Java Threads

Computer Science Department, University of Crete

Parallel Programming

Βασίζεται σε slides των J. Foster, M. Hicks, D. Holmes, D. Lea

Τί είναι Thread

- Διαισθητικά:
  - Ένας από τους πολλούς παράλληλους υπολογισμούς που γίνονται εντός μιας διεργασίας
- Υλοποίηση:
  - Ένας καταχωρητής Program Counter και ένα stack
  - Το heap και οι static περιοχές μνήμης είναι κοινές μεταξύ όλων των threads του process
- Όλα τα προγράμματα έχουν τουλάχιστον ένα thread (`main()`)

Υλοποίηση των Threads

- Ένας program counter και μια στοίβα
  - Ο stack pointer και ο program counter αποθηκεύονται στη μνήμη όσο το thread δεν εκτελείται
  - Περιέχονται σε hardware registers (esp, eip) κάποιου πυρήνα όσο το thread εκτελείται

Πλεονεκτήματα-Μειονεκτήματα

- Τα threads μπορούν να αυξήσουν την ταχύτητα
  - Δημιουργία παραλληλισμού σε παράλληλους επεξεργαστές
  - Καλός τρόπος να γίνουν ταυτόχρονα το I/O και computation
- Φυσικό για πολλά προγραμματιστικά μοντέλα
  - Event processing
  - Simulations
- Μειονεκτήματα: Αυξημένη πολυπλοκότητα
  - Ο προγραμματιστής πρέπει να σκέφτεται για safety, liveness, composability
  - Κοινό heap, σύνθετα interleavings και σειρές εκτέλεσης
- Αυξημένη χρήση πόρων
  - Μπορεί να γίνει πρόβλημα με oversubscription

Thread Programming Model

- Threads υπάρχουν σε πολλές γλώσσες
  - C, C++, C#, Java, Smalltalk, Objective Caml, F#, ...
- Σε κάποιες γλώσσες (π.χ., C, C++) τα threads υπάρχουν ως add-on library
  - Δεν είναι μέρος του language specification
  - Για αναλυτική παρουσίαση των προβλημάτων, δείτε το paper: “Threads Cannot be Implemented as a Library” (προαιρετικό)
- Στη γλώσσα Java τα threads είναι μέρος του language specification
  - Υπάρχει μαθηματικοποίηση του “Java Memory Model” με τυπικό ορισμό

Java Threads

- Κάθε πρόγραμμα έχει τουλάχιστον ένα thread, τη main
  - Ξεκινά από το JVM για να εκτελέσει τη μέθοδο `main()` της εφαρμογής
- Το `main()` thread μπορεί να δημιουργήσει περισσότερα threads
  - Άμεσα: χρησιμοποιώντας την `Thread` class
  - Έμμεσα: καλώντας βιβλιοθήκες που χρησιμοποιούν threads
    - RMI, Applets, Swing/AWT, ...

Τα Java Threads ως Objects

- Η Java είναι Object Oriented γλώσσα
  - Χρησιμοποιεί ΟΟ για να εκφράσει τα threads
  - Όπως και οι περισσότερες ΟΟ γλώσσες προγραμματισμού
- Για να δημιουργήσετε ένα νέο Java Thread:
  - Instantiate ένα αντικείμενο `Thread`
    - Αντικείμενο της κλάσης `Thread` *ή* οποιουδήποτε subclass του `Thread`
  - Καλέστε τη μέθοδο `start()` του αντικειμένου
    - Αυτό θα δημιουργήσει νέο thread
    - Το νέο thread θα ξεκινήσει (ασύγχρονα) εκτελώντας τη μέθοδο `run()` του
    - Η εκτέλεσή του θα γίνει παράλληλα με το “parent” thread
  - Το νέο thread τερματίζει όταν τελειώσει η μέθοδος `run()` του

Παράδειγμα: Alarms

- Στόχος: ορισμός alarms που θα ενεργοποιηθούν στο μέλλον
  - Input: χρόνος σε `t` δευτερόλεπτα, το μήνυμα `m` που θα τυπωθεί
  - Result: το μήνυμα `m` θα τυπωθεί μετά από `t` δευτερόλεπτα

Παράδειγμα: Synchronous Alarms

```
...
while (true) {
  System.out.print("Alarm> ");

  // read user input
  String line = b.readLine();
  parseInput(line); // sets timeout

  // wait (seconds)
  try {
    Thread.sleep(timeout * 1000);
  } catch (InterruptedException e) { }
  System.out.println("(" + timeout + ") " + msg);
}
...
```

Παραλληλοποίηση

```
public class AlarmThread extends Thread {
  private String msg = null;
  private int timeout = 0;

  public AlarmThread(String msg, int time) {
    this.msg = msg;
    this.timeout = time;
  }

  public void run() {
    try {
      Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) { }
    System.out.println("(" + timeout + ") " + msg);
  }
}
```

Παραλληλοποίηση

```
...
while (true) {
  System.out.print("Alarm> ");

  // read user input
  String line = b.readLine();
  parseInput(line); // sets timeout

  if (m != null) {
    // start alarm thread
    Thread t = new AlarmThread(msg, timeout);
    t.start();
  }
}
...
```

Εναλλακτική: Runnable Interface

- Η κληρονομικότητα από την κλάση `Thread` αποκλείει διαφορετικό parent class
- Αντί για class extend, υλοποίηση του interface `Runnable`
  - Δηλώνει ότι η κλάση έχει μια μέθοδο `void run()`
- Δημιουργία ενός `Thread` από ένα `Runnable`
  - Constructor `Thread(Runnable target)`
  - Constructor `Thread(Runnable target, String name)`

Παράδειγμα

```
public class AlarmRunnable implements Runnable {
  private String msg = null;
  private int timeout = 0;

  public AlarmRunnable(String msg, int time) {
    this.msg = msg;
    this.timeout = time;
  }

  public void run() {
    try {
      Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) { }
    System.out.println("(" + timeout + ") " + msg);
  }
}
```

Παράδειγμα

```
...
while (true) {
  System.out.print("Alarm> ");

  // read user input
  String line = b.readLine();
  parseInput(line); // sets timeout

  if (m != null) {
    // start alarm thread
    Thread t = new Thread(new AlarmRunnable(msg, timeout));
    t.start();
  }
}
...
```

