Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte N: Il Nucleo del Sistema Operativo

cap. N2 – Meccanismi Hardware di supporto

N.2 Funzionalità Hardware e compilazione

1. Considerazioni generali

La realizzazione di un SO multiprogrammato come Linux o Windows richiede da parte dell'Hardware la disponibilità di alcuni meccanismi fondamentali, in assenza dei quali è impossibile realizzare pienamente le funzionalità richieste.
Nel seguito vengono descritte tali funzionalità generali, con riferimento in particolare alle funzionalità richieste da Linux e fornite dal x64.
In alcuni casi le funzionalità del x64 sono inutilmente complesse rispetto ai fini di questo testo e quindi ne verrà fornita una versione semplificata.
Registri
Nel x64 esistono numerosi registri a 64 bit usabili dal programmatore, che citeremo solo quando serviranno; due registri di uso particolare sono il PC (indicato come rip nell'assembler) e lo SP (indicato come rsp nell'assembler).
Pila
Come nel MIPS, anche nel x64 la pila cresce da indirizzi alti verso indirizzi bassi. A differenza del MIPS, il decremento e l'incremento del SP sono svolti nella stessa istruzione di scrittura in memoria, quindi push e pop della pila richiedono una sola istruzione ciascuna.
Salto a funzione
Si tenga presente che il x64, a differenza del MIPS, salva il valore dell'indirizzo di ritorno sulla pila, non in un registro. Pertanto l'istruzione di salto a funzione esegue automaticamente le seguenti operazioni:

- il registro SP viene decrementato
- il valore del PC incrementato viene salvato sulla pila

e il ritorno da funzione preleva il valore di PC dalla pila incrementando poi lo SP.

Registri e strutture dati per la rappresentazione del contesto
I meccanismi utilizzati dal sistema operativo richiedono che l'Hardware utilizzi numerose informazioni automaticamente. A questo scopo esistono opportuni registri e strutture dati; in parte tali registri e strutture dati sono leggibili o scrivibili dal Software, permettendo in tal modo il controllo da parte del sistema operativo della operazioni svolte automaticamente dall'Hardware.
Le Strutture Dati utilizzate automaticamente dall'Hardware saranno chiamate nel seguito per brevità ***Strutture Dati ad accesso HW***.
Nel x64 alcuni di questi registri e strutture dati ad accesso HW sono eccessivamente complesse, a causa delle esigenze di compatibilità con altri modi di funzionamento. Dato che questi aspetti non ci interessano, noi ***semplificheremo drasticamente*** il modello definendo delle strutture più semplici.

In particolare, supporremo che esista un registro di stato, **PSR**, che contiene tutta l'informazione di stato che caratterizza la situazione del processore, escluse alcune informazioni per le quali indicheremo esplicitamente dei registri dedicati a contenerle.
Nel seguito, le funzionalità descritte nei riquadri con il titolo x64 corrispondono esattamente a quelle del x64, quelle prive di tale indicazione o con l'indicazione esplicita x64_semplificato costituiscono appunto semplificazioni.

Approfondimento
La maggior parte delle informazioni per le quali noi definiremo dei registri appositi (PSR, SSP, USP, ecc) sono contenute nel x64 reale in una struttura dati detta Task State Segment (TSS), accessibile tramite un descrittore (TSSD) contenuto nella Global Descriptor Table (GDT).

2. 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 **kernel** o **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; 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; in questo modo si evita che un processo possa svolgere funzioni che danneggerebbero gli altri processi.

x64
Nel x64 esistono 4 modi di funzionamento del processore, da 3 (non privilegiato) a 0 (massimo privilegio); il livello al quale il processore sta funzionando (Current Privilege Level – CPL) è memorizzato in un registro interno del processore inaccessibile dal Software.
La modifica del valore di CPL avviene indirettamente in momenti particolari, che vedremo nel seguito. Dato che il modello generale di architettura gestito da Linux prevede solamente 2 livelli, sotto Linux x86 può funzionare solo con **CPL3** (o **modo U**) oppure **CPL0** (o **modo S**).

Noi supporremo che il CPL sia memorizzato nel registro PSR.

3. Chiamata al supervisore e modi di funzionamento

Quando un processo ha bisogno di un servizio del SO esso esegue una particolare istruzione chiamata **SYSCALL** in molti calcolatori, inclusi x64 e MIPS; tale istruzione realizza una salto ad una particolare funzione del SO, ed equivale quindi ad una particolare chiamata di funzione.

A differenza di un normale salto a funzione, la SYSCALL sostituisce oltre al PC anche il PSR della CPU con valori nuovi, salvando i valori precedenti sulla pila. I valori nuovi vengono prelevati da un'opportuna struttura dati ad accesso HW, costituita da 2 celle di memoria poste a un indirizzo noto all'Hardware. Chiameremo tale struttura **Vettore di Syscall**.
Quindi nell'esecuzione di una SYSCALL avvengono automaticamente le seguenti operazioni:

- il valore del PC incrementato viene salvato sulla pila (push del PC – include l'opportuno decremento del registro SP)
- il valore del PSR viene salvato sulla pila (idem)
- nel PC e nel PSR vengono caricati i valori presenti nel Vettore di Syscall

Le differenze tra una SYSCALL ed una normale invocazione di funzione sono poche, ma fondamentali:

- la SYSCALL, a differenza di un normale salto a funzione, non indica l'indirizzo d'inizio della funzione che viene attivata, ma è il processore che lo determina leggendolo in una struttura dati; il SO LINUX inizializza tale struttura dati durante la fase di avviamento con l'indirizzo della funzione **system_call( )**, che costituisce il punto di entrata unico per tutti i servizi di sistema di LINUX.
- la SYSCALL (ovviamente) è un'istruzione non privilegiata, perché altrimenti non sarebbe utilizzabile dai processi, ma dopo la sua esecuzione il processore passa automaticamente in modo privilegiato, perché l'esecuzione del SO deve avvenire in modo privilegiato. Questa istruzione costituisce l'unico modo per un programma funzionante in modo non privilegiato di passare al modo privilegiato – cedendo il controllo al SO

Esiste un'istruzione, chiamata **SYSRET** nel x64, che permette di svolgere l'operazione inversa della SYSCALL, in modo da ritornare dal SO al processo che lo ha invocato.

- nel PSR viene caricato il valore presente sulla pila (pop del PSR – include l'opportuno incremento del registro SP)
- nel PC viene caricato il valore presente sulla pila (idem)

In Linux l'istruzione SYSRET è eseguita alla fine della funzione system_call( ) e costituisce quindi l'unico punto di uscita dal Sistema Operativo e di ritorno al processo che ha invocato un servizio.

