

Cap. 10

10. PROGRAMMAZIONE CONCORRENTE E SISTEMI OPERATIVI REAL-TIME

10.1 INTRODUZIONE

Gli obiettivi di un sistema di elaborazione possono essere svariati e concomitanti, come ad esempio:

- - semplicità ed economia
- - massimo sfruttamento di alcune risorse (video, dischi)
- - massimo *throughput*
- - adeguatezza alle esigenze di una multiutenza
- - rispetto di vincoli temporali nelle interazioni con l'esterno
- - massima affidabilità

La “personalità” del sistema dipende notevolmente dal tipo di macchina virtuale che il sistema operativo implementa sulla macchina fisica.

Il SW applicativo a sua volta si appoggia al supporto del S.O.

In questa sezione focalizziamo l'attenzione su quei sistemi per cui è qualificante il comportamento temporale, come è richiesto per una corretta interazione con un mondo esterno che è sede di fenomeni asincroni.

È interessante notare che in uno stesso sistema di elaborazione generalmente convivono diverse attività diversamente caratterizzate rispetto agli obiettivi sopra citati. Ad esempio anche in un calcolatore personale monoprogrammato e monoutente dovranno essere svolte anche alcune attività strettamente legate al tempo, quali, ad esempio, le interazioni con le unità periferiche.

In appendice sono sinteticamente richiamati alcuni importanti concetti che dovrebbero essere noti dai corsi di sistemi operativi.

10.2 NECESSITÀ DI PARALLELISMO (Per il tempo reale)

Quali carichi di lavoro/prestazioni richiedono parallelismo fisico
quali accettano parallelismo virtuale (con preemption)
quali accettano pseudo-parallelismo (senza preemption)

Un ambiente di elaborazione in tempo reale è caratterizzato da un carico di lavoro e da vincoli temporali da rispettare perchè l'esecuzione del lavoro sia considerabile come corretta.

Il carico di lavoro di un sistema di elaborazione che debba interagire in tempo reale con fenomeni esterni è scomponibile, come già presentato nel cap. 2, in:

- - attività costituite da azioni periodiche
- - azioni aperiodiche (generalmente sporadiche)
- - attività di sottofondo.

Chiamiamo **scopo temporale** di un'azione l'intervallo di tempo tra l'istante in cui si verifica l'evento (esterno, temporale o interno) che consente o richiede l'esecuzione dell'azione (*triggering event*) e l'istante di tempo in cui tale attività è, o deve (*deadline*) essere, completata.

Chiamiamo **concorrenti** attività le cui azioni hanno scopi temporali che presentano sovrapposizioni (*overlapping*), in contrapposizione al termine **sequenziali** che designa azioni con scopi temporali senza intersezioni. La concorrenza, così definita, deriva dall'interazione con fenomeni esterni tra loro asincroni.

È importante qui ricordare che un'elaborazione ad attività concorrenti non è necessariamente di tipo real-time. Assume la caratteristica di real-time solo un'elaborazione in cui l'inizio e la fine di alcuni o tutti gli scopi temporali delle azioni è vincolata da specifiche temporali. L'istante temporale che non deve essere superato dallo scopo temporale di un'azione è detto "*deadline*" (= scadenza finale).

Si noti che talora il termine *deadline* è usato per indicare l'intervallo di tempo concesso ad un'azione, cioè il suo massimo ritardo accettabile dopo l'evento *trigger*.

I fondamenti della teoria dei processi sequenziali concorrenti e comunicanti (CSP = *Communicating Sequential Processes*) sono per lo più dovuti ai lavori di Hoare (1978).

Chiamiamo **processo sequenziale** un'attività internamente sequenziale e caratterizzata da un proprio **contesto**, cioè l'insieme delle risorse ad essa allocate ed i rispettivi stati.

Le risorse possono essere **attive** (processori di vari tipi) o **passive** (periferiche, memoria, dati, ecc.).

L'esecuzione di un processo richiede una risorsa attiva ed eventualmente un certo numero di risorse passive.

Le definizioni di sistemi di elaborazione in tempo reale implicano che tali sistemi siano dotati di una capacità di **parallelismo** tra diverse elaborazioni, cioè la capacità di iniziare una nuova azione anche se altre sono già in corso.

È intuitivo infatti che la sovrapposizione di scopi temporali di diverse azioni implica la necessità di qualche forma di parallelismo di elaborazione.

Chiamiamo:

- **TD_i** la durata dello scopo temporale dell'azione i-esima (esprime la rapidità **richiesta** al sistema di elaborazione).
- **TE_i** il massimo tempo di esecuzione netto dell'azione i-esima (esprime la velocità **offerta** dal sistema di elaborazione).

Condizione sufficiente perchè sia richiesto un **parallelismo** di esecuzione (eventualmente con *preemption*) è che esistano almeno due eventi tra loro aperiodici (e quindi potrebbero verificarsi anche contemporaneamente) che iniziano gli scopi temporali rispettivamente dell'azione **i** e **j**, tali per cui sia:

$$TD_i < TE_i + TE_j$$

Condizione sufficiente perchè si possa evitare la *preemption*, se TD_j è la più breve *deadline*, è che sia $\Sigma TE_i < TD_j$

Un parallelismo totalmente **reale** (o fisico) è ottenibile solo se sono disponibili tante risorse circuitali, tra cui in particolare i processori (**multiprocessing**), quante sono richieste dalle diverse attività potenzialmente concomitanti.

Questo parallelismo implica un'assegnazione (*scheduling*) **spaziale** delle attività alle risorse.

Ogni risorsa rimane inattiva (*idle*) per tutto il tempo in cui non è richiesto lo svolgimento di azioni dell'attività assegnatale, e ciò può portare ad uno scarso sfruttamento di tali risorse.

Un'applicazione **non è fisicamente realizzabile**, neppure con un sistema a parallelismo fisico, se per almeno un'azione si ha:

$$TE_i > TD_i$$

che corrisponde a potenza di calcolo insufficiente per l'algoritmo adottato, rispetto alle prestazioni temporali richieste.

Molto interessante è il parallelismo **virtuale** (o logico) che è ottenibile con l'assegnazione opportunamente distribuita nel tempo delle (scarse) risorse alle azioni in corso durante intervalli di tempo interni ai rispettivi scopi temporali (**multitasking**).

Questo parallelismo implica quindi uno **scheduling temporale** che consente un migliore sfruttamento delle risorse rispetto ad un parallelismo esclusivamente fisico.

Chiamiamo **coefficiente di utilizzazione del processore** il valore:

$$U = \sum_1^n \frac{TE_i}{TA_i}$$

n = numero di azioni potenzialmente richieste al processore

TE_i = tempo di esecuzione netto dell'azione i-esima

TA_i = intervallo tra le attivazioni dell'azione i-esima.

La realizzabilità fisica con un solo processore impone che sia $U < 1$ che è condizione necessaria, ma non sufficiente, perchè possano essere rispettati tutti i vincoli temporali.

Si noti che con un carico di lavoro imposto esclusivamente da eventi periodici i valori TA_i corrispondono ai periodi dei rispettivi eventi, mentre con eventi sporadici si può parlare di utilizzazione "media" o "di picco" a seconda che per i vari TA_i si adottino i valori medi o minimi di ripetizione del corrispondente evento sporadico.

Un parallelismo che possiamo chiamare "misto" si realizza con un sistema dotato di più processori, ognuno dei quali esegue più attività in parallelismo virtuale.

In questo caso, disponendo di N processori, condizione necessaria per la fisica realizzabilità è che il carico di lavoro globale del sistema, imposto dalle m potenziali azioni, sia

$$G = \sum_1^m \frac{TE_i}{TA_i} < N$$

Torniamo a considerare il caso di un sistema con un unico processore.

Purchè il coefficiente di utilizzo della CPU sia

$$U < 1$$

e lo scheduling temporale sia tale da contenere l'esecuzione di ogni azione all'interno del proprio scopo temporale, l'effetto complessivo osservabile è "equivalente" a quello ottenibile da diversi processori in parallelo.

Ciò è possibile in quanto le **specifiche temporali richieste** riguardano solo certe azioni delle elaborazioni ed in particolare quasi sempre **riguardano solo l'azione conclusiva** di emissione del risultato finale, mentre nulla viene specificato sul comportamento temporale nelle fasi intermedie.

Nella fig.10.1 viene riportato un esempio di uso delle funzioni “utilità temporale della risposta” e “bontà del risultato” presentate nel cap.2, per descrivere l'esecuzione di azioni con scopi temporali sovrapposti.

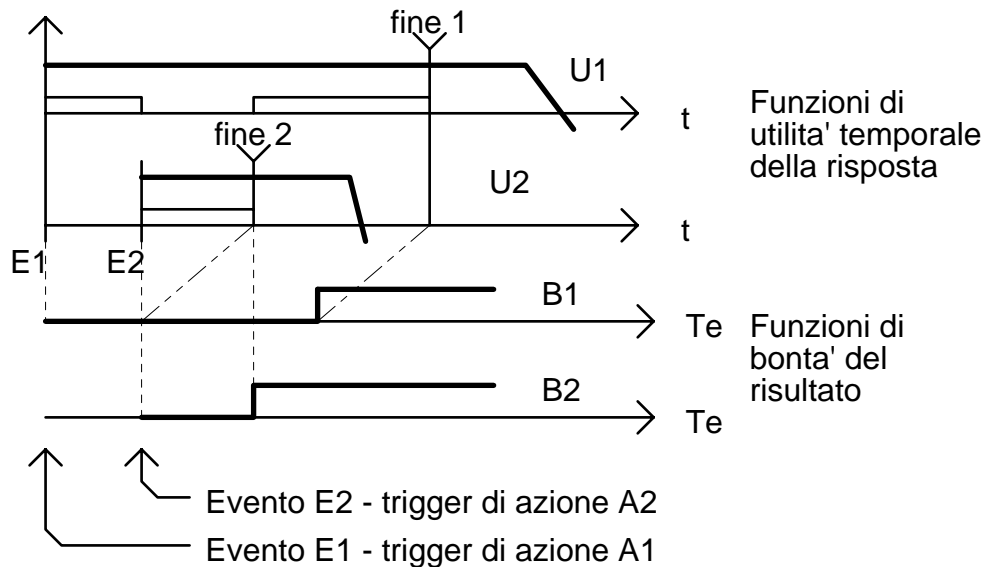


Fig. 10.1 Esecuzione di azioni con scopi temporali sovrapposti.

L'evento E1 inizia l'azione A1. Dopo un certo tempo l'evento E2 provoca l'attivazione dell'azione A2 (con *preemption* di A1) che viene completata entro la sua scadenza. Infine viene ripresa e completata l'esecuzione dell'azione A1 che pure rispetta la sua scadenza.

In questa situazione se A2 fosse stata rimandata alla conclusione di A1 (quindi senza *preemption*), A2 non avrebbe rispettato la sua scadenza.

Si noti che la scala dei tempi rispetto a cui si giudica l'equivalenza di comportamento temporale è quella degli scopi temporali, quindi molto più grossolana di quella in cui si misurano i tempi di esecuzione di singole istruzioni. È proprio questa differenza di granularità temporale che consente la virtualizzazione del parallelismo.

Se il carico di lavoro, con il processore che si intende utilizzare, supera complessivamente il fattore di utilizzazione unitario è inevitabile dover ricorrere ad un

sistema con più processori. Tuttavia il parallelismo virtuale presenta una particolare importanza anche nei sistemi multiprocessore, ed è molto utilizzato per vari motivi che possono essere riassunti nei seguenti punti:

- diventano sempre più economici processori molto più potenti di quanto sarebbe richiesto dalla maggior parte delle singole attività e che quindi sarebbero poco sfruttati da una singola attività;
- le attività assegnate ad uno stesso processore sono favorite nelle reciproche sincronizzazioni e comunicazioni in termini di semplicità ed efficienza, rispetto a quelle assegnate a processori diversi.

Come conseguenza il parallelismo virtuale è generalmente un obiettivo perseguito anche nei sistemi a più processori, eventualmente anche distribuiti, che sempre maggior interesse stanno suscitando per vari motivi che ora non analizziamo.

10.3 MODELLI IMPLEMENTATIVI DI PARALLELISMO VIRTUALE

Si è detto che il parallelismo virtuale si può ottenere dedicando la CPU per opportuni intervalli di tempo alle diverse attività da eseguire. Presentiamo ora brevemente una classificazione delle principali tecniche utilizzate a questo scopo, analizzandone le caratteristiche e le attitudini a realizzare prestazioni di tipo tempo-reale.

