Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

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

Parte N: Il Nucleo del Sistema Operativo

cap. N3 – Gestione dello stato dei processi

N.3 Gestione dello stato dei processi

1. Il modello fondamentale di gestione dei processi

In questo capitolo trattiamo il modello di gestione dei processi nei suoi aspetti fondamentali, senza tener conto dei molteplici aspetti di dettaglio che verranno poi arricchiti in capitoli successivi.

Stato dei processi.

Da un punto di vista generale 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.

Esecuzione di un processo - contesto

Normalmente il processore esegue il codice del processo corrente 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.

Normalmente un processo è in esecuzione in modo U; 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.

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

- quando un servizio di sistema richiesto dal processo deve porsi in attesa di un evento, esso abbandona esplicitamente lo stato di esecuzione passando 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 e quindi il servizio si pone in attesa dell'evento "arrivo del dato dal terminale del processo P". Si noti che *un processo si pone in stato di attesa quando è in esecuzione un servizio di sistema per suo conto*, e non quando è in esecuzione normale in modo U.
- 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

Il componente del Sistema Operativo che decide quale processo mettere in esecuzione è lo scheduler. Lo scheduler è un componente costituito da diverse funzioni.

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 in un capitolo successivo; al momento vogliamo analizzare solamente la funzione di Context Switch, che è svolta dalla funzione **`schedule( )`** dello scheduler.

Lo Scheduler gestisce una struttura dati fondamentale (per ogni CPU): la **`runqueue.L`**a `runqueue` contiene a sua volta 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 dei processi

Linux assegna ad ogni processo una sPila. Sul x64 la sPila allocata ad ogni processo ha una dimensione fissa di 2 pagine (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 che serve fondamentalmente a ricostruire lo stato al momento del ritorno al modo U.

Abbiamo già visto come il meccanismo HW permetta 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 e richiede di salvare i valori di SSP e USP durante la sospensione tra una esecuzione di un processo e la successiva. Per questo motivo il Descrittore di un Processo P contiene i seguenti 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 (ed è un valore relativo alla sPila perché una sospensione può avvenire solo quando il processo è in modo S)

La funzione di Context Switch gestisce SSP e USP nel modo seguente:

- quando il processo è in esecuzione in modo U, la sPila è vuota, quindi 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) USP contiene il valore corretto per il ritorno al modo U
- se, durante l'esecuzione in modo S, viene eseguita una commutazione di contesto, USP viene salvato sulla sPila di P e poi il valore corrente di SP viene salvato nel campo `sp` del descrittore di P
- quando P riprenderà l'esecuzione, lo Stack Pointer 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

Esempio 1
In questo esempio partiamo da una situazione iniziale (Figura 1) nella quale è in esecuzione un processo P, mentre un altro processo Q è in stato di PRONTO.

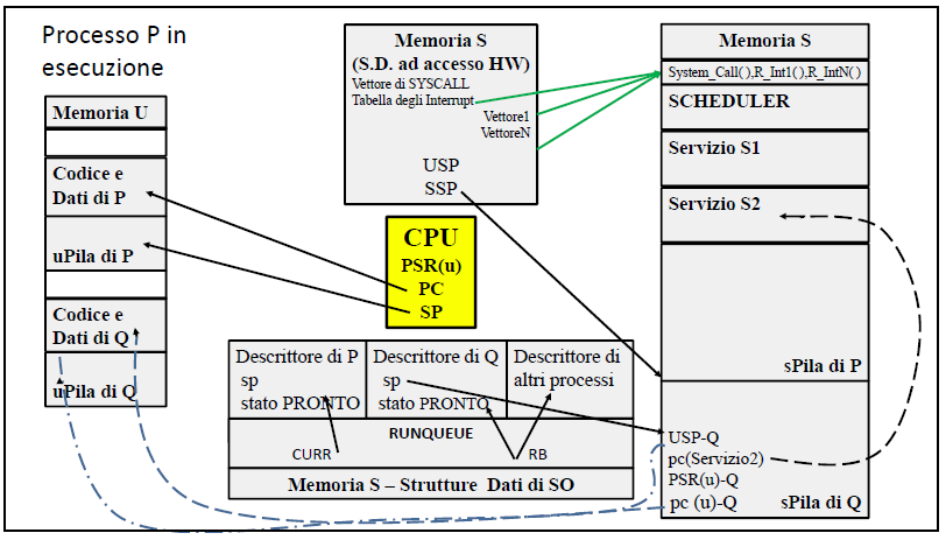

Figura 1

La figura mostra i componenti principali coinvolti:

- la CPU contiene i registri PC, SP e PSR, il cui contenuto è evidentemente quello relativo all'esecuzione di P
- Il registro SSP punta alla base della sPila di P; il registro USP non ha un valore utile

- la variabile CURR punta al descrittore di P
- la RUNQUEUE contiene i riferimenti a tutti gli altri processi PRONTI, tra cui il processo Q

E' importante analizzare le informazioni relative al processo Q, perché queste informazioni dovranno permettere di riprendere la sua esecuzione correttamente. Il processo Q è stato sospeso mentre eseguiva il servizio di sistema S2,

- nel descrittore di Q il campo `sp` punta alla cima della sPila di Q
- in tale sPila sono memorizzati:
  - il PC e il PSR per il ritorno al modo U
  - l'indirizzo di ritorno al Servizio S2; tale indirizzo è stato salvato al momento in cui il Servizio S2 ha invocato la funzione `schedule( )` per richiedere un context switch
  - la funzione `schedule( )` ha salvato sulla sPila di Q il valore di USP e poi ha salvato il valore del registro SP nel campo `sp` del descrittore di Q
  - poi la funzione `schedule( )` ha messo in esecuzione P caricando i registri SP, PC e PSR e ripristinando tramite `sp0` il valore della base di sPila di P.

Si tenga presente che durante il context switch che ha sospeso l'esecuzione di Q *il contesto HW che è stato salvato sulla pila di Q include anche tutti i registri del processore che si devono ritrovare integri al ritorno in esecuzione del processo Q*. Questa parte del contesto salvato non viene presa in considerazione nell'esempio.

In Figura 2 è mostrato l'effetto della esecuzione di una istruzione SYSCALL da parte del processo P.