Πέρασμα παραμέτρων

- Η μέθοδος `run()` δεν δέχεται παραμέτρους
- Για να "περάσουμε παραμέτρους" στο νέο thread, αποθηκεύουμε τα δεδομένα ως private fields
  - Στο extended class
  - Στο `Runnable` object
  - Για παράδειγμα, τα `timeout` και `msg` είναι private fields του `AlarmThread` class

Concurrency

- *Concurrent* πρόγραμμα είναι αυτό που έχει πολλά threads ενεργά ταυτόχρονα
  - Μπορεί να εκτελείται σε ένα CPU
    - Ο επεξεργαστής αλλάζει μεταξύ threads
    - Ο Thread scheduler αποφασίζει
    - Context-switching μπορεί να συμβεί οποιαδήποτε στιγμή
  - Μπορεί να εκτελείται *παράλληλα* σε πολυπύρηνη μηχανή
    - Κάθε CPU core εκτελεί ένα thread
    - Μπορεί να υπάρχουν πάνω από ένα threads ανά CPU core
    - Τα Threads που είχαν διακοπεί μπορεί να συνεχίσουν στον ίδιο ή σε διαφορετικό CPU core
    - Η πολιτική του scheduler μπορεί να διαφέρει ανάλογα με το JVM

Concurrency και Κοινή Μνήμη

- Το concurrency είναι εύκολο αν τα threads δεν αλληλεπιδρούν
  - Κάθε thread χρησιμοποιεί δικά του αντικείμενα, κάνει τοπικά ό,τι θέλει
  - Τυπικά, τα threads πρέπει να επικοινωνήσουν
- Η επικοινωνία γίνεται έχοντας *shared* data
  - Πολλά threads μπορούν να έχουν πρόσβαση στο heap ταυτόχρονα
  - Η επικοινωνία γίνεται γράφοντας και διαβάζοντας τα ίδια αντικείμενα
  - Writes και reads μπορεί να γίνουν με οποιαδήποτε interleaved σειρά
    - Το hardware αλλάζει τη σειρά των instructions ή των μηνυμάτων μεταξύ hardware components
    - Ο scheduler μπορεί να προκαλέσει απρόβλεπτα interleavings
    - Ο compiler μπορεί να αλλάξει τη σειρά του κώδικα
    - Χρειάζεται προσοχή για την αποφυγή προβλημάτων!

Παράδειγμα Data Race σε Java Threads

```
public class Example extends Thread {
  private static int counter = 0; // shared state

  public void run() {
    int y = counter;
    counter = y + 1;
  }

  public static void main(String args[]) {
    Thread t1 = new Example();
    Thread t2 = new Example();
    t1.start();
    t2.start();
  }
}
```

Συγχρονισμός

- Μηχανισμοί που ελέγχουν τη σειρά εκτέλεσης μεταξύ threads
- “Φιλοσοφικά”:
  - Τα threads “παράγουν” όλα τα δυνατά executions με όλα τα δυνατά interleavings και timings
  - Κάποιες τέτοιες εκτελέσεις είναι σωστές, κάποιες όχι
  - Κάθε μηχανισμός συγχρονισμού έχει στόχο να “αφαιρέσει” λάθος εκτελέσεις από το χώρο των πιθανών εκτελέσεων
  - Περιορίζοντας τα πιθανά interleavings
- Διαφορετικές γλώσσες χρησιμοποιούν διαφορετικούς μηχανισμούς συγχρονισμού – μερικές φορές με το ίδιο όνομα!
- Η Java έχει αρκετούς τέτοιους μηχανισμούς

Java Locks

```
interface Lock {
  void lock();
  void unlock();
  ...
}

class ReentrantLock implements Lock { ... }
```

- Μόνο ένα thread μπορεί να κρατά το lock ανά πάσα στιγμή
  - Άλλα threads που προσπαθούν να πάρουν το ίδιο lock θα σταματήσουν μέχρι να είναι διαθέσιμο
- Reentrant lock: μπορεί να το ξαναπάρει το ίδιο thread που το έχει ήδη
  - Όσες φορές θέλει
  - Δεν μπορεί άλλο thread να κλειδώσει το lock μέχρι να γίνει release τον ίδιο αριθμό φορών που κλειδώθηκε
  - (Γι αυτό λέγεται “re-entry” (και χρειάζεται “re-exit”)

Παράδειγμα Συγχρονισμού με Locks

```
public class Example extends Thread {
  private static int counter = 0;
  static Lock lock = new ReentrantLock();

  public void run() {
    lock.lock();
    int y = counter;
    counter = y + 1;
    lock.unlock();
  }

  ...
}
```

Δεν υπάρχει συγχρονισμός μεταξύ διαφορετικών locks

```
static int counter = 0;
static Lock l = new ReentrantLock();
static Lock m = new ReentrantLock();

public void inc1() {
  l.lock();
  counter++;
  l.unlock();
}

public void inc2() {
  m.lock();
  counter++;
  m.unlock();
}
```

- Αυτό το πρόγραμμα έχει race condition
- Τα threads περιμένουν μόνο αν προσπαθήσουν να πάρουν lock που έχει άλλο thread

Ερώτηση

```
static int counter = 0;
static int x = 0;
```

Thread 1

```
while (x != 0) ;
x = 1;
counter++;
x = 0;
```

Thread 2

```
while (x != 0) ;
x = 1;
counter++;
x = 0;
```

Ερώτηση

```
static int counter = 0;
static int x = 0;
```

Thread 1

```
while (x != 0) ;
x = 1;
counter++;
x = 0;
```

Thread 2

```
while (x != 0) ;
x = 1;
counter++;
x = 0;
```

- Ένα thread μπορεί να σταματήσει μετά το `while` αλλά πριν γράψει στο `x`
- Θα υπάρχουν δύο threads που νομίζουν ότι έχουν το “lock”!
- Αυτό λέγεται και busy waiting: καταναλώνει κύκλους του επεξεργαστή περιμένοντας

Παράδειγμα Reentrant Lock

```
static int c = 0;
static Lock l =
     new ReentrantLock();

void inc() {
  l.lock();
  c++;
  l.unlock();
}
```

```
void returnAndInc() {
  int temp;

  l.lock();
  temp = c;
  inc();
  l.unlock();
}
```

