LEZIONE N.7
LPR-THREADS: SCHEDULING E SINCRONIZZAZIONE

21/04/2008

Laura Ricci

STATI DI UN THREAD

- initial state: è stato creato l'oggetto thread, ma l'esecuzione del thread non è ancora stata avviata (non è stata eseguita la start())
- runnable: il thread è pronto per essere eseguito, cioè è stata eseguita la start( ).
- running:il thread è in esecuzione
- blocked: il thread è in attesa di qualche evento
  - ha eseguito un metodo come sleep(), join(), wait(),..
  - è in attesa del termine di una operazione di I/O: esempio ha tentato di leggere da un socket su cui non sono presenti dati
  - ha tentato di acquisire una lock posseduta da un altro thread
- exiting: l'esecuzione metodo run() è terminata

THREAD SCHEDULING

- JAVA non definisce una politica vincolante riguardo la schedulazione dei threads
- JAVA 'suggerisce' di schedulare i threads secondo la loro priorità
- Esecuzioni dello stesso programma multi-threaded su diverse JVM possono produrre diversi interleaving dei threads
- La correttezza del programma non deve dipendere da un particolare ordinamento dell'esecuzione dei threads
- Implementazioni esistenti
  - green threads
  - implementazioni basate su native threads

GREEN THREADS (USER LEVEL)

- La gestione dei threads è completamente demandata alla JVM
- La JVM emula l'ambiente multithreaded senza utilizzare le funzionalità del sistema operativo ospite
- Il programma JAVA multithreaded 'è visto' dal sistema operativo come un unico processo
- La JVM mantiene per ogni thread un descrittore in cui salve il contenuto dello stack associato al thread, il valore del PC....
- La JVM è responsabile del cambiamento di contesto tra i threads: sceglie i threads da mandare in esecuzione, salva lo stato di un thread quando la sua esecuzione viene interrotta, carica lo stato del nuovo thread,.....
- Dal punto di vista del sistema operativo esiste un singolo processo che esegue la JVM. Il fatto che il codice eseguito emuli diversi theads non è visibile all'esterno della JVM.

POLITICHE DI SCHEDULAZIONE

- Green threads: implementano una politica di schedulazione basata su priorità e prerilascio
  - priorità: assegnate dal programmatore o dalla JVM
  - prerilascio: quando viene schedulato un thread T1 di priorità più alta del thread in esecuzione T2, l'esecuzione di T2 è interrotta e viene eseguito T1
- JAVA non richiede che venga implementato time slicing. Infatti le implementazioni della JVM basate su green threads non utilizzano time slicing
  - Se un thread non si blocca in atteso di un evento (es: compute intensive thread) e non viene interrotto da threads a più alta priorità, rilascia la CPU solo quando il metodo run() è completata
  - I threads possono essere eseguiti, in certi casi, in modo sequenziale

POLITICHE DI SCHEDULAZIONE

- Green threads: si può utilizzare il metodo yeld() per implementare il time slicing dei threads
  - Un thread interrompe autonomamente la propria esecuzione, dopo un certo intervallo di tempo
- Green threads: Utilizzati nelle prime versioni della JVM, attualmente poco utilizzati
- In ambiente windows ed per le nuove versioni del kernel LINUX si utilizzano approcci alternativi
  - Approccio basato su native threads (windows, LINUX, Solaris,...): mapping dei threads JAVA sui thread del sistema operativo
  - Lo scheduling dei threads è fortemente influenzato dalla politica di scheduling del sistema operativo su cui viene eseguita la JVM

NATIVE THREAD SCHEDULING

- Il sistema di schedulazione basato su prerilascio e priorità deve essere 'mappato' sui meccanismi offerti dal sistema operativo
- Le politiche di schedulazione del sistema operativo in genere implementano time slicing
- Gestione delle priorità:
  - I livelli di priorità definiti da JAVA devono essere mappati su un numero inferiore di livelli di priorità.
  - threads a cui è stato assegnato un diverso livello di priorità dal programmatore JAVA possono essere eseguiti con lo stesso livello di priorità
  - la priorità reale di un thread è ottenuta mediante formule complesse che tengono in considerazione, oltre alla priorità assegnata in JAVA, di altri fattori (es: tempo di attesa della CPU,.....)

