Corso di Programmazione Concorrente

Problemi Classici

Valter Crescenzi
crescenzi@inf.uniroma3.it
http://crescenzi.inf.uniroma3.it

Sommario

- Introduzione ai Problemi Classici
- I Cinque Filosofi Mangiatori
  - richieste incrementali
  - seriale
  - prerilascio
  - richieste non incrementali
  - alloc. gerachica risorse: mancino
- Il Barbiere Dormiente
- Produttori/Consumatori
  - Mono-Produttore/Mono-Consumatore
  - P-Produttori/C-Consumantori con Buffer

I Problemi Classici (1)

- Alcuni problemi di concorrenza sono talmente consolidati da potersi oramai considerare dei “classici”
  - I Cinque Filosofi Mangiatori
  - Il Barbiere Dormiente
  - Il problema dei Produttori / Consumatori
  - …

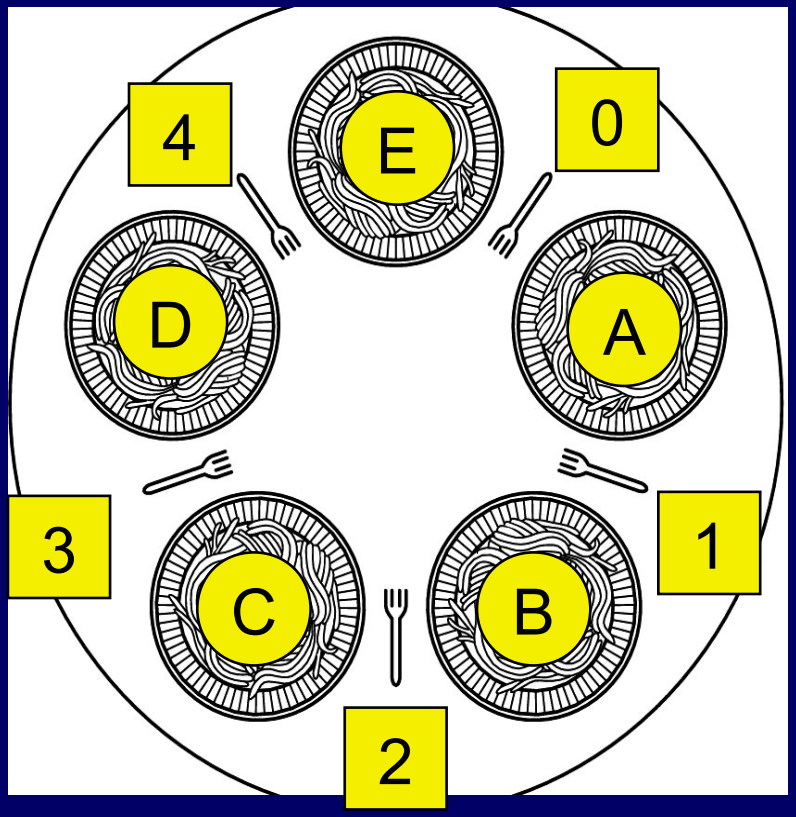
I Problemi Classici (2)

- Sono rilevanti per almeno questi motivi:
  - permettono di illustrare alcuni problemi della programmazione concorrente come lo stallo e la starvation
  - costituiscono una base di confronto per valutare l'espressività e l'eleganza di eventuali nuovi linguaggi e/o costrutti
  - taluni hanno chiarissime applicazioni pratiche
  - soluzioni consolidate si possono riutilizzare adattandole ai problemi contingenti

I Cinque Filosofi Mangiatori

Problema classico originariamente proposto da E.W. Dijkstra.
Cinque filosofi trascorrono la vita alternando, ciascuno indipendentemente dagli altri, periodi in cui pensano a periodi in cui mangiano degli spaghetti.

Per raccogliere gli spaghetti ogni filosofo necessita delle due forchette poste rispettivamente a destra ed a sinistra del proprio piatto. Trovare una strategia che consenta ad ogni filosofo di ottenere sempre le due forchette richieste.



