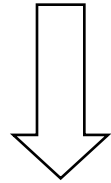


SINCRONIZZAZIONE TRA PROCESSI

Qualche concetto di base

Processi cooperanti *condividono uno spazio di indirizzi*

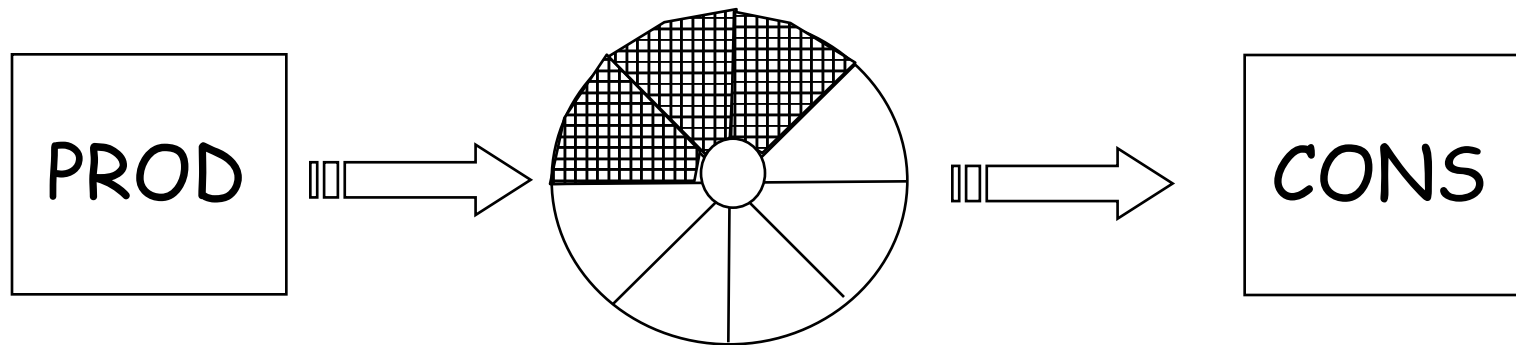


Bisogna *in qualche modo* garantire
consistenza di dati...

... e questo richiede meccanismi che assicurino
l'esecuzione in ordine appropriato dei
processi cooperanti (*IPC*)

Di nuovo su PROD-CONS

La prima soluzione vista del problema produttore-consumatore permetteva al più *$n-1$ elementi nel buffer* alla volta



Proviamo allora a trovare una soluzione che ne permetta *n* :
variabile ***counter*** che conta il numero di elementi nel buffer

Una nuova soluzione

Variabili condivise:

```
type item = ... ;  
var buffer array [0..n-1] of item;  
in, out : 0..n-1 ;  
counter : 0..n ;  
in, out, counter := 0;
```

Processo produttore:

```
repeat  
    ...  
    produci un elemento in nextp  
    ...  
while counter = n do no-op;  
buffer[in] := nextp;  
in := in + 1 mod n;  
counter := counter + 1;  
until false;
```

Processo consumatore:

repeat

while *counter* = 0 **do** *no-op*;

nextc := *buffer* [*out*];

out := *out* + 1 **mod** *n*;

counter := *counter* - 1;

...

consuma l'elemento presente in *nextc*

...

until *false* ;

La *condivisione* vera e propria *si sposta sulla variabile counter* :

i due processi possono aggiornare la variabile *counter* “allo stesso tempo” senza garanzia di consistenza...

Esempio di sequenza potenzialmente “dannosa” di istruzioni

Condizione per la consistenza dei dati

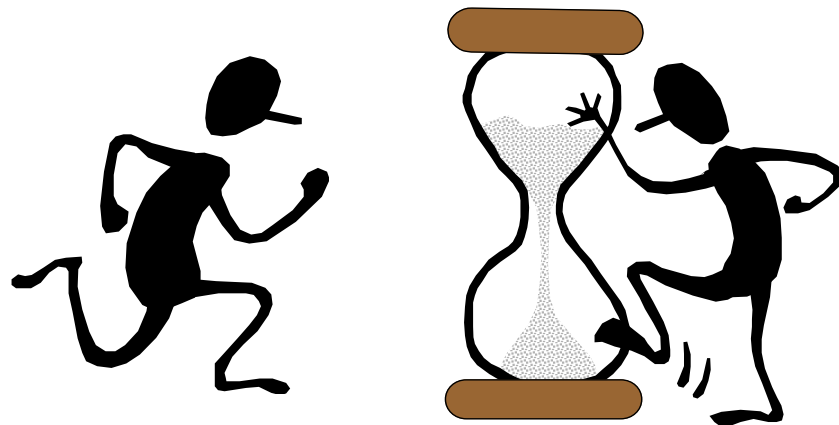
Le istruzioni

counter := counter + 1;
counter := counter - 1;

devono essere eseguite *in maniera atomica*

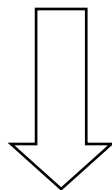
Dove sono piazzate le istruzioni critiche?

