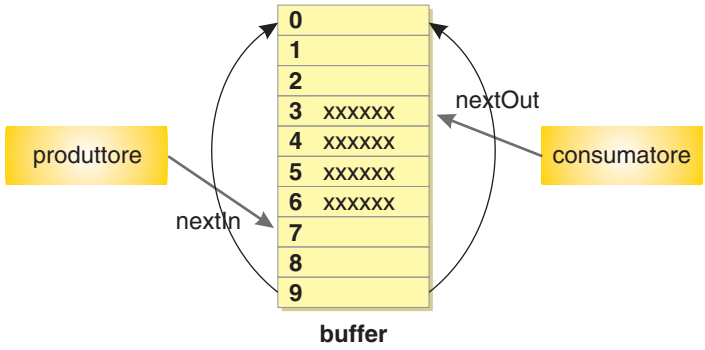
Il problema del produttore-consumatore con un buffer circolare

La soluzione al problema del *produttore-consumatore*, presentata nel Paragrafo 6, deve essere modificata nel caso in cui il buffer sia di dimensione maggiore di 1. La seguente figura mostra un buffer in grado di memorizzare i dati di 10 cicli di produzione.



Il buffer può essere definito come un array di stringhe:

```
String buffer[] = new String[10];
```

Il produttore deposita i dati in *buffer[0]*, *buffer[1]* e così via sino a *buffer[9]*, per poi ricominciare da *buffer[0]*.
Il processo consumatore si comporta nello stesso modo, prelevando i dati a partire da *buffer[0]* sino a *buffer[9]*, per riprendere da *buffer[0]*. Per questa ragione il buffer si chiama **buffer circolare**.
Per gestire il buffer circolare si usano due puntatori *nextIn* e *nextOut*:

- *nextIn* indica l'elemento di *buffer* dove il produttore deposita i dati;
- *nextOut* indica al consumatore la posizione del dato da prelevare.

Nel buffer circolare in figura *nextIn* ha valore 7 e *nextOut* ha valore 3. Il produttore depositerà il prossimo dato in *buffer[7]*, il consumatore preleverà il prossimo dato da *buffer[3]*. Nella figura si vede anche che nel buffer ci sono quattro dati da prelevare.

Nel buffer circolare, dopo ogni inserimento, il valore di *nextIn* deve essere modificato nel seguente modo:

```
nextIn = (nextIn + 1) % n;
```

Con *n* si indica il numero di elementi del buffer (nell'esempio della figura *n* vale10) e il carattere % indica il resto della divisione intera tra due numeri. Il consumatore, dopo ogni inserimento, aggiorna il valore di *nextOut* in modo analogo:

```
nextOut = (nextOut + 1) % n;
```

Se il valore iniziale di entrambi i puntatori è 0, per effetto dei precedenti assegnamenti, essi assumeranno i seguenti valori:

0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,...

I puntatori puntano ciclicamente a tutti gli elementi del buffer, che sarà riempito e svuotato circolarmente a partire da *buffer[0]*. I processi produttore e consumatore procedono in parallelo: il produttore inserisce dati nel buffer, ma solo se il buffer non è pieno, e il consumatore li preleva, a condizione che il buffer non sia vuoto.

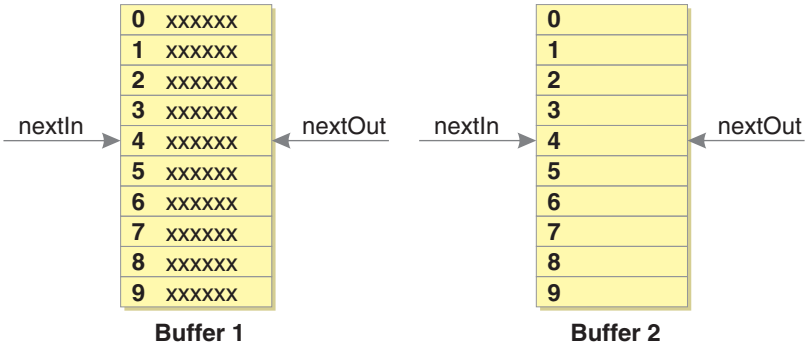
La classe *BufferCircolare*, presentata di seguito, implementa il comportamento dei due processi e utilizza due semafori *pieno* e *vuoto*: essi sono inizializzati in modo da dare via libera per *n* volte al produttore e di bloccare inizialmente il consumatore per impedire che acceda a un buffer vuoto.

```
/*-------------------------------------------------------------------
 * produttore-consumatore con un buffer circolare
 *-------------------------------------------------------------------*/
class BufferCircolare
{
   static final int n = 10;
   static int nextIn = 0;
   static int nextOut = 0;
   static Semaforo pieno(10);
   static Semaforo vuoto(0);
   String buffer[] = new String[10];

   public static void main(String args[])
   {
      /*----------------------------------------------------------------
       * produttore e consumatore sono eseguiti in parallelo
       *----------------------------------------------------------------*/
      produttore();
      consumatore();
   }
/*-------------------------------------------------------------------
 * produttore esegue                consumatore esegue
 *-------------------------------------------------------------------*/
   void produttore()                void consumatore()
   {                                {
      String dato;                     String dato;
      while (ciSonoDati)               while (ciSonoDati)
      {                                {
         produce(dato);                   vuoto.wait();
         pieno.wait();                    dato = buffer[nextOut];
         buffer[nextIn] = dato;           nextOut = (nextOut+1) % n;
         nextIn = (nextIn+1) % n;         pieno.signal();
         vuoto.signal();                  consuma(dato);
      }                                }
   }                                }
}
```