Struttura della Soluzione al Problema dei Cinque Filosofi Mangiatori

- Prevediamo che ogni filosofo segua un protocollo di questo tipo:

  loop

  <<pensa>>;

  <<impossessati delle due forchette>>;

  <<mangia>>;

  <<rilascia le due forchette>>;

  end

- le forchette sono modellate con una variabile condivisa

  var F**: shared array**[0..4] **of** *semaforo***;**

CINQUE_FILOSOFI_MANGIATORI

```
concurrent program
   CINQUE_FILOSOFI_MANGIATORI;
type filosofo = concurrent procedure (I: 0..4);
   begin /* … */  end
var A, B, C, D, E: filosofo;
     J: 0..4;
var F: shared array[0..4] of semaforo;
begin
   for J ← 0 to 4 do INIZ_SEM(F[J],1);
   cobegin A(0) || B(1) || C(2) || D(3) || E(4) coend
end
```

Una Soluzione con Stallo:
type *filosofo*

```
concurrent program
   CINQUE_FILOSOFI_MANGIATORI;
type filosofo = concurrent procedure (I: 0..4);
   begin loop
        <<pensa>>;
        P(F[i]);  P(F[(i+1) mod 5]);
        <<mangia>>;
        V(F[(i+1) mod 5]); V(F[i]);
   end
end; /* ... */
```

Commenti a CINQUE_FILOSOFI_MANGIATORI

- Una situazione di stallo che si può facilmente presentare
  - tutti i filosofi possiedono solo la forchetta alla propria destra senza poter ottenere quella alla propria sinistra già posseduta dal vicino di tavola

- N.B. se invertissimo l'ordine delle due P e delle due V lo stallo si avrebbe comunque, ma i filosofi rimarrebbero con l'altra forchetta in mano

Una Semplice Soluzione Senza Stallo

- Cerchiamo di eliminare il problema usando un solo semaforo binario

…[omissis]…

```
var S: semaforo;
begin
   INIZ_SEM(S,1);
   cobegin A(0) || B(1) || C(2) || D(3) || E(4) coend
end
...
concurrent program CINQUE_FILOSOFI_MANGIATORI;
type   filosofo = concurrent procedure (I: 0..4);
begin loop
      <<pensa>>;
     P(S);    <<mangia>>;   V(S);
end
```

Commenti a CINQUE_FILOSOFI_MANGIATORI

- Questa soluzione
  - evita qualsiasi stallo
  - gode della proprietà di fairness
- Ma in effetti non fa altro che serializzare l'ordine con il quale i filosofi mangiano
  - può mangiare un solo filosofo alla volta
  - S associato all'intero tavolo
  - invece le forchette basterebbero per due
    - basso grado di parallelismo

Un Approccio Ottimistico

- Se un filosofo trova la seconda forchetta già impegnata, allora rilascia anche la prima, ed attende prima di riprovare
- Non compromette il livello di parallelismo ed evita lo stallo invalidando la non-prerilasciabilità delle risorse

- Questa soluzione non godrebbe comunque della proprietà di fairness: non esiste alcuna garanzia algoritmica che tutti i filosofi riescano a mangiare; qualcuno potrebbe essere sistematicamente sopraffatto dai vicini
- Se i filosofi attendono un tempo casuale prima di riprovare, al più avremmo una bassa probabilità delle seq. *unfair*, ma comunque il sistema non godrebbe della proprietà di fairness

Esercizio: provare a scrivere il codice di questa soluzione utilizzando i semafori di Dijkstra.

Approccio Conservativo (1)

- In alternativa, per evitare lo stallo senza compromettere il livello di parallelismo, rendiamo indivisibile l'operazione di acquisizione delle risorse

- Cambiamo la rappresentazione dello stato come segue:

  - **var** F**: shared array**[0..4] **of** *semaforo***;**
    - d'ora in avanti rappresenta la *coppia* di forchette che servono all'i-esimo filosofo e *non* una sola
  - **var** FP**: shared array**[0..4] **of** *boolean***;**
    - FP[i] è vero se e solo se il l'i-esima *coppia* di forchette è prenotata

Approccio Conservativo (2)

- Oltre al concetto di coppia di forchette prenotata, vengono rappresentati i filosofi “affamati”:
  - **var** A**: shared array**[0..4] **of** *boolean***;**
    - A[i] è vero se e solo se il l’i-esimo filosofo è affamato
  - **var** R**:** *semaforo***;**
    - un semaforo binario R per accedere allo stato corrente *(vettori A ed FP*) in mutua esclusione