Qualunque soluzione per la sincronizzazione tra processi deve *prescindere dalla velocità relativa di esecuzione dei processi*



Cos' e' una *Sezione Critica*

Ogni zona di codice in cui *si accede ad una variabile condivisa* viene detta *sezione critica*



Bisogna *far qualcosa* prima e dopo la sezione critica per *garantire il corretto funzionamento*

repeat

entry section

sezione critica

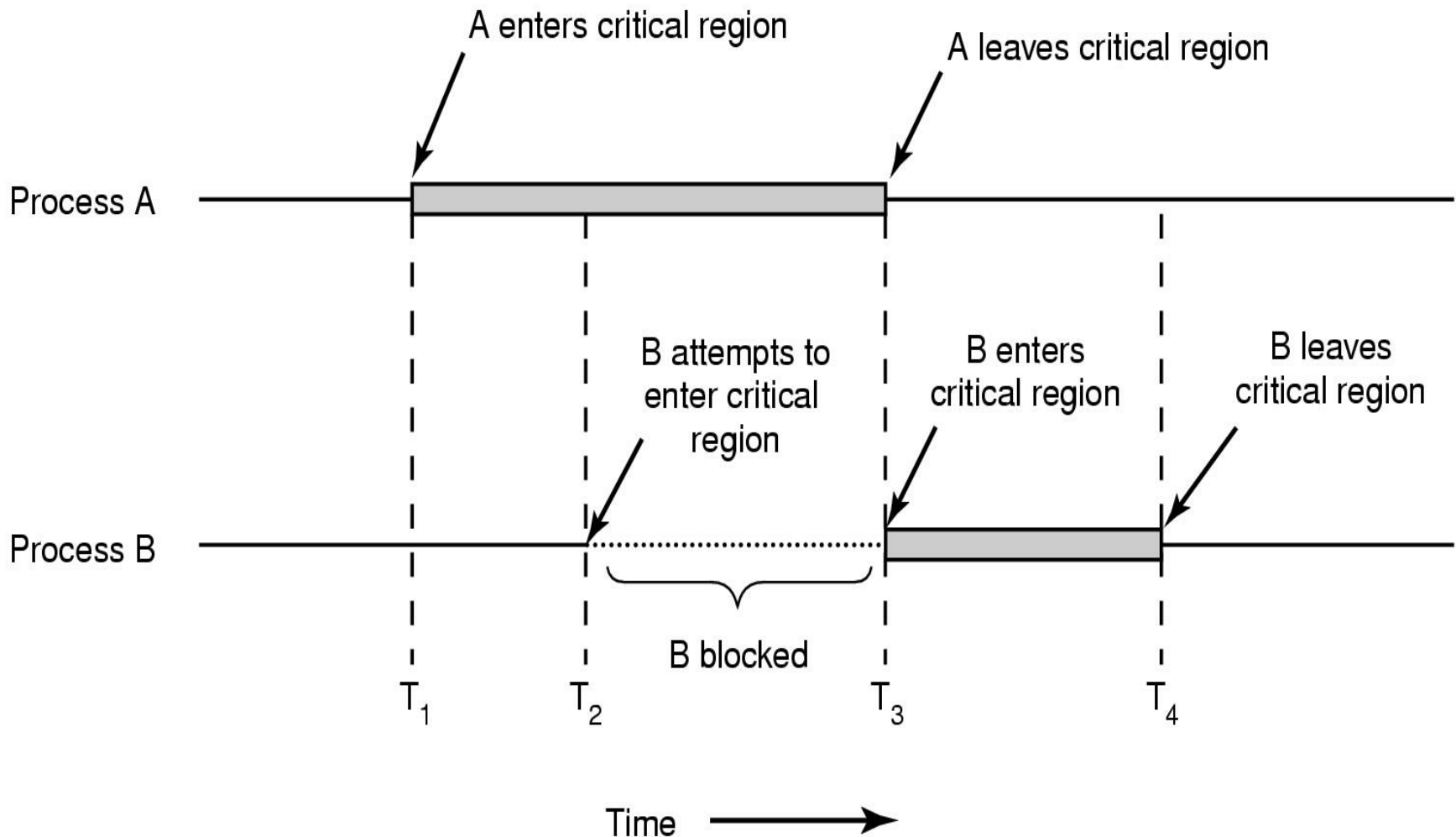
exit section

sezione "normale"

Tutto cio' che non e' la sezione critica

until *false* ;

