POLITECNICO
MILANO 1863

Architettura dei calcolatori e sistemi operativi

Il Nucleo del Sistema Operativo

N3 – Gestione dello stato dei Processi

04.01.2016

Stato dei processi

un processo può trovarsi in uno dei due stati fondamentali seguenti:

- **ATTESA**: un processo in questo stato non può essere messo in esecuzione, perché deve attendere il verificarsi di un certo ***evento***. Ad esempio, un processo che ha invocato una "*scanf( )*" ed attende che venga inserito un dato dal terminale
- **PRONTO**: un processo pronto è un processo che può essere messo in esecuzione se lo *scheduler* lo seleziona

Tra i processi in stato di pronto ne esiste uno che è effettivamente in esecuzione, chiamato **processo corrente**

Lo stato di un processo è registrato nel suo descrittore

Contesto di un processo

Normalmente un processo è in esecuzione in modo U

Se il processo corrente richiede un **servizio di sistema** (tramite l'istruzione SYSCALL) viene attivata una funzione del SO che esegue il servizio ***per conto di tale processo***

- ad esempio, se il processo richiede una lettura da terminale, il servizio di lettura legge un dato dal terminale ***associato al processo in esecuzione***
- i servizi sono quindi in una certa misura parametrici rispetto al processo che li richiede
- faremo riferimento a questo fatto dicendo che un servizio è svolto ***nel contesto*** di un certo processo

si usa dire che ***un processo è in esecuzione in modo S*** quando il SO è in esecuzione nel contesto di tale processo, sia per eseguire un servizio, sia per servire un interrupt

Abbandono dell'esecuzione di un processo

Il processo in stato di esecuzione abbandona tale stato solamente a causa di uno dei due eventi seguenti

1. quando un servizio di sistema richiesto dal processo deve porsi in attesa di un evento
   - Il processo passa in **stato di attesa** di un evento
     - ad esempio, il processo P ha richiesto il servizio di lettura (*read*) di un dato dal terminale ma il dato non è ancora disponibile
     - il servizio si pone in attesa dell'evento "arrivo del dato dal terminale del processo P"
   - *un processo si pone in stato di attesa quando è in esecuzione un servizio di sistema per suo conto*, non quando è in esecuzione normale in modo U

2. quando il SO decide di sospenderne l'esecuzione a favore di un altro processo (preemption); in questo caso il processo passa dallo stato di esecuzione allo **stato di pronto**

Scheduler

E' il componente del Sistema Operativo che decide quale processo mettere in esecuzione

Lo scheduler svolge 2 tipi di funzioni:

- determina quale processo deve essere messo in esecuzione, quando e per quanto tempo, cioè realizza la **politica di scheduling** del sistema operativo
- esegue l'effettiva **Commutazione di Contesto** (**Context Switch**), cioè la sostituzione del processo corrente con un altro processo in stato di PRONTO

La politica di scheduling verrà affrontata più avanti

Il Context Switch è svolto dalla funzione **schedule( )** dello scheduler

Lo Scheduler gestisce una struttura dati fondamentale (per ogni CPU), la **runqueue** che contiene due campi

- **RB**: è una lista di puntatori ai descrittori di tutti i processi pronti (escluso quello in esecuzione)
- **CURR**: è un puntatore al descrittore del processo corrente

La sPila dei processi

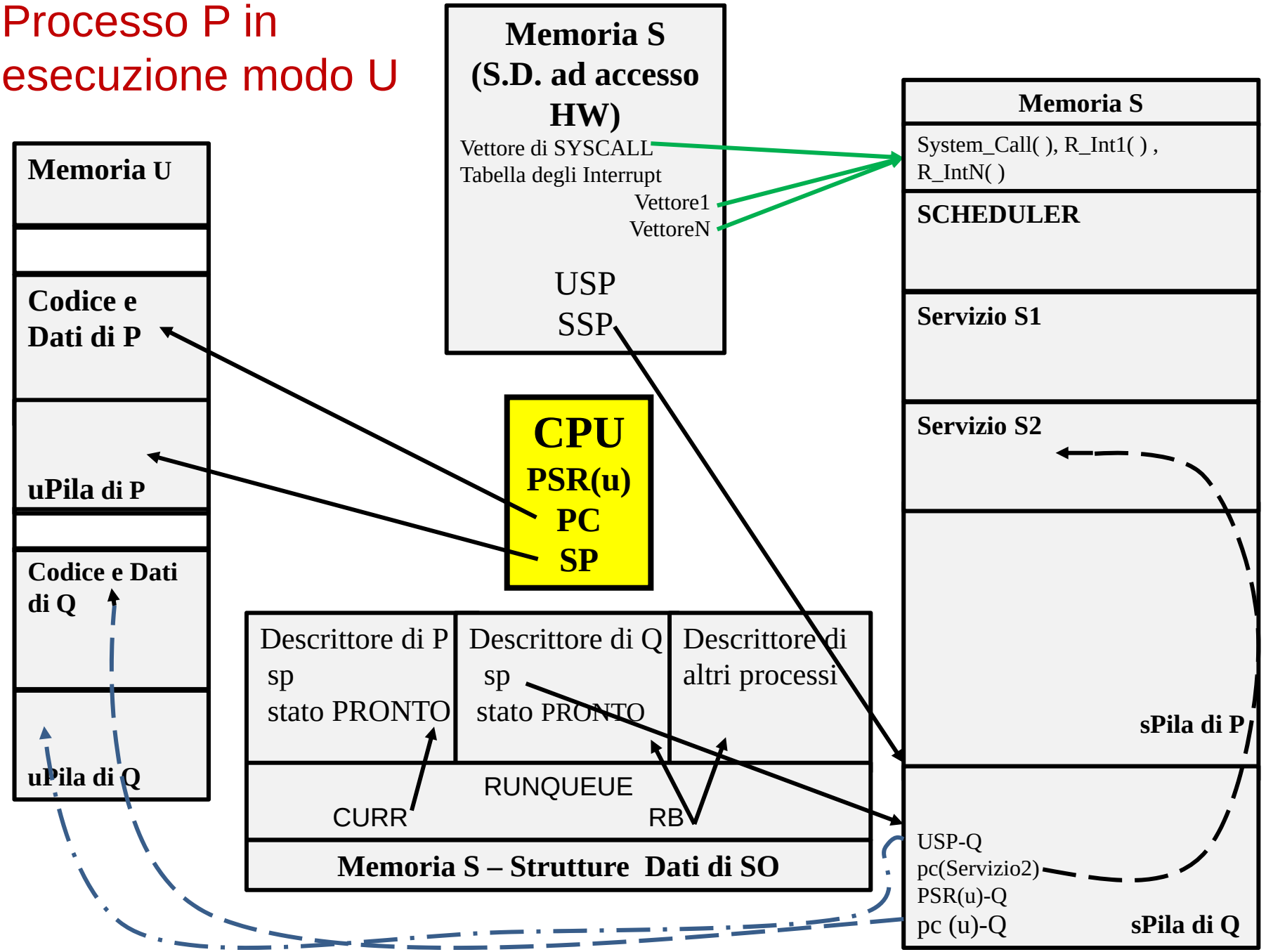
- Linux assegna ad ogni processo una sPila di 8Kb
- Durante l'esecuzione di un servizio di sistema per conto di un processo la sua sPila contiene una parte del contesto Hardware del processo (per ripristinare lo stato al ritorno in modo U)
- il meccanismo HW permette la commutazione corretta da uPila a sPila e viceversa, a condizione che SSP e USP contengano i valori corretti da assegnare al registro SP
- dato che il SO mantiene ***una diversa sPila per ogni processo***, la gestione di questo meccanismo diventa più complessa:
  - richiede di ***salvare i valori di SSP e USP*** durante la sospensione tra una esecuzione di un processo e la successiva
  - a questo scopo il Descrittore di un Processo P contiene i campi
    - ***sp0***: contiene l'indirizzo di base della sPila di P
    - ***sp***: contiene il valore dello SP salvato al momento in cui il processo ha sospeso l'esecuzione

