6. IL NUCLEO DEL SISTEMA OPERATIVO

1. Meccanismo base di funzionamento del nucleo

La funzione principale che il SO deve svolgere è la virtualizzazione dei processi. Per comprendere il modo in cui il SO realizza questa funzione è opportuno afferrare prima gli aspetti generali di funzionamento, senza tener conto dei dettagli, poi procedere ad analizzare come ogni singola operazione possa essere svolta utilizzando i meccanismi messi a disposizione dal calcolatore, descritti nel precedente capitolo.

Da un punto di vista generale il SO funziona nel modo seguente:

- il SO alloca una zona di memoria ad ogni processo che viene creato e vi carica il programma eseguibile del processo stesso;
- il SO sceglie un processo e lo pone in **stato di esecuzione**;
- quando il processo in esecuzione richiede un servizio di sistema (tramite l'istruzione SVC) 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.
- il processo in stato di esecuzione abbandona tale stato solamente a causa di uno dei due eventi già citati nel capitolo sugli aspetti generali di LINUX, cioè:
    1. 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.

2. quando il SO rileva che è scaduto il quanto di tempo a disposizione del processo e quindi 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** (che differisce dallo stato di attesa, perchè un processo in stato di attesa non può essere posto in stato di esecuzione, invece un processo in stato di pronto può essere posto in esecuzione).

- quando il processo in esecuzione passa in stato di attesa o di pronto il SO seleziona in base a certi parametri un altro processo, attualmente in stato di pronto, e lo passa in esecuzione. Questa operazione è detta **commutazione di contesto**. Le operazioni riprendono come descritto sopra con la nuova situazione, in cui il processo in esecuzione è diverso da quello iniziale.

Nella descrizione fornita sopra non è stato affrontato un aspetto fondamentale, cioè cosa accade quando si verificano degli interrupt. La gestione degli interrupt segue i seguenti principi:

- quando si verifica un interrupt esiste generalmente un processo in stato di esecuzione. Tuttavia la situazione può appartenere ad uno dei 3 seguenti sottocasi:
  1. l'interrupt interrompe il processo mentre funziona in modalità U;
  2. l'interrupt interrompe un servizio di sistema che è stato invocato dal processo in esecuzione;
  3. l'interrupt interrompe una routine di interrupt relativa ad un interrupt verificatosi precedentemente;
- 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 (in realtà, più avanti vedremo le motivazioni di questa regola e dovremo anche osservare che la regola non viene applicata esattamente ma con una necessaria approssimazione).

- 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 risulterà molto importante nella corretta progettazione delle routine di interrupt, trattata nel capitolo relativo ai gestori di periferiche.

Per comprendere più in dettaglio come il SO riesca a realizzare i meccanismi indicati analizziamo ora alcune sue strutture dati fondamentali.

2. Strutture dati fondamentali

Il SO possiede una struttura dati, la **Tabella dei processi** (**ProcTable)**, nella quale per ogni processo che è stato creato viene inserito un record **ProcRec** contenente i dati fondamentali che lo caratterizzano. Una variabile intera **CurProc** contiene l'indice (in ProcTable) del record relativo al processo in esecuzione. La definizione in linguaggio C di queste strutture dati è riportata in figura 1. E' evidente che queste strutture definiscono un certo numero di informazioni importanti relative ad ogni processo esistente; inoltre, qualsiasi funzione del SO può fare riferimento ad un qualsiasi attributo A del processo in esecuzione individuandolo come *ProcTable[Curproc].A*.

Il SO contiene anche, per ogni processo che è stato creato, un'area adibita a **pila di sistema di modo S**; in altre parole, *il SO non ha una sola pila di sistema, ma ne ha una per ogni processo creato*. In ogni momento il SO opera utilizzando la pila di sistema associata al processo che è in esecuzione in quel momento. Quest'area dati può essere dichiarata in C come un array di dimensione MAXPROC di array costituiti da MAXPILA parole:

long PileDiSistema [MAXPROC] [MAXPILA]

```
ProcRec     ProcTable[MAXPROC]
int         CurProc
struct      ProcRec {
                …Stato /*indica lo stato del processo*/
                …Pila  /*contiene il valore del puntatore alla pila di modo S,
                         salvato quando l'esecuzione  del processo viene sospesa*/
                …Evento /*contiene l'identificatore dell'evento sul quale il
                         processo è in attesa*/
                …Base /*contiene la base del processo */
                …File Aperti  /* è una tabella che contiene i riferimenti ai file aperti
                         del processo (trattata nel capitolo sul file system);  dato che le
                         periferiche sono associate a file speciali, anche il terminale del
                         processo (standard input e standard output) è definito da
                         questa tabella */
                /*altre variabili che non interessano per il momento*/
                }
```

Figura 1 – Alcune strutture dati del SO LINUX

La struttura ProcTable è di fondamentale importanza per il funzionamento del SO; un errore nel suo aggiornamento avrebbe infatti conseguenze molto gravi per il funzionamento del sistema.

Le operazioni fondamentali che modificano il contenuto di ProcTable sono ovviamente quelle centrali della gestione dei processi, cioè il cambiamento di stato di un processo e la commutazione di contesto. Per evitare che tali funzioni, che sono richieste dai numerosi servizi di sistema e dalle routine di interrupt, vengano scritte molte volte, esistono alcune funzioni speciali del nucleo che possono venire invocate per svolgerle; si tratta delle funzioni seguenti:

- **sleep_on**: pone il processo corrente in stato di attesa
- **change**: esegue una commutazione di contesto
- **wakeup**: risveglia un processo, passando il suo stato da attesa a pronto
- **preempt**: sospende un processo per esaurimento del suo quanto di tempo

Queste funzioni del nucleo possono scrivere in ProcTable, mentre normalmente i vari servizi e routine di interrupt possono solamente leggerne il contenuto. Si tenga presente che la maggior parte dei servizi di sistema e delle routine di interrupt appartengono ai gestori di periferiche, cioè a quella parte del SO che deve essere continuamente

aggiornata. E' quindi importante evitare che queste parti in continua evoluzione possano per errore danneggiare il nucleo del sistema.

Per comprendere il funzionamento delle funzioni del nucleo e il significato delle strutture dati, analizziamo un esempio di funzionamento del SO.

3. Esempio di funzionamento

Quando la macchina viene accesa il SO viene automaticamente caricato da un file particolare, il processore viene posto in modo S e viene avviata l'esecuzione di una parte del SO che inizializza tutto il sistema. In particolare, durante questa fase il SO inizializza tutte le sue strutture dati, compresi i Vettori di Interrupt, e crea, in base ad un file di configurazione, un processo per ogni terminale sul quale un utente potrebbe eseguire un Login.

Lasciamo per il momento in sospeso la descrizione di come tutto ciò avvenga e consideriamo di avere raggiunto una situazione di funzionamento a regime come quella rappresentata in figura 2, con due processi P e Q che sono stati creati (MAXPROC=2), dei quali P è quello in esecuzione.

La figura mostra che:

- Il SO è allocato sequenzialmente a partire dall'inizio della memoria
- Il modo del processore è U, quindi il processo in esecuzione sta operando in modo U e gli indirizzi vengono rilocati con la base contenuta nel Registro Base
- I vettori di interrupt, le variabili CURPROC e ProcTable e PileDiSistema appartengono alla memoria del SO; i vettori di interrupt sono posti a partire dall'indirizzo 0 della memoria.
- il valore di CURPROC è 0, perchè per ipotesi ProcTable[0] si riferisce al processo P e ProcTable[1] si riferisce al processo Q
- il valore dei campi Pila ed Evento di ProcTable[0] non sono specificati, perchè non servono per un processo in esecuzione; i campi Stato e Base hanno un ovvio valore;
- il valore del campo Pila di ProcTable[1] è il valore posseduto dal registro sSP al momento in cui Q era stato sospeso, è cioè l'indirizzo della prima cella libera della pila di sistema del processo Q; Q è pronto all'esecuzione, quindi il campo Evento non ha un valore significativo

