Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2018-2019

Pietro Frasca

Lezione 24

Martedì 15-01-2019

Allocazione dei dispositivi e tecniche di spooling

I dispositivi, secondo le loro caratteristiche, possono essere allocati ad un processo alla volta o essere condivisi tra più processi contemporaneamente.

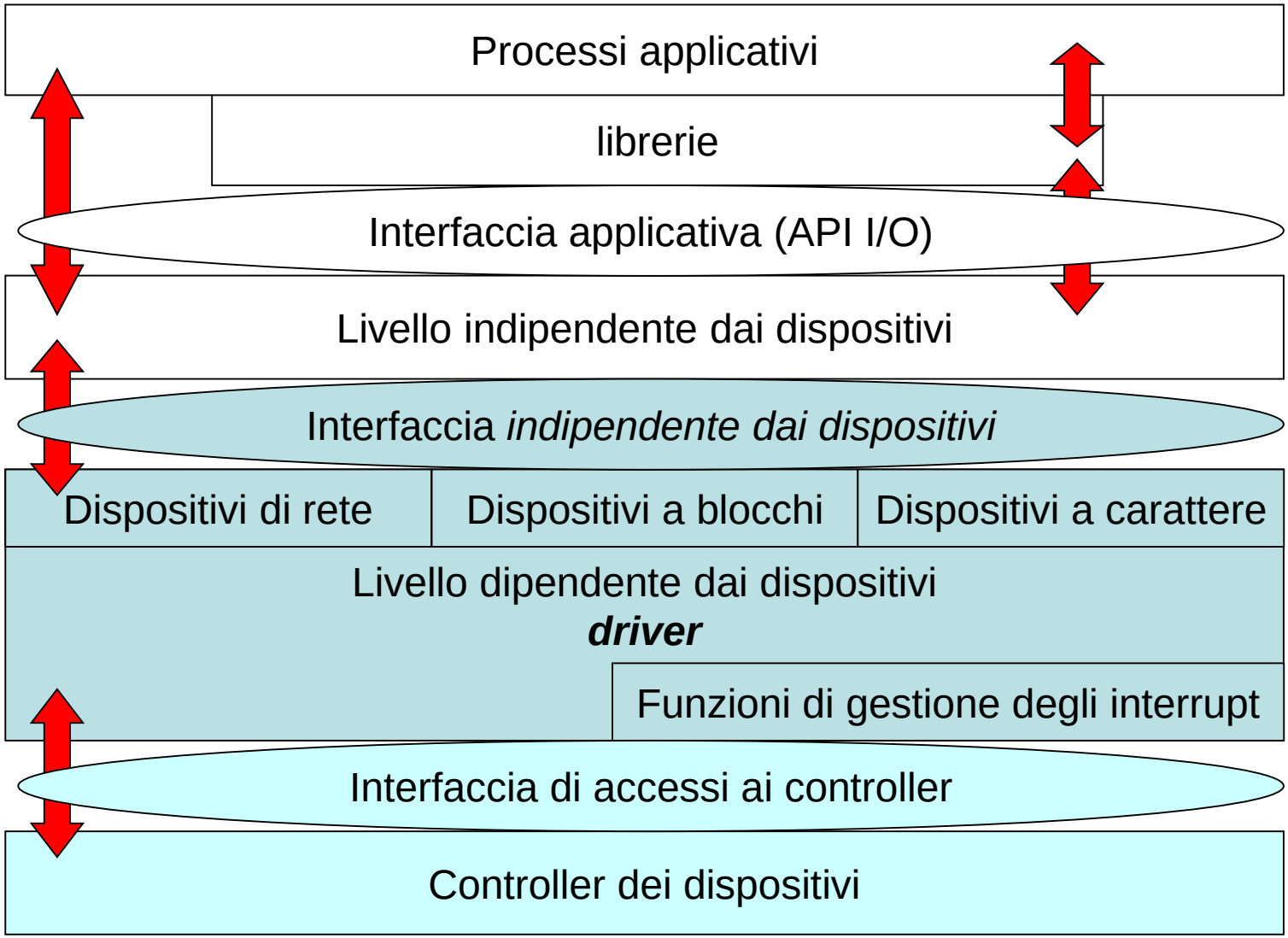
- Per esempio, un disco è una risorsa condivisa poiché più processi, leggono e/o scrivono file memorizzati sullo stesso disco.
- In altri casi, ad esempio nel caso di una stampante, il dispositivo deve essere assegnato a un processo per volta per evitare che pagine appartenenti a un processo siano alternate a pagine di altri processi.
- Per i dispositivi lenti, una tecnica spesso utilizzata al posto dell'allocazione dinamica di un dispositivo è la tecnica di ***spooling (spool* - Simultaneous Peripheral Operations On-line**) che consente di ridurre i tempi di attesa a un processo applicativo che, nel richiedere l'uso esclusivo del dispositivo, e trovandolo già occupato, deve attendere il suo rilascio per un tempo spesso lungo e difficilmente prevedibile.

- Tale tecnica è spesso usata per risorse come la stampante. In questo caso il processo applicativo non invia dati direttamente alla stampante ma genera file di stampa in una directory del disco, detta **directory di spool**.
- I file contenuti in questa directory saranno successivamente elaborati da un processo di sistema che si occupa della gestione della stampante.
- Dinamicamente si viene quindi a creare una **lista di *file di spool*** in attesa di essere letti e inviati in stampa dal processo di sistema uno alla volta.
- La coda dei file che si crea è resa visibile all'utente che in ogni momento può gestire tale coda (in base a determinati diritti di accesso) rimuovendo, ad esempio, alcuni dei file in attesa.
- Ad esempio, in unix un *demone* (processo di sistema che svolge un servizio) di stampa molto usato è **lpd (line printer daemon)**, e il client di stampa è **lpr (line printer)**. L'utente per gestire la coda di stampa può usare l'utility **lpq** (**line printer queue**) che consente di visualizzare i job in coda ed eventualmente rimuoverli con **lprm.**