Gestione di SSP e USP nel Context Switch

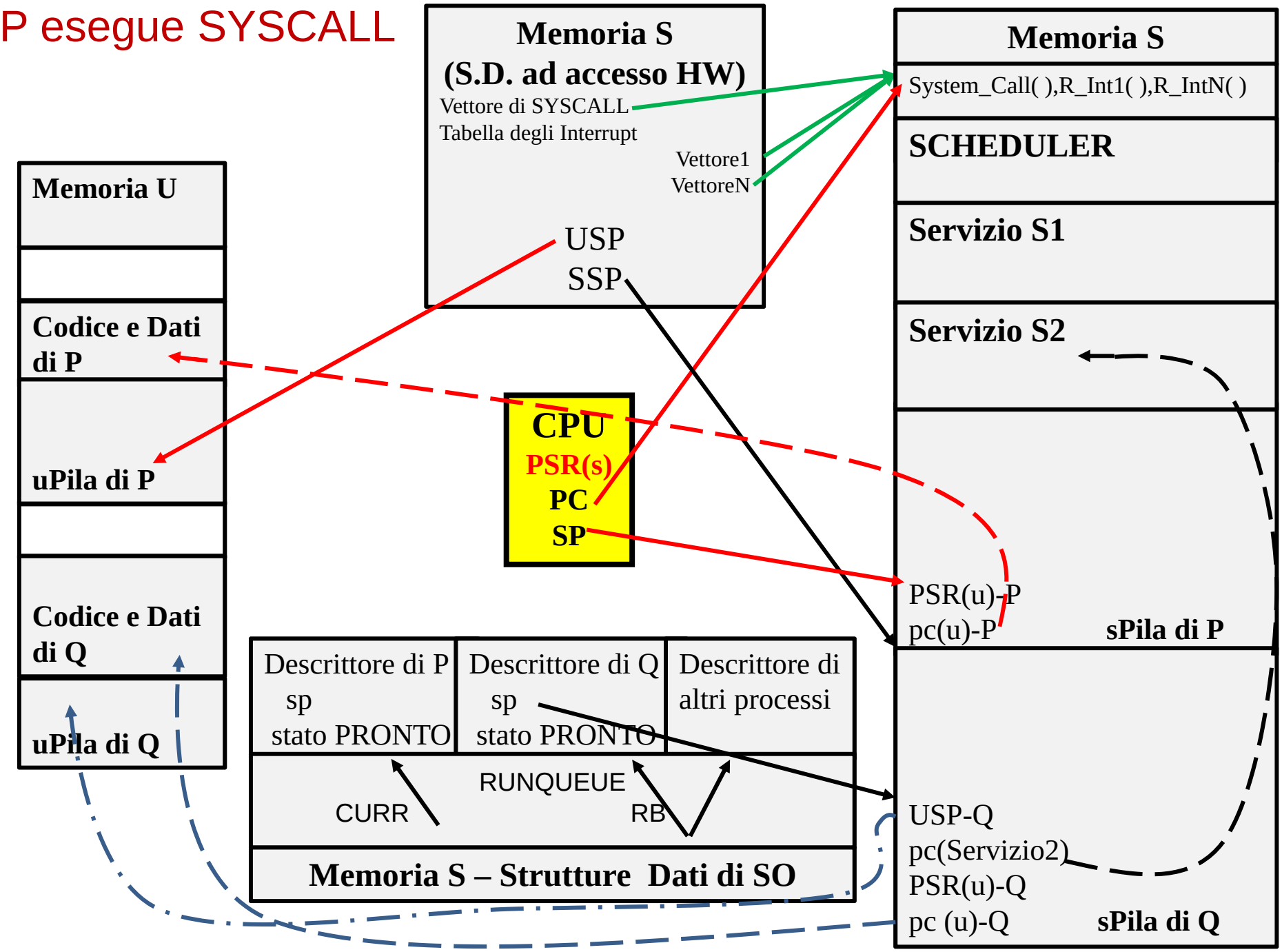
- quando il processo P è in esecuzione in modo U, la sPila è vuota; in SSP viene messo il valore di base preso da *sp0* del descrittore di P
- quando la CPU passa al modo S (SYSCALL o Interrupt) in USP viene scritto (caricato) automaticamente dall'HW il valore corretto - il valore di SP corrente - per il ritorno al modo U
- se, durante l'esecuzione in modo S, viene eseguita una *commutazione di contesto* (va in esecuzione il processo Q), Linux opera nel modo seguente
  - salva USP sulla sPila di P
  - poi salva il valore del registro SP nel campo *sp* del descrittore di P
- quando P riprenderà l'esecuzione
  - SP verrà ricaricato dal campo *sp* del descrittore, puntando alla cima della sPila
  - USP verrà ricaricato prendendolo dalla sPila
  - SSP verrà ricaricato prendendolo dal campo *sp0* del descrittore

Vediamo un esempio: inizialmente è in esecuzione un processo P, mentre un altro processo Q è in stato di PRONTO

