UNIVERSITÀ DI PISA

Appunti di Reti di Calcolatori
Laboratorio A
A.A. 2020/2021

Lezioni prof.ssa L. Ricci - prof. A. Michienzi
October 01

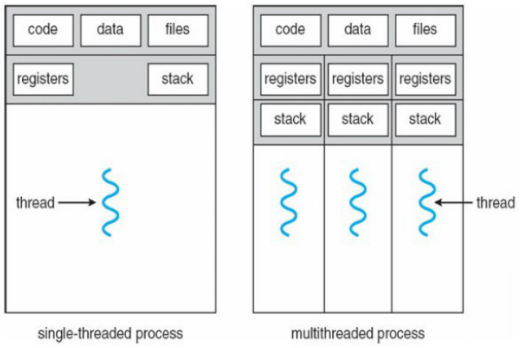
A cura di L. G. Pinta

Tali appunti sono stati pubblicati solo a fini orientativi e non rappresentano in alcun modo materiale sostitutivo al materiale consigliato o alla frequenzatione del corso e delle relative esercitazioni.

Contents

1 Introduzione ai threads

Un thread o light weight process è un flusso di esecuione all'interno di un processo. Viene definito light weight process poiché consente una commutazione di contesto (context switch) meno onerosa di un processo.



Con multitasking ci riferiamo a processi o thread:

- a livello di processo è controllato esclusivamente dal SO
- a livello thread è controllato, almeno in parte, dal programmatore

I vantaggi principali di thread multitasking rispetto a process multitasking risiede nel fatto che i thread condividono lo stesso spazio degli indirizzi e generalmente sono meno costosi in termini di context switch e comunicazioni tra threads.

A runtime i thread comportano diversi vantaggi in relazione al tipo di architettura:

- Single core: multiplexing, interleaving (tramite meccanismi di time sharing delle risorse)
- Multicore: più flussi di esecuzione eseguiti in parallelo consentono simultaneità di esecuzione

1.1 Multithreading

La gestione di funzionalità che richiedono esecuzione simultanea è realizzabile tramite una decomposizione del programma in thread. Ciò implica una modularizzazione della struttura dell'applicazione e consente un aumento della responsiveness.

Ciò consente di catturare la struttura dell'applicazione semplificandola in componenti interagenti e ogni componente viene gestita da thread distinti.

L'utilizzo del multithrading tuttavia comporta la gestione di alcuni problemi:

- difficoltà nel debugging e nella manutenzione rispetto ad un software single-threaded

- race conditions e sincronizzazioni
- deadlock, livelock, starvation

1.2 Creazione e attivazione di thread

Quando si manda in esecuzione un programma Java la JVM crea un thread che invoca il metodo main del programma. Dunque esiste sempre almeno un thread per ogni programma.

Successivamente altri thread sono attivati automaticamente da Java quali gestore eventi, interfaccia, garbage collector, etc.

La creazione esplicità di un thread segue due metodologie principali:

- **Metodo 1**: implements Runnable
- **Metodo 2**: extends Threads

1.3 Metodo 1: implements Runnable

Si implementa tramite i seguenti passi:

1. Definire un task (codice che il thread deve eseguire) tramite la definizione di una classe C che implementi l'interfaccia Runnable.
2. Creare un'istanza R di tale classe C
3. Creare un oggetto thread passandogli l'istanza R

L'interfaccia runnable appartiene al package java.language e contiene solo la segnatura del metodo void run(). Il task che il thread andrà ad eseguire andrà implementato all'interno di tale metodo void run().

Un istanza della classe che implementa Runnable è un task ovvero un frammento di codice che può essere eseguito in un thread. La creazione dei task non implica la creazione di un thread che lo esegua. Lo stesso task può essere eseguito da più thread.

```
// Task.java

public class Task implements Runnable {

    public Task(){}

    public void run () {
        System.out.println("I'm the thread " +
            Thread.currentThread().getName());
    }
}
```

Poiché più thread possono eseguire lo stesso codice (task) è possibile identificare il thread che sta eseguendo una determinata istanza del task tramite il metodo public static native Thread.currentThread().

```
// Main.java

public class Main {

    public static void main(String[] args){

        Task t = new Task(); // Istanziazione task
        Thread thread1 = new Thread(t); //Istanziazione thread
        thread.start(); //Avvio thread

    }

}
```

Se erroneamente, piùttosto che invocare thread.start() avessimo invocato thread.run() non avremmo avviato alcun thread poiché ogni metodo run() viene eseguito all'interno del flusso del thread attivato per l'esecuzione del programma principale.
L'invocazione del metodo run() rende il flusso di esecuzione sequenziale (il controllo dal main passa al metodo run() e tornerà unicamente al main al termine dell'esecuzione del metodo run()). Solo il metodo start() comporta l'avvio di un nuovo thread.

L'invocazione del metodo start() provoca l'esecuzione del metodo run(), che, a sua volta, provoca l'esecuzione del metodo run() di Runnable. Il metodo start():

- segnala allo schedulatore (tramite la JVM) che il thread può essere attivato (invoca un metodo nativo)
- l'ambiente del thread viene inizializzato
- restituisce il controllo al chiamante senza attendere che il thread attivato inizi la sua esecuzione

1.4 Metodo 2: extends Thread

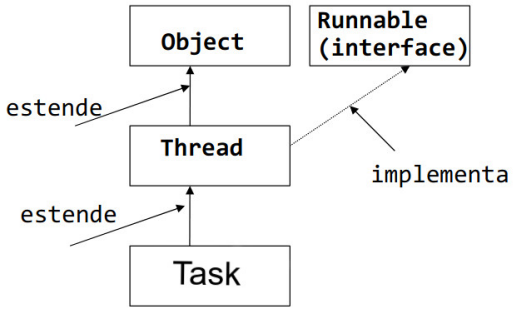
Si implementa tramite i seguenti passi:

- creare una classe C che estenda Thread
- effettuare l'overriding del metodo run() definito di quella classe
- istanziare un oggetto di quella classe: questo oggetto è un thread il cui comportamento è quello definito nel metodo run ridefinito

- invocare il metodo start() sull'oggetto istanziato

Ricordiamo che l'overriding consiste nel definire e implementare un metodo della sottoclasse con stesso nome e segnatura del metodo della superclasse.

```
// Task.java
public class Task extends Thread {

    public void run(){
    //Task here
        System.out.println("I'm thread " +
            Thread.currentTHread().getName());

    }

}

// Main.java
public class Main {

    public static void main(String[] args){

    Task t = new Task();
    t.start();


    }

}
```



Viene effettuato l'overriding del metodo run() all'interno della classe Task che estende Thread. Il comportamento del thread è definito dal metodo run() di Task.

