Διάλεξη 22: Συγχρονισμός (Concurrency)

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Διεργασίες (processes) και Νήματα (threads)
- Συγχρονισμός Νημάτων, Προβλήματα, Λύσεις
- Οι τροποποιητές <synchronized>, <volatile>
- Διαχείριση Συγχρονισμού

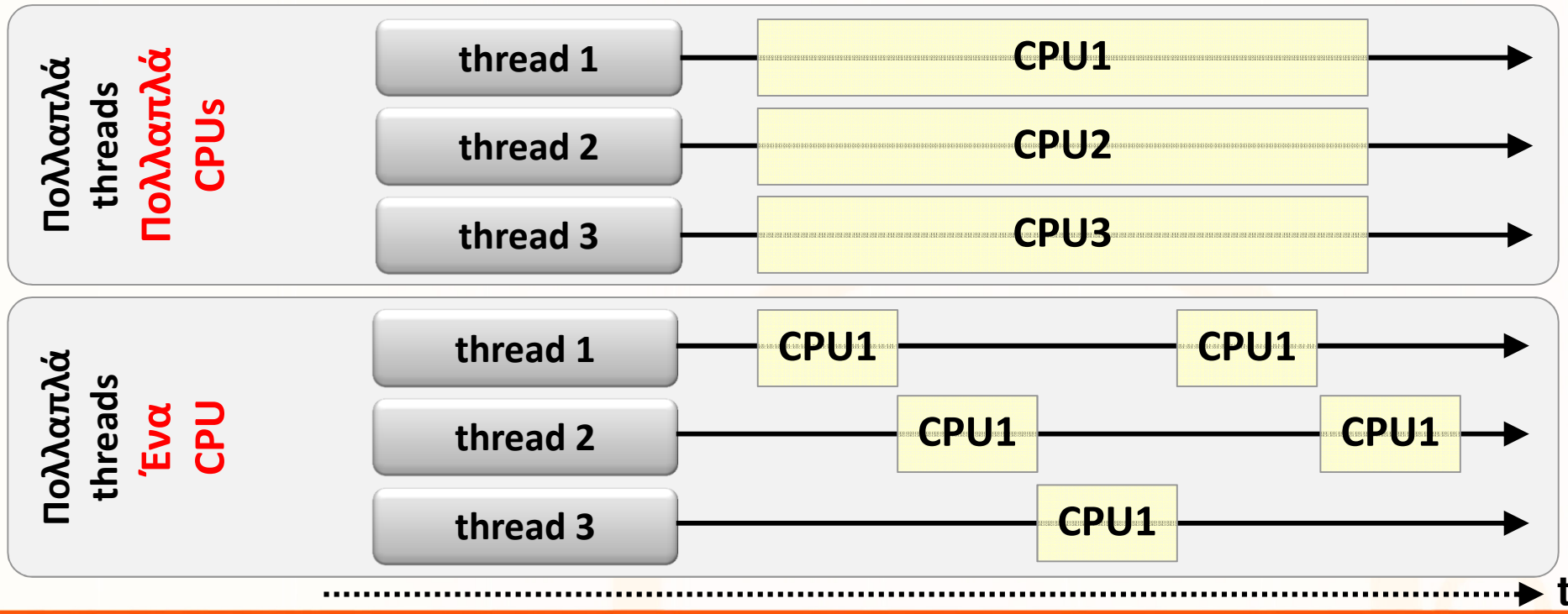
Διδάσκων: Παναγιώτης Ανδρέου

Εισαγωγή

- Τι είναι ένα νήμα?
  - Νήμα αποκαλείται η ελάχιστη οντότητα επεξεργασίας που μπορεί να δρομολογηθεί από το λειτουργικό σύστημα. Σε κάποια λειτουργικά συστήματα αποκαλούνται και Lightweight Processes
- Ποίο είναι το πιο κοινό πρόβλημα που μπορεί να προκύψει με την ταυτόχρονη εκτέλεση διεργασιών?
  - Η ταυτόχρονη πρόσβαση σε κοινούς πόρους του συστήματος
- Πως αντιμετωπίζεται?
  - Με τεχνικές αμοιβαίου αποκλεισμού των προσβάσεων.
- Ποια λύση προτείνεται από την Java?
  - Αρκετές, ανάμεσα στις οποίες είναι οι: Synchronized methods, Synchronized blocks, Atomic access, Locks, …

Νήματα (Threads)

- Το νήμα (thread) αποτελεί μία μοναδική, ακολουθιακή ροή ελέγχου μέσα σε ένα πρόγραμμα.
- Στα σύγχρονα συστήματα βρίσκονται σχεδόν σε όλες τις εφαρμογές (π.χ., DOS vs. Windows)
- Τα threads μπορούν να τρέχουν σε ένα υπολογιστή ή σε πολλαπλούς υπολογιστές



Διεργασίες (processes) vs. Νήματα (threads)

