Allocatore di memoria

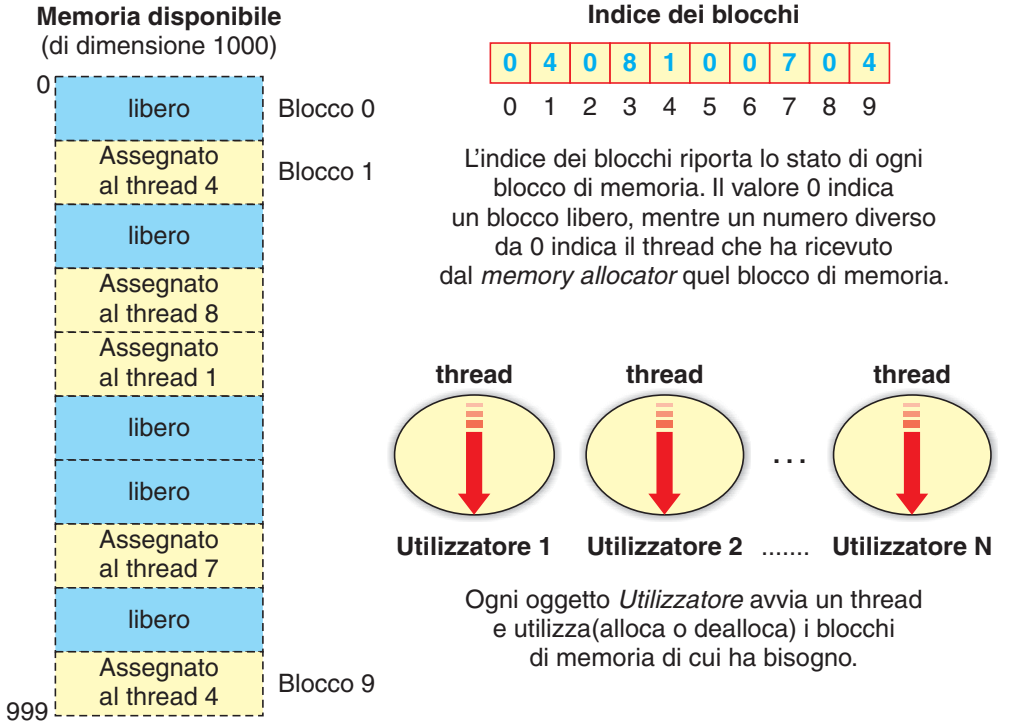
Un sistema operativo deve mettere a disposizione dei programmi le aree di memoria, o di disco, che essi richiedono, utilizzano e poi liberano, restituendole al sistema operativo stesso. Si pensi alle aree di memoria che i programmi possono utilizzare per i propri dati dinamici, oppure per memorizzare temporaneamente pacchetti di dati che arrivano dalla rete. Oppure diversi programmi possono condividere un'area di disco su cui memorizzare temporaneamente dei dati che vengono cancellati dopo essere stati elaborati.

Il programma che realizza funzionalità di gestione di aree di memoria condivise viene generalmente chiamato **allocatore di memoria** (*memory allocator*).

Progetto

Implementare un allocatore di memoria che gestisce 10 blocchi di memoria, messi a disposizione di più thread (utilizzatori) che alternativamente richiedono, utilizzano e restituiscono tali blocchi. Si identifichino i vari thread concorrenti con un numero, e si marchi ogni blocco di memoria con l'identificativo del thread che sta utilizzando quel blocco.

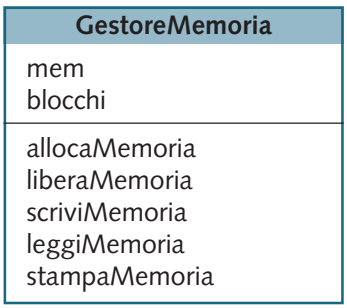
In questo esempio si immagina di gestire un'area di memoria contenente 1000 numeri interi, suddivisi in 10 blocchi da 100.
Per tener traccia dei blocchi liberi e di quelli occupati, si crea un indice rappresentato con un array di dieci elementi, uno per ogni blocco.
Il valore di un elemento dell'indice è 0 se il blocco corrispondente è libero.
Se un blocco di memoria è stato invece assegnato a un thread utilizzatore, il corrispondente elemento dell'indice riporta il numero che identifica l'utilizzatore stesso.



La classe *GestoreMemoria* definisce le variabili locali:
- *mem*, contenente la memoria da gestire (di dimensione 1000 in questo esempio);
- *blocchi*, contenente l'indice che riporta i blocchi liberi e quelli occupati.

