Livelli del software dell'Input e Output

Il software dell'I/O è organizzato in quattro livelli, ciascuni dei quali ha un'interfaccia ben definita verso i licelli adiacenti. I quattro livelli sono, partendo dal basso:

- **Gestore degli interrupt**
- **Driver del dispositivo**
- **Software del sistema operativo indipendente dal dispositivo**
- **Software per L'I/O a livello utente**

Gestori degli interrupt

Per la maggior parte dei dispositivi di I/O gli interrupt sono un male necessario e solo una parte del SO dovrebbere esserne a conoscenza.
Per nasconderli possiamo fare che il driver che ha iniziato l'operazione di I/O si blocchi fino al termine dell'operazione e al verificarsi dell'interrupt.

```
Interrupt --> Procedura lo gestisce --> Sblocco del driver.
```

Questo modello funziona se i **driver sono strutturati come processi**, con stati, stack e contatori di programma propri.

Di seguito vengono descritti una serie di passaggi da eseguire sul software dopo il completamento dell'interrupt hardware:

> ⚠ **Questi passaggi possono essere diversi su macchine diverse.**

1. Salvataggio dei registri non ancora salvati dell'interrupt hardware.
2. Impostazione di un contesto e di uno stack per la procedura di servizio dell'interrupt.
3. Conferma al controller degli interrupt, se manca il controller centralizzato degli interrupt, riabilitiamo tutti gli interrupt.
4. Copia dei registri da stack alla tabella dei processi.
5. Esecuzione procedura di servizio dell'interrupt.
6. Scelta di quale processo eseguire come successivo.
7. Impostazione del contesto della MMU e del TLB per il processo successivo da eseguire.
8. Caricamento dei nuovi registri del processo.
9. Avvio dell'esecuzione del nuovo processo.

Elaborazione di un interrupt --> Tante istruzioni della CPU, specialmente su macchine con memoria virtuale dove bisogna impostare tabelle delle pagine, MMU, TLB, cache della CPU quanda si passa da kernel a modalità utente.

Driver di dispositivo

Esistono diversi dispositivi di I/O e ognuno di questi necessita di un certo codice specifico, cioè il **driver di dispositivo**.
Questo è scritto dal produttore del dispositivo che deve scrivere il codice per ogni sistema operativo, poiché ognuno di questi necessita dei propri driver.

Ogni driver gestisce un dispositivo o una classe di dispositivi, ma non ci sono restrizioni sul fatto che un driver possa controllare dure tipi di dispositivi completamente diversi tra loro.
Driver diversi possono, però essere basati sulla stessa tecnologia.

> **USB (Universal Serial Bus)**
>
> Dispositivi della categoria USB come dischi, chiavette, fotocamere, termometri, mouse, astiere, disco ball usano tutti **driver USB**.
> Il trucco sta nel disporre i dispositivi su uno stack (Come quello del TCP/IP) partendo dalla base abbiamo:
>
> - **Livello di collegamento** -> Gestisce questioni hardware (Segnalazione e decodifica dei flussi di segnali in pacchetti USB).
> - **Livelli superiori** -> Usano il livello di collegamento e sono comuni a tutti i dispositivi USB.
> - **API** -> Interfaccie per i dispositivi specifici.

Per accedere all'hardware del dispositivo, il driver del dispositivo deve essere parte del kernel del SO.

> **Nota**
>
> E' possibile creare driver che funzionano nello spazio utente, ma la maggior parte delle architetture non lavora in questo modo.

Il modello sopra descritto, per quanto non venga usato, elimina un grande problema di crash nel sistema: **Driver difettosi che interferiscono con il kernel.**

L'architettura del SO deve permettere installazione di codice scritti da terzi, questo comporta un modello ben definito di quello che fa un driver e di come interagisce con il resto del sistema. Posizioniamo i driver sotto il resto del SO, il SO poi li classifica in:

- **Dispositivi a blocchi** --> Contenenti molteplici blocchi di dati indirizzabili e indipendenti.
- **Dispositivi a caratteri** --> Generano un flusso di caratteri.

