# 5. FUNZIONALITA' DEL CALCOLATORE

Per comprendere come il SO possa realizzare le proprie funzioni è necessario conoscere le funzionalità fornite dal calcolatore a livello Hardware. Dato che esistono molti tipi di calcolatori e il SO LINUX è in grado di funzionare su molti di essi, in questo paragrafo non viene descritto un calcolatore specifico ma vengono analizzati i meccanismi fondamentali che, sia pure con moltissime variazioni, esistono in tutti i tipi di Hardware. Tutti i processori moderni realizzano in qualche forma i meccanismi descritti nel seguito.

Si tenga presente che la descrizione dei meccanismi fornita qui è minimale e solamente orientata a comprendere la realizzazione del SO; una descrizione più accurata dovrebbe fare riferimento a dettagli relativi a specifici processori ed è oggetto di testi sulle architetture dei calcolatori.

### 1. Meccanismo fondamentale di funzionamento di un processore

Il meccanismo base di funzionamento di un **processore** è illustrato in figura 1. Il processore è l'elemento attivo che opera eseguendo le istruzioni in linguaggio macchina che sono contenute nella memoria.

La **memoria** è costituita da una sequenza di **celle** o **parole**, ognuna delle quali è capace di contenere un certo numero di **bit**. Tale numero è detto lunghezza della parola. Ogni parola è identificata da un numero detto **indirizzo** (una trattazione più approfondita della memoria verrà svolta nel capitolo dedicato a questo argomento).

Il processore contiene al proprio interno dei **registri**, capaci anch'essi di memorizzare un certo numero di bit come le celle di memoria. Il numero e il tipo dei registri posseduto varia da un tipo a un altro di processore, ma praticamente tutti i processori possiedono un registro particolare detto **contatore di programma** o **PC** (**Program Counter**), mostrato in figura 1, che è destinato a contenere l'indirizzo della parola di memoria che contiene la prossima istruzione da eseguire.

Il processore opera eseguendo un ciclo fondamentale di interpretazione delle istruzioni, che può essere schematizzato nel modo seguente:

1. legge in memoria l'istruzione contenuta all'indirizzo contenuto nel registro PC;
2. interpreta l'istruzione letta e la esegue;
3. determina il nuovo valore di PC e torna al passo 1.

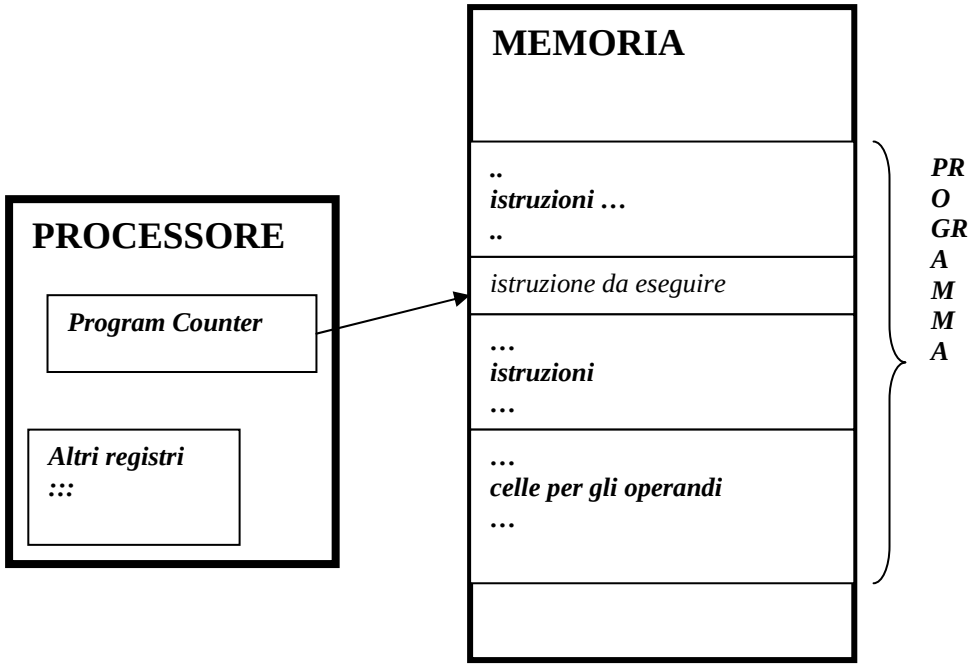



***Figura 1 – Funzionamento di Processore e Memoria***

E' importante tenere presenti alcune osservazioni rispetto a questo ciclo.

- all'inizio (cioè all'avviamento del processore) il registro PC deve contenere l'indirizzo della prima istruzione eseguibile del programma;
- durante l'interpretazione dell'istruzione il processore può accedere a celle di memoria che contengono gli operandi; la memoria contiene quindi sia le istruzioni che gli operandi del programma;
- nel determinare il nuovo valore del PC normalmente il processore suppone che la prossima istruzione sia contenuta nella cella successiva di memoria,

quindi si limita ad incrementare il valore del PC stesso; però, se l'istruzione appena eseguita è un salto, allora il prossimo valore del PC viene determinato dall'istruzione stessa.

Le istruzioni che possono essere eseguite sono molto elementari, ad esempio:

- traferisci il contenuto di una cella di memoria in un registro o viceversa
- somma il contenuto di due registri (o analogamente altre operazioni aritmetiche e logiche)
- salta all'esecuzione di un'istruzione posta a un certo indirizzo (questa istruzione modifica il registro PC)
- come sopra, ma il salto deve essere eseguito solamente se vale una certa condizione (salto condizionato); una tipica condizione è che un certo registro abbia un contenuto uguale (oppure maggiore o minore) di zero;

Le istruzioni di salto (condizionato e non) sono di fondamentale importanza per realizzare in linguaggio macchina i costrutti di controllo (cicli, if-then, ecc…) di un linguaggio come il C. Si tenga presente che, anche se la maggior parte dei processori possiede un numero elevato di istruzioni, un linguaggio macchina completo, nel senso di un linguaggio sufficiente per realizzare tutte le funzioni richieste dalla traduzione di qualsiasi programma in linguaggio C, può essere costituito da 5 o 6 istruzioni solamente, di cui almeno una deve essere un salto condizionato. Moltissime operazioni possono infatti essere realizzate utilizzando istruzioni più elementari (ad esempio, la sottrazione tramite negazione e somma; la moltiplicazione tramite somme ripetute, ecc…).

I diversi processori differiscono moltissimo in base al tipo di istruzioni che possono eseguire, ma possiamo affermare che tutti i processori operano secondo lo schema illustrato sopra e che i tipi di istruzioni che eseguono sono sufficienti per svolgere tutte le funzioni richieste da un programma eseguibile ottenuto per traduzione (compilazione e collegamento) di un programma qualsiasi scritto in un linguaggio di alto livello (ad esempio, C).

Nel seguito noi dovremo spesso rappresentare il contenuto della memoria facendo riferimento a un programma eseguibile ottenuto per compilazione di un programma in linguaggio C; in generale useremo le seguenti convenzioni:

- per ogni funzione esiste un'area di memoria che contiene il codice, cioè le istruzioni macchina, ottenute per traduzione della funzione;
- con il termine indirizzo della funzione intenderemo l'indirizzo della prima istruzione eseguibile della funzione;
- nelle figura indicheremo la funzione direttamente nella memoria, intendendo che il suo codice è in memoria,
- la rappresentazione dei dati delle funzioni è trattata nel prossimo paragrafo

Inoltre, dal punto di vista della realizzazione grafica delle figura, la memoria sarà sempre rappresentata come un vettore di celle con l'indirizzo 0 in alto e l'indirizzo massimo in basso; in questo modo l'ordinamento delle istruzioni in memoria è dall'alto verso il basso.

