Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχ. και Μηχανικών Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Συγχρονισμός

Συστήματα Παράλληλης Επεξεργασίας
9ο Εξάμηνο

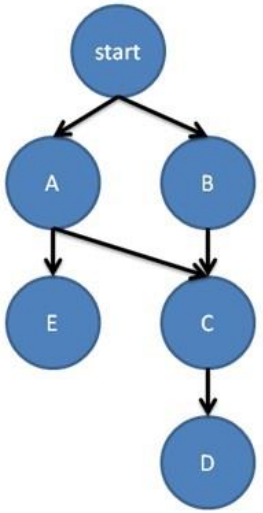
National Technical University of Athens
CSLab

Σύνοψη

- Το πρόβλημα του συγχρονισμού
- Βασικοί μηχανισμοί
  - Κλειδώματα και υλοποιήσεις
  - Condition variables
- Τακτικές συγχρονισμού
  - Coarse-grain locking
  - Fine-grain locking
  - Optimistic synchronization
  - Lazy synchronization
  - Non-blocking synchronization

Η ανάγκη για συγχρονισμό

- Ταυτόχρονη πρόσβαση σε κοινά δεδομένα μπορεί να οδηγήσει σε ασυνεπή εκτέλεση
  - Το πρόβλημα του **κρίσιμου τμήματος (critical section):** το πολύ μία διεργασία πρέπει να βρίσκεται στο κρίσιμο τμήμα σε κάθε χρονική στιγμή
  - Δεν μας ενδιαφέρει η σειρά εκτέλεσης, αλλά η συντονισμένη πρόσβαση στα κοινά δεδομένα
- Η σωστή παράλληλη εκτέλεση απαιτεί συγκεκριμένη σειρά εκτέλεσης, όπως π.χ. επιβάλλεται από το γράφο των εξαρτήσεων
  - Το πρόβλημα της **σειριοποίησης (ordering)**
  - Υπάρχει αυστηρά προκαθορισμένη σειρά με την οποία πρέπει να εκτελεστούν οι εργασίες



Άλλα patterns συγχρονισμού

- **Φράγμα συγχρονισμού (barrier)**
  - Όλες οι διεργασίες που συμμετέχουν συγχρονίζονται σε ένα συγκεκριμένο σημείο κώδικα
  - Ειδική περίπτωση της σειριοποίησης
- **Αναγνώστες-εγγραφείς (readers – writers)**
  - Στο «κρίσιμο τμήμα» επιτρέπεται να είναι είτε:
    - Καμία διεργασία
    - Οσεσδήποτε διεργασίες αναγνώστες
    - Μία διεργασία εγγραφέας
- **Παραγωγός-καταναλωτής (producer – consumer)**
  - Φραγμένος αριθμός από προϊόντα ($0 < items < N$)
  - Ο παραγωγός εισέρχεται στο «κρίσιμο τμήμα» όταν $items < N$
  - Ο καταναλωτής εισέρχεται στο «κρίσιμο τμήμα» όταν $items > 0$
  - Χρειάζεται συγχρονισμένη πρόσβαση στη δομή που κρατάει τα προϊόντα και συγχρονισμένη ενημέρωση του μετρητή items

Μηχανισμοί συγχρονισμού

- **Κλειδώματα (locks)**
  - Προστατεύει ένα τμήμα κώδικα από ταυτόχρονη πρόσβαση
  - Μόνο η διεργασία που έχει λάβει το κλείδωμα μπορεί να προχωρήσει
- **Σημαφόροι (semaphores)**
  - Ακέραιος αριθμός στον οποίο επιτρέπονται 3 ενέργειες
    - Αρχικοποίηση
    - Αύξηση της τιμής κατά 1
    - Μείωση της τιμής κατά 1 και μπλοκάρισμα αν η νέα τιμή είναι 0 (ή αρνητική – ανάλογα με την υλοποίηση)
- **Παρακολουθητές (monitors) και μεταβλητές συνθήκης (condition variables)**
  - Ζεύγος κλειδώματος και condition variable (m, c)
  - Μία διεργασία αναστέλλει τη λειτουργία της (λειτουργία wait) μέχρι να ισχύσει η κατάλληλη συνθήκη
  - Μία διεργασία «ξυπνάει» (λειτουργία signal) κάποια από (ή όλες) τις διεργασίες που περιμένουν

Το πρόβλημα του κρίσιμου τμήματος
(και οι ιδιότητές του)

1. Αμοιβαίος αποκλεισμός (mutual exclusion)
2. Πρόοδος (deadlock – free)
3. Πεπερασμένη αναμονή (starvation – free)

Τα 1 και 2 είναι αναγκαία, το 3 επιθυμητό

Το κλείδωμα ως λύση του κρίσιμου τμήματος

```
do {
    lock(mylock);
    /* critical section */
    code
    unlock(mylock);
    remainder section
} while (TRUE);
```

```
do {
    lock(mylock);
    /* critical section */
    other code
    unlock(mylock);
    other remainder section
} while (TRUE);
```

Το κλείδωμα ως λύση του κρίσιμου τμήματος

```
do {
   lock(mylock);
   /* critical section */
   code
   unlock(mylock);
   remainder section
} while (TRUE);
```

```
do {
   lock(mylock);
   /* critical section */
   other code
   unlock(mylock);
   other remainder section
} while (TRUE);
```

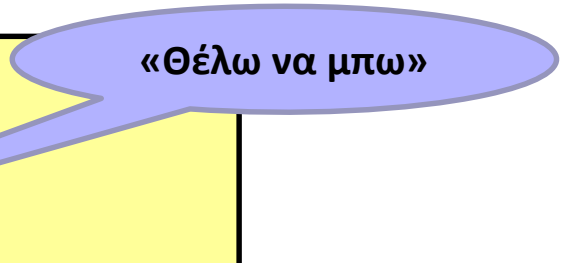
Πώς υλοποιείται ένα σωστό και «καλό» κλείδωμα;

Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα

```
// i = εγώ
// j = το άλλο νήμα
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```

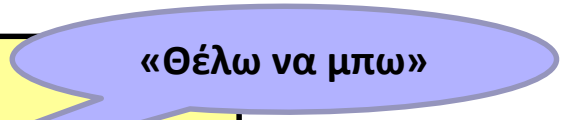
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα

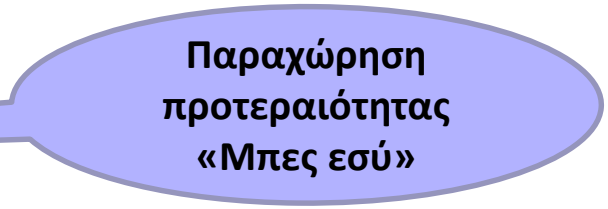
```
// i = εγώ
// j = το άλλο νήμα
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```



Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα

```
// i = εγώ
// j = το άλλο νήμα
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```





Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα

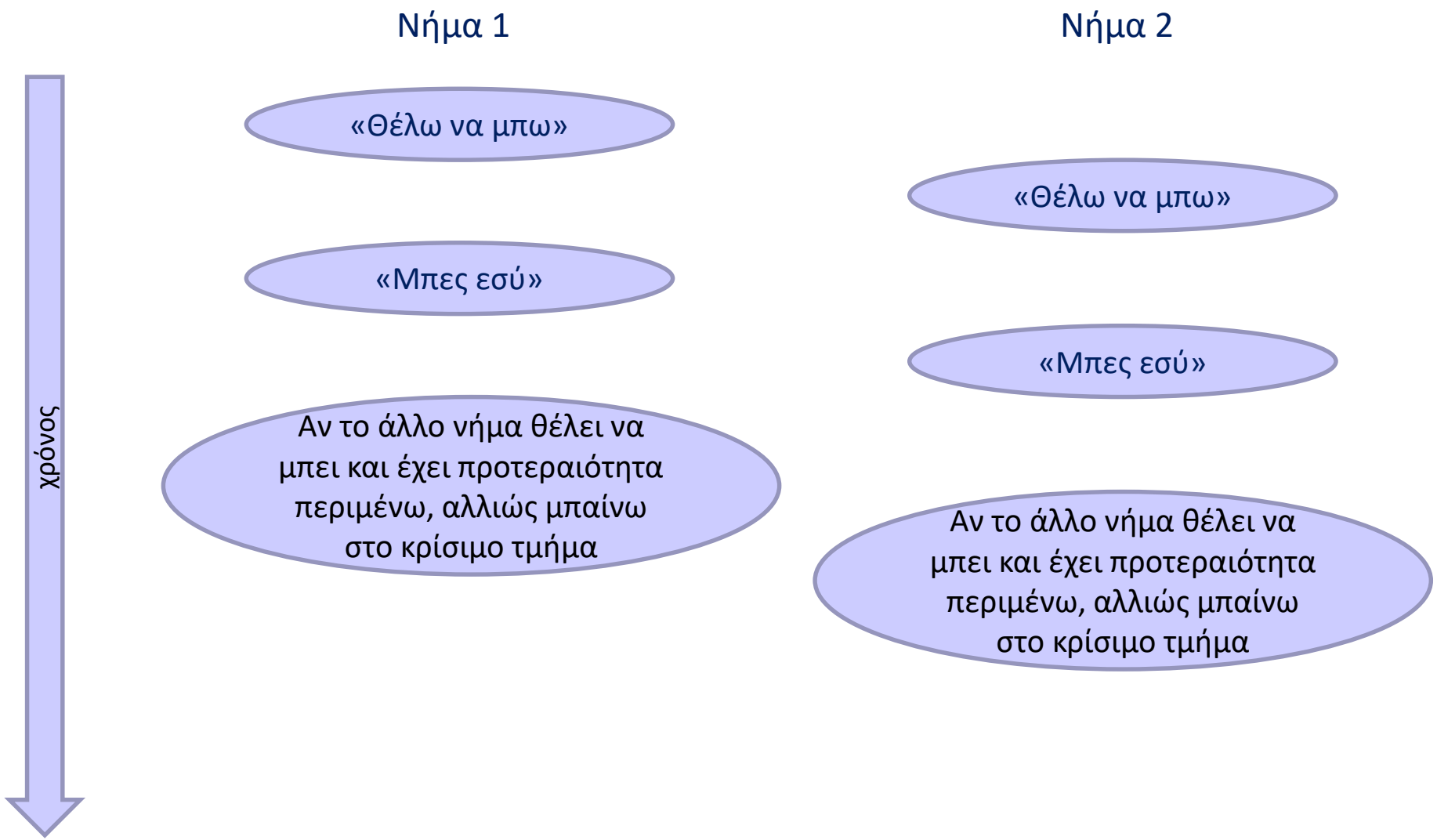
```
// i = εγώ
// j = το άλλο νήμα
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```

«Θέλω να μπω»

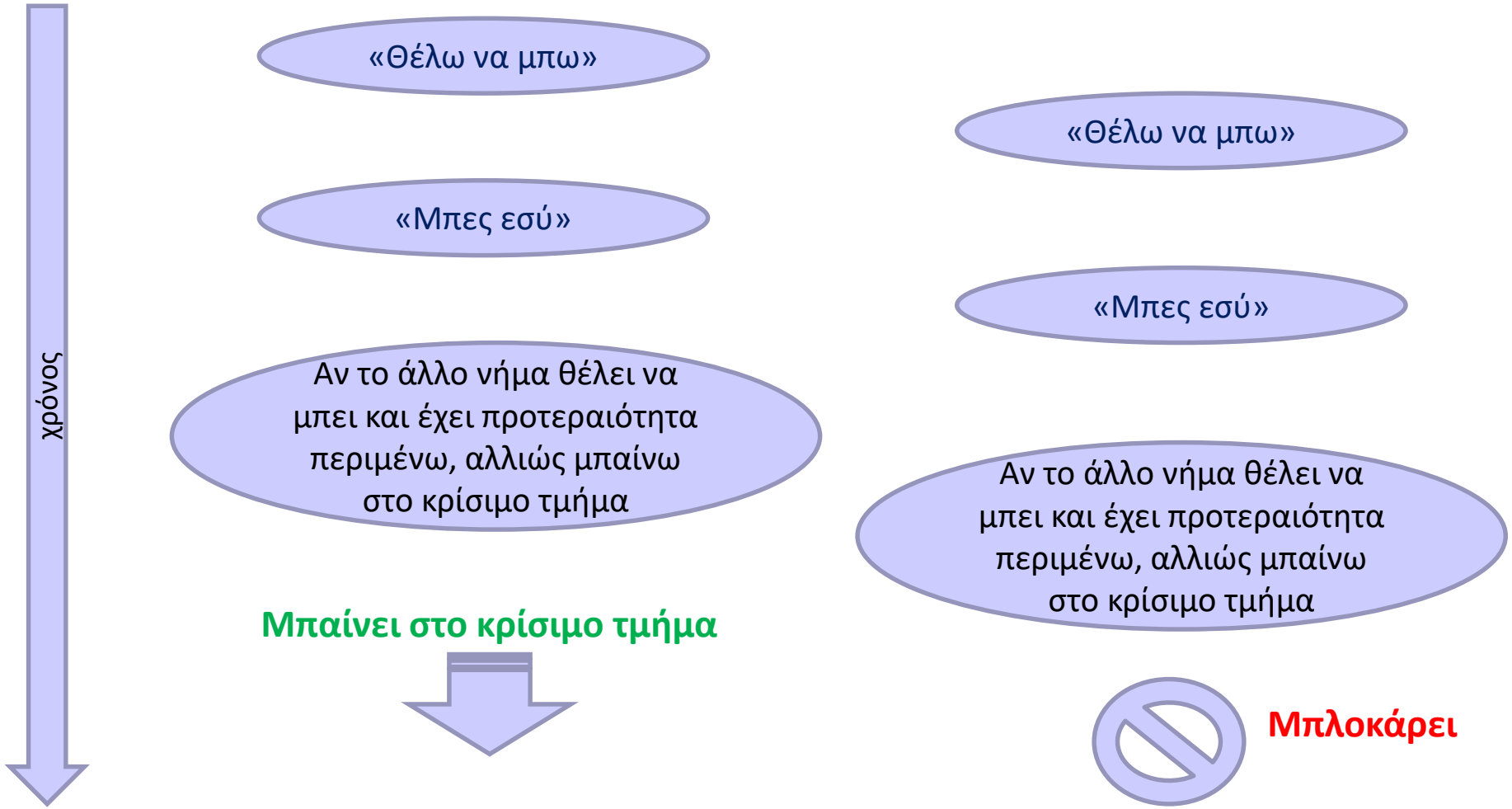
Παραχώρηση προτεραιότητας «Μπες εσύ»

Αν το άλλο νήμα θέλει να μπει και έχει προτεραιότητα περιμένω, αλλιώς μπαίνω στο κρίσιμο τμήμα

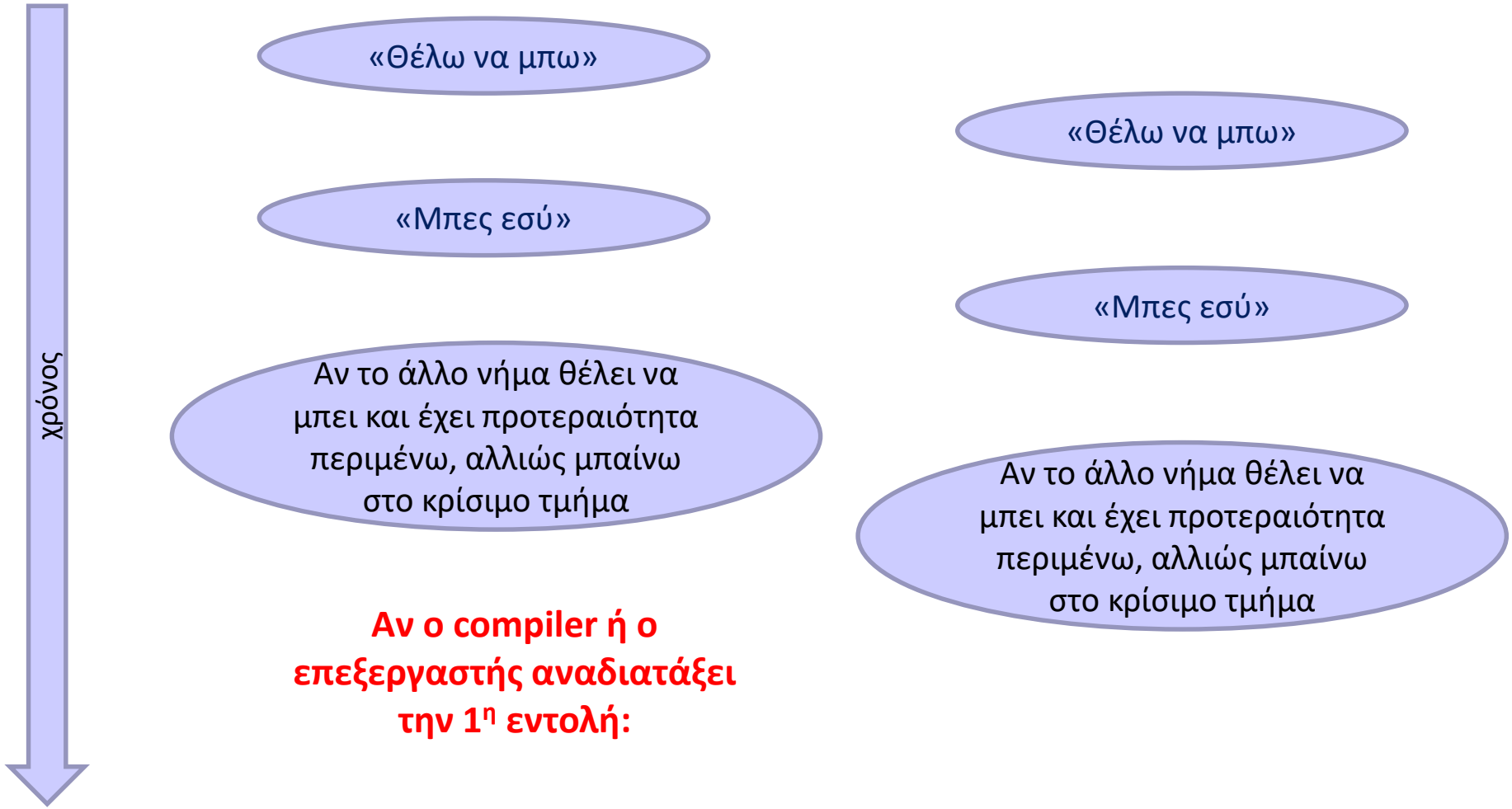
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



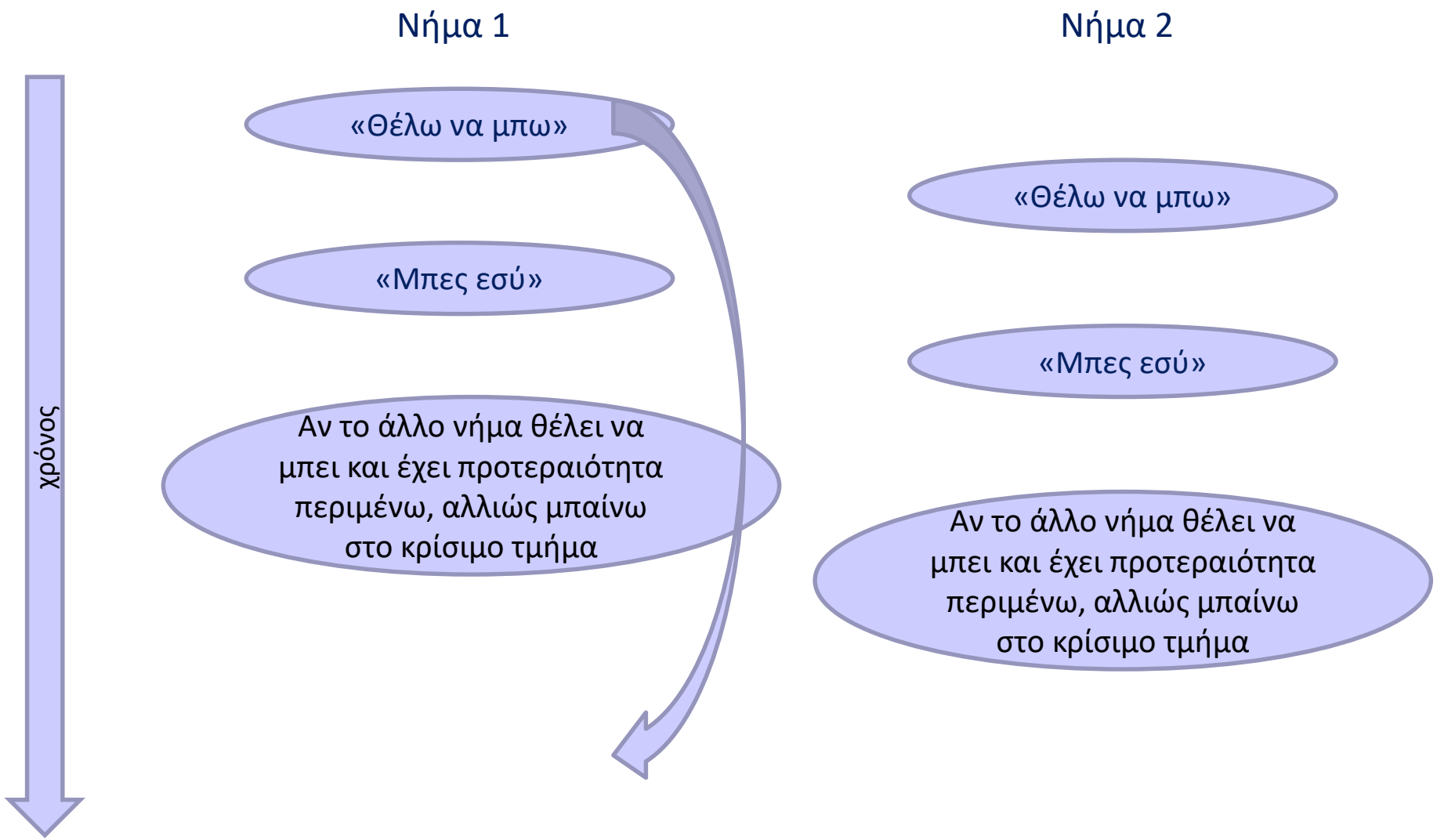
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



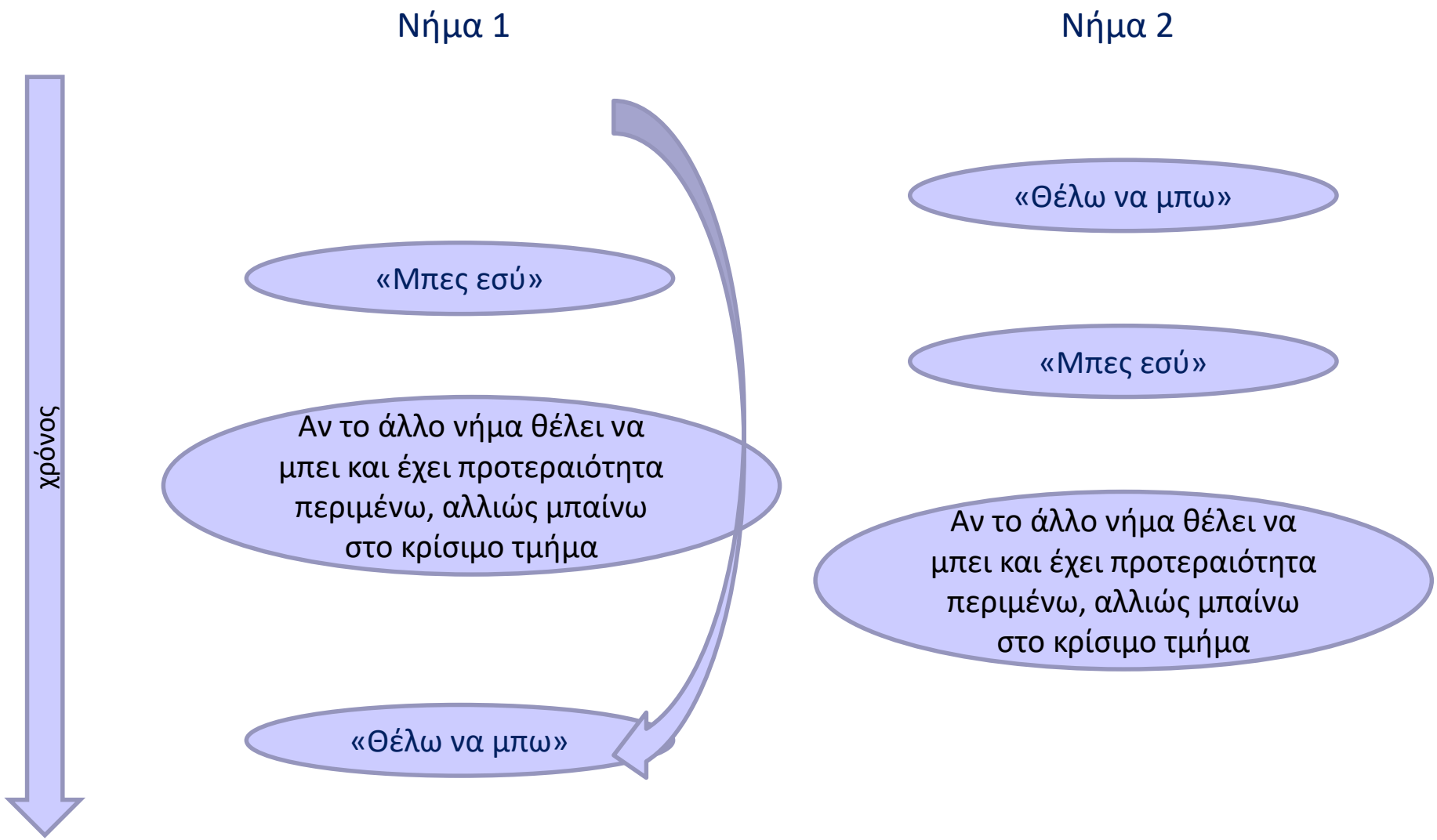
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



