

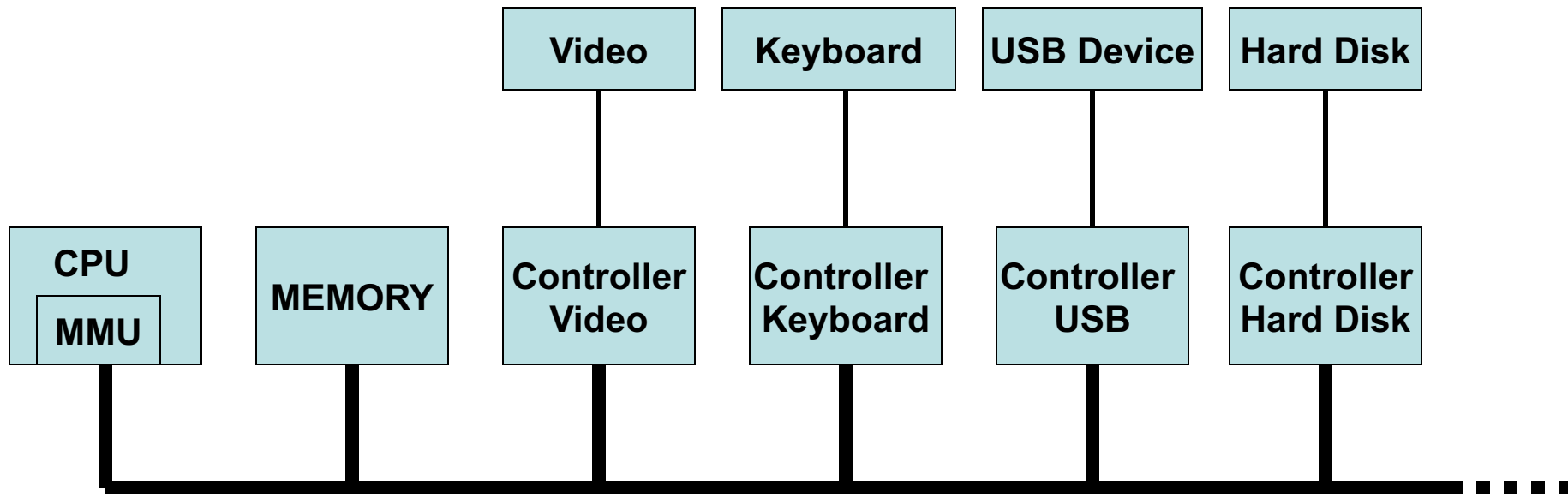
# **RICHIAMI DI ARCHITETTURA E** **SYSTEM CALL**

- CPU ed interrupt
- Dispositivi di I/O e DMA
- System call

# CPU E INTERRUPT

Il S.O. interagisce direttamente con l'hardware: il progettista del S.O. deve conoscere l'hardware su cui gira il S.O..

Semplificando, rappresentiamo l'hardware come segue, tenendo conto che, concettualmente, Hard Disk Drive e Solid State Drive sono equivalenti:



# CPU

- La CPU **preleva** le istruzioni dalla memoria, le **decodifica** e le **esegue**.
- Ciclo “tradizionale” di funzionamento della CPU:  
1: fetch; 2: decodifica; 3: esecuzione; 4: goto 1;
- La CPU si avvale di **registri generali** (general purpose registers), che costituiscono il primo livello della gerarchia di memoria e che memorizzano variabili e risultati temporanei.
- La CPU si avvale di **registri di controllo** (control registers), che servono per controllarne il funzionamento.

## CPU – registri di controllo:

- **Program Counter (PC)**: indirizzo istruzione da prelevare.
- **Stack Pointer (SP)**: puntatore al top dello stack contenente i frame delle procedure già iniziate ma non ancora terminate.
- **Program Status Word (PSW)**:
  - \* **Privileged Mode (PM)**: bit di modalità user/kernel (Pag. 8);
  - \* **Condition Code (CC)**: codici di condizione impostati da istruzioni di confronto / codici di proprietà di operazioni aritmetiche;
  - \* **Interrupt Mask (IM)**: bit usati per gestire interrupt (Pag. 19);
  - \* **Interrupt Code (IC)**: bit usati per gestire interrupt (Pag. 22);
  - \* **Memory Protection Information (MPI)** – informazioni sulla porzione di memoria accessibile (e.g. LBR/UBR – Cap 2 e Pag. 6).

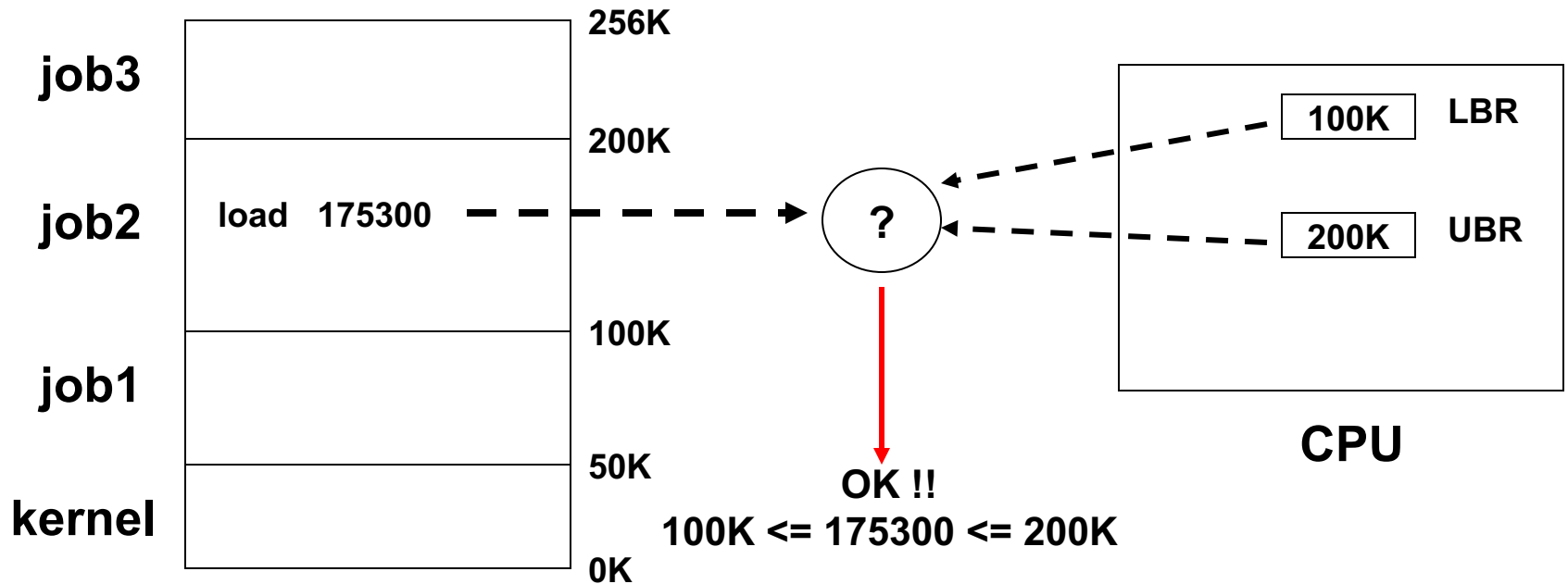
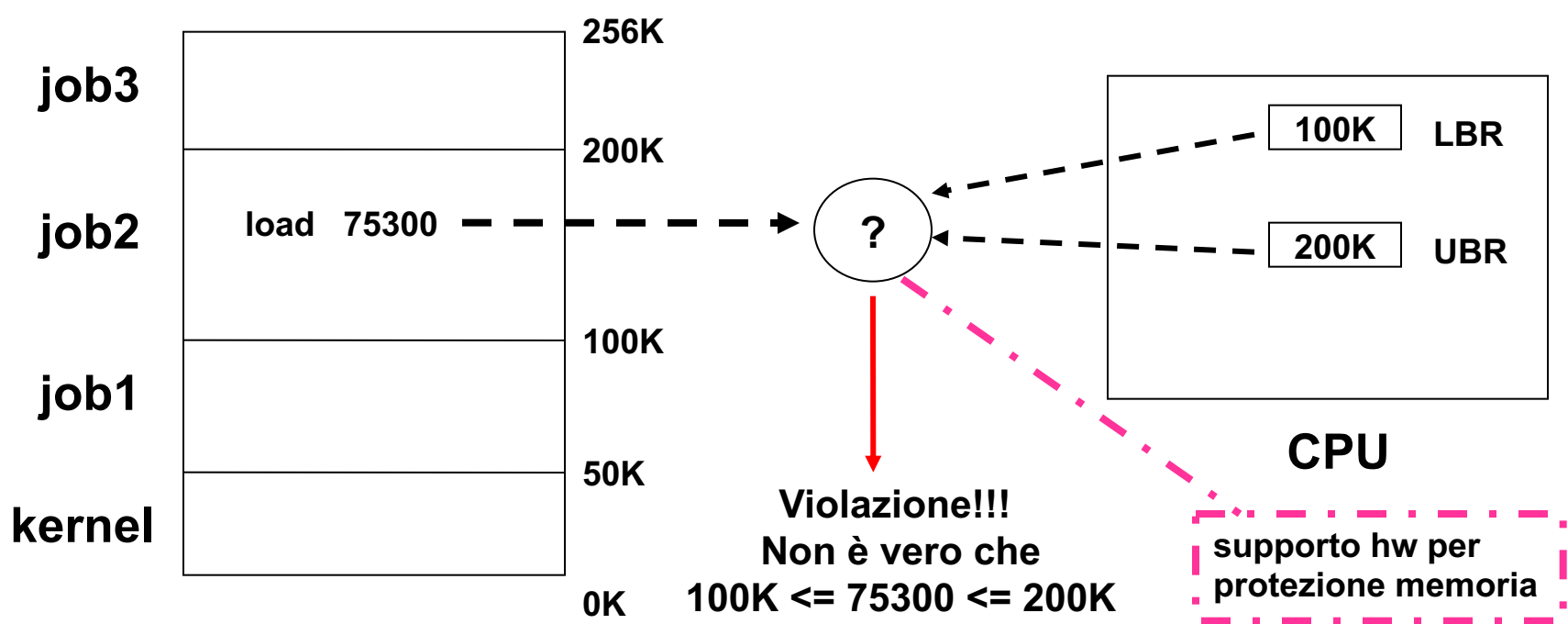
N.B. Alcuni autori considerano **PC** e **SP** parte di **PSW**.

