Capitolo 1

Introduzione

Il paradigma della programmazione sequenziale rispecchia la struttura ed il funzionamento della Macchina di von Neumann che, a sua volta, affonda le proprie origini nella ben nota *Macchina di Turing.* Si ricorda, a questo proposito, che l'esecuzione di un programma per una Macchina di von Neumann é basato su due registri fondamentali, il *program counter* **PC** e l'*instruction register* **IR** i quali, ad ogni ciclo *fetch-execute*, permettono di ottenere l'istruzione corrente per la successiva decodifica del *codice operativo* e attivarne la corrispondente rete logica di esecuzione.

E' altrettanto noto che l'eseguibilitá di un programma richiede la presenza contemporanea in memoria centrale sia del programma che dei dati su cui deve operare, ai quali si accede *per indirizzo*[1], mentre il *program counter* stabilisce il controllo sequenziale del flusso esecutivo e che si rispecchia fedelmente nel modo con cui i programmi sono pensati e realizzati.

1.1 MACCHINE DATAFLOW

L'ordinamento *totale* delle istruzioni in formato assembler si presenta in forma pressoché immutata nella programmazione strutturata del Pascal, o di qualunque altro linguaggio evoluto della stessa famiglia, imponendo una sequenzialitá stretta sulle istruzioni. Tuttavia, nella pratica programmativa si verificano spesso situazioni in cui la scelta dell'ordine con cui manipolare un insieme di variabili é del tutto arbitraria potendo, in realtá, accedervi contemporaneamente. In generale, la corretta esecuzione di un programma richiede l'ordinamento solamente per un certo sottoinsieme di istruzioni; le rimanenti, per le quali non é prevista alcuna forma di restrizione temporale, possono essere eseguite in qualunque ordine e, quindi, contemporaneamente o, *concorrentemente*, come si usa dire.

Sorge a questo punto spontanea la questione di come si debbano rappresentare i programmi affinché sia reso esplicita tale forma di ordinamento **parziale** fra le istruzioni. I grafi di dipendenza di dati, con la loro struttura planare, forniscono la rappresentazione richiesta per la loro capacitá di mettere in evidenza l'ordinamento parziale che si realizza attraverso la **rete di comunicazione** che interconnette i nodi *istruzione* del grafo. A questo punto non si ha piú memoria condivisa perché i *dati fluiscono da un'istruzione all'altra direttamente*[2] in forma di **token**, il cui scopo é quello di incapsulare il *dato* con la sua *destinazione*, cioé un indirizzo, nella memoria in cui é immagazzinato il programma, contenente l'istruzione che utilizzerá quel dato.

E' bene tener presente che il grafo sará rappresentato in memoria come una serie di istruzioni[3] con l'informazione del numero e il tipo degli operandi[4] a cui si aggiungono gli indirizzi di destinazione[5] dei valori prodotti. Naturalmente si puó fare sempre in modo che, casi particolari a parte, la destinazione

[1] by reference
[2] by value
[3] i nodi del grafo
[4] corrispondenti agli archi entranti del grafo
[5] gli archi uscenti del grafo

sia unica se si aggiunge un'istruzione speciale di *duplicazione* **dup** il cui effetto é quello di inviare una copia[6] del dato ad una seconda destinazione specificata.

L'esecuzione di un programma di questo tipo non potrá piú essere basata su un registro *program counter*, come avviene nelle Macchine di von Neumann, ma su un particolare meccanismo di ricostruzione delle istruzioni nell'*instruction register* dell'*Unitá di Elaborazione* basato sulla **disponibiltá dei dati** di ciascuna istruzione. Il ciclo di esecuzione di questo nuovo tipo di macchine generalizza quello di von Neumann nel senso che la semplice fase di *fetch* viene sostituita con la coppia *select-examine* dove

> **select** seleziona tutte le istruzioni che hanno ricevuto almeno un dato nel ciclo precedente, mentre
>
> **examine** esamina queste istruzioni per verificare se dispongono di tutti gli operandi di ingresso, altrimenti vengono messe in attesa.

A causa dell'assenza di un registro *program counter* per la selezione immediata dell'istruzione da mandare in esecuzione, l'architettura piú complessa di queste macchine, che prendono il nome di **macchine dataflow** in quanto l'esecuzione di un grafo di dipendenza dei dati é basato sulla *generazione e mantenimento del traffico di token sulla rete di comunicazione che connette i nodi istruzioneÆ*, é costituita da un *pipeline* di unitá specifiche di elaborazione, come appare dalla figura 1.1. In essa si evidenzia

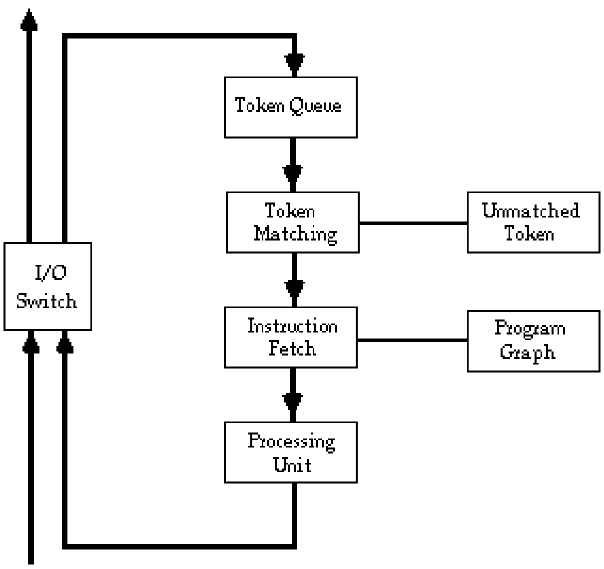

Figure 1.1: architettura di una macchina dataflow

in che modo il traffico di token é realmente gestito. L'unitá elementare di informazione che transita attraverso lo *switch* é il cosiddetto token pacchetto[7], che rappresenta un token segnato[8] in transito su un arco del grafo di programma, e che é costituito da un tipo, un valore e un colore. Tale pacchetto confluisce nel modulo di accodamento, posizionato in prossimitá dello switch, all'ingresso dell'anello.

Uno per volta i token presenti nella coda vengono rimossi e trasferiti verso l'unitá di *matching*, che é responsabile del raggruppamento dei token aventi lo stesso colore per la stessa istruzione di destinazione. Se si limita a due il numero massimo di operandi per istruzione, é sufficiente raggruppare

[6]clonazione
[7]token package
[8]tagged token

i token singolarmente o a coppie. Quelli che richiedono il token complementare, ma che giungono all'unitá di matching in anticipo, sono temporaneamente immagazzinati nell'unitá dei token spaiati[9].

A questo punto, sia nel caso di istruzioni a due operandi con entrambi i token disponibili, sia le istruzioni ad operando singolo, con l'unico token in arrivo, si passa all'invio dei dati richiesti[10] verso l'unitá che gestisce il prelievo delle istruzioni in forma di *pacchetto di token appaiati*[11].

Il campo *destinazione* di un pacchetto di questo tipo viene utilizzato come indirizzo per prelevare la prossima istruzione, contenente il codice operativo e fino a due campi destinazione. Tutto ció produce un pacchetto eseguibile completo che viene inoltrato all'unitá di elaborazione determinando l'esecuzione dell'operazione relativa al corrispondente codice che impiega come operandi le coppie dati-colore preventivamente collezionate e che produce come risultato uno o piú token che vengono instradati verso lo *switch*.

1.2 GRAFI DI DIPENDENZA DEI DATI

Prima di affrontare il problema della realizzazione di un meccanismo alternativo per l'esecuzione di programmi in forma di grafi di dipendenza dei dati prendiamo in esame alcuni semplici esempi. Allo scopo consideriamo il seguente sottoinsieme di istruzioni per un'ipotetica macchina dataflow, scelte con l'intento di illustrare alcuni grafi di programma per semplici algoritmi e per schematizzarne l'implementazione a livello di linguaggi procedurali evoluti.

A meno che non sia specificamente indicato, i token prodotti dall'esecuzione di ciascuna istruzione hanno lo stessa *segnatura*[12] o *colore*, dei rispettivi token di input. Le marcature *LV* e *RV* denotano, rispettivamente, i valori sinistro e destro dei due archi di input dell'istruzione.

Le seguenti sono tipiche *istruzioni logico-aritmetiche*

Istruzione	tipo di dati		Commento
ADD	Int	Int	Somma due valori interi; il risultato è LV + RV
INC	Int		Incremento unitario; il risultato è LV + 1
SUB	Int	Int	Sottrae due valori interi; il risultato è LV - RV
DEC	Int		Decremento unitario; il risultato è LV - 1
MLT	Int	Int	Moltiplica due valori interi; il risultato è LV * RV
TST	Int		Verifica se il valore intero è nullo; il risultato è LV = 0
CMP	Int	Int	Confronta due valori interi; il risultato è LV = LR
GTR	Int	Int	Verifica se il primo valore è maggiore del secondo; il risultato è LV > LR;
GEQ	Int	Int	Verifica se il primo valore non è minore del secondo; il risultato è LV >= LR;
LSS	Int	Int	Verifica se il primo valore è minore del secondo; il risultato è LV < LR;
LEQ	Int	Int	Verifica se il primo valore non è maggiore del secondo; il risultato è LV <= LR;

Per quanto riguarda, invece, le *istruzioni di controllo* verranno prese in esame solamente le tre seguenti

Istruzione	tipo di dati		Commento
DUP	Any		DUPlicate;
BRR	Any	Bool	Branch; if RV then R1 := LV else R2 := LV
MRG	Any	Any	Merge;

[9] unmatched-token unit
[10] caratterizzati dallo stesso colore per la stessa destinazione
[11] token- pair package
[12] tag

la seconda delle quali riceve in ingresso sull'arco sinistro un dato, che puó essere di qualunque tipo, ed un segnale di controllo sull'arco destro, che determina se il dato uscirá sul lato destro o sinistro. Dell'istruzione MRG verrá considerato solamente il caso in cui un solo dato é presente in ingresso e che, pertanto, verrá selezionato come dato in uscita.

Le istruzioni di *commutazione* BRR e MRG sono molto potenti quando sono utilizzate per implementare cicli e funzioni e permettono di ricondurre computazioni potenzialmente complesse a programmi relativamente coincisi analogamente a quanto avviene per gli stessi costrutti della programmazione sequenziale.

Si consideri, ad esempio, il seguente frammento di programma Pascal per il calcolo della funzione fattoriale in forma iterativa.

```
k   := 1;
i   := n;
while ( i > 0 ) do
    begin
        k   := k * i;
        i   := i - i;
    end;
Fact := k;
```

```
    k   := 1;
    i   := n;
1:  if  ( i = 0 ) then goto 2;

    k   := k * i;
    i   := i - 1;
    goto 1;
2:  Fact := k;
```

Per facilitá di comprensione, accanto ad esso, compare lo stesso programma in cui l'istruzione *while* é espressa solamente in termini di salto condizionato. La sua traduzione in grafo di dipendenza dei dati a livello macchina appare in figura 1.2.

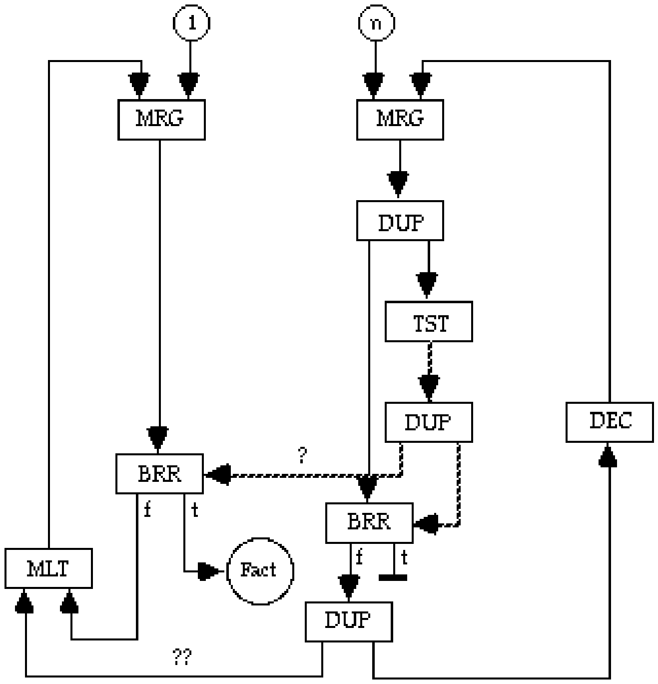

Figure 1.2: fattoriale

Un'analisi dettagliata di questo grafo rivela la presenza di due cicli, attivati entrambi contemporaneamente, attraverso le istruzioni MRG e che poi si sincronizzano attraverso le istruzioni di salto

BRR. Sebbene tale modello costituisca il modo piú naturale di estendere la programmazione sequenziale verso un'esplicita rappresentazione del parallelismo, solamente un massiccio parallelismo a livello delle singole istruzioni[13] giustifica la realizzazione effettiva dela corrispondente macchina dataflow.

Quando, invece, si considera un livello di parallelismo *a grana grossa*, dove un nodo istruzione puó contenere un numero arbitrario di istruzioni semplici, l'esecuzione del grafo di dipendenza di dati é ottenibile 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.

1.3 CONTROLLO PARALLELO

Si faccia riferimento, a questo proposito, un grafo di dipendenza dei dati i cui nodi denominano sezioni arbitrarie di istruzioni sequenziali. Per quanto si é visto 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.

Dall'esempio del fattoriale e da quelli che verranno trattati nei capitoli seguenti, 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 del 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 esplicitare 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 in precedenza. 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.

[13] parallelismo a *grana fine*

1.4 MODELLI MULTISEQUENZIALI

La discussione precedente ha messo in luce la possibilitá di realizzare l'esecuzione di un programma, in forma di grafo di dipendenza di dati, mediante un certo numero di *flussi sequenziali* di computazione che sono identificabili nel grafo con altrettanti cammini di un appropriato *albero* 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 corrispondenza di 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 quella istruzione 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 di tipo *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 complesso 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.

1.5 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**.

Per quanto si é visto la comprensione di un programma concorrente in termini dei suoi *processi sequenziali componenti* e delle loro*interazioni*, senza fare riferimento al modo con cui sono eseguiti, richiede che *non si debba fare alcuna assunzione sulla velocitá di esecuzione* dei processi eseguiti concorrentemente, tranne il fatto che siano tutte *positive*. Questa assunzione é nota anche come **ipotesi di avanzamento finito**.

I processi sequenziali che compongono l'esecuzione di un programma parallelo si dicono anche **cooperanti**. Per poter cooperare tali processi devono*comunicare* e *sincronizzarsi*. Mentre la comunicazione fa si che l'esecuzione di un processo possa influenzare l'esecuzione dell'altro, la loro esecuzione concorrente é ottenibile con diverse soluzioni sintattiche, le quali possono classificarsi in base alla capacitá di *separare* la *definizione* dei processi dalla loro *sincronizzazione*. Se si impongono 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.

1.6 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 informazione sufficiente sullo *stato della coroutine* da permettere, al momento del ritorno del controllo, di eseguire l'istruzione immediatamente seguente l'istruzione*resume*. 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.

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* 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.

L'istruzione **cobegin** é un metodo *strutturato* per denotare l'esecuzione parallela[14] di un insieme di istruzioni. L'esecuzione di

$$\textbf{cobegin} S_1; S_2; ...; S_n \textbf{coend}$$

termina solo quando l'esecuzione di tutti gli 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

$$\begin{array}{lcl}
 & & \mathrm{K} := \mathrm{n}; \\
 & & \textbf{fork } L_2; \;....\; ; \textbf{fork } L_n; \\
 & & S_1; \textbf{ goto } L_{n+1}; \\
L_2 & : & S_2; \textbf{ goto } L_{n+1}; \\
 & & ..... \\
L_{n-1} & : & S_{n-1}; \textbf{ goto } L_{n+1}; \\
L_n & : & S_n; \\
L_{n+1} & : & \textbf{join } \mathrm{K};
\end{array}$$

[14]concorrente

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.

Capitolo 2

Macchina Concorrente Multipila

Si é visto che il meccanismo delle coroutine permette di realizzare una forma di parallelismo fra processi che possono essere eseguiti su un singolo processore. In questo caso, se si esclude il trattamento delle interruzioni[1] é sufficiente realizzare un supporto *run-time* in cui lo scambio di contesto fra processi venga programmato esplicitamente. Alternativamente, lo scambio di contesto puó essere lasciato a discrezione dell'implementazione che lo produce implicitamente utilizzando il meccanismo dell'interruzione d'orologio.

Mentre un processo é *in esecuzione*, gli altri processi devono attendere il loro turno nella *coda dei processi pronti*. Fissato un certo intervallo[2] di tempo al termine del quale l'elaboratore sospende l'esecuzione del processo corrente, il cambio di contesto esecutivo[3] dá l'illusione che tali processi vengano eseguiti simultaneamente.

A tale scopo il nucleo rappresenta 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 un processo *riprende* l'esecuzione. Si supponga, allora, di avere a disposizione una macchina a pila per l'esecuzione di programmi sequenziali e la si ampli in modo da permettere che piú processi concorrenti possanno essere in esecuzione contemporanea. Si tenga presente, comunque, che in ogni istante di tempo un solo processo per volta puó essere attivo.

2.1 LA MACCHINA A PILA

La macchina che verrá considerata nel seguito é l'esecutore di programmi **pcode**[4] i quali rappresentano il risultato della compilazione di programmi PASCAL nell'originaria definizione di Wirth. L'elaboratore a pila SC é costituito da 12 registri e una memoria che puó pensarsi come un insieme di sequenze lineari le cui singole unitá variano a seconda del tipo di informazione da immagazzinare. La memoria é strutturata in *area del codice*, etichettata con *Mem* e contenente il programma da eseguire, e *area dei dati* dove sono contenute tutte le informazioni per la corretta esecuzione del programma.

L'area dei dati, a sua volta, é organizzata come una pila in cui sono mantenute le costanti reali, le stringhe, gli identificatori, i blocchi, i descrittori di array e record e l'ambiente esecutivo vero e proprio, come indicato in figura 2.1.

Per quanto riguarda, invece, i registri questi sono divisi in registri che si riferiscono alla *memoria*, intesa come area del codice, e alla *pila*, intesa come supporto esecutivo. In questo senso, l'accesso in memoria é ottenuto con i registri

[1] l'unico insieme di eventi asincroni che puó influenzare la normale esecuzione dei processi
[2] o, quanto
[3] riprendendo l'esecuzione di un altro processo
[4] nell'accezione di Wirth

Figure 2.1: organizzazione della pila

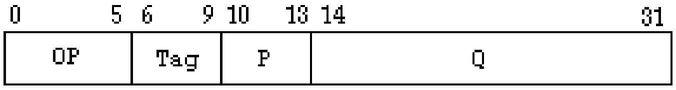

Figure 2.2: struttura delle istruzioni

PC	program counter
IR	instruction register

mentre, per accedere alla pila, si utilizzano i registri

SP	stack pointer
FP = D0	frame pointer
D1,..., D7	index register
SL	stack limit

l'impiego dei registri indice essendo dovuto solamente a motivi di miglioramento della velocitá di accesso alla pila.