- Inoltre può usare **lpc (line printer control)** per visualizzare lo stato delle code e intervenire su alcune problematiche del funzionamento delle stampanti (per il completo controllo dei comandi lcp è necessario avere i diritti di root).
- Come già detto, un disco è condiviso tra più processi. Le operazioni di lettura e/o scrittura dei vari processi non avvengono contemporaneamente, ma in sequenza. Pertanto si gestiscono le richieste mediante una lista. Quindi, dinamicamente si forma una coda di richieste di accesso ai settori del disco da parte di processi diversi, che devono essere soddisfatte una alla volta, in un ordine stabilito da qualche strategia.
- Nasce quindi il problema di come ordinare le esecuzioni di tutti i trasferimenti pendenti.
- Le politiche di schedulazione più semplici sono la **FIFO** che evita sicuramente lo ***starvation,*** oppure quelle **basate sulle priorità** dei processi che hanno richiesto il trasferimento.

- Spesso, però, tenendo conto delle caratteristiche hardware dei dischi si preferisce realizzare una **diversa politica di schedulazione**.
- Infatti il tempo medio di accesso al disco dipende soprattutto da due parametri: il primo, e più rilevante è detto **tempo di *seek*** che è l'intervallo di tempo necessario per spostare la testina dalla posizione corrente alla posizione desiderata. Questo tempo di spostamento meccanico della testina è di alcuni ordini di grandezza superiore al tempo effettivo per il trasferimento dei dati. C'è, inoltre, anche da considerare il **tempo di rotazione** necessario affinché il settore su cui operare passi sotto la testina. In base a queste considerazioni, per ridurre il tempo medio di accesso, è necessario ridurre il tempo di seek e pertanto si preferiscono algoritmi di schedulazione che, istante per istante, selezionano il trasferimento che riguarda una delle tracce più vicine a quella su cui è posizionata la testina.

Struttura logica del sottosistema di I/O



Livello dipendente dai dispositivi

- Questo livello del sottosistema di I/O ha il compito di nascondere ai livelli soprastanti tutti i dettagli relativi ai controller e ai relativi dispositivi definendo un'**interfaccia *indipendente dai dispositivi (driver)***.
- Ogni dispositivo è rappresentato mediante una **struttura dati** *detta* ***descrittore di dispositivo*** a cui è possibile accedere mediante le funzioni che fanno parte del driver.
- Ciascuna di queste funzioni, spesso realizzata in linguaggio assembly, invia gli opportuni comandi ai registri del controllore del dispositivo mediante le istruzioni di I/O, e coordina in tal modo le operazioni del dispositivo.
- Poiché esiste una grande varietà di dispositivi e molteplici modi di funzionare, i driver hanno interfacce con caratteristiche differenti, in base al tipo di dispositivo.

- Per esempio generalmente le funzioni di accesso ai **dispositivi a blocchi** sono diverse dalle funzioni con cui si accede ai **dispositivi a carattere** o ai **dispositivi di rete**. Nel caso di dispositivi a blocchi si accede ai dispostivi tramite funzioni del tipo **read, write**, **seek**, mentre l'accesso a un dispositivo a carattere avviene con funzioni del tipo **get o put**.
- Alcune funzioni appartenenti all'interfaccia del driver hanno le stesse *"firme"* delle funzioni del livello di interfaccia applicativa (spesso si aggiunge il carattere _ davanti al nome). Ad esempio le due tipiche funzioni di accesso in lettura e scrittura su un dispositivo, possono avere la seguente firma:

  n = _read (dispositivo, buffer, nbytes);

  n = _write (dispositivo, buffer, nbytes);

  dove, rispetto alle corrispondenti funzioni dell'interfaccia applicativa è diverso il modo di identificare il dispositivo, non più mediante nomi simbolici ma con indirizzi (fisici o virtuali) e ora il buffer è posto nel kernel e non nell'area virtuale del processo applicativo.

