06β Πολυνηματικός προγραμματισμός

Τεχνολογία Λογισμικού

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο

Χειμερινό εξάμηνο 2017-18

Δρ. Κώστας Σαΐδης (saiko@di.uoa.gr)

Πολυνηματικός (multi-threaded) προγραμματισμός

Η γνώση συγγραφής πολυνηματικού κώδικα είναι απολύτως απαραίτητη στην ανάπτυξη σύνθετων έργων λογισμικού.

Γιατί;

- Μεγαλύτερος παραλληλισμός (ακόμα και τα κινητά έχουν multiple CPUs πλέον)
- Καλύτερο CPU utilization
- Μείωση του latency
- Αύξηση του throughput

Πολυνηματικό προγραμματισμός στο JVM

- Θα βασιστούμε στο JVM για την κουβέντα μας
- Ένας από τους λόγους που έχουμε πλέον πολλές γλώσσες στο JVM είναι γιατί υλοποιεί καλά τον πολυνηματικό προγραμματισμό!

Διεργασίες (processes)

Από τα ΛΣ: προγράμματα που τρέχουν ανεξάρτητα και απομονωμένα το ένα από το άλλο και διαθέτουν:

- ξεχωριστό περιβάλλον εκτέλεσης (π.χ. ξεχωριστή στοίβα, καταχωρητές και PC).
- ξεχωριστό χώρο μνήμης (memory space).

Νήματα (threads)

Παρόμοια με τις διεργασίες αλλά πιο "ελαφριά" (light-weight).

- Έχουν και αυτά το δικό τους περιβάλλον εκτέλεσης.
- Είναι "φθηνότερα" από τις διεργασίες.
- Ζουν "μέσα" στις διεργασίες (κάθε διεργασία έχει τουλάχιστον ένα νήμα).
- Ξεχωριστά μονοπάτια εκτέλεσης του ίδιου προγράμματος, που εκτελούνται ταυτόχρονα και ασύγχρονα μεταξύ τους.

Διεργασίες, νήματα και η πλατφόρμα της Java

- Το ίδιο το JVM υλοποιείται συνήθως ως μια διεργασία.
- Τόσο η γλώσσα Java όσο και το JVM υποστηρίζουν εγγενώς πολυνηματικά προγράμματα.
- Σήμερα θα δώσουμε έμφαση στις βασικές έννοιες και μηχανισμούς του JVM και όχι τόσο στις βιβλιοθήκες της Java (θα τις καλύψουμε σε επόμενο φροντιστήριο)

Το κύριο νήμα (main thread) του JVM

- Κάθε πρόγραμμα στο JVM ξεκινά με ένα νήμα, το ονομαζόμενο κύριο, το οποίο δημιουργείται αυτόματα (δεν είναι ευθύνη του προγραμματιστή να το αρχικοποιήσει).
- Το κύριο νήμα έχει την ευθύνη εκτέλεσης της main μεθόδου `public static void main(String[] args)`, η οποία θα πρέπει να είναι διαθέσιμη στην "κύρια" κλάση της εφαρμογής.

```
$ java -cp CLASSPATH package.of.MainClass
```

- Στο παρασκήνιο, το JVM μπορεί να εκτελεί διάφορα εσωτερικά νήματα, τα οποία δεν είναι "ορατά" από τον προγραμματιστή (όπως;).

Προγραμματιστική διαχείριση των νημάτων

- java.lang.Thread: Στιγμιότυπα της κλάσης αναπαριστούν διαφορετικά νήματα εκτέλεσης στην εφαρμογή. Πολλές και χρήσιμες μέθοδοι που θα καλύψουμε σήμερα.
- java.lang.Runnable: Αφαίρεση (interface) που αναπαριστά τι θα πρέπει να εκτελέσει ένα νήμα.

```
public interface Runnable {
  void run();
}
```

Thread is Runnable

```
public class Thread implements Runnable {
   ...
   public void run() {
     //do nothing
   }
   ...
}
```

Κατασκευή νημάτων

- Πολύ απλό: στιγμιοτύπιση ενός Thread object και επίκληση της μεθόδου `start()`.
- Αρκεί να υλοποιήσουμε τι θέλουμε να εκτελεστεί από το νήμα.
- Με δύο τρόπους:
  - Υλοποιούμε το Runnable ξεχωριστά και το δίνουμε ως παράμετρο στο Thread.
  - Εξειδικεύουμε την κλάση Thread, επαναορίζοντας (override) τη μέθοδο run.