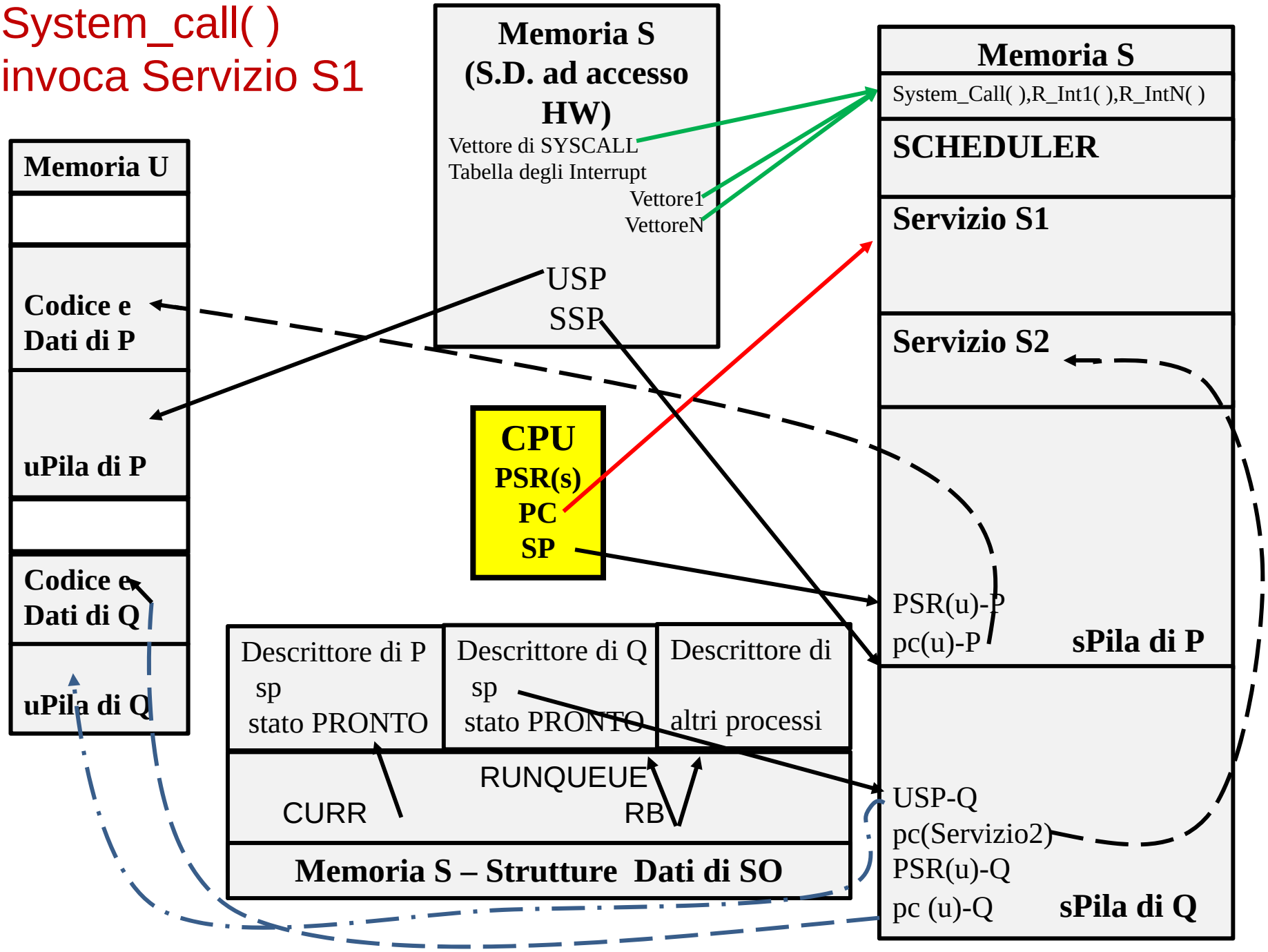
Processo P in esecuzione modo U



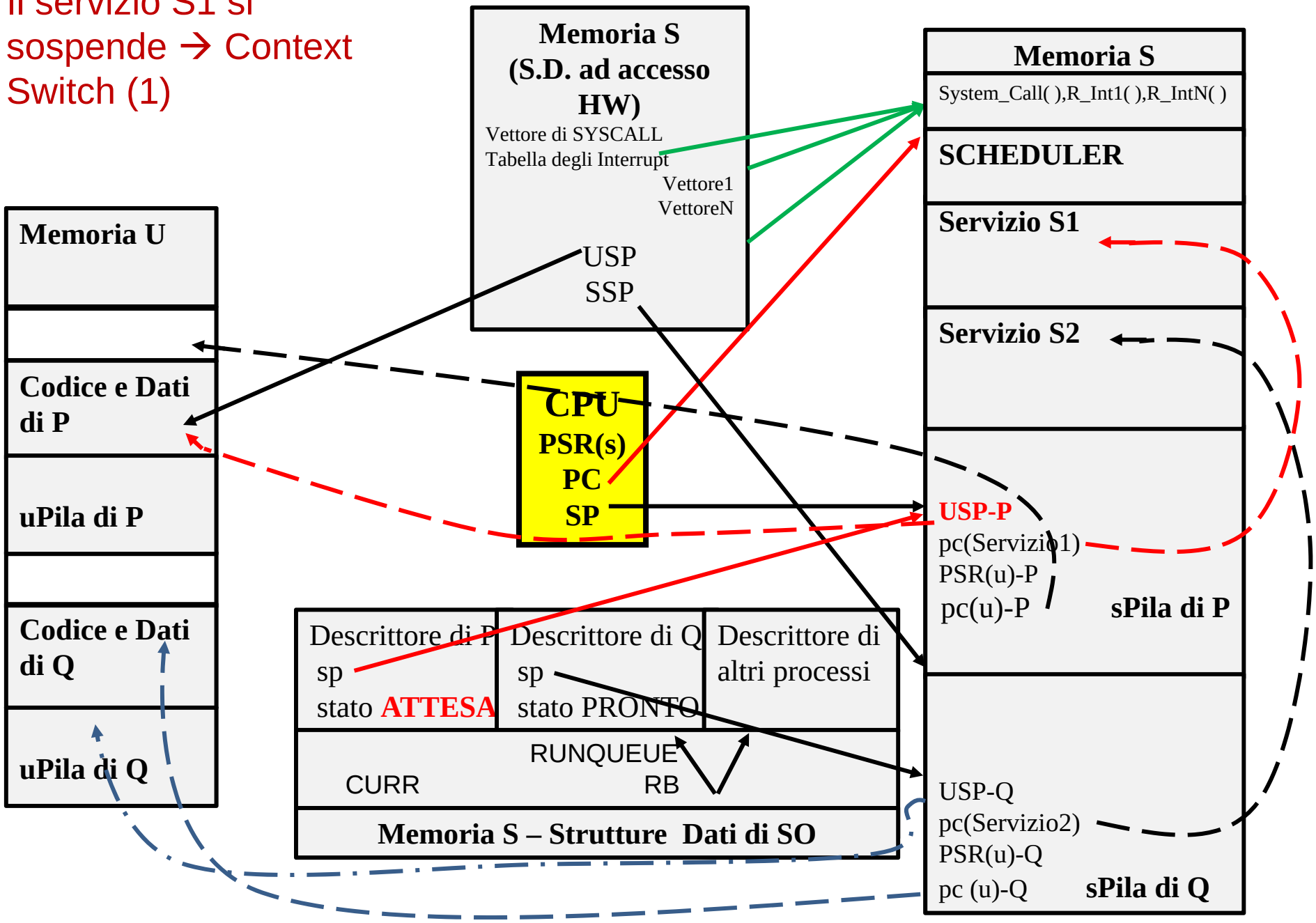
P esegue SYSCALL



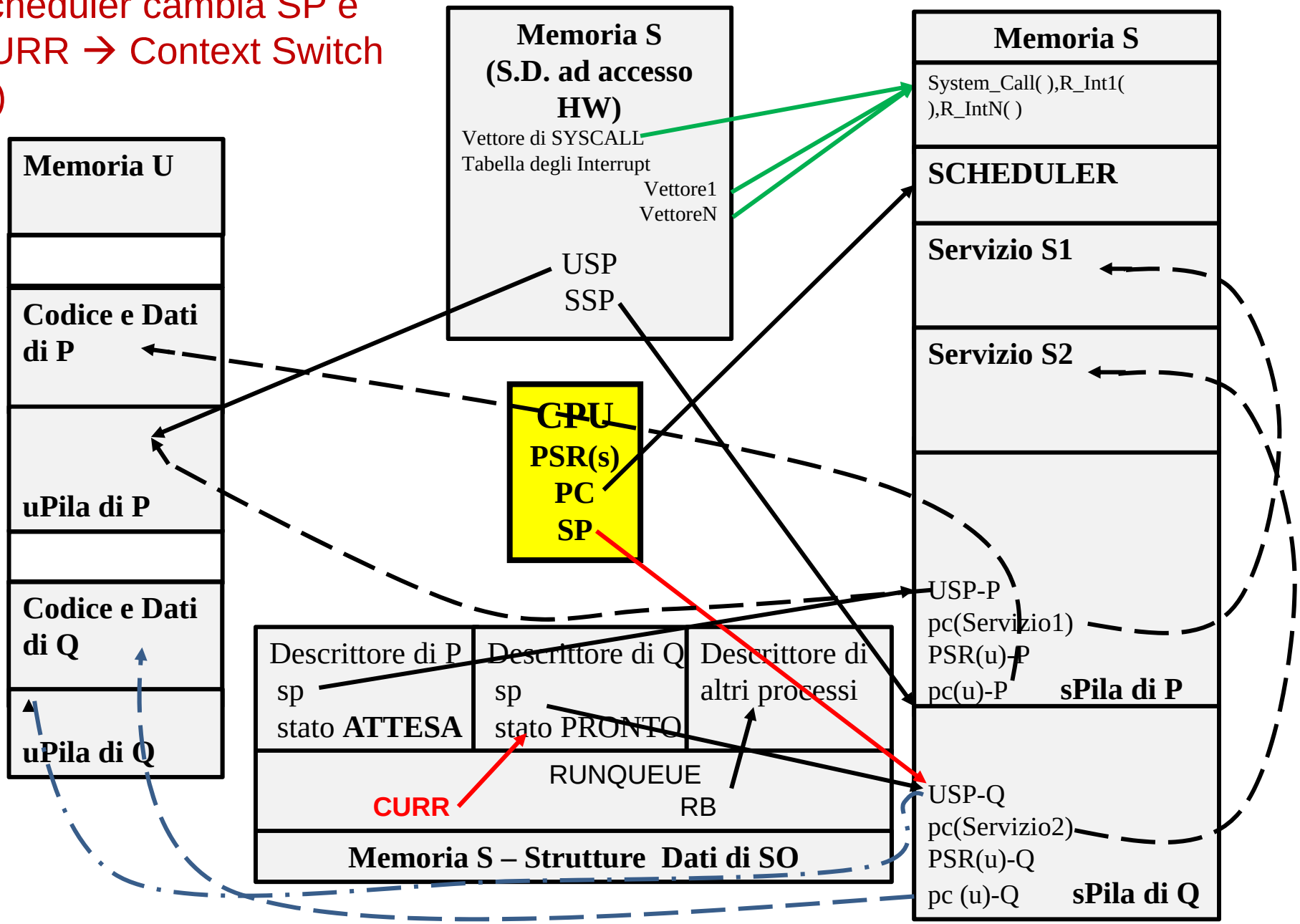
System_call( ) invoca Servizio S1