- Le principali funzioni svolte dal driver sono

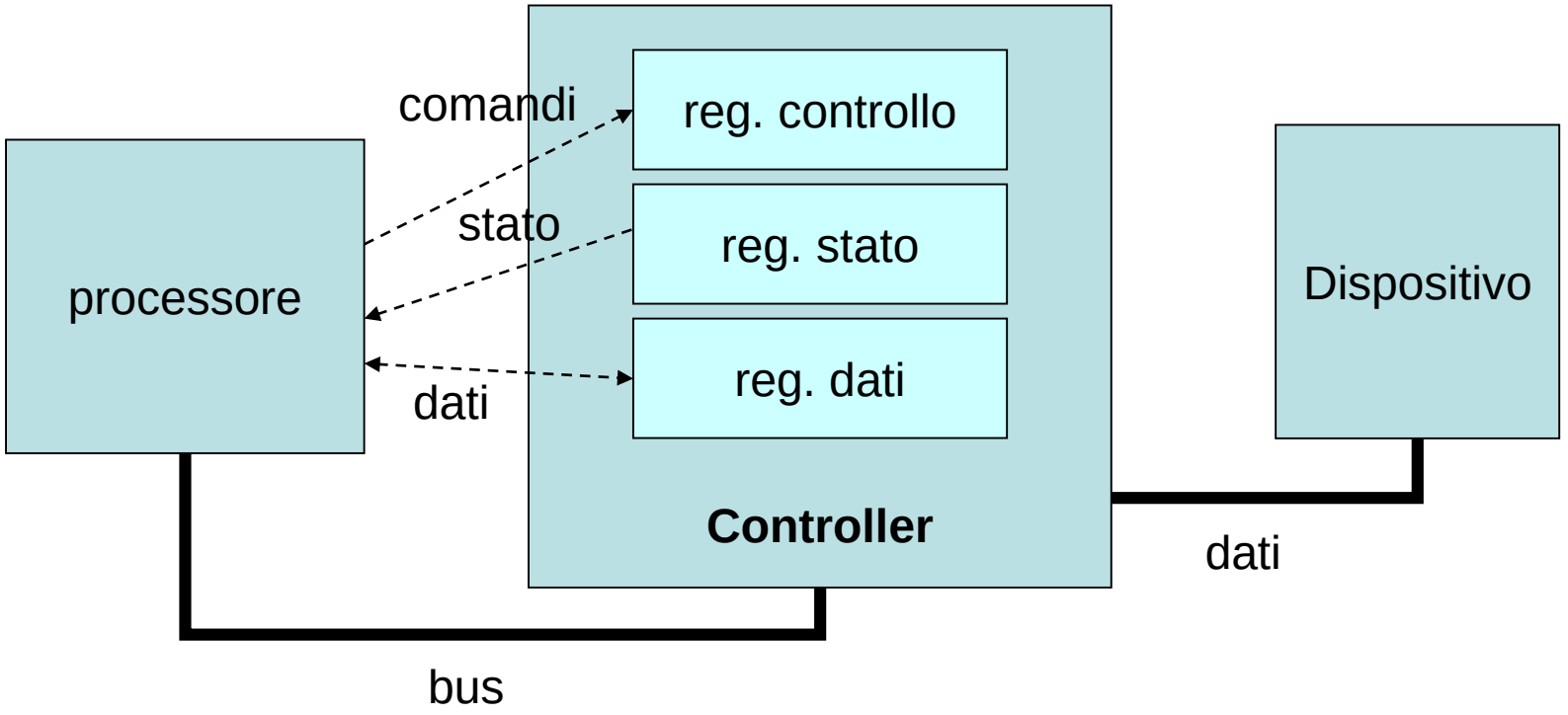
  - **attivare il dispositivo;**
  - **eseguire la funzione richiesta;**
  - **gestire localmente le eccezioni che si possono risolvere o, altrimenti, interrompere l'operazione e propagarle al livello superiore;**
  - **sincronizzare il processo applicativo che ha richiesto l'operazione con la fine dell'operazione stessa;**
  - **disattivare il dispositivo alla fine dell'operazione.**

- E’ bene notare che le operazioni svolte da un dispositivo sono concorrenti e asincrone con le operazioni svolte dal processore. Infatti, generalmente il controller di un dispositivo utilizza un **processore specializzato***,* che esegue i microprogrammi memorizzati nel suo **firmware**, che svolgono particolari operazioni e in modo indipendente dalla CPU.
- Pertanto le funzioni di *I/O* possono essere realizzate in modo che si comportino in modo **asincrono** nei confronti del processo applicativo che le chiama. Ad esempio la funzione **read**, quando è chiamata da un processo applicativo, attiva la lettura di dati da un dispositivo e termina restituendo il controllo al processo chiamante, il quale continua la sua esecuzione in parallelo con le operazioni svolte dal dispositivo.
- Il funzionamento asincrono è generalmente più efficiente del sincrono, ma è anche più complesso poiché, durante la sua esecuzione il processo deve poi stabilire quando termina l’operazione.

- Con la ***modalità sincrona****,* invece*,* un processo è sospeso dopo aver chiamato una funzione di I/O, per essere riattivato alla fine dell'operazione.
- Poiché in realtà le operazione svolte da un processo applicativo sono asincrone a quelle svolte da un dispositivo, è necessario sincronizzare le loro attività ricorrendo al meccanismo hardware delle interruzioni per garantire il comportamento sincrono.
- Se infatti un dispositivo funziona a **interruzione di programma**, alla fine di ogni operazione il controller invia alla CPU un segnale di interruzione per ottenere la corretta sincronizzazione. Per questo motivo, l'insieme delle funzioni di servizio delle interruzioni provenienti dai dispositivi *(interrupt handler)* sono incluse nel ***livello dipendente dai dispositivi****.*

Controller di un dispositivo

- La figura seguente mostra una schema semplificato di un controller di un dispositivo.
- Come già descritto, la CPU comunica con il controller tramite i registri di cui esso è dotato e che sono indirizzati mediante le istruzioni macchina di I/O e/o con le stesse istruzioni usate per l'accesso alla memoria principale (memory mapped).
- Il numero e il tipo dei registri presenti nel controller varia a seconda del dispositivo e dipende dalla complessità delle funzioni che il dispositivo è in grado di svolgere.
- In modo molto semplificato possiamo distinguere la presenza di tre registri (o gruppi di registri) indicati con i nomi di ***registro di controllo***, ***registro di stato*** e ***registro dati*** (o buffer del controller).



Controllore di un dispositivo

- Il **registro di controllo** consente alla CPU di programmare il funzionamento del dispositivo. Tipicamente, è un **registro in sola scrittura** nel quale la CPU può scrivere valori che rappresentano i **codici operativi** che specificano le operazioni che il dispositivo deve svolgere.
- Spesso è presente un bit ***(bit di abilitazione alle interruzioni)*** che consente di abilitare il controller a inviare un segnale di interruzione alla CPU (o al DMA) alla fine dell'operazione svolta dal dispositivo.
- Il **registro di stato** è un registro in **sola lettura** ed è utilizzato dal controller per segnalare la stato in cui si trova il dispositivo, e che può quindi essere letto dalla CPU.
- Nel registro di stato è presente generalmente un bit ***(flag end_flag)*** che il controller setta alla **fine di un'operazione** da parte del dispositivo. Quando il valore del flag passa da zero a uno, se il dispositivo è stato abilitato alle interruzioni, viene inviato un segnale d'interruzione alla CPU.
- Sono presenti inoltre uno o più bit che specificano il codice dell'eccezione che può verificarsi durante le operazioni del dispositivo.

- Infine, il **registro dati** rappresenta il **buffer del controllore** nel quale la CPU inserisce i dati da trasferire in output, o dal quale preleva i dati letti in fase di input.