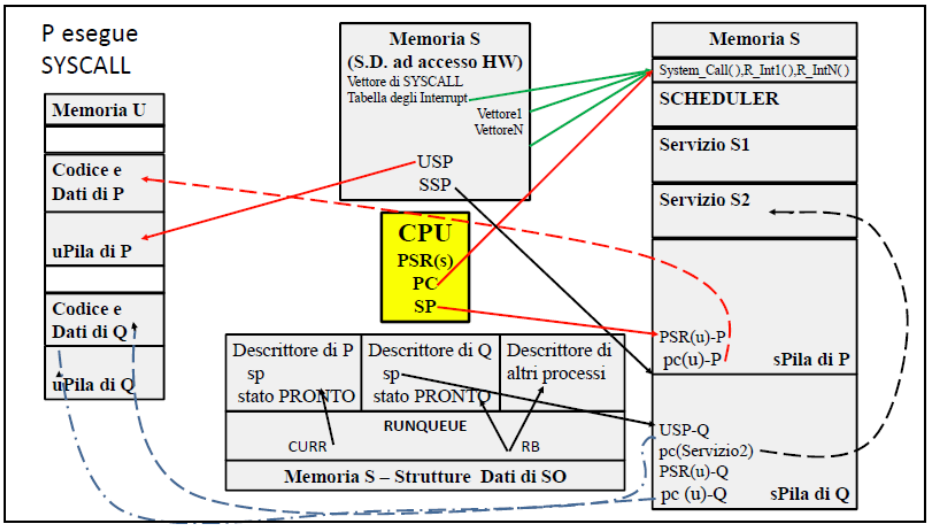

Figura 2

L'effetto è quello già visto analizzando il funzionamento dell'Hardware, con la precisazione che adesso la sPila e la uPila sono quelle del processo P.

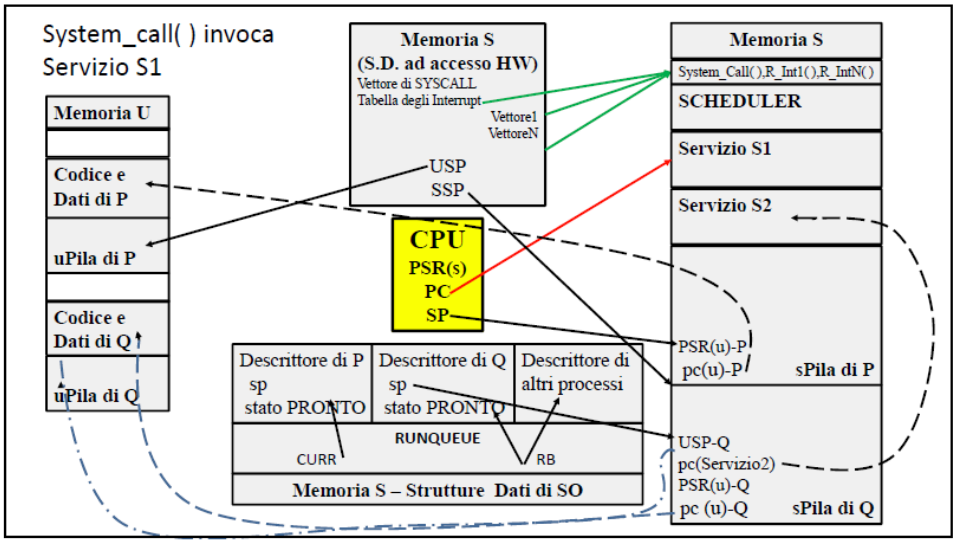

Figura 3

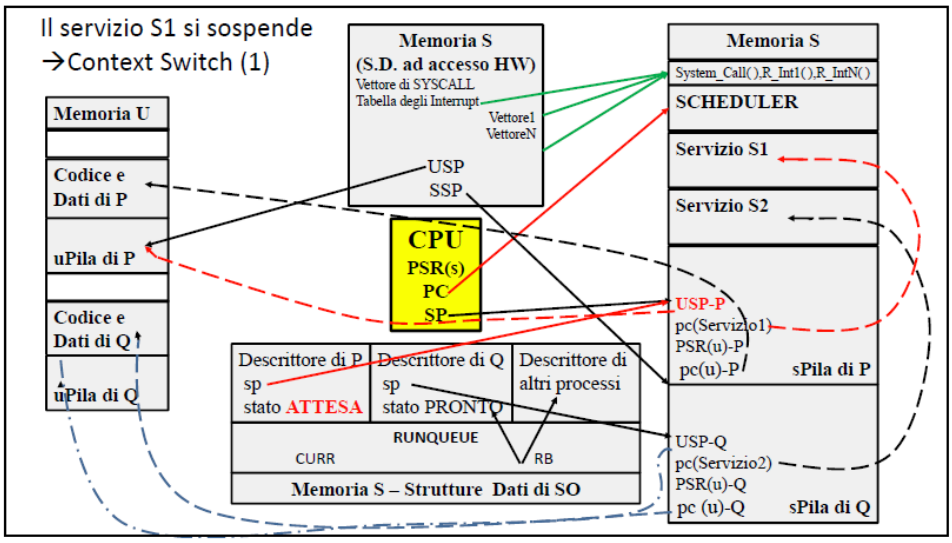

Figura 4

In Figura 3 la funzione `system_call()` ha invocato il servizio S1; l'indirizzo di ritorno salvato dalle invocazioni dei servizi e da successive chiamate di funzioni non viene rappresentato nelle figure per semplicità.

In Figura 4 è mostrata la situazione dopo che il servizio S1 ha invocato la funzione `schedule()` per richiedere una commutazione di contesto e tale funzione ha eseguito la prima parte delle operazioni, cioè *il salvataggio del contesto di P.*

pelagatti 9/12/2015 15:59
Deleted:

Sulla sPila di P è presente l'indirizzo di ritorno al servizio S1 (salvato automaticamente dall'invocazione di `schedule`()), poi `schedule`() ha svolto le seguenti operazioni:

- ha salvato USP sulla sPila
- ha posto lo stato di P al valore di ATTESA (infatti, vedremo più avanti che un servizio si sospende autonomamente solo per attendere un evento)
- ha tolto P dal puntatore CURR – vedremo più avanti come sono resi ritrovabili i processi in ATTESA
- ha salvato nel campo sp del descrittore di P il valore corrente del registro SP

A questo punto `schedule`() procede scegliendo un processo PRONTO e mettendolo in esecuzione; in Figura 5 è mostrato l'effetto di questa operazione ipotizzando che in base alla politica di scheduling e alla situazione dei processi pronti il processo scelto sia Q:

- `CURR` punta al descrittore di Q, che è stato rimosso dalla `RUNQUEUE`
- nel registro SP è stato posto il valore presente nel campo `sp` del descrittore di Q; questa operazione costituisce il momento centrale della commutazione di contesto, perché la sPila in uso non è più quella del processo precedente (P), ma quella del processo successivo (Q).

pelagatti 9/12/2015 16:02
Deleted: .

