LIVELLI DEL SOFTWARE DI I/O

Danilo Croce

Gennaio 2025

OVERVIEW

- Principi dell'hardware di I/O
- **Livelli del software di I/O**

OBIETTIVI DEL SOFTWARE DI I/O

- **Indipendenza dal Dispositivo**:
  - Il software di I/O dovrebbe permettere l'accesso a diversi dispositivi senza specificare il tipo di dispositivo in anticipo.
  - **Esempio**: un programma che legge un file dovrebbe funzionare indifferentemente con dischi fissi, SSD o penne USB.
- **Denominazione Uniforme**:
  - I nomi di file o dispositivi dovrebbero essere stringhe o numeri indipendenti dal dispositivo.
  - **Esempio**: in UNIX, l'integrazione dei dispositivi nella gerarchia del file system consente un indirizzamento uniforme tramite nomi di percorso.
    - Non vogliamo digitare ST6NM04 per indirizzare il primo disco rigido.
      - `/dev/sda` **è meglio**
      - `/mnt/movies` **ancora meglio**

OBIETTIVI DEL SOFTWARE DI I/O (2)

- **Gestione degli Errori**:
  - Gli errori **vanno gestiti il più vicino possibile all'hardware**, idealmente dal controller stesso o dal driver del dispositivo.
  - **Errori transitori** (come quelli di lettura) spesso **scompaiono ripetendo l'operazione**.
- **Trasferimenti Sincroni vs Asincroni**:
  - La maggior parte dell'I/O fisico è asincrono, ma per semplicità, molti programmi utente trattano l'I/O come se fosse sincrono (bloccante).
  - Il sistema operativo rende operazioni asincrone sembranti bloccanti, ma fornisce anche l'accesso all'I/O asincrono per applicazioni ad alte prestazioni.
  - Il sistema operativo deve gestire DMA

OBIETTIVI DEL SOFTWARE DI I/O (3)

- **Buffering**:
  - Spesso i dati da un dispositivo non vanno direttamente alla destinazione finale, richiedendo un buffer temporaneo.
    - Un pacchetto che arriva su un'interfaccia di rete deve essere ricevuto e analizzato prima di capire quale applicazione (esempio browser) può usarlo
    - Un segnale audio deve essere posizionato preventivamente in un buffer per evitare interruzioni
  - L'uso di buffer può influenzare le prestazioni, soprattutto per dispositivi con vincoli real-time.
- **Dispositivi Condivisibili vs Dedicati**:
  - Dispositivi come dischi e SSD possono essere condivisi da più utenti, mentre altri come stampanti e scanner sono tipicamente dedicati.
  - Il sistema operativo deve gestire entrambe le categorie per evitare problemi come i deadlock.

