# Gestione dello Stato dei Processi

# Stato del Processo

- un processo può trovarsi in uno dei due stati fondamentali seguenti:
  - ***ATTESA***: un processo in attesa non può essere messo in esecuzione, perché deve attendere il verificarsi di un certo ***evento***
  - per esempio, un processo ha invocata la funzione di ingresso *scanf* ( ) e attende che da terminale venga inserito un dato
  - ***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*** (***current process***)
- lo stato di ciascun processo è registrato nel suo descrittore

# Contesto di un Processo

- un processo è in esecuzione in modo U per la maggior parte del suo tempo, e allora si dice che ***il processo è in esecuzione in modo U***
- se il processo corrente richiede un ***servizio di sistema*** (tramite l'istruzione macchina *SYSCALL*), viene attivata una funzione del *SO*, la quale esegue il servizio ***per conto di tale processo***
  - per esempio se il processo richiede una lettura da terminale, il servizio di lettura del *SO* legge un dato dal terminale ***associato al processo in esecuzione***
  - pertanto i servizi sono parametrici, in una certa misura, rispetto al processo che li richiede
  - si intenderà questo fatto dicendo che un servizio è svolto ***nel contesto*** di un certo processo
- si dice 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 lascia tale stato solo a causa di uno di questi due eventi:
  1. quando un servizio di sistema richiesto dal processo corrente si deve porre in attesa di un evento
     - in questo caso il processo passa dallo stato di esecuzione allo **stato di *ATTESA***
     - esempio: se il processo P ha richiesto il servizio di lettura (*read*) di un dato dal terminale, ma il dato non è ancora disponibile, allora il servizio si pone in attesa dell'evento *arrivo del dato dal terminale del processo P*
     - nota: *un processo va in stato di attesa quando esegue un servizio di sistema* (cioè quando il *SO* esegue un servizio di sistema per conto del processo stesso), *non quando esegue il suo codice utente in modo U*
  2. quando il *SO* decide di sospendere l'esecuzione del processo corrente a favore di un altro processo (*preemption*)
     - in questo caso il processo passa dallo stato di esecuzione allo **stato di *PRONTO***

# Scheduler

- lo *scheduler* è il componente del sistema operativo che decide quale processo mettere in esecuzione
- lo *scheduler* svolge due tipi di funzione:
  - determina quale processo va messo in esecuzione, in quale momento e per quanto tempo, cioè realizza la ***politica di scheduling*** (*scheduling policy*) 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***, che diventa il nuovo processo corrente
- la politica di *scheduling* verrà affrontata più avanti (si può impostare secondo vari criteri)
- la commutazione di contesto è svolta dalla funzione ***schedule*** ( ) dello *scheduler*
- lo *scheduler* gestisce la ***runqueue,*** una struttura dati fondamentale del nucleo
- ciascun processore ha una sua *runqueue*, la quale consiste di due campi:
  - ***RB***: è una lista di puntatori ai descrittori di tutti i processi pronti, escluso quello corrente
  - ***CURR***: è un puntatore al descrittore del processo corrente

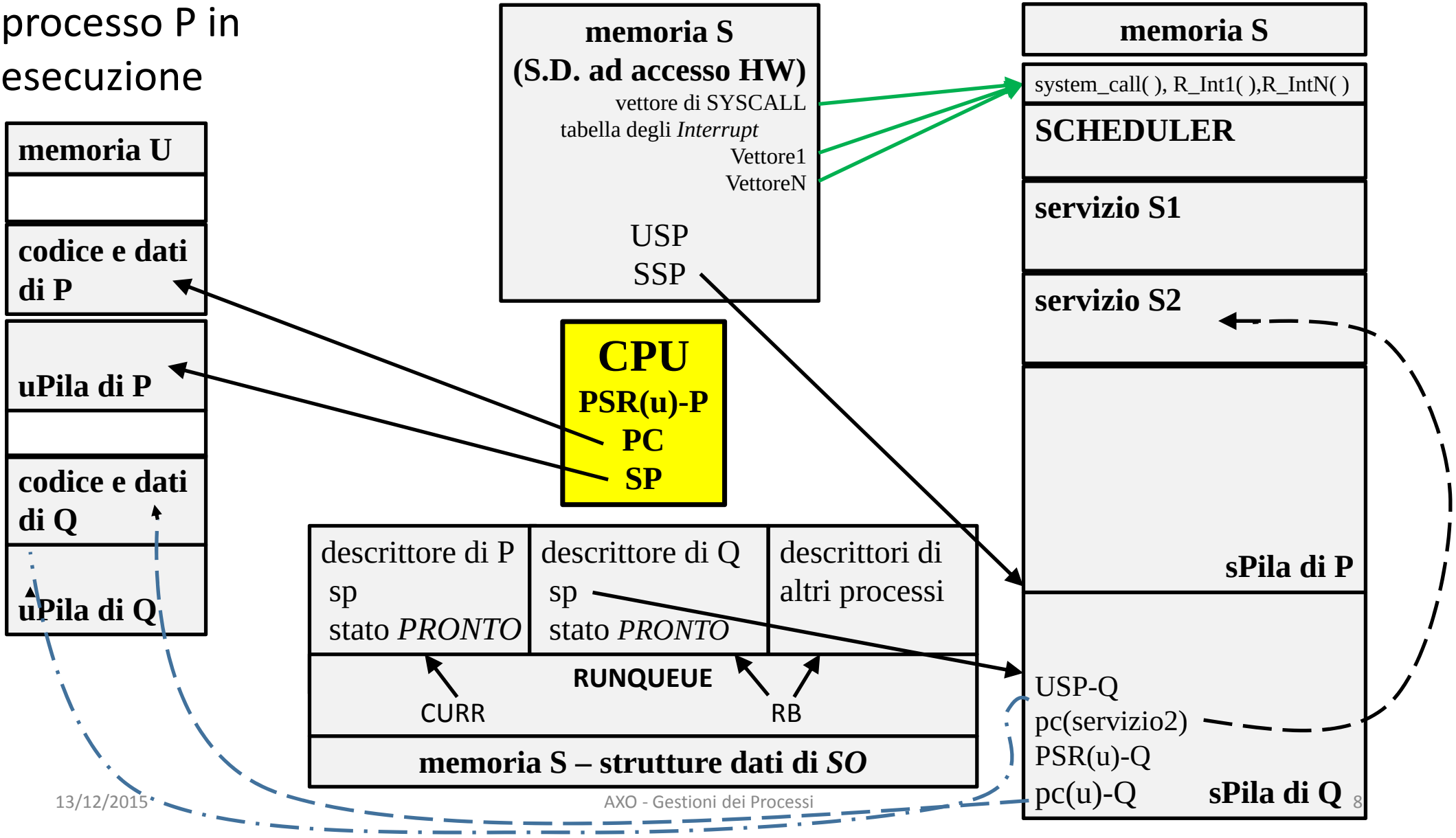
# La sPila del Processo

- il *SO* Linux assegna a ciascun processo una pila di sistema (sPila) di **8 K byte max**
- durante l'esecuzione di un servizio di sistema per conto di un processo, la sPila del processo contiene una parte del contesto hardware del processo stesso
- il meccanismo *HW* che sopraintende ai passaggi di modo permette la commutazione corretta da uPila a sPila e viceversa, a condizione che le celle *SSP* e *USP* contengano i valori corretti da assegnare al registro *SP* del processore
- però, dato che il *SO* mantiene una diversa sPila per ogni processo, la gestione di questo meccanismo per la commutazione di contesto diventa piuttosto complessa:
  - richiede di salvare i valori delle celle *SSP* e *USP* nell'intervallo di tempo tra la sospensione dell'esecuzione di un processo e la successiva ripresa dell'esecuzione dello stesso processo
- a tale scopo il descrittore di un processo P contiene i seguenti campi di appoggio:
  - ***sp0***: indirizzo della base della sPila di P (è definito al momento della creazione di P)
  - ***sp***: valore del registro *SP* salvato nel momento in cui il processo P sospende l'esecuzione

