Processi e Thread

Processi e Thread: Il Caso Java.

Walter Cazzola
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano.

e-mail: cazzola@disi.unige.it

Processi

Def. Un processo è una sequenza di codice in esecuzione assieme al suo ambiente.

È un'attività controllata da un programma che si svolge su un processore, quindi, un programma può essere composto da più processi.

Nota: I processi sono l'unità di esecuzione in ambiente distribuito.

Processi: Caratteristiche

Distribuiti

Sono eseguiti su computer e ambienti diversi.

Autonomi

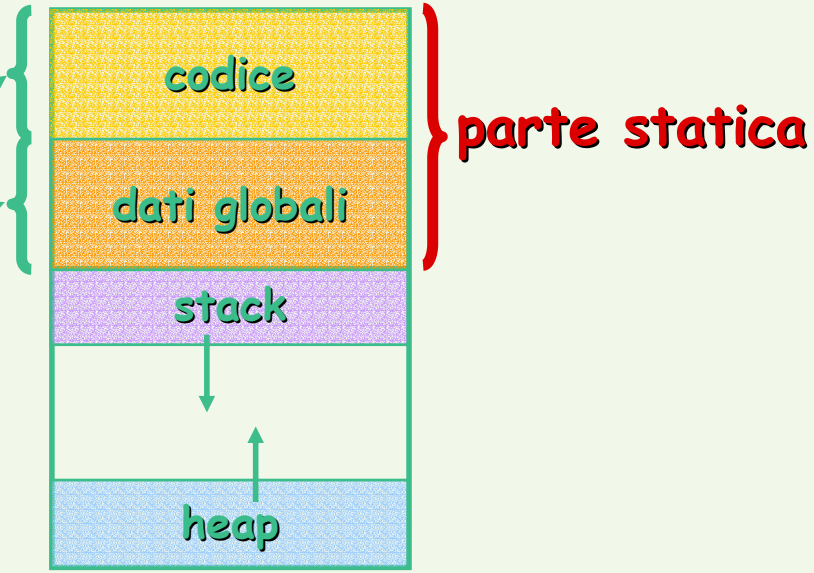
Ogni processo ha un proprio ciclo di vita indipendente da quello degli altri processi. Durante il proprio ciclo di vita ogni processo porta a termine il proprio compito.

Interagenti

I processi durante la propria computazione necessitano di servizi offerti da altri processi. Se ci sono molte interazioni tra due processi si parla di accoppiamento forte, altrimenti di accoppiamento lasco.