- Στον **συντρέχον προγραμματισμό (concurrent programming)**, υπάρχουν δύο βασικά είδη εκτέλεσης: **διεργασίες και νήματα**.
- Ένα σύστημα έχει τυπικά **πολλά ενεργά processes and threads**.
- Αν υπάρχει **μόνο ένας επεξεργαστής**, τότε ο **χρόνος (και πόροι) του μοιράζονται** στα processes and threads μέσω της τεχνικής **time slicing** που παρέχεται από το λειτουργικό σύστημα.
- Διεργασία (process)
  - Ένα process έχει το δικό της αυτοδύναμο περιβάλλον εκτέλεσης: ξεχωριστό κομμάτι μνήμης και προσωπικούς πόρους
  - Μία εφαρμογή περιλαμβάνει ένα σύνολο από processes που συνεργάζονται
  - Η επικοινωνία μεταξύ τους (εντός και εκτός του ίδιου συστήματος) γίνεται μέσω τεχνικών όπως Pipes και Sockets (Inter Process Communication (IPC)
  - Στην JAVA, μία εφαρμογή τρέχει σαν ένα process αλλά μπορεί να δημιουργήσει περισσότερα processes μέσω της κλάσης ProcessBuilder
    Παράδειγμα:
    ```
    ProcessBuilder builder = new ProcessBuilder("calc.exe");
    builder.start();
    ```

Διεργασίες (processes) vs. Νήματα (threads) (συν.)

- Νήμα (thread)
  - Τα threads ονομάζονται και lightweight processes.
  - Τα processes και threads παρέχουν ένα περιβάλλον εκτέλεσης
  - Ένα thread χρειάζεται (δεσμεύει και χρησιμοποιεί) πιο λίγους πόρους από ένα process.
  - Τα threads υπάρχουν μέσα σε ένα process και ένα process έχει τουλάχιστον ένα thread.
  - Τα threads μοιράζονται τους πόρους μίας διεργασίας (μνήμη, αρχεία, κτλ.)
  - Η πολυπλοκότητα τους βρίσκεται στο θέμα της επικοινωνίας
  - Στην JAVA κάθε εφαρμογή ξεκινάει αρχικά με ένα thread (main thread)
  - Αυτή μπορεί να ξεκινήσει πολλά άλλα threads

- Στα μοντέρνα συστήματα, υπάρχουν πολλαπλοί επεξεργαστές, ή και επεξεργαστές με πολλαπλούς πυρήνες που υποστηρίζουν πιο αποδοτική εκτέλεση του συντρέχων προγραμματισμού

Δημιουργία και Εκκίνηση threads

- **Κάθε thread σχετίζεται με ένα αντικείμενο της κλάσης Thread**
- Μία εφαρμογή μπορεί να δημιουργήσει threads με δύο τρόπους

1. **Με τη δημιουργία ενός αντικειμένου που υλοποιεί την διαπροσωπεία (interface) Runnable**
   - Το interface Runnable ορίζει μία μέθοδο (run()) η οποία θα έχει τον κώδικα εκτέλεσης του συγκεκριμένου thread
   - Το αντικείμενο Runnable περνάει σαν παράμετρος στον κατασκευαστή της Thread
   - Παράδειγμα

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        ( new Thread( new HelloRunnable() ) ).start();
    }
}
```

Δημιουργία και Εκκίνηση threads (συν.)

2. **Με την δημιουργία ενός αντικειμένου που κληρονομεί από την Thread**
   - Η κλάση Thread υλοποιεί την Runnable αλλά η μέθοδος run δεν κάνει τίποτα. method does nothing. An application can subclass Thread, providing its own implementation of run
   - Παράδειγμα

```
public class HelloThread extends Thread {
    public void run() {
       System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
       ( new HelloThread() ).start();
    }
}
```

- Ο πρώτος τρόπος είναι πιο γενικός και επίσης επιτρέπει την κληρονομικότητα από άλλες κλάσεις εκτός του Thread

Η κλάση Thread

- Πιο σημαντικοί Κατασκευαστές
  - **Thread():** Δημιουργία ενός thread
  - **Thread(Runnable target):** Δημιουργία thread για κάποιο task
- Πιο σημαντικές Μεθόδοι
  - **void start():** εκκίνηση ενός thread
  - **void yield():** επιτρέπει σε άλλα threads να τρέξουν πρώτα
  - **void sleep(long millis):** το thread "κοιμάται" για κάποιο χρόνο
  - **void setPriority(int newP):** αλλαγή προτεραιότητας (1..10)
  - **void interrupt():** διακόπτει το thread
  - **void join():** περιμένει μέχρι το thread να τελειώσει την εκτέλεσή
  - **boolean isAlive(), isInterrupted():** επιστρέφουν την κατάσταση του thread (π.χ., αν τρέχει, αν είναι τύπου daemon, αν έχει διακοπεί)
  - **void wait():** το thread μπαίνει σε κατάσταση waiting μέχρι να γίνει **notify() ή notifyAll()**

Παράδειγμα: Δημιουργία και Εκκίνηση threads

```
public class TaskThreadDemo {
  public static void main(String[] args) {

    Runnable printA = new PrintChar('a', 10);
    Runnable printB = new PrintChar('b', 10);
    Runnable print10 = new PrintNum(10);