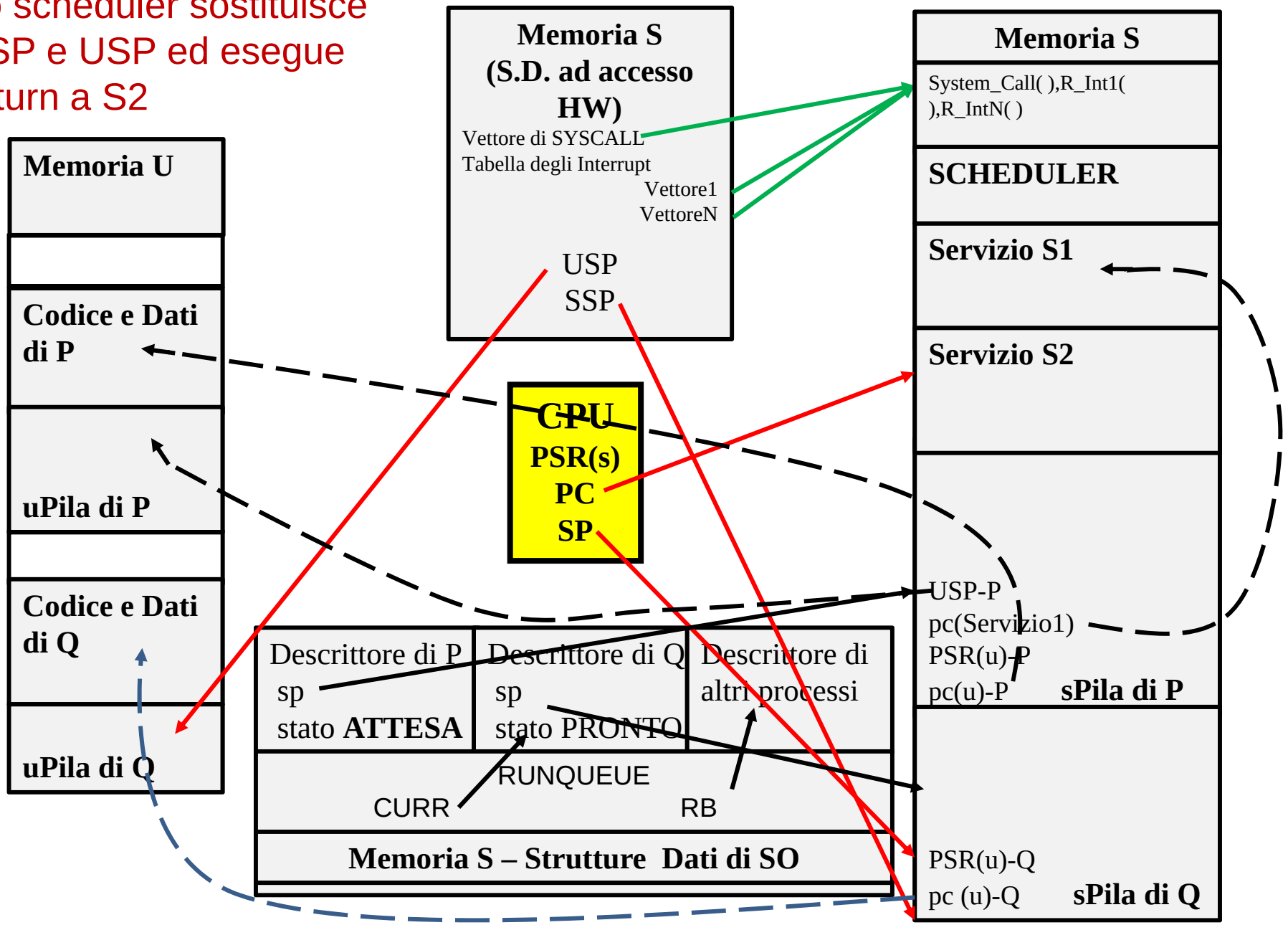
Il servizio S1 si sospende → Context Switch (1)



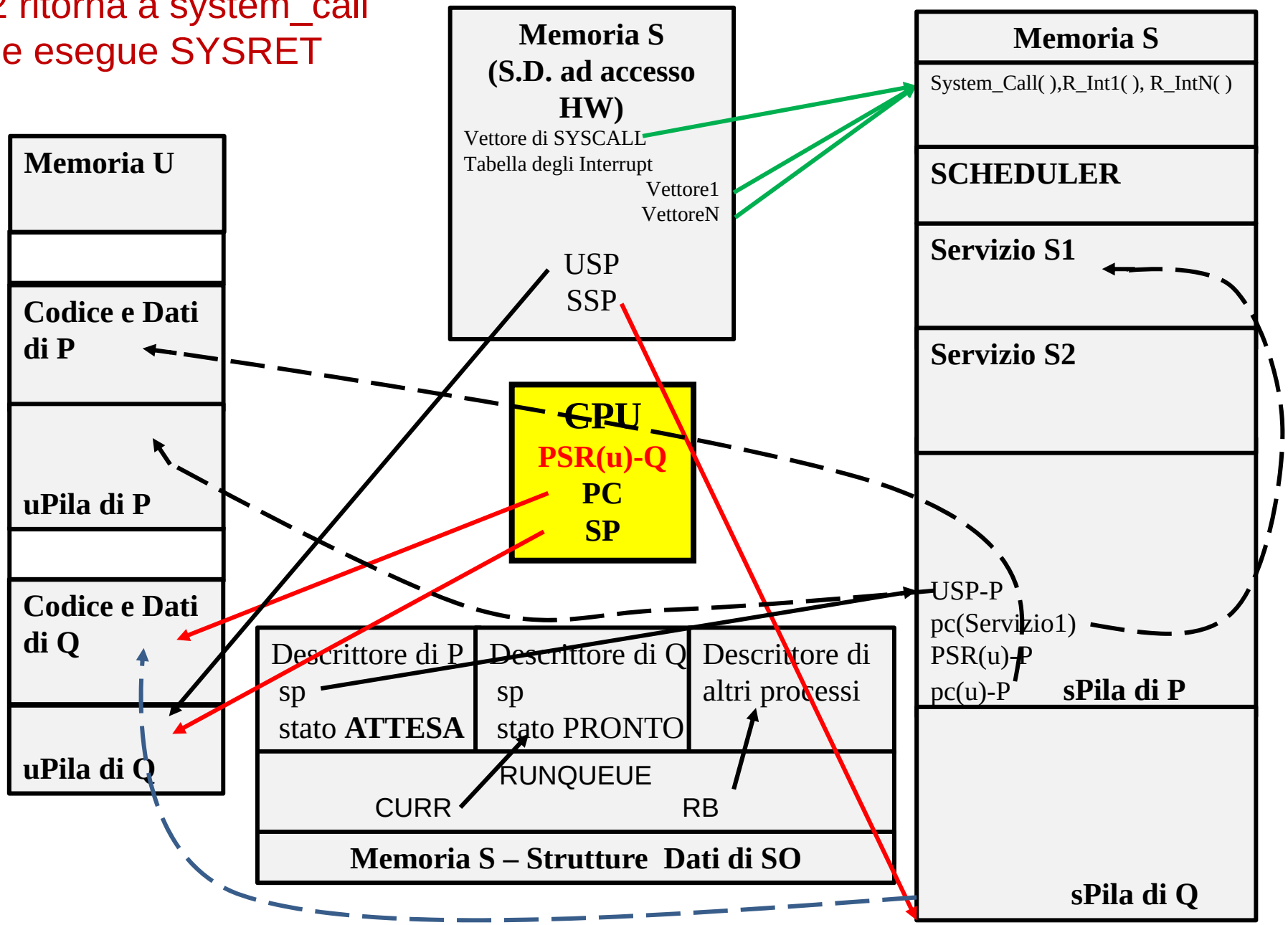
Scheduler cambia SP e CURR → Context Switch (2)