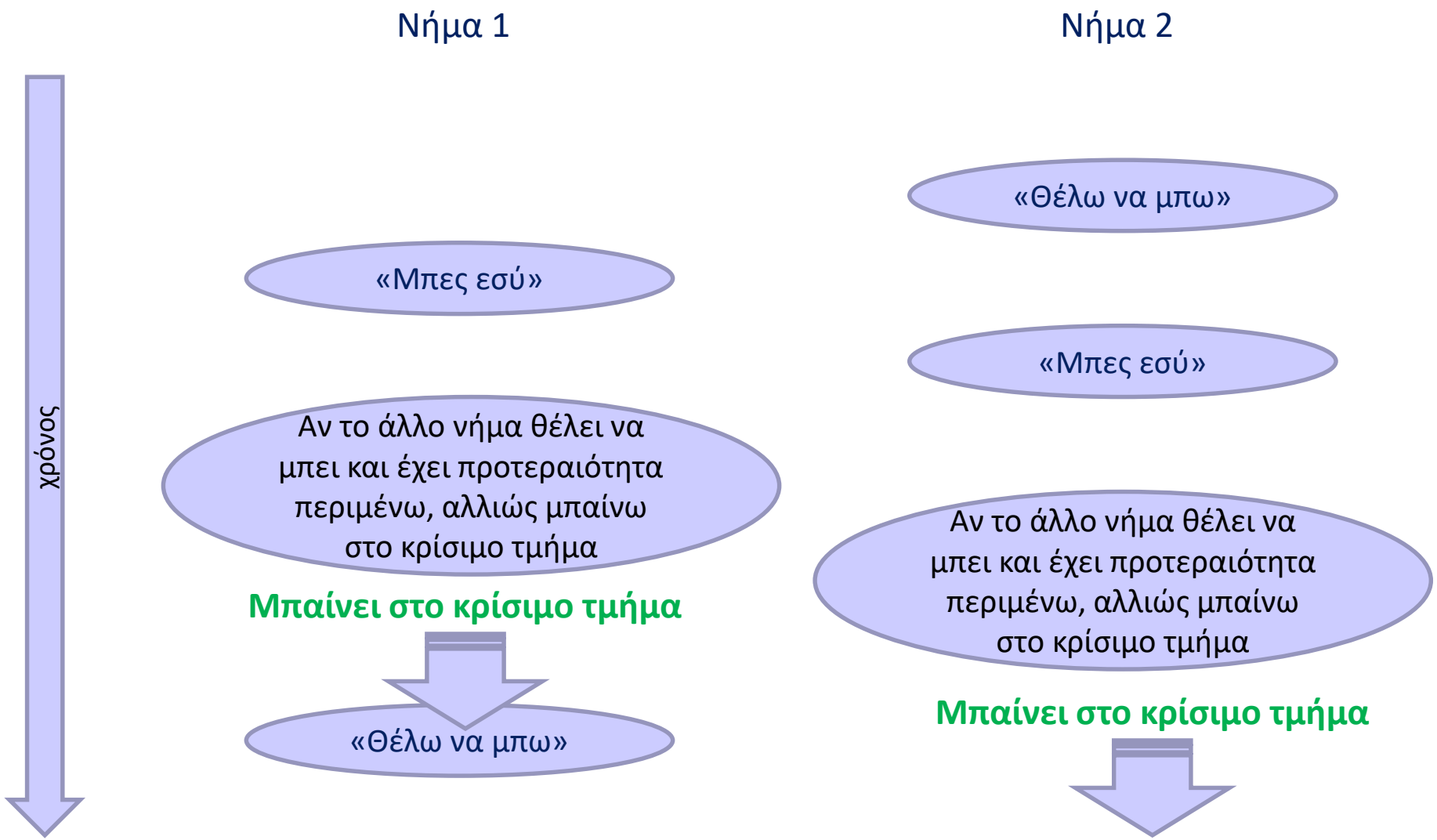
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



Λύση στο λογισμικό: Lamport's Bakery Algorithm

```
class Bakery {
 boolean[] flag;
 Label[] label;
 int size;

 public Bakery (int n) {
   flag = new boolean[n];
   label = new Label[n];
   for (int i = 0; i < n; i++) {
      flag[i] = false; label[i] = 0;
   }
 }

 public void lock() {
   int i = ThreadID.get();
   flag[i] = true;
   label[i] = max(label[0],…, label[n-1]) + 1;
   while ((∃k != i)(flag[k] && (label[k],k) << label[i],i)))
    {};
 }

 public void unlock() {
   flag[ThreadID.get()] = false;
 }
}
```

Λύση στο λογισμικό: Lamport's Bakery Algorithm

```
class Bakery {
 boolean[] flag;
 Label[] label;
 int size;

 public Bakery (int n) {
   flag = new boolean[n];
   label = new Label[n];
   for (int i = 0; i < n; i++) {
      flag[i] = false; label[i] = 0;
   }
}

 public void lock() {
   int i = ThreadID.get();
   flag[i] = true;
   label[i] = max(label[0],…, label[n-1]) + 1;
   while ((∃k != i)(flag[k] && (label[k],k) << label[i],i)))
    {};
 }

 public void unlock() {
   flag[ThreadID.get()] = false;
 }
}
```

«Θέλω να μπω»

Λύση στο λογισμικό: Lamport's Bakery Algorithm

```
class Bakery {
 boolean[] flag;
 Label[] label;
 int size;

 public Bakery (int n) {
   flag = new boolean[n];
   label = new Label[n];
   for (int i = 0; i < n; i++) {
      flag[i] = false; label[i] = 0;
   }
}

 public void lock() {
   int i = ThreadID.get();
   flag[i] = true;
   label[i] = max(label[0],…, label[n-1]) + 1;
   while ((∃k != i)(flag[k] && (label[k],k) << label[i],i)))
    {};
 }

 public void unlock() {
   flag[ThreadID.get()] = false;
 }
}
```

«Θέλω να μπω»

Παίρνει σειρά προτεραιότητας

Λύση στο λογισμικό: Lamport's Bakery Algorithm

```
class Bakery {
 boolean[] flag;
 Label[] label;
 int size;

 public Bakery (int n) {
   flag = new boolean[n];
   label = new Label[n];
   for (int i = 0; i < n; i++) {
      flag[i] = false; label[i] = 0;
   }
}

 public void lock() {
   int i = ThreadID.get();
   flag[i] = true;
   label[i] = max(label[0],…, label[n-1]) + 1;
   while ((∃k != i)(flag[k] && (label[k],k) << label[i],i)))
    {};
 }

 public void unlock() {
   flag[ThreadID.get()] = false;
 }
}
```

«Θέλω να μπω»

Παίρνει σειρά προτεραιότητας

Αναμονή μέχρι να έρθει η προτεραιότητά μου

Αν 2 νήματα έχουν αποκτήσει τον ίδιο αριθμό προτεραιότητας, λαμβάνεται υπόψη το id τους

Λύση στο λογισμικό: Lamport's Bakery Algorithm

```
class Bakery {
 boolean[] flag;
 Label[] label;
 int size;

 public Bakery (int n) {
   flag = new boolean[n];
   label = new Label[n];
   for (int i = 0; i < n; i++) {
      flag[i] = false; label[i] = 0;
   }
}

 public void lock() {
   int i = ThreadID.get();
   flag[i] = true;
   label[i] = max(label[0],…, label[n-1]) + 1;
   while ((∃k != i)(flag[k] && (label[k],k) << label[i],i)))
    {};
 }

 public void unlock() {
   flag[ThreadID.get()] = false;
 }
}
```

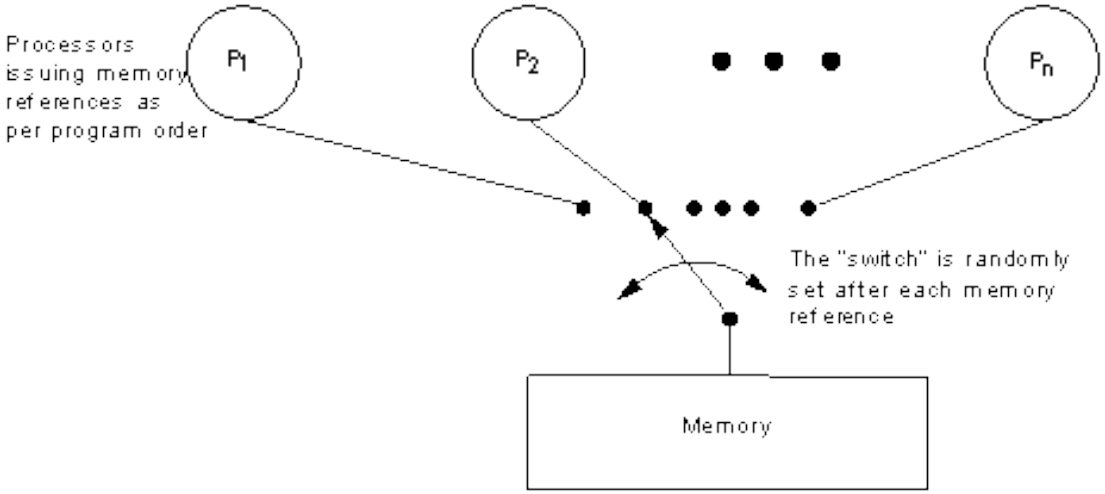
«Δεν θέλω να μπω»

Ζητήματα λύσεων στο λογισμικό

- Βασίζονται σε απλές αναγνώσεις και εγγραφές στη μνήμη (+)
- Το κλείδωμα του Peterson:
  - Λειτουργεί μόνο για 2 διεργασίες (-)
- Το κλείδωμα του Lamport
  - Λειτουργεί για N διεργασίες (+)
  - Διατρέχει N θέσεις μνήμης και αυτό μπορεί να είναι πολύ αργό και μη κλιμακώσιμο για μεγάλα N (-)
- Και τα 2 κλειδώματα δεν λειτουργούν αν ο μεταγλωττιστής ή ο επεξεργαστής αναδιατάξουν εντολές (υποθέτουν ότι το σύστημα υλοποιεί ακολουθιακή συνέπεια – **sequential consistency**) (-)
  - Στο μεταγλωττιστή μπορώ να το επιβάλλω
  - Στον επεξεργαστή χρειάζεται ειδική υποστήριξη

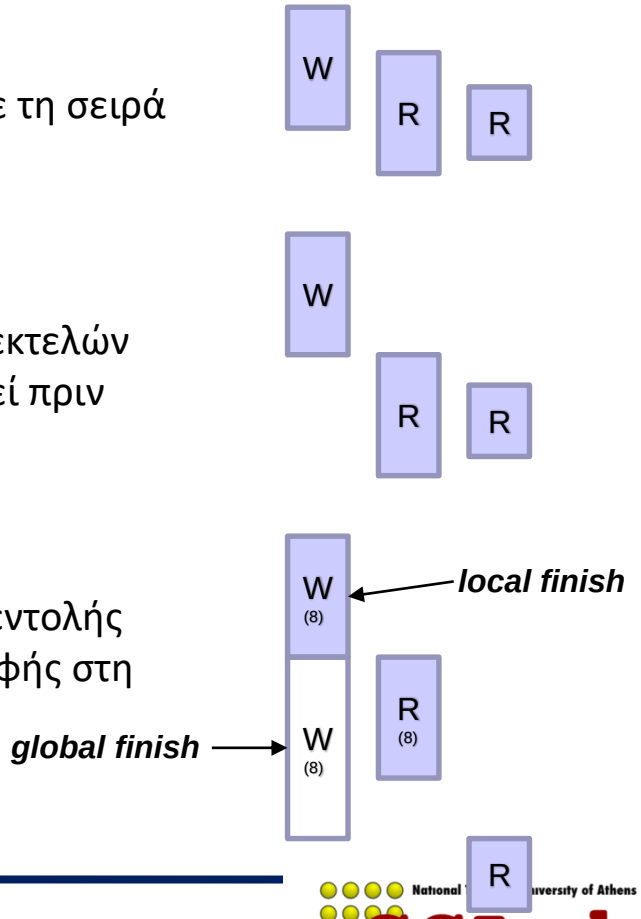
Ακολουθιακή συνέπεια (sequential consistency)

- *Ένας πολυεπεξεργαστής είναι ακολουθιακά συνεπής αν το αποτέλεσμα οποιασδήποτε εκτέλεσης είναι το ίδιο ως εάν οι εντολές όλων των επιμέρους επεξεργαστών εκτελούνταν σειριακά και οι εντολές του κάθε επιμέρους επεξεργαστή εκτελούνται με τη σειρά που ορίζεται στο πρόγραμμα* (Lamport, 1979).



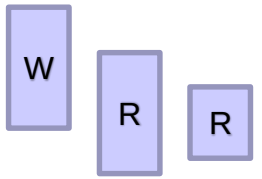
Ακολουθιακή συνέπεια (sequential consistency)

- *Ένας πολυεπεξεργαστής είναι ακολουθιακά συνεπής αν το αποτέλεσμα οποιασδήποτε εκτέλεσης είναι το ίδιο ως εάν οι εντολές όλων των επιμέρους επεξεργαστών εκτελούνταν σειριακά και οι εντολές του κάθε επιμέρους επεξεργαστή εκτελούνται με τη σειρά που ορίζεται στο πρόγραμμα* (Lamport, 1979).
- Ικανές συνθήκες:
  - Κάθε διεργασία εκκινεί εντολές πρόσβασης στη μνήμη με τη σειρά που ορίζεται στο πρόγραμμα
  - Μετά την εκκίνηση μιας εντολής εγγραφής στη μνήμη ο εκτελών επεξεργαστής περιμένει μέχρι η εγγραφή να ολοκληρωθεί πριν εκτελέσει την επόμενη εντολή
  - Μετά την εκκίνηση μιας εντολής ανάγνωσης, ο εκτελών επεξεργαστής πρέπει να περιμένει την ολοκλήρωση της εντολής ανάγνωσης αλλά και την ολοκλήρωση της εντολής εγγραφής στη θέση μνήμης στην οποία γίνεται η ανάγνωση

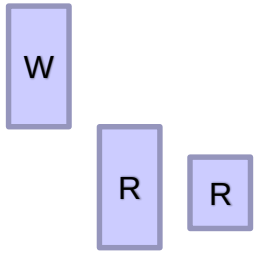


Ακολουθιακή συνέπεια (sequential consistency)

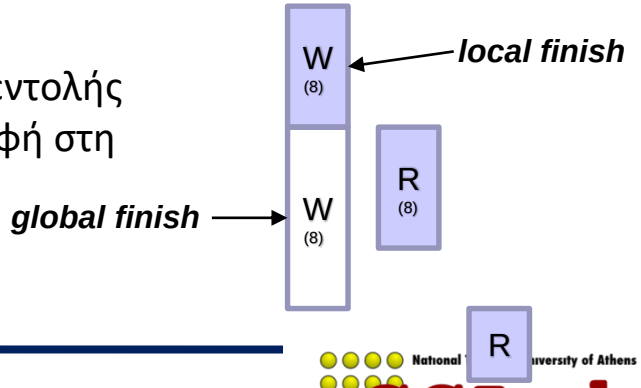
- *Ένας πολυεπεξεργαστής είναι ακολουθιακά συνεπής αν το αποτέλεσμα οποιασδήποτε εκτέλεσης είναι το ίδιο ως εάν οι εντολές όλων των επιμέρους επεξεργαστών εκτελούνταν σειριακά και οι εντολές του κάθε επιμέρους επεξεργαστή εκτελούνται με τη σειρά που ορίζεται στο πρόγραμμα* (Lamport, 1979).
- Ικανές συνθήκες:
  - Κάθε διεργασία εκκινεί εντολές πρόσβασης στη μνήμη με τη σειρά που ορίζεται στο πρόγραμμα **(περιορισμός out-of-order execution)**



  - Μετά την εκκίνηση μιας εντολής εγγραφής στη μνήμη ο εκτελών επεξεργαστής περιμένει μέχρι η εγγραφή να ολοκληρωθεί πριν εκτελέσει την επόμενη εντολή **(επιπλέον σειριοποίηση)**



  - Μετά την εκκίνηση μιας εντολής ανάγνωσης, ο εκτελών επεξεργαστής πρέπει να περιμένει την ολοκλήρωση της εντολής ανάγνωσης αλλά και την ολοκλήρωση της εντολής εγγραφή στη θέση μνήμης στην οποία γίνεται η ανάγνωση **(επιπλέον σειριοποίηση και καθολική λειτουργία)**



Ακολουθιακή συνέπεια (sequential consistency)

- *Ένας πολυεπεξεργαστής είναι ακολουθιακά συνεπής αν το αποτέλεσμα οποιασδήποτε εκτέλεσης είναι το ίδιο ως εάν οι εντολές όλων των επιμέρους επεξεργαστών εκτελούνταν σειριακά και οι εντολές του κάθε επιμέρους επεξεργαστή εκτελούνται με τη σειρά που* ***ορίζεται στο πρόγραμμα*** (Lamport, 1979).
- Η σειρά που ορίζεται στο πρόγραμμα είναι η σειρά που έχει παράξει ο μεταγλωττιστής. Ποιος εγγυάται ότι ο μεταγλωττιστής δεν έχει αναδιατάξει εντολές;
- **Προσοχή:** Οι παραπάνω αλγόριθμοι στο λογισμικό είναι κατανεμημένοι, δηλαδή οι εγγραφές σε έναν επεξεργαστή επηρεάζουν την ορθή εκτέλεση σε έναν άλλο επεξεργαστή. Αυτή η σημασιολογία όμως δεν είναι γνωστή ούτε στον επεξεργαστή, ούτε στο μεταγλωττιστή που με βάση την ***τοπική ανάλυση εξαρτήσεων*** έχουν τη δυνατότητα να κάνουν αναδιατάξεις για λόγους βελτιστοποίησης

Μοντέλα μνήμης

- Η ακολουθιακή συνέπεια μπορεί να υποστηρίξει κατανεμημένους αλγορίθμους συγχρονισμού στο λογισμικό
- Έχει όμως πολύ μεγάλο κόστος στην επίδοση
  - Γιατί να το πληρώσουν και τα σειριακά προγράμματα ή τα προγράμματα που δεν συγχρονίζονται;
- **Λύση:** Ο προγραμματιστής θα πρέπει να αναλάβει να επισημαίνει τα σημεία όπου χρειάζεται συγχρονισμός **(release consistency)**
  - Ο μεταγλωττιστής και ο επεξεργαστής κάνουν βελτιστοποιήσεις ανάμεσα στις περιοχές όπου επισημαίνονται με «φράγματα συγχρονισμού»
- Οι παράλληλες γλώσσες προγραμματισμού και οι επεξεργαστές ορίζουν «μοντέλα μνήμης», δηλαδή τις επιτρεπτές αναδιατάξεις στις εντολές πρόσβασης στη μνήμη

Το κλείδωμα ως λύση του κρίσιμου τμήματος (revisited)

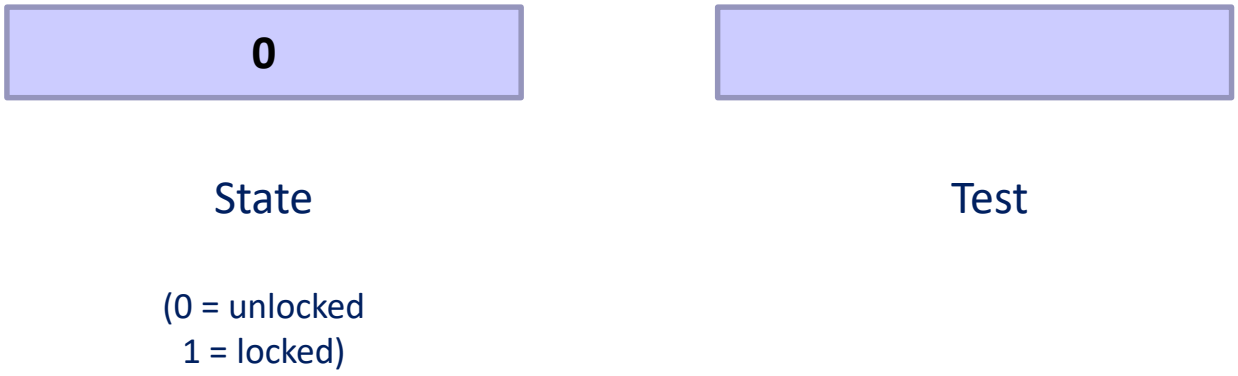
```
do {
   lock(mylock);
   /* critical section */
   code
   unlock(mylock);
   remainder section
} while (TRUE);
```

```
do {
   lock(mylock);
   /* critical section */
   other code
   unlock(mylock);
   other remainder section
} while (TRUE);
```

- Δύο ζητήματα που πρέπει να αντιμετωπιστούν:
  1. Κίνδυνος αναδιάταξης εντολών:
     - **Προσέγγιση 1:** Το σύστημα (ISA) υποστηρίζει sequential consistency (+: ο προγραμματιστής απλά υλοποιεί τον αλγόριθμο συγχρονισμού, -: χαμηλή επίδοση για ΟΛΕΣ τις εφαρμογές – ακόμα και τις σειριακές)
     - **Προσέγγιση 2:** Το σύστημα (ISA) δεν υποστηρίζει sequential consistency (-: ο προγραμματιστής πρέπει να επισημάνει στον κώδικα ότι εισέρχεται σε αλγόριθμο συγχρονισμού, +: υψηλή επίδοση)
  2. Προσπέλαση πολλών θέσεων μνήμης (π.χ. Bakery)
- **Λύση:** Υποστήριξη από το υλικό
  - Κίνδυνος αναδιάταξης εντολών -> **Release consistency + memory fences**
  - Προσπέλαση πολλών θέσεων μνήμης -> **Atomic operations**

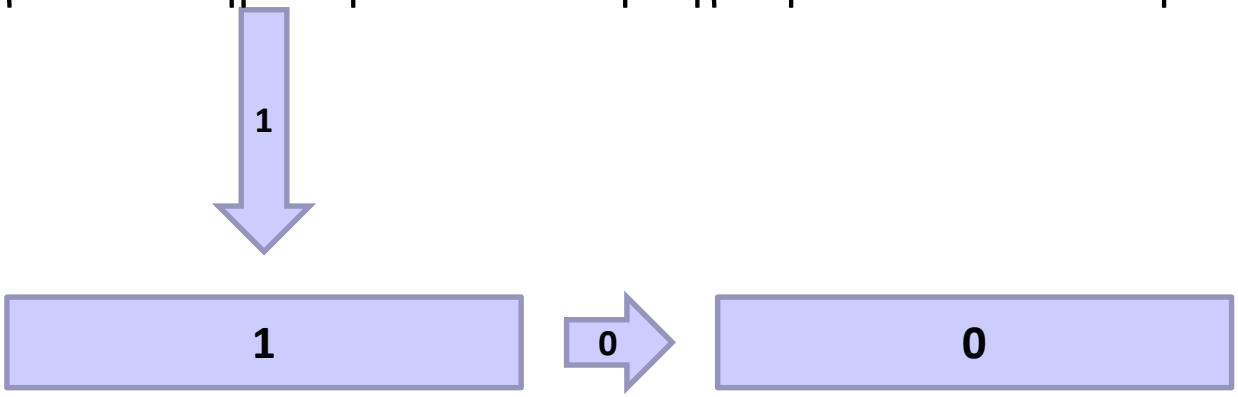
Υποστήριξη από το υλικό:
atomic operations, π.χ. test-and-set, compare-and-swap

- **Ατομικά** ενεργούν σε **2** μεταβλητές
- Στις υλοποιήσεις τους τυπικά περιλαμβάνουν και **memory fence** (επιβολή για ολοκλήρωση όλων των προηγούμενων εντολών πρόσβασης στη μνήμη)



