo inverso, con il vecchio PSW e il contatore di programma prelevati dallo stack e la continuazione del processo precedente. In questo modello era implicito che, se si fosse verificato un interrupt appena dopo un’istruzione, tutte le istruzioni fino a quel punto sarebbero state eseguite completamente, mentre quelle che seguivano non lo sarebbero state.

Sulle macchine più vecchie questa affermazione era sempre vera, ma su quelle moderne potrebbe non esserlo.

Che cosa accade se si verifica un interrupt mentre la pipeline è piena (caso frequente)? Molte istruzioni si troverebbero in un diverso stato di esecuzione. Quando si verifica l’interrupt, il valore del contatore di programma potrebbe non riflettere il limite corretto fra le istruzioni eseguite e quelle non eseguite. Infatti, molte istruzioni potrebbero essere state eseguite parzialmente.

Un interrupt che lascia la macchina in uno stato ben definito è detto interrupt preciso.

Questo interrupt ha quattro proprietà:

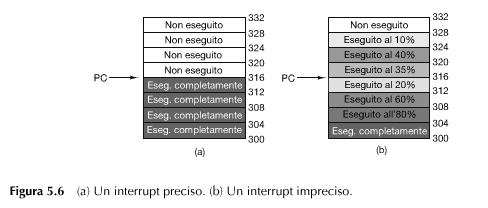
1) il PC (program counter) è salvato in un posto conosciuto;

2) tutte le istruzioni eseguite prima di quella puntata dal PC sono state completamente eseguite;

3) nessuna istruzione oltre a quella puntata dal PC è stata eseguita;

4) lo stato di esecuzione dell’istruzione puntata dal PC è conosciuto.

Un interrupt che non sottostà a queste specifiche è detto interrupt impreciso.

****

Macchine con interrupt imprecisi di solito delegano il sistema operativo per capire come gestire la situazione e riversano sullo stack grandi quantità di dati sullo stato interno. Solitamente, il codice necessario a riavviare la macchina è estremamente complicato. Inoltre salvare una gran quantità di informazioni in memoria a ogni interrupt rende gli interrupt lenti e ripristinare è anche peggio.

Questo genera una situazione paradossale in cui talvolta le CPU superveloci e superscalari non sono adatte a lavori real-time a causa di interrupt lenti.

**Princìpi del software di I/O**

Prima analizzeremo gli obiettivi di un software di I/O.

Poi ci dedicheremo ai diversi metodi per eseguire un I/O:

- I/O programmato;

- I/O guidato dagli interrupt (interrupt-driven);

- I/O con Direct Memory Access

**Obiettivi del software di I/O**

**Indipendenza dal dispositivo:** i programmi che gestiscono l’I/O devono poterlo fare indipendentemente dal tipo di dispositivo.

**Denominazione uniforme:** l’identificatore di un file o di un dispositivo non deve dipendere dal tipo di dispositivo.

**Gestione degli errori:** gli errori andrebbero gestiti il più possibile a livello hardware (controller, driver, …). Molti errori (come quelli di lettura dal disco) sono transitori e scompaiono se si ripete l’operazione.

**Trasferimenti sincroni/asincroni:** i trasferimenti possono essere bloccanti (sincroni) o gestiti con interrupt (non bloccanti, asincroni).

A causa della differente velocità, i dispositivi di I/O sono tipicamente asincroni. Tuttavia è molto più facile scrivere programmi utenti con primitive bloccanti (es. read()), quindi sta al SO dare al programmatore l’illusione di utilizzare chiamate sincrone quando in realtà i dispositivi sono trattati con primitive asincrone.

**Bufferizzazione:** i dati che vengono da un dispositivo non possono essere memorizzati direttamente nella loro destinazione finale e dovrebbero essere bufferizzati.

Per esempio, il sistema operativo non sa a chi è destinato un pacchetto che arriva dalla rete finché non è assemblato ed esaminato.

**Condivisione:** alcuni dispositivi di I/O (es. disco) sono risorse condivise e possono essere utilizzati da molti utenti contemporaneamente. Altri dispositivi, come ad esempio le unità a nastro, sono dedicate ad un singolo utente: nessuno può utilizzare il dispositivo finché l’utente non ha finito.

