

Principi dell'hardware di Input e Output

Il codice per I/O rappresenta una **parte significativa** della totalità del SO.


Il SO deve anche supportare l'utente nel controllo dei dispositivi di input e output, quindi deve:

- **Inviare comandi ai dispositivi.**
- **Intercettare interrupt.**
- **Fornire interfaccia semplice per gli utenti.**

 **L'interfaccia dovrebbe essere la stessa per tutti i dispositivi**

Possiamo analizzare I/O da due punti di vista:

- **Componenti fisici** --> Ingegneri elettronici che mettono a disposizione i componenti **fisici** (Chip, Cavi, Alimentatori).
- **Interfaccia software** -> Programmatori mettono a disposizione i **comandi** che possono essere eseguito dal dispositivo.

 **Il nostro interesse è focalizzato sulla programmazione dell'hardware e non su funzionamento interno dei dispositivi.**

Dispositivi di I/O

Esistono due diverse categorie di dispositivi di I/O:

- **Dispositivi a blocchi** --> Astraggono i dati a **forma di blocchi** di dimensione fissa (Da 512 a 32.768 byte), ognuno con il proprio indirizzo. Questi dati possono essere trasferiti in qualsiasi punto del output. I blocchi possono essere **letti o scritti in maniera indipendente**.
- **Dispositivi a caratteri** --> Flusso non strutturato sia in ingresso che in uscita di caratteri. (Stampanti, tastiere). Non è indirizzabile e **non ha alcuna operazione di ricerca**.

Alcuni dispositivi rimangono fuori da questa classificazione come schermi, clock, touch screen.

Il **file system**, per esempio, si occupa solo di **dispositivi a blocchi astratti** e lascia la parte dipendente dai dispositivi a software di livello più basso.

Uno dei problemi principali per il software di gestione è gestire la **differenza di velocità di trasferimento tra le differenti periferiche**.

Controller dei dispositivi

Le componenti dei dispositivi dell' I/O:

- **Parte meccanica**(dispositivi stesso)
- **Parte elettronica** (controller del dispositivo o adattatore).

La componente elettronica è detta **controller del dispositivo** spesso presente sottoforma di **chip** sulla scheda madre o come scheda aggiuntiva su slot PCIe.

La scheda del controller ha un connettore in cui si inserisce un cavo che si collega al dispositivo stesso.

L'interfaccia tra il controller e il dispositivo è un' interfaccia **conforme ad uno standard:**

- **ANSI**
- **IEEE**
- **ISO**

Possono essere anche interfacce *de facto*:

- **SATA**
- **SCSI**
- **USB**
- **ThunderBolt**

Il compito del controller è **convertire il flusso seriale di bit in un blocco di byte** ed eseguire le correzioni degli errori necessarie.

Il blocco dei byte è prima assemblato bit per bit, in un **buffer interno al controller**, dopo aver controllato l'assenza degli errori, il blocco può essere **copiato nella memoria principale**.

Senza il controller, **il programmatore dovrebbe gestire dettagli complessi**.

Il controller fortunatamente **gestisce autonomamente** dettagli complessi.

 **Esistono decine di bus che fanno comunicare bus e periferiche.**

 **Alcuni chip della CPU**

Northbridge -> Serve a connetterci a componenti veloci.

Southbridge -> Serve a connetterci a componenti lente.

"Vecchia" porta parallela

Interfaccia utilizzata inizialmente per collegare un computer ad una stampante (Comunicazione Unidirezionale), in seguito, usata per **comunicazioni bidirezionali** con altre periferiche.

Abbiamo **25 o 36 spinotti** usate per spedire bit simultaneamente su più canali.

- pin 1-8 -> Servono a trasmettere i **dati**.
- pin 9-16 -> Servono per **controlli e status**.
- pin 17-25 -> Forniscono il **ritorno elettrico e la connessione di terra**.

Una delle caratteristiche è la **velocità variabile a seconda del dispositivo collegato**.