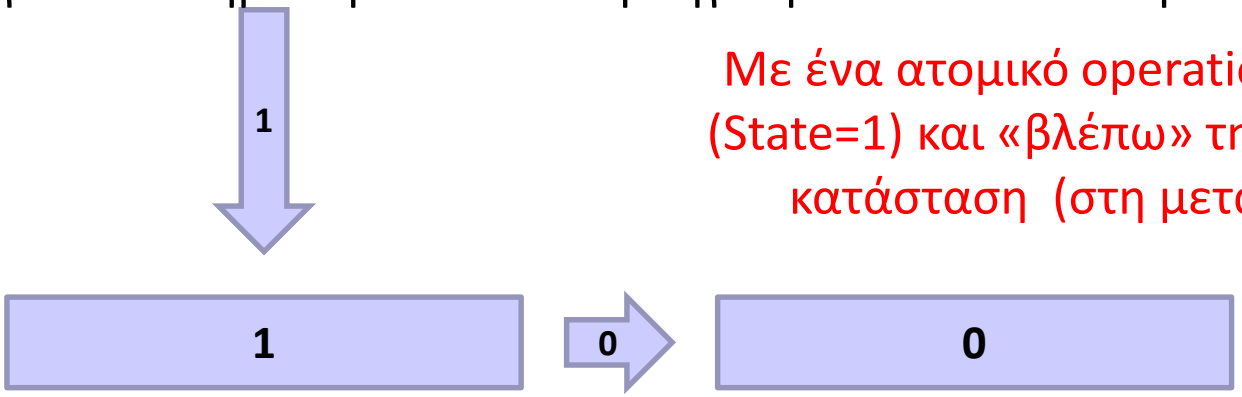
Υποστήριξη από το υλικό:
atomic operations, π.χ. test-and-set, compare-and-swap

- **Ατομικά** ενεργούν σε **2** μεταβλητές
- Στις υλοποιήσεις τους τυπικά περιλαμβάνουν και **memory fence** (επιβολή για ολοκλήρωση όλων των προηγούμενων εντολών πρόσβασης στη μνήμη)



Υποστήριξη από το υλικό:
atomic operations, π.χ. test-and-set, compare-and-swap

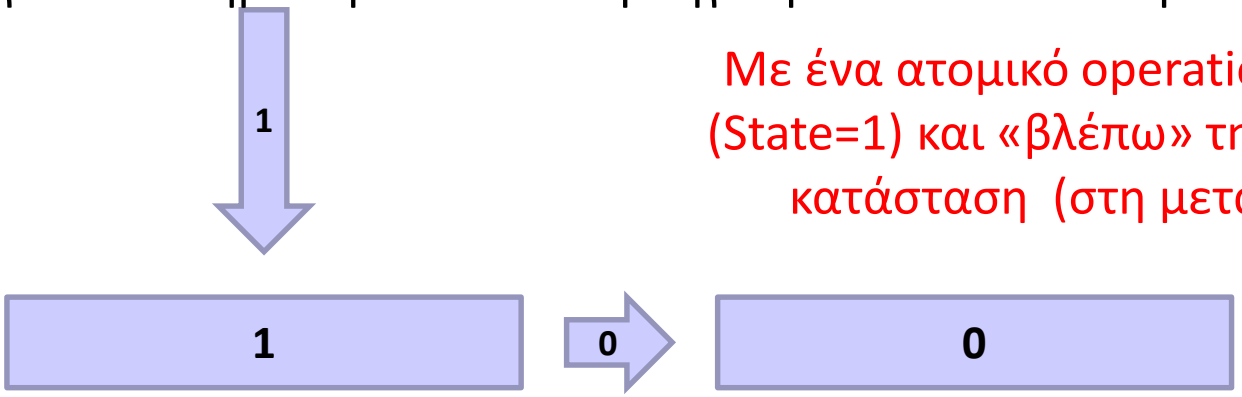
- **Ατομικά** ενεργούν σε **2** μεταβλητές
- Στις υλοποιήσεις τους τυπικά περιλαμβάνουν και **memory fence** (επιβολή για ολοκλήρωση όλων των προηγούμενων εντολών πρόσβασης στη μνήμη)



Με ένα ατομικό operation θέτω το lock (State=1) και «βλέπω» την προηγούμενη κατάσταση (στη μεταβλητή Test)

Υποστήριξη από το υλικό:
atomic operations, π.χ. test-and-set, compare-and-swap

- **Ατομικά** ενεργούν σε **2** μεταβλητές
- Στις υλοποιήσεις τους τυπικά περιλαμβάνουν και **memory fence** (επιβολή για ολοκλήρωση όλων των προηγούμενων εντολών πρόσβασης στη μνήμη)



Με ένα ατομικό operation θέτω το lock (State=1) και «βλέπω» την προηγούμενη κατάσταση (στη μεταβλητή Test)

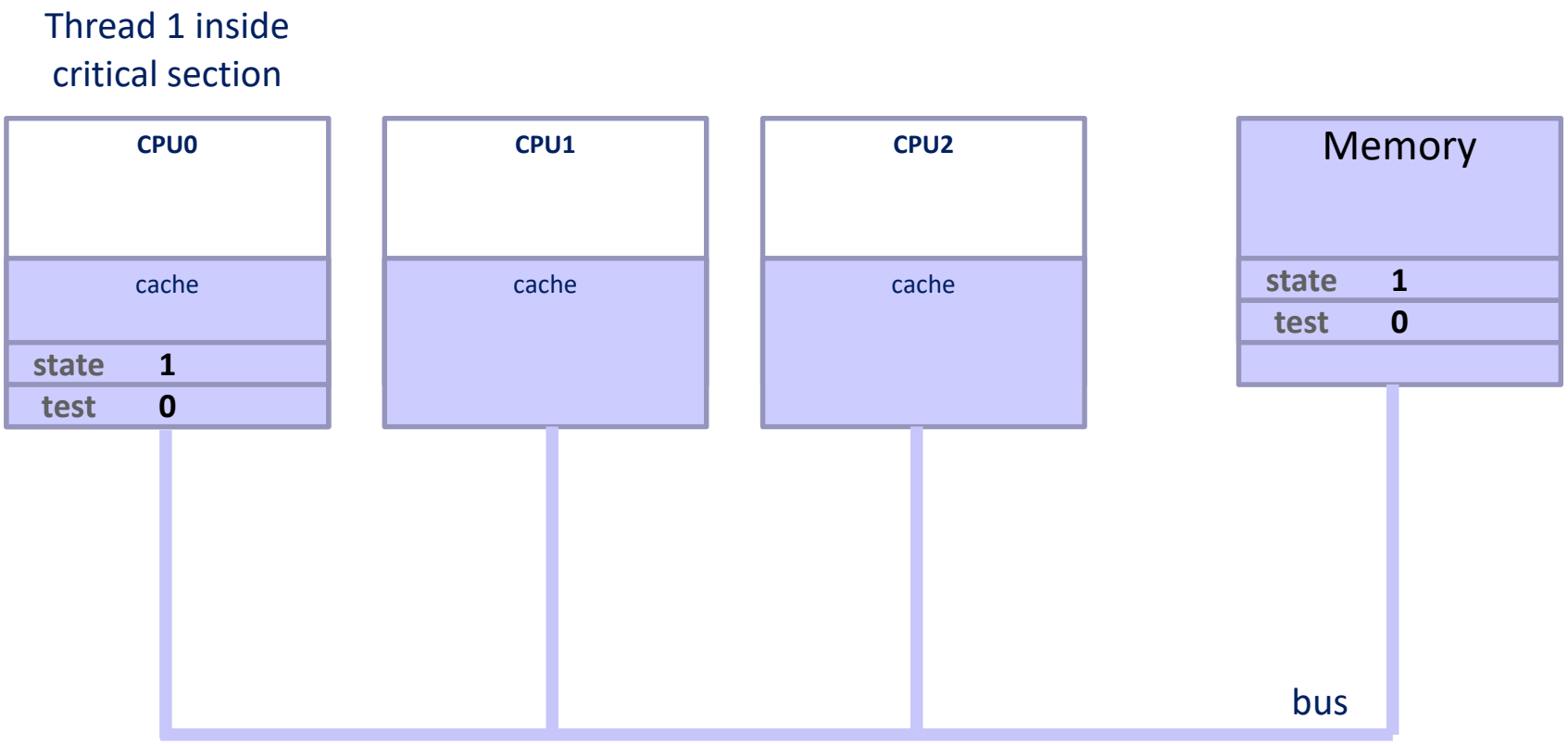
(0 = unlocked
1 = locked)

Αν test = 0 (ήταν «ξεκλείδωτα») μπαίνω στο critical section, αλλιώς περιμένω

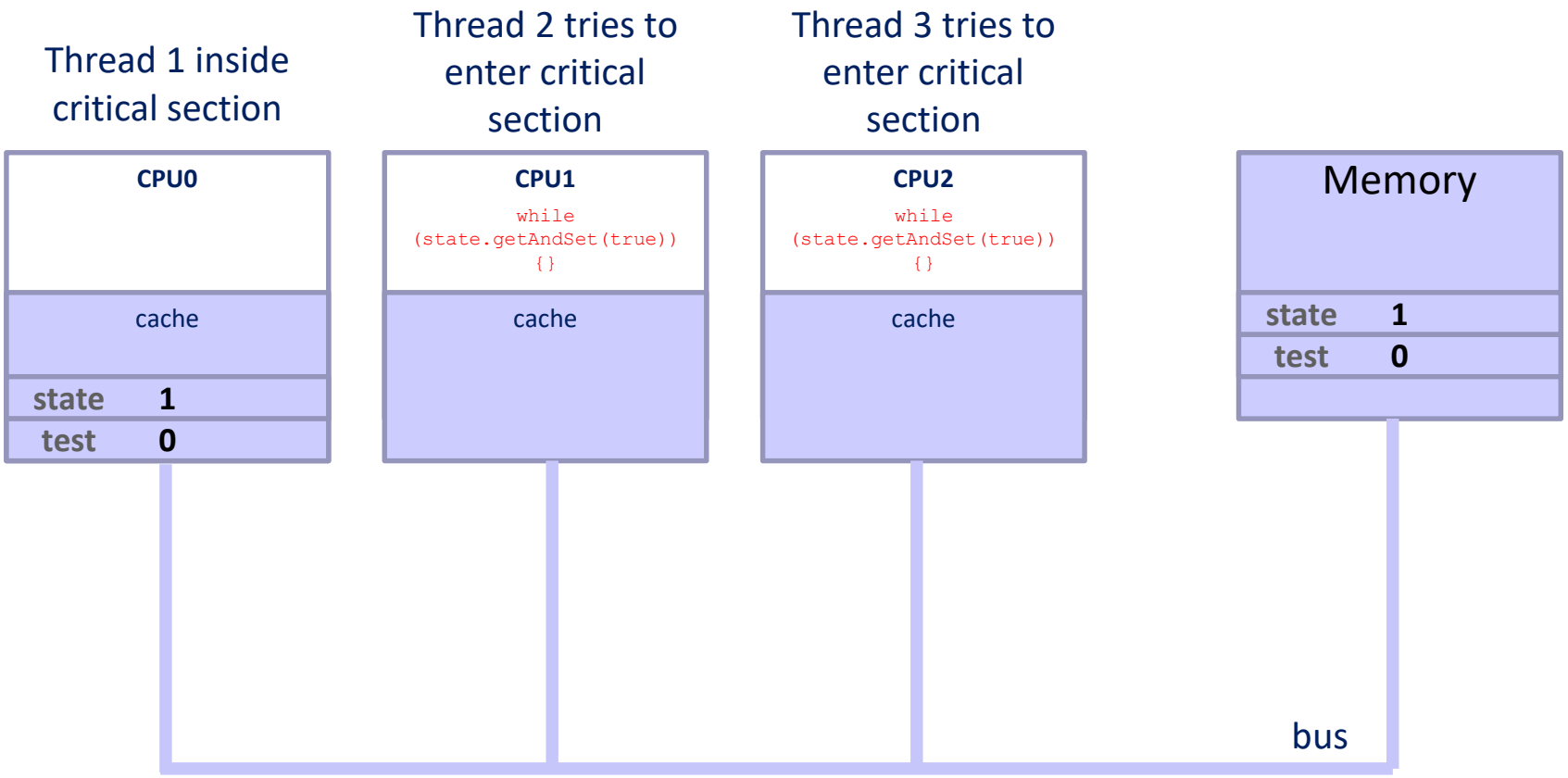
Test-and-set (TAS) lock

```
class TASlock {

 AtomicBoolean state = new AtomicBoolean(false);

 void lock() {
   while (state.getAndSet(true)) {}
 }

 void unlock() {
   state.set(false);
 }
}
```

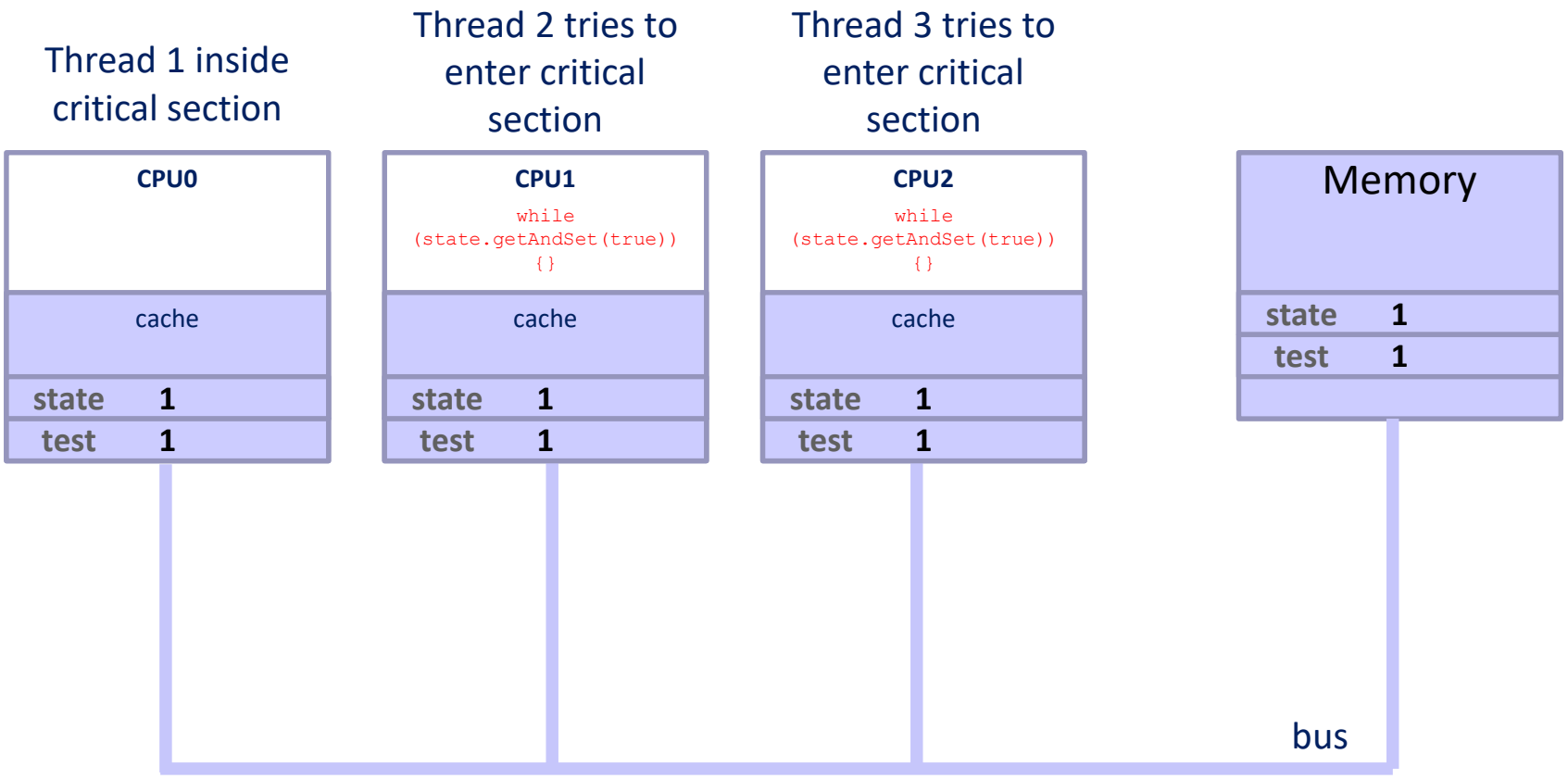
TAS lock in a bus-based multiprocessor



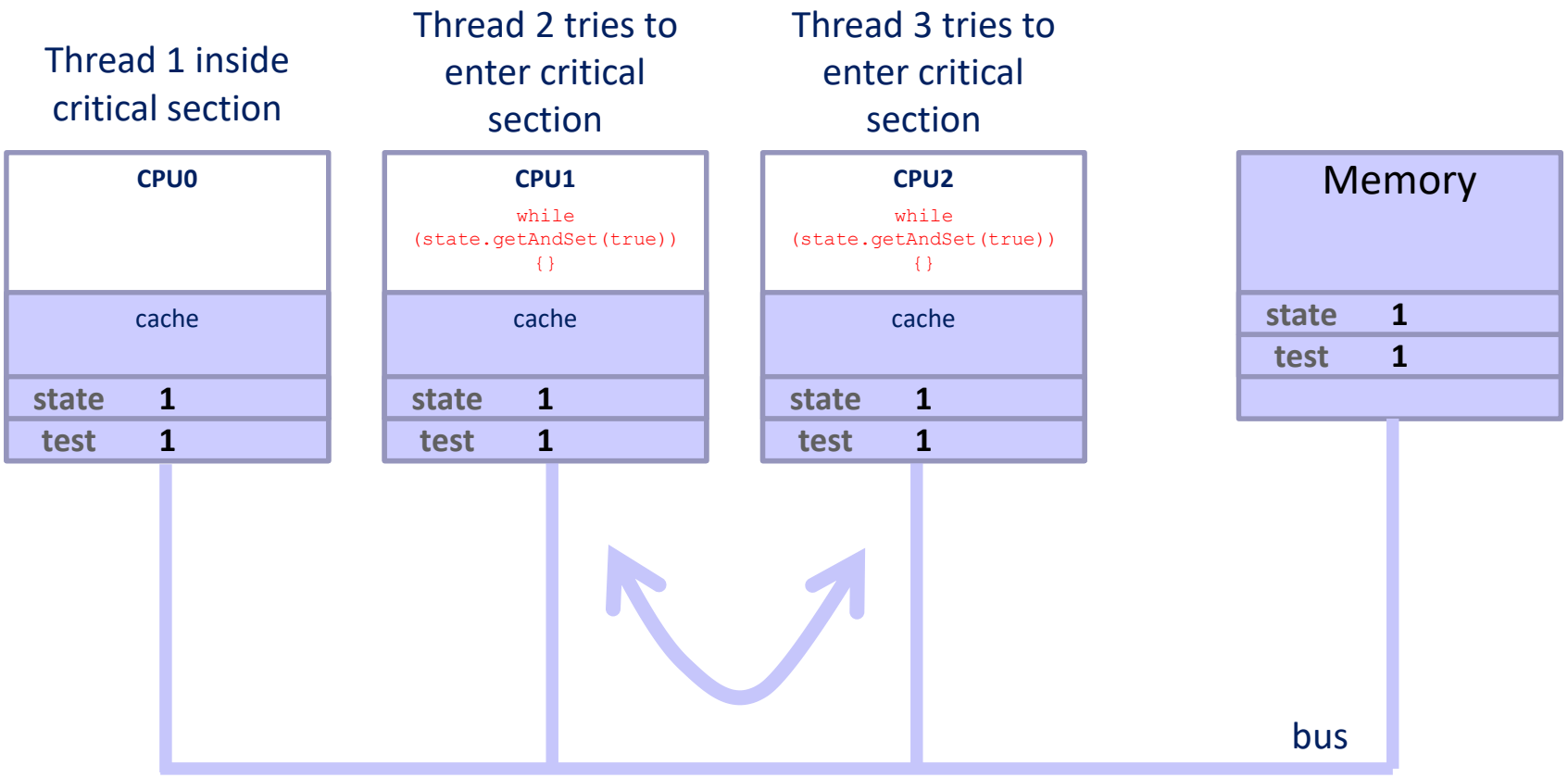
TAS lock in a bus-based multiprocessor



TAS lock in a bus-based multiprocessor



TAS lock in a bus-based multiprocessor



Υπερβολική χρήση του διαδρόμου λόγω πρωτοκόλλου συνάφειας κρυφής μνήμης

Test and test and set (TTAS) lock

```
class TTASlock {

 AtomicBoolean state = new AtomicBoolean(false);

 void lock() {
   while (true) {
     while (state.get()) {};
       if (!state.getAndSet(true))
         return;
 }

 void unlock() {
    state.set(false);
 }
}
```

Test and test and set (TTAS) lock

```
class TTASlock {

 AtomicBoolean state = new Atomi

 void lock() {
   while (true) {
     while (state.get()) {};
       if (!state.getAndSet(true))
         return;
 }

 void unlock() {
    state.set(false);
 }
}
```

«Επίμονο» διάβασμα της κατάστασης του lock

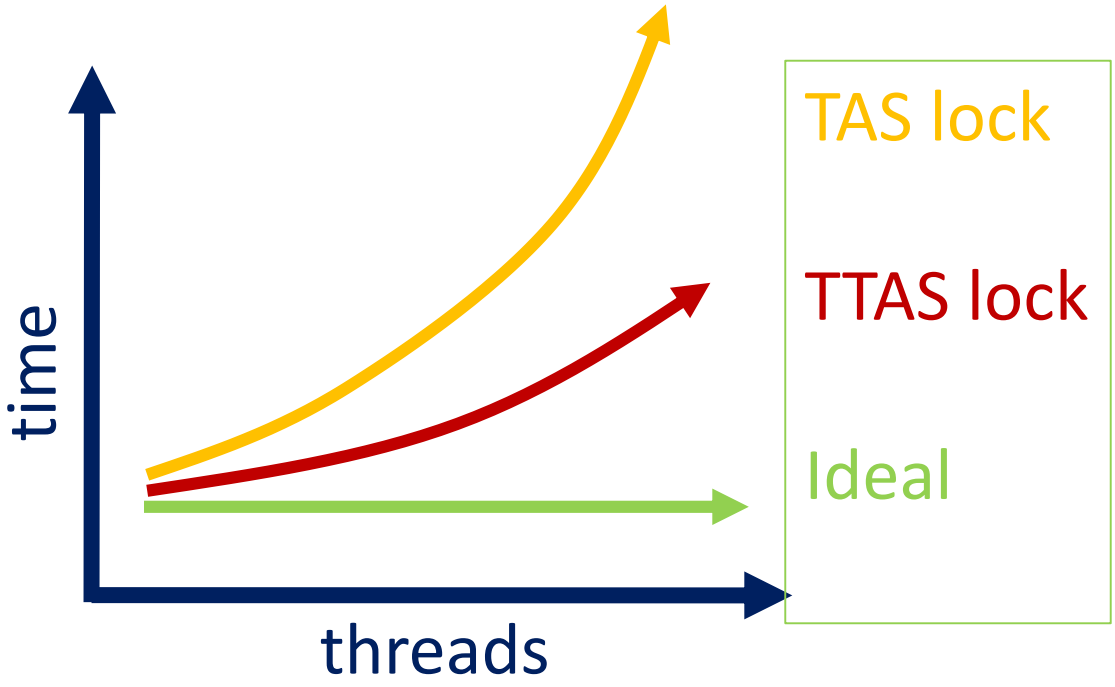
Test and test and set (TTAS) lock

```
class TTASlock {

 AtomicBoolean state = new Atomi

 void lock() {
   while (true) {
     while (state.get()) {};
       if (!state.getAndSet(true))
         return;
 }

 void unlock() {
    state.set(false);
 }
}
```

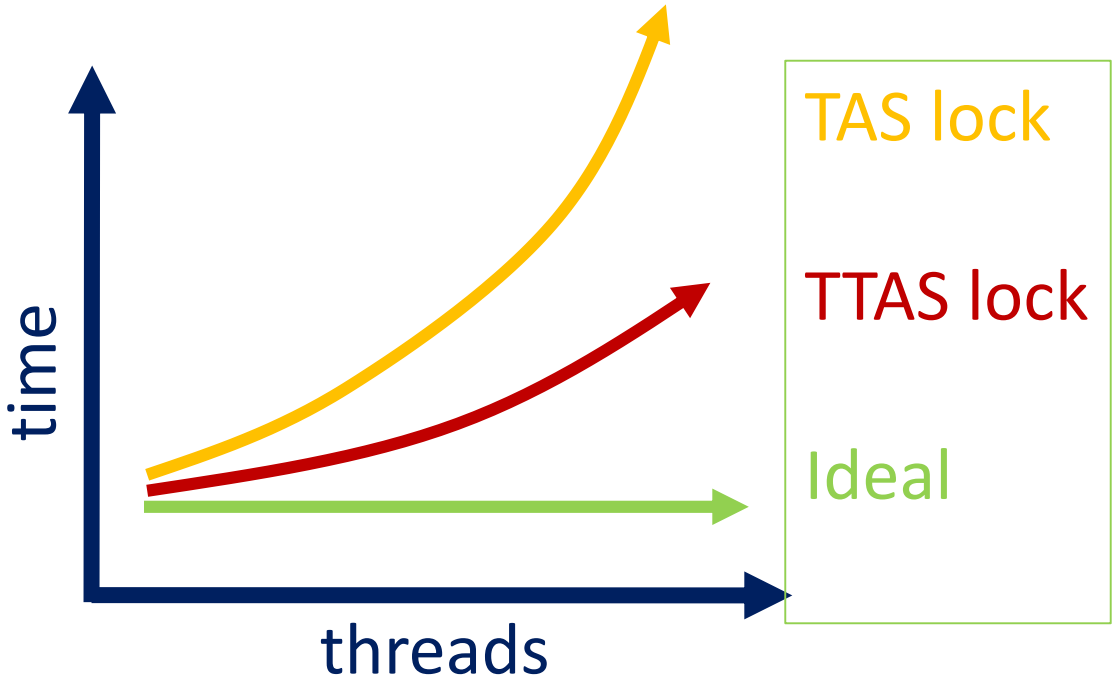
«Επίμονο» διάβασμα της κατάστασης του lock

