CAPITOLO 4

PROCESSI

4.1: Memory layout dei programmi
4.2: Il concetto di processo
4.3: Parallelismo e concorrenza
4.4: Stati di un processo
4.5: Implementazione dei processi – process control block
4.6: Implementazione dei processi – context switch
4.7: Creazione e terminazione di processi
4.8: System call in UNIX

4.1: MEMORY LAYOUT DEI PROGRAMMI

L'esecuzione di un programma necessita di tre aree di memoria, che in Unix si chiamano tradizionalmente **regioni**:

- **Area di testo**. Contiene il **testo del programma**, in linguaggio macchina. Non è modificabile dal programma.

- **Area dati**. Contiene le **variabili globali** del programma, cioè le variabili condivise da tutte le procedure.
  - Contenuto variabile – assegnamenti alle variabili.
  - Dimensione variabile – alcune strutture dati sono dinamiche.

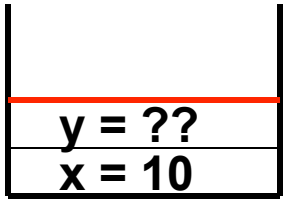
- **Area di stack**. Contiene i **record di attivazione** (frame) **delle procedure** già chiamate ma non ancora terminate.
  - Contenuto variabile - assegnamenti alle variabili.
  - Dimensione variabile - chiamata/terminazione di procedure.

Per le strutture dinamiche delle procedure si usa una parte dell'area dati, detta **area heap**, cui le variabili fanno riferimento.

```
#include <stdio.h>
int a=5; /*variabile globale - area dati*/
int b=6; /*variabile globale - area dati*/

main(  ){
    int x = 10;   /*variabile locale*/
      /*(punto 1)*/
    int y = f1(x,b); /*variabile locale*/
      /*(punto 5)*/
    printf("ecco il valore di y: %d\n",y);
}

int f1(int s, int t){
    int u = a+s+t; /*variabile locale*/
      /*(punto 2)*/
    int v = f2(u); /*variabile locale*/
      /*(punto 4)*/
    return v;
}

int f2 (int h){
    int k = h+15; /*variabile locale*/
      /*(punto 3)*/
    return k;
}
```

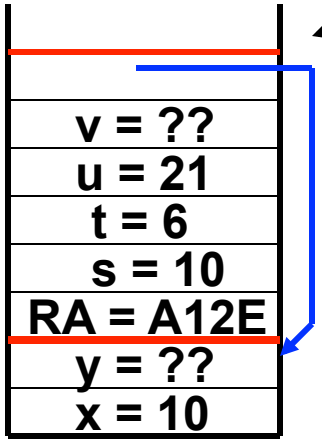
Esempio di programma in C. Nella prossima slide vedremo la foto dello stack quando il programma si trova nei punti segnati in rosa.
Ogni frame contiene, oltre a componenti che trascuriamo:
- Parametri attuali,
- Variabili locali,
- Return Address (RA),
- Pointer al frame della procedura chiamante.

Assunzioni: il programma compilato è tale che:
- l'istruzione (macchina) di main che segue la chiamata a f1 si trova all'indirizzo A12E,
- l'istruzione (macchina) di f1 che segue la chiamata a f2 si trova all'indirizzo A39F.



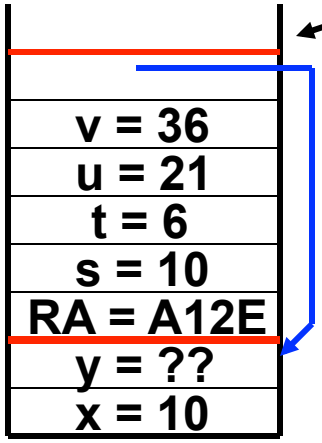
stack top

(1)



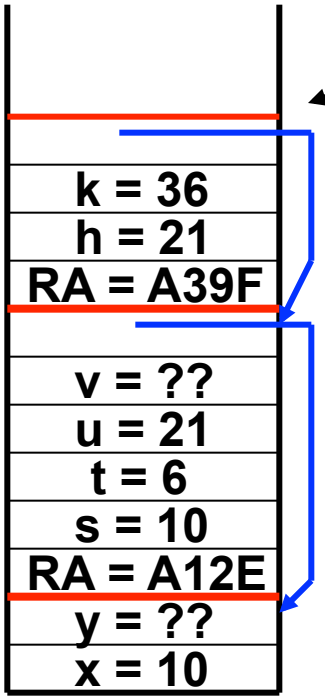
stack top

(2)



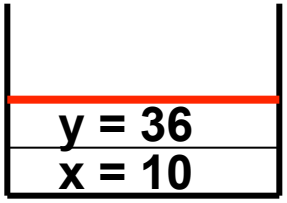
stack top

(4)



stack top

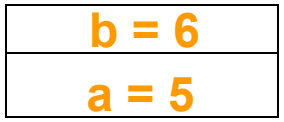
(3)



stack top

(5)

Area dati:



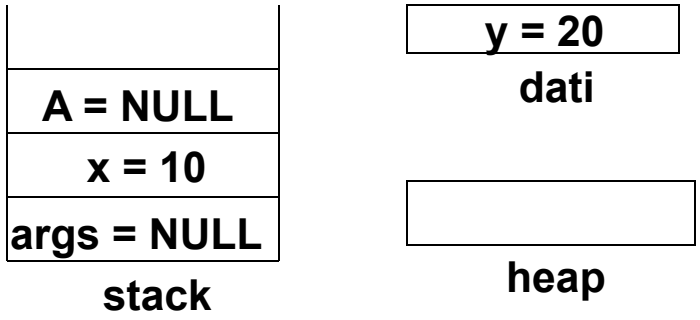
N.B. il valore delle variabili non inizializzate è indicato con "??". In realtà un valore di default viene assegnato, dipende dal linguaggio.

Assumiamo architettura a 16 bit: gli indirizzi sono rappresentabili con 4 cifre esadecimali.
Esempio:
A12E = 1010 0001 0010 1110.

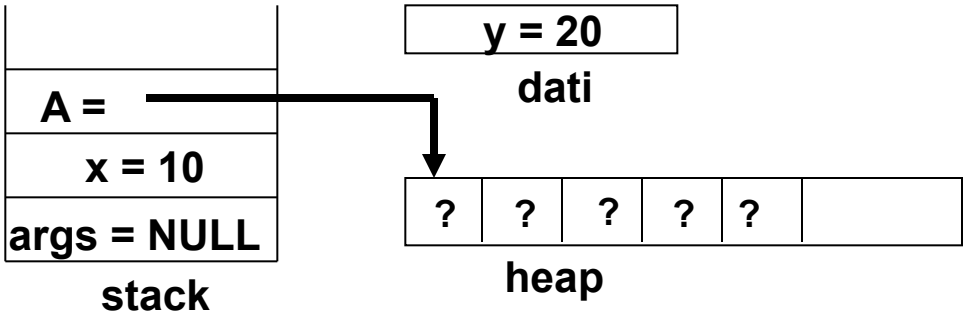
```
class Cl{
   int y = 20;
   public static void main(String[] args){
      int x = 10;
      int[ ] A;
        (punto 1)
      A = new int[5];
        (punto 2)
      for (int i=0,i<5,i++){
         A[i] = y+x+i;
      }
        (punto 3)
   }
}
```

Esempio di programma in Java. Nella prossima slide vedremo il contenuto dello stack e dello heap quando il programma si trova nei punti contrassegnati in rosa. Assumiamo di non assegnare valore iniziale ad args.

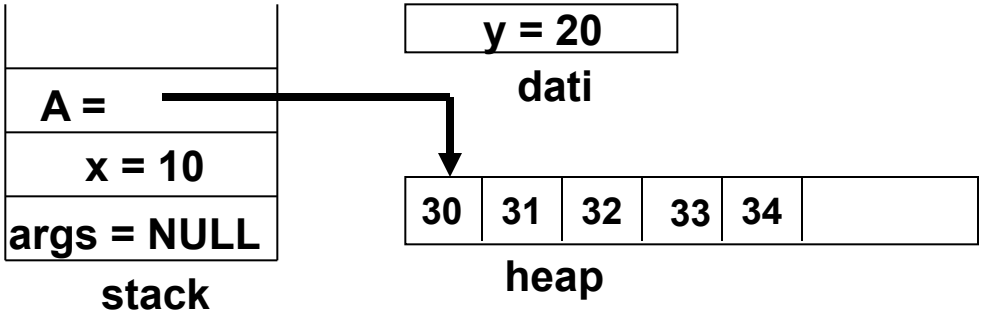
Ricordiamo che in Java gli array sono, di fatto, pointers.



(1): nessun assegnamento ad A. Il pointer è nullo.

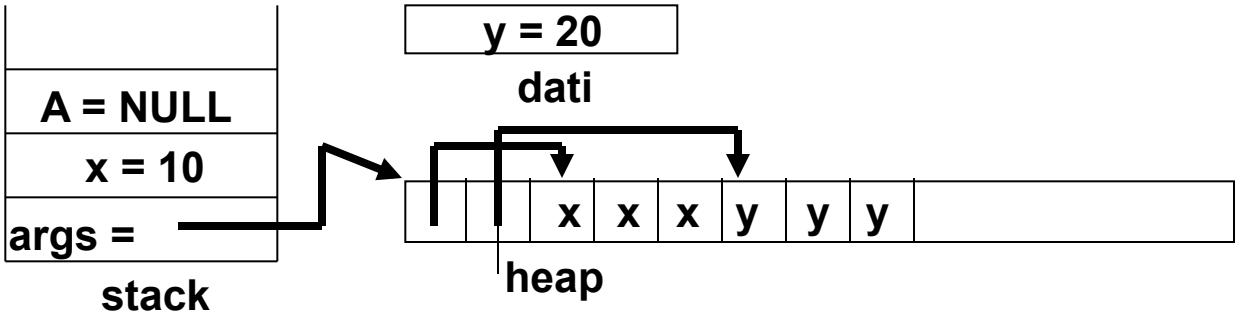


(2): A ha un valore, gli interi in A non ancora (in realtà hanno il valore di default assegnato agli interi da Java).

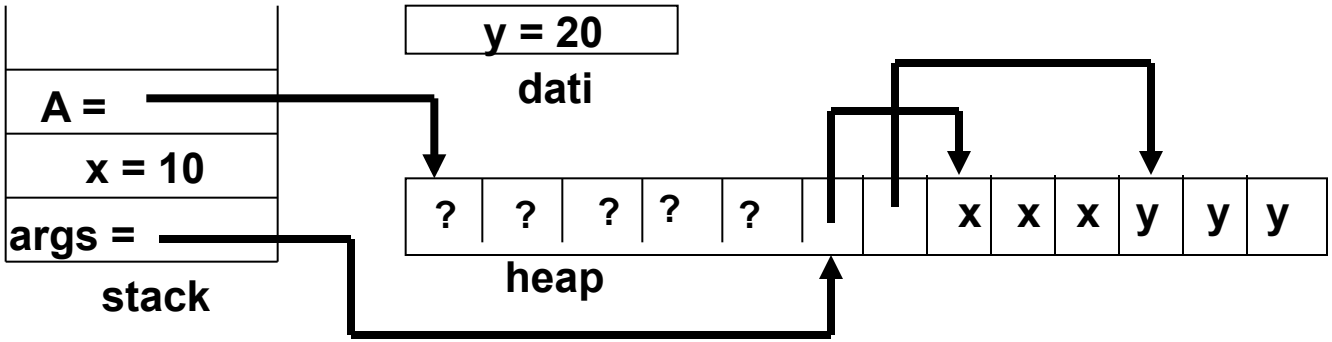


(3): Gli interi in A hanno i valori assegnati nel for.

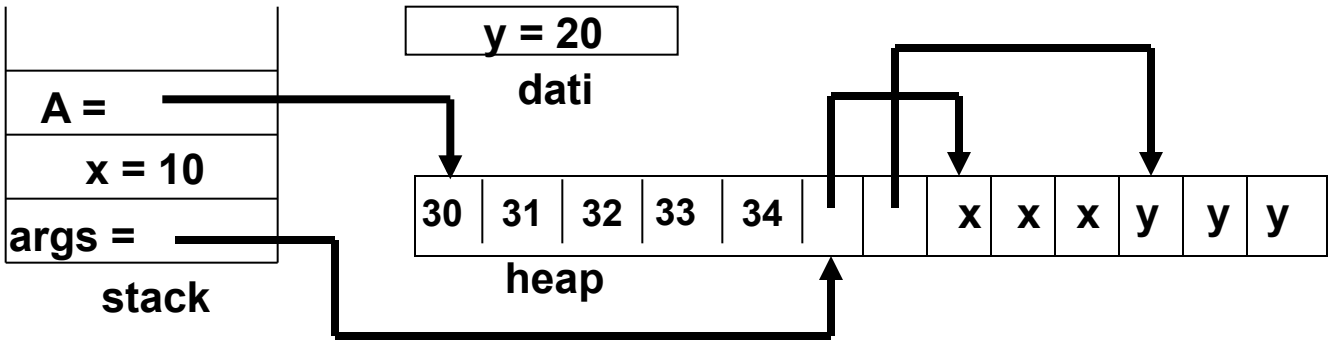
- E se chiamassi il programma con parametri xxx,yyy?



(1)



(2)



(3)

Osservazione: il codice di un programma contiene indirizzi di memoria che si riferiscono:

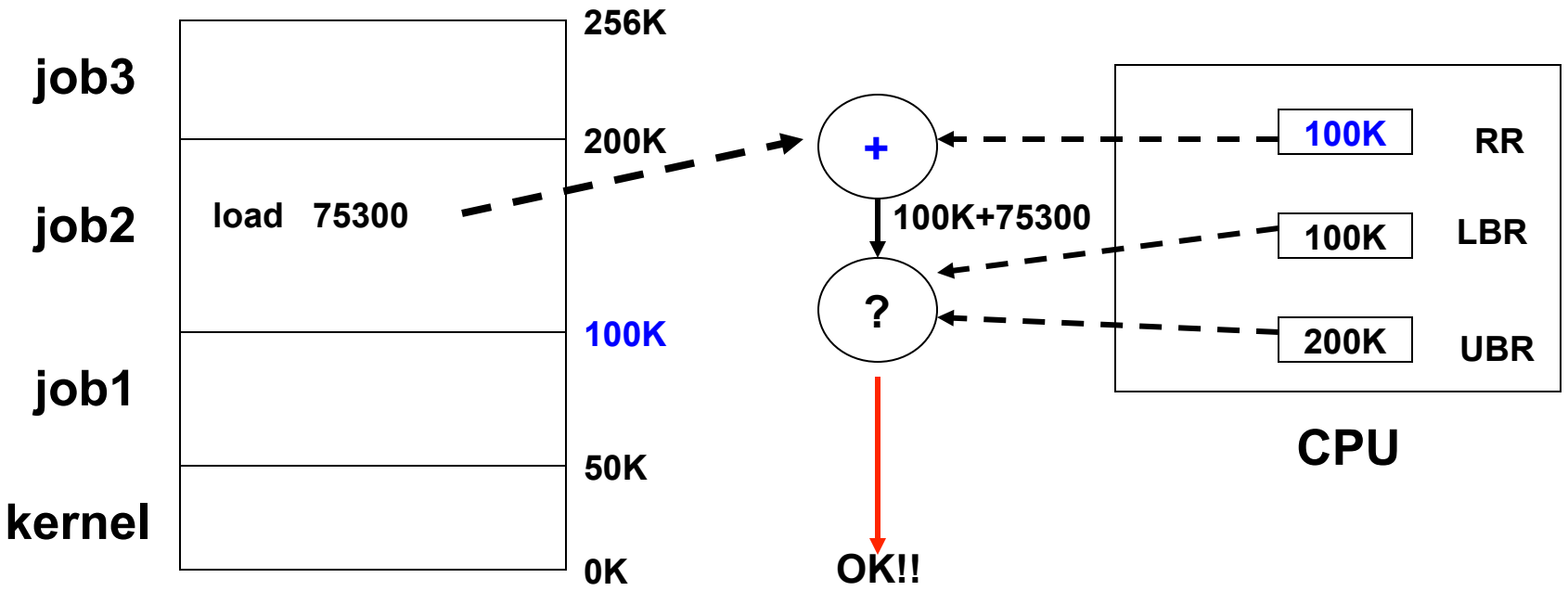
- all'area di testo (e.g. istruzioni jump);
- all'area dati (e.g. variabili globali);
- all'area di stack (e.g. variabili/parametri delle procedure).

Importante: questi non sono indirizzi reali, ma **indirizzi virtuali** che sono tradotti in indirizzi reali dalla MMU a tempo di esecuzione.

Pertanto le tre aree testo, dati e stack possono essere caricate in qualsiasi luogo della memoria, a patto che la MMU venga impostata in modo corretto.

Questi argomenti verranno trattati nel Cap. 7 (gestione della memoria). Al momento ci limitiamo a fare un esempio banale.

Esempio semplice di memoria virtuale: **Relocation Register (RR)**. Gli indirizzi virtuali di ogni job partono dall'indirizzo 0. Ogni job viene caricato in un'area di memoria contigua, a partire da un indirizzo **a**. Il RR assume valore **a**. Il valore di **a** viene sommato agli indirizzi virtuali per ottenere gli indirizzi reali. La MMU in questo caso comprende il RR ed il modulo per la somma. Mostriamo anche la protezione con i registri LBR/ UBR. NB: Si potrebbe usare LBR come RR.



E se il codice contenesse indirizzi reali?

Non servirebbe la MMU, però:

- Non potremmo caricare in memoria due programmi con overlapping di memoria, in particolare non potremmo caricare più istanze del medesimo programma.

- I compilatori dovrebbero generare indirizzi evitando gli overlapping: praticamente impossibile su computer general purpose dove vogliamo eseguire grosse quantità di programmi, magari scaricabili dalla rete.

- I programmi non sarebbero portabili: impensabile in un mondo in cui i programmi vengono scaricati dal web.

4.2: IL CONCETTO DI PROCESSO

Nei sistemi operativi gioca un ruolo essenziale il concetto di **processo**. Vediamo 2 definizioni di processo, equivalenti:

Definizione: **un processo è un'esecuzione di un programma** (M.J. Bach).

Definizione: **un processo è un'istanza di un programma in esecuzione** (D.M. Dhamdhere, A.S. Tanenbaum).

Possiamo avere **più processi relativi al medesimo programma**, in quanto possiamo avere più esecuzioni, anche "in contemporanea", del programma.

Ognuna di queste esecuzioni non deve interferire con le altre, anche se si sovrappongono dal punto di vista temporale, come può accadere nei sistemi con multiprogrammazione.

Ciò significa, per esempio, che ogni processo può leggere/ modificare solo le proprie aree di memoria.

Osservazioni:

- Il programma è un'entità passiva che non esegue nessuna azione di per sé.
- L'esecuzione del programma, chiamata processo, concretizza le azioni specificate nel programma.
- **Le entità che vengono schedulate dal S.O. sono pertanto i processi, non i programmi**.
- Nei S.O. con multiprogrammazione, più processi possono coesistere ed eseguire "contemporaneamente".

Altre osservazioni:

- Le **aree di testo, dati e stack** si riferiscono al programma in esecuzione, quindi **sono associate al processo**, non al programma.

  Mandando in esecuzione un programma 18 volte, avremo 18 processi, ognuno con le proprie aree di memoria.

- Anche le **risorse** logiche, come i file, e le risorse fisiche, come i dispositivi (tastiera, video, scheda di rete,…), sono **associate al processo**.

- Anche lo **stato della CPU**, cioè il contenuto dei registri di controllo e dei registri generali, è relativo all'esecuzione del programma, quindi è **associato al processo**.

Riassumendo, possiamo fornire la seguente definizione.

Definizione: Un **processo** consiste delle seguenti componenti:

- stato della CPU;
- area di testo;
- area dati;
- area di stack;
- risorse logiche e fisiche assegnate al processo.