Porta USB

Interfaccia standardizzata per la **connessione e comunicazione** tra dispositivi e computer. Utilizzata per trasferire dati e fornire alimentazione elettrica.

Composto da 4 pin:

- Pin 1 (**Vcc**) -> Fornisce alimentazione.
- Pin 2 (**D-**) -> Dati negativi.
- Pin 3 (**D+**) -> Dati positivi.
- Pin 4 (**GND**) -> Terra.

Solitamente USB 2.0 è usato per le periferiche come tastiere e mouse.

Uno dei vantaggi principali è **l'ampia applicazione**, infatti la possiamo ritrovare in molti dispositivi ed è inoltre **semplice** da usare. E' anche **retrocompatibile**.

Sguardo ai circuiti del disco

- **PCB** -> La Printed Circuit Board è realizzata in vetro verde e rame, dotato di connettori SATA e di alimentazione. Questa **comunica con il controller SATA e i bus** e supporta i componenti elettronici dell' HDD.
- **MCU** -> Al centro della PCB, essenziale per le operazioni con HDD, inoltre include: una CPU e un **canale di lettura e scrittura per convertire segnali analogici in digitali**.
- **Memoria cache** -> chip di memoria di tipo DDR SDRAM che definisce la cache dell'HDD.
- **Controller VCM** -> **Controlla la rotazione del motore del disco** e i movimenti delle testine, consuma la maggior parte di energia.
- **Chip flash** -> Memorizza **parte del firmware dell'HDD**, essenziale per l'avvio.

- **Sensori di shock e diodi TVS** -> **Proteggono HDD da urti e sovratensioni**, il sensore invia rileva urti inviando segnali al controllo VCM. Mentre i diodi si sacrificano in caso di picco di tensione.

I/O mappato in memoria

I controller dispongono di **registri** usati per comunicare con la CPU. Il SO può ordinare al dispositivo di **scrivere** nei registri o può **leggerli** verificando lo stato del dispositivo.

Molti dispositivi hanno dei buffer di dati su cui il SO legge e scrive

Esistono due approcci principali per far sì che la CPU e il dispositivo comunichino:

- **Port-mapped I/O**
- **Memory mapped I/O**
- **Approccio ibrido**

Il modo in cui implementiamo la comunicazione è **fondamentale** nel SO, poiché incide sulla progettazione e sulle prestazioni del sistema.

Port mapped I/O

Ciascun registro di controllo è assegnato un numero di **porta I/O**, l'insieme di tutte le porte forma lo **spazio delle porte di I/O**. Al quale vi può accedere solo il Sistema Operativo tramite delle istruzioni speciali, come:

```
IN REG, PORT
OUT PORT, REG
```

In questo modo la CPU legge il registro di controllo port e inserisce il risultato nel registro della CPU chiamato REG. Allo stesso modo con OUT scriviamo il contenuto di REG in un registro di controllo.

In questo metodo **lo spazio degli indirizzi di memoria e dell'I/O sono diversi**.

Il tipo di linguaggio usato deve essere **riconosciuto dal device**.

Vantaggio principale

Siccome lo spazio degli indirizzi di memoria e quello delle porte è diverso **non abbiamo alcun problema nella gestione della memoria**.

Memory mapped I/O

Mappiamo tutti i registri di controllo nello spazio di memoria, ad ogni registro di controllo assegniamo uno spazio di memoria univoco e libero.

 Gli indirizzi assegnati sono quelli nella parte superiore dello spazio di indirizzi.

Vantaggi:

- Elimina necessità di istruzioni speciali (Assembly)
- Registri possono essere trattati come variabili in C.
- Processi utenti non possono accedere direttamente ai registri di controllo.
- Controllo selettivo dei dispositivi.

Possibile loop infinito

Nasce un problema nel momento in cui interviene la cache. Infatti quando facciamo un riferimento alla parte di memoria destinata al registro di controllo, la cache si salverà il dato del registro in modo tale da riprenderlo se serve.