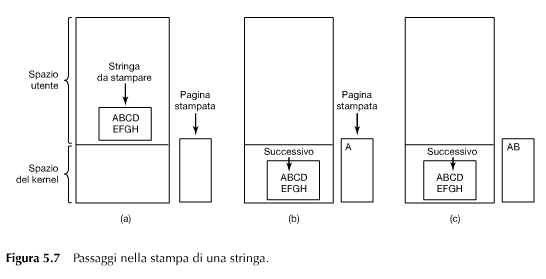
**I/O programmato**

L’I/O programmato è la tecnica più semplice perché viene tutto delegato alla CPU.

È più semplice illustrare l’I/O programmato con un esempio. Considerate un processo utente che voglia scrivere la stringa di otto caratteri “ABCDEFGH” sulla stampante.

Per prima cosa assembla la stringa in un buffer dello spazio utente, come mostrato dalla

Figura 5.7.

****

Il processo utente acquisisce poi la stampante per la scrittura, facendo una chiamata di sistema per aprirla. Se la stampante al momento è usata da un altro processo utente, questa chiamata fallirà, restituendo un codice di errore o si bloccherà finché la stampante non sarà disponibile, a seconda del sistema operativo e dei parametri della chiamata.

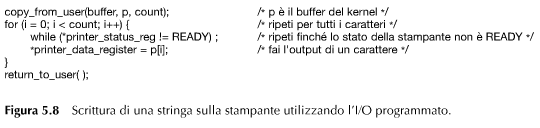
Una volta che ha la stampante, il processo utente esegue una chiamata di sistema richiedendo

al sistema operativo di stampare la stringa sulla stampante.

Il sistema operativo copia poi (di solito) il buffer con la stringa in un array, diciamo *p*, nello spazio kernel, dove è più facilmente accessibile. Controlla poi se la stampante è attualmente disponibile. Se non lo è aspetta finché lo sia. A quel punto il sistema operativo copia il primo carattere nel registro dati della stampante. Questa azione attiva la stampante.

La CPU esegue il polling della stampante per controllare se è pronta ad accettare un altro carattere (polling o busy waiting).

Le azioni seguite dal sistema operativo sono sintetizzate nella Figura 5.8.

****

L’I/O programmato è semplice ma ha lo svantaggio di utilizzare il tempo della CPU finché tutte le operazioni di I/O non sono finite.

Generalmente, il busy waiting è una prassi che andrebbe limitata ai soli casi di I/O veloce.

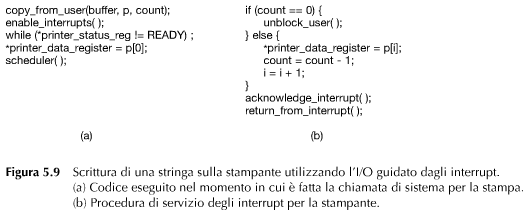
Esistono metodi migliori.

**I/O guidato dagli interrupt**

Andiamo ora a considerare il caso di una stampa su una stampante che non ha buffer ma stampa un carattere dopo l’altro man mano che arrivano. Se la stampante può stampare per esempio 100 caratteri/s, ogni carattere impiega 10 ms per essere stampato. Ciò significa che, dopo l’operazione di scrittura nel registro dei dati della stampante di ciascun carattere, la CPU si fermerà in un ciclo di inattività di 10 ms, in attesa che le sia consentito l’output del carattere successivo. Questo tempo è più che sufficiente per attivare un cambio di contesto ed eseguire qualche altro processo in quei 10 ms che altrimenti andrebbero persi.

Per permettere alla CPU di fare qualcos’altro nell’attesa che la stampante sia pronta è possibile usare gli interrupt. Avvenuta la chiamata di sistema per stampare la stringa, il buffer è copiato nello spazio del kernel, come abbiamo mostrato prima, e il primo carattere è copiato nella stampante appena è in grado di accettarlo. A questo punto la CPU richiama lo scheduler e viene eseguito un altro processo. Il processo che ha richiesto la stampa della stringa è bloccato finché non è stampata l’intera stringa.