ESERCIZIO

Scrivere un programma che attivi k threads. Ogni thread calcola l'ennesimo numero della successione di Fibonacci, con n molto grande.

Ogni thread, al momento dell'attivazione, visualizza il tempo restituito dall'orologio di sistema, effettua la computazione, poi visualizza di nuovo il tempo letto dall'orologio di sistema ed il tempo impiegato per eseguire la computazione richiesta. Devono essere provate diverse versioni del programma

1) ai threads non viene/viene assegnata una priorità,
2) ogni thread rilascia/non rilascia autonomamente il processore (metodo yeld( ))

Eseguire il programma sul computer di casa ed in laboratorio ed illustrare eventuali differenze riscontrate.

BLOCCHI SINCRONIZZATI

- Sincronizzazione di un blocco di codice

```
synchronized (obj)
     { // blocco di codice
     }
```

- L'oggetto obj può essere quello su cui è stato invocato il metodo che contiene il codice (this) oppure un altro oggetto
- Il thread che esegue il blocco sincronizzato deve acquisire la lock su l'oggetto obj
- La lock viene rilasciata nel momento in cui il thread termina l'esecuzione del blocco (es: return, throw, esecuzione dell'ultima istruzione del blocco)
- La lock non viene rilasciata nel caso in cui si invochi un altro metodo all'interno del blocco di codice

BLOCCHI SINCRONIZZATI

```
synchronized (obj)
        { // blocco di codice ...
        }
```

- L'oggetto obj può essere quello su cui è stato invocato il metodo (this) oppure un altro oggetto
- I seguenti frammenti di codice sono equivalenti

```
public synchronized void set(int x)
  {this.x = x}
```

```
public void set(int x)
        synchronized(this)
         {this.x=x }
```

BLOCCHI SINCRONIZZATI

- Utilizzo:
  - Ridurre la lunghezza di una sezione critica
  - Rendere indivisibile un frammento di codice che invoca due o più metodi synchronized
- L'esempio presentato nei lucidi successivi mostra in modo operativo il vantaggio di diminuire la lunghezza di una sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public void setValues (int x, double r)
    double tempA= // questa elaborazione richiede molto tempo...
    double tempB= // questa elaborazione richiede molto tempo...
    synchronized ( this ) {
        A = tempA;
        B = tempB; }
```

- L'elaborazione che richiede molto tempo viene eseguita in modo concorrente (utile se i threads vengono eseguiti in parallelo)
- i risultati vengono assegnati a variabili locali
- la sezione critica contiene solo l'aggiornamento delle variabili di istanza del metodo

OTTIMIZZAZIONE DI SEZIONI CRITICHE

La classe Pair implementa una coppia di valori interi.

I threads devono aggiornare i componenti della coppia una coppia in modo atomico . Si definiscono due diverse implementazioni diverse della modifica

```
public class Pair {
    private int x,y;   public Pair (int x, int y)  {  this.x=x;   this.y=y;  };
    public synchronized void increment_syn_method()
    {x++;  y++;
    //questa sleep simula una computazione C molto costosa
    try{Thread.sleep(100);} catch(Exception e){};   }
    public void increment_syn_block()
    {synchronized(this)
             {x++;  y++;}
    try {Thread.sleep(100); } catch(Exception e){} }  }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

La classe Pair definisce inoltre una variabile intera accesscount che deve essere incrementata in modo mutuamente esclusivo rispetto agli aggiornamenti effettuati sulla coppia di valori, ma che può essere effettuata in modo concorrente rispetto all'operazione C.

```
private int accesscount=0;
public synchronized void accessincrement( ){accesscount++;};
public synchronized int accessreturn( ){return accesscount;};
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class Thread_Syn_Method  extends Thread
{Pair p;
public Thread_Syn_Method(Pair p) {this.p=p;};
  public void run(   ){
   while (true)  {p.increment_syn_method();   }}}

public class Thread_Syn_Block extends Thread{
 Pair p;
public Thread_Syn_Block(Pair p) {this.p=p;};
  public void run(   ){
while (true)  {p.increment_syn_block();  }}}
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class CountAccess extends Thread {
        Pair p;
        public CountAccess(Pair p) {this.p=p;}
        public void run(){
        while (true)
            p.accessincrement( );  }  }
```