A e B accedono alle loro rispettive *Sezioni Critiche*



Condizioni per il *corretto funzionamento*

1. Mutua esclusione:

Se un processo sta eseguendo la sua sezione critica, allora *nessun altro processo* puo' stare ad *eseguire la sua sezione critica*

Condizioni per il *corretto funzionamento*

2. Progresso:

Se nessun processo sta eseguendo la sua sezione critica e c'è qualche processo che vuole entrare nella sua, allora la *decisione su quale processo far entrare* nella sezione critica deve essere: (i) *presa in un tempo finito*, e (ii) *solo dai processi che non stanno nella loro sezione "normale"*

Condizioni per il *corretto funzionamento*

3. Attesa limitata: Dopo che un processo ha fatto richiesta di ingresso nella sua sezione critica e prima che la sua richiesta venga soddisfatta deve essere *limitato il numero di volte* che ad *altri processi* e' consentito *entrare nelle loro sezioni critiche*

Caso di due processi : *Alternanza*

Variabili condivise :

var turn : (0..1);

inizialmente *turn = 0*

turn = i \Rightarrow *Pi* puo' entrare nella sua sezione critica

Processo *Pi* :

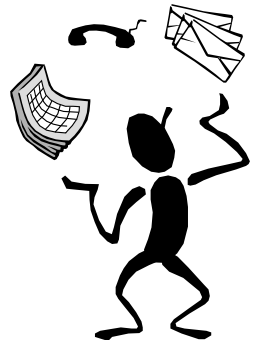
repeat

[while turn \neq i do no-op ;]
sezione critica

[turn := j ;]

sezione "normale"

until false ;



Valutazione di *Alternanza*

Sono garantite :

- *mutua esclusione* (1)
- *attesa limitata* (3)

Non e' garantito progresso (2)

Un processo che sta nella sua sezione “normale” puo' condizionare un altro all' ingresso nella sezione critica

Esempio : se P1 vuole entrare due volte di seguito non può, condizionato dal fatto che P2 sta ancora nella sua sezione “normale”

Piu' conoscenza reciproca : *Segnalazioni*

Variabili condivise :

var flag : array [0..1] of boolean ;

inizialmente *flag [0] = flag [1] = false*

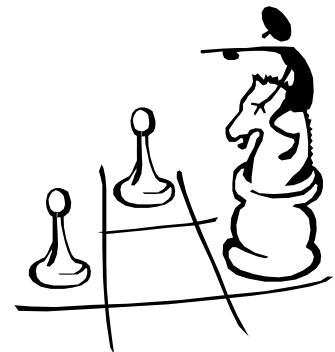
flag [i] = true \Rightarrow *Pi* e' pronto ad entrare nella
sua sezione critica

Processo *Pi* :

repeat

flag[i] := true ;
while flag[j] do no-op ;
sezione critica
flag[i] := false ;
sezione "normale"

until false ;



Valutazione di *Segnalazioni*

Sono garantite :

- *mutua esclusione (1)*
- *attesa limitata (3)*

Comunque non e' garantito progresso (2)!!!

La decisione su chi deve entrare nella propria sezione critica potrebbe non essere presa mai

- *Esempio di sequenza “dannosa”*
- *Cosa succede se si invertono le due istruzioni della entry section?!*

Soluzione ottimale : *Passo*

repeat

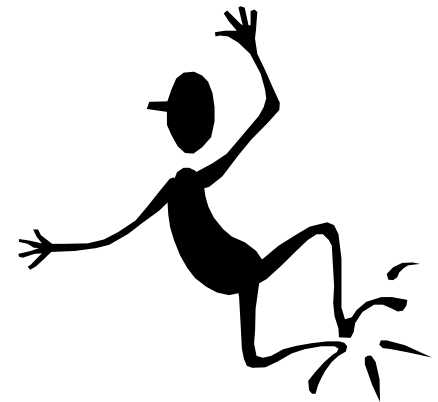
```
flag[i] := true ;  
turn := j ;  
while (flag[j] and turn = j ) do no-op ;
```

sezione critica

```
flag[i] := false ;
```

sezione “normale”

until false ;



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 P_i e P_j ricevono lo stesso numero, se $i < j$, allora P_i entra per primo, altrimenti P_j .
- ✓ 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 P_i :

repeat

[while Test-and-Set (lock) do no-op ;

sezione critica

[lock := false ;

sezione "normale"

until false ;

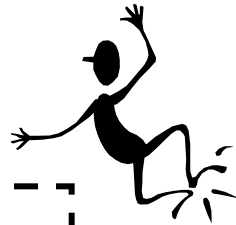
Non e' garantita l'*attesa limitata* (3) in quanto non c' e' controllo sul numero di processo, ma il primo che esegue il test entra e tiene fuori gli altri

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”