pieno: via libera 10 volte al produttore
vuoto: blocca subito il consumatore

Il precedente schema di algoritmo è corretto e non ci sono corse critiche nell'accesso ai dati nel buffer: infatti se *nextIn* è diverso da *NextOut*, l'accesso avviene in posizioni differenti del buffer, escludendo quindi la possibilità di avere corse critiche. Queste si possono verificare solo quando *nextIn* è uguale a *nextOut*, se il produttore e il consumatore accedono al buffer contemporaneamente. Occorre però notare che quando *nextIn* e *nextOut* sono uguali l'accesso al buffer è consentito al solo produttore, oppure al solo consumatore e quindi, anche in questo caso, le corse critiche sono escluse. Infatti *NextIn* è uguale a *NextOut* in due casi: quando il buffer è pieno oppure quando il buffer è vuoto. Le due situazioni sono schematizzate in figura.



A sinistra (*buffer 1*) è mostrato un caso di buffer pieno. Questa situazione si può presentare se, per qualche motivo, il consumatore si ferma. La situazione di un buffer vuoto mostrata nella figura a destra (*buffer 2*) si può presentare se si ferma il produttore.
Analizziamo il caso di buffer pieno: il produttore dopo ogni inserimento ha eseguito *pieno.wait()*, che ha decrementato *pieno.count*. Quando il buffer è stato completamente riempito, *pieno.count* ha valore 0 e un ulteriore tentativo di accedere al buffer causa la sospensione del produttore. Se il buffer è pieno, solo il consumatore
può accedere al buffer. Analogamente, se il buffer è vuoto, solo il produttore può accedere al buffer. Riassumendo, se *nextIn* è uguale a *nextOut*, l'accesso al buffer è consentito a uno solo dei due processi.
La programmazione concorrente deve essere gestita con molta attenzione, perché spesso una piccola modifica può generare nuovi problemi. Si consideri, per esempio, il problema precedente e si pensi alla sua generalizzazione nel caso in cui ci siano più processi produttori e più processi consumatori che accedono al buffer. Questa situazione trova riscontri in casi reali: si pensi a un magazzino che ha una capacità limitata, come il buffer del problema precedente; in genere, ci sono più processi produttivi che effettuano carichi di prodotti in magazzino e altri che li scaricano.
Per generalizzare la soluzione al caso di molti produttori e molti consumatori, occorre tener conto del fatto che più produttori/consumatori possono accedere al buffer. Di conseguenza le due sequenze di istruzioni:

```
buffer[nextIn] = dato;          dato = buffer[nextOut];
nextIn = (nextIn+1) % n;        nextOut = (nextOut+1) % n;
```

possono dare luogo a corse critiche e, per evitare l'inconsistenza dei dati, devono essere eseguite in mutua esclusione. La soluzione al problema del buffer circolare deve essere modificata in modo che l'accesso al buffer avvenga in mutua esclusione. Nella classe *ProduttoriConsumatori* si usano le notazioni *produttore(i)* e *consumatore(i)* per indicare il generico processo produttore/consumatore.

```
/*-------------------------------------------------------------------
 * ProduttoriConsumatori con un buffer circolare
 *-------------------------------------------------------------------*/
class ProduttoriConsumatori
{
   static final int n = 10;
   static int nextIn = 0;
   static int nextOut = 0;
   static Semaforo pieno(10);
   static Semaforo vuoto(0);
   static Semaforo mutex(1);
   String buffer[] = new String[10];
```

pieno: via libera 10 volte al produttore
vuoto: blocca subito il consumatore
mutex: accesso al buffer in mutua esclusione

```
   public static void main(String args[])
   {
      /*-------------------------------------------------------------------
       * tutti i processi sono eseguiti in parallelo
       *-------------------------------------------------------------------*/
      produttore(1); produttore(2);   . . . ;
      consumatore(1); consumatore(2);   . . . ;
   }
/*--------------------------------------------------------------------
 * produttore(i) esegue                consumatore(i) esegue
 *--------------------------------------------------------------------*/
   void produttore(int i)              void consumatore(int i)
   {                                   {
      String dato;                        String dato;
      while (ciSonoDati)                  while (ciSonoDati)
      {                                   {
         produce(dato);                      vuoto.wait();
         pieno.wait();                       mutex.wait()
         mutex.wait();                          dato = buffer[nextOut];
            buffer[nextIn] = dato;              nextOut = (nextOut+1) % n;
            nextIn = (nextIn+1) % n;         mutex.signal();
         mutex.signal();                     pieno.signal();
         vuoto.signal();                     consuma(dato);
      }                                   }
   }                                   }
}
```

Le parti evidenziate in giallo indicano le porzioni di codice eseguite in mutua esclusione.
La mutua esclusione è ottenuta con il semaforo *mutex* inizializzato a 1, con la tecnica presentata nel Paragrafo 6.