Struttura dei Processi in Memoria

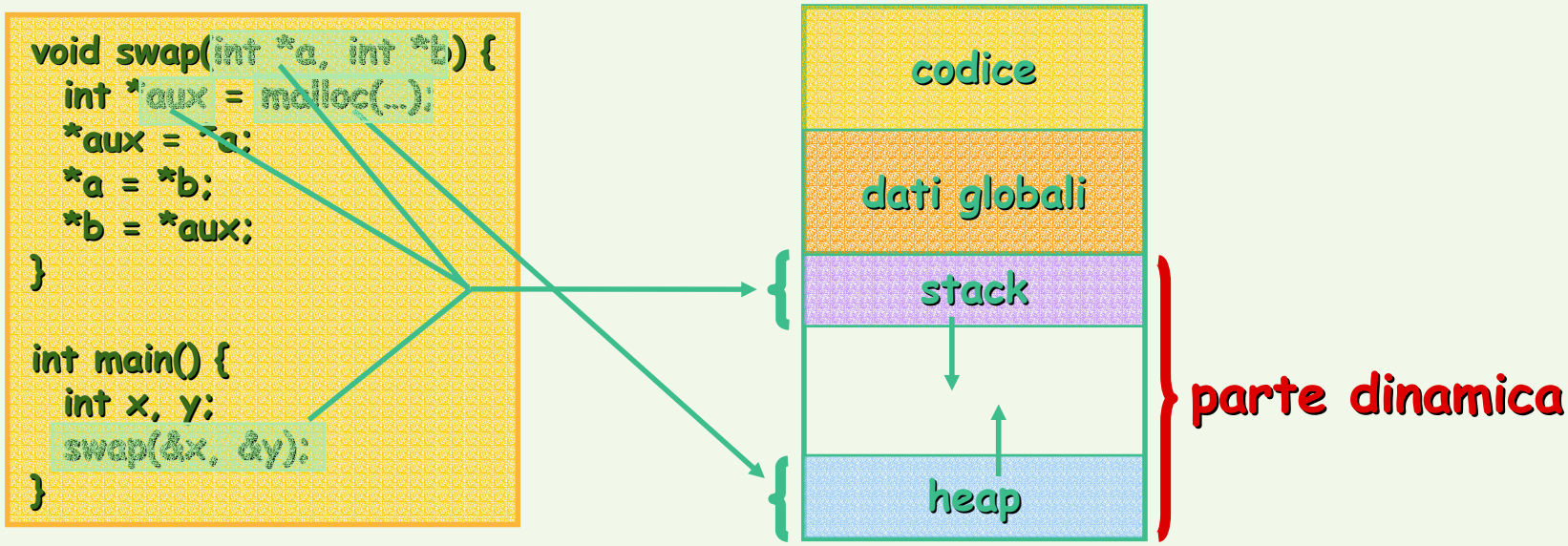
```
void swap(int *a, int *b) {
  int *aux = malloc(…);
  *aux = *a;
  *a = *b;
  *b = *aux;
}

int main() {
  int x, y;
  swap(&x, &y);
}
```



Un processo è composto da una parte statica:

- Area Codice
- Area Dati Globali

Struttura dei Processi in Memoria



Un processo è composto da una parte dinamica:

- Stack, ed
- Heap

Thread: Definizione

Cos'è un Thread?

- Un thread (o processo leggero) è una sequenza di istruzioni di un programma in corso di esecuzione.

Praticamente il flusso esecutivo di un processo viene scomposto in più flussi concorrenti.

I Threads di uno stesso processo condividono l'area dati e codice. È, pertanto, necessaria una sincronizzazione nell'accesso ai dati globali.

Vantaggio: il cambio di contesto tra threads è più veloce che tra processi.

Threads (Lifecycle)

Un thread viene creato ed inizia la propria esecuzione

- il nuovo thread avrà una propria identità ed il SO lo tratterà come qualsiasi altro processo, applicandogli le proprie politiche di schedulazione. Il thread è running.

Può mettersi in attesa del verificarsi di una condizione

- il passaggio dallo stato di dormiente (sleeping o waiting) a quello di attivo (running) è regolato dal SO tramite segnali.

Conclude la propria computazione

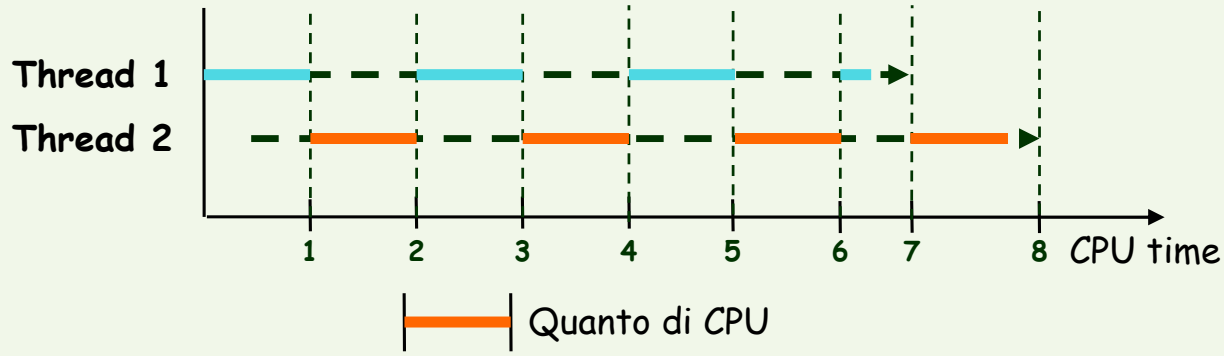
- il flusso esecutivo torna a confluire in quello che lo ha originato. Il thread è stopped.