Il lavoro fatto sulla chiamata di sistema è mostrato nella Figura 5.9(a).

****

Quando la stampante ha stampato il carattere ed è pronta ad accettare il successivo, genera un interrupt. Questo interrupt ferma il processo attuale e salva il suo stato. Poi è eseguita la procedura di servizio dell’interrupt della stampante. Una versione piuttosto grezza di questo codice è mostrata nella Figura 5.9(b).

Se non ci sono più caratteri da stampare, il gestore degli interrupt esegue qualche azione per sbloccare l’utente. Altrimenti, esegue l’output del carattere successivo, avvisa l’interrupt e ritorna al processo che stava eseguendo appena prima dell’interrupt, continuando da dove era stato lasciato.

**I/O con DMA**

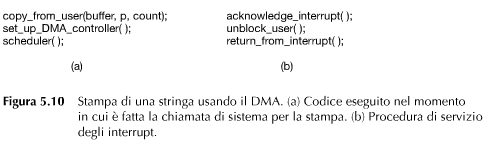
Uno svantaggio ovvio dell’I/O guidato dagli interrupt è che avviene un interrupt a ogni carattere. Gli interrupt richiedono tempo, pertanto questo schema spreca una certa quantità di

tempo CPU. Una soluzione è l’uso del DMA.

In questo caso l’idea è di lasciare che il controller DMA invii i caratteri alla stampante uno alla volta, senza disturbare la CPU. In sostanza, il DMA è I/O programmato, con solo il controller DMA che fa tutto il lavoro, invece della CPU principale.

Questa strategia richiede sì un hardware speciale (il controller DMA), ma lascia che la CPU faccia altre cose durante l’I/O.

Uno schema sommario del codice è dato nella Figura 5.10.

****

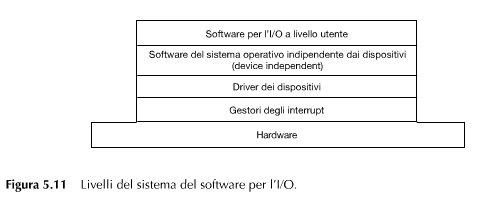
Il successo del DMA consiste nella riduzione del numero degli interrupt da uno per carattere a uno per buffer stampato. Nel caso di numerosi caratteri e interrupt lenti, si tratta di un grande miglioramento.

Dall’altra parte il controller DMA è generalmente molto più lento della CPU principale. Se il controller DMA non è in grado di condurre il dispositivo a velocità massima o se la CPU non ha comunque nulla da fare mentre è in attesa dell’interrupt del DMA, allora potrebbe andar meglio l’I/O gestito dagli interrupt o anche l’I/O programmato. La maggior parte delle volte vale la pena usare il DMA.

**Livelli del software di I/O**

Il software di I/O è generalmente organizzato in quattro livelli, come mostrato nella

Figura 5.11, ciascuno dei quali ha una funzione ben definita da eseguire e un’interfaccia ben definita verso i livelli adiacenti.

****

**Gestori degli interrupt**

Anche se l’I/O programmato è talvolta utile, si utilizzano diffusamente gli interrupt per gestire i dispositivi di I/O.

Gli interrupt dovrebbero rimanere così nascosti in profondità che il sistema operativo potrebbe quasi ignorarne l’esistenza. Il miglior modo per nasconderli è fare in modo che il driver che avvia un’operazione di I/O si blocchi finché l’I/O non è terminato e si verifica l’interrupt.

Il driver può bloccarsi facendo per esempio una down su un semaforo, una wait su una variabile condizione, una receive su un messaggio o qualcosa di simile.

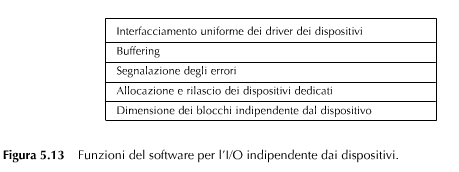
Quando si verifica l’interrupt, la procedura fa quanto deve per gestirlo. Quindi può sbloccare il driver che l’ha fatto partire. In alcuni casi si completerà con una up su un semaforo. In altri invierà una signal su una variabile condizione in un monitor. In altri ancora invierà un messaggio al driver bloccato. In ogni caso l’effetto dell’interrupt sarà che un driver in precedenza bloccato potrà essere adesso in esecuzione.