Il formato di ciascuna istruzione nell'area del codice é dato da quattro campi OP, Tag, P e Q di cui il secondo non é attualmente utilizzato. La figura 2.2 illustra la situazione: il campo OP contiene il codice dell'istruzione e il suo valore é compreso fra 0 e 63, il campo P puó assumere un valore compreso fra 1 e 7 che permette di determinare l'indirizzo assoluto di un elemento dello stack con la formula

$$Indirizzo = D_k + Q$$

nell'ipotesi che il campo Q contenga un indirizzo. Il contenuto di ciascun campo dipende dalla singola istruzione.

La realizzazione della macchina a pila SC é ottenuta mediante simulazione con un programma PASCAL che utilizza opportune strutture dati che caratterizzano le singole aree di memoria dell'elaboratore SC e un insieme di procedure per ottenere il suo corretto funzionamento. Nel seguito viene descritta con maggior dettaglio ciascuna strutture dati.

2.1.1 Area del Codice

Le definizioni di tipo e le dichiarazioni di variabili sono quelle che si evincono dalla discussione precedente. In particolare, non compare il campo Tag perché non utilizzato dall'interprete.

```
const  MemSize = 1024;
         OpMax = 63;         {2^6 - 1}
          LMax = 7;          {2^3 - 1}
          Nmax = 131071;     {2^17 - 1}

type    MemRef = 0 .. MemSize;
        OpCode = 0 .. OpMax;
         Index = 1 .. LMax;
         Value = -Nmax .. Nmax;

   Instruction = record
                   f : OpCode;
                   x : Index;
                   y : Value
                 end;

       Memory = array[MemRef] of Instruction;

var         IR : Instruction;
            PC : MemRef;
           Mem : Memory;
```

Come si puó notare l'attuale implementazione prevede una dimensione massima dell'area del codice di 1024 istruzioni.

2.1.2 Spazio degli Identificatori

Ogni identificatore non riservato che compare nel programma, ad eccezion fatta per le procedure e funzioni predefinite, é rappresentato da un elemento della tabella degli identificatori costituito da cinque campi, i primi due dei quali ne determina univocamente la posizione nella pila a partire dall'indirizzo di base assegnato.

Il terzo campo é un entry point nella tabella dei blocchi ed é utilizzato nel caso che l'identificatore sia il nome di una procedura o di una funzione. In corrispondenza di ogni nuovo identificatore viene associata la pseudoistruzione

```
 L    .NAME    lev adr ref link name
```

dove L indica l'entry point nella relativa tabella.

```
const   NameRef = 10;
       AtomSize = 100;

type    AtomRef = 0 .. AtomSize;
        NameRef = 1 .. NameRef;
          Alpha = packed array[NameRef] of char;

           Atom = record
```

```
                    lev: Index;
                    adr: MemRef;
                    ref: BlockRef;
                    link: AtomRef;
                    name: Alpha
                  end;

        AtomTable = array[AtomRef] of Atom;

   var       AtomT: AtomTable;
```

Si tenga presente che, attualmente, soltanto i primi dieci caratteri degli identificatori dichiarati nel programma sorgente sono significativi.

2.1.3 Vettori

Ogni vettore é rappresentato da un'opportuno descrittore nella tabella dei vettori costituito da tre campi che servono a determinare lo spazio totale da riservare nella pila per memorizzare il vettore. La seguente pseudoistruzione

```
    L    .BLCK    low high elsize
```

definisce l'entry point nella tabella nella posizione indicata da L.

```
  const      AMax = 30;

  type   ArrayRef = 1..AMax;

        ArrayType = record
                      low: Value;
                      high: Value;
                      elsize: Value
                    end;

       ArrayTable = array[ArrayRef] of ArrayType;

  var        ATab : ArrayTable;
```

2.1.4 Costanti Reali

Anche le costanti reali sono memorizzate in un'apposita tabella il cui indirizzo viene caricato sulla pila quando la costante é utilizzata dal programma. La seguente pseudoistruzione

```
    L    .LONG    adr
```

definisce, pertanto, un entry point nella tabella delle costanti reali nella posizione L.

```
  const   MaxData = 20;

  type   DataRef = 1..MaxData;
       DataTable = array[DataRef] of real;

  var       RTab : DataTable;
```

2.1.5 Stringhe

Le stringhe sono memorizzate come sequenze di caratteri contigui in una tabella appropriata a partire dalla posizione indicata dall'etichetta L della pseudoistruzione

```
L    .DATA    "stringa di caratteri"
```

i cui caratteri componenti sono delimitati da doppio apice.

```
const  StringSize = 600;

type    StringRef = 0..StringSize;
      StringTable = packed array[StringRef] of char;

var          STab : StringTable;
```

2.1.6 Descrizione degli Ambienti

Ad ogni blocco di istruzioni, corrispondenti ad una procedura, fuzione oppure al programma principale, é associato un descrittore di tre campi che viene aggiunto nella tabella degli ambienti con la psedoistruzione

```
L    .WORD    psize vsize last
```

nella posizione indicata da L e con i parametri che rappresentano, nell'ordine, lo spazio nella pila da riservare agli eventuali parametri della procedura o funzione, il numero totale di locazioni di memoria da riservare nella pila per le variabili locali e la posizione dell'ultimo identificatore locale al blocco di istruzioni nella tabella degli identificatori.

```
const    MaxBlock = 20;

type     BlockRef = 1..MaxBlock;
            Block = record
                        psize: StackRef;
                        vsize: StackRef;
                        last: AtomRef
                     end;

      Environment = array[BlockRef] of Block;

var          Envr : Environment;
```

2.1.7 Ambiente di esecuzione

L'ambiente di esecuzione dei programmi per una macchina a pila é, ovviamente, la pila che é una struttura dati avente un meccanismo di accesso LIFO in modo tale che la cima contenga tutte le informazioni relative all'esecuzione delle istruzioni correnti e che, quindi, rappresenta l'ambiente naturale in cui eseguire i programmi sequenziali. Per maggiori dettagli sulla strutturazione delle informazioni nella pila si veda nella sezione 4.2.

```
const   StackSize = 4600;
        WorkSpace = 460;
```

```
type     StackRef = 0..StackSize;

         Contents = record
                      case types of
                        ints: (i: Value);
                        reals: (r: real);
                        bools: (b: boolean);
                        chars: (c: char);
                        stacks: (s: StackRef)
                     end;

         Display = array[Index] of StackRef;

           Stack = array[StackRef] of Contents;

var           SP : StackRef;
              FP : StackRef;
              SL : StackRef;

              Dx : Display;
               S : Stack;
```

Come si puó notare, ciascun elemento della pila puó contenere un intero, un numero reale, un valore logico oppure un carattere. Gli interi sono utilizzati anche come indici per accedre alla pila.

2.1.8 Controllo dell'Esecuzione

Per quanto riguarda l'esecuzione dei programmi questa é ottenuta mediante la ripetizione continua di un ciclo fetch-execute che preleva, all'indirizzo indicato dal program counter PC, l'istruzione dalla memoria ed introducendola nel registro IR delle istruzioni per passarla, successivamente, all'esecuzione. L'operazione ClockInterrupt che precede il ciclo serve ad estendere la macchina all'esecuzione concorrente di piú programmi. Per maggiori dettagli si veda la sezione 5.

Il registro **PSW**[5] é utilizzato per mantenere lo stato corrente dell'esecuzione la cui terminazione puó essere causata da programma con appropriata istruzione o per un errore che si é verificato durante l'esecuzione di un'istruzione.

```
const     LineLength = 80;
           TimeSlice = 10;

type   ProgramStatus = (run, stop, caschk, divchk, inxchk, stkchk,
                               linchk, lngchk, redchk, deadlocked);
           ClockTick = integer;

var              PSW : ProgramStatus;
                Seed : longint;
               lncnt : integer;
                Tick : ClockTick;
              blkcnt : integer;                  {counter}
              chrcnt : integer;                  {counter}
```

[5]program status word

```
                DebugF : boolean;
                   fld : array[1..4] of integer;   {default field widths}

begin
  Prologue;

  InitMainContext;
  for Pool := 1 to MaxProcess do InitContext(Pool);
  SetRandom(Seed);

  Tick := 0;
  Pool := 0;
  PFlag := false;
  PSW := run;

  Serve(MAIN);

  repeat
    if not Executing(MAIN) then ClockInterrupt;
    IR := Mem[PC];
    PC := PC + 1;
    Execute(IR.f)
  until PSW <> run;

  if PSW <> stop then PostMortemDump;

  Epilogue
end.
```

Si noti che in caso di terminazione anomala e, quindi, non generata da programma con l'istruzione *HLT*, viene prodotto un *dump* della memoria che fotografa lo stato della pila al momento del verificarsi dell'errore.

2.2 ISTRUZIONI

Come si é gia visto ciascuna istruzione é caratterizzata da un codice *f* il cui valore é compreso fra 0 e 63 e i cui mnemonici sono riportati di seguito

0	LDA	*Load Address*
1	LDC	*Load Value*
2	LDI	*Load Indirect*
3	UDX	*Update Display*
4	FORK	*Fork at Cobegin*
5	JOIN	*Join at Coend*
6	WAIT	*Wait on Semaphore*
7	SGNL	*Signal to Semaphore*
8	CSP	*Call Standard Procedure*
9	LOA	*Load Offset*
10	UJP	*Unconditional Jump*

11	FJP	*Conditional Jump*
12	SWI	*Switch*
13	XJP	*Indexed Jump*
14	CHK T	*Check Toward*
15	NXT T	*Next Toward*
16	CHK D	*Check Downward*
17	NXT D	*Next Downward*
18	MST	*Mark Stack*
19	CUP	*Call User Procedure*
20	INX	*Index*
21	IXA	*Indexed Address*
22	LDB	*Load Block*
23	CPB	*Copy Block*
24	ENT	*Enter Literally*
25	LDR	*Load Real*
26	FLT	*Float*
27	RDD	*Read Data*
28	WRS	*Write String*
29	WRD	*Write Unformatted Data*
30	WRC	*Write Formatted Constant*
31	HLT	*Halt*
32	RET P	*Return from Procedure*
33	RET F	*Return from Function*
34	STR	*Store Indirect*
35	NOT	*Boolean Not*
36	NGI	*Negate Integer*
37	WRR	*Write Real*
38	STO	*Store*
39	EQL R	*Equal Real*
40	NEQ R	*Not Equal Real*
41	LSS R	*Less Real*
42	LEQ R	*Less or Equal Real*
43	GTR R	*Greater Real*
44	GEQ R	*Greater or Equal Real*
45	EQL I	*Equal Integer*
46	NEQ I	*Not Equal Integer*
47	LSS I	*Less Integer*
48	LEQ I	*Less or Equal Integer*
49	GTR I	*Greater Integer*
50	GEQ I	*Greater or Equal Integer*
51	IOR	*Inclusive Or*
52	ADI	*Add Integer*
53	SBI	*Subtract Integer*
54	ADR	*Add Real*
55	SBR	*Subtract Real*
56	AND	*Boolean And*
57	MPI	*Multiply Integer*
58	DVI	*Divide Integer*
59	MOD	*Module*
60	MPR	*Multiply Real*
61	DVR	*Divide Real*
62	RLN	*Read Line*
63	WLN	*Write Line*

Come si puó facilmente notare le istruzioni della macchina a pila non riguardano solamente le usuali operazioni aritmetiche e logiche ma si riferiscono anche alla loro gestione in termini di indirizzamento e immagazzinamento oltre al controllo dell'esecuzione del programma e della pila stessa. Le istruzioni per la concorrenza sono trattate a parte nella sezione 5. Per le altre viene data una breve descrizione nei paragrafi seguenti.

Ciascuna istruzione viene presentata secondo lo schema usuale di utilizzo nei programmi assembler, per cui l'etichetta L che precede l'istruzione indica l'*entry point* nella di memoria che contiene il programma.

2.2.1 Istruzioni Base

Le istruzioni che seguono servono a gestire le operazioni base della macchina per quanto riguarda variabili e costanti relativamente al loro immagazzinamento sulla pila, gestione degli indirizzi, salto condizionato e incondizionato.

LDA : Load Address

Carica sulla cima della pila l'indirizzo indicato nella forma *lev*, *adr* secondo lo schema riportato

```
L    LDA       lev adr
```

essendo *lev* l'appropriato registro indice e *adr* l'offset relativamente all'indirizzo contenuto nel registro FP, che funge da indirizzo base.

```
procedure LoadAddress;
begin
   SP := SP + 1;
   if SP > SL then PSW := stkchk
              else S[SP].i := Dx[IR.x] + IR.y
end;
```

LDC : Load Value

Carica sulla cima della pila il contenuto all'indirizzo indicato nella forma *lev*, *adr* secondo lo schema riportato

```
L    LDC       lev adr
```

essendo *lev* l'appropriato registro indice e *adr* l'offset relativamente all'indirizzo contenuto nel registro FP, che funge da indirizzo base.

```
procedure LoadValue;
begin
  SP := SP + 1;
  if SP > SL then PSW := stkchk
             else S[SP] := S[Dx[IR.x] + IR.y]
end;
```

LDI : Load Indirect

Carica sulla cima della pila il contenuto dell'indirizzo che si trova all'indirizzo indicato nella forma *lev*, *adr* secondo lo schema riportato

```
L    LDI       lev adr
```

essendo *lev* l'appropriato registro indice e *adr* l'offset relativamente all'indirizzo contenuto nel registro FP, che funge da indirizzo base.

```
procedure LoadIndirect;
begin
  SP := SP + 1;
  if SP > SL then PSW := stkchk
             else S[SP] := S[S[Dx[IR.x] + IR.y].i]
end;
```

UDX : Update Display

Aggiorna il contenuto dei registri indice secondo lo schema

```
L    UDX       lev adr
```

essendo *lev* l'appropriato registro indice e *adr* l'offset relativamente all'indirizzo contenuto nel registro FP, che funge da indirizzo base.

```
procedure UpdateDisplay;
  var h1, h2, h3: integer;
begin
  h1 := IR.y;
  h2 := IR.x;
  h3 := FP;
  repeat
    Dx[h1] := h3;
    h1 := h1 - 1;
    h3 := S[h3 + 2].i
  until h1 = h2
end;
```

LOA : Load Offset

Aggiorna l'indirizzo contenuto sulla cima della pila incrementandolo del valore di offset secondo lo schema

```
L    LOA       adr
```

essendo *adr* il parametro passato dall'istruzione.

```
procedure OffSet;
begin
  S[SP].i := S[SP].i + IR.y
end;
```

UJP : Unconditional Jump

Salta incondizionatamente all'istruzione il cui indirizzo di memoria é passato come parametro secondo lo schema

```
L    UJP       adr
```

senza modificare il contenuto della pila.

```
procedure Jump;
begin
  PC := IR.y
end;
```

FJP : False Jump

Se il valore booleano della cima della pila é *false* vai all'istruzione il cui indirizzo di memoria é passato come parametro secondo lo schema

```
L    FJP       adr
```

rimuovendo tale valore dalla cima della pila.

```
procedure FalseJump;
begin
  if not S[SP].b then PC := IR.y;
  SP := SP - 1;
end;
```

ENT : Enter Literally

Carica sulla cima della pila il valore passato come parametro secondo lo schema seguente

```
L    ENT       int
```

con la restrizione che tale valore possa essere solamente di tipo intero.

```
procedure Literal;
begin
  SP := SP + 1;
  if SP > SL then PSW := stkchk
              else S[SP].i := IR.y
end;
```

LDR : Load Real

Carica sulla cima della pila l'*entry point* nella tabella delle costanti reali secondo lo schema seguente

```
L    LDR       ref
```

essendo tale riferimento passato come parametro

```
procedure LoadReal;
begin
  SP := SP + 1;
  if SP > SL then PSW := stkchk
             else S[SP].r := RTab[IR.y]
end;
```

FLT : Float

Converti il valore intero, posto nella pila al di sotto della cima, nella posizione indicata dal parametro *ref* secondo lo shema

```
L   FLT     ref
```

lasciando immutato lo stato della pila

```
procedure Float;
  var  h : integer;
begin
  h := SP - IR.y;
  S[h].r := S[h].i
end;
```

HLT : Halt

Blocca l'esecuzione del programma secondo lo schema

```
L   HLT
```

provocando l'arresto del ciclo fetch-execute della macchina a pila.

```
procedure DoStop;
begin
  PSW := stop
end;
```

STR : Store Indirect

Immagazzina sulla cima della pila il contenuto all'indirizzo presente sulla cima stessa secondo lo schema

```
L   STR
```

lasciando immutata la posizione della cima

```
procedure StoreIndirect;
begin
  S[SP] := S[S[SP].i]
end;
```

STO : Store

Carica il valore inserito sulla cima della pila all'indirizzo contenuto subito al di sotto della cima secondo lo schema

```
L   STO
```

rimuovendo tali elementi dalla cima della pila.

```
procedure Store;
begin
  S[S[SP - 1].i].i := S[SP].i;
  SP := SP - 2
end;
```

2.2.2 Istruzioni di Gestione della Pila

Le istruzioni che seguono sono impiegate per la gestione della pila durante le chiamate di procedure e di funzioni. La loro implementazione riflette esattamente il modo con cui sono organizzate le informazioni all'interno della pila le quali, a loro volta, sono determinate dalla corrispondenza fra la valutazione dinamica di un programma Pascal ed il suo testo statico, nel senso che vengono gestite dinamicamente tutte quelle connessioni necessarie a rendere accessibili soltanto le strutture dati che il testo del programma prevede.

Scendendo in maggior dettaglio, si puó dire che la pila consiste di una sequenza di *segmenti di dati*, ciascuno dei quali appartiene all'attivazione di qualche procedura o funzione, eccetto il primo segmento che inizia alla locazione 0 e che appartiene al blocco piú esterno, cioé il programma principale.

Ogni segmento di dati é costituito di tre parti, secondo quanto evidenzia la figura 2.3. La prima prende il nome di *mark stack*[6], la seconda rappresenta la *sezione dei parametri* se ce ne sono nella definizione della procedura o della funzione cui il segmento di dati appartiene, mentre l'ultima definisce la *sezione dei dati locali*, quando questi sono dichiarati all'interno di essa. Inoltre, possono essere presenti elementi temporanei richiesti durante la valutazione del programma e riguardanti solamente la procedura o funzione corrente.

