Meccanismi *HW* di Supporto al Sistema Operativo

Aspetti Generali dell'Architettura *x64*

- la realizzazione di un *SO* multi-programmato come Linux o Windows richiede da parte dello hardware la disponibilità di alcuni meccanismi fondamentali
- qui si analizzano tali funzionalità, con riferimento specifico all'architettura *x64*
- semplificazione: alcune funzionalità di *x64* sono inutilmente complesse, per motivi di compatibilità con diversi modi di funzionamento che qui non interessano, e quindi ne verrà fornita una versione semplificata
- in *x64* si trovano numerosi registri a 64 bit:
  - registri usabili dal programmatore, citati solo quando serviranno
  - registro *PC* (*Program Counter*)
  - registro *SP* (*Stack Pointer*)

Pila e Salto a Funzione

- struttura e funzionamento della pila:
  - in *x64* la pila cresce da indirizzi alti verso indirizzi bassi (come in *MIPS*)
  - a differenza di *MIPS*, il decremento e l'incremento di *SP* sono svolti nella stessa istruzione di scrittura in memoria
  - le operazioni *push* e *pop* della pila richiedono una sola istruzione ciascuna
- salto a funzione:
  - il processore *x64*, a differenza di *MIPS*, salva il valore dell'indirizzo di ritorno sulla pila, non in un registro
  - l'istruzione di salto a funzione esegue le seguenti operazioni:
    - il registro *SP* viene decrementato
    - il valore di *PC* incrementato viene salvato sulla pila
  - l'istruzione di ritorno da funzione preleva il valore di *PC* dalla pila e poi incrementa *SP*

Relazione *MIPS – x64*

per capire la relazione tra le rispettive istruzioni, si supponga che l'area di attivazione del *chiamato* (*callee*) contenga solo l'indirizzo di ritorno

x64	MIPS
`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`

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 lo Hardware accede autonomamente per eseguire alcune operazioni
- il sistema operativo può accedere tali strutture per:
  - impostare dei valori che governano il funzionamento dello *HW*
  - leggere dei valori per conoscere lo stato dello *HW*
- registro di stato, chiamato ***PSR*** (*Program Status Register*):
  - contiene tutta l'informazione di stato che caratterizza la situazione del processore
  - escluse alcune informazioni per le quali si indicheranno esplicitamente dei registri dedicati a contenerle
  - tutti gli aspetti descritti relativamente a *PSR* non corrispondono ai reali meccanismi di *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*** (detto anche ***non privilegiato***)
  - modo ***supervisore*** (detto anche ***kernel*** 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*** (per 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 quattro modi, con livelli crescenti di privilegio, ma Linux ne usa solo i due estremi
- il modo di funzionamento è rappresentato da un bit di *PSR*

Chiamata a Sistema Operativo

- c'è un'istruzione macchina, chiamata ***SYSCALL***, non privilegiata (ma con un comportamento assai speciale), che realizza un salto al *SO*
- l'istruzione macchina *SYSCALL* opera nel modo seguente:
  - il valore di *PC* incrementato viene salvato sulla pila
  - il valore di *PSR* viene salvato sulla pila
  - in *PC* e in *PSR* vengono caricati i valori presenti in una struttura dati ad accesso *HW* detta ***vettore di syscall***
- il *SO* Linux inizializza il ***vettore di syscall*** durante la fase di avviamento del sistema, con la coppia
  - l'indirizzo della funzione C chiamata ***system_call*** ( )
  - *PSR* opportuno per l'esecuzione di *system_call* ( )
- pertanto la funzione *system_call* ( ) costituisce il punto di entrata unico per tutti i servizi di sistema di Linux

Ritorno da Sistema Operativo

- c'è un'istruzione macchina, chiamata ***SYSRET***, privilegiata, che esegue queste operazioni:
  - carica in *PSR* il valore presente sulla pila
  - carica in *PC* il valore presente sulla pila
- in Linux l'istruzione *SYSRET* è eseguita alla fine della funzione C *system_call* ( )
- pertanto tale funzione costituisce l'unico punto di uscita dal sistema operativo e di ritorno al processo che ha invocato un servizio

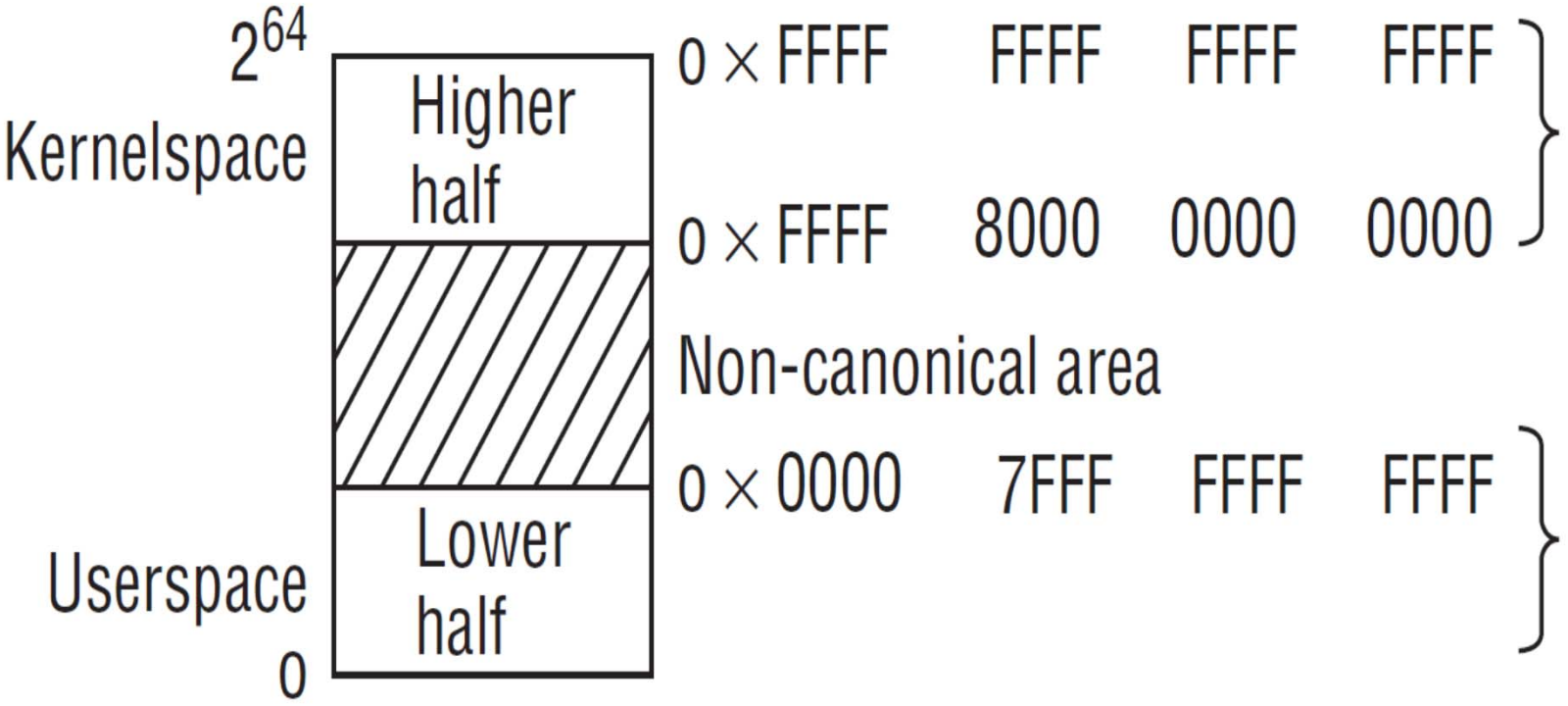
Modello di Memoria – Protezione del *SO*