## Ruolo dei bit **MPI**:

- Supporto hardware per costringere i job ad accedere solo al proprio spazio di memoria.

Di fatto, impediscono alla CPU di accedere agli indirizzi di memoria che non fanno parte dello spazio di memoria del job in esecuzione.

- Il numero ed il significato di tali bit dipende da come è organizzata la memoria.
- Vediamo nella prossima slide un esempio banale dove MPI = registri LBR/UBR ed assumiamo *allocazione contigua* della memoria (no paginazione e/o segmentazione, il job viene caricato in una porzione di memoria continua, per i dettagli bisogna aspettare il Cap. 7 – Gestione della Memoria).



## Ruolo del bit **PM** (**P**rivileged **M**ode).

- Le **istruzioni privilegiate**, cioè le istruzioni che modificano la **PSW**, che modificano i valori della MMU, che operano sulle porte I/O per comunicare con i dispositivi (Pag. 34),.... sono eseguibili solo in **stato kernel**.
- Esempio: le istruzioni per modificare i registri LBR/UBR devono essere privilegiate.  
Tipicamente il S.O. le esegue al momento del context switch.
- L'accesso alle risorse (memoria, dispositivi) è limitato da un opportuno **sistema di protezione** (e.g. non è possibile accedere alle strutture dati del S.O.) quando la CPU è in stato user, mentre è illimitato quando la CPU è in stato kernel.
- Pertanto i **programmi critici**, cioè i programmi che svolgono funzioni del S.O., sono eseguibili solo in stato kernel.



## Definizione:

Gli **interrupt** costituiscono un meccanismo per **notificare alla CPU un evento o una condizione** avvenuti nel sistema che devono essere trattati dal S.O..

L'obiettivo è duplice:

- **interrompere il normale ciclo di esecuzione** della CPU, cioè interrompere l'esecuzione del programma;
- **richiedere l'intervento del S.O.**, cioè eseguire codice che fa parte del S.O..

Osservazione: di norma interrompere il programma non vuol dire forzarne la terminazione. Una volta che il S.O. ha terminato l'intervento, il programma dovrà riprendere l'esecuzione.

Aspetti da considerare:

- Come si manifesta l' interrupt?

Risposta: con un segnale su una linea del bus di sistema.

- Chi invia questo segnale?

Due casi: il clock o il controller di un dispositivo (**hardware interrupt**, Pag. 11), oppure, con modalità che chiariremo, il programma in esecuzione (**program interrupt**, Pag. 41).

- Come si fa a “sottrarre” la CPU al programma in esecuzione?
- Come si fa a far partire il codice del S.O. che deve gestire l'evento/condizione?
- Come fa il programma interrotto a riprendere la propria esecuzione dopo che il S.O. ha terminato l'intervento?

**Hardware interrupt.** Distinguiamo tra:

- **I/O interrupt:** usati dai dispositivi I/O per notificare eventi alla CPU che richiedono di essere trattati dal S.O. (e.g.: terminazione di un'operazione di lettura/scrittura su disco).
- **Timer interrupt:** usati dal clock per notificare il tick.

Si tratta di **eventi asincroni** rispetto al programma in esecuzione: il programma non ha modo di prevedere se e quando questi eventi si verificano.

Gli eventi vanno gestiti dal S.O. Esempi:

- il S.O. deve “risvegliare” il programma che attendeva la terminazione della lettura/scrittura su disco,
- il S.O. controlla se il tick ha causato l'esaurimento del time slice del programma in esecuzione.

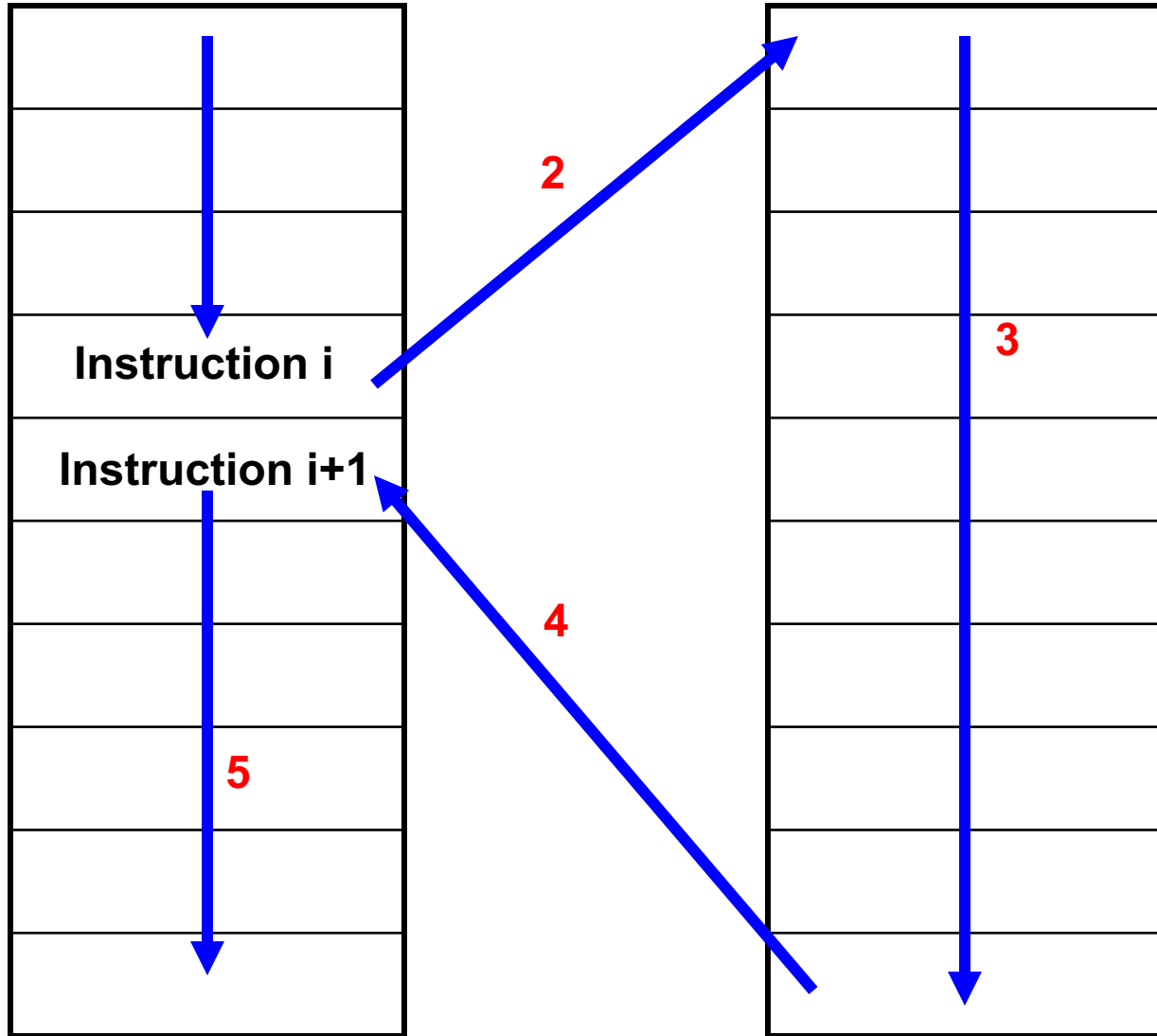
## Fasi dell' interrupt hardware - semplificate:

1. La CPU, mentre sta eseguendo l'istruzione **i** del programma **P**, riceve un **interrupt request** su una linea del bus, che verrà presa in considerazione prima di prelevare l'istruzione **i+1**;
2. terminata l'esecuzione dell'istruzione **i**, la CPU sospende l'esecuzione di **P**, cioè non preleva l'istruzione **i+1**, e salta alla procedura di gestione dell' interrupt, detta **interrupt handler**;
3. l' interrupt handler, che è parte del S.O., gestisce l' interrupt;
4. l' interrupt handler restituisce il controllo a **P**;
5. **P** riprende l'esecuzione prelevando l'istruzione **i+1**, come se nulla fosse accaduto.