A sua volta, la parte mark stack é costituita da cinque locazioni successive della pila: la prima riserva lo spazio necessario a contenere il *valore restituito* dopo la chiamata della funzione e non é utilizzato dalle procedure ma viene incluso per ragioni di uniformitá e facilitá di implementazione; la seconda locazione contiene un puntatore nella pila che prende il nome di *indirizzo di ritorno*; le due locazioni successive contengono i cosiddetti *link statico* e *dinamico*, che sono puntatori nella memoria, il secondo dei quali registra la storia dinamica delle attivazioni mentre il primo connette quelle sezioni che si riferiscono a procedure dichiarate staticamente una dentro l'altra; infine, la locazione designata come *table index* é un puntatore alla tabella dei simboli da cui sono ricavate ulteriori informazioni per completare la costruzione del segmento. La catena relativa al link statico designa tutti i segmenti che sono correntemente accessibili. Per velocizzare l'accesso ai dati tale catena, che inizia con la procedura correntemente attiva, viene copiata anche nei registri indice.

L'esecuzione di un programma inizia sempre caricando sulla pila la parte *mark stack* relativa al programma principale che, avendo la struttura di procedura, non necessita della locazione relativa al risultato della funzione. Pertanto, la pila viene inizializzata nel modo seguente

```
procedure InitStack;
begin
  S[1].i := 0;
  S[2].i := 0;
  S[3].i := -1;
  S[4].i := Envr[1].last
end;
```

tenendo presente, fra l'altro, che non cé indirizzo di ritorno né link statico.

MST : Mark Stack

Durante l'attivazione di una procedura o funzione viene iniziata l'allocazione di un nuovo segmento di dati eseguendo l'istruzione

```
  L    MST       ref
```

[6] o, mark section, marcatura della pila

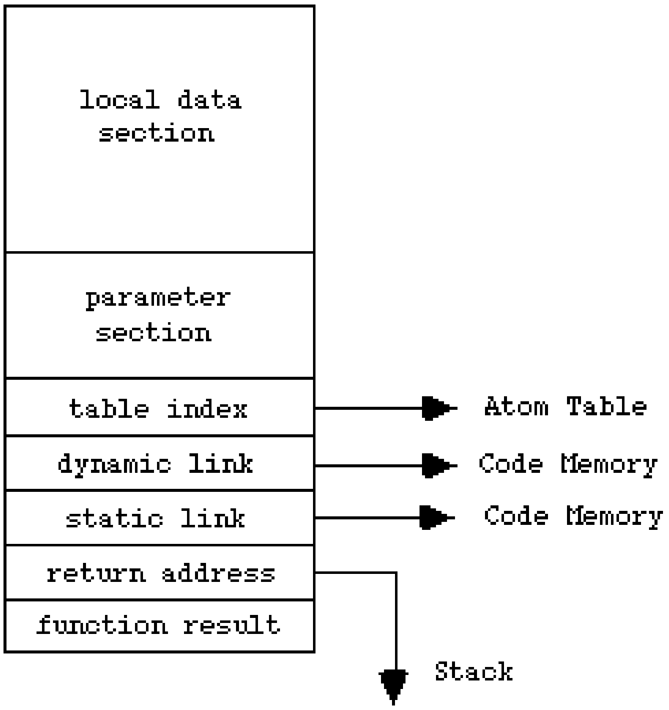

Figure 2.3: record di attivazione delle procedure

il cui effetto é quello di *marcare la pila*, cioé di riservare spazio per i 5 elementi costituenti il *block mark*. Si noti che il parametro *ref* individua l'elemento della tabella dei simboli che identifica il nome della procedura o funzione chiamata.

```
procedure MarkStack;
  var  h1 : Blockref;
       h2 : StackRef;
begin
  h1 := AtomT[IR.y].ref;
  h2 := Envr[ h1 ].vsize;
  if SP + h2 > SL then PSW := stkchk else
    begin
      SP := SP + 5;
      S[SP - 1].i := h2 - 1;
      S[SP].i := IR.y
    end
end;
```

Accedendo alla tabella dei simboli si ricava la dimensione del segmento da creare; successivamente tale parametro viene salvato nell'appropriata locazione dello stack.

CUP : Call User Procedure

Dopo aver processato i parametri attuali, questi vengono caricati sulla pila come valori nel caso di *parametri valore*, oppure come indirizzi, nel caso di *paramentri indirizzo*. Successivamente viene eseguita la chiamata di procedura secondo lo schema

```
L    CUP       psize
```

il cui effetto é quello di modificare i registri FP[7] e SP[8] e assegnare i *link* e l'indirizzo di ritorno ai corrispondenti campi del *block mark*.

```
procedure Call;
  var  h1 : StackRef;
       h2 : StackRef;
       h3 : Level;
       h4 : StackRef;
begin
  h1 := SP - IR.y;
  h2 := S[h1 + 4].i;
  h3 := AtomT[h2].lev;
  Dx[h3 + 1] := h1;
  h4 := S[h1 + 3].i + h1;
  S[h1 + 1].i := PC;
  S[h1 + 2].i := Dx[h3];
  Ready := true;
  if PFlag then S[h1 + 3].i := PSBTab[0].FPImage
          else S[h1 + 3].i := FP;
  for h3 := SP + 1 to h4 do S[h3].i := 0;
  FP := h1;
  SP := h4;
  PC := AtomT[h2].adr
end;
```

Si noti che l'operando *psize* dell'istruzione rappresenta la dimensione dell'area dei parametri sulla pila diminuita di una unitá.

LDB : Load Block

Copia sulla cima della pila, a partire dall'indirizzo ivi contenuto, una sequenza di elementi della pila secondo lo schema

```
L    LDB       psize
```

essendo *psize* la lunghezza di tale sequenza.

```
procedure LoadBlock;
  var  h1 : StackRef;
       h2 : StackRef;
begin
   h1 := S[SP].i;
   SP := SP - 1;
   h2 := IR.y + SP;
   if h2 > SL then PSW := stkchk else
      while SP < h2 do
        begin
```

[7]frame pointer, o puntatore alla base del segmento
[8]stack pointer, o puntatore alla cima del segmento

```
            SP := SP + 1;
            S[SP] := S[h1];
            h1 := h1 + 1
          end
  end;
```

Questa istruzione viene utilizzata quando si passa come parametro di una chiamata di procedura o funzione un array o un record.

CPB : Copy Block

Copia, secondo le direttive contenute nelle due locazioni in prossimitá della cima della pila, una sequenza di elementi della pila con lo schema

```
L    CPB       psize
```

essendo *psize* il numero di tali elementi. La cima della pila contiene l'indirizzo di partenza mentre nella locazione subito al di sotto é contenuto l'indirizzo di destinazione.

```
procedure CopyBlock;
  var  h1, h2, h3 : integer;
begin
  h1 := S[SP - 1].i;
  h2 := S[SP].i;
  h3 := h1 + IR.y;
  while h1 < h3 do
    begin
      S[h1] := S[h2];
      h1 := h1 + 1;
      h2 := h2 + 1
    end;
  SP := SP - 2
end;
```

CSP : Call Standard Procedure

Quando la procedura o funzione da chiamare é predefinita, lo schema seguente

```
L    CSP       code
```

permette di realizzare tale chiamata, essendo *code* il codice della procedura.

```
procedure CallStandProc (P: integer);

  procedure IsEof (var f: text);
  begin
    SP := SP + 1;
    if SP > SL then PSW := stkchk
                else S[SP].b := eof(f)
  end;

  procedure IsEoln (var f: text);
```

```
  begin
    SP := SP + 1;
    if SP > SL then PSW := stkchk
               else S[SP].b := eoln(f)
  end;

begin
  case P of
     0 : S[SP].i := abs(S[SP].i);
     1 : S[SP].r := abs(S[SP].r);
     2 : S[SP].i := sqr(S[SP].i);
     3 : S[SP].r := sqr(S[SP].r);
     4 : S[SP].b := odd(S[SP].i);
     7 : S[SP].c := succ(S[SP].c);
     8 : S[SP].c := pred(S[SP].c);
     9 : S[SP].i := round(S[SP].r);
    10 : S[SP].i := trunc(S[SP].r);
    11 : S[SP].r := sin(S[SP].r);
    12 : S[SP].r := cos(S[SP].r);
    13 : S[SP].r := exp(S[SP].r);
    14 : S[SP].r := ln(S[SP].r);
    15 : S[SP].r := sqrt(S[SP].r);
    16 : S[SP].r := arctan(S[SP].r);
    17 : IsEof(input);
    18 : IsEoln(input);
  end
end;
```

RET P : Return from Procedure

Al termine della procedura, l'istruzione di ritorno inverte le operazioni eseguite durante la fase di chiamata

```
L    RET P
```

ristabilendo il corretto contenuto dei registri. Eventualmente viene fatta seguire un'istruzione di *aggiornamento del display*, *UDX*, nel caso che il livello statico della procedura corrente é inferiore a quello della procedura cui ritorna il controllo.

```
procedure ExitProcedure;
begin
  SP := FP - 1;
  PC := S[FP + 1].i;
  if PC <> 0 then FP := S[FP + 3].i else
    begin
      Pool := Pool - 1;
      Ready := false;
      Tick := 0;
      PSBTab[0].ReadyImage := (Pool = 0)
    end
end;
```

RET F : Return from Function

Anche nel caso di terminazione di una funzione, l'istruzione di ritorno inverte le operazioni eseguite durante la fase di chiamata

```
L    RET F
```

ristabilendo sia il *contatore di programma* che i registri che puntano all'estremitá del segmento dati della pila.

```
procedure ExitFunction;
begin
  SP := FP;
  PC := S[FP + 1].i;
  FP := S[FP + 3].i
end;
```

Riassumendo, l'esecuzione di una chiamata di procedura o funzione viene realizzata con una sequenza di istruzioni la prima delle quali, MST, serve a marcare l'inizio di un nuovo segmento sulla pila. Successivamente vengono caricati, se ve ne sono, i parametri attuali e le variabili locali e, infine, viene eseguita la chiamata di procedura o funzione.

Per accedere ai dati del segmento si utilizza un indirizzo di offset relativo alla prima locazione del segmento, oppure il puntatore alla cima della pila la quale puó contenere risultati temporanei generati durante la valutazione delle espressioni.

2.2.3 Istruzioni di Salto Indiciato

Le istruzioni che seguono sono impiegate per realizzare i costrutti strutturati di ripetione **for** e di selezione **case** e per accedere ai singoli elementi di un **array**.

SWI : Switch; XJP : Indexed Jump

Le due istruzioni indicate permettono di realizzare l'istruzione di selezione **case** del Pascal

```
L    SWI       ref                L    XJP       ref
```

essendo *ref* un parametro che permette di accedere direttamente all'area di codice da eseguire.

```
procedure Switch;
  var  h1, h2, h3 : integer;
begin
  h1 := S[SP].i;
  SP := SP - 1;
  h2 := IR.y;
  h3 := 0;
  repeat
    if Mem[h2].f <> 13 then
      begin
        h3 := 1;
        PSW := caschk
      end else if Mem[h2].y = h1 then
      begin
```

```
        h3 := 1;
        PC := Mem[h2 + 1].y
      end else
      h2 := h2 + 2
  until h3 <> 0
end;
```

CHK T : Check Toward

Verifica se il valore finale non é minore del valore iniziale per la variabile indice del ciclo **for-to**, poste in quest'ordine sulla cima della pila, con lo schema

```
L    CHK T    ref
```

per cui, in caso affermativo il valore finale viene assegnato alla variabile mentre, in caso negativo, l'esecuzione passa all'istruzione indicata dal parametro e i tre elementi rimossi dalla cima della pila.

```
procedure CheckToward;
   var  h : integer;
begin
  h := S[SP - 1].i;
  if h <= S[SP].i then S[S[SP - 2].i].i := h else
    begin
      SP := SP - 3;
      PC := IR.y
    end
end;
```

NXT T : Next Toward

Incrementa l'indice del ciclo **for-to** e verifica se non é superiore al valore finale contenuto sulla cima della pila usando lo schema

```
L    NXT T    ref
```

per cui, in caso affermativo, ripete la sequenza di istruzioni, altrimenti continua e valore finale, valore iniziale e indirizzo indice sono rimossi dalla cima della pila.

```
procedure NextToward;
   var  h1, h2 : integer;
begin
  h2 := S[SP - 2].i;
  h1 := S[h2].i + 1;
  if h1 <= S[SP].i then
    begin
      S[h2].i := h1;
      PC := IR.y
      end else
    SP := SP - 3
end;
```

CHK D : Check Downward

Verifica se il valore finale non é maggiore del valore iniziale per la variabile indice del ciclo **for-downto**, poste in quest'ordine sulla cima della pila, con lo schema

```
    L    CHK D    ref
```

per cui, in caso affermativo il valore finale viene assegnato alla variabile mentre, in caso negativo, l'esecuzione passa all'istruzione indicata dal parametro e i tre elementi rimossi dalla cima della pila.

```
procedure CheckDownward;
  var  h : integer;
begin
  h := S[SP - 1].i;
  if h >= S[SP].i then S[S[SP - 2].i].i := h else
    begin
      SP := SP - 3;
      PC := IR.y
    end
end;
```

NXT D : Next Downward

Decrementa l'indice del ciclo **for-downto** e verifica se il suo valore é superiore al valore iniziale contenuto sulla cima della pila usando lo schema

```
    L    NXT D    ref
```

per cui, in caso affermativo, ripeti la sequenza di istruzioni, altrimenti continua e valore finale, valore iniziale e indirizzo indice sono rimossi dalla cima della pila.

```
procedure NextDownward;
  var  h1, h2 : integer;
begin
  h2 := S[SP - 2].i;
  h1 := S[h2].i - 1;
  if h1 >= S[SP].i then
    begin
      S[h2].i := h1;
      PC := IR.y
      end else
    SP := SP - 3
end;
```

INX : Index

Accedi all'elemento di un **array**, costituito da elementi di dimensione unitaria e riferito dal parametro *ref* secondo lo schema

```
    L    INX      ref
```

essendo l'indice dell'elemento posto sulla cima della pila il quale viene rimosso al termine dell'operazione.

```
procedure Index;
  var  h1, h2, h3 : integer;
begin
  h1 := IR.y;
  h2 := ATab[h1].low;
  h3 := S[SP].i;
  if h3 < h2 then PSW := inxchk else
    if h3 > ATab[h1].high then PSW := inxchk else
      begin
        SP := SP - 1;
        S[SP].i := S[SP].i + (h3 - h2)
      end
end;
```

IXA : Indexed Address

Accedi all'elemento di un **array**, costituita da elementi di dimensione definita nel suo descrittore e riferito dal parametro *ref* secondo lo schema

```
L    IXA      ref
```

essendo l'indice dell'elemento posto sulla cima della pila il quale viene rimosso al termine dell'operazione.

```
procedure IndexedAddress;
  var  h1, h2, h3 : integer;
begin
  h1 := IR.y;
  h2 := ATab[h1].low;
  h3 := S[SP].i;
  if h3 < h2 then PSW := inxchk else
    if h3 > ATab[h1].high then PSW := inxchk else
      begin
        SP := SP - 1;
        S[SP].i := S[SP].i + (h3 - h2) * ATab[h1].elsize
      end
end;
```

2.2.4 Operazioni Logico-Aritmetiche

Le istruzioni aventi codice di operatore compreso fra 39 e 61 rappresentano le usuali operazioni aritmetiche e logiche su valori interi o reali. In esse si assume che i dati siano posti sulla cima della pila che, al termine dell'operazione, vengono rimossi per lasciar posto al risultato. La forma generale dell'operazione é la seguente

```
procedure OperazioneAritmetica;
begin
  SP := SP - 1;
  S[SP].? := S[SP].? @ S[SP + 1].?
end;

procedure OperazioneLogica;
```

```
begin
  SP := SP - 1;
  S[SP].b := S[SP].? @ S[SP + 1].?
end;
```

dove @ rappresenta una delle operazioni =, ¡¿, ¡, ¡=, ¿, ¿=, or, +, -, and, *, div, / mentre *?* determina il tipo di dato sulla pila. A queste si aggiungono le istruzioni di `not` e `negate` con un solo operando

```
procedure DoNot;
begin
  S[SP].b := not S[SP].b
end;

procedure Negate;
begin
  S[SP].i := -S[SP].i
end;
```

che servono ad invertire un valore booleano o intero.