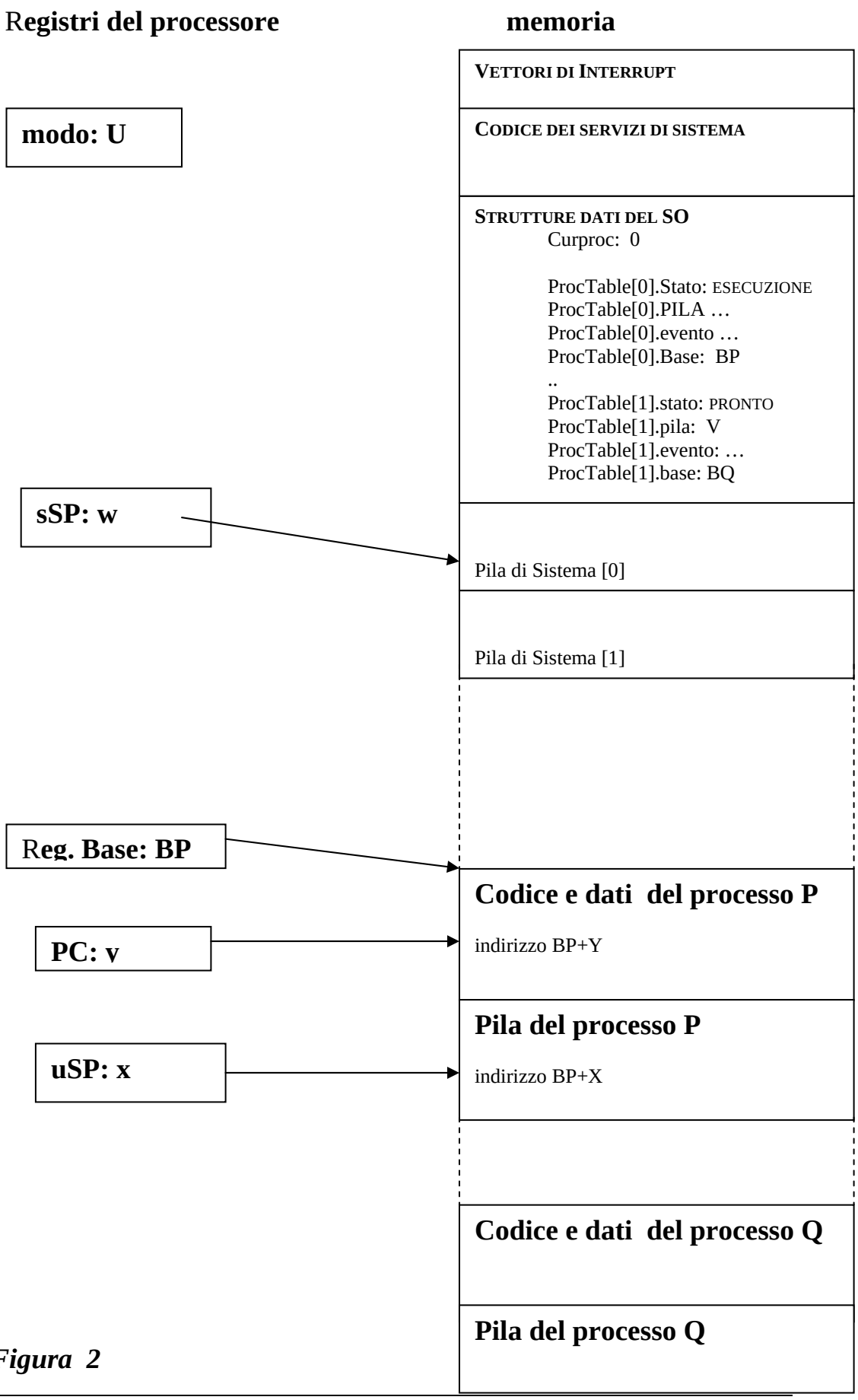

Registri del processore
memoria
modo: U
VETTORI DI INTERRUPT
CODICE DEI SERVIZI DI SISTEMA
STRUTTURE DATI DEL SO
Curproc: 0
ProcTable[0].Stato: ESECUZIONE
ProcTable[0].PILA …
ProcTable[0].evento …
ProcTable[0].Base: BP
..
ProcTable[1].stato: PRONTO
ProcTable[1].pila: V
ProcTable[1].evento: …
ProcTable[1].base: BQ
sSP: w
Pila di Sistema [0]
Pila di Sistema [1]
Reg. Base: BP
Codice e dati del processo P
PC: v
indirizzo BP+Y
Pila del processo P
uSP: x
indirizzo BP+X
Codice e dati del processo Q
Pila del processo Q

Figura 2

- Il registro sSP punta all'interno della pila di sistema di P
- Ogni processo è allocato in una zona di memoria; l'inizio di tale zona è detta base; la base di P è indicata come BP, quella di Q come BQ
- Il processo in esecuzione è P, e il registro base contiene il valore BP
- Il registro program counter PC contiene l'indirizzo y della prossima istruzione da eseguire; tale indirizzo verrà trasformato nell'indirizzo BP+y dall'unità di rilocazione al momento di accedere alla prossima istruzione

E' di fondamentale importanza comprendere bene il significato delle diverse pile presenti in figura 2; si noti che vi sono 2 pile nello spazio S e 2 pile nello spazio U (in quanto i processi sono solo 2 in questo esempio; se i processi creati fossero N si avrebbero N pile in modo S e N pile in modo U).

Quando il processore esegue un'istruzione con riferimento alla pila, una sola pila deve essere indirizzata, quindi devono esistere delle regole che rendono assolutamente univoca tale scelta.

Come mostrato in figura, quando è in esecuzione un processo (P in questo caso), i puntatori alla pila sSP e uSP puntano alle pile di modo S e di modo U relative al processo P stesso, quindi tutte le altre pile non sono utilizzate; inoltre, dato che la macchina sceglie di usare sSP oppure uSP in base al modo corrente del processore, che è unico, la pila da utilizzare risulta univocamente determinata.

Il contenuto della pila del processo Q non è indicato in figura; tale contenuto è quello lasciato sulla pila di Q al momento in cui Q è stato sospeso l'ultima volta; più avanti faremo le opportune ipotesi relativamente a tale contenuto.

Si ribadisce che lo stato di figura 2 costituisce una ipotesi di partenza, anche se non sappiamo al momento come sia stato creato; analizzando una possibile sequenza di eventi significativi possiamo convincerci che tale stato, assunto inizialmente per ipotesi, viene effettivamente creato e mantenuto dal SO.

Analizziamo quindi cosa avviene in seguito ad una serie di eventi a partire dalla situazione di figura 2:

a. Il processo P richiede un servizio di sistema, quindi l'istruzione all'indirizzo y+BP è una SVC

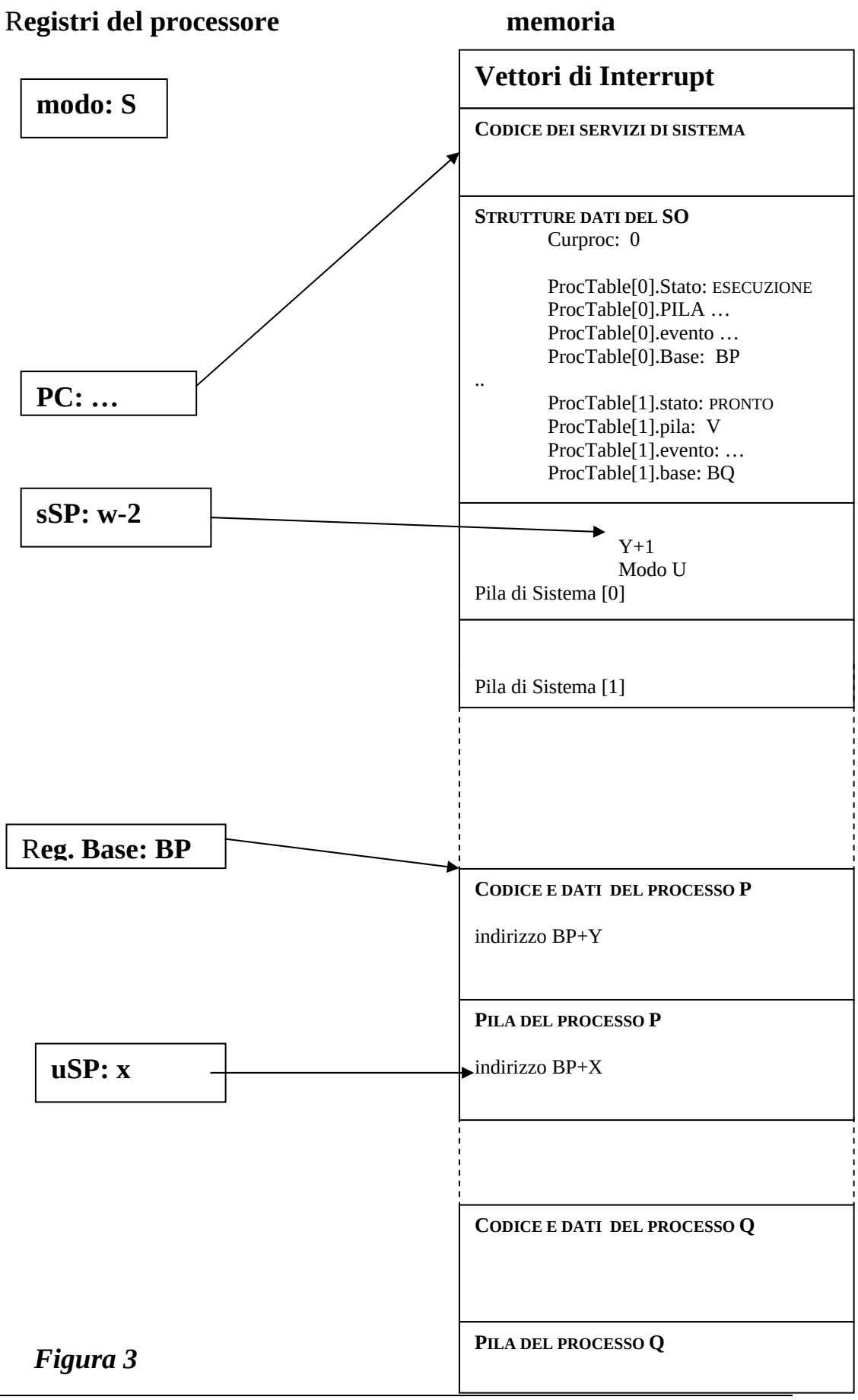

Registri del processore
memoria
modo: S
PC: …
sSP: w-2
Reg. Base: BP
uSP: x
Vettori di Interrupt
CODICE DEI SERVIZI DI SISTEMA
STRUTTURE DATI DEL SO
Curproc: 0
ProcTable[0].Stato: ESECUZIONE
ProcTable[0].PILA …
ProcTable[0].evento …
ProcTable[0].Base: BP
..
ProcTable[1].stato: PRONTO
ProcTable[1].pila: V
ProcTable[1].evento: …
ProcTable[1].base: BQ
Y+1
Modo U
Pila di Sistema [0]
Pila di Sistema [1]
CODICE E DATI DEL PROCESSO P
indirizzo BP+Y
PILA DEL PROCESSO P
indirizzo BP+X
CODICE E DATI DEL PROCESSO Q
PILA DEL PROCESSO Q

Figura 3

L'esecuzione di una SVC comporta, come mostrato in figura 3, il salvataggio sulla pila di modo S dell'indirizzo e del modo di ritorno al processo P, poi la commutazione del processore al modo S e il caricamento nel PC dell'indirizzo del gestore dei servizi di sistema del SO (G_SVC). Il registro sSP è stato decrementato di 2 a causa delle due scritture sulla pila.