I metodi da implementare sono i seguenti:
- *allocaMemoria* viene invocato da un thread che necessita di un blocco di memoria, e assegna il primo blocco libero al thread richiedente. Questo metodo ritorna il numero di blocco libero. Se nessun blocco è libero, questo metodo ritorna il valore -1.
- *liberaMemoria* viene invocato da un thread quando esso non ha più bisogno del blocco di memoria che gli era stato allocato. Questo metodo marca come libero (valore zero nell'indice) il blocco "liberato" dal thread.
- *scriviMemoria* scrive un dato numerico intero in una certa posizione del blocco di memoria allocato al thread chiamante;
- *leggiMemoria* legge il dato numerico intero presente in una certa posizione del blocco di memoria allocato al thread chiamante;
- *stampaMemoria* riporta a video il contenuto dell'indice blocchi.

Diagramma della classe



Implementazione della classe

```
class GestoreMemoria
{
   final int dimMem = 1000;     // dimensione della memoria disponibile
   final int dimBlocco = 100;   // dimensione del singolo blocco di memoria
   final int numBlocchi = 10;   // numero di blocchi di memoria disponibili
   int mem[];                   // spazio di memoria da gestire
   int blocchi[];               // indice dei blocchi occupati o liberi

   public GestoreMemoria()
   {
      mem = new int[dimMem];
      blocchi = new int[numBlocchi];
      for (int i = 0; i < numBlocchi; i++)
      {
         blocchi[i] = 0;         // inizio: tutti i blocchi liberi
      }
   }

   synchronized int allocaMemoria(int idUtilizzatore)
   {
      int bloccoLibero = 0;
      while (bloccoLibero < numBlocchi)
```

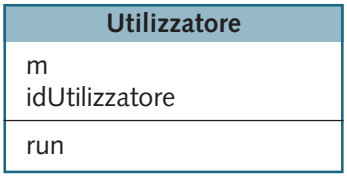
```
      {
         if (blocchi[bloccoLibero] != 0)
         {
            bloccoLibero++;
         }
         else
         {
            break;
         }
      }
      if (bloccoLibero < numBlocchi)
      {
         // il blocco di memoria viene marcato
         // con l'identificatore dell'utilizzatore
         blocchi[bloccoLibero] = idUtilizzatore;
         return bloccoLibero;
      }
      else
      {
         return -1;
      }
   }

   synchronized void liberaMemoria(int bloccoDaLiberare)
   {
      blocchi[bloccoDaLiberare] = 0;
   }

   synchronized void scriviMemoria(int blocco, int posizione, int dato)
   {
      mem[blocco*dimBlocco + posizione] = dato;
   }

   synchronized int leggiMemoria(int blocco, int posizione)
   {
      return(mem[blocco*dimBlocco + posizione]);
   }

   synchronized void stampaMemoria()
   {
      System.out.println("\t\t\t\tStato della memoria: "  + blocchi[0]
               +blocchi[1]+blocchi[2]+blocchi[3]+blocchi[4]+
               blocchi[5]+blocchi[6]+blocchi[7]+blocchi[8]+blocchi[9]);
   }
}
```

La classe *Utilizzatore* implementa i thread che richiedono l'allocazione e la deallocazione di blocchi di memoria. Essa definisce due variabili:

- *m*, che fa riferimento al *memory allocator*, istanza di *GestoreMemoria*;
- *idUtilizzatore*, il numero assegnato a ogni istanza di *Utilizzatore* (corrispondente a un thread) per identificarla, e per marcare i blocchi di memoria ad esso assegnati. Per semplicità, in questo esempio si assume di avere un massimo di 9 utilizzatori concorrenti, per rendere più facile la lettura di quanto contenuto nell'indice blocchi del gestore di memoria.

Ogni volta che viene creata un'istanza di *Utilizzatore*, viene creato un nuovo thread che esegue il metodo *run* di questa classe.

Diagramma della classe



Implementazione della classe (*Utilizzatore.java*)

```
class Utilizzatore implements Runnable
{
   GestoreMemoria m;
   int idUtilizzatore;
   Utilizzatore(GestoreMemoria miaMemoria, String nomeThread, int id)
   {
      m = miaMemoria;
      idUtilizzatore = id;
      new Thread(this, nomeThread).start();
   }

   public void run()
   {
      int blocco1, blocco2;
      String nomeThread = Thread.currentThread().getName();
      for (int i = 0; i < 3; i++)
      {
         blocco1 = m.allocaMemoria(idUtilizzatore);
         if (blocco1 != -1)
         {
            System.out.println(nomeThread + " ha allocato il blocco " + blocco1);
         }
         else
         {
            System.out.println(nomeThread + " non ha trovato blocchi liberi");
         }
         m.stampaMemoria();

         blocco2 = m.allocaMemoria(idUtilizzatore);
         if (blocco2 != -1)
         {
            System.out.println(nomeThread + " ha allocato il blocco " + blocco2);
         }
         else
         {
            System.out.println(nomeThread + " non ha trovato blocchi liberi");
         }
         m.stampaMemoria();

         if (blocco2 != -1)
         {
         m.liberaMemoria(blocco2);
            System.out.println(nomeThread + " ha liberato il blocco " + blocco2);
         m.stampaMemoria();
         }
```

```
            if (blocco1 != -1)
            {
               m.liberaMemoria(blocco1);
               System.out.println(nomeThread + " ha liberato il blocco " + blocco1);
               m.stampaMemoria();
            }
         }
      }
   }
```