1ος τρόπος

```
public class MyRunnable implements Runnable {
  private final String message;
  public MyRunnable(String message) {this.message = message;}
  public void run() {
    while(true) {
      System.out.println(
        message + " from " + Thread.currentThread()
      );
      try {
        Thread.sleep(1000);
      }
      catch(InterruptedException ie) {
        System.out.println("Interrupted");
      }
    }
  }
}
```

Χρήση

```
  public static void main(String[] args) {
    if (args!=null && args.length==1) {
      new Thread(new MyRunnable(args[0])).start();
    }
    else {
      System.out.println("No message for MyRunnable");
    }
  }
}
```

2ος τρόπος

```
public class MyThread extends Thread {
  private final String message;
  public MyThread(String message) {this.message = message;}
  public void run() {
    while(true) {
      System.out.println(
        message + " from " + Thread.currentThread()
      );
      try {
        Thread.sleep(1000);
      }
      catch(InterruptedException ie) {
        System.out.println("Interrupted");
      }
    }
  }
}
```

Χρήση

```
public static void main(String[] args) {
  if (args!=null && args.length==1) {
    new MyThread(args[0]).start();
  }
  else {
    System.out.println("No message for MyThread");
  }
}
```

Παραλλαγή του 2ου τρόπου

```
public static void main(final String[] args) {
  if (args!=null && args.length==1) {
    Thread t = new Thread() { //Anonymous inner class
      @Override
      public void run() {
        while(true) {
          System.out.println(
              args[0] + " from " + Thread.currentThread()
          );
          try {
            Thread.sleep(1000);
          }
          catch(InterruptedException ie) {
            System.out.println("Interrupted");
          }
        }
      }
    };
    t.start();
  }
}
```

Runnable.run()

- Δε τρέχουν "μαγικά" όλα τα Runnables ως νήματα
- Η `run()` είναι μια "κανονική" μέθοδος, που μπορεί να κληθεί σε οποιοδήποτε thread

```
public class RunNotMagic {
  public static void main(String[] args) {
    MyRunnable r = new MyRunnable("I am running");
    r.run();
  }
}
```

Τα "μαγικά" γίνονται στο Thread (a)

Τα νήματα στο JVM:

- έχουν καταστάσεις (states)
- έχουν προτεραιότητες (priorities)
- μπορεί να είναι "δαίμονες" (daemons)
- μπορεί να "κοιμηθούν" (sleep)
- μπορεί να διακοπούν (interrupted)
- μπορούν να συγχρονιστούν (synchronized)

Τα "μαγικά" γίνονται στο Thread (b)

Τα νήματα στο JVM:

- μπορούν να παραχωρήσουν προτεραιότητα (yield)
- μπορούν να σμίξουν/συνδεθούν (join)
- μπορούν να κωλυσιεργούν (block)
- μπορούν να περιμένουν (wait)
- μπορούν να ενημερωθούν (notify)

Κατάσταση ενός νήματος

- NEW: Νέο στιγμιότυπο νήματος που δεν έχει ακόμα ξεκινήσει.
- RUNNABLE: Τρέχει.
- BLOCKED: Κώλυμα για την απόκτηση ενός monitor/lock (παρακάτω).
- WAITING: Αναμονή (επ' άπειρο) για κάποιο άλλο νήμα.
- TIMED_WAITING: Όπως προηγουμένως, αλλά με άνω όριο αναμονής.
- TERMINATED: Η run() μέθοδος του νήματος ολοκληρώθηκε/ επέστρεψε.

Βλ. java.lang.Thread.State enum

Thread.yield(), getPriority(), setPriority()

Δε χρησιμοποιούμε ούτε τη yield() ούτε τις προτεραιότητες (priorities) των νημάτων για "εγγυημένο" χρονοπρογραμματισμό τους, καθώς αποτελούν μόνο ενδείξεις για το JVM και δεν είναι δεσμευτικές.

Μόνο επικουρική η χρήση τους.

