APPENDICE AI COSTRUTTI DI SINCRONIZZAZIONE

1. REGIONI CRITICHE

Il costrutto di **REGIONE CRITICA** è stato introdotto per garantire la mutua esclusione dall'accesso ad una struttura dati condivisa fra più processi (***Brinch Hansen e Hoare, 1972***)

SINTASSI (Pascal-like e NON C-like):
region V do
<SEZIONE CRITICA>
end;

dove
V è il nome della variabile che identifica la struttura dati (la risorsa) che viene usata all'interno della SEZIONE CRITICA

La variabile V deve essere *opportunamente* dichiarata come **var V: shared T;**

Il ***compilatore*** controlla che V sia acceduta **SOLO** all'interno della regione critica per garantire il rispetto del vincolo di mutua esclusione

Il costrutto di **REGIONE CRITICA** assicura che, durante la esecuzione della SEZIONE CRITICA, nessun altro processo può accedere alla variabile V
Quindi se:
P1 ===> region V do SC1 end;
P2 ===> region V do SC2 end;
questo è equivalente alla esecuzione sequenziale di SC1 seguita da SC2 (o *viceversa*)

(segue **1. REGIONI CRITICHE**)

ESEMPIO DI USO DI REGIONE CRITICA

Supponiamo di avere un insieme di processi che facciano uso della stessa risorsa R

Quindi R deve essere dichiarata come

var R: **shared** Tipo-Risorsa;

Allora i processi dovranno avere tutti questo schema per poter accedere in modo mutuamente esclusivo alla risorsa R ***(sintassi Pascal-like e NON C-like)***: Õ

PROCESSO Pi

```
begin

 region R do

    <uso di R>

 end;

end;
```

Ö Note su sintassi Pascal-like:

1) La parola-chiave `var` serve per dichiarare variabili nella forma `nome-var: Tipo-Var`
2) La parola-chiave `begin` è simile alla `{` e la `end` a `}` del C, mentre le `{}` servono per delimitare i commenti

(segue **1. REGIONI CRITICHE**)

REALIZZAZIONE

Il costrutto di **REGIONE CRITICA** viene trattato dal compilatore che lo sostituisce con la coppia di primitive **wait** e **signal** su un ***semaforo*** di mutua esclusione

Quindi,

per ogni dichiarazione

```
var V: shared T;
```

il compilatore genera un **SEMAFORO MUTEX** inizializzato ad 1

e per ogni istruzione

```
region V do S end;
```

viene sostituito il codice

```
wait(mutex);
   S;
signal(mutex);
```

(segue **1. REGIONI CRITICHE**)

OSSERVAZIONI

* **COSTRUTTO DI ALTO LIVELLO**

* Il campo di applicazione del costrutto di REGIONE CRITICA è limitato al solo caso di **MUTUA ESCLUSIONE**

* **Vantaggi** rispetto all'uso di un SEMAFORO
  ===> trattamento a livello di compilatore e quindi esente da errori

* Garanzia di correttezza nell'accesso alla variabile **SHARED**
  ===> controllo del compilatore che venga acceduta solo nella REGIONE CRITICA

* Possibilità di controllo, da parte del compilatore, di blocchi critici dovuti ad **innestamenti** di regioni critiche

Ad esempio:

```
var V1, V2: shared T;
```

processo P1

```
region V1 do
     ...
     region V2 do
     ...
     end;
end;
```

processo P2

```
region V2 do
     ...
     region V1 do
     ...
     end;
end;
```

Se P1 e P2 entrassero nelle regioni critiche più esterne si verificherebbe una situazione di blocco critico
⇨ compilatore può segnalare l'innestamento

(segue **COSTRUTTI DI SINCRONIZZAZIONE**)

2. REGIONI CRITICHE CONDIZIONALE

A differenza della REGIONE CRITICA, la **REGIONE CRITICA CONDIZIONALE** consente di risolvere qualunque tipo di problema di sincronizzazione (***Hoare, 1972***)

SINTASSI (Pascal-like e NON C-like):

```
region V when B do
<SEZIONE CRITICA>
end;
```

dove
B è una espressione logica ===> ***condizione***

La REGIONE CRITICA CONDIZIONALE è una REGIONE CRITICA e, quindi, assicura l'accesso esclusivo a V
INOLTRE
tale accesso è consentito **SE E SOLO SE** la condizione espressa da B è vera

REGOLE:

1) Un processo, una volta entrato nella regione critica, per prima cosa deve **valutare** la condizione **B**
2) **Se B è vera**, il processo esegue la sezione critica e quindi esce dalla REGIONE CRITICA
3) **Se B è falsa**, il processo viene ritardato e la REGIONE CRITICA liberata
   ===> potrà proseguire solo quando B sarà vera e nessun processo è nella REGIONE CRITICA

(segue **2. REGIONI CRITICHE CONDIZIONALI**)