OTTIMIZZAZIONE SEZIONI CRITICHE

```
public class MainPair {  public static void main(String args[])
    {Pair syn_method_pair= new Pair(1,2);
    Pair syn_block_pair= new Pair(3,4);
    new Thread_Syn_Method(syn_method_pair).start();
    new CountAccess(syn_method_pair).start();
    new Thread_Syn_Block(syn_block_pair).start();
    new CountAccess(syn_block_pair).start();
    try{Thread.sleep(200);}catch(Exception e){}
    System.out.println(
    "Accessi con blocco sincronizzato "+syn_block_pair.accessreturn( )+
    "     Accessi con metodo sincronizzato
      "+syn_method_pair.accessreturn( ));  }}
```

OTTIMIZZAZIONE SEZIONI CRITICHE

Risultati di alcune esecuzioni:

Accessi con blocco sincronizzato 2258106 Accessi con metodo sincronizzato 2843

Accessi con blocco sincronizzato 2242580 Accessi con metodosincronizzato 2

Accessi con blocco sincronizzato 3749759 Accessi con metodo sincronizzato 1252

Conclusioni:

- il thread che accede all'oggetto coppia con blocco sincronizzato (invece che con un metodo sincronizzato) consente un maggior numero di accessi al thread Countaccess, che viene eseguito concorrentemente
- La versione che utilizza il blocco sincronizzato aumenta il numero di accessi concorrenti effettuati sull'oggetto coppia
- NOTA BENE: La differenza nel numero di accessi dipende ovviamente dal fatto che il programma simula la cumputazione costosa con una sleep()
- In un caso reale: la differenza minore poichè il thread rilascerebbe la CPU solo in seguito allo scadere del suo time slice

COLLEZIONI SINCRONIZZATE

- JAVA Collections: strutture dati predefinite incluse nel package java.util

  a partire da JAVA 1.2
- Alcuni esempi: HashTables, Vectors
- Synchronized Collections: Definiscono strutture dati thread safe, cioè garantiscono che lo stato della struttura risulti corretto anche nel caso in cui la struttura venga acceduta in modo concorrente da più threads
- Thread Safety: la struttura dati viene incapsulata e ogni metodo pubblico viene sincronizzato
- Se l'operazione che il client (il thread che utilizza la collezione) è composta da una serie di operazioni elementari, il client deve spesso implementare ulteriori sincronizzazioni.
- Si possono definire blocchi di codice sincronizzati che incapsulano più operazioni elementari effettuate su una collezione sincronizzata

COLLEZIONI SINCRONIZZATE

```
import java.util.*;
public class threadremover extends Thread{
        Vector v;
        public threadremover(Vector v){this.v=v;}
            public void run( ) {
            int lastIndex = v.size(   ) - 1;
        Object o=v.remove(lastIndex); }   }
```

- il thread threadremover elimina da un Vector l'elemento che si trova nella sua ultima posizione
- Le operazioni size( ) e remove( ) sono definite come operazioni sincronizzate

COLLEZIONI SINCRONIZZATE

```
import java.util.Vector;
public class threadget extends Thread{
    Vector v;
    public threadget(Vector v){this.v=v;}
        public void run( ){
        int lastIndex = v.size( ) -1;
        try {Thread.sleep(5000);}  catch(InterruptedException e){  };
        v.get(lastIndex);   } }
```

- Il thread threadget restituisce l'ultimo elemento del Vector
- La sleep( ) è stata introdotta per costringere threadget a rilasciare il processore a threadremover
- In questo modo si ottiene un interleaving scorretto tra i due threads ed il programma segnala la seguente eccezione

