Capitolo 2

PROCESSI CONCORRENTI

Sebbene il modello dei grafi di dipendenza dei dati costituisca il modo piú naturale di estendere la programmazione sequenziale verso un'esplicita rappresentazione del parallelismo, la realizzazione effettiva del corrispondente sistema di calcolo si rende necessaria solamente quando si é in presenza di un massiccio parallelismo a livello delle singole istruzioni o, come si dice spesso, parallelismo a *grana fine*.

Se, d'altra parte, si considera un livello di parallelismo *a grana grossa*, dove un'istruzione puó essere costituita da un gran numero di istruzioni semplici, la loro esecuzione si puó ottenere estendendo l'usuale definizione classica di *macchina di Von Neumann* permettendo la coesistenza di un certo numero di *macchine virtuali* sull'unica macchina fisica di Von Neumann disponibile.

2.1 CONTROLLO PARALLELO DELL'ESECUZIONE

Si consideri un grafo di dipendenza dei dati i cui nodi denominano sezioni arbitrarie di istruzioni sequenziali. Per quanto si é visto nel capitolo precedente, il grafo puó contenere uno o piú cicli ed, inoltre, devono essere evidenziati alcuni archi lungo i quali vanno introdotti i *token iniziali* dei dati di ingresso e che sono responsabili dell'attivazione delle varie linee computazionali lungo gli archi del grafo.

A questo punto, con un effetto di moltiplicazione a valanga, il numero dei token attivi all'interno del grafo tende a crescere durante le computazioni evidenziando tutto il parallelismo possibile insito nell'algoritmo ed esplicitato nella struttura a grafo. Naturalmente, se la computazione termina, si deve arrivare ad una situazione in cui il numero dei token prodotti dai nodi istruzione é inferiore rispetto a quelli distrutti determinando una diminuzione del loro numero fino ad arrivare ai *token risultato* che vengono raccolti attraverso lo `switch` e trasferiti in uscita all'anello caratterizzante l'evoluzione computazionale delle macchine dataflow.

Dagli esempi forniti nel capitolo precedente si vede che, tipicamente, le istruzioni *merge* svolgono il compito di attivare il ciclo all'interno del quale sono inseriti. Inoltre, ció puó accadere in computazioni parallele multisequenziali quando un dato di ingresso viene reso disponibile contemporaneamente ad un certo numero di istruzioni.

Considerando il fatto che in entrambi i casi le computazioni vengono attivate per la disponibilitá iniziale di un certo numero di token, eventualmente duplicati nel numero necessario di utilizzo, si puó pensare di evidenziare nell'istruzione dataflow di duplicazione *DUP* la parte relativa al controllo implicitamente definito dalla disponibilitá del token stesso.

In effetti, ciascun *data token* é in grado di evidenziare la parte del controllo rispetto al valore del dato: é sufficiente introdurre, a questo proposito, il token di controllo, definito come un *data token la cui componente relativa al dato é nulla.* Si parla, allora, di controllo parallelo esplicito della computazione e ricompare, nuovamente, la memoria condivisa in cui devono essere memorizzati i dati temporaneamente in uso.

In questo scenario l'istruzione dataflow *DUP* prende il nome di *istruzione* fork e il suo effetto é quello di duplicare il *flusso sequenziale di controllo* che la sta eseguendo. Entrambi i flussi condividono lo stesso dato a meno che non si assegnino rispettivamente due copie dello stesso dato ai due flussi in questione. Si constata immediatamente che l'istruzione *fork* ha la capacitá di estendere la computazione sequenziale di una macchina di Von Neumann alla computazione parallela di due macchine *virtuali* di Von Neumann.

Naturalmente, l'istruzione *fork* non é di per se sufficiente ad estendere la computazione sequenziale in modo da generare un *modello equivalente debole* delle macchine dataflow viste nel capitolo precedente. Si deve poter disporre, a tal fine, di istruzioni che siano in grado di far cooperare attivamente i flussi sequenziali generati dalle istruzioni *fork*. A questo proposito si puó procedere in due modi diversi; nel primo, con l'uso di un'istruzione join si ricongiungono i flussi le cui computazioni parziali contribuiscono a determinare il valore di una certa variabile globale. Il secondo verrá trattato nella sezione seguente.