Thread e Concorrenza

Multitasking permette di eseguire diversi compiti entro un unico programma.

Su un computer "sequenziale" i thread di un programma condividono l'unica CPU del computer.

Time slicing: i thread si alternano in esecuzione per una fetta (quanto) del tempo di CPU.



Scope delle Variabili nei Thread

Variabili locali ad un metodo sono locali ai thread

- Modifiche ad una variabile locale ad un thread non sono visibili agli altri thread.
- Variabili locali - variabili dichiarate in un metodo.

Nota: non sono visibili perché i thread non condividono lo stack.

Variabili globali sono condivise da tutti i thread

- Variabili globali: campi statici o di istanza della classe.

Nota: sono condivise perché non sono allocate sullo stack.

Threads in Java

Il multi-threading permette di scrivere server che gestiscono più richieste contemporaneamente.

Java ha i thread nativi

- i thread sono supportati a livello di linguaggio.

Java ha la classe Thread per gestire i thread

- C'è una corrispondenza tra i thread forniti dal linguaggio e la classe Thread.

I thread sono controllati tramite i metodi della classe Thread.

La Classe Thread

È una rappresentazione a run-time di un thread

- tramite un'istanza di questa classe è possibile manipolare il thread associato.

I metodi principali sono:

- Start
- Stop
- Run
- Interrupt
- Join

Thread in Java: la Classe Thread

```
public class Thread extends Object implements Runnable {
   // costruttori
      public Thread();
      public Thread(Runnable target);
      public Thread( String name );
   // metodi di classe
      public static native void sleep(long ms) throws InterruptedException;
      public static native void yield();
   // metodi
      public final String getName();
      public final int getPriority();
      public void run();                          // unico metodo dell'interfaccia Runnable
      public final void setName();
      public final void setPriority();
      public synchronized native void start();
      public final void stop();
}
```

I Metodi della Classe Thread

void run()

- Viene chiamato automaticamente quando il thread è attivato. Deve essere ridefinito per costruire thread con comportamenti particolari.

void start()

- Permette di attivare un thread. Una chiamata a questo metodo chiama a sua volta il metodo run().

```
Thread execution = new Thread(anObject);
execution.start();
```

I Metodi della Classe Thread

void stop()

- Permette di fermare un thread. Un thread stoppato non può successivamente essere fatto ripartire. L'uso di questo metodo viene sconsigliato perché può lasciare un oggetto in uno stato inconsistente.

void interrupt()

- Permette di interrompere l'esecuzione di un thread, solo quando lo stato dell'oggetto corrispondente è consistente. Cioè quando il thread è dormiente o in attesa di un evento.

void join()

- Permette di bloccare il chiamante fintanto che il thread associato termina la propria esecuzione.

Thread: Creazione

Associo un identificatore ad ogni thread che viene stampato quando il thread parte.

- IdThread ridefinisce Thread.run().

Quando un thread parte, esegue il suo metodo run().

```
public class IdThread extends Thread {
    int num;
    public IdThread(int n) {
        num = n;
    }
    public void run() {
        System.out.print(num);
    } // run()
} // IdThread
```

```
public class IdThreads {
  public static void main(String args[]) {
      IdThread idT1, idT2, idT3;
      idT1 = new IdThread(1); idT1.start();
      idT3 = new IdThread(3); idT3.start();
      idT2 = new IdThread(2); idT2.start();
  } // main()
} // IdThreads
```

Output: i thread sono eseguiti nell'ordine con cui sono fatti partire.

132

Interfaccia Runnable

Implementare l'interfaccia Runnable è un modo alternativo per rendere una classe multi-thread.

- In questo modo una classe qualsiasi può diventare un thread.