## User program

## Interrupt handler (S.O.)

Interrupt  
request



- Osservazione: Le fasi 2 e 4 sono tutt'altro che banali, in quanto è immediato notare che il valore dei registri della CPU, ed in particolare del **PC**, va “salvato” nella fase 2 e “ripristinato” nella fase 4.
- I dettagli delle fasi 2 e 4 verranno chiariti a Pag. 20.
- Siamo stati un po' imprecisi:  
In base alle politiche di scheduling, al punto 4 l' interrupt handler potrebbe restituire il controllo ad un programma **P'** interrotto in passato, anziché a **P**.  
In tal caso **P** riprenderebbe l' esecuzione in seguito.  
I punti 4 e 5 vanno pertanto riscritti come segue:

## Fasi dell' interrupt hardware – semplificate - bis:

1. vedi Pag. 12;

2. vedi Pag. 12;

3. vedi Pag. 12;

4. l' interrupt handler restituisce il controllo a **P**, oppure ad un altro programma **P'** interrotto in precedenza;

5. Il programma schedulato al punto 4 riprende la propria esecuzione dal punto in cui era stato interrotto.

Nel caso venga schedulato **P**, viene prelevata l' istruzione **i+1**.

- Cosa accade se arriva un interrupt request mentre si sta eseguendo un interrupt handler?
- Esempio, cosa accade se arriva un interrupt request dal clock mentre è in esecuzione l' interrupt handler per gestire un interrupt generato dall' hard disk?
- La nuova request può essere soddisfatta come mostrato nella slide seguente, ma servono accorgimenti che dettaglieremo a breve.

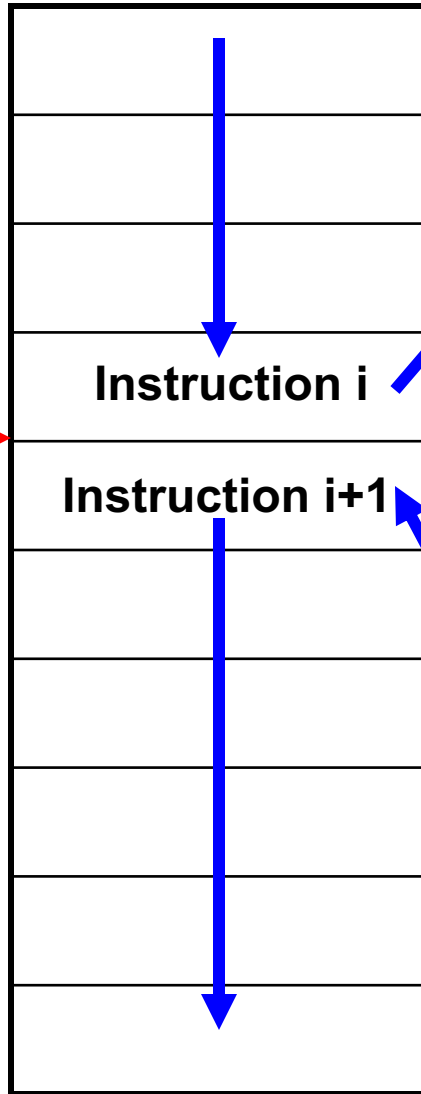


## User Program

## Disk Interrupt handler

## Clock Interrupt handler

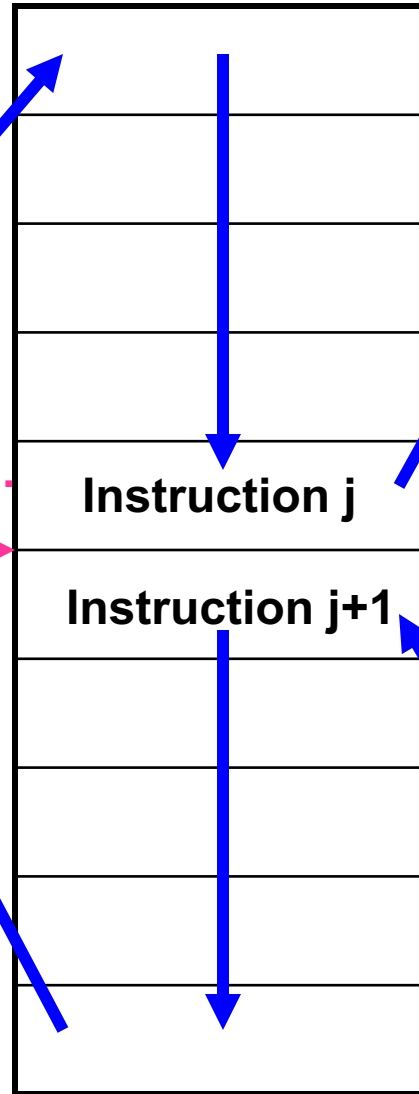
Disk  
Interrupt  
Request



Instruction i

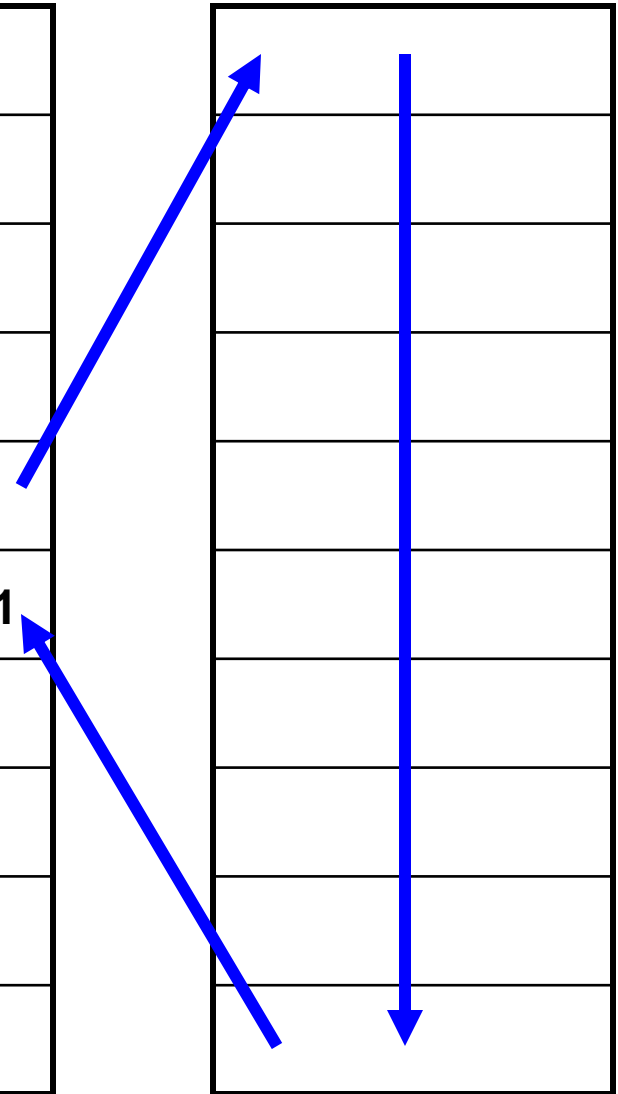
Instruction i+1

Clock  
Int. Req.




Instruction j

Instruction j+1



# Come gestire gli interrupt a cascata?

- Gli interrupt vengono organizzati in **classi di priorità**.
  - Quando si gestisce un interrupt request si ignorano *temporaneamente* le richieste di interrupt della medesima classe e delle classi con minor priorità, la cui gestione viene rimandata. Tali richieste rimangono **pendenti**.
  - Gerarchia tipica degli interrupt hardware:
    - machine error
    - clock interrupt
    - disk interrupt
    - fast device interrupt
    - slow device interrupt
- 

# Come ignorare gli interrupt request?

- Il registro **PSW** contiene alcuni bit che svolgono il ruolo di **Interrupt Mask (IM)** e che determinano le classi di interrupt abilitate (**enabled**) e quelle non abilitate (**masked off**). Implementazioni possibili:
  - l' **IM** contiene un bit (enabled/masked off) per ogni classe di interrupt;
  - l' **IM** contiene un valore ***m*** per abilitare tutti e soli gli interrupt di priorità  $\geq m$ .
- Quando il programma gira in modo user, tutti gli interrupt sono abilitati. Il programma non può modificare la **PSW**, pertanto non può disabilitare gli interrupt.
- Quando il controllo passa ad un interrupt handler, l' **IM** viene settato in modo opportuno. Come? Chiariamo subito.