    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print10);

    thread1.start(); thread2.start(); thread3.start();
  }
}
class PrintChar implements Runnable {
  private char charToPrint; // character to print
  private int times; // The times to repeat
  public PrintChar(char c, int t) {
    charToPrint = c;     times = t; }
  //Override the run() method
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.println(charToPrint);
    }
  }
}
class PrintNum implements Runnable {
  private int lastNum;
  public PrintNum(int n) { lastNum = n; }
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.println(" " + i);
    }
  }
}
```

Δημιουργία tasks: Αντικείμενα κλάσεων που υλοποιούν την Runnable

Δημιουργία threads: Πέρασμα στον κατασκευαστή της Thread των αντικειμένων που υλοποιούν την Runnable

Εκκίνηση των 3 threads

Υλοποίηση της διαπροσωπείας Runnable από δύο κλάσεις

Υπερκάλυψη (override) της μεθόδου run()

Συγχρονισμός Νημάτων (Thread Synchronization)

- **Τα νήματα επικοινωνούν μεταξύ τους μέσω κοινών πόρων (π.χ., μία μεταβλητή)**
- **Αυτή η μορφή επικοινωνίας είναι πολύ αποδοτική** γιατί δεν χρειάζεται η δημιουργία αντιγράφων των κοινών πόρων

- **Όμως**, μπορεί να παρουσιάσει **δύο είδη προβλημάτων**:
  - **Παρεμβολές (Thread Interference):** δύο threads έχουν πρόσβαση στα ίδια δεδομένα ταυτόχρονα
  - **Λάθη Ασυνέπειας στην Μνήμη (Memory Consistency Errors):** όταν υπάρχουν ασυνέπειες στην όψη της μνήμης, π.χ., παρουσίαση δύο διαφορετικών τιμών για την ίδια μεταβλητή

- **Η λύση** για να αποφύγουμε αυτά τα προβλήματα είναι **ο συγχρονισμός**

Παράδειγμα Παρεμβολών (Thread Interference)

```
class Counter {
    private int c = 0;
    public int getValue() {
        return c;
    }
    public void setValue(int newC) {
        c=newC;
    }
}

public class Main {
  public static void main(String[] args) {
    final Counter c = new Counter();

    (new Thread( new Runnable() {
      public void run() {
        int x = c.getValue();
        x = x + 1;                    Thread A
        c.setValue(x); … } }
      )).start();

    (new Thread( new Runnable() {
      public void run() {
        int x = c.getValue();
        x = x - 1;                    Thread B
        c.setValue(x); … } }
      )).start();
}
```

Οι παρεμβολές συμβαίνουν υπάρχουν περισσότερα το ενός βημάτων σε πολλαπλά threads και αυτά υπερκαλύπτονται μεταξύ τους

<u>Σενάριο</u>

1. Thread A: Ανάκτηση του c
2. Thread B: Ανάκτηση του c
3. Thread A: Αύξηση του x (x=1)
4. Thread B: Μείωση του x (x=-1)
5. Thread A: Αποθήκ. του x (c=1)
6. Thread B: Αποθήκ. του x (c=-1)
7. Thread A: … **(Πρόβλημα)**

Παράδειγμα Λάθη Ασυνέπειας στην Μνήμη

```
class Counter {
    private int c = 0;
    public void increment() {
        c++;
        Object o = new Object();
    }
    public void decrement() {
        c--;
    }
    public void value() {
        System.out.println("c="+c);
    }
}
public class Main {
  public static void main(String[] args) {
    final Counter c = new Counter();
    (new Thread( new Runnable() {
        public void run() {
          c.increment();
          c.value(); }
      } )).start();
    (new Thread( new Runnable() {
        public void run() {
          c.decrement();
          c.value(); }
      } )).start();
}
```

Thread A

Thread B

Οι ασυνέπειες συμβαίνουν όταν τα νήματα έχουν διαφορετικές όψεις των ίδιων δεδομένων

Πιθανές εκδοχές εκτύπωσης του διπλανού προγράμματος:

- c=1, c=0 (Σωστό)
- c=-1, c=0 (**Λάθος**)
- c=0, c=0 (**Λάθος**)
- c=-1, c=1 (**Μπορεί να γίνει;;;**)

Μπορούν να αποφευχθούν (όχι πάντα) με την χρήση της σχέσης (happens-before)

Προσδιοριστής <synchronized>

- Στα προηγούμενα παραδείγματα, παρουσιάζεται πρόβλημα επειδή **δύο threads έχουν πρόσβαση και μεταβάλλουν τον ίδιο πόρο**
- **Το πρόβλημα αυτό ονομάζεται race condition**
- **Για να αποφύγουμε τα race conditions** μπορούμε να χρησιμοποιήσουμε τον **προσδιοριστή synchronized** **➔thread-safe κώδικας**
- **synchronized**: κανένα thread δεν μπορεί να έχει πρόσβαση στους πόρους της εμβέλειας του ταυτόχρονα με το τρέχων thread
  - Μπορεί να δηλωθεί **στο πεδίο εμβέλειας μίας μεθόδου** Όλοι οι πόροι που σχετίζονται με το αντικείμενο που βρίσκεται η μέθοδος κλειδώνεται (lock) μέχρι να τερματίσει η μέθοδος
  - Μπορεί να δηλωθεί **σε συγκεκριμένο πεδίο εμβέλειας κώδικα** Μόνο οι πόροι (αντικείμενο) που καθορίζονται από το συγκεκριμένο πεδίο εμβέλειας κώδικα κλειδώνονται

Παράδειγμα <synchronized> μεθόδοι