Maggior parte dei SO definisce un'interfaccia standard supportata da tutti dispositivi a blocchi e una per i dispositivi a caratteri, queste interfaccie sono **procedure** che il SO può chiamare affinchè il driver faccia il suo lavoro.

Procedure tipiche

Lettura di un blocco -> Dispositivi a blocchi.
Scrittura di una stringa di caratteri --> Dispositivi a caratteri.

In alcuni sistemi il SO è un singolo binario contenente al suo interno tutti i driver compilati che gli servono, nel momento in cui bisogna aggiungere un nuovo dispositivo bisogna ricompilare il kernel con il nuovo driver.

Questo modello smise di funzionare con l'avvento del personal computer quindi, a partire dal MS-DOS si optò per il caricamento dinamico dei driver nel sistema durante l'esecuzione.

Funzioni di un driver di dispositivo

- Accettare richieste di lettura e scrittura astratte provenienti dal soptarstante software indpendente dal dispositivo e portarle a termine.
- Inizializzare il dispositivo.
- Gestire i requisiti di alimentazione.
- Gestire il registro degli eventi.

Un driver classico nello svolgere la prima funzione:

- Verifica se i parametri di input sono validi o meno, se non lo sono restituisce un errore.
- Se sono validi dobbiamo **tradurre dall'astratto al concreto.**
- Verificare se il dispositivo è in uso, se lo è, la richiesta andrà in coda.
- Se il dispositivo è inattivo, viene **esaminato lo stato dell'hardware** per vedere se la richiesta può essere gestita immediatamente.

Una volta che il dispositivo è accesso e pronto a partire può iniziare il controllo.

Controllare significa inviare una serie di comandi, questa serie di comandi si scrive nel driver.
Una volta che il driver sa quali comandi inviare, inizia a scriverli nei registri del controller.
Dopo la scrittura di ciascun comando nel controller, è necessario capire se il controller ha accettato il comando e se è pronto per quello succesivo, questa sequenza continua finché non sono stati inviati tutti i comandi.

> ⓘ **Ad alcuni controller**
>
> Possiamo passare una lista concatenata di comandi con l'ordine di leggerli ed elaborarli tutti autonomamente, senza alcun aiuto del SO.

Dopo l'invio di ciscun comando si possono verificare due possibili situazioni:

- Il driver del dispositivo deve aspettare che il controller esegua un pò di lavoro, quindi si blocca finché non arriva un interrupt a sbloccarlo.
- L'operazione finisce senza ritardo e quindi il driver non ha bisogno di bloccarsi.

In entrambi i casi il driver, al completamento delle operazioni, deve controllare eventuali errori.
Se è tutto a posto, il driver potrebbe avere dei **dati da passare al software** indipendente dal dispositivo.
Alla fine restituirà informazioni di stato per comunicare errori al chiamante e se ha richieste in coda potrà selezionarne una nuova, in caso contrario il driver si blocca.

Questo modello descritto è un'approssimazione della realtà, ci sono diversi fattori che rendo l'esecuzione del codice del driver molto complicato.

> ⚠ **Esempio**
>
> Dispositivo di I/O termina, mentre il driver è in esecuzione, quindi il dispositivo invia un interrupt che potrebbe causare l'esecuzione di un driver di dispositivo.

Per questo motivo i driver sono **rientranti**, cioè che i driver in esecuzione devono aspettarsi di essere chiamati una seconda volta prima di aver completato la prima chiamata.

Altre situazioni sconvenienti avvengono nei sistemi **a caldo** dove può aggiungere o rimuovere dispositivi a computer accesso funzionante.

> ⚠ **Sistemi a caldo**
>
> Se un driver che sta leggendo da un dispositivo viene informato dal sistema che il dispositivo è stato rimosso allora:

- Il trasferimento di I/O deve essere annullato.
- Bisogna eliminare qualsiasi altra richiesta fatta dal dispositivo svanito.

Inoltre l'aggiunta inaspettata di altri dispositivi obbliga il kernel a fare il giocoliere con le risorse (Es: le linee IRQ).