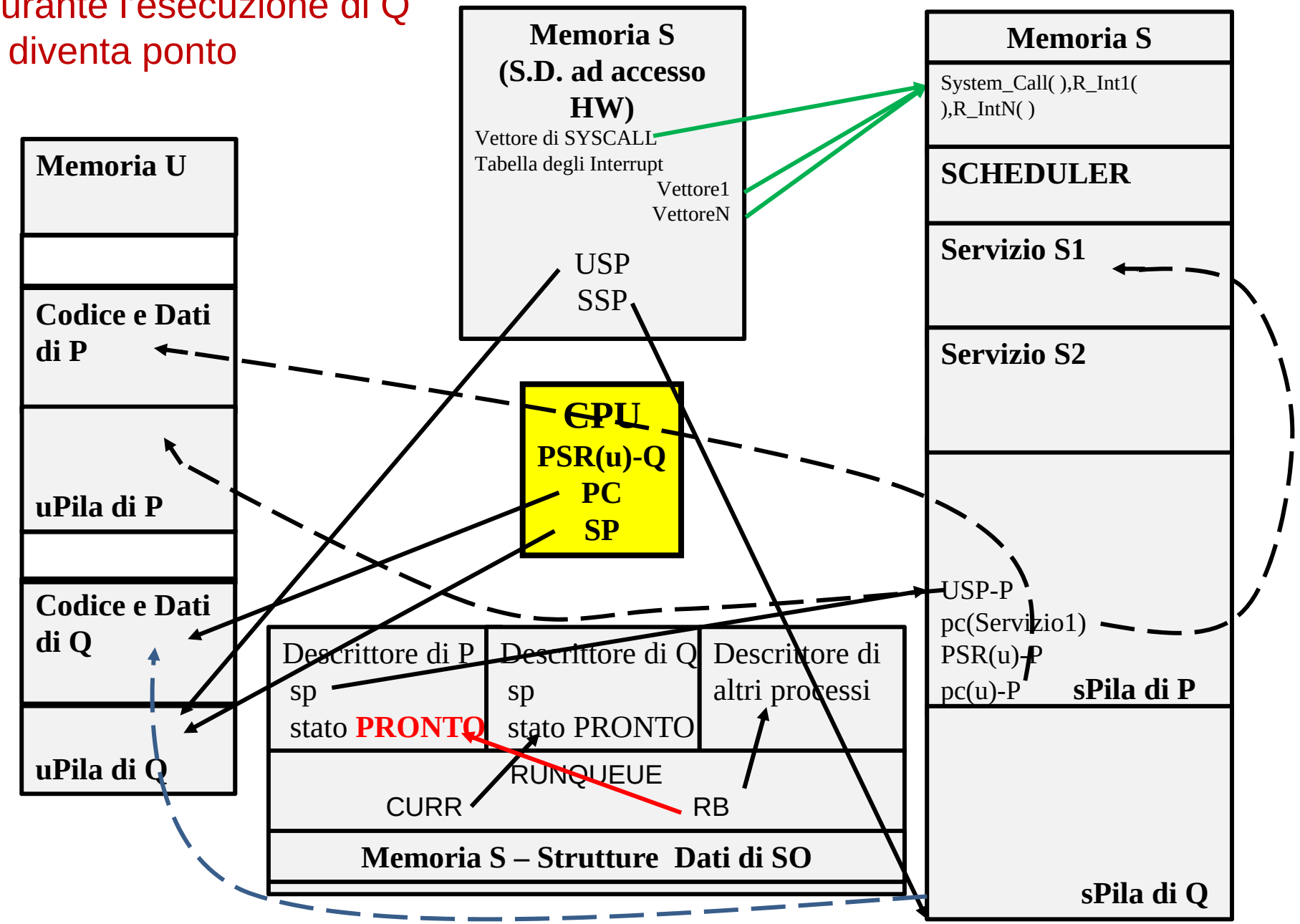
Lo scheduler sostituisce SSP e USP ed esegue return a S2



S2 ritorna a system_call che esegue SYSRET



Durante l'esecuzione di Q
P diventa ponto



Considerazioni conclusive sull'esempio

- lo stato finale raggiunto è identico a quello iniziale, scambiando i processi P e Q
- quindi lo stato iniziale ipotizzato è effettivamente quello raggiunto da un processo che si sospende in un servizio e ritorna pronto
- Il modello fondamentale appena analizzato è *semplificato* rispetto a una serie di problemi che dobbiamo affrontare
  - la *gestione degli interrupt* – cosa accade se durante il funzionamento descritto avviene un interrupt
  - la *gestione del passaggio dallo stato di ATTESA a quello di PRONTO*
  - la sospensione forzata dell'esecuzione di un processo (*preemption*) da parte dello scheduler

La gestione degli Interrupt

- quando si verifica un interrupt esiste sempre un processo in stato di esecuzione possono verificarsi i seguenti casi:
  - l'interrupt interrompe il *processo* mentre funziona in *modalità U*
  - l'interrupt interrompe un servizio di sistema che è stato invocato dal processo in esecuzione
  - l'interrupt interrompe una routine di interrupt relativa ad un interrupt con priorità inferiore
- la routine di interrupt svolge la propria funzione senza disturbare il processo in esecuzione (la routine di interrupt è ***trasparente***)
- ***non viene mai sostituito il processo in esecuzione durante l'esecuzione di un interrupt***
  - gli interrupt vengono quindi eseguiti ***nel contesto del processo in esecuzione***
- se la routine di interrupt è associata al verificarsi di un evento E sul quale è in stato di attesa un certo processo P
  - **la routine di interrupt risveglia il processo** P passandolo dallo stato di attesa allo stato di pronto
  - successivamente il processo P potrà tornare in esecuzione.

esempio:

- P era in attesa di un dato dal terminale,
- la routine di interrupt associata al terminale del processo P risveglia tale processo

Gestione dello stato di ATTESA - waitqueue

i tipi di eventi che possono essere oggetto di attesa appartengono a diverse categorie che richiedono una gestione differenziata, in particolare

- attesa del completamento di un'operazione di Input/Output
- attesa dello sblocco di un Lock (ad esempio dovuto a un MUTEX o a un semaforo)
- attesa dello scadere di un timeout (cioè del passaggio di un certo tempo)

waitqueue

- Una `waitqueue` è una lista contenente i puntatori ai descrittori dei processi in attesa di un certo evento
- Una `waitqueue` viene creata ogni volta che si vogliono mettere dei processi in attesa di un certo evento
- L'indirizzo della `waitqueue` costituisce l'identificatore dell'evento

creazione (statica): `DECLARE_WAIT_QUEUE_HEAD  nome_coda`

Attesa esclusiva e non esclusiva

- In alcuni casi conviene risvegliare tutti i processi presenti nella coda (ad esempio, processi che attendono la terminazione di un'operazione su disco)
- In altri conviene risvegliarne uno solo (ad esempio, se molti processi sono in attesa della stessa risorsa bloccata, pichè uno solo potrà utilizzare la risorsa e gli altri dovrebbero tornare immediatamente in attesa)
- I processi per i quali deve esserne risvegliato uno solo sono detti in attesa **esclusiva**
- A questo scopo i processi vengono inseriti in una `waitqueue` con il seguente accorgimento
  - esiste un flag che indica se il processo è in attesa esclusiva oppure no
  - i processi in attesa esclusiva sono inseriti alla fine della coda
- la routine di risveglio opera nel modo seguente
  - risveglia tutti i processi dall'inizio della lista fino al primo processo (incluso) in attesa esclusiva

I segnali e l'attesa interrompibile

Un segnale (***signal***) è un avviso asincrono inviato a un processo dal sistema operativo oppure da un altro processo, ad esempio tramite la chiamata di `kill`. Ogni signal è identificato

- da un numero, da 1 a 31
- da un nome che è nella maggior parte dei casi abbastanza autoesplicativo

Un segnale (***signal***) causa l'esecuzione di un'azione da parte del processo che lo riceve (simile quindi a un interrupt)

- l'azione può essere svolta *solamente quando il processo che riceve il signal è in esecuzione in modo U*
- se il processo ha definito una propria funzione destinata a gestire quel signal, questa viene eseguita, altrimenti viene eseguito il *default signal handler*

La maggior parte dei signal può essere *bloccato* dal processo; un signal bloccato rimane pendente fino a quando non viene sbloccato

Esistono due `signal` che non possono essere intercettati dal processo:

`SIGKILL` – termina immediatamente il processo

`SIGSTOP` – blocca il processo (per riprenderlo più tardi)

I segnali e l'attesa interrompibile (2)

Oltre a `kill`, alcuni signal sono inviati a causa di una particolare configurazione di tasti della tastiera:

- ctrl-C invia il signal `SIGINT` (che causa la terminazione del processo)
- ctrl-Z invia il signal `SIGTSTP` (che causa la sospensione del processo); è simile a `SIGSTOP`, ma il processo può definire un suo handler

Se un signal viene inviato a un processo che non è in esecuzione in modo U

- se il processo esegue in modo S, il signal viene processato immediatamente al ritorno al modo U
- se il processo è pronto ma non in esecuzione, il signal viene tenuto in sospeso finché il processo torna in esecuzione
- se il processo è in *stato di attesa*, ci sono due possibilità che dipendono dal tipo di attesa:
  - se l'attesa è interrompibile (stato TASK_INTERRUPTIBLE) il processo viene immediatamente risvegliato
  - altrimenti (stato TASK_UNINTERRUPTIBLE) il signal rimane pendente
- nel caso di *attesa interrompibile* il processo può essere risvegliato senza che l'evento su cui era in attesa si sia verificato
  - il processo deve controllare al risveglio se la condizione di attesa è diventata falsa e, in caso contrario, rimettersi in attesa

I segnali e l'attesa interrompibile (3)

- Alcune funzioni per mettere un processo in stato di attesa sono le seguenti:
  - wait_event(coda, condizione) – non interrompibile da signal, neppure SIGKILL
  - wait_event_killable(coda, condizione) – interrompibile solo da SIGKILL
  - wait_event_interruptible(coda, condizione) – interrompibile da tutti i signal

- Noi useremo solo le **wait_event_interruptible** (quindi lo stato di ATTESA coincide con TASK_INTERRUPTIBLE)