Approccio Conservativo (3)

```
concurrent program CINQUE_FILOSOFI_MANGIATORI;
type   filosofo = concurrent procedure (I: 0..4);
    begin /* … */   end
var A, B, C, D, E: filosofo;
     J: 0..4;
var R  : semaforo;
var F  : shared array[0..4] of semaforo;
var FP : shared array[0..4] of  boolean;
var A  : shared array[0..4] of  boolean;
begin
    INIZ_SEM(R,1);
    for J ← 0 to 4 do INIZ_SEM(F[J],0);
    for I ← 0 to 4 do FP[I] ← false;
    for K ← 0 to 4 do A[K] ← false;
    cobegin A(0) || B(1) || C(2) || D(3) || E(4) coend
end
```

Approccio Conservativo (4)

```
concurrent program
   CINQUE_FILOSOFI_MANGIATORI;
type filosofo = concurrent procedure (I: 0..4);
   begin loop
        <<pensa>>;
        prendi_forchette(i)
        <<mangia>>;
        posa_forchette(i);
   end
end; /* ... */
```

Approccio Conservativo (5)

```
concurrent procedure prendi_forchette(I:0..4)
begin
    P(R);
     A[I] ← true;
     test(I);
    V(R);
    P(F[I]);
end

concurrent procedure test(I:0..4)
begin
    if (A[I] and not FP[(I-1) mod 5] and not FP[(I+1) mod 5] )
    begin
     FP[I] ← true;
     V(F[I]);
    end
end
```

Approccio Conservativo (6)

```
concurrent procedure posa_forchette(I:0..4)
begin
    P(R);
     FP[I] ← false;
     A[I] ← false;
     test( (I-1) mod 5);   test( (I+1) mod 5);
    V(R);
end

concurrent procedure test(I:0..4)
begin
    if (A[I] and not FP[(I-1) mod 5] and not FP[(I+1) mod 5] )
    begin
     FP[I] ← true;
     V(F[I]);
    end
end
```

Approccio Conservativo: Conclusioni

- Questa soluzione
  - non presenta stallo
  - gode della proprietà di fairness?
  - offre il maggior grado di parallelismo possibile
    - nessuno attende mai in presenza delle risorse necessarie per avanzare

- Ma è anche la più articolata:
  - In particolare, manca una corrispondenza immediata tra f.d.e. e filosofi
  - ciascun f.d.e. lavora *anche* per conto dei vicini

Esercizio

Esercizio: la soluzione proposta non garantisce contro l'eventualità che un filosofo muoia di fame perché sistematicamente sopraffatto dai due vicini. Trovare almeno una sequenza di interleaving *periodica* che dimostri quanto affermato.

La Proprietà di Fairness di un Algoritmo

- Accezione *stringente*: se esiste anche una sola possibile sequenza di esecuzione ammissibile in cui un flusso non avanza mai, tale algoritmo è considerato *unfair*

- Indipendentemente dalla probabilità di tale sequenza secondo un qualsiasi modello che rispetti l'*assunzione di progresso finito*

- Chiamiamo *unfair* anche tali sequenze di esecuzione ammissibili

Conseguenze Pratiche della Unfairness

- Un algoritmo *unfair* ma che dal punto di vista probabilistico sembra produrre solo seq. *unfair* con probabilità tendenti allo zero al crescere della loro lunghezza, nella pratica può esibire uno di questi comportamenti indesiderabili:
  - Starvation
    - le seq. unfair diventano probabili nell'implementazione su una certa piattaforma a causa di fattori trascurati nella modellazione
  - Forte varianza nei tempi di attesa di un f.d.e.

- Nella pratica, spesso soluzioni unfair sono accettate se non addirittura preferite, perché più efficienti in termini di throughput (richieste evase per secondo) e questi rischi sono ritenuti tollerabili

Compromessi nella Progettazione di Moduli Concorrenti

- Il problema dei cinque filosofi testimonia quanto può essere problematico disegnare soluzioni ottimali da tutti i punti di vista
- La progettazione di soluzioni concorrenti, spesso costringe a compromessi tra
  - grado di parallelismo
  - pericolo di stallo, starvation ed interferenze
  - semplicità
  - riusabilità

Una Soluzione al Problema dei Cinque Filosofi Mangiatori

Una soluzione:

- senza stallo
- con il massimo livello di parallelismo (2)
- fair
- semplice

✔si ottiene prevedendo la richiesta incrementale delle due forchette e la presenza di un filosofo *mancino* che acquisisce le forchette nell'ordine opposto agli altri

Esercizio: spiegare perché la soluzione in questione corrisponde ad applicare la tecnica di allocazione gerarchica delle risorse

Esercizio: implementare con i semafori la soluzione

Fairness della Soluzione con Mancino

Esercizio: dimostrare che la soluzione con il filosofo *mancino* non permette la sequenza di interleaving *unfair (e periodica*) utilizzata per dimostrare la mancanza di *fairness* per la soluzione precedente ("Approccio Conservativo").

