POLITECNICO
MILANO 1863

Architettura dei calcolatori e sistemi operativi

Il Nucleo del Sistema Operativo

N2 – Meccanismi Hardware di supporto al Sistema Operativo

Aspetti generali dell'architettura x64

- La realizzazione di un SO multiprogrammato come Linux o Windows richiede da parte dell'Hardware la disponibilità di alcuni meccanismi fondamentali
- Analizziamo tali funzionalità, con riferimento specifico al x64
- Semplificazione: alcune funzionalità del x64 sono inutilmente complesse (per motivi di compatibilità con diversi modi di funzionamento, che non ci interessano) e quindi ne verrà fornita una versione semplificata

Nel x64 esistono numerosi registri a 64 bit:

- registri usabili dal programmatore, che citeremo solo quando serviranno
- **registro PC** (program counter)
- **registro SP** (stack pointer)

Pila e salti a funzione

Pila

- nel x64 la pila cresce da indirizzi alti verso indirizzi bassi (come nel MIPS)
- a differenza del MIPS, il decremento e l'incremento dello SP sono svolti nella stessa istruzione di scrittura e lettura in memoria
- *push* e *pop* della pila richiedono una sola istruzione ciascuna

Salto a funzione

- il x64, a differenza del MIPS, salva il valore dell'indirizzo di ritorno sulla pila, non in un registro
- l'istruzione di salto a funzione esegue le seguenti operazioni:
  - Decrementa il registro SP
  - Salva sulla pila valore del PC incrementato

- L'istruzione di ritorno da funzione esegue le seguenti operazioni:
  - preleva il valore di PC (indirizzo di ritorno) dalla pila
  - Incrementa poi lo SP

Relazione MIPS – x64

Per capire la relazione tra le rispettive istruzioni, si supponga che l'area di attivazione del callee contenga solo l'indirizzo di ritorno (rx è un registro generico).

x64 (ISA esteso)	MIPS (ISA ridotto)
`PUSH rx`	`addiu $sp, $sp, -4` `sw $rx, ($sp)`
`POP rx`	`lw $rx, ($sp)` `addiu $sp, $sp, 4`

`CALL FUNCT // nel caller`	`jal FUNCT // nel caller`
	`addiu $sp, $sp, -4 // nel callee`
	`sw $ra, ($sp) // nel callee`

`RET // nel callee`	`lw $ra, ($sp) // nel callee`
	`addiu $sp, $sp, 4 // nel callee`
	`jr $ra // nel callee`

Sia MIPS sia x64 poi completeranno la costruzione dell'area come da rispettivi standard.

Naturalmente i registri di x64 sono a 64 bit e quelli di MIPS sono a 32 bit.

Strutture dati ad accesso HW

sono registri e strutture dati in memoria che ***l'Hardware accede autonomamente*** per eseguire alcune operazioni

il Sistema Operativo può accedere tali strutture per:

- impostare dei valori che governano il funzionamento dell'HW
- leggere dei valori per conoscere lo stato dell'Hardware

registro di stato, **PSR (Processor Status Register)**

- 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
- tutti gli aspetti descritti relativamente al PSR non corrispondono ai reali meccanismi del x64, che sono più complessi

Modi di funzionamento – istruzioni privilegiate

Il processore ha la possibilità di funzionare in due stati o **modi** diversi:

- modo ***Utente*** (**User Mode** detto anche **non privilegiato**)
- modo ***Supervisore*** (detto anche **kernel mode** o **privilegiato**)

- il processore in modo S può eseguire tutte le proprie istruzioni e può accedere a tutta la propria memoria
- il processore 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** (ad esempio, istruzioni di I/O o di arresto della macchina)
- quando viene eseguito il SO il processore è in modo S, mentre quando vengono eseguiti i normali programmi esso è in modo U
- nel x64 esistono 4 modi, con livelli crescenti di privilegio, ma Linux ne usa solo i 2 estremi
- il modo di funzionamento è rappresentato da un bit del PSR

Chiamata al sistema operativo - SYSCALL

Esiste un'istruzione, **SYSCALL**, non privilegiata, che realizza un ***salto al SO*** (cambio di modo di esecuzione da U a S)

SYSCALL (simile a un salto a funzione) opera nel modo seguente

- il valore del PC incrementato viene salvato sulla pila
- il valore del PSR viene salvato sulla pila
- nel PC e nel PSR vengono caricati i valori presenti in una struttura dati ad accesso HW detta **Vettore di Syscall**

LINUX inizializza il **Vettore di Syscall** durante la fase di avviamento del sistema, con la coppia

- indirizzo della *funzione* **system_call( )**
- PSR opportuno per l'esecuzione di *system_call( ), modo privilegiato*

system_call( ) costituisce quindi il punto di entrata unico per tutti i servizi di sistema di LINUX

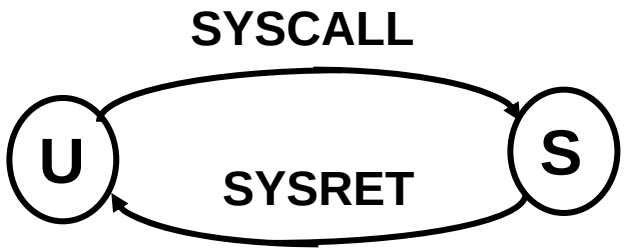
Ritorno dal sistema operativo - SYSRET

Esiste un'istruzione **SYSRET**, privilegiata, che esegue le seguenti operazioni

- carica nel PSR il valore presente sulla pila
- carica nel PC il valore presente sulla pila

In Linux l'istruzione SYSRET è eseguita alla fine della funzione system_call( )

- costituisce quindi l'unico punto di uscita dal Sistema Operativo e di ritorno al processo che ha invocato un servizio.