COLLEZIONI SINCRONIZZATE

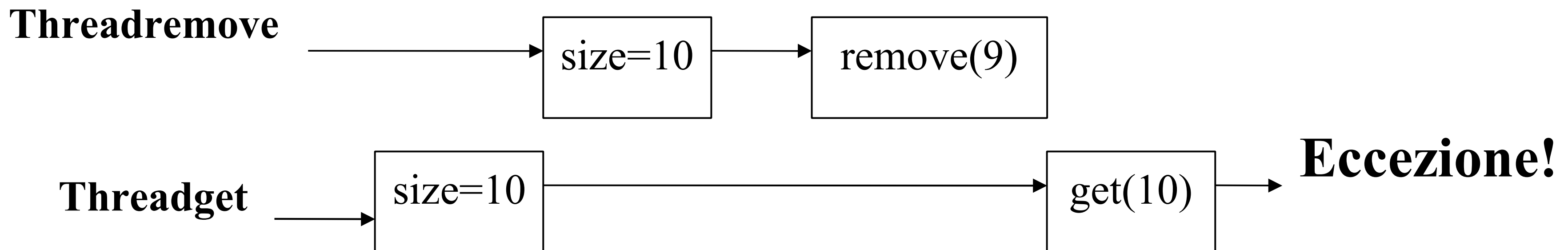
Eccezione sollevata dalla esecuzione del programma:

Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException:
Array index out of range: 9

at java.util.Vector.get(Unknown Source)

at threadget.run(threadget.java:10)

L'eccezione viene sollevata a causa del seguente interleaving (scorretto)



COLLEZIONI SINCRONIZZATE

```
import java.util.Vector;
public class threadget extends Thread{
    Vector v;
    public threadget(Vector v){this.v=v;}
    public void run ( ){
        synchronized(v){
        int lastIndex = v.size( ) -1;
        //try {Thread.sleep(5000);}catch(InterruptedException e){};
        Object x=v.get(lastIndex);
        System.out.println("oggetto prelevato"+x);
        }}}
```

COLLEZIONI SINCRONIZZATE

Versione corretta del threadremover

```
import java.util.*;
public class threadremover extends Thread{
      Vector v;
      public threadremover(Vector v){this.v=v;}
      public void run(){
            synchronized(v){
            int lastIndex = v.size()-1;
            Object o=v.remove(lastIndex);} }}
```

BLOCCHI DI CODICE SYNCHRONIZED

```
public class Coda { //....
    public synchronized boolean isSpaceAvailable() { //....
    public synchronized void add(Item i){ //....
    public synchronized Item remove( ){ //....
    ......
    Coda c =//....
    syncronized (c)
        { if (c.isSpaceAvailable()) {
                                        c.add(i); }}
```

- Se un thread entra nel blocco sincronizzato acquisisce la lock su c. Quando entra in un metodo sincronizzato, possiede già la lock e non la deve quindi richiedere nuovamente.

METODI STATICI SINCRONIZZATI

JAVA supporta due tipi di variabili

- Variabili di classe (statiche) : variabili che memorizzano lo stato di tutti gli oggetti di quella classe
- Variabili di istanza : variabili che memorizzano lo stato di un singolo oggetto della classe

Esempio:

Classe :	Conto Corrente,
Variabile di classe:	Numero progressivo di conto corrente
Variabili di istanza:	Saldo del Conto Corrente

- La sincronizzazione sulle variabili di istanza è implementata mediante le lock associate agli oggetti
- Problema: come sincronizzare gli accessi alle variabili di classe

SINCRONIZZARE METODI STATICI

- JAVA crea implicitamente un oggetto O per ogni classe definita nel programma
  - lo stato di O è definito dalle variabili statiche
  - i metodi di O sono i metodi dichiarati statici nella classe
- Class level lock: lock associata all'oggetto O e condivisa da tutte le istanze di quella classe.
- Ad ogni classe viene associata una class level lock CLL.
- Ogni metodo statico e sincronizzato, deve acquisire la CLL lock, prima di iniziare la sua esecuzione
- La CLL lock consente di implementare la mutua esclusione sulle variabili statiche
- Due metodi statici e sincronizzati non possono accedere contemporaneamente alle variabili statiche della classe

SINCRONIZZARE METODI STATICI

```
public class StaticNeedSync {
            private static int nextSerialNum = 10001;
            public static int getNextSerialNum(){
            int sn=nextSerialNum;
            try{Thread.sleep(1000);}
            catch(InterruptedException x){     }
            nextSerialNum++;
            return sn;
        }
```