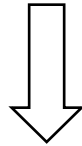
until false ;

Il processo P_i sta nel ciclo di ingresso finché un processo appena uscito non:

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

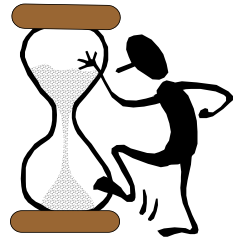
Busy Form of Waiting (BFW)

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



Evitare BFW: semaforo con coda

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;
```

Evitare BFW: semaforo con coda

Su questo semaforo si possono effettuare, di nuovo, solo operazioni di **wait** e **signal**, ma che sono diverse dalle precedenti in quanto evitano la BFW...

... e che utilizzano 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

*(campo “value”
inizializzato a 1)*

Processo P_i :

repeat

wait (mutex);

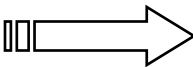
sezione critica

signal (mutex);

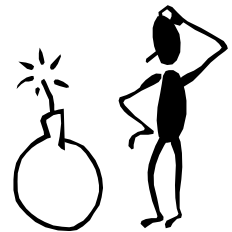
sezione “normale”

until false ;

Alcune considerazioni

- Sulla *coda dei processi ad un semaforo* puo' essere applicata qualsiasi strategia (FIFO, a prioritá', 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 :
Atomicita' delle operazioni Wait e Signal



Per la *mutua esclusione* tra n processi

Variabili condivise:

var *mutex* : *semaphore*

Processo P_i :

repeat

wait (*mutex*) ;

sezione critica

signal (*mutex*);

sezione “normale”

until *false* ;

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

Semaforo per la *precedenza*

Eseguire B in P_j solo dopo aver eseguito A in P_i

<u>P_i</u>	<u>P_j</u>
...	...
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 *precedenza* (inizializzazione a 0)

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

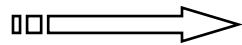


Un possibile problema : Deadlock

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

wait (S);

wait (Q);

...

signal (S);

signal (Q);

P1

wait (Q);

wait (S);

...

signal (Q);

signal (S);

Assenza di starvation

Si avrebbe starvation se un processo non fosse mai rimosso dalla coda del semaforo sul quale è waiting

Per esempio se una *strategia LIFO* governasse la coda di un semaforo

Classici problemi di sincronizzazione

1. Bounded-Buffer Problem
(PROD-CONS)
2. Readers and Writers Problem
3. Dining-Philosophers Problem

1. Bounded-Buffer Problem

Variabili condivise :

type *item* = ...

var *buffer* = ...

full, empty, mutex: semaphore ;

nextp, nextc: item ;

full := 0; empty := n; mutex := 1;

*Semaforo per numero
di locazioni piene*