Funzionamento del dispositivo

- In pratica, ogni dispositivo esegue una sequenza di operazioni relative a routine implementate nel firmware del dispositivo che funziona in parallelo alla CPU.
- Indicheremo questo processo col termine di ***processo dispositivo***. Il dispositivo esegue le seguenti operazioni:

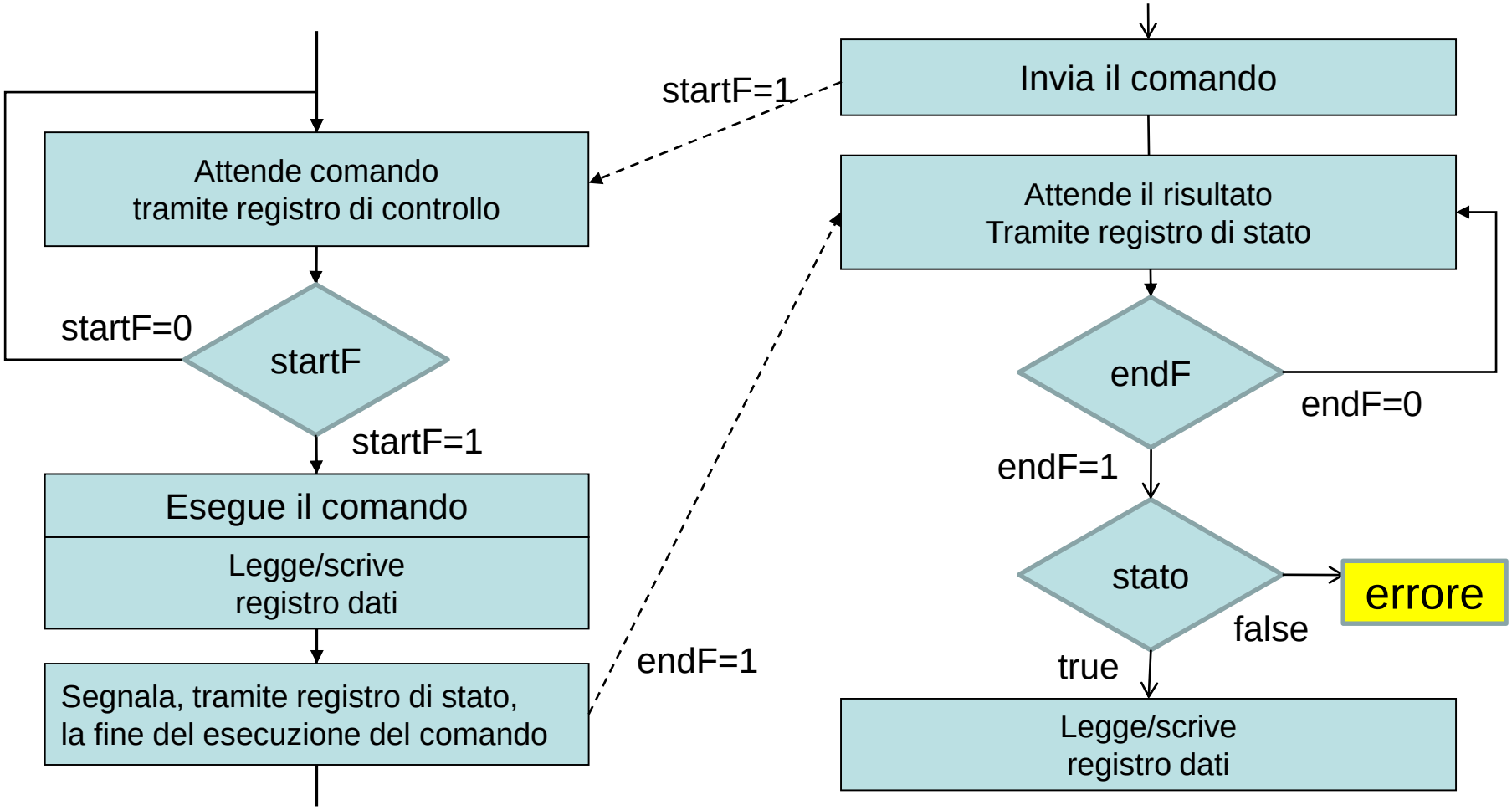
  - **attende di ricevere un comando;**
  - **Una volta ricevuto il comando, esegue le operazioni ad esso corrispondenti, comunicando con la CPU mediante il registro dati;**
  - **alla fine dell'operazione registra l'esito nel registro di stato.**
  - **Quindi ripete il ciclo e si pone di nuovo in *stand-by* in attesa di una nuovo comando.**

- In pratica è come se il processo dispositivo eseguisse il seguente codice:

```
while (true) {
    while (startF==0);//attesa invio di un comando
    <ESEGUE COMANDO>;
    <REGISTRA ESITO DEL COMANDO>
    endF=1;
}
```

Comunicazione tra processo e dispositivo

- Processo e dispositivo comunicano in base al seguente schema:



Gestione di un dispositivo mediante controllo di programma

In base allo schema della figura precedente, il processo applicativo esegue, per effettuare il trasferimento dei dati (ad esempio lettura di N dati), un codice del tipo:

```
endf=0;
for (i=0;i<N;i++){
    <PREPARA IL COMANDO>
    <INVIA IL COMANDO tra cui startF=1>;
    while (endF==0);  /* ciclo di attesa sul flag
                         endF */
    // attende la fine del comando
    <VERIFICA L'ESITO>;
    /* fa il test del registro di stato  per
       eventuali errori */
    if (stato)
          <LEGGI IL DATO DAL REGISTRO DATI>;
}
```

- Il processo, per N volte, deve attendere che il dato sia disponibile nel registro dati del controllore. Si tratta di un **ciclo di attesa attivo (polling)**.
- Generalmente, questo schema **non è adatto per essere usato nei SO multiprogrammati**, dove è invece conveniente sospendere un processo quando è in attesa di un particolare evento.