SINCRONIZZARE METODI STATICI

```
public static void main (String args[ ]){
        Runnable r = new Runnable () {
                public void run() {
                String Name=Thread.currentThread().getName();
                System.out.println(Name+"SerialNum"+getNextSerialNum());}};
       Thread threadA = new Thread (r, "threadA");
        threadA.start();
        try{Thread.sleep(1500);}
        catch(InterruptedException x){ }
```

SINCRONIZZARE METODI STATICI

```
Thread threadB = new Thread (r, "threadB");
threadB.start();
try{Thread.sleep(500);}
catch(InterruptedException x){ }
Thread threadC= new Thread (r, "threadC");
threadC.start();
try{Thread.sleep(2500);}
catch(InterruptedException x){ }
Thread threadD= new Thread (r, "threadD");
threadD.start( ); } }
```

SINCRONIZZARE METODI STATICI

- Un possibile output del programma:

  threadA SerialNum 10001

  threadB SerialNum 10002

  threadC SerialNum 10002

  threadD SerialNum 10004

- Se modifico il programma

  public static synchronized int getNextSerialNum()

  l'output sarà qullo corretto

  threadASerialNum10001

  threadBSerialNum10002

  threadCSerialNum10003

  threadDSerialNum10004

SINCRONIZZARE METODI STATICI

```
public class classlevellock {
    private static int nextSerialNum=10001;
    public static synchronized int getNextSerialNum( ){
        nextSerialNum++;
        String Name = Thread.currentThread( ).getName();
        System.out.println(Name+"SerialNum"+nextSerialNum);
        try{Thread.sleep(1000);}
        catch(InterruptedException x){};
        return nextSerialNum;}
    public synchronized void setNum( ){
        nextSerialNum ++;}
```

SINCRONIZZARE METODI STATICI

```
public static void main (String args[  ]){
        Runnable r = new Runnable ()
                { public void run( ){String Name=Thread.currentThread().getName();
                        System.out.println(Name+"SerialNum"+getNextSerialNum());}};
        Runnable r1 = new Runnable ( )
                {public void run(){
                classlevellock ans = new classlevellock();
                ans.setNum( ); }};
Thread threadA = new Thread (r, "threadA"); threadA.start();
try{Thread.sleep(1500);} catch(InterruptedException x){}
Thread threadB = new Thread (r, "threadB"); threadB.start();
try{Thread.sleep(100);} catch(InterruptedException x){}
Thread threadF= new Thread (r1, "threadF");threadF.start();}}
```

SINCRONIZZARE METODI STATICI

Output del Programma

threadA SerialNum 10002

threadA SerialNum 10002

threadB SerialNum 10003

threadB SerialNum 10004

Cosa è accaduto?

- i metodi getNextSerialNum e setNum sono entrambi sincronizzati, ma accedono a lock diverse
- I metodi sono stati eseguiti in interleaving

SINCRONIZZARE METODI STATICI

- E' possibile anche utilizzare esplicitamente un lock a livello di classe (CLL)

```
synchronized ( ClassName.class )
        { // all'interno di questo blocco modifico i campi statici
        }
```

- Un metodo sincronizzato, ma non statico, può accedere ai campi statici senza richiedere la class level lock

- E' compito del programmatore assicurare la correttezza dell'applicazione

METODI SYNCHRONIZED

- Importante: la lock( ) è associata all'istanza di un oggetto, non al metodo o alla classe (a meno di metodi statici !!)
- Diversi metodi sincronizzati invocati sull'istanza dello stesso oggetto competono per la stessa lock(), quindi risultano mutuamente esclusivi
- Metodi sincronizzati che operano su istanze diverse dello stesso oggetto possono essere eseguiti in modo concorrente
- All'interno della stessa classe possono comparire contemporaneamente metodi sincronizzati e non
  - I metodi non sincronizzati possono essere eseguiti in modo concorrente
  - In ogni istante, su un certo oggetto, possono essere eseguiti concorrentemente piu' metodi sincronizzati e solo uno dei metodi sincronizzati della classe