1.5 Vantaggi e svantaggi dei due metodi

In Java una classe può estendere una sola altra classe secondo l'ereditarità singola: se si estende la classe Thread, la classe i cui oggetti devono essere eseguiti come thread non può estendere altre classi.
Ciò rappresenta una limitazione all'utilizzo del metodo 2: impedirebbe la gestione di eventi di interfaccia poiché la classe che getisce un evento deve estendere una classe C predefinita di Java dunque se il gestore deve essere eseguito in un thread separato, occorrerebbe definire una classe che estenda sia C che Thread, ma non è permesso poiché Java adotta l'ereditarietà singola.
Viene dunque generalmente utilizzato il metodo 1.

1.6 Thread Demoni

Sono thread a **bassa priorità** adatti a **jobs non-critici** da eseguire in background. Sono generalmente utilizzati per servizi in background utili fino a che il programma è in esecuzione (ex: garbage collector) ma possono anche essere user-defined.
Non appena tutti i thread non demoni del programma sono terminati, la JVM termina il programma, forzando la terminazione dei thread demoni.Un esempio di thread daemon tramite l'implementazione del primo metodo è:

```
// JavaDaemonThread
public class JavaDaemonThread {

    public static void main(String[] args) throws InterruptedException {

    //primo metodo creazione thread
    //dunque la classe Task deve estendere Runnable

    Thread daemon = new Thread(new Task(),"daemon-thread");
    daemon.setDaemon(true);
    daemon.start();

    }

}
```

E' possibile assegnare ad un thread un nome specifico indicandolo come parametro nel costruttore. In questo caso il nome del thread è "daemon-thread".
Un programma Java termina quando terminano tutti i thread non demoni. Se il thread main (che esegue il metodo main()) termina, i restanti thread ancora attivi e non demoni continuano la loro esecuzione fino alla terminazione. Se un thread usa l'istruzione **System.exit()** per terminare, allora tutti i threads terminano la loro esecuzione.

2 Gestione delle interruzioni

Java mette a disposizione un meccanismo per interrompere un thread e diversi meccanismi per intercettare l'interruzione in relazione a:

- lo stato del thread (running, blocked)
- se il thread è sospeso l'interruzione solleva una InterruptedException
- se in esecuzione, può testare un flag che segnala se è stata inviata una interruzione

In generale è il thread che decide in autonomia come rispondere alla interruzione.

2.1 Interruzione thread

Il metodo interrupt() viene utilizzato per lanciare un eccezione ad un determinato thread:

- imposta a true un valore booleano nel descrittore del thread
- flag vale true se vi sono interrupts pendenti

Tale flag è utilizzabile dal thread tramite due metodi:

- public static boolean Interrupted(): metodo statico che si invoca con il nome della classe Thread.Interrupted. Se il flag è true, lo resetta a false.
- public boolean isInterrupted(): deve essere invocato su un'istanza di oggetto di tipo thread (daemon.isInterrupted()). Non cambia il valore del flag se già true.

Entrambi i metodi restitusiscono un valore booleano che segnala se il thread ha ricevuto un'interruzione.

3 Classe Thread

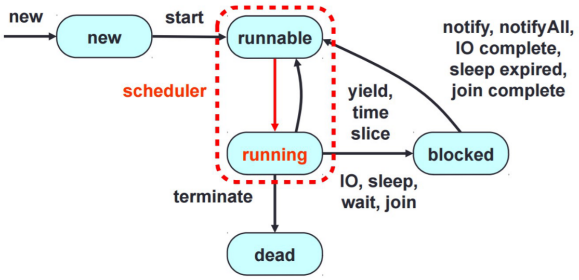
La classe thread fornisce una serie di metodi per l'interruzione, la sospensione e l'attesa della terminazione di un thread. Tuttavia **non contiene** metodi per la sincronizzazione tra thread (vedi java.lang.object). Inoltre è dotata di costruttori diversi differenti per i parametri utilizzati (nome thread, gruppo thread, etc).

Un thread viene posto nello stato i blocked tramite l'invocazione del metodo void sleep(long M) che sospense l'esecuzione del thread per M millisecondi. Durante la sleep il thread può essere interrotto. Tale metodo, essendo statico, non può essere invocato su una istanza di un thread.

Vi sono inoltre una serie di metodi get() e set() che consentono di reperire le caratteristiche di un thread. La classe Thread salva alcune informazioni:

- **ID**: identificatore thread

- **nome**: nome thread
- **priorità**: valore da 1 a 10 (1 priorità bassa)
- **stato**: uno dei possibili stati (new, runnable, blocked, waiting, time waiting o terminated).



Nel diagramma degli stati si evidenzia come, a partire dall'invocazione di specifici metodi Java, viene istanziato e messo in esecuzione. D'interesse è la transizione dallo stato running allo stato blocked: viene eseguita in caso di operazioni di IO (come scrittura su socket), sleep, wait (permette sospensione condizionata del thread in relazione ad un evento. Condizioni inverse per la transizione da blocked a runnable (notify (simile ad una signal), notifyAll (simil broadcast), IO complete, sleep expired, join complete). Si passa da running a runnable nel caso al termine del time slice.