Questo modello funziona meglio se i driver sono strutturati come processi kernel, con i loro stati, stack e contatori di programma personali.

**Software per l’I/O indipendente dal dispositivo**

Sebbene parte del software per l’I/O sia specifico di un determinato dispositivo, altre parti sono indipendenti dal dispositivo stesso (device independent). Il limite esatto fra i driver e il software indipendente dal dispositivo dipende dal sistema e dal dispositivo, poiché alcune funzioni che potrebbero essere svolte in modalità indipendente dal dispositivo sono effettivamente svolte dai driver, sia per efficienza sia per altre ragioni.

Le funzioni mostrate nella Figura 5.13 sono eseguite tipicamente da software indipendente dal dispositivo.

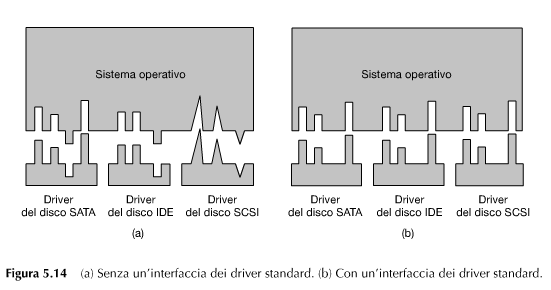
****

La funzione base del software indipendente dal dispositivo è quella di eseguire tutte quelle funzioni di I/O trasversali a tutti i dispositivi e di fornire un’interfaccia uniforme al software a livello utente.

**Interfacciamento uniforme dei driver dei dispositivi**

Una questione fondamentale di un sistema operativo è come rendere tutti i dispositivi e i driver più o meno simili tra loro. La modalità di interfacciamento tra driver e sistema operativo deve essere standard per ogni dispositivo. Senza standard l’interfacciamento di un nuovo driver richiede uno sforzo di programmazione enorme.

Nella Figura 5.14(a) è illustrata una situazione in cui ogni driver ha una diversa interfaccia verso il sistema operativo.

****

Diversamente, nella Figura 5.14(b) è illustrato un diverso modello in cui tutti i driver hanno la stessa interfaccia, dove è molto più semplice inserire un nuovo driver, facendo in modo che sia conforme alla loro interfaccia.

Un altro aspetto dell’uniformità di un’interfaccia riguarda la denominazione dei dispositivi di I/O. Il software indipendente dal dispositivo si prende cura di mappare i nomi simbolici dei dispositivi nel driver adatto.

Per esempio, in UNIX un nome di dispositivo, come */dev/disk*0, specifica in modo univoco l’i-node per un file speciale e questo i-node contiene il major device number (numero di dispositivo primario), usato per localizzare il driver adeguato. L’i-node contiene anche il minor device number (numero di dispositivo secondario), passato come un parametro al driver al fine di specificare l’unità da leggere o da scrivere. Tutti i dispositivi hanno un numero di dispositivo

primario e secondario e l’accesso a tutti i driver avviene selezionandoli usando il numero

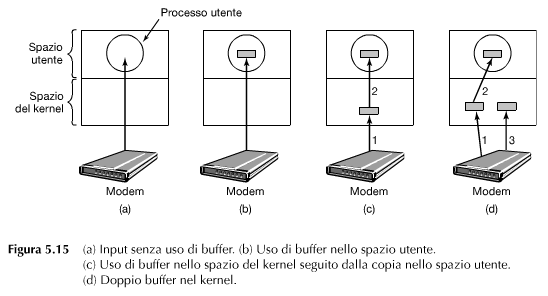
di dispositivo primario.

**Buffering**

Il buffering costituisce un problema, sia per i dispositivi a blocchi sia per quelli a caratteri.

Consideriamo un processo che voglia leggere dei dati da un modem. Una possibile strategia per gestire i caratteri in ingresso consiste nell’avere un processo utente che esegue una chiamata di sistema read e si blocca in attesa di un carattere. Ogni carattere in arrivo causa un interrupt.