Daemon threads

Στο JVM:

- Κάθε νήμα είναι είτε daemon thread είτε user thread.
- Κατά την κατασκευή τους (by default), τα νήματα είναι user threads.
- Τα "εσωτερικά" νήματα του JVM είναι daemon threads (π.χ. garbage collection).

Χρήση της μεθόδου `Thread.setDaemon(boolean)` για να ορίσουμε το είδος του νήματος, πριν την επίκληση της start().

```
Thread t = new Thread(...)
t.setDaemon(true)
t.start()
```

Πότε ολοκληρώνεται η εκτέλεση ενός προγράμματος από το JVM;

Το JVM τερματίζει όταν όλα τα νήματα που εκτελούνται είναι daemon threads.

Νήματα και γραφικές διεπαφές χρήστη

- Το διαβόητο event thread!
- Όλες οι εφαρμογές με γραφική διεπαφή χρήσης χρησιμοποιούν ένα ξεχωριστό νήμα για τη λήψη και επεξεργασία των "γεγονότων" διάδρασης του χρήστη με την εφαρμογή (γιατί;).
- Στη Java, έχουμε το AWT/Swing event thread, το οποίο είναι user thread (γιατί;)

Κουίζ

Γιατί δεν τερματίζει ο παρακάτω κώδικας αφού η main μέθοδος είναι κενή;

```
import javax.swing.*;
public class EmptyMain {
  private static final EmptyMain self = new EmptyMain() ;
  private EmptyMain ( ) {
    JFrame jf = new JFrame ("oops!") ;
    jf.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE) ;
    jf.pack() ;
    jf.setVisible(true) ;
  }
  public static void main( String [ ] args ) {
    //nothing here!!!!!
  }
}
```

Thread.sleep()

- Στατική μέθοδος που προκαλεί το τρέχων νήμα να παύσει την εκτέλεσή του για ορισμένο χρονικό διάστημα
- Είδαμε παραδείγματα προηγουμένως (MyRunnable, MyThread)

Thread.interrupt()

- Instance μέθοδος που συμβατικά σημαίνει "διέκοψε αυτό που κάνεις".
- Όταν ένα νήμα λάβει σήμα διακοπής η μέθοδος isInterrupted() επιστρέφει true.
- Είναι ευθύνη του προγραμματιστή να ερμηνεύσει το σήμα διακοπής και να το χειριστεί με βάση τη λογική του προγράμματός του (π.χ. τερματισμός; επαναπροσπάθεια; κάτι άλλο;).

Διακοπή νημάτων

- Οι μοναδικές περιπτώσεις που το "σήμα διακοπής" έχει συμπεριφορά μη οριζόμενη από τον προγραμματιστή είναι οι εξής:
  - Όταν το νήμα βρίσκεται στη sleep ή στη join (παρακάτω)
  - Όταν το νήμα κωλυσιεργεί γιατί αναμένει κάποιο monitor/lock (wait)
- Τότε το JVM εγείρει ένα InterruptedException και επαναθέτει την κατάσταση διακοπής (η isInterrupted() επιστρέφει false).

java.lang.Thread source code

```
public boolean isInterrupted() {
  return isInterrupted(false);
}

/*
Tests if some Thread has been interrupted.
The interrupted state is reset or not based on the value
of clear that is passed.
*/
private native boolean isInterrupted(boolean clear);


public static boolean interrupted() {
  return currentThread().isInterrupted(true);
}
```

Παράδειγμα

```
public class SumOfFactorials implements Runnable {
  private static final long factorial(long l) {
    if (l == 0) return 1L;
    else return l * factorial(l-1);
  }

  public void run() {
    long sumOfFactorials = 0L;
    int counter = 0;
    while (true) {
      sumOfFactorials = sumOfFactorials + factorial(counter);
      counter++;
      if ((counter % 100 == 0) && Thread.interrupted()) {
        System.out.printf(
          "The sum of the first %d factorials is: %d",
          counter,
          sumOfFactorials);
        return;
      }
    }
  }
}
```

Χρήση

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

  Thread calculator = new Thread(new SumOfFactorials());
  calculator.start();