3.1 Thread State Monitoring

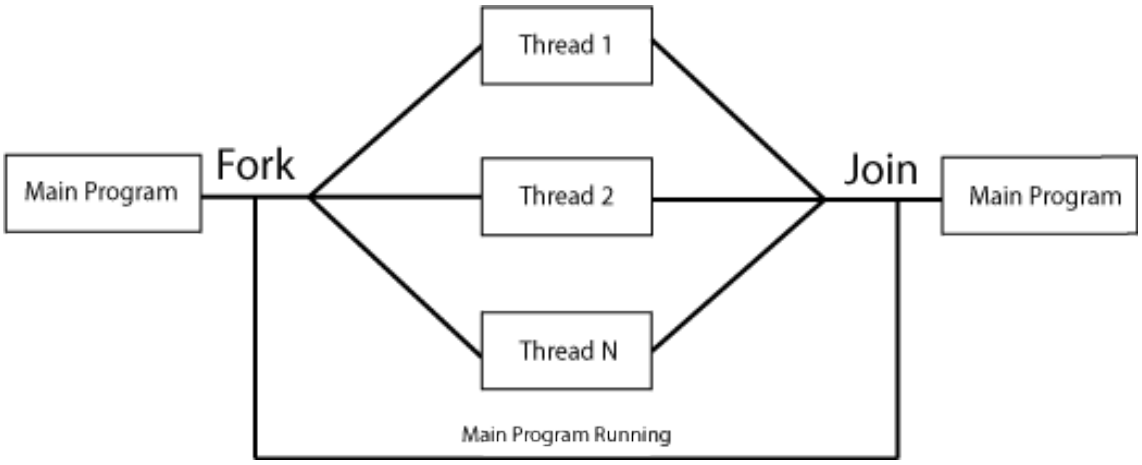
Si riporta un esempio di programma Java in grado di monitorare lo stato di più thread nel caso in cui questo vari durante la sua esecuzione, memorizzando nome, priorità, stato precedente, nuovo stato in un file "log.txt". Il monitor termina quanto tutti i thread sono terminati.

```
public static void main(String[] args) throws Exception
{
    Thread threads[]=new Thread[10];
    Thread.State status[]=new Thread.State[10];
    for (int i=0; i<10; i++){
        threads[i]=new Thread(new Calculator(i));
        if ((i%2)==0){
            threads[i].setPriority(Thread.MAX_PRIORITY);
        }else {
            threads[i].setPriority(Thread.MIN_PRIORITY);
        }
        threads[i].setName("Thread "+i);
```

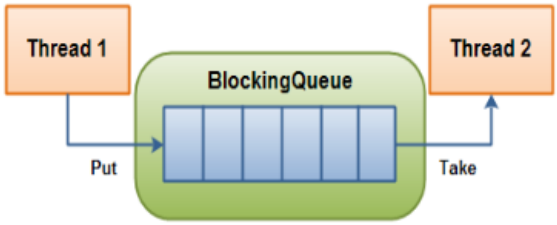
```
}


FileWriter file = new FileWriter("log.txt");
PrintWriter pw = new PrintWriter(file);
pw.printf("**********************************\n");
for (int i=0; i<10; i++){
       pw.println("Status of Thread"+i+":"+threads[i].getState());
       status[i]=threads[i].getState();
    }
for (int i=0; i<10; i++){
       threads[i].start();
}

boolean finish=false;
while (!finish) {
   for (int i=0; i<10; i++){
       if (threads[i].getState()!=status[i]) {
           pw.printf("Id %d -
               %s\n",threads[i].getId(),threads[i].getName());
           pw.printf("Priority: %d\n",threads[i].getPriority());
           pw.printf("Old State: %s\n",status[i]);
           pw.printf("New State: %s\n",threads[i].getState());
           pw.printf("**********************************\n");
           pw.flush();
           status[i]=threads[i].getState();
           }
       }
finish=true;
for (int i=0; i<10; i++){
       finish=finish && (threads[i].getState()==
            Thread.State.TERMINATED);
       }
    }
  }
}
```

3.2 Thread Join

La classe Thread fornisce un metodo **join()** che invocato sull'istanza di un thread t determina la sospensione del chiamante che rimane in attesa della terminazione del thread t.
E' possibile specificare un *timeout di attesa*: finita l'attesa, se il thread non è ancora terminato si riprenderà l'esecuzione a partire dalle istruzione successive a **t.join(maxTime)**. Se il thread sospeso sulla join() riceve un'interruzione viene sollevata una eccezione dunque si utilizza in un **blocco try-catch**.



3.3 Blocking Queue: interazione tra threads



Java definisce ***BlockingQueue*** (java.util.concurrent package) che consente la definizione di una **coda sincronizzata thread-safe** per quanto riguarda inserimenti e rimozioni. Il produttore può inserire elementi nella coda fino a che la dimensione della coda non raggiunge un limite, dopo di che si blocca e rimane bloccato fino a che un consumatore non rimuove un elemento. Il consumatore può rimuovere elementi della coda, ma se tenta di eliminare un elemento dalla coda si blocca fino a che il produttore inserisce un elemento nella coda.
Sono definiti 4 metodi per operare sulla BlockingQueue:

	Throws Exception	Special Value	Blocks	Times Out
Insert	`add(o)`	`offer(o)`	`put(o)`	`offer(o, timeout, timeunit)`
Remove	`remove(o)`	`poll()`	`take()`	`poll(timeout, timeunit)`
Examine	`element()`	`peek()`		

BlockingQueue è un'interfaccia, alcune implementazioni:

- **ArrayBlockingQueue**: coda di dimensione limitata. Memorizza gli elementi in un array il cui upper bound è definito a tempo di inizializzazione
- **LinkedBlockingQueue**: mantiene gli elementi in una struttura linkata che può avere un upper bound che se non specificato è la costante Java intera MAX VALUE.
- **SynchronousQueue**: non possiede capacità interna e le operazioni di inserzione devono attendere per una corrispondente operazione di rimozione e viceversa (fuorviante chiamarla coda).

```
// BlockingQueueExample
import java.util.concurrent.*;
    public class BlockingQueueExample {
        public static void main(String[] args) throws Exception {
            BlockingQueue queue = new ArrayBlockingQueue(1024);

            Producer producer = new Producer(queue);
            Consumer consumer = new Consumer(queue);

            new Thread(producer).start();
            new Thread(consumer).start();
            Thread.sleep(4000);
    }
}
```

```
// Producer
import java.util.concurrent.*;
    public class Producer extends Thread{
        protected BlockingQueue queue = null;

            public Producer(BlockingQueue queue) {
                this.queue = queue;
                }

        public void run() {
                try {
                    queue.put("1");
                    Thread.sleep(1000);
                    queue.put("2");
                    Thread.sleep(1000);
                    queue.put("3");
            } catch (InterruptedException e) {
        e.printStackTrace();
        }
}
```

```
// Consumer

import java.util.concurrent.*;
    public class Consumer extends Thread {
        protected BlockingQueue queue = null;

        public Consumer(BlockingQueue queue) {
            this.queue = queue;
            }

    public void run() {
        try {
                System.out.println(queue.take()); //la take potrebbe
                    sospendere il thread
                System.out.println(queue.take()); //dunque si fa uso del
                    try-catch
                System.out.println(queue.take());
        } catch (InterruptedException e) {
                    e.printStackTrace();
            }
    }
```

4 Thread Pooling

Istanziare un thread per ogni task comporta alcuni svantaggi:

- **Thread life cycle overhead**: la creazione/distruzione dei thread richiede un'interazione tra JVM e SO. Il ciclo di vita varia a seconda della piattaforma dunque non è mai trascurabile. In caso di richieste frequenti e 'lightweight' ciò può impattare negativamente sulle prestazioni dell'applicazione poiché costa più la gestione del thread che l'esecuzione del task.
- **Resource consumption**: molti threads idle quando il numero supera il numero di processori disponibili, comportando un alta occupazione di risorse (memoria). Utilizzo elevato del garbage collector e dello scheduler.
- **Stability**: limitazione al numero di threads imposto dalla JVM o dal SO.