Queste tecniche sono realizzate dal nucleo (*kernel*) del Sistema Operativo che è preposto alla gestione dei Processi.

10.3.1 TIME-DRIVEN

Esecuzione ciclica semplice o multipla

Nella maggior parte delle applicazioni è pensabile che le informazioni in ingresso al sistema di elaborazione siano ottenute mediante una **periodica** osservazione (campionamento) di stati continui (analogici) e discreti (digitali), come già visto trattando delle problematiche di interfacciamento.

Con l'approccio *time-driven* ogni attività può essere implementata come una successione di azioni consistenti nella valutazione, ripetuta ad ogni campionamento, di una stessa funzione dei valori catturati in ingresso (I_i) ed eventualmente di uno stato interno (S_i) e che produce un valore di uscita (O_i) e l'eventuale nuovo valore dello stato interno (S_{i+1}).

Si noti che per **stato interno** qui si intende l'insieme di variabili interne i cui valori "ricordano" ciò che è risultato dalle elaborazioni del ciclo attuale e che deve essere utilizzato nel ciclo prossimo, come è necessario per realizzare **funzioni sequenziali** (o con una dinamica) e non solo combinatorie.

Ad esempio in un algoritmo che implementa un PID appartengono allo stato interno la variabile che mantiene il valore precedente della misura, per il calcolo della derivata, e la variabile che mantiene il valore corrente, da aggiornare ad ogni ciclo, dell'integrale.

Assumiamo cioè che un'attività A consista nell'esecuzione in istanti discreti e ad intervalli regolari $T_0, T_1, \dots, T_i, \dots$ delle azioni $a_0, a_1, \dots, a_i, \dots$ che consistono nel valutare la stessa espressione E_a in cui le variabili assumono i valori campionati in quegli istanti.

Questa modellizzazione corrisponde a una rete sequenziale sincrona o, che è poi quasi lo stesso, ad una funzione di trasferimento a tempo discreto.

Supponiamo che si debbano eseguire le attività concorrenti A, B e C con lo stesso periodo di campionamento. È spontaneo organizzare l'esecuzione delle azioni come segue:

- all'istante T_0
 - si utilizzano i valori I_0 e S_0
 - per eseguire le azioni a_0, b_0, c_0
 - che producono i valori O_0 e S_1
- all'istante T_1
 - si utilizzano i valori I_1 e S_1
 - per eseguire le azioni a_1, b_1, c_1
 - che producono i valori O_1 e S_2
- all'istante T_i
 - si utilizzano i valori I_i e S_i
 - per eseguire le azioni a_i, b_i, c_i
 - che producono i valori O_i e S_{i+1}

Implementativamente il verificarsi dell'evento temporale "scadenza dell'istante T_i " attiva l'esecuzione della sequenza di azioni:

- leggi i valori I_i con campionamento in ingresso
- esegui le azioni a_i, b_i, c_i
- emetti i valori O_i appena calcolati e aggiorna lo stato S_i .

Questo approccio è detto "guidato dal tempo" (*time-driven*) perchè tutte le azioni sono eseguite solo in base a **eventi temporali** ed indipendentemente dal verificarsi di eventi interni od esterni.

Si noti che gli eventi esterni giocano in questo caso un **ruolo passivo**, non provocano direttamente nessuna azione, ma possono essere rilevati deducendoli dalle differenze di valori degli stati campionati in ingresso. E' dominante il concetto di **stato**.

Lo **scheduling delle azioni è statico** (*off-line*), dato che il programmatore, con la stesura del programma, decide la sequenza di esecuzione delle azioni nell'ambito del ciclo, e non viene effettuata nessuna *preemption*.

I pregi non trascurabili di questo approccio lo rendono interessante in molte semplici applicazioni e ne hanno fatto il modello di esecuzione tipico dei PLC (*Programmable Logic Controller*). I pregi sono sostanzialmente la semplicità e la predicibilità.

-- **Semplicità**: il supporto "*run-time*" è costituito da un sistema operativo molto ridotto che si limita alle funzioni di lettura delle porte di ingresso, scrittura delle porte di uscita e alla gestione di un temporizzatore di attivazione ciclica.

-- **Predicibilità**: i tempi di esecuzione molto regolari e occupati ad intervalli regolari rendono facile verificare se il fattore di utilizzazione del processore è < 1 . Con tale ipotesi, detto **T_p il periodo di ripetizione** ciclica, il **tempo di risposta T_r** del sistema di elaborazione soddisfa la relazione:

$$T_r < 2 \cdot T_p$$

E' infatti facile verificare che il caso peggiore si ha quando un ingresso cambia subito dopo essere stato "campionato" all'inizio del ciclo i -esimo. Le uscite ottenute alla fine del ciclo i -esimo (cioè dopo il tempo T_p) corrisponderanno ancora al vecchio valore di quell'ingresso, mentre solo alla fine del ciclo $i+1$ (quindi dopo un tempo $2 \cdot T_p$) le uscite terranno conto del nuovo valore.

In realtà la validità di questa relazione è condizionata al verificarsi di ipotesi più restrittive che ora discutiamo brevemente.

Ipotesi a): le azioni a_i, b_i, c_i , ecc. sono tra loro indipendenti

In questo caso nessuna delle azioni utilizza dati prodotti da altre azioni e il tempo di risposta T_r non dipende dall'ordine con cui le azioni sono eseguite all'interno del ciclo.

Ipotesi b): le azioni a_i, b_i, c_i , ecc. presentano relazioni di precedenza rappresentabili con un grafo non ciclico e siano eseguite all'interno del ciclo di ripetizione in una sequenza che non viola alcuna precedenza. Anche in questo caso il tempo di risposta soddisfa la relazione $T_r < 2 \cdot T_p$

Le relazioni di precedenza tra azioni si hanno quando alcune di esse utilizzano dati prodotti da altre. Possiamo rappresentare queste relazioni con lati orientati dal nodo che rappresenta l'azione produttrice dei dati al nodo dell'azione utilizzatrice di tali dati, come in un grafo *data-flow*. In particolare il caso b) richiede che tale grafo sia aciclico e ciò rende **possibile** individuare (e ciò è compito del progettista) almeno una sequenza tale che all'interno del ciclo di ripetizione ogni azione sia eseguita dopo tutte quelle che producono i dati che essa utilizza.

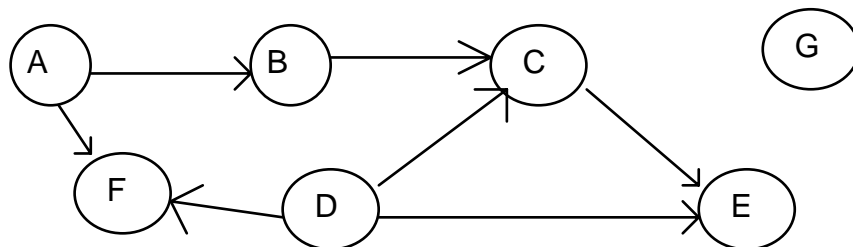


Fig. 10.2 - Esempio di grafo di dipendenza di azioni.

Il grafo della fig. 10.2 è aciclico, e quindi esiste almeno una sequenza di azioni che non introduce ritardi ulteriori. E' facile verificare che soddisfano a questo requisito ad esempio le sequenze:

- A-B-D-C-F-G-E
- D-G-A-F-B-C-E

nelle quali ogni azione è preceduta da tutte quelle da cui riceve dati e seguita da tutte quelle a cui fornisce dati.

Si noti bene che azioni che presentano un grafo di dipendenze **ciclico non** costituiscono un errore di progetto, anzi sono la situazione tipica delle reti sequenziali con retroazione e quindi memoria interna (stati interni). Gli **archi** che rappresentano variabili **di stato interno** da utilizzarsi nel ciclo successivo, **dovranno** partire da azioni eseguite dopo le azioni a cui sono diretti. Semplicemente in questi casi i valori degli ingressi non influenzano solo l'uscita entro il tempo $2 \cdot T_p$, ma anche in tempi successivi.

Come ulteriore spunto di riflessione si lascia al lettore individuare un algoritmo che dato il grafo delle dipendenze e data una sequenza di azioni calcoli quanti cicli di ulteriore ritardo questa sequenza introduce sulle uscite di quali azioni. Ad esempio la sequenza F-A-B-D-C-G-E introduce ritardo **aggiuntivo zero** sulle uscite delle azioni A, B, C, D, E, mentre le uscite di F presentano un **ritardo aggiuntivo Tp**.

Il difetto principale dell'approccio ciclico, che poi è il rovescio della medaglia dei pregi, è costituito dalla notevole rigidità che si può scomporre nei due aspetti seguenti:

- -- Tutte le informazioni di ingresso godono dello stesso trattamento temporale, indipendentemente dalla loro urgenza e criticità, obbligando il progettista ad adottare la cadenza imposta dalle attività più critiche anche per tutte le altre, con uno sfruttamento della CPU lungi dall'ottimale se le attività sono caratterizzate da una notevole dispersione dei valori di urgenza e criticità.
 - -- Tutte le attività devono essere ricondotte alla forma sopra indicata, cioè di ripetizione ciclica degli stessi calcoli. Questo rende piuttosto macchinoso realizzare costrutti di controllo come quelli a cui siamo abituati con i linguaggi di alto livello orientati al software.
- Inoltre si nota facilmente come i principi di alta coesione e basso accoppiamento, caratteristici di una buona modularità, siano decisamente disattesi, almeno nella forma.
- -- Le modifiche richiedono molta attenzione nella verifica che continuino ad essere rispettate le precedenze tra le azioni.

In realtà con questa tecnica di programmazione ciclica si assume, come accennato, una visione più vicina alle strutture hardware, di impostazione "dichiarativa", piuttosto che alle sequenze software generalmente orientate al paradigma "imperativo" (v. 10.3.4). Questo aspetto è considerabile come positivo in certi settori applicativi, tipicamente di semplici automazioni, e per le attitudini dei relativi tecnici e progettisti, come l'evoluzione dei PLC (*Programmable Logic Controller*) ha dimostrato.

Altre considerazioni sull'approccio time-driven verranno fatte nel capitolo dedicato ai PLC.

10.3.2 EXECUTION-DRIVEN MULTITASKING

Coroutine che si chiamano tra loro

Un'altra tecnica relativamente semplice per assegnare la CPU alle varie attività concorrenti consiste nel predisporre un supporto di esecuzione in grado di commutare, su richiesta, dal contesto di un'attività a quello di un'altra e di lasciare al programmatore applicativo la possibilità, e quindi la responsabilità, di invocare queste commutazioni di contesto nei punti "strategici" delle varie attività.

L'implementazione delle azioni componenti le varie attività assume la forma di "**coroutine**", cioè di processi che si "chiamano" reciprocamente, ed i tempi di attivazione derivano da **eventi interni** (le invocazioni di commutazione di contesto) generati in dipendenza dai **tempi di esecuzione** delle azioni stesse.

Come per il caso precedente gli eventi esterni sono **passivi** ed il loro rilievo è affidato a "volontarie" analisi delle variazioni degli stati esterni acquisibili in ingresso.

Il sistema in questo caso non fornisce alcuna garanzia di rispetto di eventuali requisiti temporali, che è invece totalmente affidata alle valutazioni, e verifiche, del programmatore. Per questo motivo questa tecnica viene raramente utilizzata, ed esclusivamente in applicazioni semplici e con requisiti di tempo reale "lascio".

10.3.3 EVENT-DRIVEN MULTITASKING (= DATA DRIVEN)

Processi attivati da eventi (con o senza preemption)

Ruolo e caratteristiche degli interrupt

Nei sistemi detti "reattivi" perchè la loro principale funzionalità consiste nel reagire agli stimoli costituiti da **eventi esterni**, viene naturale associare a tali eventi il compito di **attivare** (*trigger*), o almeno "prenotare" (*release*), le esecuzioni delle azioni che forniscono le "risposte" da essi richieste.

Le attività sono eseguite da procedure SW ben aggregate internamente e tra loro ben disaccoppiate, ed assumono la veste di **processi**, dotate come sono ognuna di un proprio contesto.

Gli eventi esterni sono messi in grado di giocare un **ruolo attivo** generalmente mediante meccanismi di generazione di richieste di interruzione, mediante i quali sono in grado di "forzare" azioni SW che a loro volta possono generare eventi interni di produzione di informazioni e di cambiamento di contesto tra i processi, sia pure con le eventuali relative latenze.