```
class SynchronizedCounter {

  private int c = 0;

  public
  synchronized void increment() {
        c++;
    }

  public
  synchronized void decrement() {
        c--;
   }

  public
  synchronized int value() {
        return c;
   }

}
```

- Τι έχουμε πετύχει;
  - **Δεν υπάρχει περίπτωση υπερκάλυψης των μεθόδων του ίδιου αντικειμένου**
  - Όταν καλέσουμε μία μέθοδο, τότε αφού αυτή πρέπει να τελειώσει για να αφήσει το lock, **έχουμε πετύχει αυτόματα την σχέση happens-before**, δηλ. ότι μέθοδος ακολουθήσει θα έχει την ίδια όψη αντικειμένου
- **Απλή** και **αποδοτική μέθοδος**

Συγχρονισμός συγκεκριμένων αντικειμένων

- Ο μηχανισμός του συγχρονισμού σχετίζεται με το κλείδωμα συγκεκριμένων αντικειμένων
- Όταν προσθέτουμε το synchronized σε μία μέθοδο σημαίνει ότι έμμεσα κλειδώνουμε το αντικείμενο που βρίσκεται στην μέθοδο
- Μερικές φορές όμως μπορεί να θέλουμε να κλειδώσουμε άλλα αντικείμενα
- Μπορούμε να το κάνουμε με την δήλωση: synchronize(object){ … }
- Ονομάζεται και synchronized block.
- Αν το αντικείμενο είναι ήδη κλειδωμένο από κάποιο άλλο thread B τότε όταν καλεστεί η μέθοδος από το thread A θα πρέπει να περιμένει μέχρι να τελειώσει την εκτέλεση το thread B (ξεκλείδωμα)
- Παρατήρηση:
  Μία δήλωση `synchronized method() { … }` είναι ισοδύναμη με
  `method() { synchronized(this) { … } }`

Συγχρονισμός με προστατευόμενα πλαίσια

- **Συγχρονισμός με προστατευόμενα πλαίσια (guarded blocks)** επιτυγχάνεται με τις εντολές **wait(), notify() και notifyAll()**
- Δεν μπορούν ποτέ να υπερσκελιστούν (δηλώνονται final).
- Μπορούν να κληθούν μόνο μέσα από τομείς που δηλώνονται ως synchronized.
- Υλοποιούν τη λογική της αναμονής κάποιου γεγονότος το οποίο θα σηματοδοτηθεί από κάποιο άλλο Thread.
- Όταν ένα Thread εισέλθει σε κατάσταση wait, αναστέλλεται μέχρι να αφυπνιστεί από κάποιο άλλο Thread.