Modello della memoria – protezione del SO

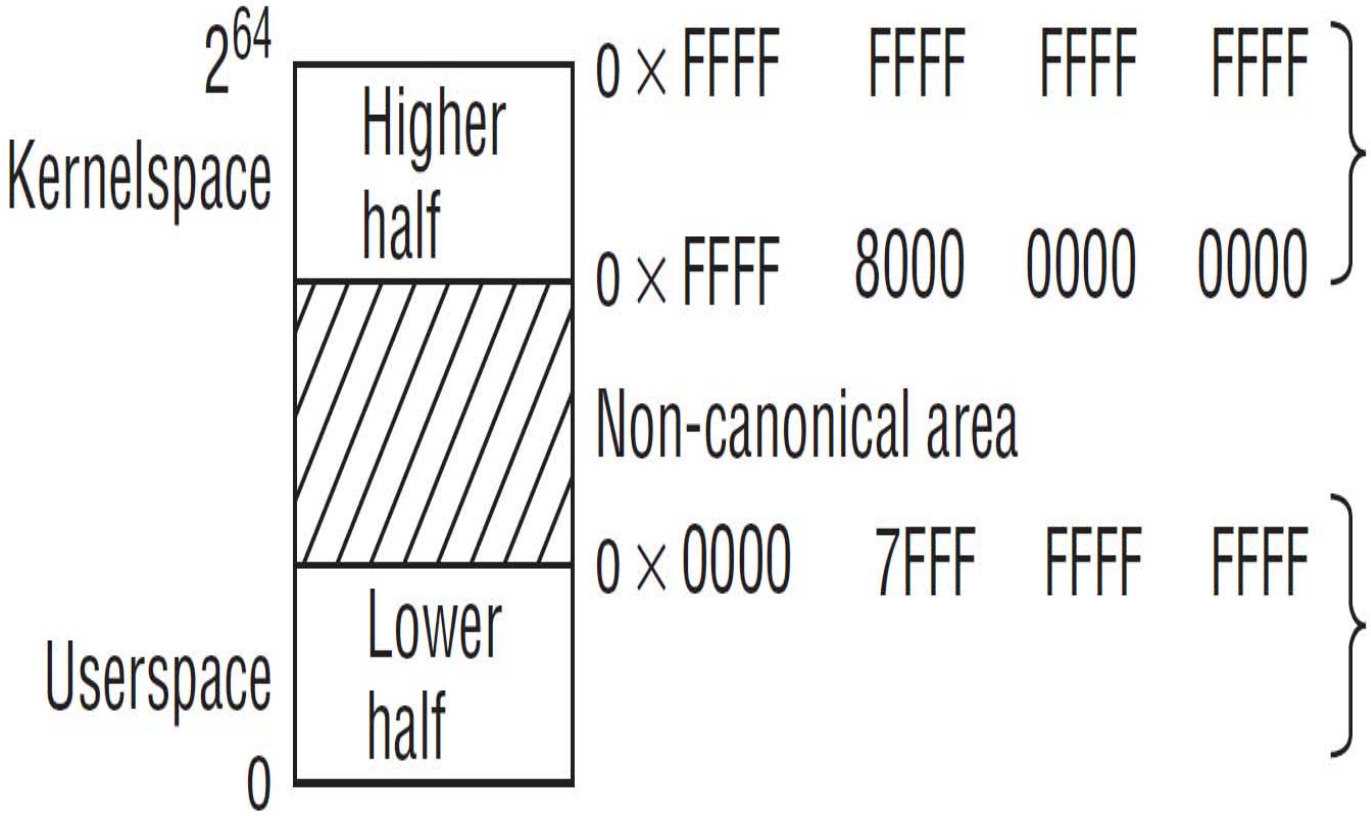
- Quando il processore è in modo U non deve poter accedere alle zone di memoria riservate al SO
- Viceversa, quando il processore è in modo S deve poter accedere sia alla memoria del SO, sia alla memoria dei processi

Struttura degli indirizzi

- ✓ spazio di indirizzamento potenziale del x64 è di 2^{64} byte
- ✓ 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)
  - lo spazio di modo U occupa i primi $\mathbf{2^{47}}$ byte (da 0 a 0000 7FFF FFFF FFFF)
  - lo spazio di modo S occupa i $\mathbf{2^{47}}$ byte di indirizzo più alto (da FFFF 8000 0000 0000 a FFFF FFFF FFFF FFFF)
- ✓ gli indirizzi intermedi sono detti *non-canonici* e se utilizzati generano un errore

- La CPU in modo S può utilizzare tutti gli indirizzi canonici
- in modo U un indirizzo superiore a 0000 7FFF FFFF FFFF genera un errore

Modello della memoria



Cenni alla paginazione

La memoria del x64 è gestita tramite paginazione (argomento trattato in dettaglio più avanti)

Le seguenti caratteristiche della paginazione sul x64 sono sufficienti alla comprensione del nucleo

- la memoria è suddivisa in unità dette **pagine**, di dimensione **4Kb** (12 bit di offset)
- 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**

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 dal registro 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 due pile quando è necessario
  - le 2 pile sono allocate nei corrispondenti spazi virtuali di modo U e di modo S

Linux alloca ad ogni processo una sPila costituita da 2 pagine (8K)

Esempio di indirizzamento della pila

- ✓ Si considerino i valori prodotti dal modulo axo_hello con la funzione task_explore, riportati in tabella
- ✓ Le **pagine** di memoria hanno dimensione **4KB** (12 bit di offset) => le ultime 3 cifre esadecimali sono l'offset nella pagina, quelle precedenti il numero di pagina
- ✓ la pila di sistema **sPila** va da 0xFFFF 8800 5C64 4000 a 0xFFFF 8800 5C64 6000
- ✓ la pila di utente **uPila** è nello spazio U (0x 0000 7FFF 6DA9 8C78)

variabile	indirizzo	significato
thread.sp0	0xFFFF 8800 5C64 6000	inizio sPila
ts-> stack	0xFFFF 8800 5C64 4000	fine sPila
thread.sp	0xFFFF 8800 5C64 5D68	SP di sPila
usersp	0x0000 7FFF 6DA9 8C78	SP di uPila

Commutazione di pila

nella commutazione da modo U a modo S (SYSCALL o interrupt) la *commutazione di pila* avviene ***prima*** del *salvataggio di informazioni* sulla stessa

- l'indirizzo di ritorno a modo U deve essere salvato su sPila
- nel ritorno da modo S a modo U l'informazione per il ritorno verrà prelevata da sPila, cioè prima di commutare a uPila

Per poter ***commutare da modo U a modo S e attivare la sPila*** è necessaria una opportuna «struttura dati» basata su due celle di memoria **USP** e **SSP** (usiamo un modello semplificato rispetto a quello del x64)

✓***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 (*base di*) sPila del processo in esecuzione

✓in **USP** viene salvato il valore del registro SP (valore corrente dello SP relativo a uPila) al momento del passaggio a modo S