2.2 MODELLI MULTISEQUENZIALI

Fin'ora si é visto che per realizzare l'esecuzione di un programma in forma di grafo di dipendenza di dati é necessario, identificare nel grafo un *albero* i cui cammini rappresentano *flussi sequenziali* di computazione e i cui nodi multipli, mediante l'introduzione di istruzioni *fork*, aprono nuovi flussi di controllo sequenziale. I restanti archi del grafo che non sono archi dell'albero appena evidenziato, e che si diranno appartenenti al corrispondente *coalbero*, vanno realizzati esplicitamente con una coppia di istruzioni opportune.

In pratica, a ciascun nodo dell'albero da cui diparte un arco del coalbero si affianca un'istruzione send il cui compito é quello di trasferire il token indicato verso la destinazione indicata dall'arco orientato. A sua volta, a ciascun nodo dell'albero, verso cui punta un ramo del coalbero, deve aggiungersi un istruzione recv il cui effetto é quello di trasferire all'interno del flusso sequenziale cui appartiene il valore prodotto da una diversa linea computazionale.

Ciascun ramo del coalbero definisce un canale di comunicazione fra due flussi sequenziali e il coalbero, nel suo complesso, rappresenta l'ambiente di comunicazione nel quale sono immersi i flussi sequenziali definiti dai diversi cammini dell'albero evidenziato nel grafo di dipendenza dei dati.

Quello che si ottiene é un modello computazionale che estende la macchina monosequenziale di Von Neumann verso un insieme di *computazioni multisequenziali di macchine virtuali* di von Neumann, ciascuna con la propria unitá di elaborazione e di memoria centrale, inserite in un ambiente di comunicazione a *scambio messaggi.*

Pertanto, la realizzazione di un modello multisequenziale su una macchina di Von Neumann richiede l'implementazione delle istruzioni fork, send e recv che nel suo insieme rappresenta il nucleo di un sistema operativo. Piú in dettaglio, lo *scheduler* e il *dispatcher* realizzano l'istruzione *fork* i quali si accompagnano sempre ad una qualche forma di *gestione della memoria centrale*, per un trattamento corretto dei dati privati a ciascun flusso sequenziale.

2.3 PROCESSI

I modelli dataflow multisequenziali si basano sulla realizzazione di un certo numero di macchine virtuali monosequenziali a partire dall'unica macchina di Von Neumann disponibile. Ciascuna di queste macchine esegue un unico flusso sequenziale che, per l'importanza che riveste, prende il nome di processo. Nel seguito viene precisato meglio questo concetto.

Si dice che due istruzioni sono eseguite **sequenzialmente** se il completamento dell'esecuzione di una **precede** (nel tempo) l'inizio dell'esecuzione dell'altra. Un **programma sequenziale** é una notazione sintattica che specifica l'*esecuzione sequenziale* di una lista di istruzioni. L'esecuzione di un programma sequenziale prende il nome di **processo**.

Si dice che due istruzioni sono eseguite in **modo concorrente** quando l'esecuzione di una é sovrapposta (o, inframmezzata) all'esecuzione dell'altra ossia, quando l'inizio dell'una avviene prima del completamento dell'altra. Un **programma concorrente** é una notazione sintattica che specifica come due o piú programmi sequenziali possono essere eseguiti in *modo concorrente* come processi paralleli.

Si dice che un programma concorrente é eseguito in **multiprogrammazione** se i processi sequenziali componenti condividono uno o piú processori. Si dice che il programma concorrente é eseguito in **multielaborazione** se viene assegnato un processore a ciascun processo sequenziale componente ed inoltre c'é *condivisione* di un'area comune di memoria. Se i processori sono interconnessi da una *rete di comunicazione* si dice che il programma concorrente viene eseguito in **modo distribuito**. Naturalmente sono possibili anche approcci ibridi, ad esempio, i procesori di un sistema distribuito sono spesso multiprogrammati.