- quando il processore è in modo U, gli deve essere impedito di accedere alle zone di memoria riservate al *SO*
- viceversa, quando il processore è in modo S, deve avere modo di accedere sia alla memoria del *SO* sia alla memoria dei processi

struttura dell'indirizzo di memoria

- lo spazio di indirizzamento potenziale di *x64* è di $\mathbf{2^{64}}$ byte, cioè $\mathbf{2^{24}}$ T byte (16 milioni di Tb)
- al momento l'architettura limita lo spazio virtuale utilizzabile a $\mathbf{2^{48}}$, cioè 256 Tb
- tale spazio è suddiviso nei due sottospazi di modo U ed S, ambedue da $\mathbf{2^{47}}$ 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
- gli indirizzi intermedi sono detti non-canonici e se utilizzati generano un errore
- la *CPU* è in modo S può utilizzare tutti gli indirizzi canonici
- in modo U la generazione di un indirizzo superiore a 0000 7FFF FFFF FFFF causa errore

Modello di Memoria in *x64*



il modello *x64* è simile al modello di memoria *MIPS*,
ma esteso enormemente

Cenni alla Paginazione

- la memoria di *x64* è gestita tramite paginazione, argomento trattato in dettaglio più avanti
- le seguenti caratteristiche della paginazione su *x64* sono sufficienti alla comprensione del nucleo:
  - la memoria è suddivisa in unità dette ***pagine*** (***pages***) di dimensione **4 K byte**, dunque con **12 bit** di *spiazzamento* (*offset*)
  - le pagine costituiscono unità di allocazione della memoria, per esempio della pila o dello *heap* (*memoria dinamica*)
  - ogni indirizzo prodotto dalla ***CPU***, chiamato *indirizzo virtuale*, viene trasformato in un *indirizzo fisico* prima di accedere alla memoria fisica – si chiamerà questa trasformazione ***mappatura virtuale / fisica***
  - la mappatura è descritta da una struttura dati chiamata ***tabella delle pagine***

Commutazione di Pila nel Cambio di Modo

- la pila utilizzata implicitamente dalla *CPU* nello svolgimento delle istruzioni – per 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 potere sostituire il valore di SP***
- in questo modo la *CPU* utilizza una pila diversa quando opera in modi diversi
- le pile di sistema e utente sono indicate con ***sPila*** e ***uPila***, quando è necessario
- le due pile sono allocate nei corrispondenti spazi virtuali di modo S e di modo U
- il *SO* Linux alloca a ogni processo una sPila costituita da due pagine, cioè 8 K byte

Esempio di Indirizzamento della Pila

- si considerino i valori prodotti dal modulo *axo_hello* con la funzione *task_explore*, riportati in tabella
- le ultime tre cifre indicano lo *spiazzamento* (*offset*), quelle precedenti il *numero di pagina*
- la pila di sistema va da 0x FFFF 8800 5C64 4000 a 0x FFFF 8800 5C64 6000
- la pila di utente è nello spazio U (la sua cima è 0x 0000 7FFF 6DA9 8C78)

variabile	indirizzo	significato
thread.sp0	0x FFFF 8800 5C64 6000	base della sPila
ts->stack	0x FFFF 8800 5C64 4000	limite della sPila
thread.sp	0x FFFF 8800 5C64 5D68	*SP* della sPila
usersp	0x 0000 7FFF 6DA9 8C78	*SP* della uPila

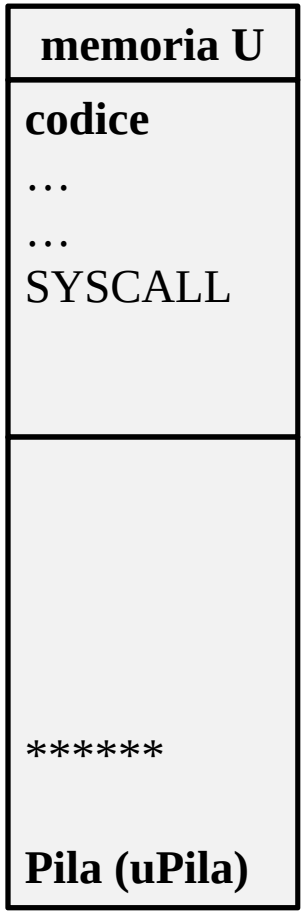
Commutazione di Pila – I

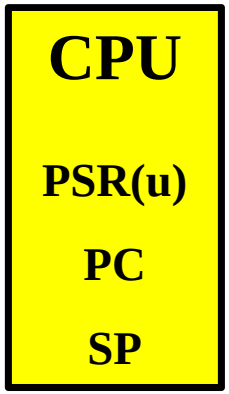
- nella commutazione da modo U a modo S, la commutazione di pila avviene *prima* del salvataggio di informazioni sulla stessa
  - l'indirizzo di ritorno a modo U deve essere salvato su sPila
  - nel ritorno da modo S a modo U, l'informazione per il ritorno verrà prelevata da sPila, cioè prima di commutare a uPila
- sono necessarie opportune strutture dati – qui si usa un modello semplificato rispetto a quello di *x64* – basato su due celle apposite chiamate ***USP*** e ***SSP***:
  - la cella *SSP* contiene il valore da caricare nel registro *SP* al momento del passaggio a modo S
  - è compito del sistema operativo garantire che il registro *SP* contenga sempre il valore corretto, cioè quello relativo alla sPila del processo in esecuzione
  - invece, nella cella *USP* viene salvato il valore del registro *SP* al momento del passaggio a modo S, dunque lo *SP* relativo alla uPila
- le celle *USP* e *SSP* sono contenute nel *TSS* (*Task State Segment*), una struttura dati di memoria mantenuta dalla *CPU* mediante un meccanismo hardware di aggiornamento (piuttosto complicato per via dei numerosi modi di compatibilità di *x64* – qui non interessa)