G_SVC invoca la funzione che esegue il servizio richiesto, la quale opera nel contesto del processo P, anche se in modo S; in particolare, se tale funzione scrive sulla pila di sistema, essa scrive sulla pila associata al processo P, perché sSP punta a tale pila; se la funzione scrive sul terminale, essa scrive sul terminale del processo P, perché utilizza la variabile ProcTable[CurProc].File Aperti, che, essendo Curproc=0, punta ai file di P.

b. Il processo P si sospende e viene lanciato in esecuzione il processo Q

In figura 4 sono riportati gli effetti delle operazioni descritte nel seguito: Il servizio attivato al punto precedente (ad esempio, il servizio READ) scopre di non poter proseguire perchè deve attendere un dato da una periferica (ad esempio dal terminale di P). Il servizio richiede allora la sospensione del processo P. Per richiedere la sospensione un servizio di sistema invoca la funzione **sleep_on**(evento) del nucleo, passandole come parametro un numero che caratterizza l'evento che il processo deve attendere. La funzione sleep_on salva tutti i registri del processore, incluso uSP, sulla pila di sistema, salva nel record ProcRec del processo alcune informazioni di stato, in particolare il valore dell'evento ricevuto come parametro, pone lo stato del processo ad attesa e invoca la funzione **change**, che esegue la commutazione di contesto.

Lo pseudocodice che riassume le operazioni svolte da Sleep_on complessivamente è mostrato a pagina seguente, ma solo la parte iniziale fino all'invocazione di Change è stata eseguita per ora nel nostro esempio. Infatti, con riferimento alla commutazione dei processi la routine Sleep_on(E) può essere considerata suddivisa in due parti, una eseguita prima di invocare Change e l'altra che verrà eseguita quando il processo riprenderà l'esecuzione; indichiamo le due parti nel modo seguente:

Sleep_on(E**)_1**: salva lo stato del processo e invoca Change;

Sleep_on(E**)_2**: ricostruisce lo stato del processo che era stato sospeso e ritorna al servizio di sistema cha l'aveva invocata tramite RFS.

Si noti che a questo punto sulla pila di P è presente l'indirizzo di ritorno in sleep_on salvato al momento della chiamata di change (vedi figura 4, contenuto di Pila di Sistema[0]; per semplicità in tale figura sono stati omessi dalla pila di sistema gli indirizzi relativi all'invocazione del servizio da parte di G_SVC e all'invocazione di Sleep_on da parte del servizio).

```
Sleep_on(identificativo_evento)
{     /*Sleep_on_1:  salva contesto processo in esecuzione */
       /*salva sulla pila di sistemai uSP e i registri del processore*/
          ....
          /*salva informazioni di stato in ProcTable[curproc] */
               ProcTable[curproc].evento = identificativo_evento;
               ProcTable[curproc].stato = attesa;
          change( );
     /*Sleep_on_2: ricostruisci lo stato: carica uSP e gli altri registri dalla pila di sistema e
ritorna  al chiamante*/
          ....
     RFS
}
```

La routine change conclude il salvataggio dello stato di P ponendo nella variabile PILA di ProcRec il valore corrente di sSP, poi carica in Curproc il valore 1 (l'indice del processo Q in ProcTable) e quindi preleva da ProcTable[1] il valore della pila e della base di Q e li carica nei registri sSP e Base. In questo modo da questo momento qualsiasi accesso alla pila di sistema è sulla pila del processo Q e qualsiasi accesso a memoria in modo U è allo spazio di memoria del processo Q.
Il seguente pseudocodice riassume le operazioni svolte da Change.

```
Change( )
{
        /* completamento salvataggio contesto processo in esecuzione */
     ProcTable[curproc].pila = sSP;
        /* assegna nuovo curproc; Scheduler è un modulo che restituisce l' indice del
        processo da mandare in esecuzione*/
     curproc=Scheduler ();
       /* carica nuovo contesto e commuta */
     RegBase = ProcTable[curproc].base;
     sSP = ProcTable[curproc].pila;
     ProcTable[curproc].stato = esecuzione;
        /*esegui un ritorno all'indirizzo presente in cima alla pila di sistema*/
     RFS
}
```

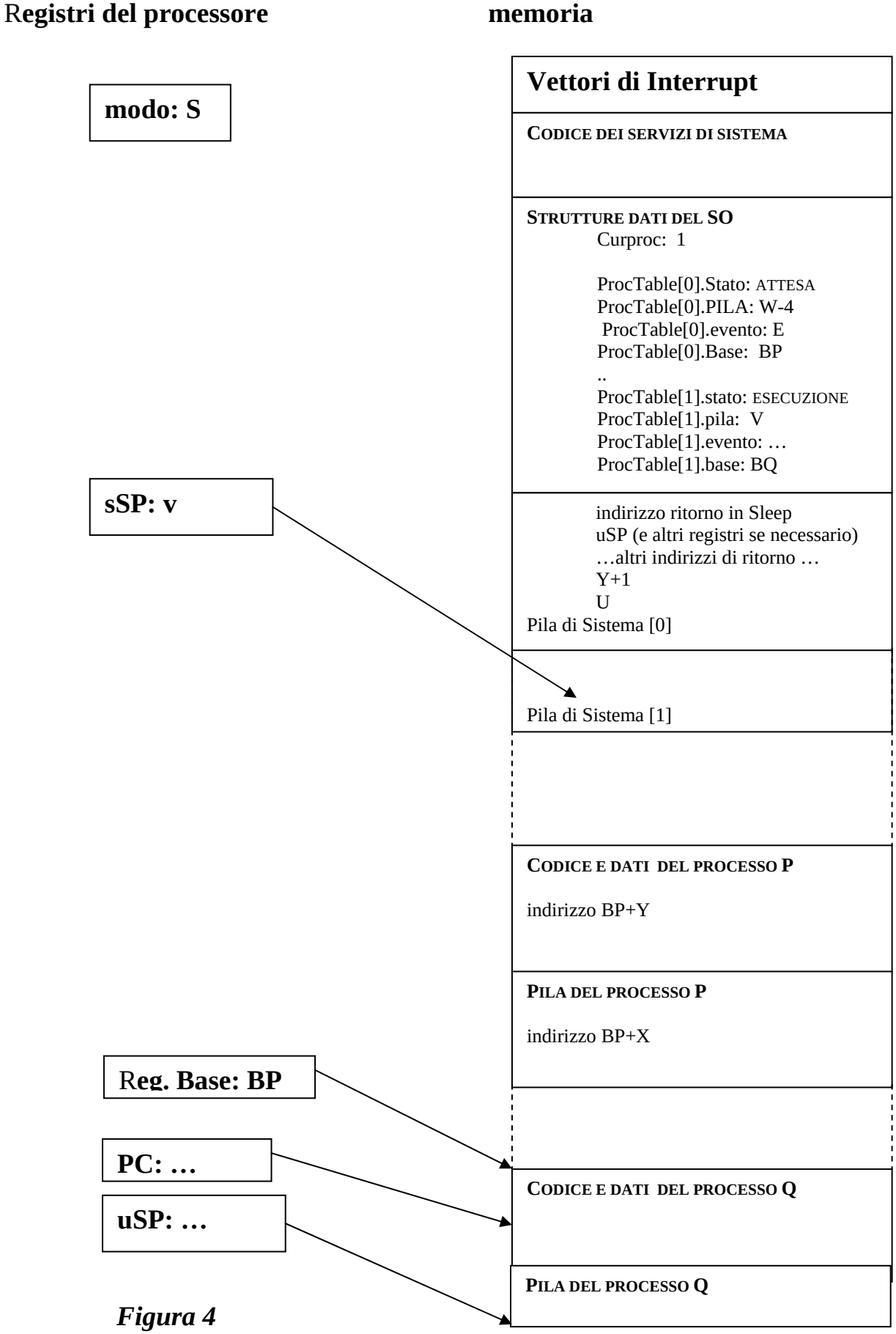

Registri del processore
memoria
modo: S
sSP: v
Reg. Base: BP
PC: …
uSP: …
Vettori di Interrupt
CODICE DEI SERVIZI DI SISTEMA
STRUTTURE DATI DEL SO
Curproc: 1
ProcTable[0].Stato: ATTESA
ProcTable[0].PILA: W-4
ProcTable[0].evento: E
ProcTable[0].Base: BP
..
ProcTable[1].stato: ESECUZIONE
ProcTable[1].pila: V
ProcTable[1].evento: …
ProcTable[1].base: BQ
indirizzo ritorno in Sleep
uSP (e altri registri se necessario)
…altri indirizzi di ritorno …
Y+1
U
Pila di Sistema [0]
Pila di Sistema [1]
CODICE E DATI DEL PROCESSO P
indirizzo BP+Y
PILA DEL PROCESSO P
indirizzo BP+X
CODICE E DATI DEL PROCESSO Q
PILA DEL PROCESSO Q

Figura 4

A questo punto la funzione change esegue un'istruzione di ritorno alla funzione chiamante e il ritorno avviene prelevando l'indirizzo di ritorno dalla pila di modo S (perchè il processore è in modo S) relativa al processo Q (perchè sSP punta a tale pila). *Pertanto il ritorno avviene a quella funzione che aveva invocato change quando era stata sospesa l'ultima esecuzione di Q*. Questo significa che il processo Q riprende l'esecuzione esattamente come se non fosse avvenuto niente dall'ultima volta in cui era stato sospeso. L'esecuzione di altri processi, avvenuta nel frattempo, non lo ha influenzato.