La velocitá di esecuzione dei processi dipende dal tipo di approccio impiegato. Quando i processi vengono eseguiti, ciascuno sul proprio processore privato, la velocitá di esecuzione rimane costante e deve ritenersi generalmente ignota. D'altra parte, se i processi condividono un unico processore, é come se ciascun processo andasse in esecuzione su un processore a velocitá variabile.

In ogni caso, se si vuol comprendere un programma concorrente in termini dei suoi *processi sequenziali componenti* e delle loro *interazioni*, senza fare riferimento al modo con cui sono eseguiti, non bisogna fare alcuna ipotesi sulla velocitá di elaborazione dei processi in esecuzione concorrente, tranne il fatto che siano tutte *positive*. Questa assunzione é nota anche come **ipotesi dell'avanzamento finito**. Soltanto se si soddisfa tale ipotesi la correttezza di un programma viene ad essere indipendente dal sistema con il quale il programma é eseguito.

2.4 INTERAZIONE FRA PROCESSI

I processi sequenziali che compongono l'esecuzione di un programma parallelo si dicono anche **cooperanti**. Per poter cooperare tali processi devono *comunicare* e *sincronizzarsi.* La comunicazione fa si che l'esecuzione di un processo possa influenzare l'esecuzione dell'altro.

Si dice che la comunicazione fra processi si basa su un meccanismo

- **control driven** se utilizza *variabili condivise* che possono, quindi, essere riferite da piú di un processo;

- **data driven** se utilizza lo *scambio messaggi* che permette ai valori di variabili riferite da un processo di essere trasferiti a variabili riferite da un altro processo.

La *sincronizzazione* é necessaria quando due o piú processi vogliono comunicare, in quanto la velocitá di esecuzione per ciascuno di essi non puó essere predefinita a priori. Per **comunicare** un processo deve eseguire qualche *azione* che un altro processo é in grado di riconoscere; ad esempio, assegnare un *particolare valore ad una variabile condivisa*, oppure *inviare un messaggio.* Questo é possibile soltanto se si impone agli eventi

(*i*) *esegui un'azione*
(*ii*) *riconosci l'azione*

di verificarsi esattamente in quest'ordine cosicché la *sincronizzazione* é riconducibile ad un **insieme di restrizioni** sull'ordine degli eventi per soddisfare i quali é necessario introdurre opportuni meccanismi di sincronizzazione che ritardino l'esecuzione dei processi.

L'analisi della *sincronizzazione* da questo punto di vista parte da un **approccio operazionale** alla semantica dei programmi. L'esecuzione di un programma concorrente viene vista come una sequenza di azioni atomiche, ciascuna delle quali é determinata dall'esecuzione di *operazioni indivisibili* la quale potrá comprendere alcune sovrapposazioni di sequenze di azioni atomiche generate dai singoli processi componenti. Molto raramente tutte le possibili sovrapposizioni daranno luogo ad un esecuzione acccettabile del programma concorrente.

L'**approccio assiomatico**, invece, fornisce un metodo alternativo in cui collocare il *ruolo della sincronizzazione.* Secondo questo punto di vista vengono introdotti un certo numero di assiomi e di regole di inferenza per definire la semantica delle istruzioni[1]. I teoremi di questa logica hanno tutti la forma

$$\{P\} \quad S \quad \{Q\}$$

la quale specifica una *relazione* fra l'insieme delle istruzioni S e due predicati, la precondizione P e la postcondizione Q. La scelta degli assiomi e delle regole di inferenza é tale da verificare la seguente condizione

Se si iniziano ad eseguire le istruzioni S a partire da uno stato che soddisfa le precondizioni P e se l'esecuzione termina, allora lo stato raggiunto soddisfa le postcondizioni Q.

Secondo questa interpretazione le istruzioni giocano il ruolo di relazioni fra predicati.