1° ESEMPIO DI USO DI REGIONE CRITICA CONDIZIONALE

Riprendiamo l'esempio di due **processi** che facciano uso di una PILA

```
var PILA: shared STACK-TYPE;
```

Allora le due procedure per accedere alla PILA (INSERIMENTO e PRELIEVO) diventano ***(sintassi Pascal-like e NON C-like)*** ♦

INSERIMENTO (y)

```
begin
 region PILA
    when top < MAX do
        top := top + 1;
        stack[top] := y;
 end;
end;
```

PRELIEVO (x)

```
begin
     region PILA
        when top > 0 do
            x := stack[top];
            top := top - 1;
     end;
    end;
```

♦ **Note su sintassi Pascal-like:**
L'assegnamento viene indicato con :=, mentre = è il simbolo di uguaglianza e <> quello di disuguaglianza

(segue **2. REGIONI CRITICHE CONDIZIONALI**)

2° ESEMPIO DI USO DI REGIONE CRITICA CONDIZIONALE

Riprendiamo l'esempio di processi **PRODUTTORE** e di processi **CONSUMATORE** che comunicano tramite un ***buffer circolare (sintassi Pascal-like e NON C-like)***

```
var    BUFFER: shared record
             buff: array[0..N-1] of messaggio;
             punt1, punt2: 0..N-1; {valore iniziale 0}
             inseriti: 0..N; {valore iniziale 0}
    end;
```

processo PRODUTTORE

```
begin
     repeat
     <produzione messaggio>
     region BUFFER when inseriti < N do
          buff[punt1] := messaggio;
          punt1 := (punt1 + 1) mod N; inseriti := inseriti
          + 1;
     end;
     until false;
end;
```

processo CONSUMATORE

```
begin
     repeat
     region BUFFER when inseriti > 0 do
          messaggio := buff[punt2];
          punt2 := (punt2 + 1) mod N;
          inseriti := inseriti - 1;
     end;
     <consumo messaggio>
     until false;
end;
```

NOTE: 1) È stato necessario introdurre la variabile **inseriti**
2) Questa soluzione non consente l'accesso contemporaneo del produttore e consumatore ===> meno concorrente di quella coi semafori

(segue **2. REGIONI CRITICHE CONDIZIONALI**)

OSSERVAZIONI

* Valgono tutte le altre considerazioni fatte per le REGIONI CRITICHE semplici

MA

* Il campo di applicazione del costrutto di REGIONE CRITICA CONDIZIONALE ***non*** è limitato al solo caso di MUTUA ESCLUSIONE

IMPORTANTE:

La riattivazione dei processi sospesi **dipende** dal supporto che realizza il costrutto

La ***semantica*** del costrutto dice solo che:
un processo sospeso può riprendere l'esecuzione quando la condizione **B** diventa vera e non ci sono altri processi attivi entro la REGIONE CRITICA

Questo significa che:
non appena un processo lascia la REGIONE CRITICA, dato che la condizione per cui un processo è stato bloccato può essere modificata, il supporto ***DEVE*** provvedere a riattivare i processi sospesi e fargli **verificare nuovamente** la condizione

Nell'esempio precedente: la variabile inseriti viene sempre modificata
===> REALIZZAZIONE PIÙ COMPLESSA DI QUELLA DELLA REGIONE CRITICA SEMPLICE

(segue **2. REGIONI CRITICHE CONDIZIONALI**)

REALIZZAZIONE

Il costrutto di **REGIONE CRITICA CONDIZIONALE** viene sempre trattato dal compilatore, ma la sua implementazione risulta più complicata rispetto a quella della regione critica semplice

Quindi,
per ogni dichiarazione

var V: shared T;

il compilatore NON SOLO genera un **SEMAFORO MUTEX** inizializzato ad 1, ma anche un **semaforo SYNCH** ed un contatore **CONT** entrambi inizializzati a **0**

e per ogni istruzione
region V when B do S end;

viene sostituito il codice

```
wait(mutex);
        while (!B) {
        cont++;
        signal(mutex);
        wait(synch);
        wait(mutex); }
S;
while (cont != 0) {
        signal(synch);
        cont --; }
signal(mutex);
```

(segue **2. REGIONI CRITICHE CONDIZIONALI - REALIZZAZIONE**)

Tutti i processi per cui la condizione non è soddisfatta si sospendono sul **semaforo SYNCH** (liberando prima la REGIONE CRITICA)

Quando un processo ***lascia*** la REGIONE CRITICA, tutti i processi sospesi sulla coda del semaforo SYNCH vengono riattivati e trasferiti nella coda del semaforo **MUTEX**

OSSERVAZIONE:
Questa soluzione **NON** garantisce che i processi sospesi, una volta riattivati, acquisiscano **immediatamente** l'accesso alla REGIONE CRITICA

INFATTI:
vi potrebbero essere già altri processi presenti nella coda del semaforo MUTEX

INOLTRE:
I processi risvegliati DEVONO rivalutare la condizione **B** e non è detto che possano proseguire