CODE BLOCCANTI (O SINCRONIZZATE)

- Code bloccanti: introdotte in JAVA 5 come supporto per il paradigmi computazionali di tipo produttore/consumatore
- Code sincronizzate:comportamento di base
  - inserimento: aggiunge un elemento infondo alla coda, se la coda non è piena, altrimenti blocca il thread che ha invocato l'operazione.
  - rimozione:elimina il primo elemento della coda, se questo esiste, altrimenti blocca il thread che ha invocato l'operazione
- L'interfaccia java.util.concurrent.BlockingQueue definisce questo tipo di code
- Sono stati definiti
  - diverse varianti della coda
  - metodi caratterizzati da diversi comportamenti per l'inserzione/rimozione di elementi dalla coda

CODE BLOCCANTI: TIPI DEFINITI

Classi che implementano l'interfaccia BLockingQueue

- LinkedBlockingQueue: non si definisce un limite superiore alla capacità della coda
- ArrayBlockingQueue: definisce un numero fisso di posizioni della coda
- SynchronousQueue: non è una vera e propria coda, in quanto non ha capacità di memorizzazione. Mantiene solo le code per la gestione dei threads che aspettano in attesa di produrre/consumare elementi. Risparmia il tempo necessario per la bufferizzazione degli elementi prodotti
- PriorityBlockingQueue:implementa una coda a priorità

CODE BLOCCANTI: METODI

Metodi che generano un'eccezione quando si tenta di aggiungere un elemento ad una coda piena o di estrarre un elemento da una coda vuota:

add, remove, element (element restituisce l'elemento in testa, e non lo rimuove)

```
import java.util.concurrent.*;
public class provaqueue {
public static void main (String args[])
{ BlockingQueue <String> queue = new ArrayBlockingQueue <String>(1);
queue.add("el1"); queue.add("el2");   }}
```

```
Exception in thread "main" java.lang.IllegalStateException: Queue full
at java.util.AbstractQueue.add(Unknown Source)
at java.util.concurrent.ArrayBlockingQueue.add(Unknown Source)
at procacrawler.main(procacrawler.java:7)
```

CODE BLOCCANTI: METODI

Metodi che generano un'eccezione quando si tenta di aggiungere un elemento ad una coda piena o di estrarre un elemento da una coda vuota:

add,remove, element (element restituisce l'elemento in testa, e non lo rimuove)

```
import java.util.concurrent.*;
public class provaqueue {
public static void main (String args[])
    {BlockingQueue <String> queue = new ArrayBlockingQueue <String>(1);
    queue.add("el1"); queue.remove(); queue.remove();
    }}
```

```
Exception in thread "main" java.util.NoSuchElementException
at java.util.AbstractQueue.remove(Unknown Source)
at procacrawler.main(procacrawler.java:9)
```

CODE BLOCCANTI: METODI

Metodi che restituiscono una segnalazione del fallimento dell'operazione di inserzione/estrazione: offer,poll, peek

```
import java.util.concurrent.*;
public class provaqueue {
public static void main (String args[])
    {BlockingQueue <String> queue = new ArrayBlockingQueue <String>(1);
    boolean success=false;
    success=queue.offer("el1");  System.out.println(success);
    try{success=queue.offer("el2", 100, TimeUnit.MILLISECONDS);
    }catch (InterruptedException e){ }
    System.out.println(success);}}
```

Ouput del programma true, false

CODE BLOCCANTI:METODI

Metodi che bloccano il thread nel caso del fallimento della operazione di inserzione/rimozione di un elemento

put, take

```
import java.util.concurrent.*;
public class provaqueue {
public static void main (String args[])
    {BlockingQueue <String> queue = new ArrayBlockingQueue <String>(1);
    try  {queue.put("el1");} catch (InterruptedException e){ }
    try{ queue.put("el2");} catch (InterruptedException e){ }
        }}
```

la seconda put blocca il thread. Il thread viene risvegliato quando l'inserzione/rimozione può essere effettuata

ESERCIZIO

Si vuole realizzare un programma Crawler che analizza tutti i files presenti in una directory specificata e nelle sue sottodirectory e visualizza tutte le righe presenti in tali file che contengono una determinata parola chiave P.