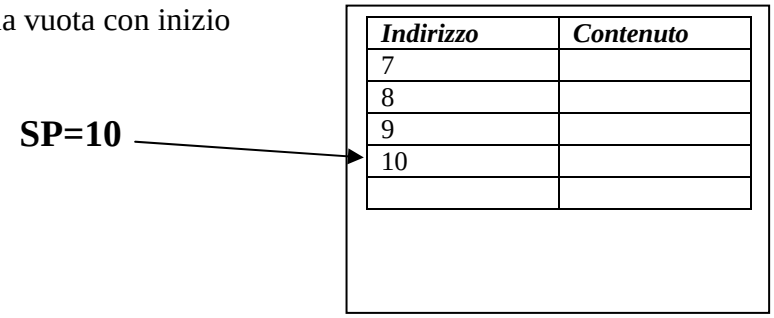
**2. Pila, puntatore alla pila e variabili locali delle funzioni**

I processori moderni possiedono in generale un meccanismo detto **pila** (**stack**). Una pila è semplicemente una zona della memoria nella quale la scrittura e la lettura si svolgono secondo delle regole ben precise, basate sull'esistenza di un particolare registro detto **puntatore alla pila** (**stack pointer** – **SP**). Il funzionamento della pila si basa sulle seguenti regole:

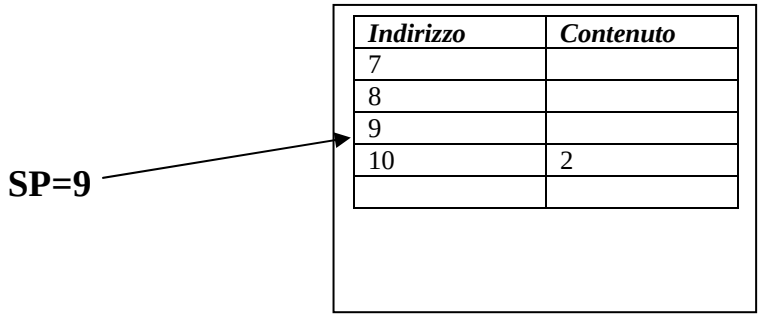
1. inizialmente SP contiene l'indirizzo della parola di memoria dalla quale si è deciso di far iniziare la pila (base della pila);
2. l'operazione di scrittura di un dato sulla pila, detta **PUSH(dato),** consiste nello scrivere il dato nella parola indirizzata da SP e decrementare SP (pila crescente dagli indirizzi grandi verso indirizzi piccoli, cioè, graficamente, dal basso verso l'alto; questa convenzione è utilizzata in questo testo in conformità con LINUX – si potrebbe adottare anche la regola opposta);
3. l'operazione di lettura di un dato dalla pila, detta **POP**, consiste nell'incrementare il contenuto di SP e leggere il contenuto della parola indirizzata;
4. gli unici modi concessi per modificare la pila sono le operazioni PUSH e POP.

In figura 2 è illustrato il funzionamento di una pila tramite una sequenza di operazioni.

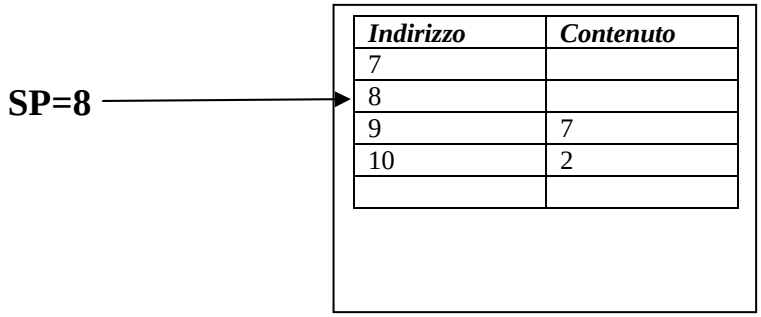
Stato Iniziale:Pila vuota con inizio nella cella 10




***PUSH (2)***




***PUSH (7)***




***POP***

***(legge il valore 7)***

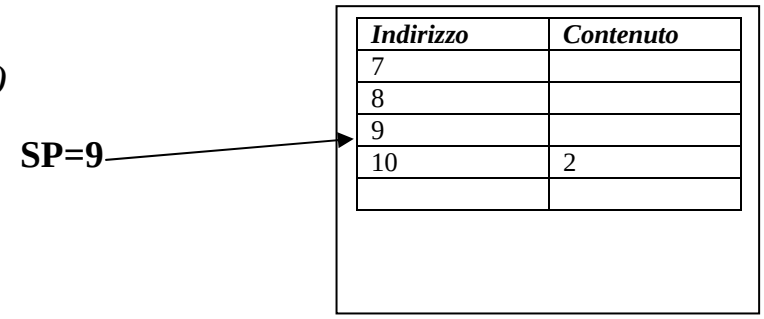
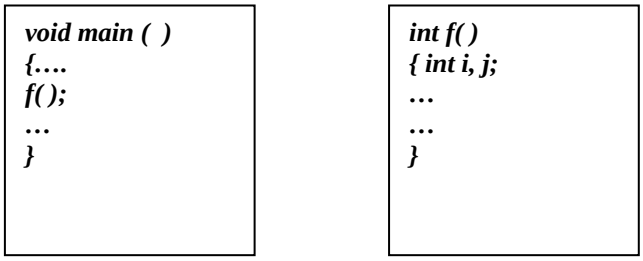


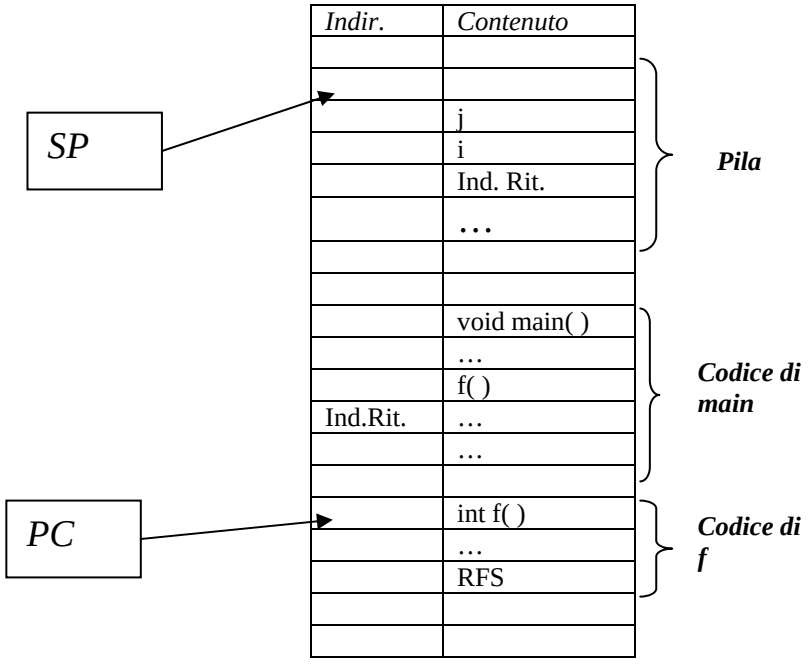

***Figura 2 – Funzionamento di una Pila***

Per rendere tale figura più leggibile, in figura 2 si è cancellato il contenuto di una cella letta da una POP, ma in realtà tale cancellazione non è necessaria, in quanto la prossima PUSH scriverà sopra il contenuto dell'ultima cella letta da una POP.

La pila costituisce un meccanismo importante nella realizzazione della struttura a funzioni di una linguaggio come il C. Si consideri infatti il programma di figura 3a, nel quale un main invoca una funzione f( ).