Esercizio: dimostrare che la soluzione con il filosofo *mancino* gode della proprietà di *fairness.*

Sugg.: scrivere le seq. di interleaving assumendo le istruzioni del pseudo-codice, per semplicità, atomiche. Osservare che gli accessi alla forchetta contesa tra il mancino ed il suo vicino sono serializzati dal semaforo che la protegge...

Il Barbiere Dormiente

In un negozio lavora un solo barbiere, ci sono N sedie per accogliere i clienti in attesa ed una sedia di lavoro. Se non ci sono clienti, il barbiere si addormenta sulla sedia di lavoro. Quando arriva un cliente, questi deve svegliare il barbiere, se addormentato, od accomodarsi su una delle sedie in attesa che finisca il taglio corrente. Se nessuna sedia è disponibile preferisce non aspettare e lascia il negozio.

Soluzione al Barbiere Dormiente

```
concurrent program BARBIERE_DORMIENTE;
type barbiere = concurrent procedure; begin /* … */ end
type cliente = concurrent procedure;   begin /* … */ end
var Dormiente: barbiere;
var CLIENTE array[0..NUM_CLIENTI] of cliente
var C, MX, B: semaforo;
var N = 5;
var in_attesa : intero;
begin
    INIZ_SEM(MX,1); INIZ_SEM(C,0); INIZ_SEM(B,0);
    fork Dormiente;
    for J ← 0 to NUM_CLIENTI do fork CLIENTE[J];
    for J ← 0 to NUM_CLIENTI do join CLIENTE[J];
    join Dormiente;
end
```

Soluzione al Barbiere Dormiente

type *barbiere*

```
concurrent program BARBIERE_DORMIENTE;
type barbiere = concurrent procedure;
   begin loop
    P(C);       // attendi clienti
    P(MX);      // aggiorna in m.e. il
    in_attesa ← in_attesa - 1;// n. di clienti in attesa
    V(B);       // segnala la disponibilità
    V(MX)       // del barbiere
    <<taglia capelli>>;
   end
end; /* ... */
```

Soluzione al Barbiere Dormiente

type *cliente*

```
concurrent program BARBIERE_DORMIENTE;
type cliente = concurrent procedure;
    begin
     <<raggiungi il negozio>>
     P(MX);
     if (in_attesa < N) begin// se non ci sono posti lascia
           in_attesa ← in_attesa + 1;
           V(C);          // sveglia il barbiere se dorme
           V(MX);
           P(B);          // aspetta che il barbiere finisca
           <<siedi per il taglio>>;
     end
     else V(MX);     // neanche un posto a sedere: meglio ripassare
    end
```

Esercizio

Esercizio: quali semafori sono usati in stile *competitivo* dal programma BARBIERE_DORMIENTE? E quali in stile *cooperativo?*

Esercizio

Esercizio: la soluzione proposta non garantisce contro l'eventualità che alcuni clienti rubino il posto a chi è arrivato prima nella bottega (ovvero riescano a fare una **P**(B) prima di chi li ha preceduti nel fare la **P**(MX) iniziale).

- Trovare almeno una seq. di interleaving che testimoni quanto affermato
- Risolvere il problema introducendo il concetto di "numeretto" elimina code: per poter essere serviti i clienti devono "strappare" un numero progressivo non appena entrano nella bottega e quindi mostrare il numero corrispondente al prossimo turno servito per poter effettuare la **P**(B)

Esercizio

Esercizio: considerando lo sviluppo del precedente esercizio, si può affermare che la soluzione senza *numeretto* goda della proprietà di fairness.