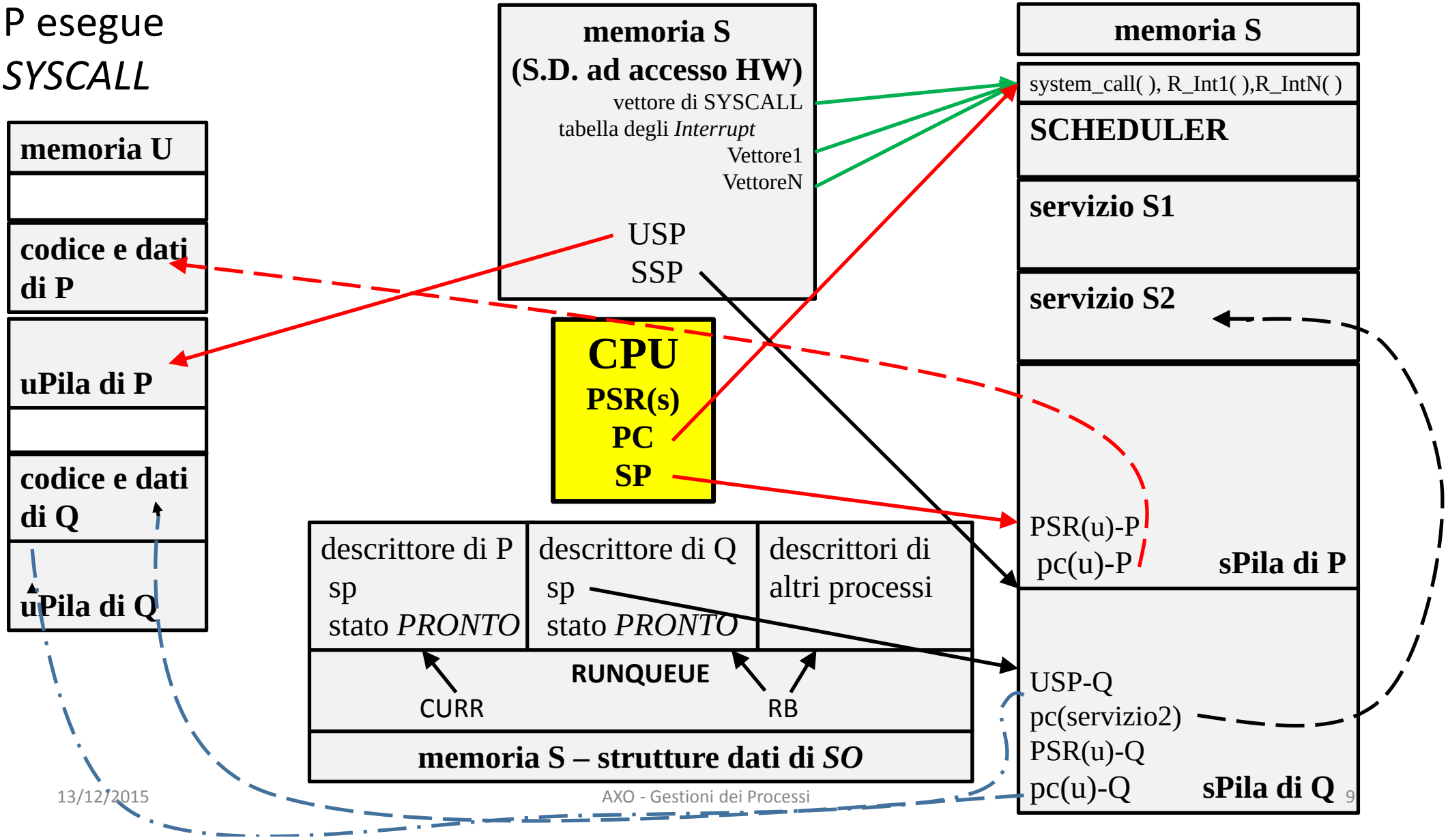
# Gestione di *SSP* e *USP* nella Commutazione di Contesto

- quando un generico processo P è in esecuzione in modo U, la sPila di P è sempre vuota
- la cella *SSP* contiene l'indirizzo base della sPila di P, preso dal campo *sp0* del descrittore di P
- quando la *CPU* passa a modo S (con istruzione macchina *SYSCALL* oppure via *interrupt*), la cella *USP* viene caricata automaticamente da parte dello *HW* con l'indirizzo della cima corrente della uPila di P, cioè con il valore che il registro *SP* ha subito prima di passare a modo S
- i registri *PC* e *SP* passano a puntare al *SO* e alla sPila di P, come visto nei precedenti esempi
- se durante l'esecuzione in modo S del processo P viene eseguita una commutazione di contesto e si passa a eseguire un altro processo, il *SO* Linux opera nel modo seguente:
  - salva il valore della cella *USP* (puntatore a uPila di P) mettendolo sulla sPila di P
  - poi salva il valore del registro *SP* mettendolo nel campo *sp* del descrittore di P
- quando prima o poi il processo P riprenderà l'esecuzione:
  - il registro *SP* verrà ricaricato prendendolo dal campo *sp* del descrittore di P, e così tornerà a puntare alla cima della sPila di P
  - la cella *USP* verrà ricaricata prendendola dalla sPila di P, che è appena tornata accessibile
  - la cella *SSP* verrà ricaricata prendendola dal campo *sp0* del descrittore di P
- esempio: un processo P è in esecuzione, mentre un altro processo Q è in stato di *PRONTO*