La procedura di servizio degli interrupt presenta il carattere al processo utente e si sblocca. Questo modello è indicato nella Figura 5.15(a).

****

Il problema che sorge da questo schema è che il processo utente deve essere riavviato a ogni carattere in ingresso. Consentire che un processo sia eseguito più volte per brevi periodi non è efficiente e quindi questo schema non è quello valido.

Un miglioramento è mostrato dalla Figura 5.15(b).

In questo caso il processo fornisce un buffer di *n* caratteri nello spazio utente e fa una lettura di *n* caratteri. La procedura di servizio degli interrupt mette i caratteri in ingresso in questo buffer finché non è pieno. Solo a quel punto risveglia il processo utente. Questo schema è molto più efficiente del precedente, ma ha un rovescio della medaglia: che accade se il buffer viene paginato in uscita all’arrivo di un carattere?

Un ulteriore approccio è creare un buffer all’interno del kernel e avere il gestore degli interrupt che vi mette i caratteri, come mostrato nella Figura 5.15(c). Quando questo buffer è pieno, la pagina con il buffer utente è portata dentro, se necessario, e il buffer copiato in una sola operazione. Questo schema è di gran lunga più efficiente.

Tuttavia, anche qui sorge un problema: che accade ai caratteri che arrivano mentre la pagina con il buffer utente viene letta dal disco? Dato che il buffer è pieno, non c’è posto dove metterli.

Una via d’uscita è avere un secondo buffer nel kernel. Dopo che si riempie il primo, ma prima che sia stato svuotato, si utilizza il secondo, come nella Figura 5.15(d). Al riempirsi del secondo lo si copia in quello utente (dando per scontato che l’utente lo abbia richiesto). Durante questa fase con il secondo buffer, il primo è riusato con i nuovi caratteri. In questo modo i due buffer fanno a turno: mentre uno viene usato per copiare nello spazio utente, l’altro accumula i nuovi input. Uno schema di buffering di questo genere è detto buffering doppio.

Un’altra forma di buffering usata diffusamente è detta buffer circolare. È composta da una zona di memoria e due puntatori, uno dei quali punta alla parola libera successiva (Pfine), dove poter mettere i nuovi dati, mentre l’altro (Pinizio) punta alla prima parola dei dati nel buffer che non è stata ancora rimossa.

Pfine avanza quando arrivano nuovi dati dal modem; Pinizio avanza quando il sistema operativo elimina ed elabora i dati. Quando il buffer è pieno Pinizio = Pfine.

Il buffering è una tecnica largamente utilizzata, ma occorre far attenzione all’eccesso di copia, da uno spazio all’altro, che rallenta le prestazioni.

**Dischi**

**Algoritmi di scheduling del braccio del disco**

In questo paragrafo analizzeremo alcune questioni generali relative ai driver dei dischi.

Per prima cosa, considerate quanto tempo serve per la lettura o la scrittura di un blocco del disco. Il tempo richiesto è determinato da tre fattori:

1. il tempo di ricerca (il tempo per muovere il braccio al giusto cilindro);

2. il ritardo rotazionale (il tempo affinché il settore giusto ruoti sotto la testina);

3. il tempo effettivo di trasferimento dei dati.

Per la maggior parte dei dischi, il tempo di ricerca è decisamente maggiore degli altri due, cosicché la riduzione del tempo medio di ricerca migliora sostanzialmente le prestazioni del

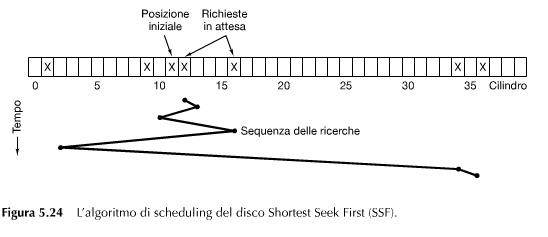
sistema.

Se il driver del disco accetta una richiesta per volta e le esegue nello stesso ordine, cioè first-come, first-served (FCFS), c’è ben poco da fare per ottimizzare il tempo di ricerca.