c. Durante l'esecuzione di Q si verifica l'evento atteso da P

Supponiamo che, mentre il processo Q è in esecuzione, si verifichi un interrupt del terminale di P. La routine di interrupt del terminale, che viene eseguita nel contesto di Q, scopre che si è verificato un evento atteso e invoca la funzione **Wakeup**(E) passandole come parametro il numero E, che identifica l'evento che si è verificato. Wakeup risveglia tutti i processi in attesa di tale evento, cioè cerca in ProcTable tutti i ProcRec con stato="attesa" ed evento="E" e cambia il loro stato a "pronto". Dopo questa operazione Wakeup termina e anche la routine di interrupt termina, restituendo il controllo al processo Q. Q prosegue normalmente, come se niente fosse accaduto (trasparenza dell'interrupt). Sorge ovviamente la domanda: chi ha stabilito quale sia il numero che identifica lo specifico evento richiesto dal servizio READ e riconosciuto dalla routine di interrupt del terminale? Questa domanda riguarda il gestore del terminale, perchè, come vedremo nel capitolo dedicato a questo argomento, chi progetta un gestore di periferica progetta sia i servizi che accedono a tale periferica, sia la routine di interrupt della periferica. Le funzioni sleep_on e wakeup del nucleo non conoscono il significato degli identificatori degli eventi; esse ricevono solo tali numeri come parametri.

d. Il processo Q esaurisce il proprio quanto di tempo

Durante l'esecuzione dei processi si verificano continuamente gli interrupt dell'**orologio di sistema**, una periferica che genera un interrupt con una certa frequenza prefissata (ad esempio, supponiamo un interrupt ogni millisecondo). Ad ogni interrupt dell'orologio la relativa routine incrementa una variabile TE che contiene il tempo di

esecuzione del processo corrente. In alcuni momenti particolari discussi più avanti viene invocata dal Sistema Operativo la funzione **preempt** del nucleo, che controlla se la variabile TE ha raggiunto il quanto di tempo e, in caso affermativo, cambia lo stato del processo Q da esecuzione a pronto, salva le informazioni necessarie in ProcRec di Q e invoca change per una commutazione di processo.

Lo pseudocodice complessivo di preempt è riportato di seguito; la struttura di Preempt è molto simile a quella di Sleep_on; in particolare anche preempt è divisa in 2 parti, delle quali al momento nel nostro esempio è stata eseguita solamente la prima.

```
Preempt( )
{
    /*verifica se è scaduto un quanto di tempo*/
    if (scaduto)
    {
    /*Preempt_1:  salva contesto processo in esecuzione */
      /*salva sulla pila di sistemai uSP e i registri del processore*/
         ....
      /*salva informazioni di stato in ProcTable[curproc] e invoca change */
         ProcTable[curproc].stato = pronto;
          change( );

    /*Preempt_2: ricostruisci lo stato: carica uSP e gli altri registri dalla pila di sistema e
ritorna  al chiamante*/
         ....
    }
    RFS
}
```

e. Il processo P riprende l'esecuzione

Possiamo ipotizzare che, essendo P l'unico processo pronto nel sistema, alla sospensione di Q venga posto in esecuzione P. A questo punto la funzione change pone Curproc=0 e carica da ProcTable[0] i valori dei registri Base e sSP. Dopo queste operazioni change torna al chiamante; l'indirizzo di ritorno viene quindi trovato sulla pila, dove in effetti troviamo l'indirizzo di ritorno alla routine sleep_on che si era interrotta. La routine sleep_on completa la propria attività eseguendo la porzione che segue l'invocazione di change nello pseudocodice, cioè la porzione Sleep_on_2.

Sleep_on_2 ritorne tramite RFS al servizio che l'aveva invocata; il servizio ritorna al gestore G_SVC, il quale a sua volta esegue una IRET che preleva dalla pila i valori del

PC (y+1) e del modo del processore (U) e li carica nei registri del processore. A questo punto riparte l'esecuzione del processo P e la situazione, per quanto riguarda P, è ritornata ad essere esattamente quella di figura 2, con la sola differenza che l'istruzione SVC e quindi il relativo servizio sono stati eseguiti e il registro PC è y+1, pronto per iniziare l'esecuzione della prossima istruzione. L'esecuzione del processo Q e la gestione degli interrupt, avvenute nel frattempo, sono completamente trasparenti per il processo.

4. Regole relative alla Preemption – priorità dinamiche

Il motivo per cui si era formulata la regola *"non viene mai svolta una commutazione di contesto durante l'esecuzione di un interrupt"* riguarda la difficoltà di scrivere software corretto ammettendo che in qualsiasi momento possa avvenire una commutazione di contesto. Si immagini di scrivere una routine di interrupt e di dover tener conto del fatto che, dopo una qualsiasi istruzione, nel corso di un'altra routine di interrupt di maggior priorità, possa essere cambiato il contesto e quindi, tra l'altro, anche la pila di sistema. Basta pensare, ad esempio, che la routine non potrebbe invocare funzioni, perchè il cambio di pila renderebbe scorretti i ritorni, per rendersi conto di quanto questa regola sia importante.

Tuttavia, come già accennato e come implicitamente già mostrato nell'esempio precedente, il momento in cui può avvenire una commutazione di contesto dovuta ad una preemption paradossalmente deve necessariamente violare la regola fondamentale indicata sopra. Come abbiamo visto nell'esempio, la preemption del processo Q è infatti avvenuta nell'esecuzione della routine di interrupt dell'orologio. Il motivo della violazione è dovuto al fatto che il SO può scoprire il raggiungimento del quanto di tempo solo grazie alla routine di interrupt dell'orologio di sistema. E' evidente che, per imporre la preemption il SO deve attuarla prima di ritornare al modo U, perchè il processo in modo U non contiene nessun controllo relativo all'eventuale superamento del quanto.

A causa dell'impossibilità di osservare 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 di interrupt solamente alla fine e solamente se il modo al quale la routine sta per ritornare è il modo U"* . In base a questa regola, la

routine di interrupt dell'orologio, quando ha terminato l'esecuzione delle proprie funzioni, se verifica di dover tornare al modo U (cioè se l'interrupt ha interrotto un processo in modo utente, e non un servizio di sistema o un altro interrupt) invoca la preempt.

Per evitare di dilazionare troppo la preemption di un processo nel caso in cui la routine di interrupt dell'orologio abbia interrotto altre routine di interrupt e quindi non ritorni al modo U, **anche tutte le altre routine di interrupt e il gestore dei servizi di sistema** controllano, prima di terminare, se il ritorno è al modo U e in tal caso invocano la preempt, che controlla se per caso il processo corrente deve essere sospeso. La regola diventa quindi la seguente: *"il SO (in particolare, le routine di interrupt e il gestore dei servizi di sistema) prima di eseguire una IRET che lo riporta al modo U invoca la funzione preempt"*.

La routine Preempt( ) procede a sospendere il processo attualmente in esecuzione in base al valore della variabile booleana *need_resched*, che indica se tale processo deve essere sospeso. La variabile *need_resched* è posta al valore 1 in due occasioni:

1. quando la routine di interrupt dell'orologio scopre, incrementando il contatore del tempo di esecuzione del processo corrente, che ha superato il quanto di tempo;
2. quando la routine wakeup( ) risveglia un processo in attesa e tale processo risulta avere una priorità superiore a quella del processo corrente

La seconda regola, insieme al meccanismo della priorità dinamica, serve per favorire i processi I/O bound, che, come già osservato al capitolo 7, sarebbero svantaggiati da un time sharing puro.

La priorità dinamica di un processo è calcolata come somma di una priorità base (o quanto di base) e del numero di clock-tick ancora disponibili per quel processo in un dato periodo di schedulazione (epoch). I dettagli del meccanismo di calcolo della priorità dinamica non vengono presentati. L'idea di base è la seguente: il tempo viene suddiviso in periodi di schedulazione (epoch); all'inizio di un periodo di schedulazione tutti i processi normali hanno la stessa priorità dinamica; tale priorità viene decrementata in base all'uso effettivo della CPU (cioè al tempo di esecuzione).

Inoltre, la priorità statica è sempre superiore a quella dinamica, quindi i processi normali vengono eseguiti solo se non ci sono processi real-time pronti per l'esecuzione.

Riprendiamo ora l'esempio dell'Editor e del Compilatore introdotto nel capitolo 7 per capire come le regole enunciate favoriscano un buon funzionamento dei processi I7O bound.

Dato che l'Editor è molto interattivo (praticamente si autosospende dopo ogni carattere ricevuto in attesa del successivo) tenderebbe ad eseguire per poco tempo e poi a sospendersi. Invece il Compilatore ogni volta che va in esecuzione tenderebbe ad eseeguire per l'intero quanto di tempo. Dato che la priorità dinamica si riduce con la riduzione dei clock-tick ancora disponibili, dopo un po' la priorità del Compilatore risulterà inferiore a quella dell'Editor e, quindi, quando arriverà un interrupt per l'Editor che lo passerà in stato di pronto, la routine Wakeup( ) attiverà la variabile need_resched; successivamente, appena verrà invocata preempt( ), l'Editor verrà posto in esecuzione, anche se il quanto di tempo del Compilatore non è scaduto.

5. Gli stati di un processo e le transizioni di stato

Quanto esposto sopra può essere rianalizzato considerando le possibili transizioni di stato di un generico processo.

In Figura 5 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 può essere in esecuzione (in modo U oppure S), ma molti processi possono essere pronti o in attesa di eventi. Nella figura 5 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 SVC, 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 che riporta il processo al modo U (transizione 2).