- Το reentrancy είναι χρήσιμο επειδή κάθε μέθοδος μπορεί να κάνει acquire/release τα locks που χρειάζεται ανεξάρτητα
  - Δεν χρειάζεται ο προγραμματιστής να ξέρει αν αυτός που καλεί τη μέθοδο έχει ήδη το lock
  - Ο κώδικας προκύπτει πιο απλός, ευανάγνωστος

Deadlock

- Προκύπτει deadlock όταν κανένα thread δεν μπορεί να προχωρήσει επειδή όλα τα threads περιμένουν για κάποιο lock
- Κανένα thread δεν εκτελείται, άρα κανένα thread δεν μπορεί να αφήσει το lock του για να επιτρέψει σε άλλο να προχωρήσει

```
Lock l = new ReentrantLock();
Lock m = new ReentrantLock();
```

Thread 1

```
l.lock();
m.lock();
...
m.unlock();
l.unlock();
```

Thread 2

```
m.lock();
l.lock();
...
l.unlock();
m.unlock();
```

Deadlock

- Μερικές εκτελέσεις προχωρούν κανονικά
  - Το thread 1 εκτελείται ως το τέλος, και μετά εκτελείται το thread 2
- Πρόβλημα όταν
  - Thread 1 acquires `l`
  - Thread 2 acquires `m`
- Deadlock:
  - Το thread 1 προσπαθεί να κλειδώσει το `m`
  - Το thread 2 προσπαθεί να κλειδώσει το `l`
  - Κανένα δεν προχωρά, επειδή το άλλο thread έχει το lock

Ο γράφος των wait

- Wait graph
  - Κάθε thread είναι ένας κόμβος
  - Κάθε lock είναι ένας κόμβος
  - Η ακμή από το `l` στο `Thread1` υπάρχει αν έχει το lock
  - Δημιουργείται ακμή `Thread1` στο `m` όταν προσπαθήσει να κλειδώσει το lock
  - Ο γράφος wait αναπαριστά ένα σημείο της εκτέλεσης
- Υπάρχει deadlock όταν υπάρχει κύκλος
- Το πρόγραμμα έχει deadlock αν οποιαδήποτε εκτέλεση θα μπορούσε να προκαλέσει κύκλο
- Δύσκολο να αναπαραχθεί, δύσκολο στο debug

Παράδειγμα Deadlock

```
static Lock l = new ReentrantLock();

void f() throws Exception {
  l.lock();
  FileInputStream f = new FileInputStream("file.txt");
  // do something with f
  f.close();
  l.unlock();
}
```

- Το lock `l` δεν γίνεται release σε όλα τα πιθανά execution paths
- Μπορεί να συμβεί File Exception και να μείναι acquired από το thread
  - Πολύ πιθανό να προκαλέσει deadlock αργότερα
  - Ακόμη δυσκολότερο να γίνει debug
  - Το deadlock θα εμφανιστεί σε πιθανώς άσχετο σημείο της εκτέλεσης

Λύση: “finally”

```
static Lock l = new ReentrantLock();

void f() throws Exception {
  l.lock();
  try {
    FileInputStream f = new FileInputStream(”file.txt”);
    // do something with f
    f.close();
  }
  finally {
    // this code is executed always,
    // regardless of how we exit the try block
    l.unlock();
  }
}
```

Synchronized blocks

- Πολύ συχνή χρήση
  - Κλειδώνω το lock, κάνω κάτι, ξεκλειδώνω το lock σε κάθε περίπτωση (π.χ., `finally`)
- Η Java έχει ειδικό language construct γι αυτό το pattern χρήσης
  - `synchronized (obj) { body }`
    - Κάθε Java object έχει ένα lock
  - Κλειδώνεται το lock του αντικειμένου `obj`
  - Εκτελείται το `body`
  - Ξεκλειδώνεται το lock με την έξοδο από το syntactic scope
    - Ακόμη και σε περίπτωση exception ή explicit return

Παράδειγμα

```
static Object o = new Object();

void f() throws Exception {
  synchronized (o) {
    FileInputStream f = new FileInputStream("file.txt");
    // do something with f
    f.close();
  }
}
```

- Το lock του αντικειμένου o κλειδώνεται πριν εκτελεστεί το block
  - Ξεκλειδώνεται βγαίνοντας από το block scope, ακόμη και με exception

Object locks

- Προσοχή: Το αντικείμενο και το lock που του αντιστοιχεί είναι διαφορετικά πράγματα!
- Το ότι το lock ενός αντικειμένου είναι κλειδωμένο δε σταματά άλλα threads από το να καλέσουν μεθόδους, να έχουν πρόσβαση στα πεδία, κλπ.

Παράδειγμα

```
class C {
  int counter;

  void inc() {
    synchronized (this) {
      counter++;
    }
  }
}
...
C c = new C();
```

Thread 1

```
c.inc();
```

Thread 2

```
c.inc();
```

- Το πρόγραμμα έχει data race;

Παράδειγμα

```
class C {
  int counter;

  void inc() {
    synchronized (this) {
      counter++;
    }
  }
}
...
C c = new C();
```

Thread 1

```
c.inc();
```

Thread 2

```
c.inc();
```

- Το πρόγραμμα έχει data race;
  - Όχι, και τα δύο threads κλειδώνουν locks στο ίδιο αντικείμενο πριν κάνουν access τα shared data

Παράδειγμα

```
class C {
  int counter;

  void inc() {
    synchronized (this) { counter++; }
  }

  void dec() {
    synchronized (this) { counter--; }
  }
}
...
C c = new C();
```

Thread 1

```
c.inc();
```

Thread 2

```
c.dec();
```

- Το πρόγραμμα έχει data race;

Παράδειγμα

```
class C {
  int counter;

  void inc() {
    synchronized (this) { counter++; }
  }

  void dec() {
    synchronized (this) { counter--; }
  }
}
...
C c = new C();
```

Thread 1

```
c.inc();
```

Thread 2

```
c.dec();
```

- Το πρόγραμμα έχει data race;
  - Όχι, και τα δύο threads κλειδώνουν locks στο ίδιο αντικείμενο πριν κάνουν access τα shared data