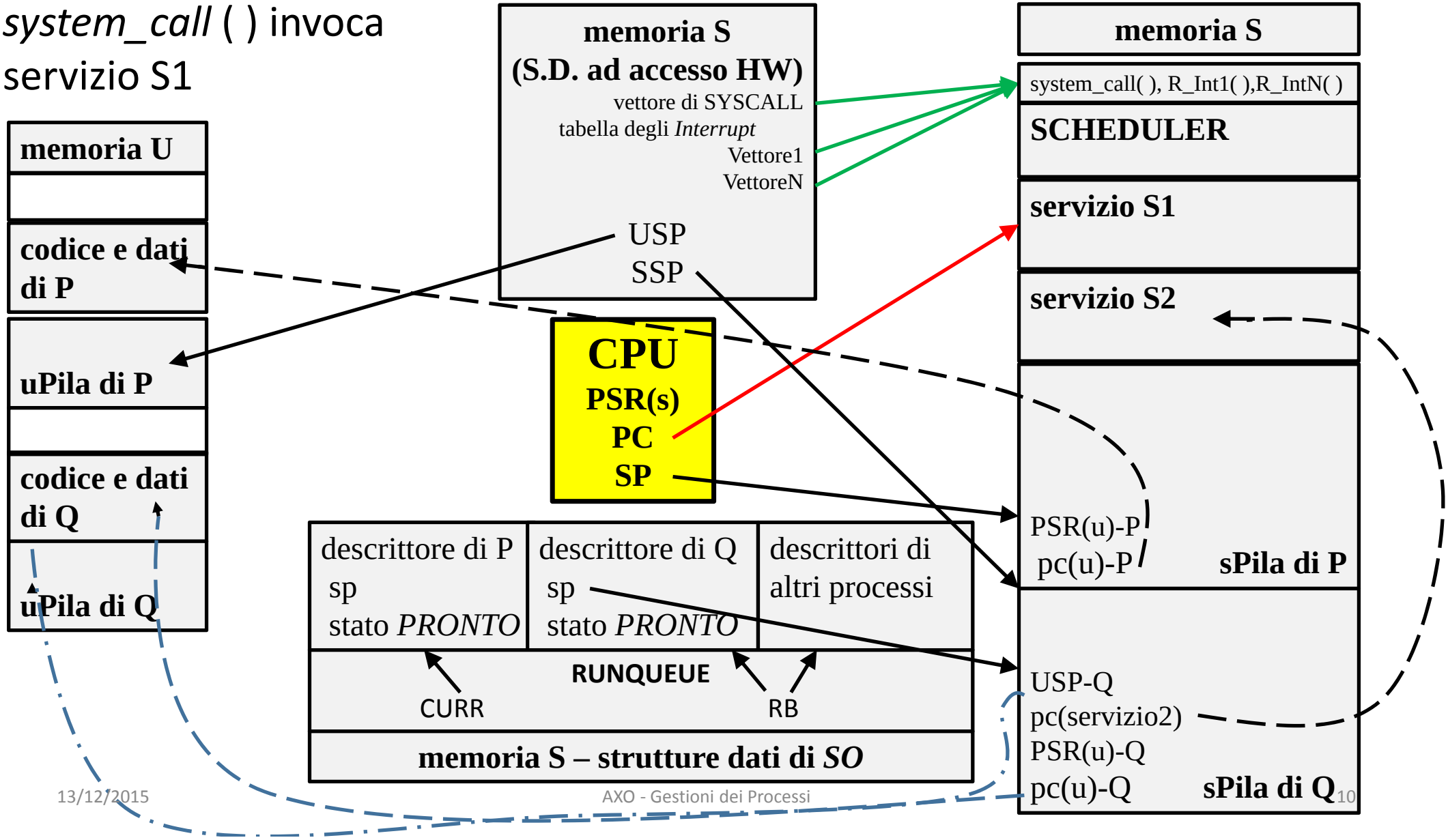
processo P in esecuzione



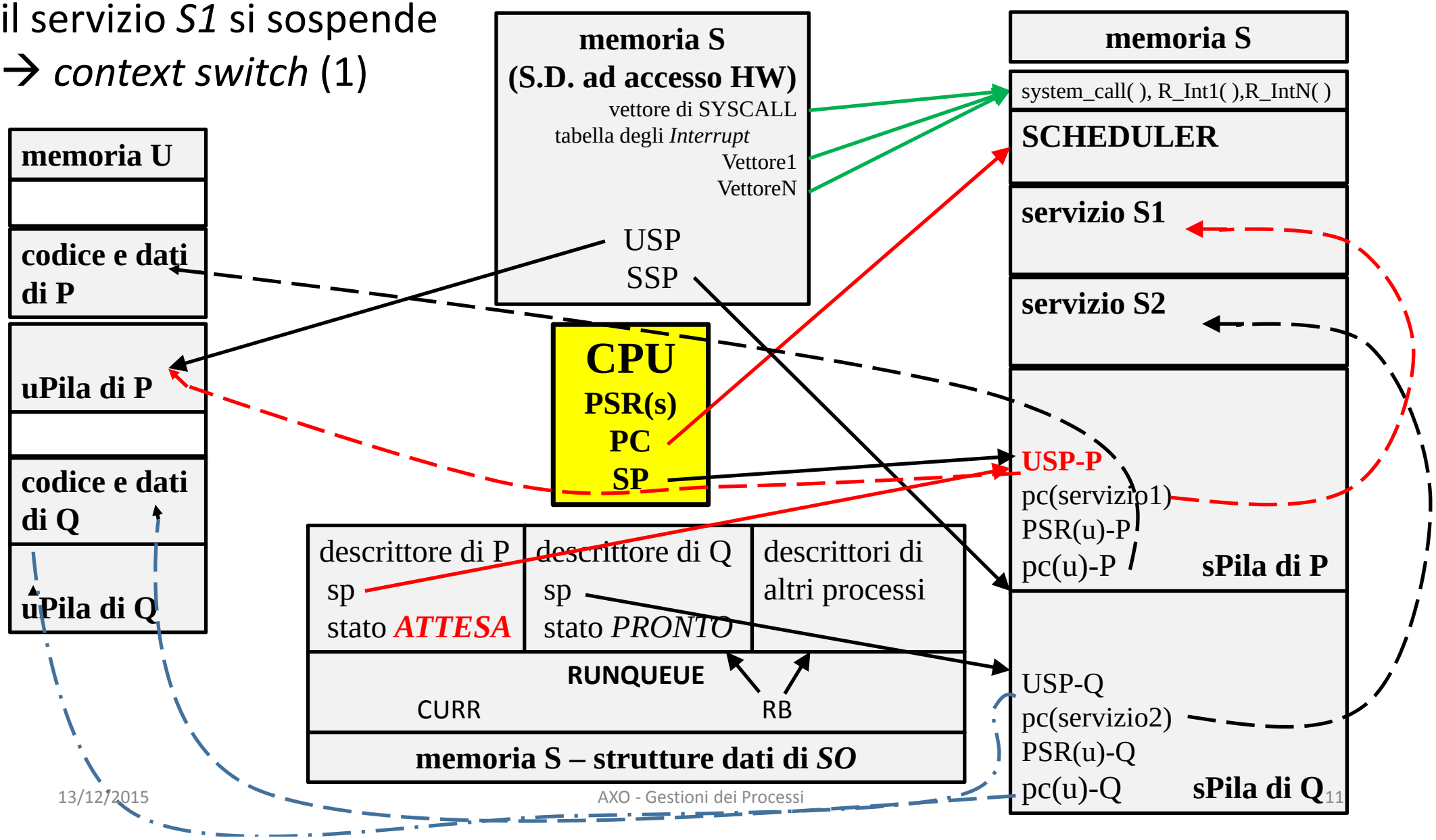
P esegue
*SYSCALL*



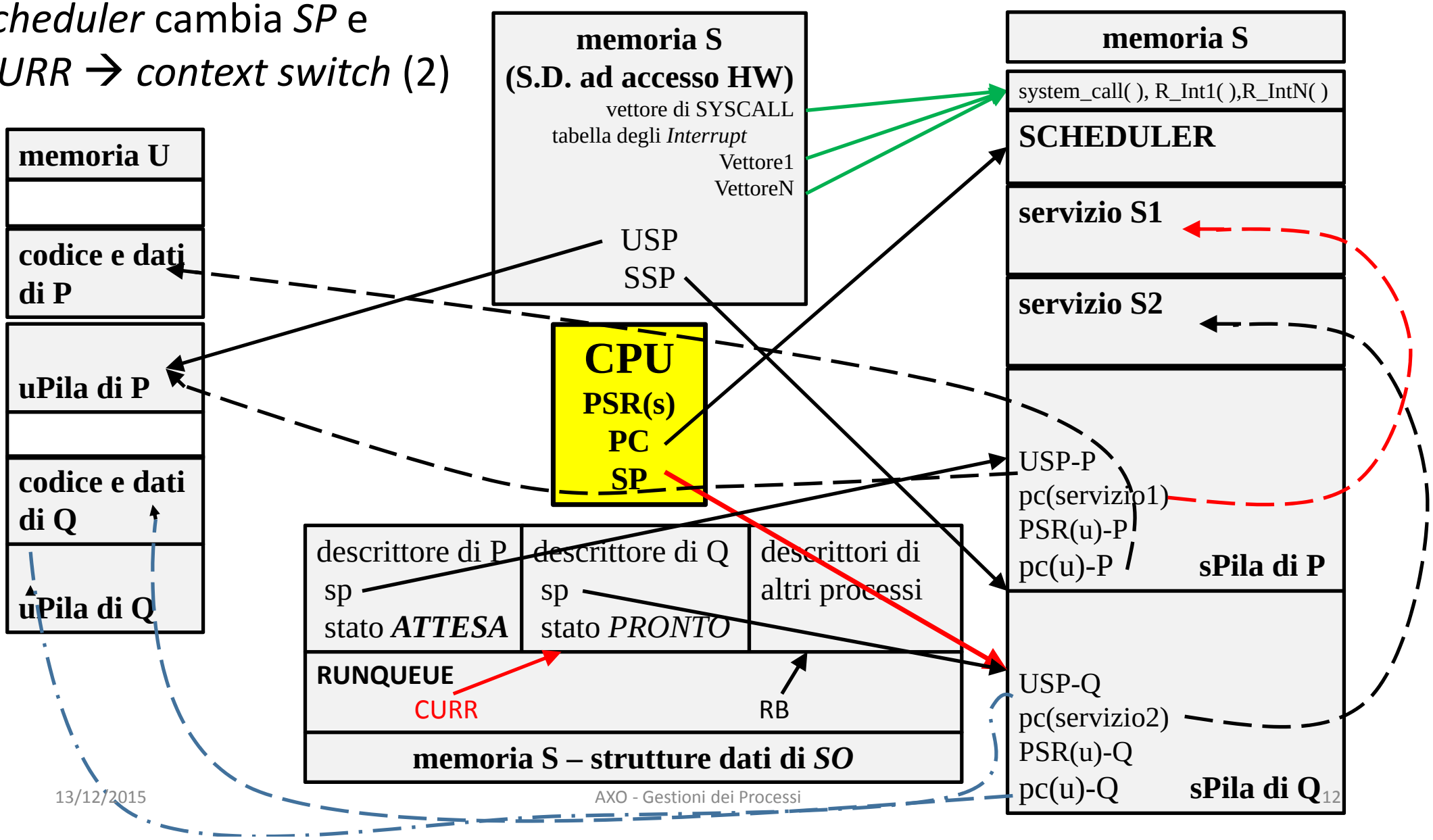
*system_call* ( ) invoca servizio S1