Ai driver non sono consentite chiamate di sistema, ma devono potrr interagire con il resto del kernel, quindi sono permesse chiamate a determinate procedure del kernel.

Software per l'I/O indipendete dal dispositivo

Parte del software per l'I/O è specifico di un dispositivo, ma esistono anche parti indipendenti dal dispositivo stesso. Il limite fra driver e software indipendente dipende dal sistema e dal dispositivo.

Funzioni tipiche eseguite dal software indipendente

- Interfacciamento uniforme dei driver dei dispositivi.
- Buffering.
- Segnalazione degli errori.
- Allocazione e rilascio dei dispositivi dedicati.
- Dimensione dei blocchi indipendente dal dispositivo.

La funzione del software indipendente dal dispositivo è quella di eseguire funzioni trasversali a tutti i dispositivi in modo da fornire un'interfaccia uniforme al software a livello utente.

Interfacciamento uniforme dei driver dei dispositivi:

E' di fondamentale importanza rendere tutti i dispositivi e i driver simili tra loro, se ogni dispositivo fosse diverso, allora dovremmo modificare il SO ogni volta.

Se ogni driver avesse un'interfaccia diversa verso il SO avremmo bisogno di fare uno sforzo gigantesco dal punto di vista della programmazione. Se ogni driver avesse un'interfaccia comune allora non ci sarebbero troppi problemi nell'aggiugere un nuovo dispositivo.

Per ogni classe di dispositivo, il SO definisce un insieme di funzioni che il driver deve supportare, il driver contiene una tabella con **puntatori a queste funzioni.**
Una volta caricato il driver, il sistema operativo registra l'indirizzo di questa tabella di puntatori a funzioni, così quando ha bisogno di chiamarne una può fare una chiamata indiretta tramite

questa tabella. Questa tabella di puntatori a funzioni definisce **l'interfaccia fra il driver e il resto del sistema operativo.**

Altro aspetto dell'uniformità dell'interfaccia riguarda la denominazione dei dispositivi di I/O.
Il software indipendente dal dispositivo ha il compito di mappare i nomi simbolici dei dispositivi ai driver corrispondenti.

nome di dispositivo ha un I-node che contiene:

Major device number --> Numero del dispositivo primario che localizza il driver.
Minor device number --> Numero del dispositivo secondario da dove leggere.

I dispositivi, inoltre appaiono nel file system come oggetti denominati, quindi le abituali regole di protezione dei file valgono anche per i dispositivi.

Buffering:

Anche il buffering costituisce un problema, consideriamo un processo che legge dati da un modem.

Consideriamo l'esempio di un processo utente che vuole leggere dati da un modem

Prima strategia senza buffer

Il processo utente fa una chiamata di sistema read per leggere il carattere, i caratteri in arrivo causano un interrupt, la procedura di interrut sblocca il processo utente e gli passa il carattere, infine il processo utente salva il carattere e legge un nuovo carattere e si blocca di nuovo.

⚠ Problema

Inefficiente poiché richiede il riavvio del processo utente per ogni carattere ricevuto.

Buffer del processo utente

Il processo utente ha un buffer di n caratteri nello spazio utente, da una read di n caratteri, la procedura di servizio degli interrupt mette i caratteri nel buffer finché non è pieno, poi risveglia il processo.

⚠ **E' più efficiente, MA**

Se il buffer è paginato in uscita all'arrivo di un carattere, allora ci sta un calo delle prestazioni.

Buffer all'interno del kernel

Il gestore degli interrupt vi inserisce caratteri, quando è pieno la pagina con il buffer utente è copiata in ingresso.

⚠ **Efficiente, MA**

Cosa succede ai caratteri che arrivano mentra la pagina con il buffer utente viene letta dal disco? Il buffer è pieno è non può ricevere altri caratteri.
Per questo usiamo un secondo buffer nel kernel, questo schema si chiama **buffering doppio.**

Un'altra forma di buffering è detta **buffer circolare** composto da una zona di memoria e due puntatori:

- Il primo punto alla parola libera successiva, dove mettere i nuovi dati.
- Il secondo punta alla prima parola dei dati nel buffer che non è stata ancora rimossa.