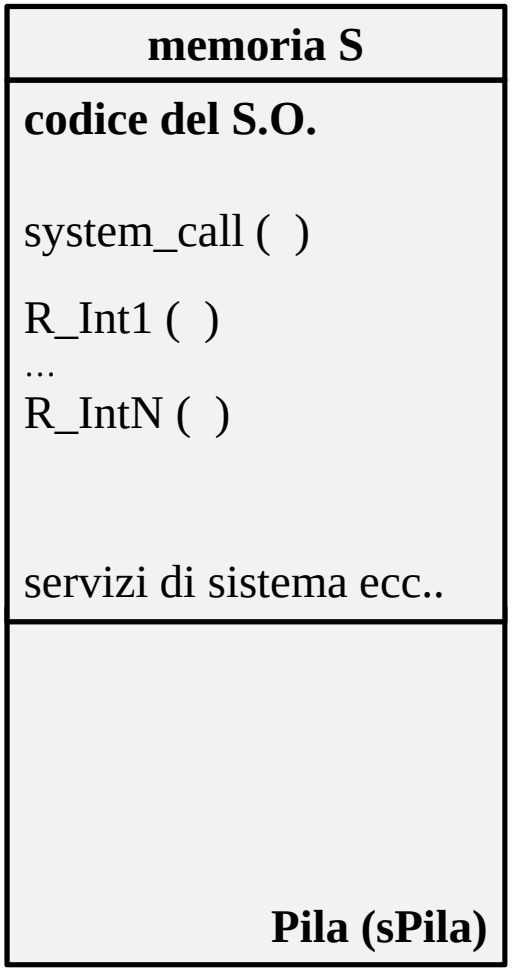
Commutazione di Pila – II

- complessivamente le operazioni svolte dall'istruzione macchina *SYSCALL* sono:
  - salva in *USP* il valore corrente di *SP*
  - copia in *SP* il valore presente in *SSP* (ora *SP* punta in sPila)
  - salva su sPila il valore del *PC* di ritorno al programma chiamante
  - salva su sPila il valore del *PSR* del programma chiamante
  - carica in *PC* e in *PSR* i valori presenti nel vettore di syscall
  - pertanto adesso il modo di funzionamento passa a S
- simmetricamente, le operazioni svolte dall'istruzione macchina *SYSRET* sono:
  - ripristina in *PSR* il valore presente su sPila
  - ripristina in *PC* il valore presente su sPila
  - copia in *SP* il valore presente in *USP*
  - pertanto adesso *SP* punta nuovamente a uPila

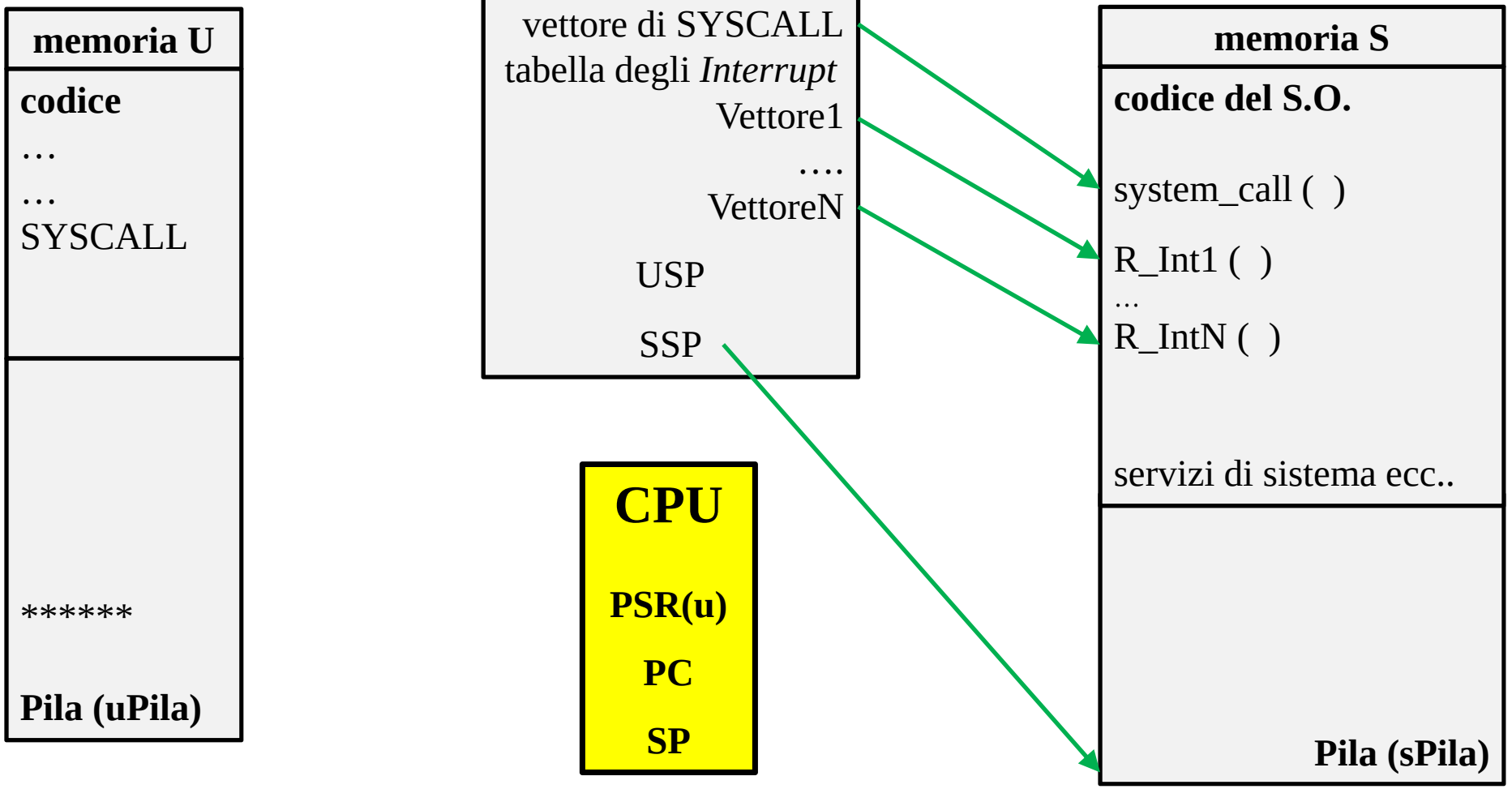
ESEMPIO 1 (componenti *HW*)