4. Protezione della memoria del SO

Quando il processore è in modo U non deve poter accedere alle zone di memoria riservate al SO. Viceversa, quando il processore è in modo S deve poter accedere sia alla memoria del SO, sia alla memoria dei processi. Questo meccanismo è realizzato con modalità diverse in diverse architetture, ma fondamentalmente si basa sulla suddivisione dello spazio di indirizzamento virtuale in due sottoinsiemi distinti, quello di modo U e quello di modo S:

x64

Lo spazio di indirizzamento potenziale del x64 è di 2^{64} byte, perché i registri sono in grado di gestire tale dimensione di indirizzamento. Tuttavia, al momento l'architettura limita lo spazio virtuale utilizzabile a 2^{48}, cioè 256 Tb. Tale spazio è suddiviso nei 2 sottospazi di modo U ed S, ambedue da 2^{47} byte (128 Tb). Il meccanismo utilizzato per distinguere gli indirizzi di modo U ed S si basa sul seguente accorgimento (vedi Figura 1): mentre lo spazio di modo U occupa i primi 2^{47} byte (da 0 a 0000 7FFF FFFF FFFF), lo spazio di modo S i 2^{47} byte di indirizzo più alto (da FFFF 8000 0000 0000). Gli indirizzi intermedi sono detti non-canonici e se utilizzati generano un errore. Quando la CPU è in modo S può utilizzare tutti gli indirizzi canonici, mentre quando è in modo U la generazione di un indirizzo superiore a 0000 7FFF FFFF FFFF genera un errore.

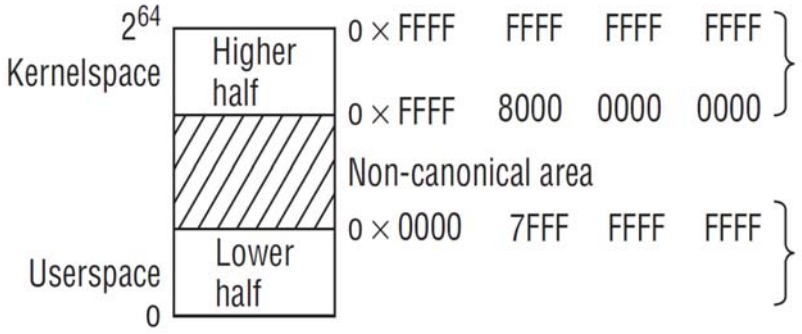


Figura 1 Spazio di indirizzamento del x64

La memoria del x64 è gestita tramite paginazione. Dato che questo argomento verrà trattato nei capitoli relativi alla gestione della memoria, ci limitiamo ad anticipare alcune proprietà della paginazione necessarie alla comprensione del nucleo:

- la memoria è suddivisa in unità dette pagine, di dimensione 4Kb; le pagine costituiscono unità di allocazione della memoria (ad esempio, della pila o dello heap)
- ogni indirizzo prodotto dalla CPU (indirizzo virtuale) viene trasformato in un indirizzo fisico prima di accedere alla memoria fisica – chiameremo questa trasformazione mappatura virtuale/fisica
- La mappatura è descritta da una struttura dati detta Tabella delle Pagine

5. Commutazione della pila nel cambio di modo

La pila utilizzata implicitamente dalla CPU nello svolgimento delle istruzioni (ad esempio nel salto a funzione) è puntata da un registro puntatore alla pila (SP). Per realizzare il SO è necessario fare in modo che la pila utilizzata durante il funzionamento in modo S sia diversa da quella utilizzata durante il funzionamento in modo U. Per questo motivo, ***quando la CPU cambia modo di funzionamento deve anche poter sostituire il valore di SP***.

In questo modo la CPU utilizza una pila diversa quando opera in modi diversi. Indicheremo con ***sPila*** e ***uPila*** le 2 pile quando è necessario. Ovviamente le 2 pile sono allocate nei corrispondenti spazi virtuali di modo U e di modo S.

Possiamo ora interpretare i valori prodotti dall'ultimo esempio del capitolo precedente, riportati in tabella 1. Il valore di usersp è un indirizzo di modo U perchè si riferisce al valore dello SP quando era attiva uPila, mentre gli altri valori sono indirizzi di modo S e si riferiscono a sPila. Linux alloca ad ogni processo una sPila costituita da 2 pagine (8K). Dato che le pagine da 4K usano 12 bit di offset, le ultime 3 cifre esadecimali di inizio sPila e fine sPila sono nulle, e le cifre precedenti rappresentano il numero di pagina virtuale (NPV). Tra inizio e fine di sPila devono esserci esattamente 2 pagine, quindi NPV deve variare di 2, e in effetti si passa da ...6 a ...4.

variabile	indirizzo	significato
thread.sp0	**0xFFFF 8**800 5C64 **6**000	inizio sPila
ts-> stack	**0xFFFF 8**800 5C64 **4**000	fine sPila
thread.sp	**0xFFFF 8**800 5C64 **5**D68	SP di sPila
usersp	**0x 0000 7**FFF 6DA9 8C78	SP di uPila
Tabella 1		

Quando il processo ha iniziato a usare sPila, in SP è stato caricato il valore 0xFFFF88005C64 **6000**, e tale valore è stato decrementato dalle operazioni di push fino al valore attuale, 0xFFFF88005C64 **5D68**.

Nella commutazione da modo U a modo S la commutazione di pila deve avvenire *prima* del salvataggio di informazioni sulla stessa; pertanto l'indirizzo di ritorno a modo U deve essere salvato su sPila e non su uPila; viceversa, quando si verifica un ritorno da modo S a modo U l'informazione per il ritorno verrà prelevata da sPila, cioè prima di commutare a uPila.

Per realizzare il meccanismo descritto è necessario che l'Hardware possa determinare automaticamente il valore del SP di modo S da sostituire a quello di modo U.
Il meccanismo utilizzato dal x64 è piuttosto complesso, quindi noi seguiremo il seguente **modello semplificato**, basato sull'esistenza di una struttura dati ad accesso HW contenente 2 celle di memoria, che chiameremo SSP e USP, con il seguente significato:

- **SSP** contiene il valore da caricare in SP al momento del passaggio al modo S; è compito del sistema operativo garantire che tale registro contenga il valore corretto, cioè quello relativo alla sPila del processo in esecuzione.
- in **USP** viene salvato il valore del registro SP al momento del passaggio a modo S

Complessivamente le operazioni svolte da SYSCALL sono quindi:

- salva il valore corrente di SP in USP
- carica in SP il valore presente in SSP (adesso SP punta in sPila)
- salva su sPila il PC di ritorno al programma chiamante
- salva su sPila il valore del PSR del programma chiamante
- carica in PC e PSR i valori presenti nel Vettore di Syscall (il modo di funzionamento passa quindi a S)

Simmetricamente, le operazioni svolte da SYSRET sono:

- carica in PSR il valore presente in sPila
- carica in PC il valore presente in sPila
- carica in SP il valore presente in USP (adesso SP punta nuovamente a uPila)

Esempio 1
Nelle seguenti figure vengono mostrate gli effetti dell'inizializzazione del Vettore di SYSCAL, dei Vettori di Interrupt (trattati più avanti) e di SSP da parte del SO e gli effetti dell'esecuzione di una SYSCALL e di una successive SYSRET. I puntatori modificati da un'operazione sono evidenziati in rosso.