Αν είναι «ξεκλείδωτο» το διεκδικώ

Επίδοση TAS vs TTAS



Επίδοση TAS vs TTAS



Περαιτέρω βελτίωση: TTAS + εκθετική οπισθοχώρηση (exponential backoff)

Scalable locks

- Σε ένα σύστημα κοινής μνήμης δεν είναι καλή ιδέα πολλά threads να «συνωστίζονται» σε μία κοινή θέση μνήμης
  - Δημιουργείται μεγάλη κυκλοφορία δεδομένων από το σύστημα συνάφειας κρυφής μνήμης
- Προτιμούμε τα νήματα να εργάζονται το καθένα στο δικό του χώρο και να έχουν πρόσβαση σε κοινές θέσεις **σπάνια** και με λογική **“point-to-point”**, π.χ. 2 [O(1)] νήματα ανά θέση μνήμης

Queue Locks: Array-based lock

```
class ALock {

 ThreadLocal<Integer> mySlotIndex;
 boolean[] flag;
 AtomicInteger tail;
 int size;

 public Alock (int capacity) {
   size = capacity;
   flag = new boolean[capacity];
   flag[0] = true;
   tail = new AtomicInteger(0);
 }

 public void lock() {
   int slot = tail.getAndIncrement() % size;
   mySlotIndex.set(slot);
   while (!flag[slot]){};
 }

 public void unlock() {
   int slot = mySlotIndex.get();
   flag[slot] = false;
   flag[(slot + 1) % size] = true;
 }
}
```

Queue Locks: Array-based lock

```
class ALock {

 ThreadLocal<Integer> mySlotIndex;
 boolean[] flag;
 AtomicInteger tail;
 int size;

 public Alock (int capacity) {
   size = capacity;
   flag = new boolean[capacity];
   flag[0] = true;
   tail = new AtomicInteger(0);
 }

 public void lock() {
   int slot = tail.getAndIncrement() % size;
   mySlotIndex.set(slot);
   while (!flag[slot]){};
 }

 public void unlock() {
   int slot = mySlotIndex.get();
   flag[slot] = false;
   flag[(slot + 1) % size] = true;
 }
}
```

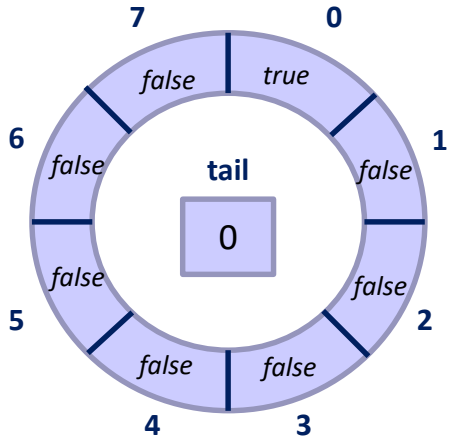
Τοπική μεταβλητή ανά thread

Πίνακας-ουρά με αριθμό θέσεων ίσο με τον αριθμό των threads

Δείκτης στο τέλος της ουράς

Queue Locks: Array-based lock

```
class ALock {

 ThreadLocal<Integer> mySlotIndex;
 boolean[] flag;
 AtomicInteger tail;
 int size;

 public Alock (int capacity) {
   size = capacity;
   flag = new boolean[capacity];
   flag[0] = true;
   tail = new AtomicInteger(0);
 }

 public void lock() {
   int slot = tail.getAndIncrement() % size;
   mySlotIndex.set(slot);
   while (!flag[slot]){};
 }

 public void unlock() {
   int slot = mySlotIndex.get();
   flag[slot] = false;
   flag[(slot + 1) % size] = true;
 }
}
```

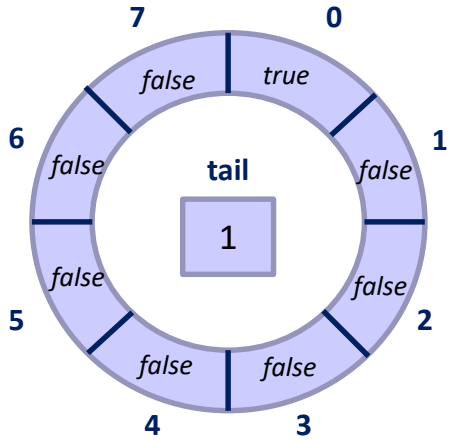



```
Alock mylock = new Alock(8);
```

```
/* Thread code */
mylock.lock();
/* critical section */
mylock.unlock();
```

Queue Locks: Array-based lock

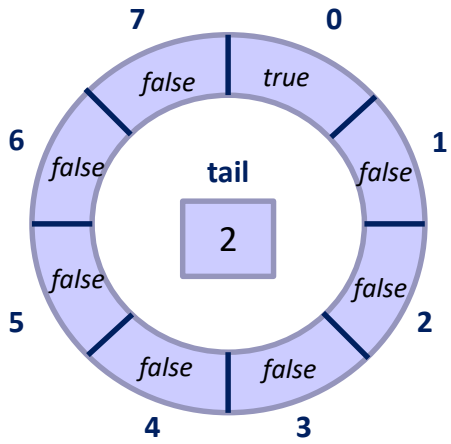
```
class ALock {

 ThreadLocal<Integer> mySlotIndex;
 boolean[] flag;
 AtomicInteger tail;
 int size;

 public Alock (int capacity) {
   size = capacity;
   flag = new boolean[capacity];
   flag[0] = true;
   tail = new AtomicInteger(0);
 }

 public void lock() {
   int slot = tail.getAndIncrement() % size;
   mySlotIndex.set(slot);
   while (!flag[slot]){};
 }

 public void unlock() {
   int slot = mySlotIndex.get();
   flag[slot] = false;
   flag[(slot + 1) % size] = true;
 }
}
```



Thread A
mylock.lock()

Queue Locks: Array-based lock

```
class ALock {

 ThreadLocal<Integer> mySlotIndex;
 boolean[] flag;
 AtomicInteger tail;
 int size;

 public Alock (int capacity) {
   size = capacity;
   flag = new boolean[capacity];
   flag[0] = true;
   tail = new AtomicInteger(0);
 }

 public void lock() {
   int slot = tail.getAndIncrement() % size;
   mySlotIndex.set(slot);
   while (!flag[slot]){};
 }

 public void unlock() {
   int slot = mySlotIndex.get();
   flag[slot] = false;
   flag[(slot + 1) % size] = true;
 }
}
```

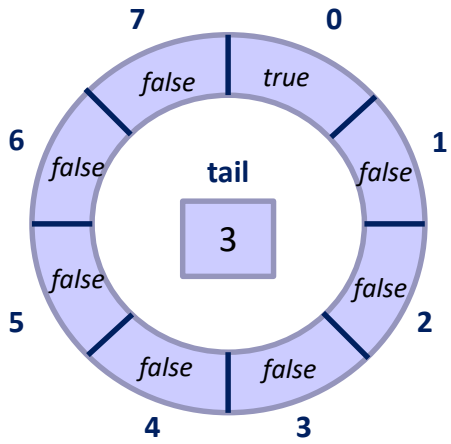


Thread A
in CS

Thread B
mylock.lock()

Queue Locks: Array-based lock

```
class ALock {

 ThreadLocal<Integer> mySlotIndex;
 boolean[] flag;
 AtomicInteger tail;
 int size;

 public Alock (int capacity) {
   size = capacity;
   flag = new boolean[capacity];
   flag[0] = true;
   tail = new AtomicInteger(0);
 }

 public void lock() {
   int slot = tail.getAndIncrement() % size;
   mySlotIndex.set(slot);
   while (!flag[slot]){};
 }

 public void unlock() {
   int slot = mySlotIndex.get();
   flag[slot] = false;
   flag[(slot + 1) % size] = true;
 }
}
```



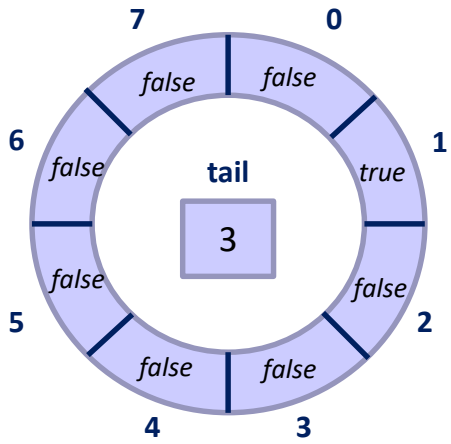
Thread A
in CS

Thread B
mylock.lock()

Thread C
mylock.lock()

Queue Locks: Array-based lock

```
class ALock {

 ThreadLocal<Integer> mySlotIndex;
 boolean[] flag;
 AtomicInteger tail;
 int size;

 public Alock (int capacity) {
   size = capacity;
   flag = new boolean[capacity];
   flag[0] = true;
   tail = new AtomicInteger(0);
 }

 public void lock() {
   int slot = tail.getAndIncrement() % size;
   mySlotIndex.set(slot);
   while (!flag[slot]){};
 }

 public void unlock() {
   int slot = mySlotIndex.get();
   flag[slot] = false;
   flag[(slot + 1) % size] = true;
 }
}
```



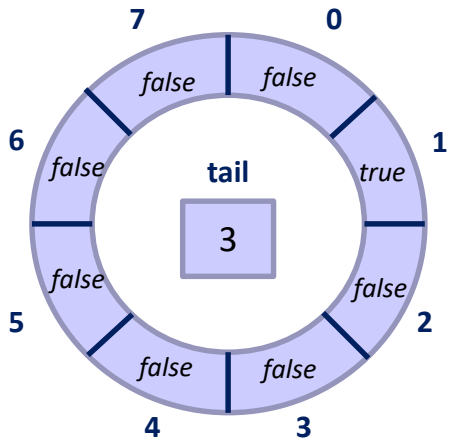
Thread A
mylock.unlock()

Thread B
mylock.lock()

Thread C
mylock.lock()

Queue Locks: Array-based lock

```
class ALock {

 ThreadLocal<Integer> mySlotIndex;
 boolean[] flag;
 AtomicInteger tail;
 int size;

 public Alock (int capacity) {
   size = capacity;
   flag = new boolean[capacity];
   flag[0] = true;
   tail = new AtomicInteger(0);
 }

 public void lock() {
   int slot = tail.getAndIncrement() % size;
   mySlotIndex.set(slot);
   while (!flag[slot]){};
 }

 public void unlock() {
   int slot = mySlotIndex.get();
   flag[slot] = false;
   flag[(slot + 1) % size] = true;
 }
}
```



Thread B
In CS

Thread C
mylock.lock()

Επισημάνσεις

- Η επίδοση κάθε μηχανισμού κλειδώματος εξαρτάται από τη συμφόρηση
  - Αριθμός νημάτων
  - Μέγεθος κρίσιμου τμήματος
  - Μέγεθος μη-κρίσιμου τμήματος
- Δεν υπάρχει ένα κλείδωμα για όλες τις περιπτώσεις
  - Υβριδικές προσεγγίσεις έχουν νόημα
- Οι παραπάνω υλοποιήσεις αφορούν **spinlocks**
  - ο επεξεργαστής είναι κατειλημμένος όσο ένα νήμα επιχειρεί να εισέλθει στο κρίσιμο τμήμα
- Μπορούν να συνδυαστούν ορθογώνια με προσεγγίσεις όπου το νήμα απελευθερώνει τον επεξεργαστή μετά από κάποιο χρονικό διάστημα
- Η υλοποίηση ενός κλειδώματος (και εν γένει μηχανισμού συγχρονισμού) απαιτεί μελέτη του memory model της αρχιτεκτονικής και κατάλληλη χρήση των διαθέσιμων εντολών του instruction set. **Σημαντικό**: Σαν προγραμματιστές δεν φτιάχνουμε κλειδώματα μόνοι μας (εκτός αν ξέρουμε καλά τι κάνουμε) αλλά χρησιμοποιούμε κλειδώματα που έχουν υλοποιηθεί από βιβλιοθήκες για την ISA στην οποία θα εκτελέσουμε το πρόγραμμά μας.

Monitors και Condition Variables

- Η ορθή υλοποίηση σύνθετων προγραμμάτων με τη χρήση κλειδωμάτων είναι δύσκολη και συχνά οδηγεί σε σφάλματα
- Είναι επιθυμητό να αναθέτουμε μέρος της υλοποίησης του συγχρονισμού σε κάποια βιβλιοθήκη ή στον compiler
  - Ο πραγματικός σχεδιασμός και ο εντοπισμός των αναγκών του συγχρονισμού παραμένει
- Ο προγραμματιστής προτιμά να περιγράφει το «τι» και όχι το «πως»

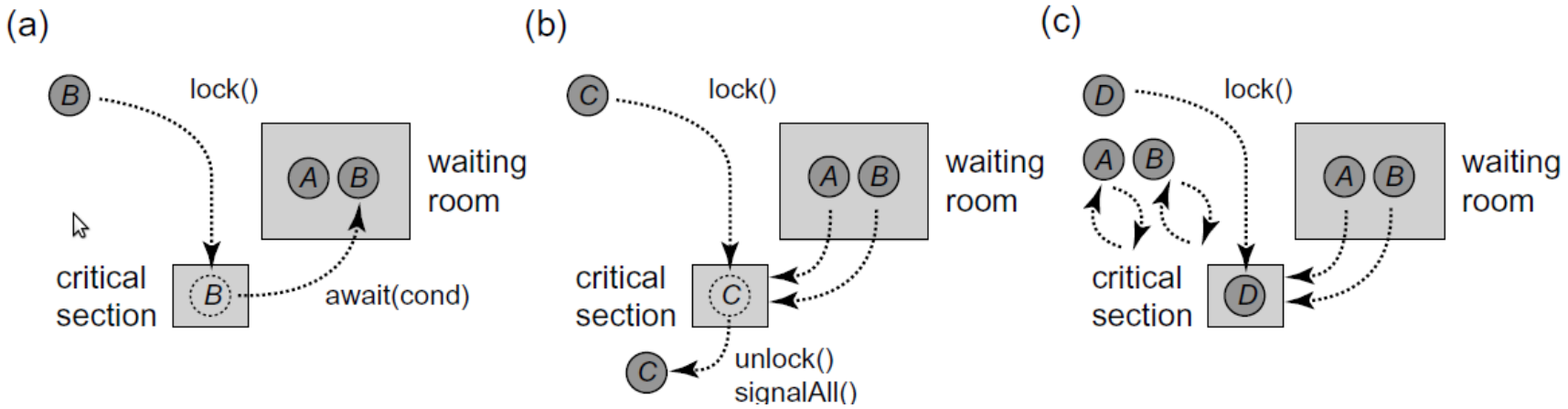
Monitors

- Τα monitors (παρακολουθητές) είναι δομές συγχρονισμού υψηλού επιπέδου
- Υποστηρίζουν:
  - **Ατομική εκτέλεση** των μεθόδων που ορίζονται εντός ενός monitor (η κατάλληλη χρήση των κλειδωμάτων για το σκοπό αυτό γίνεται από τον compiler αυτόματα)
  - Αναμονή ενός νήματος μέχρι να ικανοποιηθεί κάποια συνθήκη με τη χρήση **condition variables**

Condition variables

- Εξομοιώνουν μία αίθουσα αναμονής στην οποία εισέρχονται νήματα μέχρι να ικανοποιηθεί μία συνθήκη (π.χ. να απελευθερωθεί χώρος σε μία γεμάτη ουρά)
- Υποστηρίζουν λειτουργίες **αναστολής λειτουργίας (wait)** ενός νήματος αν δεν ισχύει μία συνθήκη και **ειδοποίησης των νημάτων σε αναμονή (signal)** όταν ισχύει η συνθήκη
- Τυπικά εμπλέκουν το λειτουργικό σύστημα για την αναστολή λειτουργίας και την ενημέρωση
- Συνδέονται με ένα κλείδωμα
  - Ο έλεγχος της συνθήκης γίνεται υπό την κατοχή κλειδώματος
  - Η αναστολή λειτουργίας έπεται της απελευθέρωσης του κλειδώματος
  - Η ανάνηψη από την αναμονή έπεται της επανάκτησης του κλειδώματος

Condition variables: Τρόπος λειτουργίας



Condition variables - διεπαφή

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws
   InterruptedException;
  boolean tryLock();
  boolean tryLock(long time, TimeUnit unit);
  Condition newCondition();
  void unlock();
}
public interface Condition {
  void await() throws InterruptedException;
  boolean await(long time, TimeUnit unit)
    throws InterruptedException;
  boolean awaitUntil(Date deadline)
    throws InterruptedException;
  long awaitNanos(long nanosTimeout)
    throws InterruptedException;
  void awaitUninterruptibly();
  void signal(); // wake up one waiting thread
  void signalAll(); // wake up all waiting threads
}
```

LockedQueue enq()

```
class Queue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;
  public Queue(int capacity) {
    items = (T[])new Object[capacity];
  }
  public void enq(T x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[tail] = x;
      if (++tail == items.length)
        tail = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

Λήψη lock

LockedQueue enq()

```
class Queue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;
  public Queue(int capacity) {
    items = (T[])new Object[capacity];
  }
  public void enq(T x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[tail] = x;
      if (++tail == items.length)
        tail = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

Έλεγχος συνθήκης και αναστολή λειτουργίας. Απελευθέρωση του lock εντός της await()

LockedQueue enq()

```
class Queue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;
  public Queue(int capacity) {
    items = (T[])new Object[capacity];
  }
  public void enq(T x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[tail] = x;
      if (++tail == items.length)
        tail = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

Εισαγωγή στοιχείου στην ουρά.
Αν η διεργασία έχει μόλις ξυπνήσει βρίσκεται με το lock σε κατοχή

LockedQueue enq()

```
class Queue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;
  public Queue(int capacity) {
    items = (T[])new Object[capacity];
  }
  public void enq(T x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[tail] = x;
      if (++tail == items.length)
        tail = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

Ειδοποίηση σε κάθε περίπτωση

LockedQueue deq()

```
class Queue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;
  public Queue(int capacity) {
    items = (T[])new Object[capacity];
  }
  public T deq() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      T x = items[head];
      if (++head == items.length)
        head = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Ανάλογα για το enq()

The lost wakeup problem

- Εμφανής βελτιστοποίηση στην υλοποίηση της προηγούμενης ουράς:
  - Ειδοποίηση των νημάτων υπό αναμονή μόνο στις οριακές καταστάσεις (εισαγωγή 1ου item στην ουρά, διαγραφή από πλήρη λίστα)
- Σε αυτή την περίπτωση μπορεί να υπάρξει η παρακάτω ακολουθία λειτουργιών σε άδεια ουρά:
  1. Consumer A.deq() // κοιμάται
  2. Consumer B.deq() // κοιμάται
  3. Producer C.enq() // items == 1
  4. Producer C.signal()
  5. Consumer A: ξυπνά αλλά δεν προλαβαίνει να πάρει το lock
  6. Producer D enq() // items = 2 (δεν κάνει signal γιατί δε βρήκε άδεια ουρά)
  7. Consumer A: παίρνει το lock και καταναλώνει ένα αντικείμενο //items = 1

  Ο Consumer B δεν ξυπνά ποτέ παρόλο που υπάρχει αντικείμενο να καταναλώσει!
- Λύση:
  - Ενημερώνουμε (ξυπνάμε) όλα τα νήματα σε αναμονή (signalAll)
  - Ορίζουμε timeout στην αναμονή

Τακτικές συγχρονισμού για δομές δεδομένων (1)

- Coarse-grain synchronization
  - Χρησιμοποιείται ένα κλείδωμα για όλη τη δομή
  - Εύκολη υλοποίηση
  - Περιορίζει την παράλληλη πρόσβαση των νημάτων στη δομή
- Fine-grain synchronization
  - Χρησιμοποιούνται περισσότερα κλειδώματα σε τμήματα της δομής
  - Εξασφαλίζεται παράλληλη πρόσβαση στη δομή από πολλαπλά νήματα
  - Δύσκολη υλοποίηση
  - Υψηλό κόστος κτήσης / απελευθέρωσης κλειδωμάτων
- Optimistic synchronization
  - Για εισαγωγή/διαγραφή: αναζήτηση του στοιχείου χωρίς κλείδωμα
  - Αν βρεθεί:
    - χρησιμοποιείται κλείδωμα
    - γίνεται έλεγχος αν το στοιχείο εξακολουθεί να είναι στη δομή
    - πραγματοποιείται η επεξεργασία του στοιχείου
  - Αναζήτηση: χρήση κλειδωμάτων

Τακτικές συγχρονισμού για δομές δεδομένων (2)

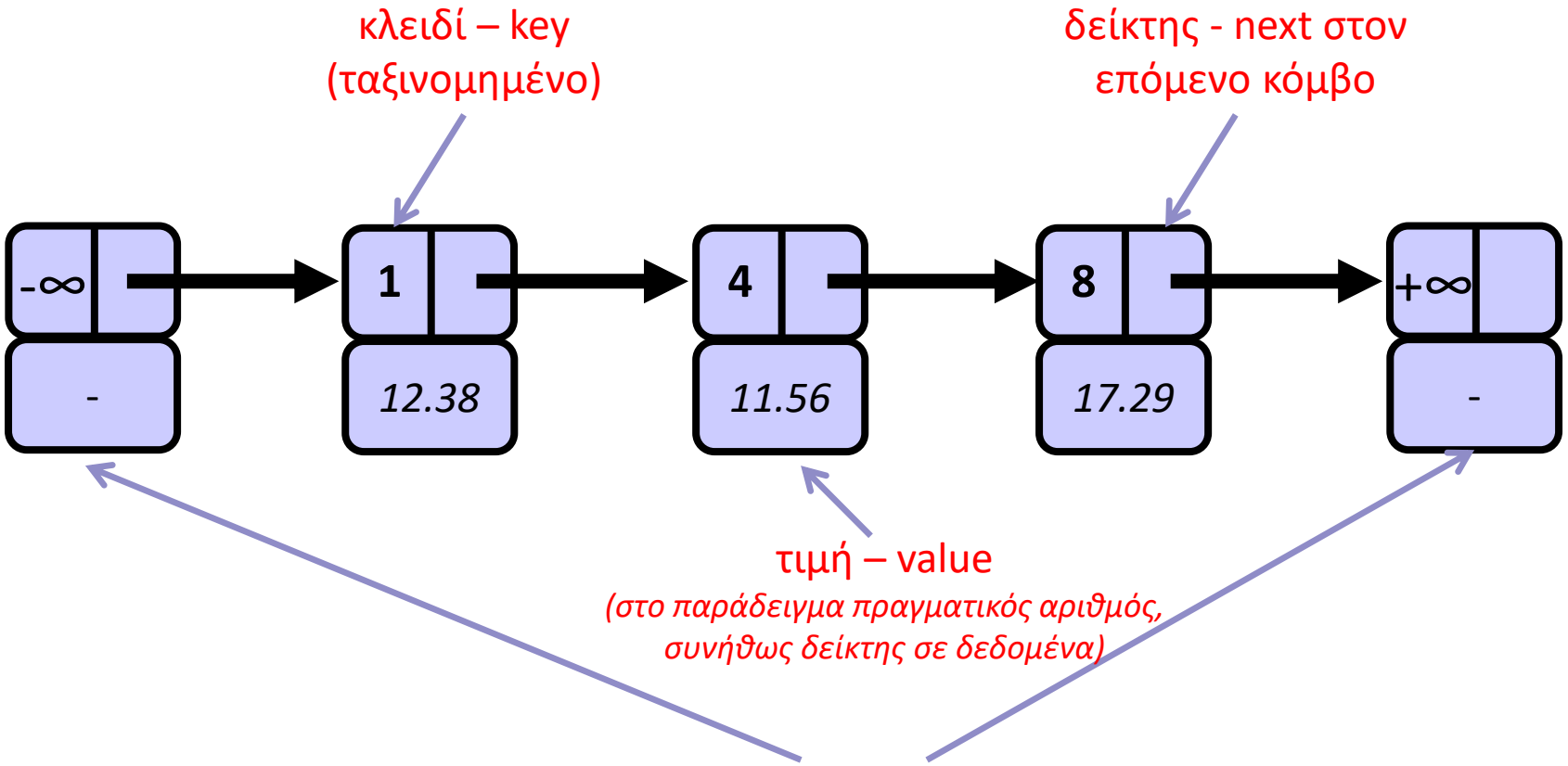
- Lazy synchronization
  - Η διαδικασία διαιρείται σε 2 μέρη (ελαφρύ, βαρύ)
  - Το ελαφρύ μέρος (π.χ. η λογική αφαίρεση ενός κόμβου μέσω της εγγραφής ενός tag) πραγματοποιείται αμέσως και με συγχρονισμό
  - Το βαρύ μέρος (π.χ. η φυσική αφαίρεση ενός κόμβου – διόρθωση των δεικτών) μπορεί να γίνει χωρίς συγχρονισμό **αργότερα (lazy)**
- Non-blocking synchronization
  - Σε κάποιες περιπτώσεις μπορούμε να αποφύγουμε τα κλειδώματα
  - Χρησιμοποιούνται οι ατομικές εντολές που παρέχει το υλικό (TAS, CAS)
- Transactional memory
  - Προγραμματιστικό μοντέλο που υποστηρίζει ατομικές εκτελέσεις τμημάτων κώδικα (transactions)
  - Τα transactions καταγράφονται
  - Αν υπήρξε ταυτόχρονη πρόσβαση η εκτέλεση αναιρείται και επανεκκινείται
  - Υλοποίηση σε λογισμικό (STM), υλικό (HTM) ή υβριδικά
  - Επεξεργαστές: SUN Rock (canceled by Oracle), IBM BlueGene/Q, IBM zEnterprise EC12, Intel Haswell/Broadwell, IBM Power