Παράδειγμα

```
class C {
  int counter;

  void inc() {
    synchronized (this) {
      counter++;
    }
  }
}
...
C c1 = new C();
C c2 = new C();
```

Thread 1

```
c1.inc();
```

Thread 2

```
c2.inc();
```

- Το πρόγραμμα έχει data race;

Παράδειγμα

```
class C {
  int counter;

  void inc() {
    synchronized (this) {
      counter++;
    }
  }
}
...
C c1 = new C();
C c2 = new C();
```

Thread 1

```
c1.inc();
```

Thread 2

```
c2.inc();
```

- Το πρόγραμμα έχει data race;
  - Όχι, τα threads κλειδώνουν διαφορετικά locks, αλλά γράφουν σε διαφορετικά αντικείμενα

Synchronized Methods

- Χρήση του synchronized keyword για ορισμό μιας μεθόδου ως synchronized
  - Το ίδιο με συγχρονισμό πάνω στο lock του αντικειμένου `this` στον κώδικα της μεθόδου
  - Ευκολότερο να εκφραστεί το ίδιο pattern
- Τα παρακάτω προγράμματα έχουν το ίδιο νόημα:

```
class C {
  int counter;

  void inc() {
    synchronized (this) {
      counter++;
    }
  }
```

```
class C {
  int counter;

  synchronized void inc() {
    counter++;
  }
}
```

Παράδειγμα Synchronized Methods

```
class C {
  int counter;

  void inc() {
    synchronized (this) {
      counter++;
    }
  }

  synchronized void dec() {
    counter--;
  }
}
...
C c = new C();
```

Thread 1

```
c.inc();
```

Thread 2

```
c.dec();
```

Synchronized static methods

- Προσοχή: Οι static methods κλειδώνουν το class object!
  - Δεν υπάρχει `this` για να γίνει lock

```
class C {
  int counter;

  synchronized void inc() {
    counter++;
  }

  static synchronized void dec() {
    counter--;
  }
}
...
C c = new C();
```

Thread 1

```
c.inc();
```

Thread 2

```
c.dec();
```

Thread Scheduling

- Όταν πολλά threads μοιράζονται ένα CPU core
  - Πότε πρέπει να σταματήσει το thread που εκτελείται;
  - Ποιό thread πρέπει να είναι το επόμενο;
- Ένα java thread μπορεί να κάνει `yield()` το CPU core που έχει
  - Η κλήση στη `yield()` μπορεί να αγνοηθεί από το JVM
- Preemptive schedulers
  - Μπορούν να κάνουν de-schedule το thread που εκτελείται οποιαδήποτε στιγμή
  - Δεν χρησιμοποιούνται από όλα τα JVMs
  - Ένα thread που έχει κολλήσει σε επανάληψη μπορεί να μην κάνει ποτέ yield
  - Μερικές φορές είναι καλό να καλείται η `yield()` από τον κώδικα μέσα στα loops
- Τα threads γίνονται de-schedule όταν κάνουν block
- Lock, I/O, etc.

Thread Lifecycle

- Το thread που εκτελείται περνάει από διαφορετικές φάσεις
  - **New:** Δημιουργήθηκε αλλά δεν άρχισε ακόμη
  - **Runnable:** Εκτελείται αυτή τη στιγμή ή μπορεί να εκτελεστεί σε οποιοδήποτε ελεύθερο CPU core
  - **Blocked:** Περιμένει για I/O, lock, ή κάποιο άλλο synchronization operation
  - **Sleeping:** Paused για κάποιο διάστημα καθορισμένο από τον προγραμματιστή
  - **Terminated:** Έχει τελειώσει, δεν εκτελείται

Ποιό thread εκτελείται?

- Αναζήτηση σε όλα τα runnable threads
  - Υπάρχει κάποιο που μόλις έγινε unblocked?
    - Κάποιο lock έγινε release
    - I/O έγινε διαθέσιμο
    - Τελείωσε το sleep
- Διαλέγει ένα thread και το εκτελεί
  - Μπορούμε να επηρεάσουμε την προτεραιότητα με την `setPriority(int)`
  - Υψηλή τιμή προτεραιότητας σημαίνει ότι θα επιλεγεί πρώτα
  - Πιθανότατα δεν υπάρχει λόγος να καθοριστεί priority

Σημαντικές μέθοδοι του Thread

- `void join() throws InterruptedException`
  - Περιμένει να τελειώσει το thread
- `static void yield()`
  - Το τρέχων thread απελευθερώνει το CPU core
- `static void sleep(long milliseconds) throws InterruptedException`
  - Το τρέχων thread κάνει sleep για τον καθορισμένο χρόνο
- `static Thread currentThread()`
  - Επιστρέφει το `Thread` object του thread που εκτελείται

Παράδειγμα: Alarm

```
...
while (true) {
  System.out.print("Alarm> ");

  // read user input
  String line = b.readLine();
  parseInput(line); // sets timeout

  // wait (seconds) asynchronously
  if (msg != null) {
    // start alarm thread
    Thread t = new AlarmThread(msg, timeout);
    t.start();
    // wait for the thread to complete
    t.join();
  }
}
...
```

Daemon Threads

- `void setDaemon(boolean on)`
  - Ορίζει ότι ένα thread είναι daemon thread
  - Πρέπει να τεθεί πριν ξεκινήσει η εκτέλεση του thread
- Κάθε νέο thread αποκτά αυτόματα την κατάσταση του thread που το δημιούργησε
- Η εκτέλεση του προγράμματος τερματίζει όταν δεν μείνουν threads που να τρέχουν
  - Εκτός από τα daemon threads