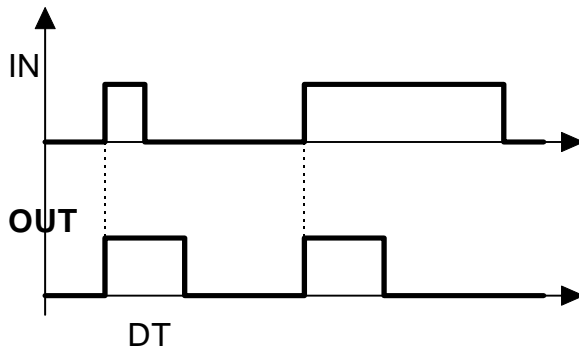
Poichè ad ogni evento (esterno, temporale o interno) sono associate informazioni, questo approccio può essere considerato anche "**data-driven**".

Il ruolo attivo degli eventi esterni, ma anche di quelli temporali, costituisce la caratteristica qualificante di questo approccio, in cui occorre adottare una politica di *scheduling*, sia delle azioni di servizio alle interruzioni che dei processi. Il supporto di esecuzione assume la forma di un vero e proprio nucleo di sistema operativo la cui corretta ed efficiente realizzazione, come è noto, non è banale.

10.3.4 Un semplice esempio

Per confrontare le diverse forme che assuma la stessa funzionalità realizzata secondo le diverse tecniche implementative time-driven e event-driven, presentiamo un semplicissimo esempio.

Supponiamo di voler realizzare la funzione di temporizzazione *non retriggerable one-shot* presentata come **Temporizzazione a durata** nel cap.6, cioè dopo ogni fronte di salita dell'ingresso IN vogliamo generare in uscita un impulso di durata DT, come descritto dalle seguenti forme d'onda.



VERSIONE TIME-DRIVEN (CICLICA TRASFORMAZIONALE)

Supponiamo di usare la variabile PREV (variabile di stato interno) per rappresentare il valore dell'ingresso nel campionamento precedente, e la variabile PRES per il valore di stato appena campionato. Queste variabili serviranno per dedurre gli eventi.

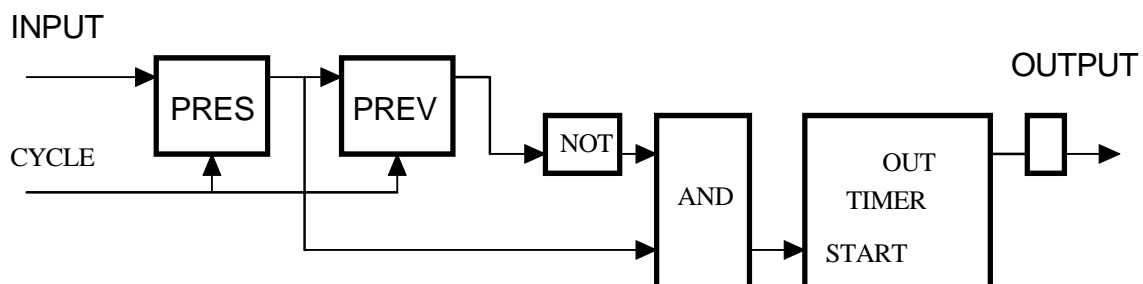
La variabile TIMER.START, che può essere vista come un registro di comando del *timer*, ne lancia la temporizzazione, mentre la variabile TIMER.OUT rappresenta il registro di stato del *timer* che ne fornisce il valore in uscita.

La sequenza è ripetuta ciclicamente con periodo PERIOD.

```

EVERY PERIOD DO
    PRES = IN
    PREV = PRES
    TIMER.START = PRES AND NOT PREV
    OUT = TIMER.OUT
END
    
```

Si noti la facile corrispondenza con una tipica descrizione con paradigma **dichiarativo**, quale è quella di uno schema di tipo “*hardware*” riportata qui sotto.



VERSIONE EVENT-DRIVEN (REATTIVA)

Questo approccio viene riportato nel seguito codificato in un linguaggio “pseudo-C”, nella forma (improbabile ma esemplificativa) a controllo di programma e nella forma ad interrupt e primitive di Sistema Operativo.

Controllo di programma

Si suppone che la funzione input (IN) riporti il valore LOW o HIGH dell'ingresso IN. TIMER è una variabile che vale TRUE durante la temporizzazione di durata DT, gestita da una funzione di sistema attivata con StartTimer (DT)

```
do
{
    while (input(IN) == LOW) {} // ciclo attesa
    StartTimer(DT);
    output(OUT, HIGH);
    while (TIMER) {} // ciclo attesa
    output(OUT, LOW);
    while(input(IN) == HIGH) {} // ciclo attesa
} while (FOREVER);
```

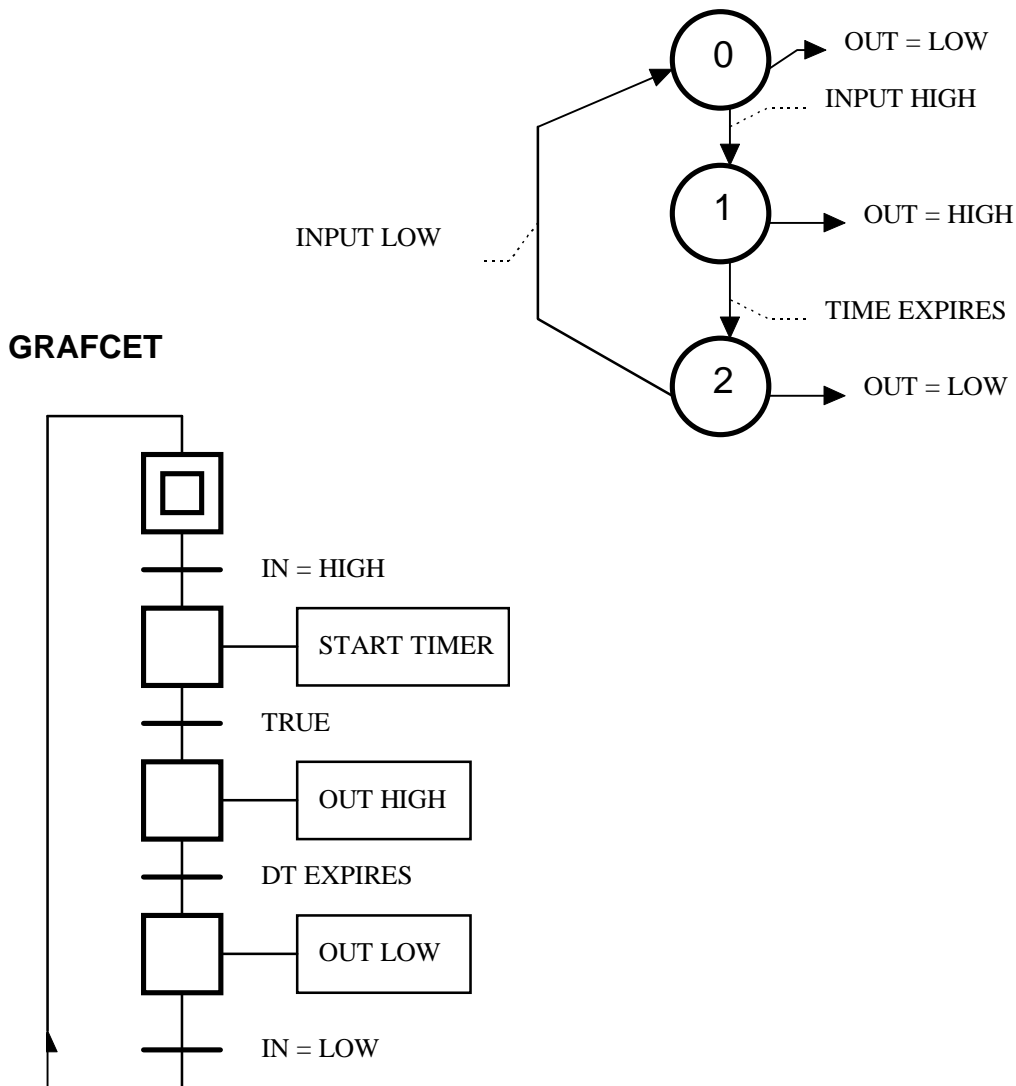
Con primitive (interrupt)

Si suppone che i fronti di salita e di discesa del segnale di ingresso generino delle richieste di interrupt che a loro volta generano (sincronizzazione di secondo livello) gli eventi fronte_salita e fronte_discesa.

```
do
{
    wait (fronte_salita)
    output(OUT, HIGH);
    wait_time (DT)
    output(OUT, LOW);
    wait (fronte_discesa)
} while (FOREVER);
```

Questo approccio, che segue un **paradigma imperativo**, presenta una facile corrispondenza con descrizioni come automa o come Grafcet.

AUTOMA A STATI



10.3.5 Considerazioni sulle tecniche presentate

È opportuno notare che l'approccio guidato dagli eventi costituisce il caso più generale (considerando eventi temporali, esterni e interni) e che, come tale, può supportare al suo interno anche sottoinsiemi di attività impostate secondo gli schemi precedenti (*time-driven* o *execution-driven*), fornendo quindi la massima flessibilità progettuale anche se a costo di una problematica molto più articolata e complessa.

In particolare un'applicazione complessa potrà comprendere:

- - un sottoinsieme di attività cicliche di controllo e supervisione (**trasformazionali**) i cui vincoli temporali consistono sostanzialmente in una attivazione ciclica regolare e perciò ben adatte ad una gestione di tipo *time-driven*;
- - alcune attività di tipo **reattivo** e con vincoli temporali sui tempi di risposta, per cui è conveniente una gestione *event-driven*;
- - delle attività di **sottofondo** senza particolari vincoli temporali e che quindi utilizzano in modo *execution-driven* i tempi lasciati liberi dalle attività precedenti.

Per questi motivi l'approccio guidato da eventi merita un ulteriore approfondimento richiamando ed integrando concetti e meccanismi già presentati, e proponendo un esempio estremamente semplice ma significativo di realizzazione di un nucleo.

10.4 MODELLO A PROCESSI SEQUENZIALI COMUNICANTI

Richiamiamo i principali aspetti del modello tipico di un ambiente *multi-task*, noto dai corsi di sistemi operativi.

I **processi** sono attività dotate di un proprio **contesto** che competono per l'uso della CPU e che possono tra loro **condividere codice, dati** ed eventualmente altre **risorse** del sistema a cui sono allocati.

In alcuni sistemi complessi si distingue tra due tipi di processi.

TASK - Processi dotati di un contesto proprio che comprende anche aree di memoria proprie. Ogni *task* è ottenuto da una separata operazione di collegamento (*linking*):

THREAD - Processi dotati di un proprio *stack*, ma che possono condividere variabili globali, essendo stati collegati nell'ambito della stessa operazione di *linking*.

Un *task* può contenere al suo interno più *thread*.

Nella maggior parte dei casi i sistemi di automazione sono basati su un unico *task* suddiviso in diversi *thread*, e quindi nel seguito i termini **processo** e **task** saranno usati come sinonimi tra loro, con il significato sopra riportato di **thread**, senza ricorrere alla sottile distinzione che può essere utile, lo ripetiamo, solo per sistemi complessi.

La macchina virtuale su cui girano i processi presenta un set di istruzioni costituito dal **set di istruzioni** del processore fisico arricchito dalle **primitive** realizzate dal sistema operativo.

I processi sono caratterizzati nella loro evoluzione da uno **stato interno** e uno **stato esterno**.

- Lo **stato interno** di un processo è gestito dal processo stesso eseguendo istruzioni del set base ed è descritto dal valore del *Program Counter*, dei registri e delle sue variabili private, come per i normali programmi, per cui non ne parleremo ulteriormente.
- Lo **stato esterno** di ogni processo, che ne descrive la situazione rispetto all'ambiente complessivo, è gestito dal Sistema Operativo, in occasione dell'esecuzione di primitive o di interrupt, ed in ogni istante appartiene ad una delle classi sotto indicate.

10.4.1 STATI DEI PROCESSI

- -- INESISTENTE 0

Stato opzionale dei processi destinati ad un calcolatore ma il cui codice principale non è stato ancora caricato nella memoria operativa.

- -- DORMIENTE 1

Stato obbligatorio dei processi potenziali, cioè presenti in memoria ma non ancora dotati di un loro contesto completo e quindi non ancora presi in carico dal sistema operativo (non ancora "creati").