  Thread.sleep(500);
  calculator.interrupt();
}
```

```
The sum of the first 19200 factorials is: 1005876315485501978
```

Thread.join()

- Μηχανισμός που ένα νήμα αναμένει την ολοκλήρωση ενός άλλου.
- Έστω δύο νήματα `t1`, `t2`, όπου η `t1.run()` περιέχει μια κλήση στην `t2.join()`.
  - Όταν η εκτέλεση φτάσει στην επίκληση `t2.join()`, το `t1` θα παύσει την εκτέλεσή του.
  - Όταν το `t2` οκοκληρώσει την εκτέλεσή του, το `t1` θα συνεχίσει.
- Η επίκληση στη `join()` μπορεί να λάβει και άνω όριο αναμονής.

Παράδειγμα

```
public static void main(String[] args)
                    throws InterruptedException{
  Worker worker = new Worker();
  System.out.println("Starting the worker...");
  worker.start();
  System.out.println("Waiting for the worker to finish...");
  worker.join(3000);
  if (worker.isAlive()) {
    System.out.println("Not yet, telling worker to exit");
    worker.exit();
  }
}
```

Επίσης: παράδειγμα ορθής χρήσης ενός flag για έξοδο από τη run().

Worker

```
public class Worker extends Thread {
  private volatile boolean exit;
  void exit() {
    exit = true;
    System.out.println("Exit is set to true");
  }

  public void run() {
    while(!exit) {
      System.out.println(this.getName() + " is working!");
      try {
        Thread.sleep(1000); //compute something
      }
      catch(InterruptedException ie) {
        return; //interrupted on sleep
      }
    }
    //graceful completion
    System.out.println(this.getName() + ": Worker exits");
  }
}
```

Output

```
Starting the worker...
Waiting for the worker to finish...
Thread-5 is working!
Thread-5 is working!
Thread-5 is working!
Thread-5 is working!
Not yet, telling worker to exit
Exit is set to true
Thread-5: Worker exits
```

Συγχρονισμός (synchronization)

- Τα νήματα μπορούν να μοιράζονται την πρόσβαση σε κοινά αντικείμενα ή πεδία (θυμηθείτε ότι στο JVM υπάρχει ένα κοινό heap για όλα τα στιγμιότυπα).
- Σε αυτή τη διαδικασία, μπορεί να προκύψουν τα εξής δύο είδη σφαλμάτων:
  - Παρεμβολή/παρέμβαση νημάτων (thread interference)
  - Ασυνέπεια μνήμης (memory inconsistency)
- Ο συγχρονισμός αποτρέπει την εμφάνιση αυτών των σφαλμάτων.

Πιο συγκεκριμένα

Thread interference

Two operations, running in different threads, but acting on the same data, interleave (two operations that consist of multiple steps, overlap each other's sequences of steps).

Memory inconsistency

Different threads have inconsistent views of what should be the same data. This may happen during data transfer from/to the CPU registers/caches, the main memory and the thread-local registers/caches.

Οι βασικοί μηχανισμοί συγχρονισμού στη Java

- Συγχρονισμένες μέθοδοι και blocks (`synchronized` keyword):
  - Το JVM διασφαλίζει ότι ένα νήμα θα εκτελεί τον συγχρονισμένο κώδικα τη φορά.
- Πεδία `volatile` (light-weight συγχρονισμός)
  - Το JVM διασφαλίζει ατομικές αναγνώσεις/εγγραφές (atomic reads/writes) σε volatile μεταβλητές.
- Από την έκδοση 1.0

Ατομικότητα (atomicity)

- Την είδαμε και σε προηγούμενη διάλεξη (το A στις ACID δοσοληψίες)
- Μια λειτουργία είναι "ατομική" όταν είτε ολοκληρώνεται επιτυχώς, είτε αποτυγχάνει πλήρως.
- Τα αποτελέσματά της, αν επιτύχει, γίνονται διαθέσιμα με την ολοκλήρωσή της (δεν μπορεί ούτε να "μείνει στη μέση", ούτε να παράγει εσφαλμένα ενδιάμεσα αποτελέσματα ορατά προς τρίτους).

Κουίζ

Είναι thread-safe η παρακάτω υλοποίηση (μπορεί να εκτελεστεί από πολλά νήματα χωρίς προβλήματα thread interference ή memory inconsistency);