Tramite i costruttori della classe Thread, ad una classe X che implementa l'interfaccia Runnable può essere associato un thread.

L'attivazione del metodo start() dell'oggetto thread attiverà il metodo run() della classe X.

Thread: Creazione (Runnable)

Un thread può essere creato a partire da un oggetto qualsiasi purché implementi l'interfaccia Runnable.

```
public class IdThread implements Runnable {
   int num;
   public IdThread(int n) {
      num = n;
   }
   public void run() {
      System.out.print(num);
   } // run()
} // IdThread
```

Un oggetto Runnable implementa run().

```
Thread idT1;
idT1 = new Thread(new IdThread(1));
idT1.start();
```

Crea un oggetto Runnable.

Controllo sui Thread

setPriority(int) assegna ai thread una priorità compresa tra: Thread.MIN_PRIORITY e Thread.MAX_PRIORITY.

```
public class IdPriThread extends Thread {
   private int num;

   public IdPriThread(int n) {
      num = n;
      setPriority(n);
   }

   public void run() {
     for (int k = 0; k < 2000000; k++)
       if (k % 1000000 == 0)
          System.out.print(num);
   } // run()
} // IdPriThread
```

Inizializza la priorità del thread al suo identificativo.

5544332211

i thread sono eseguiti rispettando l'ordine dato dalla priorità.

Un thread a priorità più alta interromperà l'esecuzione dei thread a priorità più bassa.

L'implementazione dei Thread è dipendente dalla piattaforma:

- un thread con priorità alta e che non rilascia mai la CPU potrebbe creare un livelock.

Thread: sleep()

Il metodo sleep() forza il thread ad attendere per un fissato periodo di tempo.

Es., la run() di IdThread modificata per stampare in ordine casuale.

```
public void run() {
  for (int k=0; k < 10; k++) {
    try {
       Thread.sleep((long)(Math.random() * 1000));
     } catch (InterruptedException e) {
       System.out.println(e.getMessage());
     }
     System.out.print(num);
  } // for
} // run()
```

attende al più 1000 millisecondi.

i thread sono eseguiti in ordine casuale.

```
14522314532143154232152423541243235415523113435451
```

La Natura Asincrona dei Thread

I thread sono asincroni, la loro durata ed il loro ordine di esecuzione è impredicibile.

Non è possibile prevedere quando un thread verrà interrotto.

Sincronizzazione

La condivisione delle variabili globali da parte di più thread può creare problemi di consistenza.

Es.

```
int balance;
void withdraw(int amount) {
  if (balance - amount ≥ 0)
     balance -= amount;
}
```

$thread_1$	$thread_2$	balance
if (balance - amount ≥ 0)		15
	if (balance - amount ≥ 0)	15
balance -= amount;		5
	balance -= amount;	-5

Sincronizzazione

Ogni oggetto ha un semaforo associato.

Ogni semaforo ha due operazioni:

- Get, e
- Release

Ogni thread quando accede ad una zona critica tenta di prendere possesso del semaforo (Get); se non ci riesce verrà messo a dormire e ci ritenterà quando un altro thread rilascerà (Release) tale semaforo.

Sincronizzazione

I semafori sono nascosti al programmatore

- una regione critica viene introdotta dallo statement:

```
Synchronized (anObject) {
        // regione critica
}
```

- si possono sincronizzare sia classi che metodi.

I thread prima di entrare nella regione critica devono ottenere il via libero dal semaforo relativo.

Nota: un thread che sta eseguendo una sezione critica e chiede di eseguirne un'altra, ottiene immediatamente l'accesso, evitando i deadlock.

Thread: Stati e Ciclo di Vita

Il ciclo di vita dei thread consiste nel passaggio in diversi stati.

Descrizione degli Stati

Ready: pronto per l'esecuzione ma in attesa della CPU.