- Επίσης ελευθερώνει το αντικείμενο το οποίο είχε κλειδώσει όταν εισήλθε στο synchronized block.
- Η αφύπνιση του Thread προκαλείται από κάποιο άλλο Thread το οποίο έχει τα δικαιώματα πρόσβασης στο ίδιο αντικείμενο.

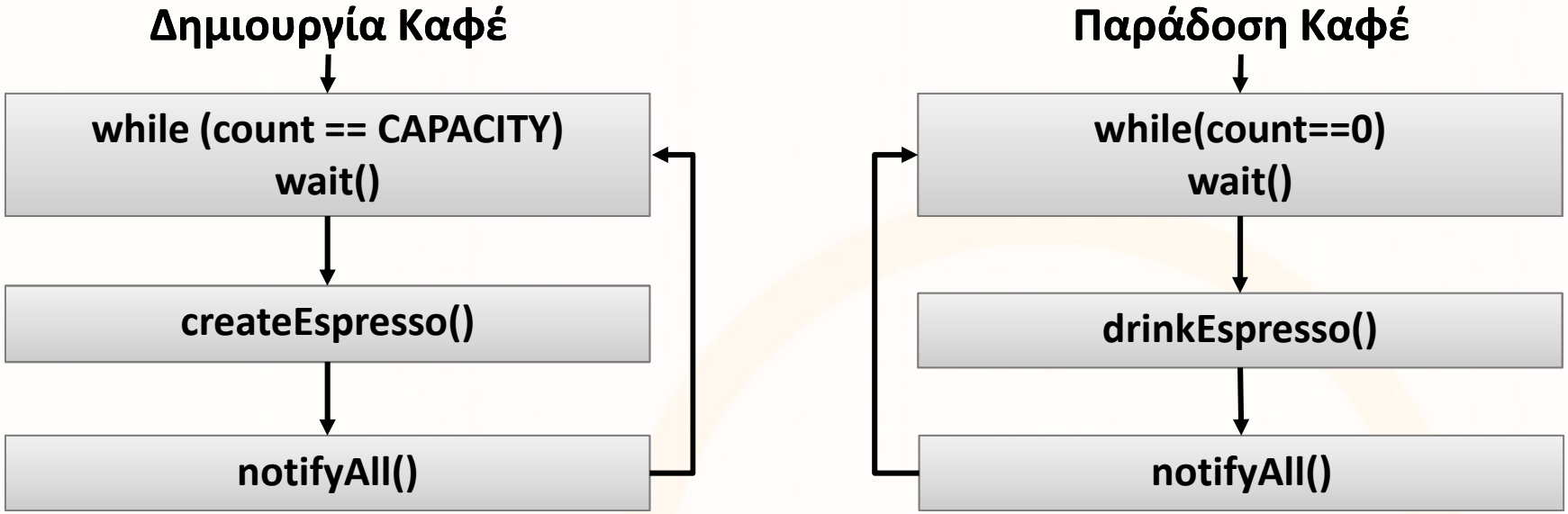
Συγχρονισμός με προστατευόμενα πλαίσια (συν.)

- Η αφύπνιση γίνεται με την εντολή notify()
- Όταν κληθεί η notify() το Thread που είχε το δικαίωμα πρόσβασης το παραχωρεί στο Thread που είχε ανασταλεί στη συνάρτηση wait().
- **Προσοχή**: Στη συνάρτηση notify() δε δηλώνουμε τι να αφυπνιστεί. Για το λόγο αυτό αν περιμένουν περισσότερα του ενός Thread δε ξέρουμε ποιο θα ξυπνήσει.
- Παρατηρήσεις
  - Η notify() αφυπνίζει μόνο 1 thread.
  - Η notifyAll() αφυπνίζει όλα τα threads.
  - Η εντολή notify() μπορεί μόνο να αφυπνίσει ένα Thread που υπήρχε στην ουρά αναμονής.
- Η notify() είναι ασφαλής υπό 2 συνθήκες μόνο:
  - Όταν και μόνο όταν υπάρχει ένα thread σε αναμονή.
  - Όταν πολλά thread είναι σε αναμονή αλλά δεν υπάρχει ιδιαίτερη σημασία ποιο από όλα θα ξυπνήσει.

Παράδειγμα συγχρονισμού με guarded blocks

Espresso Bar (Παραγωγή / Κατανάλωση)

- Χρησιμοποιούμε ένα πάγκο (buffer) όπου τοποθετούμε τα espressos που παράγει ο barrista.
- Ο πάγκος (buffer) έχει περιορισμένη χωρητικότητα (CAPACITY)
- Υπάρχει μία μεταβλητή count που μετράει τα διαθέσιμα espresso στον πάγκο
- Ο barrista έχει τη μέθοδο createEspresso() που δημιουργεί ένα espresso και ο πελάτης την μέθοδο drinkEspresso() που πίνει ένα espresso



- **Ερώτηση:** Χρειαζόμαστε τους βρόγχους while και τις συνθήκες τους;

Συγχρονισμός με locks

- Όπως έχουμε πει, μία synchronized μέθοδος έμμεσα κλειδώνει το στιγμιότυπο της κλάσης (αντικείμενο) πριν να εκτελέσει την μέθοδο
- **Το JDK 1.5 (και αργότερα) επιτρέπουν στον προγραμματιστή να δημιουργήσει άμεσα τα locks**
- Αυτό δίνει **περισσότερη ευελιξία για τον έλεγχο και το συγχρονισμό threads**