**a) Un programma e una funzione C**




**b) Situazione subito dopo l'invocazione della funzione f( ) da parte di main**

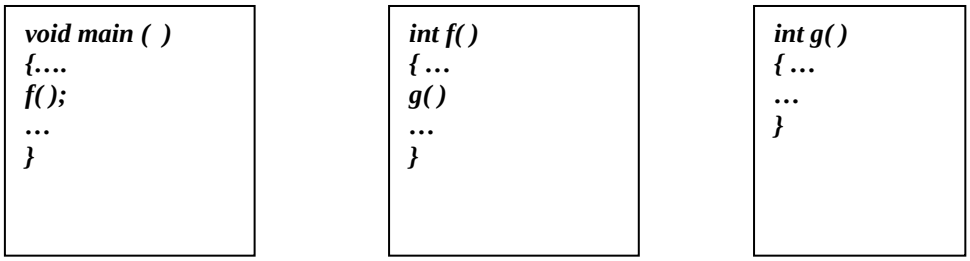
***Figura 3 – Uso della pila per le funzioni di un programma***

Quando il main invoca la funzione f vengono allocate sulla pila sia le variabili automatiche della funzione che l'indirizzo dell'istruzione successiva al salto alla funzione, detto **indirizzo di ritorno**. La situazione risultante è mostrata in figura 3b. Durante l'esecuzione la funzione f opera sulle variabili I e J allocate sulla pila.

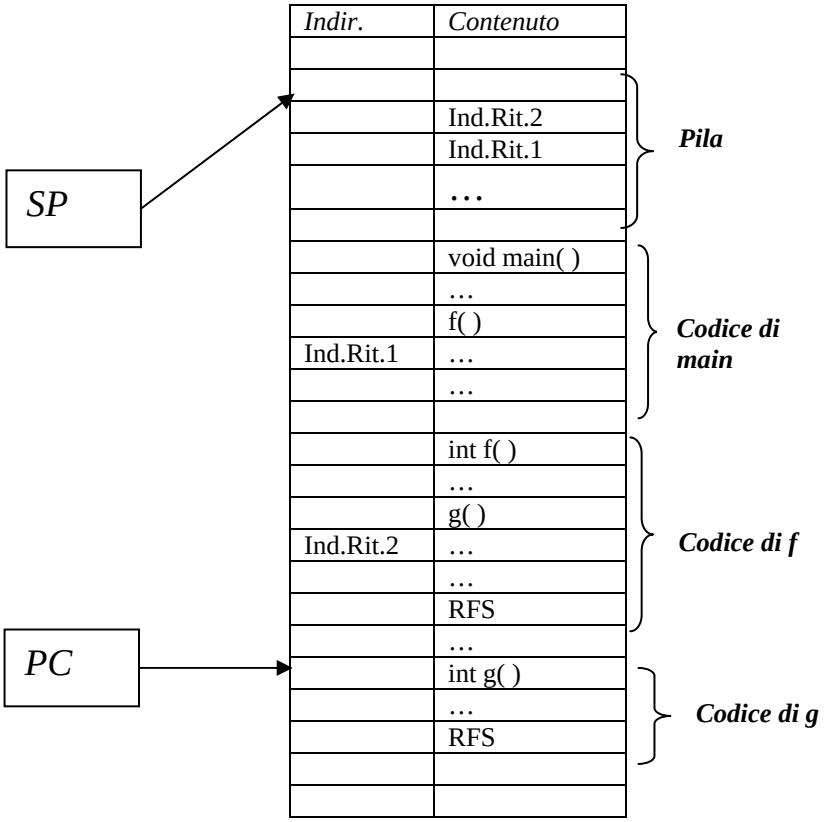
Quando l'esecuzione della funzione termina vengono cancellate dalla pila le variabili automatiche, perchè non servono più, e poi viene eseguita un'istruzione, che esiste in diverse forme in tutti i processori e che chiameremo qui **RFS** (return from subroutine), la quale esegue il prelievo del valore presente in cima alla pila e il suo caricamento nel registro PC. In questo modo l'esecuzione riprende nel main dall'istruzione successiva all'invocazione della funzione f( ), come richiesto da una corretta interpretazione del programma C. L'istruzione RFS corrisponde quindi, nel codice eseguibile dal processore, all'istruzione return delle funzioni C.

L'utilizzazione della pila per la realizzazione della nozione di funzione presente in tutti i linguaggi di alto livello è universale nei processori moderni, perchè tale meccanismo realizza ottimamente le caratteristiche fondamentali di gestione delle variabili automatiche e del **funzionamento nidificato e ricorsivo** proprio di tali funzioni. In figura 4 è illustrato tale funzionamento, mostrando come cresce la pila quando vengono attivate due funzioni nidificate. La figura mostra che, nel momento in cui la funzione f invoca la funzione g, sulla pila viene salvato il nuovo indirizzo di ritorno (in questa figura sono state eliminate le variabili automatiche delle funzioni, per semplicità). Partendo dallo stato mostrato in figura, è chiaro che al termine dell'esecuzione della funzione g nel registro PC verrà ricaricato il valore Ind.Rit.2 e l'esecuzione riprenderà all'interno della funzione f dall'istruzione successiva all'invocazione g( ); sulla pila rimarrà solamente il valore Ind.Rit.1, che permetterà di ritornare al main quando terminerà anche f.

Questo argomento rientra nella trattazione del modello di esecuzione di un linguaggio di alto livello da parte di un processore, e non viene ulteriormente approfondito qui. Dal punto di vista degli argomenti relativi al SO è importante tenere presente che il meccanismo a pila permette di gestire correttamente l'invocazione nidificata e ricorsiva di funzioni.




**a) Un programma e due funzioni**




**b) Situazione subito dopo l'invocazione della funzione g( ) da parte di f**

***Figura 4 – Chiamata nidificata di 2 funzioni***

### 3. Accesso alle periferiche

Per svolgere funzioni utili il processore deve poter interagire con le periferiche del sistema. A questo scopo molti processori utilizzano delle istruzioni macchina specializzate, ad esempio IN e OUT, che fanno riferimento agli indirizzi dei registri delle periferiche. Ogni periferica infatti possiede alcuni registri che servono alla sua gestione; alcuni registri, detti **registri dati**, servono a contenere i dati che la periferica deve leggere o scrivere, altri registri, detti di **registri di controllo e stato**, servono a contenere delle indicazioni sulle operazioni che la periferica deve svolgere oppure sullo stato della periferica (ad esempio, periferica "pronta" oppure periferica "occupata").

Si tenga presente che il significato dello stato di pronto di una periferica è relativo alla possibilità, per il processore, di svolgere un'operazione di lettura o scrittura di un dato. Pertanto una periferica di ingresso, come una tastiera, è in stato di pronto quando un tasto è stato premuto e quindi esiste un carattere che il processore può leggere tramite l'istruzione IN; una periferica di uscita, come una stampante, è pronta se è disponibile per stampare un dato, in modo che il processore possa eseguire un'istruzione OUT verso di essa.

Tramite l'esecuzione delle istruzioni IN e OUT è possibile leggere e scrivere tali registri. Normalmente il programma che gestisce una periferica funziona secondo il seguente schema (detto schema a **controllo di programma**):

1. leggi il registro di stato della periferica (IN registro di stato)
2. se lo stato è "pronta", procedi, altrimenti ritorna a leggere il registro di stato
3. esegui l'operazione voluta (ad esempio, lettura di un dato dalla tastiera oppure invio di un dato alla stampante) (IN oppure OUT sul registro dati)

Ad esempio, in figura 5 è illustrata la sequenza di istruzioni di ingresso/uscita, cioè di interazioni tra il processore e una stampante durante l'esecuzione di un programma di stampa di un carattere "A" sulla stampante. Si noti che il processore è obbligato a restare nella situazione iniziale fino a quando la stampante non cambierè stato.