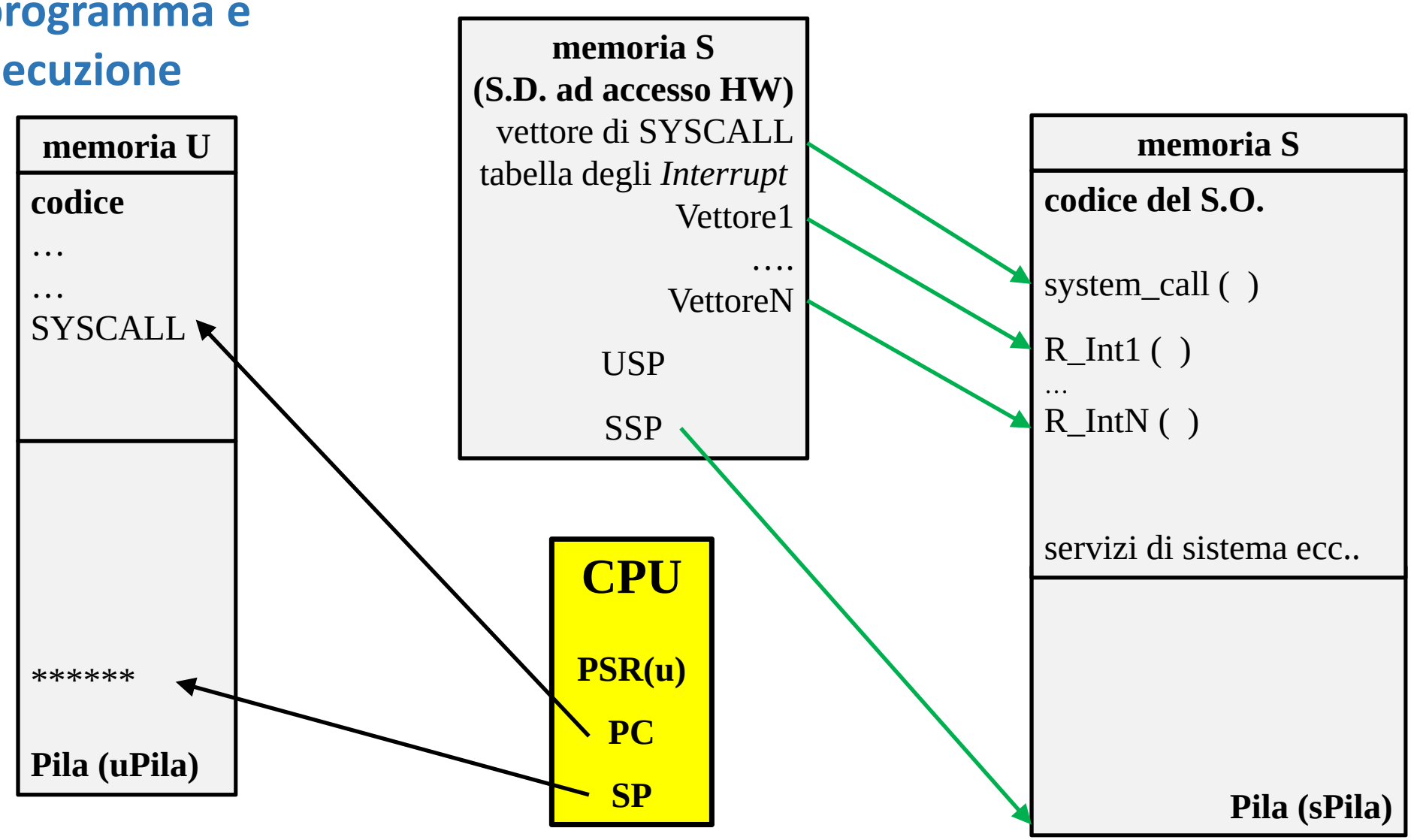




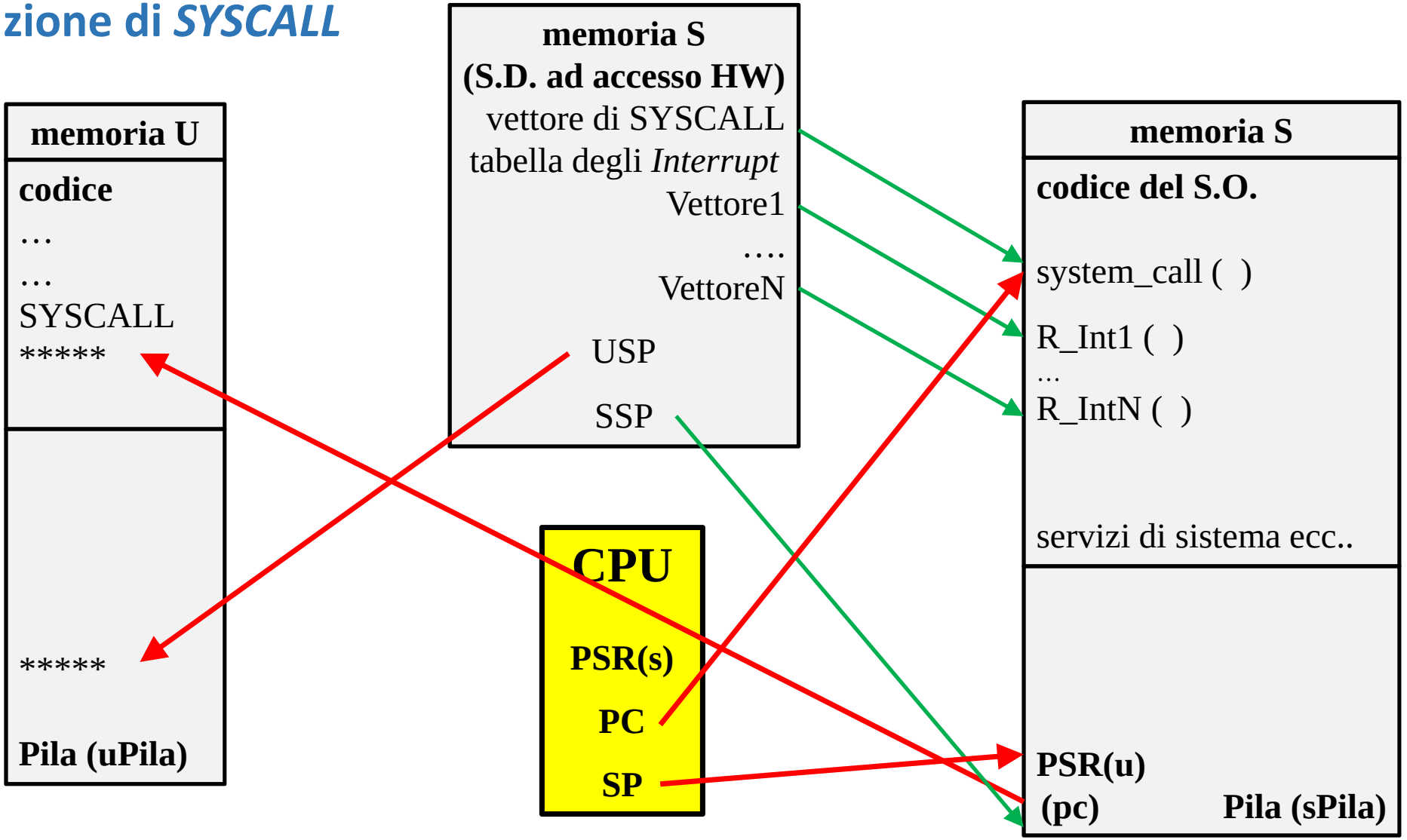
Dopo l'inizializzazione del Sistema operativo