È tuttavia possibile una diversa strategia quando il disco è utilizzato frequentemente. È probabile che, mentre il braccio sta compiendo una ricerca sulla base di una richiesta, altri processi generino altre richieste per il disco. Molti driver dei dischi mantengono una tabella, indicizzata per numero di cilindro, con tutte le richieste in attesa di ciascun cilindro. Data questo tipo di struttura dei dati, è possibile migliorare l’algoritmo di scheduling FCFS.

Per vedere in che modo, considerate un disco immaginario con 40 cilindri. Arriva una richiesta di lettura di un blocco sul cilindro 11. Mentre è in corso la ricerca del cilindro 11, arrivano nuove richieste dei cilindri 1, 36, 16, 34, 9 e 12, in quest’ordine.

Le richieste sono mostrate nella Figura 5.24.

****

Al termine della richiesta attuale (per il cilindro 11), il driver deve scegliere quale richiesta gestire come successiva. Se usasse l’FCFS come cilindro successivo avrebbe l’1, poi il 36 e così via. Questo algoritmo richiederebbe rispettivamente movimenti del braccio di 10, 35, 20, 18, 25 e 3 cilindri, per un totale di 111.

In alternativa, per ridurre al minimo il tempo di ricerca potrebbe sempre gestire come richiesta seguente la più vicina. Date le richieste della Figura 5.24, la sequenza è 12, 9, 16, 1, 34 e 36, illustrate nella Figura 5.24 dalla linea a zig zag alla base.

Con questa sequenza gli spostamenti del braccio sono 1, 3, 7, 15, 33 e 2, per un totale di 61 cilindri. Questo algoritmo, shortest seek first (SSF - “la ricerca più breve per prima”), riduce lo spostamento totale del braccio quasi della metà in confronto all’FCFS.

Sfortunatamente l’SSF comporta un problema. Supponete che continuino ad arrivare più richieste durante il processo di quelle della Figura 5.24. Se, per esempio, dopo essere andato al cilindro 16, giunge una nuova richiesta per il cilindro 8, quella richiesta avrà priorità rispetto a quella del cilindro 1. Se arriva poi una richiesta del cilindro 13, il braccio vi si porterà escludendo di nuovo il cilindro 1. Con un disco che subisce un forte carico di lavoro, il braccio avrà la tendenza a restare per la maggior parte del tempo nel mezzo del disco, pertanto le richieste a entrambi gli estremi dovranno aspettare finché una fluttuazione statistica del carico di lavoro porterà a non avere richieste nel mezzo. Le richieste lontane dal centro avranno un servizio scadente. In questo caso gli obiettivi di un tempo di risposta basso e di imparzialità sono in contrasto.

Anche gli edifici alti hanno a che fare con questo problema. Il problema dello scheduling di un ascensore in un edificio con tanti piani è simile a quello del braccio di un disco.

Per trovare un punto d’incontro nell’eterno conflitto tra efficienza e imparzialità, la maggior parte degli ascensori usa tuttavia un algoritmo diverso: continuano a muoversi in una direzione finché non vi sono più richieste inevase in quella direzione, quindi cambiano direzione.

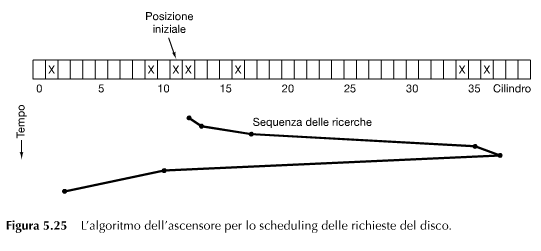
Questo algoritmo, noto sia nel mondo dei dischi sia in quello degli ascensori con il nome di algoritmo dell’ascensore (elevator algorithm), richiede al software di gestire un bit: il bit della direzione attuale, UP o DOWN. Al termine di una richiesta, il driver del disco o dell’ascensore controlla il bit.

Se è UP, il braccio o la cabina sono spostati alla richiesta successiva in attesa verso l’alto. Se non vi sono richieste in attesa più in alto allora viene invertito il bit della direzione.