2.2.5 Istruzioni di Ingresso-Uscita

Le operazioni di ingresso-uscite consentite riguardano solamente lo standard input e lo standard output. Non sono ammessi altri canali di comunicazione con l'esterno. Possono essere letti o scritti, senza ulteriori precisazioni, i tipi predefiniti di intero, reale, boolean e carattere.

```
procedure InitField;
begin
  fld[1] := 7;        {integer}
  fld[2] := 12;       {real mantissa}
  fld[3] := 6;        {real exponent}
  fld[4] := 1         {char}
end;
```

RDD : Read Data

Leggi un valore intero, reale o un carattere dallo standard input e caricalo sulla cima della pila utilizzando lo schema

```
    L    RDD       code
```

essendo *code* il codice del tipo di dato

```
procedure ReadData;
begin
  if eof(input) then PSW := redchk else
    case IR.y of
      1 : read(S[S[SP].i].i);
      2 : read(S[S[SP].i].r);
      4 : read(S[S[SP].i].c)
   end;
  SP := SP - 1
end;
```

WRS : Write String

Scrivi una stringa costante di caratteri sullo standard output secondo lo schema

```
L    WRS       ref
```

essendo il parametro *ref* un indice al primo carattere della stringa nella tabella delle stringhe e la sua lunghezza contenuta sulla cima della pila.

```
procedure WriteString;
  var  h1, h2 : integer;
begin
  h1 := S[SP].i;
  h2 := IR.y;
  SP := SP - 1;
  chrcnt := chrcnt + h1;
  if chrcnt > LineLength then PSW := lngchk;
  repeat
    write(STab[h2]);
    h1 := h1 - 1;
    h2 := h2 + 1
  until h1 = 0
end;
```

WRD : Write Unformatted Data

Scrivi un valore intero, reale, booleano o un carattere sullo standard output, con la formattazione predefinita, utilizzando lo schema

```
L    WRD       code
```

essendo *code* il codice del tipo di dato, posto sulla cima della pila.

```
procedure WriteData;
begin
  chrcnt := chrcnt + fld[IR.y];
  if chrcnt > LineLength then PSW := lngchk else
    case IR.y of
      1 : write(S[SP].i : fld[1]);
      2 : write(S[SP].r : fld[2]);
      3 : write(S[SP].b : fld[3]);
      4 : write(S[SP].c)
    end;
  SP := SP - 1
end;
```

WRC : Write Formatted Constant

Scrivi un valore intero, reale, booleano o un carattere sullo standard output, con la formattazione definita sulla cima della pila, utilizzando lo schema

```
L    WRC       code
```

essendo *code* il codice del tipo di dato, posto sulla cima della pila.

```
procedure WriteConstant;
begin
  chrcnt := chrcnt + S[SP].i;
  if chrcnt > LineLength then PSW := lngchk else
    case IR.y of
      1 : write(S[SP].i : S[SP - 1].i);
      2 : write(S[SP].r : S[SP - 1].i);
      3 : write(S[SP].b : S[SP - 1].i);
      4 : write(S[SP].c : S[SP - 1].i)
    end;
  SP := SP - 2
end;
```

WRR : Write Real

Scrivi un dato reale sullo standard output con la notazione mantissa-esponente con le direttive contenute sulla cima della pila, secondo lo schema

```
L    WRR
```

rimuovendo dato e direttive.

```
procedure WriteReal;
begin
  chrcnt := chrcnt + S[SP - 1].i;
  if chrcnt > LineLength
    then PSW := lngchk
     else write(S[SP - 2].r:S[SP - 1].i:S[SP].i);
  SP := SP - 3
end;
```

RLN : Read Line

Termina la lettura della riga corrente e passa alla riga successiva, purché il file di input non sia terminato, secondo lo schema

```
L    RLN
```

Si tenga presente, comunque, che nel caso di input interattivo da terminale il flag di eof é sempre falso.

```
procedure ReadLine;
begin
  if eof(input) then PSW := redchk
                else readln
end;
```

WLN : Write Line

Termina la scrittura della linea corrente con la marca di eoln e passa alla riga seguente, secondo lo schema

```
L    WLN
```

facendo attenzione di non superare il massimo numero di righe di output che é permesso scrivere dal programma.

```
procedure WriteLine;
begin
  writeln;
  lncnt := lncnt + 1;
  chrcnt := 0;
  if lncnt > LineLength then PSW := linchk
end;
```

Riassumendo, l'esecuzione delle istruzioni della macchina a pila é simulata con una chiamata selettiva, dipendente dal codice dell'istruzione, di una procedura che manipola i dati contenuti nella pila e in alcuni registri, seguendo uno schema del tipo indicato

```
procedure Execute (Op: OpCode);
begin
  if PFlag then NewProcess(Running, Op);
    case Op of
      {esegui l'istruzione avente codice Op\}
    end
end;
```

L'esecuzione di NewProcess consente di trasformare la chiamata di procedura nell'attivazione di una coroutine secondo le caratteristiche che saranno meglio specificate nella sezione seguente.

2.3 PROCESSI

Supponiamo, ora, di voler estendere la macchina a pila presentata nelle sezioni precedenti al fine di permettere l'esecuzione contemporanea di piú programmi. E' noto che la contemporaneitá é fittizia in quanto, solo un programma per volta puó essere eseguito se si ha a disposizione una sola *CPU*.

Nell'introduzione si é definito *processo l'esecuzione di un programma* e questo comporta la disponibilitá dell'Unitá Centrale di Elaborazione oltreché, ovviamente, della Memoria Centrale in cui deve essere caricato il programma assieme ai propri dati. Per quanto riguarda quest'ultima viene, assegnato un segmento dei dati sulla pila di lunghezza fissa per ciascuno dei processi generabili, per cui viene stabilito a priori l'indirizzo della prima e dell'ultima locazione.

L'impiego della *CPU*, invece, é ripartito fra i vari processi con l'usuale tecnica della multiprogrammazione, nel senso che viene assegnato, al processo selezionato per l'esecuzione, un quanto di tempo al termine del quale, se il processo non viene bloccato su un semaforo o é terminato, é costretto al rilascio della *CPU*.

2.3.1 Descrittore di Processo

Il descrittore di processo, oltre a contenere l'immagine dei 10 registri che puntano sulla pila e del *contatore di programma* PC[9], consiste di due ulteriori campi, indicati come *Blocked* e *Ready*, il primo dei quali, se non nullo, contiene l'indirizzo di un semaforo e il secondo un valore booleano che é posto a *true* quando il processo é attivo. Queste due informazioni sono utilizzate dallo scheduler.

```
const   MaxProcess = 9;
              MAIN = 0;

type       Process = 0..MaxProcess;

           PSBlock = record
                       SPImage: StackRef;
                       FPImage: StackRef;
                       PCImage: MemRef;
                       SLImage: StackRef;
                       DxImage: Display;
                       BlockedImage: integer;
                       ReadyImage: boolean
                     end;

var          PFlag : boolean;
           Running : Process;
              Pool : Process;
           Blocked : integer;
             Ready : boolean;

            PSBTab : array[Process] of PSBlock;
```

Le variabili globali *Running* e *Pool* sono utilizzate, rispettivamente, per contenere l'indice del processo correntemente in esecuzione e il numero di processi attivi.

2.3.2 Orologio

Se vogliamo simulare un'accettabile parallelismo fra le coroutine, é necessario disporre di un meccanismo d'interruzione d'orologio che permetta la realizzazione di una sequenza pseudocasuale di intervalli di tempo da assegnare a ciascuna di esse in modo che questa sia sempre diversa ad ogni attivazione del programma.

La procedura *SetRandom* é in grado di stabilire un diverso valore inziale della sequenza pseudocasuale in quanto questo valore é determinato dall'ora corrente ricavabile accedendo all'orologio interno della CPU mediante la funzione di sistema *GetTime*.

```
procedure SetRandom (var K: longint);
   var  A : integer;
        T : DateTimeRec;
begin
  with T do
    begin
      GetTime(T);
```

[9]e per questo é utilizzata dal dispatcher

```
      A := 11 * minute + 41 * second;
      K := 10 * A + 49 * hour + 31 * day;
    end
end;
```

La funzione *Randomize* restituisce un valore pseudocasuale, compreso fra 0 e 1, utilizzando la funzione predefinita *random* e il valore generato dalla procedura di cui si é detto e passato come parametro.

```
function Randomize (S: longint): real;
  var  K, L, M: longint;
begin
  K := (32768 + random + S) mod 65536;
  L := 65536 - K;
  M := 65536 + K;
  Randomize := L / M
end;
```

Definito un valore massimo *TimeSlice* di *quanto di tempo* assegnabile ad ogni coroutine, la procedura *SetClock* sceglie in modo casuale di assegnarne una parte in base al valore corrente della funzione Randomize.

```
procedure SetClock (S: longint; var T: ClockTick);
begin
  T := trunc(Randomize(S) * TimeSlice) + 1
end;
```

Si noti come tale funzione sia realizzata in modo che tale valore non possa essere mai nullo.

2.3.3 Gestione della Coda

La scelta della politica di gestione della coda dei processi gioca un ruolo decisivo nella caratterizzazione dell'ambiente di esecuzione delle coroutine. Si é osservato in precedenza come lo schema dovrebbe dipendere dal meccanismo di scambio di contesto normalmente utilizzato. In questo caso, poiché *tale scambio é implicito ed é basato sull'interruzione d'orologio*, é conveniente considerare la coda dei processi organizzata circolarmente, con un puntatore mobile al primo di essi la cui posizione corrente viene aggiornata ad ogni richiesta di schedulazione, spostandolo in avanti di un numero di processi casuale.

La procedura *SetProcess* ha proprio il compito di eseguire tale aggiornamento con una chiamata alla funzione Randomize in maniera che il nuovo processo cui far riferimento non sia mai quello precedente.

```
procedure SetProcess (S: longint; var P: Process);
begin
  P := trunc(Randomize(S) * MaxProcess) + 1
end;
```

Le condizioni che devono essere verificate affinché un processo sia eseguibile e che sia attivo[10] e non sia bloccato su alcun semaforo

```
function Runnable (P: Process): boolean;
begin
  with PSBTab[P] do
    Runnable := ReadyImage and (BlockedImage = 0)
end;
```

[10] espresso dal campo booleano Ready

La politica seguita dallo scheduler é quella di generare, in modo casuale, un nuovo processo di riferimento a partire dal quale viene percorsa tutta la coda circolare alla ricerca di un processo pronto per l'esecuzione. Se, dopo una scansione completa, si ritorna allo stesso processo vuol dire che non esistono processi pronti, ossia si é verificata una condizione di stallo. Viceversa, il processo é selezionato e gli viene assegnato un quanto di tempo.

```
function Schedule (P: Process) : Process;
  var  Q : Process;
       Count: integer;
begin
  if not Runnable(MAIN) then
    begin
      SetProcess(Seed, Q);
      Q := (P + Q) mod (MaxProcess + 1);
      Count := MaxProcess + 1;
      while not (Runnable(Q) or (Count < 0)) do
        begin
          Count := Count - 1;
          Q := (Q + 1) mod (MaxProcess + 1)
        end;
      if Count < 0 then PSW := deadlocked
                   else SetClock(Seed, Tick);
      Schedule := Q
    end else
    Schedule := MAIN
end;
```

Indipendentemente dal fatto che il programma principale dará luogo ad una o piú coroutine, questo deve essere trattato come processo[11] e vedere inizializzato il proprio contesto con la procedura *InitMainContext* la quale, comunque, riserverá solo una porzione della pila.

```
procedure InitMainContext;
begin
  with PSBTab[0] do
    begin
      FPImage := 0;
      BlockedImage := 0;
      DxImage[1] := 0;
      SPImage := Envr[2].vsize - 1;
      PCImage := AtomT[S[4].i].adr;
      ReadyImage := true;
      SLImage := StackSize - MaxProcess * WorkSpace
    end
end;
```

Per quanto riguarda, invece, i contesti dei possibili altri processi, la procedura di inizializzazione *InitContext* deve solo stabilire la dimensione e i limiti della parte di pila utilizzabile assieme al fatto che questo non puó essere impiegato prima che venga attivato dal programma principale.

[11] al quale viene assegnato l'indice 0

```
procedure InitContext (P: Process);
begin
  with PSBTab[P] do
    begin
      ReadyImage := false;
      DxImage[1] := 0;
      PCImage := 0;
      BlockedImage := 0;
      FPImage := PSBTab[P - 1].SLImage + 1;
      SLImage := FPImage + WorkSpace - 1;
      SPImage := FPImage - 1
    end
end;
```

Le due procedure che seguono realizzano le funzioni del dispatcher, in quanto hanno il compito di caricare i registri della SC con i valori precedentemente salvati nell'apposito descrittore

```
procedure LoadContext (P: Process);
var  I : integer;
begin
  with PSBTab[P] do
    begin
      SP := SPImage;
      FP := FPImage;
      PC := PCImage;
      SL := SLImage;
      for I := 1 to LMax do Dx[I] := DxImage[I];
    end
end;
```

e di salvare i valori correnti della coroutine quando questa viene interrotta, sia per aver terminato il quanto di tempo assegnato, sia perché bloccata da un'operazione di semaforo.

```
procedure SaveContext (P: Process);
  var  I : integer;
begin
  with PSBTab[P] do
    begin
      SPImage := SP;
      FPImage := FP;
      PCImage := PC;
      SLImage := SL;
      for I := 1 to LMax do DxImage[I] := Dx[I];
    end
end;
```

Le ulteriori due procedure riportate di seguito hanno il compito di rendere esplicito il meccanismo di gestione della coda dei processi pronti, sospendendo il processo correntemente in esecuzione

```
procedure Preempted (var Pid: Process);
begin
```

```
    SaveContext(Pid)
    with PSBTab[P] do
      begin
        BlockedImage := Blocked;
        ReadyImage := Ready
      end;
    Pid := Running
  end;
```

oppure, servendone un altro, dopo che é stato selezionato dallo scheduler

```
  procedure Serve (Pid: Process);
  begin
    LoadContext(Pid);
    with PSBTab[P] do
      begin
        Blocked := BlockedImage;
        Ready := ReadyImage
      end;
    Running := Pid
  end;
```

in seguito al sopravvenire di interruzioni di orologio, generate dalla procedura *ClockInterrupt* non appena il quanto di tempo assegnato alla coroutine corrente é scaduto.

```
  procedure ClockInterrupt;
  var OldPid, NewPid : Process;
  begin
    if (Tick = 0) or Runnable(MAIN) then
      begin
        Preempted(OldPid);
        NewPid := Schedule(OldPid);
        Serve(NewPid)
      end else
      Tick := Tick - 1
  end;
```

Si noti come l'interruzione di orologio possa determinare la ripresa dell'esecuzione del programma principale, quando questo si trova nello stato di pronto, perché é in fase di creazione delle coroutine dichiarate all'interno dell'istruzione **cobegin**/**coend** del programma principale.

Con la procedura *NewProcess* il sistema associa alla marcatura della pila, che rappresenta il primo *record di attivazione* della coroutine in esame, la corretta sincronizzazione dei dati nel corrispondente descrittore. Si osservi come nell'attuale implementazione sia assegnato un *segmento di lunghezza fissa*[12] per ciascun processo creato.

```
  procedure NewProcess (Op: OpCode);
  var OldPid : Process;
  begin
    Preempted(OldPid);
    if Op = 18 then Pool := Pool + 1;
    Serve(Pool)
  end;
```

[12] partizione statica della memoria

2.3.4 Istruzioni di Processo

In base alle considerazioni precedenti, la gestione delle coroutine come processi concorrenti é ottenuta implementando lo schema cobegin/coend con due istruzioni di inizio e fine computazione multipla e ampliandolo con i semafori per realizzare, cosí, un generico grafo di interdipendenza fra processi.

L'unica limitazione introdotta é quella di consentire la creazione dei processi solo nel programma principale in base al fatto che questi sono gestiti con un'allocazione statica della memoria.

FORK : Fork at Cobegin

Attiva il meccanismo che permette di eseguire le procedure che seguono questa istruzione come coroutine

```
L    FORK
```

permettendo la corretta preparazione dei dati sul relativo segmento.

```
procedure CoBegin;
begin
  PFlag := true
end;
```

JOIN : Join at Coend

Disattiva il meccanismo che consente di eseguire le procedure come coroutine secondo lo schema indicato

```
L    JOIN
```

essendo questa l'ultima istruzione del blocco **cobegin-coend**.

```
procedure CoEnd;
begin
  PFlag := false;
  PSBTab[0].ReadyImage := false
end;
```

WAIT : Wait on Semaphore

Attendi il verificarsi di un evento sul semaforo il cui indirizzo é passato sulla cima della pila

```
L    WAIT
```

nel caso che questo abbia valore nullo, altrimenti continua decrementandone il suo valore.

```
procedure Wait;
  var  h : integer;
begin
  h := S[SP].i;
  SP := SP - 1;
  if S[h].i > 0 then S[h].i := S[h].i - 1 else
    begin
      Blocked := h;
      Tick := 0
    end
end;
```

SGNL : Signal to Semaphore

Segnala al semaforo in cui indirizzo é contenuto sulla cima della pila che si é reso disponibile un nuovo segnale temporale

```
L    SGNL
```

il quale viene accumulato, incrementando il valore del semaforo, se nessuna coroutine é bloccata su quel semaforo.

```
procedure Signal;
  var  h1, h2 : integer;
       h3 : Process;
begin
  h1 := S[SP].i;
  SP := SP - 1;
  h2 := MaxProcess + 1;
  SetProcess(Seed, h3);
  while (h2 >= 0) and (PSBTab[h3].BlockedImage <> h1) do
    begin
      h3 := (h3 + 1) mod (MaxProcess + 1);
      h2 := h2 - 1
    end;
  if h2 < 0 then S[h1].i := S[h1].i + 1
              else PSBTab[h3].BlockedImage := 0
end;
```

In conclusione, l'estensione della macchina a pila per consentire l'esecuzione concorrente di piú coroutine si ottiene fornendo quattro nuove istruzioni, due per la loro creazione e due per permetterne la sincronizzazione per mezzo di variabili intere globali, i semafori. Questi ultimi verranno trattati piú diffusamente in un capitolo successivo.

2.4 ASSEMBLATORE

I programmi che possono essere eseguiti dalla macchina a pila SC risiedono come file su disco, in formato testo, con la classica struttura di un linguaggio assembler, vale a dire

Label OpCode Parm1 Parm2 ....

dove ciascuna riga rappresenta una delle istruzioni discusse nelle sezioni precedenti con il corretto numero di paramentri.

2.4.1 Caricamento del Programma

Prima dell'esecuzione, il programma deve essere caricato in memoria dal file su cui risiede, con estensione **.as**, e questo viene realizzato dalla seguente procedura.

```
procedure OpenFile (var f, g: text);
  type   FileName = string;

  var           K : integer;
             Name : FileName;
```

```
            Target : FileName;

begin
  IOCheck(false);
  writeln;
  write(' Enter filename : ');
  repeat
    readln(Name);
    Target := concat(Name, '.as');
    reset(f, Target);
    K := IOResult;
    if K <> 0 then
      begin
        writeln(' File not found');
        write(' Enter filename : ')
      end
  until K = 0;
  IOCheck(true);
  if DebugF then
     begin
        Name := concat(Name, '.db');
        rewrite(g, Name)
     end
 end;
```

2.4.2 Assemblaggio del Programma

L'operazione di assemblaggio del programma viene ottenuta leggendo riga per riga il programma sorgente in formato assembler e codificando le sue istruzioni che, a seconda del tipo vengono caricate in sezioni diverse di memoria, agli indirizzi indicati dalla corrispondente etichetta.