Σχήματα συγχρονισμού για συνδεδεμένες λίστες

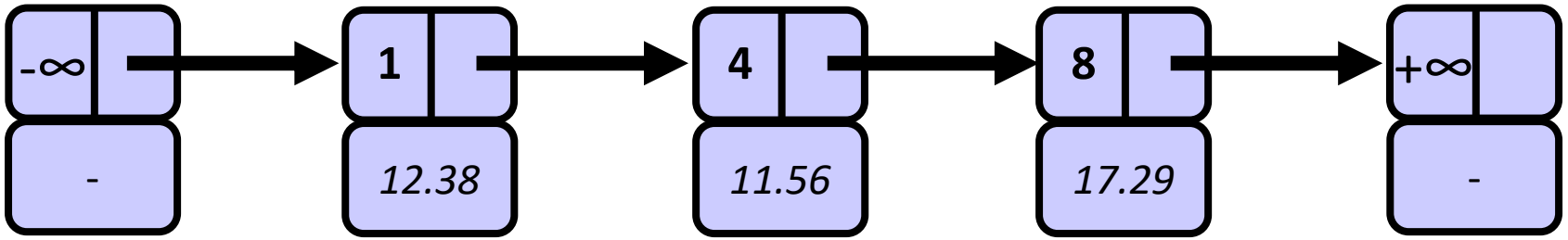
Δομή υπό εξέταση:

- Συνδεδεμένη λίστα με ταξινομημένα στοιχεία
- Κάθε κόμβος περιλαμβάνει:
  - κλειδί - key (συνήθως hash key, ταξινομημένο)
  - τιμή - value (τυπικά ένας δείκτης στο περιεχόμενο του κόμβου – εδώ χρησιμοποιούμε έναν πραγματικό αριθμό για απλότητα)
  - δείκτη στον επόμενο κόμβο
- Λειτουργίες:
  - **add** (value), προστίθεται αν δεν υπάρχει
  - **remove** (value), αφαιρείται αν υπάρχει
  - **contains** (value), επιστρέφει true αν υπάρχει το κλειδί (υπάρχει μονοπάτι που καταλήγει στο κλειδί), αλλιώς false
- Περιλαμβάνει 2 κόμβους φρουρούς για την αρχή και το τέλος με τη μέγιστη και ελάχιστη τιμή
- Η δομή μπορεί να εμπλουτιστεί / τροποποιηθεί προκειμένου να υποστηρίξει κάποιο σχήμα συγχρονισμού

Συνδεδεμένες λίστες (λεπτομέρειες υλοποίησης)

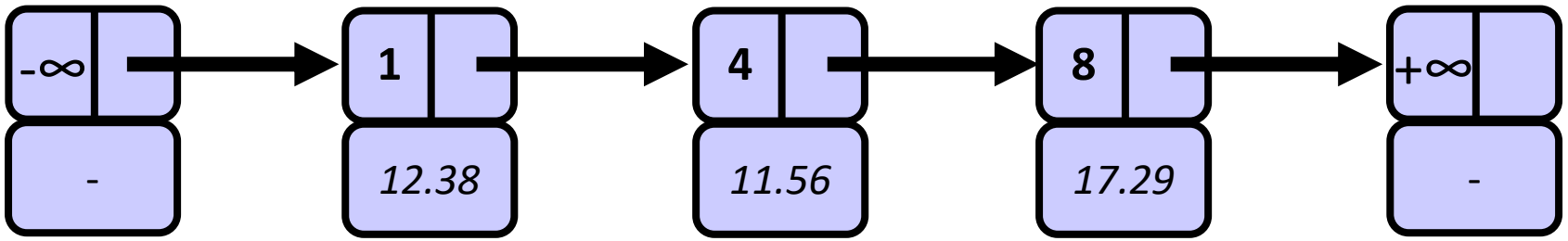


Συνδεδεμένες λίστες (λεπτομέρειες υλοποίησης)

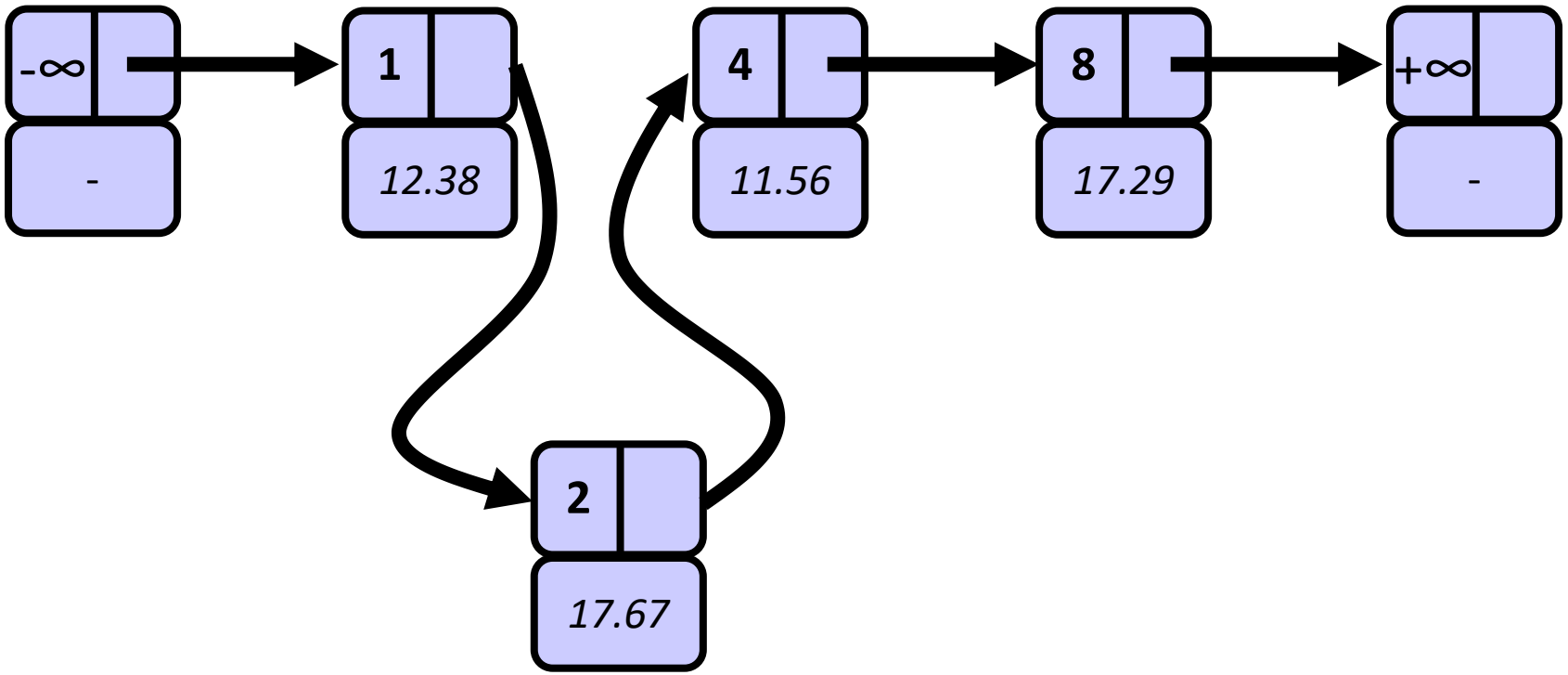



```
public class Node {
 public T value;
 public int key;
 public Node next;
}
```

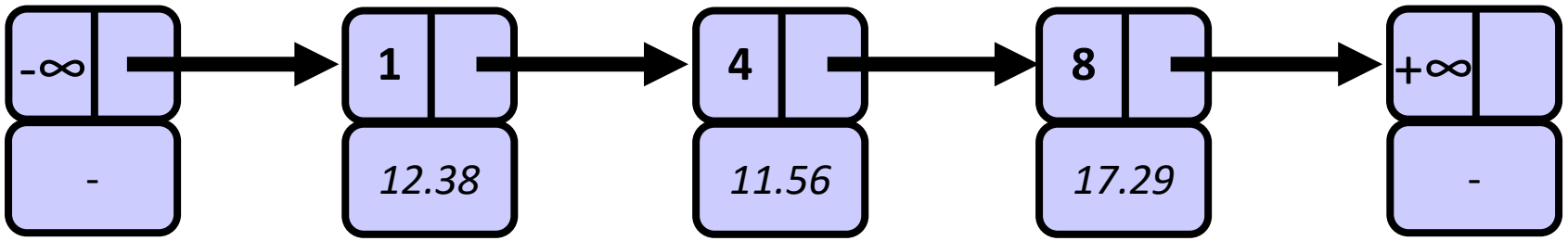
Συνδεδεμένες λίστες (προσθήκη στοιχείου)



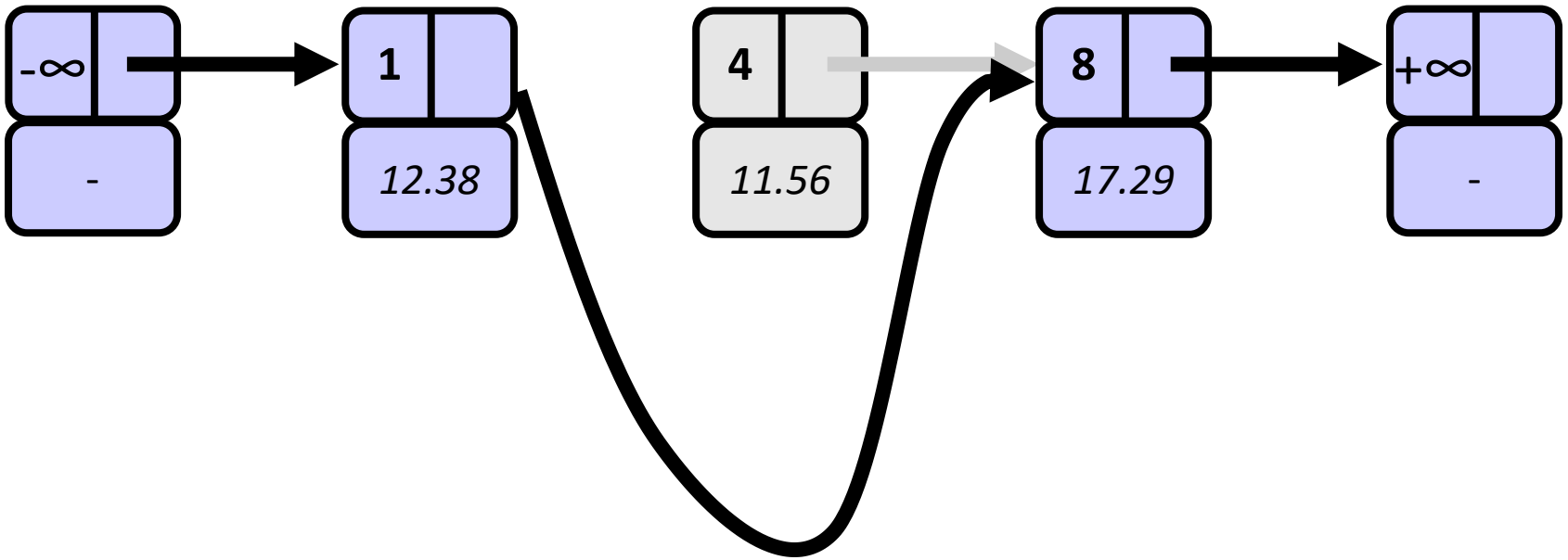
Συνδεδεμένες λίστες (προσθήκη στοιχείου)



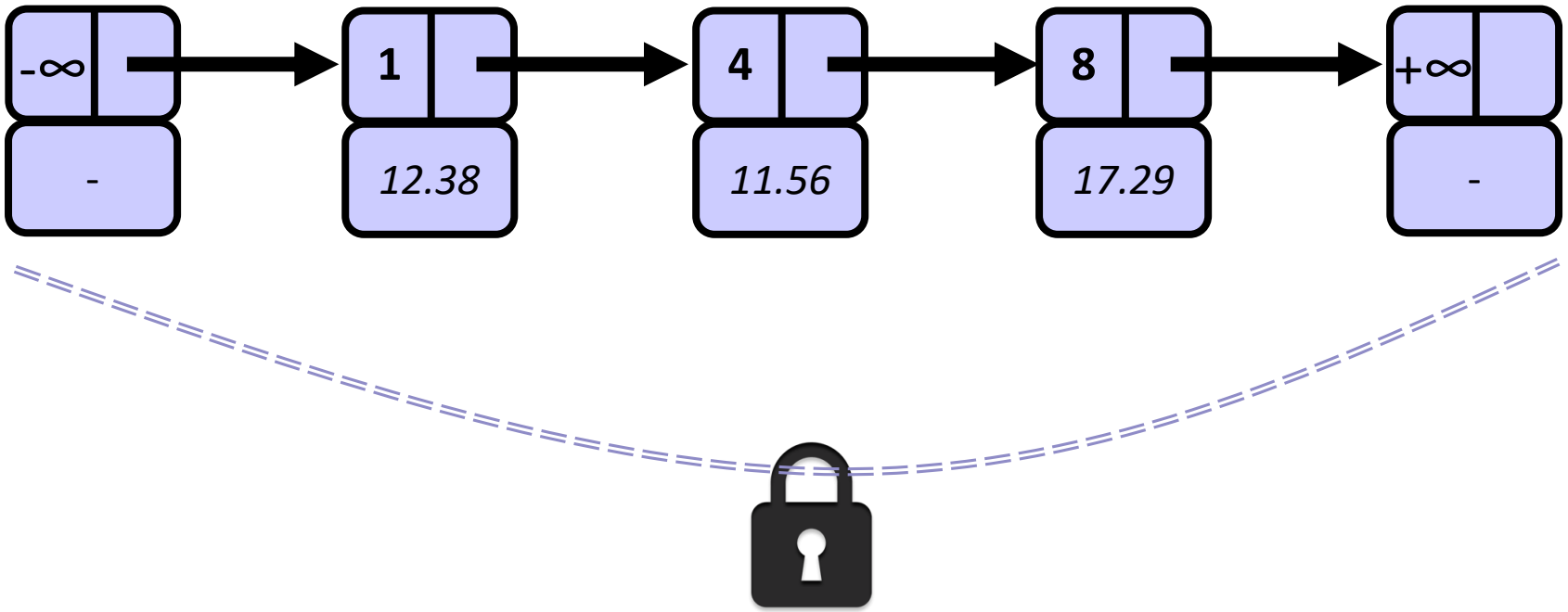
Συνδεδεμένες λίστες (αφαίρεση στοιχείου)



Συνδεδεμένες λίστες (αφαίρεση στοιχείου)



Coarse-grain synchronization



Ένα κλείδωμα για όλη τη δομή

Coarse-grain synchronization

```
public class CoarseList<T> {
  private Node head;
  private Node tail;
  private Lock lock = new
   ReentrantLock();

  public CoarseList() {
    head  = new
   Node(Integer.MIN_VALUE);
    tail  = new
   Node(Integer.MAX_VALUE);
    head.next = this.tail;
  }
…
}
```

Coarse-grain synchronization: add

```
public class CoarseList<T> {
  …
  public boolean add(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      if (key == curr.key) {
        return false;
      } else {
        Node node = new Node(item);
        node.next = curr;
        pred.next = node;
        return true;
      }
    } finally {
      lock.unlock();
    }
  }…
}
```

Coarse-grain synchronization: add

```
public class CoarseList<T> {
  …
  public boolean add(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      if (key == curr.key) {
        return false;
      } else {
        Node node = new Node(item);
        node.next = curr;
        pred.next = node;
        return true;
      }
    } finally {
      lock.unlock();
    }
  }…
}
```

Ολόκληρη η εκτέλεση της μεθόδου περιλαμβάνεται σε ένα κρίσιμο τμήμα

Coarse-grain synchronization: remove

```
public class CoarseList<T> {
  …
 public boolean remove(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = this.head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      if (key == curr.key) {
        pred.next = curr.next;
        return true;
      } else {
        return false;
      }
    } finally {
      lock.unlock();
    }
 }
…
}
```

Ολόκληρη η εκτέλεση της μεθόδου περιλαμβάνεται σε ένα κρίσιμο τμήμα

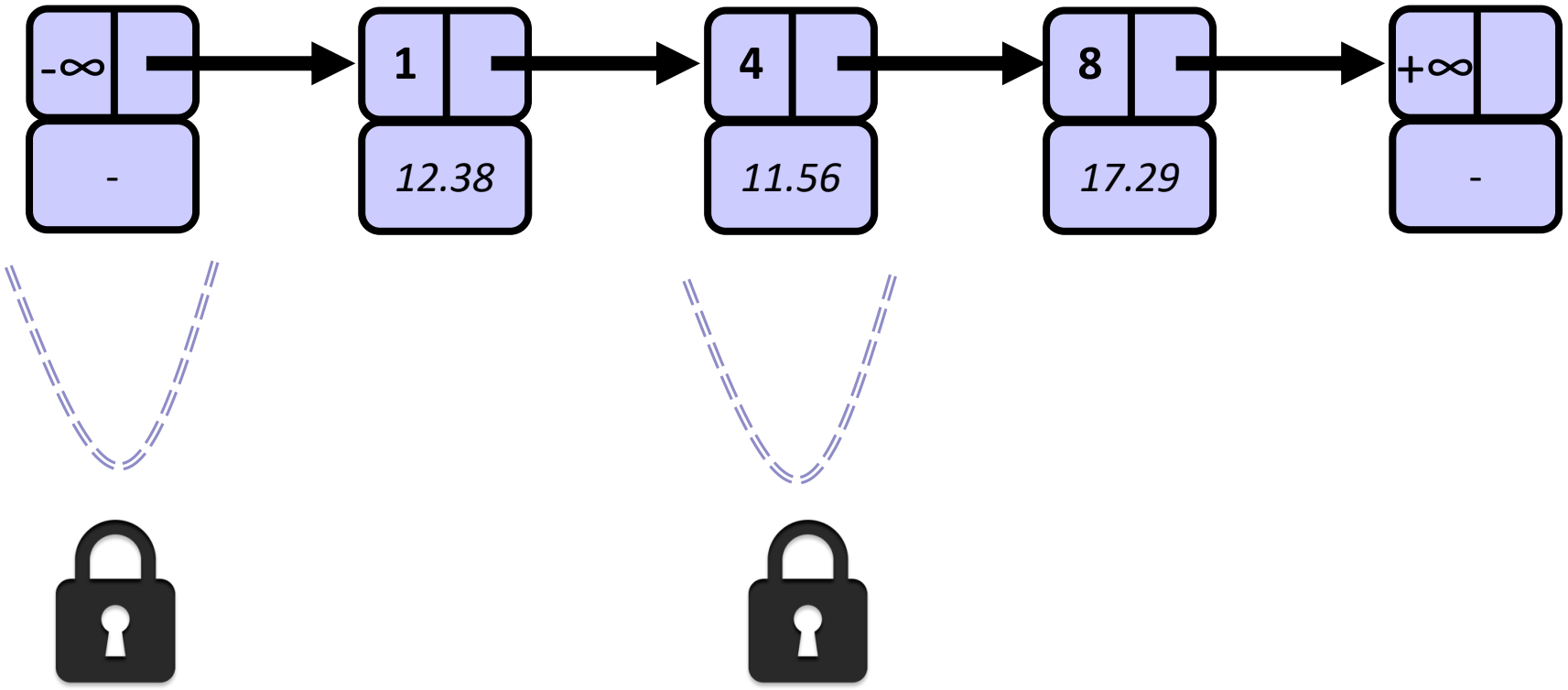
Ομοίως και για τη μέθοδο contains

Οι μέθοδοι διεκδικούν ένα κοινό lock άρα μόνο μία μπορεί να βρίσκεται σε εκτέλεση. Προφανής βελτίωση: readers-writers lock

Fine-grain synchronization

- Γιατί να κλειδώνουμε ολόκληρη τη δομή αν τα νήματα εργάζονται σε πλήρως ανεξάρτητα τμήματά της;
- **Ιδέα:** να χρησιμοποιήσουμε πολλαπλά κλειδώματα
- Τι ακριβώς πρέπει να κλειδώσουμε;

Fine-grain synchronization

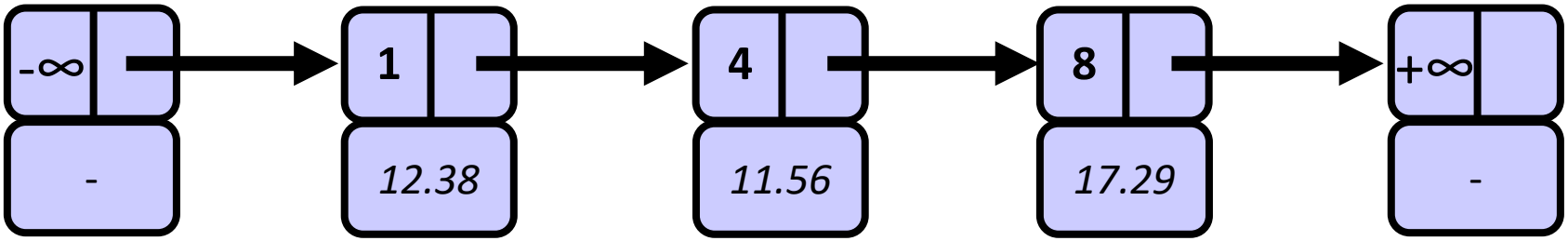


Πολλαπλά κλειδώματα για τη δομή

Fine-grain synchronization

Αρκεί ένα κλείδωμα ανά κόμβο;

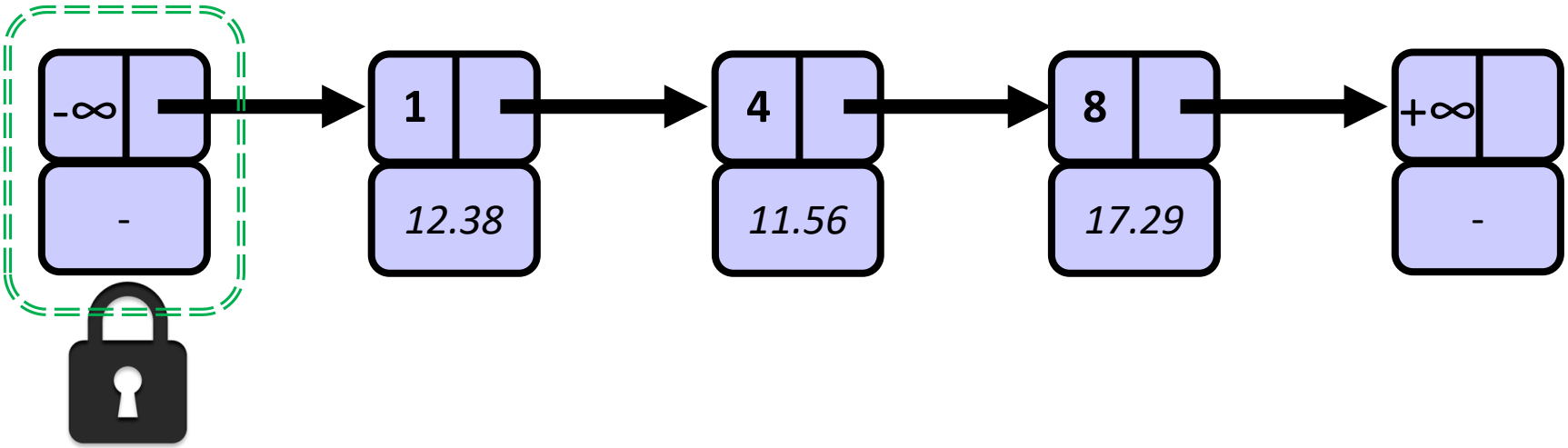
Έστω ότι το νήμα A θέλει να διαγράψει το **1** και το νήμα B θέλει να διαγράψει το **4**