L'hardware fa avanzare il primo ogni volta che riempe la memoria, il SO fa avanzare il secondo man mano che rimuove dati, i due puntatori faranno il giro tornando in fondo quando hanno raggiunto la cima.

L'uso di buffer è importante anche in output, prendiamo in esempio il secondo modello:
Quando l'utente ha fatto la richiesta dei caratteri il sistema ha due possibilità:

- Bloccare l'utente finché non sono scritti tutti i caratteri -> Richiede molto tempo.
- Rilasciare utente e gestire I/O mentre l'utente fa qualcos'altro -> Ma come fa l'utente a sapere quando l'output è pronto?

Possiamo copiare i dati in un buffer kernel e sloccare immediatamente il chiamante. Non importa quando I/O effettivo è stato completato; l'utente è libero di usare il buffer nel momento in cui è sbloccato.

Il buffering è ampiamente usato, ma se i dati vengono messi troppo volte nel buffer, le prestazioni ne risentono, poiché le continue copie richieste per il trasferimento rallentano la velocità di trasmissione.

Quindi la sequenzialità delle operazioni di copia aumenta il tempo di trasmissione.

Segnalazione degli errori:

Gli errori sono comuni in ambito dell'I/O, molti sono specifici del dispositivo e devono esser gestiti tramite driver apposito, ma la struttura per la gestione è indipendente dal dispositivo.

Esistono diverse classi di errori:

- **Errori di programmazione** --> Il pr0cesso richiede qualcosa di impossibile (Lettura/Scrittura/Fornire parametri non validi/Specificare dispositivo sbagliato). In questo caso dobbiamo solo mandare un codice d'errore al chiamante.
- **Errori di I/O** -->Tentativo di scrittura su un blocco danneggiato, in questo caso il driver decide cosa fare, se non lo sa, passa il problema al software indipendente dal dispositivo.

Ciò che fa il software **dipende dalla natura dell'errore**, se ci sta un utente interattivo attivo può mostrare una finestra per chiedere all'utente cosa fare.

> **Alcune opzioni**
>
> Riprovare, ignorare l'errore, terminare il processo chiamante.

Se non ci sta un utente allora la chiamata di sistema fallisce e riporta un codice di errore.
Alcuni errori non possono essere gesetiti in questo modo, come quelli che richiedeono la distruzione di una struttura dati critica, in questo caso ci sta solo un messaggio di errore.

Allocazione e rilascio dei dispositivi dedicati:

Alcuni dispositivi possono essere usati da un solo utente in un determinato momento, il SO esamina le richieste e decide di accetarle o rifiutarle.

Come gestire le richieste --> richiedere ai processi di fare la open direttamente su file speciali per i dispositivi. La chiusura di tali dispositivi dedicati rilascia i file.

Un approccio alternativo è quello di avere meccanismi speciali per richiedere e rilasciare i dispositivi dedicati, i tentativi di acquisizione rifiutati bloccano il processo chiamante.
I processi bloccati sono messi in una **coda** da cui verrano ripresi quando il dispositivo sarà libero.

Software di I/O nello spazio utente

La maggior parte del software I/O risiede nel SO, ma ci sta una piccola parte composta da librerie collegate con i programmi utente e programmi eseguiti al di fuori del kernel.

Le chiamate di sistema sono procedure di libreria:

```
count = write(fd, buffer, nbytes)
```

La procedura di libreria write sarà collegara con il programma e contenuta nel programma binario. La raccolta delle procedure di librerie fa parte dell'I/O.

Molte di queste procedure non fanno altro che mettere i parametri nel posto opportuno per la chiamata di sistema, ma ci sono alcune di queste che servono per la formattazione dell'input e dell'output.

Esempio con printf:

```
printf("Il quadrato di %3d è %6d\n", i, i*i);
```

printf --> Accetta come input una stringa di formato e alcune variabili, costruisce una stringa ASCII e chiama write sull'output della stringa.