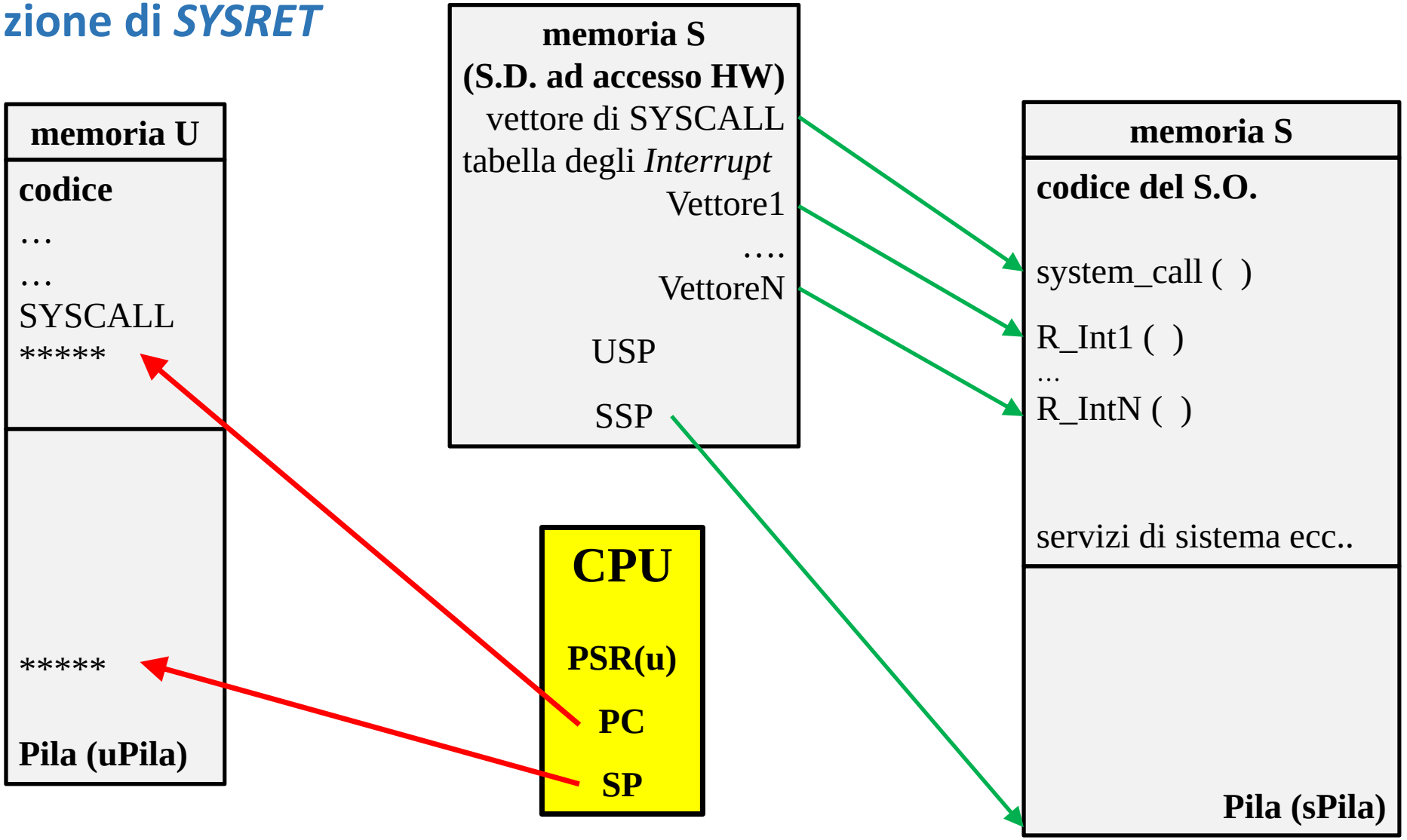
Un programma è in esecuzione



Esecuzione di *SYSCALL*



Esecuzione di *SYSRET*



Commutazione della Mappatura Virtuale/Fisica della Memoria

- si ricorda che in *x64* la memoria è divisa in pagine di 4 K byte ciascuna
- il *SO* Linux associa a ciascun processo una diversa tabella delle pagine
- in questo modo gli indirizzi virtuali di ciascun processo sono mappati su aree (pagine) indipendenti della memoria fisica
- in *x64* c'è un registro, chiamato ***CR3*** (***CR*** sta per ***Control Register***), che contiene l'indirizzo del punto di partenza della tabella delle pagine utilizzata per la mappatura degli indirizzi di memoria
- pertanto, per cambiare la mappatura è sufficiente cambiare il contenuto di *CR3*, facendolo puntare a una diversa tabella delle pagine

Meccanismo di Interruzione (Interrupt) – I

- esiste un insieme di ***eventi*** rilevati dallo hardware, per esempio un particolare segnale proveniente da una periferica, una condizione di errore, ecc
- a ciascun evento è associata una particolare funzione detta ***gestore dell'interrupt*** o ***routine di interrupt***
- tutte le routine di interrupt fanno parte del *SO*
- quando il processore rileva un evento, esso interrompe il programma correntemente in esecuzione ed effettua un salto all'esecuzione della funzione associata a tale evento
- quando la funzione termina, il processore riprende l'esecuzione del programma che è stato interrotto

Meccanismo di Interruzione (Interrupt) – II

- per riprendere l'esecuzione il processore salva sulla pila, al momento del salto alla routine di interrupt, l'indirizzo della prossima istruzione del programma interrotto
- dopo l'esecuzione della routine di interrupt tale indirizzo è disponibile per eseguire il ritorno al programma interrotto
- l'istruzione macchina che esegue il ritorno da interrupt è chiamata ***IRET***
- il meccanismo di interrupt è a tutti gli effetti simile a un'invocazione di funzione o all'esecuzione dell'istruzione macchina *SYSCALL*
- dunque le routine di interrupt sono completamente asincrone rispetto al programma interrotto, come le funzioni dei thread
- pertanto è necessario trattare le routine di interrupt con tutti gli accorgimenti della programmazione concorrente