Concentriamoci sulla fase 2.

Per ogni classe di interrupt abbiamo un **interrupt vector**, memorizzato nella system area, che memorizza:

- L'indirizzo della prima istruzione dell' interrupt handler, da assegnare al **PC** per attivare l' interrupt handler, cioè per avviare la fase 3.
- Il valore da assegnare al registro **PSW** durante l' esecuzione dell' interrupt handler. Il valore dei bit **IM** sono tali da abilitare/disabilitare in modo opportuno gli interrupt delle varie classi mentre si esegue l' interrupt handler.

Gli interrupt vector vengono inizializzati in memoria durante la fase di booting. Sono un esempio di struttura dati del kernel, non accessibile ai programmi user.

Rimaniamo concentrati sulla fase 2.

Per ogni classe di interrupt abbiamo una **Saved Register Information Area (SRIA)**, che serve per memorizzare i valori dei seguenti registri quando la CPU riceve l' interrupt:

- **PC**
- **SP**
- **PSW**

La SRIA può essere allocata (chiariremo nel Cap. 4):

- in una zona di memoria apposita;
- nello stack.

Dettagli fase 2: cosa accade quando arriva un interrupt abilitato?

- A. L'hardware imposta l'interrupt code (**IC**) della **PSW**, che serve per comunicare al S.O. le cause dell'interrupt e che verrà esaminato dall'interrupt handler;
- B. I valori dei registri **PC**, **SP** e **PSW** vengono salvati nell'apposita **SRIA**. Saranno sfruttati nella fase 4. Precisiamo che i registri generali non sono stati ancora salvati.
- C. I registri **PC** e **PSW** vengono impostati in base ai valori memorizzati nell'interrupt vector apposito. L'interrupt handler può pertanto partire. La fase 3 ha inizio.

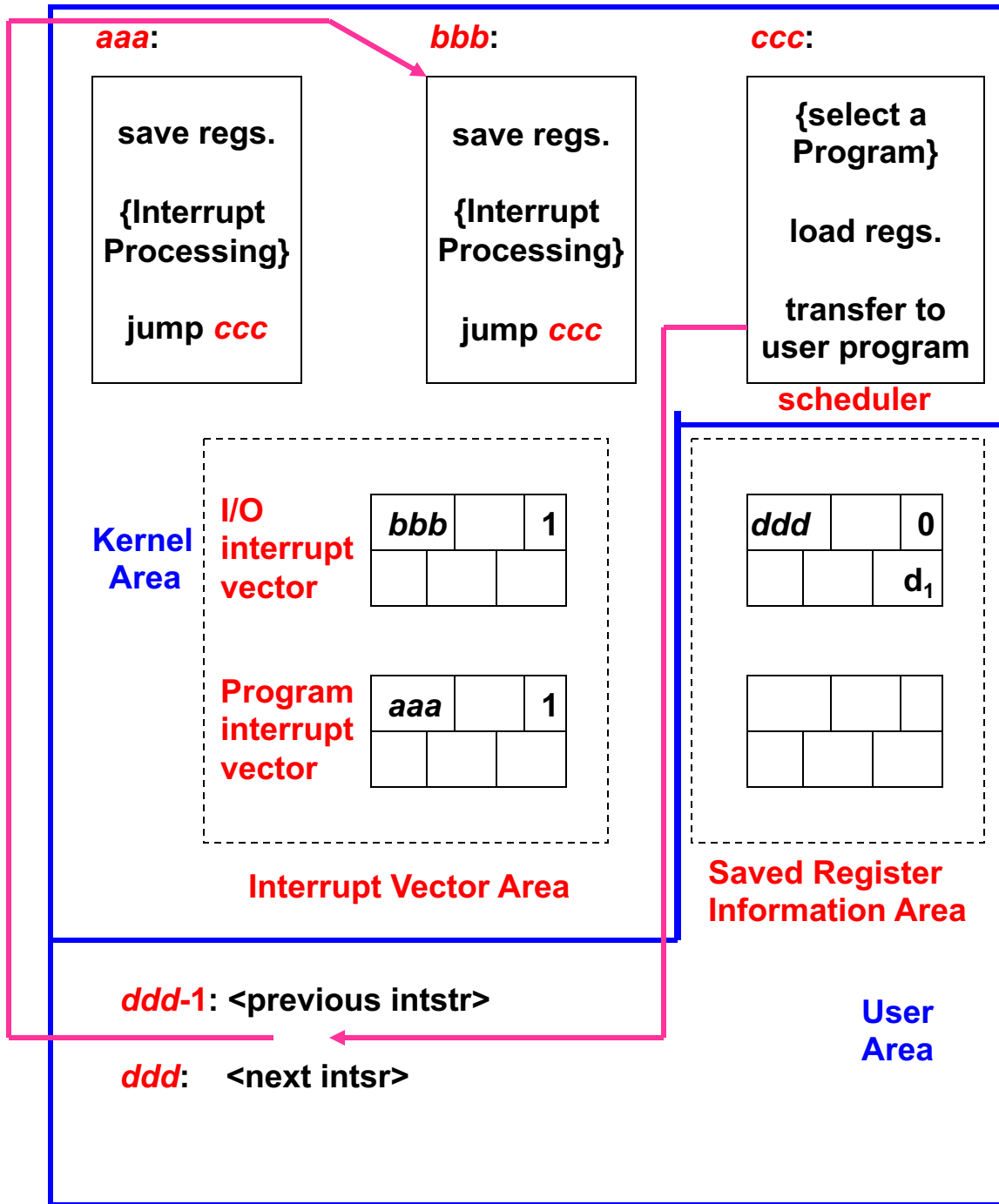
## Dettagli fase 3 – cosa fa l' interrupt handler:

- A. Salva i registri generali della CPU. A questo punto i valori di tutti i registri della CPU relativi all'esecuzione del programma **P** sono stati salvati e potranno essere ripristinati quando **P** riprenderà l' esecuzione (fase 4).  
Vedremo nel Cap.4 dove i registri generali vengono salvati.
- B. Esegue il codice apposito per gestire l' interrupt. Questo codice sfrutta il valore dell' **IC**. Esempio, se abbiamo 2 dischi ed arriva un disk interrupt, l' **IC** contiene l' indicazione circa il disco che ha causato l' interrupt, informazione indispensabile per l' interrupt handler.
- C. Salta allo scheduler, che seleziona il programma **P'** (può essere che **P' = P**) cui assegnare la CPU nella fase 4.

Dettagli fase 4: cosa fa lo scheduler dopo che ha selezionato il programma **P'** che deve riprendere l'esecuzione:

- A. I valori dei registri generali per eseguire il programma schedulato **P'** erano stati salvati (non abbiamo ancora detto dove, attendiamo il Cap. 4) e vanno ripristinati.
- B. Vanno ripristinati anche i registri di controllo che erano stati salvati nella SRIA. Spesso l'architettura offre l'istruzione **Interrupt RETURN (IRET)** per ripristinare, tutti insieme, i registri di controllo.





## Esempio:

Arriva un I/O interrupt dal dispositivo numero 1 prima che il programma utente esegua l'istruzione di indirizzo **ddd**.

Assumiamo, per facilitare il disegno, che esista un'unica classe di I/O interrupt.

Nel disegno mancano alcuni interrupt vector, quali quello del clock.

Nel disegno mancano alcuni elementi degli interrupt vector, sono riportati solo i valori da assegnare a **PC** e **PM**.

- salvataggio hw dei registri di controllo (**PC=ddd**, **PM=0**, cioè user, **IC = d<sub>1</sub>**, cioè device numero 1) nella **SRIA**;
- settaggio dei registri di controllo sfruttando l' interrupt vector (**PC=bbb**, **PM=1**, cioè kernel);
- l' interrupt handler (indirizzo **bbb**):
  - salva il contenuto dei registri generali (vedremo dove nel Cap. 4),
  - controlla l' **IC** per individuare il device (cioè **d<sub>1</sub>**),
  - esegue le sue funzioni,
  - invoca lo scheduler, il cui codice è all' indirizzo **ccc**;
- se lo scheduler seleziona il programma interrotto:
  - ripristina i registri generali precedentemente salvati,
  - ripristina i registri di controllo salvati nella S.R.I.A. (istruzione **IRET**). Il **PC** assume valore **ddd**.

