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

Συγχρονισμός

Συστήματα Παράλληλης Επεξεργασίας
9ο Εξάμηνο

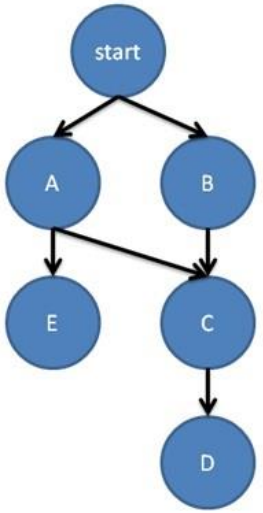
National Technical University of Athens
CSLab

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);
```

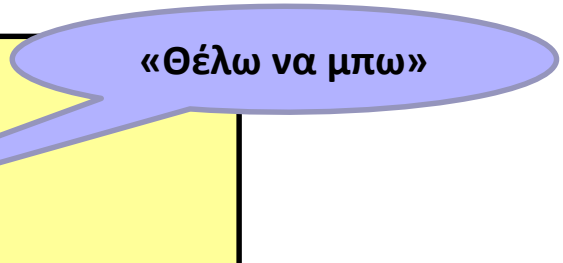
Πώς υλοποιείται ένα σωστό και «καλό» κλείδωμα;

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;
}
```

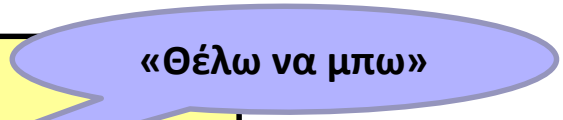
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα

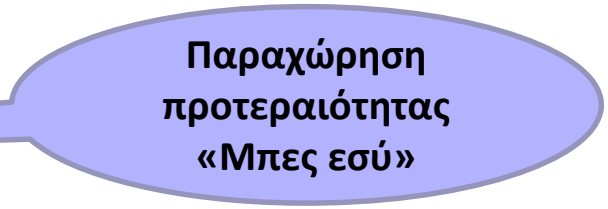
```
// i = εγώ
// j = το άλλο νήμα
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```



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

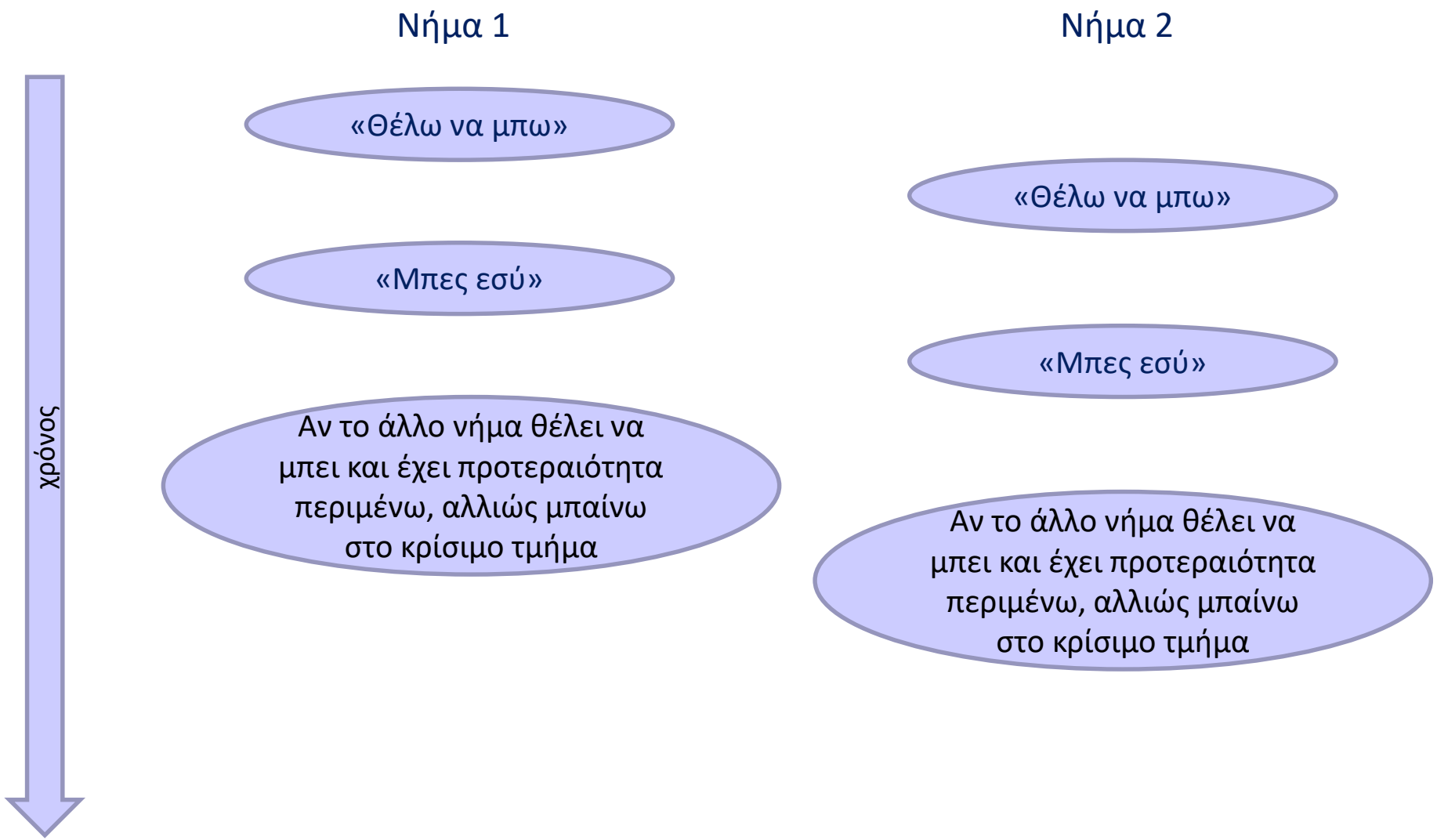
```
// i = εγώ
// j = το άλλο νήμα
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```