Gestione di un dispositivo mediante interruzione

- Questa tecnica, consente ad un processo applicativo di sospendersi fino a quando il dato è pronto.
- Il processo si sospende utilizzando un **semaforo inizializzato a zero** e chiamando una **wait** su tale semaforo.
- Chiamando il **semaforo *dato_pronto*** si ha:

```
semaforo dato_pronto;
dato_pronto.value = 0;
for (int=0;i++;i<N){
    <PREPARA IL COMANDO>;
    <INVIA IL COMANDO tra cui startF=1>;
    wait(dato_pronto); //attesa del dato
    <VERIFICA L'ESITO>;
    // fa il test del registro di stato status
       per eventuali errori
    if (stato)
          <LEGGI IL DATO DAL REGISTRO DATI>;
 }
```

- In base allo pseudo-codice precedente, il processo chiamando la funzione **wait(dato_pronto)** viene sospeso, in quanto il semaforo è inizializzato a **0**.
- Dovrà essere il controller del dispositivo quindi a **risvegliare il processo applicativo**, quando il dato sarà pronto.
- Il dispositivo deve essere programmato in modo che possa generare un **segnale di interruzione** quando il dato è pronto (settando il bit di abilitazione delle interruzioni nel registro di controllo).
- Quando il controller genera l'interruzione va in esecuzione la ***funzione di servizio relativa all' interruzione del dispositivo***. Questa funzione conterrà la chiamata di sistema **signal** sul semaforo che consentirà di risvegliare il processo sospeso. Per il precedente esempio eseguirà quindi **signal(dato_pronto).**

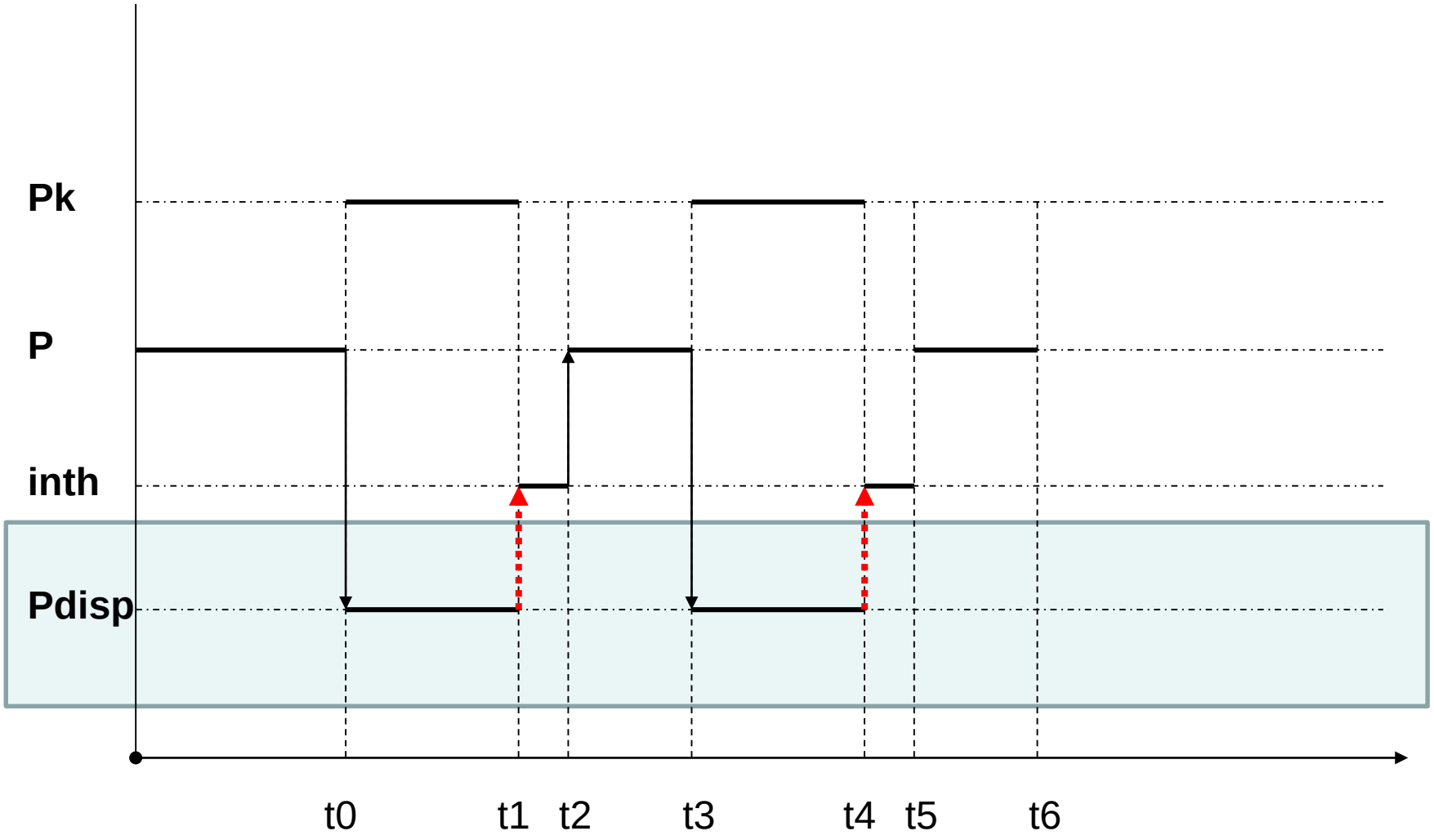


Diagramma temporale della gestione a interruzione

- La tecnica di **gestione di un dispositivo a interruzione di programma**, è poco efficiente poiché **sospende per N volte consecutive il processo applicativo** che deve eseguire il trasferimento dei dati, generando continui cambi di contesto i quali producono un eccessivo overhead.

Descrittore di un dispositivo

- Una soluzione al problema di cui sopra consiste nel fornire al processo applicativo una funzione di sistema che consenta di specificare il numero di dati da trasferire e l'indirizzo di memoria dove i dati devono essere trasferiti (o da prelevare nel caso di operazione di scrittura).
- Il diagramma temporale della figura seguente mostra il funzionamento di tale soluzione.
- Ora il processo **P** che esegue il trasferimento di dati resta bloccato fino al termine del trasferimento.