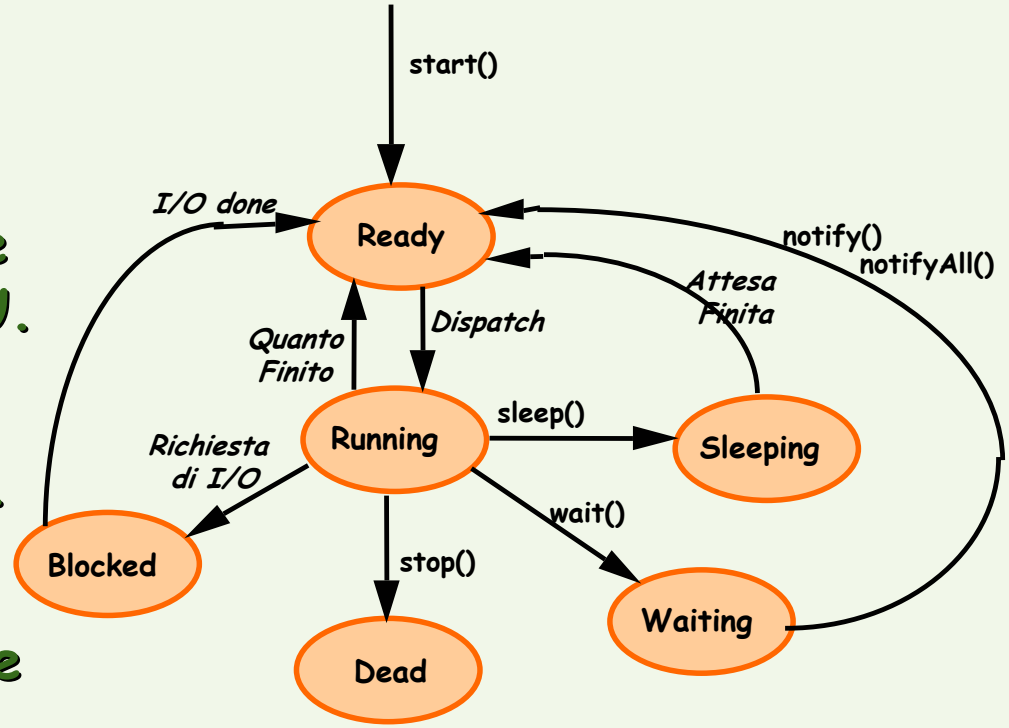
Running: in esecuzione.

Waiting: in attesa di un evento.

Sleeping: sospeso un periodo.

Blocked: in attesa di completare l'I/O.

Dead: terminato.



Legenda: *il controllo del sistema*
il controllo del programma

Studio di Caso: Thread Cooperanti

La cooperazione tra thread richiede una sincronizzazione esplicita ed un coordinamento.

Problema: considerare lo scenario "coda dal panettiere" con un commesso ed uno o più clienti. Usare un dispositivo che distribuisce i biglietti per gestire la coda.

Decomposizione del Problema:

- Bakery: programma principale, inizia i thread.
- TakeANumber: tiene traccia di chi è il prossimo.
- Clerk: serve il cliente successivo.
- Customer: attende in coda di essere servito.

Classe TakeANumber (1ª Versione)

Risorsa Condivisa: un oggetto di tipo TakeANumber verrà condiviso da molti thread.

```
class TakeANumber {
    private int next = 0;       // ultimo cliente da servire
    private int serving = 0;    // prossimo cliente da servire

    public synchronized int nextNumber() {
        next = next + 1;
        return next;
    } // nextNumber()

    public int nextCustomer() {
        ++serving;
        return serving;
    } // nextCustomer()

} // TakeANumber
```

utilizzato dai clienti.

utilizzato dal commesso.

un metodo sincronizzato non può essere interrotto.

Monitors e Mutua Esclusione in Java

Monitor: un meccanismo che assicura che solo un thread alla volta possa eseguire un metodo definito come synchronized.

Quando un metodo synchronized (es., nextNumber()) è invocato, l'oggetto corrispondente (cioè TakeANumber) è locked.

Mutua esclusione: fintanto che un oggetto è locked, nessuno dei suoi metodi synchronized può essere eseguito.