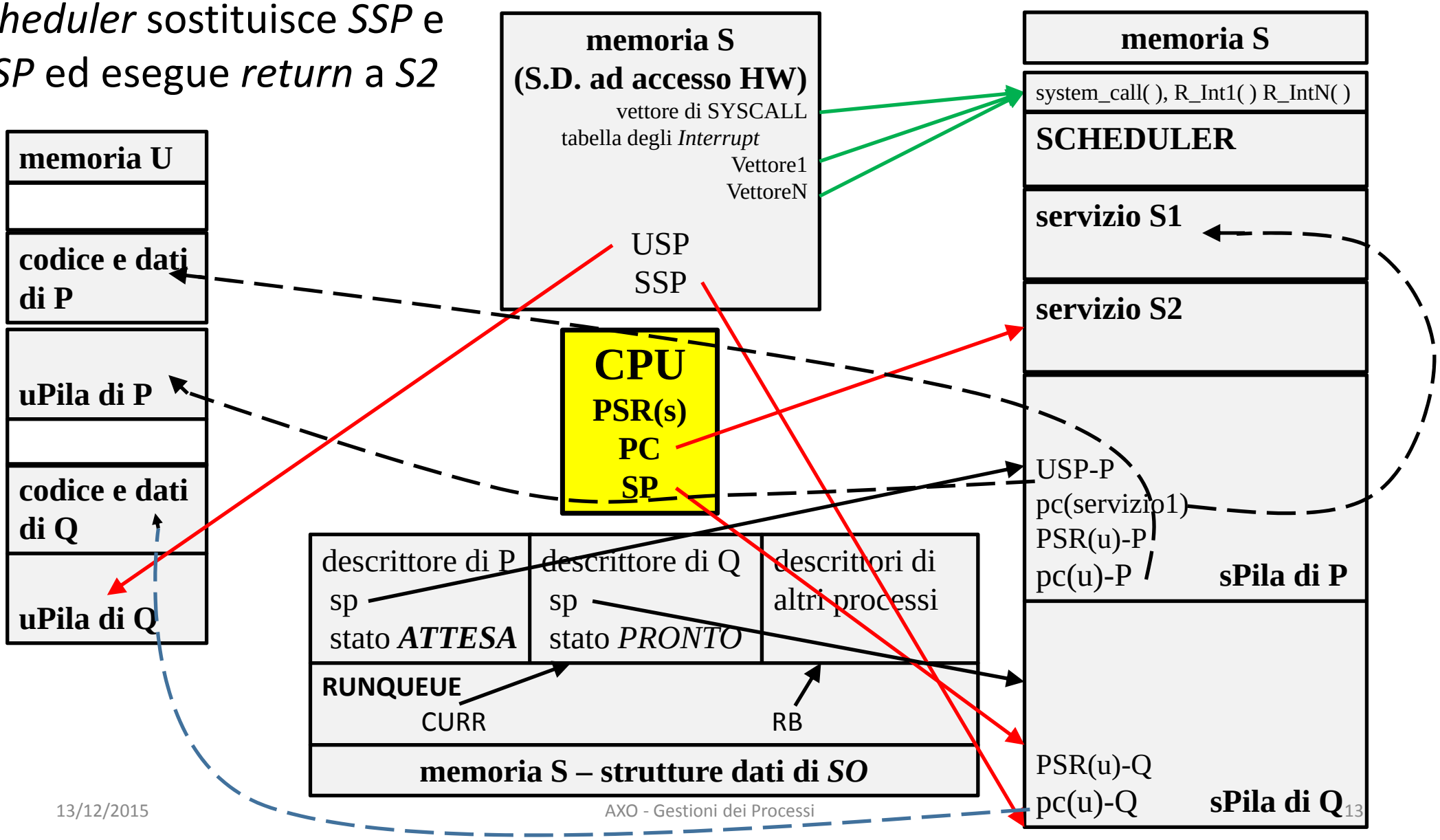
il servizio *S1* si sospende
→ *context switch* (1)



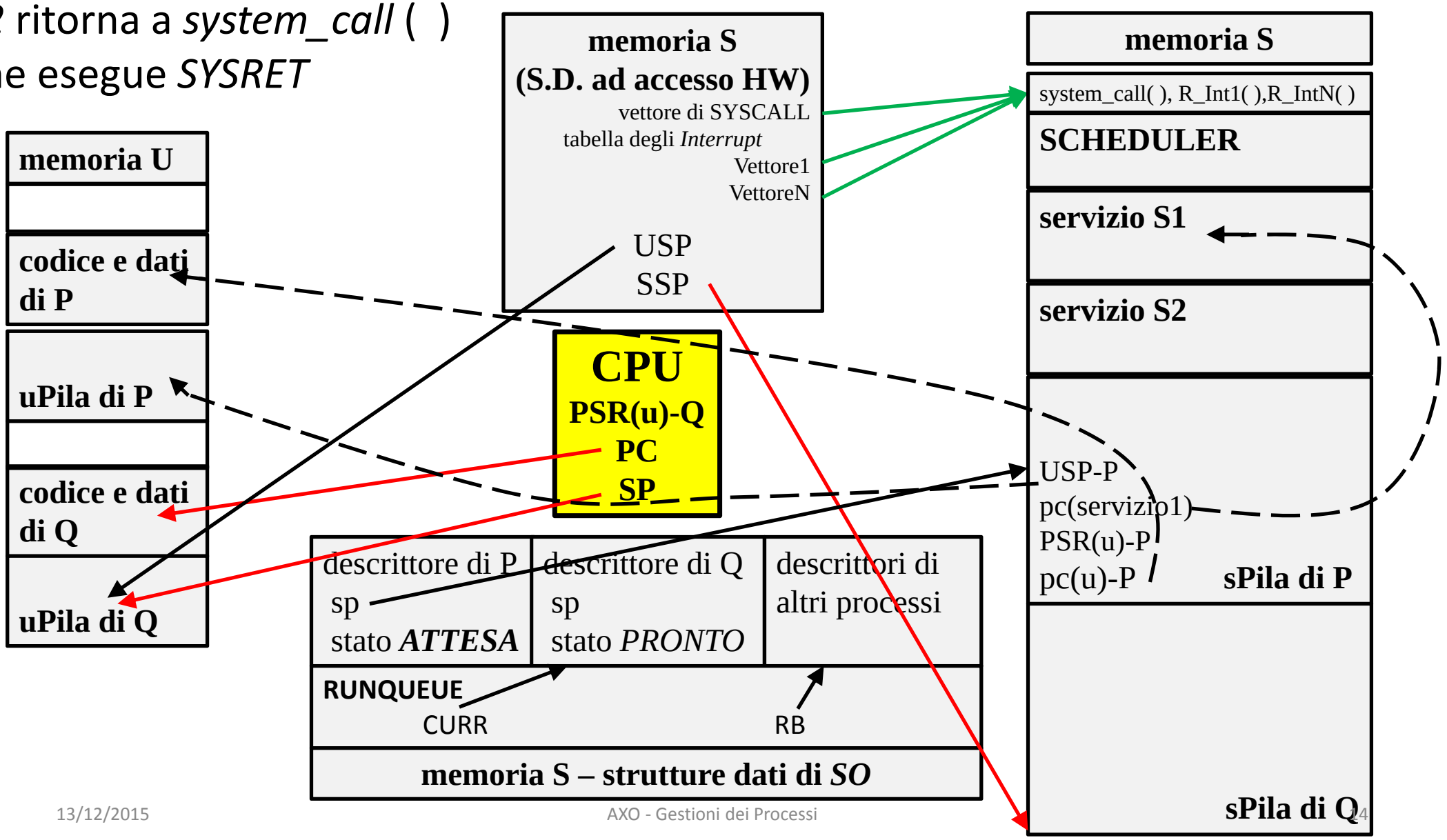
*scheduler* cambia *SP* e
*CURR* → *context switch* (2)