«Θέλω να μπω»

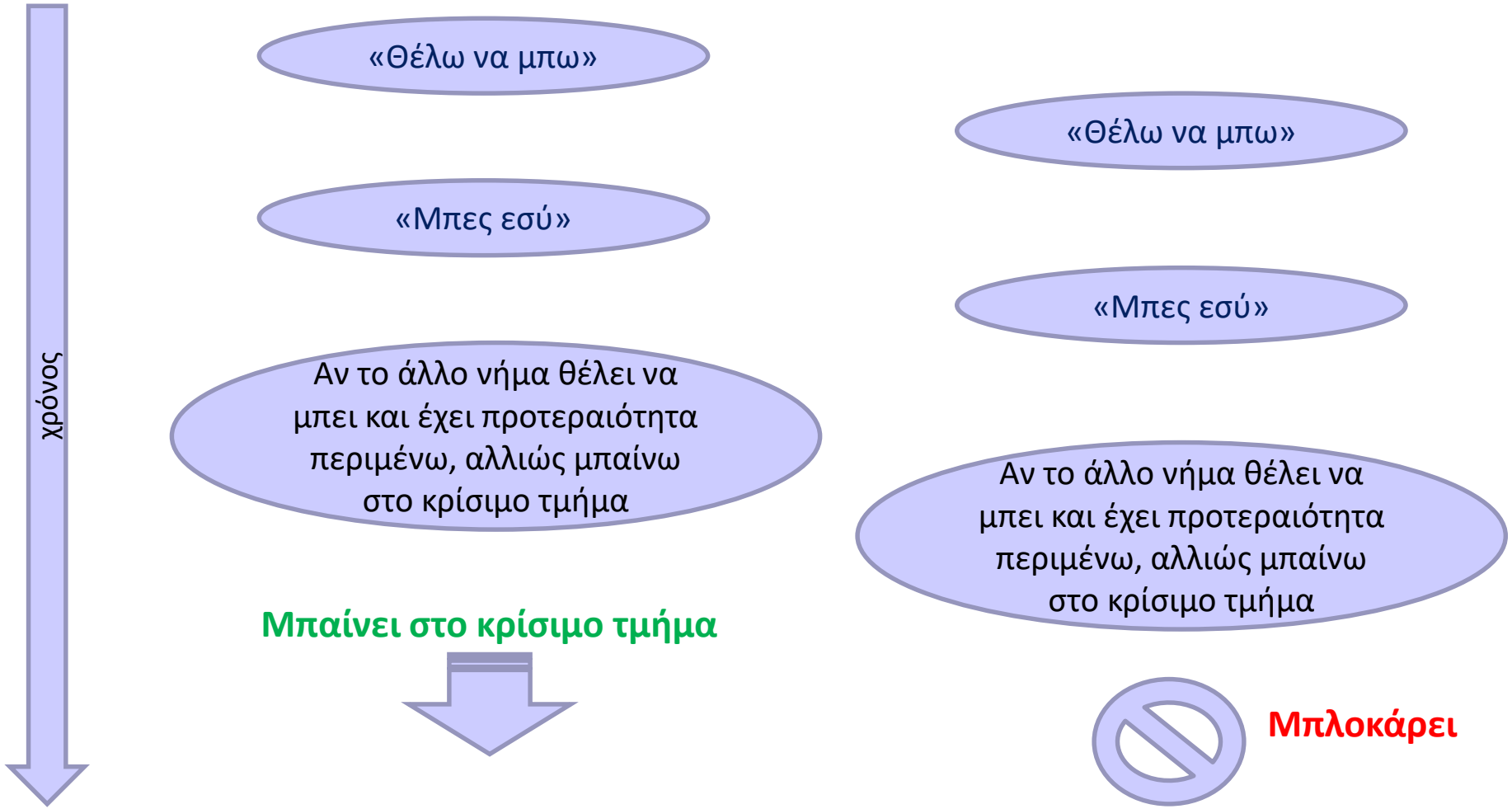
Παραχώρηση προτεραιότητας «Μπες εσύ»