Come si puó notare, a parte le istruzioni che rappresentano operazioni proprie della macchina SC e che vengono caricate nella memoria riservata al codice, tutte le altre[13] servono ad inizializzare le opportune tabelle interne, immagazzinate alla base della pila, per permettere la corretta esecuzione del programma.

```
procedure LoadCode (var f: text);
  const     SMax = 5;

  type    SIndex = 1..SMax;
         PString = packed array[SIndex] of char;
        Mnemonic = array[OpCode] of PString;

  var          K : integer;
              sx : StringRef;
              Op : OpCode;
              Dx : Index;
              Da : Value;
               W : PString;
```

[13]che si distinguono dalle precedenti perché iniziano con un punto e, perció, vengono dette anche direttive o pseudoistruzioni

```
             CH : char;
            Mne : Mnemonic;

 procedure InitMnemonic;
 begin
   {viene inizializzato il vettore dei codici mnemonici}
 end;

 procedure SkipComment (var f: text; var CH: char);
 begin
   repeat
     if eoln(f) then
       begin
         readln(f);
         read(f, CH);
         if CH = ';' then
           begin
             while not eoln(f) do read(f, CH);
             CH := ' '
           end
       end else
     read(f, CH);
   until CH <> ' '
 end;

 procedure GetLabel (var f: text; var CH: char; var K: integer);
   var  Zero : integer;
 begin
   if CH in ['0'..'9'] then
     begin
       Zero := ord('0');
         K := 0;
         repeat
           K := 10 * K + (ord(CH) - Zero);
           read(f, CH)
         until not (CH in ['0'..'9'])
     end
 end;

function GetCmd (var f: text; CH: char; var W: PString): boolean;
   var  I : SIndex;
begin
  while CH = ' ' do read(f, CH);
  for I := 1 to SMax do
    begin
      W[I] := CH;
      read(f, CH)
    end;
  GetCmd := W[1] = '.'
end;
```

```
procedure LookUp (T: PString; var Op: OpCode);
  var  Found : boolean;
begin
  Op := 0;
  repeat
    Found := T = Mne[Op];
    if not Found then Op := Op + 1;
  until Found
end;

procedure LoadDirective (var f: text; W: PString);
begin
  if W = '.NAME' then
    with AtomT[K] do
      readln(f, lev, adr, ref, link, name)
  else if W = '.WORD' then
    with Envr[K] do
      readln(f, psize, vsize, last)
  else if W = '.BLCK' then
    with ATab[K] do
      readln(f, low, high, elsize)
  else if W = '.LONG' then
    readln(f, RTab[K])
  else if W = '.DATA' then
    begin
      repeat read(f, CH) until CH = '"';
      read(f, CH);
      while not eoln(f) do
        begin
          STab[sx] := CH;
          sx := sx + 1;
          read(f, CH)
        end
    end else
    if W = '.END ' then
      begin
        while not eoln(f) do read(f, CH);
        readln(f)
      end
 end;

 procedure LoadInstruction (var f: text; K: MemRef; Op: OpCode);
 begin
   if Op < 4 then
     begin
       readln(f, Dx, Da);
       with Mem[K] do
         begin
           f := Op;
```

```
                x := Dx;
                y := Da
              end
        end else
        if Op < 31 then
          begin
            readln(f, Da);
            with Mem[K] do
              begin
                f := Op;
                y := Da
              end
          end else
          begin
            readln(f);
            with Mem[K] do f := Op;
          end
    end;

  begin
    sx := 0;
    InitMnemonic;
    while not eof(f) do
      begin
        SkipComment(f, CH);
        GetLabel(f, CH, K);
        if GetCmd(f, CH, W) then LoadDirective(f, W) else
          begin
            LookUp(W, Op);
            LoadInstruction(f, K, Op)
          end
      end
  end;
```

Si noti come il corpo principale della procedura definisce il semplice algoritmo di assemblaggio utilizzato, il quale prevede anche la possibilitá che compaiano righe di commento purché inizino con il punto e virgola.

2.4.3 Terminazione Anomala

Qualora si dovesse verificare un errore durante l'esecuzione del programma il cui tipo comparirebbe nel registro *PSW*, viene prodotta un'immagine del suo stato esecutivo con la seguente procedura.

```
  procedure PostMortemDump;
    var  h1, h2, h3 : integer;

  begin
    writeln;
    write(' halt at', PC : 5);
    write(' in process ', Running : 2, ' because of ');
    case PSW of
```

```
    deadlocked : writeln('deadlock');
        caschk : writeln('undefined case');
        divchk : writeln('division by zero');
        inxchk : writeln('invalid index');
        stkchk : writeln('storage overflow');
        linchk : writeln('too much output');
        lngchk : writeln('line too long');
        redchk : writeln('reading past end of file')
  end;
  writeln;
  writeln(' Process    PC   Blocked  Ready');
  for h1 := 0 to Pool do
    with PSBTab[h1] do
      if ReadyImage then
        writeln(h1 : 8, PC : 6, Blocked : 10, '   ', Ready) else
        begin
          write(h1 : 8, PCImage : 6);
          write(BlockedImage : 10, '   ', ReadyImage)
        end;
  writeln;
  writeln(' Global Variables');
  h1 := FP;
  blkcnt := 10;
  repeat
    blkcnt := blkcnt - 1;
    if blkcnt = 0 then h1 := 0;
    h2 := S[h1 + 4].i;
    if h1 <> 0 then
      begin
        write('   ', AtomT[h2].name, '  called at ');
        writeln(S[h1 + 1].i : 5)
      end;
    h2 := Envr[AtomT[h2].ref].last;
    h1 := S[h1 + 3].i
  until h1 < 0;
  writeln
end;
```

Per avere maggiori dettagli sul valore delle variabili é necessario conservare tutte le informazioni della *tabella dei simboli* generata dal compilatore.

2.5 OSSERVAZIONI

La descrizione della macchina a pila presentata nelle sezioni precedenti non é intesa tanto ad essere utilizzata direttamente per programmare algoritmi concorrenti ma, piuttosto, a prendere confidenza con gli aspetti piú strettamente implementativi che accompagnano sempre la stesura di tali algoritmi e che si riferiscono al funzionamento del nucleo dei Sistemi Operativi.

Rispetto alla programmazione sequenziale va posta particolare cura nello scegliere il modello di cooperazione fra le componenti sequenziali tenendo presente che il parallelismo puó innescare errori dipendenti dal tempo se non viene circoscritto e ben controllato. La tecnica dei grafi di dipendenza dei dati fornisce un ottimo strumento per identificare e gestire le forme di parallelismo regolare degli algoritmi.

Nei capitoli che seguono queste problematiche verranno affrontate in modo sistematico utilizzando la forza descrittiva del **concurrent pascal**, il linguaggio sorgente in cui verranno espressi gli algoritmi paralleli. Per ragioni di semplicitá implementativa del compilatore e di efficienza di esecuzione dell'interprete della *macchina a pila* il linguaggio PASCAL preso in esame sará un sottoinsieme del Pascal Standard, corrispondente al **pascals** di Wirth, esteso con il costrutto **cobegin**/**coend** per la creazione di processi e le operazioni **wait** e **signal** su semafori, realizzati come variabili intere, definite all'interno del programma principale.

Nel capitolo seguente verrá data una breve presentazione del linguaggio, le sue caratteristiche salienti, la sua grammatica ed alcuni esempi compilati in **pcode** di semplici algoritmi sequenziali.

Capitolo 3

Programmare in Concurrent Pascal

Sebbene il linguaggio *pcode* utilizzato dalla macchina a pila presentata nel capitolo precedente possa considerarsi completo dal punto di vista della capacitá di codifica degli algoritmi concorrenti, compresi quelli usati normalmente nell'implementazione dei Sistemi Operativi, non é certo il piú adatto ad esprimere tali algoritmi in modo semplice e coinciso.

D'altra parte la semplicitá espressiva di un linguaggio difficilmente si coniuga con la semplicitá d'implementazione del suo compilatore e del relativo supporto run-time. Nel caso del Concurrent Pascal la scelta delle caratteristiche *sequenziali* da incorporare sono strettamente legate alla necessitá di avere uno strumento sufficientemente ricco per codificare in modo diretto ed elegante i piú comuni algoritmi concorrenti, che non sia a discapito di un compilatore troppo oneroso da realizzare, specialmente riguardo alla sua robustezza di fronte ai piú comuni errori che possono intervenire nel normale utilizzo del linguaggio.

3.1 IL LINGUAGGIO

L'impiego del *Pascal-S* come piattaforma sequenziale di base, a partire dalla quale estendere il linguaggio verso i costrutti della concorrenza, é stata motivato da ragioni di convenienza sia per la capacitá espressiva che per la semplicitá implementativa, nel senso che si presenta come un buon compromesso nel fornire un ragionevole supporto di strutture dati e costrutti di controllo pur rimanendo duttile ed efficiente nell'esecuzione.

Nel Pascal-S sono ammessi solo due tipi di strutture dati: gli **array** e i **record** senza varianti. Non compaiono, invece, né gli insiemi né i file ad eccezion fatta per i *file testuali* che rappresentano lo standard input e lo standard output i quali, sebbene dichiarati implicitamente, devono essere listati nell'intestazione del programma. Inoltre sono stati volutamente omessi tutti i tipi di puntatore e, quindi, anche le strutture dinamiche. Nella figura 3.1 sono schematizzati i tipi di dati ammessi.

Naturalmente sono state tralasciate tutte le opzioni di packing[1]. Per quanto riguarda invece le strutture dati predefinite, compaiono tutti e quattro gli usuali tipi primitivi del Pascal: **integer**, **real**, **boolean** e **char** mentre sono stati omessi gli scalari e i *subrange*.

La scelta delle strutture dati e dei relativi tipi predefiniti contribuisce in modo determinante alla complessitá del processo di elaborazione che, invece, é poco sensibile alle strutture di controllo utilizzate. Per questa ragione sono presenti pressocché tutte le istruzioni di controllo del Pascal ad eccezion fatta per le istruzioni *with* e *goto*. Le procedure e funzioni possono essere pienamente utilizzate se si esclude il fatto che non possono essere passate come parametri di altre procedure e/o funzioni. La figura 3.3 dá una precisa immagine delle modalitá d'impiego delle strutture di controllo ammesse.

Poiché non é riconosciuta la dichiarazione di file, sono state altresí escluse le operazioni primitive di *get* e *put* restringendo le operazioni sui file alle sole procedure di **read** sullo standard *input* e di

[1] packed record, packed array

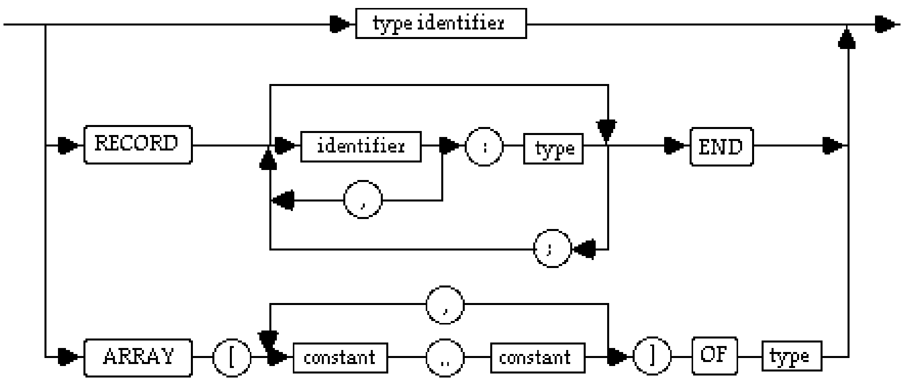

Figure 3.1: strutture dati

write sullo standard *output*. Pertanto non sono nemmeno disponibili le operazioni di *reset* e *rewrite* mentre ci si puó avvelere degli usuali strumenti di formattazione dell'output, giá presenti nel Pascal.

Riassumendo, l'attuale versione del Concurrent Pascal[2], basata su un'appropriata estensione del Pascal-S, mette a disposizione i seguenti oggetti predefiniti

costanti :	*true*, *false*
tipi :	*integer*, *real*, *boolean*, *char*
funzioni :	*abs*, *sqr*, *odd*
	chr, *ord*, *succ*, *pred*
	round, *trunc*
	sin, *cos*, *exp*, *ln*, *sqrt*, *arctan*
	eof, *eoln*
	wait, *signal*
procedure :	*read*, *readln*, *write*, *writeln*

con la restrizione che le funzioni *succ* e *pred* sono applicabili solamente ad argomenti di tipo *char*. L'argomento di *ord* puó essere di tipo *char*, *boolean* oppure *integer*.

L'estensione concorrente é ottenuta per mezzo dell'istruzione di controllo **cobegin**/bf coend che puó apparire solamente nel programma principale, come appare chiaramente dalla figura 3.2, e il cui effetto é quello di *eseguire le procedure che compaiono al suo interno come coroutine*. Questo significa che l'ambiente di esecuzione dell'attuale implementazione del Concurrent Pascal ammette solamente una creazione statica di processi i quali durante l'esecuzione possono sincronizzarsi fra loro per mezzo di variabili intere globali[3] e che prendono il nome di **semafori**. Su di esse vanno eseguite solamente le operazioni *wait* e *signal* a parte la loro inizializzazione che deve essere effettuata prima della creazione dei processi.

Poiché ciascuno processo alloca staticamente un segmento privato di stack, per permettere lo scambio di dati é necessario definire oppurtune variabili di comunicazione a livello del programma principale. Nei capitoli seguenti verrá adeguatamente esemplificato come procedere a riguardo.

Un discorso a parte andrebbe fatto per i messaggi d'errore che il compilatore produce durante la fase di analisi lessicale e sintattica. In questo caso, ovviamente, non viene prodotto alcun codice eseguibile[4] ma viene prodotto un listato del programma sorgente con l'indicazione nel testo della

[2]brevemente, CPascal
[3]pertanto, definite nel programma principale
[4]interpretato dal simulatore della macchina a pila

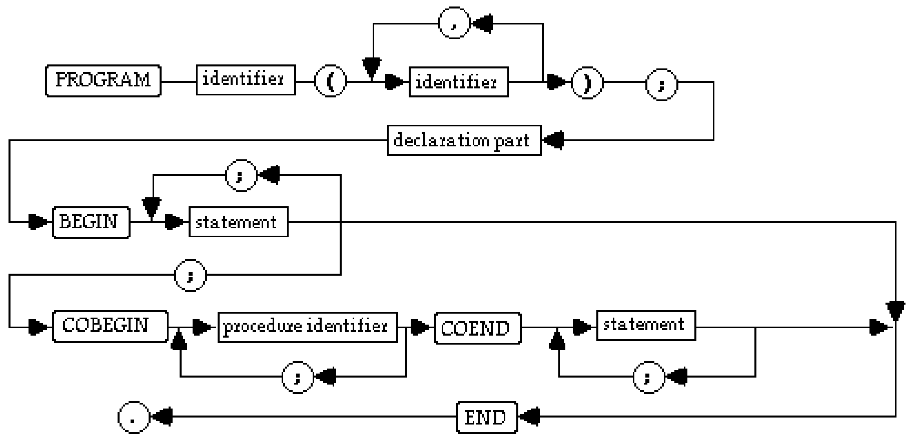

Figure 3.2: blocco principale

posizione dell'errore ed un mnemonico che informa sul tipo di errore rilevato.

Si noti, comunque, che il codice eseguibile dalla macchina a pila SC risiede su un file testuale che ha lo stesso nome del programma sorgente in Concurrent Pascal con l'estensione **.as** che sostituisce l'originale **.p**, obbligatoria se si vuole che venga riconosciuto dal compilatore.

3.2 COMPILAZIONE IN PCODE

Per comprendere meglio il tipo di corrispondenza fra programma sorgente CPascal ed il suo compilato in formato **pcode**, viene riportato nel seguito il listato di un semplice programma sequenziale che calcola il fattoriale in varie forme al quale é stato aggiunto il risultato commentato della compilazione. Vale la pena di osservare che l'*eseguibile* cosí ottenuto presenta alcune differenze sostanziali rispetto ad un file *oggetto* standard, perché mantiene tutte le pseudoistruzioni, discusse nel capitolo 2, le quali altro non sono che descrizioni del contenuto di tabelle[5] e che sono impiegate dall'interprete[6] durante l'esecuzione del programma.

Il programma in esame dá un'idea abbastanza chiara dell'organizzazione della macchina a pila e del modo di procedere della computazione. Per i dettagli delle istruzioni si consulti il capitolo 2. Si tenga presente, comunque, che per ottenere il compilato, commenti a parte, nella forma seguente é necessario introdurre nel file sorgente la direttiva $\{\$V+\}$ che istruisce il compilatore a produrre il file assemblato in un formato piú leggibile, organizzato come una sequenza di procedure, funzioni e programma principale.

Ciascuna sezione di codice é strutturata in due parti, la prima comprendente tutte le dichiarazioni necessarie a definire l'area dei dati su cui lavorano le istruzioni che appaiono nella seconda parte. Lo schema generale é quello riportato di seguito

[5]quelle rappresentate alla base della pila nella fig. 2.1
[6]il simulatore della macchina a pila

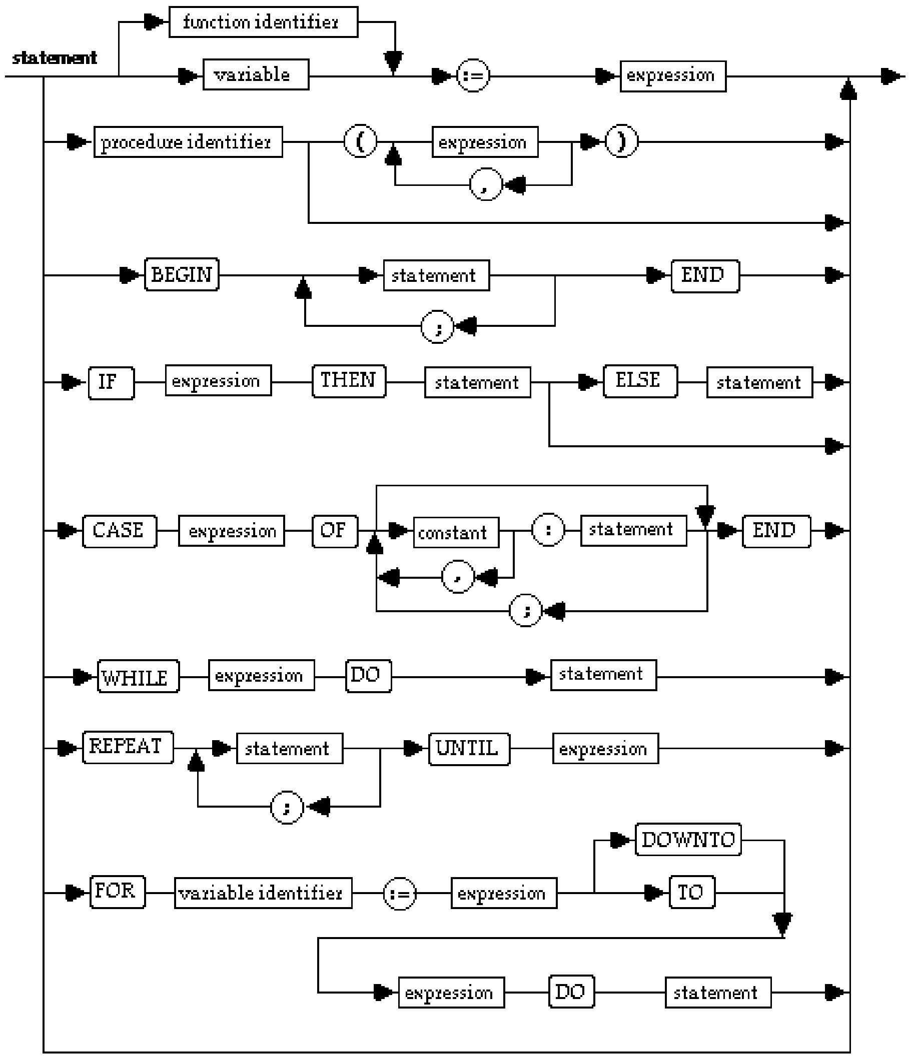

Figure 3.3: strutture di controllo

L	.NAME	Lev	N	M	...	procname
L+1	.NAME	Lev+1	5	...	0	var1
...	...	...	...	...	...	...
L+k	.NAME	Lev+1	5+r	...	...	vark
M	.WORD	p+5	k+5	L+k		
N	OpCode	....				
...	...	....				
...	...	....				
N+s	RET P					

dove *Lev* indica il livello di incassamento sintattico della procedura e/o funzione, **r** lo spazio delle variabili, **s** la dimensione del codice e **p** il numero dei parametri.