Meccanismo di Interruzione (Interrupt) – III

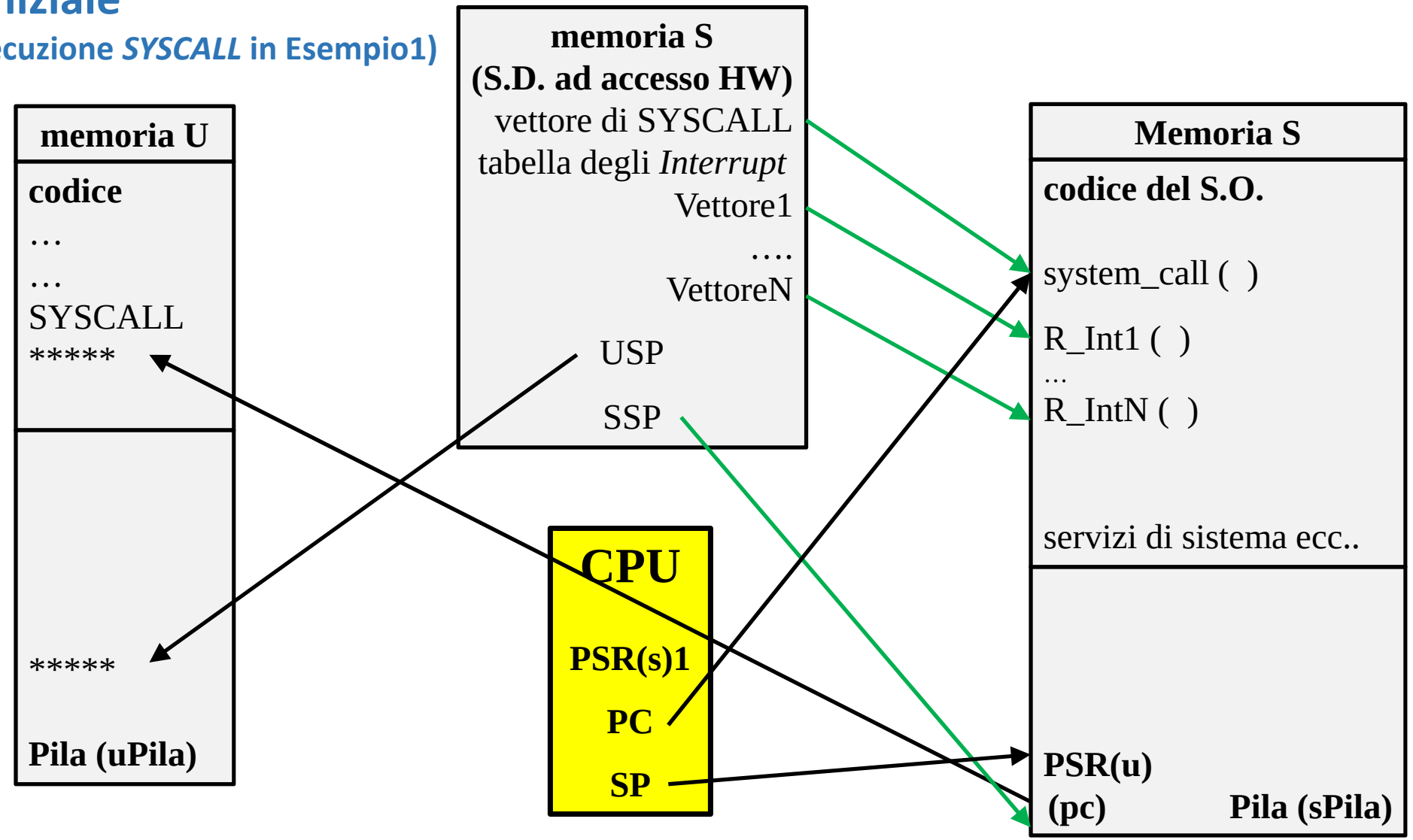
- il meccanismo di interrupt si combina con il doppio modo di funzionamento S e U in maniera simile a quello dell'istruzione macchina *SYSCALL*
- dal punto di vista hardware non c'è differenza sostanziale tra un interrupt e un'istruzione macchina *SYSCALL*
- in ambedue i casi è necessario passare a modo 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 di prima che l'interrupt si verificasse, prelevando il *PSR* dalla sPila

Meccanismo di Interruzione (Interrupt) – IV

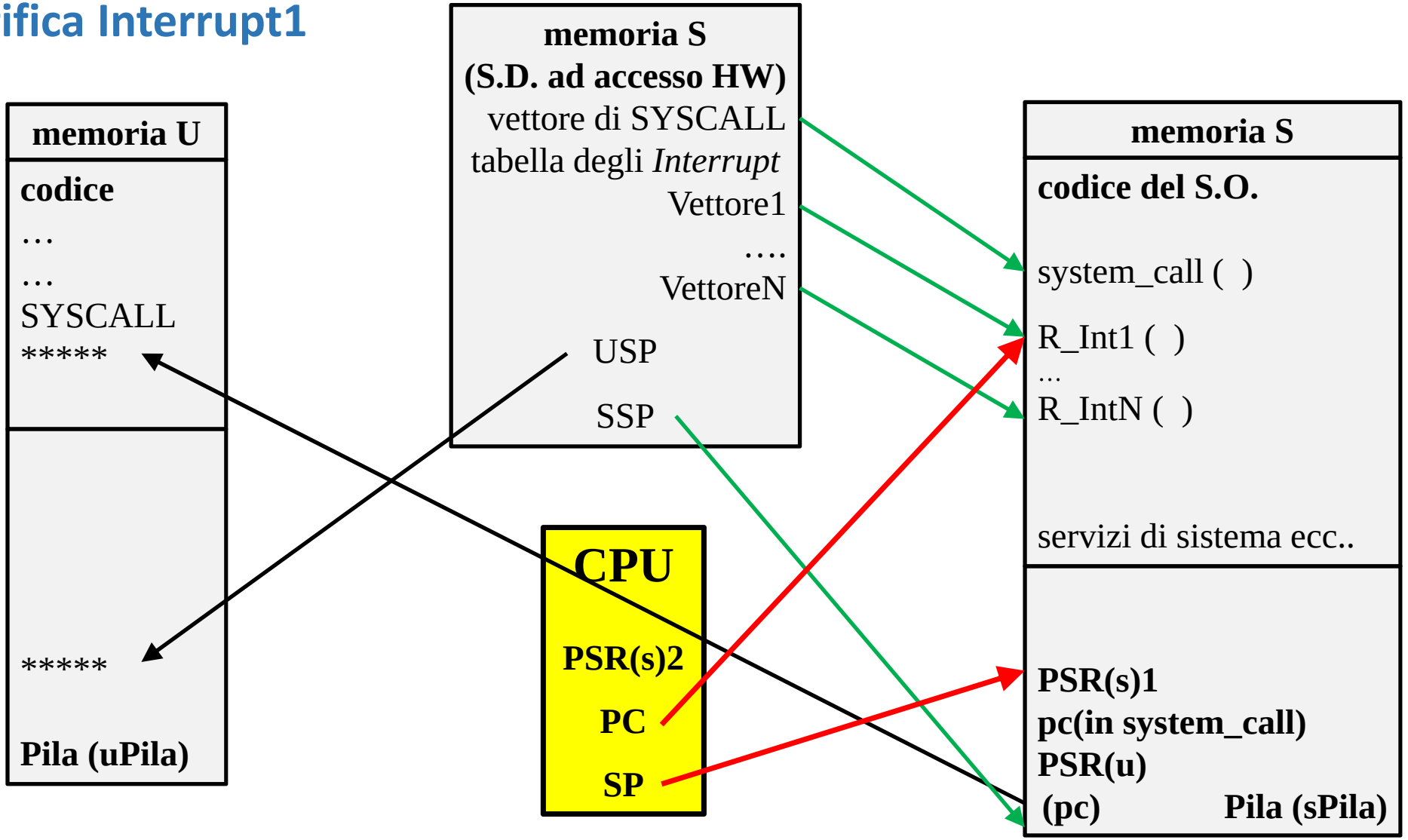
- il processore deve sapere qual è l'indirizzo della routine di interrupt da eseguire quando si verifica un certo evento, e anche qual è il valore di *PSR* da utilizzare
- la ***tabella degli interrupt***, una struttura dati ad accesso *HW*, contiene un certo numero di ***vettori di interrupt*** costituiti, come il vettore di *syscall*, da una coppia $\langle PC, PSR \rangle$
- c'è 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 funzione