Deviamo dal filo del discorso per 4 slide, al fine di mostrare quel che intendiamo per “processo che dispone di un file”, tenendo presente che:

- l’argomento non verrà trattato quindi stiamo guardando un argomento che non fa parte del programma ufficiale del corso e non serve a capire quanto seguirà;

- usiamo un esempio in **C**, dove, contrariamente all’esempio visto alla fine del Cap. 3, non usiamo le funzioni di libreria **fnctl.h**, ma le funzioni della **stdio.h**. Ciò consente al programma di essere portabile (gira in UNIX e anche in Windows).

```c
#include <stdio.h>

int main( ){

   char stringa[20];
   FILE *str1, *str2;  /*accessi a file: variabili di tipo pointer a FILE*/

   str1 = fopen("prova.txt","wt");  /*accesso read/write a prova.txt*/
   for(int i=0; i<5; i++){
        printf("dammi una parola: "); /*stampa a video*/
        gets(stringa);  /*lettura dati da tastiera*/
        fwrite(stringa, sizeof(stringa), 1, str1); /*scrittura su prova.txt*/
   }
   fclose(str1);  /*accesso al file non più valido*/

   str2 = fopen("prova.txt","r");  /*accesso solo read a prova.txt /
   for(int i=0; i<5; i++){
         fread(&stringa, sizeof(stringa), 1, str2);  /*lettura da prova.txt*/
         printf("parola numero %d: %s\n", i, stringa);  /*stampa a video*/
   }
   fclose(str2); /*accesso al file non più valido*/

}
```

- Confrontando l'esempio con quello del Cap. 3, si nota che i tipi dei parametri della **open** e della **fopen** sono diversi, mentre la **read** e la **fread** si differenziano anche per il numero dei parametri.

- Usando **stdio.h**, gli accessi ai file sono **puntatori al tipo FILE**, e non interi.

- I processi relativi a questo programma avranno i propri accessi al file **prova.txt** (**str1**, accesso in lettura/scrittura, e **str2**, accesso in sola lettura), il proprio accesso alla tastiera per ottenere dati ed il proprio accesso al video per mostrare messaggi. Gli accessi a video/tastiera sono aperti di default.

Altre precisazioni:

- Se il programma precedente gira su UNIX, avremo due file descriptor in corrispondenza di **str1** e **str2**;

- In UNIX i dispositivi vengono visti come file, quindi anche per i dispositivi abbiamo i file descriptor.

  Pertanto se il programma gira su UNIX abbiamo i file descriptor anche per la tastiera ed il video.

  In verità ogni processo dispone di default dei file descriptor per la tastiera, detto **standard input**, e per il video, detto **standard output**.

4.3: PARALLELISMO E CONCORRENZA

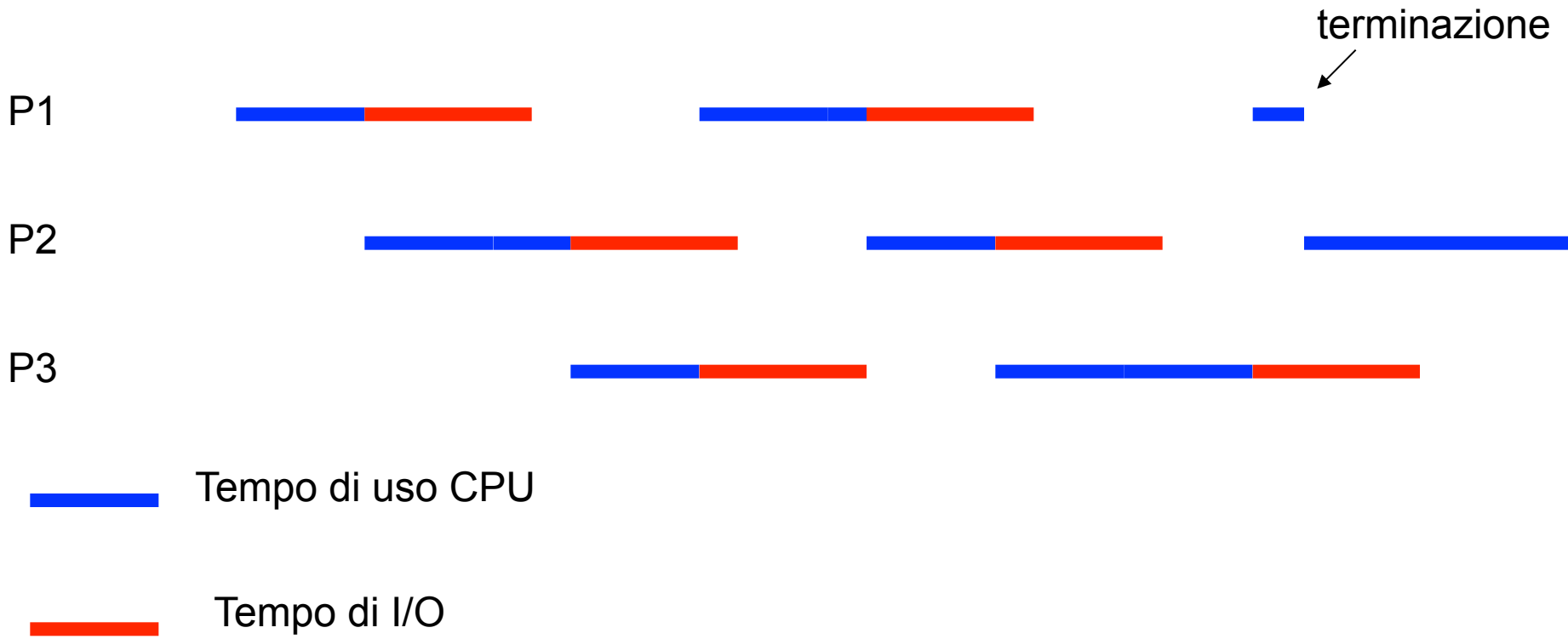
Definizione: **Due eventi sono paralleli se occorrono nello stesso momento**. Due attività sono parallele se vengono portate avanti contemporaneamente.

Definizione: **La concorrenza è l'illusione del parallelismo**. Due attività sono concorrenti se si ha l'illusione che vengano eseguite in parallelo, mentre, in realtà, in ogni singolo istante solo una di esse viene eseguita.

Assumiamo che l'hardware offra una sola CPU ed un DMA controller.

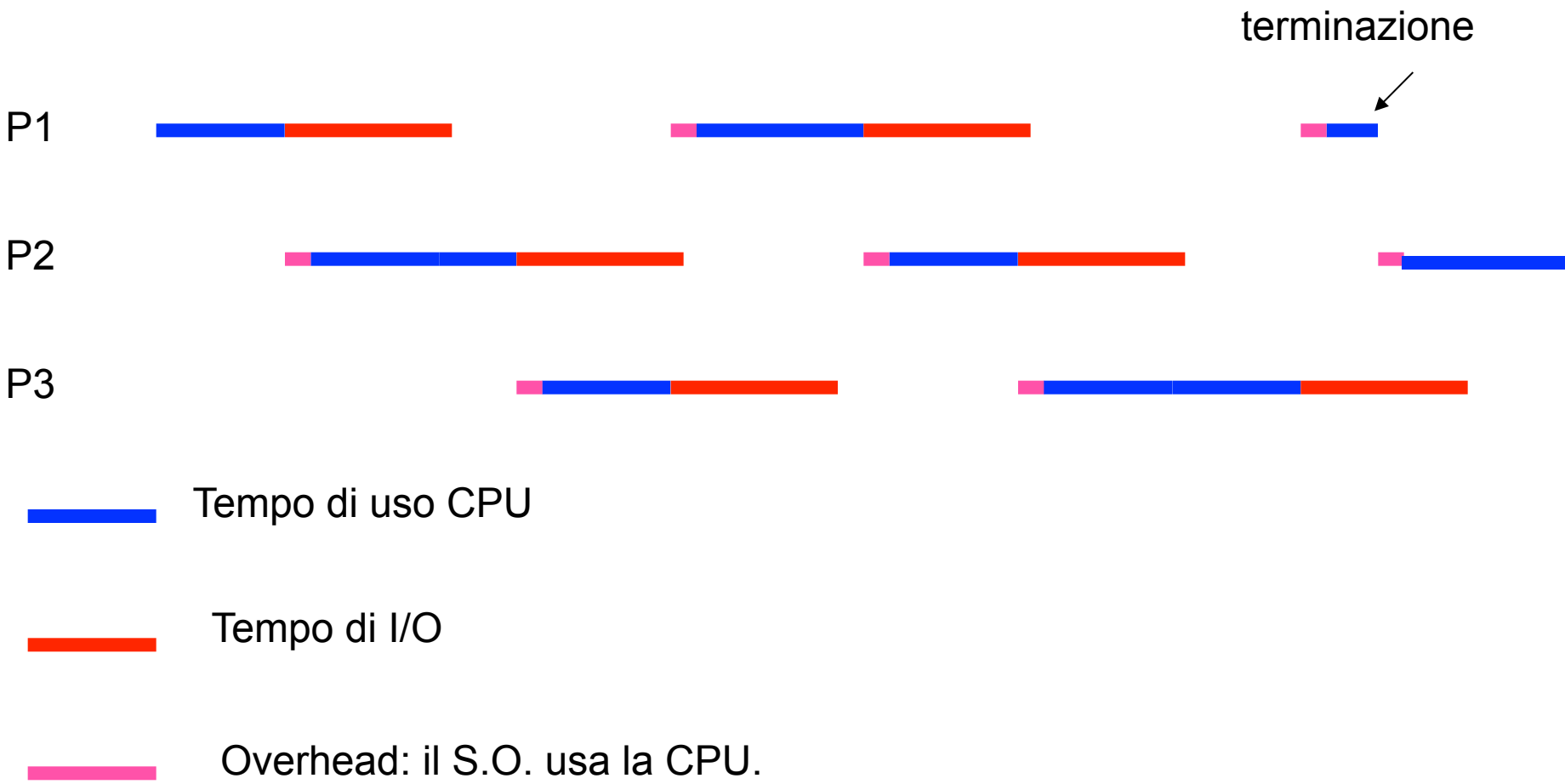
Assegnando la CPU "a turno" ai vari processi (si parla di **interleaving**), il S.O. realizza **l'esecuzione concorrente** di tali processi. Il parallelismo è solo simulato, in quanto in ogni istante al più un solo processo può usare la CPU.

Esempio di esecuzione concorrente di 3 processi.

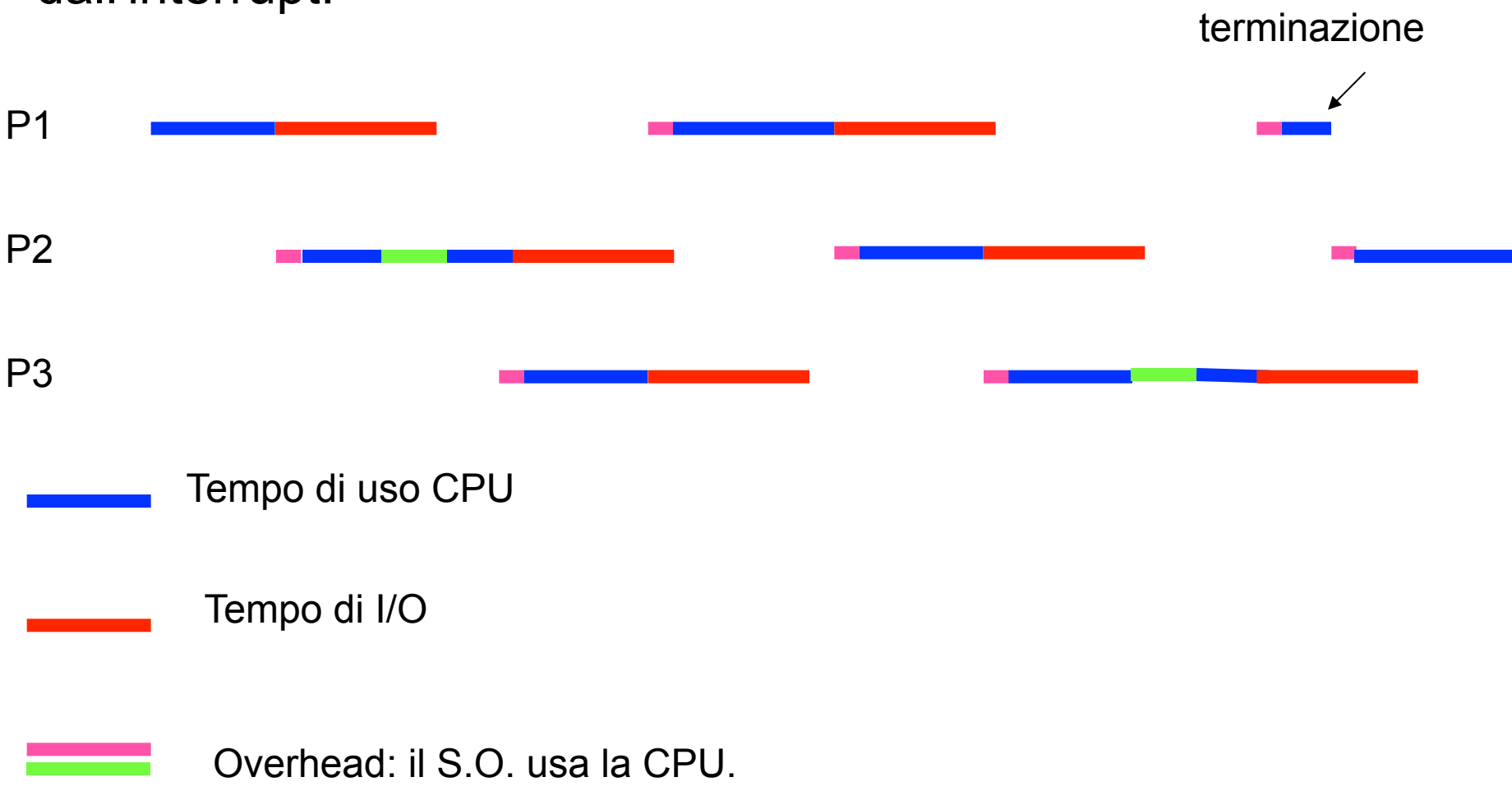


In ogni istante, solo un processo dispone della CPU. Gli altri processi fanno operazioni di I/O, oppure non fanno nulla.

Siamo stati imprecisi: quando il S.O. sottrae la CPU ad un processo e l'assegna ad un altro, lo fa usando la CPU, che non lavora per nessun processo. Dobbiamo pertanto considerare l'overhead (già noto dal Cap. 3):



Siamo stati ancora imprecisi. L'overhead della slide precedente è sempre conseguente ad un interrupt. Può capitare che lo scheduler, invocato dall'interrupt handler, scheduli lo stesso processo che era stato interrotto dall'interrupt:



Osservazioni:

- **La velocità di avanzamento di un processo non è uniforme nel tempo**, in quanto in alcuni momenti dispone della CPU ed in altri no.

- La **velocità di avanzamento** di un processo dipende dal numero di processi presenti e dal loro comportamento (CPU-bound, I/O-bound). Pertanto **non è riproducibile**.

- Anche la velocità di avanzamento relativa dei processi (cioè come avanzano l'uno rispetto all'altro) non è riproducibile.

Ciò non sorprende neanche chi non conosce la teoria dei S.O.: se sul PC intendo guardare un film su un DVD, di solito evito di masterizzare un CD, scaricare file dalla rete, scaricare filmati da un hard disk esterno, trasferire dati da una USB key ad un'altra,…..

E se avessimo più CPU? Distingueremmo due casi:

- **Multiprocessing**: più processi possono eseguire su una macchina dotata di più CPU.
- **Distributive Processing**: più processi possono eseguire su più macchine distribuite ed indipendenti.

In questi casi il parallelismo può essere reale, nel senso che più processi possono eseguire in contemporanea su CPU diverse.

Comunque il numero di processi può essere maggiore del numero di CPU: abbiamo ugualmente concorrenza.

I sistemi risultano complessivamente più efficienti, ma il S.O. deve essere più complesso. **NON** affronteremo queste problematiche.

4.4: STATI DI UN PROCESSO

In ogni istante, ogni processo gestito dal S.O. si trova in un certo **stato**. Lo stato è un indicatore dell'attività che sta svolgendo il processo.

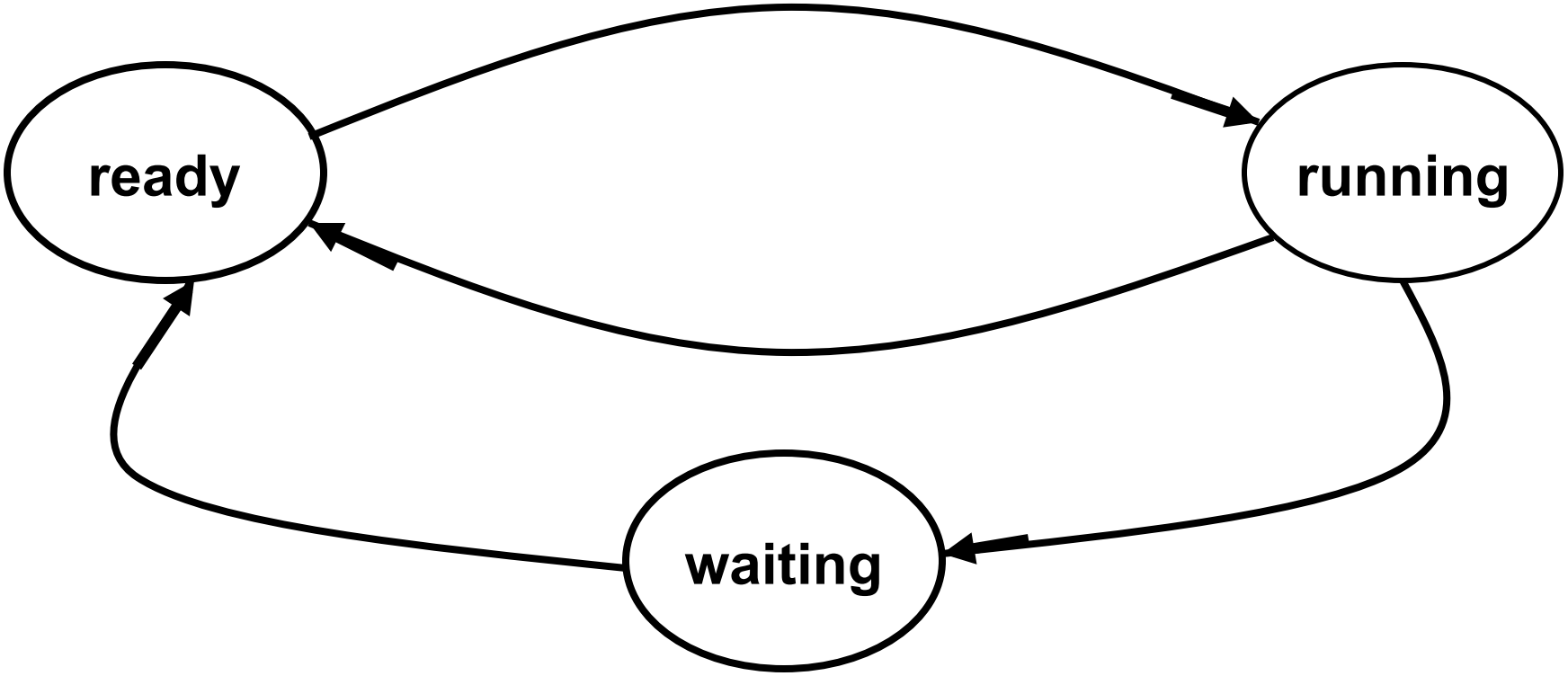
Possiamo distinguere almeno tre stati:

- **running**: il processo è in esecuzione, nel senso che la CPU sta eseguendo il programma associato al processo;

- **ready**: il processo non è in esecuzione, ma sarebbe in grado di eseguire se ottenesse la CPU. In pratica, il processo è costretto a rimanere inattivo in quanto il numero di processi è maggiore del numero di CPU;