Commutazione di pila

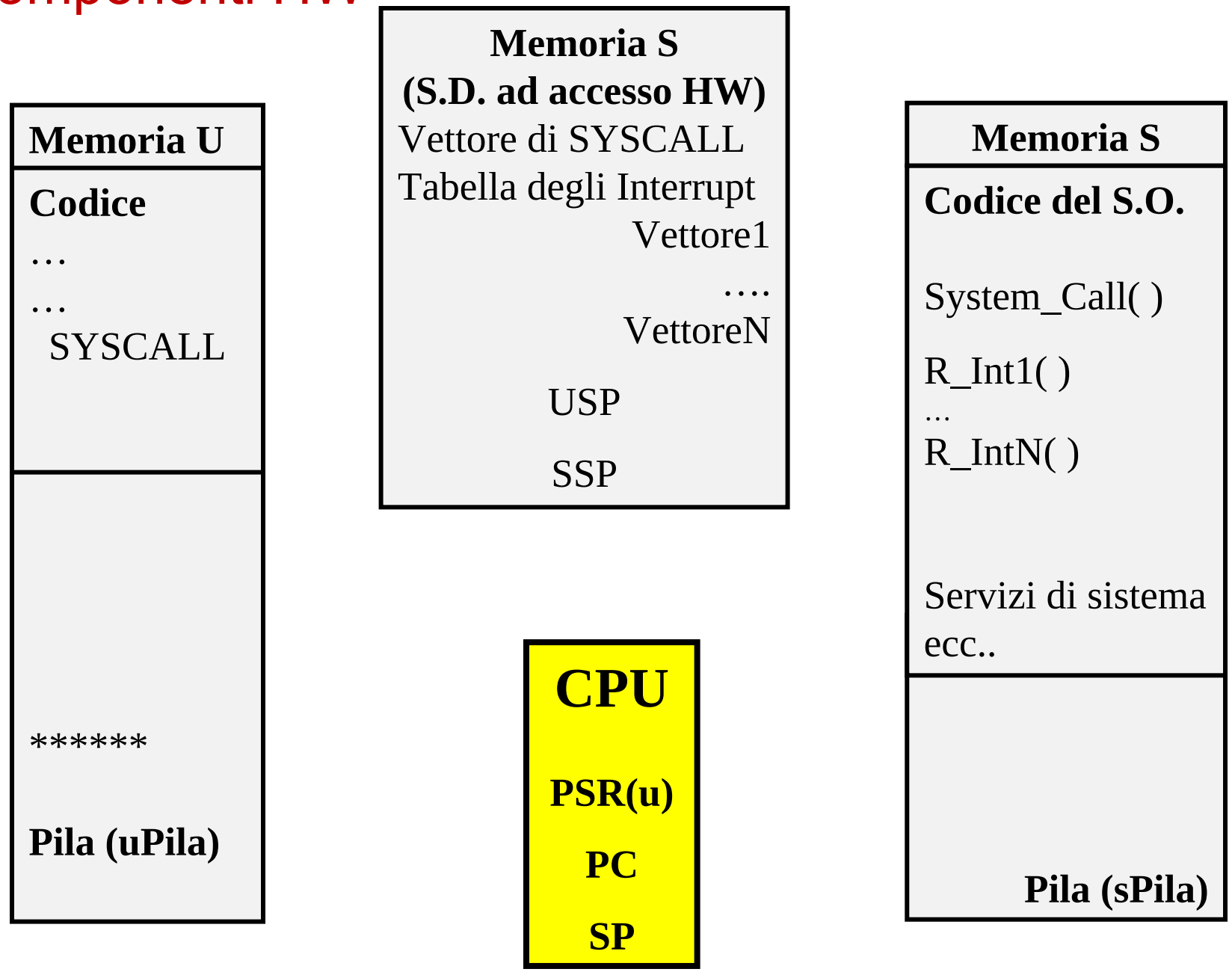
Complessivamente le operazioni svolte da **SYSCALL** sono quindi

- salva il valore corrente di SP in USP
- carica in SP il valore presente in SSP (adesso SP punta alla *base di* 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)

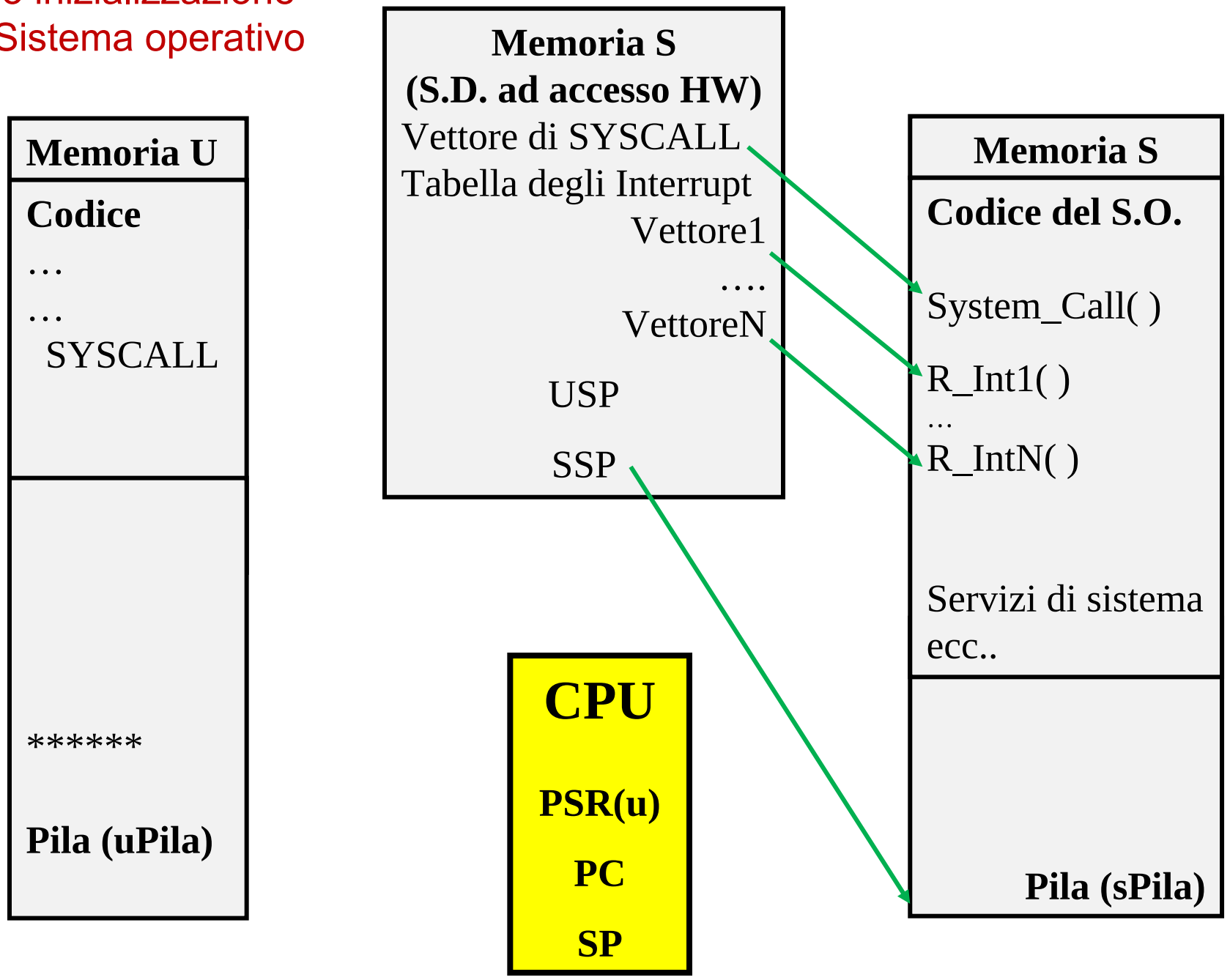
Simmetricamente, le operazioni svolte da **SYSRET** sono

- carica in PSR il valore presente in sPila (modo U)
- carica in PC il valore presente in sPila
- carica in SP il valore presente in USP (adesso SP punta nuovamente a uPila, *valore corrente*)