L'idea alla base del thread pool consiste nel porre un **limite** oltre il quale non risulta conveniente creare ulteriori threads in modo da:

- sfruttare al meglio i processori disponibili
- evitare una contesa delle risorse disponibili da troppi threads
- diminuire il costo per l'attivazione/terminazione threads

L'utente struttura l'applicazione mediante un insieme di task: il thread pool è una **struttura dati** la cui dimensione massima può essere prefissata *(che*

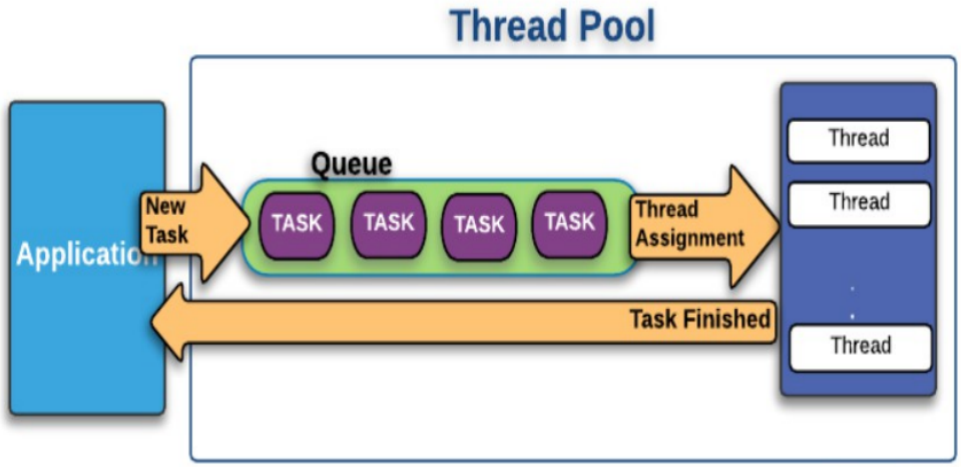


Figure 1: Schema generale

contiene riferimenti ad un insieme di threads). I thread pool possono essere riutilizzati per l'esecuzione di più **task diversi**: la sottomissione di un task al pool viene **disaccopiata** dall'esecuzione del thread infatti l'esecuzione del task può essere ritardata se non vi sono risorse attualmente disponibili.

4.1 Concetti generali e implementazione

L'utente, al momento della sottomissione:

- crea un pool e stabilisce una politica di gestione del thread pool
- l'attivazione di un thread può avvenire alla creazione del pool oppure on demand, all'arrivo di un nuovo task, etc.
- se è quando è opportuno può terminare l'esecuzione di un thread (in caso di un numero non sufficiente di tasks da eseguire)
- l'utente può sottomettere i task per l'esecuzione del thread pool

Il supporto, *al momento della sottomissione del task*, può:

- utilizzare un thread attivato in precedenza, inattivo in quel momento
- creare un nuovo thread
- memorizzare il task in una struttura dati (coda) in attesa dell'esecuzione

- respingere la richiesta di esecuzione del task

Il numero di threads attivi nel pool può variare dinamicamente.
Alcune interfacce definisco servizi generici di esecuzione: la classe **Executor** che opera come una **Factory** in grado di generare oggetti di tipo ExecutorService con comportamenti predefiniti. I tasks devono essere incapsulati in oggetti di tipo Runnable e passati a questi esecutori, mediante invocazione del metodo **execute()**.

```
//Main

public class Main {
   public static void main(String[] args) throws Exception
        {
         Server server=new Server();
         for (int i=0; i<10; i++){
            Task task=new Task("Task "+i); //Sottometto 10 task
            server.executeTask(task);
           }
         server.endServer();
        }
}


//Server

import java.util.concurrent.*;
    public class Server {
       private ThreadPoolExecutor executor;
        public Server( ){
            executor=(ThreadPoolExecutor)
                 Executors.newCachedThreadPool(); //Creazione thread pool
                 con una politica specifica
        }

        public void executeTask(Task task){
           System.out.printf("Server: A new task has arrived\n");
           executor.execute(task); //Sottomissione task
           System.out.printf("Server:Pool
                Size:%d\n",executor.getPoolSize());
           System.out.printf("Server:Active
                Count:%d\n",executor.getActiveCount());
           System.out.printf("Server:Completed Tasks:%d\n",
           executor.getCompletedTaskCount());

        public void endServer() {
                   executor.shutdown(); //Terminazione server

        }
```

```
}
```

La primitiva **NewCachedThreadPool** crea un pool con un comportamento predefinito: *alla sottomissione di un nuovo task, se tutti i thread del pool sono occupati nell'esecuione di altri task allora ne crea uno nuovo.*
Se disponibile un thread, viene riutilizzato per il nuovo task tuttavia se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina.
Ciò implementa l'**elastiticità**: un pool che può espandersi all'infinito ma si contrae quando la domanda di esecuzione di task diminuisce. E' possibile aumentare il riuso distanziando appropriatamente la sottomissione di task successivi in modo da riutilizzare lo stesso thread (ex: ogni 5 secondi una nuova sottomissione di task).
La primitiva **newFixedThreadPool(int N)** crea un pool in cui vengono creati N thread, al momento della inizializzazione del pool, riutilizzati per l'esecuzione di più task.
Alla sottomissione di un task T, se tutti i thread sono occupati nell'esecuzione di altri tasks, T viene inserito in una coda, gestita automaticamente dall'ExecutorService. Tale coda è illimitata e se almeno un thread è inattivo verrà utilizzato quel thread per l'esecuzione del task T.