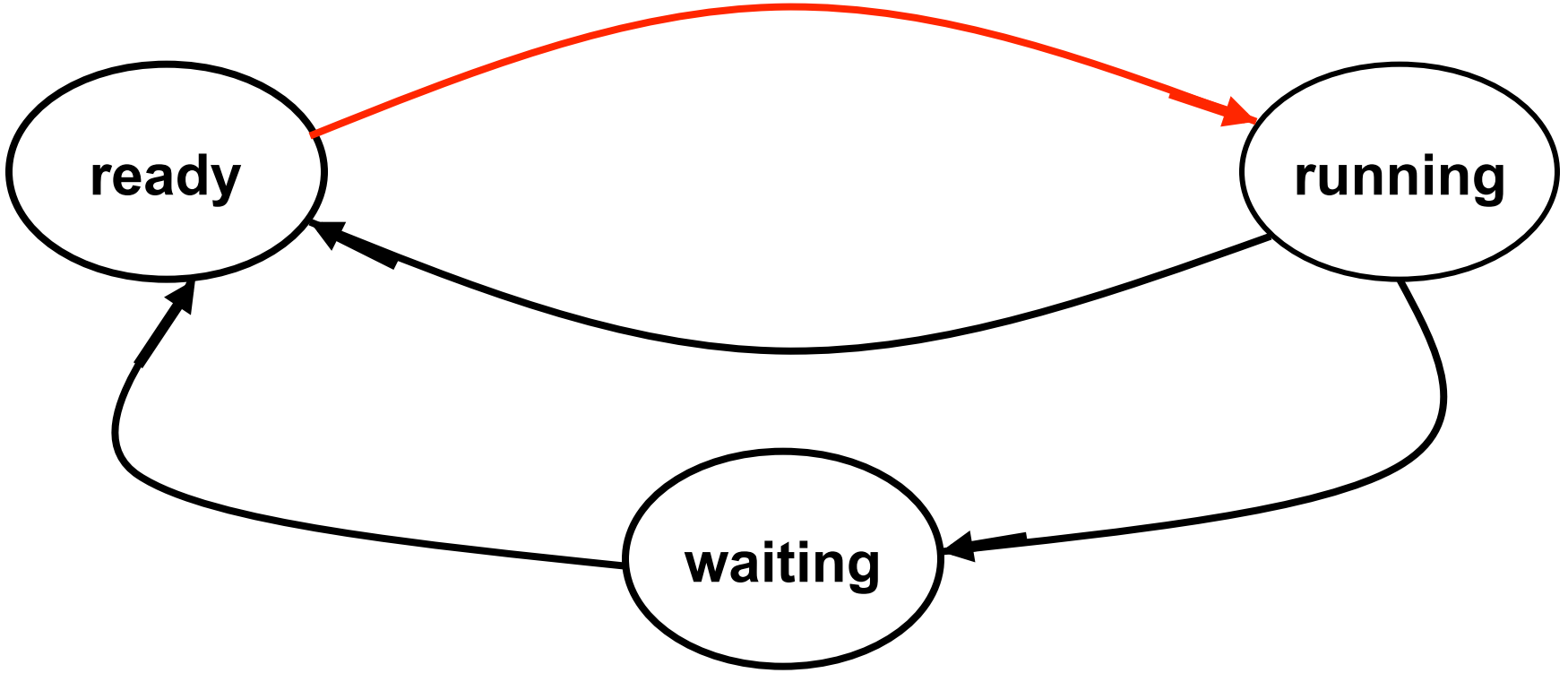
- **waiting**: il processo non è in esecuzione, non sarebbe in grado di eseguire se ottenesse la CPU, ed acquisirà la capacità di eseguire se e quando si verificherà un evento, che il processo sta pertanto attendendo.

  Esempio: il processo è in attesa che si completi un'operazione di I/O, quale una scrittura su disco.



Abbiamo 4 possibili transizioni da stato a stato, che sono determinate da eventi che avvengono nel sistema.

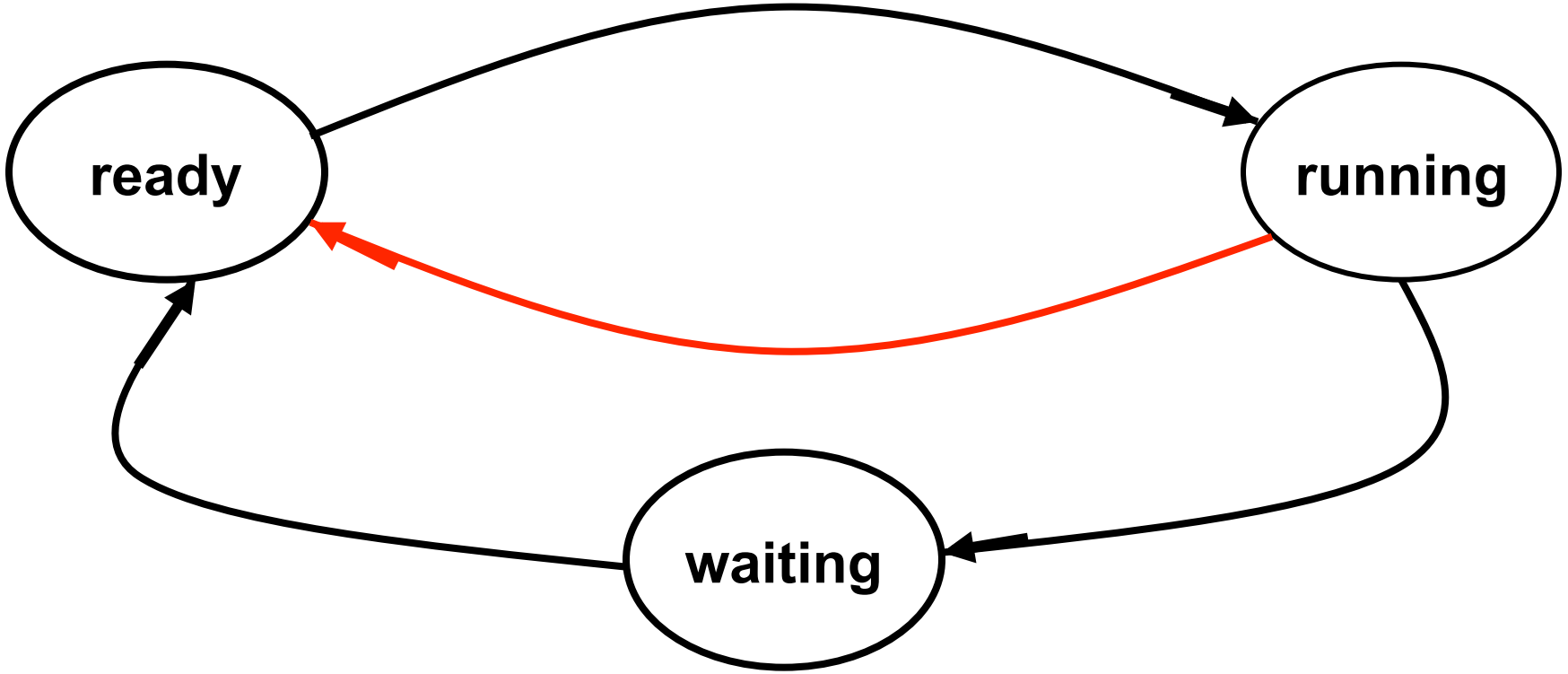
Andiamo ad esaminarle una ad una.



Caso **ready → running**: **lo scheduler seleziona il processo** per l'esecuzione.

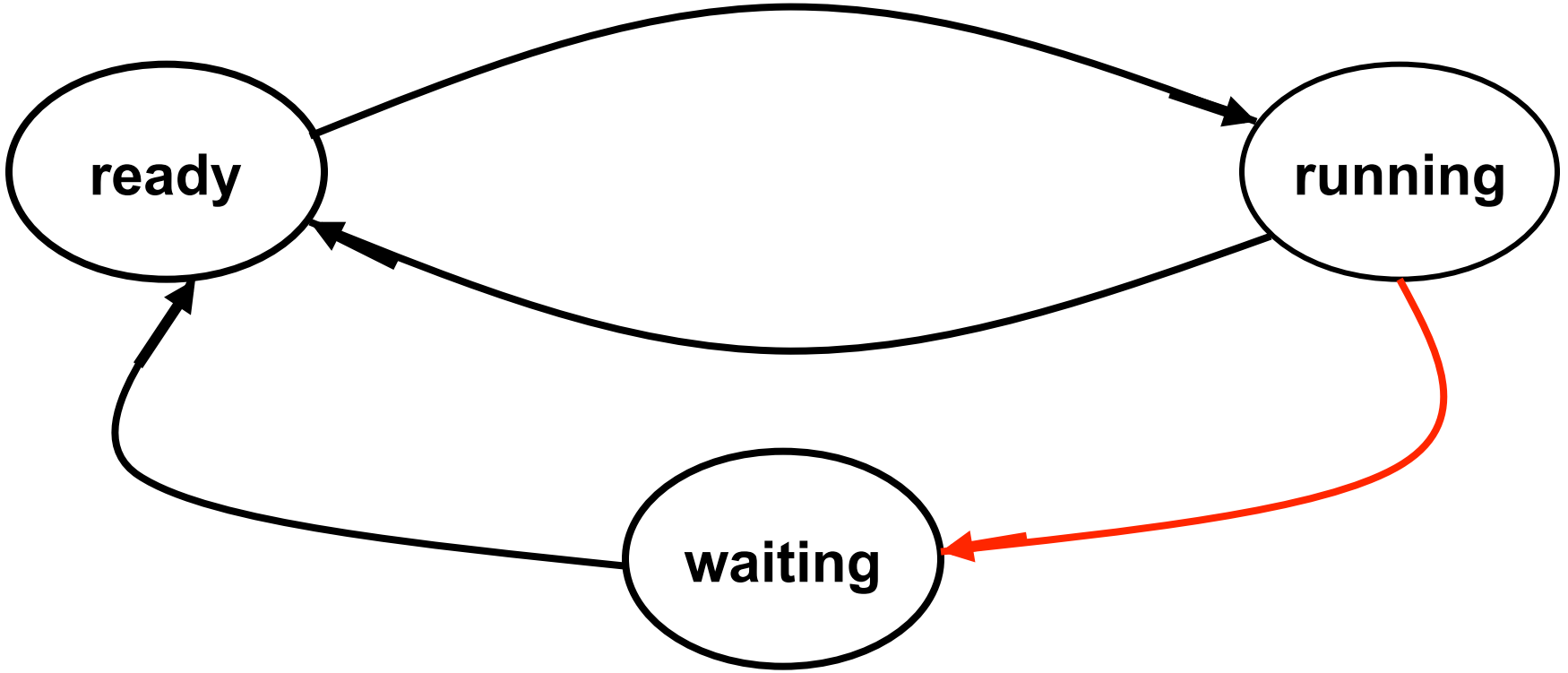
Se ci sono più processi in stato di ready, come accennato nel Cap. 3 lo scheduler sceglie sulla base della **politica di scheduling**.

Il processo non può forzare lo scheduler, il tempo di permanenza in ready dipende dalle politiche di scheduling e dal numero/tipo di processi presenti nel sistema.



Caso **running → ready**: il processo subisce una **preemption** perchè un processo con priorità maggiore diventa ready. In un sistema con timesharing può anche verificarsi la **scadenza del time slice**.

In entrambi i casi il processo subisce l'evento passivamente, non può far nulla per opporsi. Se non si verificasse l'evento, il processo potrebbe proseguire il proprio lavoro.



Caso **running → waiting**: il processo si trova **impossibilitato ad eseguire**. Si blocca in attesa di un **evento sbloccante**. Esempi:

- Avvio di un'operazione di I/O. Evento atteso: completamento I/O.
- Lettura di un messaggio da un processo partner non ancora inviato. Evento atteso: invio del messaggio.

N.B. per "impossibilitato ad eseguire" si intende "impossibilitato a sfruttare la CPU". Nel frattempo, se è il caso, il DMA può lavorare

Osservazione:

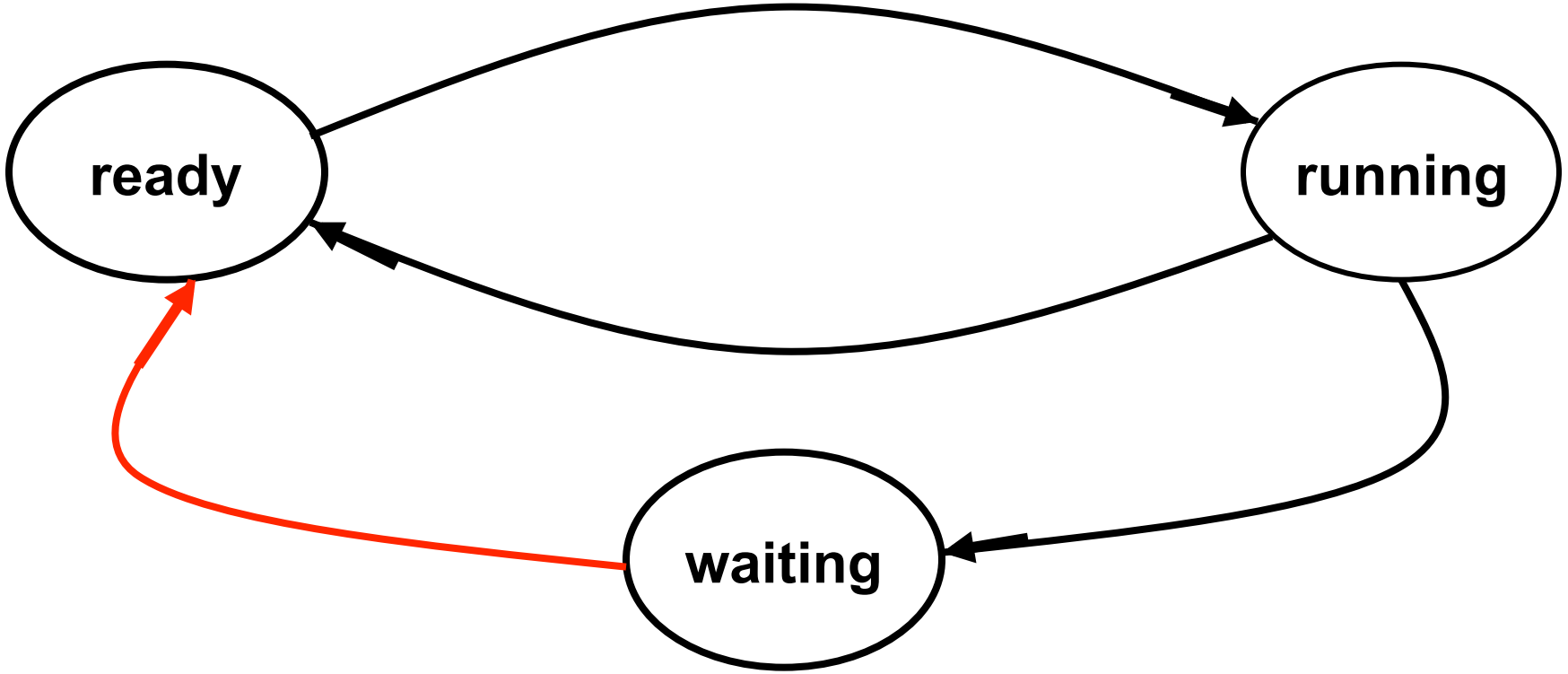
- Abbiamo già discusso che quando un processo **P** è running ed arriva un interrupt, il controllo passa al S.O., precisamente ad un interrupt handler, che, a sua volta, chiamerà lo scheduler.

- Durante l'esecuzione dell'interrupt handler e dello scheduler la CPU non sta eseguendo il programma di nessun processo, quindi, di fatto, nessun processo è in esecuzione.

- Per convenzione, anche durante l'esecuzione dell'interrupt handler e dello scheduler il processo **P** che era stato interrotto è considerato running. Andrà in ready oppure in waiting se lo scheduler selezionerà un altro processo **P'**, che diventerà il processo running.

Spesso le transizioni **running → waiting** avvengono mentre il S.O. sta eseguendo una system call per conto del processo. Le cause principali sono le seguenti:

- Il processo richiede un'operazione di I/O;
- Il processo richiede una risorsa (e.g. file, stampante,..);
- Il processo chiede esplicitamente di essere messo in waiting per un intervallo di tempo (istruzione **pause**);
- Il processo richiede un messaggio da un altro processo;
- Il processo chiede esplicitamente di attendere che un altro processo abbia eseguito una determinata azione.



Caso **waiting → ready**: si verifica l'evento sbloccante atteso dal processo. Esempi:

- il DMA completa l'operazione di I/O avviata in precedenza,
- il processo partner invia il messaggio atteso.

Il processo non ha la possibilità di influire sui tempi di permanenza in stato di waiting, perché non può far nulla per forzare/favorire il verificarsi dell'evento atteso.

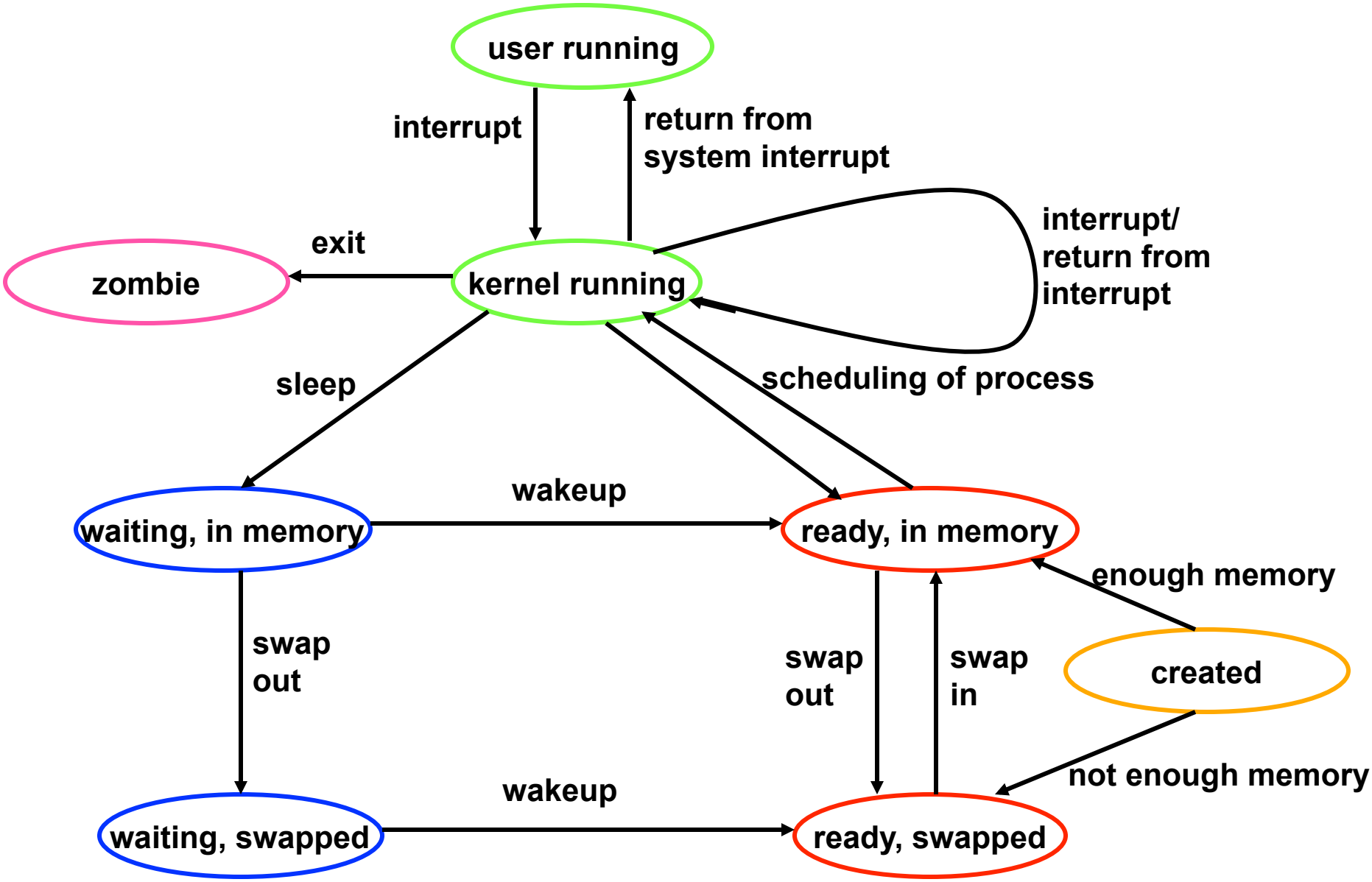
Osservazioni:

- Sarebbe insensato schedulare i processi in waiting: non saprebbero come usare la CPU.