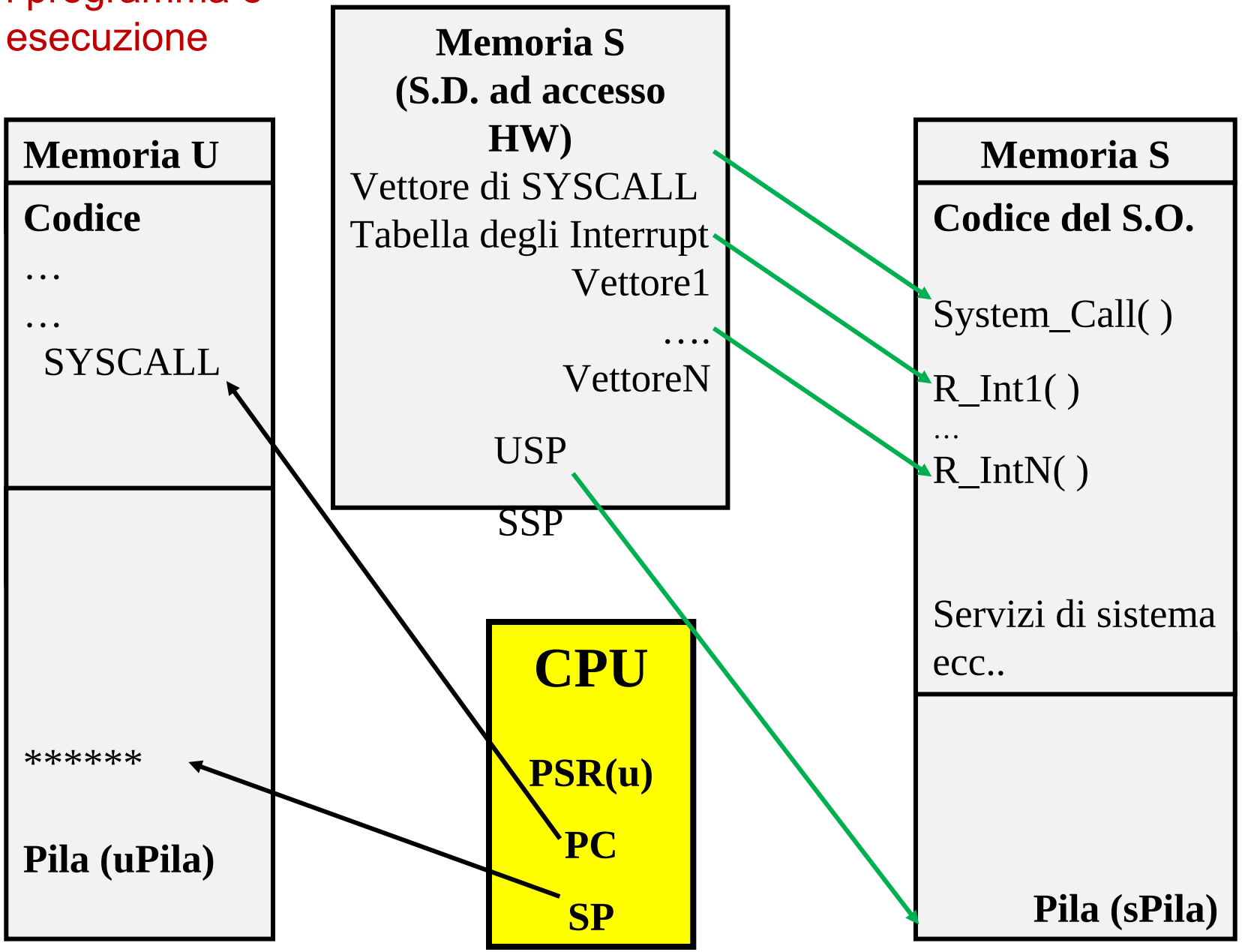
Componenti HW



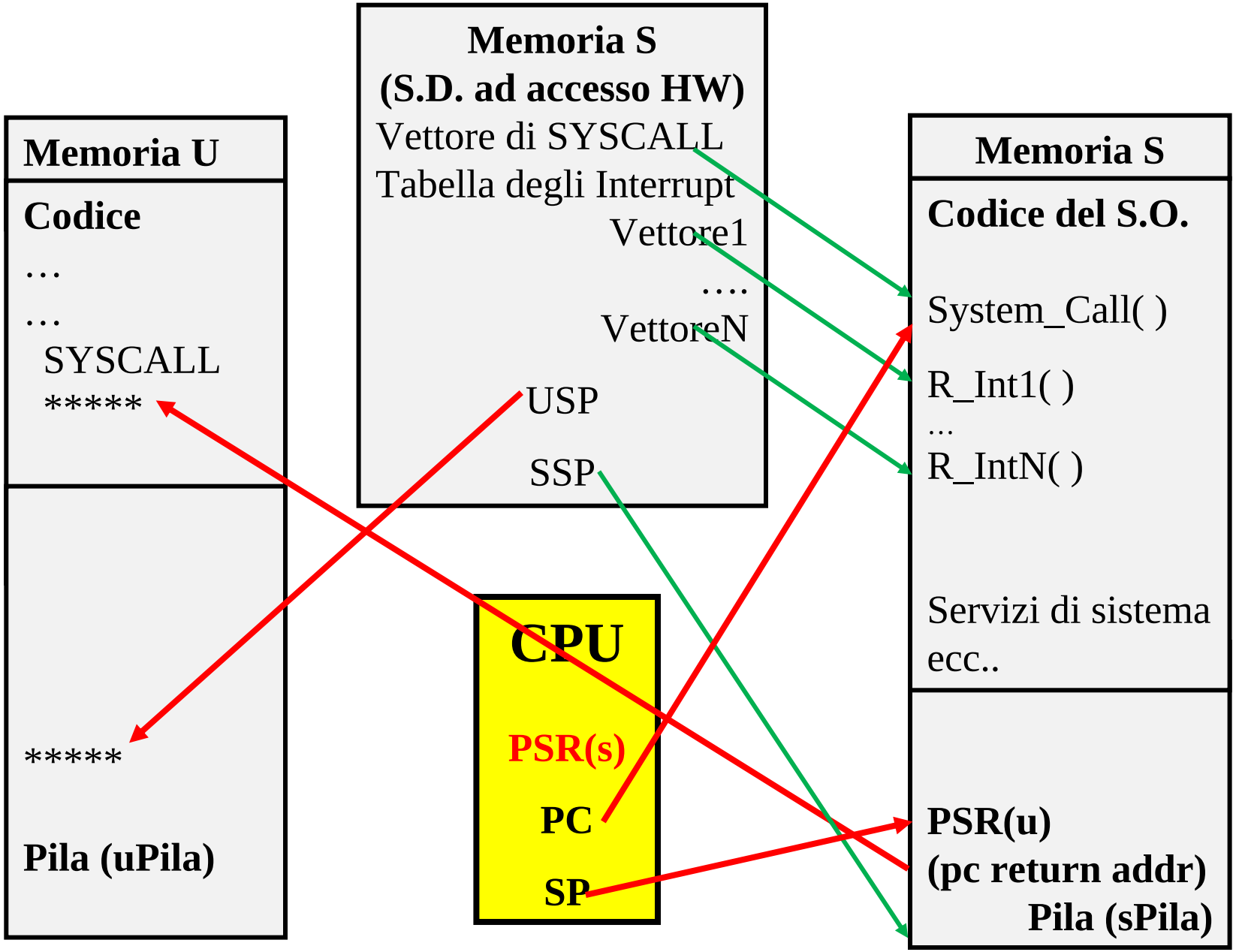
Dopo inizializzazione
del Sistema operativo