Funzioni per mettere un processo in attesa e risvegliarlo

Esempio di attesa:

```
DECLARE_WAIT_QUEUE_HEAD  coda_della_periferica
wait_event_interruptible(coda_della_periferica, buffer_vuoto);
```

Risveglio

`wake_up(wait_queue_head_t * wq)`

risveglia tutti i task in attesa non-esclusiva e un solo task in attesa esclusiva. Per ogni task risvegliato

- cambia lo stato a pronto
- lo elimina dalla `waitqueue` e lo pone nella `runqueue`

se un nuovo task aggiunto alla `runqueue` ha maggiori diritti di esecuzione rispetto a quello corrente:

`wake_up` pone a 1 il flag **`TIF_NEED_RESCHED`**

La preemption

Dato che il nucleo è **non-preemptive**:

- non viene eseguita immediatamente una commutazione di contesto quando il sistema scopre che un task in esecuzione dovrebbe essere sospeso e portato in stato di pronto
- ma viene semplicemente settato il flag `TIF_NEED_RESCHED`
- successivamente, al momento opportuno questo flag causerà la commutazione di contesto

La realizzazione concreta della preemption paradossalmente deve *apparentemente* violare la regola fondamentale di non-preemption del nucleo

In realtà **si applica una regola più debole**, cioè la seguente:

- "*una commutazione di contesto viene svolta durante l'esecuzione di una routine del Sistema Operativo solamente alla fine e solamente se il modo al quale la routine sta per ritornare è il modo U*".
- Ovvero: *"il SO prima di eseguire una IRET o una SYSRET che lo riporta al modo U se necessario esegue una preemption",* dove il termine "se necessario" si riferisce alla esistenza di un altro processo con maggiori diritti di esecuzione.

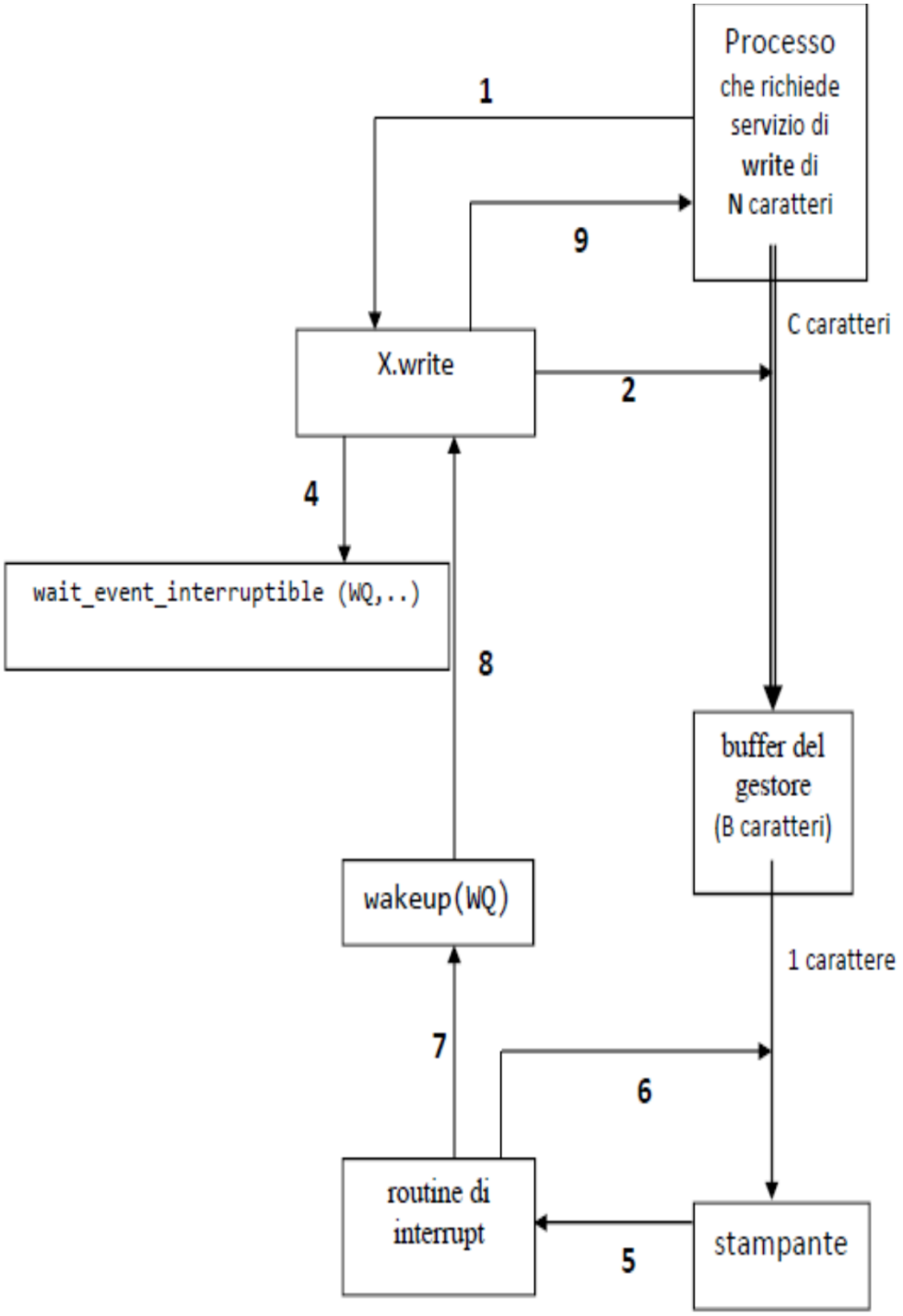
Esempio di attesa non esclusiva: gestori (driver) di periferica

- Consideriamo un processo P che richiede un servizio di scrittura (*write*) di N caratteri relativo a una periferica PX gestita dal gestore X
- Tramite un meccanismo che vedremo nella trattazione dei gestori, l'invocazione del servizio *write*(…) viene trasformato nella invocazione della funzione *X.write( )* del gestore di X
- Quando un processo P richiede un servizio a una periferica PX, se PX non è pronta il processo P viene posto in stato di attesa, e un diverso processo Q viene posto in esecuzione.
- Sarà l'interrupt della periferica a segnalare il verificarsi dell'evento che porterà il processo P dallo stato di attesa allo stato di pronto. Quindi*, quando si verificherà l'interrupt della periferica PX il processo in esecuzione sarà Q (o un altro processo subentrato a Q), ma non P*, cioè non il processo in attesa dell'interrupt stesso
- Questo aspetto è fondamentale nella scrittura di un gestore di periferica, perché implica che, al verificarsi di un interrupt, i dati che la periferica deve leggere o scrivere non possono essere trasferiti al processo interessato, che non è quello corrente, ma devono essere conservati nel gestore stesso

Esecuzione di una write da parte del driver di una stampante

write (fd, vet, N)

1. SYSCALL e invocazione di X.write
2. trasferisci max B caratteri da vet a buffer gestore
3. invia 1 carattere alla stampante
4. mette il processo in attesa che la stampante abbia completato la scrittura di N caratteri, un altro processo va in esecuzione
5. Interrupt per fine stampa carattere e attivazione Routine interrupt
6. Se buffer non vuoto invia prox carattere alla stampante
7. Se buffer vuoto, risveglia processo (PRONTO)
8. Schedule manda in esecuzione X.write.
9. Se sono stati stampati N caratteri il processo torna in modalità utente,
10. Altrimenti torna al passo 2



Esempio di attesa esclusiva: implementazione dei MUTEX

- Linux utilizza un meccanismo detto **futex** (**Fast Userspace Mutex**) per realizzare i Mutex in maniera efficiente
- Un Futex ha due componenti
  - una variabile intera nello spazio U che rappresenta il mutex
  - una `waitqueue WQ` nello spazio S
- Il test della variabile intera e il decremento (lock) sono svolti in maniera atomica in spazio U (deve esistere una sola istruzione in linguaggio macchina che esegue l'operazione, oppure è necessario implementare il Mutex con l'algoritmo di Peterson, ad esempio)
  - ✓ se il lock può essere acquisito l'operazione ritorna senza bisogno di invocare il SO
  - ✓ se il lock è bloccato allora viene invocata una *system call*, chiamata **sys_futex( )** con parametro `wait` (la unlock invoca **sys_futex( )** con parametro `wake`)
  - ✓ ***sys_futex( ) - wait***
    - invoca `wait_event_interruptible_exclusive(WQ…)` e pone quindi il processo in *attesa esclusiva* sulla `waitqueue` fino a quando il lock non viene rilasciato
- l'implementazione dei Futex costituisce un esempio di uso conveniente dell'attesa esclusiva, perché causerà il risveglio di uno solo degli N processi