- **Ένα lock είναι ένα στιγμιότυπο κλάσης που υλοποιεί την διαπροσωπεία java.util.concurrent.locks.Lock**
  - Η διαπροσωπεία Lock δηλώνει μεθόδους για την δημιουργία και απελευθέρωση locks
  - Επίσης, παρέχει την μέθοδο newCondition() για την δημιουργία προϋποθέσεων (αντικειμένων τύπου Condition) για την επικοινωνία και συντονισμό threads.
  - Γνωστή κλάση που υλοποιεί την Lock είναι η **ReentrantLock** που επιβάλει μία δίκαιη πολιτική locks ώστε να μην παρουσιάζονται προβλήματα starvation (π.χ., threads που περιμένουν το περισσότερο έχουν προτεραιότητα)

Συγχρονισμός με locks (συν.)

- **Δημιουργία και αρχικοποίηση** αντικειμένου τύπου **Lock**
  Lock lock = **new** ReentrantLock();
- Μεθόδοι
  - **lock(), lockInterruptibly():** απόκτηση lock και απόκτηση lock αν δεν έχει γίνει interrupt το τρέχον thread
  - **unlock():** απελευθέρωση του lock
  - **tryLock(), tryLock(long time, TimeUnit unit):** προσπάθεια απόκτησης του lock αν είναι διαθέσιμο (και σε συγκεκριμένο χρονικό διάστημα)
  - **newCondition():** δημιουργία καινούριας συνθήκες (επόμενη διαφάνεια)
- Παράδειγμα χρήσης σε μία μέθοδο

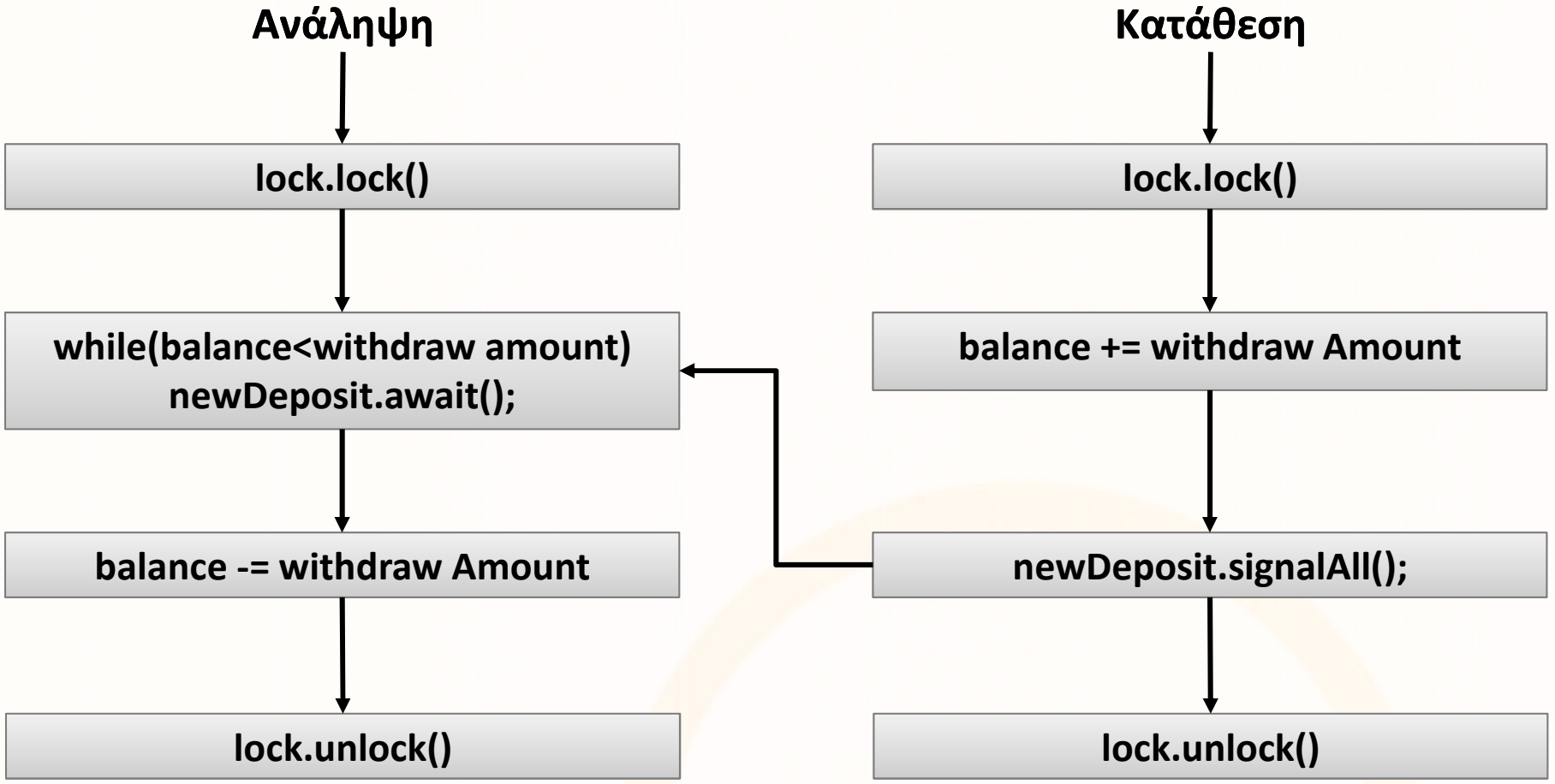
```
lock.lock();  /* … δηλώσεις …*/   lock.unlock();
```

Συγχρονισμός με locks (συν.)

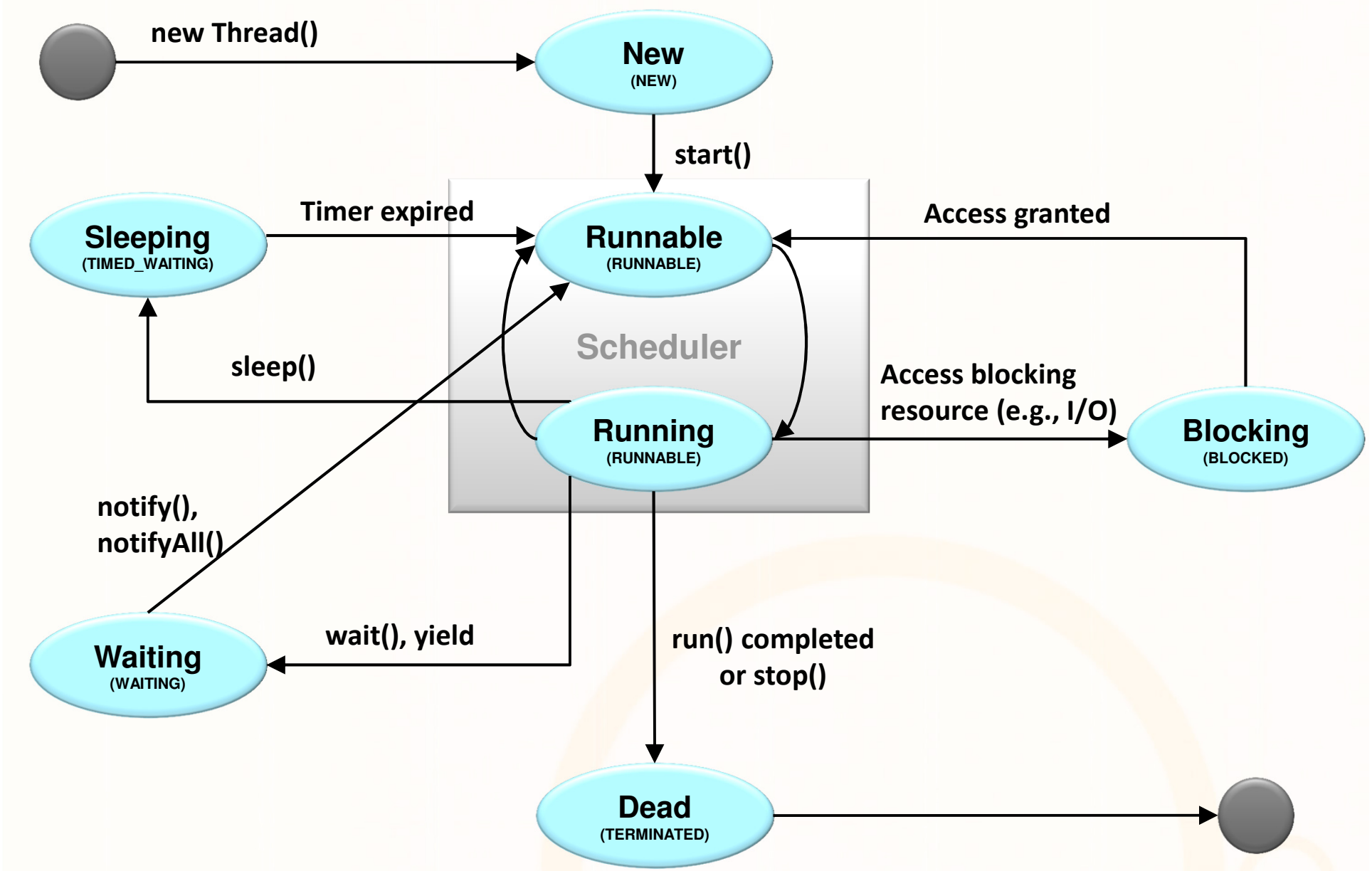
- **Τα conditions χρησιμοποιούνται για επίτευξη επικοινωνίας μεταξύ threads**
- Ένα thread μπορεί να προσδιορίσει τι θα κάνει κάτω από συγκεκριμένες συνθήκες
- Δημιουργούνται με την μέθοδο newCondition() σε Lock αντικείμενα
- Για την επικοινωνία χρησιμοποιούνται οι μεθόδοι:
  - **await():** το thread περιμένει μέχρι η συνθήκη να γίνει σηματοδότηση (signal)
  - **signal():** Σηματοδοτεί ότι ένα thread που περιμένει πρέπει να ξύπνησε
  - **signalAll():** Όπως το signal() αλλά όλα τα threads που περιμένουν
  - await(long time, TimeUnit unit)