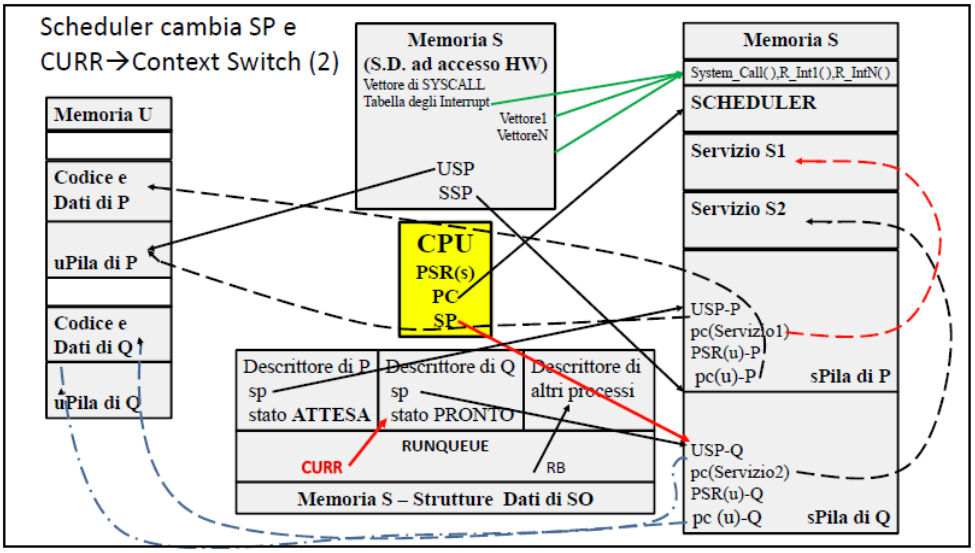

Figura 5

A questo punto lo scheduler ripristina il valore di SSP che deve ora puntare alla base della Spila di Q e può copiare il valore USP-Q presente sulla sPila in USP ed eseguire il ritorno da funzione; dato che sulla sPila è presente il valore salvato al momento in cui il servizio S2 (eseguito per conto di Q) aveva invocato lo scheduler, il ritorno è a tale servizio (Figura 6).

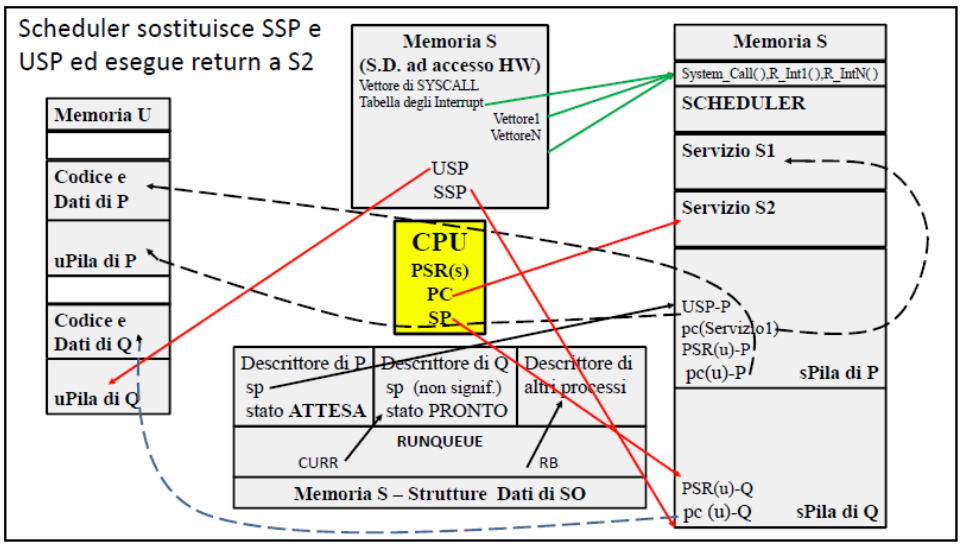

Figura 6

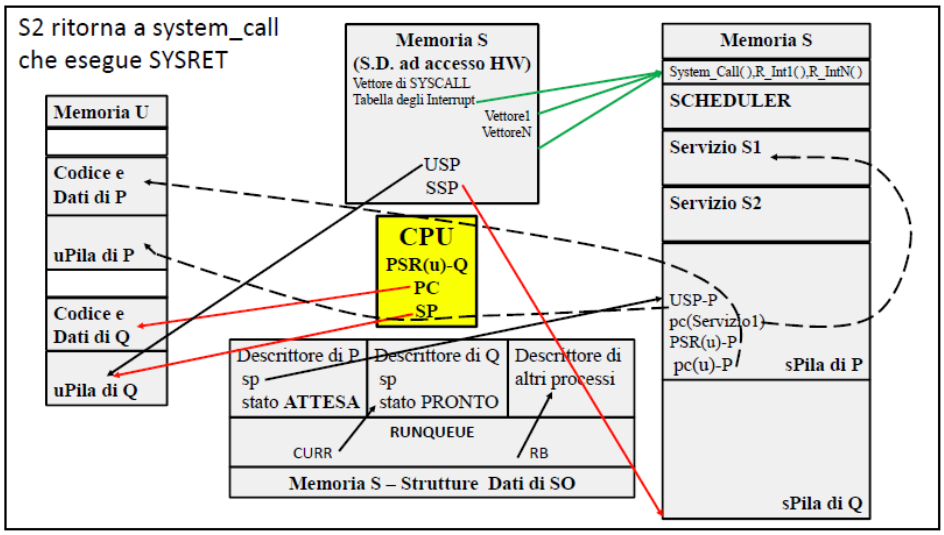

Figura 7

Il servizio S2 può terminare la sua esecuzione rientrando in system_call ( ) ed eseguendo l'istruzione SYSRET (Figura 7).

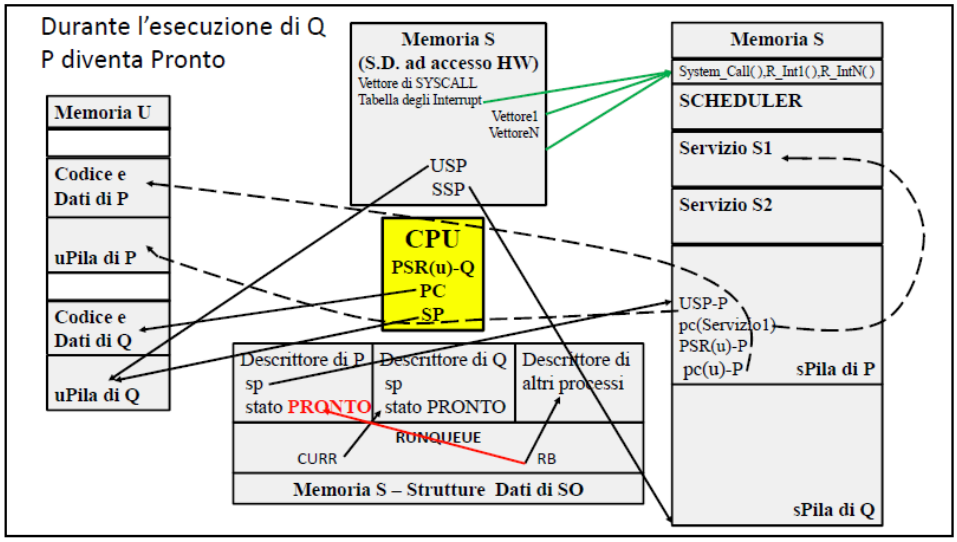

Figura 8

Adesso è in esecuzione il processo Q in modalità U; durante tale attività è possibile che si verifichi l'evento atteso dal servizio S1 (per conto del processo P); in tal caso lo stato di P viene cambiato da ATTESA a PRONTO e il descrittore di P viene inserito nuovamente nella RUNQUEUE (Figura 8). Si noti che lo stato rappresentato in figura 8 è identico a quello di figura 1, scambiando i processi P e Q. Questo fatto dimostra che lo stato iniziale ipotizzato in Figura 1 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 nel seguito, in particolare:

- 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

2. La gestione degli Interrupt

La gestione degli interrupt segue i seguenti principi:

- quando si verifica un interrupt esiste sempre un processo in stato di esecuzione. Tuttavia la situazione può appartenere ad uno dei 3 seguenti sottocasi:
  - 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;
- in tutti questi casi il SO, cioè la routine di interrupt, svolge la propria funzione senza disturbare il processo in esecuzione (la routine di interrupt è trasparente) e *non viene mai sostituito il processo in esecuzione (cioè non viene mai svolta una commutazione di contesto) durante l'esecuzione di un interrupt*; si dice che gli interrupt vengono 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 (ovviamente diverso dal processo in esecuzione), la routine di interrupt risveglia il processo P passandolo dallo stato di attesa allo stato di pronto, in modo che successivamente il processo P possa tornare in esecuzione. Ad esempio, se il processo P era in attesa di un dato dal terminale, la routine di interrupt associata al terminale del processo P risveglia tale processo quando un interrupt segnala l'arrivo

del dato. Si osservi che *la routine di interrupt è associata ad un evento atteso dal processo P ma si svolge nel contesto di un diverso processo*. Questo aspetto risulta molto importante nella corretta progettazione delle routine di interrupt.

La gestione degli interrupt verrà approfondita nel capitolo relativo all'Input/Output.

3. La gestione dei cambiamenti di stato ATTESA/PRONTO e viceversa – modello fondamentale

Il problema fondamentale da affrontare nella gestione dello stato di ATTESA di un evento E è costituito dalla necessità di supportare il passaggio allo stato di PRONTO (detto risveglio o ***wakeup***) quando l'evento E si verifica. Si tenga presente che a un certo momento possono esistere più di un processo in attesa di uno stesso evento (ad esempio, diversi processi in attesa dello sblocco dello stesso Lock).

Tale supporto è reso più complesso dal fatto che 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)

Wait_queue

Una `waitqueue` è una lista contenente i 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; in tale coda vengono inseriti i descrittori dei processi posti in attesa dell'evento. L'indirizzo della `waitqueue` costituisce l'identificatore dell'evento.

La coda può essere definita staticamente come variabile globale tramite la *macro* seguente (esiste anche un modo di definizione dinamica, che trascuriamo):

```
DECLARE_WAIT_QUEUE_HEAD  nome_coda
```

Il Nucleo mette a disposizione delle altre funzioni del SO numerose funzioni per mettere correttamente in stato di ATTESA un processo; l'esistenza di numerose funzioni è dovuta alla varietà di modalità di attesa. Tutte queste funzioni richiedono l'indicazione di 2 parametri:

- la coda
- una condizione (serve per evitare problemi di concorrenza, cioè che la condizione di attesa si verifichi durante la sospensione del processo, sospendendolo per sempre – in altri termini, serve per rendere atomica l'operazione di messa in attesa)

Nel seguito vedremo solo alcune di queste funzioni, trascurandone molte altre.

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, se si risvegliassero tutti quando la risorsa diventa disponibile solo uno potrebbe acquisire la risorsa e gli altri dovrebbero tornare in attesa).

I processi per i quali deve esserne risvegliato uno solo sono detti in attesa **esclusiva**.

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

In questo modo la routine di risveglio può operare nel modo seguente: risveglia tutti i processi dall'inizio della lista fino (incluso) al primo processo in attesa esclusiva.

Per mettere un processo in attesa non esclusiva la funzione più usata è la seguente:

`wait_event_interruptible()`, invece per metterlo in attesa esclusiva:

`wait_event_interruptible_exclusive( )`

Il motivo del termine "interruptible" verrà spiegato più avanti.

Esempio:

```
DECLARE_WAIT_QUEUE_HEAD  coda_della_periferica
wait_event_interruptible(coda_della_periferica, buffer_vuoto ==  1);
```

Wakeup

La funzione fornita dal nucleo per risvegliare i processi in attesa su una coda è

`wake_up(wait_queue_head_t * wq)`,

che risveglia tutti i task in attesa non-esclusiva e un solo task in attesa esclusiva sulla coda wq, svolgendo per ogni task 2 operazioni:

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

Risvegliando dei processi `wakeup` può creare una situazione in cui il processo corrente in esecuzione dovrebbe essere sostituito da uno nuovo, appena risvegliato e dotato di maggiori diritti di esecuzione. `Wakeup` non invoca però mai direttamente schedule; se il nuovo task aggiunto alla `runqueue` ha diritto di sostituire quello corrente, `wakeup` pone a uno il flag `TIF_NEED_RESCHED`. La funzione schedule verrà invocata alla prima occasione possibile.

4. 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. Ogni `signal` è identificato da un numero, da 1 a 31, e 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 *bloccata* dal processo; un `signal` bloccato rimane pendente fina a quando non viene sbloccato.

Esistono 2 `signal` che non possono essere bloccati dal processo:

- `SIGKILL` – termina immediatamente il processo
- `SIGSTOP` – sospende il processo (per riprenderlo più tardi)

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 oppure bloccarlo

Talvolta un `signal` viene inviato a un processo che non è in esecuzione in modo U; in questi casi le azioni si svolgono nel modo seguente:

- se il `signal` viene inviato a un processo che esegue in modo S, viene processato immediatamente al ritorno al modo U
- se il `signal` viene inviato a un processo pronto ma non in esecuzione, viene tenuto in sospeso finché il processo torna in esecuzione.
- se il `signal` viene inviato a un processo in stato di attesa, ci sono 2 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

Si osservi che nel caso di attesa interrompibile il processo può essere risvegliato senza che l'evento su cui era in attesa si sia verificato; pertanto il processo deve controllare al risveglio se la condizione di attesa è diventata falsa e, in caso contrario, rimettersi in attesa.

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** viste in precedenza (quindi lo stato di ATTESA coincide con `TASK_INTERRUPTIBLE`).