```
graph TD
    S1[Semaforo per numero di locazioni piene] -.-> full
    S2[Semaforo per numero di locazioni vuote] -.-> empty
    S3[Semaforo di mutua esclusione] -.-> mutex
```

*Semaforo per numero
di locazioni vuote*

*Semaforo di mutua
esclusione*

1. Bounded-Buffer Problem

Processo produttore :

repeat

...

produci un elemento in *nextp*

...

wait(empty);

wait(mutex);

...

inserisci *nextp* in *buffer*

...

signal(mutex);

signal(full);

until false;

1. Bounded-Buffer Problem

Processo consumatore:

repeat

wait(full)

wait(mutex);

...

rimuovi l'elemento di *buffer* in *nextc*

...

signal(mutex);

signal(empty);

...

consumo l'elemento in *nextc*

...

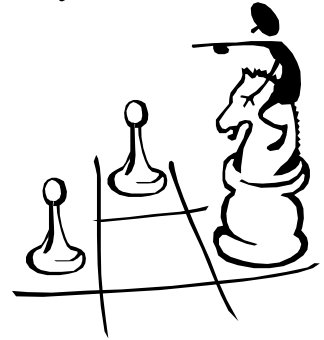
until *false;*

*Si noti la simmetria del
codice con il produttore!*

2. Readers and Writers Problem

PROBLEMA

*Un'area di memoria deve essere condivisa tra vari processi che possono scrivere su (**writer**) o leggere da (**reader**) tale area*



*Piu' processi **reader** possono leggere contemporaneamente* ma, per problemi di consistenza, un **writer** *non puo' accedere* all'area *in contemporaneita'* con nessun altro processo (reader o writer)

2. Readers and Writers Problem

SOLUZIONI

Prima versione - precedenza ai reader :
nessun reader deve attendere a meno che
un writer non sia già nella sezione critica

Seconda versione - precedenza ai writer :
se un writer è in attesa inizierà al più
presto, e cioè quando i reader che stavano
già dentro avranno terminato; gli altri
reader attenderanno

Possibile starvation in entrambi i casi !!!

2. Readers and Writers Problem (prima versione)

Variabili condivise :

*var mutex, wrt : semaphore (=1);
 readcount : integer (=0);*

Processo writer :

wait(wrt);

...

esegue la scrittura

...

signal(wrt);

2. Readers and Writers Problem

Processo reader :

wait(mutex);

readcount := readcount + 1;

if readcount = 1 then wait(wrt);

signal(mutex);

...

esegue la lettura

...

wait(mutex);

readcount := readcount - 1;

if readcount = 0 then signal(wrt);

signal(mutex);

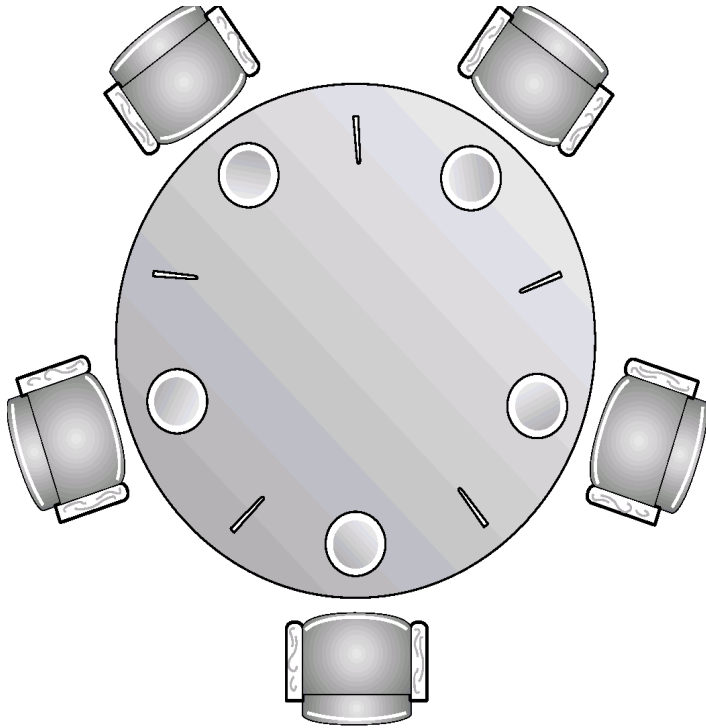
*Mentre un
writer e'
dentro il
primo lettore
si accoda su
wrt mentre
tutti i lettori
seguenti
all'ingresso
del primo
entrano
direttamente*

2. Readers and Writers Problem (seconda versione)

**CHE NE DITE DI PROVARCI
DA SOLI, CHE VI FAREBBE
TANTO BENE !?!?!?**



3. Dining-Philosophers Problem



Ogni filosofo esegue questa sequenza:

- Si procura le due bacchette
- Mangia
- Ripone le bacchette
- Pensa

Variabili condivise :

var chopstick : array [0..4] of semaphore ;
(=1 inizialmente)

3. Dining-Philosophers Problem

Philosopher i:

repeat

wait (chopstick[i])

wait (chopstick[i+1 mod 5])

...

mangia

...

signal (chopstick[i]);

signal (chopstick[i+1 mod 5]);

...

pensa

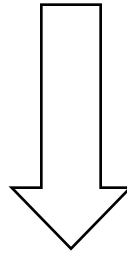
...

until false ;

Soluzione inaccettabile : possibile deadlock! Ci ritorniamo...

Strutture ad alto livello per la gestione di sezioni critiche

L'uso dei *semafori* si presta comunque ad *errori* introdotti dal programmatore



Servono *costrutti a piu' alto livello* per gestire in maniera piu' sicura i problemi di sincronizzazione

- 1. Regioni critiche*
- 2. Monitors*

1. Regioni critiche: definizione

Una *variabile condivisa* v di tipo T viene dichiarata come:

var v : shared T

... si accede alla variabile v *solo all'interno di questo tipo di istruzione*:

region v when B do S

*sezione
critica*



dove B e' un'espressione booleana.

Mentre si esegue S nessun altro processo puo' accedere alla variabile v .

1. Regioni critiche: realizzazione

Quando un processo prova ad eseguire l'istruzione *region*, l'espressione booleana *B* viene valutata

Se *B* e' vera, il codice *S* viene eseguito se nessun altro processo e' in una regione associata con *v*

Se *B* e' falsa, il processo e' sospeso finche' *B* diventa vera e nessun altro processo e' in una regione associata con *v*

1. Regioni critiche: PROD-CONS

Variabili condivise :

```
var buffer :  
    shared record  
        pool : array [0..n-1] of item ;  
        count,in,out: integer  
    end;
```



Processo produttore :