- -- PRONTO (*ready*) **2**

Stato dei processi che dispongono di tutti i requisiti e risorse per essere eseguibili, ad eccezione della CPU di cui sono in attesa.

- -- IN ESECUZIONE (*running*) **3**

Stato del processo che dispone della CPU. Nei sistemi "*preemptive*" il processo in esecuzione ha priorità non inferiore a quella di ogni eventuale processo pronto.

È significativo suddividere questo stato nei due sottostati "in esecuzione modo utente" e "in esecuzione modo sistema".

- UTENTE **3U**

Il processo sta eseguendo procedure applicative.

- SISTEMA **3S**

Il processo sta eseguendo procedure appartenenti al sistema operativo che costituiscono "servizi" (o primitive) che esso mette a disposizione dei processi.

- -- INTERROTTO **4**

In questo stato il processo può essere considerato formalmente in esecuzione, ma in realtà è latente perchè interrotto da una richiesta di servizio di interrupt, in attesa di riprendere l'esecuzione attiva o di subire una *preemption*. Si ricordi che la routine di servizio ISR può produrre eventi o risorse (ad es. dati) che possono portare nello stato *ready* altri processi che ne erano in attesa.

- -- IN ATTESA DI EVENTO/I (*waiting*) **5**

Stato dei processi destinati a reagire ad eventi che non si sono ancora verificati, o che necessitano di risorse non ancora loro assegnate.

Questa categoria comprende in genere diversi stati corrispondenti ai diversi tipi di eventi o risorse di cui i processi possono essere in attesa.

È significativo prevedere il caso di processi in attesa **disgiuntiva** di diversi eventi, ed in particolare di: evento *OR* scadenza temporale (*time-out*).

- -- SOSPESO (*suspended*) **6**

È lo stato in cui possono eventualmente confluire i processi che non competono più per l'uso di risorse nè della CPU. I processi possono passare volontariamente a questo stato se sono definitivamente terminati, oppure possono esservi collocati "di forza" dal sistema operativo in seguito ad eccezioni irrecuperabili.

10.4.2 EVENTI → TRANSIZIONI DI STATO

Le transizioni di ogni processo da uno stato all'altro sono causate da:

- eventi interni (azioni eseguite dal processo in esecuzione),
- eventi esterni (azioni eseguite in risposta ad interruzioni),
- eventi temporali (azioni eseguite in risposta ad interruzioni del dispositivo di temporizzazione al raggiungimento di scadenze).

Un tipico diagramma stati - transizioni dei processi è rappresentato in fig. 10.3.

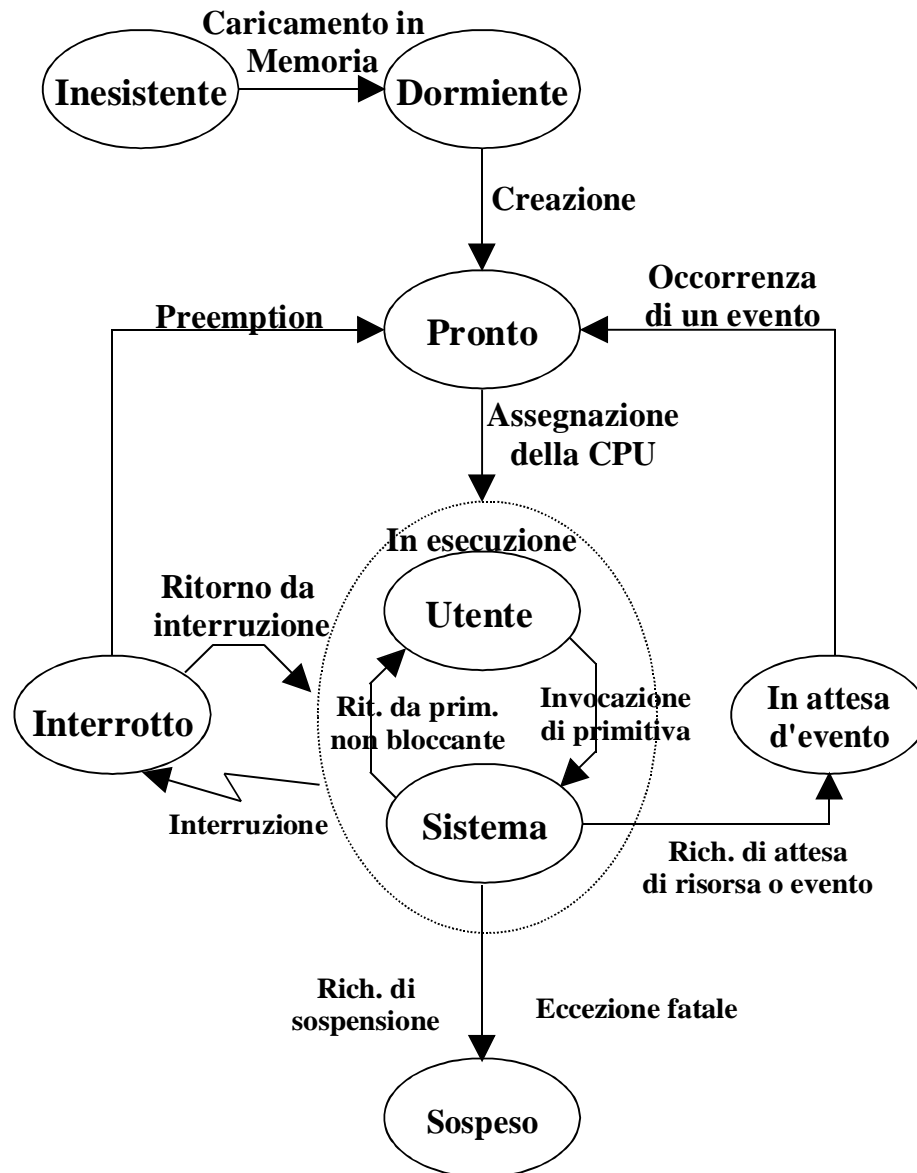


Fig. 10.3 - Diagramma Stati - Transizioni dei Processi.

- -- CARICAMENTO IN MEMORIA **A**

Questa fase non è necessaria per quei sistemi che sono dotati di memorie non volatili (ad es. EPROM) e che spesso sono privi di memorie di massa, come è il caso di molti controllori e sistemi "*embedded*".

- -- CREAZIONE **B**

È una funzione eseguita dal sistema operativo autonomamente in fase di inizializzazione, o su richiesta di un processo in esecuzione. In questa fase viene creato il contesto del processo consistente in un'area di "*stack*" e un descrittore dinamico (talora detto TCB = *Task Control Block*) con opportuna inizializzazione.

- -- ASSEGNAZIONE DELLA CPU **C**

Questa transizione è preceduta da una fase di scelta del processo da attivare tra tutti quelli che eventualmente si trovino nello stato di pronto.

I criteri su cui si basa questa scelta, detta "*scheduling*", risultano molto influenti sulla correttezza del comportamento temporale del sistema e vengono trattati più a fondo nel seguito del capitolo.

- --- INVOCAZIONE DI PRIMITIVA **D**

È l'operazione, detta talvolta *Supervisor Call*, con cui il processo in esecuzione invoca uno dei servizi messi a disposizione dal sistema operativo. Dal punto di vista del processo le primitive sono particolari istruzioni (o procedure) che gli consentono di astrarre dai dettagli che sono eseguiti nel modo *sistema*.

In alcuni sistemi questa invocazione è implementata mediante istruzioni di "*software interrupt*" (quindi "interruzioni sincrone").

Tipici servizi e le relative primitive vengono trattati nel seguito. Qui anticipiamo che alcuni servizi possono completarsi subito, con ritorno all'esecuzione in modo utente, oppure portare il processo in stato di attesa degli eventi che consentano la conclusione del servizio richiesto.

- --- RITORNO DA PRIMITIVA NON BLOCCANTE **E**

Avviene se il servizio può essere completato e se esso non ha comportato il passaggio allo stato di pronto di un processo più prioritario del processo invocante.

- -- INTERRUZIONE **F**

Avviene al termine dell'istruzione corrente (o del gruppo di istruzioni non interrompibili) quando nel frattempo sia stata segnalata una richiesta di interruzione e la CPU sia abilitata ad onorare tale richiesta. Questa transizione si verifica nello stato in esecuzione, sia in modo utente che in modo sistema.

- -- RITORNO DA INTERRUZIONE **G**

È il normale percorso a conclusione di un servizio ad interruzione che non abbia posto in stato di pronto processi più prioritari di quello interrotto, che quindi riprende la propria esecuzione "ignaro dell'interruzione".

- -- PREEMPTION **H**

Si verifica nell'ambito di un servizio di risposta a richiesta di interruzione, durante il quale venga portato nello stato di pronto un processo più prioritario di quello interrotto che a sua volta viene posto nello stato di pronto.

- -- ECCEZIONE FATALE **L**

Le "eccezioni" (anomalie in esecuzione) vengono rilevate con diversi meccanismi che in generale passano il controllo al sistema operativo. Se questo non è in grado di "recuperarle" è costretto a forzare la sospensione del processo responsabile che è generalmente quello in esecuzione.

- -- RICHIESTA DI ATTESA DI RISORSA O EVENTO **M**

Se il servizio invocato comporta la richiesta di attesa di una risorsa non disponibile o di un evento, temporale o esterno, il processo richiedente viene posto in stato di attesa.

- -- OCCORRENZA DI UN EVENTO **N**

Quando si verifica un evento atteso, la disponibilità della risorsa richiesta o una scadenza temporale attesa o di *time-out*, il processo interessato viene posto nello stato di pronto.

- -- RICHIESTA DI SOSPENSIONE **P**

In genere è prevista la possibilità che un processo inoltri al sistema operativo la richiesta di terminare la propria esecuzione. Nei sistemi di automazione questa terminazione non è quasi mai prevista.

10.5 SCHEDULING TEMPORALE - PREEMPTION

L'operazione di *scheduling* consiste nello scegliere quale processo mandare in esecuzione.

- CHI

Si tratta di un'operazione eseguita dal **sistema operativo**, ovviamente quando esso ha il controllo della CPU, il che avviene tipicamente in due diverse circostanze:

- QUANDO

- 1) - in seguito all'invocazione di una **primitiva** da parte del processo in esecuzione (quindi è nel modo sistema);
- 2) - in occasione di risposte a richieste di **interruzione**, nel caso frequente che tali risposte siano parte del sistema operativo stesso. Un caso particolare molto importante è costituito dall'interruzione del RTC (*real time clock*).

- COME

I criteri di scheduling sono realizzati con **opportuni algoritmi** di cui i principali sono presentati nel seguito.

Innanzitutto è molto importante la distinzione tra "*preemptive scheduling*" e "*non-preemptive scheduling*" che si applica ai due possibili diversi comportamenti nel caso 2) sopra citato (interruzioni), come descritto nei punti seguenti.

10.5.1 NON-PREEMPTIVE SCHEDULING

Con questo approccio al termine del servizio di ogni interruzione il controllo viene nuovamente ceduto al processo interrotto mentre era in esecuzione. Come importante conseguenza si ha che ogni processo in esecuzione rimane tale fino a quando "volontariamente" invoca servizi del sistema operativo. Vengono così molto **semplificati i rapporti tra i processi**, dato che i **cambiamenti di contesto** vengono effettuati **solo** in punti ben precisi, appunto in occasione di **chiamate al sistema operativo**.

Un altro pregio è dato dalla minimizzazione dei cambiamenti di contesto, e quindi dell'*overhead* associato, che va in favore del massimo sfruttamento delle risorse e del massimo *throughput*.

Il limite di questo approccio sta nel fatto che le prestazioni di tempo reale devono essere responsabilità del programmatore dei processi, con la necessità di frequenti inserimenti di invocazioni di primitive, e non possono essere garantite dal sistema operativo.

10.5.2 PREEMPTIVE SCHEDULING

Costituisce la soluzione tipicamente adottata nei sistemi operativi per sistemi in tempo reale. In effetti ogni volta che assume il controllo, il sistema operativo ha l'occasione di individuare il processo più opportuno da attivare, in base alle scadenze temporali da rispettare per le varie elaborazioni.