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

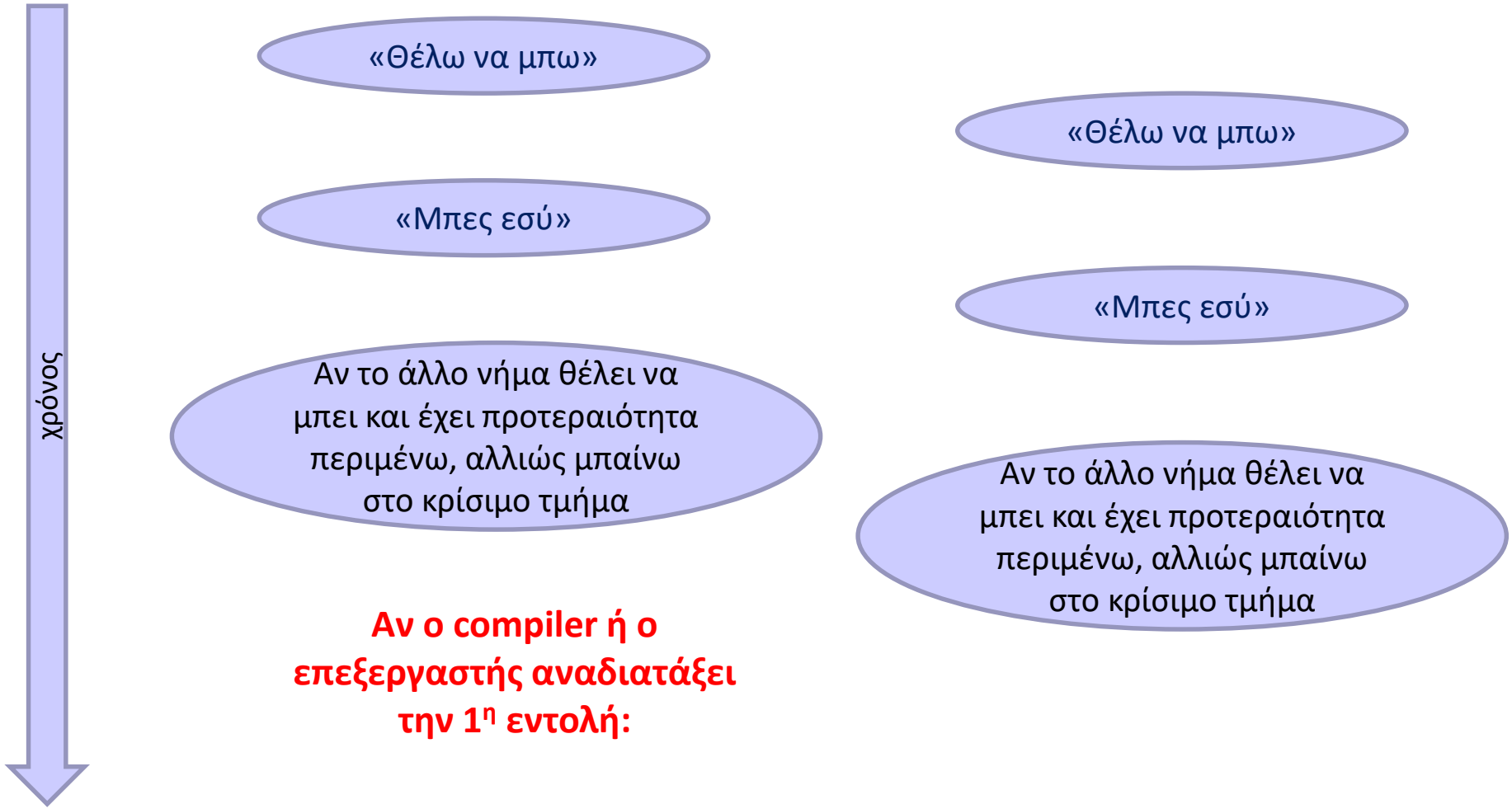
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