- Per ottenere questo tipo di funzionamento è necessario che la routine di interruzione **inth** riesca a distinguere tra le interruzioni intermedie che provvedono a trasferire un blocco di dati da quella finale che provvede a risvegliare il processo P.
- Una soluzione a questo problema consiste nel utilizzare una struttura dati che descrive il dispositivo alla quale possano accedere sia il processo applicativo, mediante le funzioni dell'interfaccia ***indipendente dai dispositivi***, sia la funzione **inth** che gestisce le interruzioni lanciate dal dispositivo.
- Il descrittore del dispositivo insieme alle funzioni del sistema dipendenti dal dispositivo e alla funzione di interruzione costituiscono il **driver del dispositivo**.
- Il descrittore del dispositivo ha il compito di:
  - **Nascondere le informazioni associate al dispositivo**
  - **Consentire la comunicazione di informazioni tra il processo applicativo e il dispositivo.**

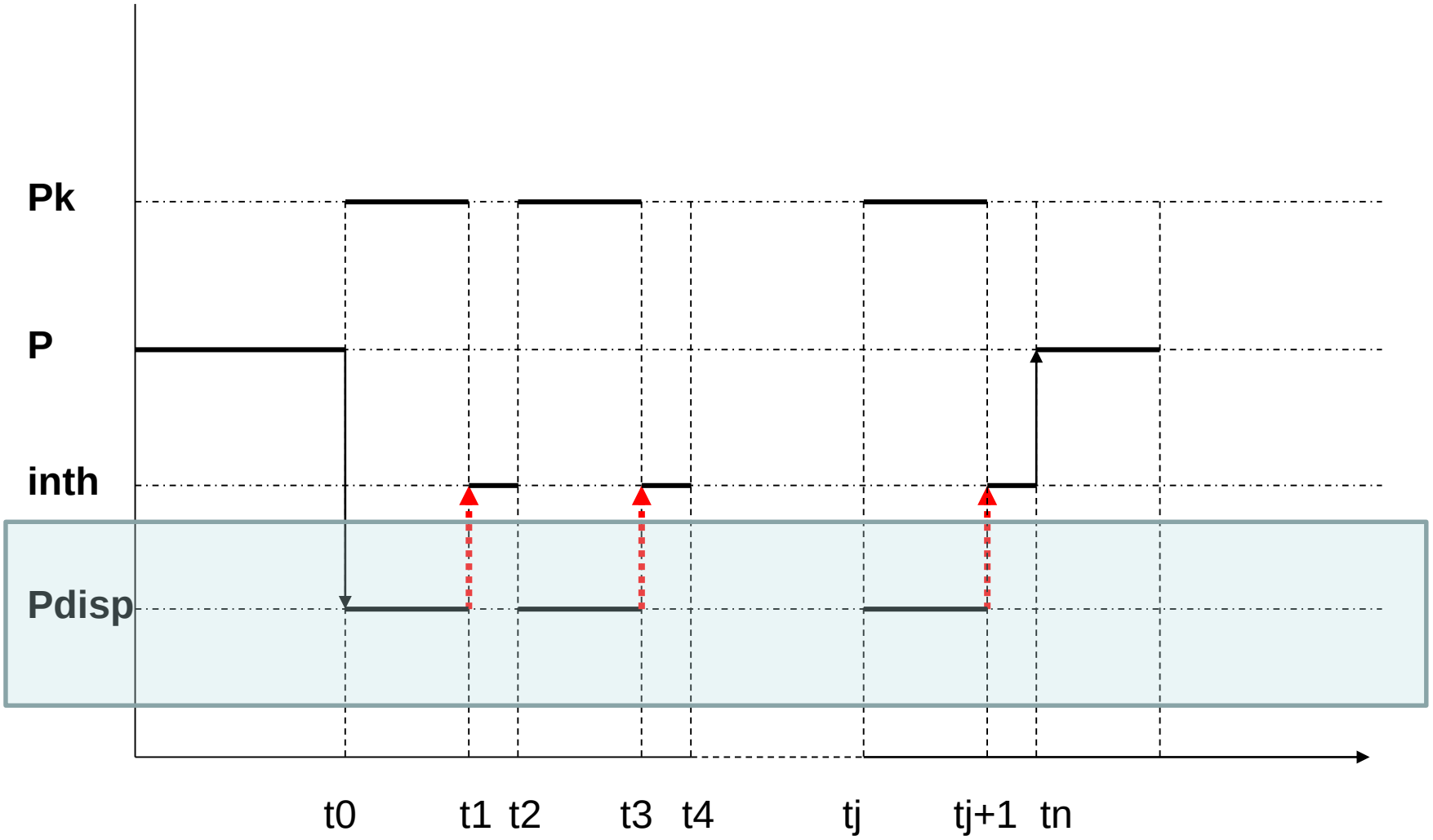


Diagramma temporale della gestione a interruzione
Trasferimento di N dati consecutivi