Altri tipi di attesa

- l'uso di una `wake_up` (che è riferita a una `waitqueue`) è possibile nelle situazioni in cui la funzione che scoprirà il verificarsi dell'evento atteso conosce la coda relativa all'evento
- esistono però situazioni nelle quali l'evento atteso è scoperto da una funzione che non ha modo di conoscere la `waitqueue`
- in questi casi viene invocata una variante di `wake_up` che riceve come argomento direttamente un puntatore al descrittore del processo da risvegliare:

```
wakeup_process(task_struct * processo)
```

- due esempi sono i seguenti:
  - un processo P esegue exit( ) e deve essere risvegliato il relativo processo padre che si era posto in wait( ). Poiché P possiede nel descrittore un puntatore `parent_ptr` al proprio processo padre, è sufficiente che *exit* invochi `wakeup_process(parent_ptr)`
  - un processo deve essere risvegliato a un *preciso istante di tempo*

Attesa di un timeout

- Un **timeout** definisce una scadenza temporale
- esistono servizi per definire i *timeout* in vario modo, ma il principio di base è sempre quello di specificare un intervallo di tempo a partire da un momento prestabilito
- il tempo interno del sistema è rappresentato dalla variabile `jiffies` che registra il numero di tick del clock di sistema intercorsi dall'avviamento del sistema
- la durata effettiva dei `jiffies` *dipende quindi dal clock del sistema*
- i servizi di sistema permettono di specificare l'intervallo di tempo secondo diverse rappresentazioni esterne, che dipendono dalla scala temporale e dal livello di precisione desiderato
- supponiamo che la rappresentazione sia basata sul tipo `timespec`
- il servizio `sys_nanosleep(timespec t)` definisce una scadenza posta un tempo t (in nanosecondi) dopo la sua invocazione e pone il processo in stato di ATTESA fino a tale scadenza

Gestione dei timer

`sys_nanosleep` svolge le seguenti azioni

```
current->state = ATTESA;
schedule_timeout(timespec_to_jiffies(&t) )
```

dove
`timespec_to_jiffies(timespec * t)` converte t dalla rappresentazione esterna ai jiffies

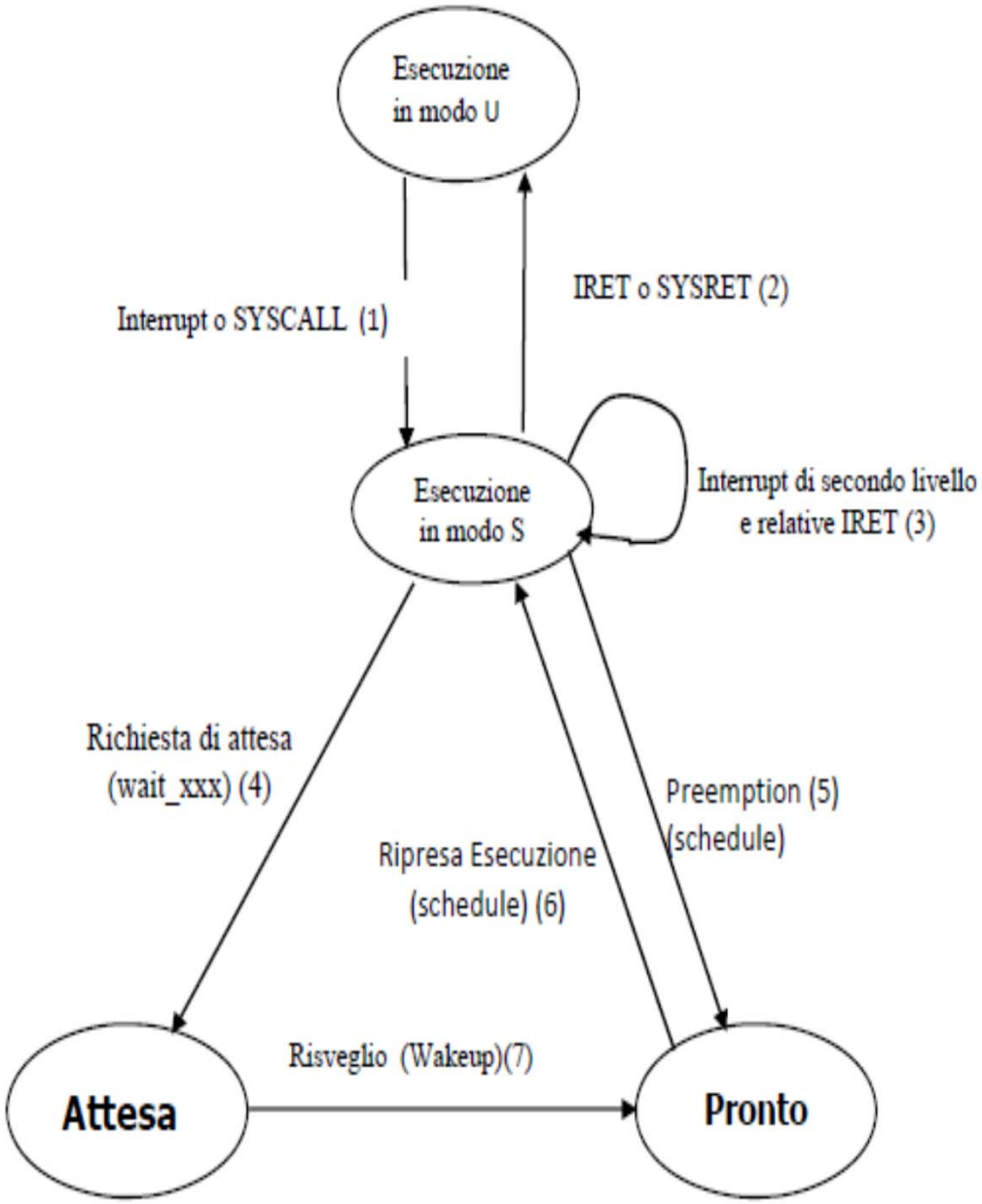
```
schedule_timeout(timeout t){
    struct timer_list timer;          //definisce un elemento timer
    init_timer(&timer)                //inizializza il timer
    timer.expire = t + jiffies;       //calcola la scadenza
    timer.data = current;             //puntatore al descrittore del processo
    timer.function = wakeup_process; //funzione da invocare alla scadenza
    add_timer(&timer)        //aggiunge il nuovo timer alla lista dei timer
    schedule( );   //il processo viene sospeso, perché il suo stato è ATTESA
    delete_timer(&timer);    //quando riparte il processo, elimina il timer
}
```

Risveglio quando scade il timer

- L'interrupt del clock aggiorna i `jiffies`
- Il controllo della scadenza dei timeout non può essere svolto ad ogni tick per ragioni di efficienza
- la soluzione è complessa e noi ipotizzeremo semplicemente che esista una routine Controlla_timer( ) che controlla periodicamente la lista dei timeout (ordinata in ordine di scadenze) per verificare se qualche timeout è scaduto
- un timer scade quando il valore corrente dei `jiffies` è maggior di `timer.expire`
- quando il timer scade, Controlla_timer invoca la funzione `timer.function` passandole `timer.data` come parametro, quindi in questo caso

```
wakeup_process(timer.data)
//risveglia il processo che aveva invocato il servizio
```

Riassunto delle transizioni di stato



Funzioni dello scheduler utilizzate dalla gestione dello stato - 1

schedule ( )

- if (CURR.stato == ATTESA) dequeue_task(CURR) (questo caso si verifica se schedule è stata invocata da una funzione di tipo wait_xxx)
- esegui il *context switch*

check_preempt_curr ( )

- verifica se il task deve essere preempted (in tal caso pone TIF_NEED_RESCHED a 1); invocata da wake_up che modifica l'insieme dei processi pronti

enqueue_task ( )

- inserisce il task nella runqueue

dequeue_task ( )

- elimina il task dalla runqueue

Funzioni dello scheduler utilizzate dalla gestione dello stato - 2

resched( )

- pone TIF_NEED_RESCHED a 1
- in tutti i punti in cui in precedenza abbiamo detto che una funzione pone TIF_NEED_RESCHED a 1, in realtà l'operazione è realizzata invocando resched( )

task_tick( )

- scheduler periodico, invocata dall'interrupt del clock
- interagisce indirettamente con le altre routine del nucleo
- aggiorna vari contatori e determina se il task deve essere preempted perchè è scaduto il suo quanto di tempo (in tal caso invoca resched).