Viene effettuata una **preemption** quando al ritorno da un **interrupt HW** (evento esterno o temporale) l'esecuzione **non** torna al processo interrotto, ma ad uno più **prioritario** diventato ready in conseguenza del servizio dell'interrupt. Quindi la *preemption* può verificarsi in tutte le parti del processo in cui è abilitato l'*interrupt*.

La possibilità di preemption richiede al programmatore applicativo un'analisi **molto più accurata** per assicurare la correttezza nelle mutue interazioni tra processi:

- occorrono operazioni esplicite per rendere **atomiche** (non interrompibili) le "regioni critiche";
- tutte le funzioni (in particolare di libreria) eseguibili da **diversi** processi devono essere implementate in modo **rientrante**.
- se si ha gestione degli interrupt con annidamento, **tutti i vettori di interrupt devono passare attraverso il sistema operativo** e non possono essere connessi direttamente alle routine di risposta scritte dal programmatore applicativo

10.5.3 ALGORITMI DI SCHEDULING TEMPORALE

Gli obiettivi dello scheduling possono essere diversi, come ad esempio:

- - massima semplicità dello *scheduler*
- - massimo *throughput* del sistema
- - *real-time* (massima predicibilità temporale)
- - minimo rischio di anomalie gravi.

Lo studio di algoritmi semplici, efficaci e predicibili, è tuttora molto attivo, soprattutto per le situazioni applicative critiche e complesse. L'analisi approfondita delle numerose proposte della letteratura va oltre gli scopi di questo testo, ma si ritiene opportuno presentare brevemente almeno i più semplici e diffusi algoritmi adottati nei sistemi operativi per applicazioni di automazione.

10.5.3.1 ROUND-ROBIN

I processi sono collocati in una sequenza fissa che viene ciclicamente scandita. Ad ogni operazione di *scheduling* si incrementa l'indice del vettore di processi e viene selezionato per l'attivazione il primo trovato pronto. Giunti in fondo al vettore si riprende dalla prima posizione.

In alcuni sistemi viene assegnato ad ogni processo un quanto di tempo (*time slice*), al termine del quale il processo, se è ancora in esecuzione subisce una *preemption*.

Questa soluzione, tipica anche dei sistemi in *time-sharing*, consente una migliore predicibilità dei casi peggiori dei tempi di esecuzione dei singoli processi.

I tempi massimi di esecuzione possono essere infatti calcolati pensando che ad ogni processo è concessa la frazione di $1/N$ della potenza di calcolo.

Pregi:

- - semplicità e rapidità di esecuzione
- - nessun processo rischia di attendere indefinitamente (*starvation*)

Difetti:

- - nessuna garanzia di tempo reale (tranne che con il *time-slicing*)
- - il trattamento omogeneo non distingue tra attività più o meno importanti o urgenti

10.5.3.2 FIFO

I processi quando diventano pronti sono collocati in fondo ad una coda FIFO (*First In First Out*). L'operazione di *scheduling* consiste nell'estrarre per l'attivazione il primo processo della coda, cioè quello pronto da un tempo maggiore.

Pregi:

- - semplicità e rapidità di esecuzione
- - nessun processo rischia di attendere indefinitamente (*starvation*)

Difetti:

- - nessuna garanzia di tempo reale
- - il trattamento omogeneo non distingue tra attività più o meno importanti o urgenti

10.5.3.3 FIXED PRIORITY (PREEMPTIVE)

Ad ogni processo è assegnato **staticamente** un livello di **priorità**.

I processi pronti sono in una coda **ordinata per priorità decrescenti** e viene attivato il processo (più prioritario) in testa alla coda. Nei casi in cui si prevede la possibilità di più processi con la stessa priorità, si adotta per questi un criterio FIFO.

Pregi:

- - semplicità e rapidità di esecuzione
- - si tiene conto in modo differenziato dell'urgenza o importanza dei processi.

Difetti:

- - la garanzia di tempo reale è affidata alla buona attribuzione delle priorità relative ai vari processi da parte del progettista;
- - eventuali fasi di attività di uno stesso processo, che abbiano una diversa importanza, vengono trattate in modo indistinto;
- - può verificarsi la cosiddetta **inversione di priorità** (*priority inversion*) che possiamo esemplificare come segue. Supponiamo che un processo di priorità alta (A) attenda il rilascio di una risorsa impegnata da un processo a priorità bassa (B). Tutti i processi a priorità media (M) passeranno davanti al

processo B ritardandone il rilascio della risorsa e quindi ritardando indirettamente la possibilità di esecuzione del processo A che invece dovrebbe essere privilegiato rispetto ai processi M.

10.5.3.4 RATE MONOTONIC

Si tratta di un interessante caso particolare di **priorità fissa**, applicabile quando i processi siano caratterizzati da:

- - esecuzione periodica con periodo T_i per l' i -esimo processo
- - scadenza temporale (*deadline*) costituita dal termine del periodo T_i
- - tempo di esecuzione netto C_i eguale, o limitato superiormente, in ogni ciclo.

Un interessante studio [LiL73] ha dimostrato che viene garantito il rispetto delle scadenze di tutti i processi se:

- - le priorità sono assegnate in ordine inverso alle durate dei periodi di attivazione T_i ;
- - lo scheduling è di tipo *preemptive*
- - il coefficiente di utilizzazione della CPU (v. 10.2) è minore di circa 0.7.

Questo algoritmo è **ottimo** (definizione nel seguito).

Si noti che il coefficiente di utilizzazione della CPU nel caso in esame è dato dalla sommatoria estesa a tutti i processi, dei loro coefficienti di utilizzazione parziali che valgono **C_i/T_i** .

Nel caso di processi attivati da eventi sporadici (e quindi non periodici) la situazione è diversa, ma rimane valido il criterio di assegnazione delle priorità in base al minimo intervallo di occorrenza dei rispettivi eventi (caso peggiore).

10.5.3.5 EARLIEST DEADLINE

Questo algoritmo assegna **dinamicamente** la priorità ai processi, considerando di volta in volta più prioritario quello che in quel momento ha la *deadline* **più vicina** nel tempo. Ovviamente lo scheduler deve **conoscere la scadenza** da rispettare per ogni processo pronto.

Quando un processo diventa pronto, se ha la deadline più vicina di quello in esecuzione, passa subito in esecuzione con una *preemption*, altrimenti viene inserito in una coda ordinata per deadline crescenti. Nelle operazioni di *scheduling* viene attivato il primo processo di questa coda.

Questo algoritmo è **ottimo** (definizione nel seguito).

10.5.3.6 SHORTEST SLACK (minimum laxity)

Anche questo algoritmo assegna **dinamicamente** la priorità ai processi, considerando più urgente quello **meno dilazionabile**.

La dilazionabilità (*laxity*) di un processo è calcolata come differenza tra la distanza della sua *deadline* ed il tempo netto di elaborazione che esso deve ancora eseguire prima della scadenza.

Lo *scheduler* deve **conoscere** quindi, oltre alla **scadenza** dei processi pronti, anche i **tempi di esecuzione** di ognuno di essi. Deve inoltre tener nota del tempo di esecuzione concesso ad ogni processo per mantenere aggiornato il conto del tempo di esecuzione rimanente.

Questo algoritmo è **ottimo** (definizione nel seguito).

Definizione: si dice "**ottimo**" un algoritmo di scheduling che garantisce il rispetto di tutte le scadenze in tutte le situazioni in cui è possibile (*feasible*) uno scheduling che porta al rispetto di tutte le scadenze.

10.6 INTERRUZIONI - PRIORITÀ - ANNIDAMENTO

La gestione delle interruzioni assume una particolare importanza nei sistemi operativi *multi-task*, soprattutto se di tipo *real-time*. Per questo motivo riprendiamo qui in esame da un punto di vista più generale questo discorso già trattato con una visione "locale" nel capitolo dell'interfacciamento.

Abbiamo già detto dell'opportunità che tutte le interruzioni, e non solo quelle del *Real Time Clock*, siano gestite dal sistema operativo, in modo da concedergli l'opportunità di effettuare anche le operazioni di transizione di stato dei processi ed in particolare la schedulazione di tipo *preemptive*.

Nella maggior parte dei calcolatori "normali" le interruzioni sono connesse con unità periferiche, le cui modalità di gestione sono standard e note al sistema operativo, che può quindi farsi carico completamente delle operazioni di servizio a tali interruzioni.

Ben diverso è il caso della maggior parte delle applicazioni di automazione in cui il servizio alle interruzioni è fortemente dipendente dai particolari dispositivi e non può quindi essere eseguito da procedure standard del sistema operativo. In questi casi è il programmatore applicativo che deve farsi carico della stesura delle relative routine di risposta (ISR).

Una soluzione a queste **esigenze contrastanti** è ottenibile con il seguente approccio implementativo, esemplificato nel paragrafo seguente.

- - Il sistema operativo si fa carico della fase **iniziale** e **conclusiva** di tutte le routine di risposta ad interruzioni, con porzioni di codice standard, cioè indipendenti dalle particolari operazioni applicative.
- - I processi applicativi interessati possono chiedere al sistema operativo, con opportune primitive, di abilitare particolari livelli di interruzione e **agganciare ad essi specifiche routine applicative**. Naturalmente queste routine vanno scritte tenendo presenti i criteri da adottare nella stesura di risposte ad interruzioni: brevità, limitazione di accesso a risorse, evitare situazioni di blocco, chiamate di sole funzioni rientranti, ecc.

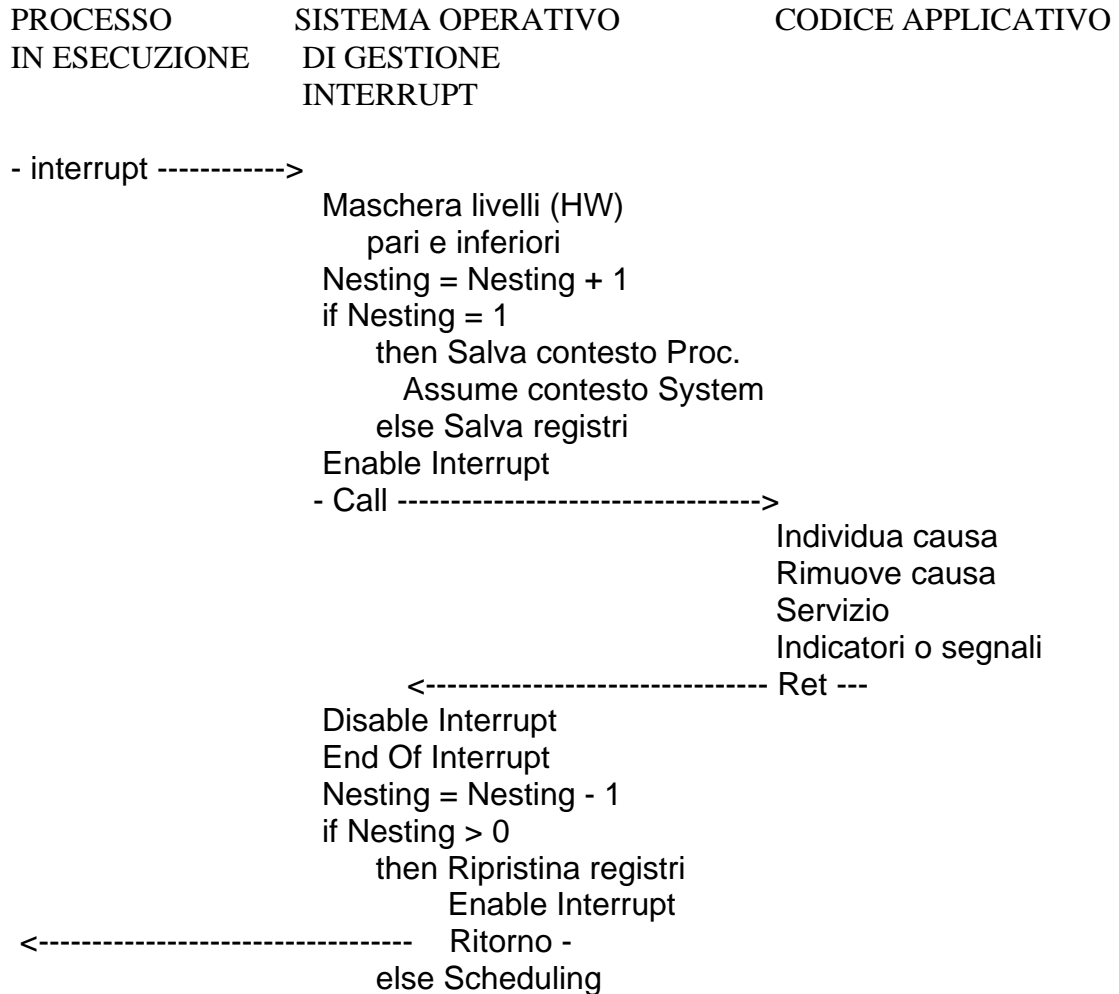
10.6.1 LIVELLI DI ANNIDAMENTO

La soluzione più semplice, e per questo adottata in alcuni piccoli sistemi o quando non vi siano stringenti requisiti temporali, consiste nell'evitare che una routine di servizio di un'interruzione venga a sua volta interrotta. Ciò si ottiene facilmente, dato che in tutti i processori l'attivazione di una risposta ad interruzione contestualmente disabilita la sensibilità della CPU ad altre interruzioni; è quindi sufficiente relegare l'apposita istruzione di riabilitazione (**EI** = Enable Interrupt o **STI** = Set Interrupt flag) alla fine della routine stessa, subito prima dell'istruzione di ritorno (**IRET**).