TIPOLOGIE DI SW PER I/O

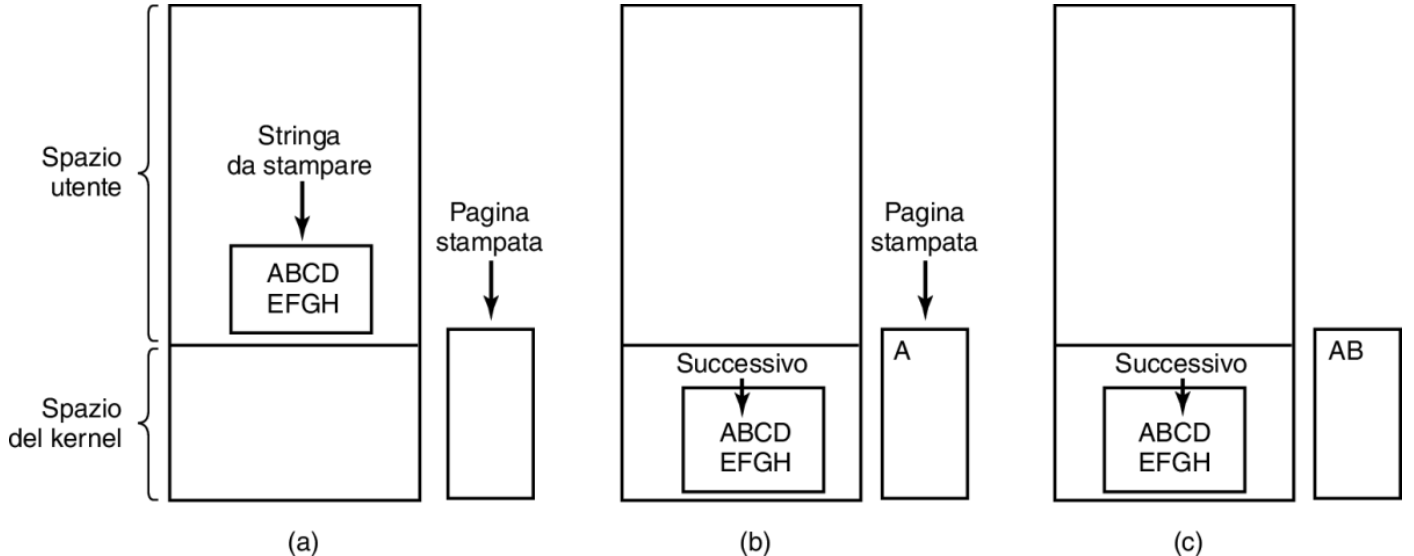
I/O Programmato

I/O Guidato dagli Interrupt

I/O con DMA

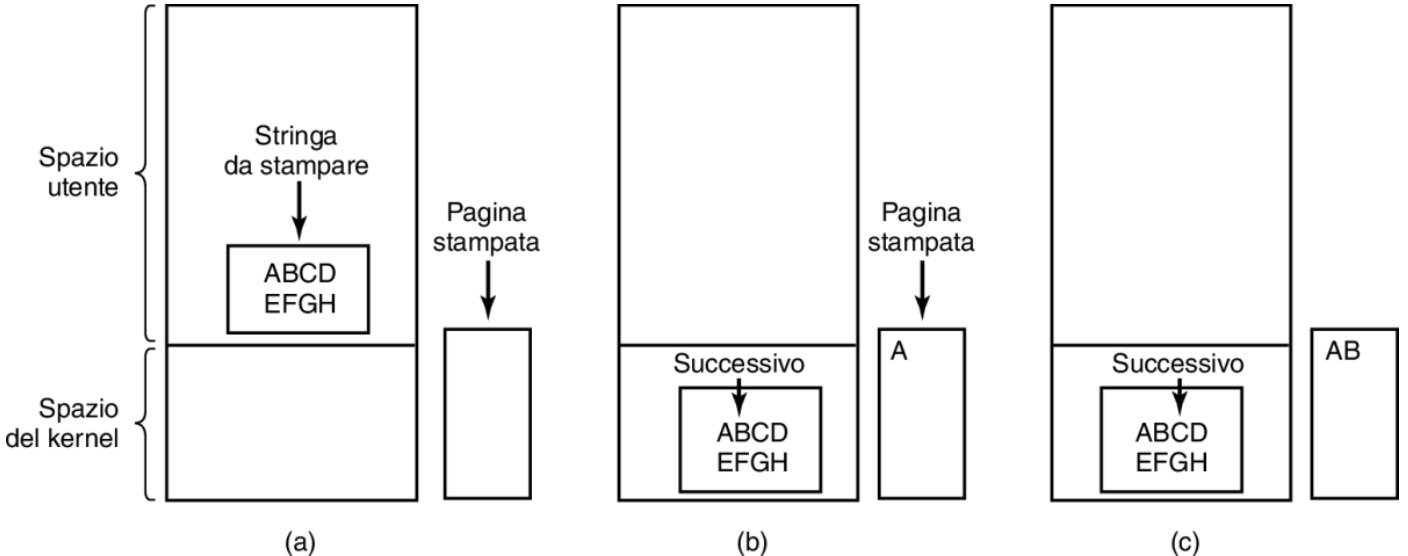
I/O PROGRAMMATO - PROCESSO E ESEMPIO PRATICO

- **Definizione di I/O Programmato**: la CPU gestisce direttamente tutto il processo di trasferimento dei dati.
- **Esempio Pratico con Riferimento**:
  - Un processo utente prepara una stringa "ABCDEFGH" in un buffer dello spazio utente.
  - Il processo effettua una chiamata di sistema per stampare la stringa, dopo aver ottenuto l'accesso alla stampante (a).



I/O PROGRAMMATO - PROCESSO E ESEMPIO PRATICO

- **Azione del Sistema Operativo**: copia il buffer in uno spazio del kernel.
  - Invia i caratteri alla stampante uno alla volta, aspettando che questa sia pronta per ogni carattere (b e c).
- **Polling o Busy Waiting**:
  - Il sistema operativo entra in un ciclo di polling, controllando il registro di stato della stampante e inviando un carattere alla volta.



ESEMPIO DI CODICE E LIMITI DELL'I/O PROGRAMMATO

```
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];               /* invia in output ogni carattere */
}
return_to_user();
```

- **Esempio di Codice per I/O programmato**:
  - Utilizza `copy_from_user(buffer, p, count)` per copiare i dati dal buffer utente a quello del kernel.
  - Un ciclo `for (i = 0; i < count; i++)` gestisce il trasferimento carattere per carattere alla stampante.
- **Svantaggi dell'I/O Programmato**:
  - **Occupa la CPU a tempo pieno durante il processo di I/O**, facendo continuamente polling sullo stato della stampante.
  - **Inefficiente** in sistemi complessi dove la CPU ha altre attività importanti da gestire.