Si noti che la prima riga contiene tutte le informazioni necessarie all'istruzione *MST* di marcatura della pila per costruire correttamente il nuovo segmento nel quale andrá ad operare la procedura. Le righe successive si riferiscono alle variabili utilizzate con ultima pseudoistruzione adibita a mantenere le informazioni vitali sul numero dei parametri e la dimensione dello spazio delle variabili.

Ritornando all'esempio proposto si puó osservare come il programma metta in evidenza la differenza fra definizione ricorsiva e iterativa di una funzione che, successivamente, viene presentata in forma di procedura. La numerazione che accompagna ciascuna riga serve ad identificare facilmente la sezione di codice prodotta durante la fase di compilazione.

```
;    0   program DoFact( input, output );
;    0   {$V+}
;    0   var   N : integer;
;    0         T1, T2, T3 : integer;
;    0
;    0   {---- fattoriale come funzione ricorsiva ----}
;    0
;    0     function RFact( K : integer ) : integer;
;    0     begin
;    0     if K = 0
;    2       then RFact := 1
;    6       else RFact := K * RFact(K - 1)
;   14     end;
;   18
;   18   {---- fattoriale come funzione iterativa ----}
;   18
;   18     function IFact( K : integer ) : integer;
;   19     var X : integer;
;   19     begin
;   19       X := 1;
;   22       while K > 0 do
;   26         begin
;   26           X := K * X;
;   31           K := K - 1
;   34         end;
;   37       IFact := X
;   38     end;
;   40
;   40   {---- fattoriale come procedura per evidenziare il ---}
;   40   {--- passaggio di parametri per valore e indirizzo ---}
;   40
;   40     procedure PFact( K : integer; var X: integer );
;   41     begin
;   41       X := 1;
;   44       while K > 0 do
;   48         begin
;   48           X := K * X;
;   53           K := K - 1
;   56         end;
;   59     end;
;   59
;   59   begin
;   60     writeln('Calcolo del Fattoriale');
```

```
;  63     repeat
;  63       write('Enter N : ');
;  65       readln( N );
;  68       PFact(N,T1);
;  72       T2 := IFact( N );
;  77       T3 := RFact(N);
;  82       writeln( T1, T2, T3 )
;  88     until N = 0;
;  93     writeln
;  93   end.
```

Il risultato della compilazione della funzione fattoriale in forma ricorsiva é quello che appare di seguito. La prima riga definisce la funzione, la seconda l'unico parametro utilizzata mentre la terza serve a stabilire la dimensione del segmento di pila da riservare durante la chiamata.

```
35     .NAME    1    0    3   34  rfact
36     .NAME    2    5    0    0  k
 3     .WORD    6    6   36
 0     LDC      2    5     ; carica il contenuto di k
 1     ENT           0     ; carica il valore 0
 2     EQL I               ; k = 0 ?
 3     FJP           8     ; in caso negativo salta ad 8
 4     LDA      2    0     ; carica l'indirizzo di k
 5     ENT           1     ; carica il valore 0
 6     STO                 ; memorizza
 7     UJP          18     ; vai a 18
 8     LDA      2    0     ; carica l'indirizzo di rfact
 9     LDC      2    5     ; carica il contenuto di k
10     MST          35     ; prepara la chiamata di rfact
11     LDC      2    5     ; carica il contenuto di k
12     ENT           1     ; carica il valore 1
13     SBI                 ; sottrai
14     CUP           5     ; chiama rfact
15     UDX      1    2     ; aggiorna i registri indice
16     MPI                 ; moltiplica
17     STO                 ; immagazzina
18     RET F               ; ritorna dalla chiamata
```

In modo del tutto analogo viene compilata la forma iterativa del fattoriale con l'unica differenza che intestazione prevede la presenza di una variabile locale al segmento di attivazione.

```
37     .NAME    1   19    4   35  ifact
38     .NAME    2    5    0    0  k
39     .NAME    2    6    0   38  x
 4     .WORD    6    7   39
19     LDA      2    6     ; carica l'indirizzo di x
20     ENT           1     ; carica il valore 1
21     STO                 ; immagazzina
22     LDC      2    5     ; carica il contenuto di k
23     ENT           0     ; carica il valore 0
24     GTR I               ; k > 0 ?
```

```
25     FJP          37     ; in caso negativo salta a 37
26     LDA      2    6     ; carica l'indirizzo di x
27     LDC      2    5     ; carica il contenuto di k
28     LDC      2    6     ; carica il contenuto di x
29     MPI                 ; moltiplica
30     STO                 ; immagazzina
31     LDA      2    5     ; carica l'indirizzo di k
32     LDC      2    5     ; carica il contenuto di k
33     ENT           1     ; carica il valore 1
34     SBI                 ; sottrai
35     STO                 ; immagazzina
36     UJP          22     ; salta a 22
37     LDA      2    0     ; carica l'indirizzo di ifact
38     LDC      2    6     ; carica il contenuto di x
39     STO                 ; immagazzina
40     RET F               ; ritorna dalla chiamata
```

Del tutto simile si presenta la compilazione del fattoriale come procedura con la sola differenza che il risultato della computazione non viene memorizzato nella prima locazione del segmento di stack associato.

```
40     .NAME    1   41    5   37  pfact
41     .NAME    2    5    0    0  k
42     .NAME    2    6    0   41  x
 5     .WORD    7    7   42
41     LDC      2    6     ; carica il contenuto di x
42     ENT           1     ; carica il valore 1
43     STO                 ; immagazzina
44     LDC      2    5     ; carica il contenuto di k
45     ENT           0     ; carica il valore 0
46     GTR I               ; k > 0 ?
47     FJP          59     ; in caso negativo salta a 59
48     LDC      2    6     ; carica il contenuto di x
49     LDC      2    5     ; carica il contenuto di k
50     LDI      2    6     ; carica indirettamente x
51     MPI                 ; moltiplica
52     STO                 ; immagazzina
53     LDA      2    5     ; carica l'indirizzo di k
54     LDC      2    5     ; carica il contenuto di k
55     ENT           1     ; carica il valore 1
56     SBI                 ; sottrai
57     STO                 ; immagazzina
58     UJP          44     ; salta a 44
59     RET P               ; ritorna dalla chiamata
```

Un disorso a parte va fatto per il programma principale, che deve contenere tutte le variabili globali necessarie per l'attivazione delle procedure e funzioni coinvolte, oltre a stabilire l'accesso da e per l'esterno. Si noti, fra l'altro, la differenza fra meccanismo di chiamata delle funzioni da quello delle procedure.

```
30     .NAME    0   60    2   29  $Main
```

```
31      .NAME    1    5    0    0  n
32      .NAME    1    6    0   31  t1
33      .NAME    1    7    0   32  t2
34      .NAME    1    8    0   33  t3
 2      .WORD    5    9   40
60      ENT          22      ; carica il valore 22
61      WRS           0      ; scrivi la stringa lunga 22 a partire da 0
62      WLN                  ; scrivi "fine linea"
63      ENT          10      ; carica il valore 10
64      WRS          22      ; scrivi la stringa lunga 10 a partire da 22
65      LDA      1    5      ; carica l'indirizzo di n
66      RDD           1      ; leggi un valore intero e mettilo in n
67      RLN                  ; leggi il "fine linea"
68      MST          40      ; prepara la chiamata di pfact
69      LDC      1    5      ; carica il contenuto di n
70      LDA      1    6      ; carica l'indirizzo di t1
71      CUP           6      ; chiama la procedura pfact
72      LDA      1    7      ; carica l'indirizzo di t2
73      MST          37      ; prepara la chiamata di ifact
74      LDC      1    5      ; carica il contenuto di n
75      CUP           5      ; chiama ifact
76      STO                  ; immagazzina
77      LDA      1    8      ; carica l'indirizzo di t3
78      MST          35      ; prepara la chiamata di rfact
79      LDC      1    5      ; carica il contenuto di n
80      CUP           5      ; chiama rfact
81      STO                  ; immagazzina
82      LDC      1    6      ; carica l'indirizzo di t1
83      WRD           1      ; scrivi il suo contenuto intero
84      LDC      1    7      ; carica l'indirizzo di t2
85      WRD           1      ; scrivi il suo contenuto intero
86      LDC      1    8      ; carica l'indirizzo di t
87      WRD           1      ; scrivi il suo contenuto intero
88      WLN                  ; scrivi "fine linea"
89      LDC      1    5      ; carica il contenuto di n
90      ENT           0      ; carica il valore 0
91      EQL I                ; n = 0 ?
92      FJP          63      ; in caso negativo salta a 63
93      WLN                  ; scrivi "fine linea"
94      HLT                  ; termina l'esecuzione

 0      .DATA   "calcolo del fattoriale"
23      .DATA   "enter n : "
        .END
```

Una certa attenzione merita il trattamento delle stringhe che in CPascal, come nel Pascal Standard, sono predefinite solamente come costanti utilizzabili in connessione con l'operazione di scrittura su file di testo. L'istruzione *WRS* del **pcode** produce una stringa sullo standard output copiandola dalla *Tabella delle Stringhe* **STab** a partire dall'indirizzo base memorizzato come parametro e per una lunghezza di caratteri che si trova come valore sulla cima della pila.

3.3 ESEMPI DI PROGRAMMI SEQUENZIALI

A conclusione di capitolo vengono presentati alcuni semplici programmi sequenziali di algoritmi molto comuni per il trattamento degli interi con il duplice scopo di fornire ulteriori esempi d'utilizzo della macchina a pila e di confrontarli, nei capitoli successivi, con la corrispondente implementazione in forma parallela ottenuta a partire dal relativo grafo di dipendenza dei dati.

Per ciascuno di essi viene dato, dapprima il listato completo in formato CPascal e, successivamente, fornita l'implementazione della funzione che realizza l'algoritmo indicato.

3.3.1 Somma dei Primi Numeri Interi

Nel semplice programma che segue la funzione che valuta la somma dei primi n numeri interi utilizza una variabile ausiliaria per il transito del parametro valore durante la chiamata.

```
;    0   program DoSum( input, output );
;    0   {$V+}
;    0     var   A : integer;
;    0           T : integer;
;    0
;    0     function Sum( N : integer ) : integer;
;    0     var I : integer;
;    0         S : integer;
;    0     begin
;    0       I := N;
;    3       S := 0;
;    6       while I > 0 do
;   10         begin
;   10           S := S + I;
;   15           I := I - 1
;   18         end;
;   21       Sum := S
;   22     end;
;   24
;   24   begin
;   25     writeln('Somma dei primi N numeri interi');
;   28     repeat
;   28       write('Enter N : ');
;   30       readln( A );
;   33       T := Sum( A );
;   38       writeln( T )
;   40     until A = 0;
;   45     writeln
;   45   end.
```

3.3.2 Divisione Intera

L'algoritmo di Euclide realizzato dal seguente programma per il calcolo della divisione intera utilizza due variabili locali, a parte la coppia di parametri valore, una per il controllo del ciclo, l'altra per memorizzare il risultato finale richiesto.

```
;    0   program IntegerDivision( input, output );
```

```
;   0   {$V+}
;   0     var   A : integer;
;   0           B : integer;
;   0           T : integer;
;   0
;   0     function IDiv( N, M : integer ) : integer;
;   0     var Q : integer;
;   0         R : integer;
;   0     begin
;   0       Q := 0;
;   3       R := N;
;   6       while R > M do
;  10         begin
;  10           Q := Q + 1;
;  15           R := R - M
;  17         end;
;  21       IDiv := Q
;  22     end;
;  24
;  24   begin
;  25     writeln('Calcolo della Divisione Intera');
;  28     repeat
;  28       write('Enter N, M : ');
;  30       read( A );
;  32       if A <> 0 then
;  36         begin
;  36           readln( B );
;  39           T := IDiv( A, B );
;  45           writeln( T )
;  47         end else
;  48         readln
;  49     until A = 0;
;  54     writeln
;  54   end.
```

3.3.3 Radice Quadrata Intera

Con una semplice variazione dell'algoritmo di Euclide si ottiene immediatamente il programma seguente per la determinazione della radice quadrata intera k di un numero intero n, definita con la coppia di diseguaglianze $k_2 \leq n < (k+1)_2$. L'algoritmo si basa sull'ovvia identitá $(k+1)_2 = k_2 + 2k + 1$ cosicché é sufficiente generare la sequenza dei primi numeri dispari ed arrestarla al k-esimo termine.

```
;   0   program SquareRoot( input, output );
;   0   {$V+}
;   0     var   A : integer;
;   0           T : integer;
;   0
;   0     function Sqrt( N : integer ) : integer;
;   0     var I : integer;
;   0         P : integer;
;   0         K : integer;
```

```
;   0     begin
;   0       I := 1;
;   3       P := 1;
;   6       K := 1;
;   9       while K <= N do
;  13         begin
;  13           I := I + 1;
;  18           P := P + 2;
;  23           K := K + P
;  25         end;
;  29       Sqrt := I - 1
;  32     end;
;  34
;  34   begin
;  35     writeln('Calcolo della radice Quadrata Intera');
;  38     repeat
;  38       write('Enter A : ');
;  40       readln( A );
;  43       T := Sqrt( A );
;  48       writeln( T )
;  50     until A = 0;
;  55     writeln
;  55   end.
```

3.3.4 Somma dei Primi Numeri Interi

```
 1    .WORD    0    0   30

33    .NAME    1    0    3   32   sum
34    .NAME    2    5    0    0   n
35    .NAME    2    6    0   34   i
36    .NAME    2    7    0   35   s
 3    .WORD    6    8   36
 0    LDA      2    6
 1    LDC      2    5
 2    STO
 3    LDA      2    7
 4    ENT           0
 5    STO
 6    LDC      2    6
 7    ENT           0
 8    GTR I
 9    FJP          21
10    LDA      2    7
11    LDC      2    7
12    LDC      2    6
13    ADI
14    STO
15    LDA      2    6
16    LDC      2    6
17    ENT           1
18    SBI
19    STO
20    UJP           6
21    LDA      2    0
22    LDC      2    7
23    STO
24    RET F

30    .NAME    0   25    2   29   $Main
31    .NAME    1    5    0    0   a
32    .NAME    1    6    0   31   t
 2    .WORD    5    7   33
25    ENT          31
26    WRS           0
27    WLN
28    ENT          10
29    WRS          31
30    LDA      1    5
31    RDD           1
32    RLN
33    LDA      1    6
34    MST          33
35    LDC      1    5
36    CUP           5
37    STO
38    LDC      1    6
39    WRD           1
40    WLN
41    LDC      1    5
42    ENT           0
43    EQL I
44    FJP          28
45    WLN
46    HLT

 0    .DATA   "somma dei primi n
               numeri interi"
32    .DATA   "enter n : "
      .END
```

3.3.5 Divisione Intera

```
 1    .WORD    0    0   30

34    .NAME    1    0    3   33   idiv
35    .NAME    2    5    0    0   n
36    .NAME    2    6    0   35   m
37    .NAME    2    7    0   36   q
38    .NAME    2    8    0   37   r
 3    .WORD    7    9   38
 0    LDA      2    7
 1    ENT           0
 2    STO
 3    LDA      2    8
 4    LDC      2    5
 5    STO
 6    LDC      2    8
 7    LDC      2    6
 8    GTR I
 9    FJP          21
10    LDA      2    7
11    LDC      2    7
12    ENT           1
13    ADI
14    STO
15    LDA      2    8
16    LDC      2    8
17    LDC      2    6
18    SBI
19    STO
20    UJP           6
21    LDA      2    0
22    LDC      2    7
23    STO
24    RET F

30    .NAME    0   25    2   29   $Main
31    .NAME    1    5    0    0   a
32    .NAME    1    6    0   31   b
33    .NAME    1    7    0   32   t
 2    .WORD    5    8   34
25    ENT          30
26    WRS           0
27    WLN
28    ENT          13
29    WRS          30
30    LDA      1    5
```

```
31    RDD          1
32    LDC     1    5
33    ENT          0
34    NEQ I
35    FJP         49
36    LDA     1    6
37    RDD          1
38    RLN
39    LDA     1    7
40    MST         34
41    LDC     1    5
42    LDC     1    6
43    CUP          6
44    STO
45    LDC     1    7
46    WRD          1
47    WLN
48    UJP         50
49    RLN
50    LDC     1    5
51    ENT          0
52    EQL I
53    FJP         28
54    WLN
55    HLT

 0    .DATA   "calcolo della
               divisione intera"
31    .DATA   "enter n, m : "
      .END
```

3.3.6 Radice Quadrata Intera

```
 1    .WORD   0    0    30

33    .NAME   1    0     3   32   sqrt
34    .NAME   2    5     0    0   n
35    .NAME   2    6     0   34   i
36    .NAME   2    7     0   35   p
37    .NAME   2    8     0   36   k
 3    .WORD   6    9    37
 0    LDA     2    6
 1    ENT          1
 2    STO
 3    LDA     2    7
 4    ENT          1
 5    STO
 6    LDA     2    8
 7    ENT          1
 8    STO
 9    LDC     2    8
10    LDC     2    5
11    LEQ I
12    FJP         29
13    LDA     2    6
14    LDC     2    6
15    ENT          1
16    ADI
17    STO
18    LDA     2    7
19    LDC     2    7
20    ENT          2
21    ADI
22    STO
23    LDA     2    8
24    LDC     2    8
25    LDC     2    7
26    ADI
27    STO
28    UJP          9
29    LDA     2    0
30    LDC     2    6
31    ENT          1
32    SBI
33    STO
34    RET F

30    .NAME   0   35     2   29   $Main
31    .NAME   1    5     0    0   a
32    .NAME   1    6     0   31   t
 2    .WORD   5    7    33
35    ENT         36
36    WRS          0
37    WLN
38    ENT         10
39    WRS         36
40    LDA     1    5
41    RDD          1
42    RLN
43    LDA     1    6
44    MST         33
45    LDC     1    5
46    CUP          5
47    STO
48    LDC     1    6
49    WRD          1
50    WLN
51    LDC     1    5
52    ENT          0
53    EQL I
54    FJP         38
55    WLN
56    HLT

 0    .DATA   "calcolo della radice
               quadrata intera"
37    .DATA   "enter a : "
      .END
```

Capitolo 4

Ambienti di Programmazione Concorrente

Affinché i processi sequenziali, generati dall'attivazione di un programma parallelo eseguito su una macchina di von Neumann multisequenziale, siano in grado di cooperare é necessario che abbiano la capacitá di *comunicare*, in modo che l'esecuzione dell'uno possa influenzare l'esecuzione dell'altro. Quando il valore di una variabile riferita da un processo é direttamente trasferito ad una variabile riferita da un altro processo si dice che la comunicazione innesca un coordinamento di tipo *data driven* basato sullo **scambio messaggi**.