## Alcune considerazioni:

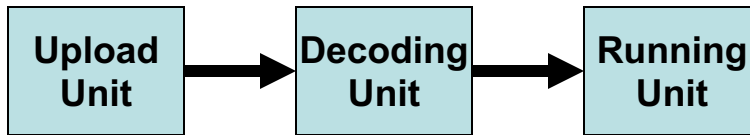
- I S.O. assumono che l'hardware supporti gli interrupt. Si tratta di un esempio di evoluzione dell'hardware dettata dallo sviluppo dei S.O..
- Riguardo al salvataggio dei registri della CPU che viene interrotta:
  - il **PC** deve essere salvato via hardware;
  - gli altri registri possono essere salvati via software, cioè dall' interrupt handler.

Il salvataggio via software è più lento, ma richiede un hardware meno complicato. Come compromesso, spesso l'hardware offre la possibilità di salvare tutti i registri di controllo, ma non i registri generali. Questo è l'approccio discusso in precedenza. Anche la **IRET** va in questa direzione.

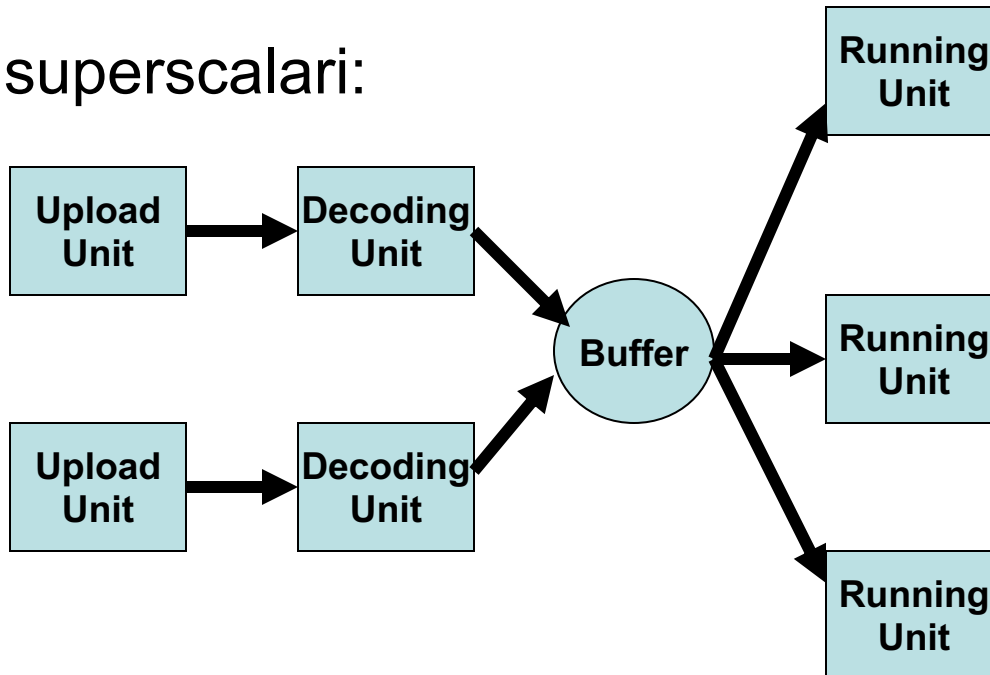
- Ovviamente i registri della CPU devono essere salvati anche quando è un interrupt handler ad essere interrotto da un interrupt.
- La gestione degli interrupt con classi di priorità fa sì che solo un interrupt di priorità maggiore possano interrompere un interrupt handler. Ne consegue che, per ogni programma **P**, una **SRIA** per ogni classe di interrupt è sufficiente.

Il ciclo tradizionale – fetch, decodifica, esecuzione – ormai non è più attuale:

- Organizzazione pipeline:



- CPU superscalari:



Con queste tipologie di CPU le prestazioni aumentano.

Però, quando arriva l' interrupt:

- possono esserci istruzioni già prelevate ma non ancora eseguite;
- nel caso delle CPU superscalari, possono esserci istruzioni eseguite solo parzialmente.

Non approfondiremo l' argomento, limitiamoci ad avere presente che il caso che noi consideriamo è quello più semplice.

# **DISPOSITIVI DI I/O E DMA**

Dispositivi di I/O. Distinguiamo tra:

- **Controller** (detto anche **adapter**): componente elettronica che comunica con la CPU (e le altre unità) via bus.
- Dispositivo vero e proprio: componente meccanica che viene gestita dal controller. In generale, un controller può gestire più dispositivi (uguali o simili) contemporaneamente.

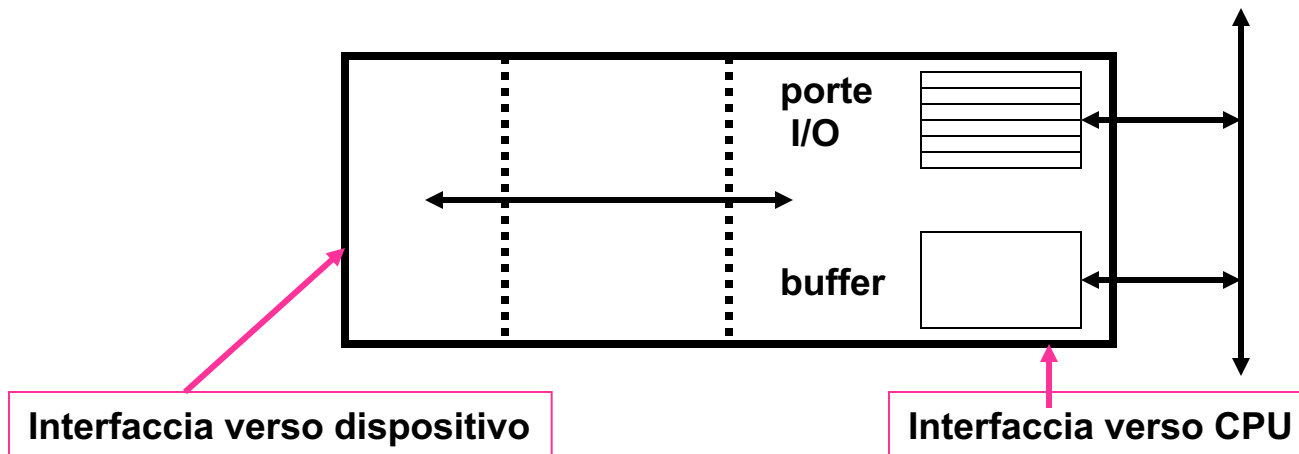
Le interfacce tra controller e dispositivo sono solitamente standard (SATA, SCSI, USB, FireWire). Pertanto si possono produrre dispositivi per una data interfaccia standard.



L'interfaccia tra CPU e controller viene usata dai programmi del S.O. che gestiscono i dispositivi (**driver**).

Questa interfaccia prevede:

- **registri di controllo**, detti anche **porte di I/O**:
  - vengono usati dalla CPU per inviare comandi al controller,
  - vengono usati dal controller per comunicare i risultati dei comandi e lo stato del dispositivo alla CPU.
- un **buffer**: serve per memorizzare dati durante le operazioni di I/O.



Abbiamo due possibili soluzioni per le comunicazioni tra la CPU e le porte di I/O:

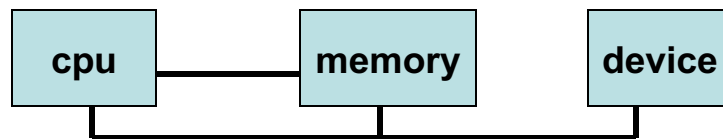
Soluzione 1: **porte di I/O gestite con istruzioni macchina ad hoc**, e.g.:

IN     R, P    il valore della porta P viene caricato nel registro R  
OUT   P, R    il valore del registro R viene caricato nella porta P

- Queste **istruzioni** devono essere **privilegiate**, per impedire che in modalità user si riesca ad accedere ai dispositivi.
- Alcune parti dei driver devono essere in assembly, perché nei linguaggi ad alto livello non si possono avere istruzioni corrispondenti.

Soluzione 2 - **Memory Mapped I/O**: ad ogni porta di I/O è assegnato un indirizzo di memoria e non servono istruzioni ad hoc:

- questi indirizzi non sono visibili dai programmi, che devono pertanto invocare il S.O. per accedere ai dispositivi;
- i driver possono essere scritti in C;
- è più complicata la gestione della cache, in quanto il contenuto di queste locazioni è modificabile dal dispositivo e non solo dalla CPU;
- tutti i moduli di memoria ed i dispositivi di I/O devono esaminare tutti i riferimenti di memoria – il tutto si complica se usiamo un bus dedicato tra CPU e Memoria, tipo:



Non indaghiamo ulteriormente.