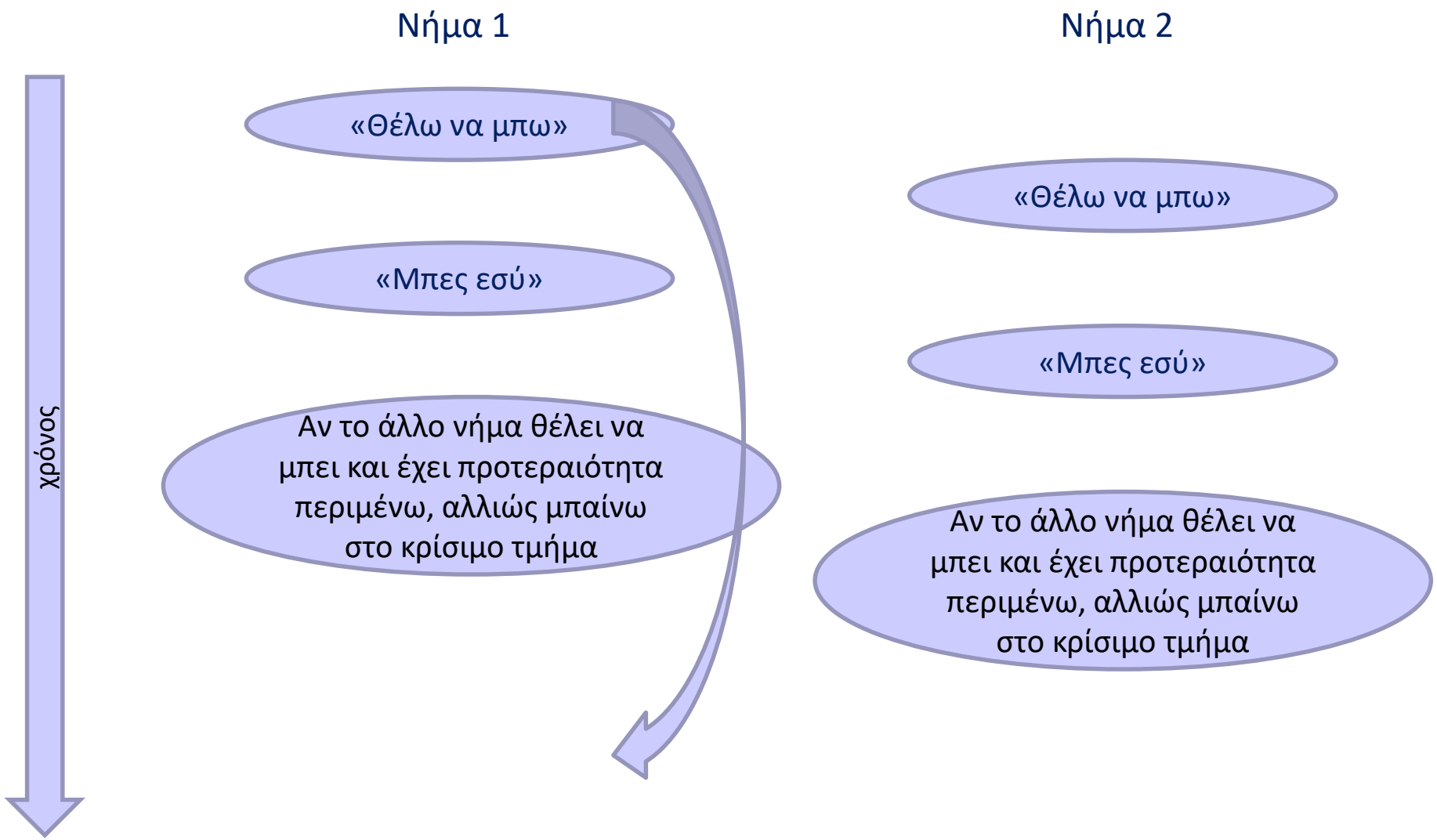
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