Nelle applicazioni in tempo reale è però spesso molto significativo tener conto del diverso grado di urgenza associato a diversi servizi, cercando **di ridurre la latenza dei servizi più urgenti**. Per ottenere ciò non è sufficiente assegnare una priorità ai diversi livelli di interruzioni, ma è necessario consentire anche una "*preemption dei servizi*", cioè un annidamento dei servizi di risposta.

Il criterio adottato consiste nel consentire che una risposta ad interruzione sia a sua volta interrompibile da interruzioni più prioritarie, e che solo quando essa dichiara concluso (con apposite istruzioni **EOI** = *End Of Interrupt*) il suo compito, siano accettate interruzioni di pari o inferiore priorità. Si noti che l'istruzione EOI è diretta ad agire sul circuito di arbitraggio delle priorità delle richieste di interruzione, che ovviamente deve essere previsto per la gestione degli annidamenti.

In conclusione una tipica gestione di interruzioni annidate e **con compiti suddivisi tra sistema operativo e parte applicativa** potrà assumere la seguente forma.



Si noti il ruolo della variabile *Nesting* (con classe di allocazione *statica*) che serve a tener conto del livello di annidamento, perchè al ritorno da un'interruzione si dovrà tornare all'esecuzione interrotta in tutti i livelli annidati, mentre **solo al ritorno dal primo livello** si dovrà o potrà fare la *preemption*.

10.7 COMPETIZIONE E COOPERAZIONE

Dal corso di sistemi operativi sono note, almeno a livello elementare, le problematiche di correttezza nei rapporti tra processi concorrenti. Non si intende quindi rifare un'analisi organica di tali problematiche, ma solo puntualizzare alcuni aspetti significativi.

È importante soprattutto evidenziare la differenza tra applicazioni in cui i processi sono prevalentemente indipendenti tra loro e si trovano ad interagire per motivi di "**competizione**" nell'accesso alle risorse, dalle applicazioni i cui processi presentano più o meno strette relazioni di "**cooperazione**".

Il primo caso è tipico di applicazioni "**gestionali**" multiutente o anche a singolo utente. Un esempio potrebbe consistere nella concorrenza tra la stampa del listato di un programma, la compilazione di un secondo programma e l'edizione di un terzo. Si tratta di attività che bene convivono tra loro, dato che usano prevalentemente diverse risorse, e che costituiscono diversi "*job*" (lavori) totalmente indipendenti e privi di aree di memoria operativa condivise e scambio di informazioni.

In ogni caso, e a maggior ragione se si hanno utenti diversi, una barriera di inaccessibilità delle informazioni di altri processi è considerata un pregio, anzi una protezione essenziale.

Il caso di processi destinati a **cooperare** è invece tipico delle **applicazioni di automazione** ed in generale di interazione con fenomeni esterni. In questo caso un unico "*job*" è scomposto in diversi processi (tipicamente di tipo *thread*) principalmente per consentire la concorrenza (parallelismo virtuale) necessaria a supportare interazioni in tempo reale con fenomeni asincroni.

Potremo avere ad esempio un processo che gestisce un'interfaccia di acquisizione dati, un altro che realizza funzioni di controllo, un'altro che interagisce con l'operatore ed infine uno che mantiene le comunicazioni con sistemi remoti. La necessità di comunicazione e sincronizzazione tra tali processi è evidente, come pure è facile intuire l'importanza di tecniche che contemporaneamente garantiscano correttezza ed efficienza.

È soprattutto per questo tipo di applicazioni che sono previste le primitive analizzate nel seguito.

10.7.1 CORRETTEZZA - SINCRONIZZAZIONE - MUTUA ESCLUSIONE

La correttezza di un insieme di processi cooperanti, quindi interagenti, ma anche in competizione per l'uso di risorse, almeno della CPU, richiede che vengano effettuate operazioni di sincronizzazione e mutua esclusione. Inoltre le azioni che vanno eseguite con particolari relazioni rispetto al tempo richiedono una temporizzazione.

La **temporizzazione** consiste nell'eseguire particolari azioni in particolari istanti di tempo assoluto (ad una certa ora) o relativo rispetto ad altri istanti (dopo un certo intervallo di tempo).

La **sincronizzazione** serve a garantire che vengano rispettati i vincoli di precedenza tra le azioni che producono e le azioni che utilizzano informazioni (eventi interni), in modo da realizzare un **parziale ordinamento** temporale tra azioni.

La **mutua esclusione** è necessaria perchè siano svolte in modo corretto operazioni che richiedono l'uso di risorse che in ogni istante possono essere in uso ad un solo processo. Esempi di risorse che richiedono mutua esclusione.

Stampante. Un solo processo può inviare caratteri di stampa. Terminato il modulo, la stampante può essere assegnata ad un altro processo.

Aree di dati in memoria. Un solo processo può utilizzare un'area di dati finchè si verifica una delle condizioni:

- i dati sono stati completamente utilizzati
- i dati sono stati portati in una configurazione consistente
- la memoria viene rilasciata

Procedure e funzioni **non rientranti**. La non rientranza deriva generalmente dall'uso di aree dati **non locali alla singola istanza di esecuzione**.

Le funzioni di temporizzazione, sincronizzazione e mutua esclusione possono essere realizzate in uno dei modi seguenti.

1. Inglobate in un **linguaggio di programmazione** appositamente progettato per la concorrenza (*multitasking*) rendendole così in parte invisibili al programmatore. Con questo tipo di piattaforma il programmatore adotta per lo più una tecnica dichiarativa per esprimere i vincoli tra i processi. Il compilatore tiene conto delle dichiarazioni per inserire le operazioni necessarie per la correttezza. Molte situazioni che

richiedono mutua esclusione possono essere individuate direttamente anche dal compilatore.

2. Eseguite come **primitive** del sistema operativo che il programmatore invoca in modo esplicito. In questo caso l'approccio è di tipo imperativo, il linguaggio usato può essere anche uno di quelli progettati per la normale esecuzione sequenziale, come C e Pascal. Questo secondo approccio è il più diffuso e ulteriormente analizzato nel seguito.

10.8 PRIMITIVE

Generalmente sono dette primitive i servizi che il sistema operativo mette a disposizione dei processi, **sotto forma di procedure considerate elementari** a livello della macchina virtuale vista dai processi.

Come si è accennato, una buona parte delle attività del sistema operativo vengono proprio svolte **all'interno di queste procedure**, mentre le rimanenti funzionalità sono attivate dalle **interruzioni** del Real-Time-Clock e delle interfacce HW.

Come richiamo dai corsi di Sistemi Operativi, vediamo una sintetica rassegna delle primitive tipicamente previste per applicazioni di elaborazione in tempo reale e adatte a supportare applicazioni nell'ambito dell'automazione.

Per le varie primitive si sono adottati nomi inglesi generici, cioè non specifici di un particolare sistema operativo, ma caratterizzanti secondo una terminologia piuttosto diffusa.

10.8.1 PRIMITIVE DI GESTIONE PROCESSI

Queste primitive supportano la corretta esecuzione dei processi con il rispetto di vincoli temporali, di parziale ordinamento delle azioni e di correttezza delle comunicazioni.

Nel seguito viene sinteticamente presentato un possibile insieme di primitive, con cenno ad una possibile semantica. Si noti infatti che i vari nuclei di sistemi operativi possono presentare anche importanti differenze nel set di primitive e nella loro implementazione.

10.8.1.1 SERVIZI GENERALI

- **CREATE (Processo)**

Processo è un **descrittore statico**, tipicamente strutturato a *record*, che contiene i valori degli attributi iniziali del Processo:

- priorità
- ampiezza dell'area di stack richiesta
- entry point
- eventuali risorse private permanenti

L'effetto della primitiva è di assegnare al processo citato come parametro un suo contesto e di inizializzare un suo **descrittore dinamico**, ponendo il processo nello stato di Pronto. Poi, a seconda del sistema operativo, ritorna al processo chiamante oppure chiama lo *scheduler*.

- **SUSPEND**

Questa primitiva pone il processo invocante nello stato Sospeso e cede il controllo allo scheduler per scegliere il processo *ready* da attivare.

10.8.1.2 TEMPORIZZAZIONE

Le funzioni di temporizzazione possono essere classificate nel modo seguente.

Si noti che l'esatta semantica delle primitive di temporizzazione dipende dalla particolare implementazione del sistema operativo e dei dispositivi di temporizzazione (*timer*) disponibili nel sistema (v. cap.6).

- **ABS_WAIT (abs_time)**

Attesa di istanti assoluti. - Pone il processo chiamante in stato di Attesa, e associa al suo descrittore dinamico il valore *abs_time*. Poi cede il controllo allo *scheduler*.

Sotto risposta a interrupt di RTC il S.O. mette nello stato *ready* il processo invocante quando $t = \text{abs_time}$.

- **REL_WAIT (rel_time)**

Attesa di intervalli di tempo. - Pone il processo chiamante in stato di Attesa, e associa al suo descrittore dinamico un temporizzatore impostato sul valore *rel_time*. Poi cede il controllo allo *scheduler*.

Sotto risposta a interrupt di RTC il S.O. mette il processo nello stato *ready* se è trascorso un intervallo di durata pari a *rel_time*.

- **CYC_WAIT (period)**

Riattivazione ciclica e periodica ad ogni intervallo pari a *period*.

- **__TOUT (...tout_time)**

L'attesa con *time-out* è associata all'attesa di altri eventi (vedere primitive di sincronizzazione e di comunicazione).

Il processo invocante chiede di essere comunque svegliato dopo il tempo relativo *tout_time*, se non si verifica l'evento atteso.

È opportuno sottolineare l'importanza di disporre della variante con *time-out* per tutte le primitive bloccanti. In molte situazioni, infatti, il mancato verificarsi di un evento o la mancata disponibilità di una risorsa entro un dato intervallo di tempo, può costituire un'anomalia di cui il processo richiedente deve prendere coscienza per adottare i provvedimenti del caso.

Talora è anche molto utile che il processo che riceve l'evento prima dello scadere del *time-out*, possa conoscere il tempo rimanente, da adottare come scadenza per la successiva richiesta di altri eventi. Ciò consente, ad esempio, di imporre un intervallo di tempo globalmente concesso all'acquisizione di una sequenza di eventi o risorse, come nell'esempio seguente.

```
t_restante = t_tot;
ok = wait (risorsa1, &t_restante);
if (ok)
    ok = wait (risorsa2, &t_restante)
    if (not ok) gestisci_anomalia2();
else
    gestisci_anomalia1();
```

10.8.1.3 SINCRONIZZAZIONE

Le primitive di sincronizzazione sono destinate a realizzare i meccanismi con cui i processi sono eseguiti rispettando i vincoli di parziale ordinamento tra le loro azioni o di **mutua esclusione** nell'accesso a risorse condivise su scala temporale a grana grossa, ma non condivisibili con tempi a grana fine.

L'approccio tipico consiste di primitive di basso livello che si basano sull'uso di semafori.

- **WAIT** (semaphore [,tout])

Attesa della segnalazione, eventualmente con gestione di *time-out*.

Decrementa **semaphore** e se poi è **semaphore** > 0 torna al processo invocante, altrimenti mette il processo nello stato di attesa, indicando nel suo descrittore dinamico il semaforo su cui è in attesa e il tempo di **tout** eventuale.