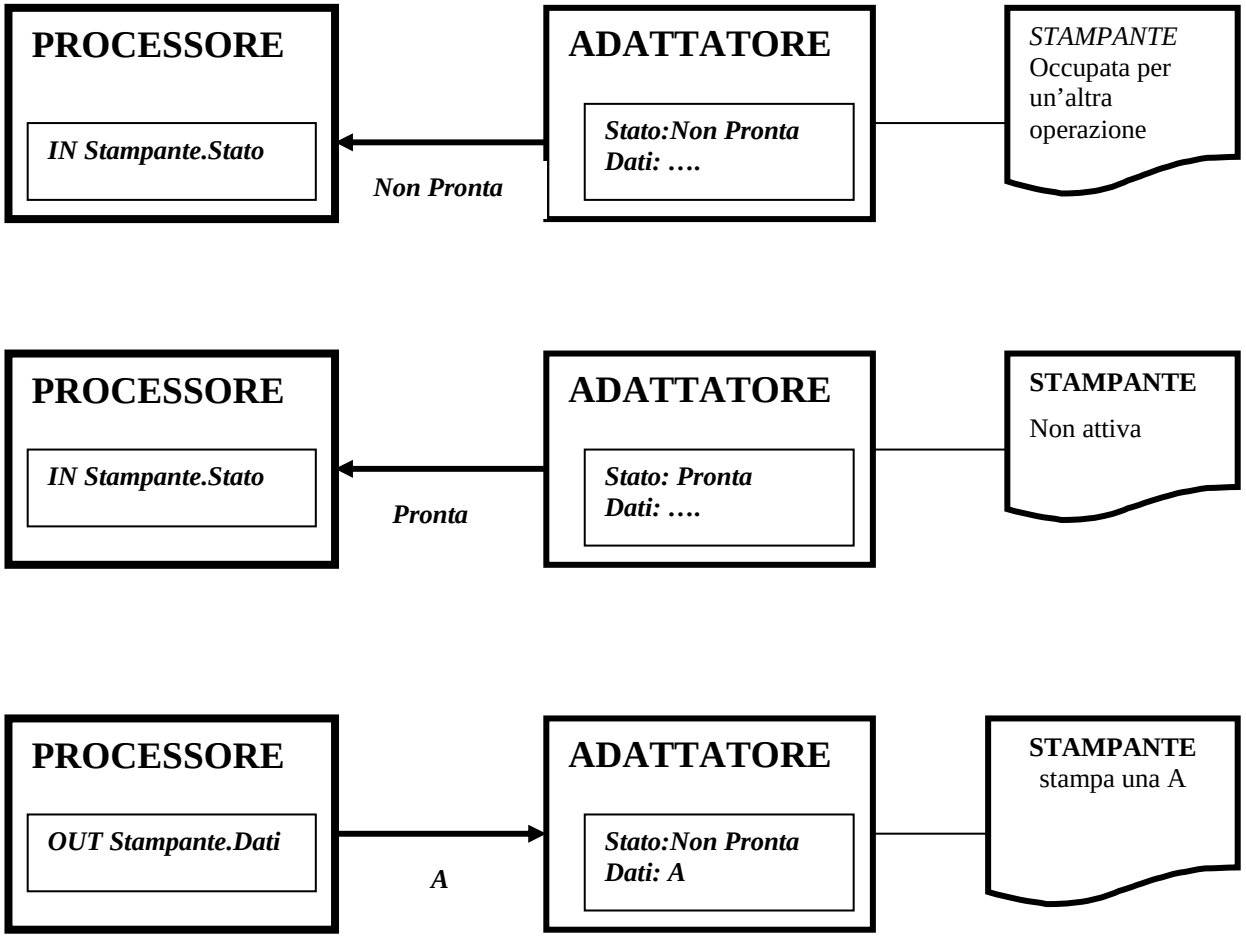



***Figura 5 – Stampa di una "A" a controllo di programma***

La figura mostra inoltre che è opportuno distinguere tra l'**adattatore** della stampante, che interagisce con il processore e contiene i registri di stato e dati, e la stampante vera e propria. Un approfondimento di questa distinzione e dei meccanismi di comunicazione tra gli oggetti riguarda lo studio del bus del calcolatore, ma non è rilevante per comprendere il Sistema Operativo. Dal punto di vista della terminologia, esiste molta confusione in questo campo e in molti testi e documentazioni di sistemi vengono utilizzati termini come interfaccia, controllore o anche semplicemente scheda

(board) al posto del termine adattatore. In questo testo utilizzeremo sempre il termine adattatore.

**4. Meccanismo di Interruzione (Interrupt)**

La gestione delle periferiche a controllo di programma ha il difetto di obbligare il processore a restare in un ciclo di attesa che la periferica diventi pronta; per questo motivo tutti i processori possiedono un meccanismo diverso **detto meccanismo di interrupt**: analizziamo prima come funziona tale meccanismo, poi vedremo perchè esso migliora grandemente la gestione delle periferiche rispetto al controllo di programma.

Il meccanismo di interrupt si basa sulla definizione di un insieme di eventi rilevati dall'Hardware (ad esempio, un particolare segnale proveniente da una periferica, una condizione di errore, ecc...) ad ognuno dei quali è associata una particolare funzione detta **gestore dell'interrupt** o **routine di interrupt;** quando il processore si accorge del verificarsi di un particolare evento E, esso esegue il salto all'esecuzione della funzione associata a tale evento. Quando la funzione termina, il processore riprende l'esecuzione del programma che è stato interrotto. Per poter riprendere tale esecuzione il processore ha salvato, al momento del salto alla routine di interrupt, sulla pila l'indirizzo della prossima istruzione di tale programma, in modo che, dopo l'esecuzione della routine di interrupt tale indirizzo sia disponibile per eseguire il ritorno. Come si vede, il meccanismo di interrupt è a tutti gli effetti equivalente ad un'invocazione di funzione, con la differenza che le funzioni normali sono attivate dal programma in esecuzione mentre le routine di interrupt sono attivate da eventi riconosciuti dal processore. Per un motivo che verrà spiegato più avanti, le routine di interrupt non utilizzano la stessa istruzione RFS per terminare, ma un'istruzione diversa, che chiameremo **IRET**. Per il momento possiamo pensare all'istruzione IRET come identica alla RFS.

L'uso di una meccanismo a pila uguale a quello utilizzato per le funzioni normali comporta in particolare che il verificarsi di un nuovo interrupt durante l'esecuzione di una routine di interrupt (**interrupt nidificati**) viene gestito correttamente, esattamente come l'annidamento delle invocazioni di funzioni.