```
region buffer when count < n
do begin
    pool[in] := nextp;
    (in := in + 1) mod n;
    count := count + 1;
end;
```

Processo consumatore :

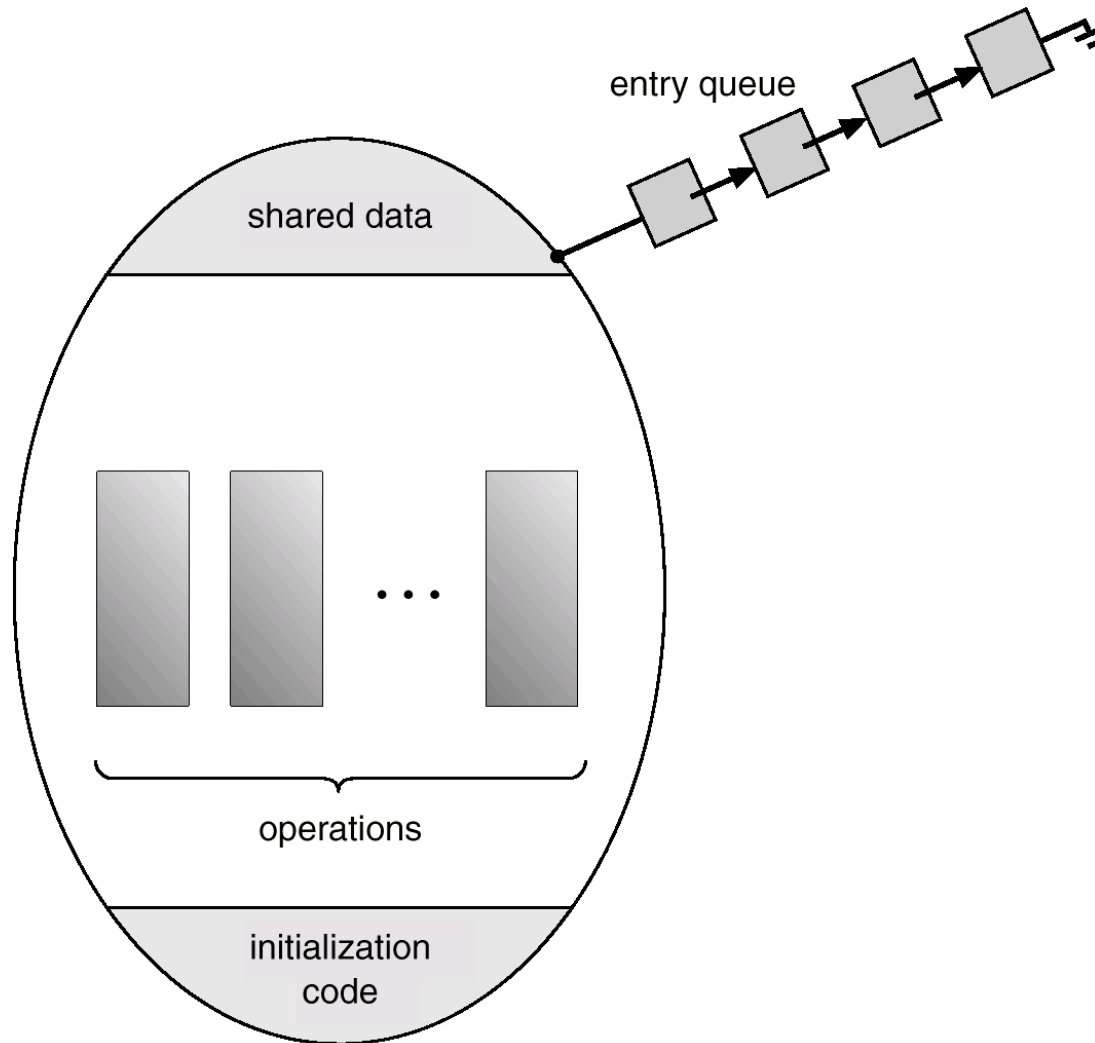
```
region buffer when count > 0
do begin
    nextc := pool[out];
    (out := out + 1) mod n;
    count := count - 1;
end;
```

2. Monitors : definizione

Costrutti di alto livello per la sincronizzazione che permettono la sicura condivisione di tipi di dati astratti tra processi concorrenti