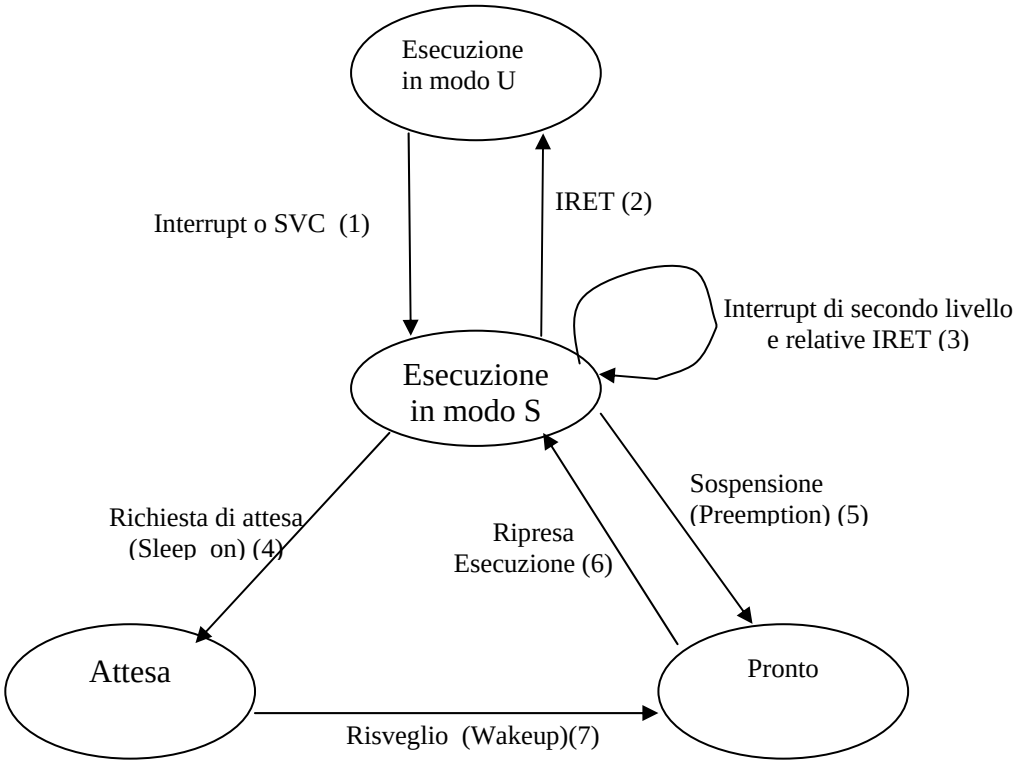


Figura 5 – Stati e transizioni di un processo

Durante l'esecuzione in modo S possono verificarsi altri interrupt di 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 una sleep_on (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 preempt richiede un commutazione di contesto (transizione 5).

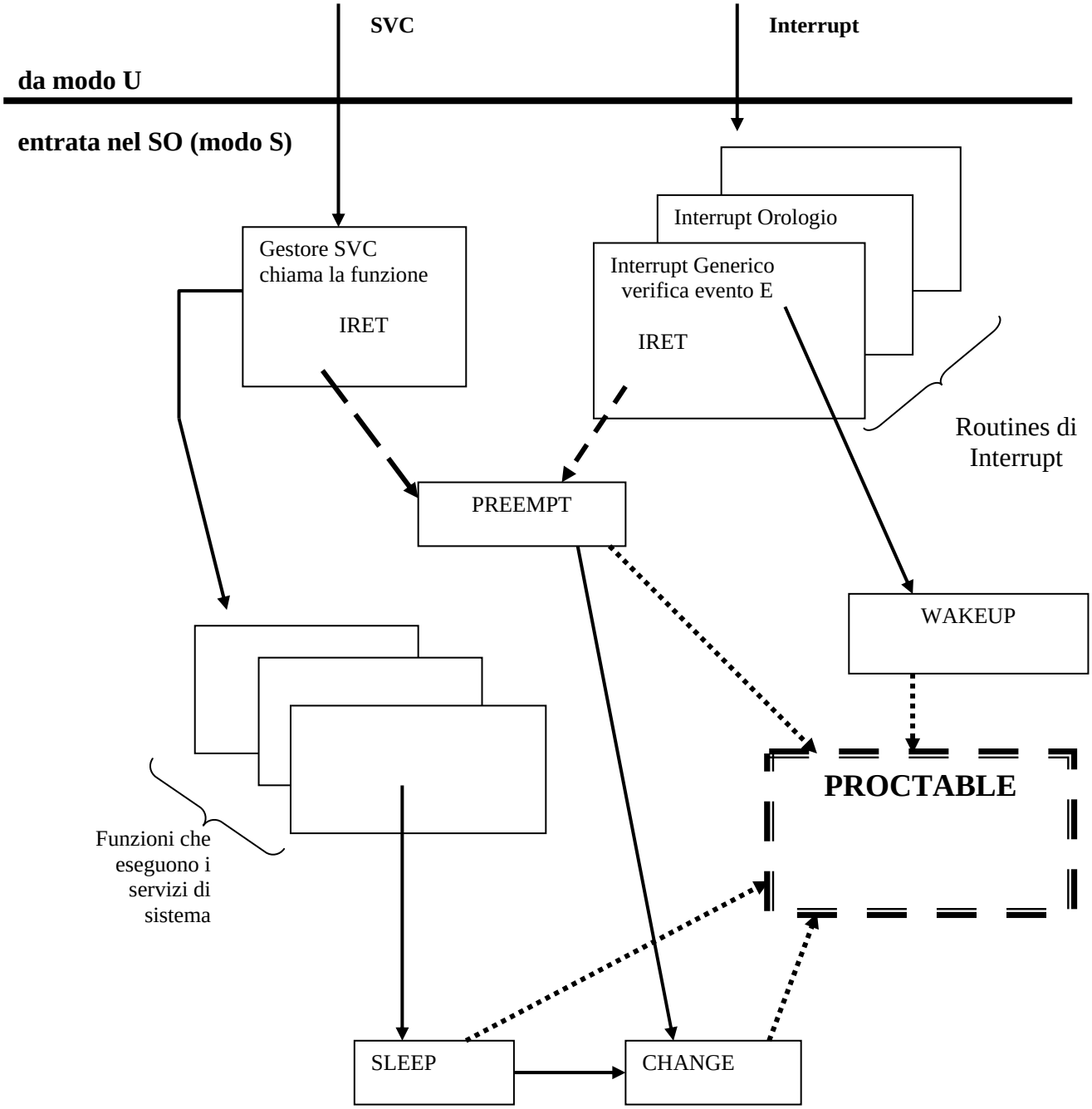
La ripresa dell'esecuzione di un processo (transizione 6) avviene se il processo è pronto, cioè non è in attesa, e se è quello avente maggior diritto all'esecuzione tra tutti quelli pronti. Si osservi che il momento in cui avviene la transizione 6 di un processo P non dipende da P stesso, ma dal processo 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 rsveglio 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.

6. Struttura modulare e funzionale del sistema

La struttura delle principali routine del SO e delle loro interazioni per quanto riguarda la gestione dello stato dei processi è illustrata in figura 6, che mostra anche quali di queste funzioni leggono o scrivono informazioni in ProcTable.

La struttura funzionale complessiva del sistema è mostrata in figura 7, che differisce dalla struttura modulare per il fatto che i servizi sono raggruppati in base ad alcune grandi funzionalità: la gestione dei processi (trattata in questo capitolo), e la gestione della memoria, dei file (file system) e delle periferiche (device drivers), trattate nei capitoli successivi. Le routine di servizio degli interrupt delle periferiche appartengono funzionalmente ai gestori delle periferiche.



Legenda:

Normale invocazione di funzioni
Invocazione di preempt al momento di eseguire un IRET
Accesso ai dati di ProcTable