Quindi, quando un cliente sta prendendo un numero non potrà essere interrotto da nessun altro cliente.

La Classe Customer

I clienti prendono il numero successivo.

```
public class Customer extends Thread {
    private static int number = 10000;      // ID iniziale
    private int id;
    private TakeANumber takeANumber;
    public Customer( TakeANumber gadget ) {
        id = ++number;                      // ID unico per ogni cliente
        takeANumber = gadget;
    }
    public void run() {
        try {
            sleep( (int)(Math.random() * 1000 ) );
            System.out.println("Il cliente " + id + " prende il numero " +
                               takeANumber.nextNumber());
        } catch (InterruptedException e) {
            System.out.println("Errore: " + e.getMessage());
        }
    } // run()
} // Customer
```

La Classe Clerk

Il commesso continua a servire i clienti rispettando la coda.

```
public class Clerk extends Thread {
    private TakeANumber takeANumber;
    public Clerk(TakeANumber gadget) {
      takeANumber = gadget;
    }
    public void run() {
        while (true) {
            try {
                sleep( (int)(Math.random() * 50));
                System.out.println("Il commesso sta servendo il cliente"+
                  " col biglietto numero "+takeANumber.nextCustomer());
            } catch (InterruptedException e) {
                System.out.println("Errore " + e.getMessage() );
            }
        } //while
    } //run()
} // Clerk
```

La Classe Bakery

Il panettiere da il via ai thread rappresentanti i clienti e il commesso, tutti puntano allo stesso dispositivo per distribuire i biglietti, istanza di TakeANumber.

```
public class Bakery {
    public static void main(String args[]) {
        System.out.println( "Inizializzo i thread del commesso e dei clienti." );
        TakeANumber numberGadget = new TakeANumber();
        Clerk clerk = new Clerk(numberGadget);
        clerk.start();                  // un commesso
        for (int k = 0; k < 5; k++) {
            Customer customer = new Customer(numberGadget);
            customer.start();       // cinque clienti
        }
    } // main()
} // Bakery
```

Problema: Clienti Fantasma

Problema: il commesso non attende l'arrivo dei clienti per iniziare a servirli.

```
Inizializzo i thread del commesso e dei clienti.
        Il commesso sta servendo il cliente col biglietto numero 1
        Il commesso sta servendo il cliente col biglietto numero 2
        Il commesso sta servendo il cliente col biglietto numero 3
        Il commesso sta servendo il cliente col biglietto numero 4
        Il commesso sta servendo il cliente col biglietto numero 5
  Il cliente 10004 prende il numero 1
  Il cliente 10002 prende il numero 2
        Il commesso sta servendo il cliente col biglietto numero 6
  Il cliente 10005 prende il numero 3
        Il commesso sta servendo il cliente col biglietto numero 7
        Il commesso sta servendo il cliente col biglietto numero 8
        Il commesso sta servendo il cliente col biglietto numero 9
        Il commesso sta servendo il cliente col biglietto numero 10
  Il cliente 10001 prende il numero 4
  Il cliente 10003 prende il numero 5
```

Sincronizzazione: Commesso in Attesa di Clienti

Soluzione: modificare il metodo run() della classe Clerk ...

```
public void run() {
   while (true) {
      try {
        sleep((int)(Math.random() * 50));
        while (!takeANumber.customerWaiting());
        System.out.println("Il commesso sta servendo il cliente"+
              " col biglietto numero " + takeANumber.nextCustomer());
      } catch (InterruptedException e) {
        System.out.println("Exception " + e.getMessage() );
      }
   } // while
} // run()
```

il commesso attende l'arrivo di clienti da servire.