Quando il bit è impostato a DOWN, lo spostamento è nella posizione successiva inferiore richiesta, qualora ve ne sia una.

La Figura 5.25 mostra l’algoritmo dell’ascensore usando le stesse sette richieste della

Figura 5.24, considerando che il bit della direzione sia inizialmente UP.

****

L’ordine in cui sono serviti i cilindri è 12, 16, 34, 36, 9 e 1, il che comporta spostamenti del braccio di 1, 4, 18, 2, 27 e 8 cilindri, per un totale di 60. In questa situazione l’algoritmo dell’ascensore è leggermente migliore dell’SSF, sebbene in genere sia peggiore.

Una caratteristica simpatica dell’algoritmo dell’ascensore è che, data una qualunque raccolta di richieste, il limite superiore dello spostamento massimo è fisso: è due volte il numero dei cilindri.

**Gestione degli errori**

I produttori di dischi si spingono sempre più oltre ai limiti della tecnologia, incrementando

le densità lineari dei bit.

Sfortunatamente è impossibile produrre un disco senza alcun difetto. I difetti di produzione presentano settori difettosi (bad sectors) ossia settori che non rileggono correttamente il valore che vi è appena stato scritto.

Se si tratta di un errore molto piccolo, diciamo di pochi bit, è possibile utilizzare il settore difettoso e far si che l’ECC(Hamming) corregga ogni volta gli errori.

Se il difetto è più grande, non può essere mascherato.

Gli approcci generali ai blocchi difettosi sono due: trattarli a livello di controller oppure

a livello di sistema operativo.

**Memoria stabile**

Come abbiamo visto, i dischi talvolta commettono degli errori. Settori buoni possono rivelarsi improvvisamente difettosi. Interi dischi possono smettere di funzionare inaspettatamente.

I RAID proteggono da qualche settore che diviene difettoso o anche dal crash di un intero disco, ma non proteggono errori di scrittura che alterano i dati originali senza sostituirli dai nuovi.

L’ideale sarebbe un disco che lavorasse tutto il tempo senza errori. Sfortunatamente, è irrealizzabile.

Quello che si può avere è un sistema stabile che quando gli arriva un comando di scrittura, ha due possibilità: scrivere i dati correttamente o non fare nulla. L’obiettivo è la consistenza del disco a ogni costo.

La memoria stabile utilizza una coppia di dischi identici con i blocchi corrispondenti che lavorano insieme a formare un blocco esente da errori. In assenza di errori i blocchi corrispondenti di entrambi i dischi sono identici. Qualunque dei due sia letto, il risultato è

lo stesso.

Per ottenere questo risultato sono definite le tre operazioni seguenti.

1. Operazioni di scrittura stabili. Un’operazione di scrittura stabile consiste nello scrivere prima il blocco sull’unità 1, quindi leggerlo per verificare che sia stato scritto correttamente.

Se non è stato scritto correttamente allora l’operazione di scrittura e la lettura sono rifatte per *n* volte finché non funzionano. Dopo *n* tentativi falliti il blocco è rimappato con uno di riserva e l’operazione ripetuta finché non funziona, non importa quanti blocchi di riserva debbano essere utilizzati.

Non appena l’operazione di scrittura sul disco 1 è andata a buon fine, il blocco corrispondente viene scritto sul disco 2 e quindi riletto, ripetutamente se necessario, finché anch’esso non funziona.

2. Operazioni di lettura stabili. Un’operazione di lettura stabile legge per primo un blocco dal disco 1. Se questo porta a un ECC scorretto, l’operazione di lettura si ritenta di nuovo, per *n* volte. Se risulta ancora un ECC sbagliato, viene letto il blocco corrispondente dal disco 2 (è altamente improbabile che siano entrambi guasti).

3. Crash recovery (ripristino da un crash). Dopo un crash, un programma di ripristino fa la scansione di entrambi i dischi per fare il confronto dei blocchi corrispondenti.

Se una coppia di blocchi è valida e sono identici, non viene fatto nulla.

Se uno di loro presenta un errore di ECC, il blocco difettoso è sovrascritto con quello valido.