Βασικές Αρχές

- Πολλά threads που εκτελούνται ταυτόχρονα
  - Είτε πραγματικά σε πολλά CPU cores
  - Είτε γίνονται schedule ώστε να χρησιμοποιούν έναν επεξεργαστή
    - Ένα thread που εκτελείται μπορεί να γίνει pre-empted οποιαδήποτε στιγμή
  - Ή συνδυασμός των παραπάνω
- Τα threads μπορούν να μοιράζονται δεδομένα
  - Στην Java, μόνο τα fields μπορούν να είναι shared
  - Πρέπει να αποφύγουμε το interference
    - Καλή πρακτική 1: Το thread κρατά ένα lock όταν γράφει ή διαβάζει shared δεδομένα
    - Καλή πρακτική 2: Το lock δεν γίνεται release μέχρι τα shared δεδομένα να ξαναβρεθούν σε valid state
  - Υπερβολική χρήση synchronization μπορεί να προκαλέσει deadlocks
    - Rule of thumb: Δεν μπορεί να υπάρξει deadlock αν κάθε thread κρατά ένα μόνο lock κάθε στιγμή

Producer – Consumer Design Pattern

- Έστω ότι δύο threads επικοινωνούν με μια κοινή μεταβλητή
  - Π.χ., κάποιου είδους buffer που κρατά μηνύματα
  - Ένα thread *παράγει* είσοδο στον buffer
  - Ένα thread *καταναλώνει* δεδομένα από τον buffer
  - Πώς υλοποιείται αυτό;
    - Με χρήση condition variables

Conditions

```
interface Lock { Condition newCondition(); ... }
interface Condition {
  void await();
  void signalAll();
   ...
}
```

Conditions

- Ένα `Condition` δημιουργείται χρησιμοποιώντας ένα `Lock` object
- Η `await()` καλείται με το lock να είναι *acquired*
  - Απελευθερώνει το lock
    - Αλλά όχι άλλα locks που κρατά αυτό το thread
  - Βάζει το τρέχον thread στο wait set γι αυτό το lock
  - Σταματά (Block) το thread
- Η `signalAll()` καλείται με το lock να είναι *acquired*
  - Ξεκινά όλα τα threads στο wait set του lock
  - Αυτά τα threads προσπαθούν να ξανακλειδώσουν το lock πριν συνεχίσουν
    - Μόνο ένα θα τα καταφέρει
    - Αν το thread που τα καταφέρει είχε γίνει block μέσα στην `await()` συνεχίζει με το lock να είναι *acquired*

Παράδειγμα: Producer – Consumer

```
Lock lock = ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;
```

Thread 1

```
void produce(Object o) {
  lock.lock();
  while (valueReady)
    ready.await();
  value = o;
  valueReady = true;
  ready.signalAll();
  lock.unlock();
}
```

Thread 2

```
Object consume() {
  lock.lock();
  while (!valueReady)
    ready.await();
  Object o = value;
  valueReady = false;
  ready.signalAll();
  lock.unlock();
}
```

Προτιμήστε αυτό το design pattern

- Είναι η σωστή λύση στο πρόβλημα
  - Μπορεί να φαίνεται πιο εύκολο να χρησιμοποιήσετε τα locks άμεσα στον κώδικα
  - Είναι πολύ δύσκολο να γίνει σωστά
  - Τα προβλήματα που προκύπτουν με άλλες υλοποιήσεις είναι συχνά subtle, όχι προφανή
    - Π.χ., το double-checked locking έχει bug

Παράδειγμα: BROKEN code (1)

```
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;
```

Thread 1

```
void produce(Object o) {
  lock.lock();
  while (valueReady);
  value = o;
  valueReady = true;
  lock.unlock();
}
```

Thread 2

```
Object consume() {
  lock.lock();
  while (!valueReady);
  Object o = value;
  valueReady = false;
  lock.unlock();
}
```

Παράδειγμα: BROKEN code (1)

```
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;
```

Thread 1

```
void produce(Object o) {
  lock.lock();
  while (valueReady);
  value = o;
  valueReady = true;
  lock.unlock();
}
```

Thread 2

```
Object consume() {
  lock.lock();
  while (!valueReady);
  Object o = value;
  valueReady = false;
  lock.unlock();
}
```

- Αυτό το πρόγραμμα είναι *λάθος*
- Deadlock: τα threads περιμένουν ενώ κρατούν το lock, δεν προχωρά κανείς

Παράδειγμα: BROKEN code (2)

```
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;
```

Thread 1

```
void produce(Object o) {
  while (valueReady);
  lock.lock();
  value = o;
  valueReady = true;
  lock.unlock();
}
```

Thread 2

```
Object consume() {
  while (!valueReady);
  lock.lock();
  Object o = value;
  valueReady = false;
  lock.unlock();
}
```

- Κι αυτό το πρόγραμα είναι *λάθος*
- Data Race: η πρόσβαση στη `valueReady` γίνεται χωρίς το thread να έχει το lock

Παράδειγμα: BROKEN code (3)

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;
```

Thread 1

```
void produce(Object o) {
  lock.lock();
  if (valueReady) ready.await();
  value = o;
  valueReady = true;
  ready.signalAll();
  lock.unlock();
}
```

Thread 2

```
Object consume() {
  lock.lock();
  if (!valueReady) ready.await();
  Object o = value;
  valueReady = false;
  ready.signalAll();
  lock.unlock();
}
```

- Κι αυτό το πρόγραμμα είναι *λάθος*!
- Correctness: Τί θα γίνει αν προκύψουν πάνω από ένα producer και consumer threads;

Condition Interface

```
interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  void signal();
  void signalAll();
  ...
}
```

- `await(t, u)` περιμένει για χρόνο `t` και μετά σταματά την αναμονή
  - Boolean result: `false` αν ο χρόνος αναμονής τελείωσε πριν να επιστρέψει η μέθοδος, αλλιώς `true`
- `signal()` ξυνπά μόνο *ένα* από τα thread που περιμένουν
  - Δύσκολο να εκτελεστεί σωστά
    - Προτιμότερο όλα τα threads να είναι ίσα, και να χειρίζονται exceptions σωστά
  - Προτιμότερη η χρήση του `signalAll()`

Θέματα στη χρήση await και signalAll

- Η `await()` *πρέπει* να είναι μέσα σε επανάληψη
  - Μην υποθέσετε ότι όταν επιστρέψει το condition είναι αληθές
  - Μπορεί να υπάρχουν πολλά threads που “καταναλώνουν” το condition
- Αποφύγετε να κρατάτε άλλα locks όσο ένα thread περιμένει
  - Η `await()` απελευθερώνει μόνο τα locks του αντικειμένου που περιμένει
- Δεν υπάρχει `Condition` object για δύο locks
- Μπορούμε να έχουμε δύο `Condition` objects στο ίδιο lock

Blocking Queues

- Interface του Producer–Consumer pattern