Fine-grain synchronization

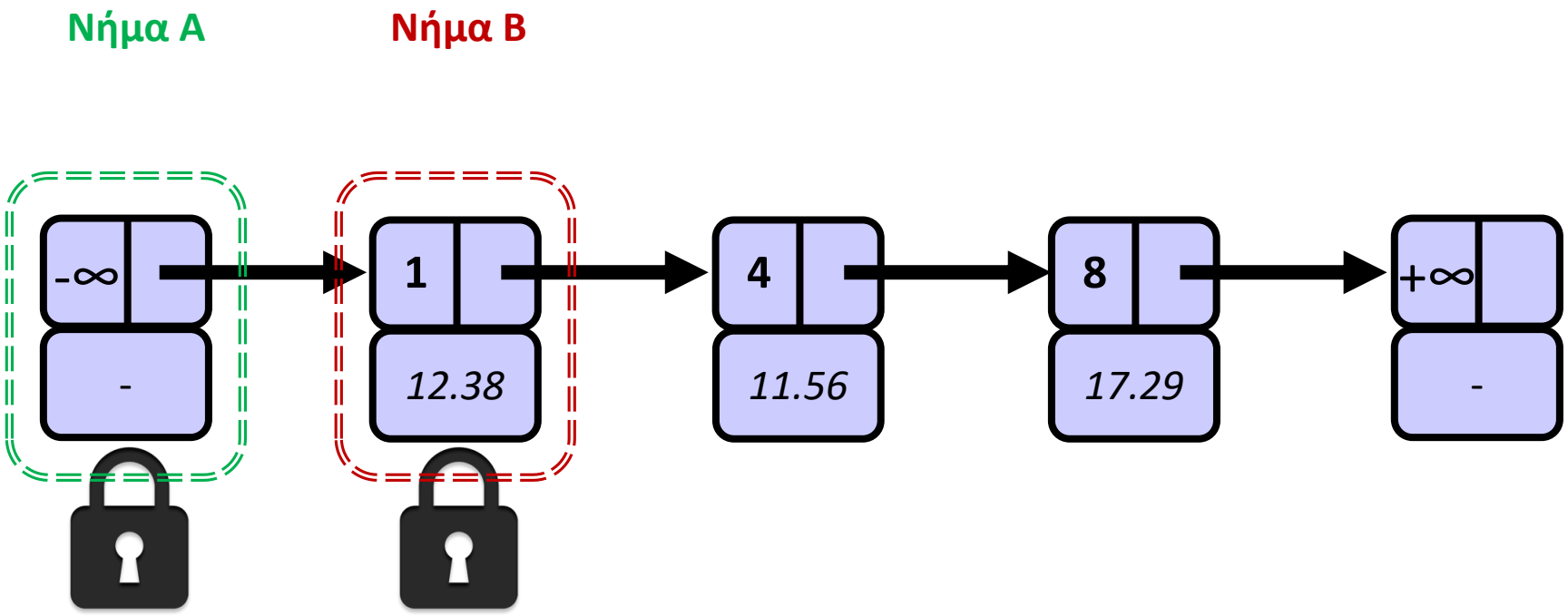
Αρκεί ένα κλείδωμα ανά κόμβο;

Νήμα Α



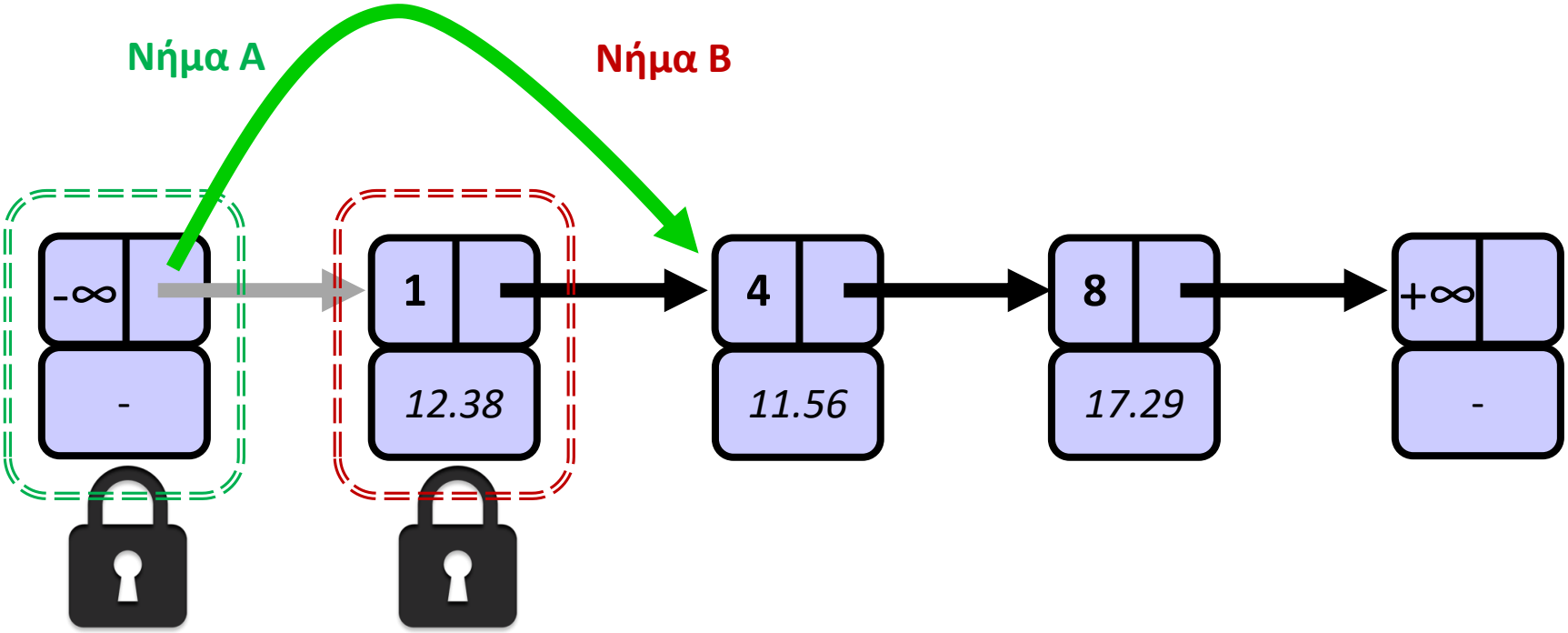
Fine-grain synchronization

Αρκεί ένα κλείδωμα ανά κόμβο;



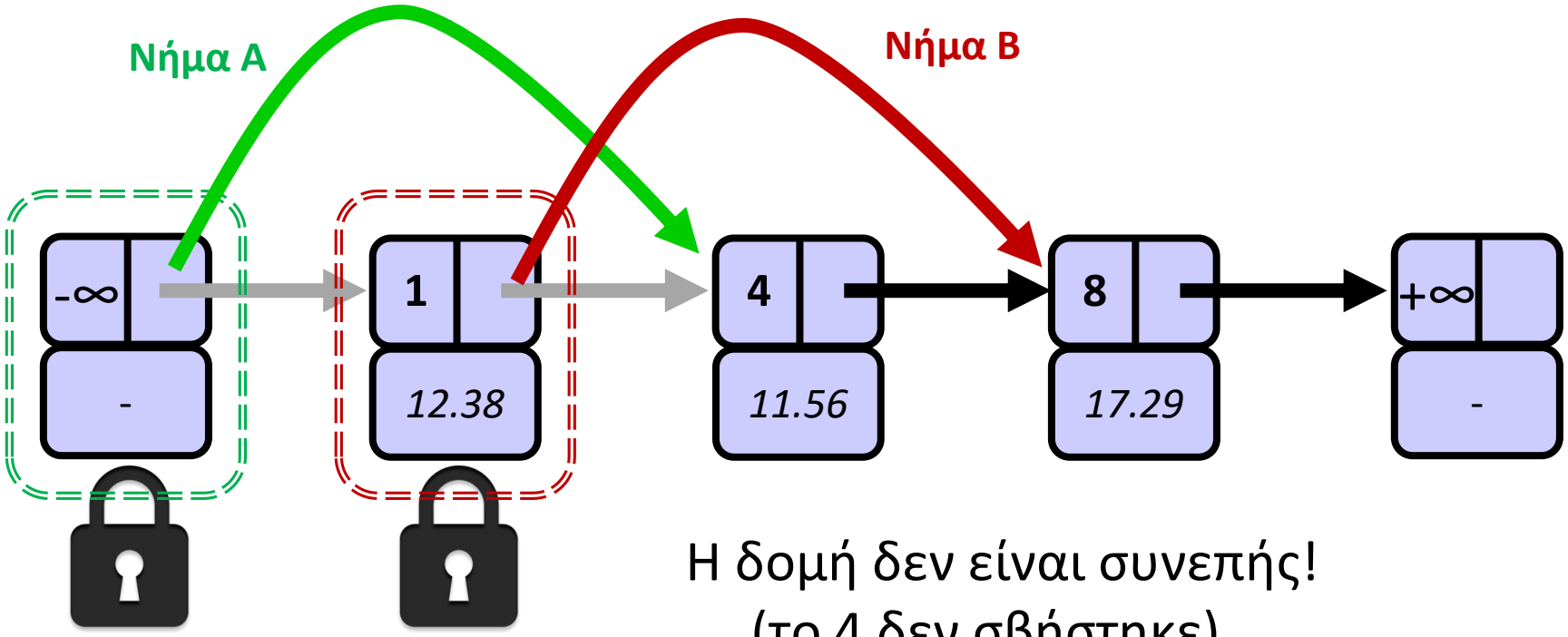
Fine-grain synchronization

Αρκεί ένα κλείδωμα ανά κόμβο;



Fine-grain synchronization

Αρκεί ένα κλείδωμα ανά κόμβο;



Η δομή δεν είναι συνεπής!
(το 4 δεν σβήστηκε)

Fine-grain synchronization

Αρκεί ένα κλείδωμα ανά κόμβο;

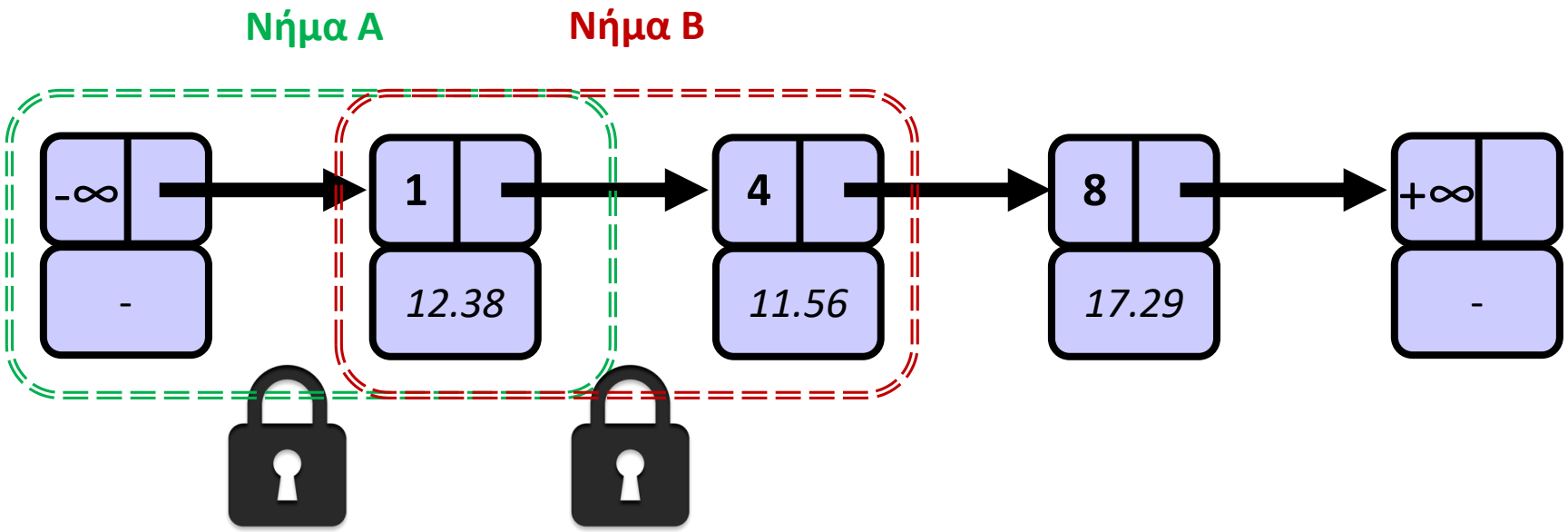
ΟΧΙ

Fine-grain synchronization

- **Παρατήρηση:**
- Αν ένας κόμβος είναι κλειδωμένος (για να εγγραφούν τα πεδία του), κανείς δεν πρέπει να αφαιρεί τον επόμενό του → το νήμα πρέπει να κλειδώνει τον προς αφαίρεση κόμβο και τον προηγούμενό του
- Αν πρόκειται να προστεθεί ένα κόμβος ανάμεσα σε δύο κόμβους, κανείς δεν πρέπει να προσθέσει κόμβο στο ίδιο σημείο → το νήμα πρέπει να κλειδώνει τους κόμβους που βρίσκονται εκατέρωθεν αυτού που θα προστεθεί (τον αμέσως μικρότερο και τον αμέσως μεγαλύτερο)

Λύση: *"hand-over-hand"* locking

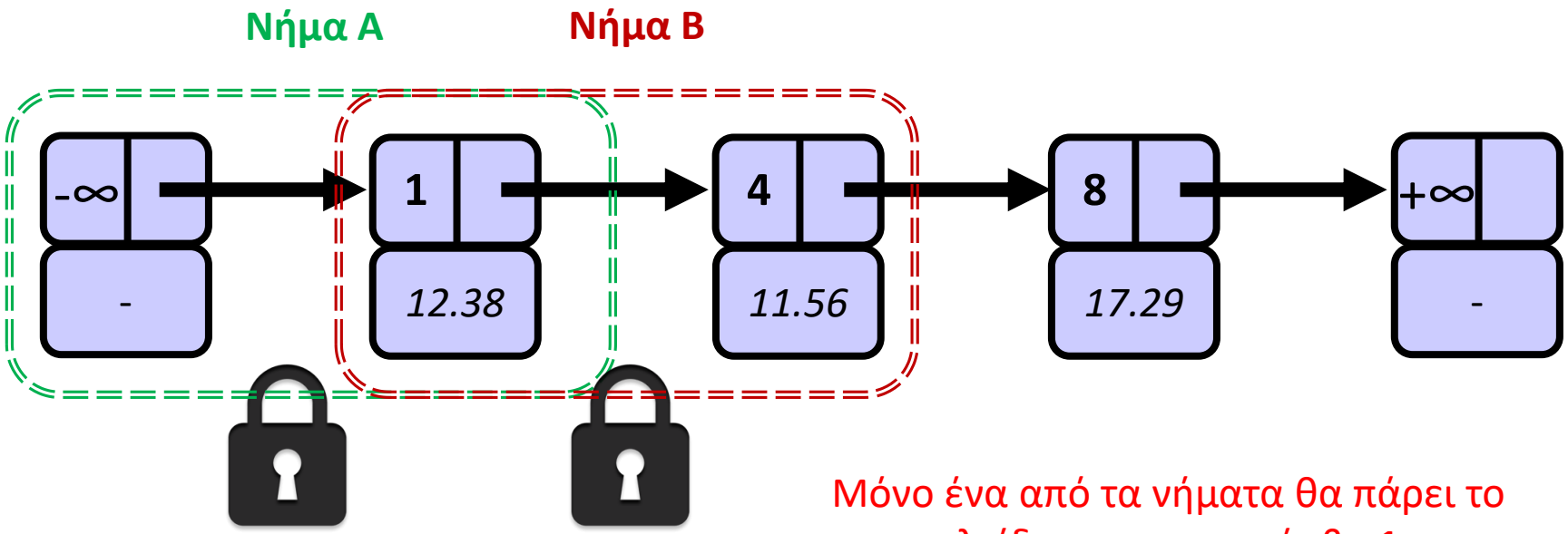
Έστω ότι το νήμα Α θέλει να διαγράψει το **1** και το νήμα Β θέλει να διαγράψει το **4**



Fine-grain synchronization

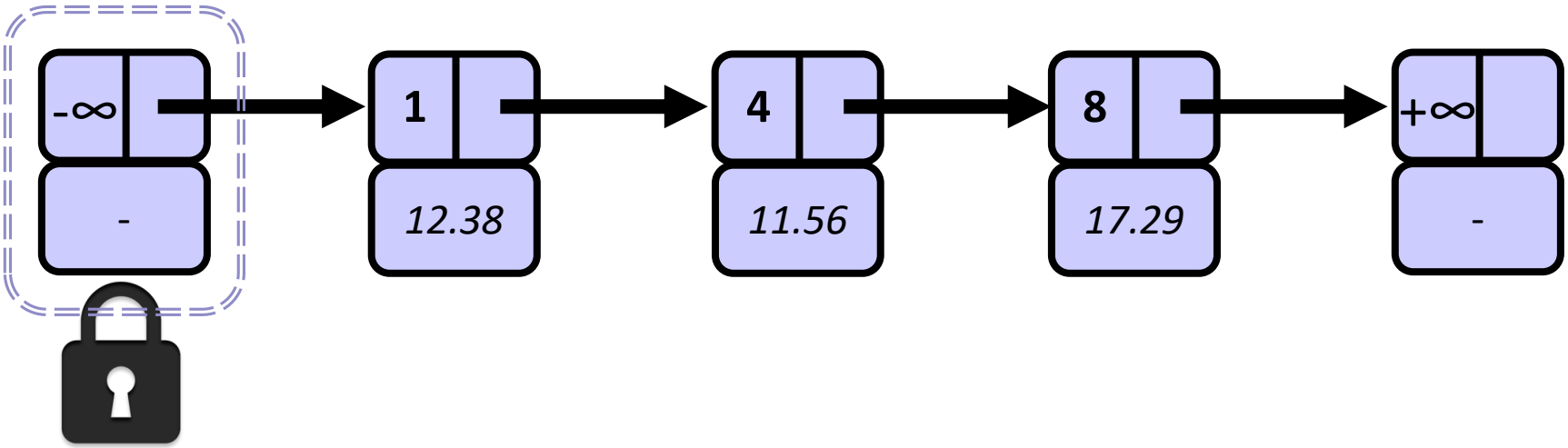
Λύση: “hand-over-hand” locking

Έστω ότι το νήμα A θέλει να διαγράψει το **1** και το νήμα B θέλει να διαγράψει το **4**