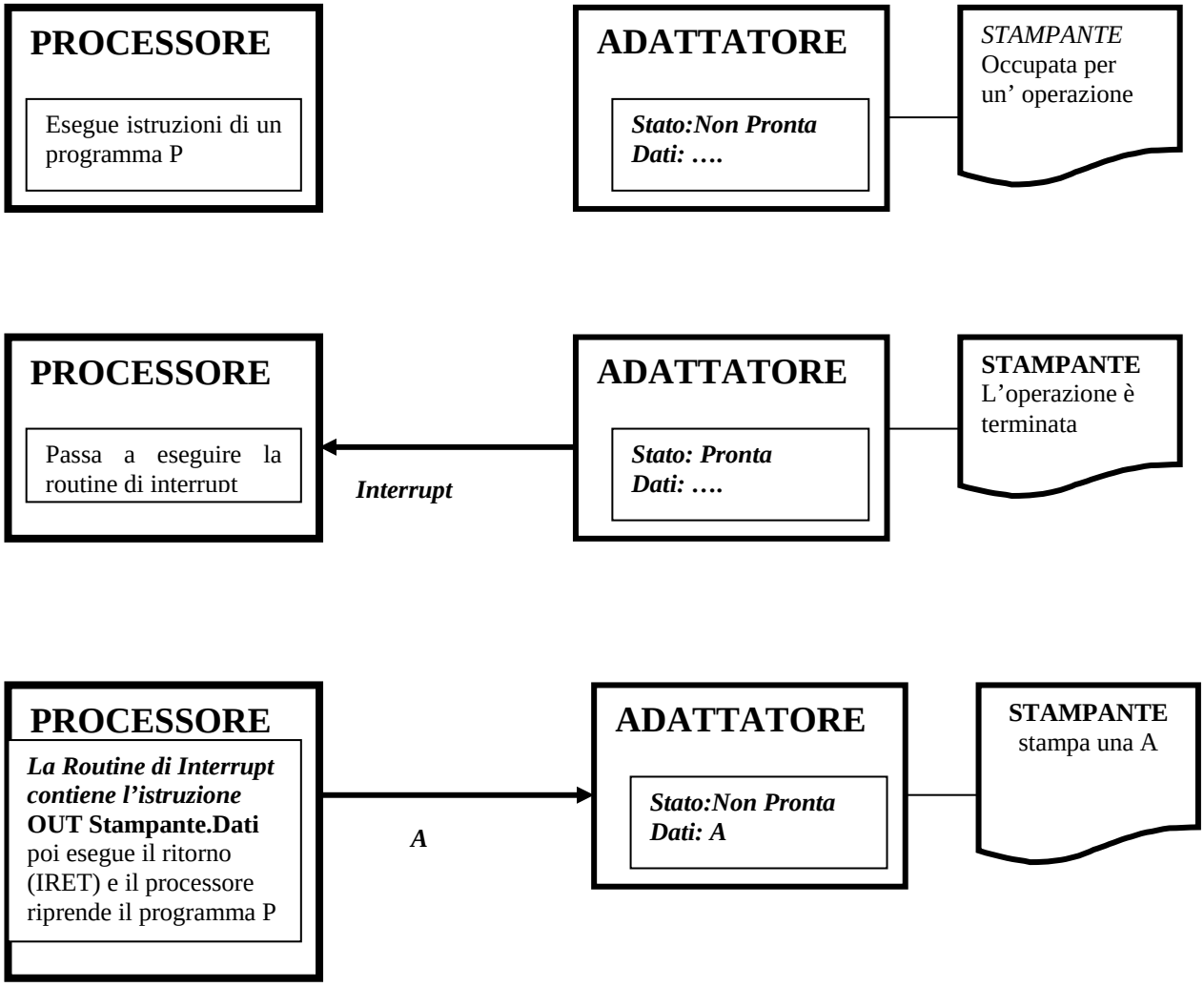



***Figura 6 – Stampa di una "A" tramite meccanismo di interrupt***

In figura 6 questo meccanismo è applicato alla stampa di un carattere A sulla stampante; la figura illustra un possibile svolgimento temporale degli eventi. La differenza fondamentale rispetto allo svolgimento di figura 5 consiste nell'assenza del ciclo di attesa iniziale; il processore viene impegnato solamente per svolgere funzioni utili.

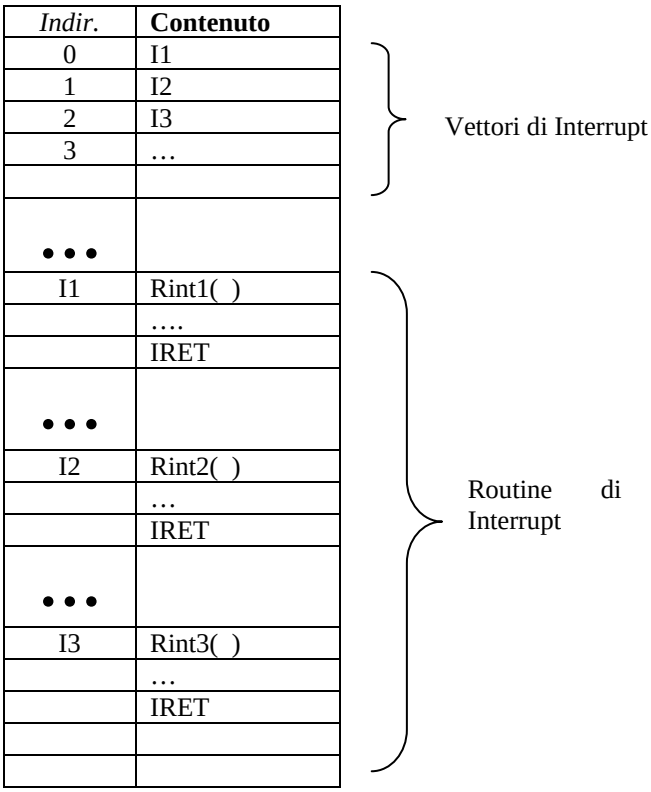



***Figura 7 – Vettori di interrupt e routine di interrupt***

Come fa il processore a sapere quale sia l'indirizzo della routine di interrupt che deve essere eseguita quando si verifica un certo evento? Tale indirizzo è scritto dal SO durante la fase di inizializzazione in una particolare cella di memoria, detta **vettore di interrupt**, che è nota al processore perchè è associata univocamente all'evento che ha causato l'interrupt. Quindi, quando si verifica un evento, il processore legge nel vettore di interrupt l'indirizzo della routine di interrupt, esegue la routine stessa e poi ritorna al programma interrotto. Ad esempio, in figura 7 sono mostrati alcuni vettori di interupt e le corrispondenti routine di interrupt; in tale figura si è adottata l'ipotesi, valida per molti processori, che i vettori di interrupt siano posizionati nelle celle iniziali della memoria.

Interrupt e gestione degli errori

Durante l'esecuzione delle istruzioni possono verificarsi degli errori che impediscono al processore di proseguire; un esempio classico di errore è l'istruzione di divisione con divisore 0, altri errori sono legati ad indirizzi di memoria non validi o al tentativo di eseguire istruzioni non permesse (più avanti, nella descrizione dei modi del processore, emergeranno numerose situazioni di errore possibili).

La maggior parte dei processori prevede di trattare l'errore come se fosse un particolare tipo di interrupt. In questo modo, quando si verifica un errore che impedisce al processore di procedere normalmente con l'esecuzione delle istruzioni, viene attivata, attraverso un opportuno vettore di interrupt, una routine del SO che decide come gestire l'errore stesso. Spesso la gestione consiste nella terminazione forzata (abort) del programma che ha causato l'errore, eliminando il processo.

Priorità e abilitazione degli interrupt

Abbiamo visto che gli interrupt possono essere nidificati, cioè che un interrupt può interrompere una routine di interrupt relativa ad un evento precedente. Non sempre è opportuno concedere questa possibilità – ovviamente è sensato che un evento molto importante e che richiede una risposta urgente possa interrompere la routine di interrupt che serve un evento meno importante, ma il contrario invece deve essere evitato. Un meccanismo molto diffuso per gestire questo aspetto è il seguente:

1. il processore possiede un livello di priorità che è scritto in un opportuno registro all'interno del processore stesso (**registro di stato** – **PSR**)
2. il livello di priorità del processore può essere modificato dal software tramite opportune istruzioni macchina che scrivono nel registro di stato
3. anche agli interrupt viene associato un livello di priorità, che è fissato nella configurazione fisica del calcolatore
4. un interrupt viene accettato, cioè si passa ad eseguire la sua routine di interrupt, solo se il suo livello di priorità è superiore al livello di priorità del processore in quel momento, altrimenti l'interrupt viene tenuto in sospeso fino al momento in cui il livello di priorità del processore non sarà stato abbassato sufficientemente

Utilizzando questo meccanismo Hardware il sistema operativo può alzare e abbassare la priorità del processore in modo che durante l'esecuzione delle routine di interrupt più importanti non vengano accettati interrupt meno importanti.