- Figura 2: mostra i componenti HW coinvolti nelle operazioni; si noti che le Strutture dati ad accesso HW sono nella memoria S, quindi accessibili solo dal SO, non dai programmi in modo utente – Alcuni di questi componenti (Tabella degli Interrupt e Routine di Interrupt verranno spiegati più avanti)
- Figura 3: mostra l'effetto della inizializzazione svolta dal SO
- Figura 4: mostra una situazione di normale esecuzione di un programma: il modo è U, il PC punta a istruzioni nel segmento codice, SP punta alla pila di modo U – si ipotizza che la prossima istruzione da eseguire sia SYSCALL

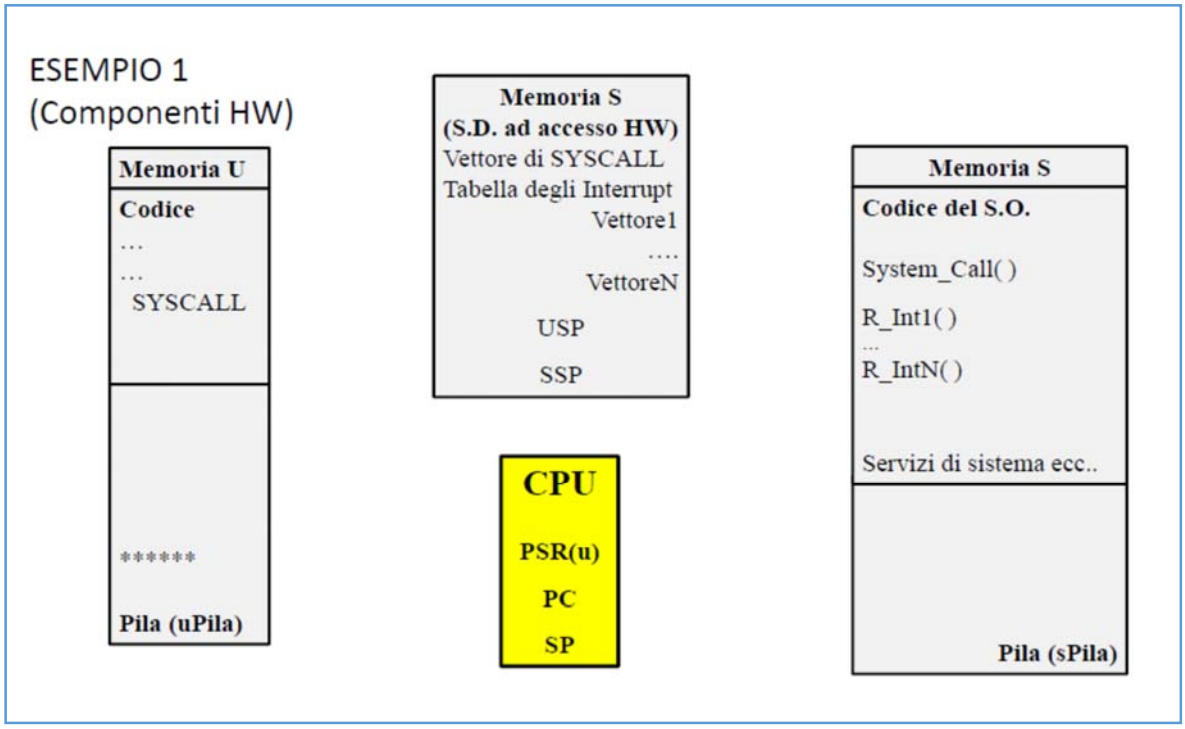


Figura 2

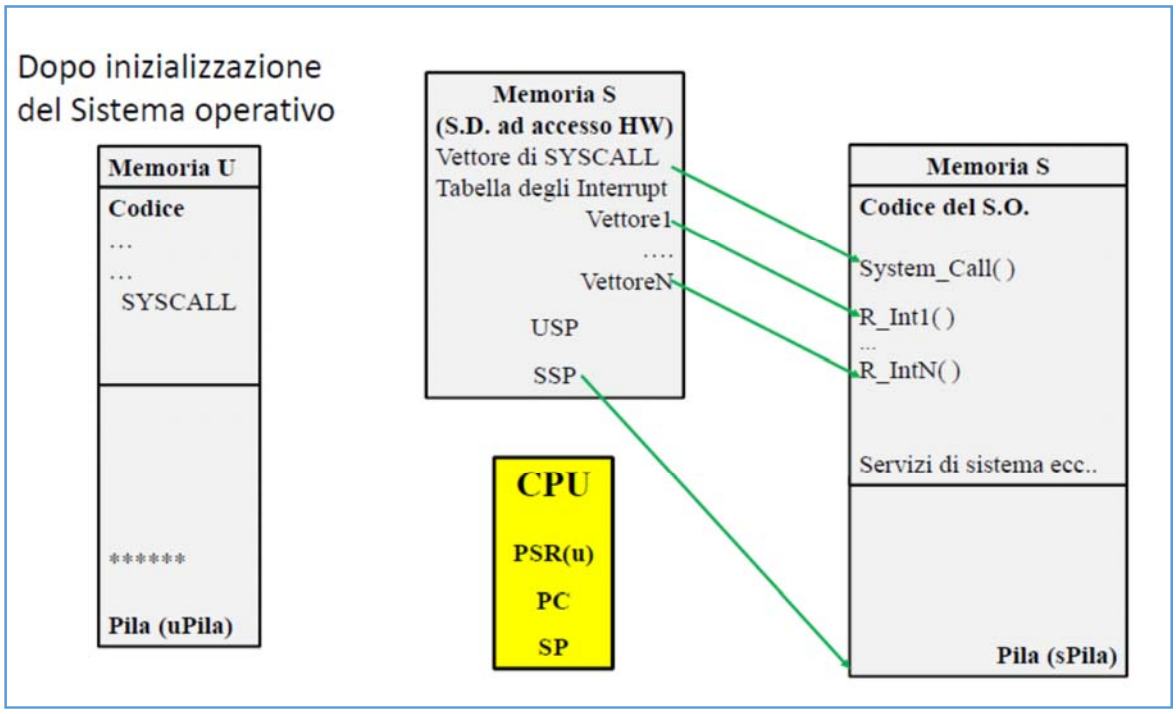


Figura 3

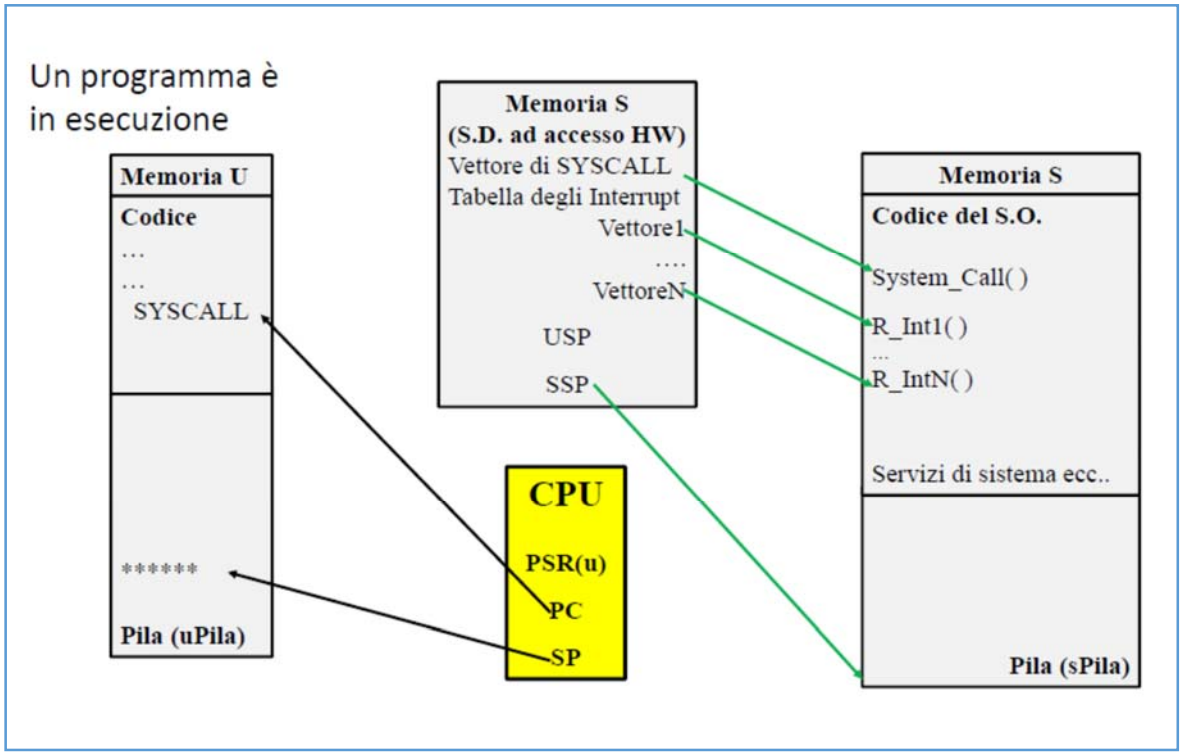


Figura 4

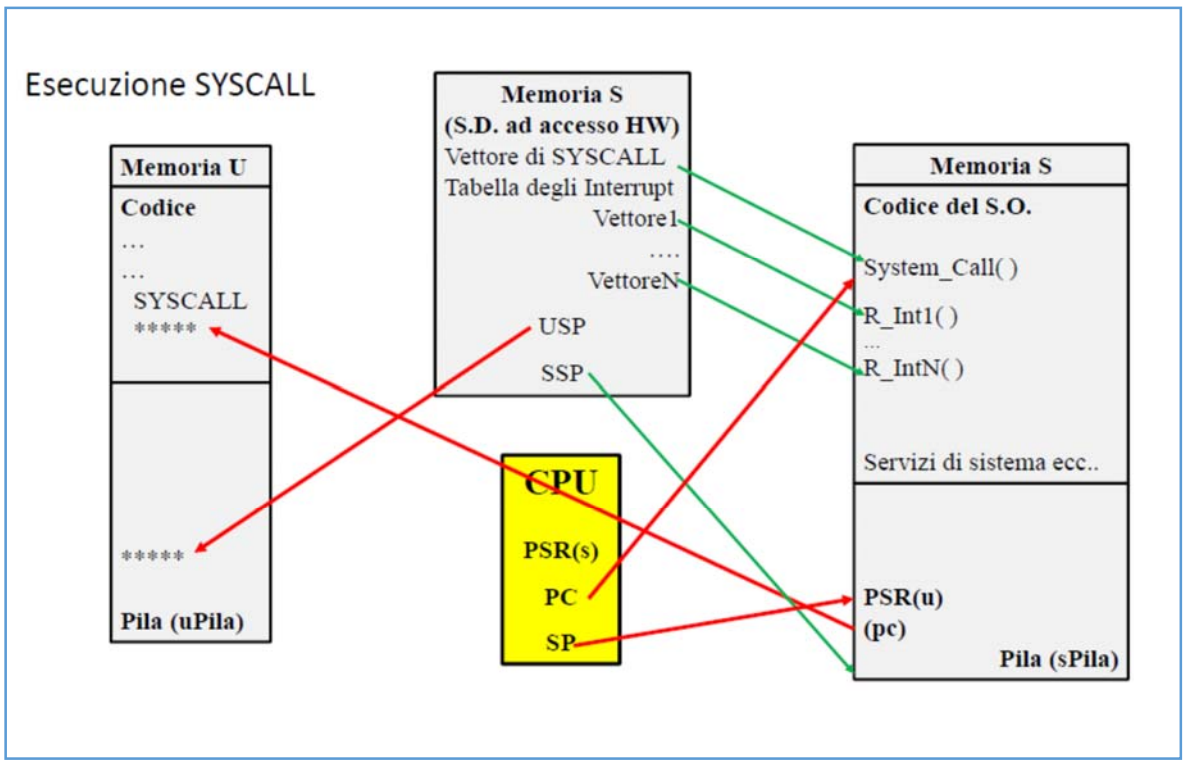


Figura 5

- Figura 5: mostra l'effetto dell'esecuzione di SYSCALL
- Figura 6: mostra l'effetto di una successiva SYSRET

Si noti che la situazione di Figura 6 è identica a quella di Figura 4 – il programma di modo U riprende l'esecuzione dopo che il SO ha terminato il servizio richiesto come se fossa stata invocata una normale funzione.