```
public class Counter {

  private int counter;

  public void increase() {
    counter++;
  }

  public void decrease() {
    counter--;
  }

  public int get() {
    return counter;
  }
}
```

Όχι

Γιατί;

Διότι οι λειτουργίες `counter++` και `counter--` δεν είναι ατομικές!

Όντως

`counter++`

1. Διαβάζει την τρέχουσα τιμή του counter
2. Την αυξάνει κατά ένα
3. Και αποθηκεύει τη νέα τιμή στον counter

Μας καλύπτει το volatile keyword;

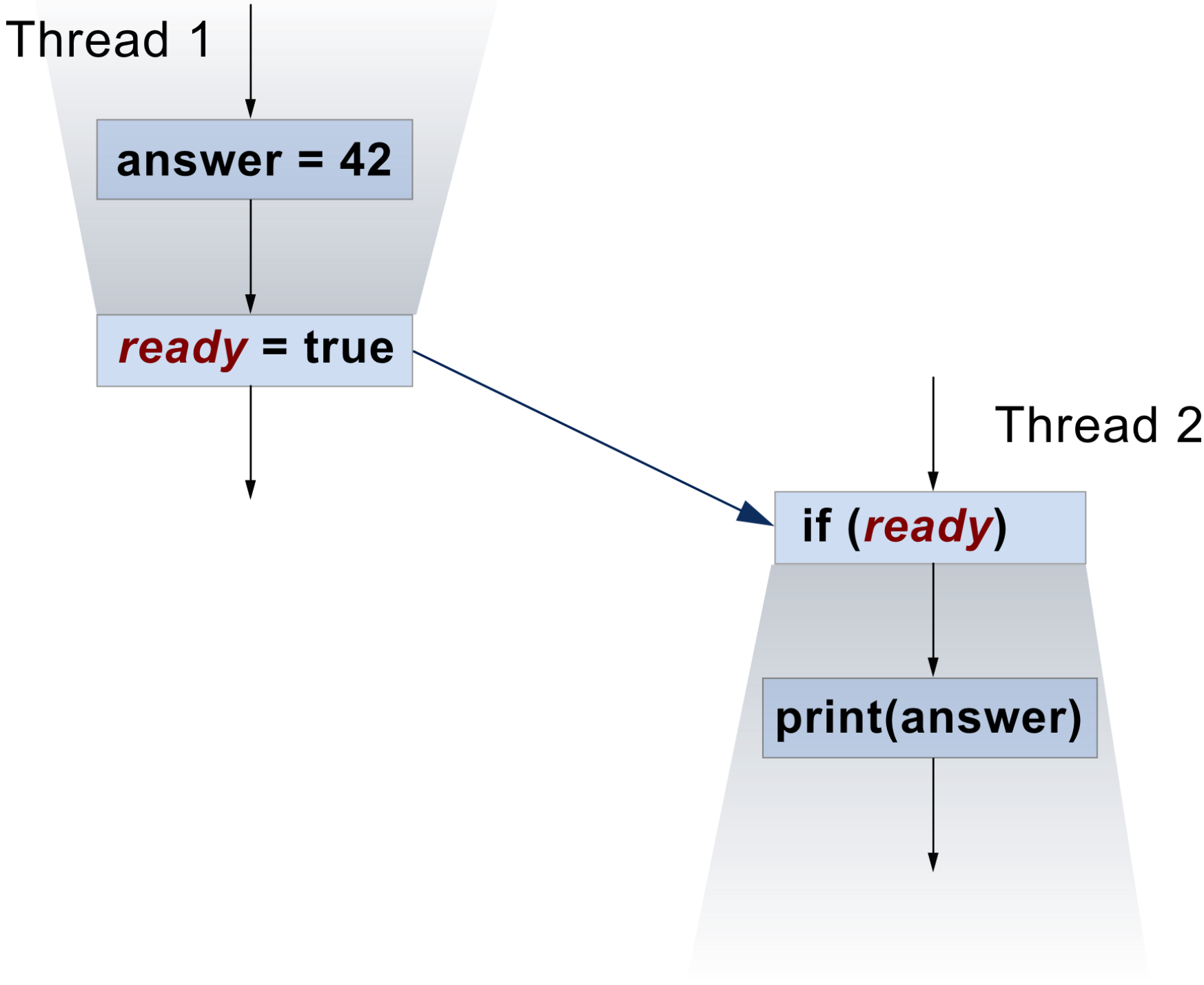
Volatile πεδία

- Η τιμή ενός volatile πεδίου δεν θα αποθηκευθεί προσωρινά στην τοπική μνήμη των νημάτων (thread-local cache), αλλά θα δρομολογηθεί απευθείας στην κύρια μνήμη.
- Ένδειξη ότι όλες οι αναγνώσεις/εγγραφές τιμών θα πρέπει να είναι ατομικές.
- Οι αλλαγές σε volatile μεταβλητές είναι πάντα ορατές από τα άλλα νήματα.

Παράδειγμα

- Έστω δύο νήματα t1, t2
- Ένα volatile πεδίο `ready`
- Ένα κανονικό πεδίο `answer`
- Το t1 γράφει στα πεδία
- Το t2 διαβάζει τα πεδία

http://jeremymanson.blogspot.gr/2008/11/what-volatile-means-in-java.html


Thread 1
answer = 42
ready = true
Thread 2
if (ready)
print(answer)

Χρήση του volatile

- Όταν το σύνολο των πιθανών τιμών που είναι να γραφούν στη μεταβλητή δεν εξαρτώνται από άλλες τιμές στο πρόγραμμα (συμπεριλαμβανομένης της τωρινής τιμής της μεταβλητής)

Παράδειγμα (Worker)