Pseudocodice di wait_event_interruptible_xxx

```
void wait_event_interruptible(coda, condizione) {
     //costruisci un elemento della waitqueue che punta
     //al descrittore del processo corrente
     //aggiungi il nuovo elemento alla waitqueue
     //poni i flag a indicare Non Esclusivo oppure
     //Esclusivo, in base a XXX
     //poni stato del processo (current->state) a ATTESA

      schedule( );          //richiedi un context switch;

      //quando il processo riparte: rimuovi il processo
      // corrente dalla waitqueue
}
```

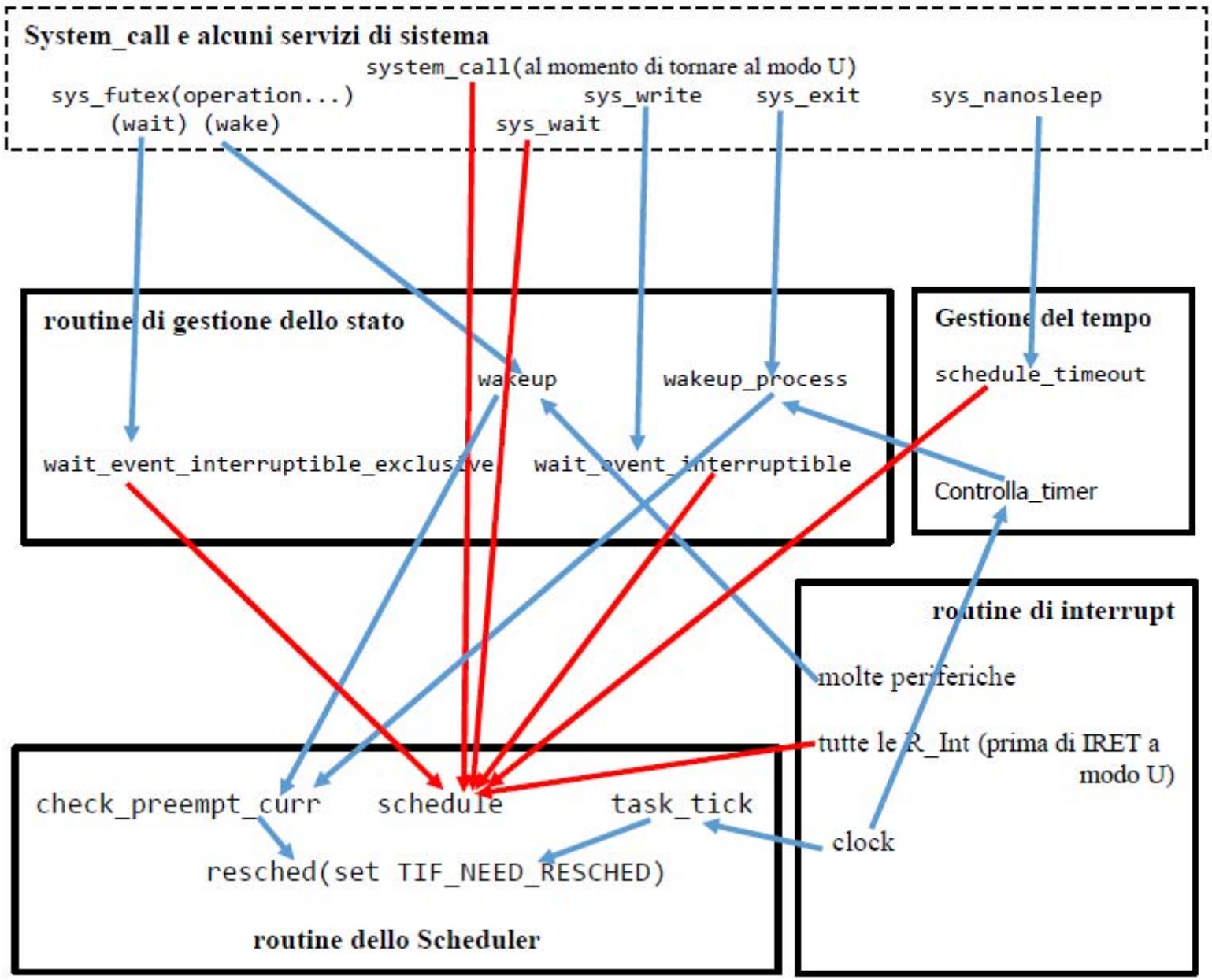
Pseudocodice di wakeup

```
void wake_up(wait_queue_head_t *wq)
{
    //per ogni descrittore puntato da un elemento di wq
    {
        //cambia lo stato a PRONTO
        enqueue( );          //inseriscilo nella runqueueu
        //eliminalo dalla waitqueue
        //se flag indica esclusivo, break
    }

    check_preempt_curr();
     //verifica se è necessaria la preemption
}
```

Pseudocodice di R_int_clock (..)

```
void R_int_clock(… )
{  // attivata dall'interrupt di real time clock

   //gestisce i contatori di tempo reale (data, ora …)

   //con peridicità opportuna
       task_tick( );                 //controlla  se è scaduto il qdt
                                       //del processo corrente

   //con peridicità opportuna
       Controlla_timer( );           //controlla lista dei timeout

    if (modo di rientro == U && TIF_NEED_RESCHED == 1)
        schedule();

    IRET
}
```

Mappa delle funzioni trattate finora



Codice assembler nei sorgenti Linux

Esempio: commutazione di contesto

La funzione `schedule( )` invoca la funzione `context_switch()`, che a sua volta contiene la macro assembler

```
#define switch_to(prev, next)
```

la lettura del codice di questa macro richiede di conoscere

- lo specifico assembler
- le regole del "*gnu inline assembler*", che serve a collegare le variabili C con gli indirizzi di memoria e i registri assembler

l'operazione centrale consiste nella sostituzione della sPila del processo uscente con la sPila del processo nuovo - nel x64 si tratta delle due seguenti istruzioni

```
movq  %rsp, threadrsp( %prev)
movq  threadrsp(%next), %rsp
```

Gestione della concorrenza nel nucleo

Cause di concorrenza:

a. l'esistenza di molte CPU che eseguono in parallelo
b. la sospensione di un'attività a causa di una commutazione di contesto, con partenza di una nuova attività

esempio (in assenza della regola di non-preemption del nucleo):

- un processo P ha chiesto di eseguire un servizio di sistema servizio1( )
- durante l'esecuzione di servizio1( ) si esegue la preemption di P, interrompendo servizio1( ) in un momento arbitrario
- un nuovo processo Q parte e richiede l'esecuzione dello stesso servizio1( ) o di un altro servizio2( ) che comunque interagisce con le strutture dati usate da servizio1( )
- si è creata una situazione simile a quella di due thread che eseguono una funzione in parallelo, con tutti i problemi legati alla concorrenza

Non preemption del nucleo

La regola di non-preemption del Kernel riduce i problemi di esecuzione concorrente tra diverse funzioni del Kernel, perché

- l'autosospensione di un servizio avviene in maniera controllata
- la commutazione di contesto avviene quando non ci sono *attività del Kernel* in corso

Nei sistemi mono-processore:

- la causa (a) di concorrenza non sussiste
- il controllo della commutazione di contesto rende impossibile l'esistenza di due servizi arbitrari in esecuzione concorrente, semplificando molto i problemi di concorrenza

La regola di non-preemption oltre a risolvere il problema della sincronizzazione rendere più semplice il salvataggio e il ripristino dello stato di un processo nelle commutazioni di contesto

Primitive di sincronizzazione interne al nucleo

- Linux implementa i Mutex mettendo il processo che trova una risorsa bloccata in stato di attesa su una `waitqueue`
- questo approccio non è utilizzato per la maggior parte delle sincronizzazioni interne al SO, perché:
  - si tratta di attese molto brevi
  - mentre l'operazione di cambio di contesto è onerosa
- Linux implementa un diverso tipo di primitive di lock: gli spinlock basati su un ciclo di attesa (busy waiting):
  - se il task non ottiene il lock, continua a tentare (spinning) finchè lo ottiene
  - gli spinlock sono molto piccoli e veloci e possono essere utilizzati ovunque nel kernel
- il difetto degli spinlock consiste quindi nel fatto che impediscono di passare all'esecuzione di un altro thread finchè:
  - non sono riusciti ad ottenere il lock
  - oppure il thread che sta tentando di ottenerlo viene preempted dal SO
- questo meccanismo ha senso solo perchè esistono i multiprocessori:
  - nei sistemi monoprocessore non-preemptive un processo che esegue in modo S non ha bisogno di utilizzare i lock nei servizi di sistema (qualche accorgimento!!!)