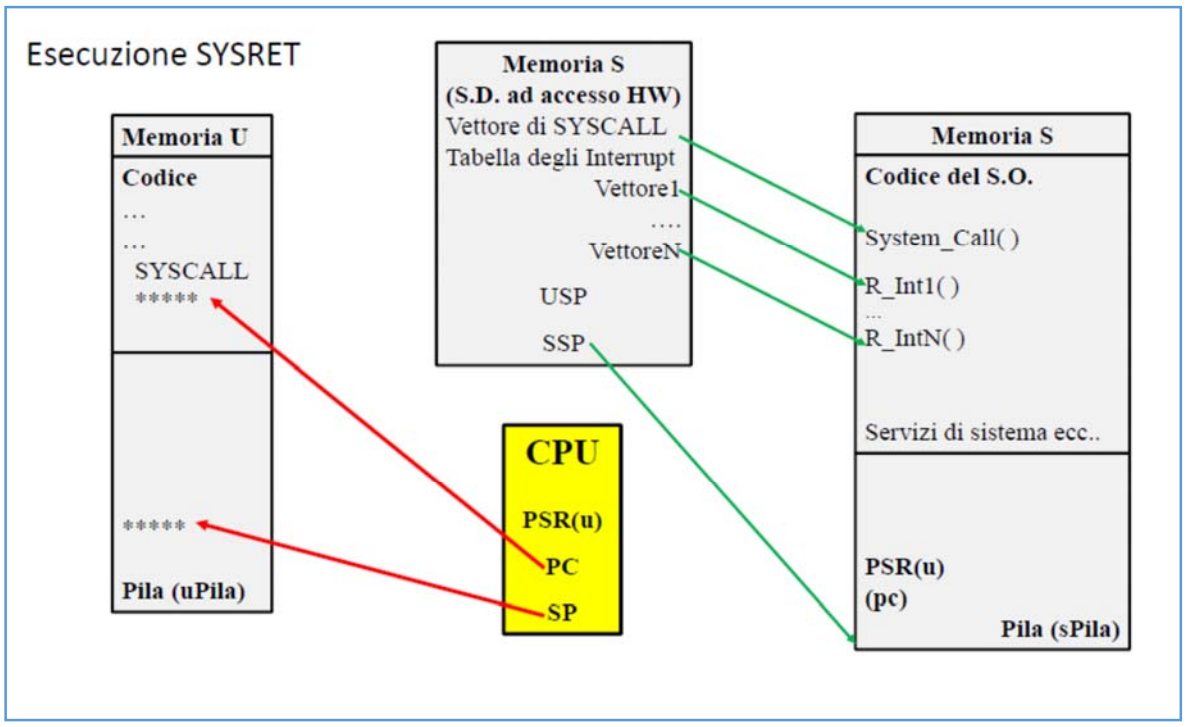


Figura 6

6. Commutazione della mappatura virtuale/fisica della memoria

Linux associa ad ogni processo una diversa Tabella delle Pagine; in questo modo gli indirizzi virtuali di ogni processo sono mappati su aree indipendenti della memoria fisica. Deve esistere un meccanismo efficiente per sostituire la mappatura virtuale/fisica della memoria passando da un processo a un altro. Tale meccanismo può differire notevolmente tra diverse architetture; nel x64 è molto semplice.

x64.
Nel x64 esiste un registro, **CR3** (CR sta per Control Register) che definisce il punto di partenza della tabella delle pagine utilizzata per la mappatura degli indirizzi. Per cambiare la mappatura è quindi sufficiente cambiare il contenuto di CR3. L'organizzazione della Tabella delle Pagine basata su CR3 verrà descritta nel capitolo relativo alla gestione della memoria.

7. Meccanismo di Interruzione (Interrupt)

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** e, come già indicato nelle Figure 2-6, le routine di interrupt fanno parte del SO. Quando il processore si accorge del verificarsi di un particolare evento, esso "interrompe" il programma correntemente in esecuzione ed 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 sulla pila, al momento del salto alla routine di interrupt, 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. L'istruzione che esegue il ritorno da interrupt è detta **IRET.**
Il meccanismo di interrupt è a tutti gli effetti simile ad un'invocazione di funzione o a una SYSCALL, con la differenza che le funzioni normali e le SYSCALL sono attivate esplicitamente dal programma, mentre le routine di interrupt sono attivate da eventi riconosciuti dal processore. *Le routine di interrupt sono quindi completamente asincrone rispetto al programma interrotto, come le funzioni dei thread, e quindi è necessario trattarle con tutti gli accorgimenti della programmazione concorrente.*

Meccanismo di Interrupt e Modi di funzionamento
Il meccanismo di interrupt si combina con il doppio modo di funzionamento S ed U in maniera simile a quello della SYSCALL. In effetti, dal punto di vista Hardware non c'è sostanziale differenza tra un interrupt e una SYSCALL: in ambedue i casi è necessario passare al modo S e salvare l'informazione di ritorno sulla sPila.
Se il modo del processore al momento dell'interrupt era già S alcune operazioni non sono necessarie, ma il registro di stato viene comunque salvato su sPila.
L'istruzione di ritorno da interrupt (IRET) riporta la macchina al modo di funzionamento in cui era prima che l'interrupt si verificasse, prelevando le necessarie informazioni dalla pila sPila.

Il processore deve sapere quale sia l'indirizzo della routine di interrupt che deve essere eseguita quando si verifica un certo evento. Nel caso della SYSCALL tale indirizzo è memorizzato nel Vettore di Syscall; per gli interrupt, che sono numerosi, viene utilizzata una struttura dati ad accesso HW, la **Tabella degli Interrupt**, che contiene un certo numero si **vettori di interrupt** costituiti, come il vettore di Syscall, da una coppia <PC,PSR>. Esiste un meccanismo Hardware che è in grado di convertire l'identificativo dell'interrupt nell'indirizzo del corrispondente vettore di interrupt.
L'inizializzazione della Tabella degli Interrupt con gli indirizzi delle opportune routine di interrupt deve essere svolta dal SO in fase di avviamento.
L'uso per le routine di interrupt 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 annidati**) venga gestito correttamente, esattamente come l'annidamento delle invocazioni di funzioni.

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.
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 nel registro PSR
2. il livello di priorità del processore può essere modificato dal software tramite opportune istruzioni macchina che scrivono nel PSR
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.

Esempio 2

Nella Figure da 7 a 10 sono illustrate le operazioni svolte a causa del verificarsi di un Interrupt. Nell'esempio si ipotizza che l'interrupt si verifichi quando la CPU è già in modo S (interrupt annidato), perché sta eseguendo un servizio di sistema.

Anzitutto si richiama il fatto che, come mostrato già in Figura 3, il SO ha inizializzato i Vettori di Interrupt con i puntatori alle corrispondenti routine di interrupt, R_Int( ).

- Figura 7 rappresenta lo stato al momento del verificarsi dell'interrupt: questa figura è identica a Figura 5, cioè lo stato è quello presente dopo aver iniziato l'esecuzione della funzione system_call
- Figura 8 mostra le operazioni svolte dall'HW al verificarsi di un Interrupt1; si noti che sulla sPila sono a questo punto salvati sia i valori di ritorno da R_Int1( ) a System_call( ), sia quelli per il ritorno al programma di modo U. Si noti anche che il valore di PSR presente nella CPU è quello caricato dal Vettore di Interrupt, e quindi diverso da quello della SYSCALL, anche se ambedue indicano modo U (possono essere infatti diverse altre componenti dello stato, ad esempio la priorità)
- Figura 9 mostra l'effetto dell'istruzione IRET svolta alla fine della routine R_Int1( ); ritorna in esecuzione system_call
- Figura 10 è identica a Figura 6 – il ritorno al programma di modo U non risente dell'esecuzione della routine R_Int1( ).

Se l'interrupt si fosse verificato durante il funzionamento del programma in modo U la sequenza di operazione sarebbe invece stata simile a quella che abbiamo visto nel caso di SYSCALL.