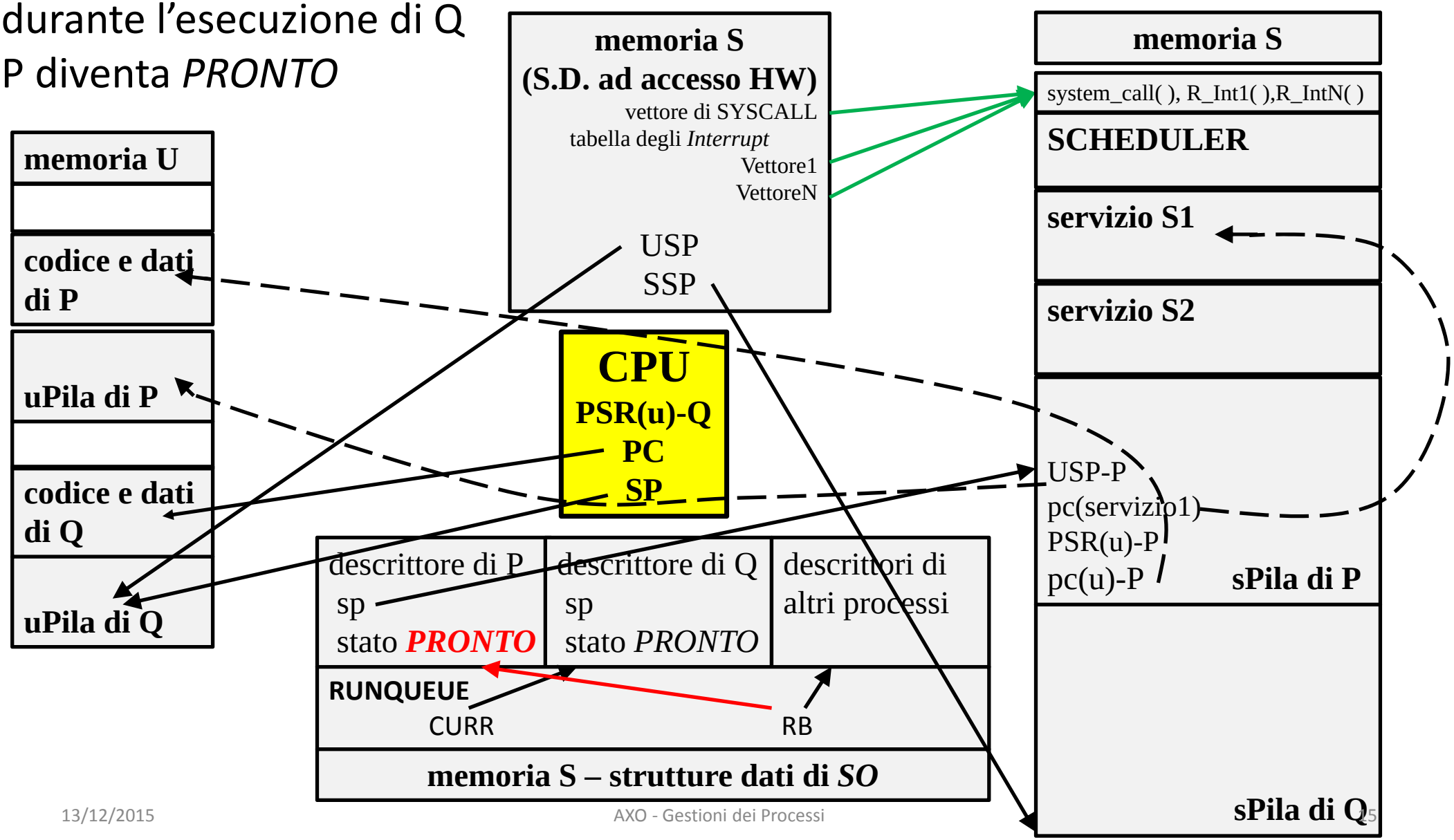
*scheduler* sostituisce *SSP* e *USP* ed esegue *return* a *S2*



*S2* ritorna a *system_call* ( )
che esegue *SYSRET*



durante l'esecuzione di Q
P diventa *PRONTO*



# Considerazioni Conclusive sull'Esempio

- lo stato finale raggiunto è identico a quello iniziale, scambiando i processi P e Q
- dunque lo stato iniziale ipotizzato è effettivamente quello raggiunto da un processo che si sospende in un servizio di sistema e in seguito ritorna pronto
- il modello fondamentale appena analizzato è semplificato rispetto a una serie di aspetti e problemi ancora da chiarire e risolvere:
  - la gestione dell'interrupt, cioè che cosa accade se durante il funzionamento descritto si verifica un evento di 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*, a favore di un altro processo

# Gestione dell'Interrupt

- quando si verifica un evento di interrupt, c'è sempre un processo in stato di esecuzione
- si possono verificare i tre casi seguenti:
  - l'evento interrompe il processo in esecuzione mentre questo funziona in modalità U
  - l'evento interrompe un servizio di sistema che è stato invocato dal processo in esecuzione
  - l'evento interrompe una routine di interrupt relativa a un interrupt con priorità inferiore
- la routine di interrupt svolge la propria funzione senza disturbare il processo in esecuzione (si dice che la routine di interrupt è ***trasparente***)
- *durante il servizio di un interrupt non viene mai sostituito il processo in esecuzione*
  - dunque gli interrupt vengono eseguiti (serviti) ***nel contesto del processo corrente***
- se la routine di interrupt è associata al verificarsi di un evento sul quale un certo processo P si trova in stato di attesa, allora:
  - la routine di interrupt risveglia il processo P spostandolo dallo stato di attesa allo stato di pronto
  - successivamente il processo P potrà tornare in esecuzione (cioè ritornerà processo corrente)
- ecco un esempio:
  - un processo P è in stato di attesa di un dato da terminale
  - la routine di interrupt associata al terminale di P risveglia P

# Gestione dello Stato di *ATTESA – waitqueue*

- i tipi di evento che possono essere oggetto di attesa appartengono a svariate categorie che richiedono una gestione differenziata, in particolare:
  - attesa del completamento di un'operazione di Input / Output
  - attesa dello sblocco di un *lock* (per esempio dovuto a un *mutex* o a un *semaforo*)
  - attesa dello scadere di un *timeout* (cioè il passaggio di un certo intervallo di tempo)
- struttura dati ***waitqueue***
  - una *waitqueue* è una lista contenente i descrittori dei processi in attesa di un determinato evento
  - viene creata una *waitqueue* ogniqualvolta si vogliono mettere dei processi in attesa di un determinato 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 di attesa
- esempio: tutti i processi che attendono la terminazione di un'operazione su disco
- in altri casi conviene risvegliare un solo processo tra quelli presenti nella coda di attesa
- esempio: nel caso che numerosi processi siano in attesa della stessa risorsa bloccata, giacché uno solo potrà utilizzare la risorsa e gli altri dovrebbero tornare subito 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:
  - un indicatore (*flag*) specifica se il processo è in attesa esclusiva oppure no
  - i processi in attesa esclusiva sono inseriti alla fine della coda
- la routine di risveglio dei processi risveglia tutti i processi dall'inizio della *waitqueue* fino al primo processo (incluso) in attesa esclusiva

# Segnale e Attesa Interrompibile – I

- un ***segnale*** (***signal***) è un avviso asincrono inviato a un processo da parte del sistema operativo oppure da parte di un altro processo (tramite un'apposita primitiva di sistema)
- il *SO* Linux supporta 31 diversi segnali e ciascun segnale è identificato così:
  - con un numero, da 1 a 31
  - con un nome, che nella maggior parte dei casi è abbastanza autoesplicativo
- un segnale causa l'esecuzione di un'azione da parte del processo che lo riceve, dunque è simile a un evento di interrupt
  - l'azione associata al segnale può essere svolta solamente *quando il processo che riceve il segnale è in esecuzione in modo U*
  - il processo può definire una sua funzione specifica (*signal handler*) per gestire un determinato segnale, altrimenti viene eseguita una funzione standard (*default signal handler*)
- la maggior parte dei segnali può essere *bloccata* dal processo
- un segnale bloccato rimane pendente fino al momento in cui viene sbloccato
- esistono due segnali il cui effetto sul processo è predeterminato e non si può modificare, cioè il processo non può definire un suo *handler* per gestire questi due segnali:
  - `SIGKILL` – fa immediatamente terminare il processo
  - `SIGSTOP` – blocca il processo (per riprenderlo più tardi)
- i segnali hanno vari ambiti di applicazione, uno caratteristico è per sistemi e applicazioni *real-time*

# Segnale e Attesa Interrompibile – II

- certi segnali vengono inviati a seguito di una particolare combinazione 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
  - nota: `SIGTSTP` è simile a `SIGSTOP`, ma il processo può definire un suo *handler*
- se un processo riceve un segnale mentre non è in esecuzione in modo U, allora:
  - se il processo è in esecuzione in modo S, gestirà il segnale non appena sarà tornato a modo U
  - se il processo è in stato di pronto è ma non in esecuzione, il segnale viene tenuto in sospeso fino al momento in cui il processo tornerà 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 risvegliato immediatamente
    - altrimenti (stato TASK_UNINTERRUPTIBLE) il segnale rimane pendente
- in caso di attesa interrompibile può dunque accadere che il segnale risvegli il processo senza che l'evento su cui il processo si era posto in attesa si sia ancora verificato
- pertanto al suo risveglio il processo deve ricontrollare la sua condizione di attesa e, se essa è ancora verificata, rimettersi subito in stato di attesa

# Segnale e Attesa Interrompibile – III

- ecco le quattro funzioni principali per mettere un processo in stato di *ATTESA*:

1. *wait_event* (*coda*, *condizione*) // non interrompibile da segnale, neppure SIGKILL
2. *wait_event_killable* (*coda*, *condizione*) // interrompibile solo da SIGKILL
3. *wait_event_interruptible* (*coda*, *condizione*) // interrompibile da qualsiasi segnale
4. *wait_event_interruptible_exclusive* (*coda*, *condizione*) // come (3) con attesa esclusiva

- parametri della famiglia di funzioni *wait_event*:
  - *coda*: nome di una *waitqueue* creata (staticamente) con DECLARE come visto prima
  - *condizione*: condizione booleana, se vera (o ancora vera) il processo va in attesa, altrimenti no
- qui si useranno solo le funzioni (3) e (4) (*interruptible* e *interruptible_exclusive*), dunque lo stato di *ATTESA* coincide con lo stato TASK_INTERRUPTIBLE