Figura 6 – Struttura modulare del Sistema Operativo

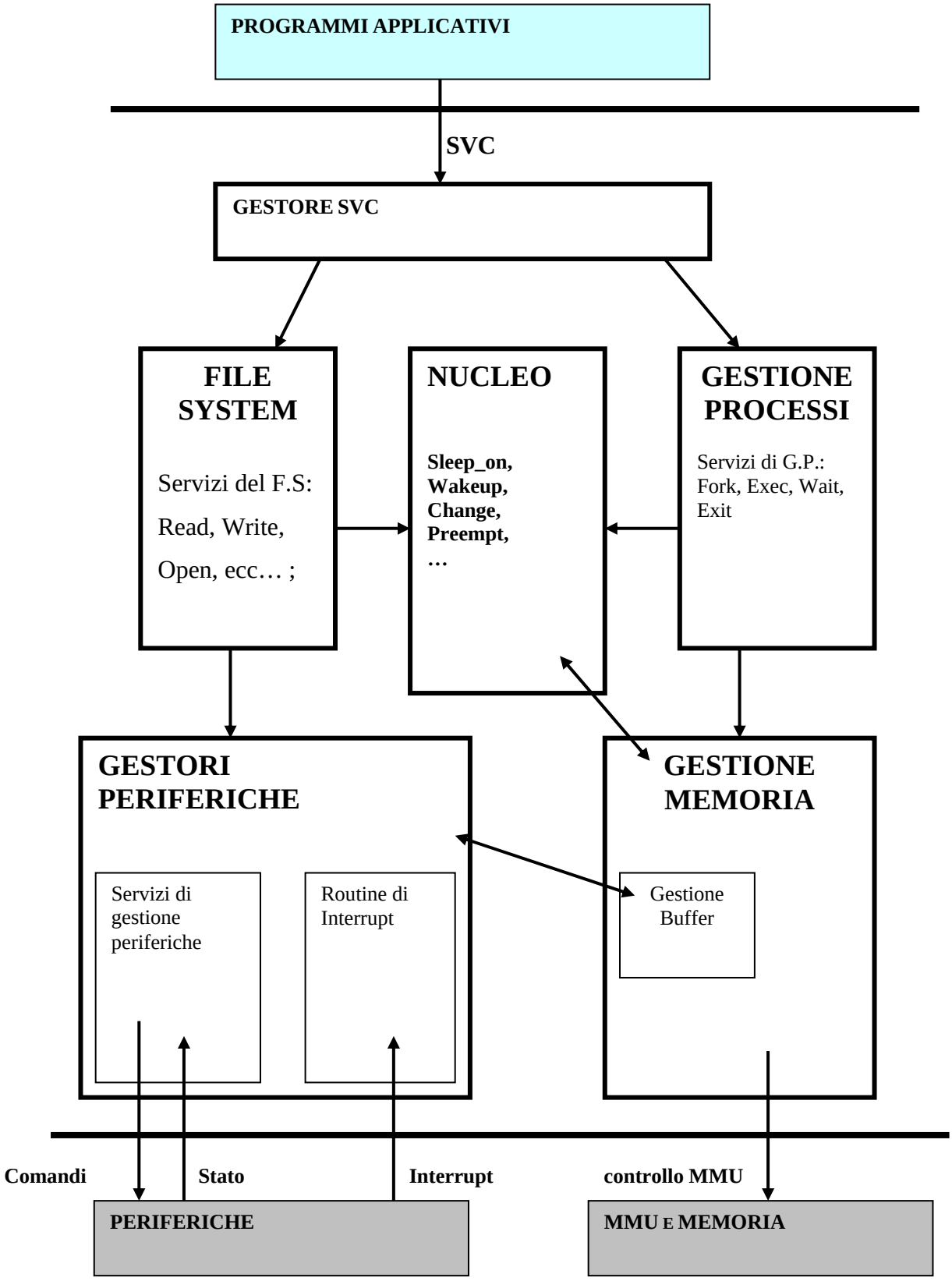


Figura 7 – Struttura funzionale del SO

7. Avviamento, inizializzazione e interprete comandi

Al momento dell'avviamento del sistema operativo, cioè del suo caricamento in memoria (bootstrap), il sistema deve svolgere alcune operazioni di avviamento che producono la situazione di funzionamento che è stata analizzata in questo capitolo. In realtà, tale avviamento consiste essenzialmente nell'inizializzazione di alcune strutture dati e nella creazione di un processo iniziale (processo 1, che esegue il programma init).

Abbiamo infatti visto che tutti i processi sono creati da un altro processo tramite l'esecuzione di una fork( ), ma ovviamente deve esistere almeno un processo iniziale che viene creato direttamente dal sistema operativo. Un aspetto interessante di LINUX è costituito dal fatto che tutte le operazioni di avviamento successive alla creazione del processo 1 sono svolte dal programma init, cioè da un normale programma non privilegiato, utilizzando i meccanismi descritti della programmazione di sistema normale (cioè di modo utente). Init è infatti un normale programma, e il suo processo differisce dagli altri processi solamente per il fatto di non avere un processo padre.

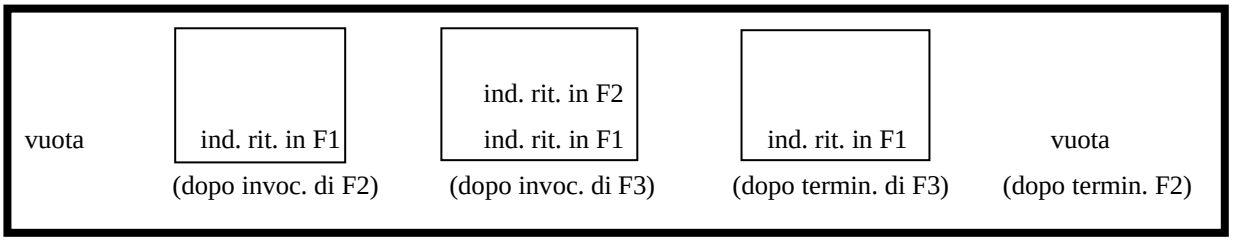
Il processo 1, eseguendo init, crea un processo per ogni terminale sul quale potrebbe essere eseguito un login; questa operazione è eseguita leggendo un apposito file di configurazione. Quando un utente si presenta al terminale ed esegue il login, se l'identificazione va a buon fine, il processo che eseguiva il programma di login lancia in esecuzione il programma shell (interprete comandi), e da questo momento la situazione è quella di normale funzionamento.

Il programma init può trovare nel file di configurazione anche l'indicazione di creare dei processi permanenti (demoni), come ad esempio i server di rete. Ovviamente, la creazione di tali processi è realizzabile tramite i servizi disponibili.

In conclusione, il nucleo del sistema operativo mette a disposizione dei normali processi un insieme di servizi sufficientemente potente da permettere di realizzare molte funzioni generali tramite processi normali, evitando di incorporare nel nucleo del sistema funzionalità che è comodo poter modificare o sostituire per adattarle alle diverse situazioni di impiego (come esempio particolare di questo fatto, si consideri che sono stati realizzati, da diversi programmatori e in tempi diversi, numerosi interpreti comandi per sistemi UNIX).

8. Struttura dettagliata di alcune Routine del S.O. e sequenze di esecuzione

Il meccanismo di commutazione dei processi descritto ha alcune conseguenze particolari sulla struttura sulle sequenze di esecuzione delle routine del Sistema Operativo. Con sequenza di esecuzione delle routine intendiamo l'ordine in cui tali routine sono invocate e quindi terminano e ritornano. In un normale processo l'ordine di terminazione è ovviamente inverso a quello di invocazione e si riflette nella sequenza di caricamento di indirizzi di ritorno sulla pila e successivo prelievo di tali indirizzi; ad esempio se F1 invoca F2, F2 invoca F3, poi termina F3 e poi termina F2, la pila assume gli stati seguenti:



Se analizziamo però la pila di sistema in presenza di commutazioni di contesto la situazione si complica, perché esistono numerose pile di sistema e la loro crescita e decrescita si intreccia, riflettendo il fatto che le sequenze di invocazioni di routine nel contesto di un processo si possono interrompere in certi punti prestabiliti permettendo la partenza di altre sequenze nell'ambito di altri processi.

Per arrivare, nel prossimo sottocapitolo, alla descrizione di sequenze di invocazioni e ritorni di routine nei diversi processi, iniziamo qui con l'analisi delle sequenze di routine di un singolo processo evidenziando però i punti in cui possono inserirsi, tramite commutazione di contesto, sequenze di altri processi.

La seguente tabella indica la notazione di base per rappresentare tali sequenze di routine di sistema. Tale notazione di base deve essere estesa per rappresentare i punti in cui si inseriranno routine eseguite da altri processi.

Notazione abbreviata	Modulo (o frammento di modulo) di sistema
G_SVC	Gestore SVC (chiamata a supervisore, supervisor call)
R_Int (Disp)	Routine di interruzione; Disp può valere: CK = orologio, RETE_acc = scheda rete per accept, S_out = Standard output, S_in = Standard input, DMA_in = disco_in_lettura, DMA_out = disco_in_scrittura
<nome routine di sistema>	può essere: fork, write, read, wait, exit, open, sleep, exec, preempt, change, accept ecc...
Sleep_on (E_n)	Sleep_on: per indicare l'evento su cui viene sospeso il processo si scriva convenzionalmente E_1, E_2, E_3, ..., E_n, ...
Wake_up (E_n)	Wake_up: il simbolo E_n indica l'evento, come in Sleep_on
Codice Utente	nessun modulo di sistema, il processo esegue codice utente

Suddivisione di alcune routine in diverse parti:

Abbiamo visto che la routine Sleep_on(E_n) può essere considerata suddivisa in due parti, una eseguita prima di invocare Change e l'altra quando il processo riprende l'esecuzione; indichiamo le due parti nel modo seguente:

Sleep_on(E_n)_1: salva lo stato del processo e invoca Change;

Sleep_on(E_n)_2: ricostruisce lo stato del processo che era stato sospeso e ritorna, tramite RFS, al servizio di sistema che l'aveva invocata.

Nel caso in cui la commutazione di processo sia causata da Preempt (suddivisa in Preempt_1 e Preempt_2) invece che da Sleep-on si verifica una situazione simile, ma con alcune particolarità. La routine Preempt è sempre invocata prima del ritorno al modo U; tale invocazione è quindi eseguita da G_SVC oppure da una routine di interrupt R_Int non nidificata e darà luogo ad una commutazione di contesto solamente se il quanto di tempo del processo è esaurito.

Quando la Preempt è invocata da una routine di interrupt non nidificata la sequenza di invocazioni e terminazioni è la seguente:

R_Int_1 *(R_Int è eseguita quasi completamente, ma al momento di tornare al modo U invoca IRET)*

Preempt_1 *(se il quanto è esaurito salva lo stato del processo e invoca Change)*

Change,

(a questo punto vengono eseguiti altri processi,....)

(alla ripresa di P verranno eseguite le seguenti routine)

Preempt_2 *(ricostruisce lo stato del processo e esegue RFS)*

R_Int_2 *(esegue solamente la IRET)*

Il gestore dei servizi, G_SVC, assume nel caso più generale, cioè se il servizio invocato va in attesa e se il quanto di tempo del processo è scaduto, una struttura in 3 parti rispetto alla commutazione di processo:

G_SVC_1 *(il gestore dei servizi di sistema parte e invoca il servizio Serv)*

Serv *(se il servizio Serv richiede un'attesa, esso esegue fino all'invocazione di Sleep_on)*

Sleep_on(E_n)_1 *(salva lo stato del processo e invoca Change)*

Change

(a questo punto vengono eseguiti altri processi,....)