Stato Iniziale

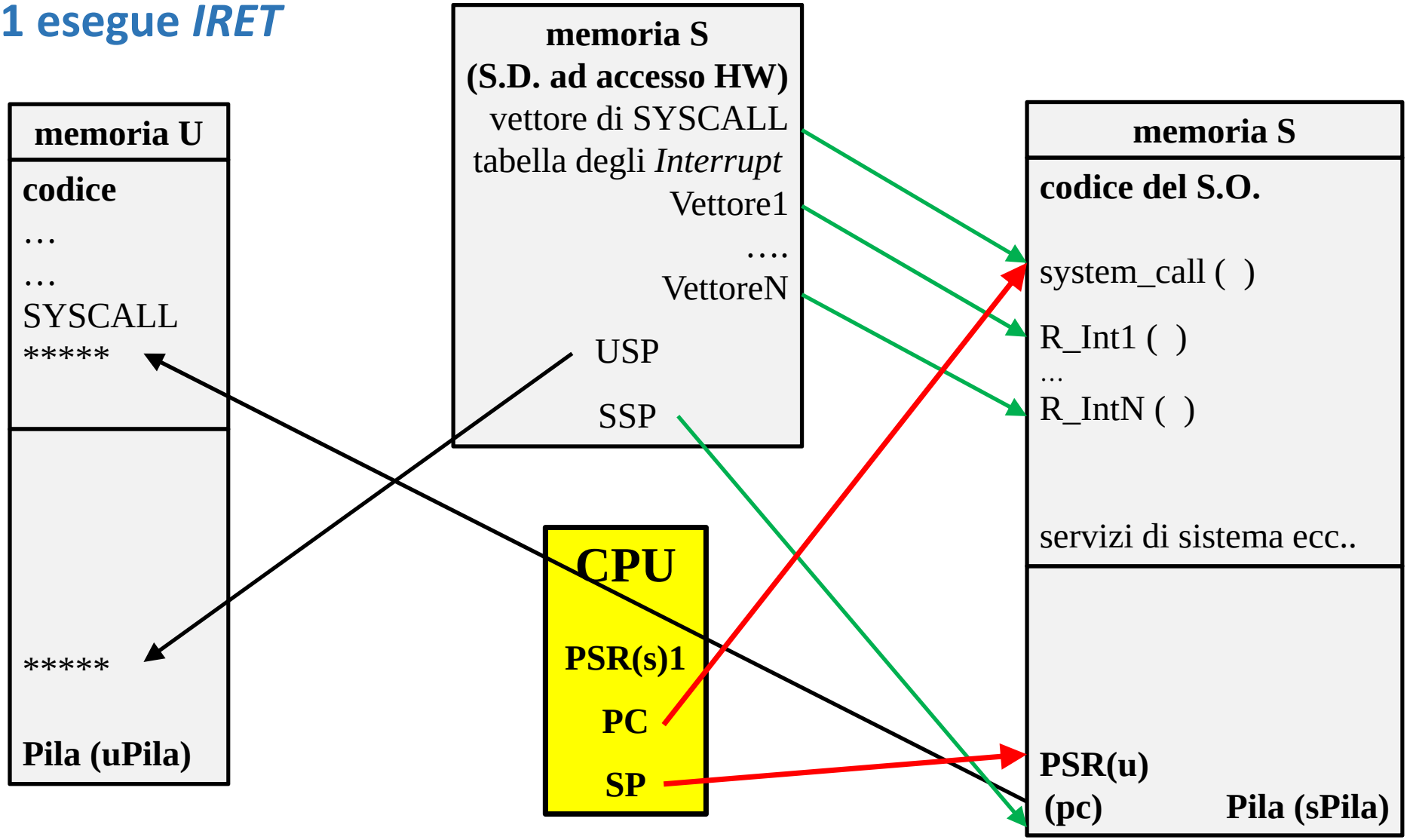
(dopo esecuzione *SYSCALL* in Esempio1)



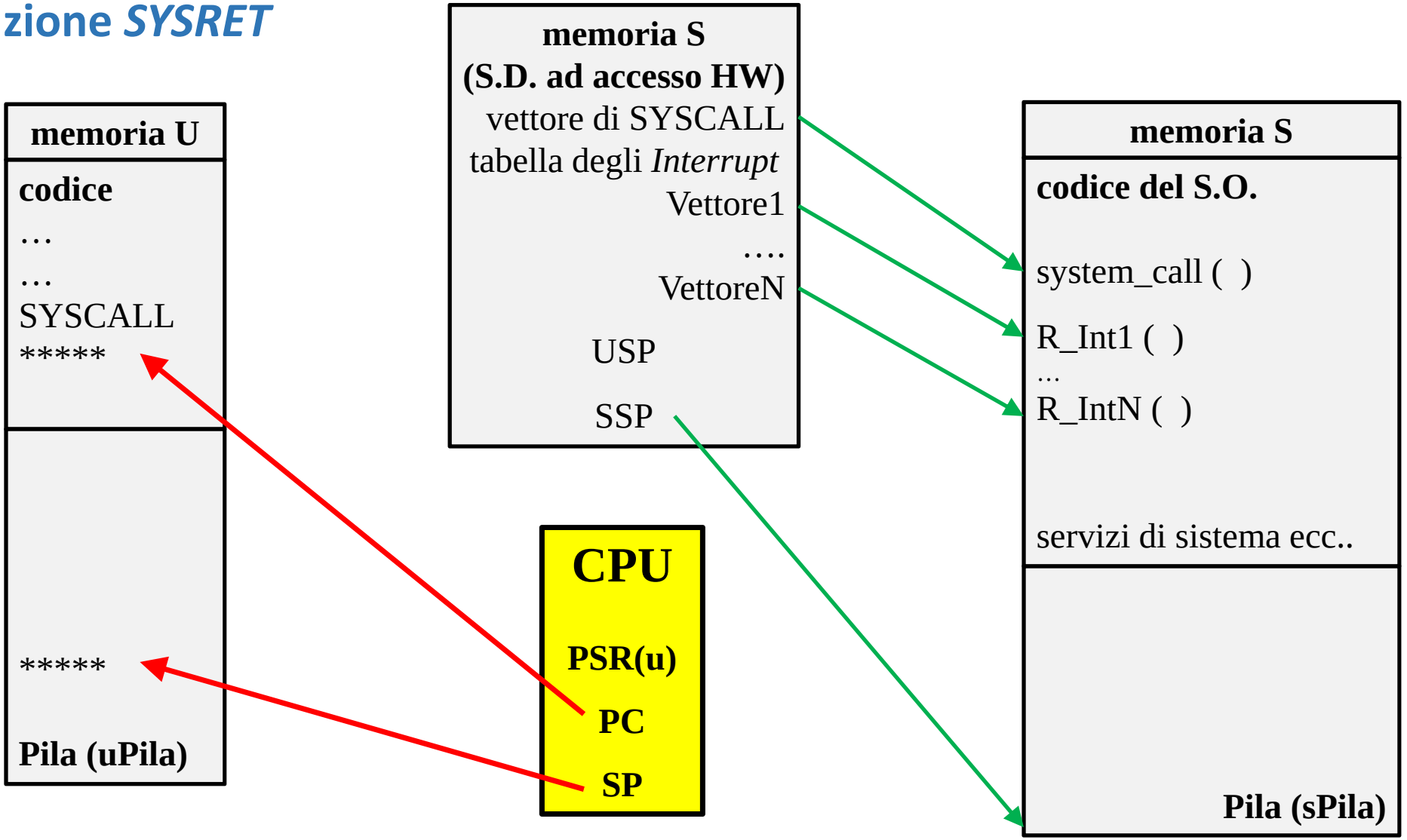
Si verifica Interrupt1



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, come per esempio:
  - divisione per zero
  - uso di indirizzi di memoria non validi
  - tentativo di eseguire istruzioni vietate
- la maggior parte dei processori prevede di trattare l'errore come se fosse un particolare tipo di interrupt (interrupt di tipo *eccezione*)
- in questo modo, quando si verifica un errore che impedisce al processore di procedere normalmente con l'esecuzione delle istruzioni, viene attivata, tramite un opportuno vettore di interrupt, una routine del *SO* che decide come gestire l'errore stesso
- spesso la gestione dell'errore consiste nella terminazione forzata (*abort*) del programma che ha causato l'errore, eliminando il processo