# Funzione per Risvegliare un Processo

- si supponga di avere messo un processo in stato di *ATTESA*, p. es. come visto prima:

  *DECLARE_WAIT_QUEUE_HEAD coda_della_periferica*

  *wait_event_interruptible* (*coda_della_periferica*, *buffer_vuoto == 1*)

- la funzione base per risvegliare un processo in stato di *ATTESA* è questa:

  *wake_up* (*wait_queue_head_t * wq*)

tipica condizione di attesa: il buffer della periferica (qui una di ingresso) è vuoto

- parametri: *wq* è un puntatore a una *waitqueue* (p. es. la *coda_della_periferica*)
- *wake_up* risveglia tutti i task in attesa non esclusiva e uno solo in attesa esclusiva:
  - pone tutti i task risvegliati in stato di *PRONTO*
  - toglie i task risvegliati dalla *waitqueue* e li mette nella *runqueue*
- se un nuovo task aggiunto alla *runqueue* ha diritti di esecuzione maggiori rispetto a quello corrente, allora *wake_up* pone a uno l'indicatore (*flag*) `TIF_NEED_RESCHED`

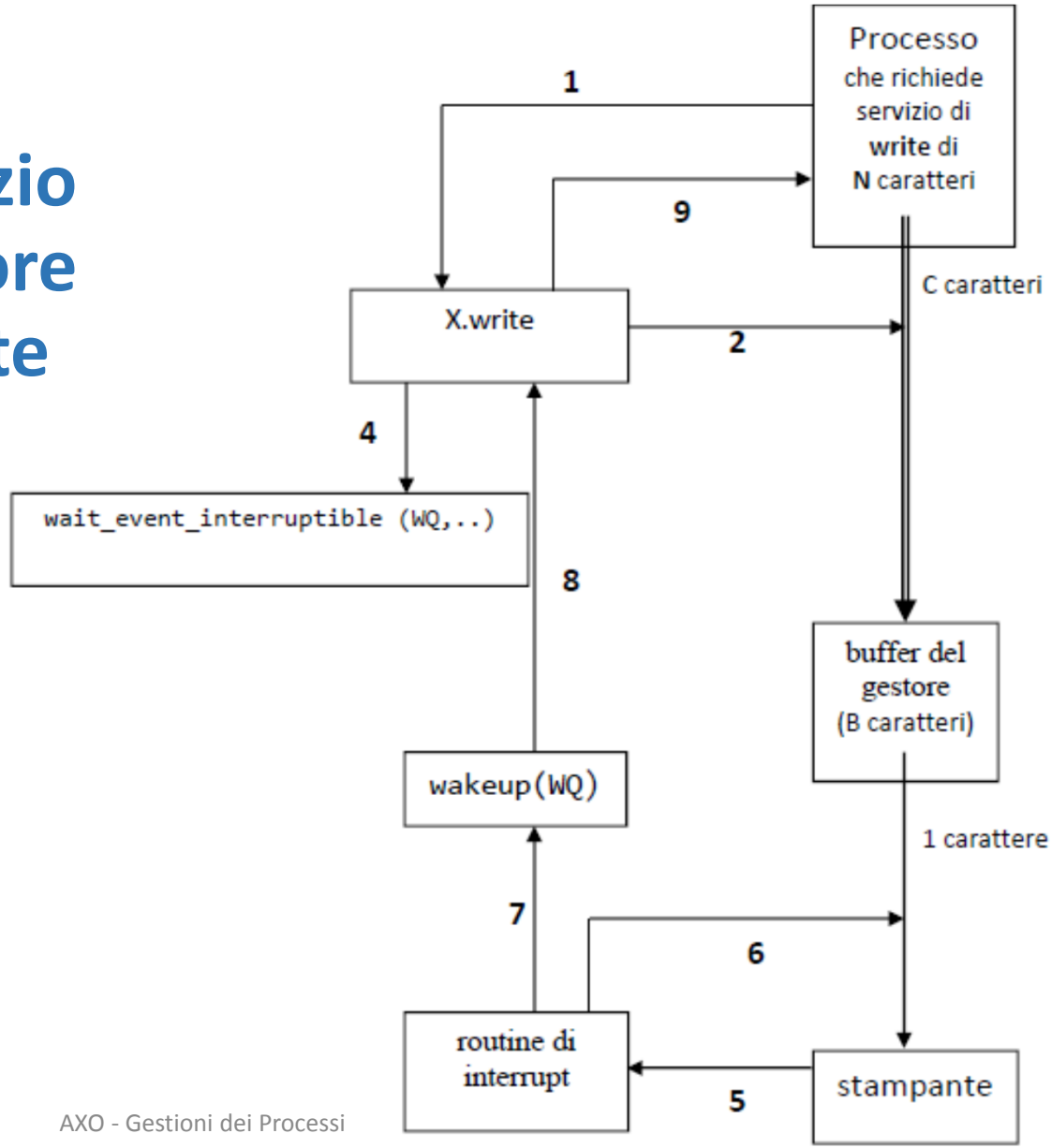
# Regola di *Preemption*

- il nucleo di Linux è di tipo è ***non-preemptive***, dunque:
  - quando il sistema scopre che un task in esecuzione dovrebbe essere sospeso, non esegue immediatamente una commutazione di contesto, ma semplicemente pone a uno l'indicatore `TIF_NEED_RESCHED` (in pratica è una variabile booleana)
  - in seguito, al momento opportuno l'indicatore causerà la commutazione di contesto
- paradossalmente, la realizzazione concreta della *preemption* deve *apparentemente* violare la regola fondamentale di *non-preemption* del nucleo
- in realtà si applica una regola di *preemption più debole*, cioè la seguente:
  - *una commutazione di contesto viene svolta durante l'esecuzione di una routine del sistema operativo solamente alla fine della routine e solamente se il modo al quale la routine sta per ritornare è il modo U*
  - ovvero: *il SO, prima di eseguire un'istruzione macchina IRET o SYSRET che lo riporta al modo U, esegue una preemption se è necessario*, dove la clausola *se è necessario* si riferisce all'esistenza di un altro processo in stato di *PRONTO* e con diritti di esecuzione maggiori di quello corrente

# Esempio di Attesa non Esclusiva: Gestore (*Driver*) di Periferica

- si consideri un processo P che richiede un servizio di scrittura (*write*) di N caratteri relativo a una periferica PX gestita dal *device driver* X
- tramite un meccanismo che si vedrà trattando i *device driver*, l'invocazione del servizio *write* (...) viene trasformata nell'invocazione della funzione *X.write* ( ) del driver di X
- quando un processo P richiede un servizio a una periferica PX, se questa non è pronta il processo P viene posto in attesa, e un diverso processo Q viene posto in esecuzione
- prima o poi un interrupt proveniente dalla periferica segnalerà il verificarsi dell'evento che porterà il processo P dallo stato di attesa allo stato di pronto
- dunque*, quando si verificherà l'interrupt della periferica PX il processo in esecuzione sarà Q* (o addirittura un altro processo nel frattempo subentrato a Q), *non P*, cioè non il processo in attesa dell'arrivo dell'interrupt stesso
- questo aspetto è fondamentale nella scrittura di un *device driver* (*gestore di periferica*)
- esso implica che, al verificarsi di un interrupt, i dati che la periferica deve leggere o scrivere (in sintesi trasferire) non possono essere trasferiti con il processo interessato, che non è più quello corrente, ma vanno conservati nel gestore stesso fino a quando potranno essere trasferiti con il processo corretto, quando sarà di nuovo il processo corrente

# esecuzione del servizio *write* da parte del gestore (*driver*) di una stampante



# Esempio di Attesa Esclusiva – Realizzazione del *mutex*

- il *SO* Linux utilizza un meccanismo chiamato ***futex*** (***fast userspace mutex***) per realizzare il meccanismo di *mutex* in maniera efficiente
- un *futex* è simile a un *semaforo* e ha due componenti:
  - una variabile intera nello spazio U
  - una *waitqueue* WQ nello spazio S
- l'incremento e il test della variabile intera sono svolti in maniera atomica in spazio U
  - 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*** ( )
- la funzione *sys_futex* ( ) invoca *wait_event_interruptible_exclusive* (WQ, ...) e pone il processo in *attesa esclusiva* sulla *waitqueue* fino a quando il *lock* non viene rilasciato
- l'implementazione del *futex* costituisce un esempio di uso conveniente dell'attesa di tipo esclusivo, perché causerà il risveglio di uno solo degli N processi nella *waitqueue*