Esempio simile è quello con *scanf.*

Un'altra categoria importante, oltre alle procedure di librerie, è quello dei **sistemi di spooling.**

> ⓘ **Spooling**
>
> Modo per interagire con i dispositivi dedicati in un sistema multiprogrammato.

Esempio con stampante:

La stampante è un dispositivo che usa lo spooling, è semplice permettere a qualunque processo di aprire il file speciale a caratteri per la stampante, ma qualcuno potrebbe bloccarla per ore senza usarla, nessun altro processo avrebbe diritto di accesso.

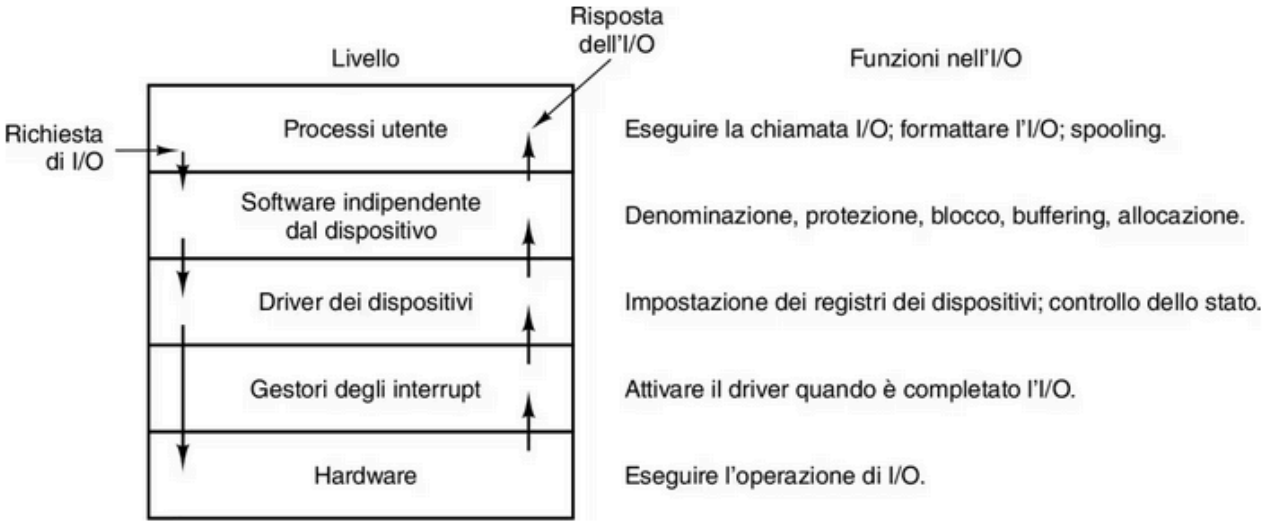
Vengono creati quindi un processo **deamon** e una **directory di spooling**.

Per stampare il processo genera il file sulla directory di spooling.

Il compito del **deamon** è quello di essere l'unico processo a poter aprire il file speciale della stampante che dovrà stampare ciò che sta nella directory.

Esempio del funzionamento di tutti i livelli

Figura 5.17



Livelli del sistema di I/O e principali funzioni di ogni livello.

Le frecce nella Figura 5.17 mostrano il flusso del controllo. Quando per esempio un programma utente prova a leggere un blocco da un file, il sistema operativo viene invocato per realizzare la chiamata. Il software indipendente dal dispositivo cerca il blocco, ad esempio, nella cache del buffer. Se il blocco non è lì, chiama il driver del dispositivo per inviare la richiesta all'hardware di prelevarlo dall'SSD o dal disco. Il processo viene poi bloccato sino al completamento dell'operazione, che può durare millisecondi (un tempo troppo lungo per lasciare inattiva la CPU).

Quando l'SSD o il disco ha finito, l'hardware genera un interrupt. Il gestore degli interrupt viene eseguito per scoprire che cos'è accaduto, ossia quale dispositivo richiede attenzione immediata. Estrae quindi lo stato dal dispositivo e risveglia il processo dormiente per concludere la richiesta di I/O e permettere al processo utente di proseguire.