Sotto risposta a RTC decrementa il tempo **tout** e se questo scade mette il processo nello stato di Pronto. Se qualcuno fa *Signal* sul semaforo mette Pronto il processo. Ovviamente il processo deve poter distinguere tra le due cause di “risveglio” (*signal* o *time-out*).

- **SIGNAL** (semaphore)

Produce una segnalazione. Incrementa **semaphore** e se poi è ≤ 0 mette nello stato di Pronto il primo processo in attesa su quel semaforo.

Poichè la **SIGNAL** non è bloccante, si ha una sincronizzazione lasca, nel senso che il processo che esegue tale primitiva non rimane bloccato in attesa di quello che eseguirà la **WAIT** sullo stesso semaforo. Per questo motivo la **SIGNAL** può essere invocata anche all'interno di una ISR (risposta a interrupt).

Con queste primitive si può usare un semaforo per ottenere la mutua esclusione (*mutex*) inizializzandolo al valore 1 e adottando la sequenza:

```
WAIT (mutex_x)
    operazioni in mutua esclusione
SIGNAL (mutex_x)
```

In alcuni casi sono previste primitive di livello più alto, come ad esempio il **RENDEZ_VOUS** del linguaggio Ada, che crea una **sincronizzazione stretta** tra i due processi richiedente ed accettante che si devono attendere l'un l'altro e ripartono solo ad incontro avvenuto.

10.8.1.4 COMUNICAZIONE

Il corretto svolgimento di comunicazioni tra processi concorrenti all'interno dello stesso processore richiede anche una sincronizzazione tra i processi, oltre naturalmente al supporto dei meccanismi di comunicazione veri e propri, quali accodamenti e ricopiature delle informazioni.

Queste primitive sono quindi in genere considerabili un'estensione delle primitive di sincronizzazione sopra citate, a cui è però associato anche il trasporto di informazioni.

10.8.1.4.1 PRODUTTORE - CONSUMATORE DI EVENTI

Le informazioni considerate **eventi** richiedono la gestione di un meccanismo di tipo produttore-consumatore, senza o con *bufferizzazione*.

- Senza bufferizzazione.

In questo caso è necessaria una sincronizzazione stretta e quindi la comunicazione è basata su *rendez-vous*.

- Con bufferizzazione di messaggi.

La sincronizzazione lasca tra produttore e consumatore richiede un accodamento dei messaggi, generalmente effettuato mediante liste di accodamento FIFO.

- **SEND_MESS** (message, queue)

Invia un messaggio. Accoda il messaggio *message* nella coda *queue* (detta spesso *mailbox*), senza attendere il consumatore. Se la coda è a lista non c'è pericolo di "coda piena", come nel caso di coda basata su *array*.

- **RECEIVE_MESS** (queue, message [,tout])

Riceve un messaggio. Il processo riceve il primo messaggio nella coda *queue*, eventualmente attendendo che ne arrivi uno.

È spesso necessario prevedere una gestione con *time-out*.

- Con bufferizzazione di singoli caratteri.

In questi casi si utilizzano spesso **buffer circolari** basati su *array* con opportuna gestione degli indici di produzione e di consumo.

- **PUT_CHAR** (char, buffer [,tout])

Invia un carattere. Inserisce il carattere *char* nel *buffer* (circolare) se c'è posto, altrimenti attende che si liberi un posto, eventualmente con il tempo limite *tout*.

- **GET_CHAR** (buffer, char [,tout])

Aspetta un carattere. Attende, eventualmente con *time-out*, che ci sia un carattere nel *buffer* (circolare).

10.8.1.4.2 SCRITTORE - LETTORI DI STATI

- Se le informazioni sono di tipo **stato**, si deve instaurare il rapporto scrittore-lettori.

In questo caso non si hanno copie o accodamenti, ma le informazioni sono basate su **variabili globali**. La corretta gestione richiede una **mutua esclusione** tra le operazioni dello scrittore e quelle dei lettori, che generalmente non vengono supportate da specifiche primitive, essendo utilizzabili quelle sopra citate e adatte, appunto, per imporre una mutua esclusione.

10.8.1.4.3 CLIENT - SERVER

Le comunicazioni di questo tipo si svolgono tipicamente in due fasi.

Client produce una richiesta

Server consuma la richiesta

Server esegue il servizio richiesto

Server produce un esito

Client consuma l'esito

Le due fasi possono quindi essere basate su opportune implementazioni del rapporto produttore-consumatore.

10.8.2 PRIMITIVE DI GESTIONE DI RISORSE

Le primitive per la gestione di risorse, memoria, unità periferiche e comunicazioni con l'esterno, sono generalmente di **livello superiore**, e basano le temporizzazioni, sincronizzazioni e comunicazioni necessarie al loro interno, sulle primitive di base sopra riportate.

È significativo evidenziare un problema di potenziale conflitto tra la comodità, per il programmatore applicativo, di disporre di primitive che nascondano tutti i meccanismi di dettaglio e le esigenze di visibilità e controllo di tali meccanismi per applicazioni con stringenti requisiti temporali.

I sistemi operativi adatti al supporto del tempo reale generalmente supportano in modo adeguato almeno la gestione delle risorse "convenzionali", quali memoria ad allocazione dinamica (*heap*), memorie di massa, tastiera, linee di comunicazione, stampanti, ecc.

Le risorse specifiche dell'applicazione in genere vengono invece gestite dal programmatore applicativo usando le primitive (di livello inferiore) precedentemente riportate.

10.8.3 GESTIONE DELLE ECCEZIONI

La gestione delle eccezioni assume una particolare importanza per i sistemi di elaborazione in tempo reale per applicazioni critiche, come è spesso il caso per i sistemi di automazione.

Come si è accennato, questi sistemi devono mantenere una vitalità anche in presenza di errori, tentandone il recupero o almeno il confinamento, e comunque continuando le attività possibili (persistenza, v. cap.1).

Una tecnica spesso usata si basa su una collezione di gestori di eccezioni (*exception handlers*) scritti dal programmatore applicativo, a cui il sistema operativo cede il controllo su invocazione di apposite primitive, in occasione di specifiche interruzioni sincrone (*trap*) o quando rilevi direttamente situazioni anomale.

Tipici problemi sono:

- Spesso le anomalie sono rilevate dalle routine di livello più basso, mentre i provvedimenti vanno presi a livello alto.
- In ambiente multi-task occorrono meccanismi per attribuire ogni anomalia possibilmente ad un ben preciso processo. Questo è il processo *running* se le anomalie riguardano i calcoli in esecuzione, mentre per le anomalie sull' I/O, che si verificano tipicamente nelle risposte ad interrupt, è il processo responsabile dell'attivazione delle operazioni fallite.
- È in genere complessa una buona gestione delle situazioni eccezionali, che generalmente passa attraverso la scelta tra:
 - **abort** - cioè sospendere l'attività fallita (privilegia **l'integrità**);
 - **retry** - cioè rieseguire l'attività dopo eventuali reinizializzazioni, modifiche di contesto, ecc. (compromesso tra integrità e persistenza);

- **ignore** - cioè procedere dopo aver eventualmente "rappezzato" (privilegia la **persistenza**).

- In molti casi è desiderabile che i gestori di eccezione da attivare **dipendano dal contesto** del momento in cui si verifica l'anomalia. Una soluzione interessante è basata su una gestione annidata, in cui ogni processo è corredato di una pila di gestori di eccezioni, in cui a diversi livelli di annidamento delle procedure si inseriscono e si estraggono i gestori adatti.

- Il gestore di eccezione attivato dal sistema operativo deve trovare il contesto (stack e visibilità di variabili) corrispondente al processo e al livello di interventi che deve effettuare.

Tipiche primitive per la gestione di eccezioni sono le seguenti.

- **ON_EXCEPTION** (exception, exc_handler)

Inserisce gestore di eccezione. Primitiva con cui un processo chiede al sistema operativo di creare l'aggancio tra **exception** e il relativo gestore **exc_handler**.

- **RAISE** (exception)

Primitiva di **sollevamento di eccezione**. Invocata dalla routine che rileva l'anomalia *exception*.

Il linguaggio Ada è uno dei pochi linguaggi di programmazione che prevede direttamente una discreta ricchezza di funzioni per la gestione di eccezioni.

10.8.4 DESCRITTORE DINAMICO DI PROCESSO

I sistemi operativi utilizzano vari tipi di **strutture di dati** per descrivere lo stato e il contesto dei processi. Un approccio tipico è basato su una struttura di tipo *record*, spesso chiamata TCB = *Task Control Block*, associata ad ogni processo all'atto della sua creazione e che contiene nei suoi campi le informazioni che caratterizzano l'evoluzione del processo.

Senza riferimento ad alcuna particolare implementazione, riportiamo una breve presentazione di quelle che possono essere le tipiche informazioni contenute in un descrittore dinamico di processo.

10.8.4.1 STATO

Lo stato (esterno, cioè rispetto al SO) del processo può essere descritto in vari modi, eventualmente anche con l'accodamento del TCB in code di attesa di eventi o risorse. Se si utilizza un indicatore di stato è in genere necessario utilizzare informazioni più ricche di quelle schematiche riportate nei classici diagrammi degli stati, come in fig. 10.3. In particolare lo stato di attesa deve essere specializzato con l'indicazione di quale evento il processo attende, e se l'attesa è con *time-out*.

10.8.4.2 TEMPO DA ATTENDERE

Una variabile che indica il tempo di attesa rimanente è necessaria perchè il sistema operativo possa gestire le attese temporizzate o con *time-out*.

Talvolta si adottano variabili distinte per i normali tempi di attesa, per i tempi di *time-out* e per i tempi delle attivazioni cicliche, in modo da consentire particolari gestioni di questi casi, eventualmente con diverse granularità temporali, ecc.

10.8.4.3 INIZIO STACK

L'indirizzo di inizio dell'area di stack assegnata al processo non è a rigore necessario, ma costituisce un'informazione utile in sede di debug o per operazioni di verifica automatica che non si abbiano superi della pila (*stack overflow*).

10.8.4.4 STACK POINTER

Il valore del registro Stack-Pointer costituisce la parte minima di contesto che non può (ovviamente) venire salvato sullo *stack*, come invece avviene generalmente per tutti gli altri registri della CPU.

10.8.4.5 RISORSE

Le risorse allocate al processo sono una parte importante del contesto del processo stesso, e la loro descrizione è estremamente variabile per i vari sistemi operativi.

Nei casi più semplici si hanno nel TCB puntatori alle risorse (o a loro descrittori) quando queste sono in quantità limitata, mentre nei casi più complessi possono essere previste delle liste, la cui testa è prevista in un apposito campo del descrittore dinamico del processo.

Tra le risorse allocate al processo ci possono essere:

- - messaggi ricevuti
- - code private
- - aree di memoria
- - vettori di interrupt
- - regioni critiche impegnate
- - file in uso
- - periferiche condivise

La descrizione delle risorse allocate serve per:

- - verificare i diritti di accesso alle risorse
- - realizzare particolari protocolli di scheduling
- - effettuare i necessari rilasci delle risorse in caso di sospensione dei processi

10.8.4.6 TEMPO DI ESECUZIONE

Uno o più contatori correnti del tempo che il processo trascorre in esecuzione possono servire per:

- - una valutazione del tempo di CPU mediamente utilizzato dal processo, cioè la sua componente del **coefficiente di utilizzazione**;
- - la gestione di un eventuale *time-slicing* per i processi di *background*, che consiste nell'effettuare una *preemption* dei processi che rimangano in esecuzione oltre la porzione (*slice*) di tempo loro assegnata;
- - il calcolo del tempo di esecuzione residuo, necessario per particolari politiche di scheduling, come quella di tipo *minimum slack*.

10.8.4.7 PRIORITÀ

Se la priorità dei processi è staticamente prefissata, come avviene spesso, non è necessario includere tale informazione nel descrittore dinamico. Alcuni sistemi operativi però consentono una modifica dinamica della priorità, che quindi va annotata, per i seguenti motivi:

- i processi possono chiedere che la priorità venga modificata in base all'importanza o urgenza delle azioni che devono di volta in volta eseguire;
- il sistema operativo cambia dinamicamente la priorità dei processi per realizzare particolari politiche di scheduling, come ad esempio per limitare l'inversione di priorità di cui si è accennato precedentemente.

10.9 Appendice

Si riportano nel seguito sintetiche ed informali descrizioni del significato di alcuni termini, su cui è importante avere chiarezza.

FUNZIONE COMBINATORIA - Una funzione il cui risultato dipende solo dagli ingressi (o dai parametri).