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

Il meccanismo di attesa su una `waitqueue` è usato molto nei gestori di periferica. La trattazione dei gestori di periferica verrà svolta nell'ambito dell'Input/Output; qui vogliamo solamente analizzare come le funzioni e i meccanismi di gestione dello stato possano essere utilizzati da un gestore di periferica per svolgere ad esempio una scrittura di dati sulla 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. Tale funzione viene attivata nel contesto del processo che ne ha richiesto il servizio, e quindi l'eventuale trasferimento di dati tra il gestore e il processo non pone particolari problemi. Tuttavia, dato che le periferiche operano normalmente tramite interrupt, 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.

L'esecuzione dell'operazione è descritta nel seguito con riferimento allo schema di figura 9. Il buffer che compare in tale figura è una zona di memoria appartenente al gestore stesso, non è un buffer generale di sistema. Supponiamo che tale buffer abbia una dimensione di B caratteri. Le operazioni svolte saranno le seguenti:

1. Il processo richiede il servizio tramite una funzione *write( )* e il sistema attiva la routine *X.write()* del gestore;
2. la routine *X.write* del gestore copia dallo spazio del processo nel buffer del gestore un certo numero C di caratteri; C sarà il minimo tra la dimensione B del buffer e il numero N di caratteri dei quali è richiesta la scrittura;
3. la routine *X.write* manda il primo carattere alla periferica con un'istruzione opportuna;
4. a questo punto la routine *X.write* non può proseguire, ma deve attendere che la stampante abbia stampato il carattere e sia pronta a ricevere il prossimo; per non bloccare tutto il sistema, *X.write* pone il processo in attesa creando una `waitqueue` WQ e invocando la funzione `wait_event_interruptible(WQ, buffer_vuoto)` – dove buffer_vuoto è una variabile booleana messa a true ogni volta che il buffer è vuoto
5. quando la periferica ha terminato l'operazione, genera un interrupt;
6. la routine di interrupt del gestore viene automaticamente messa in esecuzione; se nel buffer esistono altri caratteri da stampare, la routine di interrupt esegue un'istruzione opportuna per inviare il prossimo carattere alla stampante e termina (IRET);
   altrimenti, il buffer è vuoto e si passa al prossimo punto;
7. la routine di interrupt, dato che il buffer è vuoto, risveglia il processo invocando la funzione `wake_up(WQ)`, cioè passandole lo stesso identificatore di `waitqueue` che era stato passato precedentemente alla `wait_event_interruptible`;
8. `wake_up` ha risvegliato il processo ponendolo in stato di pronto; prima o poi il processo verrà posto in esecuzione e riprenderà dalla routine *X.write* che si era sospesa tramite `wait_event_interruptible`; se esistono altri caratteri che devono essere scritti, cioè se N è maggiore del numero di caratteri già trasferiti nel buffer, *X.write* copia nuovi caratteri e torna al passo 2, ponendosi in attesa, altrimenti procede al passo 9;
9. prima o poi si deve arrivare a questo passo, cioè i caratteri già trasferiti raggiungono il valore N e quindi il servizio richiesto è stato completamente eseguito; la routine *X.write* del gestore esegue il ritorno al processo che la aveva invocata, cioè al modo U

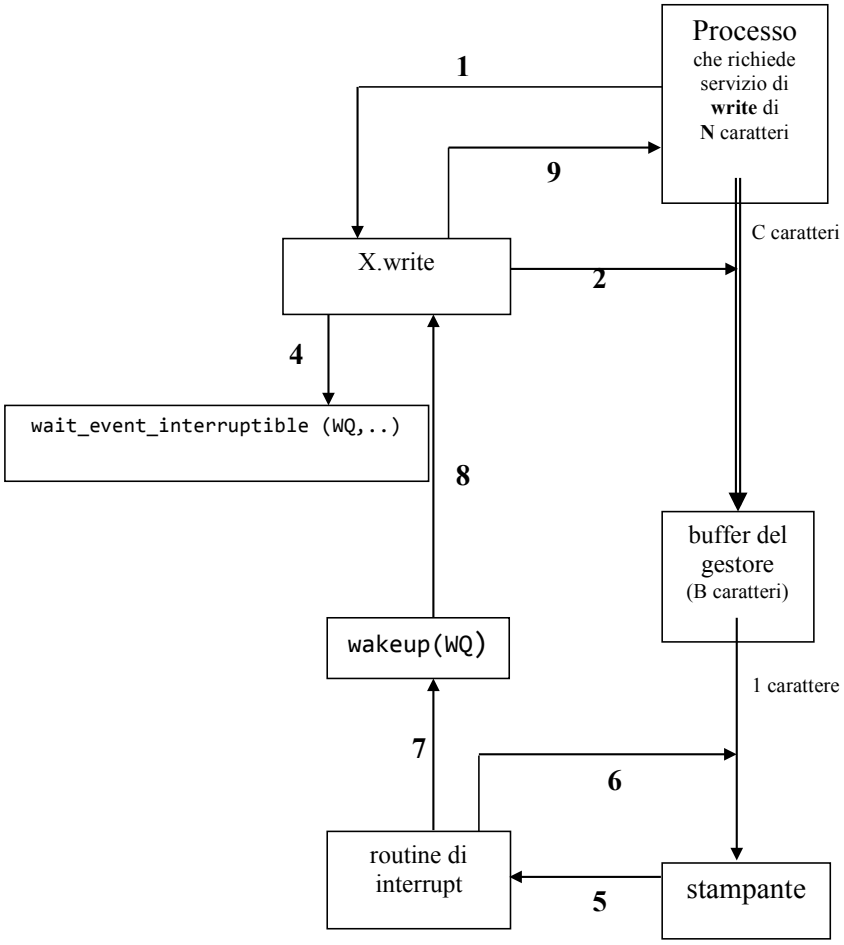
Processo
che richiede
servizio di
write di
N caratteri
1
9
C caratteri
X.write
2
4
wait_event_interruptible (WQ,..)
8
buffer del
gestore
(B caratteri)
wakeup(WQ)
1 carattere
7
6
routine di
interrupt
5
stampante

Figura 9

6. Esempio di attesa esclusiva: implementazione dei MUTEX

Quando un programma applicativo in modo U esegue un'operazione di lock su un mutex già bloccato il suo processo deve essere posto in stato di ATTESA. Dato che il cambiamento di stato deve avvenire nel SO, l'esecuzione di ogni operazione di lock richiederebbe di passare al modo S tramite una system call. Per evitare l'onere di eseguire una system call anche quando il mutex è libero e non è necessaria alcuna attesa, Linux utilizza il meccanismo dei **`futex`** (**Fast Userspace Mutex**); sopra questo meccanismo la libreria di modo U può realizzare i Mutex in maniera efficiente.