Il problema nasce quando facciamo nuovamente un riferimento alla stessa parte di memoria, infatti il dispositivo di I/O cambia costantemente il suo stato o dati senza aggiornare la cache quindi, con il nuovo riferimento potremmo andare a prendere il dato che si trova nella cache che però non è aggiornato.

Se il programma eseguito si aspetta un cambiamento dei dati del dispositivo, allora aspetterà finché non verranno aggiornati, ma poiché li va a prendere da una cache che non viene aggiornata rimarrà in un loop infinito.

Per evitare il problema dobbiamo disattivare la cache per alcuni indirizzi e capire anche quali pagine sono destinate ai dispositivi hardware. (**Maggiore complessità**).

Quindi avendo un solo spazio di indirizzi, tutti i moduli in memoria e tutti i dispositivi I/O devono esaminare tutti i riferimenti alla memoria per capire a quali rispondere.

Problema

I computer moderni hanno un bus ad alta velocità tra CPU e memoria che migliora le prestazioni di quest'ultima.

Il problema è che i dispositivi I/O non possono vedere gli indirizzi di memoria che vanno sul bus e quindi non possono rispondere.

Soluzioni

1. Inviemo alla memoria **tutti i riferimenti**, se questa non risponde li inviamo sugli altri bus, è semplice da implementare, ma inefficiente.
2. Mettiamo sul bus della memoria una **spia** che intercetta gli indirizzi che potrebbero interessare dei dispositivi di I/O, il problema è che alcuni dispositivi potrebbero non essere in grado di elaborare le richieste alla velocità della memoria.
3. Usiamo un **controller della memoria** che filtra gli indirizzi per decidere se vanno alla memoria o ai dispositivi di I/O, potrebbero essere complicata da implementare in dispositivi legacy.

In conclusione I/O mappato in memoria richiede un bilanciamento tra:

- **Prestazioni** -> Minimizzare i ritardi negli accessi a memoria e dispositivi
- **Complessità** -> Mantenere un design efficiente senza overhead.

Soluzione ibrida

Combina i due tipi di approcci visti:

- **PMIO** --> Utilizza indirizzamento separato se il dispositivo ha istruzioni dedicate.
- **MMIO** --> I registri dei dispositivi sono mappati nello spazio di memoria.

La configurazione iniziale avviene tramite PMIO e usa l'indirizzo delle porte, mentre l'accesso ai dati avviene tramite MMIO.

Vantaggi

1. Flessibilità.
2. Ottimizzazione delle istruzioni.
3. Compatibilità legacy.
4. Separazione logica.

Svantaggi

1. Aumento della complessità.
2. Overhead iniziale.
3. Limitazioni architetturali.

DMA

La CPU deve **accedere** ai controller dei dispositivi per scambiare dati. Potrebbe richiedere i dati un byte alla volta, ma sarebbe troppo lento, per questo motivo usiamo una schema chiamata **DMA**.

Il SO può usare il DMA solo se l'hardware ha un controller DMA, questo può gestire trasferimenti a più dispositivi ed è spesso situato sulla scheda madre.

Il controller DMA ha accesso al bus di sistema indipendentemente dalla CPU, contiene all'interno tre registri che possono essere letti e scritti dalla CPU:

- **Registro degli indirizzi di memoria**
- **Registro dei conteggi dei byte**
- **Uno o più registri di controllo**

Registri di controllo del DMA

Specificano:

- Le **porte I/O** da usare.
- La **direzione** del trasferimento.
- **unità** di trasferimento.
- Numero di byte da trasferire alla volta.

Consideriamo la lettura dei dati sul disco per capire come funziona il DMA:

Senza DMA

1. Controller del disco legge serialmente il blocco, **memorizzandolo** nel buffer interno del controller.
2. Calcola la **somma di controllo**.
3. Provoca un **interrupt**.
4. Il SO legge il blocco del disco dal buffer del controller **salvandolo in memoria**.

Il funzionamento con il DMA è diverso, per prima cosa la **CPU programma il controller DMA impostandone i registri**. Poi dice al controller del disco di leggere i dati dal suo disco ed eseguire la **checksum**, quando ci sono dei dati validi nel buffer, allora il DMA può partire.