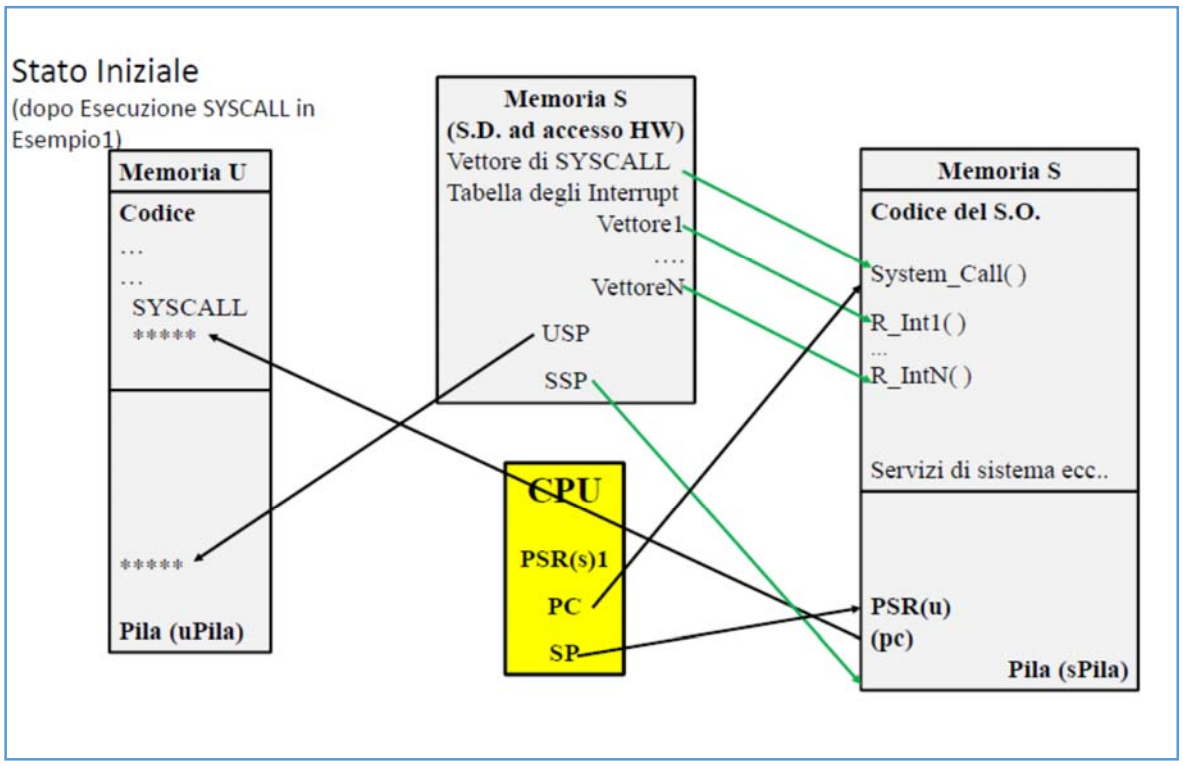


Figura 7

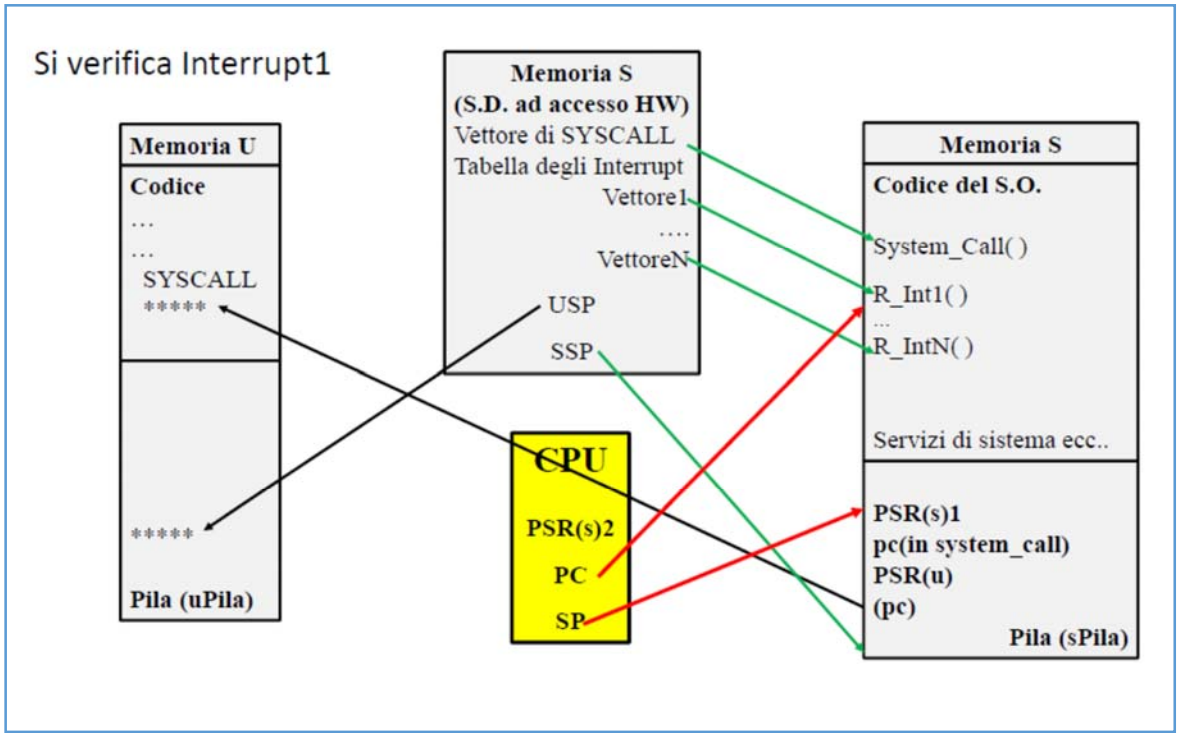


Figura 8

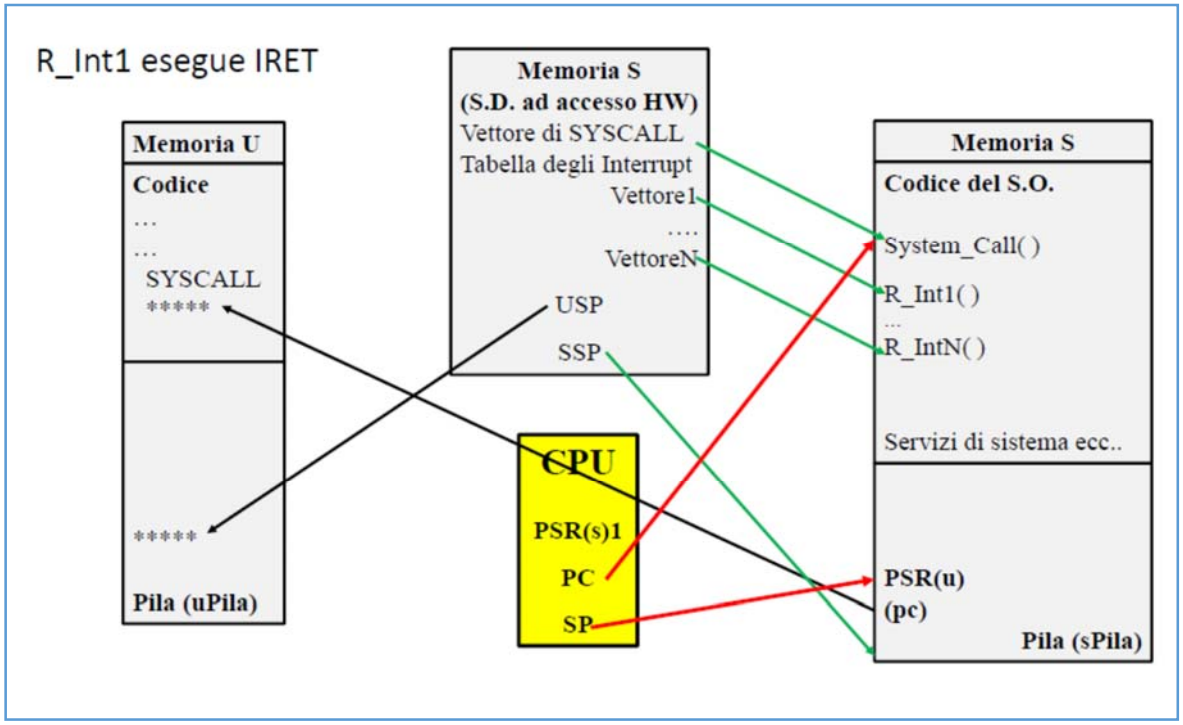


Figura 9

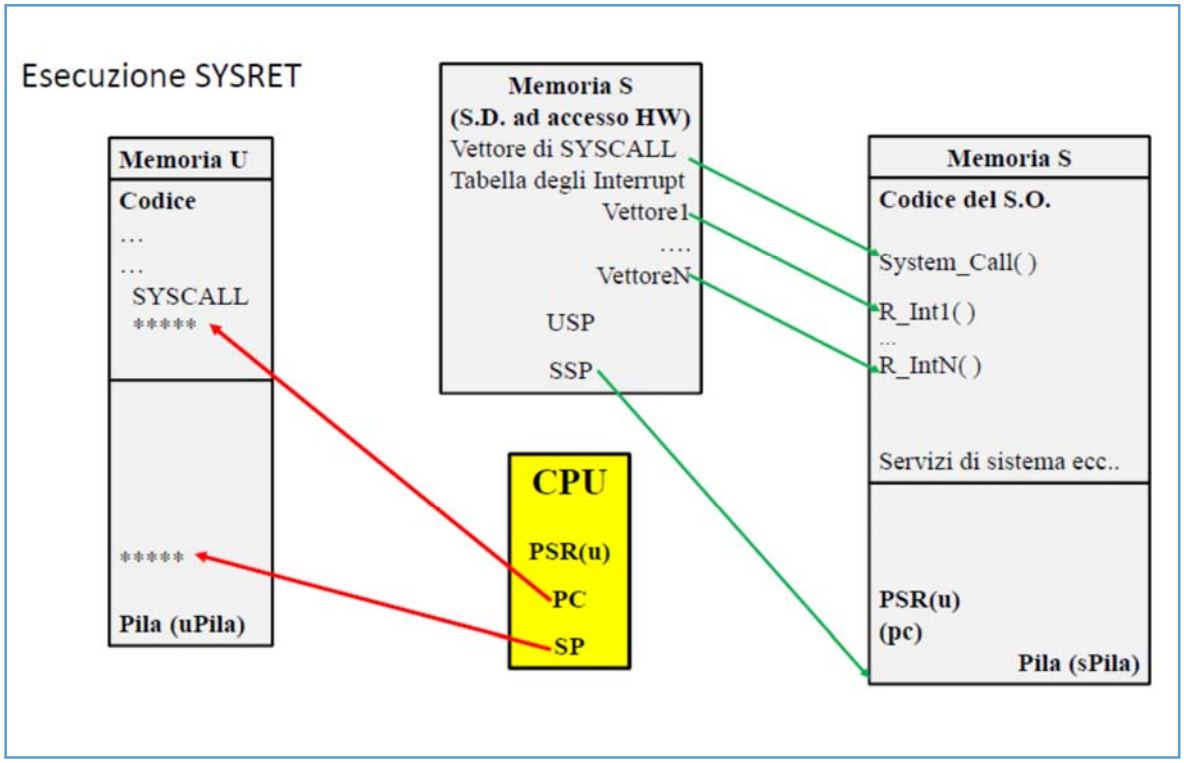


Figura 10

8. Ingresso/Uscita

Il nucleo non è influenzato da come l'architettura gestisce l'Input/Output. E' sufficiente che le istruzioni di I/O siano privilegiate. La trattazione di questo argomento è perciò rinviata ai capitoli sul Input/Output.

9. Riassunto delle modalità di cambio di modo

Riassumiamo i meccanismi che comportano un cambiamento del modo di funzionamento del processore:

- quando si esegue una istruzione SYSCALL oppure si verifica un interrupt che viene accettato 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)
- quando si esegue l'istruzione IRET, che viene utilizzata alla fine di una routine di interrupt, o l'istruzione SYSRET alla fine della funzione system_call, queste eseguono il ritorno al programma che era stato interrotto o che aveva eseguito una SYSCALL prelevando dalla pila l'indirizzo di ritorno e ricostituiscono il modo originario del processore.

Meccanismo di salto	**Modo di partenza**	**Modo di arrivo**	**Meccanismo di ritorno**	**Modo dopo il ritorno**
salto a funzione normale	U S	U S	istruzione di ritorno	U S
SYSCALL	U	S	SYSRET	U
interrupt	U S	S S	IRET	U S

Tabella 2

In tabella 2 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 SYSCALL oppure quando si verifica un interrupt;
- Il passaggio dall'esecuzione del SO a un generico processo avviene tramite le istruzioni SYSRET e IRET, che ritornano al processo in modo U

10. Interfacce Standard e Application Binary Interface (ABI)

