Principi del software di Input e Output

Concetto fondamentale --> **Indipendenza dal dispositivo.**

> ⚠ **Indipendenza dal dispositivo**
>
> Dobbiamo poter scrivere programmi in grado di accedere a qualsisasi dispositivo I/O senza dover specificare in anticipo il dispositivo.

Dovremmo poter digitare il seguente comando:

```
sort <input >output
```

Un comando che funzioni a prescindere dal dispositivo di input e output. E' compito del SO gestire i dispositivo usati nel comando.

Un secondo obiettivo è quello della **denominazione uniforme.**

> ⚠ **Denominazione uniforme**
>
> Il nome di un file o di un dispositivo non dovrebbe essere in alcun modo dipendente dal dispositivo.

> ⓘ **In UNIX usiamo dei nomi di percorso**
>
> Montando la penna USB questa viene aggiunta al percorso.

Altra cosa importante è la **gestione degli errori**, questi dovrebbero essere gestiti il più vicino all'hardware.
Se il controller scopre un errore, allora dovrebbe essere lui a correggerlo, altrimenti dovrebbe farlo il driver del dispositivo.
Molti errori sono **transitori** --> Scompaiono al ripetersi dell'operazione.
Quando i livelli inferiori non riescono a gestire il problema possiamo avvisare i livelli superiori.

Altra questione è quella dei **trasferimenti asincroni rispetto a quelli sincroni.**

> ⚠ **Warning**

La maggior parte dell'I/O è asincrono, la CPU si dedica ad altro finché non arriva un interrupt.

Programmi utente --> I/O bloccanti, dopo la chiamata read il programma è sospeso finchè i dati non sono disponibili nel buffer.
I sistemi operativi cercano di rendere l'I/O asincrono "trasparente" ai programmi, facendogli sembrare che siano bloccanti anche quando, in realtà, funzionano in modo asincrono dietro le quinte.

Altro problema --> **bufferizzazione**

⚠ **Bufferizzazione**

I dati che escono da un dispositivo non possono essere memorizzati nella destinazione finale.

Quindi i dati devono essere inseriti momentaneamente in un buffer, il SO può esaminarli, se si tratta di dati con vincoli Real-TIme possono essere processati, inoltre dobbiamo evitare un **buffer underrun.** (Mancanza di dati nel buffer durante la lettura).
L'uso dei buffer ha un impatto pesante sulle prestazioni di I/O.

L'ultimo problema di cui dobbiamo parlare è la **condivisione di dispositivi**

Alcuni dispositivi possono essere condivisi senza problemi (Es: Dischi).
Alcuni dispositivi NON possono essere condivisi senza problemi (Es: Stampanti).

Quindi dobbiamo riuscire a gestire i dispositivi tenendo conto anche la condivisione.

I/O programmato

Esistono tre metodi principali per eseguire l'I/O, il primo è quello programmato.
Consiste nel delegare tutto il lavoro alla CPU.

🖊 **Esempio di I/O programmato**

Abbiamo un processo utente che vuole scrivere "ABCDEFGH" sulla stampante attraverso un interfaccia seriale:

1. Il software assembla la stringa in un buffer dell spazio utente.
2. Il processo utente, tramite chiamata di sistema, acquisisce la stampante per la scrittura.

Se è occupata allora la chiamata fallirà restituendo un codice di errore o si bloccherà finché la stampante non sarà libera.

3. Ottenuta la stampante, il processo esegue una chiamata di sistema richiendendo al SO di stampare la stringa.
4. Il SO copia il buffer con la stringa nello **spazio kernel**, per accedervi più facilmente.
5. Controlla se la stampante è disponibile, se non lo è aspetta.
6. Il SO copia il primo carattere nel registro dei dati della stampante.
7. Dopo aver copiato il primo carattere, il SO controlla se la stampante è pronta per quello successivo (La stampante di solito ha un secondo registro di stato).
8. Quando è pronta ricomincia da capo la sequenza.

Il SO entra in uno stato di **polling** o **busy waiting**, cioè controlla ogni volta lo stato della stampante e poi invia un carattere.

```
copy_from_user(buffer, p, count); /* p è il buffer del kernel */
for (i = 0; i < count; i++) { /* ripeti per tutti i caratteri */
        while (*printer_status_reg != READY) ; /* ripeti finché lo stato
diventa
        READY */ *printer_data_register = p[i]; /* fai l'output di un
carattere */ } return_to_user();
```

I/O programmato è molto semplice, ma occupa la CPU a tempo pieno finché tutto l'I/O non è terminato. Busy waiting è efficiente solo per tempi brevi.
Estremamente adatto a **sistemi embedded** dove la CPU non ha altre attività da scolgere.

I/O guidato dagli interrupt

Consideriamo il caso di stampa su stampante che non ha buffer, ma stampa caratteri appena arrivano.

Esempio

Se una stampante stampa 100 caratteri al secondo, ogni carattere impiega 10ms per essere stampato.
Quindi dopo ogni scrittura la stampante si fermerà per 10ms per consentire l'output successivo.

Questo è abbastanza tempo per fare un cambio di contesto e eseguire altri processi.
Per permettere alla CPU di fare altro usiamo gli interrupt.

Funzionamento

1. Chiamata di sistema.
2. Buffer copiato nel kernel.
3. Primo carattere copiato.
4. CPU chiama lo scheduler.
5. Processo che ha richiesto la stampa è bloccato finché non è stampata l'intera stringa.

```
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler();

if (count == 0) {
  unblock_user();
} else {
  *printer_data_register = p[i];
  count = count - 1;
  i = i + 1;
}
  acknowledge_interrupt();
  return_from_interrupt();
```

Prima parte --> Codice eseguito al momento della chiamata di sistema per la stampa.
Seconda parte --> Procedura di servizio interrupt per la stampante.

Quando la stampante ha stampato il primo carattere genera un interrupt che ferma il processo attuale e ne salva lo stato.
Viene eseguita la procedura di servizio di interrupt della stampante, se non ci sono caratteri da stampare **sblocca l'utente**, altrimenti esegue **l'output del carattere successivo**, riconosce l'interrupt e ritorna al processo che stava eseguendo.

I/O con DMA

Svantaggio dell'I/O guidato dagli interrupt -> Interrupt per ogni carattere.

Una soluzione è l'uso del DMA, l'idea è che il controller DMA invii i caratteri alla stampante uno alla volta, senza disturbare la CPU.

I/O con DMA è uguale a quello programmato con la differenza che il lavoro lo fa il controller DMA e non la CPU.

```
copy_from_user(buffer, p , count);
set_up_DMA_controller();
sceduler();

acknoledge_interrupt();
unblock_user();
return_from_interrupt();
```

Prima parte -> Codice eseguito al momento della chiamata di sistema per la stampa.
Seconda parte -> Proedura di servizio degli interrupt.

Il grosso vantaggio è la riduzione di interrupt da uno per carattere ad uno per buffer.

> ⚠ **NOTA CHE**
>
> Se i caratteri sono molti e gli interrupt lenti --> Grande miglioramento.
> Ma il **controller DMA è più lento della CPU**, quindi il dispositivo non sarà al massimo della velocità.
> Se la CPU nel frattempo non ha nulla da fare potrebbe andare meglio un I/O programmato o con interrupt.