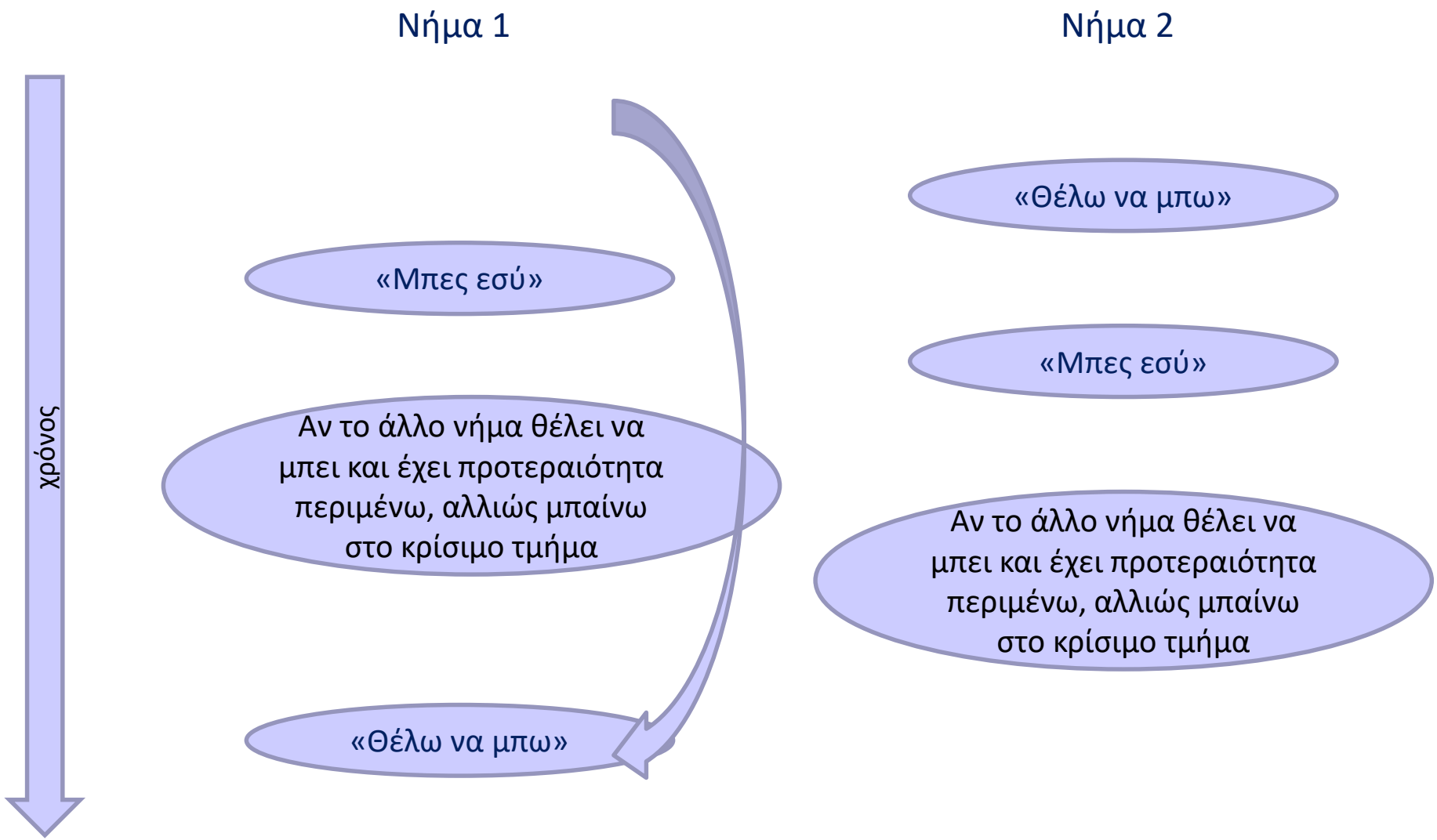
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