```
interface Queue<E> extends Collection<E> {
  boolean offer(E x); // produce
    // waits for queue to have capacity

  E remove();         // consume
    // waits for queue to become non-empty
   ...
}
```

- Δύο χρήσιμα implementations
  - `LinkedBlockingQueue` (FIFO, may be bounded)
  - `ArrayBlockingQueue` (FIFO, bounded)
  - Υπάρχουν κι άλλα, δείτε στο documentation

Wait και NotifyAll (1)

- Παλιότερο synchronization (Java 1.4)
- Στην Java 1.4, υπήρχε μόνο η `synchronized` για κλείδωμα lock
  - Κάθε αντικείμενο έχει ένα lock
  - Κάθε αντικείμενο έχει ένα *wait set*

Wait και NotifyAll (2)

- `o.wait()`
  - Πρέπει να είναι κλειδωμένο το lock του `o` (μέσα σε `synchronized` block)
  - Απελευθερώνει το lock
    - Κανένα άλλο
  - Προσθέτει το thread στο wait set του lock
  - Κάνει block το thread
  - Όταν επιστρέψει το lock θα είναι ξανά acquired
- `o.notifyAll()`
  - Πρέπει το lock του `o` να είναι κλειδωμένο
  - Ξεκινά όλα τα threads στο wait set του `o`
  - Αυτά θα προσπαθήσουν να ξανακλειδώσουν το lock πριν συνεχίσουν (πριν επιστρέψει η `wait()`)

Producer – Consumer στη Java 1.4

```
public class ProducerConsumer {
  private boolean valueReady = false;
  private Object value;

  synchronized void produce(Object o) {
    while (valueReady) wait();
    value = o;
    valueReady = true;
    notifyAll();
  }

  synchronized Object consume() {
    while (!valueReady) wait();
    valueReady = false;
    Object o = value;
    notifyAll();
    return o;
  }
}
```

InterruptedException

- Exception που προκύπτει αν κάποια concurrency operations διακοπούν
  - `wait()`, `await()`, `sleep()`, `join()`, and `lockInterruptibly()`
  - Επίσης προκύπτει αν κληθούν ενώ έχει γίνει set το interrupt flag
- Το exception *δεν* προκύπτει αν το thread περιμένει σε Java 1.4 lock ή σε I/O

InterruptedException

```
class Object {
  void wait() throws InterruptedException;
  ...
}
interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  ...
}
interface Condition {
  void await() throws InterruptedException;
  void signalAll();
  ...
}
```

Designing Objects for Concurrency

- Isolation
  - Avoid interference by not sharing
- Immutability
  - Avoid inteference by avoiding change
- Locking
  - Dynamically guarantee exclusive access
- Splitting Objects
  - Changing representation to facilitate concurrency control
- Containment
  - Guarantee exclusive control of internal components
  - Manage ownership
  - Protect unhidden components
- Alternatives to Synchronization
  - `volatile` variables and the Java Memory Model

Isolation

- Objects that are not shared cannot suffer interference
  - Heap objects accessible only from current thread
  - Parameters and local variables
    - Applies to *references*, not the objects to which they refer
  - `java.lang.ThreadLocal`
    - Simplifies access from other objects running in the same thread
  - No need for *any* synchronization
- Objects can be accessed by multiple threads as long as they are isolated to one thread at any given time
  - Transfer of ownership protocols
    - Thread 1 uses the object, hands off to Thread 2 and then never accesses the object again
  - Transfer still requires synchronization

Thread Local Data

- Suppose you want to run multiple web servers, each on one thread, each using a different document directory
- Could define a `documentRoot` field in the `WebServer` class
- Or, define the document root as a variable tied to each `Thread` object
  - The easiest way: use `java.lang.ThreadLocal`
  - Equivalent to adding instance variables to all `Thread` objects
  - No need to define subclasses or control thread creation
- All methods running can access thread local data when needed
  - Frequent use: package accessible statistics
- No interference when *all* accesses happen within the same thread

Example: ThreadLocal

```
public class WebServer {
  static final ThreadLocal documentRoot = new ThreadLocal();
  ...
  public WebServer(int port, File root) throws IOException {
    ...
    documentRoot.set(root);
  }

  private void processRequest(Socket sock) throws IOException {
    File  root = (File) documentRoot.get();
    ...
  }
  ...
}
...
```

When to use ThreadLocal

- Variables that apply *per activity*, not *per object*
  - E.g., timeout value, transaction ID, current dirctory, default parameters
- Replacement for `static` variables
  - When different threads should use different values
- Tools to eliminate the need for synchronization
  - Used internally in JVM to optimize memory allocation, lock implementations, etc.
  - E.g., per-thread caches, slabs

Stateless Objects

```
class StatelessAdder {
  int addOne (int i) { return i + 1; }
  int addTwo (int i) { return i + 2; }
}
```

- There are no special concurrency concerns
  - No per-instance state, therefore no storage conflicts
  - No data representation, therefore no representation invariants
  - Multiple concurrent executions, therefore no liveness problems
  - No interaction with other objects, therefore no requirement for synchronization protocol
- Example: `java.lang.Math`

Immutable Objects

```
class ImmutableAdder {
  private final int offset;
  ImmutableAdder(int offset) { this.offset = offset; }
  int add(int i) { return i + offset; }
}
```

- Object state frozen upon initialization
  - Still no safety or liveness concerns
  - No interference as per-instance state never changes
  - Java `final` fields enforce most senses of immutability
- Immutability often suitable for closed Abstract Data Types
  - E.g., `String`, `Integer`, etc.

Containment

- Strict containment creates islands of objects
  - Applies recursively
- Allows code of “inner” objects to run faster
  - Works with legacy sequential code
- Requires inner code to be *communication closed*
  - No unprotected calls into or out of island
- Requires outer objects to never leak inner references
  - Or uses ownership transfer protocol
- By convention, can be difficult to enforce and check

Example: Containment (1)