- Il S.O. può schedulare in base alla priorità, e può capitare che un processo appena sbloccato sia quello a maggior priorità: in tal caso la transizione ready → running segue immediatamente la waiting → ready, ed il processo che era in running subisce la preemption e va in ready.

- Lo stato di waiting è chiamato anche **blocked** o **sleeping**.

- I tre stati ready, running, waiting sono considerati da tutti i S.O.. Alcuni S.O. possono distinguere sottostati per ognuno dei tre. Vediamo ora lo schema degli stati di UNIX System V.


user running
interrupt
return from
system interrupt
interrupt/
return from
interrupt
exit
zombie
kernel running
sleep
scheduling of process
wakeup
waiting, in memory
ready, in memory
enough memory
swap
out
swap
out
swap
in
created
not enough memory
wakeup
waiting, swapped
ready, swapped

Spiegazione:

- Il S.O. può gestire un numero di processi che richiedano più memoria di quella disponibile: alcuni processi hanno le aree testo, dati e stack **interamente in memoria**, altri processi hanno le aree testo, dati e stack **interamente su uno swap device** (solitamente una porzione di disco).

  N.B.: questo schema è ormai obsoleto, nei S.O. attuali è possibile avere processi parzialmente in memoria (e.g. demand paging), approfondiremo nel Cap. 7.

- I processi in **ready** o **waiting** possono essere **in memory** oppure **swapped out**. Lo **swapping out** è deciso dal S.O. quando serve memoria. Lo **swapping in** viene effettuato a certi intervalli di tempo dal S.O., solo per i processi ready ed in base alla quantità di memoria libera.

- Le transizioni running → waiting (**sleep**), waiting → ready (**wakeup**) e ready → running (**scheduling of process**) sono come nello schema precedente.

- Lo stato running è diviso in **user running** e **kernel running**.

- Gli interrupt sono gestiti dal S.O.: quando un processo è in user running, in caso di hardware interrupt, system call o eccezione, avviene una transizione **user running → kernel running**. Quando l'interrupt handler termina, si ha la transizione opposta **kernel running → user running**. (Vedi la discussione di Pag. 34).

- Un interrupt di maggior priorità può interromperne uno di priorità inferiore: ciò motiva la transizione **kernel running → kernel running**.

- Quando un processo termina entra nello stato di **zombie**, ciò serve prevalentemente per collezionare statistiche prima della cancellazione totale.

- I processi appena creati vanno in memoria o nello swap device a seconda della disponibilità.

4.5: IMPLEMENTAZIONE DEI PROCESSI – PROCESS CONTROL BLOCK

Il S.O. deve ovviamente tener traccia di tutti i processi presenti nel sistema, avvalendosi di apposite strutture dati.

Per esempio, ciò è necessario in fase di scheduling, perché il S.O. deve sapere quali processi chiedono la CPU (cioè sono ready) e quali no (sono waiting), e quali sono le relazioni di priorità tra i processi ready.

Il S.O. mantiene una **process table**, con una voce, **Process Control Block** (**PCB**), per ogni processo.

Le voci contenute nel PCB variano in base al S.O. considerato. Per il momento ci limitiamo ad elencare un sottoinsieme delle voci previste da tutti i S.O..

N.B.: i PCB sono nella kernel area – inaccessibile in modalità user.

Voci certamente presenti in ogni PCB:

- identificativo del processo (**Process IDentifier - PID**);
- **stato del processo** (running, waiting, ready, …….);
- **stato della CPU**: quando il processo non è running, nel PCB sono memorizzati i valori da assegnare ai registri quando il processo tornerà running;
- **priorità** ed altri parametri da usare per lo scheduling;
- puntatore ad una zona di memoria che contiene le informazioni necessarie per accedere all'**area di testo (*)**;
- puntatore ad una zona della memoria che contiene le informazioni per accedere all'**area dati (*)**;

- puntatore ad una zona della memoria che contiene le informazioni per accedere all'**area di stack** **(*)**;

- puntatori a zone della memoria che contengono le informazioni per accedere ai file aperti **(*)**. Questi puntatori si chiamano **file descriptor**. Non indaghiamo oltre.

- puntatori a zone della memoria che contengono le informazioni per accedere ai **dispositivi aperti** **(*)**. Sono file descriptor nei S.O. che trattano i dispositivi come file, quali ad esempio UNIX.

- un **PCB pointer**, che serve per creare:

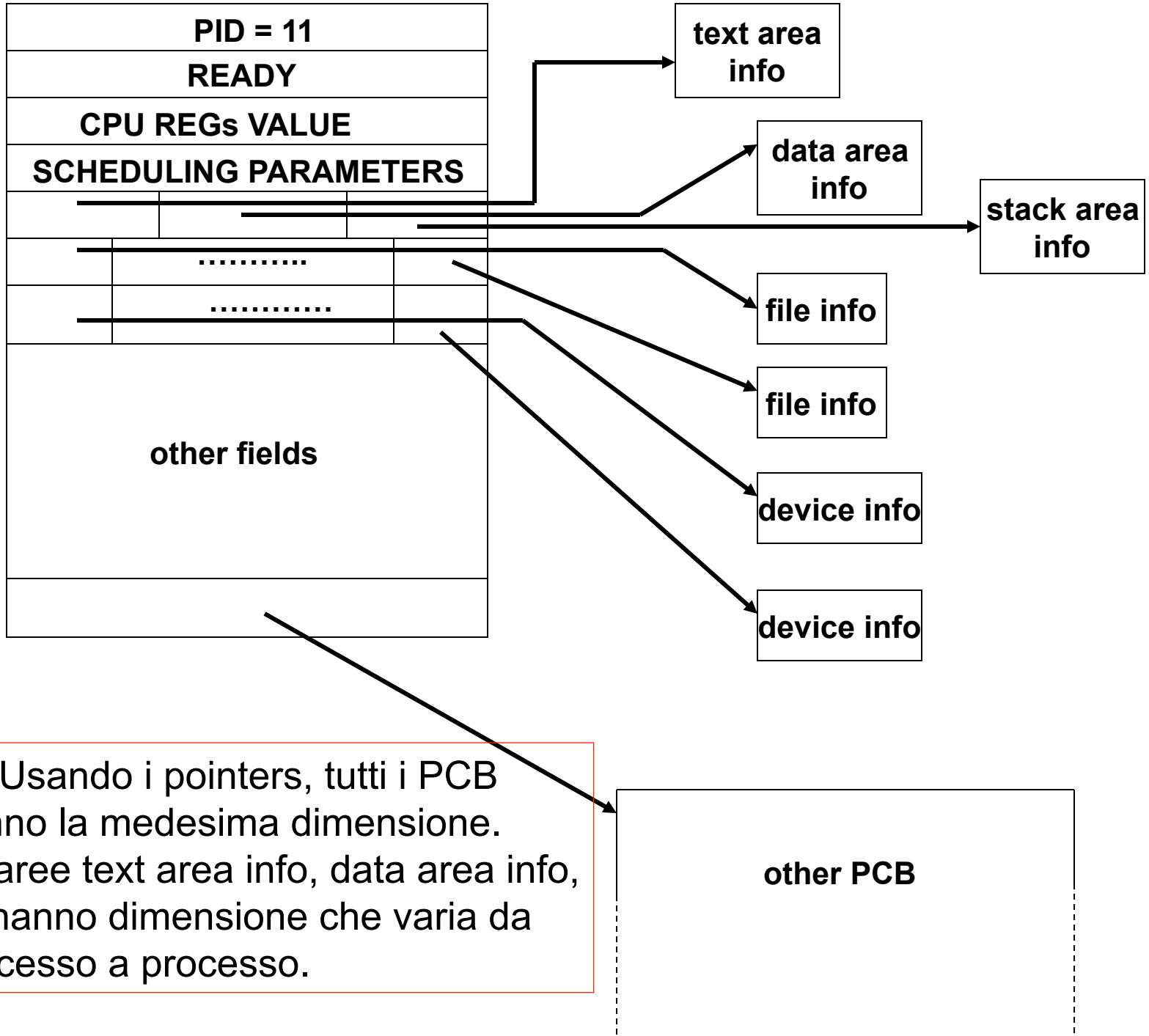
  - la/le **lista/e dei processi in stato di ready**;

  - per ogni evento che può essere atteso dai processi, una **lista dei processi in waiting** di tale evento.

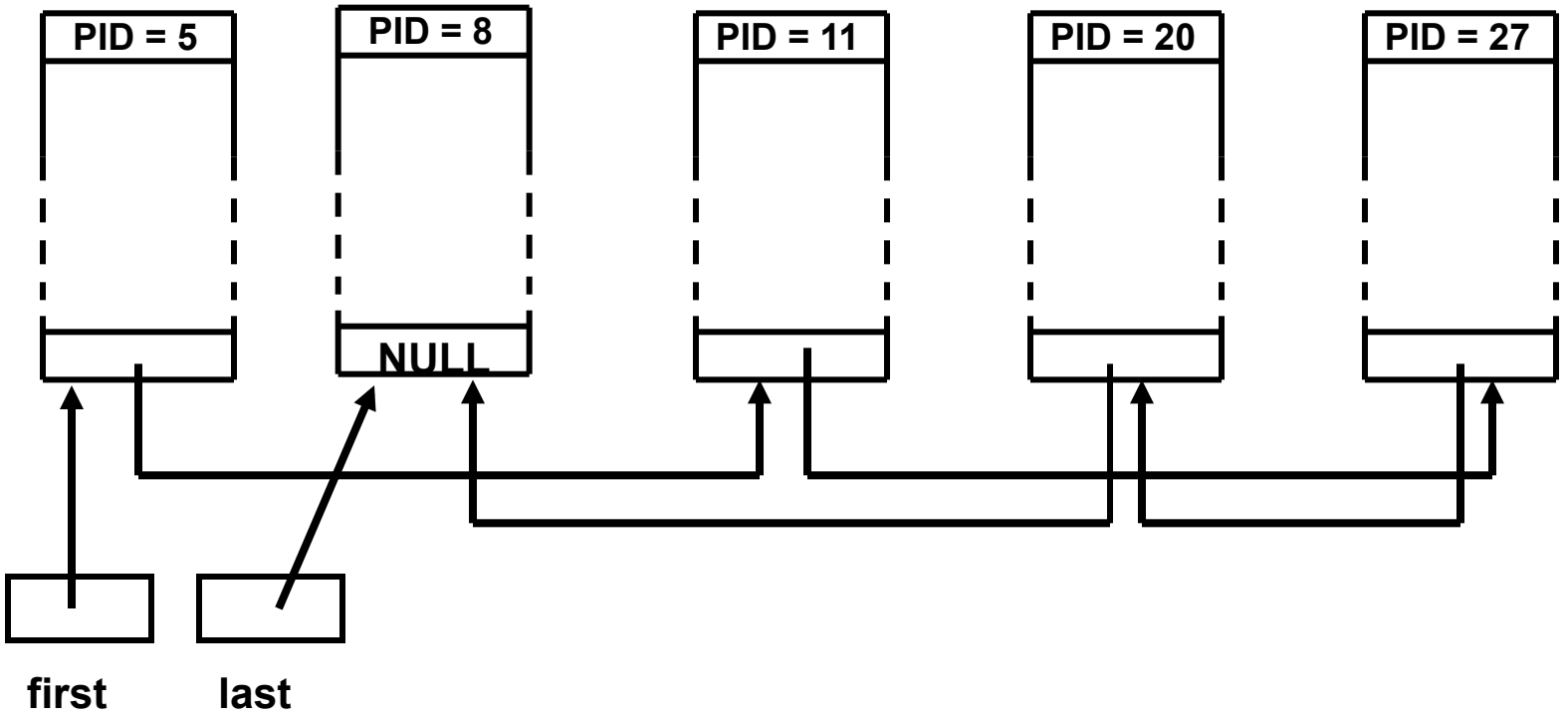
Vediamo ora:

- una rappresentazione grafica di un PCB ed un esempio di inserimento e rimozione dalla coda dei processi pronti,

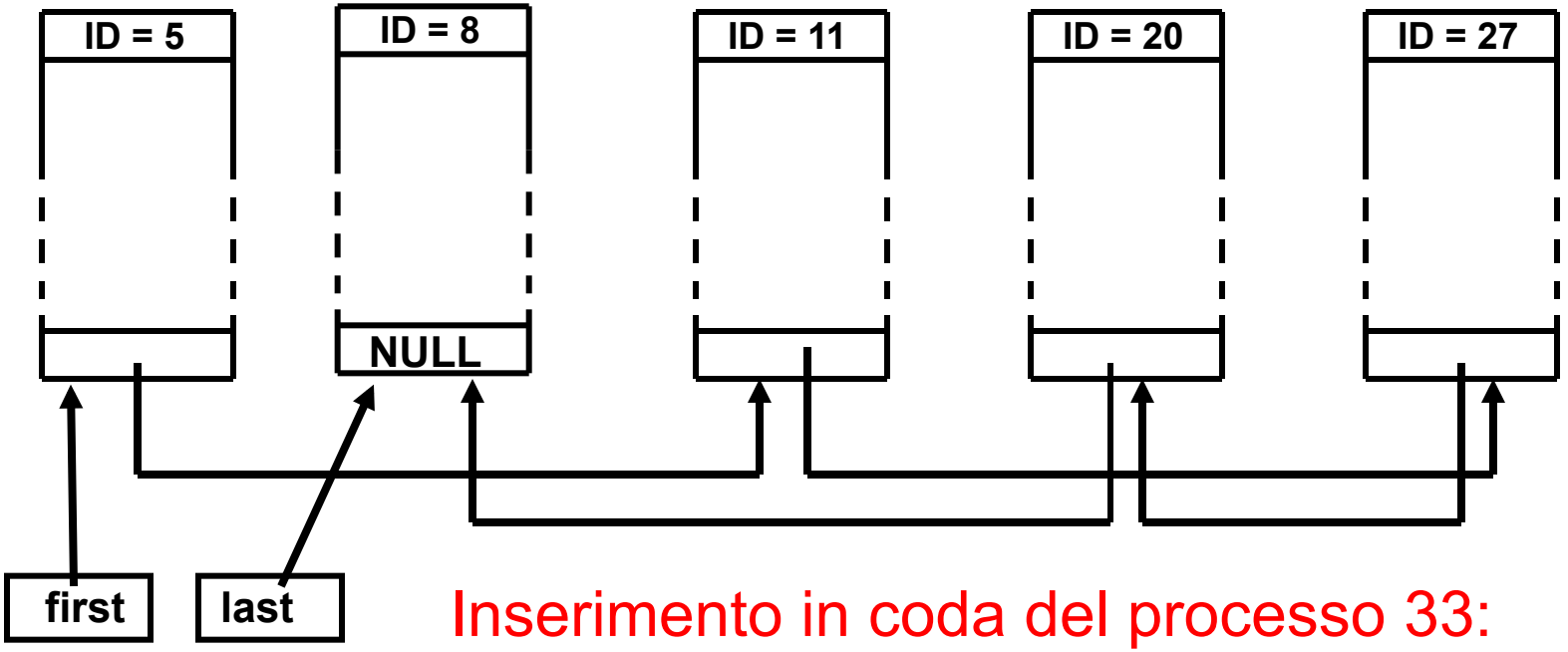
- un esempio relativo alle informazioni relative alle aree di memoria.



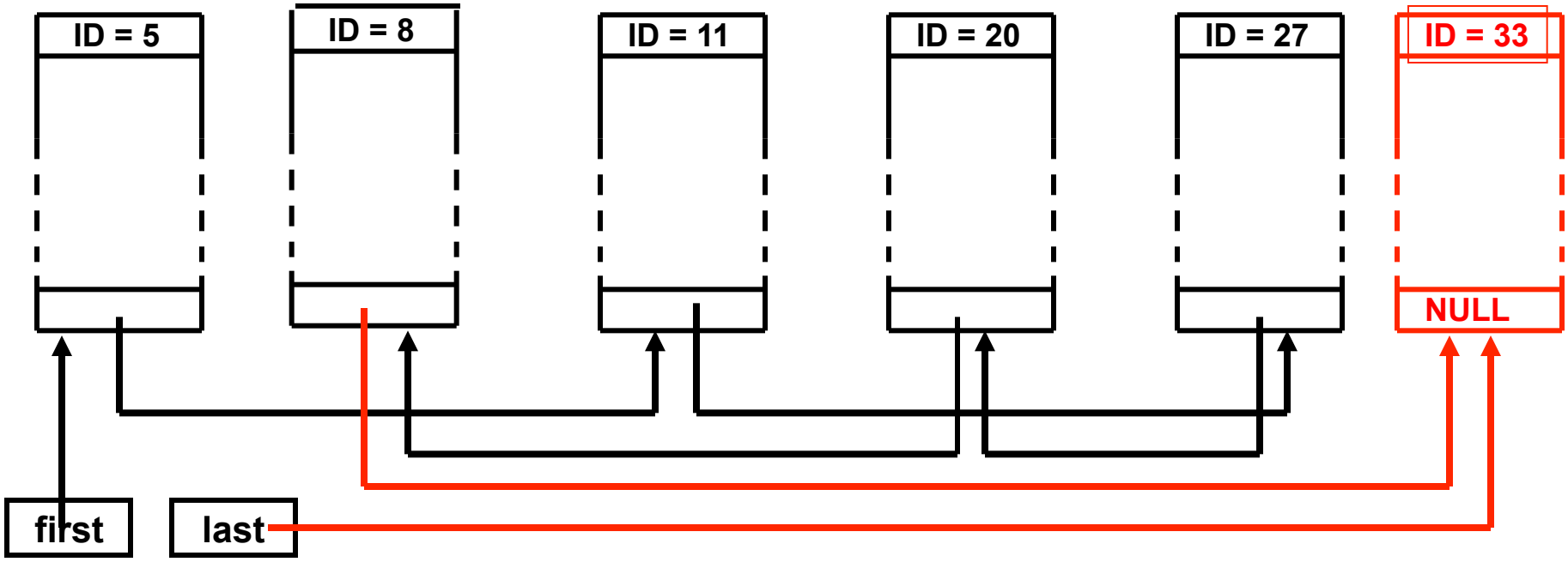
(*) Usando i pointers, tutti i PCB hanno la medesima dimensione. Le aree text area info, data area info, … hanno dimensione che varia da processo a processo.

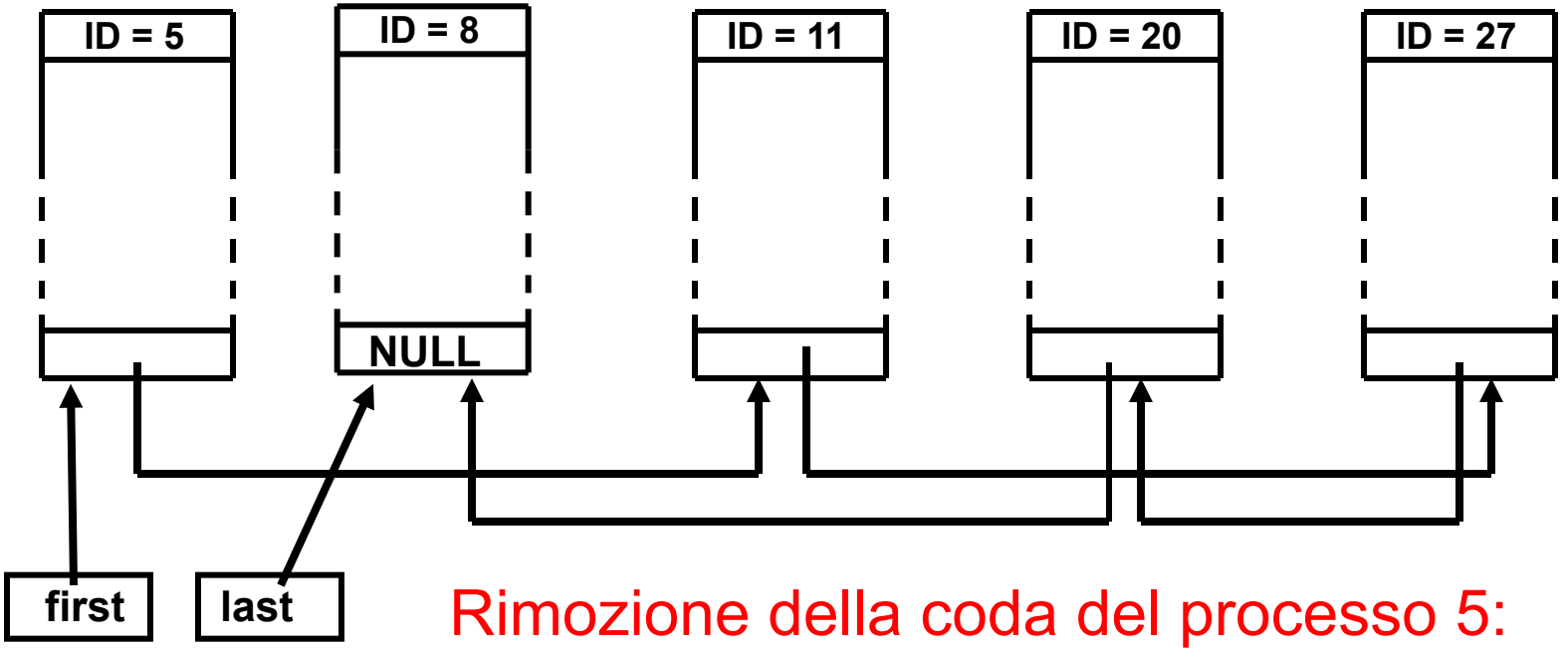


Esempio di lista di PCB realizzata con PCB pointer.
Assumiamo che sia la lista dei processi pronti, che tutti i processi abbiano ugual priorità e che vengano schedulati in base all'anzianità di presenza in lista. Cioè, la lista è in realtà una **coda**.
L'ordine di anzianità è pertanto: 5, 11, 27, 20, 8.
Vediamo ora come gestire un inserimento in coda ed una rimozione dalla coda.