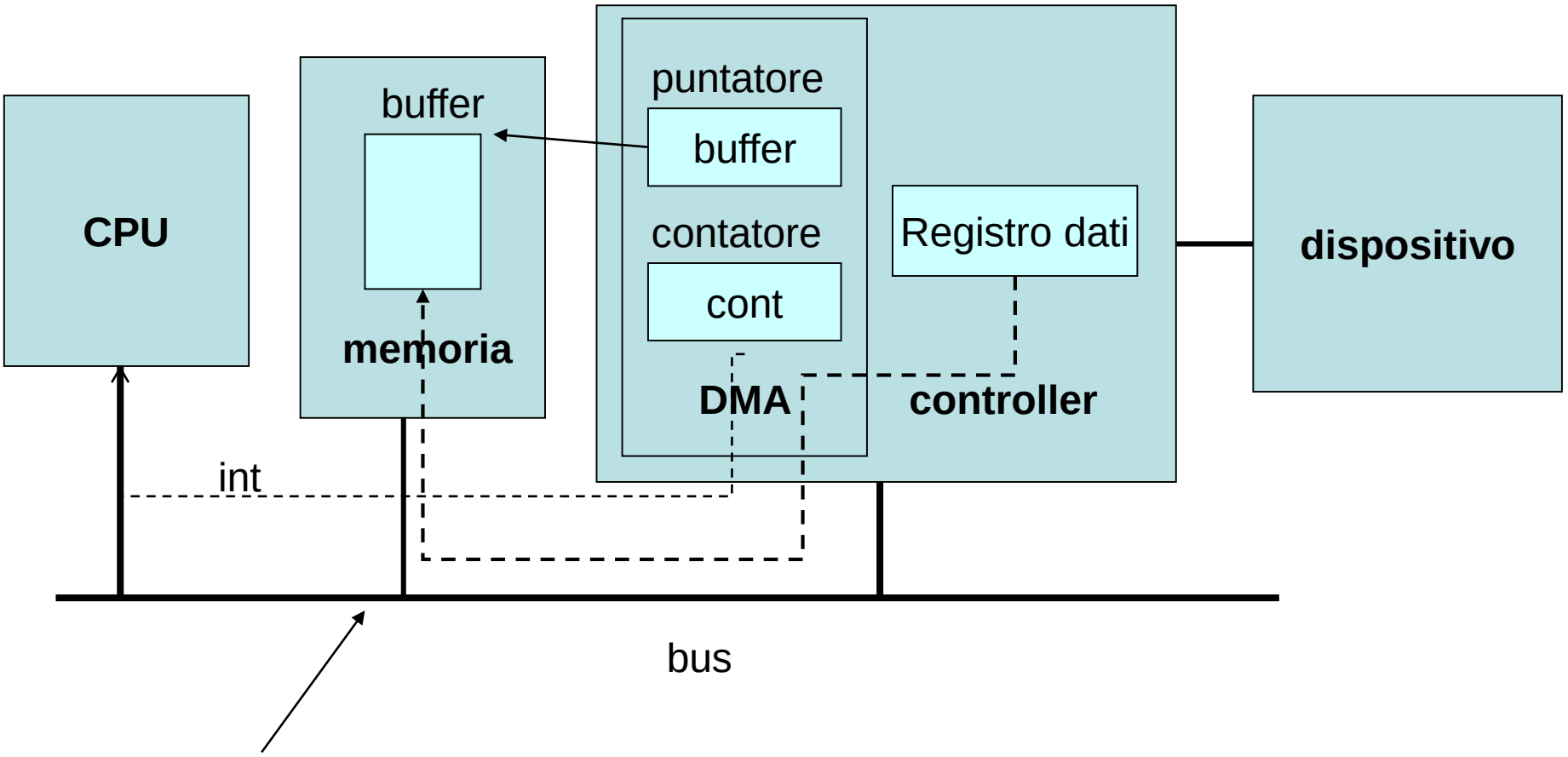
La struttura dati del descrittore di dispositivo

- Il descrittore varia molto in dipendenza della complessità del dispositivo. Una tipica struttura contiene i seguenti campi:
  - tre campi contenenti gli indirizzi dei registri del controllore del dispositivo **controllo**, **dati e stato**;
  - un campo **dato_pronto** di tipo semaforo per la sincronizzazione tra il processo applicativo e la funzione **inth** di risposta alle interruzioni generate dal dispositivo;
  - un campo **contatore** per indicare il numero di byte da trasferire;
  - un campo **pBuffer** per contenere l'indirizzo di memoria del buffer in cui (o da cui) trasferire i dati;
  - un campo **stato** per memorizzare l'esito delle operazioni svolte dal dispositivo.

Esempio di descrittore di dispositivo

Gestione di un dispositivo con DMA

- Il DMA consente di trasferire i dati dal registro del controllore direttamente in memoria tramite i bus di sistema operando in cycle stealing (cioè condivide il bus di sistema con la CPU).
- Questa tecnica riduce il tempo di trasferimento di un blocco di dati e consente alla CPU di svolgere altri compiti.
- Il DMA genera un segnale di interruzione per ogni blocco di dati, pronto per essere trasferito in memoria.
- Nel DMA sono presenti due registri **contatore** e **puntatore.** Nel primo sarà scritto il numero di byte (o parole) da trasferire e nel secondo l'indirizzo di memoria ove iniziare il trasferimento.
- Il contatore viene decrementato e il puntatore sarà incrementato per ogni byte trasferito. Quando il contatore assume il valore 0, il DMA invia alla CPU un segnale di interruzione.
- I DMA sono usati nei dispositivi a blocchi.


CPU
buffer
memoria
puntatore
buffer
contatore
cont
DMA
Registro dati
controller
dispositivo
int
bus
L’uso del bus viene condiviso
tra CPU e DMA

Driver di un dispositivo

- Vediamo come è strutturato un driver di dispositivo, mostrando uno schema di funzione di lettura (la scrittura è simile) dell'interfaccia *e la funzione di risposta all'interruzione.*

  int _read(int disp, char *pbuf, int cont)

  - **disp** identifica il dispositivo
  - **pbuf** il puntatore al buffer di sistema in cui trasferire i dati letti
  - **cont** il numero di byte da leggere

- La funzione read ritorna il valore -1 nel caso si verifichi un'eccezione.

```
int _read(int disp, char *pbuf, int cont){
     //numero di dati da leggere
     descrittore[disp].contatore = cont;
     //indirizzo del buffer di sistema
     descrittore[disp].pbuffer = pbuf;
     // bit start=1; bit r=1 (lettura)
    descrittore[disp].controllo = <opcode_read>
     <ATTIVAZIONE DEL DISPOSITIVO>;
     // sospensione del processo
     wait(descrittore[disp].dato_pronto);

     // il processo torna in esecuzione
     if (descrittore[disp].stato == <codice errore>)
            return -1;
     else
        return cont-descrittore[disp].contatore;
}
```