4.2 Thread Pool Executor

E' definito con 5 parametri in ingresso: i primi 4 controllano la gestione del thread pool, l'ultimo è uno struttura dati per la memorizzazione di eventuali task in attesa di esecuzione.

```
import java.util.concurrent.*;
public class ThreadPoolExecutor implements ExecutorService{

    public ThreadPoolExecutor(int CorePoolSize,
                              int MaximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue <Runnable> workqueue);

    }
```

- **CarePoolSize**: dimensione minima del pool, definisce il core del pool.
- **MaxPoolSize**: indica la dimensione massima del pool. Non si avranno mai più di MaxPoolSize thread nel pool anche se vi sono task da eseguire e tutti i threads sono occupati nell'elaborazione di altri task.

Generalmente i thread del core possono venire creati secondo varie modalità:

- **PreStartAllCoreThreads()**: al momento della creazione del pool

- **on demand**: alla sottomissione di un task si crea un nuovo thread anche se qualche thread già creato dal core è inattivo (obiettivo:riempire il pool il prima possibile).

- quando sono stati creati tutti gli n thread del core, questi saranno tutti ed n sempre attivi

Se tutti i thread del sono sono stati già creati e viene sottomesso un nuovo task:

- se un thread del core è inattivo, il task gli viene assegnato altrimenti se la coda passata ultimo parametro del costruttore non è piena, il task viene inserito nella coda: i task verranno poi prelevati dalla coda ed inviati ai thread disponibili

- se coda piena e tutti i thread del core stanno eseguendo un task allora si crea un nuovo thread attivando così k thread t.c. corePoolSize <= k <= MaxPoolSize

- se coda piena e sono attivi MaxPoolSize thread allora il task viene respinto

E' possibile scegliere diversi tipi di coda (*tipi derivati da BlockingQueue*). Il tipo di coda scelto influisce sullo scheduling.

Supponendo che un thread termini dopo l'esecuzione di un task e che il pool contenga k threads:

- se **k $\leq$ core**: il thread si mette in attesa di nuovi task da eseguire (attesa è indefinita)

- se **k $>$ core ed il thread non appartiene al core**: si considera il timeout T definito al momento alla costruzione del thread pool. Se nessun viene sottomesso entro T, il thred termina la sua esecuzione riducendo il numero di threads del pool. Per definire un timeout occorre specificare un valore e l'unità di misura utilizzata (*ex. TimeUnit.MILLISECONDS*).

Tra i diversi tipi di coda vi sono:

- **SynchronousQueue**: dimensione uguale a 0. Ogni nuovo task T viene eseguito immediatamente o respinto. Alternativamente viene eseguito immediatamente se esiste un thread inattivo oppure se è possibile creare un nuovo thread (*numero di threads* $\leq$ *MaxPoolSize*).

- **LinkedBlockingQueue**: dimensione limitata. E' sempre possibile accodare un nuovo task, nel caso in cui tutti i threads siano attivi nell'esecuzione di altri task. Limite imposto è che la dimensione del pool non può superare core.

- **ArrayBlockingQueue**: Dimensione limitata stabilita dal programmatore.

Il comportamento del newFixedThreadPool o del newCachedThreadPool si può ottenere come istanza del metodo più generale **ThreadPoolExecutor**:

```
//newFixedThreadPool
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L,
        TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
   }
```

Il ***newFixedThreadPool*** avendo n thread sempre attivi allora i primi due parametri saranno uguali (***dimensione core = dimensione massima)***, con 0L non ucciderà mai i thrad. Infine fa uso di una *LinkedBLockingQueue<Runnable>* poiché è sempre possibile accodare nuovi task.

```
//newCachedThreadPool
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L,
        TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
   }
```

Il ***newCachedThreadPool*** lo ottengo similmente ponendo a 0 i thread del pool poiché vogliamo che al termine di un task un thread muoia.Con MAX VALUE non poniamo alcun limite al numero di thread del pool e inoltre dopo 60L (*60 secondi*) viene ucciso per convenzione. Infine si fa uso di una *SynchronousQueue<Runnable>* poiché i task non vanno in coda ma bensì ad ogni nuovo task avvio un thread per quello specifico task, evitando accodamento.

4.3 Executor Lifecycle

La JVM termina la sua esecuzione quando tutti i thread (**non demoni**) terminano la loro esecuzione. E' tuttavia necessario analizzare il concetto di **terminazione**, nel caso si utilizzi un *Executor Service* poiché i task vengono eseguiti in modo **asincrono** rispetto alla loro sottomissione dunque in un certo istante, alcuni task sottomessi precedentemente possono essere completati, altri in esecuzione, altri in coda. Un thread pool può rimanere attivo anche quando ha terminato l'esecuzione di un task.
Poiché alcuni threads possono essere sempre attivi, Java mette a disposizione dell'utente metodi che consentono di terminare l'esecuzione del pool. La terminazione può avvenire:

- **shutdown() - *modo graduale***: nessun task viene accettato dopo che la shutdown è stata invocata e tutti i task sottomessi precedentemente e non ancora terminati vengono eseguiti, compresi quelli in coda. Successivamente tutti i threads del pool terminano la loro esecuzione.

- **shutdownNow() - *modo istantaneo***: non accetta ulteriori task ed elimina tasks non ancora iniziati (*dalla coda*). Restituisce una lista dei

task eliminati dalla coda e tenta di terminare l'esecuzione dei thread che stanno eseguendo i task.

Nello specifico, ***shutdownNow()*** non garantisce la terminazione immediata dei threads del pool: invia una **interruzione** al thread in esecuzione nel pool ma se un thread non risponde all'interruzione non termina.
Il ciclo di vita di un execution service è dunque **running, shutting down** e infine **terminated**. Un pool viene creato in running, quando viene invocata una shutdown() o shutdownNow() passa allo stato shutting down e quando tutti i thread sono terminati passa nello stato terminated.
I thread sottomessi per l'esecuzione ad un pool in stato di shuttind down o terminated possono essere gestiti da un **rejected execution handler** che può semplicemente scartarli, sollevare una eccezione o adottare politiche più complesse e mirate alla gestione.