... e TakeANumber:

```
public boolean customerWaiting() {
      return next > serving;
}
```

Cooperazione dei Thread

La cooperazione tra thread DEVE essere progettata nell'algoritmo.

```
Inizializzo i thread del commesso e dei clienti.
   Il cliente 10003 prende il numero 1
      Il commesso sta servendo il cliente col biglietto numero 1
   Il cliente 10005 prende il numero 2
      Il commesso sta servendo il cliente col biglietto numero 2
   Il cliente 10001 prende il numero 3
      Il commesso sta servendo il cliente col biglietto numero 3
   Il cliente 10004 prende il numero 4
      Il commesso sta servendo il cliente col biglietto numero 4
   Il cliente 10002 prende il numero 5
      Il commesso sta servendo il cliente col biglietto numero 5
```

il servizio è successivo all'arrivo di un cliente.

Problema: Sezioni Critiche

Una sezione critica è una sezione di un thread che non dovrebbe essere interrotta.

Se noi simulassimo l'interruzione di:

```
System.out.println("Il cliente " + id + " prende il numero " +
                                    takeANumber.nextNumber());
```

Cosa succederebbe se un altro thread fosse eseguito in questo punto?

```
public void run() { // Customer.run()
    try {
        int myturn = takeANumber.nextNumber();
        sleep( (int)(Math.random() * 1000 ) );
        System.out.println("Il cliente " + id + " prende il numero " + myturn);
    } catch (InterruptedException e) {
        System.out.println("Exception " + e.getMessage());
    }
} // run()
```

Problema: Output Errato

Se ci fosse un'interruzione tra quando un cliente sta prendendo un numero e quando mostra il numero preso, noi potremmo avere:

```
Inizializzo i thread del commesso e dei clienti.
      Il commesso sta servendo il cliente col biglietto numero 1
      Il commesso sta servendo il cliente col biglietto numero 2
      Il commesso sta servendo il cliente col biglietto numero 3
   Il cliente 10004 prende il numero 4
      Il commesso sta servendo il cliente col biglietto numero 4
      Il commesso sta servendo il cliente col biglietto numero 5
   Il cliente 10001 prende il numero 1
   Il cliente 10002 prende il numero 2
   Il cliente 10003 prende il numero 3
   Il cliente 10005 prende il numero 5
```

Logicamente, il ns codice assicura che il commesso non si metta a servire finché un biglietto non è stato preso ma …

… questo output non riflette il vero stato della simulazione.

Creare una Sezione Critica

Sarà TakeANumber a mostrare lo stato del sistema.

Sezione Critica: i metodi synchronized non possono essere interrotti.

```
public class TakeANumber {
    private int next = 0;       // ultimo cliente da servire
    private int serving = 0;    // prossimo cliente da servire

    public synchronized int nextNumber(int custId) {
        next = next + 1;
        System.out.println( "Il cliente " + custId + " prende il numero " + next );
        return next;
    }

    public synchronized int nextCustomer() {
        ++serving;
        System.out.println(
                "Il commesso sta servendo il cliente col biglietto numero " + serving );
        return serving;
    }

    public synchronized boolean customerWaiting() {
        return next > serving;
    }
} // TakeANumber
```

Classi Customer e Clerk Modificate

Il commesso ed i clienti non mostrano alcunché.

```
public void run() { // Customer.run()
    try {
        sleep((int)(Math.random() * 2000));
        takeANumber.nextNumber(id);
    } catch (InterruptedException e) {
        System.out.println("Errore: " + e.getMessage() );
    }
} // run()
```

si limita a prendere un numero.

si limita a servire i clienti.

```
public void run() {     // Clerk.run()
    while (true) {
        try {
            sleep( (int)(Math.random() * 1000));
            while (!takeANumber.customerWaiting());
            takeANumber.nextCustomer();
        } catch (InterruptedException e) {
                System.out.println("Errore: " + e.getMessage());
        }
    } // for
} // run()
```

Coordinare i Thread

Output Corretto:

```
Inizializzo i thread del commesso e dei clienti.
   Il cliente 10001 prende il numero 1
      Il commesso sta servendo il cliente col biglietto numero 1
   Il cliente 10003 prende il numero 2
   Il cliente 10002 prende il numero 3
      Il commesso sta servendo il cliente col biglietto numero 2
      Il commesso sta servendo il cliente col biglietto numero 3
   Il cliente 10005 prende il numero 4
   Il cliente 10004 prende il numero 5
      Il commesso sta servendo il cliente col biglietto numero 4
      Il commesso sta servendo il cliente col biglietto numero 5
```

i clienti vengono serviti nell'ordine corretto, non importa come arrivino.