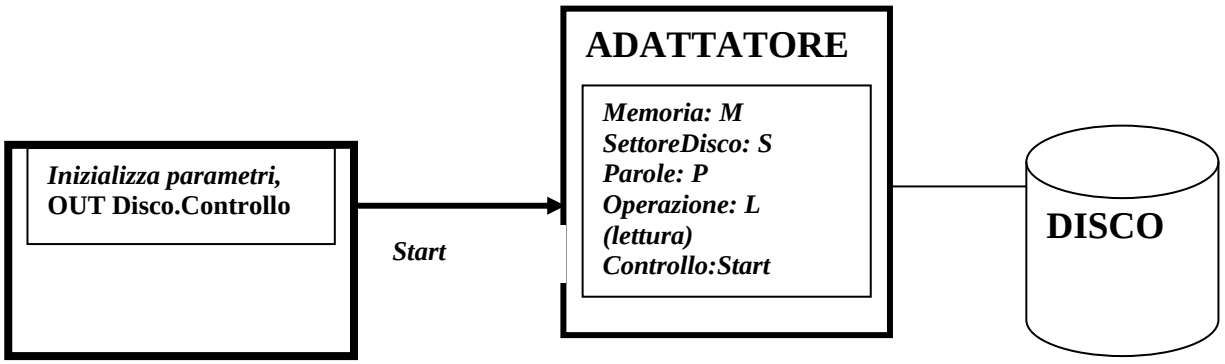
**5. Accesso diretto alla memoria da parte delle periferiche (DMA)**

Alcune periferiche veloci (ad esempio i dischi magnetici) non richiedono al processore di intervenire per la lettura o scrittura di ogni singolo dato ma possono traferire il contenuto di molte parole da (o alla) memoria autonomamente. Tipicamente la periferica possiede dei registri nei quali il processore scrive le seguenti informazioni:

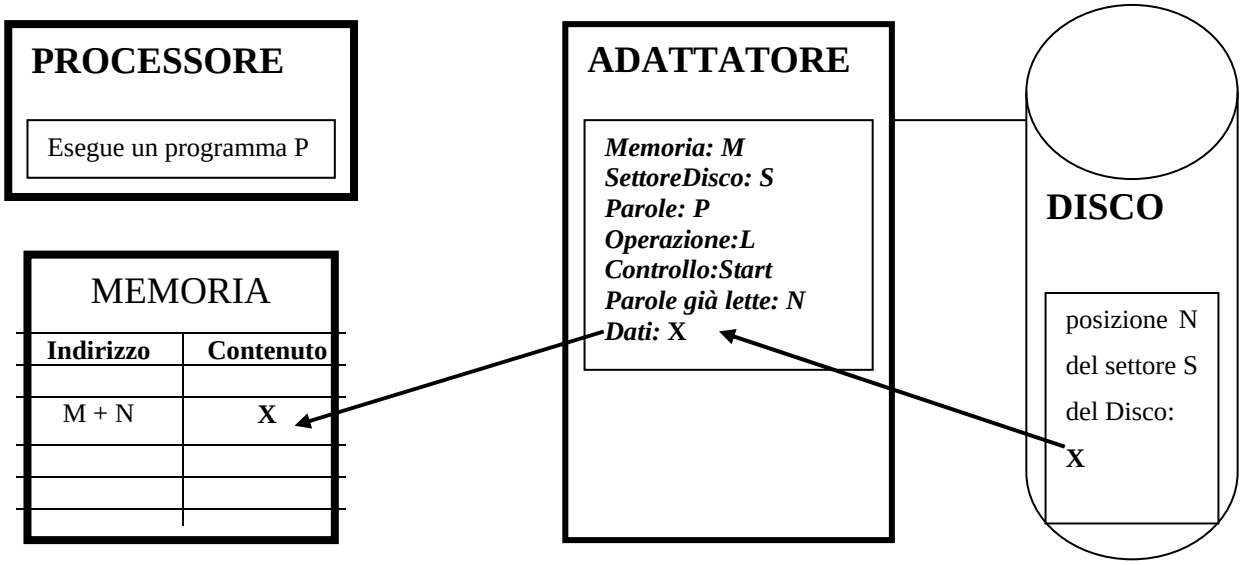
1. l'indirizzo della memoria dal quale iniziare il trasferimento
2. l'indirizzo sulla periferica dal quale iniziare il trasferimento
3. il numero di parole da trasferire
4. la direzione del trasferimento (lettura o scrittura in memoria)

Il processore, dopo aver inizializzato tali registri, ordina alla periferica di iniziare l'operazione scrivendo il comando "start" nel registro di comando e poi passa ad eseguire altri programmi; dopo aver completato tutta l'operazione la periferica genera un interrupt per avvisare il processore. In figura 8 è mostrato questo meccanismo con riferimento a un disco al quale viene richiesta un'operazione di lettura. Si tenga presente che i dati sul disco sono organizzati in settori; ogni settore è identificato dal proprio numero.

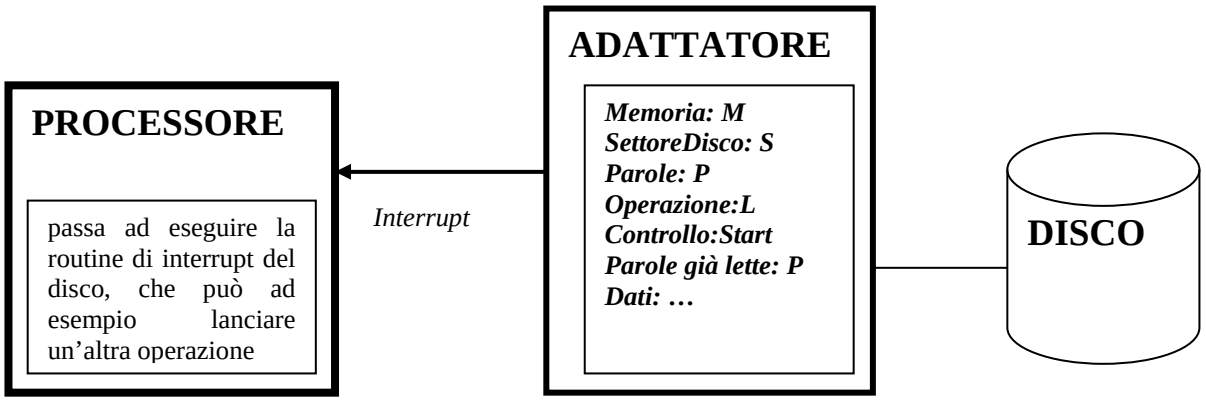
Ai fini della realizzazione del SO l'esistenza delle periferiche in DMA è importante per due motivi: primo, perchè richiede che il SO gestisca delle aree di memoria adatte al trasferimento in DMA dalle periferiche (in particolare i dischi), dette **buffer**, e secondo, perchè il SO distingue tra i gestori di periferiche in DMA (block devices) e quelli di periferiche normali (**character devices**). Quest'ultimo aspetto verrà ripreso nel capitolo sui gestori di periferiche.




*a) il processore inizializza l'adattatore del disco e avvia l'operazione*




***b)l'adattatore DMA sta trasferendo dal disco alla memoria;N parole sono già state trasferite, l'indirizzo corrente in memoria è M+N, l'indirizzo corrente sul disco è l'N-esimo dall'inizio del settore***




***c)l'operazione è terminata (parole lette=P) e l'adattatore genera un interrupt***

***Figura 8 – Lettura di un disco tramite DMA***

## 6. Meccanismi specifici per i Sistemi operativi multiprogrammati

I meccanismi illustrati fino a questo punto sono ovviamente indispensabili per permettere a un qualsiasi sistema di funzionare con un minimo di efficienza, ed erano infatti posseduti anche dai piccoli calcolatori gestiti da sistemi operativi non multiprogrammati; i meccanismi descritti nel seguito sono invece posseduti da tutti i processori moderni che devono supportare un SO multiprogrammato come LINUX. e sono quindi orientati più specificamente al supporto del SO.

Modi di funzionamento – istruzioni privilegiate

Il processore ha la possibilità di funzionare in due stati o **modi** diversi: modo **Utente** (detto anche **non privilegiato**) e modo **Supervisore** (detto anche **privilegiato**). Quando il processore è in modo S può eseguire tutte le proprie istruzioni e può accedere a tutta la propria memoria; quando invece è in modo U può eseguire solo una parte delle proprie istruzioni e può accedere solo a una parte della propria memoria. Le istruzioni eseguibili solo quando il processore è in modo S sono dette istruzioni privilegiate, le altre sono dette non privilegiate.