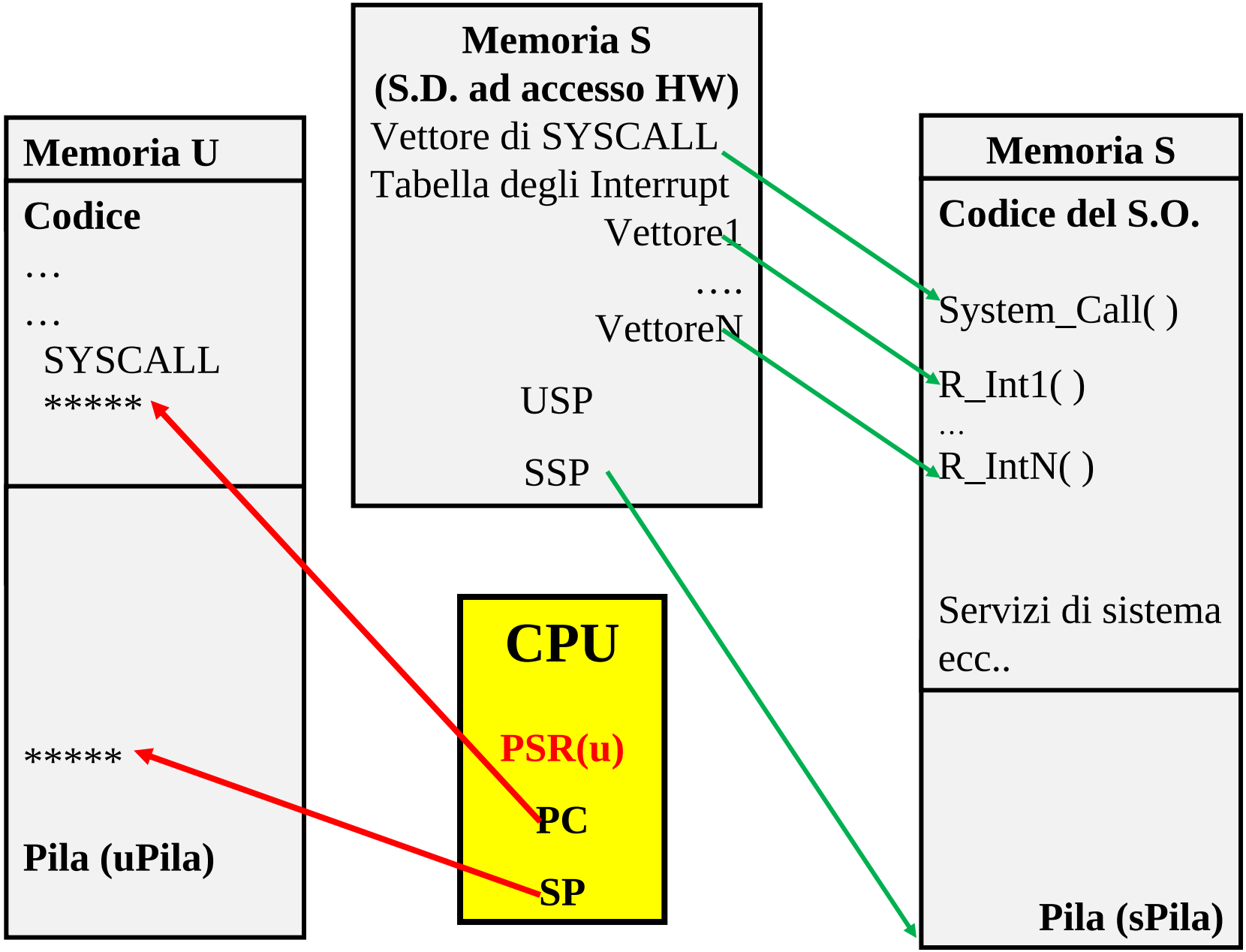
Un programma è in esecuzione



Esecuzione SYSCALL



Esecuzione SYSRET



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.
- Nel x64 esiste un registro, **CR3** (CR sta per Control Register) che contiene l'indirizzo del 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, facendolo puntare a una diversa tabella delle pagine

Meccanismo di Interruzione (Interrupt) - 1

- Esiste un insieme di **eventi** rilevati dall'Hardware (ad esempio, un particolare segnale proveniente da una periferica, una condizione di errore, ecc...)
- Ad ogni evento è associata una particolare funzione detta **gestore dell'interrupt** o **routine di interrupt**
- Le ***routine di interrupt fanno parte del SO***
- quando il processore rileva un evento, esso "interrompe" il programma correntemente in esecuzione ed esegue un salto all'esecuzione della funzione associata a tale evento
  - l'esecuzione di una *routine di interrupt* comporta il passaggio da modo U a modo S oppure quello da modo S a modo S: il modo di arrivo è sempre S
- quando la routine di interrupt termina, il processore riprende l'esecuzione del programma che è stato interrotto

Meccanismo di Interruzione (Interrupt) - 2

- Per poter riprendere l'esecuzione il processore ha salvato sulla pila, al momento del salto alla routine di interrupt, l'***indirizzo della prossima istruzione del programma interrotto***
  - oltre al salvataggio dell'indirizzo al programma interrotto viene anche salvato il ***modo***, cioè il registro PSR
- Dopo l'esecuzione della routine di interrupt tale indirizzo è disponibile per eseguire il ritorno
- l'istruzione ***privilegiata*** che esegue il ritorno da interrupt, ripristinando anche il modo, è detta **IRET**
- Il meccanismo di interrupt è a tutti gli effetti simile ad un'invocazione di funzione o a una SYSCALL
- **ma** le routine di interrupt sono 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 Interruzione (Interrupt) - 3

- Il meccanismo di interrupt si combina con il doppio modo di funzionamento S ed U in maniera simile a quello della SYSCALL
- 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 il PSR dalla pila sPila

Meccanismo di Interruzione (Interrupt) - 4

- Il processore deve sapere quale è *l'indirizzo della routine di interrupt* che deve essere eseguita quando si verifica un certo evento e il *valore di PSR da utilizzare* (anche per ***le priorità*** degli interrupt)
- la **Tabella degli Interrupt** è una struttura dati ad accesso HW e contiene un certo numero di ***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
- Il verificarsi di un nuovo interrupt durante l'esecuzione di una routine di interrupt (**interrupt annidati**) viene gestito correttamente, esattamente come l'annidamento delle invocazioni di funzioni

Commutazione di pila: Interrupt e IRET

Se già in modo S le prime due operazioni non vanno eseguite

Complessivamente le operazioni svolte dall'**Interrupt** sono :