(alla ripresa di P verranno eseguite le seguenti routine)

Sleep_on(E_n)_2 *(ricostruisce lo stato del processo e ritorna al chiamante)*

Serv *(il servizio Serv riprende l'esecuzione: deve tornare a G_SVC, ma potrebbe prima avere necessità di invocare di nuovo Sleep_on per attendere un ulteriore evento)*

G_SVC_2 *(prima o poi G_SVC riprende l'esecuzione e, prima di ritornare al modo U, invoca Preempt)*

Preempt_1 *(se il quanto è esaurito salva lo stato del processo e invoca Change)*

Change

(a questo punto vengono eseguiti altri processi,....)

(alla ripresa di P verranno eseguite le seguenti routine)

Preempt_2 *(ricostruisce lo stato del processo e esegue RFS)*

G_SVC_3 *(esegue solamente la IRET)*

Struttura del servizio Fork

Ipotizziamo che un processo P crei un processo figlio F. Le azioni svolte dalla Fork sono essenzialmente le seguenti:

- aggiorna le strutture dati del S.O. creando un elemento nuovo in *ProcTable[F]* e uno in in *PileDiSistema[F]*, dedicati al nuovo processo F
- assegna come contenuto di *PileDiSistema[F]* una copia di quello del padre *PileDiSistema[P]*;
- teoricamente viene creata per il figlio F una copia della memoria utente del padre (come vedremo nel prossimo capitolo, il termine teoricamente qui vuole indicare che in realtà questa operazione molto onerosa viene semplificata con opportuni artifici)

Dopo l'esecuzione di queste operazioni abbiamo due processi che potrebbero essere ambedue in esecuzione, ma ovviamente solo uno dei due può essere effettivamente eseguito, quindi l'altro dovrà essere in stato di pronto. *Noi ipotizzeremo che dopo la fork il padre sia in stato di esecuzione e il figlio sia in stato di pronto.*

Il compito più delicato della fork( ) è quello di creare una situazione opportuna per il figlio, in modo che possa essere posto in esecuzione da una successiva Change come se fosse un normale processo pronto, sospeso da una precedente Preempt.

Un problema da risolvere è il seguente: come restituire al processo F il valore 0 per la variabile pid.

In generale i servizi di sistema possono restituire dei valori al processo che li ha invocati copiandoli sulla pila di modo U del processo (tutti i tipi di hardware possiedono un modo di copiare dei dati da spazio del sistema operativo a spazio utente), tuttavia tale operazione si basa sul valore corrente del registro base e quindi può avvenire solamente verso lo spazio utente del processo in esecuzione (per inciso, per questo motivo le routine di interrupt, che avvengono nel contesto di un diverso processo, non possono restituire direttamente dati al processo servito).

In questo caso abbiamo ipotizzato che il processo in esecuzione sia P (il padre), quindi la fork potrebbe restituire direttamente al padre il pid del figlio, copiandolo sulla sua pila di modo U, ma non può fare altrettanto per restituire il valore 0 al figlio. Per questo motivo ipotizziamo che la fork lasci i valori da restituire sulle 2 pile di sistema di P e di

F, e che G_SVC esegua, dopo l'invocazione di una fork, la copia del valore presente in cima alla pila di sistema di modo S sulla pila di sistema di modo U, come mostrato dal seguente pseudocodice:

```
G_SVC
{  ...
     /* G_SVC_1: scelta del servizio da invocare */
      ...
     /* invocazione di fork() */
     case "fork":  {  fork();
           /* quando la fork( ) è terminata, preleva il valore restituito da fork dallo stack
           di sistema e lo impila nello stack utente */
           ...  }
     case …: ...
     ...

     /* parti restanti di G-SVC */

     IRET
}
```

Per capire questo pseudocodice si deve tenere presente che la funzione fork viene invocata una volta, ma ritorna due volte!

Infatti il codice della fork, copiando il contenuto della pila di sistema di P in quella di F, copia anche l'indirizzo di ritorno al G_SVC posto in cima a tale pila. Il ritorno avviene quindi due volte:

1) quando la fork stessa esegue RFS, nel contesto del processo padre P
2) quando la funzione change sceglie il processo figlio F come processo da eseguire e lo lancia in esecuzione caricando l'indirizzo di ritorno dalla sua pila di sistema

Naturalmente la funzione fork dopo aver copiato la pila di P in quella di F deve modificare le due pile, ponendo il pid di F sulla pila di P e il valore 0 sulla pila di F (in ambedue i casi nella posizione del valore restituito, cioè subito sotto l'indirizzo di ritorno).

Lo pseudocodice della funzione fork( ), a pagina seguente, riassume le operazioni descritte.

Dal punto di vista delle sequenze di invocazioni di routine del sistema, la fork da luogo alle seguenti situazioni:

1)nel contesto del padre (P), se non è esaurito il suo quanto di tempo (altrimenti bisogna tener conto di quanto visto sopra relativamente alla preempt)

G_SVC_1

fork

G_SVC_2

...

2)quando il figlio viene lanciato in esecuzione per la prima volta

Change *(nel contesto di qualsiasi processo, sceglie F come prossimo)*

G_SVC_2 *(nel contesto del figlio F)*

...

```
pid_t fork( )
{
  /* crea le varie strutture dati S.O. per il figlio */

       /* 1 - crea elemento in ProcTable per figlio*/
       /* 2 - alloca memoria utente per figlio e assegna base */
            ProcTable[figlio].base = base_figlio;
         /* creazione(salvataggio) contesto del figlio – simile a  Preempt */
           /*salva sulla pila di sistema del figlio uSP e gli altri  registri del processore*/
           ....
           /*salva informazioni di stato in ProcTable[figlio] */
            ProcTable[figlio].stato = pronto;
       /* 3 - alloca stack di sistema per figlio, copia PilaS padre in PilaS figlio e aggiorna
       sSP figlio – attenzione che sSP del figlio è diverso da sSP del padre, quindi la seguente
       operazione deve essere svolta quando sSP punta alla pila del figlio, alla fine
       dell'operazione di copia*/
            ProcTable[figlio].pila = sSP;

       /* 4 - predisponi il ritorno: gestisci i valori restituiti secondo le convenzioni
       dell'hardware utilizzato */
           /* carica il valore restituito (PID figlio) in PilaS padre*/
           /* carica il valore restituito (0) in PilaS figlio*/
          /* sSP deve puntare a PilaS del padre */
 RFS /*ritorna al padre */
}
```

Struttura del servizio Exit

La funzione exit, pur essendo abbastanza semplice, deve tenere conto di due aspetti non banali:

1) deve gestire il valore di ritorno al processo padre, sia nel caso in cui tale processo sia già in wait, sia in caso contrario;
2) deve far partire un nuovo processo al posto di quello terminato (questa funzione è sostanzialmente identica alla seconda parte della funzione change)

Il seguente pseudocodice riassume le funzioni di exit. Si osservi che l'indirizzo di ritorno salvato dall'invocazione di exit non verrà mai usato, perchè è stato eliminato insieme alla pila stessa del processo terminato.

```
exit( )
{
      /* dealloca memoria utente processo e dealloca stack sistema processo */

      if (esiste padre in wait o waitpid) {
           /* dealloca l'elemento del processo (figlio) in ProcTable (e a
              questo punto il processo non esiste) */
           Wake_up (evento_exit);
       }
      else {          /* non esiste padre già in wait ...*/
        /* il processo diventa zombie e il valore restituito viene
         salvato per usi futuri */
      }

    /* manda in esecuzione un nuovo processo – come seconda parte di change */
          curproc=Scheduler ();
          /* carica nuovo contesto e commuta */
                    RegBase = ProcTable[curproc].base;
                    sSP = ProcTable[curproc].pila;
                    ProcTable[curproc].stato = esecuzione;
   /* ritorna alla routine il cui indirizzo è sulla pila del nuovo processo in esecuzione */
 RFS
}
```

Comportamento dello Scheduler

Il comportamento dello Scheduler non viene analizzato in questo testo; ci limitiamo a fare le seguenti ipotesi molto semplici relative a tale comportamento:

- ad ogni processo è associata una priorità

- la priorità è un valore che permette allo Scheduler di scegliere, tra più processi in stato di pronto, quale mandare in esecuzione

Il processo "Idle"

Talvolta si verifica la situazione in cui nessun processo utile è pronto per l'esecuzione; in tali casi viene posto in esecuzione il processo 1, quello che viene creato all'avviamento del sistema, che viene chiamato convenzionalmente Idle, perchè non svolge alcuna funzione utile dopo aver concluso l'avviamento del sistema.

Il processo Idle, dopo aver concluso le operazioni di avviamento del sistema, assume le seguenti caratteristiche:

1) la sua priorità è inferiore a quella di tutti gli altri processi,
2) non ha mai bisogno di sospendersi tramite Sleep_on, quindi non è mai in stato di attesa,
3) il suo quanto di tempo è sempre scaduto, perchè di valore molto piccolo

Quando il processo Idle è in stato di esecuzione non fa niente di utile – in base al tipo di processore potrebbe eseguire un ciclo infinito oppure aver eseguito un'istruzione speciale privilegiata, che sospende l'esecuzione delle istruzioni da parte del processore in attesa che si verifichi un interrupt.

Il processo Idle va in esecuzione quando tutti gli altri processi sono in stato di attesa; questo fatto si può verificare in qualsiasi momento, ad esempio quando è terminato l'avviamento del sistema e tutti i processi creati all'avviamento hanno concluso le operazioni di inizializzazione e sono in attesa di eventi (ad esempio, tutti i server di rete sono in attesa di richieste di connessione).