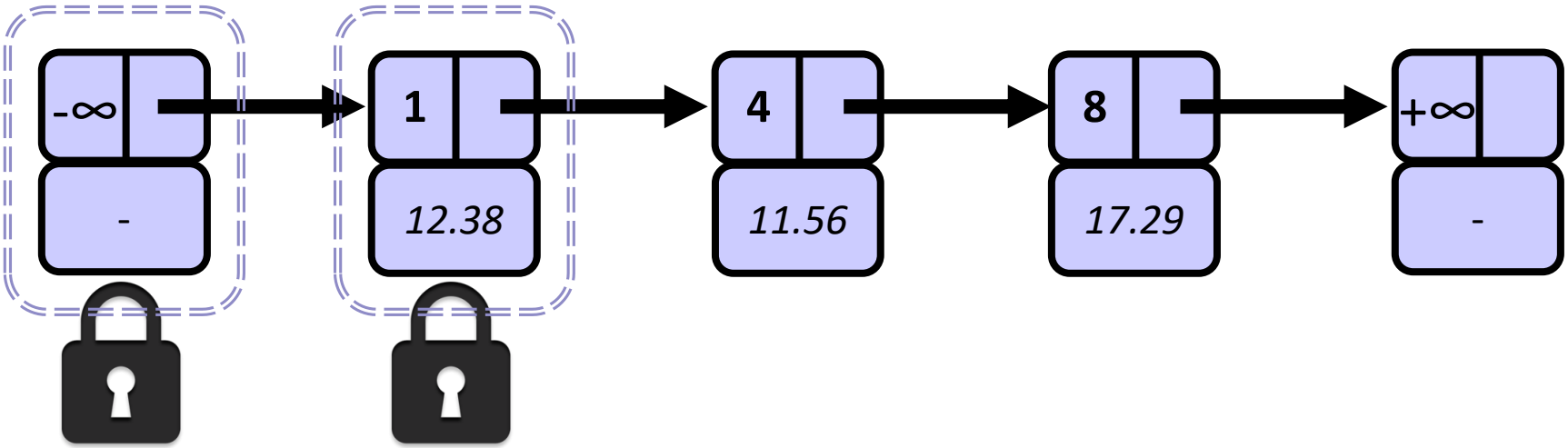


Μόνο ένα από τα νήματα θα πάρει το κλείδωμα για τον κόμβο **1**

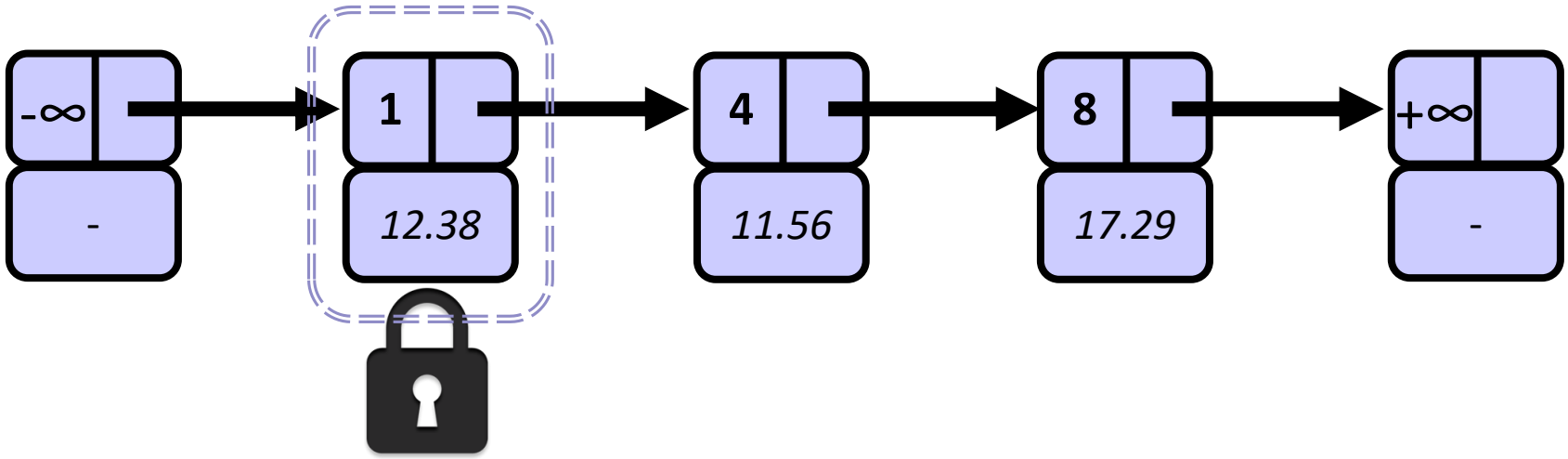
Διάσχιση λίστας



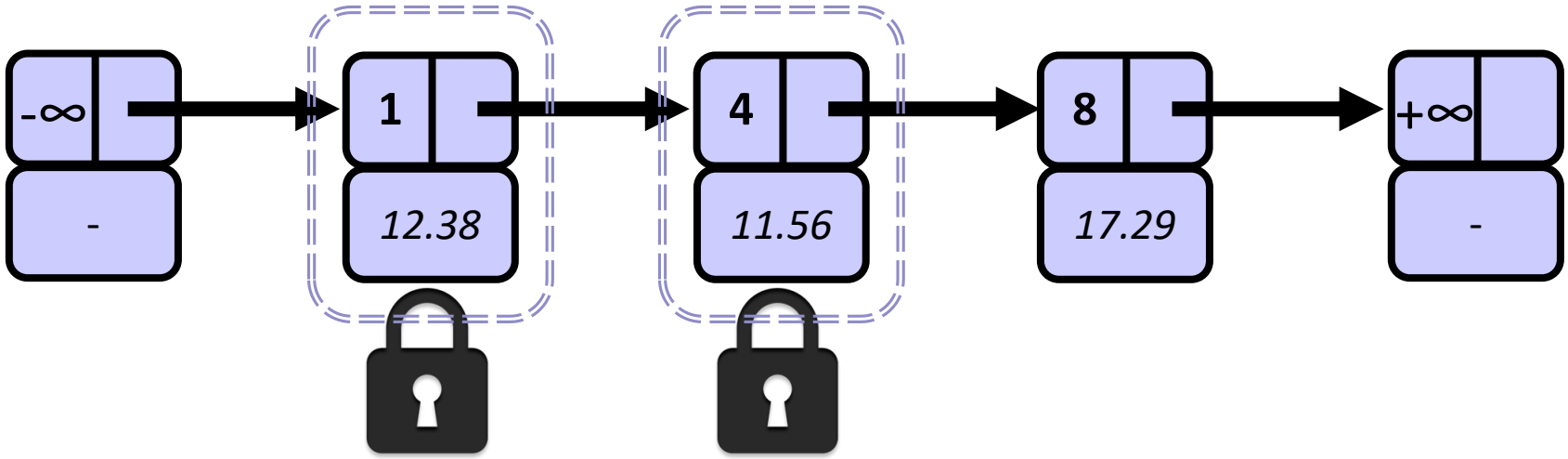
Διάσχιση λίστας



Διάσχιση λίστας

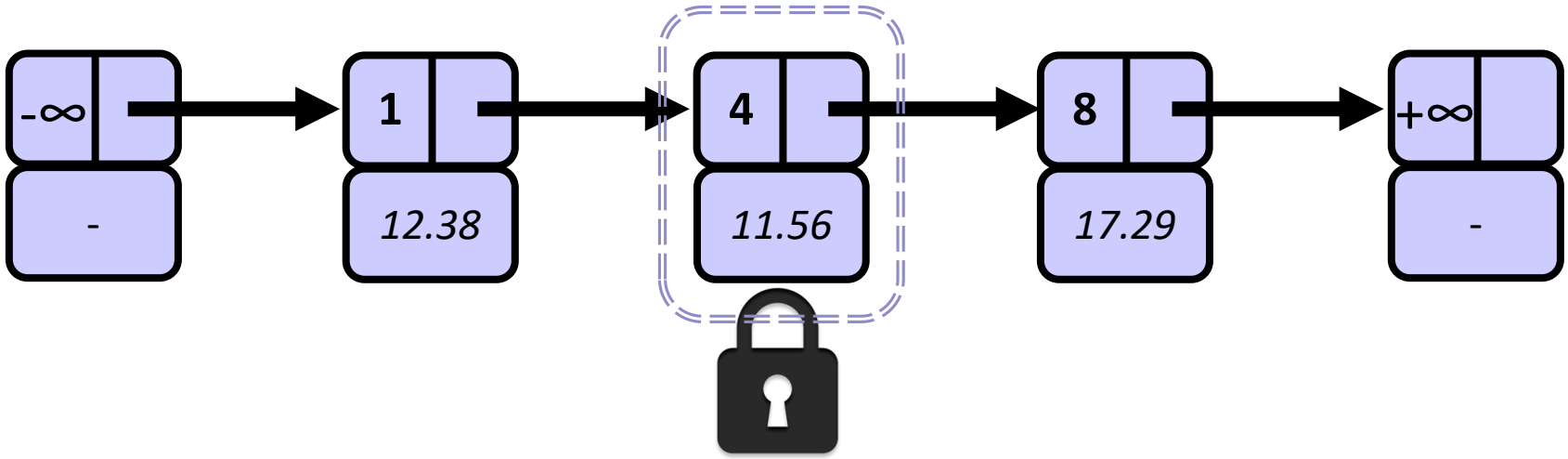


Διάσχιση λίστας

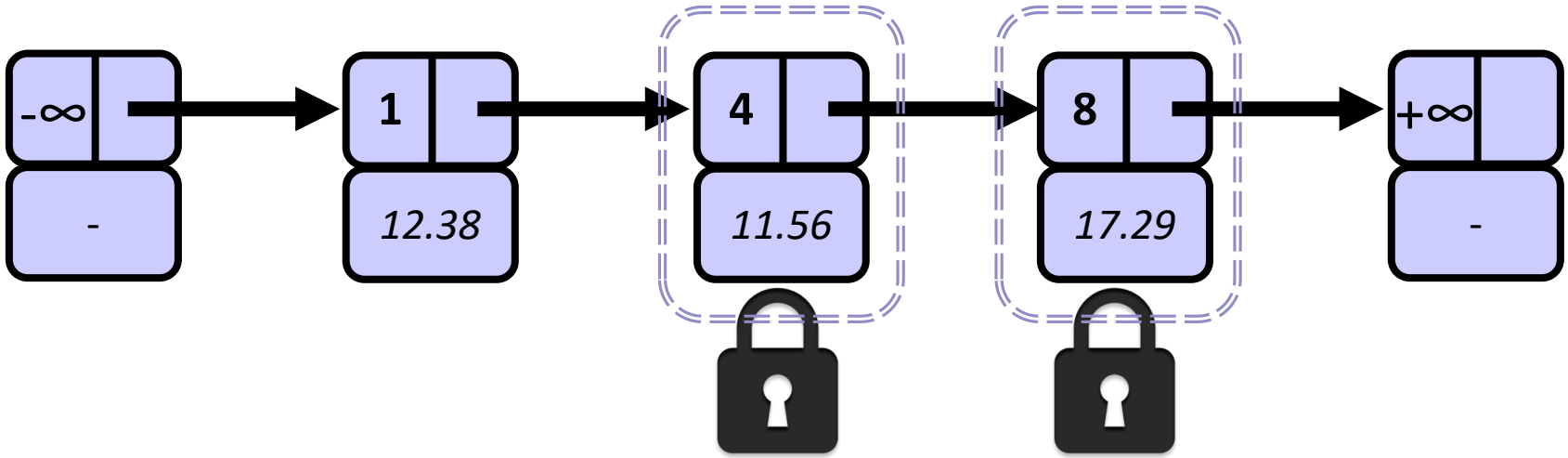


Fine-grain synchronization

Διάσχιση λίστας



Διάσχιση λίστας



Fine-grain synchronization: remove

```
public boolean remove(T item) {
   Node pred, curr;
   int key = item.hashCode();
   head.lock();
   try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            pred.next = cur.next;
            return true;
         }
         return false;
      } finally {
         curr.unlock();
      }
   } finally {
      pred.unlock();
   }
}
```

Fine-grain synchronization: remove

```
public boolean remove(T item) {
   Node pred, curr;
   int key = item.hashCode();
   head.lock();
   try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            pred.next = cur.next;
            return true;
         }
         return false;
      } finally {
         curr.unlock();
      }
   } finally {
      pred.unlock();
   }
}
```

Κλείδωμα της κεφαλής της λίστας

Fine-grain synchronization: remove

```
public boolean remove(T item) {
   Node pred, curr;
   int key = item.hashCode();
   head.lock();
   try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            pred.next = cur.next;
            return true;
         }
         return false;
      } finally {
         curr.unlock();
      }
   } finally {
      pred.unlock();
   }
}
```

Πρόσβαση στην κεφαλή
Πρόσβαση στο 2ο στοιχείο
Κλείδωμα 2ου στοιχείου

Fine-grain synchronization: remove

```
public boolean remove(T item) {
   Node pred, curr;
   int key = item.hashCode();
   head.lock();
   try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            pred.next = cur.next;
            return true;
         }
         return false;
      } finally {
         curr.unlock();
      }
   } finally {
      pred.unlock();
   }
}
```

Διασχίζεται η λίστα με hand-over-hand locking

Fine-grain synchronization: remove

```
public boolean remove(T item) {
   Node pred, curr;
   int key = item.hashCode();
   head.lock();
   try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            pred.next = cur.next;
            return true;
         }
         return false;
      } finally {
         curr.unlock();
      }
   } finally {
      pred.unlock();
   }
}
```

Διαγραφή στοιχείου

ΣΗΜΕΙΩΣΗ: Στη συγκεκριμένη περίπτωση η φυσική διαγραφή του κόμβου πραγματοποιείται από το μηχανισμό garbage collection της γλώσσας

Fine-grain synchronization: remove

```
public boolean remove(T item) {
   Node pred, curr;
   int key = item.hashCode();
   head.lock();
   try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            pred.next = cur.next;
            return true;
         }
         return false;
      } finally {
         curr.unlock();
      }
   } finally {
      pred.unlock();
   }
}
```

Απελευθέρωση κλειδωμάτων

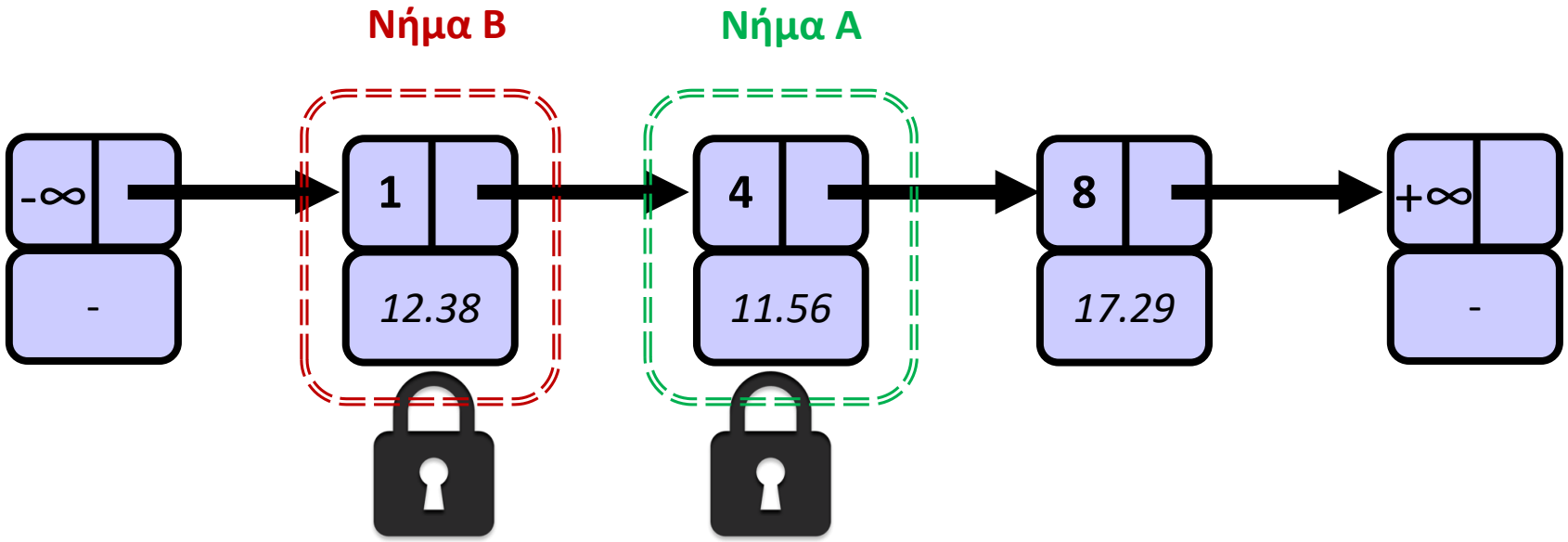
Fine-grain synchronization: remove

```
public boolean remove(T item) {
   Node pred, curr;
   int key = item.hashCode();
   head.lock();
   try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            pred.next = cur.next;
            return true;
         }
         return false;
      } finally {
         curr.unlock();
      }
   } finally {
      pred.unlock();
   }
}
```

Παρόμοια λογική για τις μεθόδους add και contains

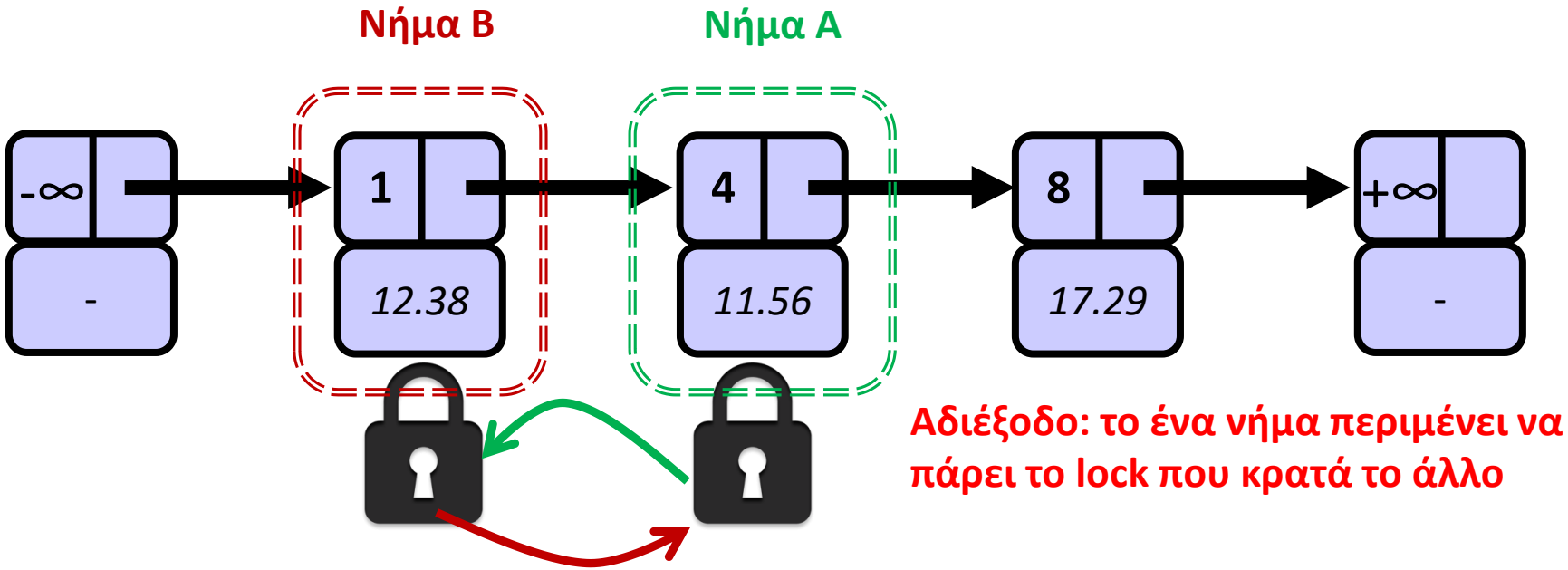
Fine-grain synchronization

- Όλες οι μέθοδοι πρέπει να καταλαμβάνουν τα locks με την ίδια σειρά για να αποφευχθεί αδιέξοδο
- Έστω ότι το νήμα Α θέλει να προσθέσει το **2** και το νήμα Β να διαγράψει το **4** και το νήμα Α καταλαμβάνει locks από δεξιά προς τα αριστερά ενώ το νήμα Β από δεξιά προς τα αριστερά



Fine-grain synchronization

- Όλες οι μέθοδοι πρέπει να καταλαμβάνουν τα locks με την ίδια σειρά για να αποφευχθεί αδιέξοδο
- Έστω ότι το νήμα Α θέλει να προσθέσει το **2** και το νήμα Β να διαγράψει το **4** και το νήμα Α καταλαμβάνει locks από δεξιά προς τα αριστερά ενώ το νήμα Β από δεξιά προς τα αριστερά



Optimistic synchronization

- Ο fine-grain συγχρονισμός αποτελεί βελτίωση σε σχέση με τον coarse-grain
  - Παρόλα αυτά περιλαμβάνει μια μεγάλη σειρά από λήψεις και απελευθερώσεις κλειδωμάτων
  - Νήματα που προσπελαύνουν διακριτά μέρη της λίστας μπορεί να αδυνατούν να εργαστούν ταυτόχρονα (π.χ. ένα νήμα που προσπελαύνει ένα στοιχείο στην αρχή της λίστας, εμποδίζει άλλα στοιχεία που θέλουν να προσπελάσουν τη μέση ή το τέλος της λίστας)
- Ας σκεφτούμε «αισιόδοξα», δηλαδή τα πράγματα έχουν λίγες πιθανότητες να πάνε στραβά:
  - **Εισαγωγή/διαγραφή:** αναζήτηση του στοιχείου χωρίς κλείδωμα
  - Κλείδωμα των κατάλληλων κόμβων
  - Έλεγχος αν η δομή παραμένει συνεπής
    - Αν όχι απελευθέρωση των κλειδωμάτων και **νέα προσπάθεια**
  - **Αναζήτηση:** (μέθοδος contains) χρησιμοποιεί κλείδωμα

Optimistic synchronization: validate

- Ελέγχει αν η δομή είναι συνεπής, δηλαδή:
  - Αν τα δύο στοιχεία (*pred, curr*) βρίσκονται ακόμα στη δομή και αν εξακολουθούν να είναι διαδοχικά
    - Αν ο προηγούμενος κόμβος (*pred*) είναι προσβάσιμος από την αρχή της δομής, **και**
    - Αν ο επόμενος του προηγούμενου είναι ο τρέχοντας (*pred.next == curr*)

Optimistic synchronization: validate

```
private boolean
  validate(Node pred, Node curr) {
   Node node = head;
   while (node.key <= pred.key) {
      if (node == pred)
         return pred.next == curr;
      node = node.next;
   }
   return false;
}
```

Optimistic synchronization: validate

```
private boolean
  validate(Node pred, Node curr) {
   Node node = head;
   while (node.key <= pred.key) {
      if (node == pred)
         return pred.next == curr;
      node = node.next;
   }
   return false;
}
```

Πρόσβαση στον προηγούμενο κόμβο από την αρχή της δομής

Optimistic synchronization: validate

```
private boolean
  validate(Node pred, Node curr) {
   Node node = head;
   while (node.key <= pred.key) {
      if (node == pred)
         return pred.next == curr;
      node = node.next;
   }
   return false;
}
```

Έλεγχος αν το προηγούμενο δείχνει στο τρέχον

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Επαναληπτική προσπάθεια μέχρι να επιτύχει

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Διάσχιση της λίστας χωρίς κλειδώματα

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Κλειδώνει τον προηγούμενο και τον τρέχοντα

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Αν η δομή είναι συνεπής

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Αν η δομή είναι συνεπής, και υπάρχει το στοιχείο το αφαιρεί και επιστρέφει

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Αν η δομή είναι συνεπής, και δεν υπάρχει το στοιχείο, επιστρέφει

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Αν η δομή ΔΕΝ είναι συνεπής, ξεκλειδώνει

Optimistic synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Αν η δομή ΔΕΝ είναι συνεπής, ξεκλειδώνει, και ξαναπροσπαθεί

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Optimistic synchronization: remove

Ανάλογα λειτουργεί η add

Η contains χρησιμοποιεί κλειδώματα (γιατί;)

ΔΕΝ ΧΡΕΙΑΖΕΤΑΙ!!!

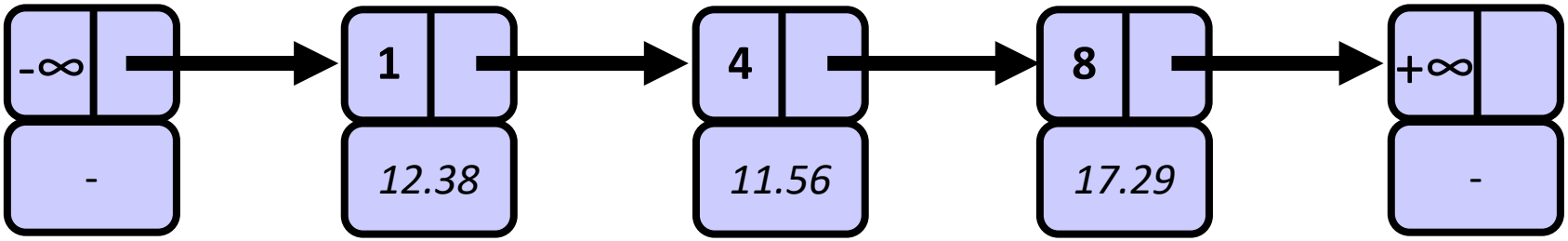
Lazy synchronization

- Ο optimistic συγχρονισμός λειτουργεί καλύτερα από το fine-grain συγχρονισμό αν:
  - το κόστος να διατρέξουμε τη δομή 2 φορές χωρίς κλειδώματα είναι χαμηλότερο από το να την διατρέξουμε μία φορά με κλειδώματα
- **Μειονέκτημα:** η μέθοδος contains χρησιμοποιεί κλειδώματα
  - Θέλουμε να το αποφύγουμε δεδομένου ότι αναμένουμε η contains να καλείται με πολύ μεγάλη συχνότητα
- Επόμενο βήμα βελτιστοποίησης:
  - **Η validate να μη διατρέχει τη λίστα από την αρχή**
  - Η contains να μη χρησιμοποιεί καθόλου κλειδώματα (wait-free)

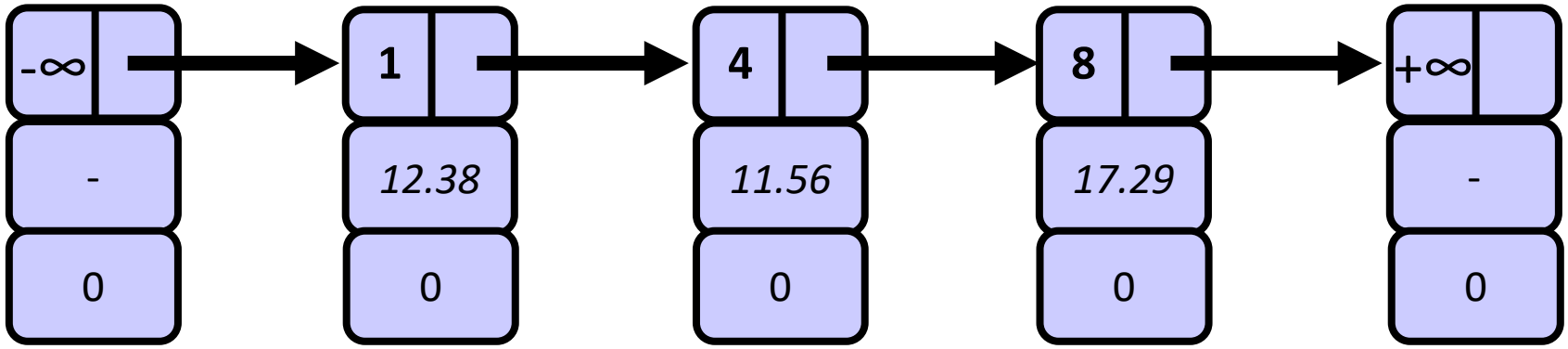
Lazy synchronization

- Lazy synchronization:
  - Προσθέτουμε στη δομή μία boolean μεταβλητή που δείχνει αν ο κόμβος βρίσκεται στη λίστα ή έχει διαγραφεί
  - Η contains διατρέχει τη λίστα χωρίς να κλειδώνει. Ελέγχει και την επιπλέον Boolean μεταβλητή
  - Η add διατρέχει τη λίστα, κλειδώνει και ελέγχει τη συνέπεια της δομής καλώντας τη validate
  - Η validate δεν διατρέχει τη λίστα από την αρχή αλλά κάνει τοπικούς ελέγχους στους κόμβους pred και curr
  - Η remove είναι **lazy**, δηλαδή λειτουργεί σε 2 βήματα:
    - Το 1ο βήμα αφαιρεί **λογικά** τον κόμβο (θέτοντας τη boolean μεταβλητή σε false)
    - Το 2ο βήμα αφαιρεί **φυσικά** τον κόμβο επαναθέτοντας του δείκτες