Interazione tra F.d.E. Concorrenti

- Due f.d.e. possono essere:
  - disgiunti
  - interagenti
    - *competizione*: due o più f.d.e. chiedono l'uso di una risorsa comune riusabile e di molteplicità finita
    - *cooperazione*: due o più f.d.e. cooperano per raggiungere un obiettivo comune

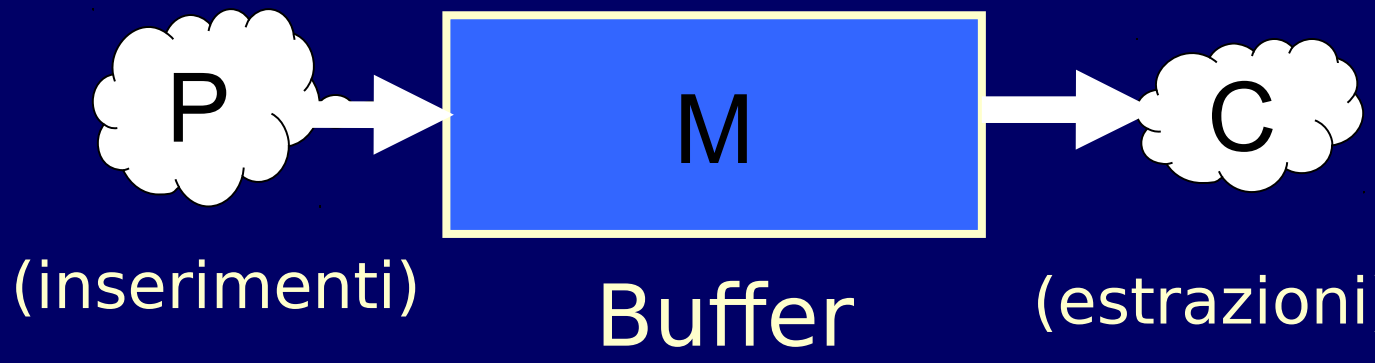
Cooperazione tra F.d.E.

- I f.d.e. interagiscono non solo competendo per le risorse comuni ma anche collaborando per raggiungere obbiettivi comuni
  - Interazione tra f.d.e.
    - diretta o cooperazione
    - indiretta o competizione

- Semafori come strumenti per *anche* problemi di cooperazione
- Il più semplice dei problemi di cooperazione >>

Problema Produttore / Consumatore

- Un f.d.e. P (produttore) deve continuamente inviare messaggi ad un altro f.d.e. C (consumatore) che li elabora nello stesso ordine in cui li ha ricevuti
- Viene utilizzato un buffer di scambio
  - P scrive i messaggi nel buffer
  - C legge i messaggi dal buffer



Produttore / Consumatore: Sincronizzazioni Necessarie

- I due f.d.e. devono essere opportunamente sincronizzati contro queste eventualità:
  - C legge un messaggio senza che P ne abbia depositato alcuno
  - P sovrascrive un messaggio senza che C sia riuscito a leggerlo
  - C legge lo stesso messaggio più di una volta

- Essenzialmente bisogna sincronizzarsi su due eventi a cui associamo due diversi semafori binari
  - DEPOSITATO:
    - ad 1 se e solo se un messaggio è stato depositato ed è prelevabile
  - PRELEVATO:
    - ad 1 se e solo se il buffer è vuoto e pronto ad accogliere un nuovo messaggio

Un Produttore / Un Consumatore

```
concurrent program COOPERAZIONE;
type   messaggio=…;
var    M: messaggio;
       DEPOSITATO, PRENOTATO:semaforo_binario;
```

```
concurrent procedure PROD
loop begin
    <produci un messaggio in M>
    P(PRELEVATO);
    BUFFER ← M;
    V(DEPOSITATO);
end
```

```
concurrent procedure
CONS loop begin
    P(DEPOSITATO);
    M ← BUFFER;
    V(PRELEVATO);
    <consuma il messaggio in M>
end
```

```
begin
    INIZ_SEM(PRELEVATO,1); INIZ_SEM(DEPOSITATO,0);
    cobegin PROD || CONS coend
end
```

Esercizi

Esercizio: Descrivere gli effetti di ciascuno dei seguenti errori sulla soluzione di Produttore / Consumatore

- Il semaforo PRELEVATO viene inizializzato a 0 ed il semaforo DEPOSITATO viene inizializzato ad 1
- La **V**(DEPOSITATO) viene sostituita da una **V**(PRELEVATO)
- Manca la **V**(PRELEVATO) nel consumatore

Accoppiamento tra Produttore e Consumatore

- La soluzione presentata, sebbene corretta, comporta la perfetta alternanza di produzione e consumazione
  - C non può consumare fino a quando il produttore non produce
  - P non può produrre fino a quando il consumatore non consuma
  - Unica possibile sequenza di esecuzione ammissibile:

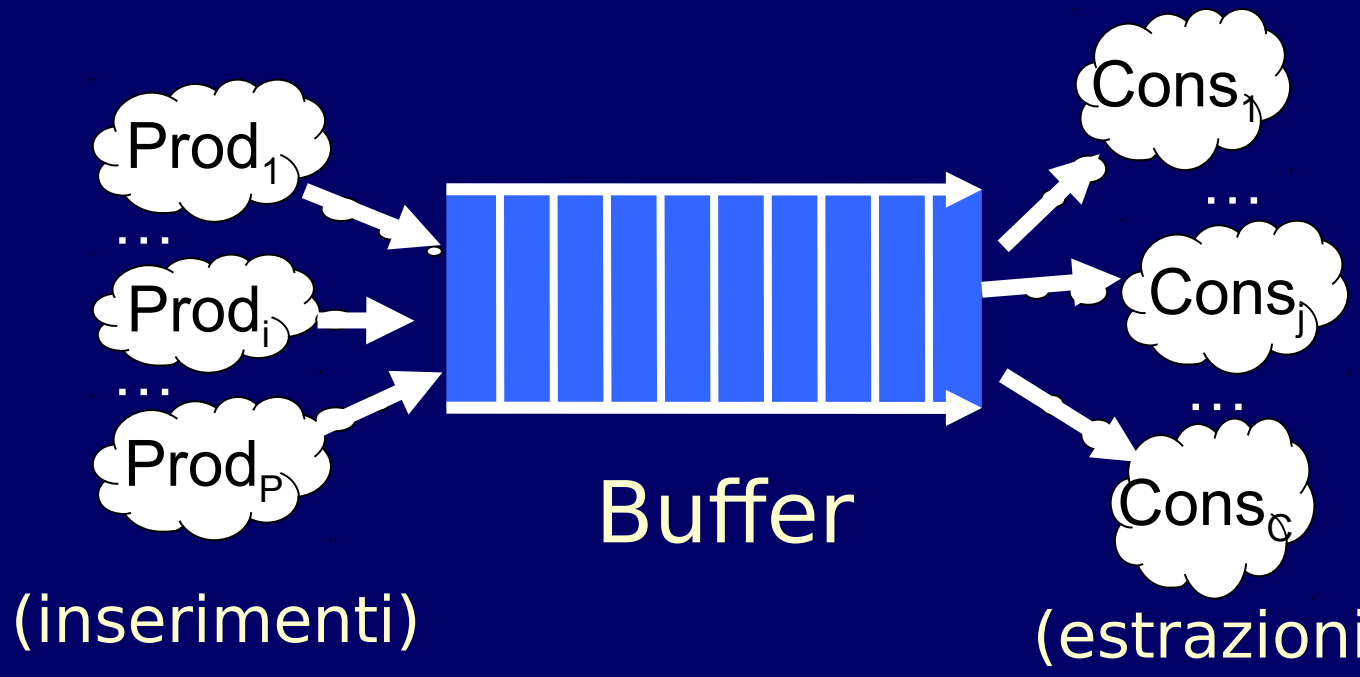
  produci-consuma-produci-consuma-…-
- A meno che produzione e consumazione abbiano esattamente la stessa durata, entrambi i f.d.e. passano del tempo sospesi uno in attesa dell'altro
  - su una macchina multi-processore questo può voler dire non utilizzare completamente la capacità di elaborazione disponibile

Disaccoppiamento dei F.d.E.

- Conviene minimizzare le attese (passive) dei f.d.e.
- In alcune situazioni, si cerca addirittura di eliminarle
  - per sfruttare appieno i processori fisici disponibili laddove non si possono prevedere tempi di produzione e consumazione paragonabili
  - per esigenze di specifici f.d.e.
    - es.: gestore di un masterizzatore
  - per fornire garanzie sulle prestazioni (ad es. sul tempo massimo di elaborazione per unità)
- meglio evitare l'eccessivo sincronizzazione tra la produzione e la consumazione permettendo al buffer di conservare molteplici messaggi