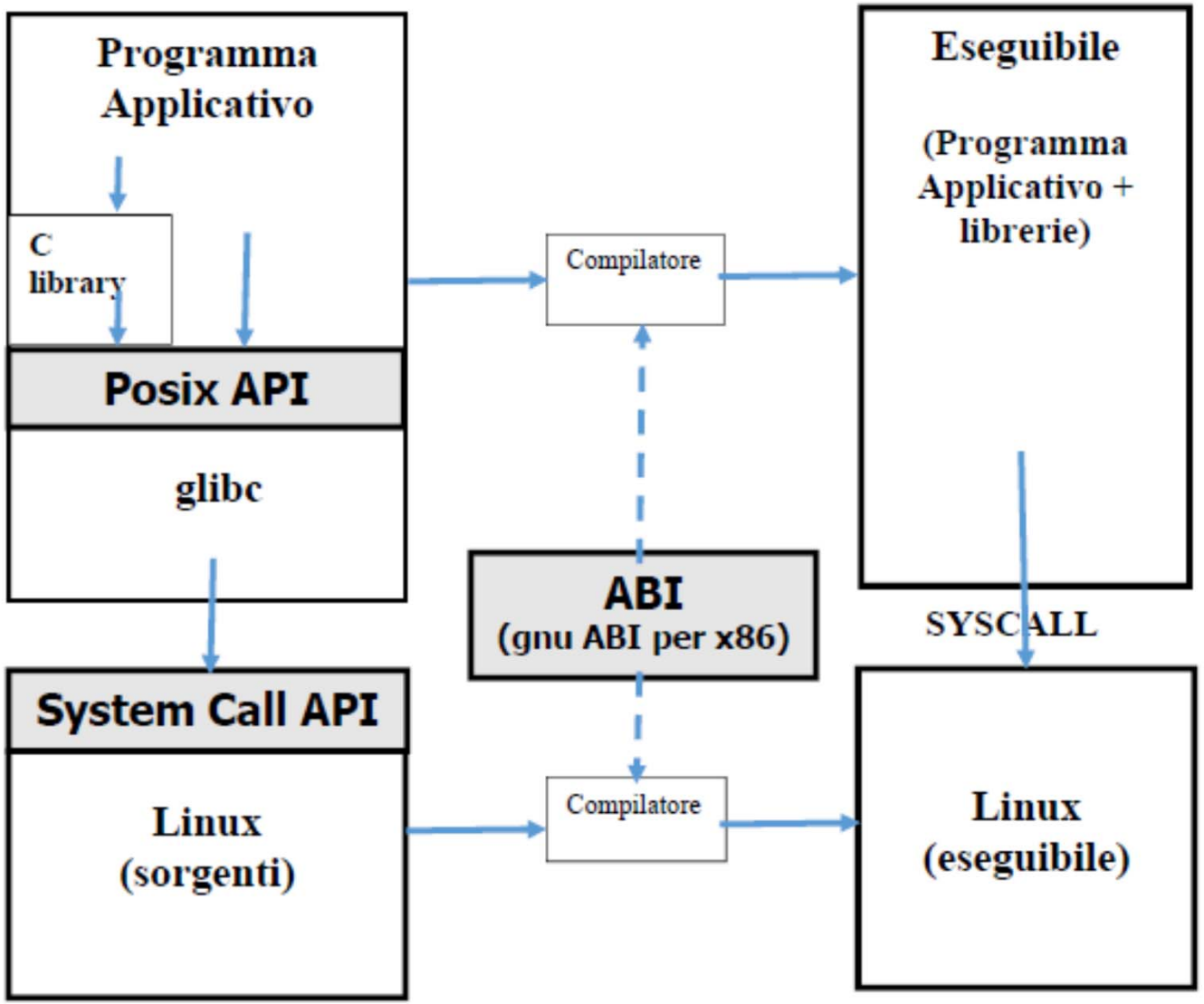
Priorità e Abilitazione degli Interrupt

- talvolta è sbagliato o inopportuno permettere a un interrupt di interrompere la routine che al momento sta servendo un altro interrupt
- ha senso che un evento molto importante e che richiede una risposta urgente possa interrompere la routine di interrupt che sta servendo un evento meno importante, ma non il contrario
- ecco il meccanismo di priorità dell'interrupt, per realizzare tale comportamento:
  - il processore possiede un ***livello di priorità*** che è scritto nel registro *PSR*
  - il livello di priorità del processore può essere modificato dal software tramite opportune istruzioni macchina (naturalmente privilegiate) che scrivono nel *PSR*
  - anche a ciascun evento di interrupt è associato un certo livello di priorità
  - un interrupt viene accettato e servito 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 a quando il livello di priorità del processore sarà diminuito sufficientemente
- con questo meccanismo hardware, il *SO* può aumentare o diminuire il livello di priorità del processore, in modo che durante l'esecuzione delle routine di interrupt più importanti gli eventi di interrupt meno importanti non vengano accettati e serviti
- se il processore ha livello di priorità massimo, nessun interrupt viene accettato e servito, cioè il meccanismo di interrupt è *disabilitato* (ciò può servire in sequenze molto critiche)

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 normale	U S	U S	istruzione di ritorno - *RET*	U S
SYSCALL	U	S	*SYSRET*	U
interrupt	U S	S S	*IRET*	U S

Interfacce Standard e Application Binary Interface (*ABI*)



ABI – Regole di Invocazione del *SO*

- ecco come si passano i parametri alla funzione C ***system_call* ( )**:
  - il numero del servizio da invocare va messo nel registro **rax**
  - eventuali parametri accessori (che dipendono dal servizio richiesto) vanno messi ordinatamente nei registri **rdi, rsi, rdx, r10, r8** e **r9**
- di solito un programma applicativo non invoca la funzione C *system_call* ( ) direttamente
- esso invoca invece una funzione della libreria ***glibc***, la quale funzione a sua volta effettua la chiamata di sistema
- nella libreria *glibc* sono presenti funzioni C che corrispondono ai servizi offerti dal *SO*, per esempio *fork* ( ), *open* ( ), ecc
- queste funzioni dei servizi invocano una funzione C della libreria *glibc*, la quale incapsula l'istruzione macchina *SYSCALL*; quest'ultima funzione C è dichiarata nel modo seguente

```
long syscall (long numero_servizio, ... parametri del servizio ...)
```

- i numeri dei servizi sono codificati nella tabella seguente (ad oggi ci sono 322 servizi !)

%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* (SYS_read, fd, buf, len):
   pone SYS_read nel registro rax
   pone fd, buf, e len nei registri rdi, rsi e rdx
   esegue istruzione macchina *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 macchina *SYSRET* per tornare al processo che ha richiesto il servizio