Un Futex ha 2 componenti:

1. una variabile intera nello spazio U
2. una `waitqueue` WQ nello spazio S

L'incremento e il test della variabile intera sono svolte in maniera atomica in spazio U; se il lock può essere acquisito l'operazione ritorna senza alcun bisogno di invocare il SO. Solo se il lock è bloccato è necessario invocare una system call, `sys_futex(...).`

La system call `sys_futex(int op, ...)` possiede un parametro `op` (operazione richiesta) che può assumere i seguenti valori: `FUTEX_WAIT` oppure `FUTEX_WAKE`; la prima operazione serve a richiedere che il processo venga posto in attesa, la seconda per richiedere che venga risvegliato.

L'operazione di lock di un mutex già bloccato si realizza invocando `sys_futex(FUTEX_WAIT, …),` che 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'operazione di unlock del mutex invoca invece `sys_futex(FUTEX_WAKE,…)` risvegliando i processi presenti nella `waiqueue` tramite l'invocazione di `wakeup(WQ)`. Dato che potrebbero esserci N processi in attesa di accesso al Futex, i quali appena risvegliati tenterebbero tutti di eseguire il lock, causando N risvegli ed N-1 ritorni in stato di attesa, l'implementazione dei Futex costituisce un esempio di uso conveniente dell'attesa esclusiva, che causerà il risveglio di uno solo degli N processi.

N.B. La realizzazione in maniera atomica dell'incremento e il test della variabile intera non può ovviamente utilizzare i Mutex, perché siamo nella implementazione dei Mutex; x64 fornisce un'istruzione apposita per realizzare operazioni atomiche di questo tipo (cfr. Patterson Hennessy 2.11) – su HW privo di questa caratteristica deve essere utilizzata una realizzazione basata su un algoritmo come quello visto nel capitolo relativo alla gestione della concorrenza.

7. La preemption

In diverse situazioni il sistema scopre che un task in esecuzione dovrebbe essere sospeso. La regola base del kernel non preemptive è*: durante il funzionamento in modo S non viene mai eseguita una commutazione di contesto, quindi sPila non cambia*.

L'assenza di preemption del sistema semplifica molti aspetti nella sua realizzazione; l'uso della sPila del processo corrente per gli interrupt è incompatibile con la preemption – casa accadrebbe se si cambiasse contesto, quindi sPila, quando ci sono routine di interrupt annidate in esecuzione? Inoltre nei sistemi monoprocessore non-preemptive un servizio di sistema non trova mai un lock bloccato da un altro servizio, perchè un servizio non può essere preempted durante un'operazione critica.

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, 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. Il motivo della violazione è dovuto al fatto che il SO può decidere di eseguire una preemption solo quando è effettivamente in esecuzione, quindi in modo S. E' evidente che, per imporre la preemption il SO deve attuarla prima di ritornare al modo U, perché il processo in modo U esegue semplicemente il suo programma, senza nessun motivo per autosospendersi.

A causa dell'impossibilità di osservare letteralmente la regola indicata, 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, formulando la regola con riferimento alle istruzioni che possono eseguire un ritorno al modo U: *"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.

8. Altri tipi di attesa – attesa di exit e attesa di un timeout

Il meccanismo di attesa basato sulla creazione di una `waitqueue` è conveniente nelle situazioni in cui la funzione che scoprirà il verificarsi dell'evento atteso conosce la coda relativa all'evento; abbiamo visto che questo è il caso ad esempio in un driver di periferica o nell'implementazione dei `futex`, dove la funzione che pone in attesa e quella che risveglia sono fortemente correlate.

Esistono però altre 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 `wakeup` che riceve come argomento direttamente un puntatore al descrittore del processo da risvegliare: `wakeup_process(task_struct * processo)`.

Due esempi di questo approccio sono i seguenti:

- quando un processo P esegue *exit( )* e deve essere risvegliato il relativo processo padre, se ha eseguito una *wait( )*; dato che P possiede nel descrittore un puntatore `parent_ptr` al proprio processo padre, è sufficiente che *exit ( )* invochi `wakeup_process(parent_ptr)`.
- quando un processo deve essere risvegliato a un preciso istante di tempo

Gestione di timeout

Un timeout definisce una scadenza temporale; la definizione di un timeout è quindi la definizione di un particolare istante nel futuro. 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` (Jiffi è un termine informale per un tempo molto breve); questa variabile 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`.

Ad esempio, il servizio `sys_nanosleep(timespec t)` definisce una scadenza posta un tempo t dopo la sua invocazione e pone il processo in stato di ATTESA fino a tale scadenza.

Quando questo servizio viene invocato, svolge le seguenti azioni:

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

La funzione `timespec_to_jiffies(timespec * t)` converte t dalla rappresentazione esterna ai jiiffies.

Il codice (semplificato) di `schedule_timeout` è il seguente

```
schedule_timeout(timeout t){
      struct timer_list timer; //definisce un elemento timer
      init_timer(&timer)
      timer. expire = timeout + 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
}
```

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 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 expire.

Quando il timer scade, viene eseguita la funzione `timer.function` passandole `timer.data` come parametro, quindi in questo caso

```
wakeup_process( (timer.data);
```

cioè risveglia il processo che aveva invocato il servizio.

9. Riassunto delle transizioni di stato

In Figura 10 sono riportati gli stati di un processo e le transizioni di stato possibili; in tale figura lo stato di esecuzione è stato diviso in 2: lo stato di esecuzione normale in modo U e lo stato di esecuzione in modo S, cioè lo stato in cui non viene eseguito il programma del processo ma un servizio o una routine di interrupt nel contesto del processo stesso.

Ovviamente, ad un certo istante un solo processo per CPU può essere in esecuzione (in modo U oppure S), ma molti processi possono essere pronti o in attesa di eventi. Nella figura 10 sono anche indicate le principali cause di transizione tra gli stati. Quando un processo è in esecuzione in modo U, le uniche cause possibili di cambiamento di stato sono gli interrupt o l'esecuzione di una SYSCALL, che lo fanno passare all'esecuzione in modo S (transizione 1).

Nel più semplice dei casi viene eseguita una funzione del SO che termina con un'istruzione IRET o SYSRET che riporta il processo al modo U (transizione 2). Durante l'esecuzione in modo S possono verificarsi interrupt annidati a maggior priorità; questi interrupt vengono eseguiti restando in modo S e nel contesto dello stesso processo (transizione 3). Durante questi interrupt non viene mai eseguita una commutazione di contesto, in base alla regola già vista.

L'abbandono dello stato di esecuzione può avvenire solo dal modo S per uno di due motivi: un servizio richiede qualche tipo di `wait_xxx` (transizione 4) oppure durante un servizio di sistema o un interrupt di primo livello, cioè un interrupt che ha causato la transizione 1 e non la 3, si verifica che è scaduto il quanto e la funzione schedule esegue una commutazione di contesto (transizione 5).