La *presentazione di una prova* consiste di un testo di programma le cui righe sono inframmezzate da asserzioni tali che, per ogni istruzione S, la tripla formata da

(*i*) *l'asserzione che precede testualmente S,*
(*ii*) *l'istruzione S,*
(*iii*) *l'asserzione che segue testualmente S,*

é un teorema della logica della programmazione. In questo modo, la presenza di di un'asserzione R nella presentazione della prova significa che R é vera per lo stato di esecuzione del programma quando il controllo raggiunge quel punto.

Quando é possibile un'*esecuzione concorrente*[2], la prova di correttezza di un processo sequenziale é valida solamente se l'esecuzione di un altro processo non falsifica le asserzioni che compaiono nella dimostrazione. Un modo di ottenere questo risultato é di assumere che

il codice fra due asserzioni consecutive é eseguito in modo indivisibile, dimostrando una serie di teoremi che mostrano come nessuna istruzione di un processo falsifica le asserzione presenti nella dimostrazione di un altro processo.

Questi teoremi addizionali costituiscono quello che si chiama una *dimostrazione di non interferenza.*

Pertanto, i *meccanismi di sincronizzazione* controllano l'interferenza in due modi:

- **ritardano** l'esecuzione di un processo finché una data *condizione*[3] *é vera.* Cosí facendo si

[1] sistema logico-formale noto come *logica della programmazione*
[2] o, *parallela*
[3] o, asserzione

assicura che la precondizione dell'istruzione successiva é vera purché non vi sia interferenza con altre asserzioni.

- **garantiscono** che un blocco di istruzioni sia un'*operazione indivisibile*. Questo elimina la possibilitá che istruzioni di altri processi interferiscano con le asserzioni che compaiono nella dimostrazione di quel blocco di istruzioni.

Entrambi gli approcci, operazionale e assiomatico, sono utili. Il primo é particolarmente adatto a definire il modo con cui funzionano i meccanismi di sincronazzione e sará usato diffusamente nel resto di questa breve rassegna.

2.5 MECCANISMI DI CREAZIONE DEI PROCESSI

L'esecuzione concorrente di due o piú istruzioni[4] puó essere caratterizzata sintatticamente in diversi modi i quali possono classificarsi in base alla capacitá di separare la *definizione* dei processi dalla loro *sincronizzazione*. Imponendo opportune restrizioni sintattiche alla struttura di un programma concorrente é possibile identificare facilmente le sezioni di programma che possono essere eseguite concorrentemente.

Ciascun costrutto puó essere usato per caratterizzare computazioni aventi un numero fissato, o **statico**, di processi oppure in combinazione con *meccanismi di creazione di processi* per dar luogo a computazioni aventi un numero variabile, o **dinamico**, di processi.

2.5.1 Coroutine

Si dicono **coroutine** due o piú procedure nelle quali il trasferimento del controllo avviene in modo simmetrico anziché strettamente gerarchico, eseguendo l'istruzione *resume* la quale é realizzata in modo da salvare abbastanza informazione sullo stato della coroutine da permettere, al momento del ritorno del controllo, di continuare l'esecuzione all'istruzione immediatamente successiva all'istruzione *resume*[5].

Quando si vuole restituire il controllo alla coroutine chiamante si esegue ancora un'istruzione *resume* al posto di un'istruzione *return* come accade nel caso di una procedura. Inoltre, qualunque altra coroutine puó restituire il controllo alla prima coroutine per il semplice fatto che *resume* é l'unico modo con cui due coroutine possono scambiarsi il controllo e ciascuna coroutine puó sceglierne qualunque altra come successiva ad essere riattivata.

Ciascuna coroutine puó essere vista come l'implementazione di un processo in cui l'esecuzione di **resume** dá luogo ad una sincronizzazione fra processi. Usate con attenzione le coroutine rappresentano un modo accettabile di organizzare i programmi concorrenti che condividono un singolo processore. La multiprogrammazione puó essere implementata usando coroutine sebbene queste non siano adeguate a definire una *reale elaborazione parallela*. In essenza, le coroutine sono processi concorrenti caratterizzati da un *controllo esplicito* nello scambio di contesto fra processi, piuttosto che essere lasciato a discrezione dell'implementazione.