```
public class Worker extends Thread {
  private volatile boolean exit;
  void exit() {
    exit = true;
    System.out.println("Exit is set to true");
  }

  public void run() {
    while(!exit) {
      System.out.println(this.getName() + " is working!");
      try {
        Thread.sleep(1000); //compute something
      }
      catch(InterruptedException ie) {
        return; //interrupted on sleep
      }
    }
    //graceful completion
    System.out.println(this.getName() + ": Worker exits");
  }
}
```

Αποφυγή χρήσης του volatile

- Σε arrays - η αναφορά στο array θα είναι volatile κι όχι οι τιμές του
- Σύνθετες λειτουργίες που απαιτούν τον αποκλεισμό της πρόσβασης στη μεταβλητή (ή μεταβλητές) καθ όλη τη διάρκειά τους
- Σε Read-Update-Write λειτουργίες, όπως είναι η αύξηση ενός μετρητή (γιατί;)

Synchronized keyword

Μηχανισμός που διασφαλίζει την αποκλειστική πρόσβαση ενός νήματος ανά φορά σε συγκεκριμένα σημεία του κώδικα (critical sections)

```
public void synchronized foo() { ... }
```

ή

```
public void foo() {
  ...
  synchronized(object) {
    ...
  }
  ...
}
```

Πως λειτουργεί

- Κάθε στιγμιότυπο (και κάθε φορτωμένη κλάση) στο JVM συνδέεται με ένα lock/monitor
- Ο κώδικας μέσα στο synchronized block προστατεύεται αυτόματα από τον compiler με ειδικές κλήσεις που αποκτούν (acquire) το lock πριν ξεκινήσει η εκτέλεση και το απελευθερώνουν (release) με την ολοκλήρωσή της (κανονικής ή μη)

Επίσης

- Στο χρονικό διάστημα ανάμεσα στην απόκτηση και στην απελευθέρωση του lock, το νήμα λέμε ότι είναι ιδιοκτήτης του (owner)
- Αν κάποιο νήμα εισέλθει στο ίδιο synchronized block και δεν έχει το lock, θα πρέπει να περιμένει τον ιδιοκτήτη του lock να το απελευθερώσει
- Τα νήματα μπορούν να ξανα-αποκτήσουν το ίδιο lock (re-entrant locking)

Παράδειγμα

Κλάση που είναι σχεδιασμένη να είναι thread-safe.

```
public class ThreadSafeCounter {

  private int counter;

  public synchronized void increase() {
    counter++;
  }

  public synchronized void decrease() {
    counter--;
  }

  public synchronized int get() {
    return counter;
  }
}
```

Εναλλακτικά

Χρήση του syncrhonized keyword για να διασφάλισουμε thread-safe χειρισμό σε κλάσεις που δεν το υποστηρίζουν.