4.4 Callable e Future

Un oggetto di tipo **Runnable** incapsula un'attività che viene eseguita in *modo asincrono*. La Runnable è un metodo asincrono, senza parametri e che non restituisce un valore di ritorno.
Per definire un task che restituisca un valore di ritorno si deve ricorrere all'**interfaccia Callable** che può restituire un risultato e sollevare eccezioni.
Ricordiamo che per le Callable non è consentito il ritorno di un tipo primitivo (*int*) ma bensì tipi definiti come oggetti (*Integer*).
Immaginiamo di definire un thread che esegue un calcolo: il risultato vorrei gestirlo in modo asincrono in fasi successive alla sua computazione dunque definisco quindi un thread Callable (*il task*) ed il risultato sarà memorizzato in un oggetto di tipo ***Future***. La classe *FutureTask* fornisce una implementazione dell'interfaccia Future.

```
public interface Callable <V>{
      V call() throws Exception;
   }
```

L'interfaccia Callable contiene solo **call()** *(analogo al metodo run() di Runnable).* Il codice del task è implementato nel metodo call() e può restituire un valore e sollevare eccezioni. Il parametri di tipo $<\mathbf{V}>$ indica il tipo del valore restituito.

```
import java.util.concurrent.*;

public class Task implements Callable<Integer>{

   private Integer age;
   public Task(Integer age){
       this.age = age;
   }

   public Integer call(){
```

```
        Integer year_of_birthday = <calculate year>;
        return year_of_birthday;
    }

}
```

Il risultato restituito sarà nella forma:

```
//....
ExecutorService pool = Executors.newCachedThreadPool();
//....
Task t1 = new Task(10); //Istanzio nuovo task
Future<Integer> year = pool.submit(t1); //Sottometto al pool
```

Dunque il valore restituito dalla Callable, acceduto mediante un oggetto di tipo <Future>, rappresenta il risultato della computazione. Se si usano i threads pools si sottomette direttamente l'oggetto di tipo Callable al pool tramite il metodo **submit()** e la sottomissione restituisce un oggetto di tipo <**Future**>.

5 Sincronizzazione e risorse condivise

Scrivere programmi con i threads che si sincronizzano tra loro è un'attività complessa e pericolosa in confronto all'uso di nuove tecnologie tuttavia le locks sono ancora utilizzate per scrivere programmi concorrenti e sono solitamente alla base di costrutti ad alto livello (anche in Java). Le locks sono di fatto una formalizzazione dei meccanismi hardware sottostanti i cui vantaggi sicuramente risiedono nell'assenza di vincoli sul loro utilizzo ma ne implicano anche le difficoltà di implementazione e manutenzione.

Uno scenario tipico di un programma concorrente è quando un insieme di thread condividono una risorsa e l'accesso non controllato a tale risorsa può provocare situazioni di errore ed incosistenze (*come le race conditions*). Definiamo sezione critica un blocco di codice che viene eseguito un thread per volta e all'interno del quale si effettua l'accesso ad una risorsa condivisa. I meccanismi Java di sincronizzazione per l'implementazione delle **sezioni critiche** sono:

- **interfaccia Lock** (e relative implementazioni)
- concetto di **monitor**

Caratteristiche tipiche delle **race conditions** sono il **non determinismo** dovuto ad una commutazione di contesto prima della terminazione di un certo metodo che modifica lo stato condiviso e può generare un *risultato incosistente*. Chiaramente l'incosistenza non si presenta necessariamente ad ogni esecuzione e dunque è un comportamento dipendente dal tempo (*interleaving, context switch, etc.*).

5.1 Classe Thread Safe

L'esecuzione concorrente dei metodi definiti in una classe **thread safe** non provoca comportamenti anomali o race conditions. Per rendere una classe thread safe è necessario garantire che le istruzioni contenute all'interno dei metodi che utilizzano risorse condivise vengano eseguite in modo **atomico / indivisibile / in mutua esclusione**. Java offre diversi meccanismi per la sincronizzazione di threads:

- **meccanismi a basso livello**: lock e variabili condizione associate
- **meccanismi ad alto livello**: syncronized() oppure wait(), notify(), notifyAll() ed i monitors.

5.2 Sincronizzazione: Lock

In Java una **lock** è un oggetto che può trovarsi in due stati diversi **locked** o **unlocked**. Lo stato viene impostati dai relativi metodi *lock() e unlock()*. Un solo thread alla volta può impostare lo stato a locked cioè ottenere la lock mentre gli altri thread che tentano di acquisire una lock si bloccano. Se un thread tenta di acquisire una lock rimane bloccato fintanto che la lock è detenuta da un altro thread mentre se un thread rilascia una lock uno dei thread in attesa la acquisisce.

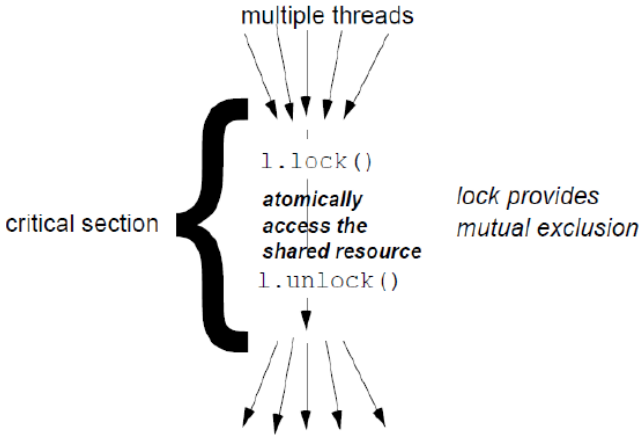


Figure 2: Schema generale

L'interfaccia in Java per le lock è *java.util.concurrent.locks.Lock* con relativa implementazione *java.util.concurrent.locks.ReentrantLock* (vedi API). Si riporta un esempio in Java in cui si fa uso della lock:

```
import java.util.concurrent.locks.*;
```

```
public class Account {
    private double balance;
    private final Lock accountLock = new ReentrantLock();

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public void addAmount(double amount) {

            accountLock.lock();
            double tmp=balance;

            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            tmp+=amount;
            balance=tmp;
            accountLock.unlock();
    }

    public void subtractAmount(double amount){

        accountLock.lock();
        double tmp=balance;
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
                    e.printStackTrace();
            }

        tmp-=amount;
        balance=tmp;
        accountLock.unlock();
    }
}
```

Si possono verificare delle situazioni di **deadlock** quando un thread A acquisice una lock X e il thread B acquisice una Y. Se rispettivamente A tenta di acquisire Y e B tenta di acquisire X entrambi i thread rimarranno bloccati all'infinito. L'interfaccia *Lock* e la classe *ReentrantLock* che implementa include un altro metodo utilizzato per ottenere il controllo della lock: il metodo **try-**

Lock() (*non bloccante*) tenta di acquisire la lock e se essa è già posseduta da un altro thread, il metodo termina immediatamente e restituise il controllo al chiamante. Restituisce invece un booleano con valore true se è riuscito ad acquisire la lock, false altrimenti.