  - **long awaitNanos(long nanosTimeout), void awaitUninterruptibly(), boolean awaitUntil(Date deadline)**

Συγχρονισμός με locks (συν.)

- Παράδειγμα BankAccount με διαδικασίες Κατάθεση/Ανάληψη



Καταστάσεις ενός thread (java.lang.Thread.State)



Ατομική Πρόσβαση (Atomic Access)

- Ένας άλλος τρόπος για επίτευξη του συγχρονισμού είναι μέσα από τον **καθορισμό ατομικών μεταβλητών**
- Μία **ατομική ενέργεια (atomic action) είναι μία ενέργεια η οποία συμβαίνει όλη την ίδια στιγμή**, αλλιώς δεν συμβαίνει καθόλου, δηλ. δεν μπλοκάρει
- Τα atomic actions δεν μπορούν να υπερκαλυφθούν γι' αυτό και δεν μπορεί να υπάρξει πρόβλημα παρεμβολών
- Αυτό δεν σημαίνει ότι δεν μπορούν να υπάρξουν ασυνέπειες μνήμης γι' αυτό και μπορεί να πρέπει να καθοριστούν σαν synchronized
- Χρησιμοποιώντας πρόσβαση ατομικών μεταβλητών είναι πιο αποδοτικό από synchronized κώδικα αλλά χρειάζονται περισσότερη προσπάθεια από τον προγραμματιστή
- Κάποιες από τις κλάσεις που βρίσκονται στην κεντρική βιβλιοθήκη του συγχρονισμού (java.util.concurrent) περιλαμβάνουν ατομικές μεθόδους που δεν βασίζονται στον συγχρονισμό.

Ατομική Πρόσβαση (Atomic Access) (συν.)

- Τα αντικείμενα και οι Αρχέγονοι τύποι μπορούν να δηλωθούν ως ατομικοί χρησιμοποιώντας τον τροποποιητή **volatile**
- Η λέξη κλειδί **volatile** ενημερώνει τον compiler ότι η συγκεκριμένη κατάσταση (μεταβλητή) θα έχει πρόσβαση από περισσότερα του ενός Threads.
- **Η μεταβλητή που δηλώνεται volatile δεν αποθηκεύεται ποτέ στην cache του Thread.**
- **Διαφορές μεταξύ volatile και synchronized**

Χαρακτηριστικό	synchronized	volatile
Τύπος Μεταβλητής	Αντικείμενο	Αντικείμενο, Αρχέγονοι Τύποι
Επιτρέπεται το null;	ΟΧΙ	ΝΑΙ
Μπορεί να μπλοκάρει;	ΝΑΙ	ΌΧΙ
Πότε γίνεται ο συγχρονισμός;	Κάλεσμα/Εκτέλεση της μεθόδου	Πρόσβαση στην μεταβλητή
Συνδυασμός πολλαπλών λειτουργιών σε ένα ατομικό block	ΝΑΙ	JAVA 1.4: ΟΧΙ JAVA 1.5: ΝΑΙ

Παράδειγμα τροποποιητή <volatile>