In Figura 11 sono riportate le interfacce più importanti per la comprensione dei diversi componenti del sistema a livello di programmi sorgenti e eseguibili

Consideriamo prima il lato dei sorgenti.
Alcune di queste interfacce sono già note dalla programmazione in C e sui thread, in particolare la libreria C e l'API POSIX.
L'API POSIX costituisce un'interfaccia alle funzioni della libreria glibc, che è una implementazione di tale interfaccia. La libreria glibc a sua volta invoca le interfacce standard del sistema operativo LINUX.

Passiamo ora al lato degli eseguibili.
Il programma applicativo viene compilato e collegato in modo da produrre un eseguibile.
Anche i sorgenti di LINUX vengono compilati e collegati, in modo da costituire un sistema operativo eseguibile, ma il suo formato non è quello di un eseguibile normale, perchè gli eseguibili normali vengono caricati dal SO, invece il SO deve essere in grado di partire da solo (bootstrap).
Le regole che governano il modo in cui un compilatore conforme a gnu deve tradurre i sorgenti sono definite dalla Application Binary Interface (ABI), in particolare, con riferimento a LINUX e all'architettura x64, nel documento *"System V Application Binary Interface – AMD64 Architecture Processor Supplement"*. Come dice il titolo, le regole sono dipendenti dall'architettura.

Le regole dell'ABI servono a garantire che tutti i moduli siano tradotti in modo coerente; ad esempio per poter collegare una funzione applicativa a una funzione di libreria è necessario che le convenzioni adottate per il passaggio dei parametri siano uguali nel chiamante e nel chiamato. Abbiamo visto un esempio di tali convenzioni nella prima parte del corso, relativamente all'architettura MIPS.