# Altri Tipi di Attesa

- l'uso di una *waitqueue* è conveniente nelle situazioni dove la funzione che scoprirà il verificarsi dell'evento atteso conosce il nome della coda di attesa relativa all'evento
- però ci sono 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:

$$wake\_up\_process\ (task\_struct * processo)$$

- ecco due esempi di questo approccio:
  - quando un processo P esegue la funzione *exit* ( ) e di conseguenza occorre risvegliare il relativo processo padre Q, che con la funzione *wait* ( ) si era sospeso in attesa delle terminazione di un figlio, dato che nel suo descrittore il processo P ha un puntatore *parent_ptr* al suo processo padre Q, al processo figlio P basta invocare *wake_up_process* (*parent_ptr*)
  - quando un processo P deve essere risvegliato a un preciso istante di tempo, come si spiega dopo

# Attesa di un *Timeout*

- un ***timeout*** definisce una *scadenza temporale* (*deadline*)
- esistono vari servizi per definire *timeout* di diverso tipo, ma il principio di base è 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
- pertanto la durata effettiva di un *jiffy dipende dal clock del sistema*
- i servizi di sistema specificano l'intervallo di tempo secondo diverse rappresentazioni esterne, che dipendono dalla scala temporale adottata e dal livello di precisione desiderato
- per esempio, si faccia l'ipotesi che la rappresentazione sia basata sul tipo *timespec*
- il servizio *sys_nanosleep* (*timespec t*) definisce una scadenza posta a un tempo *t* (espresso in nanosecondi) dopo l'invocazione del servizio, e pone il processo in stato di *ATTESA* fino a tale scadenza

# Gestione di un *Timer*

- la funzione *sys_nanosleep* svolge le azioni seguenti:

  *current->state = ATTESA* // cambia lo stato del processo

  *schedule_timeout* (*timespec_to_jiffies* (&*t*)) // imposta il timer e commuta contesto

- la funzione *timespec_to_jiffies* (*timespec * t*) converte il tempo *t* in numero di *tick* (*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 = wake_up_process; // funzione da invocare alla scadenza
  add_timer (&timer) // aggiunge il nuovo timer alla lista dei timer
  schedule ( );       // il processo viene sospeso poiché ha stato ATTESA
  delete_timer (&timer); // quando il processo riparte, elimina il timer
} /* schedule_timeout */
```

# Risveglio di un Processo alla Scadenza del *Timer*

- l'interrupt del clock del sistema aggiorna il conteggio dei *jiffies*
- per ragioni di efficienza, il controllo della scadenza dei *timeout* non può essere svolto a ogni *tick* del clock di sistema
- la soluzione è complessa e qui si ipotizzerà semplicemente che esista una routine *controlla_timer* ( ) che esamina la lista dei *timeout* (ordinata secondo le scadenze) per verificare se uno o più *timeout* è scaduto
- un *timeout* scade quando il numero corrente di *jiffies* è maggiore della var *timer.expire*
- quando un *timeout* scade, la funzione *controlla_timer* invoca la funzione *timer.function* passandole la var *timer.data* come parametro, ed ecco che cosa accade:

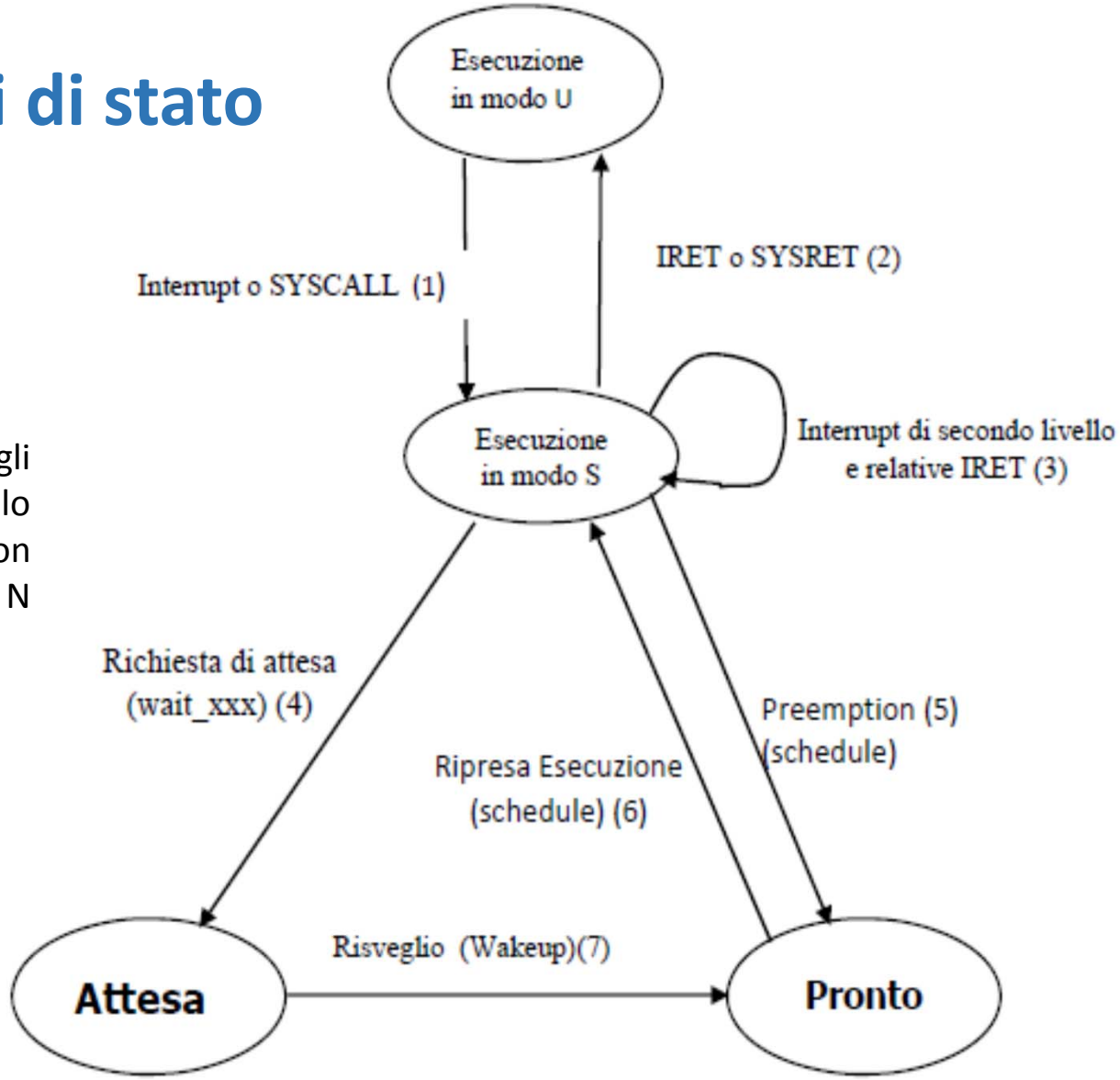
  // *controlla_timer* risveglia il processo che aveva invocato il servizio *sys_nanosleep*

  *wake_up_process* (*timer.data*)

- infatti, come si è visto in *schedule_timeout*, la variabile *timer.data* punta al processo che era quello corrente all'invocazione di *sys_nanosleep* , cioè quello ora da risvegliare

# riassunto delle transizioni di stato

in un sistema monoprocessore, il grafo degli stati è popolato da un solo processo in uno solo degli stati, e in un sistema multiprocessore con N processi, è popolato da esattamente N processi distribuiti tra i vari stati



# Funzioni dello *scheduler* usate per la Gestione dello Stato – I

lo scheduler di Linux è una collezione di funzioni per la commutazione di contesto

*enqueue_task* ( ) inserisci il task nella *runqueue*

*dequeue_task* ( ) elimina il task dalla *runqueue*

*schedule* ( ) // verifica le condizioni per il *context switch* e se del caso lo realizza

**if** (*CURR->stato* == ATTESA) ***then***

// questo caso si verifica se *schedule* è stata invocata da una funzione di tipo *wait_event*

*dequeue_task* (CURR)

esegui la *commutazione di contesto* (*context switch*)

***endif***

*check_preempt_curr* ( )

// è la funzione che realizza la regola debole di *preemption* di Linux espressa prima

verifica se il task deve essere *preempted* e in tale caso poni TIF_NEED_RESCHED a 1

# Funzioni dello *scheduler* usate per la Gestione dello Stato – II

*resched* ( )

// chiamata ovunque prima si è detto che occorre porre TIF_NEED_RESCHED a 1

poni TIF_NEED_RESCHED a 1

*task_tick* ( )

// è lo *scheduler* periodico, invocato dall'interrupt del clock di sistema

// esso interagisce indirettamente con varie altre routine del nucleo

aggiorna i vari contatori e determina se il task deve essere *preempted*

perché è scaduto il suo quanto di tempo, e in tale caso invoca *resched*

# Pseudo-codice di *wait_event_interruptible*