```
class Statistics { // Mutable!
  public long requests;
  public double avgTime;
  public Statistics(long requests, double avgTime) {
    this.requests = requests;
    this.avgTime = avgTime;
  }
}
```

- Fields are *public* and *mutable*
  - Therefore, instances cannot be shared
- Can be safely contained within a `WebServer` instance

Example: Containment (2)

```
class WebServer {
  ...
  private final Statistics stats = new Statistics(0, 0.0);
  public synchronized Statistics getStatistics() {
    return new Statistics(stats.requests, stats.avgTime);
  }
  private void processRequest(Socket sock) throws IOException {
    synchronized(this) {
      double total = stats.avgTime * stats.requests + elapsed;
      stats.avgTime = total / (++stats.requests);
    }
  }
}
```

- Cannot expose mutable state
  - Instead, make copies

Hierarchical Containment Locking

- Applies when logically contained parts are not hidden from clients
- Avoids deadlocks that could occur if parts were fully synchronized
- All parts use lock provided by the common owner
- Can use either internal or external conventions

Internal Containment Locking (1)

```
class Part {
  protected Container owner_ ; // Never null
  public Container owner() { return owner_; }
  private void bareAction() { /* unsafe */ }
  public void m() {
    synchronized (owner()) { bareAction(); }
  }
}
```

- Visible components protect themselves using their owner's locks
  - Parts do not deadlock when invoking each other's methods
  - Parts must be aware that they are contained

Internal Containment Locking (2)

```
class Container {
  class Part {
    ...
    public void m() {
      synchronized (Container.this) { bareAction(); }
    }
  }
}
```

- Implemented using inner classes
- Do not require `synchronized` blocks synchronization
  - Shared `Lock` objects
  - Transaction locks
  - etc.

External Containment Locking

```
class Client {
  void f(Part p) {
    synchronized (p.owner()) { p.bareAction(); }
  }
}
```

- External: rely on clients to provide locking (client-side)
- Used in AWT
  - `java.awt.Component.getTreeLock()`
- Can sometimes avoid more locking overhead
- ... at price of fragility
  - Can manually minimize use of `synchronized`
  - Requires all callers to obey convention
  - Effectiveness depends on context
    - Breaks encapsulation
    - Does not work with fancy schemes that do not rely on `synchronized` blocks or similar methods of locking

Subclassing Unsafe Code (1)

- Assume a method written in native code

```
class HandlerHelper {
  native void mountFileSystem();
}
```

- Suppose our method `processRequest` invokes `mountFileSystem();`

Subclassing Unsafe Code (2)

- We do not trust this class to be thread-safe
  - Wrap calls in `synchronized` blocks (i.e., containment)
  - Or, create a simple subclass that adds synchronization and instantiate that class instead

```
class SafeHandlerHelper extends HandlerHelper {
  synchronized void mountFileSystem() {
    super.mountFileSystem();
  }
}
```

  - Localizes synchronization control where it is required
- Subclassing is usually the most convenient way to do that
  - Can also use unrelated wrapper classes and delegate
  - Can generalize to “template method” schemes (later)

State Dependent Actions

- State Dependence
- Balking
- Guarded Suspension
- Optimistic Retries
- Specifying Policies

Examples of State Dependent Actions

- Operations on collections, streams, databases
  - Remove an element from an empty queue
  - Add an element to a full buffer
- Operations on objects maintaining constrained values
  - Withdraw money from an empty bank account
- Operations requiring resources
  - Print a file
- Operations requiring particular message orderings
  - Read an unopened file
- Operations on external controllers
  - Shift to reverse gear in a moving car

Policies for State Dependent Actions

- Policy choices for dealing with preconditions and postconditions
  - **Blind action**: Proceed anyway, no guarantee of outcome
  - **Inaction**: Ignore request if not in the right state
  - **Balking**: Fail via exception if not in the right state
  - **Guarding**: Suspend until in the right state
  - **Trying**: Proceed, check if successful, roll back if not
  - **Retrying**: Keep trying until successful
  - **Timeout**: Wait or retry for a while, then fail
  - **Planning**: First initiate activity that will achieve the right state
- How to convey policy in code?

Interfaces and Policies

```
public interface Buffer {
  int capacity();      // Inv: capacity() > 0
  int size();          // Inv: 0 ≤ size() ≤ capacity()
                       // Init: size() == 0
  void put(Object x); // Pre: size() < capacity()
  Object take();       // Pre: size() > 0
}
```

- Interfaces alone cannot convey policy
- Can suggest policy
  - E.g., should `take()` throw exception? What kind?
  - Different methods can support different policies for same base actions
- Can use manual annotations
  - Declarative constraints form the basis of the implementation

Balking

- Check state upon method entry
  - Must not change state in course of checking state
  - Relevant state must be explicitly represented
    - So it can be checked on entry
- Exit immediately if not in the right state
  - Throw exception or return special value
    - In these examples, throw `Failure`
  - Client is responsible for handling failure
- The simplest policy for synchronized objects
  - Useable in both sequential and concurrent contexts
    - Often used in `Collection` classes, e.g., `Vector`
  - In concurrent contexts the host must always take responsibility for entire check-act/check-fail sequence
    - Clients cannot preclude state changes between check and act, so host must control

Example: Balking Bounded Buffer

```
public Class BalkingBoundedBuffer implements Buffer {
  private List data;
  private final int capacity;
  public BalkingBoundedBuffer(int capacity) {
    data = new ArrayList(capacity);
    this.capacity = capacity;
  }
  public synchronized Object take() throws Failure {
    if (data.size() == 0) throw new Failure("Buffer Empty");
    Object temp = data.get(0);
    data.remove(0);
    return temp;
  }
  public synchronized void put(Object o) throws Failure {
    if (data.size() == capacity) throw new Failure("Buffer Full");
    data.add(o);
  }
  public synchronized int size() { return data.size(); }
  public int capacity() { return capacity; }
}
```

Guarding

- Generalization of locking for state dependent actions
  - Locked: wait until ready (not engaged in other methods)
  - Guarded: Wait until an arbitrary state predicate holds
- Check state upon entry
  - If not in right state, wait
  - Some other action in some other thread may eventually cause a state change that enables resumption
- Introduces liveness concerns
  - Relies on actions of other threads to make progress
- Useless in sequential programs
  - Client must ensure correct state before calling

Guarding Mechanisms: Busy wait

- Thread continually spins until a condition holds