FUNZIONE SEQUENZIALE - In contrapposizione a combinatoria. Una funzione il cui risultato dipende non solo dagli ingressi, ma anche dall'evoluzione precedente (storia). Questo tipo di funzioni deve essere dotato di un proprio stato basato su variabili statiche, che contiene l'informazione significativa sulla storia.

VARIABILI AUTOMATICHE - Sono variabili locali di sottoprogrammi, la cui vita termina con il termine dell'esecuzione del sottoprogramma e quindi non mantengono il loro valore tra una chiamata e l'altra, ma anzi devono venire inizializzate ogni volta.

VARIABILI STATICHE - Sono variabili (eventualmente anche locali di sottoprogrammi) che mantengono il loro valore anche tra una chiamata e l'altra. Sono necessarie perchè un sottoprogramma possa avere una sua storia interna (sia sequenziale, in contrapposizione a combinatorio).

STACK - Area di memoria, con gestione LIFO, associata ad ogni processo, in cui possono essere temporaneamente salvati i registri, gli indirizzi di ritorno da sottoprogrammi o da interruzioni e in cui possono essere allocati i record di attivazione dei sottoprogrammi con il relativo spazio per parametri e variabili locali automatiche.

HEAP - Area di memoria globale di un'applicazione, quindi accessibile da tutti i suoi processi, porzioni della quale vengono associate a dei puntatori mediante opportune funzioni (malloc()).

RISORSA - Entità interna (variabile, area dello *heap*, sottoprogramma) o esterna (unità periferica) che un processo può utilizzare come lettore o come lettore-e-scrittore.

CONDIVISIONE STRETTA - Situazione in cui una risorsa viene utilizzata da un processo mentre il suo uso da parte di altri processi non è ancora terminato.

CONDIVISIONE LASCA - Situazione in cui una risorsa può essere usata da diversi processi, ma solo da uno alla volta.

RIENTRANZA - Proprietà di un sottoprogramma di poter essere interrotto e richiamato, mantenendo la correttezza di esecuzione per ogni sua istanza. Si ottiene facendo in modo che ogni istanza possa modificare solo variabili non condivise con altre istanze. Consente la condivisione stretta.

La rientranza pone vincoli più restrittivi della ricorsione, dato che deve poter gestire correttamente l'interruzione in qualsiasi punto (asincrona) del sottoprogramma, e non solo in occasione di un ben preciso punto (sincrona) di chiamata ricorsiva.

CONCORRENZA - Parallelismo virtuale tra processi (attività) le cui azioni possono essere effettuate in modo intercalato le una tra le altre.

COOPERAZIONE - Rapporto tra processi (*thread*) che correlano le loro azioni in modo da perseguire correttamente un obiettivo comune (es. automazione di una macchina).

COMPETIZIONE - Rapporto tra processi (*thread* e *task*) che condividono delle risorse e quindi devono correlare le loro azioni in modo da non interferire tra loro, ed evitando così di compromettere la correttezza delle rispettive elaborazioni.

TEMPORIZZAZIONE - Meccanismo tramite cui si fa in modo che l'esecuzione di un'azione sia correlata con istanti temporali assoluti, o avvenga a distanza temporale relativa rispetto ad un altro evento.

SINCRONIZZAZIONE STRETTA - Meccanismo che assicura un ordinamento tra le azioni di due processi. Ad esempio l'azione B1 potrà essere eseguita solo se l'azione A1 è completata e l'azione A2 a sua volta sarà eseguita solo dopo B1.

SINCRONIZZAZIONE LASCA - Meccanismo tramite cui si assicura un parziale ordinamento temporale tra azioni. Cioè ad esempio si garantisce che l'azione B1 non sia eseguita prima che l'azione A1 sia completata, ma nel frattempo possono essere eseguite anche le azioni A2, A3, ecc..

MUTUA ESCLUSIONE - Tecnica adottata per rendere atomiche (non interrompibili) sequenze di operazioni che operano su risorse non accessibili contemporaneamente, come ad es. i sottoprogrammi non rientranti, per cui è corretta solo una condivisione lasca.

DEADLOCK - (Stallo, blocco critico) Situazione in cui due o più processi non possono più evolvere perchè ognuno blocca parte delle risorse necessarie agli altri.

STARVATION - Situazione in cui un processo può essere impedito dal proseguire per un tempo non limitato, perchè le risorse di cui necessita gli vengono continuamente sottratte da altri processi.

PRIMITIVA - Funzione del sistema operativo, che i processi eseguono nel modo sistema, nell'ambito della cui esecuzione il sistema operativo svolge le sue funzioni, ma che è vista come operazione elementare dal processo applicativo.

PRODUTTORE-CONSUMATORE - Ogni informazione generata dal processo produttore viene successivamente utilizzata, nello stesso ordine, dal processo consumatore. Richiede sincronizzazione per ottenere il parziale ordinamento temporale tra le azioni di produzione e di consumo, con eventuale accodamento FIFO se la sincronizzazione è lasca. Le informazioni sono viste come eventi incrementali. Se le variabili utilizzate per comunicare le informazioni non sono accessibili ad entrambi i processi è necessaria un'operazione di copiatura a carico del S.O.

SCRITTORE-LETTORI - Un insieme di variabili è considerato rappresentativo di uno stato che può essere modificato da un processo scrittore ed utilizzato da più processi lettori. Ogni modifica dei valori da parte del processo scrittore costituisce un evento assoluto, che deve essere reso atomico eventualmente con mutua esclusione. Non è necessario alcun ordinamento temporale tra i vari accessi in lettura o scrittura. Le variabili devono essere accessibili direttamente da tutti i processi interessati.

CLIENT-SERVER - I processi *Client* producono eventi di richiesta di servizi diretti al *Server* in grado di svolgere tali servizi. Il processo *Server* a sua volta produrrà eventi di esito del servizio svolto, diretti al *Client* che l'aveva richiesto.

FARMER-WORKERS - Il processo *Farmer* svolge il ruolo di coordinatore di attività le cui azioni possono essere eseguite in parallelo, affidandone l'esecuzione a diversi processi *Worker*. Il *Farmer* funge quindi da distributore di dati e collettore di risultati. Questo approccio è significativo in sistemi con parallelismo fisico, in cui i processi *Worker* risiedono su diversi processori.

10.10 Esercizi

- Perchè la condizione $T_{di} < T_{Ei} + T_{Ej}$ è sufficiente per richiedere un parallelismo? Perchè non è anche necessaria?.
- Supponendo di realizzare la (banale) funzione $OUT = IN$ mediante ripetizione *time-driven*, e supponendo che IN sia un'onda quadra di periodo T_{IN} e il periodo di ripetizione sia T_p , valutare il *jitter* dell'onda in uscita OUT .
- Relativamente all'esempio del par. 10.3.4 discutere gli errori di temporizzazione sulle uscite, nelle implementazioni *time-driven* e *event-driven*.
- Nell'esempio del par. 10.3.4 perchè si dichiara che la versione a controllo di programma è improbabile? Soprattutto in quali casi?.
- Discutere le relazioni tra *preemption* e rientranza.
- Discutere le relazioni tra interrupt e rientranza e dire se l'annidamento di interrupt ha effetto su questo aspetto.
- Discutere le analogie e le diversità tra esecuzione *time-driven* ed esecuzione a processi schedulati con il criterio *Rate Monotonic*.
- Discutere i vantaggi e gli svantaggi di una gestione di interrupt annidati.
- Discutere quali informazioni è opportuno inserire nel descrittore statico e in quello dinamico dei processi
- Discutere le differenze di comportamento temporale tra le seguenti due implementazioni di un ciclo con periodo (nominale) di 20 unità.

<pre> loop REL_WAIT (20) elaborazioni endloop </pre>	<pre> loop CYC_WAIT (20) elaborazioni endloop </pre>
--	--
- Quando e perchè è necessario disporre della funzionalità di *time-out* associata ad alcune primitive? Quali?.
- Confrontare le realizzazioni del rapporto Produttore - Consumatore senza e con bufferizzazione, rispetto alle problematiche di automazione.
- Perchè le eccezioni sono rilevate ai bassi livelli e vanno gestite ai livelli superiori del SW?.
- Discutere le opzioni *abort*, *retry*, *ignore* nell'ambito della gestione delle eccezioni
- Perchè le funzioni chiamabili da diversi processi, in ambiente di *preemptive scheduling*, devono essere rientranti? E senza *preemption*?.

-- Se una funzione non è rientrante, è corretto eseguirla ad interrupt disabilitati? Ci sono svantaggi con questa soluzione?

-- In una risposta ad interrupt si possono usare funzioni non rientranti? Con quali ipotesi?.

10.11 ALTRE LETTURE

P. Ancillotti, M. Boari

Principi e tecniche di programmazione concorrente.

Ed. UTET Libreria 1987

M. Ben - Ari

Principi di programmazione concorrente e distribuita

Ed. Jackson 1991

Hoare C.A.R.

Communicating Sequential Processes.

Comm. ACM V.21 n.8 Aug.1978

T. Axford

Concurrent programming.

Ed. John WILEY & S. 1989

[LiL73]

C.L. Liu, J.W. Layland

Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment.

JACM - 1973

A. Tanenbaum

Distributed Operating Systems

Ed. Prentice-Hall 1995

10. PROGRAMMAZIONE CONCORRENTE E SISTEMI OPERATIVI REAL-TIME.....	10-1
10.1 INTRODUZIONE	10-1
10.2 NECESSITÀ DI PARALLELISMO (PER IL TEMPO REALE)	10-2
10.3 MODELLI IMPLEMENTATIVI DI PARALLELISMO VIRTUALE.....	10-5
10.3.1 TIME-DRIVEN.....	10-5
10.3.2 EXECUTION-DRIVEN MULTITASKING.....	10-9
10.3.3 EVENT-DRIVEN MULTITASKING (= DATA DRIVEN)	10-9
10.3.4 Un semplice esempio.....	10-10
10.3.5 Considerazioni sulle tecniche presentate.....	10-12
10.4 MODELLO A PROCESSI SEQUENZIALI COMUNICANTI.....	10-13
10.4.1 STATI DEI PROCESSI	10-13
10.4.2 EVENTI → TRANSIZIONI DI STATO.....	10-14
10.5 SCHEDULING TEMPORALE - PREEMPTION	10-17
10.5.1 NON-PREEMPTIVE SCHEDULING	10-18
10.5.2 PREEMPTIVE SCHEDULING.....	10-18
10.5.3 ALGORITMI DI SCHEDULING TEMPORALE.....	10-18
10.5.3.1 ROUND-ROBIN	10-19
10.5.3.2 FIFO	10-19
10.5.3.3 FIXED PRIORITY (PREEMPTIVE).....	10-19
10.5.3.4 RATE MONOTONIC	10-20
10.5.3.5 EARLIEST DEADLINE	10-20
10.5.3.6 SHORTEST SLACK (minimum laxity).....	10-20
10.6 INTERRUZIONI - PRIORITÀ - ANNIDAMENTO	10-21
10.6.1 LIVELLI DI ANNIDAMENTO	10-21
10.7 COMPETIZIONE E COOPERAZIONE	10-23
10.7.1 CORRETTEZZA - SINCRONIZZAZIONE - MUTUA ESCLUSIONE.....	10-24
10.8 PRIMITIVE	10-25
10.8.1 PRIMITIVE DI GESTIONE PROCESSI.....	10-25
10.8.1.1 SERVIZI GENERALI.....	10-25
10.8.1.2 TEMPORIZZAZIONE.....	10-26
10.8.1.3 SINCRONIZZAZIONE	10-27
10.8.1.4 COMUNICAZIONE	10-27
10.8.2 PRIMITIVE DI GESTIONE DI RISORSE	10-29
10.8.3 GESTIONE DELLE ECCEZIONI.....	10-29
10.8.4 DESCRITTORE DINAMICO DI PROCESSO	10-30
10.8.4.1 STATO.....	10-30
10.8.4.2 TEMPO DA ATTENDERE	10-30
10.8.4.3 INIZIO STACK.....	10-31
10.8.4.4 STACK POINTER	10-31
10.8.4.5 RISORSE	10-31
10.8.4.6 TEMPO DI ESECUZIONE.....	10-31
10.8.4.7 PRIORITÀ	10-31
10.9 APPENDICE	10-33
10.10 ESERCIZI	10-35
10.11 ALTRE LETTURE	10-37