```
void someMethod(Counter counter) {

  synchronized(counter) {
    //thread safe
  }

}
```

Syntactic sugar

```
public class ThreadSafeCounter2 {

  private int counter;

  public void increase() {
    synchronized(this) {
      counter++;
    }
  }

  public void decrease() {
    synchronized(this) {
      counter--;
    }
  }

  public int get() {
    synchronized(this) {
      return counter;
    }
  }
}
```

Better be safe than sorry

Τα syncrhonization bugs είναι από τα πιο δύσκολα στον εντοπισμό και στη διόρθωσή τους!

Ασφαλής κανόνας: αν ένα αντικείμενο είναι ορατό σε πολλά νήματα, όλες οι αναγνώσεις και οι εγγραφές στις μεταβλητές του να γίνονται μέσα από synchronized μεθόδους.

Μη αποδοτικός: κλειδώνοντας όλο το αντικείμενο, μειώνουμε τον υποστηριζόμενο παραλληλισμό!

Αδιέξοδο (deadlock)

Η κατάσταση στην οποία δύο ή περισσότερα νήματα περιμένουν το ένα το άλλο επ' άπειρον!

Συνθήκες δημιουργίας αδιεξόδου

1. **Mutual exlusion**: Αμοιβαίος αποκλεισμός νημάτων για την πρόσβαση ενός πόρου
2. **Hold & wait**: Ένα νήμα κατέχει έναν πόρο και περιμένει να αποκτήσει έναν άλλο
3. **No-preemption**: Ένας πόρος που έχει κλειδώσει δεν μπορεί να "επιστραφεί"
4. **Circular wait**: Το ένα νήμα περιμένει την απελευθερώση κάποιου πόρου που κρατείται από άλλο

Πρέπει να ισχύουν και οι τέσσερις

Παράδειγμα

```
public static void main(String[] args) {
  BankAccount acc1 = new BankAccount(1, 1000.0d);
  BankAccount acc2 = new BankAccount(2, 500.0d);

  BankTransfer t1 = new BankTransfer(acc1, acc2, 100.0d);
  BankTransfer t2 = new BankTransfer(acc2, acc1, 200.0d);

  t1.start();
  t2.start();
}
```

Thread-safe BankAccount

```
public final class BankAccount {
  private double balance;
  public final int id;

  public BankAccount(int id, double initialBalance) {
    this.id = id;
    this.balance = initialBalance;
  }

  public synchronized void deposit(double amount) {
    balance = balance + amount;
  }

  public synchronized void withdraw(double amount) {
    balance = balance - amount;
  }

  @Override
  public String toString() { return "Account" + id; }
}
```

Deadlock-prone BankTransfer

```
public class BankTransfer extends Thread{
  private BankAccount src, dest;
  private double amount;

  public BankTransfer(BankAccount s,BankAccount d,double a){
    super(s + "-to-" + d);
    this.src = s;
    this.dest = d;
    this.amount = a;
  }
```

```
  public void run() {
    System.out.println(
      "Transfering from " + src + " to " + dest
    );

    synchronized(src) { //lock 1

      try { Thread.sleep(100); } //allow deadlock to occur
      catch(InterruptedException ie) {}

      synchronized(dest) { //lock 2 (nested)

        System.out.println(
          "Withdrawing " + amount + " from " + src
        );
        src.withdraw(amount);

        System.out.println(
          "Depositing " + amount + " into " + dest
        );
        dest.deposit(amount);
      }
    }
  }
}
```

Για να το δούμε

Αποφυγή αδιεξόδου

Συγχρονίζουμε με μια συγκεκριμένη σειρά κι όχι τυχαία!

Deadlock-free BankTransfer

```
public void run() {
  //Acquire locks based on the bank account's id
  BankAccount first, second;
  if (src.id < dest.id) {
    first = src;
    second = dest;
  }
  else {
    first = dest;
    second = src;
  }

  synchronized(first) { //ordered locking

    try{Thread.sleep(100);} catch(InterruptedException ie) {}

    synchronized(second) { //ordered locking
      src.withdraw(amount);
      dest.deposit(amount);
    }
  }
}
```