Il codice del metodo *run* della classe *Utilizzatore* genera un'attività di allocazione e deallocazione di memoria da parte del thread in esecuzione. Esso ha un ciclo *for* che per tre volte esegue due allocazioni e due deallocazioni di blocchi di memoria, invocando ogni volta il metodo *stampaMemoria* del gestore di memoria. Si osservi che questo metodo visualizza le 10 cifre che rappresentano l'allocazione attuale dei blocchi. Un blocco libero viene indicato dalla cifra 0, mentre una cifra diversa da 0 indica l'identificativo dell'*Utilizzatore* che ha preso possesso del blocco di memoria.
Il *main* crea un *memory allocator* con il nome *areaMemoria*, e poi crea 8 utilizzatori: in questo modo viene generato un traffico di allocazioni e deallocazioni, fino a saturare i blocchi disponibili.

Programma Java (*Allocatore.java*)

```
public class Allocatore
{
   public static void main(String args[])
   {
      GestoreMemoria areaMemoria = new GestoreMemoria();
      new Utilizzatore(areaMemoria, "Utilizzatore 1", 1);
      new Utilizzatore(areaMemoria, "Utilizzatore 2", 2);
      new Utilizzatore(areaMemoria, "Utilizzatore 3", 3);
      new Utilizzatore(areaMemoria, "Utilizzatore 4", 4);
      new Utilizzatore(areaMemoria, "Utilizzatore 5", 5);
      new Utilizzatore(areaMemoria, "Utilizzatore 6", 6);
      new Utilizzatore(areaMemoria, "Utilizzatore 7", 7);
      new Utilizzatore(areaMemoria, "Utilizzatore 8", 8);
   }
}
```

Un esempio di output del programma è il seguente:

```
Utilizzatore 1 ha allocato il blocco 0
                    Stato della memoria: 1000000000
Utilizzatore 1 ha allocato il blocco 1
                    Stato della memoria: 1100000000
Utilizzatore 1 ha liberato il blocco 1
                    Stato della memoria: 1000000000
.
.
.
```

```
Utilizzatore 2 ha allocato il blocco 1
Utilizzatore 3 ha allocato il blocco 2
Utilizzatore 4 ha allocato il blocco 3
Utilizzatore 5 ha allocato il blocco 4
Utilizzatore 6 ha allocato il blocco 5
Utilizzatore 7 ha allocato il blocco 6
Utilizzatore 8 ha allocato il blocco 7
                    Stato della memoria: 1234567800
.
.
.
Utilizzatore 3 ha allocato il blocco 9
                    Stato della memoria: 1234567823
Utilizzatore 4 non ha trovato blocchi liberi
Utilizzatore 5 non ha trovato blocchi liberi
Utilizzatore 6 non ha trovato blocchi liberi
.
.
.
Utilizzatore 1 ha liberato il blocco 0
                    Stato della memoria: 0234567823
Utilizzatore 7 ha allocato il blocco 0
                    Stato della memoria: 7234567823
Utilizzatore 3 ha liberato il blocco 9
Utilizzatore 5 ha liberato il blocco 4
Utilizzatore 2 ha liberato il blocco 8
                    Stato della memoria: 7234067800
.
.
.
.
.
Utilizzatore 7 ha liberato il blocco 0
                    Stato della memoria: 0044000000
Utilizzatore 4 ha liberato il blocco 3
                    Stato della memoria: 0040000000
Utilizzatore 4 ha liberato il blocco 2
                    Stato della memoria: 0000000000
```

Per motivi di sintesi non sono state riportate tutte le scritte generate dall'esecuzione dal programma. Si noti che a circa metà del tracciato alcuni utilizzatori hanno trovato la memoria completamente piena, cioè nessun blocco libero risulta disponibile.
Ogni esecuzione del programma fornisce sequenze di allocazione e deallocazione diverse, a seconda di come i vari thread vengano schedulati dal sistema.
Per verificare il comportamento del programma, si possono provare diverse esecuzioni con diverse quantità di utilizzatori.

Si osservi anche che:

- non si può dare a un *Utilizzatore* l'identificativo 0, in quanto questo valore ha un significato particolare per l'indice dei blocchi;
- se si vogliono utilizzare più di 9 utilizzatori occorre modificare la modalità di visualizzazione del metodo *stampaMemoria*, in quanto non basta più una sola cifra per identificare l'utilizzatore e quindi l'indice dei blocchi deve contenere numeri a una cifra e numeri a due cifre.