Inevitabilmente l'uso delle lock introduce un overhead che causano una **performance penalty** dovuta a più fattori quali *contention, bookkeeping, scheduling, blocking o unblocking*. Tale penalità nelle performance sono caratteristiche di tutti i costrutti ad alto livello di Java basati su lock.

Le **reentrant locks** (*o recursive lock*) utilizzano un contantore che viene incrementato ogni volta che un thread acquisce la lock e decrementanto ogni volta che un thread rilascia la lock. Tale lock è definitivamente rilasciata quando il contatore diventa 0 e un thread può acquisire più volte la lock su uno stesso oggetto senza bloccarsi.

Un altro tipo di lock sono le **read/write locks**: la struttura dati condivisa può essere letta e acceduta da più thread tuttavia vi può essere solo uno scrittore (che modifica la struttura dati) o più lettori contemporaneamente (senza alcun thread scrittore).
Java implementa le **read/write locks** come *interfaccia ReadWriteLock* che mantiene una coppia di lock associate, una per le *operazioni di lettura* e una per le *scritture*: chiaramente la read lock può essere acquisita da più lettori purché non vi siano scrittori mentre la write lock è esclusiva. Vediamo un esempio di read/write lock:

```
import java.util.concurrent.locks.*;

public class SharedLocks extends Thread {

    int a =1000, b=0;
    private ReentrantReadWriteLock readWriteLock = new
        ReentrantReadWriteLock();
    private Lock read = readWriteLock.readLock();
    private Lock write = readWriteLock.writeLock();

    public int getsum (){
        int result;
        read.lock();
        result=a+b;
        read.unlock();
        return result;
    }

    public void transfer (int x){

        write.lock(); //Recinto di mutua esclusione
        a = a-x;
        b = b+x;
        write.unlock();
    }
```

```
}
```

5.3 Problema produttore-consumatore

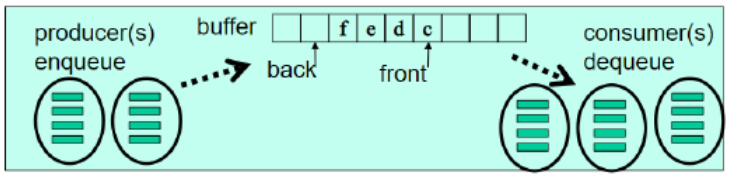


Figure 3: Produttore - Consumatore

E' un classico problema che descrive due o più thread che condivisono un buffer di dimensione fissata, usato come coda. Il **produttore P** produce un nuovo valore, lo inserisce nel buffer e torna a produrre valori. Il **consumatore C** consuma il valore (*rimuovendolo dal buffer*) e torna a richiedere valori. Necessita garantire che il produttore non provi ad aggiungere un dato nella coda se è piena ed il consumatore non provi a rimuovere un dato da una coda vuota.

Si evince che vi sono due tipi di sincronizzazione:

- **implicita**: la mutua esclusione sull'oggetto condivisa è garantita dall'uso di lock
- **esplicita**: occorrono meccanismi per attivare/riattivare threads al verificarsi di determinate condizioni

Un ipotesi non banale è l'uso di un buffer a **dimensione finita** (tramite un ArrayList o vettore di dimensione limitata) e anche se utilizzassimo una coda di dimensione illimitata sarebbe comunque necessaria la sincronizzazione per la coda vuota. L'idea di ottimizzazione utilizzata da Java è quella di *limitare la concorrenza a determinate parti di una struttura dati poiché è possibile che più thread accedano (in lettura o scrittura) a parti diverse della stessa struttura dati condivisa.*

5.4 Variabili condizione

Dunque i meccanismi forniti da Java devono consentire di definire un insieme di **condizioni sullo stato** dello'oggetto condiviso e consetire la **sospensione o riattivazione** dei threads sulla base del valore di queste condizioni.
Una prima soluzione è fornita dalla combinazione degli strumenti di basso livello visti finora: l'uso di variabili condizione e metodi per la sospensione su queste variabili in combinazione con la definizione di code associate alle variabili in cui memorizzare i threads sospesi consente l'implementazioni di tale meccanismo. Altre soluzioni verrano da meccanismi di *monitoring* ad alto livello. L'idea delle variabili condizione è data da tale schema:

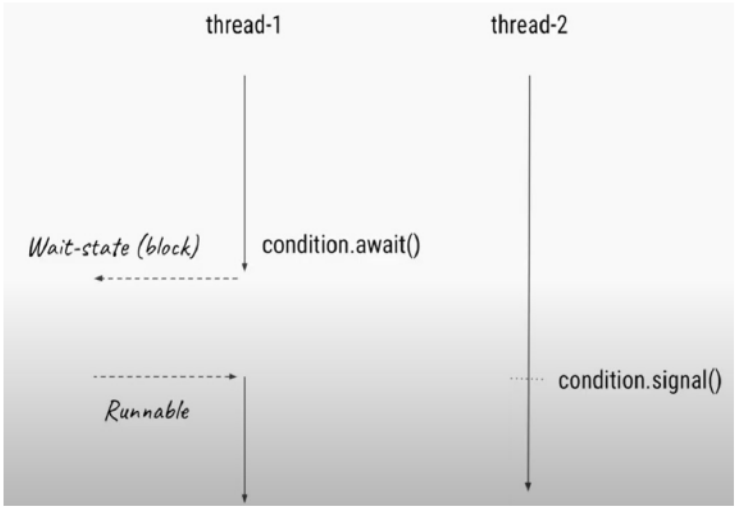


Figure 4: Schema variabili condizione

Le **variabili condizione** sono associate ad una lock e permettono ai thread di controllare se una *condizione* sullo stato della risorsa è verificata o meno. Se la condizione è falsa, rilasciano la *lock()*, si sospendono ed inseriscono il thread in una coda di attesa per quella condizione. Inoltre risvegliano un thread in attesa quando la condizione risulta verificata.