Se, d'altra parte, vengono utilizzate **variabili condivise** riferibili da piú processi si dice che la comunicazione innesca un coordinamento di tipo *control driven*, che si basa sulla sincronizzazione esplicita evidenziata da un insieme di variabili globali speciali, dette **semafori**. Nel primo caso la comunicazione é diretta ed il ruolo della sincronizzazione é giocato dalla capacitá dei processi di comunicare eseguendo azioni specificiche che altri processi sono in grado di riconoscere. Nel secondo, invece, la comunicazione é indiretta perché si utilizzano variabili esterne ai processi, il cui accesso deve essere esplicitamente controllato.

4.1 SCAMBIO MESSAGGI

Nella comunicazione diretta fra processi, la sincronizzazione implicita impone alla coppia di azioni che realizzano l'invio e la ricezione del medesimo messaggio, di avvenire esattamente in quest'ordine per cui, se necessario, viene ritardata l'esecuzione di quel processo che tende a violare tale restrizione. In questo senso il modello computazionale di riferimento é quello *data driven* perché permette l'*assegnazione remota*[1] di una variabile, riferita da un processo, con il valore di un'espressione valutata nel contesto di un altro processo.

4.1.1 Definizione

L'impiego dello schema *data driven* consente di associare automaticamente un *meccanismo di sincronizzazione* ad un *meccanismo di trasporto dei dati* che viene realizzato attraverso un'istruzione generale di invio messaggi la cui forma completa é la seguente

$$\textbf{send}(\ < dest >,\ \ < expr-list >\ )$$

dove i valori delle espressioni in <expr-list>, nell'istante in cui viene eseguita l'istruzione **send**, sono il contenuto del messaggio. Il designatore di destinazione permette di controllare il punto in cui il messaggio deve pervenire, ossia le istruzioni che devono ricevere i valori trasmessi.

[1]trasferimento di valore

Analogamente, l'istruzione generale per ricevere un messaggio si presenta nella forma

$$\textbf{receive}(\ < source >,\ \ < var - list >)$$

dove <var-list> é la lista delle variabili cui vengono assegnati i valori contenuti nel messaggio nel momento cui questo viene ricevuto. Il designatore di provenienza permette di controllare il punto di partenza del messaggio e quindi le istruzioni coinvolte nell'invio del messaggio.

La ricezione del messaggio provoca, innanzitutto, l'*assegnazione* dei valori contenuti nel messaggio alla lista delle variabili che compaiono nell'istruzione **receive** e la successiva *distruzione* del messaggio. La coppia di designatori <*source*, *dest*> definisce un **canale di comunicazione** il cui nome é identificato univocamente da tale coppia. In generale, lo schema scelto per identificare il canale caratterizza il tipo di comunicazione.

4.1.2 Canali di Comunicazione

Nella situazione piú semplice si utilizzano come designatori di provenienza e di destinazione i nomi dei processi coinvolti nella comunicazione, analogamente a quanto avviene nel caso dei grafi di dipendenza dei dati, dove ciascun arco orientato definisce il canale di comunicazione diretta fra la coppia di nodi istruzioni coinvolte. Si dice, allora, che il canale é a **denominazione diretta**. La sua facilitá d'implementazione ed uso dipende dalla capacitá che lo schema ha di controllare gli istanti temporali in cui i due processi sono coinvolti nel trasferimento del messaggio.

La generalizzazione della denominazione diretta si ha quando si permette di stabilire una comunicazione fra due classi di processi utilizzando un unico canale di comunicazione attraverso il quale transitano messaggi dello stesso tipo, aventi le stesse caratteristiche.

Si dice che uno schema di comunicazione é realizzato per mezzo di **nomi globali**, *o,* **mailbox**, *se tali nomi possono comparire come designatori di destinazione e di provenienza, rispettivamente, in istruzioni* **send** *e* **receive**.

Cosí, i messaggi inviati ad una prefissata cassetta postale possono essere ricevuti da qualunque processo che esegue una **receive** che faccia riferimento a quella cassetta postale. Se non si dispone di una rete di comunicazione specializzata, l'implementazione delle cassette postali é, in generale, complessa e inefficiente. Ció dipende dal fatto che la presenza di un nuovo messaggio deve essere notificata ad ogni istruzione **receive** che puó essere eseguita su quella cassetta postale. Questo significa implementare esplicitamente un meccanismo di negoziazione fra i processi che potenzialmente sono in grado di accettare il messaggio.

Successivamente, appena il messaggio é stato ricevuto da qualche istruzione **receive**, bisogna notificare immediatamente a tutte le altre istruzioni, che possono ricevere messaggi dalla stessa cassetta postale, che il messaggio non é ulteriormente disponibile.

Una cassetta postale prende il nome di **porta** *quando questa appare come designatore di provenienza in istruzioni* **receive** *di un solo processo.*

Le porte sono facili da implementare poiché tutte le istruzioni **receive** che designano una porta possono apparire soltanto in un unico processo. Questa restrizione permette un'immediata realizzazione dello schema a server singolo, mentre non é in grado di fornire una soluzione immediata per lo schema generale comprendente piú server.

4.1.3 Modelli di Interazione

I due paradigmi fondamentali su cui é possibile modellare l'interazione fra processi concorrenti, noti in letteratura col nome di *pipeline* e *relazione client/server*, dipendono strettamente dal modo con cui viene rappresentato il canale di comunicazione.

Si definisce **pipeline** *un insieme ordinato di processi concorrenti in cui i dati di uscita di ciascun processo sono utilizzati come dati di ingresso del successivo.* In questo caso le informazioni fluiscono fra i processi analogamente al modo con cui un liquido fluisce entro una tubatura da cui segue immediatamente che i *pipeline* sono implementabili facilmente mediante canali di comunicazione a *denominazione diretta.*

Si dice che un insieme di processi é in relazione **client/server** *se alcuni processi, detti* server*, rendono un* servizio *a qualche processo* cliente *che ne ha fatto richiesta con un opportuno messaggio. Il* server *riceve la richiesta, esegue il servizio e, se necessario, restituisce un messaggio di servizio eseguito al cliente.* Seguendo questo schema, l'interazione *client/server* é realizzata in modo tale che i clienti inviino le richieste di servizio ad una prefissata *cassetta postale* dalla quale, successivamente, i server potranno prelevarli.

4.1.4 Eventi

Il meccanismo dello *scambio messaggi* é il paradigma piú generale di comunicazione fra processi in quanto permette di realizzare implicitamente la sincronizzazione fra i processi coinvolti nel trasferimento di dati. Questo meccanismo, fra l'altro, permette di far comunicare processi che sono in esecuzione su processori diversi, ciascuno utilizzante una propria memoria di lavoro, in quanto non viene fatta alcuna ipotesi particolare sulla comunicazione se non la presenza di canali di comunicazione.

Si supponga, allora, analogamente a quanto visto per i *data token*, di scorporare dal messaggio, considerato come valore in transito, la parte che ne determina il *sincronismo*, cioé l'informazione supplementare dell'effettiva disponibilitá del dato per il processo che lo vuole utilizzare. Ció é possibile purché i processi possano condividere aree comuni di dati, come nel caso di processi in esecuzione su una singola unitá di elaborazione, avente a disposizione un'unica memoria centrale.

In questo caso, se l'impiego delle cassette postali é ristretto al transito di messaggi *vuoti*, lo scambio dell'informazione puó realizzarsi attraverso aree globali di dati che si comportano come canali di comunicazione. Poiché i messaggi vuoti sono indistinguibili é sufficiente contarli e identificarli come *segnali temporali* che, consumati dal processo che li riceve, hanno lo scopo di notificargli l'avvenuto verificarsi di un evento. In questo senso, la nozione di evento é alla base di un metodo generale di sincronizzazione fra processi che permette di risolvere qualunque problema di comunicazione fra, purché i processi condividano un'unica memoria globale di lavoro.

Si dice che una cassetta postale definisce un **evento** *quando l'informazione scambiata ha la proprietá di caratterizzare l'istante temporale in cui il sincronismo si é verificato.* In questo caso, le primitive che realizzano lo scambio messaggi sono

$$\begin{aligned}\mathbf{cause}(< Event >) &\equiv \mathbf{send}(< Event >, \mathbf{true})\\ \mathbf{await}(< Event >) &\equiv \mathbf{receive}(< Event >, Succeded)\end{aligned}$$

dove *Succeded* é una variabile booleana che permette di riconoscere se l'evento si é verificato.

4.2 SEMAFORI

Nella discussione che segue viene trattato diffusamente il caso specifico in cui i designatori di provenienza e destinazione assumono il significato particolare di *eventi*, il verificarsi dei quali viene trasmesso come *informazione temporale* scambiata fra i processi coinvolti nella comunicazione indiretta. Il fatto che la notifica di disponibilitá di un nuovo *segnale temporale* o, *sincronismo*, non implichi trasferimento alcun di valore, non toglie che questo si possa considerare un caso particolare di comunicazione fra processi in cui il verificarsi di un certo *evento* dá luogo all'invio di un *messaggio vuoto.*

Pertanto, se si trasferiscono soltanto *messaggi vuoti* i quali trasportano, per definizione, solo l'informazione della loro disponibilitá, interpretabile come *segnale temporale*, le cassette postali assumono il signicato speciale di *semafori* nella stessa accezione data da Dijkstra, che le ha introdotte come meccanismo generale di sincronizzazione.

Infatti, poiché i messaggi sono vuoti e quindi indistinguibili, basta contarli cosicché la *cassetta postale* si riduce ad una singola variabile intera non negativa, che rappresenta il numero di segnali temporali inviati ma non ancora ricevuti. Una variabile di sincronizzazione di questo tipo prende il nome di **semaforo** ed é caratterizzata dalle due operazioni primitive

$$P(\text{ s }) \equiv \textbf{receive}(\text{ s, Done })$$
$$V(\text{ s }) \equiv \textbf{send}(\text{ s, } \textbf{true} \text{ })$$

dove *Done*, analogamente al caso degli eventi, é una variabile booleana che verificare se il segnale temporale, associato al semaforo, é stato trasmesso, mentre s é la variabile che individua il semaforo.

4.2.1 Nucleo di un Sistema di Calcolo

L'implementazione dei semafori vista nel caso dell'ambiente esecutivo del Concurrent Pascal ha messo chiaramente in luce il fatto che il nucleo gestisce i semafori in modo tale che ciascun processo, nel corso della sua esistenza, si puó trovare in uno *stato* di pronto, ed andare in esecuzione sul processore, oppure, in uno *stato* bloccato in attesa di completare un'operazione P.

Le funzionalitá del nucleo sono realizzate, generalmente, da due componenti specifiche, che prendono il nome di *scheduler* e *dispatcher*. Il primo gestisce la *lista dei processi pronti*, organizzata come una coda di descrittori di processo, e distribuisce i flussi di controllo di ciascuno di essi sull'unica unitá centrale di elaborazione, facendo eseguire ogni processo sequenziale per un periodo di tempo fissato. Il secondo consente la riattivazione e la sospensione effettiva dei processi mediante salvataggio e ripristino dei **contesti**, definiti come l'immagine in memoria di tutti i registri utilizzati dalla CPU.

Gestione dei Processi

Nel caso piú generale, i descrittori dei processi che sono bloccati su un certo semaforo vengono spostati dalla coda dei processi pronti a quella associata al semaforo in modo che non possano andare in esecuzione. Le operazioni P e V sono normalmente realizzate come trap a routine del nucleo. In corrispondenza di un'operazione **P**, se il semaforo ha un valore positivo, il suo valore viene decrementato; altrimenti il descrittore del processo in esecuzione viene spostato nella coda associata al semaforo. L'operazione **P**, invece, provoca lo spostamento di un descrittore dalla coda del semaforo, se non é vuota, a quella dei processi pronti; altrimenti, il valore del semaforo viene incrementato di un'unitá.

Il codice riportato nel seguito schematizza il funzionamento del nucleo relativamente alle operazioni di semaforo, le quali sono implementate in termini delle operazioni fornite dallo *scheduler* e dal *dispatcher*.

```
type Semaphore = ↑ Descriptor;

     Descriptor = record
                        Val : integer;
                        Qu : Queue
                  end;

procedure P( var s : Semaphore );
var  Curr : Process;
     Next : Process;
begin Lock( CPU );
   with s↑ do
     if ( Val = 0 ) then
        begin
```

```
                Suspend( Curr );
                SaveContext( Curr );
                Insert( Qu, Curr );
                Awake( Next );
                LoadContext( Curr );
            end else
            Val := Val - 1
    Unlock( CPU );
end;

procedure V( var s : Semaphore );
var   Pid : Process;
begin
    Lock( CPU );
    with s↑ do
        if not IsEmpty( Qu ) then
            begin
                Remove( Qu, Pid );
                Activate( Pid );
            end else
            Val := Val + 1
    Unlock( CPU );
end;
```

Vale la pena osservare che le operazioni di semaforo P e V sono implementate come istruzioni `WAIT` e `SGNL` del **pcode** e procedure *wait* e *signal* del Concurrent Pascal.

Azioni del Nucleo

La realizzazione delle operazioni di semaforo da parte del nucleo é ottenuta utilizzando un certo numero di procedure predefinite che rappresentano le azioni dello scheduler, del dispatcher e del gestore delle code associate ai semafori, secondo quanto precisato nel seguito. In particolare,

lo **scheduler** é responsabile della gestione della coda dei processi pronti attraverso le seguenti procedure

Suspend, per sospende il processo correntemente in esecuzione, rimuovendolo dalla coda dei pronti;

Activate, per introdurre il processo indicato nella coda dei pronti;

Awake, per risvegliare un processo pronto mandandolo in esecuzione; mentre

il **dispatcher** é responsabile della gestione dei contesti con le seguenti procedure

SaveContext, per salvare i registri della CPU, relativamente al processo correntemente in esecuzione, sotto forma di contesto per essere riattivato in un istante successivo;

LoadContext, per carica i registri della CPU secondo il contenuto del contesto indicato, ripristinando l'esecuzione di quel processo. Infine,

il **gestore delle code dei semafori**, con le procedure

IsEmpty, verifica se la coda in questione é vuota;

Remove, rimuove dalla coda in esame un processo secondo la politica di gestione della coda (ad esempio, FIFO);

Insert, inserisce nella coda in questione il processo indicato.

Si ricorda che nel caso del Concurrent Pascal esiste una sola coda circolare comprendente tutti i processi attivi nella quale si identificano i processi pronti per mezzo del campo **ready** del descrittore di processo e le code dei semafori per mezzo del campo **blocked**. Non esiste, invece, il concetto di disabilitazione delle interruzioni perché le operazioni di semaforo sono implementate come istruzioni primitive del **pcode**.

4.3 COMUNICAZIONE DIRETTA

Siamo ora in grado di implementare una rete di comunicazione fra processi che si basa su un meccanismo sottostante a sincronizzazione esplicita realizzato tramite semafori. Si consideri, dapprima, il problema in astratto in termini di eventi e di scambio di segnali temporali.

4.3.1 Implementazione: Strutture Dati

Nella comunicazione indiretta i messaggi risiedono temporaneamente su **memorie tampone** la cui funzione é quella di assorbire le variazioni di velocitá di avanzamento dei singoli processi. Si consideri a questo proposito, il seguento frammento di programma PASCAL che definisce un appropriata struttura di supporto per i dati in transito fra processi.

```
type    Index  = 0 .. N - 1;
        Size   = 0 .. N;
        Buffer = array[ Index ] of Message;
     Mailbox   = record
                    B  :  Buffer;
                    l  :  Index;
                    c  :  Index;
                    s  :  Size
                 end;
```

Se si indica con *Message* la forma del generico messaggio, la cassetta postale contiene, oltre alla lista dei messaggi pervenuti non ancora smistati, il loro numero **s**, il riferimento alla prima posizione disponibile **l** per il prossimo messaggio in arrivo, il riferimento al primo messaggio da recapitare **c** non appena qualche processo ne faccia richiesta.

Naturalmente, poiché la memoria tampone ha capacitá limitata, é necessario imporre alcune *restrizioni* sulla comunicazione; in particolare, *i processi che inviano messaggi non possono superare la capacitá della memoria tampone*, *i destinatari della comunicazione non possono ricevere i messaggi prima che questi siano pervenuti*, *nessuno dei processi coinvolti puó interferire nella comunicazione.*

In alcuni casi, come per il Concurrent Pascal, l'implementazione dello scambio messaggi non é diretta ma si basa su uno schema sottostante di comunicazione indiretta, reso possibile dal fatto che i processi possono fare affidamento sull'area comune di memoria, definibile all'interno del programma principale.

4.3.2 Implementazione: Procedure Primitive

Caratterizzando gli eventi *messaggio disponibile*, *spazio disponibile* e *non interferenza*, rispettivamente, con $\mathbf{s} > 0$, $\mathbf{s} < \mathrm{N}$, *Writing* e *Reading*, le operazioni *Send* e *Receive* possono essere realizzate mediante le due procedure PASCAL riportate nel seguito. Si osservi, a questo proposito, che gli eventi sopracitati controllano la comunicazione mediante la seguente **regola di sincronizzazione**:

Se il processo che tenta di inviare un messaggio trova la memoria tampone piena, *allora questo viene ritardato fino a quando un processo destinatario non estrae un messaggio; d'altra parte, se il destinatario tenta di prelevare un messaggio quando la memoria tampone é* vuota, *allora questo viene ritardato in attesa che qualche processo ne depositi uno.*

Le operazioni di *deposito* e *prelievo* messaggio ipotizzano una organizzazione **circolare** della memoria tampone cosicché, ogni volta che é stata scritta la locazione N si attende la disponibilitá della locazione *0*.

```
procedure Send( var Mbx : Mailbox; Msg : Message );
begin
    with Mbx do
        begin
            await( s < N );
            await( Writing );
            B[ c ] := Msg;
            c := ( c + 1 ) mod N;
            s := s + 1;
            cause( Writing );
            cause( s > 0 )
        end
end;
procedure Receive( var Mbx : Mailbox; var Msg : Message );
begin
    with Mbx do
        begin
            await( s > 0 );
            await( Reading );
            Msg := B[ l ];
            l := ( l + 1 ) mod N;
            s := s - 1;
            cause( Reading );
            cause( s < N )
        end
end;
```

Quindi, se al momento dell'esecuzione di una istruzione **send**, la memoria tampone risulta essere piena, tale istruzione ritarda il processo che l'ha invocata fino a quando nella memoria tampone non c'é spazio per il messaggio. In modo del tutto analogo se, al momento dell'esecuzione di una istruzione **receive**, non é disponibile alcun messaggio la cui provenienza é quella indicata, il processo che l'ha invocata viene ritardato.