Il DMA invia sul bus una **richiesta di lettura** al controller del disco, in maniera "*anonima*".
Avviene poi la **scrittura in memoria** da parte del controller del disco, finita la scrittura il controller del disco invia un segnale di conferma al controller DMA tramite bus.
Quindi il DMA:

- Incrementa l'indirizzo di memoria da usare.
- Decrementa il conteggio dei byte.

Se il conteggio è maggiore di 0 Il DMA manda una **nuova richiesta di lettura e ripete le azioni**.
Altrimenti manda un **interrupt alla CPU** per avvisarla che il trasferimento è completo.
Il SO non deve copiare il blocco del disco perché è stato già trasferito.

Esistono diversi tipi di DMA, alcuni **semplici** (Un trasferimento alla volta), altri **complessi** (più trasferimenti) con più set di registri uno per canale, ogni canale è fatto per trasferimenti specifici e poter gestire diversi controller di dispositivi.

Quando parliamo di trasferimenti multipli il modo in cui il controller del DMA sceglie il dispositivo successivo è tramite algoritmi come il round-robin o prioritari.

Modi di operazione di un bus

- **Una parola alla volta** --> DMA trasferisce una parola alla volta, "**rubando**" dei cicli alla CPU rallentandola leggermente (**cycle stealing**).
- **A blocco** --> DMA comunica al dispositivo di **acquisire il bus**, avviare una serie di trasferimenti e rilasciare il bus. (**modalità burst**). Efficiente, ma può bloccare la CPU per lunghi periodi.

I controller DMA possono operare in entrambi i modi.

Esiste anche un terzo modello:

Fly-by Mode

Il DMA trasferisce dati direttamente alla memoria principale **senza intermediari**.
Altrimenti un altro metodo è quello di **inviare dal dispositivo al DMA** che poi fa una **richiesta di bus** per scrivere la parola dove necessario.

Questo tipo di schema richiede un **ciclo di bus extra** perché il DMA deve gestire sia il dispositivo sorgente che la memoria di destinazione.

La maggior parte dei controller DMA usa per i trasferimenti **indirizzi di memoria fisici**;
Quindi il SO converte l'indirizzo virtuale del buffer di memoria in un indirizzo fisico e poi scrive

l'indirizzo fisico nel registro degli indirizzi del controller DMA.

❗ Perché alcune periferiche hanno un buffer interno

Viene solitamente **usato per verificare i dati (checksum)** e gestire l'afflusso costante di bit dal disco. Questo approccio semplifica il design dei controller evitando problemi di temporizzazione e rischi di buffer overrun.

Ancora sugli interrupt

✍ Piccolo recap sui tipi di interruzioni

- **Interrupt hardware** --> Dispositivo invia un segnale alla CPU.
- **trap** --> Azione deliberata da parte del codice (Trap nel kernel per chiamata di sistema).
- **eccezioni o eccezione** --> Simile alla trap, ma non deliberato.

Il modo in cui vengono gestiti è simile, ma sono **causati da motivi diversi**.

Gli interrupt funzionano in questo modo:

1. I/O finisce il lavoro che gli è stato assegnato e **causa un interrupt**.
2. Il segnale è **rilevato dal chip** del controller degli interrupt.
3. Il controller invia l'interrupt che viene **riconosciuto dalla CPU**.

Se non ci sono interrupt allora viene gestito immediatamente, altrimenti viene gestito quello a **maggior priorità**. Il dispositivo ignorato continua ad inviare interrupt finché non viene preso in considerazione. Il controller assegna un numero alle linee degli indirizzi.

L'interrupt viene chiamato e la **CPU si ferma**, il numero sulle linee degli indirizzi è usato come indice in una tabella (**vettore degli interrupt**) e serve a prelevare un nuovo program counter. Questo PC punta all'inizio della **corrispondente procedura di servizio** degli interrupt. La posizione può essere fissa nella macchina o da qualche parte in memoria.