2.5.2 Istruzioni Fork e Join

L'istruzione **fork**, analogamente alle istruzioni *call* o *resume*, determina l'inizio dell'esecuzione della procedura designata come argomento con la specifica supplementare che sia la procedura chiamante

[4]o, processi

[5]naturalmente, la prima volta che si esegue resume su una coroutine, il controllo passa alla sua prima istruzione

che quella chiamata procedono in modo concorrente. Per sincronizzarsi con l'ultima istruzione della procedura chiamata, il chiamante esegue l'istruzione **join** il cui effetto é quello di ritardare il chiamante fino a quando non ha terminato l'esecuzione. L'uso attento e disciplinato di *fork* fornisce un metodo semplice e diretto per la creazione dinamica di processi, compresa l'attivazione multipla del medesimo testo di programma.

2.5.3 Istruzioni Cobegin e Coend

L'istruzione **cobegin** é un metodo strutturato per denotare l'esecuzione parallela[6] di un insieme di istruzioni. L'esecuzione di

cobegin
$S_1; S_2; ...; S_n$
coend

termina solo quando l'esecuzione di tutte le istruzioni S_i é terminata. Benché l'istruzione cobegin/coend non ha la stessa potenza delle istruzioni fork/join perché, in effetti, corrisponde alla seguente lista di istruzioni

		K := n;
		fork L_2; .... ; **fork** L_n;
		S_1; **goto** L_{n+1};
L_2	:	S_2; **goto** L_{n+1};
		.....
L_n	:	S_n ;
L_{n+1}	:	**join** K;

é, comunque, sufficiente a specificare la maggior parte di computazioni concorrenti. Inoltre, la sintassi di cobegin/coend esplicita le procedure che devono essere eseguite in modo concorrente fornendo una struttura di controllo con un singolo punto di ingresso e un singolo punto di uscita.

Senza l'introduzione di ulteriori processi o di tempi d'attesa, *cobegin* e la *sequenza* possono rappresentare solamente grafi di precedenza serie-parallelo propriamente annidati. Usando *fork/join* é possibile, invece, specificare direttamente la computazione rappresentata da qualunque grafo di precedenza. Inoltre, **fork** puó essere utilizzato per creare un numero arbitrario di processi concorrenti mentre **cobegin** puó essere usato solamente per attivare un numero fissato di processi.

2.6 IMPLEMENTAZIONE DELLE COROUTINE

Si é visto nella sezione precedente che il meccanismo delle coroutine permette di realizzare una forma debole di parallelismo fra processi che possono essere eseguiti su un unico processore. In questo caso, se si esclude il trattamento delle interruzioni[7] é sufficiente realizzare un supporto *run-time* in cui lo scambio di contesto fra processi viene programmato esplicitamente, come nel caso del Modula, oppure viene realizzato automaticamente allo scadere di un prefissato *intervallo di tempo*, come nel caso del Concurrent Pascal.

[6] concorrente
[7] l'unico insieme di eventi asincroni che puó influire sulla normale esecuzione dei processi

2.6.1 Il Supporto Run-Time del Modula

L'ambiente di esecuzione delle coroutine nel caso del Modula permette di attivare, al livello del programma principale, un certo numero di procedure[8] come processi sequenziali concorrenti. A tal fine, é necessario disporre delle due procedure predefinite

procedure NewProcess(**procedure** P; Base : Address; Size : **integer**;
var Pid : Process); **external**;
procedure Transfer(**var** From, To : Process); **external**;

la prima delle quali avente il compito di trasformare la prima chiamata alla procedura *P* nell'attivazione di una coroutine, la seconda di consentire il trasferimento esplicito dalla coroutine *From* alla coroutine *To* con tutti i salvataggi e i ripristini di contesto necessari[9].