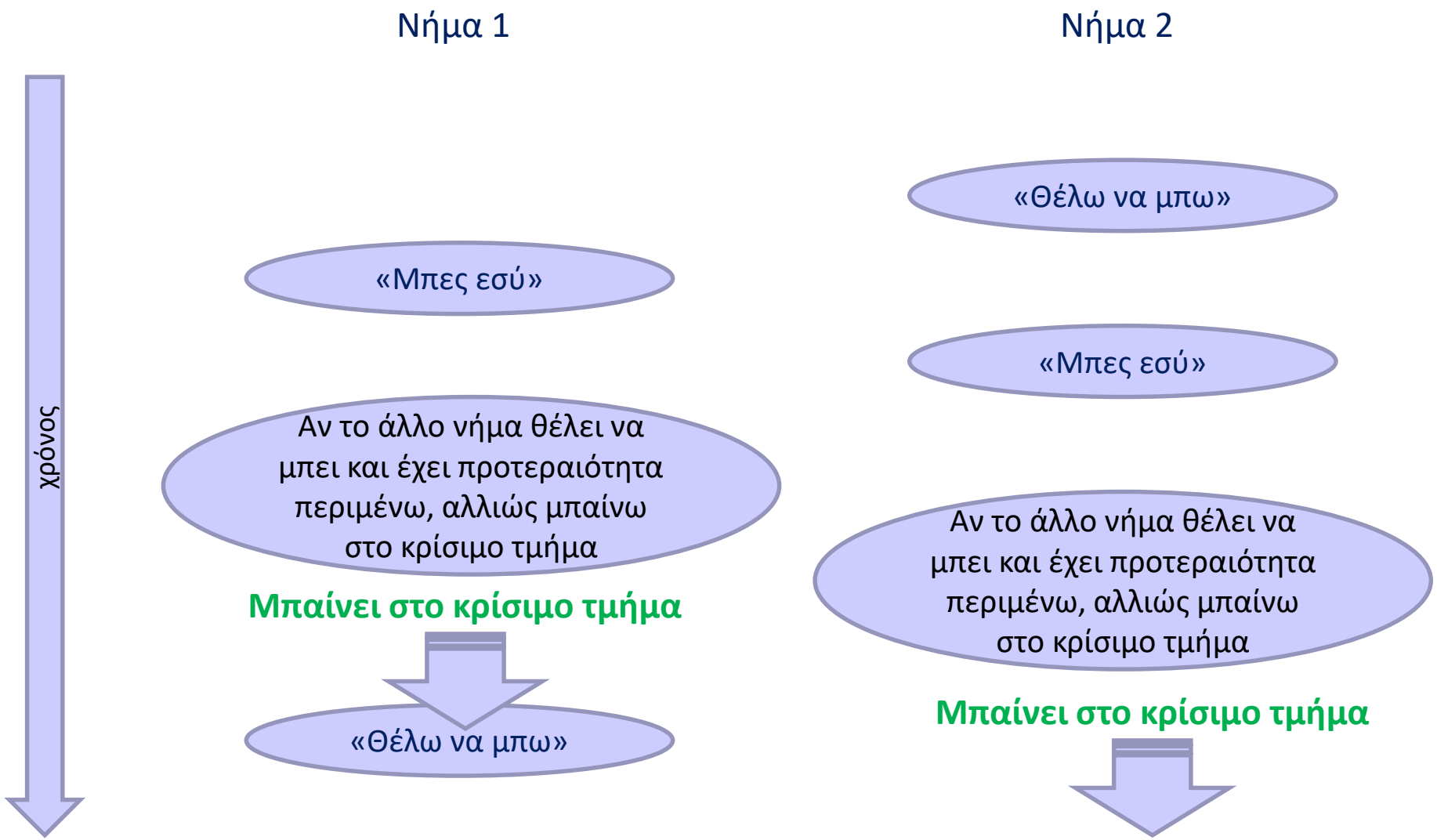
Λύση στο λογισμικό: Peterson's lock
δουλεύει για 2 νήματα