Tre possibili soluzioni per eseguire l'I/O per conto di un programma **P**. Supponiamo di voler trasferire **n** byte verso il (buffer del controller del) dispositivo da un buffer di memoria **b**:

## Soluzione 1 – Programmed I/O:

Codice eseguito dal S.O. su richiesta di **P**:

```
for(i=0; i<n, i++){  
    while(device_status_reg != READY){ } /* busy waiting */  
    buffer = b[i]  
}
```

**buffer**: buffer I/O per scambio dati.

**device\_status\_reg**: porta I/O per testare lo stato del device.

La CPU rimane inutilizzata mentre aspetta che il dispositivo abbia finito di trattare il singolo byte (**busy waiting**). Non si sfruttano gli interrupt.

## Soluzione 2 - Interrupt Driven I/O:

Codice eseguito dal S.O. su richiesta di **P**:

```
while(device_status_reg != READY){/* busy waiting, solo all' inizio*/ }  
buffer = b[0];  
c=1; i=1;  
scheduler( ); /*P lascia la CPU, la riotterrà ad operazione conclusa*/
```

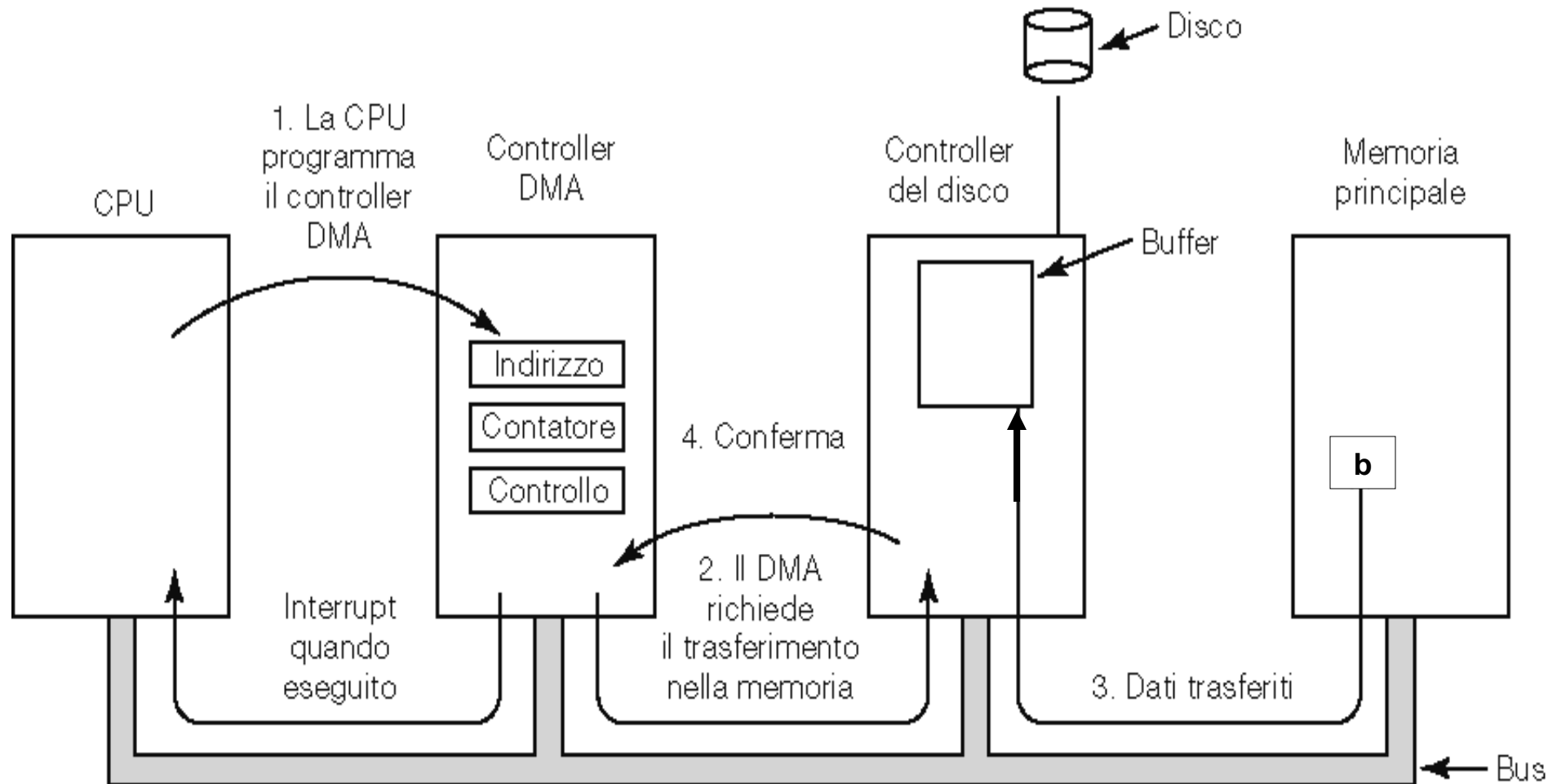
Codice eseguito dall' interrupt handler:

```
if (c==n){  
    unblock_user( ); /* il programma P può ripartire */  
}  
else{  
    buffer = b[i]; /*chiedo al device il prossimo transfer*/  
                    /*ed attendo il prossimo interrupt*/  
    i = i + 1; c = c + 1;  
}  
return_from_interrupt( )
```

Nel caso dell' interrupt driven I/O:

- Non abbiamo più busy waiting: si evita che la CPU sia impegnata solo per aspettare che il device tratti il singolo byte.
- Abbiamo però  $n$  interruzioni, una per ogni byte, che causano ovviamente overhead.

Soluzione 3: L'architettura prevede un controllore **Direct Memory Access (DMA)** capace di accedere direttamente alla memoria e di lavorare in parallelo con la CPU.



## Soluzione 3 - I/O con DMA:

Tornando all' esempio, codice eseguito dal S.O. su richiesta di P:

```
set_up_DMA_controller( ); /* imposto i registri del DMA controller */  
scheduler( ); /*P lascio la CPU, la riotterrò ad operazione conclusa*/
```

Codice eseguito dall' interrupt handler che gestisce l' interrupt inviato dal DMA per comunicare la conclusione dell' operazione:

```
unblock_user( ); /* il programma P può ripartire */  
return_from_interrupt( )
```

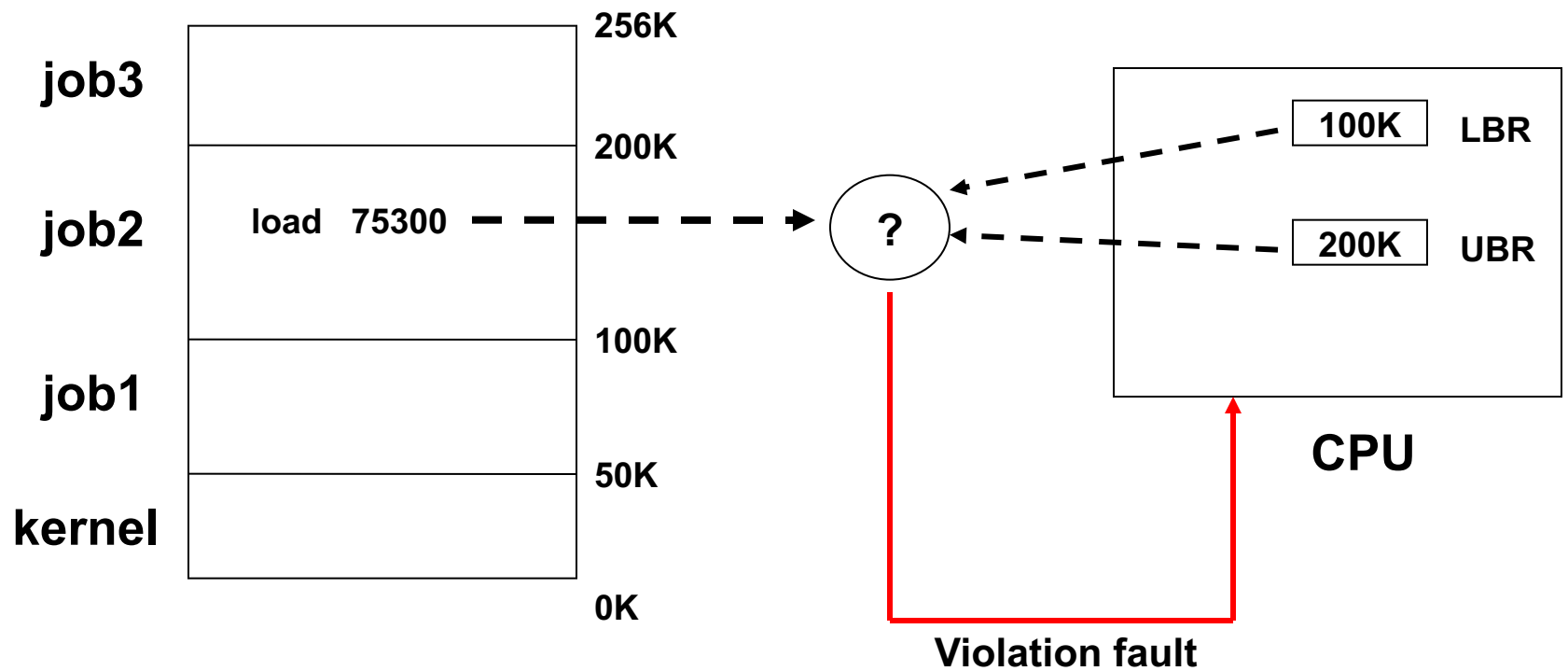
Il ciclo con il trasferimento degli **n** byte viene effettuato dal controller DMA, che, al termine del ciclo, invia un interrupt alla CPU. Il tutto mentre la CPU fa altro lavoro.