4.4 IMPLEMENTAZIONE

I canali di comunicazione nel Concurrent Pascal sono realizzati per mezzo di un meccanismo di *inter-process communication* che si basa sull'impiego di un certo numero di semafori che controllano l'accesso di opportune aree globali, definite nel programma principale, e che simulano le *cassette postali.*

Piú specificamente si utilizza un insieme di canali per il trasferimento di *valori interi* e un altro per i *valori booleani.* Il programma che segue esemplifica la situazione.

```
;    0    program ipc(input,output);
```

```
;   0   {$V-}
;   0     const Max0 = 5;
;   0           Max1 = 6;
;   0
;   0     type Semaphore = integer;
;   0
;   0          Buffer = array[0..Max0] of integer;
;   0
;   0          Timing = array[0..Max0] of boolean;
;   0
;   0         Channel = record
;   0                     NotFull : Semaphore;
;   0                     NotEmpty : Semaphore;
;   0                     lf : integer;
;   0                     ld : integer;
;   0                     Bu : Buffer
;   0                   end;
;   0
;   0           Event = record
;   0                     NotFull : Semaphore;
;   0                     NotEmpty : Semaphore;
;   0                     lf : integer;
;   0                     ld : integer;
;   0                     Bu : Timing
;   0                   end;
;   0
;   0   {******************** Trattamento degli Interi **********************}
;   0
;   0     procedure InitChannel( var Mbx : Channel );
;   0     begin
;   0       Mbx.NotFull := Max1;
;   3       Mbx.NotEmpty := 0;
;   7       Mbx.lf := 0;
;  11       Mbx.ld := 0
;  14     end;
;  15
;  15     procedure Send( var Mbx : Channel; Msg : integer );
;  16     begin
;  16       wait( Mbx.NotFull );
;  18       Mbx.Bu[ Mbx.ld ] := Msg;
;  26       Mbx.ld := ( Mbx.ld + 1 ) mod Max1;
;  36       signal( Mbx.NotEmpty )
;  39     end;
;  39
;  39     procedure Receive( var Mbx : Channel; var Msg : integer );
;  40     begin
;  40       wait( Mbx.NotEmpty );
;  43       Msg:= Mbx.Bu[ Mbx.lf ];
;  52       Mbx.lf := ( Mbx.lf + 1 ) mod Max1;
;  62       signal( Mbx.NotFull )
```

```
;  64     end;
;  64
;  64   {******************** Trattamento degli Eventi **********************}
;  64
;  64     procedure StartEvent( var Mbx : Event );
;  65     begin
;  65       Mbx.NotFull := Max1;
;  68       Mbx.NotEmpty := 0;
;  72       Mbx.lf := 0;
;  76       Mbx.ld := 0
;  79     end;
;  80
;  80     procedure Cause( var Mbx : Event; T : boolean );
;  81     begin
;  81       wait( Mbx.NotFull );
;  83       Mbx.Bu[ Mbx.ld ] := T;
;  91       Mbx.ld := ( Mbx.ld + 1 ) mod Max1;
; 101       signal( Mbx.NotEmpty )
; 104     end;
; 104
; 104     procedure Await( var Mbx : Event; var T : boolean );
; 105     begin
; 105       wait( Mbx.NotEmpty );
; 108       T := Mbx.Bu[ Mbx.lf ];
; 117       Mbx.lf := ( Mbx.lf + 1 ) mod Max1;
; 127       signal( Mbx.NotFull )
; 129     end;
; 129
; 129   begin
; 130
; 130   end.
```

Ad esso segue il risultato della compilazione in **pcode** nella quale si vede chiaramente l'utilizzo delle istruzioni per le operazioni di semaforo, definito come variabile intera globale.

```
   1    .WORD    0    0   30

  48    .NAME    1    0    5   42  initchanne
  49    .NAME    2    5    3    0  mbx
   5    .WORD    6    6   49
   0    LDC      2    5
   1    ENT           6
   2    STO
   3    LDC      2    5
   4    LOA           1
   5    ENT           0
   6    STO
   7    LDC      2    5
   8    LOA           2
   9    ENT           0
  10    STO
```

```
11      LDC      2    5
12      LOA           3
13      ENT           0
14      STO
15      RET P

50      .NAME    1   16    6   48  send
51      .NAME    2    5    3    0  mbx
52      .NAME    2    6    0   51  msg
 6      .WORD    7    7   52
16      LDC      2    5
17      WAIT
18      LDC      2    5
19      LOA           4
20      LDC      2    5
21      LOA           3
22      STR
23      INX           1
24      LDC      2    6
25      STO
26      LDC      2    5
27      LOA           3
28      LDC      2    5
29      LOA           3
30      STR
31      ENT           1
32      ADI
33      ENT           6
34      MOD
35      STO
36      LDC      2    5
37      LOA           1
38      SGNL
39      RET P

53      .NAME    1   40    7   50  receive
54      .NAME    2    5    3    0  mbx
55      .NAME    2    6    0   54  msg
 7      .WORD    7    7   55
40      LDC      2    5
41      LOA           1
42      WAIT
43      LDC      2    6
44      LDC      2    5
45      LOA           4
46      LDC      2    5
47      LOA           2
48      STR
49      INX           1
50      STR
```

```
51    STO
52    LDC      2    5
53    LOA           2
54    LDC      2    5
55    LOA           2
56    STR
57    ENT           1
58    ADI
59    ENT           6
60    MOD
61    STO
62    LDC      2    5
63    SGNL
64    RET P

56    .NAME    1   65    8   53  startevent
57    .NAME    2    5    4    0  mbx
 8    .WORD    6    6   57
65    LDC      2    5
66    ENT           6
67    STO
68    LDC      2    5
69    LOA           1
70    ENT           0
71    STO
72    LDC      2    5
73    LOA           2
74    ENT           0
75    STO
76    LDC      2    5
77    LOA           3
78    ENT           0
79    STO
80    RET P

58    .NAME    1   81    9   56  cause
59    .NAME    2    5    4    0  mbx
60    .NAME    2    6    0   59  t
 9    .WORD    7    7   60
81    LDC      2    5
82    WAIT
83    LDC      2    5
84    LOA           4
85    LDC      2    5
86    LOA           3
87    STR
88    INX           2
89    LDC      2    6
90    STO
91    LDC      2    5
```

```
 92     LOA          3
 93     LDC      2   5
 94     LOA          3
 95     STR
 96     ENT          1
 97     ADI
 98     ENT          6
 99     MOD
100     STO
101     LDC      2   5
102     LOA          1
103     SGNL
104     RET P

 61     .NAME    1  105   10   58  await
 62     .NAME    2    5    4    0  mbx
 63     .NAME    2    6    0   62  t
 10     .WORD    7    7   63
105     LDC      2   5
106     LOA          1
107     WAIT
108     LDC      2   6
109     LDC      2   5
110     LOA          4
111     LDC      2   5
112     LOA          2
113     STR
114     INX          2
115     STR
116     STO
117     LDC      2   5
118     LOA          2
119     LDC      2   5
120     LOA          2
121     STR
122     ENT          1
123     ADI
124     ENT          6
125     MOD
126     STO
127     LDC      2   5
128     SGNL
129     RET P

  2     .WORD    5    5   61
 30     .NAME    0  130    2   29  $Main
 31     .NAME    1    5    0    0  max0
 32     .NAME    1    6    0   31  max1
 33     .NAME    1    1    0   32  semaphore
 34     .NAME    1    6    1   33  buffer
```

```
  1    .BLCK   0    5    1
 35    .NAME   1    6    2   34  timing
  2    .BLCK   0    5    1
 36    .NAME   1   10    3   35  channel
 37    .NAME   2    0    0    0  notfull
 38    .NAME   2    1    0   37  notempty
 39    .NAME   2    2    0   38  lf
 40    .NAME   2    3    0   39  ld
 41    .NAME   2    4    1   40  bu
  3    .WORD   0   10   41
 42    .NAME   1   10    4   36  event
 43    .NAME   2    0    0    0  notfull
 44    .NAME   2    1    0   43  notempty
 45    .NAME   2    2    0   44  lf
 46    .NAME   2    3    0   45  ld
 47    .NAME   2    4    2   46  bu
  4    .WORD   0   10   47
130    HLT
```

Come si puó notare, vengono considerati solamente canali di comunicazione per token che portano valori interi e valori booleani. Ciascun canale é realizzato con un buffer di immagazzinamento temporaneo, al quale viene aggiunto un meccanismo di sincronizzazione per la corretta gestione della coda di token.

4.5 Esempi di programmi concorrenti

I semplici programmi sequenziali presentati nel capitolo precedente possono essere analizzati in termini di grafi *dataflow* per metterne in luce il potenziale parallelismo. Per ciascuno di essi viene riportato viene riportato il corrispondente *grafo di dipendenza dei dati* e, successivamente, viene fornita l'implementazione con un insieme di **coroutine** che ne evidenziano i cicli computazionali parzialmente indipendenti.

Il programma parallelo per ciascuna funzione é realizzato ipotizzando una comunicazione asincrona fra le coroutine, ciascuna corrispondente ad un ciclo computazionale, secondo le strutture dati e le operazioni riportate di seguito.

4.5.1 Fattoriale

L'implementazione é caratterizzata da due cicli computazionali

kLoop, corrispondente alla variabile K della rappresentazione sequenziale, ed utilizzato per accumulare i valori delle moltiplicazioni intermedie per la computazione del fattoriale,

iLoop, corrispondente alla variabile I della rappresentazione sequenziale, per determinare gli operandi della moltiplicazione e controllare il numero di iterazioni del ciclo *kLoop*

Quella che segue é l'implementazione in Concurrent Pascal della funzione che calcola il fattoriale illustrata nella figura 4.1.

```
program Fact(input,output);
{$V+}

  var iC : Channel;
```

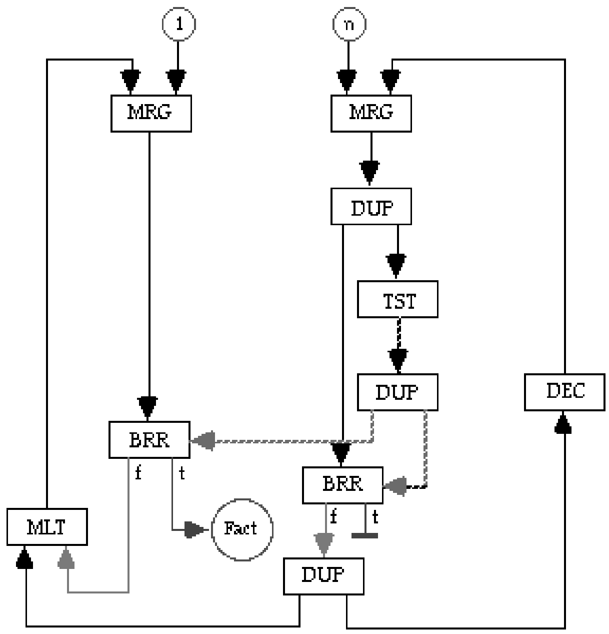

Figure 4.1: Grafo dataflow del fattoriale

```
    xC : Event;
     N : integer;
     T : integer;


procedure iLoop( n : integer );
var i : integer;
   xF : boolean;
begin
  i := n;
  repeat
    xF := i = 0;
    Cause( xC, xF );
    if not xF then
      begin
        Send( iC, i );
        i := i - 1
      end
  until xF
end;

procedure kLoop( var k : integer );
var i : integer;
   xF : boolean;
begin
  k := 1;
  repeat
```

```
      Await( xC, xF );
      if not xF then
        begin
          Receive( iC, i );
          k := k * i
        end
    until xF
  end;

begin
  InitChannel( iC );
  StartEvent( xC );
  writeln;
  writeln('Calcolo del fattoriale');
  writeln;
  write('Enter N : ');
  readln( N );
  cobegin
    iLoop( N );
    kLoop( T )
  coend;
  writeln('Fact(', N:1, ') = ', T:1 )
end.
```

Si noti che sia in questo programma come nei successivi non compaiono le definizioni di tipo, le dichiarazioni di variabili e procedure necessarie per implementare la comunicazione che, peró, devono essere aggiunte se si vuole utilizzare il compilatore **cpas** e l'esecutore **csc**.

4.5.2 Divisione Intera

L'implementazione é caratterizzata da due cicli computazionali

> **rLoop**, corrispondente alla variabile R della rappresentazione sequenziale, ed utilizzato per controllare il numero di iterazioni, nel senso che il ciclo viene mantenuto finché il valore di R non diventa il resto della divisione,
>
> **qLoop**, corrispondente alla variabile Q della rappresentazione sequenziale, per accumulare il valore che corrisponde al numero di cicli eseguiti da *rLoop* e che individua, al termine delle iterazioni, il quoziente della divisione.

Quella che segue é l'implementazione in Concurrent Pascal della funzione che calcola la divisione intera illustrata nella figura 4.2.

```
program IDiv(input,output);
{$V+}

  var xC : Event;
       A : integer;
       B : integer;
       T : integer;

  procedure rLoop( N : integer; M : integer );
```

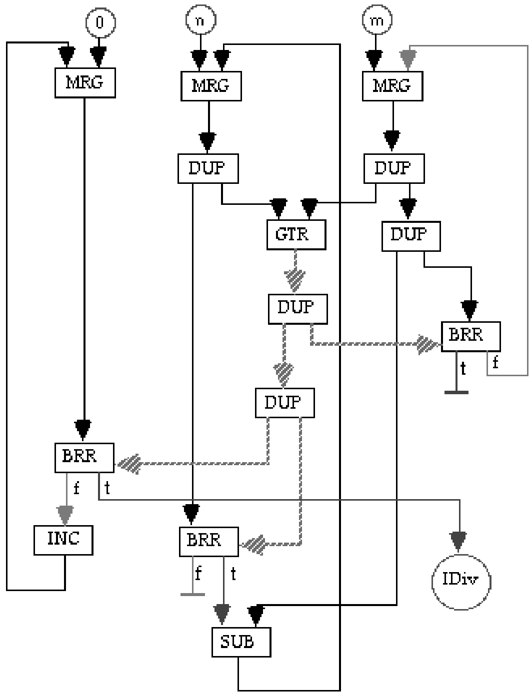

Figure 4.2: Grafo dataflow della divisione intera

```
  var R : integer;
     xF : boolean;
  begin
    R := N;
    repeat
      xF := R < M;
      Cause( xC, xF );
      if not xF then R := R - M
    until xF
  end;

  procedure qLoop( var Q : integer );
  var xF : boolean;
  begin
    Q := 0;
    repeat
      Await( xC, xF );
      if not xF then Q := Q + 1
    until xF
  end;

begin
  StartEvent( xC );
  writeln;
  writeln('Calcolo della Divisione Intera');
  writeln;
```

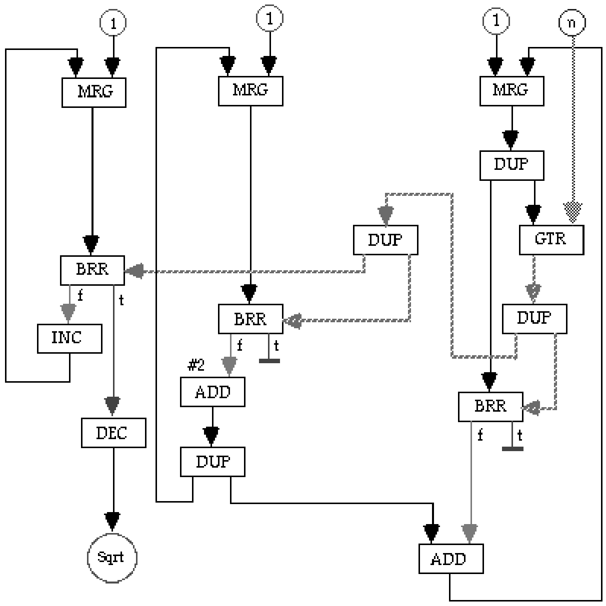

Figure 4.3: Grafo dataflow della radice quadrata intera

```
  write('Enter N, M : ');
  readln( A, B );
  cobegin
    rLoop( A, B );
    qLoop( T )
  coend;
  writeln( A:1,' div ', B:1, ' = ', T:1 )
end.
```

4.5.3 Radice Quadrata Intera

L'implementazione é caratterizzata da tre cicli computazionali

> **B¿kLoop**, corrispondente alla variabile *K* della rappresentazione sequenziale, ed utilizzato per controllare il numero di iterazioni confrontando il valore quadratico di K col valore della variabile di ingresso N,
>
> **pLoop**, corrispondente alla variabile *P* della rappresentazione sequenziale, che determina l'i-esimo numero dispari utilizzato per valutare l'i- esimo valore quadratico K,
>
> **iLoop**, corrispondente alla variabile *I* della rappresentazione sequenziale, per accumulare il valore che rappresenta il numero di cicli eseguiti da *kLoop* e che individua, al termine delle iterazioni, la radice quadrata richiesta.

Quella che segue é l'implementazione in Concurrent Pascal della funzione che calcola la radice quadrata intera illustrata nella figura 4.3.

```
program DoSqrt(input,output);
{$V+}

  var pC : Channel;
```

```
    xC1 : Event;
    xC2 : Event;
    kC : Channel;
     N : integer;
     T : integer;

procedure kLoop( N : integer );
var K : integer;
    P : integer;
   xF : boolean;
begin
  K := 1;
  repeat
    xF := K > N;
    Cause(xC1, xF );
    Cause(xC2, xF );
    if not xF then
      begin
        Receive( pC, P );
        K := K + P;
      end
    until xF
end;

procedure pLoop;
var P : integer;
    K : integer;
   xF : boolean;
begin
  P := 1;
  repeat
    Await( xC1, xF );
    if not xF then
      begin
        P := P + 2;
        Send( pC, P )
      end
  until xF
end;

procedure iLoop( var I : integer );
var  xF : boolean;
begin
  I := 1;
  repeat
    Await( xC2, xF );
    if not xF then I := I + 1
  until xF;
  I := I - 1
end;
```

```
begin
  InitChannel( pC );
  InitChannel( kC );
  StartEvent( xC1 );
  StartEvent( xC2 );
  writeln;
  writeln('Calcolo della radice quadrata intera');
  write('Enter N : ');
  readln( N );
  cobegin
    kLoop( N );
    pLoop;
    iLoop( T )
  coend;
  write('Sqrt[', N:1, '] = ', T:1 );
  writeln
end.
```