La ripresa dell'esecuzione di un processo (transizione 6) avviene se il processo è pronto, cioè non è in attesa, e se è quello avente maggior diritto all'esecuzione tra tutti quelli pronti. Si osservi che il momento in cui avviene la transizione 6 di un processo P non dipende da P stesso, ma dal processo in esecuzione, che starà eseguendo una transizione di tipo 4 oppure 5, e dall'algoritmo di selezione del processo pronto con maggior diritto all'esecuzione.

Infine, il risveglio di un processo tramite la funzione `wakeup` (transizione 7) avviene quando, nel contesto di un altro processo, si verifica un interrupt che determina l'evento sul quale il processo era in attesa.

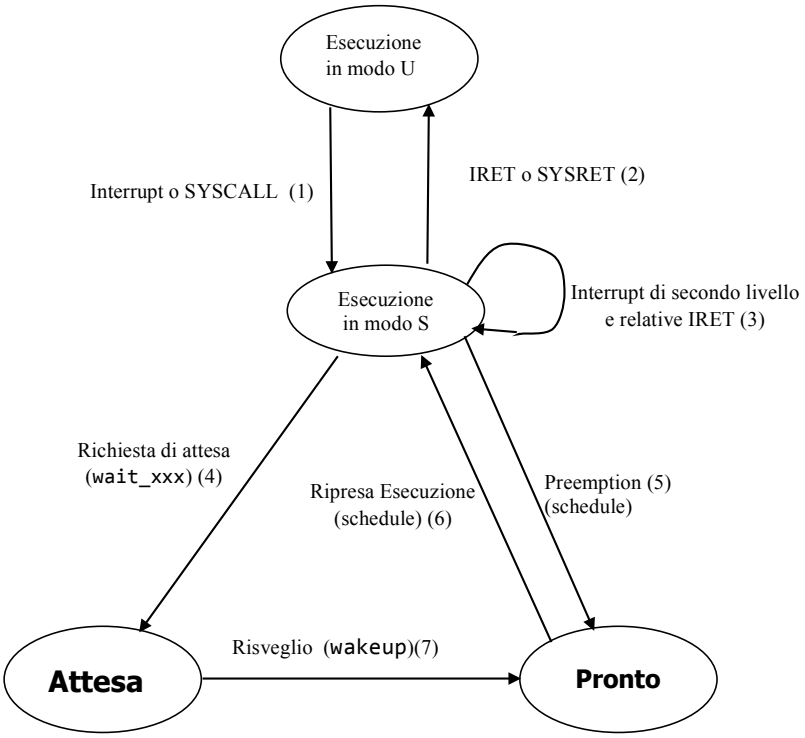
Esecuzione
in modo U
Interrupt o SYSCALL (1)
IRET o SYSRET (2)
Esecuzione
in modo S
Interrupt di secondo livello
e relative IRET (3)
Richiesta di attesa
(wait_xxx) (4)
Ripresa Esecuzione
(schedule) (6)
Preemption (5)
(schedule)
Risveglio (wakeup)(7)
Attesa
Pronto

Figura 10

10. Le funzioni dello Scheduler

Le funzioni del nucleo che gestiscono lo stato interagiscono con le funzioni dello Scheduler. Dato che lo Scheduler e le politiche che implementa saranno trattate in un capitolo successivo, qui vengono indicate per alcune funzioni dello Scheduler le operazioni utili al fine di comprendere i meccanismi della gestione dello stato dei processi – in generale queste funzioni operano anche sui dati che determinano i diritti di esecuzione dei processi.

Le funzioni utilizzate dalle routine esterne allo Scheduler sono:

- `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)
- `enqueue_task()`
  - inserisce il task nella runqueue
- `dequeue_task()`
  - elimina il task dalla runqueue
- `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).

Queste routine sono invocate dalle routine di wait_xxx e wakeup come indicato nello pseudocodice di figura 11 e 12.

```
void wait_event_interruptible(coda, condizione) {
    //costruisci un elemento delle waitqueueu che punta al descrittore del processo corrente
    //aggiungi il nuovo elemento alla waitqueue
    //poni i flag a indicare Non Esclusivo
    //poni stato del processo (current->state) a ATTESA
    schedule( );        //richiedi un context switch;

}

void wait_event_interruptible_exclusive(coda, condizione) è simile, ma i flag indicano Exclusivo
```

Figura11 - Pseudocodice di wait_event_interruptible_XXX

```
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 waitqueueu
              //se flag indica esclusivo, break
       }

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

Figura 12 – pseudocodice di wakeup

pelagatti 9/12/2015 15:44
Deleted:

pelagatti 9/12/2015 15:44
Deleted:

In Figura 13 è riportato lo pseudocodice della routine di interrupt del clock. Lo statement

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

è presente in tutte le routine di interrupt prima dell'istruzione IRET.

```
void R_int_clock(… )
{  // attivata dall'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
   //con periodicità opportuna
            Controlla_timer( );        //controlla lista dei timeout