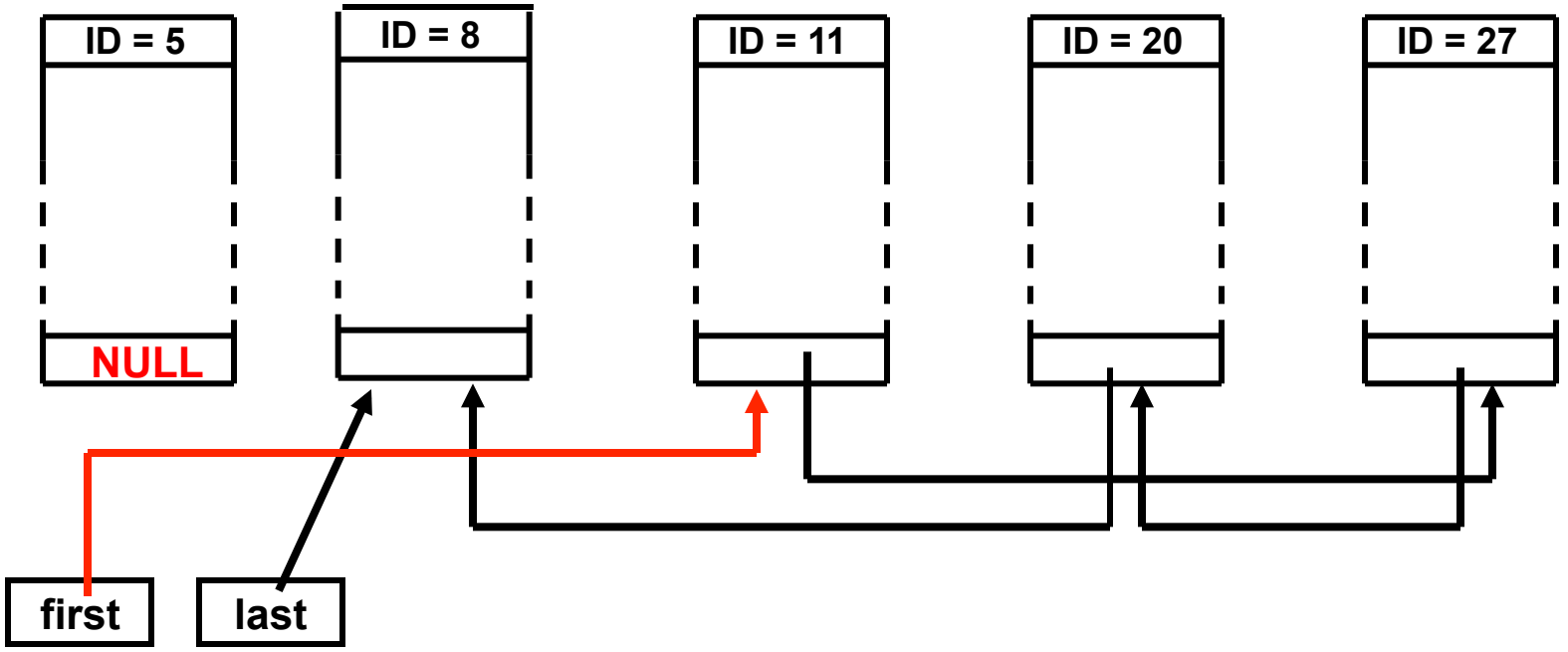


Inserimento in coda del processo 33:





Rimozione della coda del processo 5:



Se il S.O. fosse scritto in Java:

```
class PCB{

    int PID,
    ……..  /*altre variabili*/
    PCB next,  /*pointer*/

    PCB(int PID,…,PCB next){
        /*costruttore banale*/
        this.PID=PID;
        …..
        this.next=next;
    }

    ……. /*altri metodi*/

 }
```

```
class PCBQueue{
    PCB first, last;
    PCBQueue( ){ /*coda inizialment vuota*/
        first=last=null;
    }
    void insert(PCB b){
        if(first==null){
            first=last=b;
        }
        else{
            last.next=b;
            last=b;
        }
    }
    PCB remove( ){
        PCB b = first;
        first=first.next;
        if(first==null){last=null;}
        b.next=null;
        return b;
    }
}
```

Cosa si intende per “info per accedere alla memoria”?

Facciamo un esempio con **memoria paginata da** 2^{32} byte, divisa in 2^{22} pagine da 2^{10} byte ciascuna.

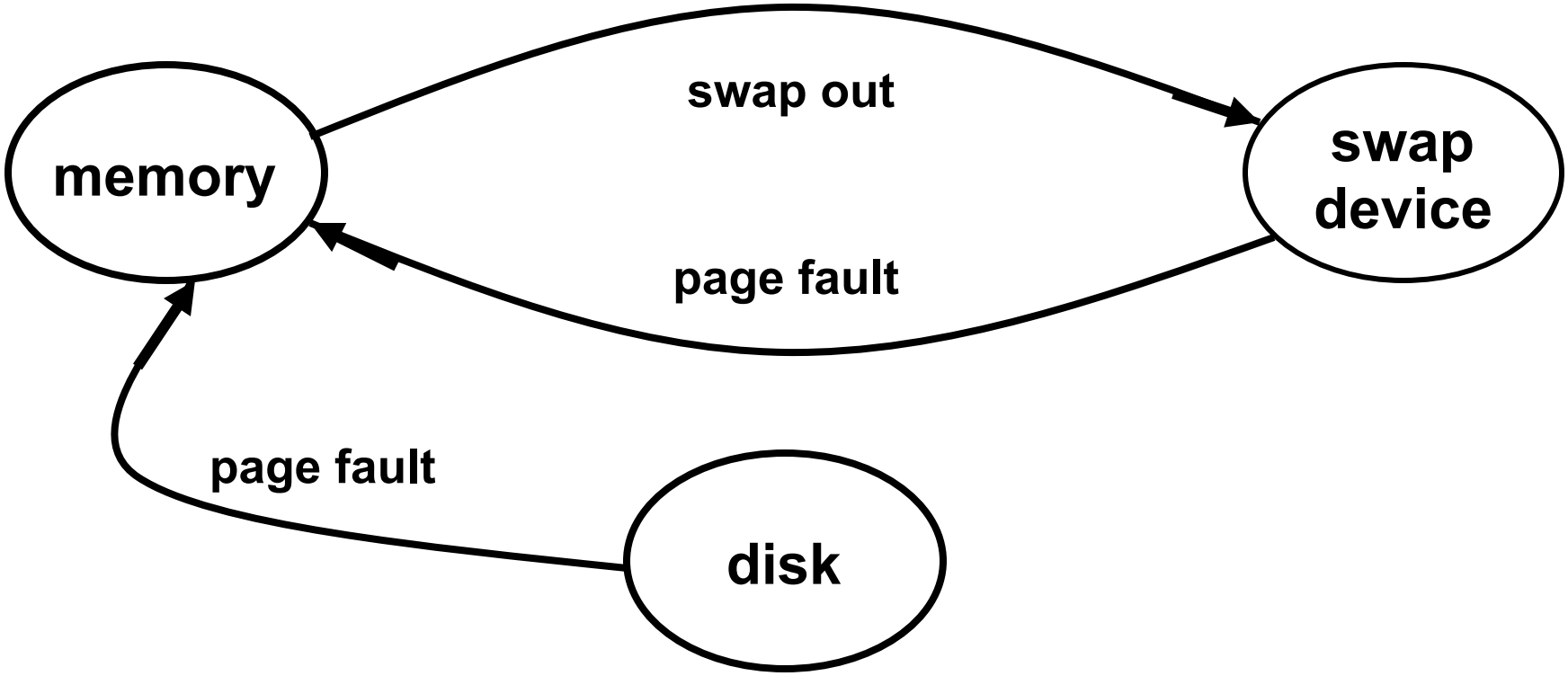
Gli indirizzi di memoria hanno il seguente significato:

22 leftmost bit = page number, 10 righmost bit = page offset.

Memoria virtuale con paginazione:

- le aree testo, dati e stack sono divise in pagine da 2^{10} byte.
- non è detto che tutte le pagine del processo siano in memoria: inizialmente sono sul disco, cioè nel file eseguibile, poi “rimbalzano” tra memoria fisica e swap device.

NB: lo swap device è un dispositivo logico. Fisicamente è una parte del disco.



Page fault: è un'eccezione causata da un accesso ad una pagina di memoria virtuale che non è nella memoria fisica. L'interrupt handler provvede a caricarla. Se non è nella cache del disco, ciò richiede un'operazione I/O che manda il processo in waiting per un tot di tempo.

Swap out: Il S.O. effettua ad intervalli regolari degli swapping out delle pagine meno utilizzate per "liberare memoria".

Per ogni pagina delle aree testo e dati, servono le seguenti info:

- Numero della pagina fisica in cui è caricata.
  Il valore NULL indica che la pagina non è attualmente in memoria, un tentativo di accesso causerà un page fault.

- Indirizzo della pagina nello swap device.
  Il valore NULL indica che la pagina non è mai stata nello swap.

- Indirizzo nel disco.

Per le pagine dell'area di stack la situazione è simile, ma possono trovarsi solo in memoria o nello swap device.
Perché?

Per i dettagli attendere Cap. 7.

4.6: IMPLEMENTAZIONE DEI PROCESSI – CONTEXT SWITCH

Definizione: il **contesto** di un processo (**process context**), detto anche **ambiente**, è costituito da:

- **register context**: il contenuto dei registri della CPU;
- **user-level context**: aree testo, dati e stack;
- **system-level context** (inaccessibile dal programma user):
  - PCB;
  - informazioni per accedere a memoria, file e dispositivi cui punta il PCB;
  - **kernel stack**: contiene i frame degli interrupt handler e delle procedure chiamate dagli interrupt handler. Vedremo tre esempi di uso del kernel stack alle Pag. 58, 64 e 70.

Il S.O. effettua 3 operazioni fondamentali sui processi:

- **Context save**: quanto il processo running va in ready o waiting, il S.O. salva nel PCB tutti i dati necessari a far ripartire il processo in futuro. Come visto nel Cap. 3, parte dell'operazione (per lo meno il salvataggio del registro **PC**) va effettuato via hardware.

- **Scheduling**: tenendo conto della politica di scheduling, il S.O. sceglie un processo tra quelli in ready per l'esecuzione.

- **Dispatching**: dopo che un processo viene schedulato, il S.O. deve ripristinarne il contesto sfruttando i dati salvati nel PCB al momento del context save. Anche in questo caso parte dell'operazione viene effettuata via hardware. Se il S.O. ammette memoria virtuale, va reimpostata anche la MMU.

Si parla di **context switch** quando il S.O. effettua il context save per un processo **P**, schedula un processo **P'** diverso da **P** e ne effettua il dispatching.

N.B.: mentre il S.O. effettua il context save di **P**, lo scheduling ed il dispatching di **P'**, il processo **P** risulta essere in running.

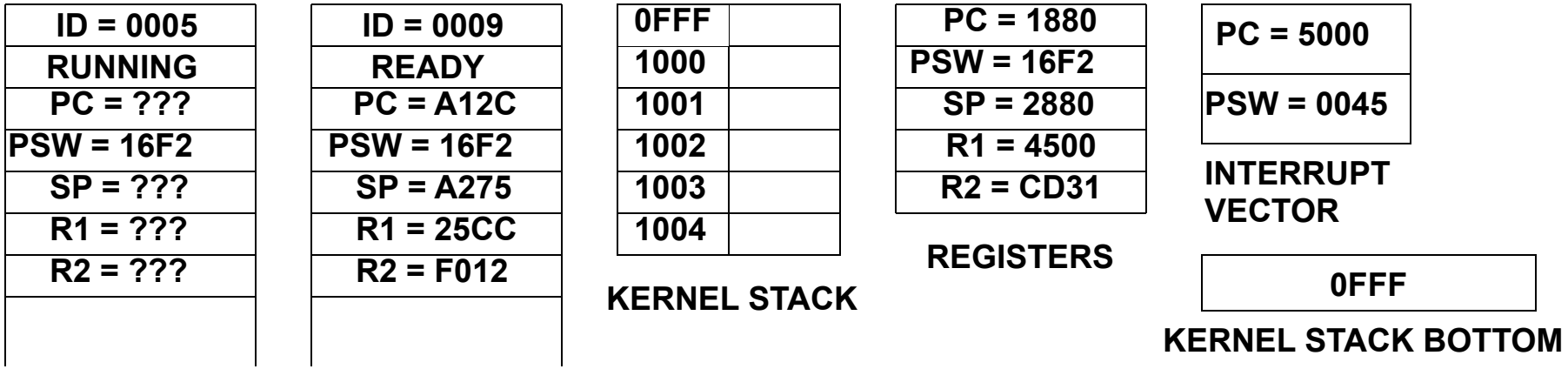
E' evidente che il context switch causi overhead, specie se l'algoritmo di scheduling è complicato.

Inoltre, dopo il context switch, il processo **P'** rischia di non avere immagini della propria memoria nella cache, immagini dei propri file nella cache del disco,… tutti fatti che influiscono negativamente sulla sua velocità di avanzamento.

ESEMPIO 1

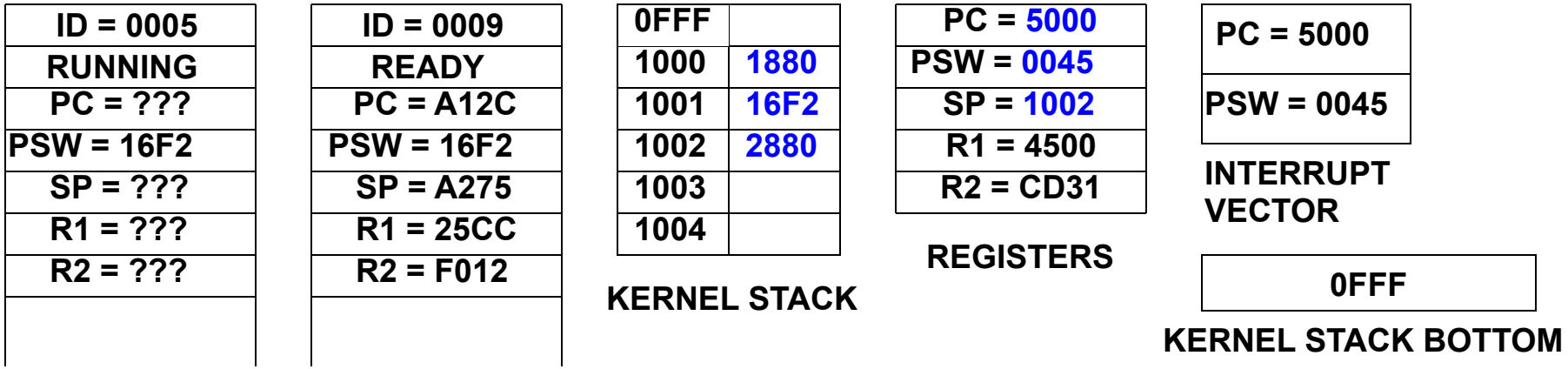
La CPU ha i control register PC, SP, PSW ed i registri generali R1, R2.
La CPU è a 16 bit, la sequenza di 16 bit è rappresentabile con 4 caratteri esadecimali. Esempio: A12C rappresenta 1010 0001 0010 1100.
Assumiamo che la SRIA sia nel kernel stack.
Il processo 5 è running (user running secondo UNIX), il processo 9 è ready.
Il processo 5 esegue una TRAP, che verrà gestita dall'apposito interrupt handler. Al ritorno dall'interrupt handler, il processo 5 riprenderà ad eseguire.

Figura 1 - Situazione iniziale, prima della TRAP – Il bit PM della PSW è 0, cioè user mode, anche se dal valore 16F2 della PSW non si capisce:



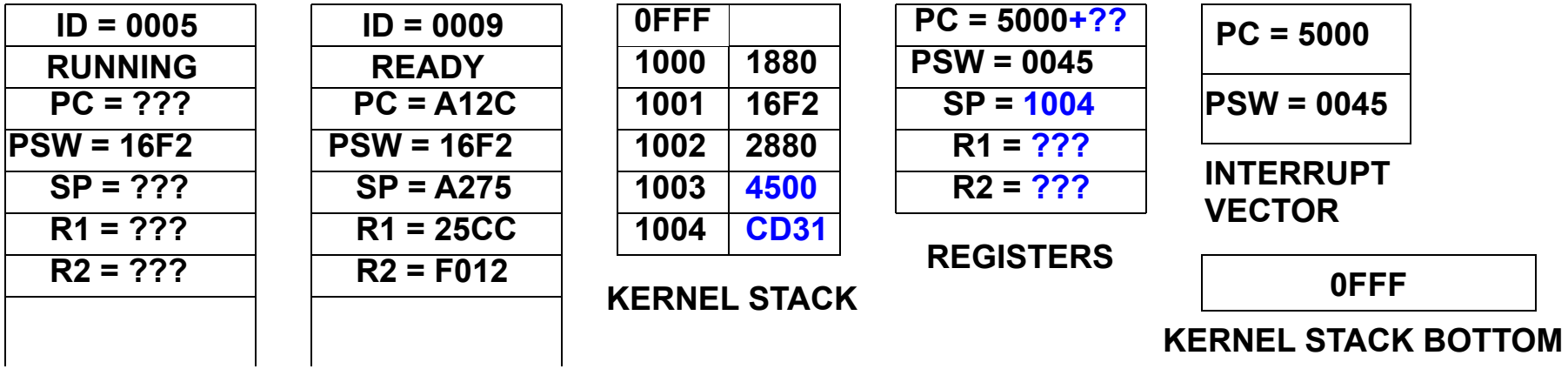
Quando la CPU riconosce l'interrupt generato dalla TRAP, l'hardware salva nel kernel stack i valori dei registri PC, PSW, SP ed inizializza i registri PC, PSW in base ai valori dell'interrupt vector. Una parte del context save è pertanto già avvenuta. Secondo UNIX, il processo 5 va in kernel running.
Il nuovo valore di PC punta, pertanto, alla prima istruzione dell'interrupt handler.
Viene aggiornato anche il registro SP, che punta ora al top del kernel stack.