E' intuibile che, come vedremo più avanti, quando viene eseguito il SO il processore sia in modo S, mentre quando vengono eseguiti i normali programmi esso sia in modo U.

Alle istruzioni privilegiate appartengono le istruzioni di ingresso e uscita IN e OUT; un normale processo non può quindi accedere direttamente a una periferica, ma deve richiedere tale funzione al SO.

Anche istruzioni che hanno un effetto globale sono privilegiate: ad esempio istruzioni che arrestano il processore (Halt); in questo modo si evita che un processo possa svolgere funzioni che danneggerebbero gli altri processi.

Chiamata al supervisore, interrupt e modi di funzionamento

Quando un processo ha bisogno di un servizio del SO esso esegue una particolare istruzione che chiameremo **SVC**[1] (**Supervisor Call**); tale istruzione realizza

[1] nei diversi processori esistono moltissime varianti di questo meccanismo, ma sono in ultima analisi tutte riconducibili alla SVC descritta qui

una salto ad una particolare funzione del SO, ed equivale quindi ad una particolare chiamata di funzione. Le differenze tra una SVC ed una normale invocazione di funzione sono poche, ma fondamentali:

1. la SVC, a differenza di un normale salto a funzione, non indica l'indirizzo della funzione che viene attivata, ma è il processore che lo determina leggendolo in uno speciale vettore di interrupt associato all'evento "chiamata al supervisore"; il SO inizializza tale vettore di interrupt con l'indirizzo della funzione di gestione dei servizi di sistema.
2. la SVC ovviamente è un'istruzione non privilegiata, perchè altrimenti non sarebbe utilizzabile dai processi, ma dopo la sua esecuzione il processore passa automaticamente in modo privilegiato, perchè l'esecuzione del SO deve avvenire in modo privilegiato.

Possiamo a questo punto completare il quadro dei meccanismi che comportano un cambiamento del modo di funzionamento del processore:

- quando si verifica un interrupt e il processore passa all'esecuzione di una routine di interrupt il modo del processore viene posto a S indipendentemente da come era prima (poteva essere U se l'interrupt aveva interrotto un normale processo oppure essere S se aveva interrotto il SO)
- l'istruzione **IRET**, che viene utilizzata alla fine di una routine di interrupt o di un servizio di sistema, ed esegue il ritorno al programma che era stato interrotto o che aveva eseguito una SVC prelevando dalla pila l'indirizzo di ritorno è leggermente diversa dalla RFS per i seguenti aspetti: essa deve ristabilire, oltre al registro PC, anche il modo di funzionamento del processore che era in vigore al momento dell'interrupt o della SVC prelevandolo dalla stessa pila; è necessario quindi che al momento di un interrupt o di una SVC il processore salvi sulla pila, oltre all'indirizzo di ritorno, anche il modo di funzionamento da ristabilire al ritorno, che chiameremo sinteticamente **modo di ritorno**. Ovviamente, l'istruzione IRET è privilegiata, dato che è usata solamente da funzioni del SO.

In tabella 1 sono riassunte le caratteristiche dei meccanismi analizzati. Da tale tabella si possono ricavare le seguenti considerazioni relative al passaggio tra l'esecuzione di un generico processo e il SO:

- Il passaggio dall'esecuzione di un processo al SO avviene solamente quando il processo lo richiede tramite SVC oppure quando si verfica un interrupt;
- Il passaggio dall'esecuzione del SO a un generico processo avviene tramite una particolare istruzione (IRET) che ritorna al processo interrotto

Ovviamente, questo è il funzionamento di base dell'Hardware; diversi aspetti devono ancora essere chiariti trattando il SO.

| **Meccanismo di salto** | **Modo di partenza** | **Modo di arrivo** | **Indirizzo di salto** | **Meccanismo di ritorno** | **Modo dopo il ritorno** |
|---|---|---|---|---|---|
| salto a funzione normale | U<br>S | U<br>S | nell'istruzione | istruzione di ritorno | U<br>S |
| SVC | U | S | vettore int. | IRET | U |
| interrupt | U<br>S | S<br>S | vettore int. | IRET | U<br>S |

***Tabella 1***

## 7. Rilocazione degli indirizzi

La gestione della memoria, che deve essere suddivisa tra il SO e molti processi, è un argomento talmente complesso che verrà trattato estesamente in un capitolo apposito. Per poter discutere il funzionamento del nucleo del SO ci basta qui definire un meccanismo minimale, che chiamiamo **rilocazione dinamica**, e si basa sul seguente principio:

> quando il processore è in modo U, gli indirizzi che esso genera vengono sommati, prima di accedere alla memoria, al valore di un particolare registro, detto **registro base**;

Si osservi che la rilocazione dinamica è applicata solamente al modo U; quando il processore è in modo S, la rilocazione non viene applicata e gli indirizzi che esso genera vengono utilizzati direttamente per accedere alla memoria.

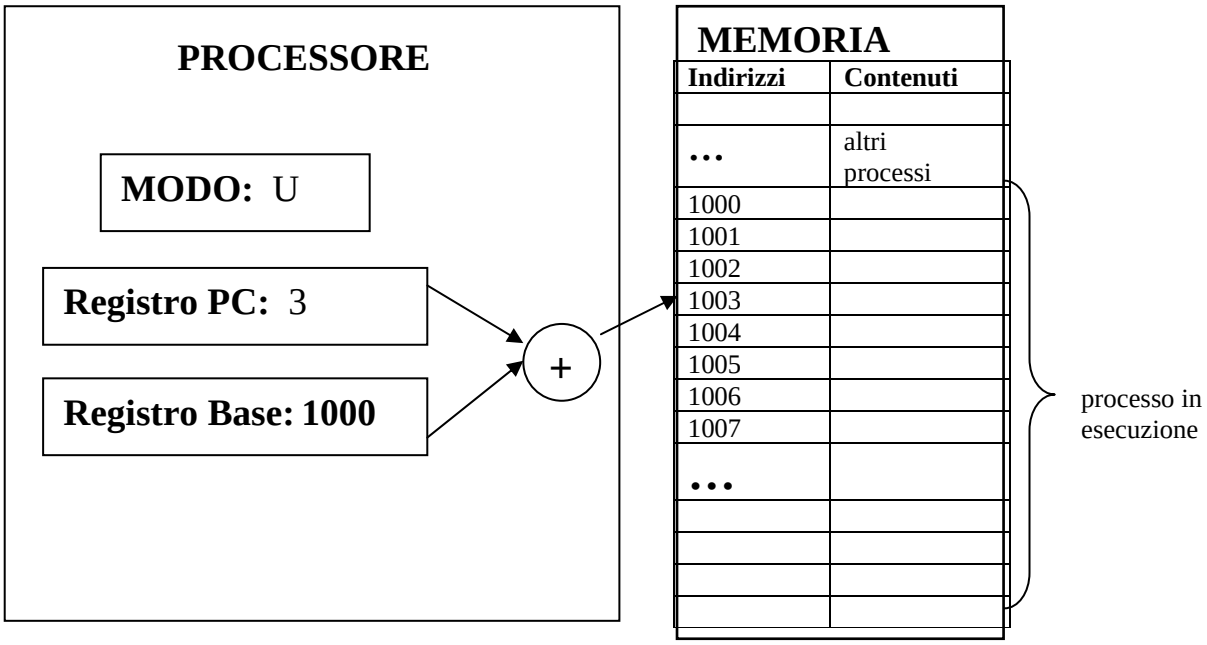



***Figura 9 – Rilocazione dinamica***

In figura 9 è mostrata la giustificazione del meccanismo appena descritto: ogni processo viene posto in una zona della memoria ad esso riservata; dato che tale zona inizia ad un indirizzo diverso da 0, gli indirizzi che vengono generati durante l'esecuzione del processo sono modificati (**rilocati**) aggiungendovi il valore della **base del processo**; tale valore è stato posto nel registro base dal SO prima di lanciare in esecuzione tale processo.