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

    IRET
}
```

Figura 13 – pseudocodice routine di interrupt del clock

In Figura 14 è riportata una mappa delle funzioni viste in questo capitolo con l'indicazione delle chiamate. Sono evidenziate le chiamate alla funzione schedule( ), cioè le invocazioni di un context switch, che avviene nei casi seguenti:

- da parte di system_call: subito prima di eseguire SYSRET, se il ritorno è al modo U
- da parte di tutte le routine di interrupt: subito prima di eseguire IRET, se il ritorno è al modo U
- da parte di tutte le wait_event_xxx, perché il processo corrente si sospende

da parte di schedule_timeout, perché il processo corrente si sospende

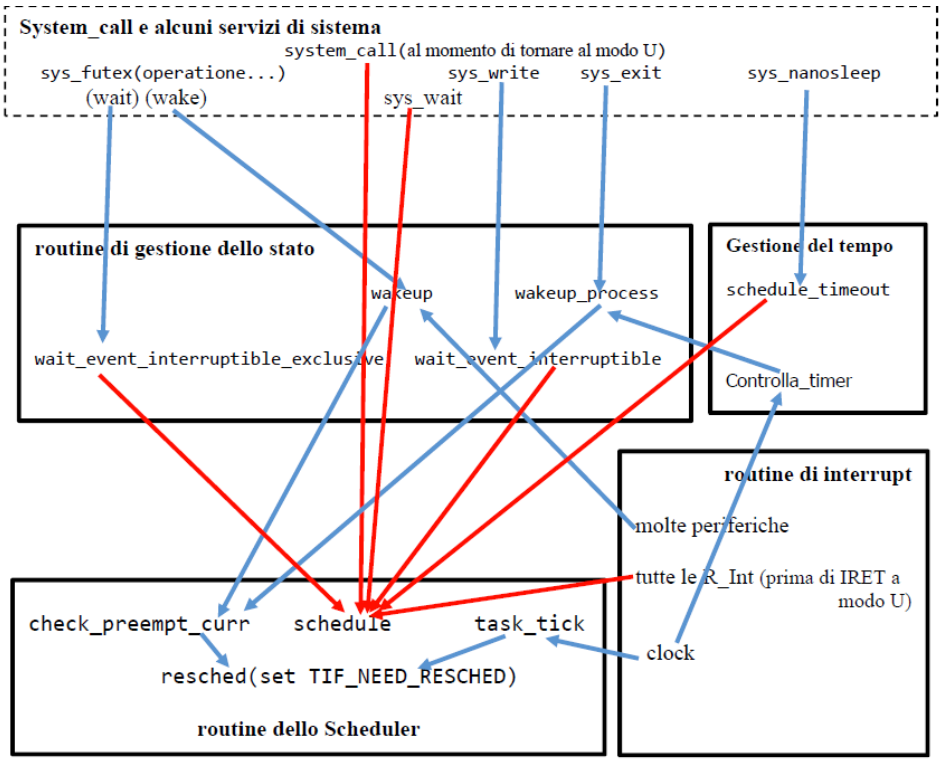

Figura 14 – mappa delle funzioni trattate in questo capitolo

L'invocazione di `resched( )` per settare `TIF_NEED_RESCHED`, indicando che è necessaria una preemption, è svolta nei casi seguenti:

- quando `task_tick`, invocata dal clock, scopre che è scaduto il quanto di tempo
- indirettamente attraverso `check_preempt_curr` quando `wakeup` o `wakeup_process` risvegliano dei processi che erano in attesa

11. Approfondimenti

11.1 Necessità di codice assembler per la commutazione di contesto

La commutazione di contesto è un'operazione che richiede di utilizzare codice assembler in diversi punti. Ad esempio consideriamo l'operazione centrale della commutazione di contesto, cioè la sostituzione della sPila del processo uscente con la sPila del processo nuovo: questa operazione non può essere realizzata da codice C puro.

La funzione `schedule( ),` definita nel file `kernel/sched/core.c,` invoca la funzione `context_switch(),` che a sua volta contiene la macro assembler

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

Il codice di questa macro è di difficile lettura, perché richiede non solo di conoscere lo specifico assembler, ma anche di conoscere le regole del "*gnu inline assembler*", che serve a collegare le variabili C con gli indirizzi di memoria e i registri assembler.

Il nucleo centrale della macro `switch_to(prev, next)` consiste nella sostituzione della sPila del processo uscente con la sPila del processo nuovo. Nel x64 si tratta delle 2 seguenti istruzioni (in assembler gnu GAS):

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

dove:

1. movq sorgente, destinazione è una istruzione a 2 operandi che copia una parola da 64 bit (q = quadword) dalla sorgente alla destinazione
2. %rsp è il registro Stack Pointer
3. threadrsp è una costante che rappresenta la distanza (offset) tra l'inizio del descrittore di un processo e il campo che contiene il valore del puntatore alla sPila di quel processo
4. %next e %prev indicano 2 registri che devono contenere l'indirizzo dei descrittori del processo nuovo (next) da mandare in esecuzione e del processo uscente (prev) da sospendere
5. i 2 modi di indirizzamento utilizzati sono identici ai corrispondenti modi del calcolatore MIPS (registro e base+registro)

Le 2 istruzioni quindi salvano il valore corrente dello SP nel descrittore del processo uscente e caricano dal descrittore del processo entrante il nuovo valore di SP. Dopo l'esecuzione di queste 2 istruzioni ogni accesso alla sPila farà riferimento alla sPila del processo nuovo – possiamo quindi dire che l'esecuzione di queste 2 istruzioni assembler costituisce il momento fondamentale della commutazione di contesto.

11.2 Gestione della concorrenza nel nucleo

Le diverse routine del nucleo operano su strutture dati condivise e possono essere eseguite in parallelo per i seguenti motivi, generando problemi di esecuzione concorrente tra diverse funzioni del Kernel:

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

Ad esempio supponiamo che

- 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 2 thread che eseguono una funzione in parallelo, con tutti i problemi legati alla concorrenza

La regole 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 e 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, quindi il controllo della commutazione di contesto rende impossibile l'esistenza di 2 servizi arbitrari in esecuzione concorrente, semplificando molto i problemi di concorrenza.
Il passaggio ai sistemi multiprocessore ha invalidato questo meccanismo, obbligando ad arricchire le funzioni del Kernel con meccanismi di sincronizzazione opportuni (tipicamente lock). Tuttavia Linux rimane un sistema non-preemptive (almeno nella sua versione standard), anche perchè il Kernel non-preemptable oltre a risolvere il problema della sincronizzazione al suo interno presenta l'ulteriore vantaggio di rendere più semplice il salvataggio e il ripristino dello stato di un processo nelle commutazioni di contesto.

Primitive di sincronizzazione interne al nucleo
Abbiamo visto che Linux implementa i Mutex mettendo il processo che trova una risorsa bloccata in stato di attesa su una waitqueue. Questo approccio è coerente con tutta la logica della gestione dei processi.
Tuttavia questo approccio non è utilizzato per la maggior parte delle sincronizzazioni interne al SO, perchè si tratta di attese molto brevi, mentre l'operazione di cambio di contesto è onerosa, quindi se l'attesa per il lock è molto breve, può non essere conveniente l'uso dei mutex.
Per le attese brevi Linux implementa un diverso tipo di primitive di lock: gli spinlock (`include/asm/spinlock.h`), 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 viene preempted dal SO. Si osservi che 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: è sufficiente che non si autosospenda mai in mezzo a una sequenza critica e che gli interrupt non accedano a strutture condivise.

Problemi di transizione
La funzione Linux equivalente a `wait_event_xxx` nel sistema originario ed è ancora presente per ragioni di compatibilità si chiama `sleep_on(queue).` L'uso di questa funzione è sconsigliato da quando Linux supporta i multiprocessori, perché può causare problemi di concorrenza se, nel momento in cui il processo viene posto in stato di attesa, concorrentemente la condizione di attesa si verifica; in questo caso il processo finirebbe in una `waitqueue` in attesa di un evento che in realtà si è già verificato.

11.3 Altri stati oltre ATTESA e PRONTO
Esistono alcuni altri stati che servono in alcune situazioni particolari (`TASK_STOPPED, EXIT_ZOMBIE, EXIT_DEAD`). Questi stati servono a gestire situazioni particolari, legate alla terminazione dei processi e alle operazioni di wait da parte del padre.
Nei descrittori esistono dei puntatori che collegano i descrittori dei processi figli al padre. I Descrittori dei processi in stato di `TASK_STOPPED, EXIT_ZOMBIE, EXIT_DEAD` non appartengono a nessuna `runqueueu o waitqueue e` vengono acceduti solo tramite PID oppure attraverso le liste che collegano i processi figli al processo padre.