Figura 2 - Situazione dopo il riconoscimento dell'interrupt, prima di eseguire l'interrupt handler (in blu i valori modificati rispetto alla figura precedente):



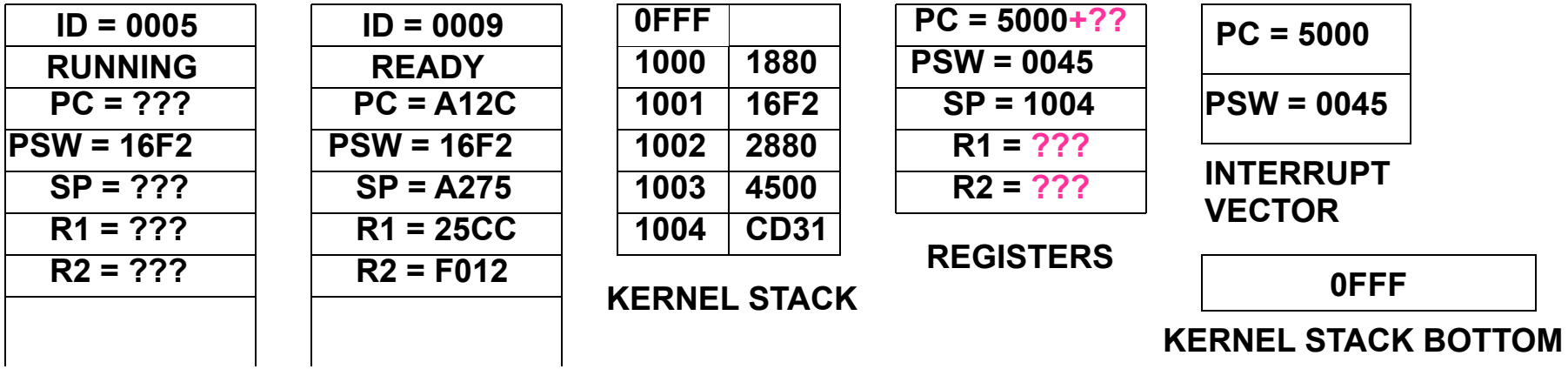
L'interrupt handler inizia il proprio lavoro salvando il contenuto dei registri generali R1, R2 nel kernel stack, aggiornando di conseguenza il valore del registro SP che punta al nuovo top dello kernel stack. Ciò completa il context save.
Dal momento che l'interrupt handler esegue alcune istruzioni per effettuare questo compito, abbiamo due conseguenze:
- il valore del registro PC risulta cambiato. Supponiamo di sapere che viene incrementato, ma di non conoscerne il valore esatto.
- R1, R2 potrebbero essere stati usati. Assumiamo di non conoscere il valore che hanno assunto.

Figura 3 - Situazione dopo il salvataggio di R1, R2:



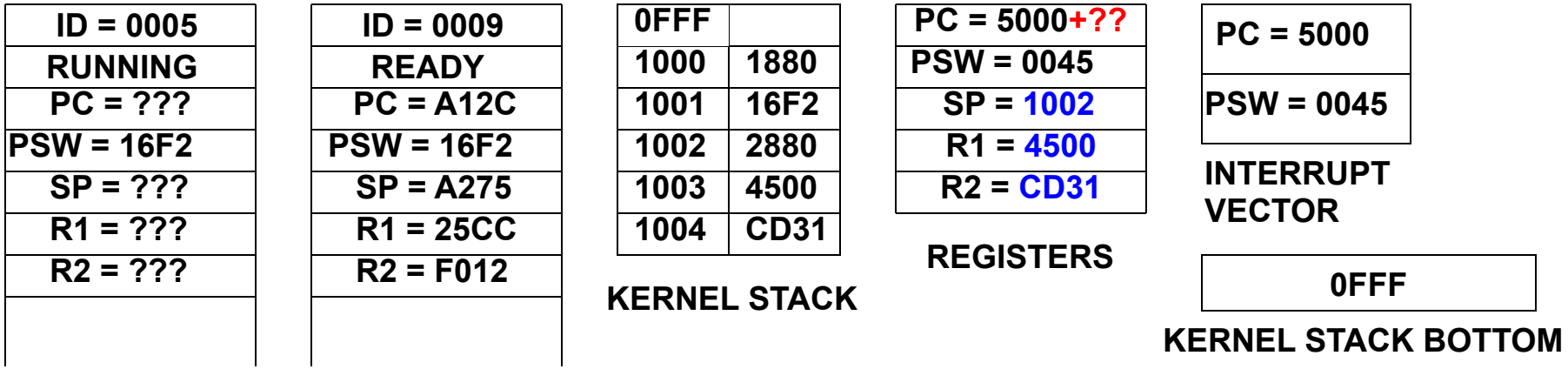
Quando l'interrupt handler termina di gestire l'interruzione, chiama lo scheduler. Abbiamo detto che il processo 5 riprende l'esecuzione. E' necessario effettuare il dispatching.
Sotto mostriamo la situazione che precede il dispatching. Rispetto alla Figura 3 sono cambiati solo i valori di PC, R1, R2, il cui cambiamento è dovuto all'esecuzione dell'interrupt handler e al "salto" allo scheduler.
N.B.: di fatto, quel che è accaduto tra la Figura 3 e la Figura 4 è la gestione vera e propria della TRAP da parte dell'interrupt handler.

Figura 4 – prima del ripristino di R1, R2:



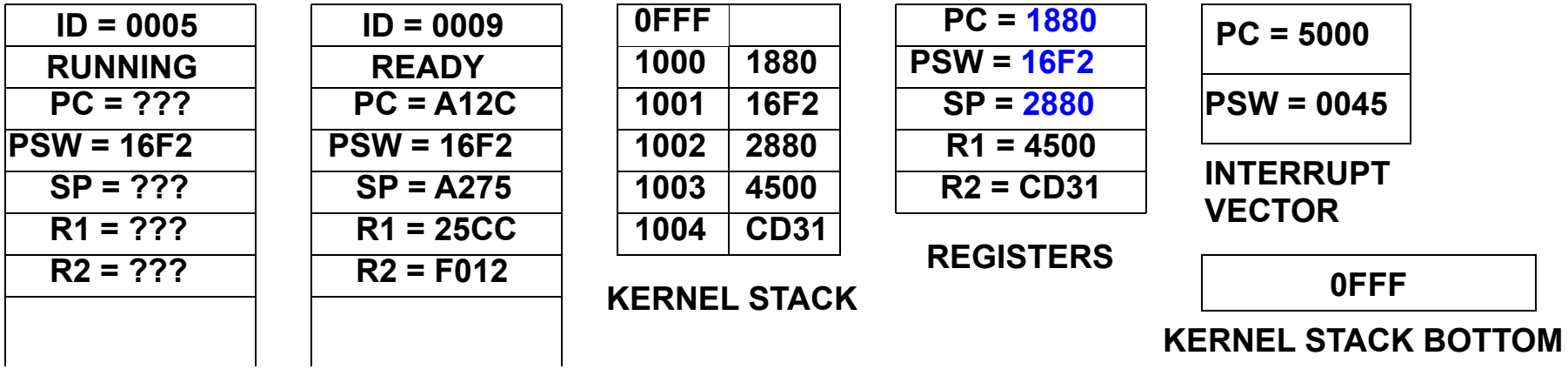
La prima fase del dispatching consiste nel ripristino del valore dei registri R1, R2, e viene effettuata eseguendo le opportune instruzioni.
Ovviamente il ripristino di R1, R2 fa si che si debba aggiornare il registro SP, perché il top del kernel stack cambia.
Il valore di PC è cambiato ulteriormente rispetto alla figura precedente.
NB. I valori degli indirizzi di memoria 1003 e 1004 (nel kernel stack) non sono stati cancellati fisicamente, ma siccome SP punta all'indirizzo 1002, dal punto di vista logico sono stati cancellati.

Figura 5 – dopo il ripristino di R1,R2:



L'ultima operazione consiste nell'eseguire l'IRET. I registri PC, PSW, SP assumono i valori che erano stati registrati nel kernel stack al momento del context save. La CPU ritorna in modalità user, il processo 5 può riprendere l'esecuzione. Secondo UNIX, il processo 5 torna in user running.

Figura 6 – dopo l'IRET.



ESEMPIO 2

Complichiamo le cose rispetto all'Esempio 1.
Supponiamo che durante l'esecuzione dell'interrupt handler che gestisce la TRAP, arrivi un disk interrupt, che deve essere trattato in quanto ha priorità maggiore del program interrupt (trap).
Assumiamo che ciò avvenga mentre PC=5200, R1=181A, R2=C23F. Ci troviamo tra la Figura 3 e la Figura 4 dell'Esempio 1. Nella figura abbiamo aggiunto l'interrupt vector per il disk interrupt.
Figura 3.a: Situazione durante l'esecuzione dell'interrupt handler per la trap al momento dell'arrivo del disk interrupt:

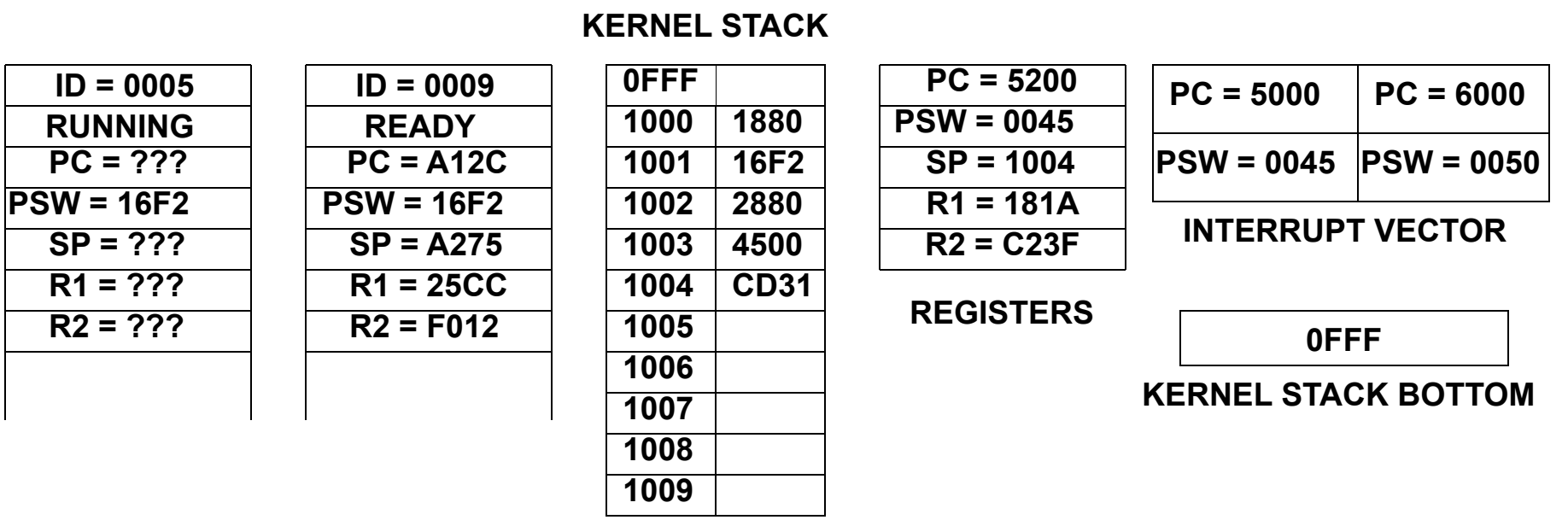


Figura 3.b – Situazione dopo:
- il riconoscimento del disk interrupt,
- il salvataggio di PC, PSW, SP nel kernel stack,
- l'impostazione di PC, PSW come da interrupt vector per disk interrupt,
- l'aggiornamento di SP al nuovo top del kernel stack.

NB. Secondo lo schema UNIX, il processo 5 ha una transizione da kernel runing a kernel running.

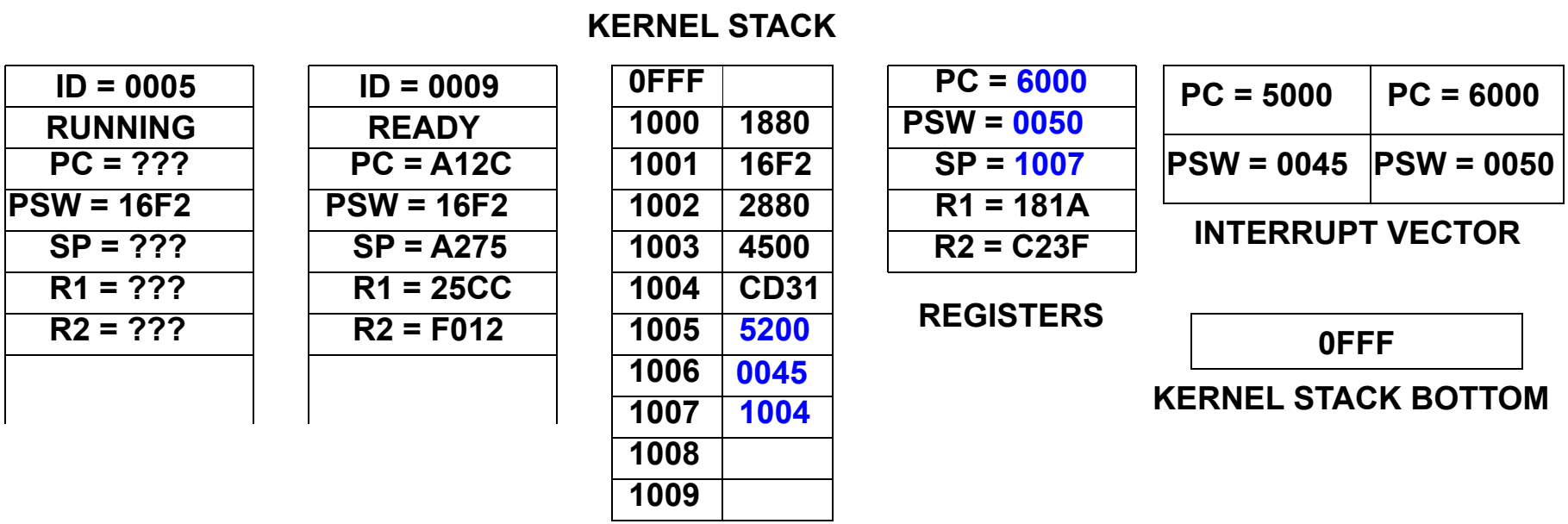


Figura 3.c – Situazione dopo:
- il salvataggio di R1, R2 effettuato dal disk interrupt handler,
- l'aggiornamento di SP al nuovo top del kernel stack.

Il valore di PC è stato incrementato, assumiamo di non conoscerlo.
I registri R1, R2 sono stati usati, assumiamo di non conoscerne il valore.

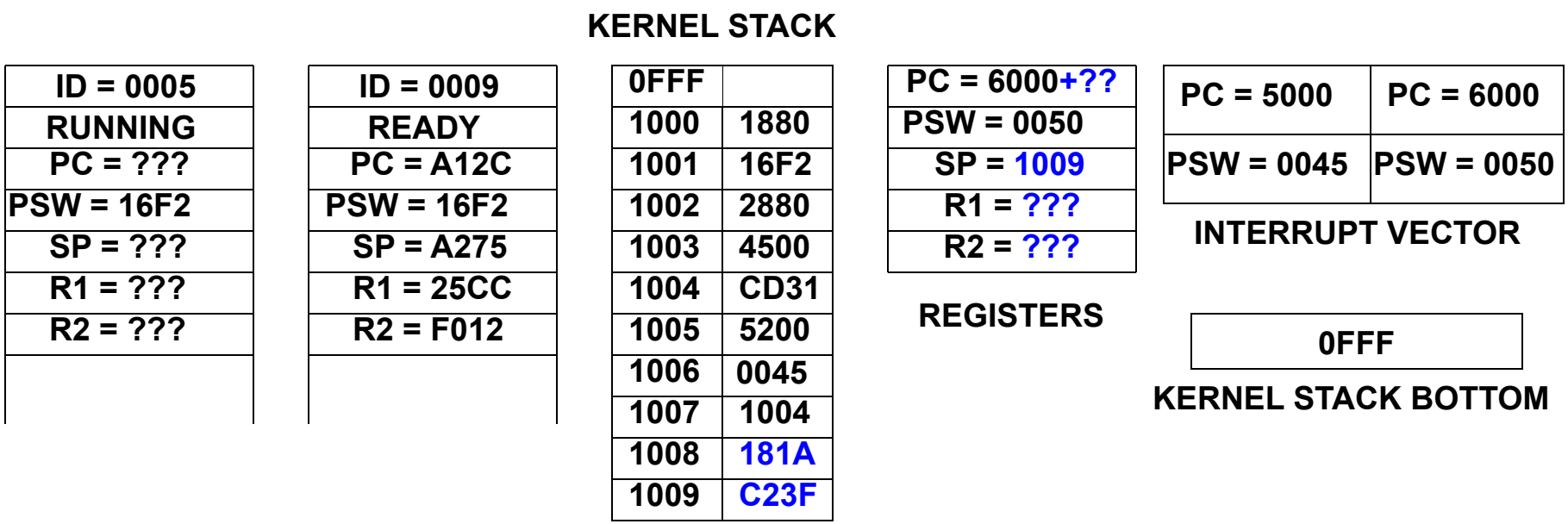


Figura 3.d – Situazione dopo:
- l'esecuzione del "cuore" del disk interrupt handler.

Il valore di PC è stato incrementato, assumiamo di non conoscerlo.

I registri R1, R2 sono stati usati, assumiamo di non conoscerne il valore.

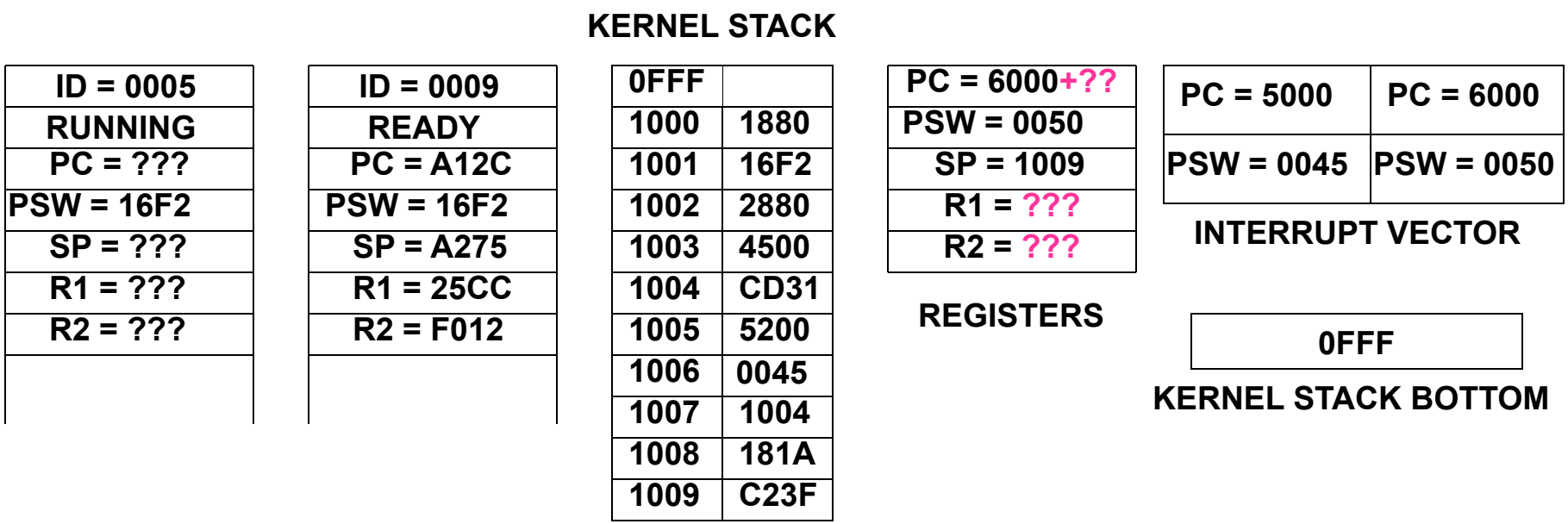


Figura 3.e – Situazione dopo:
- il ripristino di R1, R2, prima fase del dispatching
- l'aggiornamento di SP al nuovo top del kernel stack.

Il valore di PC è stato nuovamente incrementato, assumiamo di non conoscerlo.