Se sono entrambi senza errori ma diversi fra loro, il blocco sul disco 1 sovrascrive il blocco del

disco 2.

**Clock**

I clock (chiamati anche timer) sono fondamentali ai fini del funzionamento di un sistema

multiprogrammato per molte ragioni.

Mantengono l’orario del giorno e, fra le altre cose, evitano che un processo possa monopolizzare la CPU.

Il software del clock può presentarsi sotto forma di driver di dispositivo, sebbene il clock non sia né un dispositivo a blocchi né un dispositivo a caratteri.

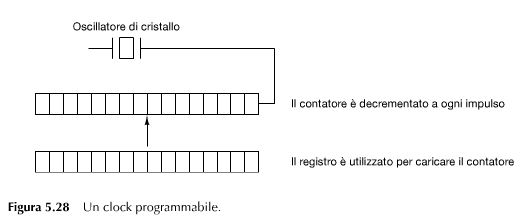
**Hardware del clock**

Nei computer sono utilizzati comunemente due tipi di clock, entrambi abbastanza diversi

dagli orologi a cui siamo abituati.

I clock più semplici sono connessi a un’alimentazione a 110 volt o a 220 volt e provocano un interrupt ogni ciclo di voltaggio, a 50 0 60 Hz. Questi clock erano i più diffusi, ma oggi sono rari.

L’altro tipo di clock è composto da tre componenti: un oscillatore a cristalli, un contatore (decrementato ad ogni impulso) e un registro (utilizzato per caricare il contatore), come mostrato nella Figura 5.28.

****

Quando un pezzo di cristallo di quarzo è messo in tensione, per la sua proprietà piezo-elettrica, genera un segnale periodico con estrema precisione, generalmente nell’intervallo di parecchie

centinaia di megahertz, a seconda del cristallo scelto.

Il segnale base può essere moltiplicato per un numero intero per arrivare a frequenze dell’ordine dei GHz. Questi circuiti sono presenti in ogni computer.

Il segnale è inviato ad un contatore che decrementa il suo valore fino a zero, genera l’interrupt e poi ricomincia.

I clock programmabili hanno due modalità di funzionamento:

- one-shot: quando il clock parte, copia il valore del registro nel contatore e quindi decrementa il contatore ad ogni impulso del cristallo. Quando il contatore arriva a zero, provoca un interrupt e si arresta finché non viene esplicitamente riavviato dal software.

- onda quadra: dopo aver raggiunto lo zero e causato l’interruzione, il registro è copiato nel contatore e il processo si ripete all’infinito. Gli interrupt periodici sono detti clock ticks.

Il vantaggio dei clock programmabili è che la frequenza di interrupt può essere controllata dal software: con un cristallo a 500MHz (T = 2 ns) il contatore decrementa ogni 2 ns, se si usa un registro a 32 bit gli interrupt si possono programmare da 2 ns a 8,6 s.

**Software del clock**

Tutto quello che l’hardware del clock fa è generare interrupt a intervalli ben precisi.

Il resto avviene tramite il software, il driver del clock.

I compiti precisi del driver del clock cambiano a seconda dei sistemi operativi; i principali consistono nel:

1) mantenere l’ora del giorno;

2) evitare che i processi siano eseguiti più tempo di quanto consentito;

3) contabilizzare l’uso della CPU;

4) gestire la system call *alarm()* invocata dai processi utente;

5) fornire il timer watchdog per le componenti del SO che ne fanno richiesta;

6) eseguire il profiling, il monitoraggio ed elaborazioni statistiche.

**Soft timer**

Quasi tutti i computer hanno un secondo orologio programmabile che può essere impostato per causare le interruzioni a qualsiasi frequenza.

I timer soft evitano interruzioni: è il kernel che, quando è in esecuzione per qualche ragione, prima di rientrare nella modalità utente controlla che non sia scaduto un timer soft.

I timer soft sono controllati con la stessa velocità con cui il kernel entra in azione per svolgere altri compiti, quindi quando:

- viene attivata una system call;

- accade una page miss della TLB;

- accade un page fault;

- accade un interrupt di I/O;

- la CPU diventa inattiva (idle).