ESEMPIO DI CODICE E LIMITI DELL'I/O PROGRAMMATO (2)

```
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];               /* invia in output ogni carattere */
}
return_to_user();
```

- **Applicazioni e Contesti Efficaci**:
  - L'I/O programmato è **efficace quando il tempo di elaborazione** di un carattere è **breve**.
  - Adatto a sistemi embedded dove la CPU non ha altre attività significative.
- **Necessità di Metodi di I/O Alternativi**:
  - Nei sistemi più complessi, il busy waiting diventa un approccio inefficiente.
  - Ricerca di metodi di I/O che liberino la CPU da costanti attività di polling.

I/O GUIDATO DAGLI INTERRUPT - PROCESSO DI STAMPA

- **Scenario di Stampa senza Buffer**:
  - Una stampante che stampa un carattere alla volta, con un ritardo di 10 ms per carattere,
  - permettendo alla CPU di eseguire altri processi durante l'attesa.
- **Utilizzo degli Interrupt**:
  - Dopo la chiamata di sistema `copy_from_user` per stampare una stringa, il buffer viene copiato nello spazio del kernel e il primo carattere viene inviato alla stampante.
  - La CPU poi passa l'esecuzione ad altri processi mentre attende che la stampante sia pronta per il carattere successivo.
- **Cambio di Contesto e Blocco del Processo**:
  - Il processo che ha richiesto la stampa viene bloccato fino a quando non è stampata l'intera stringa.
  - La CPU attiva lo scheduler per eseguire altri processi durante l'attesa.

```
/* copia dati dall'utente al kernel.*/
copy_from_user(buffer, p, count);

/*abilita gli interrupt*/
enable_interrupts();

/*attende che la stampante sia pronta
  a ricevere caratteri */
while (*printer_status_reg != READY) ;

/*invia il primo carattere alla stampante.*/
*printer_data_register = p[0];

/*passa il controllo a un altro processo.*/
scheduler();
```

Codice eseguito al momento della chiamata di sistema per la stampa.

I/O GUIDATO DAGLI INTERRUPT - PROCESSO DI STAMPA (2)

- **Generazione dell'Interrupt da Parte della Stampante**:
  - La stampante genera un interrupt quando è pronta per il carattere successivo, interrompendo il processo corrente e salvandone lo stato.
- **Esecuzione della Procedura di Servizio Interrupt**:
  - Viene eseguita la procedura di servizio di interrupt per la stampante.
  - Se ci sono altri caratteri da stampare, il gestore stampa il successivo.
- **Sblocco del Processo e Ritorno dall'Interrupt**:
  - Se tutti i caratteri sono stati stampati, il gestore degli interrupt esegue azioni per sbloccare il processo utente.
  - Riconosce l'interrupt e ritorna al processo interrotto, che riprende l'esecuzione da dove era stato lasciato.
- **Problema**: Interrupt ad ogni carattere!

```
/* Controlla se tutti i caratteri sono stati stampati. */
if (count == 0) {
    /* Sblocca il processo utente che ha richiesto la stampa */
    unblock_user();
} else {
    /* Invia il carattere successivo alla stampante. */
    *printer_data_register = p[i];
    /* Decrementa il conteggio dei caratteri rimanenti. */
    count = count - 1;
    /* Passa al carattere successivo nel buffer. */
    i = i + 1;
}
/* Riconosce l'interrupt ricevuto dalla stampante. */
acknowledge_interrupt();
/* Ritorna dall'interrupt, permettendo alla CPU di riprendere altre operazioni. */
return_from_interrupt();
```

Procedura di servizio interrupt per la stampante.

I/O CON DMA: EFFICIENZA E GESTIONE DEI PROCESSI

- **Principio del DMA**:
  - Il DMA riduce il numero di interrupt, passando da uno per ogni carattere a uno per buffer.
  - Libera la CPU per eseguire altre attività durante il trasferimento di I/O.
- **Setup e Inizio del Trasferimento (a)**:
  - **Preparazione dei Dati**: `copy_from_user`.
  - **Configurazione DMA**: `set_up_DMA_controller`.
  - **Ottimizzazione delle Risorse CPU**: `scheduler`.
- **Gestione dell'Interrupt e Conclusione (b)**:
  - **Gestione dell'interrupt generato** dal completamento del trasferimento DMA: `acknowledge_interrupt`.
  - **Ripresa del Processo Utente**: `unblock_user`.
  - **Ritorno dal Contesto dell'Interrupt**: `return_from_interrupt`.

```
/* Copia i dati dall'utente al kernel. */
copy_from_user(buffer, p, count);
/* Impostazione del controller DMA per il trasferimento */
set_up_DMA_controller();
/* La CPU esegue altri processi mentre il DMA gestisce il trasferimento. */
scheduler();
```

(a)

```
/* Riconosce l'interrupt ricevuto dal DMA. */
acknowledge_interrupt();
/* Sblocca il processo utente dopo che il trasferimento è completo. */
unblock_user();
/* Ritorna dall'interrupt, consentendo alla CPU di proseguire con altre operazioni. */
return_from_interrupt();
```

(b)