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;
 }
}
```

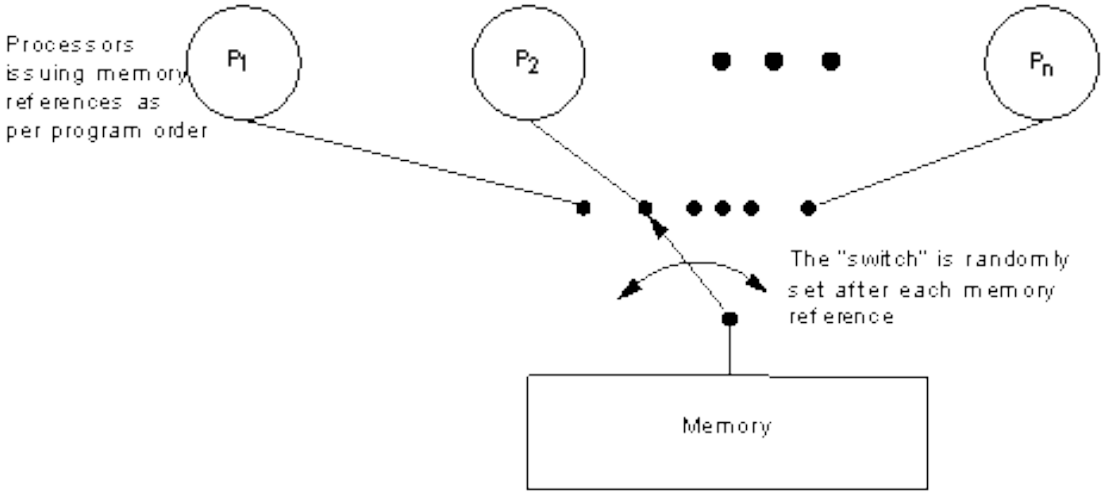
«Δεν θέλω να μπω»

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

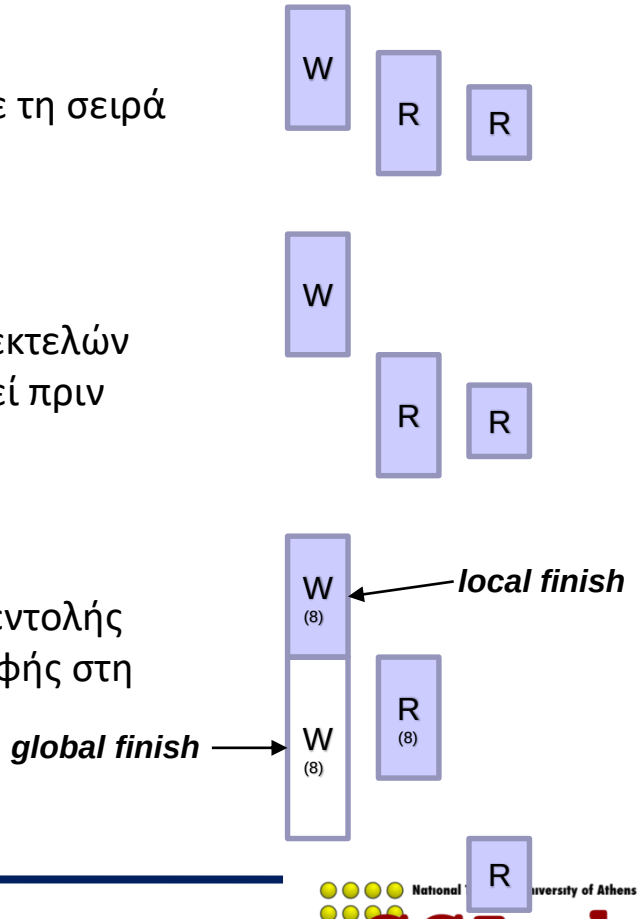
Ακολουθιακή συνέπεια (sequential consistency)

- *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).



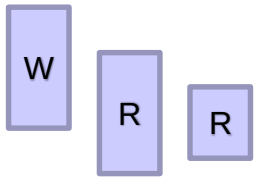
Ακολουθιακή συνέπεια (sequential consistency)

- *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

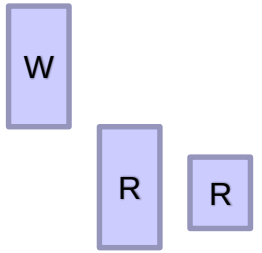


Ακολουθιακή συνέπεια (sequential consistency)

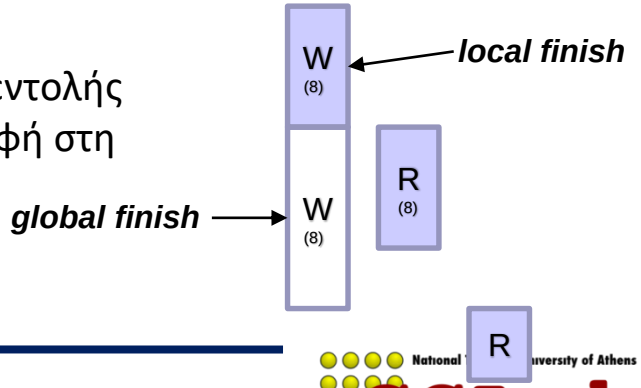
- *Ένας πολυεπεξεργαστής είναι ακολουθιακά συνεπής αν το αποτέλεσμα οποιασδήποτε εκτέλεσης είναι το ίδιο ως εάν οι εντολές όλων των επιμέρους επεξεργαστών εκτελούνταν σειριακά και οι εντολές του κάθε επιμέρους επεξεργαστή εκτελούνται με τη σειρά που ορίζεται στο πρόγραμμα* (Lamport, 1979).
- Ικανές συνθήκες: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

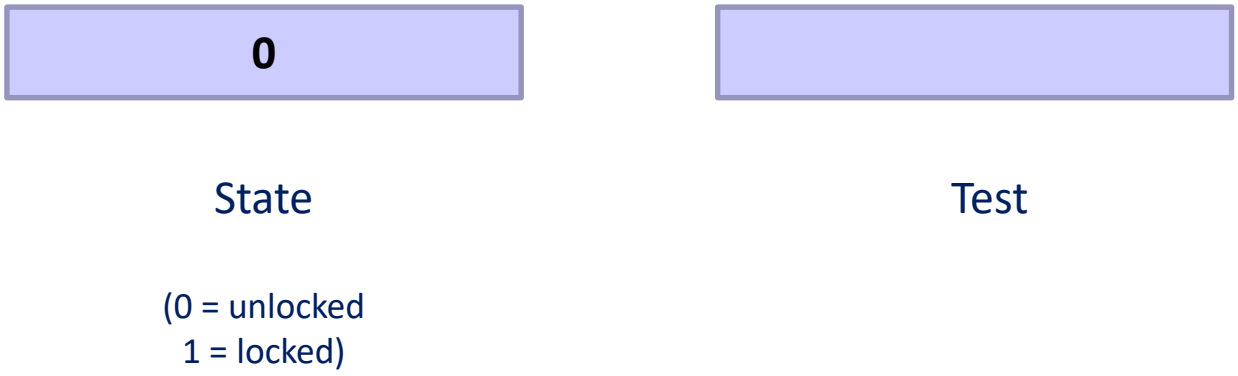
```
do {
   lock(mylock);
   /* critical section */
   code
   unlock(mylock);
   remainder section
} while (TRUE);
```

