Generalizzazione di *Passo* a n processi: *algoritmo del fornaio*

✓ ***Prima di entrare*** nella sezione critica un processo ***riceve un numero***.
✓ Il processo che detiene ***il numero piu' piccolo entra***.
✓ Se i processi ***Pi e Pj ricevono lo stesso numero***, se *i* < *j*, allora *Pi* entra per primo, altrimenti *Pj*.
✓ Lo schema di numerazione sempre genera ***numeri in ordine non decrescente***; per es. 1,2,3,3,3,3,4,5...

Variabili condivise

var *choosing* : **array** [0..*n* – 1] **of** *boolean* ;
number : **array** [0..*n* – 1] **of** *integer* ;

… tutte inizializzate a ***false*** e a ***0*** rispettivamente

IDEA DI BASE

- Quando si sceglie viene assegnato il numero piu' alto
- Si attende se qualcuno sta scegliendo il numero
- Si attende se qualcuno ha scelto il numero ed e' piu' piccolo del proprio
- All'uscita dalla sezione critica si azzera il proprio numero


```
Repeat
      choosing[i] := true ;
      number[i] := max (number[0], …, number [n – 1])+1;
      choosing[i] := false;
      for j := 0 to n – 1
            do begin
                  while choosing[j] do no-op;
                  while number[j] ≠ 0 and
                        (number[j],j) < (number[i], i) do no-op;
              end;

            sezione critica

      number[i] := 0;
            sezione "normale"
until false;
```


Soluzioni hardware : Test-and-Set

Bisogna introdurre nuove ***istruzioni atomiche***

```
function  Test-and-Set (var target: boolean): boolean;
      begin
            Test-and-Set := target ;
            target := true ;
      end;
```

Testa e modifica il contenuto di una cella di memoria in maniera atomica, ***non interrompibile***

… come si usa la Test-and-Set

Variabili condivise :
var *lock* : *boolean* ***(inizialmente false)***

Processo Pi :

```
repeat
      while Test-and-Set (lock) do no-op ;
            sezione critica
      lock := false ;
            sezione "normale"
until false ;
```

Non e' garantita ***l'attesa limitata*** (3) in quanto non c'e' controllo sul numero di processo, ma il primo che esegue il test entra e tiene fuori gli altri

Repeat ***Soluzione ottimale***

```
      waiting [i] := true ;
      key := true ;
      while (waiting[i] and key) do key := Test-and-Set(lock) ;
      waiting [i] := false ;

            sezione critica

      j := (i+1) mod n ;
      while (j ≠ i) and (not waiting[j]) do j := (j+1) mod n ;
      if j=i then lock:=false else waiting[j] := false ;

            sezione "normale"
until false ;
```

Il processo Pi sta nel ciclo di ingresso finche' un processo appena uscito non:

- setta lock a falso (non c'e' nessuno che aspetta, si agisce come prima)
- setta waiting[i] a falso (*i* viene scelto come prossimo tra quelli in attesa)

Altre soluzioni hardware

L'istruzione ***swap***, che ***scambia il contenuto di due variabili in maniera atomica***, se disponibile puo' essere utilizzata per controllare sezioni critiche

Busy Form of Waiting (BFW)

Quando un processo non puo' entrare nella sua sezione critica rimane ad ***attendere il suo turno*** "facendo qualcosa" (***forma occupata di attesa***)

⇩

Spreco di tempo di CPU

Bisogna trovare una soluzione che permetta al ***processo in attesa*** di andare in uno ***stato di waiting*** in modo da liberare la risorsa CPU

D'altro canto l'introduzione di tale ***overhead da context-switching*** conviene solo se le attese sono lunghe

Un nuovo tipo di dato: semaforo

Semaforo S – variabile intera alla quale si puo' accedere solo mediante ***due indivisibili operazioni*** la cui idea di base e':

wait (*S*): **while** *S* ≤ 0 **do** *no-op* ;
S := *S* – 1;

signal (*S*): *S* := *S* + 1;

Inizializzazione di S ?!

Evitare BFW usando semafori

Strumento di sincronizzazione che permette ai processi di ***evitare BFW***

Definiamo un semaforo come ***un record***

```
type semaphore = record
                   value : integer
                   L : list of process ;
                 end;
```

… e assumiamo due semplici operazioni:
block sospende il processo P che la invoca
wakeup(P) riattiva l'esecuzione di un processo P

Wait e signal che evitano BFW

```
wait(S ):
        S.value := S.value – 1;
        if S.value < 0
        then  begin
                    aggiungi questo processo a S.L;
                    block;
              end;

signal(S ):
        S.value := S.value + 1;
        if S.value  0
        then  begin
                    rimuovi un processo P da S.L;
                    wakeup(P);
              end;
```


Semaforo per sezione critica

Variabili condivise :
var *mutex* : *semaphore*
(inizializzata a 1)

Processo Pi :
repeat
wait (*mutex*);
sezione critica
signal (*mutex*);
sezione "normale"
until *false* ;

Alcune considerazioni

• Sulla ***coda dei processi ad un semaforo*** puo' essere applicata qualsiasi strategia (FIFO, a priorita', etc.)

• Il valore assoluto del numero intero ***S.value*** ⇨ corrisponde al numero di processi che attendono in coda, ma solo se la sezione critica e' occupata

ATTENZIONE **:** ***Wait*** e ***Signal*** sono le due ***nuove sezioni critiche***, quindi c'e' BFW su di esse, ma si e' spostata la sezione critica su un numero esiguo di istruzioni !!!

E quindi di nuovo per la *mutua esclusione* tra n processi…

Variabili condivise:

var *mutex* : *semaphore*

Processo *Pi* :

```
repeat
        wait (mutex) ;
                sezione critica
        signal (mutex);
                sezione "normale"
until false ;
```

Quindi perche' e' importante che inizialmente sia ***mutex = 1*** ?!

… ma anche per semplice *sincronizzazione*

Eseguire *B* in *Pj* solo dopo aver eseguito *A* in *Pi*

Pi	*Pj*
…	…
A	*wait* (*flag*)
signal (*flag*)	*B*
…	…

flag **inizializzato a *0* (IMPORTANTE!)**

Riassumendo quindi…

… il tipo di dato ***semaforo*** puo' essere utilizzato sia per garantire la ***mutua esclusione*** (inizializzazione a 1) sia per garantire la semplice ***sincronizzazione*** (inizializzazione a 0)

E se volessimo far accedere k processi alla volta in una sezione critica ?!

Due problemi : Deadlock…

Due o piu' processi stanno aspettando per un evento che puo' essere causato solo da uno dei processi in attesa

S e *Q* due semafori inizializzati a 1

Entrambi si possono bloccare qui ⇨

P0	*P1*
wait (*S*);	*wait* (*Q*);
wait (*Q*);	*wait* (*S*);
…	…
signal (*S*);	*signal* (*Q*);
signal (*Q*);	*signal* (*S*);

… e Starvation

Blocco indefinito :
Un processo non sara' mai rimosso dalla coda del semaforo sul quale si e' sospeso

Per esempio se una ***strategia LIFO*** e' applicata alla coda di un semaforo