A partire da questo livello base si costruisce un livello di manipolazione dei processi, nel quale il trasferimento del controllo fra una coroutine e l'altra viene realizzato soltanto quanto il processo corrente é impossibilitato a continuare la sua normale esecuzione perché deve attendere il verificarsi di una condizione legata all'esecuzione di un altro processo. A tal fine si procede secondo lo schema indicato di seguito.

Si definisce come *Signal* un sequenza di descrittori di processo[10] in modo tale che ciascuno di essi mantiene

- l'informazione che il processo é pronto a proseguire,
- l'indirizzo della prossima istruzione da eseguire quando il processo riprende l'esecuzione[11],
- il prossimo descrittore di processo facente parte dell'anello[12] di tutti i processi correntemente presenti nel sistema,
- la sequenza di tutti quei processi che sono in attesa di verificarsi di una certa condizione[13]

.

Si supponga, per semplicitá, che i descrittori di processo una volta introdotti nella coda circolare vi rimangano per tutto il tempo di esistenza del processo.

A partire dalle seguenti definizioni di tipo e dichiarazioni di variabili

type Process = ↑**integer**;
Address = **integer**;

Signal = ↑ProcessDescriptor;

ProcessDescriptor = **record**
Ready : **boolean**;
Proc : Process;
Next : Signal; { ring }

---

[8]le *coroutine* del sistema

[9]Si noti che *Base* rappresenta l'indirizzo base e *Size* la dimensione dello spazio di memoria necessario per l'esecuzione della procedura *P* come coroutine

[10]*ProcessDescriptor*

[11]si supponga a questo proposito che il tipo *Process*, definito come ↑**integer**, sia adatto allo scopo

[12]*Ring*, o coda circolare

[13]o *Event* e, pertanto, identificabile, con un'opportuna variabile di tipo *Signal*

```
                    Event : Signal { waiting queue }
                  end;

    var       Ring : Signal; { current process }

  procedure Allocate; external
```

é immediato implementare la procedura *StartRing* che ha lo scopo di inizializzare, con un descrittore di processo nullo, l'anello dei processi identificato dalla variabile globale *Ring*.

```
procedure StartRing( var Ring : Signal );
begin
  new( Ring );
  with Ring↑ do
    begin
      Next := Ring;
      Ready := false;
      Event := nil
    end
end;
```

L'operazione che segue introduce un nuovo descrittore nell'anello dei processi presenti nel sistema, attiva la procedura come coroutine trasferendone il controllo.

```
procedure StartProcess( procedure P; a : Address; n : integer );
var Curr : Signal;
begin
  Curr := Ring;
  new( Ring );
  with Ring↑ do
    begin
      Next := Curr↑.Next;
      Ready := true;
      Event := nil
      Curr↑.Next := Ring;
    end;
  NewProcess( P, a, n, Ring↑.Proc );
  Transfer( Curr↑.Proc, Ring↑.Proc )
end
```

Ad essa si devono aggiungere le due procedere di Enqueue e Select il cui effetto é, rispettivamente, quello di *accodare* un nuovo processo nella coda di processi sospesi specificata e di *selezionare* un nuovo processo da mandare in esecuzione dalla coda dei processi attivi.

```
procedure Enqueue( var Curr, Ring : Signal );
var t0, t1 : Signal;
begin
  t0 := Pid;
```

```
        t1 := t0↑.Event;
        while ( Pid <> nil ) do
            begin
                Ring := Pid;
                with Ring↑ do
                    begin
                        t0 := t1;
                        t1 := t0.Event;
                    end
                Next := Curr↑.Next;
                Ready := true;
                Event := nil
            end;
        t0.Event := Ring
    end;

    procedure Select( var Curr, Ring : Signal );
    begin
        Curr := Ring;
        Ring := Ring↑.Next;
        while not Ring↑.Ready do
            begin
                if ( Ring = Curr ) then halt;
                Ring := Ring↑.Next
            end
    end;
```

Le operazioni che gestiscono da programma lo scambio di contesto fra processi, sospendendo la coroutine corrente per attivarne un'altra fra le possibili candidate nella coda dei pronti, verranno trattate nel capitolo riguardante i monitor.