L'esecuzione di Idle può terminare quando si verifica un interrupt, in base alla seguente sequenza di eventi:

1) viene eseguita la routine R_Int (nel contesto di Idle)
2) R_Int gestisce l'evento ed eventualmente risveglia un processo tramite Wakeup
3) R_Int invoca preempt
4) Preempt invoca Change, perchè il quanto di Idle è scaduto
5) Se al precedente passo 2 è stato risvegliato un processo, allora Change lancia in esecuzione tale processo, sospendendo Idle; in caso contrario

Change rimette in esecuzione Idle e l'interrupt è stato servito nel contesto di Idle senza causare ulteriori effetti

9. Esercizio conclusivo

Dopo avere analizzato la sequenza di parti di routine di SO che vengono eseguite nel contesto di un singolo processo passiamo a considerare il problema generale di determinare la sequenza complessiva di routine di SO eseguita nell'ambito dei diversi processi, risolvendo il seguente esercizio.

Si consideri il frammento seguente di programma (gli #include necessari sono omessi):

```
/* programma main.c */
main ( ) {
    int pid1, pid2;
    int fd1;
    char v [2000];
    char c [1500];
    ...
    pid1 = fork ( );
    fd1 = open ("/user/info2/file1", O_RDWR);
    if (pid1 == 0) {
        /* codice eseguito da Q, figlio di P          */
        write (fd1, v, 2000);    /* scrive su file1    */
        exit (1);
    } else {
        /* codice eseguito da P                       */
        pid2 = fork ( );
        if (pid2 == 0) {
            /* codice eseguito da R, figlio di P      */
            fd2 = open ("/user/info2/file1", O_RDWR);
            read (fd2, c, 50);   /* legge da file1     */
            exit (2);
        } else {
            /* codice eseguito da P                   */
            pid1 = waitpid (pid1, &status, 0);
            exit (0);
        } /* end if */
    } /* end if */
} /* main.c */
```

Un processo **P** crea un figlio **Q** e poi un figlio **R**.

Nella tabella a pagina seguente sono indicati (nella prima colonna) alcuni eventi verificatisi durante l'esecuzione dei programmi da parte di P, Q e R; nella seconda colonna è aggiunta un'indicazione supplementare relativa a tali eventi. **Si completi** tale tabella indicando ordinatamente nella terza colonna tutti i moduli del S.O. che vengono eseguiti (completamente o in parte) in seguito all'evento, nella quarta colonna il contesto nel quale ciascun modulo è eseguito e, nelle ultime tre colonne, lo stato dei processi P, Q e R dopo che tutti i moduli hanno svolto la funzione e si è tornati al funzionamento in modo U.

Notazione per il riempimento della tabella:
Per indicare i moduli eseguiti parzialmente si utilizzi la notazione indicata al sottocapitolo precedente (ad esempio G_SVC_1, G_SVC_2, ecc...), ma si adottino le due seguenti semplificazioni

1. si ometta l'indicazione di preempt quando il quanto non è scaduto e quindi preempt non esegue nessuna operazione effettiva
2. nel caso in cui le 2 parti di un modulo compaiono in sequenza si usi il nome del modulo intero (ad esempio, R_Int invece della sequenza < R_Int_1, R_Int _2>)
3. la sequenza <G_SVC_2, G_SVC_3>, deve essere sostituita da G_SVC_2/3

Avvertenze per il riempimento della tabella:

1. i processi hanno priorità: **P > Q > R > Idle** (P ha priorità max) e non esistono altri processi nel sistema
2. P è stato creato da Idle
3. la dimensione di un **blocco** trasferito in DMA da o su file è di 512 byte
4. per la **lettura e scrittura** su file:
   a) le operazioni di lettura e scrittura su file accedono sempre a disco, cioè è sempre necessario eseguire trasferimenti in DMA
   b) l'interruzione di fine DMA è associata al trasferimento di un singolo blocco del file (evento DMA_in per lettura di un blocco ed evento DMA_out per scrittura di un blocco su file)
5. per l'**apertura** del file:
   c) è **sempre** necessario trasferire un totale di 5 blocchi da disco in DMA
   d) l'interruzione di fine DMA è associata al trasferimento di un singolo blocco (evento DMA_in per lettura di un blocco)
   e) una volta terminata l'operazione di apertura, l'area di memoria centrale viene resa disponibile e quindi una successiva apertura del file richiede nuovamente il trasferimento tramite DMA
6. la notazione "1 interrupt" (2 o 3 interrupt) indica che si sono verificate 1 (2 o 3) interruzioni; nella risposta si faccia riferimento all'ultima di tali interruzioni
7. la chiamata di sistema "wait / waitpid" invoca "Sleep_on" su un evento opportuno
8. la chiamata di sistema "sleep (arg)" sospende l'esecuzione del processo che la invoca per un numero di secondi specificato dal parametro; pertanto "sleep" invoca "Sleep_on" su un evento opportuno e la gestione del tempo trascorso viene eseguita dalla routine di risposta all'interruzione da orologio

Evento (è preceduto dal processo nel cui contesto l'evento si verifica)	**Informazioni aggiuntive**	**Moduli eseguiti per gestire l'evento**	**Processo / i nel cui contesto è eseguito ogni modulo**	**Stato dei processi dopo la gestione dell'evento**		
				P	**Q**	**R**
P: fork	P non ha esaurito il suo quanto di tempo					
P: open	open ha inizializzato il DMA in lettura					
Q: open	open ha inizializzato il DMA in lettura					
n.s.: 5 interrupt da disco (lettura)	trasferiti 5 blocchi in DMA associati alla open di P (l'ultimo è relativo all'ultimo blocco da trasferire)					
P: fork	P ha esaurito il suo quanto di tempo (nota bene: la fork è stata eseguita)					
R: 3 interrupt da disco (lettura)	trasferiti 3 blocchi in DMA associati alla open di Q					
R: 2 interrupt da disco (lettura)	trasferiti 2 blocchi in DMA associati alla open di Q (l'ultimo è relativo all'ultimo blocco da trasferire) **N.B.** Il quanto di tempo di R è scaduto					
P: waitpid						
Q: exit						
P: interrupt da orologio	P ha esaurito il suo quanto di tempo					

Evento (è preceduto dal processo nel cui contesto l'evento si verifica)	**Informazioni aggiuntive**	**Moduli eseguiti per gestire l'evento**	**Processo / i nel cui contesto è eseguito ogni modulo**	**Stato dei processi dopo la gestione dell'evento**		
				P	**Q**	**R**
P: fork	P non ha esaurito il suo quanto di tempo	*G_SVC_1* *fork* *G_SVC_2* *preempt* *G_SVC_3*	*P* *P* *P* *P* *P*	*esec U*	*pronto*	*non esiste*
P: open	open ha inizializzato il DMA in lettura	*G_SVC_1* *open* *Sleep_on (E1)_1* *Change* *G_SVC_2/3*	*P* *P* *P* *P – Q* *Q*	*attesa* (E1)	*esec U*	*non esiste*
Q: open	open ha inizializzato il DMA in lettura	*G_SVC_1* *open* *Sleep_on (E2)_1* *Change* *G_SVC_2/3*	*Q* *Q* *Q* *Q – Idle* *Idle*	*attesa* (E1)	*attesa* (E2)	*non esiste*
n.s.: 5 interrupt da disco (lettura)	trasferiti 5 blocchi in DMA associati alla open di P (l'ultimo è relativo all'ultimo blocco da trasferire)	*R_Int (DMA_in)_1* *Wake_up (E1)* *Change* *Sleep_on (E1)_2* *Open* *G_SVC_2/3*	*Idle* *Idle* *Idle – P* *P* *P* *P*	*esec U*	*attesa* (E2)	*non esiste*
P: fork	P ha esaurito il suo quanto di tempo (nota bene: la fork è stata eseguita)	*G_SVC_1* *fork* *G_SVC_2* *Preempt_1* *Change* *G_SVC_2/3*	*P* *P* *P* *P* *P – R* *R*	*pronto*	*attesa* (E2)	*esec U*
R: 3 interrupt da disco (lettura)	trasferiti 3 blocchi in DMA associati alla open di Q	*R_Int (DMA_in)*	*R*	*pronto*	*attesa* (E2)	*esec U*

Evento (è preceduto dal processo nel cui contesto l'evento si verifica)	**Informazioni aggiuntive**	**Moduli eseguiti per gestire l'evento**	**Processo / i nel cui contesto è eseguito ogni modulo**	**Stato dei processi dopo la gestione dell'evento**		
				P	**Q**	**R**
R: 2 interrupt da disco (lettura)	trasferiti 2 blocchi in DMA associati alla open di Q (l'ultimo è relativo all'ultimo blocco da trasferire) **N.B.**Il quanto di tempo di R è scaduto	*R_Int (DMA_in)_1* *Wake_up* (E2) *R_Int (DMA_in)_1* *Preempt_1* *Change* *Preempt_2* *G_SVC_3*	*R* *R* *R* *R* *R – P* *P* *P*	*esec U*	*pronto*	*pronto*
P: waitpid		*G_SVC_1* *waitpid* *Sleep_on (E3)_1* *Change* *Sleep_on (E2)_2* *Open* *G_SVC_2/3*	*P* *P* *P* *P – Q* *Q* *Q* *Q*	*attesa* (E3)	*esec U*	*pronto*
Q: exit		*G_SVC_1* *exit* *Wake_up (E3)* *exit* *Sleep_on (E3)_2* *Waitpid* *G_SVC_2/3*	*Q* *Q* *Q* *Q – P* *P* *P* *P*	*esec U*	*non esiste*	*pronto*
P: interrupt da orologio	P ha esaurito il suo quanto di tempo	*R_Int (CK)_1* *Preempt_1* *Change* *Preempt_2* *R_Int (DMA_in)_2*	*P* *P* *P – R* *R* *R*	*pronto*	*non esiste*	*esec U*