Dopo il suo avvio, la **procedura del servizio di interrupt conferma l'interrupt** (Scrivendo un certo valore su una delle porte di I/O del controller degli interrupt). Questo ci permette di **evitare delle race condition** tra interrupt simultanei.

Prima di avviare la procedura di servizio **dobbiamo salvare alcune informazioni**, come minimo il PC, mettendolo nei registri interni che il SO può leggere, porta a lunghi tempi di attesa.

Per questo motivo solitamente queste informazioni vengono salvate sullo stack corrente, il problema è che potremmo avere dei problemi con il puntatore dello stack (magari non è lecito) il che potrebbe portare ad avere degli **errori di pagina**.

Altrimenti usiamo lo **stack del kernel**, il che ci rende più sicuri da errori di pagina, ma potrebbe richiedere il **cambio di contesto della MMU e l'invalidazione di molte cache e del TLB**, **aumentando anche overhead**.

Interrupt precisi e imprecisi

Le moderne CPU utilizzano delle architetture pipeline superscalari il che complica la gestione degli interrupt.

Esempio su Sistemi vecchi

1. **Completamento** esecuzione di un'istruzione.
2. Microprogramma o hardware **controlla** se ci sta in interrupt.
3. Se sì, allora PSW e PC vengono spinti sullo stack, cominciava la **gestione** dell'interrupt.
4. Dopo la gestione PSW e PC vengono **prelevati** dallo stack.

Nel modello vecchio quando partiva un interrupt si era **sicuri che tutte le istruzioni avvenute prima dell'interrupt fossero eseguite completamente**.

Nei sistemi a pipeline questo **non avviene**, la pipeline contiene diverse istruzioni in un diverso stato di esecuzione.

Nelle macchine superscalari è anche **peggio**, perchè le istruzioni possono essere scomposte in micro-operazioni svolte in maniera sparsa.

Warning

Abbiamo delle istruzioni che si trovano a diversi stati di avanzamento.

Soluzione

Teniamo in dei **buffer i risultati di ciascuna istruzione** fino a quando le istruzioni precedenti non saranno completate, poi le eseguirà in ordine corretto.

In pratica:

- Ogni volta che un'istruzione viene emessa, ma non ancora completata, il processore **la mette in un buffer**.
- Quando l'interrupt finisce e il processore può riprendere, **completa le istruzioni in sospenso** (quelle che erano nel buffer) nell'ordine corretto.

Interrupt preciso --> Lascia la macchina in uno **stato definito**, ha quattro proprietà:

- Il PC è salvato in un luogo noto.
- Tutte le istruzioni prima del PC sono state completate.
- Nessuna istruzione dopo il PC è stata eseguita.
- Lo stato di esecuzione dell'istruzione alla quale punta il PC è noto.

Nella gestione degli interrupt precisi la CPU **cancella tutti gli effetti** di eventuali istruzioni transitorie eseguite dopo il PC.

La CPU continua da **dove era arrivata**, ma si assicura che tutte le operazioni incomplete vengano "annullate" prima di gestire l'interrupt.

Questo approccio è usato da architetture come x86 per garantire **compatibilità e prevedibilità**.

Interrupt impreciso --> Non rispetta i requisiti precedenti.

Le varie istruzioni al momento dell'interrupt si trovano in **stati diversi** di esecuzione.

Per gestirli le macchine **vomitano una grande quantità di stato interno sullo stack**, ciò rende il SO complesso e lento, inoltre il salvataggio di molte informazioni rallenta il processo di interrupt e ripristino.

Breve analisi degli interrupt

Interrupt precisi --> Logica di interrupt **complessa**, perchè dobbiamo assicurarci che ogni istruzione possa essere terminata senza cambiare lo stato della macchina. (Costoso in termini di **spazio del chip** e complessità progettuale).

Interrupt imprecisi --> Rendono il sistema operativo **molto più complicato, lento e meno sicuro a causa della complessità**.