Solo dopo aver acquisito la lock su un oggetto è possibile sospendersi su una variabile d condizione, altrimenti viene generata una IllegalMonitorException. La JVM mantiene più code:

- una per i threads in attesa di acquisire la lock
- una per ogni variabile condizione

```
//Schema generale

private Lock lock = new ReentrantLock();
private Condition conditionMet = lock.newCondition();

public void method1() throws InterruptedException{
    lock.lock();
    try {
    conditionMet.await(); //sospensione
    //l'esecuzione riprende da questo punto
    //operazioni che dipendevano dalla verifica della condizione
    } finally {
        lock.unlock();
    }
```

```
}

public void method2() {
    lock.lock();
    try {
        //operazioni che rendono valida la condizione
        conditionMet.signal();
        }finally {
            lock.unlock();
        }
}
```

L'**interfaccia Condition** definisce i metodi per sospendere un thread e per risvegliarlo. Le condizioni sono istanze di una classe che implementa questa interfaccia. Vediamo una soluzione del problema produttore-consumatore con le variabili condizione:

```
    public class Messagesystem {
        public static void main(String[] args) {
            MessageQueue queue = new MessageQueue(10);
                new Producer(queue).start();
                new Producer(queue).start();
                new Producer(queue).start();
                new Consumer(queue).start();
                new Consumer(queue).start();
                new Consumer(queue).start();
            }
    }
```

```
// Produttore
import java.util.concurrent.locks.*;

    public class Producer extends Thread {

        private int count = 0;
        private MessageQueue queue = null;

            public Producer(MessageQueue queue){
                this.queue = queue;
            }

    public void run(){
        for(int i=0;i<10;i++){
            queue.produce ("MSG#"+count+Thread.currentThread());
            count++;
        }
    }
}
```

```java
//Consumatore
public class Consumer extends Thread {

    private MessageQueue queue = null;

    public Consumer(MessageQueue queue){
         this.queue = queue;
    }

    public void run(){

        for(int i=0;i<10;i++){
                Object o=queue.consume();
                int x = (int)(Math.random() * 10000);
            try{
                Thread.sleep(x);
            }catch (Exception e){};
      }
    }
 }


//Implementazione coda
import java.util.concurrent.locks.*;
public class MessageQueue {

    final Lock lockcoda;
    final Condition notFull;
    final Condition notEmpty;
    int putptr, takeptr, count;
    final Object[] items;

    public MessageQueue(int size){

            lockcoda = new ReentrantLock();
            notFull = lockcoda.newCondition();
            notEmpty = lockcoda.newCondition();

            items = new Object[size];
            count=0;putptr=0;
            takeptr=0;
    }


    //Metodo per inserire in coda
    public void produce(Object x) throws InterruptedException {
            lockcoda.lock();
            try{
```

```
            while (count == items.length)
            notFull.await();

            // gestione puntatori coda
            items[putptr] = x;
            putptr++;
            ++count;

            if (putptr == items.length) putptr = 0;
            System.out.println("Message Produced"+x);
            notEmpty.signal();
        }finally {
                lockcoda.unlock();
        }
    }


    // Metodo per eliminare dalla coda
    public Object consume() throws InterrupedExeceptions {

    lockcoda.lock();
        try{
                while (count == 0)
                notEmpty.await();
        }

        \\ gestione puntatori coda
        Object data = items[takeptr];
        takeptr=takeptr+1;
        --count;

        if (takeptr == items.length) {takeptr = 0};
        notFull.signal();
        System.out.println("Message Consumed"+x);
        return x;}
        finally
        {lockcoda.unlock(); }
    }
}
```

Come si nota dall'esempio, si fa uso di un *while(condizione) wait()* a causa delle **signal spurie**: tra il momento in cui ad un thread arriva una notifica ed il momento in cui riacquisisce la lock, la condizione può diventare dinuovo falsa. In figura l'esempio del controllo di una condizione tramite (*l'errato*) uso dell'if. La prassi in uso è quella di ricontrollare la condizione dopo aver acquisito la lock e dunque inserire la *wait* in un *ciclo while*.

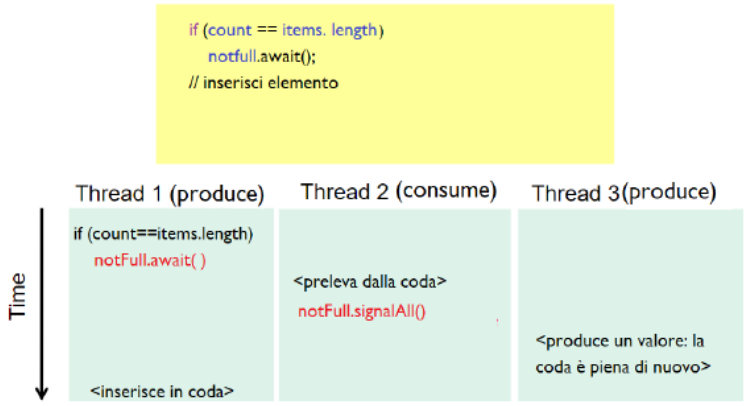


Figure 5: Signal spurie con if

5.5 Granularità delle lock

L'accesso concorrente a strutture dati deve garantire la **thread safeness** ovvero l'imperativo di mantenere consistente la struttura dati condivisa quando più thread vi accedono concorrentemente. E' possibile gestire l'accesso ad una struttura dati condivisa in relazione alla granularità delle restrizioni, in particolare tramite l'uso di:

- **Linked list**: per ogni elemento vi sono puntatori all'elemento succesivo e precedente. In questo modo inserzione ed eliminazione riguarderanno i singoli elementi della lista.
- **Coarse grain lock**: una singola lock per tutta la struttura. E' indubbiamente inefficiente poiché nessun thread può accedere alla struttura mentre un altro thread la sta modificando.
- **Hand-over-hand locking**: si implementa la mutua esclusione solo su piccole porzioni della lista, permettendo ad altri thread l'accesso ad elementi diversi della struttura (*sotto ipotesi di strutture ampie, composte da più componenti a cui thread differenti necessitano di lavorare contemporamente*).

5.5.1 Linked list

5.5.2 Coarse grain lock

5.5.3 Hand-over-hand lock

References