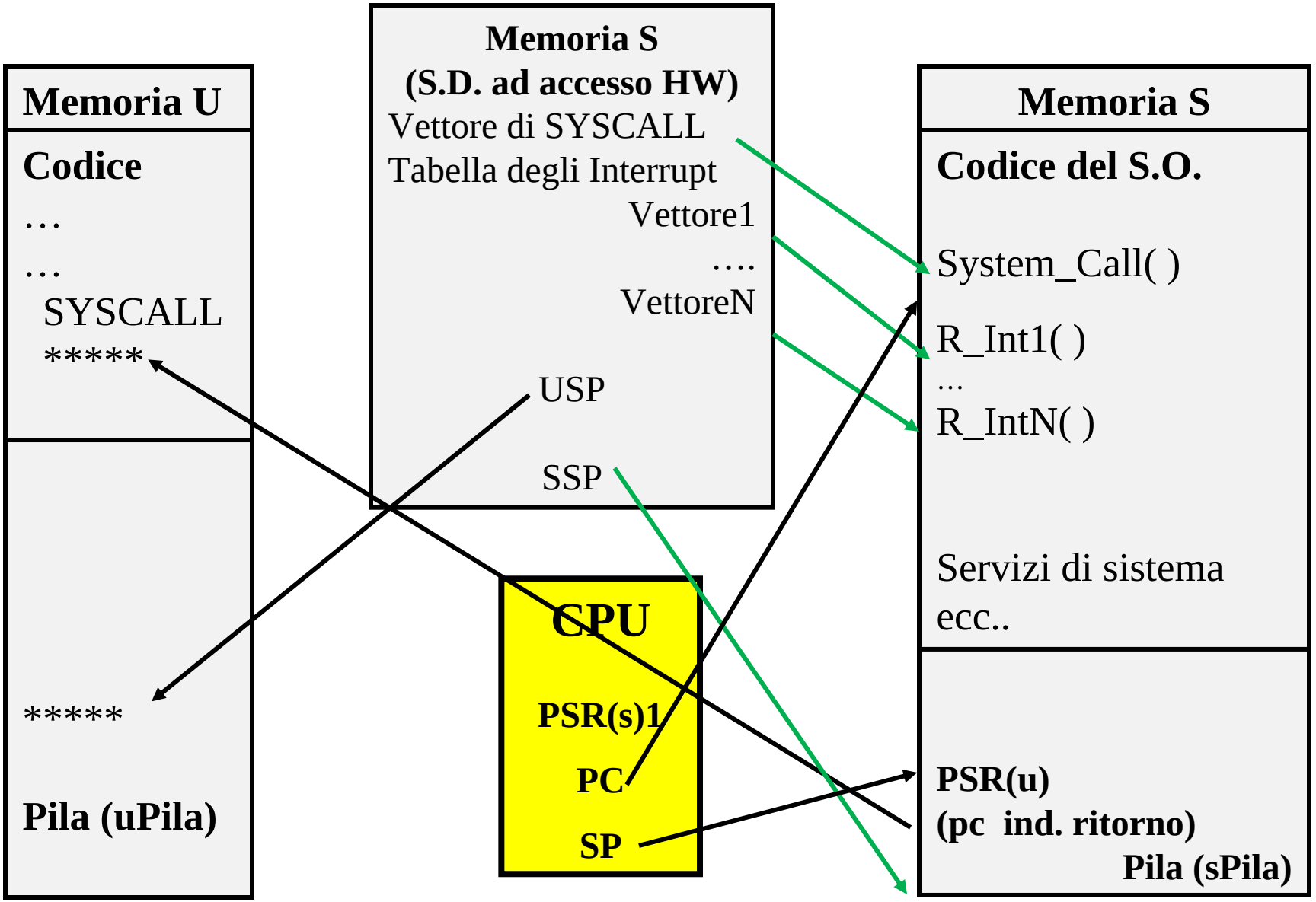
- salva il valore corrente di **SP** in **USP**
- carica in **SP** il valore presente in **SSP** (adesso **SP** punta in **sPila**)
- salva su **sPila** l'indirizzo di ritorno al programma interrotto
- salva su **sPila** il valore del **PSR** del programma interrotto
- carica in **PC** e **PSR** i valori presenti nel **Vettore di Interrupt** (il modo di funzionamento passa quindi a **S** *indipendentemente da come era prima:* poteva essere modo U se l'interrupt aveva interrotto un normale processo oppure poteva essere già modo S se l'interrupt aveva interrotto il SO)

Simmetricamente, le operazioni svolte da **IRET** sono:

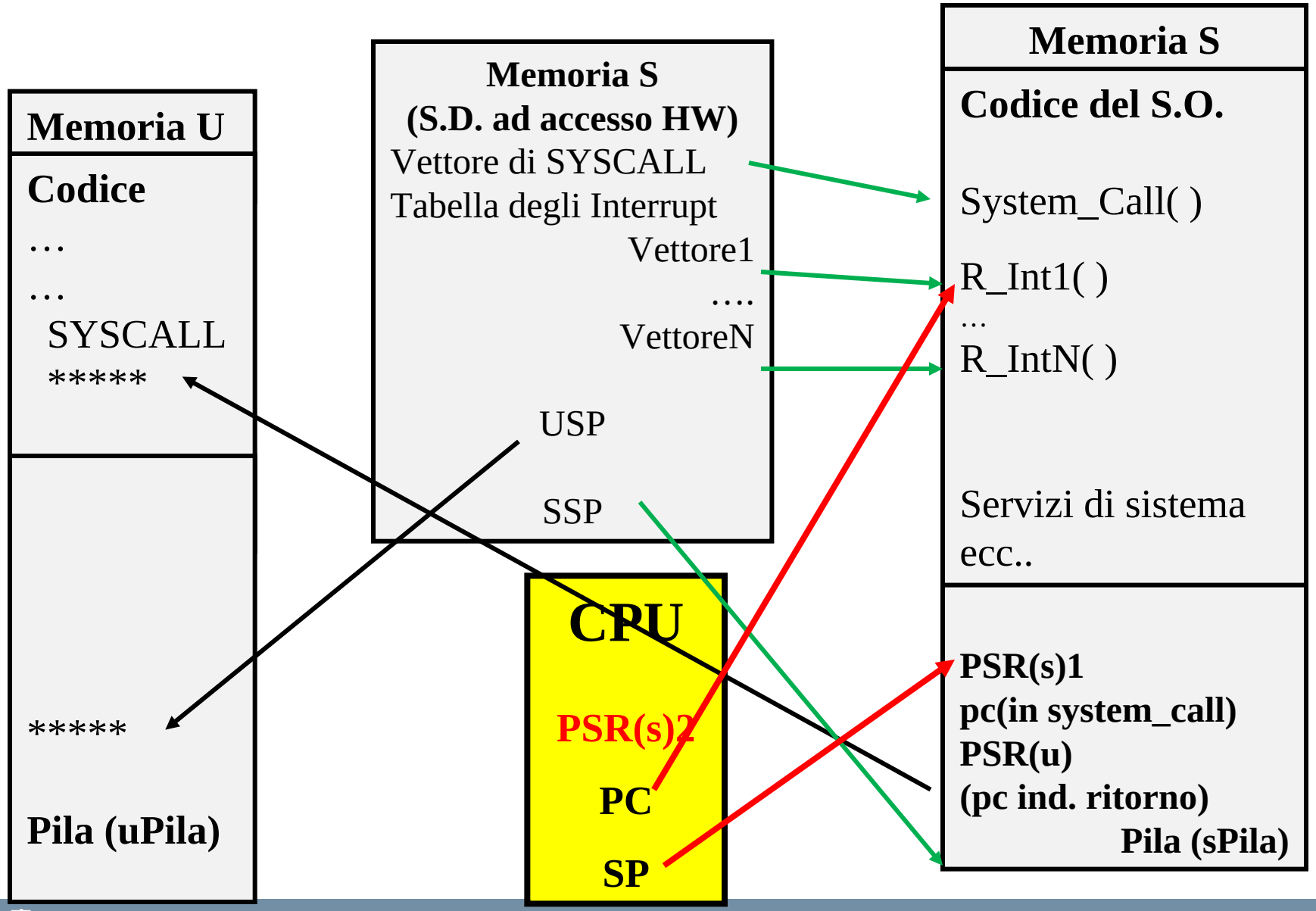
- carica in **PSR** il valore del **PSR** presente in **sPila**
- carica in **PC** il valore dell'indirizzo di ritorno presente in **sPila**
- carica in **SP** il valore presente in **USP** (adesso **SP** punta nuovamente a **uPila**)