```
void wait_event_interruptible (coda, condizione) {
    // costruisci un elemento delle waitqueue che punta al descrittore
    // del processo corrente
    // aggiungi il nuovo elemento alla waitqueue
    // poni il flag a indicare non esclusivo
    // poni stato del processo (current->state) ad ATTESA
    schedule ( );  // richiedi una commutazione di contesto
    // quando il processo riparte rimuovi il processo corrente
    // dalla waitqueue
} /* wait_event_interruptible */
```

nota: lo stesso pseudocodice vale anche per *void wait_event_interruptible_exclusive*, con l'unica differenza di porre il flag a indicare esclusivo

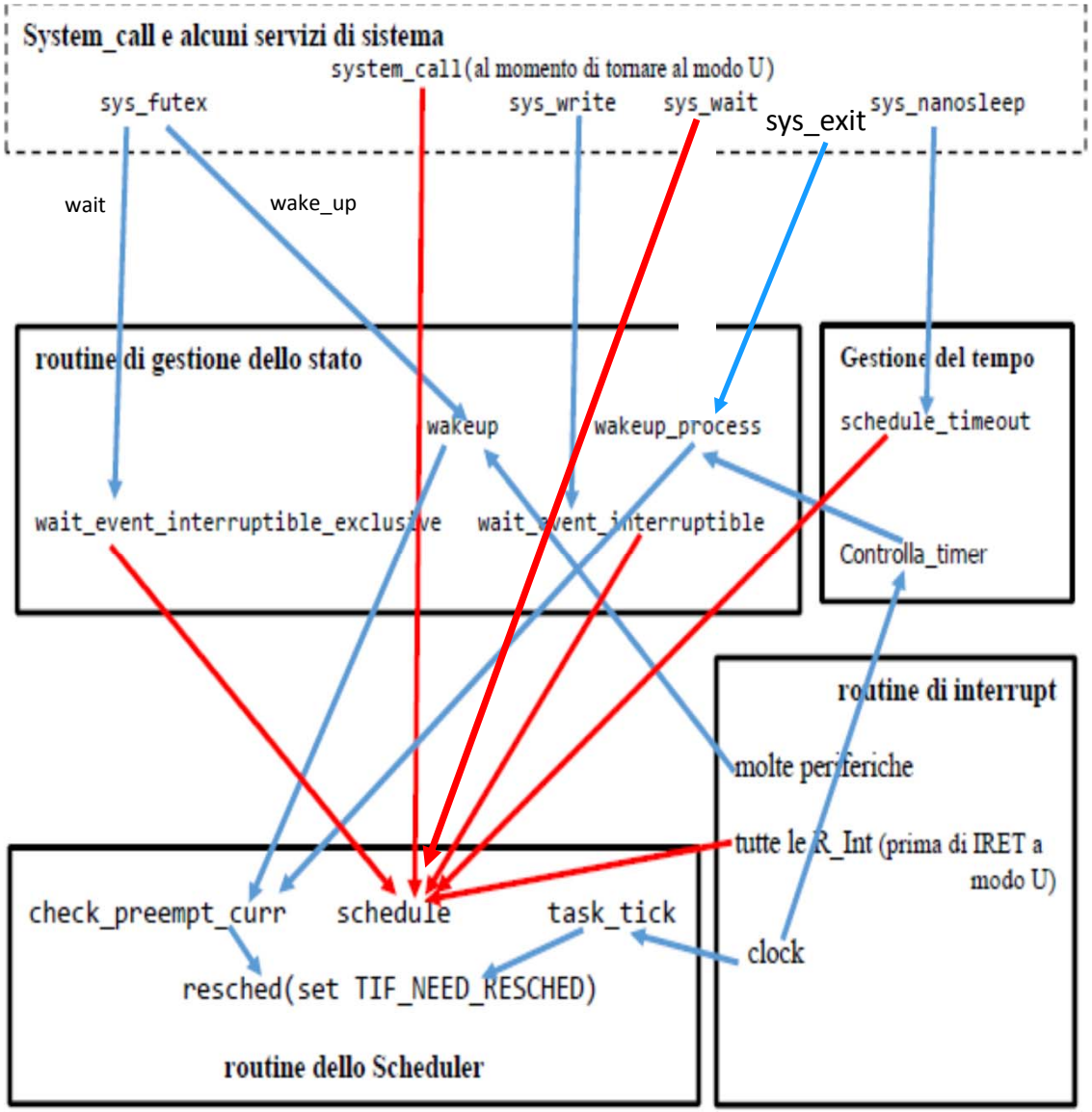
# Pseudo-codice di *wake_up*

```
void wake_up (wait_queue_head_t * wq) {
   for (per ogni descr. di proc. puntato da un elem. di wq) {
      // cambia lo stato del processo a PRONTO
      enqueue ( ); // inserisci il processo nella runqueue
      // elimina il processo dalla waitqueue
      // se flag indica esclusivo, esci subito dal ciclo
   } /* fine del ciclo sui descrittori di processo in wq */
   // verifica se è necessaria preemption
   check_preempt_curr ( );
} /* wake_up */
```

# Pseudo-codice della Routine di Interrupt *R_int_clock* ( )

```
void R_int_clock ( ) {              // viene attivata dallo interrupt di real time clock

        // gestisce i contatori di tempo reale (data, ora, ...) con periodicità opportuna

        task_tick ( );              // controlla se è scaduto il quanto di tempo del processo corrente

        controlla_timer ( );        // controlla la lista dei timeout con periodicità opportuna

        if (modo di rientro == U) schedule ( );

        IRET
} /* R_Int_clock */
```

# grafo delle funzioni di nucleo viste finora



# Codice Assemblatore nei Sorgenti di Linux

*APPROFONDIMENTO*

- esempio: commutazione di contesto
- la funzione *schedule* ( ) invoca la funzione *context_switch* ( ), che a sua volta contiene la macro in linguaggio assemblatore

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

- la lettura del codice di questa macro richiede di conoscere:
  - lo specifico linguaggio assemblatore dell'architettura hardware
  - le regole del *gnu inline assembler*, che serve a collegare le variabili C con gli indirizzi di memoria e i registri del processore
- l'operazione centrale consiste nella sostituzione della sPila del processo uscente dall'esecuzione con la sPila del processo entrante
- in *x64* si tratta delle due istruzioni macchina seguenti:

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

- dove il registro `rsp` è appunto il registro *SP* del processore

# Gestione della Concorrenza nel Nucleo

*APPROFONDIMENTO*

- cause di situazioni di concorrenza interna al nucleo:
  a. esistenza di molte *CPU* che eseguono in parallelo
  b. sospensione di un'attività a causa di una commutazione di contesto, con partenza di una nuova attività
- esempio di causa (b), 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* ( ) il sistema effettua la *preemption* del processo P, interrompendo *servizio1* ( ) in un momento arbitrario
  - parte un nuovo processo Q e richiede l'esecuzione dello stesso *servizio1* ( ), o di un altro *servizio2* ( ) che comunque interagisce con le stesse strutture dati usate da *servizio1* ( )
  - così si viene a creare 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

*APPROFONDIMENTO*

- la regola di *non-preemption* del nucleo riduce i problemi di esecuzione concorrente tra diverse funzioni del nucleo , perché:
  - l'autosospensione di un servizio avviene in maniera controllata
  - la commutazione di contesto avviene quando non ci sono attività di nucleo 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 nella commutazione di contesto

# Primitive di Sincronizzazione del Nucleo

*APPROFONDIMENTO*

- il *SO* Linux implementa il meccanismo di *mutex* mettendo il processo che trova una risorsa bloccata in stato di attesa su una *waitqueue*
- ma questo approccio non è usato per la maggior parte delle sincronizzazioni interne al *SO*, perché:
  - si tratta di attese molto brevi
  - e invece l'operazione di cambio di contesto è onerosa
- però il *SO* Linux implementa anche un diverso tipo di primitiva di *lock*:

lo *spinlock* basato su un *ciclo di attesa* (*busy waiting*)

- lo *spinlock* funziona così:
  - se il task non ottiene il *lock*, continua a tentare (*spinning*) fino a quando lo ottiene
  - gli *spinlock* sono molto piccoli e veloci e si possono utilizzare ovunque nel nucleo
- il difetto dello *spinlock* consiste pertanto nel fatto che impedisce di passare all'esecuzione di un altro thread fino al momento in cui:
  - è riuscito a ottenere il *lock*
  - oppure il thread che sta tentando di ottenerlo viene *preempted* dal *SO*
- questo meccanismo ha senso solo perché esistono i sistemi multiprocessore
- infatti nei sistemi mono-processore *non-preemptive* un processo che esegue in modo S non trova mai un *lock* occupato