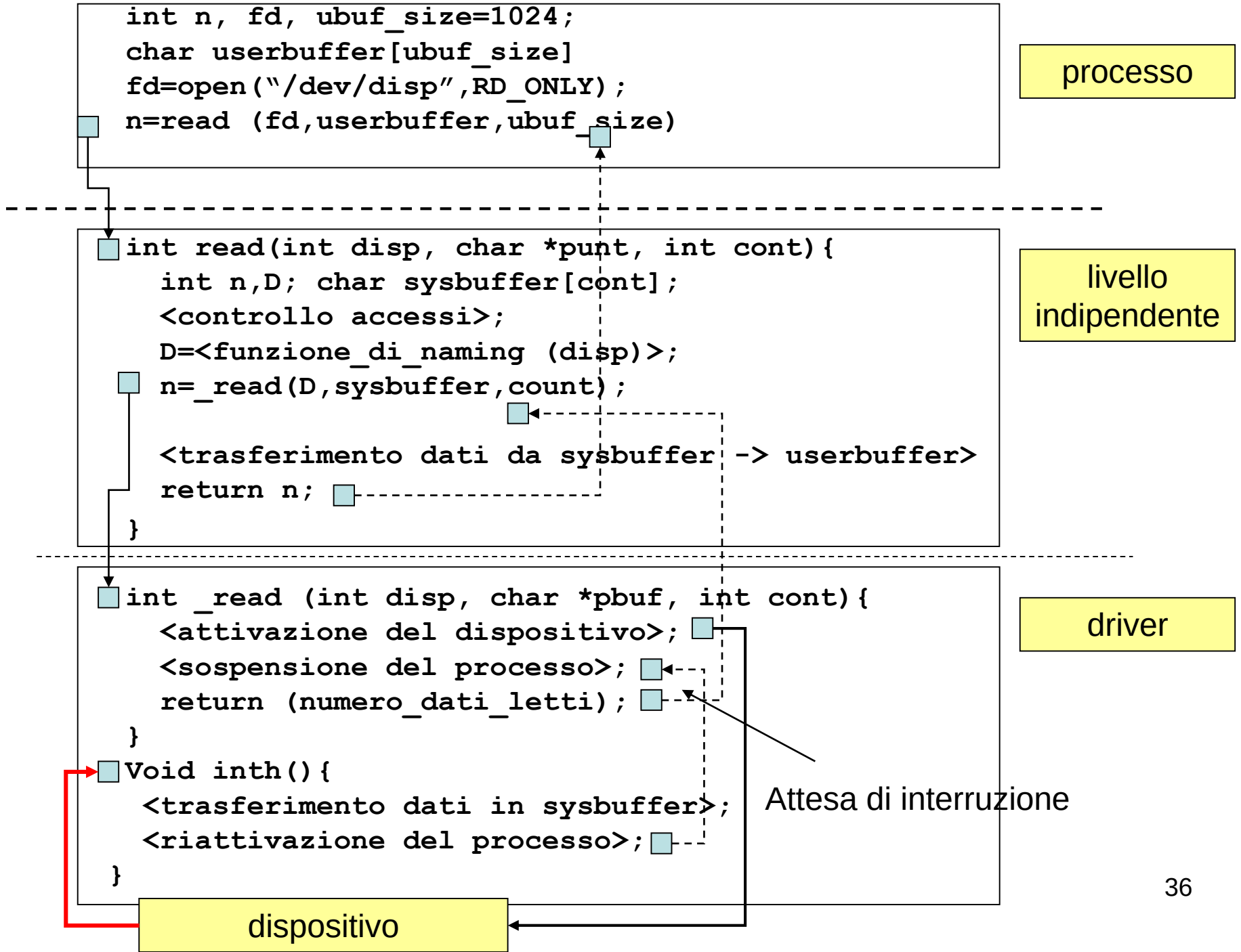
```
void inth() {
  //funzione di gestione delle interruzioni
   char dato_letto;
   if (descrittore[disp].stato != <codice errore>){
       // assenza di errori
       dato_letto = <VALORE DI REGISTRO_DATI>;
       // trasferimento di dato_letto in memoria
       *descrittore[disp].puntatore = dato_letto;
       descrittore[disp].puntatore++;
       descrittore[disp].contatore--;
       if (descrittore[disp].contatore !=0)
              <RIATTIVAZIONE DISPOSITIVO>;
       else {
              descrittore[disp].stato=<TERMINAZIONE OK>;
              <DISATTIVAZIONE DEL DISPOSITIVO>;
       }
```

```
    else {
        //presenza di errori
        <FUNZIONE GESTIONE ERRORE>;
        if (<errore_grave>)
              descrittore[disp].stato=<codice errore>

    }
    //riattivazione del processo
    signal(descrittore[disp].dato_pronto);
    return; // ritorno da interruzione
}
```

Flusso di controllo durante un trasferimento

- Vediamo un esempio di come è strutturato il flusso di controllo durante l'esecuzione di una chiamata di sistema relativa ad un trasferimento di dati.
- L'esempio mostra l'esecuzione di una chiamata di sistema che un processo P esegue per leggere, **in modalità sincrona,** un blocco di byte da un dispositivo.


int n, fd, ubuf_size=1024;
char userbuffer[ubuf_size]
fd=open("/dev/disp",RD_ONLY);
n=read (fd,userbuffer,ubuf_size)
processo
int read(int disp, char *punt, int cont){
int n,D; char sysbuffer[cont];
<controllo accessi>;
D=<funzione_di_naming (disp)>;
n=_read(D,sysbuffer,count);
<trasferimento dati da sysbuffer -> userbuffer>
return n;
}
livello indipendente
int _read (int disp, char *pbuf, int cont){
<attivazione del dispositivo>;
<sospensione del processo>;
return (numero_dati_letti);
}
Void inth(){
<trasferimento dati in sysbuffer>;
<riattivazione del processo>;
}
driver
Attesa di interruzione
dispositivo