Se ritorna a modo S l'ultima operazione non viene eseguita

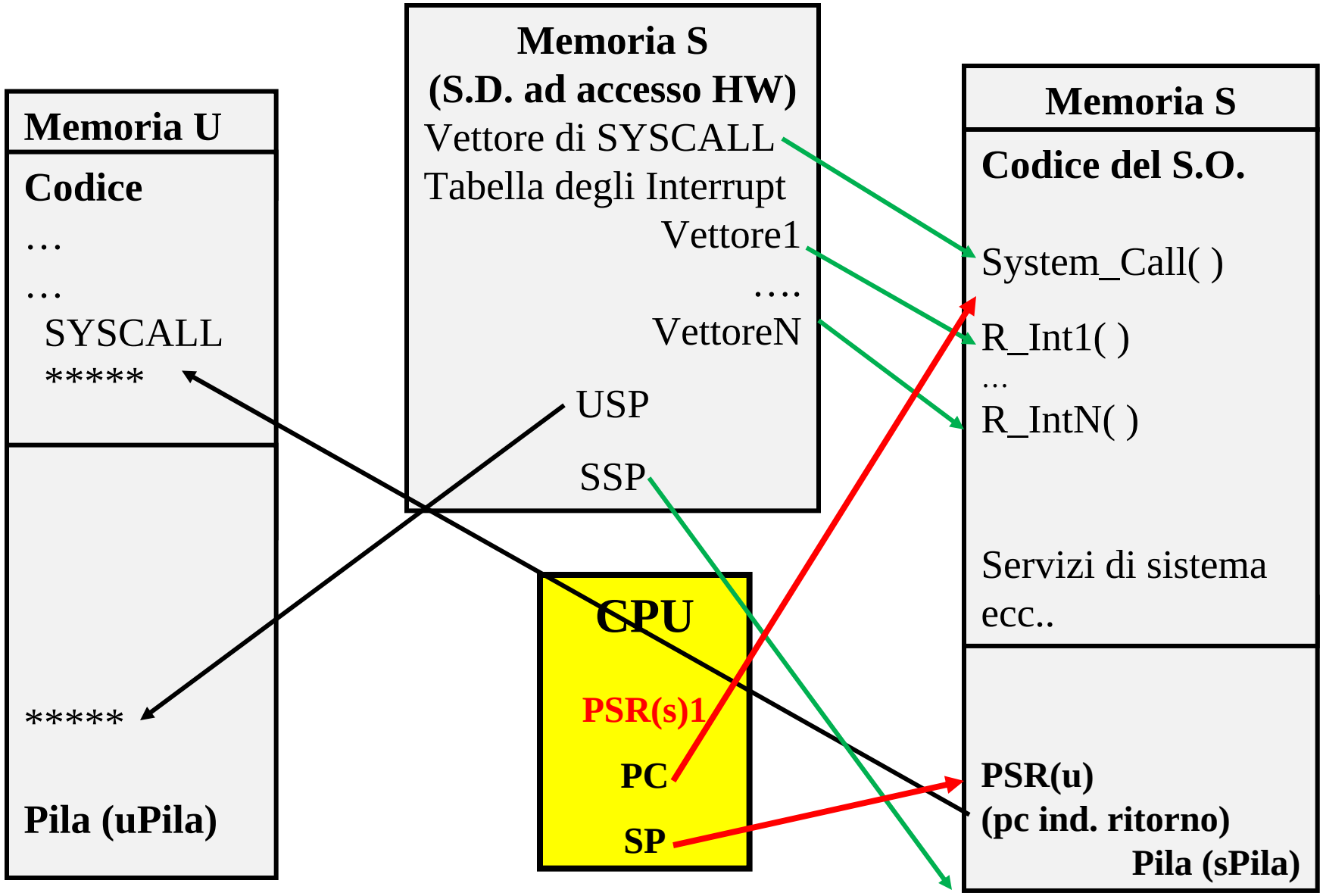
Stato Iniziale (dopo Esecuzione SYSCALL in Esempio1)



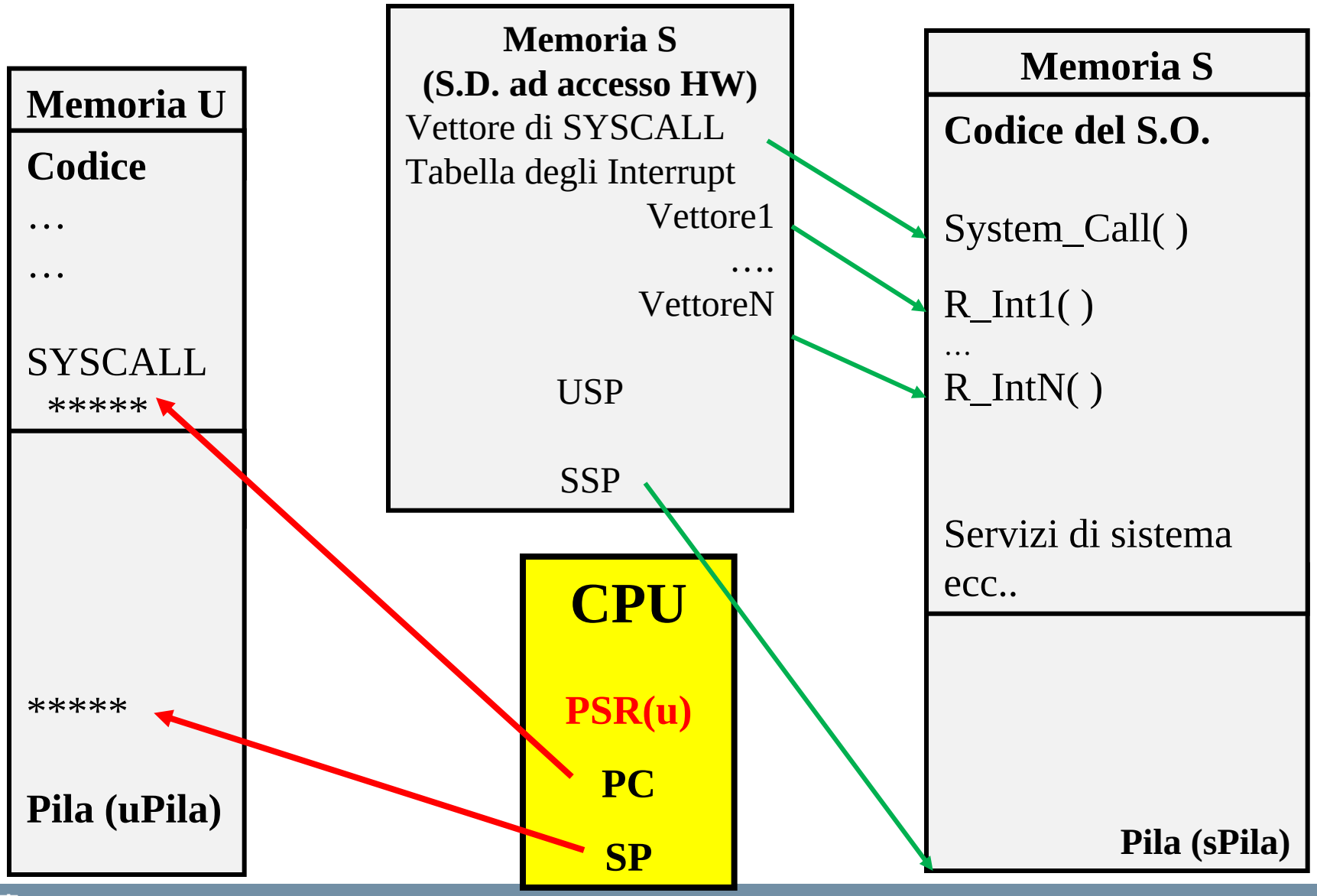
Si verifica Interrupt1 durante l'esecuzione della system_call ( )