Il contentuto degli indirizzi di memoria 1009 e 1008 non è stato cancellato fisicamente, ma SP punta all'indirizzo 1007 e, pertanto, dal punto di vista logico gli indirizzi 1008 e 1009 non fanno più parte del kernel stack.

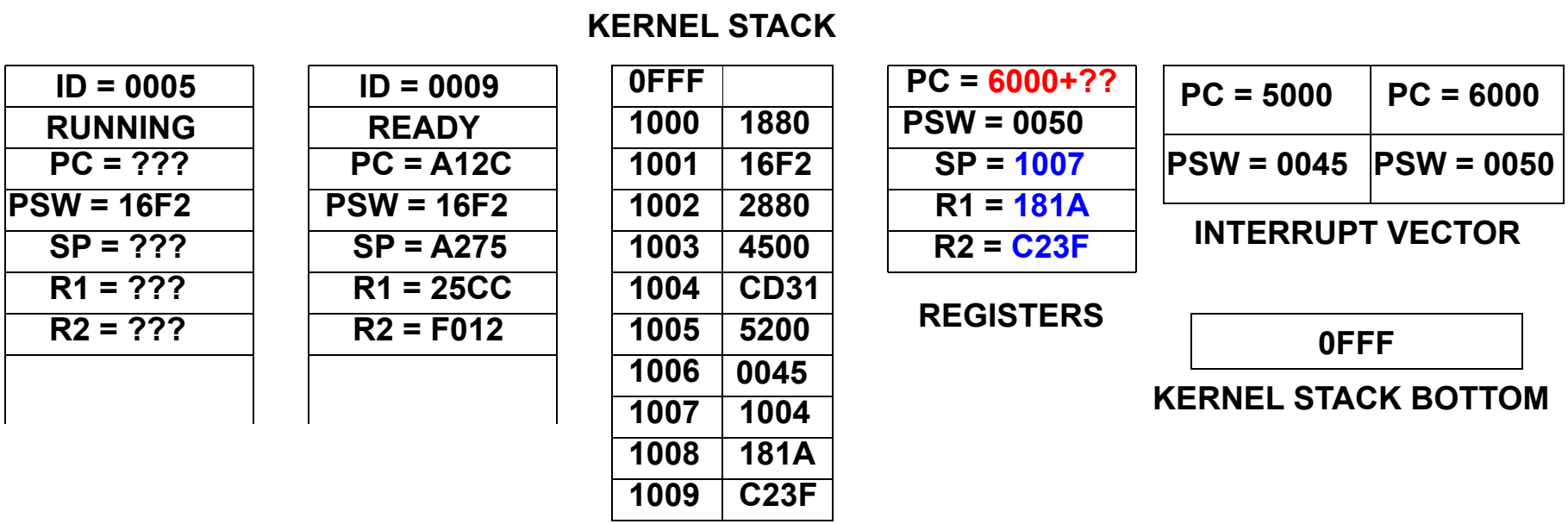
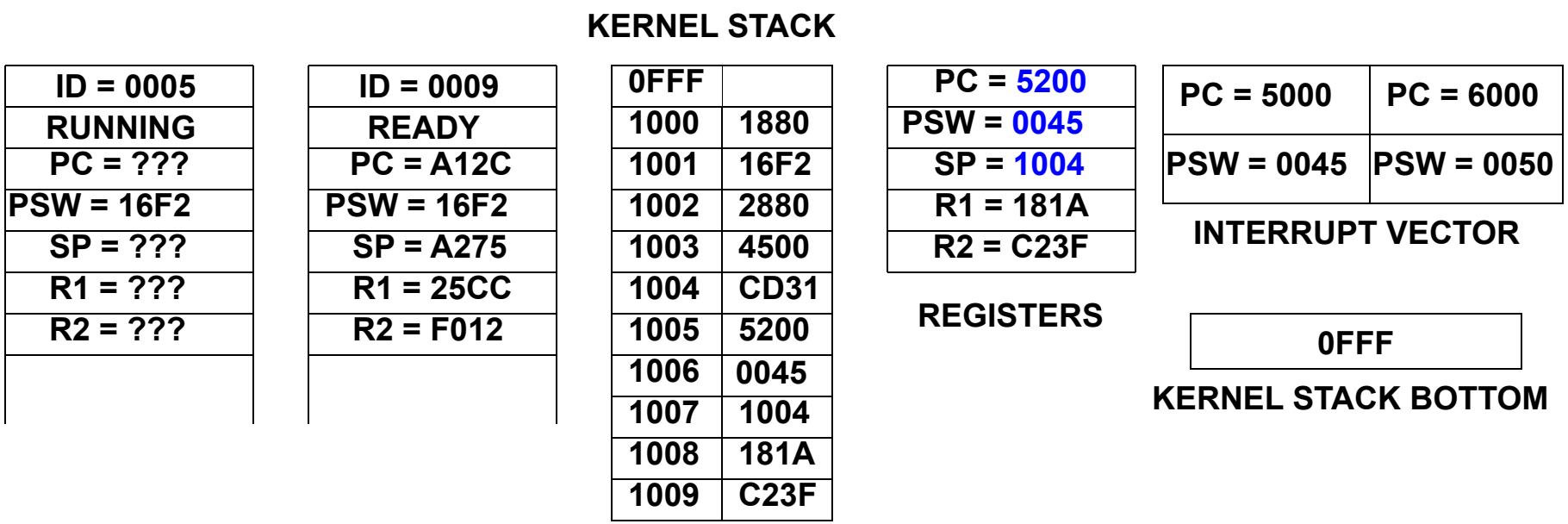


Figura 3.f – Situazione dopo l'IRET:

- PC, PSW, SP sono stati ripristinati copiandoli dal kernel stack, pertanto l'interrupt handler della TRAP può riprendere l'esecuzione, che conduce alla Figura 4.

Secondo la terminologia UNIX, abbiamo un nuova transizione del processo 5 da kernel running a kernel running.



ESEMPIO 3

Torniamo all'Esempio 1, ma supponiamo che il processo 9 sia inizialmente in waiting, la TRAP eseguita dal processo 5 risvegli il processo 9 e che lo scheduler, chiamato in causa dal trap interrupt handler, scheduli il processo 9, per esempio perchè ha maggior priorità del processo 5.

Figura 1 - Situazione iniziale, prima della TRAP.

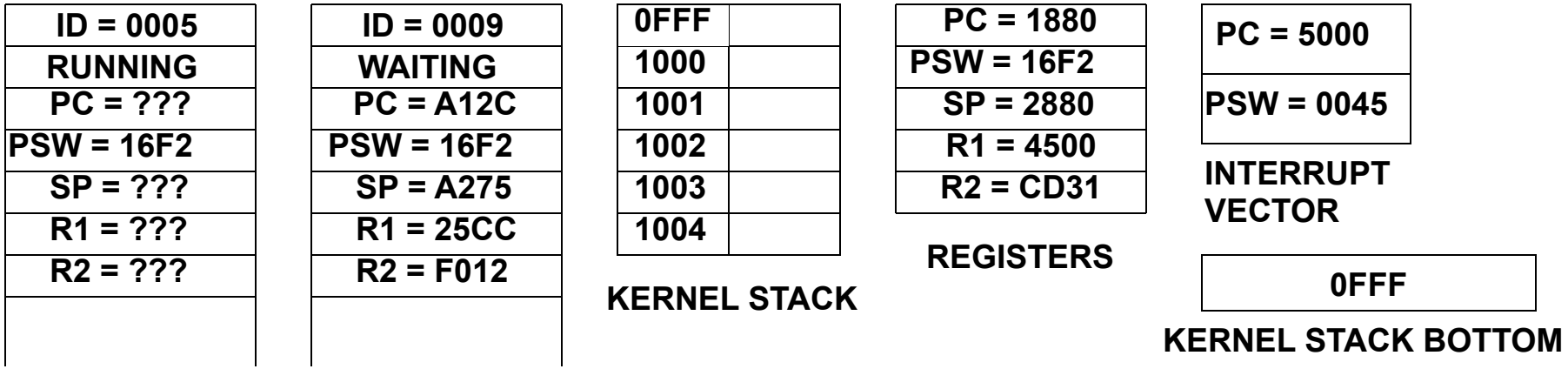


Figura 2 – Situazione dopo il salvataggio dei registri di controllo, come nell'Esempio 1.

Secondo la terminologia UNIX, il processo 5 è passato da user running a kernel running.

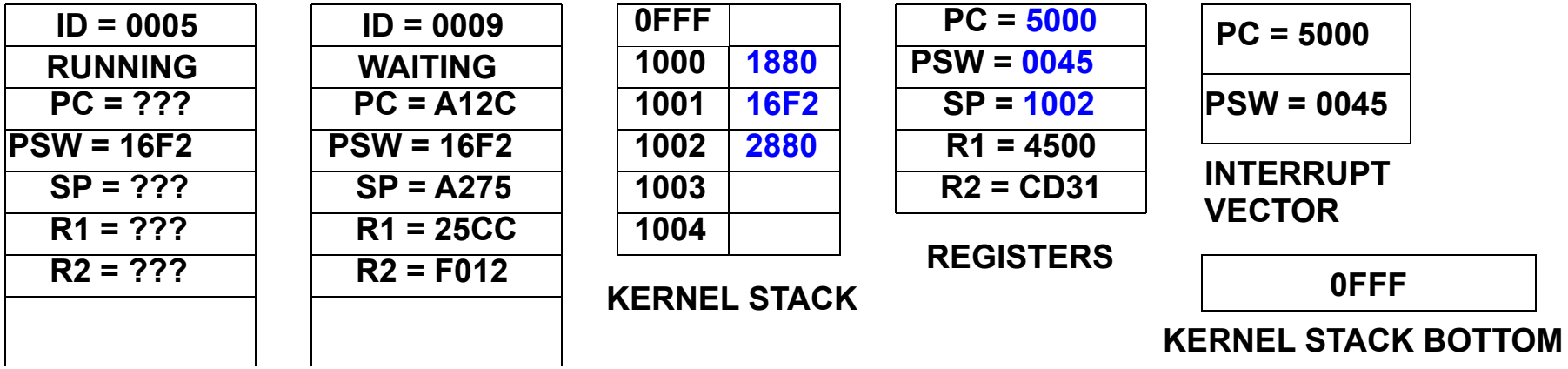


Figura 3 – Situazione dopo il salvataggio dei registri generali, come nell'Esempio 1:

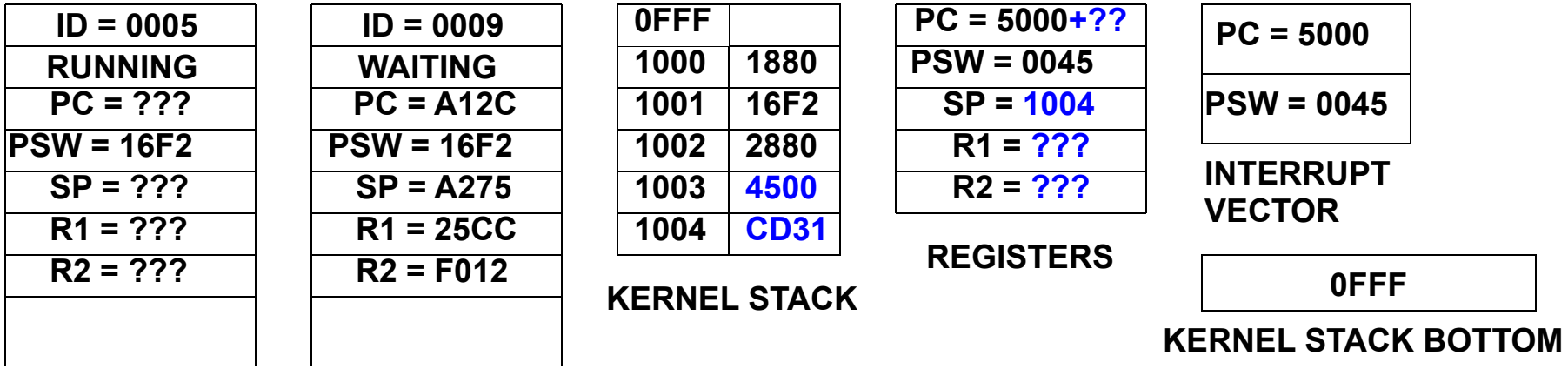
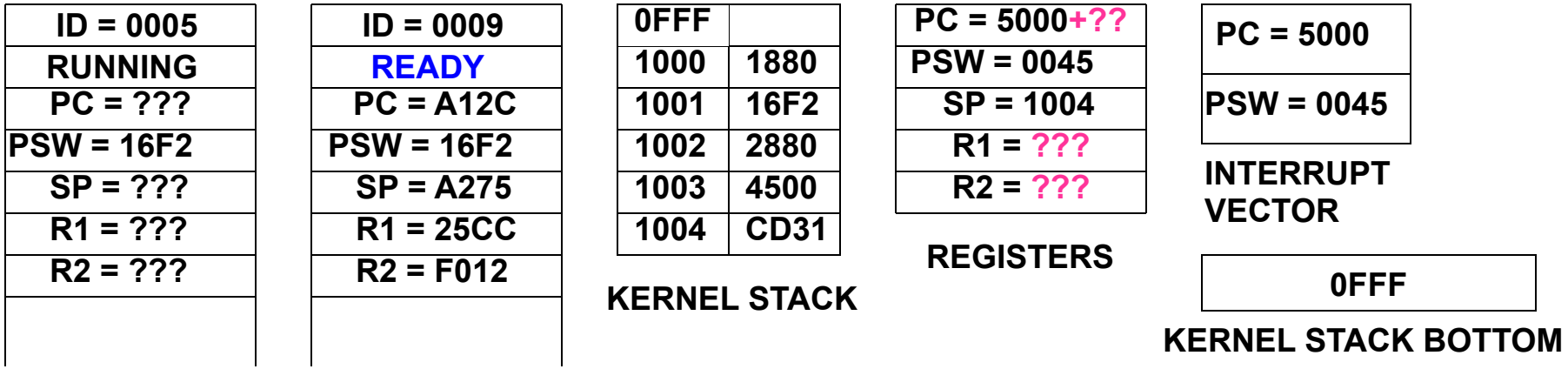
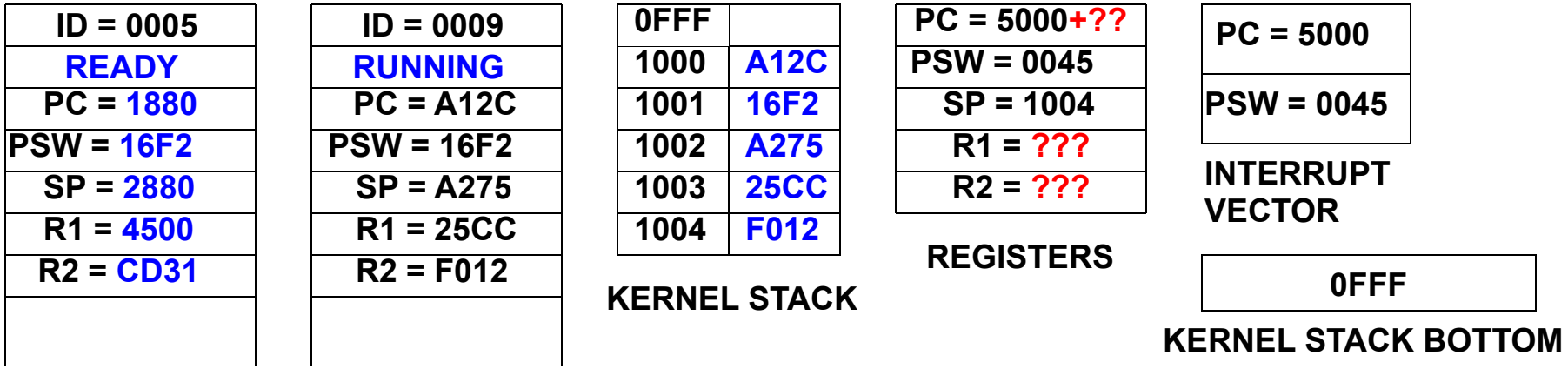


Figura 4 – Situazione prima di invocare lo scheduler, come nell'Esempio 1 ma con il processo 9 che è diventato READY durante l'esecuzione dell'interrupt handler della TRAP:



Lo scheduler seleziona il processo 9, e da luogo al context switch:
- il PCB del processo 5 assume i valori dei registri salvati nel kernel stack, lo stato diventa READY.
- nel kernel stack vengono inseriti i valori salvati nel PCB del processo 9.
- nel PCB del processo 9 lo stato diventa RUNNING (in terminologia UNIX sarebbe kernel running).
I valori di PC, R1, R2 cambiano.

Figura 5 - Situazione dopo che 5 diventa READY e 9 RUNNING:



Il context switch si completa con il salvataggio dei registri generali e la successiva IRET, operazioni che vanno effettuate anche quando non c'è il context switch, come nell'Esempio 1.

Figura 6 – Situazione dopo il ripristino di R1, R2 per il processo 9:

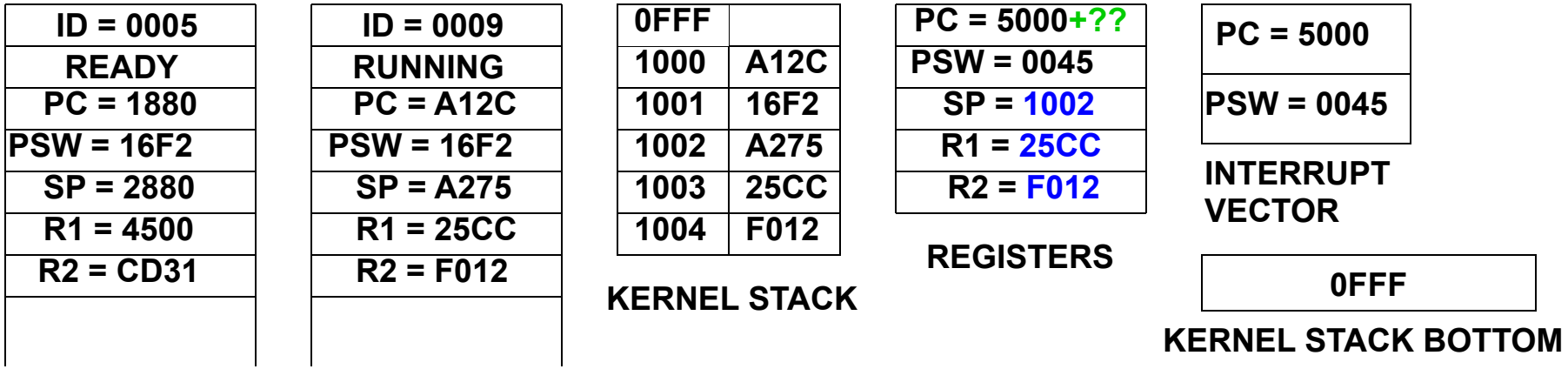
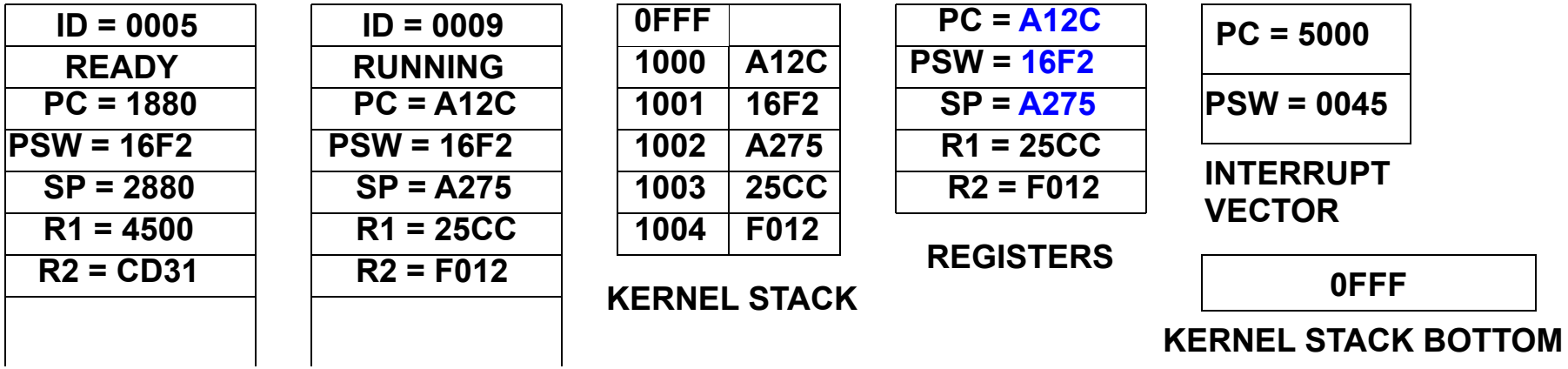


Figura 7 – Situazione dopo la IRET.
Notare che secondo la terminologia UNIX il processo 9 passa da kernel running a user running.