Nota: usare le sezioni critiche per rafforzare la mutua esclusione e per coordinare i thread.

Problema dell'Attesa Attiva (Busy Waiting Problem)

Nonostante i suoi thread siano coordinati nel modo corretto, la nostra simulazione del panettiere ha dell'attesa attiva (o busy waiting) nella classe Clerk:

```
public void run() {
    while (true) {
        try {
            sleep( (int)(Math.random() * 1000));
            while (!takeANumber.customerWaiting());
            takeANumber.nextCustomer();
        } catch (InterruptedException e) {
            System.out.println("Errore: " + e.getMessage());
        }
    } // for
} // run()
```

Attesa Attiva. Si aspetta in un ciclo.

Coordinare i Thread: wait() e notify()

Il metodo wait() forza un thread allo stato waiting, mentre notify() risveglia un thread spostandolo nella coda ready.

Alternativa per evitare il busy waiting: il commesso attende una notifica dal cliente.

Sia TakeANumber che Clerk dovranno essere modificate.

Modello Produttore/Consumatore: due thread condividono una risorsa, il primo la produrrà mentre il secondo la consumerà.

Classe TakeANumber Modificata

Quando un commesso invoca questo metodo, DEVE attendere l'arrivo di un cliente.

```
public synchronized int nextCustomer() {
      try {
          while (next <= serving)  {
              System.out.println("  Il commesso sta aspettando");
              wait();
          }
      } catch(InterruptedException e) {
          System.out.println("Errore: " + e.getMessage() );
      } finally {
          ++serving;
          System.out.println( "Il commesso sta servendo "+
                "il cliente col biglietto numero " + serving );
          return serving;
      }
} // nextCustomer()
```

Quando un cliente invoca questo metodo, notifica al commesso che è arrivato.

```
public synchronized int nextNumber(int custId) {
      next = next + 1;
      System.out.println(
         "Il cliente " + custId + " prende il numero "+ next );
      notify();
      return next;
} // nextNumber()
```

La Classe Clerk Modificata

Il metodo Clerk.run() è quindi semplificato:

```
public void run() {
    while (true) { // ciclo infinito
        try {
            sleep((int)(Math.random() * 1000));
            takeANumber.nextCustomer();
        } catch (InterruptedException e) {
            System.out.println("Errore: " + e.getMessage() );
        }
    } // while
} // run()
```

Output

```
Inizializzo i thread del commesso e dei clienti.
   Il cliente 10004 prende il numero 1
   Il cliente 10002 prende il numero 2
      Il commesso sta servendo il cliente col biglietto numero 1
      Il commesso sta servendo il cliente col biglietto numero 2
   Il cliente 10005 prende il numero 3
   Il cliente 10003 prende il numero 4
      Il commesso sta servendo il cliente col biglietto numero 3
   Il cliente 10001 prende il numero 5
      Il commesso sta servendo il cliente col biglietto numero 4
      Il commesso sta servendo il cliente col biglietto numero 5
      Il commesso sta aspettando
```

Il commesso sarà risvegliato quando un nuovo cliente arriverà.

Limiti del Meccanismo wait()/notify()

Sia wait() che notify() sono metodi della classe Object. Questo li abilita a lockare gli oggetti.

Il metodo wait() può essere usato in qualsiasi metodo synchronized, non solo in un thread.

Sia wait() che notify() DEVONO essere chiamati da un metodo synchronized. Altrimenti si incorre nell'eccezione: IllegalMonitorStateException.

Quando wait() è usato in un metodo synchronized, il lock su quell'oggetto è rilasciato, permettendo agli altri metodi di invocare gli altri metodi synchronized di quell'oggetto.