Problema Produttori / Consumatori

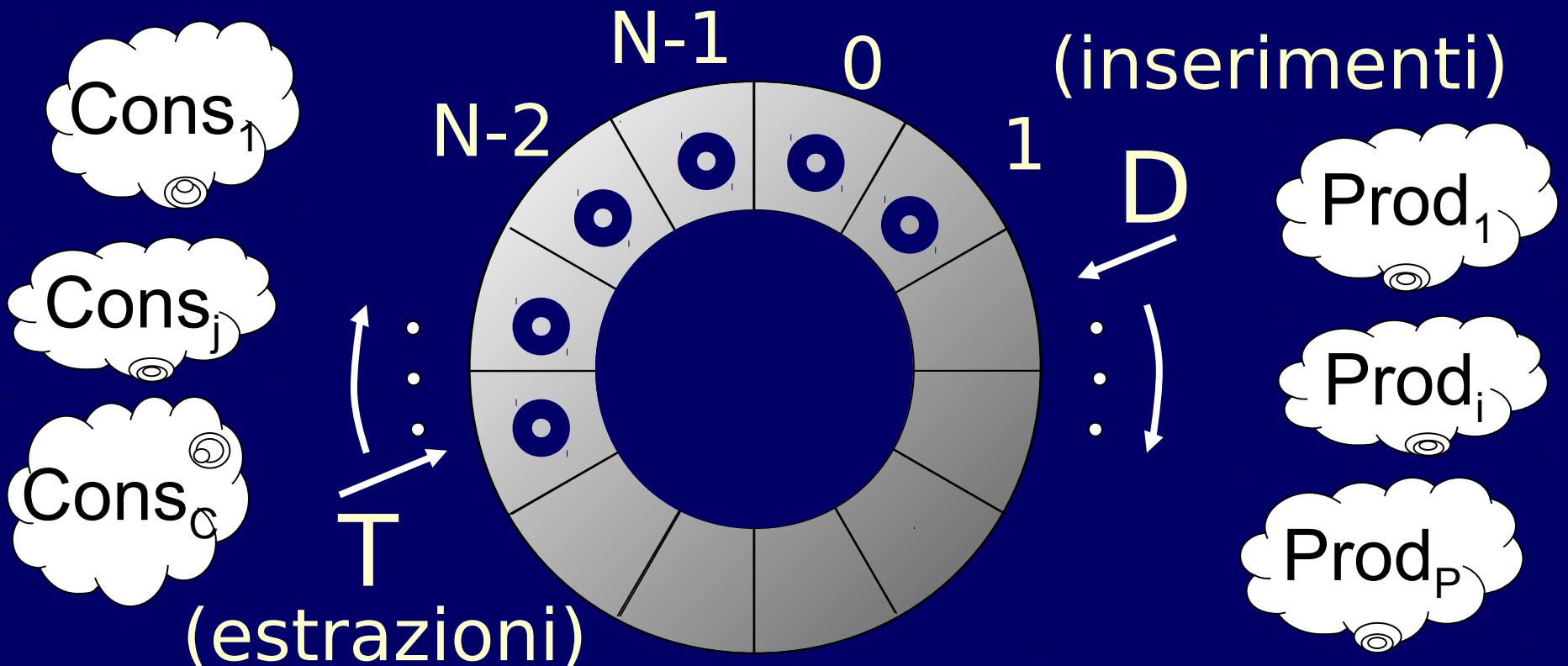
- Alcuni f.d.e. P_i (produttori) devono inviare messaggi ad altri f.d.e. C_j (consumatori) che li elaborano nello stesso ordine in cui li hanno ricevuti



Produttori / Consumatori: Sincronizzazioni Necessarie

- Sono necessarie sincronizzazioni analoghe a quelle viste nel caso di due soli f.d.e. ed inoltre bisogna evitare che:
  - un produttore scriva su un buffer già pieno
  - un consumatore legga da un buffer vuoto
- Il buffer può essere vista come una risorsa di molteplicità finita per il quale competono sia i produttori che i consumatori
  - semafori a conteggio inizializzati alla dimensione del buffer
- E' classica l'implementazione a buffer circolare

Buffer Circolare



Buffer Circolare

PRODUTTORI_CONSUMATORI

concurrent program PRODUTTORI_CONSUMATORI;
sia N=…;
type *messaggio=…;*
indice = 0..N-1;
var BUFFER: array[*indice*] of *messaggio*;
T, D:*indice*;
PIENE, VUOTE: *semaforo*;
USO_T, USO_D: *semaforo_binario;*
concurrent procedure $PROD_i$ /* generico produttore */
concurrent procedure $CONS_j$ /* generico consumatore */
begin INIZ_SEM(USO_D,1); INIZ_SEM(USO_T,1);
INIZ_SEM(PIENE,0); INIZ_SEM(VUOTE,N);
T ← 0; D ← 0;
cobegin … || $PROD_i$ || $CONS_j$ || … **coend**
end