Il programma deve attivare due thread,

- un thread produttore che enumera tutti i file e inserisce un riferimento ad ogni file individuato in una coda bloccante
- un consumatore che estrae i riferimenti ai file dalla coda e, per ognuno di essi, visualizza tutte le righe che contengono la parola chiave P.

I due thread si scambiano i dati mediante una coda bloccante. Scegliere il tipo di coda bloccante ritenuto più opportuno.

VARIABILI LOCALI DI METODI CONCORRENTI

- E' possibile che diversi thread invochino metodi su uno stesso oggetto in modo concorrente
- Le variabili locali dei metodi vengono allocate sullo stack di ogni thread
- Se non si usano opportuni meccanismi di sincronizzazione è possibile che lo stato dell'oggetto risulti inconsistente

VARIABILI LOCALI DI METODI CONCORRENTI

```
public class AccessiConcorrenti  {
    private String objID;
    public AccessiConcorrenti(String objID) { this.objID = objID; }
    public void Accedi (int val) {
        int num=val;
        System.out.println(Thread.currentThread().toString()+"variabile
                                locale num=" + num);
        try { Thread.sleep(2000); } catch ( InterruptedException x ) { };
        num = num + objID.length();
        System.out.println(Thread.currentThread().toString()+"variabile
                                locale num=" + num);
```

VARIABILI LOCALI DI METODI CONCORRENTI

```
public static void main(String[ ] args) {
   final AccessiConcorrenti ac = new AccessiConcorrenti("obj1");
   Runnable runA = new Runnable (){
                public void run() { ac.Accedi(3);}};
   Thread threadA = new Thread(runA, "threadA");
   threadA.start();
   Runnable runB = new Runnable(){
                public void run() {ac.Accedi(7);};};
   Thread threadB = new Thread(runB, "threadB");
   threadB.start();}
}
```

OUTPUT DEL PROGRAMMA

```
Thread[threadB,5,main]variabile locale num=7

Thread[threadA,5,main]variabile locale num=3

Thread[threadB,5,main]variabile locale num=11

Thread[threadA,5,main]variabile locale num=7
```

I METODI SYNCHRONIZED

- La parola chiave synchronized nella intestazione di un metodo ha l'effetto di serializzare gli accessi al metodo

```
public synchronized void Accedi (int val)
```

- Se un thread sta eseguendo il metodo Accedi, nessun altro thread eseguirà lo stesso codice finchè il primo thread non termina l'esecuzione del metodo
- Implementazione:
  - il metodo M synchronized appartiene alla classe C
  - ad ogni istanza di C (oggetto O) viene associata una lock( ) L
  - quando un thread T invoca M su O, T tenta di acquisire L, prima di iniziare l'esecuzione di M. Se T non acquisisce L, si sospende

OUTPUT DEL PROGRAMMA

Se rendiamo synchronized il metodo Accedi…..

Thread[threadA,5,main]variabile locale num=3

Thread[threadA,5,main]variabile locale num=7

Thread[threadB,5,main]variabile locale num=7

Thread[threadB,5,main]variabile locale num=11

I METODI SYNCHRONIZED

```
public static void main(String[ ] args) {
  final AccessiConcorrenti ac1 = new AccessiConcorrenti("obj1");
  final AccessiConcorrenti ac2 = new AccessiConcorrenti("obj2");
  Runnable runA = new Runnable (){
                public void run( ) { ac1.Accedi(3);}};
  Thread threadA = new Thread(runA, "threadA");
  threadA.start();
  Runnable runB = new Runnable(){
                public void run( ) {ac2.Accedi(7);};};
    Thread threadB = new Thread(runB, "threadB");
  threadB.start( );}
}
```

OUTPUT DEL PROGRAMMA

- il metodo Accedi è synchronized
- I due threads invocano entrambi Accedi, ma su oggetti diversi(ac1, ac2)

```
Thread[threadA,5,main]variabile locale num=3
Thread[threadB,5,main]variabile locale num=7
Thread[threadA,5,main]variabile locale num=7
Thread[threadB,5,main]variabile locale num=11
```