Dato che la discussione della gestione della memoria è rinviata ad un apposito capitolo, quanto descritto sopra si limita a fornirci le seguenti indicazioni, che resteranno comunque valide anche nei meccanismi più complessi di gestione della memoria:

- è possibile fare in modo che ogni processo lavori su una zona diversa della memoria;
- il SO ha il compito di adattare i meccanismi di rilocazione nei momenti in si passa dall'esecuzione di un processo a quella di un altro processo;

- infine, supporremo che il SO abbia accesso a tutta la memoria, cioè possa accedere anche alle zone dei processi; in questo modo il SO può, durante l'esecuzione di un servizio di sistema, prelevare dei dati dal processo o scrivere dei dati nel processo

**8. Pila di modo U e di modo S**

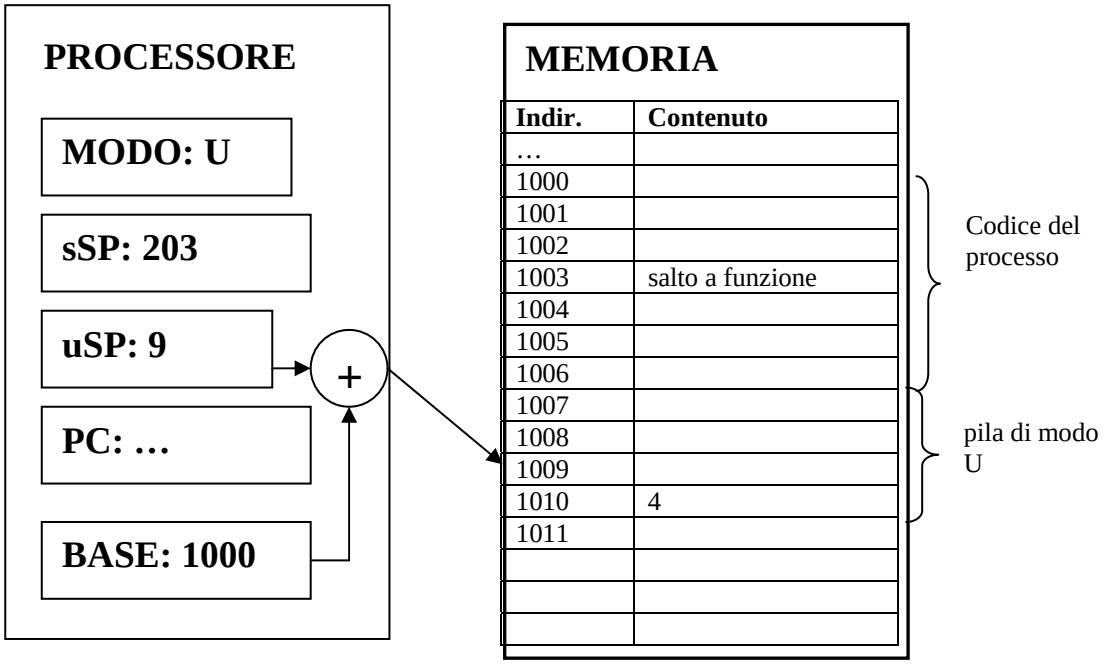
L'ultimo meccanismo che vogliamo considerare non è necessariamente presente in tutti i processori moderni, ma è molto comodo per capire come funziona il SO. Si tratta dell'esistenza di 2 pile, una utilizzata quando il modo del processore è S e l'altra utilizzata quando il processore è in modo U. A livello hardware, l'esistenza di due pile si materializza nell'esistenza di due puntatori a pila, che chiameremo **sSP** e **uSP**, e nel fatto che il processore, quando deve eseguire un'operazione relativa alla pila, utilizza sSP oppure uSP in base al modo posseduto in quel momento.

L'esistenza di due puntatori alla pila, e quindi di due pile allocate in diverse zone di memoria, significa che ogni volta che avviene un passaggio di modo da S ad U e viceversa cambia la pila utilizzata dal sistema.

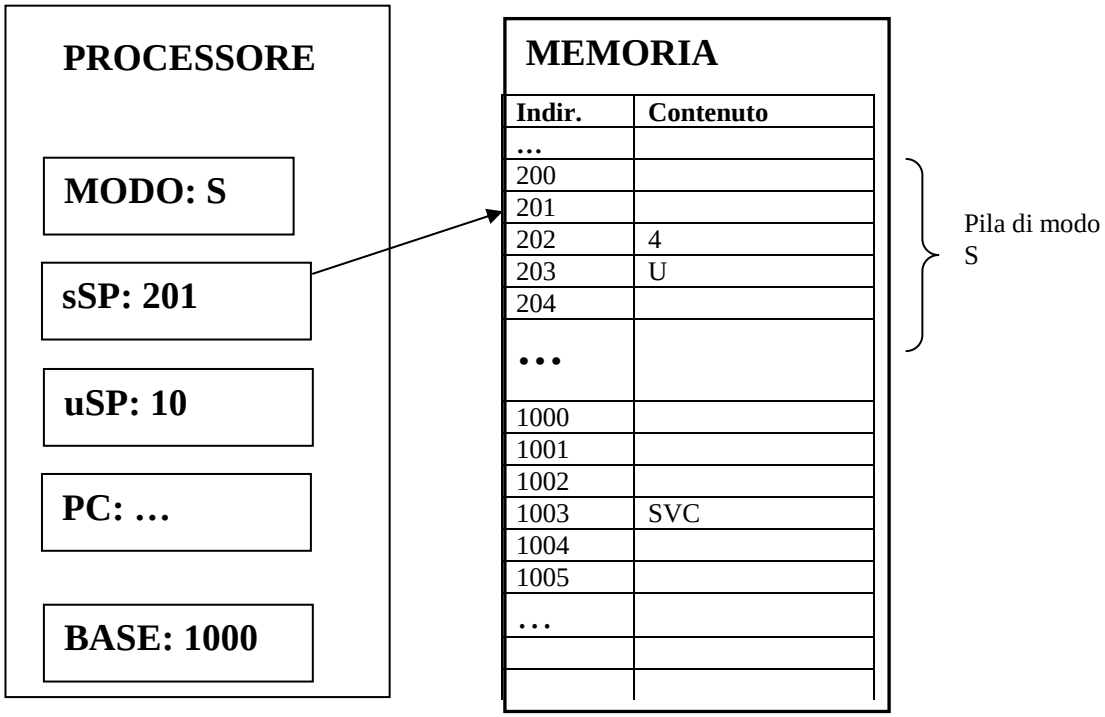
Per quanto riguarda le normali funzioni, se una funzione è eseguita nell'ambito di un processo la pila utilizzata è quella di modo U; se invece la funzione è invocata nell'ambito del SO, la pila utilizzata è quella di modo S.

Per quanto riguarda invece le situazioni di passaggio tra modi di funzionamento diversi elencate in tabella 1, è opportuno precisare che la pila utilizzata nelle operazioni di passaggio stesse è sempre quella di modo S. Ciò significa che al momento di un interrupt o di una SVC la pila sulla quale sono salvati l'indirizzo ed il modo di ritorno è quella di modo S, mentre al momento di una IRET la pila dalla quale sono prelevati il modo e l'indirizzo di ritorno è anch'essa quella di modo S.

In figura 10 è mostrato un esempio di modificazione delle diverse pile a fronte di un normale salto a funzione (figura 10a) e di una SVC (figura 10b). Nel primo caso il processore ha scritto l'indirizzo di ritorno sulla pila di modo U, nel secondo caso ha scritto il modo del processore e l'indirizzo di ritorno sulla pila di modo S.




***a) Il processore ha appena eseguito il salto a funzione contenuto all'indirizzo 1003***




***b) Il processore ha appena eseguito la SVC contenuta all'indirizzo 1003***

***Figura 10 – Uso delle pile di modo U e di modo S da parte di un salto a funzione e di una istruzione SVC; inizialmente uSP contiene 10 e sSP contiene 203***