```
public class VolatileTest {
  volatile boolean running = true;

  public void test() {
    new Thread(new Runnable() {
      public void run() {
        int counter = 0;
        while (running) {
          counter++;
        }
        System.out.println("Thread 1 finished.Counted up to "+counter);
    }}).start();
    new Thread(new Runnable() {
      public void run() {
      // Sleep for a bit so that thread 1 has a chance to start
      try { Thread.sleep(100);}
      catch (InterruptedException ignored) { }
      System.out.println("Thread 2 finishing");
      running = false;
    }}).start();
  }

  public static void main(String[] args) { new VolatileTest().test();}
}
```

Προβλήματα Συγχρονισμού

- Οι τεχνικές του συγχρονισμού μπορούν να μειώσουν κατά πολύ τα προβλήματα των παρεμβολών και της ασυνέπειας μνήμης
- Φυσικά, αυτό εναπόκειται και στις προγραμματιστικές ιδιότητες του προγραμματιστή
- Παρουσιάζουν όμως κάποια άλλα προβλήματα τα οποία έχουν σχέση με την ζωή των threads
- Τα πιο σημαντικά προβλήματα συγχρονισμού είναι:
  - **(Νεκρό) Αδιέξοδο (Deadlock):** Κάθε thread περιμένει ανενεργά ένα lock από κάποιο άλλο thread που με τη σειρά του περιμένει ένα lock από άλλο thread (διαδοχικά μέχρι το πρώτο thread)
  - **Λιμοκτονία (Starvation):** κάποια thread περιμένουν για μεγάλα χρονικά διαστήματα κάποιους locked πόρους
  - **Ζωντανό Αδιέξοδο (Livelock):** το ίδιο με το deadlock αλλά τα threads είναι ενεργά

Παράδειγμα Προβλημάτων Συγχρονισμού

- **DeadLock**

  Αλγόριθμος περάσματος διαδρόμου:

  1. Αν έχει κάποιο απέναντι σου στην ίδια λωρίδα, τότε περίμενε μέχρι να μετακινηθεί
  2. Αλλιώς προχώρησε μπροστά
     Αποτέλεσμα: καμία μετακίνηση από τα δύο άτομα, … ποτέ!

- **LiveLock**

  Αλγόριθμος περάσματος διαδρόμου:

  1. Αν έχει κάποιο απέναντι σου στην ίδια λωρίδα, μετακινήσου στην άλλη λωρίδα
  2. Αλλιώς προχώρησε μπροστά
     Αποτέλεσμα: συνεχής μετακίνηση αριστερά-δεξιά από τα δύο άτομα

- **Starvation**

  Αλγόριθμος διεκπεραίωσης παραγγελιών εστιατορίου:

  1. Κάθε χρονική στιγμή επέλεγε την παραγγελία που κοστίζει περισσότερο
     Αποτέλεσμα: Υποθέτοντας μία παραγγελία το λεπτό και χρόνο διεκπεραίωσης > του ενός λεπτό η παραγγελία με το πιο μικρό κόστος θα εξυπηρετηθεί τελευταία

Διαχείριση Συγχρονισμού

- Τα προηγούμενα παραδείγματα τρέχουν αποδοτικά για μικρές εφαρμογές
- Σε μεγάλες εφαρμογές είναι καλύτερα να διαχωρίζεται η διαχείριση των threads από την υπόλοιπη εφαρμογή (π.χ., Web Server)
- Τα υπεύθυνα αντικείμενα για την διαχείριση των threads είναι οι εκτελεστές (executors)
- Στην βιβλιοθήκη java.util.concurrent ορίζονται 3 executor interfaces:
  - **Executor:** υποστηρίζει την δημιουργία και εκτέλεση καινούριων threads
  - **ExecutorService:**,ελέγχει τον κύκλο ζωής ενός executor και των threads του.
  - **ScheduledExecutorService:** υποστηρίζει την περιοδική εκτέλεση των threads

Παράδειγμα Διαχείρισης Συγχρονισμού

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

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Shut down the executor
    executor.shutdown();
  }
}
```