Oltre agli hardware interrupt, esistono due tipi di **program interrupt**: eccezioni e software interrupt.

**Eccezioni**, dette anche **interrupt interni**:

- eccezioni aritmetiche (divisione per zero, overflow)
  - eccezioni di indirizzamento (page fault) (Cap. 7))
  - violazione delle protezioni di memoria (violation fault, esempio nella prossima slide, approfondimenti nel Cap. 7)
- 
- Contrariamente agli hardware interrupt, non sono eventi asincroni rispetto all'esecuzione del programma.
  - Sono eventi causati da **situazioni anomale** interne al programma, che devono essere trattati dal S.O.. E' possibile che il S.O. imponga la terminazione immediata del programma.



Esempio di violation fault con uso di registri LBR/UBR.

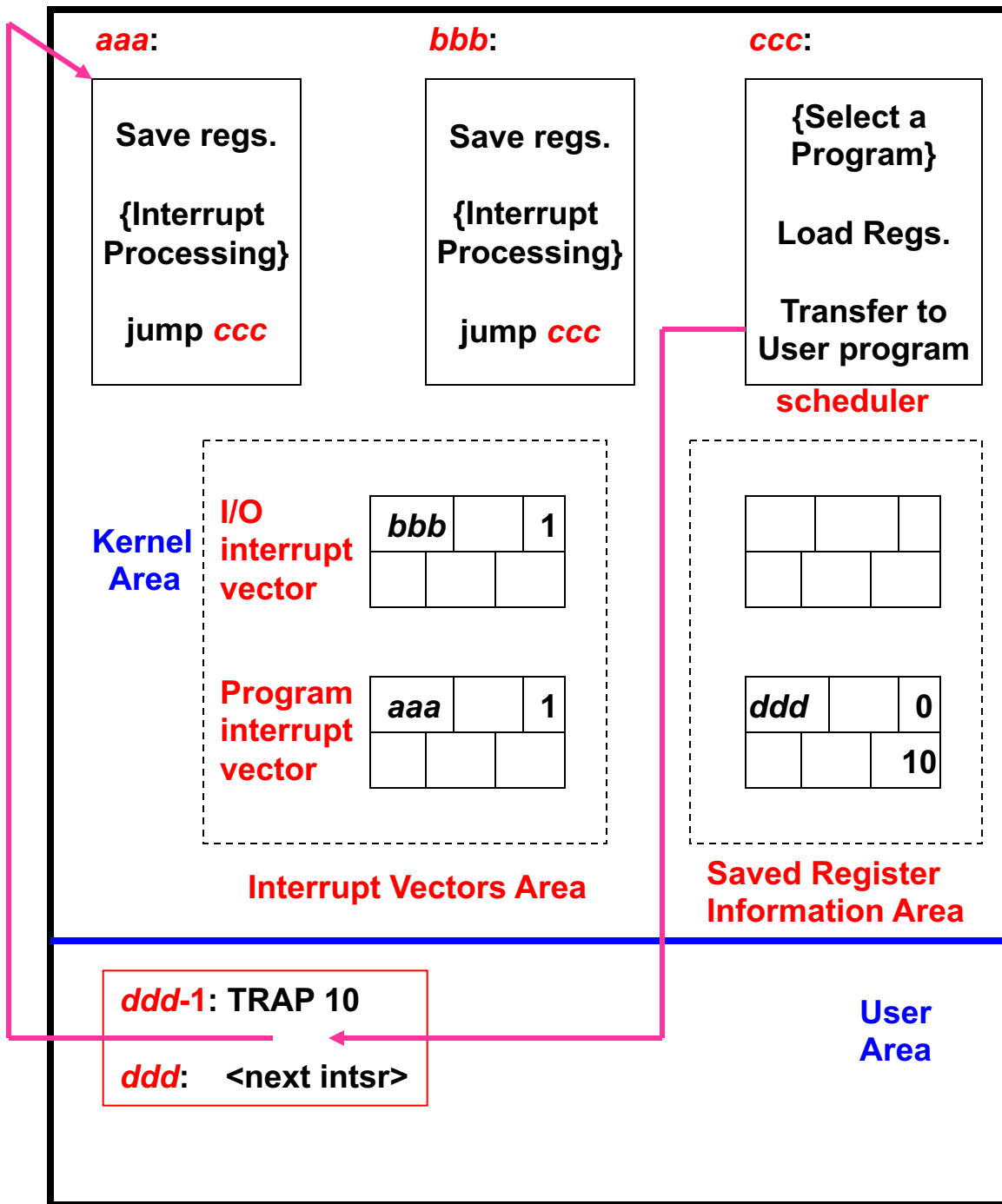
## **Software interrupt**, detti anche **trap**:

- Sono causati da un' **istruzione apposita**, solitamente chiamata **Software Interrupt Instruction**, o, semplicemente, **trap**.
- Sono usati dai programmi per chiedere esplicitamente l'intervento del S.O..
- Il linguaggio macchina prevede un'unica istruzione per chiedere tutti i possibili interventi del S.O., l'istruzione **TRAP** (ecco perché gli interrupt software vengono anche chiamati semplicemente trap).
- La TRAP ha un **parametro** che specifica il tipo di intervento richiesto. Il parametro può essere passato in 2 modi:
  - come operando della TRAP, se la TRAP prevede operandi
  - ponendolo sullo stack.

I program interrupt vengono trattati come gli hardware interrupt.  
La gerarchia di priorità tipica è aggiornata come segue:

- Machine error
- Clock Interrupt
- Disk Interrupt
- Fast device interrupt
- Slow device interrupt
- Exception
- Trap





## Esempio

L'istruzione di indirizzo **ddd-1** è una TRAP, con parametro 10.

Supponiamo che TRAP 10 serva per chiedere al S.O. la lettura del valore del clock di sistema.  
(Serve una trap perché il clock non è accessibile ai programmi in modo user).

1. salvataggio hw dei registri di controllo (**PC**=*ddd*, **PM**=0, cioè user, **IC** = 10 cioè lettura clock) nella **SRIA**;
2. settaggio dei registri di controllo sfruttando l' interrupt vector (**PC**=*aaa*, **PM**=1, cioè kernel);
3. l' interrupt handler (indirizzo *aaa*, invocato al passo 2):
  - salva il contenuto dei registri generali (vedremo dove...),
  - controlla l' **IC** per individuare il parametro (cioè 10),
  - esegue le sue funzioni (\*)
  - invoca lo scheduler, il cui codice è all' indirizzo **ccc**.
4. se lo scheduler seleziona il programma interrotto:
  - ripristina i registri generali (\*),
  - ripristina i registri di controllo (**IRET**) con i valori salvati nella **SRIA**.

(\*) Abbiamo detto che la TRAP 10 serve per leggere il valore del clock.

Domanda: dove viene memorizzato? Nel senso, come fa il programma che ha eseguito la TRAP a ritrovarsi questo valore?

Risposta: Il programma che esegue la TRAP si attende il risultato in uno dei registri generali, tipicamente il registro numero 0.