```
while(!condition) ; // spin
// use condition
```

  - Requires multiple CPUs or timeslicing
    - No way to determine this until Java 1.4

```
int nCPUs = Runtime.availableProcessors();
```

  - But busy waiting can sometimes be useful
    - When the conditions *latch*: once true, they never become false

Guarding Mechanisms: Suspension (1)

- Thread stops execution until notified that the condition *may* be true
- Supported in Java via wait sets and locks

```
synchronized (obj) {
  while (!condition) {
    try { obj.wait(); }
    catch (InterruptedException e) { ... }
  }
  // use condition
}
```

Guarding Mechanisms: Suspension (2)

- Changing a condition

```
synchronized (obj) {
  condition = true;
  obj.notifyAll();   // or obj.notify()
}
```

  - Or after Java 1.5, using `Lock` and `Condition`
- Golden rule: always test a condition in a loop
  - Change of state may not be what you need
  - Condition may have changed again
  - Break the rule only after *proving* it's safe

Wait sets and Notification (1)

- Every Java `Object` has a wait set
  - Can only be manipulated while the object lock is held
  - Otherwise, `IllegalMonitorStateException`
- Threads enter the wait set by calling `wait()`
  - `wait()` atomically releases the lock and suspends the thread
    - Including re-entrant locks held multiple times
    - *No other* held locks are released
  - Timed waiting via `wait(long milliseconds)`
    - No direct indication that a time-out occured
    - `wait()` and `wait(0)` mean wait forever
    - Nanosecond version too
- Similar for explicit `Lock` objects after Java 1.5
  - Differences in versatility: interruption, timeout notification, separate acquire – release, etc.

Wait sets and Notification (2)

- Threads are released from the wait set when
  - `notifyAll()` invoked on the object (`signalAll()` invoked on the condition)
    - Releases all threads
  - `notify()` invoked on the object (`signal()` invoked on the condition)
    - Releases one thread selected at “random”
  - The specified timeout has elapsed
  - `interrupt()` method called for current thread, causes `InterruptedException`
  - Spurious wakeup occurs when:
    - Inherited property of underlying synchronization mechanisms: POSIX threads, Windows threads, Hardware threads, etc.
- Lock is always reacquired before `wait()` returns
  - Restored lock count for re-entrant locks
  - Cannot be acquired until notifying thread releases it
  - All released threads contend for the lock

Wait sets and Notification (3)

- Avoid `notify()` (and `signal()`), only use for optimization when *all* the following hold:
  - Only one thread can benefit from the change of state
  - All threads are waiting for the same change of state
    - or else, another `notify()` is done by the released thread
  - And these conditions also hold for *all subclasses*!
- Conditional notification is another optimization
  - When you *know* for what state changes the other threads wait
  - Warning: subclasses may invalidate your “knowledge”
- Use of `wait()`, `notifyAll()`, `notify()` are similar to
  - Condition queues of classic Monitors
  - Condition variables of POSIX threads
  - But, with only one queue per object
    - May complicate some designs and lead to *nested monitor lockouts*
- Any Java object can be used just for its wait set and lock
  - After 1.5, use `Lock` objects

Example: Guarded Bounded Buffer

```
public class GuardedBoundedBuffer implements Buffer {
  private List data;
  private final int capacity;

  public GuardedBoundedBuffer(int capacity) {
    data = new ArrayList(capacity);
    this.capacity = capacity;
  }
  public synchronized Object take() throws Failure {
    while (data.size() == 0)
      try { wait(); }
      catch (InterruptedException e) { throw new Failure(); }
    Object temp = data.get(0);
    data.remove(0);
    notifyAll();
    return temp;
  }
  public synchronized void put(Object obj) throws Failure {
    while (data.size() == capacity)
      try { wait(); }
      catch (InterruptedException e) { throw new Failure(); }
    data.add(obj);
    notifyAll();
  }
  public synchronized int size() { return data.size(); }
  public int capacity() { return capacity; }
}
```

Timeout

- Intermediate points between Balking and Guarding
  - Can vary timeout parameter from zero to infinity
- Useful for heuristic detection of failures
  - Deadlocks, crashes, I/O problems, network disconnections
- But cannot be used for high-precision timing or deadlines
  - Time can elapse between wait and thread resumption
  - Time can elapse after checking the time!
- Java implementation constraints
  - `wait(ms)` does not automatically tell you if it returs because of notification or timeout
    - `await(ms)` does

Optimistic Techniques

- Variations for recording versions of mutable data
  - Immutable helper classes
  - Version numbers
  - Transaction IDs
  - Time stamps
- May be more efficient than guarded waiting
  - When conflicts are rare and running on multiple CPUs
- Retrying can livelock unless *proven* wait-free
  - Analogous to deadlock in guarded waiting
  - Should arrange to fail after a certain time or number of attempts

Example: Optimistic Bounded Counter

```
public class OptimisticBoundedCounter {
  private final long MIN, MAX;
  private Long count; // MIN <= count <= MAX

  public OptimisticBoundedCounter(long min, long max) {
    MIN = min; MAX = max;
    count = new Long(MIN);
  }
  public long value() { return count().longValue(); }
  public synchronized Long count() { return count; }

  private synchronized boolean commit(Long oldc, Long newc) {
    boolean success = (count == oldc);
    if (success) count = newc;
    return success;
  }
  public void inc() throws InterruptedException {
    for (;;) { // retry-based
      if (Thread.interrupted())
        throw new InterruptedException();
      Long c = count();
      long v = c.longValue();
      if (v < MAX && commit(c, new Long(v+1)))
        break;
      Thread.yield(); // a good idea in spin loops
    }
  }
  public void dec() { /* symmetrical */ }
```

Specifying Policies

- Some policies are per-type
  - Optimistic approaches require all methods to conform
- Some policies can be specified per-call
  - Balking *vs.* Guarding *vs.* Guarding with time-out
- Options for specifying per-call policy
  - Extra parameters
    - `void put(Object x, long timeout)`
    - `void put(Object x, boolean balk)`
  - Different name for Balking or Guarding
    - Balking: `void tryPut(Object x)`
    - Guarding: `void put(Object x)`
  - May need different exception signatures