Lazy synchronization

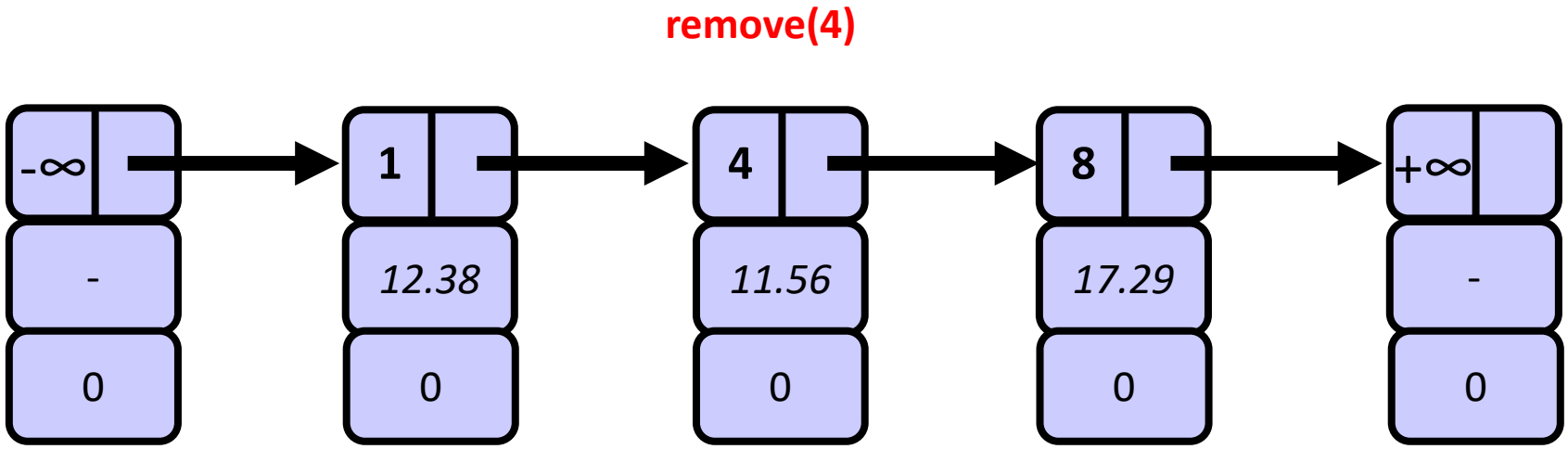


Lazy synchronization

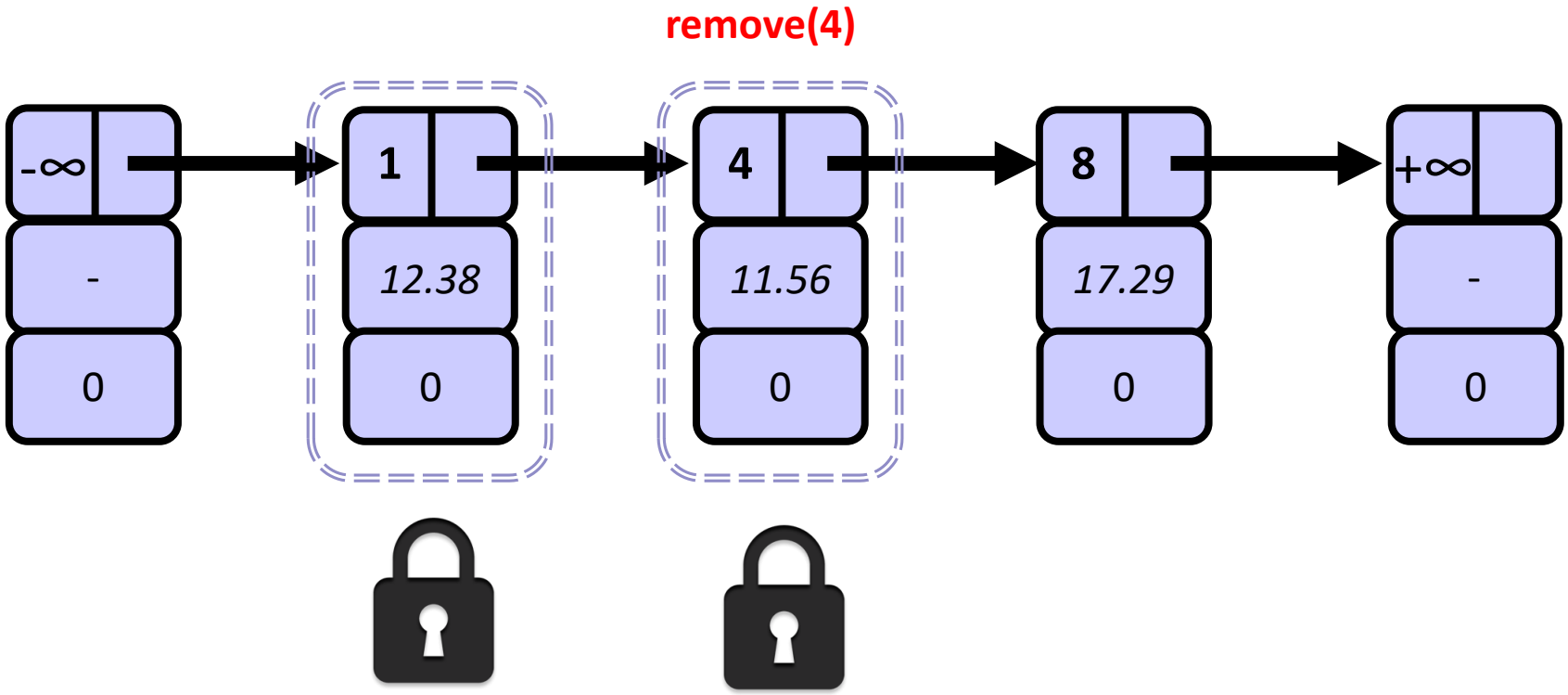


Boolean μεταβλητή που δείχνει αν ο κόμβος είναι στη δομή

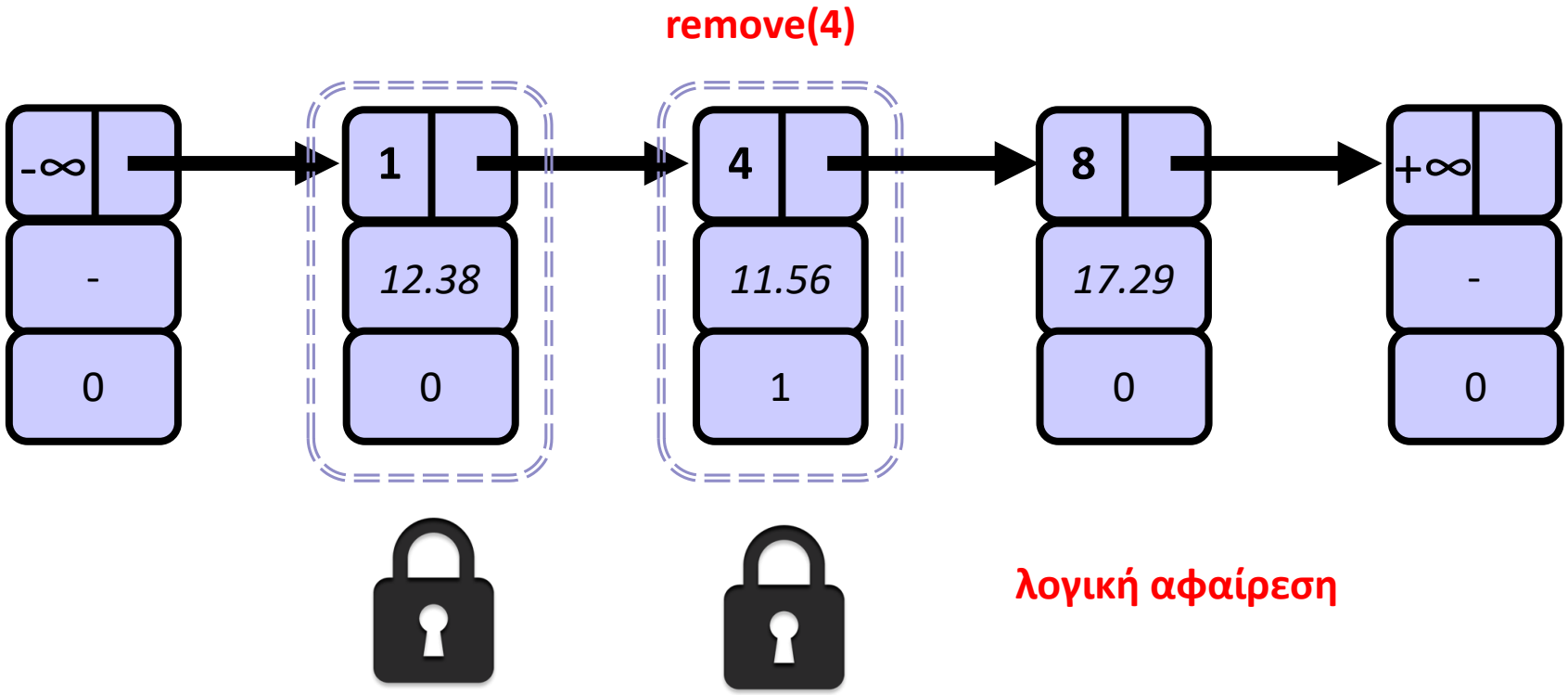
Lazy synchronization: remove



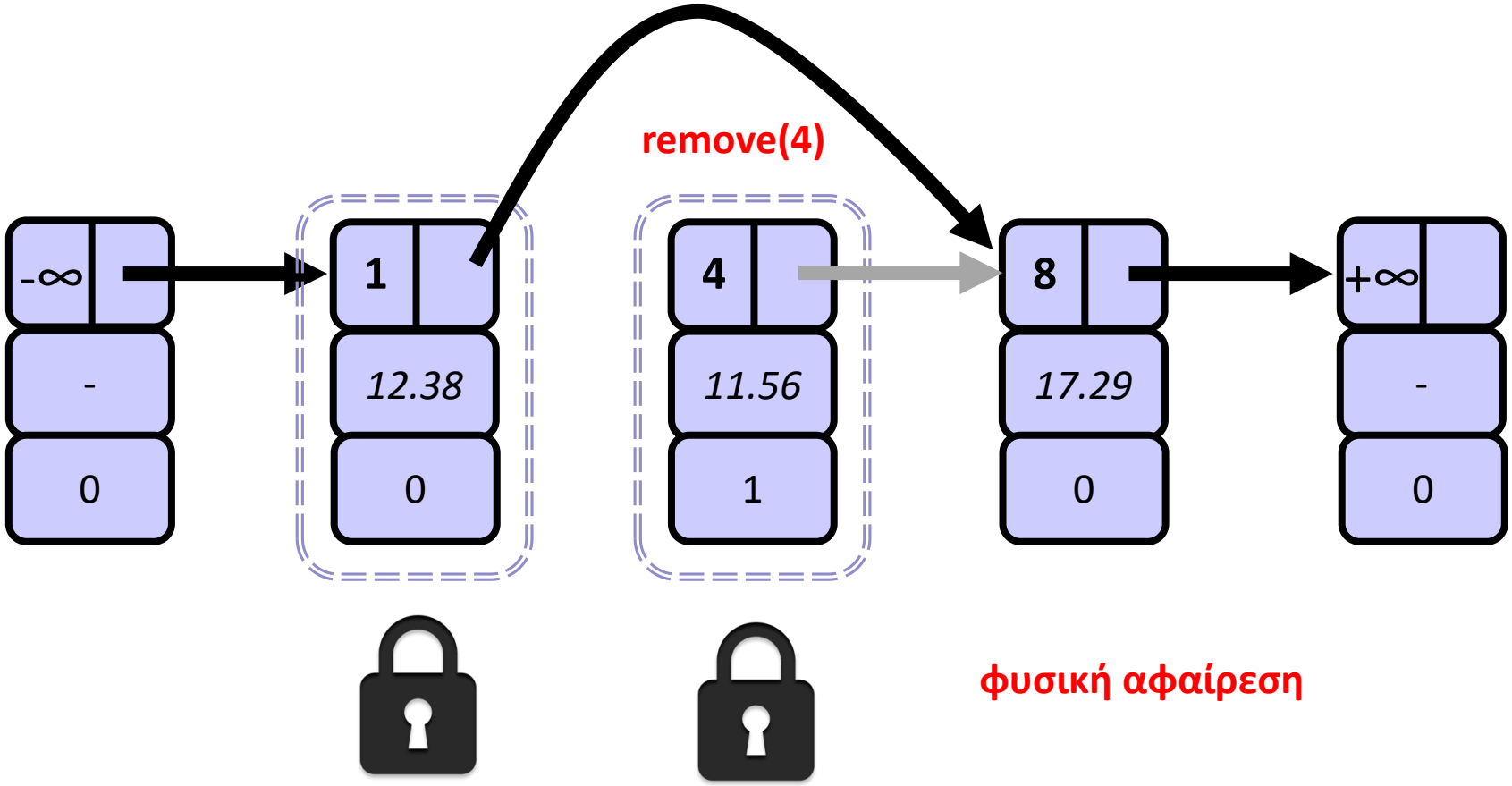
Lazy synchronization: remove



Lazy synchronization: remove



Lazy synchronization: remove



Lazy synchronization: validate

```
private boolean
  validate(Node pred, Node curr) {
   return
    !pred.marked &&
    !curr.marked &&
    pred.next == curr;
}
```

Lazy synchronization: validate

```
private boolean
  validate(Node pred, Node curr) {
   return
    !pred.marked &&
    !curr.marked &&
    pred.next == curr;
}
```

Δεν έχει σβηστεί ο προηγούμενος

Lazy synchronization: validate

```
private boolean
  validate(Node pred, Node curr) {
   return
    !pred.marked &&
    !curr.marked &&
    pred.next == curr;
}
```

Δεν έχει σβηστεί ο τρέχων

Lazy synchronization: validate

```
private boolean
  validate(Node pred, Node curr) {
   return
    !pred.marked &&
    !curr.marked &&
    pred.next == curr;
}
```

Ο προηγούμενος δείχνει τον τρέχοντα

Lazy synchronization: contains

```
public boolean contains(T item) {
   int key = item.hashCode();
   Node curr = this.head;
   while (curr.key < key) {
      curr = curr.next;
   }
   return
      curr.key == key &&
      !curr.marked;
}
```

Δεν χρησιμοποιεί κανένα κλείδωμα

Lazy synchronization: contains

```
public boolean contains(T item) {
   int key = item.hashCode();
   Node curr = this.head;
   while (curr.key < key) {
      curr = curr.next;
   }
   return
      curr.key == key &&
      !curr.marked;
}
```

Ελέγχει αν ο κόμβος έχει σβηστεί λογικά

Lazy synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.key == key) {
               curr.marked = true;
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Lazy synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.key == key) {
               curr.marked = true;
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Αναζήτηση χωρίς κλείδωμα

Lazy synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.key == key) {
               curr.marked = true;
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Τοπικό κλείδωμα

Lazy synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.key == key) {
               curr.marked = true;
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Τοπικό κλείδωμα και έλεγχος συνέπειας

Lazy synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.key == key) {
               curr.marked = true;
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Διαγραφή σε 2 βήματα:

Lazy synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.key == key) {
               curr.marked = true;
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Διαγραφή σε 2 βήματα:

1ο Βήμα: Λογική διαγραφή

Lazy synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.key == key) {
               curr.marked = true;
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Διαγραφή σε 2 βήματα:

1ο Βήμα: Λογική διαγραφή

2ο Βήμα: Φυσική διαγραφή

Lazy synchronization: remove

```
public boolean remove(Item item) {
   int key = item.hashCode();
   while (true) {
      Node pred = this.head;
      Node curr = pred.next;
      while (curr.key <= key) {
         if (item == curr.item)
            break;
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred,curr) {
            if (curr.key == key) {
               curr.marked = true;
               pred.next = curr.next;
               return true;
            } else {
               return false;
            }
         }
      } finally {
         pred.unlock(); curr.unlock();
      }
   }
}
```

Ομοίως η Add

Non-blocking synchronization

- Τελευταίο βήμα: Να απαλείψουμε πλήρως τα κλειδώματα και από τις 3 μεθόδους της δομής, add(), remove και contains().
- Γιατί μας ενδιαφέρει να απαλείψουμε πλήρως τα κλειδώματα;
  - **Επίδοση:** Υπό συνθήκες μπορεί να είναι ταχύτερο
  - **Ανθεκτικότητα (robustness):** Αποφεύγουμε καταστάσεις κατά τις οποίες η αποτυχία ενός νήματος που κρατάει ένα κλείδωμα θα οδηγήσει σε αποτυχία όλης της εφαρμογής
- Ιδέα:
  - Να χειριστούμε τα πεδία marked και next της δομής σαν μία ατομική μονάδα
  - Οποιαδήποτε προσπάθεια ενημέρωσης του πεδίου next όταν το marked είναι true θα αποτύχει

Non-blocking synchronization: Ζητήματα υλοποίησης

address

marked field

Η δομή ως τώρα

Non-blocking synchronization: Ζητήματα υλοποίησης

Συμπυκνώνονται τα δύο κρίσιμα πεδία της δομής σε μία packed μορφή την οποία ο επεξεργαστής μπορεί να χειριστεί ατομικά

Π.χ.

AtomicMarkableReference class

Java.util.concurrent.atomic package

AtomicMarkableReference

```
public T get(boolean[] marked);

public boolean compareAndSet(
   T expectedRef,
   T updateRef,
   boolean expectedMark,
   boolean updateMark);

public boolean attemptMark(
   T expectedRef,
   boolean updateMark);
```

AtomicMarkableReference

```
public T get(boolean[] marked);
```

Επιστρέφει τη διεύθυνση του αντικειμένου T και αποθηκεύει το mark στη θέση 0 του πίνακα που περνιέται σαν παράμετρος

```
  T updateRef,
  boolean expectedMark,
  boolean updateMark);

public boolean attemptMark(
  T expectedRef,
  boolean updateMark);
```

AtomicMarkableReference

```
public T get(boolean[] marked);

public boolean compareAndSet(
   T expectedRef,
   T updateRef,
   boolean expectedMark,
   boolean updateMark);
```

Ελέγχει και αν επιτύχει ενημερώνει ΚΑΙ τα δύο πεδία

```
   T expectedRef,
   boolean updateMark);
```

AtomicMarkableReference

```
public T get(boolean[] marked);

public boolean compareAndSet(
   T expectedRef,
   T updateRef,
   boolean expectedMark,
   boolean updateMark);

public boolean attemptMark(
   T expectedRef,
   boolean updateMark);
```

Ενημέρωνει το mark αν η διεύθυνση είναι η αναμενόμενη

Non-blocking synchronization

- Τελευταίο βήμα: Να απαλείψουμε πλήρως τα κλειδώματα και από τις 3 μεθόδους της δομής, add(), remove και contains().
- Ιδέα:
  - Να χειριστούμε τα πεδία marked και next της δομής σαν μία ατομική μονάδα
  - Οποιαδήποτε προσπάθεια ενημέρωσης του πεδίου next όταν το marked είναι true (ο κόμβος δεν υπάρχει) θα αποτύχει
  - Η αφαίρεση ενός κλειδιού περιλαμβάνει την ενημέρωση του πεδίου marked (και μία μόνο απόπειρα να ενημερωθούν οι διευθύνσεις)
  - Κάθε νήμα που διατρέχει τη λίστα εκτελώντας την add() και remove() αφαιρεί φυσικά τους κόμβους που συναντά να έχουν σβηστεί
  - Η αποτυχία για ατομική ενημέρωση οδηγεί στην επαναπροσπάθεια με διάτρεξη από την αρχή της λίστας

Non-blocking synchronization: βοηθητική κλάση Window και μέθοδος find()

- Χρησιμοποιείται από την add() και τη remove() για να βρεθεί το σημείο που θα γίνει η εισαγωγή ή το στοιχείο που θα αφαιρεθεί
- Επιστρέφει ένα δείκτη στο ακριβώς μικρότερο στοιχείο (από αυτό που θα προστεθεί ή αφαιρεθεί) και ένα δείκτη στο αμέσως μεγαλύτερο ή ίσο στοιχείο
- Αν συναντήσει στοιχεία που έχουν σβηστεί λογικά τα σβήνει και φυσικά

Non-blocking synchronization: find()

```
public Window find(Node head, int key) {
  Node pred = null, curr = null succ = null;
  boolean[] marked = {false};
  retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true)
      succ = curr.next.get(marked);
      while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ,
   false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
      }
      if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
  }
}
```

Non-blocking synchronization: find()

```
public Window find(Node head, int key) {
  Node pred = null, curr = null succ = null;
  boolean[] marked = {false};
  retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ,
   false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
      }
      if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
  }
}
```

Διατρέχει τη λίστα μέχρι να βρει ένα στοιχείο μεγαλύτερο από το ζητούμενο

Non-blocking synchronization: find()

```
public Window find(Node head, int key) {
  Node pred = null, curr = null succ = null;
  boolean[] marked = {false};
  retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true)
      succ = curr.next.get(marked);
      while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ,
   false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
      }
      if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
  }
}
```

Αν βρεθεί στοιχείο που έχει σβηστεί λογικά, σβήνεται και φυσικά

Non-blocking synchronization: find()

```
public Window find(Node head, int key) {
  Node pred = null, curr = null succ = null;
  boolean[] marked = {false};
  retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true)
      succ = curr.next.get(marked);
      while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ,
   false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
      }
      if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
  }
}
```

Αν αποτύχει η ατομική ενημέρωση η διαδικασία ξεκινά από την αρχή

Non-blocking synchronization: remove()

```
public boolean remove(T item) {
  boolean snip;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key != key) {
       return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.compareAndSet(succ, succ,
   false, true)
      if (!snip) continue;
      pred.next.compareAndSet(curr, succ, false,
   false);
    return true;
    }
  }
}
```

Non-blocking synchronization: remove()

```
public boolean remove(T item) {
  boolean snip;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key != key) {
       return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.compareAndSet(succ, succ,
   false, true)
      if (!snip) continue;
      pred.next.compareAndSet(curr, succ, false,
   false);
    return true;
    }
  }
}
```

Αφαιρεί λογικά τον κόμβο

Non-blocking synchronization: remove()

```
public boolean remove(T item) {
  boolean snip;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key != key) {
       return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.compareAndSet(succ, succ,
   false, true)
      if (!snip) continue;
      pred.next.compareAndSet(curr, succ, false,
   false);
    return true;
    }
  }
}
```

Αφαιρεί λογικά τον κόμβο. Αν αποτύχει (γιατί ο κόμβος έχει εντωμεταξύ αφαιρεθεί) ξαναξεκινά

Non-blocking synchronization: remove()

```
public boolean remove(T item) {
  boolean snip;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key != key) {
       return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.compareAndSet(succ, succ,
   false, true)
      if (!snip) continue;
      pred.next.compareAndSet(curr, succ, false,
   false);
    return true;
    }
  }
}
```

Κάνει μία απόπειρα να αφαιρέσει και φυσικά τον κόμβο

Non-blocking synchronization: add()

```
public boolean add(T item) {
  boolean splice;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = new AtomicMarkableRef(curr, false);
      if (pred.next.compareAndSet(curr, node, false,
   false)) return true;
    }
  }
}
```

Non-blocking synchronization: add()

```
public boolean add(T item) {
  boolean splice;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = new AtomicMarkableRef(curr, false);
      if (pred.next.compareAndSet(curr, node, false,
   false)) return true;
    }
  }
}
```

Δημιουργία και αρχικοποίηση νέου κόμβου

Non-blocking synchronization: add()

```
public boolean add(T item) {
  boolean splice;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = new AtomicMarkableRef(curr, false);
      if (pred.next.compareAndSet(curr, node, false,
   false)) return true;
    }
  }
}
```

Ο νέος κόμβος δείχνει στον current

Non-blocking synchronization: add()

```
public boolean add(T item) {
  boolean splice;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = new AtomicMarkableRef(curr, false);
      if (pred.next.compareAndSet(curr, node, false,
   false)) return true;
    }
  }
}
```

Διασύνδεση νέου κόμβου στη δομή (ο pred δείχνει στο νέο κόμβο)

Non-blocking synchronization: add()

```
public boolean add(T item) {
  boolean splice;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = new AtomicMarkableRef(curr, false);
      if (pred.next.compareAndSet(curr, node, false,
   false)) return true;
    }
  }
}
```

Αν αποτύχει, ξαναπροσπαθεί

Non-blocking synchronization: contains()

```
public boolean contains(Tt item) {
  boolean marked;
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key)
    curr = curr.next;
  Node succ = curr.next.get(marked);
  return (curr.key == key && !marked[0])
}
```

Wait-free μέθοδος

Ιδιότητες προόδου στα σχήματα Blocking και Non-blocking

	Blocking	**Non-blocking**
Κάποιος προοδεύει	*Χωρίς αδιέξοδο (deadlock-free)*	*Lock-free*
Όλοι προοδεύουν	*Χωρίς λιμοκτονία (starvation-free)*	*Wait-free*

Σύνοψη

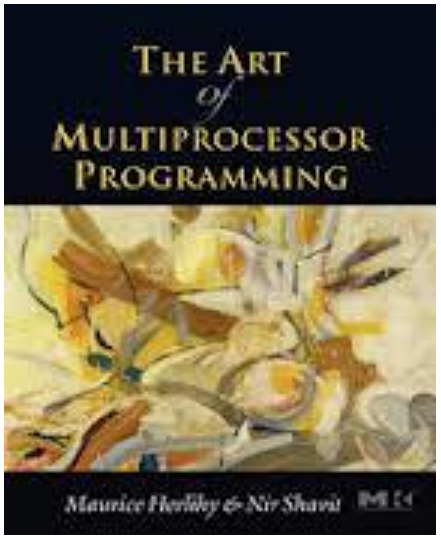
	contains	add	remove	validate
Coarse-grain	1 lock	1 lock	1 lock	no
Fine-grain	hand-over-hand locking	hand-over-hand locking	hand-over-hand locking	no
Optimistic	local locking	local locking	local locking	διάσχιση από την αρχή
Lazy	no locking	local locking	local locking	local checks
Non-blocking	wait-free	lock-free (with retries)	lock-free (with retries)	no

Further reading

Chapters 7, 8, 9 from:

The Art of Multiprocessor Programming

By Maurice Herlihy, Nir Shavit



The following joke circulated in Italy in the 1920s. According to Mussolini, the ideal citizen is intelligent, honest, and Fascist. Unfortunately, no one is perfect, which explains why everyone you meet is either intelligent and Fascist but not honest, honest and Fascist but not intelligent, or honest and intelligent but not Fascist.