Osservazione. Con l'architettura dell'Esempio 3, il context switch tra un processo **P** (processo 5 dell'Esempio 3) ed un processo **P'** (proceso 9 dell'Esempio 3) si implementa in due fasi:

1. I valori dei registri sul kernel stack, relativi a **P**, vengono salvati nel PCB di **P**.
2. Sul kernel stack vengono caricati i valori dei registri precedentemente salvati sul PCB di **P'**.

Il punto 1 sostituisce il caricamento dei registri con i valori sul kernel stack che viene fatto se non c'è il context switch.

Il punto 2 precede il caricamento dei registri con i valori sul kernel stack, che, a questo punto, non sono più quelli di **P** ma quelli di **P'**.

4.7: CREAZIONE E TERMINAZIONE DI PROCESSI

- Il S.O. mette a disposizione un'apposita **system call per la creazione di processi**.

  Esempi: **fork** in UNIX/Linux, **CreateProcess** in APIWin32.

- In alcuni S.O., come UNIX/Linux, tra il processo che esegue la sysem call, detto **processo padre**, ed il processo creato, detto **processo figlio**, si crea una **relazione gerarchica**.

  In questo caso, un processo può avere più figli ma un solo padre. I figli possono avere, a loro volta, figli.

- Quale programma esegue il processo appena creato? Dipende dal S.O. considerato:

  - nel caso della **CreateProcess** il programma è passato come parametro;

  - nel caso della **fork**, il figlio esegue il medesimo programma del padre (ciò può essere sorprendente!), ma il programma può essere cambiato con la system call **exec**.

Esempio di **fork**

```
#include<stdio.h>

main(  ){
   int p = getpid( ); /*getpid is a system call that returns the Process ID*/
   printf("sono %d, ora mi  clono\n",p);

   int x = fork( ); /*la fork restituisce 0 al figlio e l'ID del figlio al padre*/
       /*ora abbiamo 2 processi (2 PCB), il loro programma è uguale*/
   int y = 10; /*istruzione eseguita da entrambi padre e figlio*/
   printf("y vale %d\n",y); /*anche questa*/

   if(x==0){   /*ramo eseguito solo dal figlio*/
      int f = getpid( );
      printf("sono il figlio, cioè %d\n",f);
   }

   else{        /*ramo eseguito solo dal padre*/
      printf("sono %d, mio figlio è %d\n",getpid( ),x);
   }
}
```

Risultato dell'esecuzione:

```
simone$a.out
sono 5585, ora mi clono
y vale 10
sono il figlio, cioè 5586
y vale 10
sono 5585, mio figlio è 5586
simone$
```

- In UNIX il compiler C genera il file **a.out**. Digitando **a.out** dalla shell (**simone$** è il prompt) si chiede l'esecuzione.

- La **fork** restituisce 0 al figlio ed il PID del figlio al padre.

- Dopo la **fork**, **il padre ed il figlio sono due processi distinti ma *quasi* uguali**:

  1. entrambi hanno il proprio PCB (quello del figlio è creato dalla **fork**), i PID sono diversi, la **getpid** da risultati diversi;
  2. i registri generali hanno il medesimo valore, eccetto il registro usato per il risultato della system call;
  3. i registri **PC**, **PSW**, **SP** hanno i medesimi valori;
  4. le aree testo, dati e stack sono distinte ma hanno il medesimo contenuto. Ognuno ha la propria **y**, in entrambe le aree dati **y** ha valore 10 ed entrambi i processi stampano a video "**y vale 10**". Nelle due aree dati la **x** assume valori diversi (vedi punto 1), quindi i 2 processi eseguono rami dell'if-else diversi.

- Testare il risultato della **getpid** è il modo classico per far eseguire a padre e figlio porzioni di programma diverse.

- Siamo certi che dopo la **fork** padre e figlio sono entrambi READY, ma non possiamo fare assunzioni sulle velocità di esecuzione relative, che dipendono da:

  - algoritmo di scheduling
  - parametri di scheduling
  - numero e tipo di altri processi in esecuzione

  In altre parole, le esecuzioni mostrate nella prossima slide sono tutte possibili.

  (in blu le **printf** del padre, in rosso quelle del figlio)

```
simone$a.out
sono 5585, ora mi clono
y vale 10
sono il figlio, cioè 5586
y vale 10
sono 5585, mio figlio è 5586
simone$
```

```
simone$a.out
sono 5585, ora mi clono
y vale 10
sono 5585, mio figlio è 5586
y vale 10
sono il figlio, cioè 5586
simone$
```

```
simone$a.out
sono 5585, ora mi clono
y vale 10
y vale 10
sono 5585, mio figlio è 5586
sono il figlio, cioè 5586
simone$
```

```
simone$a.out
sono 5585, ora mi clono
y vale 10
y vale 10
sono il figlio, cioè 5586
sono 5585, mio figlio è 5586
simone$
```

```
simone$a.out
sono 5585, ora mi clono
y vale 10
y vale 10
sono il figlio, cioè 5586
sono 5585, mio figlio è 5586
simone$
```

```
simone$a.out
sono 5585, ora mi clono
y vale 10
y vale 10
sono 5585, mio figlio è 5586
sono il figlio, cioè 5586
simone$
```

Esempio di **exec** e **wait**

Assumiamo che **b.out** sia il programma di Pag. 18 compilato

```
#include<stdio.h>
main(  ){
    int s;
    int p = getpid( );
    printf("sono %d, ora mi clono\n",p);
    int x = fork( );

    if(x==0){
        printf("sono %d\n" , getpid( ));
        execl("b.out"  , "b.out" , NULL); /* il figlio si cambia il programma */
   }
    else{
        int  w = wait(&s); /* il padre aspetta la terminazione del figlio */
        printf("sono %d, mio figlio %d ha terminato\n"  , getpid( ) , x);
        printf("il suo stato e' %d \n"  , s);
    }
}
```

Unica esecuzione possibile (unica modulo i PID):

```
simone$a.out
sono 6389, ora mi clono
sono 6390
dammi una parola: ttt
dammi una parola: yyy
dammi una parola: eee
dammi una parola: www
dammi una parola: ggggg
parola numero 0: ttt
parola numero 1: yyy
parola numero 2: eee
parola numero 3: www
parola numero 4: ggggg
sono 6389, mio figlio 6390 ha terminato
il suo stato e' 0
simone$a.out
```

- La **system call exec** serve per “cambiarsi il programma”.

- La **execl** è una delle funzioni di libreria per invocarla. In questo esempio il figlio usa **execl** per eseguire il programma **b.out**. Tralasciamo di commentare il significato dei 3 parametri.

- Le altre funzioni che invocano la **exec** usano parametri diversi in numero e/o tipo.

- Effetti principali della **exec**:
  - il processo ottiene nuove aree testo, dati e stack,
  - i registri **PSW**, **PC**, **SP** assumono valori nuovi,
  - i registri generali vengono “azzerati”,
  - i file vengono chiusi ed i dispositivi rilasciati.

- La **system call wait** è una richiesta al S.O. di mettersi in stato di WAITING. L'evento atteso è la terminazione di un processo figlio. (Nel nostro esempio c'è solo un figlio.)

- La **wait** ha un parametro: un pointer ad una variabile dove viene memorizzato il valore che codifica la causa della terminazione del figlio. Il valore **0** significa che la terminazione è regolare.

- Quando un processo termina, esegue la **system call exit** e va in stato ZOMBIE. Il compiler C inserisce la **exit(0)** al termine del programma, anche se il programmatore la omette. La **exit** ha un parametro che codifica la causa della terminazione (**0** = regolare).

- Quando un processo esegue la **exit**, l'interrupt handler controlla se la terminazione è attesa dal padre. Se è cosi il padre passa da WAITING a READY.

Non tutti i processi sono uguali! Esempio, in UNIX abbiamo processi utente, processi deamon e processi kernel.

Gli **user process** sono associati ad un utente che lavora ad un terminale. Due esempi:

- l'eseguibile **a.out** ottenuto compilando il programma di Pag. 18 e che chiediamo alla shell di mandare in esecuzione.
- la **shell**. Vediamo come lavora:

- esecuzione classica - comando **a.out**:

  la **shell** fa una **fork**, il figlio fa la **exec** su **a.out**, il padre (cioè la **shell**) fa la **wait** ed attende la terminazione del figlio, quando ciò avviene si mette in attesa di un altro comando da parte dell'utente;

- esecuzione "in background" – comando **a.out &**:

  la **shell** fa una **fork**, il figlio fa la **exec** su **a.out**, il padre (cioè la **shell**) si mette in attesa di un altro comando da parte dell'utente.

I **daemon process** non sono associati a nessun utente:

- tipicamente non terminano,
- eseguono funzioni vitali per il sistema: controllo email, gestione coda di stampa, controllo richieste web page, …
- eseguono in modalità user,
- esempio: il processo **getty** che, iterativamente:
  - gestice il processo di login di un utente al terminale,
  - se la login va a buon fine, fa una **fork**. Il figlio fa la **exec** del programma di **shell**, il padre (cioè **getty**) fa la **wait** e si mette in attesa che il figlio termini.

I **kernel processes** sono daemon che eseguono in modalità kernel, quindi:

- accedono a procedure/strutture del kernel senza dover invocare le system call.

- sono “potentissimi”, per esempio possono controllare i propri parametri di scheduling.

- Esempi:

  - Nei sistemi con paginazione della memoria, lo **stealer process** viene svegliato ad intervalli regolari e fa lo swap out delle pagine che recentemente non hanno avuto accessi.

  - Nei sistemi con swapping, lo **swapper process** viene svegliato ad intervalli regolari e fa lo swap out dei processi con maggior anzianità in memoria, e lo swap in dei processi da più tempo in READY SWAPPED.

Al momento del boot è necessario che venga eseguito del **codice di inizializzazione che "costruisce" almeno un processo**, dal quale possono discendere tutti gli altri.

Esempio di UNIX System V:

- al momento del boot viene creato il **processo 0**, detto **swapper**, che è un kernel process;
- lo swapper crea il **processo 1**, detto **init**, il quale si crea un user context, si cambia modalità in user, e diventa un daemon "normale", cioè non kernel;
- lo **swapper** crea altri processi kernel (esempio lo stealer nei sistemi con paginazione) e, nei sistemi con swapping, fa quanto detto nella slide precedente;
- Il processo **init** crea altri daemon, per esempio un **getty** per ogni terminale (quindi un solo **getty** nei PC) e si mette in WAITING con la **wait** attendendo che terminino.

Circostanze che causano la **terminazione** di un processo:

- I S.O. offrono ai processi la possibilità di chiedere al S.O. di terminare, mediante un'apposita **system call**. Esempi: **exit** in UNIX e **ExitProcess** in APIWin32.

- L'interrupt handler delle **eccezioni** può forzare la terminazione del processo. Accade, ad esempio, in caso di errori di overflow e violazioni di protezione di memoria.

- I S.O. offrono ai processi la possibilità di chiedere al S.O. di farne terminare altri, mediante un'apposita **system call**. Esempi: **kill** in UNIX e **TerminateProcess** in APIWin32. La sysem call va a buon fine solo quando il killer ha il diritto di eseguire l'operazione. Esempio: in UNIX il padre può sempre uccidere il figlio.

Osservazione: quando un processo termina:

- **rilascia tutte le risorse**: memoria, file, CPU (non è più schedulabile).

- La distruzione del PCB non è sempre immediata. Esempio: in UNIX il PCB di un processo ucciso è cancellato dal padre. Prima che ciò avvenga il PCB è in stato ZOMBIE.

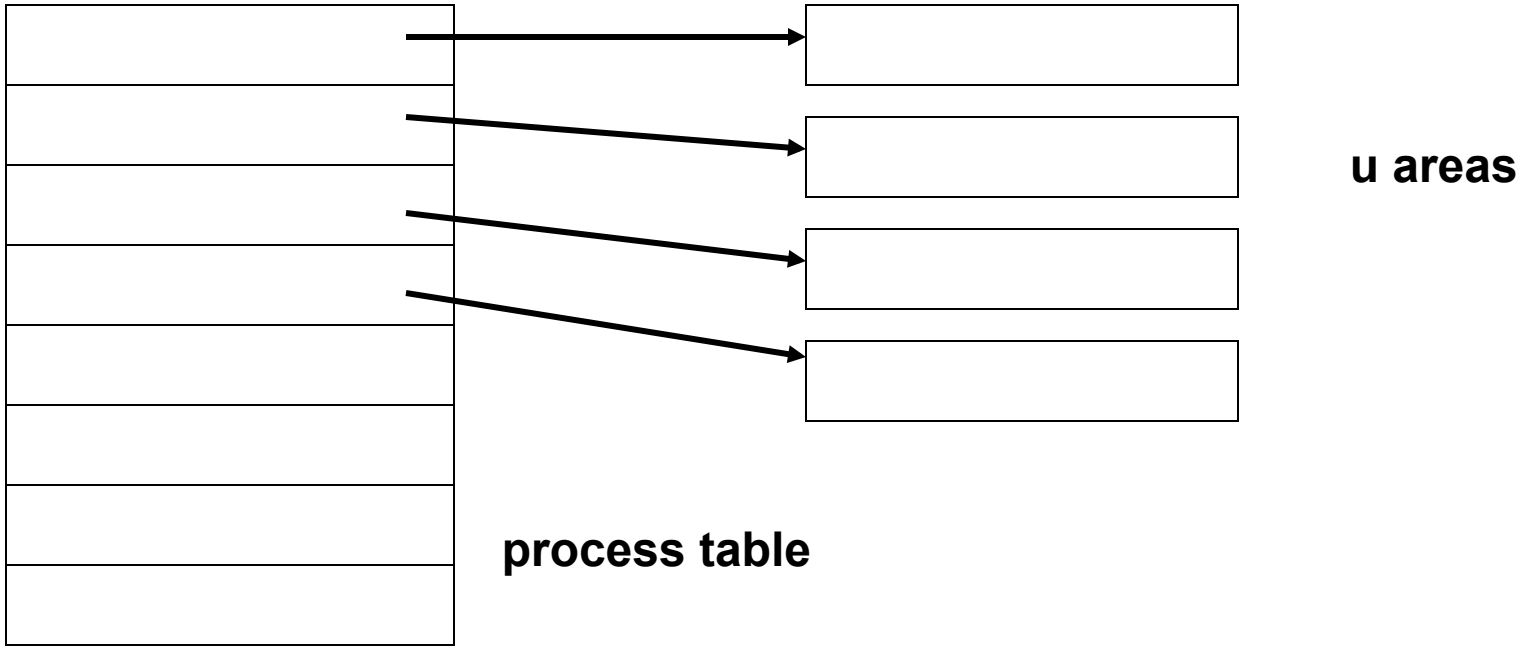
- Se un processo termina ed ha figli? In UNIX i figli non vengono uccisi, ma diventano figli del processo **init**. Anche in Windows i figli non vengono uccisi.

4.8: SYSTEM CALL IN UNIX

- Oltre al PCB, UNIX assegna ad ogni processo una **user area**, detta anche **u area**. Il PCB ha un pointer alla u area.
- Le voci elencate a Pag 43, 44 sono inserite:

  - nella u area se servono solo quando il processo è running (eg file descriptor, pointer a info su aree testo, dati e stack),

  - nel PCB, se servono sempre (e.g. stato, parametri di scheduling, pointer alla u area).
- I PCB sono allocati in una **Process Table**.



La **u area** serve anche per memorizzare i **parametri**, il **risultato** e l'eventuale **codice di errore** delle **system call**.

- Abbiamo già visto le system call a livello di linguaggio macchina (TRAP). Possiamo ora vedere come vengono trattate le system call a livello di linguaggio ad alto livello, usando le funzioni di libreria.

- Supponiamo che il **main** di un programma in C esegua la chiamata seguente:

```
int fd = creat("file_di_prova",O_RDWT);
```

La variabile **fd** assume un valore **n>=0** se la chiamata va a buon fine, **-1** altrimenti.

1. La chiamata

   int fd = creat(“file_di_prova” , O_RDWT);

è una normalissima chiamata di funzione. Possibile compilazione in linguaggio macchina Motorola 68000:

```
58:  mov  &0x1b6, %sp      # move 1b6 onto stack
5e:  mov  &0x204, -%sp     # decrement sp and
                           # move “fileDiProva” onto stack
64:  jsr    0x7a           # call C library for creat
```

- **mov** e **jsr** sono i codici della move e jump to subroutine.
- **0x** denota che il parametro che segue è in esadecimale.
- **1b6** in esadecimale = **0666** in ottale = codice di **O_RDWT.**
- **%sp** denota il registro **SP**.
- Lo stack nel Motorola 68000 cresce verso il basso.
- la chiamata di funzione si implementa con il pushing dei parametri sullo stack ed il jump alla subroutine.

2. La funzione di libreria esegue la TRAP con valore 8, assumendo che 8 identifichi la **creat**. Ecco l'assembly della **creat**:

```
7a: mov  &0x8, %d0    # move value 8 into data register 0
7c: trap &0x0         # TRAP – the parameter is in data register 0.
7e: bcc  &0x6 <86>    # if carry bit clear branch to address 86
80: jmp  0x13c        # jump to address 13c
86: rts               # return from subroutine.
```

- **%d0** denota il data register **0**.
- Il parametro della **trap** è passato in un registro che nell'architettura considerata si può scegliere, vale a dire **0** in questo caso.
- La **creat** prepara il parametro della **trap**, esegue la **trap** invocando il S.O. ed aspetta che il S.O. restituisca il controllo. Vedremo dopo quel che fa la **creat** quanto riottiene il controllo.

3. Sappiamo già che l'interrupt handler della **creat** salva i registri generali, esegue le proprie funzioni e chiama lo scheduler. "Esegue le proprie funzioni" va dettagliato:

   - L'interrupt handler copia nella u area del processo i parametri che la **main** aveva posto sullo stack.

   - L'interrupt handler esegue le proprie funzioni, memorizzando il risultato della system call ed il codice di errore nella u area.

- In caso di errore, prima di chiamare lo scheduler:

  viene settato a 1 il **Carry Bit** (è un bit del **CC**) della **PSW**

  viene inserito il codice di errore nel data register **0**

- Se non c'è errore, prima di chiamare lo scheduler:

  il Carry Bit della **PSW** rimane a **0**

  il risultato viene posto nel data register **0**.

4. Quando il processo riprende il controllo, riparte la funzione di libreria dall'istruzione ad indirizzo **7e**:

```
7a: mov   &0x8, %d0    # move value 8 into data register 0
7c: trap  &0x0         # TRAP - parameter in data register 0.
7e: bcc   &0x6 <86>    # if carry bit clear branch to address 86
80: jmp   0x13c        # jump to address 13c
86: rts                # return from subroutine.
```

La **bcc** testa il carry bit. Il codice 6 coincide con carry bit = **0**.

- Carry bit 0 → la **creat** termina e la **main** esegue dall'istruzione successiva alla **jsr**. Il valore del data register **0** è da assegnare alla variabile **fd**.
- Carry bit 1 → la **creat** chiama la subroutine che gestisce l'errore.

5. Analizziamo la gestione dell'errore:

13c: mov %d0, &0x20e # move data register 0 to location 20e
142: mov &0x-1, %d0 # move -1 to data register 0
144: rts

- Il codice dell'errore viene inserito all'indirizzo **20e**, vale a dire l'indirizzo della variabile **errno**.
- il risultato **-1** (errore) viene posto nel data register **0**,
- **main** riprende ad eseguire. Il valore del data register **0** (cioè **-1**) è da assegnare alla variabile **fd**. **main** può analizzare l'errore, trovandolo all'indirizzo **20e**.