2.6.2 Supporto Run-Time del Concurrent Pascal

Poiché le coroutine sono processi che possono essere eseguiti uno per volta sull'unica CPU disponibile é necessario realizzare un supporto esecutivo in cui se un processo é in esecuzione, tutti gli altri devono attendere il loro turno nella coda dei processi pronti. Fissato un certo intervallo di tempo[14] al termine del quale l'elaboratore sospende l'esecuzione del processo corrente, il cambio di contesto esecutivo[15] dá l'illusione che tali processi vengano eseguiti simultaneamente.

A tale scopo é necessario rappresentare i processi utilizzando un descrittore in modo tale che, quando un processo perde il controllo della CPU, tutti i suoi registri vengono salvati nel descrittore cosicché questi possono essere ristabiliti quando il processo riprende l'esecuzione.

Si supponga, ora, di definire il tipo *Process* come l'indirizzo della struttura[16] che contiene i valori dei registri del processo e si introduca il tipo *Queue* come una sequenza di descrittori di processo identificabili, ciascuno, col relativo indirizzo. Per essi si hanno le seguenti definizioni di tipo e dichiarazione di variabili

[14] o, quanto di tempo
[15] riprendendo l'esecuzione di un altro processo
[16] identificata, nel seguito, con *Image*

```
type    Image = ...;
        Process = ↑Image;
        Queue = ↑ProcessQueue;

ProcessQueue = record
                  Proc : Process;
                  Next : Queue
               end;

var   Running : Process;
        Core : Image;
```

Si supponga, inoltre, che la struttura dati *Queue* sia manipolabile soltanto dalle seguenti procedure:

```
procedure Create( var Qu : Queue ); external;
procedure Insert( var Qu : Queue; Pid : Process ); external;
procedure Remove( var Qu : Queue; var Pid : Process ); external;
function IsEmpty ( Qu : Queue ) : boolean; external;
```

il cui significato é quello che si deduce direttamente dai loro nomi. Il primo livello di gestione dei processi si ottiene introducendo due variabili globali *Running*[17] e *Core*[18] in modo tale che la loro manipolazione avvenga solamente con le seguenti procedure

```
procedure Serve( Pid : Process );
begin
   Running := Pid;
   Core := Running↑
end;

procedure Preempted( var Pid : Process );
begin
   Running↑ := Core;
   Pid := Running;
   Running := nil
end;
```

per riprendere o bloccare, rispettivamente, l'esecuzione di un processo. La coda dei processi pronti viene rappresentata dalla variabile globale *Ready* su cui sono realizzate le due operazioni seguenti

```
procedure Activate( Pid : Process );
begin
   Insert( Ready, Pid )
end;

procedure Select;
var Pid : Process;
```

[17]per rappresentare il processo correntemente in esecuzione
[18]che individua il contenuto dei relativi registri

```
begin
    while IsEmpty( Ready ) do;
    Remove( Ready , Pid );
    Serve( Pid )
end;
```

la prima utilizzata per inserire un nuovo processo, la seconda per estrarne uno da mandare in esecuzione, purché questa non sia vuota.

Inoltre, l'interruzione di orologio non ha alcun effetto se il processore é inattivo; altrimenti blocca l'esecuzione del processo corrente, introduce il suo descrittore nella coda dei pronti e ne seleziona un altro per l'esecuzione. Tale operazione é gestita dall'operazione implementata nel seguito.

```
procedure ClockInterrupt;
var Pid : Process;
begin
    if Running <> nil then
        begin
            Preempted( Pid );
            Activate( Pid );
            Select
        end
end;
```

Per quanto riguarda la gestione esplicita delle scambio di contesto fra coroutine, questa verrá trattata piú diffusamente nel capitolo riguardante i monitor.