$PROD_i$

```
concurrent procedure PROD_i //generico produttore
var M: messaggio;
  loop
    <produci un messaggio in M>
    P(VUOTE);
        P(USO_D);
            BUFFER[D] ← M;
            D ← (D+1) mod N;
        V(USO_D);
    V(PIENE);
  end
```

$CONS_j$

```
concurrent procedure CONS_j //generico consumatore
var M: messaggio;
  loop
    P(PIENE);
        P(USO_T);
            M ← BUFFER[T];
            T ← (T+1) mod N;
        V(USO_T);
    V(VUOTE);
    <consuma il  messaggio in M>
  end
```

Produttori / Consumatori: Commenti

- T: indice utilizzato per le estrazioni
- D: indice utilizzato per gli inserimenti
- Attenzione all'utilizzo dei semafori:
  - Binari (in stile competitivo)
    - USO_D: mutua esclusione dei produttori su D
    - USO_T: mutua esclusione dei consumatori su T
  - a conteggio (in stile cooperativo?)
    - PIENE: per assicurarsi la disponibilità di celle piene
      - assume un valore pari al numero di celle piene
    - VUOTE: per assicurarsi la disponibilità di celle vuote
      - assume un valore pari ad N – n. celle piene

Commenti di Chiusura

- Tra tutti i problemi classi è chiaramente quello che ha più immediata ed evidente utilità pratica
- A ben vedere, anche un semplice *hand-off*, ovvero passaggio asincrono di un messaggio, presuppone la sua soluzione (nella versione a buffer unitario)
  - Nelle librerie moderne: i *Future*
- Recentemente, per la grande diffusione di computazioni distribuite, ha acquistato rinnovato interesse la sua soluzione con il vincolo di uso di memoria limitato
  - *http://www.reactive-streams.org/*
- Due nodi di processamento procedono a velocità diverse
  - *Back-pressure:* chiedere ai produttori di rallentare (>>HWC1)

Esercizi

Esercizio: scrivere una funzione che restituisce il numero di celle correntemente occupate nel buffer, nell'ipotesi che *non* sia disponibile una primitiva per interrogare il valore dei semafori a conteggio.

Suggerimento: rivedere la versione presentata effettuando incrementi non circolari dei due indici D e T ed accessi circolari alle celle del buffer trascurando il problema di overflow sul tipo intero.

Esercizi

Esercizio: ipotizzare tempi di produzione e consumazione casuali ma della medesima durata massima per produttori e consumatori. Descrivere come cambia il tempo di consumazione medio al variare del

- del numero di produttori
- del numero di consumatori
- delle dimensioni del buffer

Esercizio: risolvere il problema produttori / consumatori nel caso che sia disponibile un buffer di dimensioni illimitate. Ripetere l'esercizio di sopra con questa ipotesi semplificativa cercando una soluzione più semplice che utilizzi *meno* semafori.

Esercizi

Esercizio: ipotizzare tempi di produzione e consumazione casuali ma della medesima durata massima per produttori e consumatori. Descrivere come cambia il tempo di attesa passiva sulle quattro invocazioni delle primitiva P() al variare:

- del numero di produttori
- del numero di consumatori
- delle dimensioni del buffer

Esercizi

Esercizio: per velocizzare la sezione critica sull'indice **D** si è proposto di utilizzare un indice di appoggio e quindi di spostare l'accesso al buffer circolare fuori della sezione critica, come segue:

concurrent procedure $PROD_i$ //generico produttore

var M: *messaggio*; app: *indice*;

```
 loop   <produci un messaggio in M>
      P(VUOTE);
       P(USO_D);
         app ← D;
         D ← (D+1) mod N;
       V(USO_D);
       BUFFER[app] ← M;
      V(PIENE);
end
```

Verificare la soluzione ed eventualmente trovare una seq. di interleaving che ne dimostri l'erroneità

Esercizi Produttori / Consumatori

Esercizio: in molti casi pratici risulta utile utilizzare una variante del *produttori / consumatori* dove le operazioni di inserimento e prelievo risultino *non-bloccanti*, ovvero dove l'operazione di inserimento su un buffer pieno e di prelievo da un buffer vuoto restituisce immediatamente un errore, anziché causare la sospensione del f.d.e. invocante. Ragionare se e come sia possibile impostare una soluzione a questo problema utilizzando i soli semafori di Dijkstra.