(segue **REGIONE CRITICA CONDIZIONALE**)

MODIFICHE ALLA REGIONE CRITICA CONDIZIONALE

Il costrutto di **REGIONE CRITICA CONDIZIONALE** consente la sospensione dei processi ***SOLO*** all'inizio della sezione critica ===> questo può rendere complessa la risoluzione di alcuni problemi di sincronizzazione

Per questa ragione il costrutto è stato **modificato** in questi termini (***Brinch Hansen***):

```
region V do
begin
    S1;
    await(B);
    S2;
end;
```

dove
la verifica della condizione **B** può essere preceduta dalla esecuzione delle istruzioni indicate con **S1**

REGOLE:

1) Un processo, una volta entrato nella regione critica, esegue le istruzioni S1
2) Quindi viene verificata la condizione **B**:
   se B è vera, il processo esegue la sezione critica e quindi esce dalla REGIONE CRITICA
3) **Se B è falsa**, il processo viene bloccato e la REGIONE CRITICA liberata
   ===> potrà proseguire solo quando **B** sarà vera e nessun processo è nella REGIONE CRITICA

(segue **2. REGIONI CRITICHE CONDIZIONALI CON AWAIT**)

ESEMPIO DI USO DI AWAIT:
Riprendiamo l'esempio dei **LETTORI e SCRITTORI**

```
var V: shared record
    num_lettori,num_scrittori: integer;  {valore iniziale 0}
    occupato: boolean;                   {valore iniziale false}
    end;

procedure Inizio-lettura;
begin
    region V do begin
        await(num_scrittori = 0);
        num_lettori := num_lettori + 1;
    end;
end;

procedure Fine-lettura;
begin
    region V do begin
        num_lettori := num_lettori - 1;
    end;
end;

procedure Inizio-scrittura;
begin
    region V do begin
        num_scrittori := num_scrittori + 1;      ➲ vedi nota
        await((not occupato) and (num_lettori = 0));
        occupato := true;
    end;
end;

procedure Fine-scrittura;
begin
    region V do begin
        num_scrittori := num_scrittori - 1; occupato := false;
    end;
end;
```

NOTA:
In questo caso, diamo **priorità** ai processi SCRITTORI

(segue **COSTRUTTI DI SINCRONIZZAZIONE**)

3. PATH EXPRESSIONS

Utilizzando le **PATH EXPRESSION**, un problema di sincronizzazione viene risolto in questa maniera:

a) viene definita la struttura dati e le operazioni su cui operare in modo indipendente dalla possibilità di avere accesso concorrente ad essa;

b) viene definito lo schema di sincronizzazione di accesso alla risorsa, in modo indipendente dal punto a)

===> **ULTERIORE MODULARITA'**

Nel caso del monitor entrambi questi aspetti vengono risolti con un unico costrutto

***SINTASSI*:**

```
PATH <espressione> end;
```

Alcuni **OPERATORI** che sono usati in <espressione> sono:

, indica possibile concorrenza

; indica un vincolo di sequenzialità

n:( ) indica che sono possibili al più **n** attivazioni concorrenti di ciò che è in parentesi

[ ] indica l'assenza di ogni restrizione, cioè ci può essere un numero qualunque di attivazioni

1° ESEMPIO:

Il problema LETTORI/SCRITTORI viene risolto associando alla risorsa accessibile tramite le operazioni Read e Write questa PATH EXPRESSION

```
path 1 : ([Read], Write) end;
```

(segue **4. PATH EXPRESSION**)

2° ESEMPIO:

Riprendiamo il problema PRODUTTORI/CONSUMATORI e supponiamo di definire un TIPO DI DATO ASTRATTO che definisce le strutture dati del buffer circolare:

```
type BUFFER_CIRCOLARE = abstract data type;
     var buff: array[0..N-1] of messaggio;
         punt1, punt2: 0..N-1;
procedure DEPOSITA(x: messaggio);
begin
     buff[punt1] := x;  punt1 := (punt1 + 1) mod N;
end;
procedure PRELEVA(var x: messaggio);
begin
     x := buff[punt2];  punt2 := (punt2 + 1) mod N;
end;
begin punt1 := 0; punt2 := 0;  end;
```

Questo tipo di dato astratto può essere protetto dagli aspetti di concorrenza mediante la PATH EXPRESSION:

```
PATH N : (1 : (DEPOSITA); 1: (PRELEVA)) end;
```

Vediamo cosa significa:

`1: (DEPOSITA)`	===>	una sola attivazione della procedura `DEPOSITA`
	===>	mutua esclusione
`1: (PRELEVA)`	===>	una sola attivazione della procedura `PRELEVA`
	===>	mutua esclusione
`N: (...)`	===>	al massimo N attivazioni di `DEPOSITA` e `PRELEVA`, partendo con una `DEPOSITA`

OSSERVAZIONE:
Questa soluzione non limita il possibile parallelismo fra un PRODUTTORE ed un CONSUMATORE