```
type monitor-name = monitor
    dichiarazioni di variabili
    procedure entry  $P1$  (...);
        begin ... end;
    procedure entry  $P2$  (...);
        begin ... end;
    ...
    procedure entry  $Pn$  (...);
        begin...end;
    begin
        codice di inizializzazione
    end
```

2. Monitors : rappresentazione



C' e' una coda di ingresso al monitor comune a tutti i processi che vogliono eseguire una delle entry procedure (operations)

2. Monitors: variabili *condition* ...

Per permettere ad un processo di *fermarsi in attesa all'interno* di un monitor deve essere dichiarata una variabile di *tipo condition*

var x: condition

... e relative operazioni

Il processo che invoca l'operazione

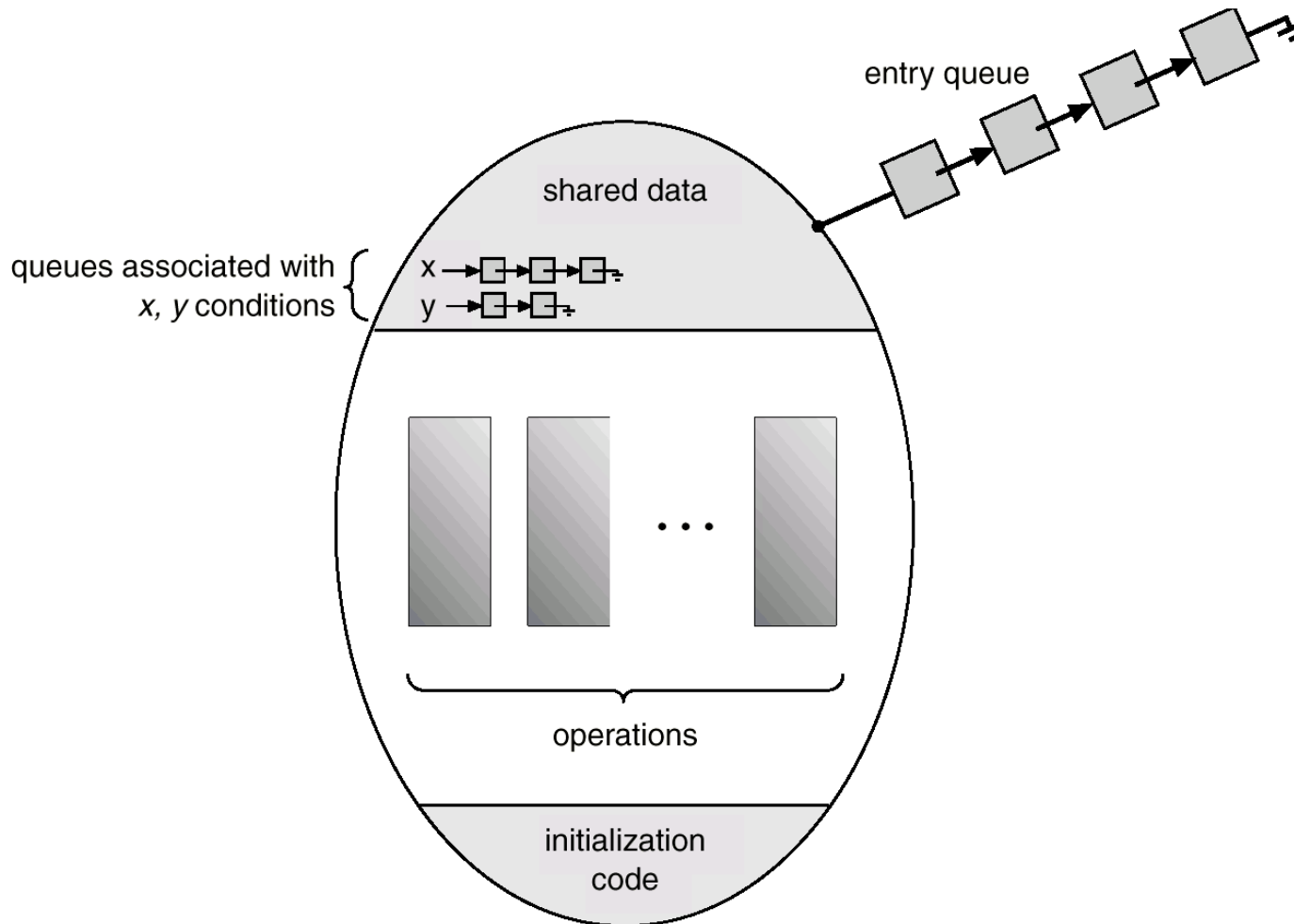
x.wait

viene sospeso finche' un altro processo invoca

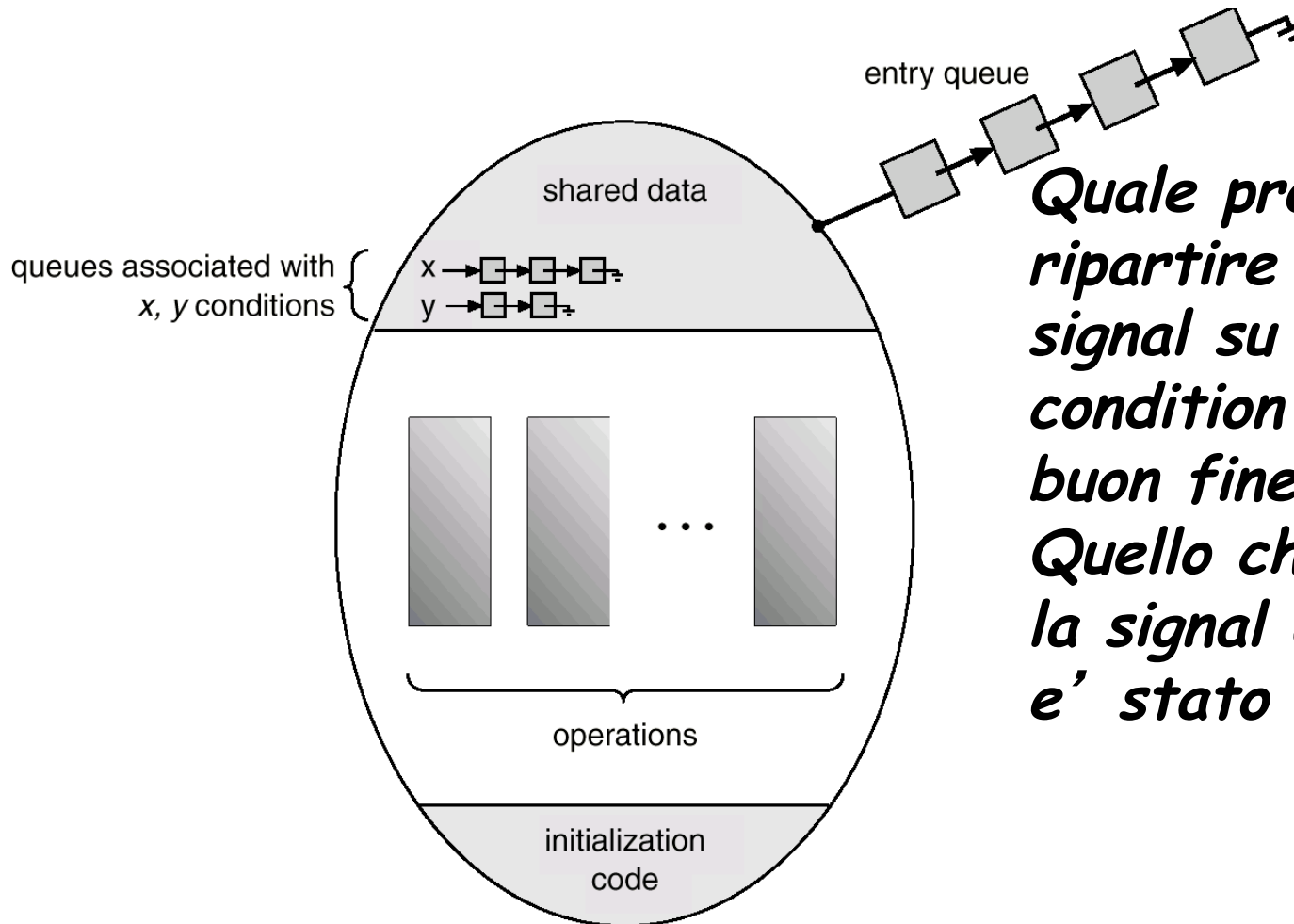
x.signal

L'operazione *x.signal* fa riprendere al piu' un unico processo. Se nessun processo era sospeso, allora l'operazione non ha effetto.

Schematica rappresentazione di un monitor con variabili condition



Schematica rappresentazione di un monitor con variabili condition



Quale processo deve ripartire dopo che una signal su una variabile condition e' andata a buon fine?!

Quello che ha eseguito la signal o quello che e' stato sbloccato?!



Un esempio : PROD-CONS...

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

... e Dining Philosophers

```
type dining-philosophers = monitor  
  var state : array [0..4] of :(thinking, hungry, eating);  
  var self : array [0..4] of condition ;  
  procedure entry pickup (i : 0..4);  
    begin  
      state[i] := hungry ;  
      test (i);  
      if state[i] ≠ eating then self[i].wait ;  
    end;  
  
  procedure entry putdown (i : 0..4);  
    begin  
      state[i] := thinking;  
      test (i+4 mod 5);  
      test (i+1 mod 5);  
    end;
```

```

procedure test(k: 0..4);
    begin
        if      state[k+4 mod 5]  $\neq$  eating
            and state[k] = hungry
            and state[k+1 mod 5]  $\neq$  eating
        then begin
            state[k] := eating;
            self[k].signal;
        end;

    end;

    begin
        for i := 0 to 4
            do state[i] := thinking;
        end
    end

```