R_Int1 esegue IRET



Esecuzione SYSRET



Interrupt e gestione degli errori

- Durante l'esecuzione delle istruzioni possono verificarsi degli errori che impediscono al processore di proseguire; ad esempio:
  - divisione con divisore 0,
  - uso di indirizzi di memoria non validi
  - 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

- Interrupt possono essere annidati
  - Non sempre è opportuno permettere a un interrupt di interrompere la routine che serve un altro interrupt
  - Necessario prevedere che un evento molto importante e che richiede una risposta urgente possa interrompere la routine di interrupt che serve un evento meno importante, ma non il contrario

Meccanismo di priorità dell'interrupt

- Il processore possiede un **livello di priorità** definito nel registro PSR
- Il livello di priorità del processore può essere modificato dal software tramite opportune istruzioni macchina che scrivono nel PSR
- Agli interrupt viene associato un livello di priorità
- Un interrupt viene accettato, se e 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

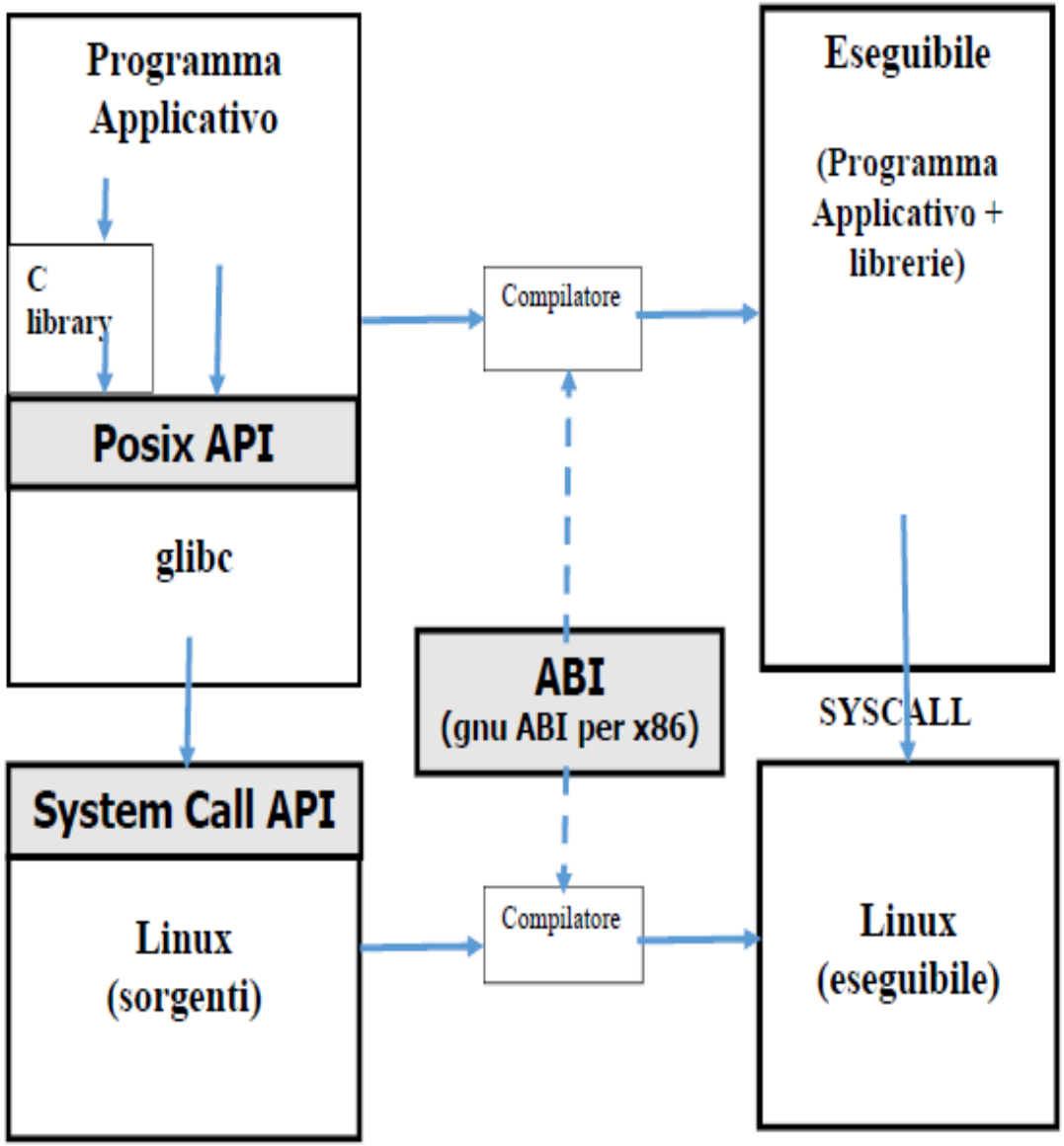
Utilizzando questo meccanismo Hardware il sistema operativo può alzare e abbassare la priorità del processore in modo che durante l'esecuzione delle routine di interrupt più importanti non vengano accettati interrupt meno importanti

Riassunto delle modalità di cambio di modo

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

Interfacce Standard e Application Binary Interface (ABI)

- Le regole che governano il modo in cui un compilatore conforme a gnu deve tradurre i sorgenti sono definite dalla **ABI**
- Queste regole servono per garantire che tutti i moduli siano tradotti in modo coerente con le convenzioni adottate per il passaggio dei parametri



API = Application Program Interface
ABI = Application Binary Interface

ABI – regole di invocazione del SO

- Passaggio di parametri alla funzione ***system_call( )*** per x86
  - 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...
- queste funzioni invocano una funzione della stessa libreria *glibc* che incapsula la SYSCALL; quest'ultima funzione (*wrapper function)* è dichiarata nel modo seguente

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

I numeri dei servizi sono codificati nella seguente tabella

%rax	System call	%rdi	%rsi	%rdx	%r10
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
10	sys_mprotect	unsigned long start	size_t len	unsigned long prot	

Esempio: invocazione 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