Queste regole definiscono numerosi aspetti all'eseguibile che viene prodotto; noi vogliamo analizzare qui le regole relative alla chiamata del sistema operativo (SYSCALL) per il x64, cioè il modo di passare i parametri alla funzione **system_call( )** che costituisce il punto di entrata dei servizi del sistema. Queste regole sono:

- passare il numero del servizio da invocare nel registro **rax**
- passare eventuali parametri (che dipendono dal servizio richiesto) ordinatamente nei registri **rdi, rsi, rdx, r10, r8, r9**

Generalmente un programma applicativo non invoca la SYSCALL direttamente, ma invoca una funzione della libreria **glibc** che a sua volta contiene la chiamata di sistema.

Nella libreria glibc sono presenti funzioni che corrispondono ai servizi offerti dal SO, ad esempio `fork( )`, `open( )`, ecc... Neppure queste funzioni eseguono direttamente la istruzione SYSCALL, ma invocano una funzione della stessa libreria glibc che incapsula la SYSCALL; quest'ultima funzione è dichiarata nel modo seguente (attenzione – ha lo stesso nome dell'istruzione del x64 – per evitare confusione l'istruzione assembler è scritta in maiuscolo e la funzione in minuscolo):.

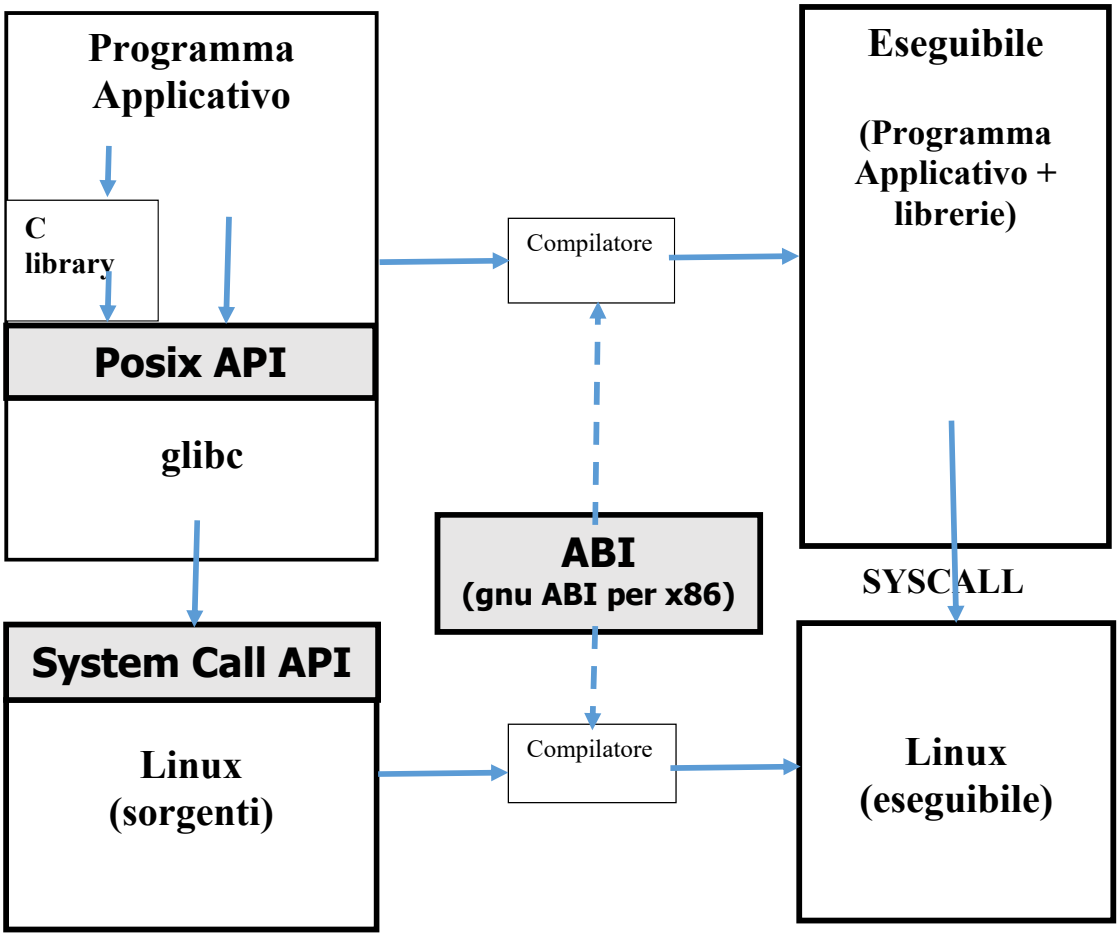
```
long syscall(long numero_del_servizio, ...parametri del servizio ...);
```

Syscall( ) è una funzione di livello molto basso. Il suo compito è solo quello di incapsulare le convenzioni per la chiamata della SYSCALL. In un programma applicativo è possibile utilizzare direttamente syscall( ) invece delle funzioni di più alto livello, ma ovviamente non è consigliabile.

Esempio: quando la funzione read( ) della libreria glibc viene invocata dai programmi applicativi, si verificano i seguenti passaggi (SYS_read è la costante simbolica che definisce il numero del servizio read):

1. programma → read(fd, buf, len) //in glibc, modo U
2. read(fd, buf, len) → syscall(SYS_read, fd, buf, len); //in glibc, modo U

3. syscall( ):
    - pone SYS_read nel registro rax
    - pone fd, buf, len nei registri rdi, rsi, rdx
    - esegue istruzione SYSCALL //passaggio a modo S
4. inizia la funzione system_call, che invoca la funzione opportuna per eseguire il servizio read
5. esecuzione del servizio read
6. il servizio ritorna alla funzione system_call
7. la funzione system_call esegue l'istruzione SYSRET per tornare al processo che ha richiesto il servizio



API = Application Program Interface
ABI = Application Binary Interface

Figura 11 – Interfacce standard

Le costanti simboliche che definiscono i numeri dei servizi sono definite nel file `/linux/include/asm-x86/unistd_64.h`. Questo file contiene delle macro la cui interpretazione ci dice che ad esempio read è

associata al numero 0 ed è implementata da un funzione chiamata `sys_read`. In figura 12 è riportata la stampa della parte iniziale di tali definizioni.

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8
0	sys_read	unsigned int fd	char *buf	size_t count		
1	sys_write	unsigned int fd	const char *buf	size_t count		
2	sys_open	const char *filename	int flags	int mode		
3	sys_close	unsigned int fd				
4	sys_stat	const char *filename	struct stat *statbuf			
5	sys_fstat	unsigned int fd	struct stat *statbuf			
6	sys_lstat	fconst char *filename	struct stat *statbuf			
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs		
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin		
9	sys_mmap	unsigned long addr	unsigned long len	unsigned long prot	unsigned long flags	unsigned long f
10	sys_mprotect	unsigned long start	size_t len	unsigned long prot		
11	sys_munmap	unsigned long addr	size_t len			
12	sys_brk	unsigned long brk				
			const struct	struct sigaction	size_t	

Figura 12 – Tabella di definizione delle System Call
(parte iniziale – attualmente le System Call sono 322)