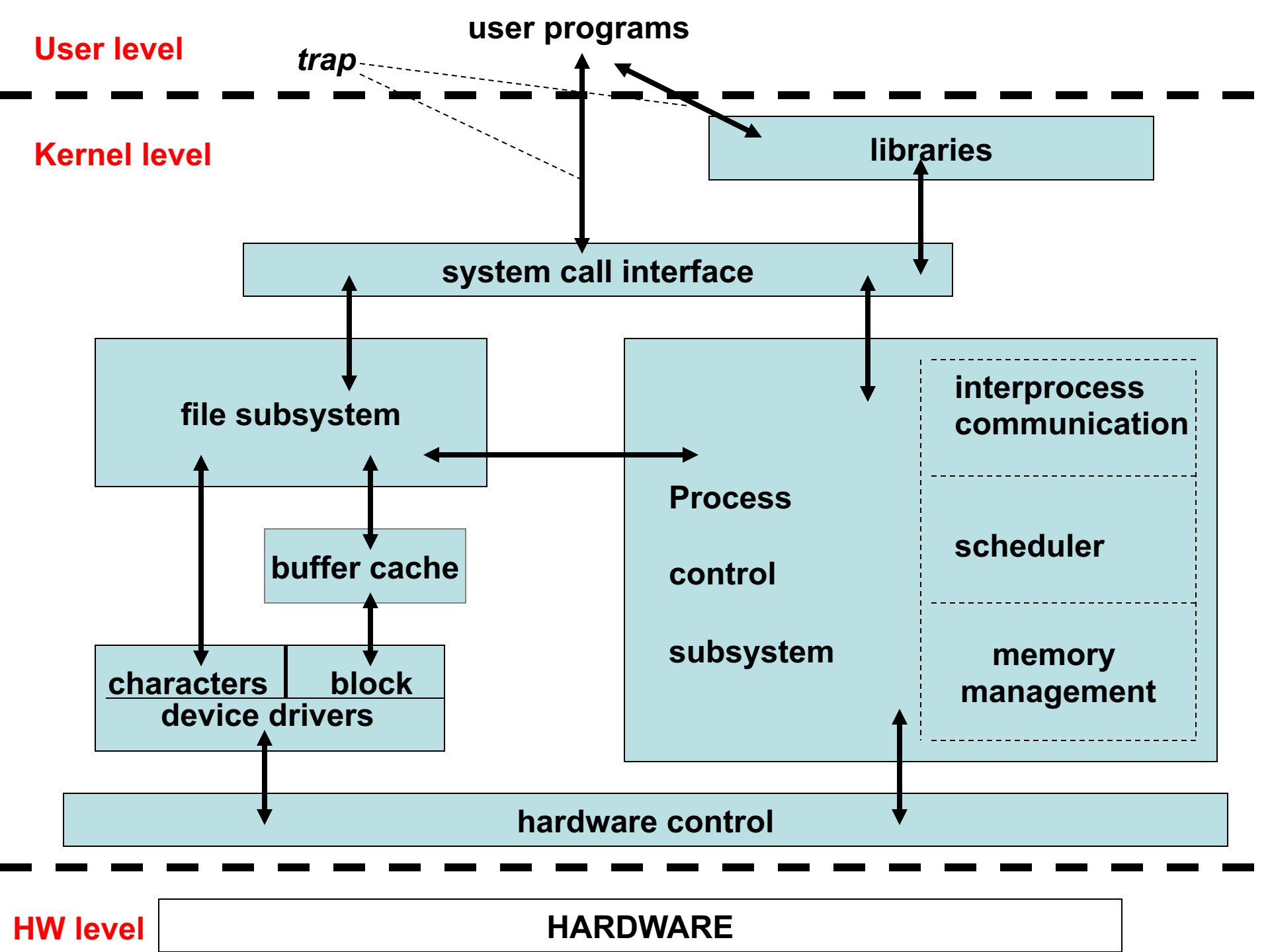
Ovviamente questo valore deve essere nel registro 0 dopo che il programma ha ripreso il controllo, quindi l'interrupt handler va nella zona di memoria dove il valore di tale registro è stato salvato e lo cambia, memorizzando il valore del clock.

Ricordiamo che nel Cap. 4 spiegheremo dove i registri generali vengono salvati.

Definizione: Una **system call** è una **richiesta al S.O.** effettuata da un programma.

- Le system call sono lo strumento messo a disposizione dei programmi per avvalersi dei servizi offerti dal S.O..
- Vengono **realizzate con la TRAP**. Abbiamo una system call per ogni parametro valido per la TRAP. Se i bit dell' **IC** sono **n**, abbiamo  $\leq 2^n$  system call. Tipicamente, **n**  $\geq 8$ .
- Il programma che invoca la system call ottiene un risultato, solitamente memorizzato in un registro.
- Rivedendo lo schema di UNIX System V già visto nel Cap. 1 e richiamato nella slide seguente, si nota come le system call siano l' interfaccia tra i programmi utente ed il S.O.:





Quante e quali system call abbiamo? Dipende dal S.O..

In UNIX/Linux abbiamo circa 100 system call. Alcune servono per interagire con il file subsystem, che gestisce file e dispositivi (in UNIX hanno la medesima interfaccia). I nomi di queste system call ne suggeriscono il significato. Esempi:

- create**, per creare un file,
- open**, per creare un accesso ad un file/device,
- read**, per leggere ***n*** byte da un file/device (***n*** è un parametro),
- write**, per scrivere ***n*** byte in un file/device (***n*** è un parametro),
- close**, per invalidare un accesso ad un file/device.

Altre system call servono per interagire con il Process Control Subsystem, e le vedremo nel Cap. 4.

Come si fa ad **invocare una system call**? La figura a Pag. 49 suggerisce due possibilità:

1. Se programmiamo **in assembly** usiamo **l'istruzione TRAP**.

Le **system call** hanno **parametri** (esempio, il numero di byte ed il nome del file per read/write), che devono essere passati opportunamente.

Non è facile, perché bisogna sapere dove l' interrupt handler della TRAP va a cercarli.

Ciò vale per la TRAP 1, TRAP 2, TRAP 3, ..... cioè per tutte le system call.

N.B. non dobbiamo confondere i parametri della system call con il parametro dell' istruzione TRAP, che serve per individuare la system call.

2. Se programmiamo **in C abbiamo una libreria di funzioni associate alle system call.**

Le funzioni di libreria sono in assembly e sono invocabili dal programma in C.

Le funzioni di libreria invocano la TRAP.

In questo caso i parametri della system call possono essere passati alle funzioni di libreria come normali parametri di funzione.

Il programmatore può ragionare come se la system call fosse una normale chiamata di funzione.

## Esempio di programma in C per UNIX/Linux

```
#include <fcntl.h>
main( ){
    int fd; /* fd è un intero */
    char buf[20], buff[30]; /* buf e buff sono array di caratteri */
    fd = open("fileDiProva",O_RDONLY); /* funzione di libreria */
    read(fd,buf,20); /* funzione di libreria */
    read(fd,buff,30); /* funzione di libreria */
}
```

- La **open** è una UNIX system call con due parametri: nome del file da “aprire” e tipo di apertura.

La **open** restituisce un “file descriptor”, cioè un accesso al file, che è un intero – non indaghiamo ulteriormente.

- La funzione di libreria **open** ha lo stesso nome della system call, è invocabile dal programma in C specificando i parametri ("fileDiProva" e O\_RDONLY, cioè apertura solo in lettura, nell' esempio).

La funzione **open** invoca la system call **open**, cioè invoca la TRAP con il parametro che corrisponde alla **open**.

```
#include <fcntl.h>
main( ){
    int fd;
    char buf[20], buff[30];
    fd = open("fileDiProva",O_RDONLY);
    read(fd,buf,20); /* leggo 20 byte */
    read(fd,buff,30; /* leggo altri 30 byte */
}
```

- La **read** è una UNIX system call con 3 parametri: nome del file, buffer di memoria su cui scrivere, numero di byte da leggere.
- la funzione di libreria **read** ha lo stesso nome della system call, è invocabile dal programma in C specificando i parametri ("fileDiProva", buf e 20 nella prima chiamata, "fileDiProva", buff e 30 nella seconda).

La funzione **read** invoca la system call **read**, cioè invoca la TRAP con il parametro che corrisponde alla **read**.

Per chi vuol familiarizzare con il C:

- **#include <fcntl.h>** serve per “importare” le definizioni del **file header fcntl.h**, dove sono definite alcune costanti, quali la costante **O\_RDONLY** usata nella **open**.
- Il programma di Pag. 53 usa le funzioni di libreria **open** e **read**, che rispettano lo **standard POSIX**: il programma può girare su qualsiasi sistema UNIX/Linux che rispetti lo standard, perché certamente il sistema ha queste funzioni di libreria.
- Il programma non gira su Windows o altri sistemi che non supportano le funzioni di libreria **open** e **read**.
- Ma allora i programmi in C che usano i file non sono portabili? Lo sono, però devo usare le funzioni della libreria quali **fopen**, **fread**, .... Queste funzioni hanno un numero e tipo di parametri diversi rispetto ad **open** e **read**. Nel caso UNIX/Linux, la **fopen** chiama la **open**, la **fread** chiama la **read**.....

Mentre in UNIX/Linux di norma abbiamo una funzione di libreria per ogni system call, in Windows non è così.

Lo standard **APIWin32** definisce alcune migliaia di procedure che i programmi dovrebbero usare per invocare i servizi del S.O. (Il ruolo delle funzioni di libreria).

Le system call sono però molte meno, alcune procedure non invocano le system call ma vengono eseguite in modo user.

Esempi di procedure APIWin32:

**CreateFile**, per creare o aprire file,

**ReadFile**, per leggere da un file,

**WriteFile**, per scrivere su un file,

**CloseHandle**, per chiudere un file.