```
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) 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**

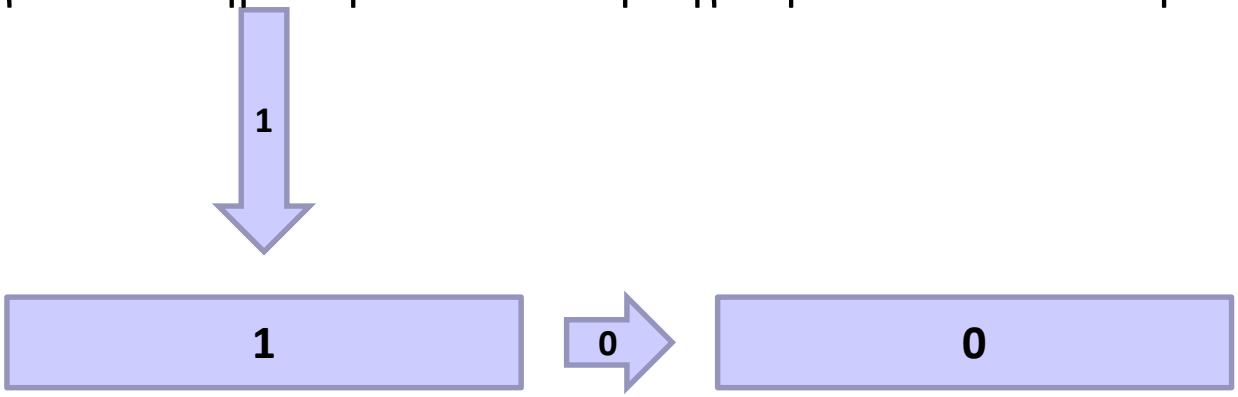
Υποστήριξη από το υλικό:
atomic operations, π.χ. test-and-set, compare-and-swap

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



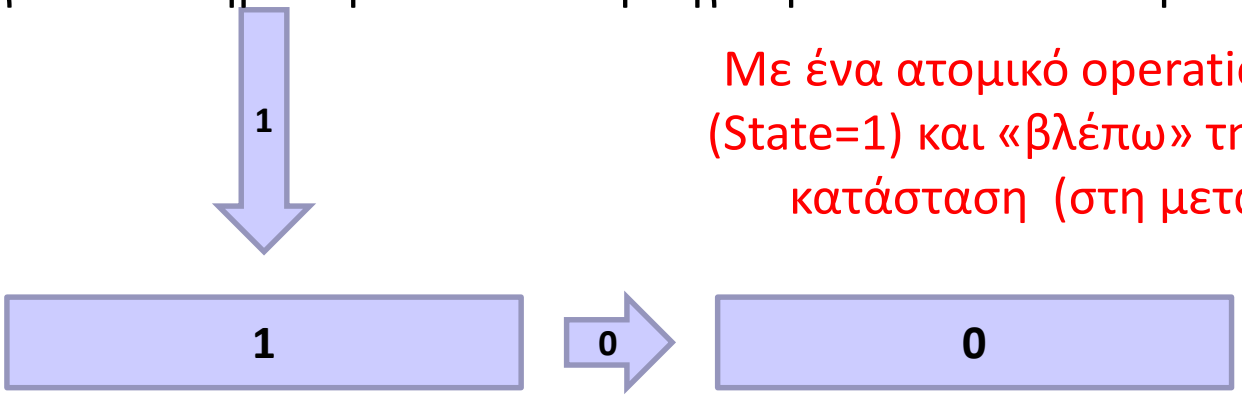
Υποστήριξη από το υλικό:
atomic operations, π.χ. test-and-set, compare-and-swap

- **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

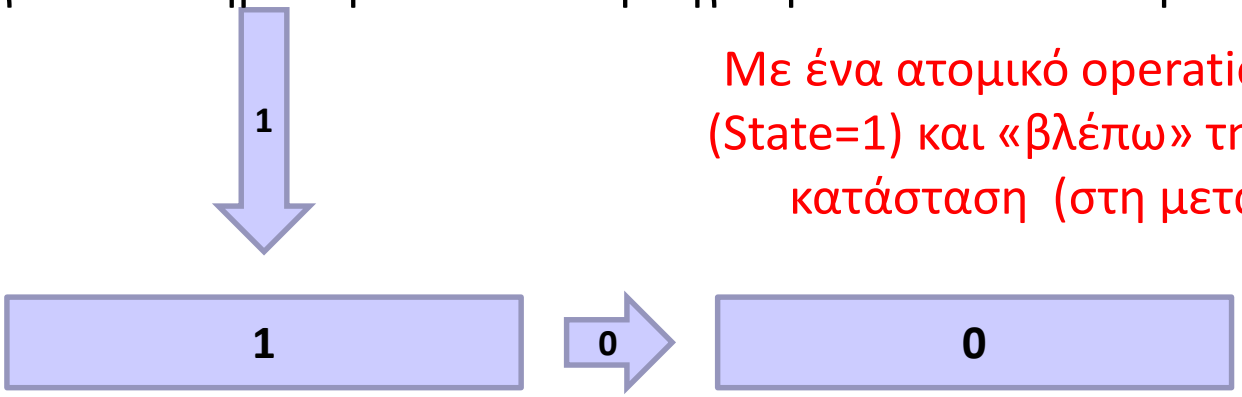
- **Ατομικά** ενεργούν σε **2** μεταβλητές
- Στις υλοποιήσεις τους τυπικά περιλαμβάνουν και **memory fence** (επιβολή για ολοκλήρωση όλων των προηγούμενων εντολών πρόσβασης στη μνήμη)



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)

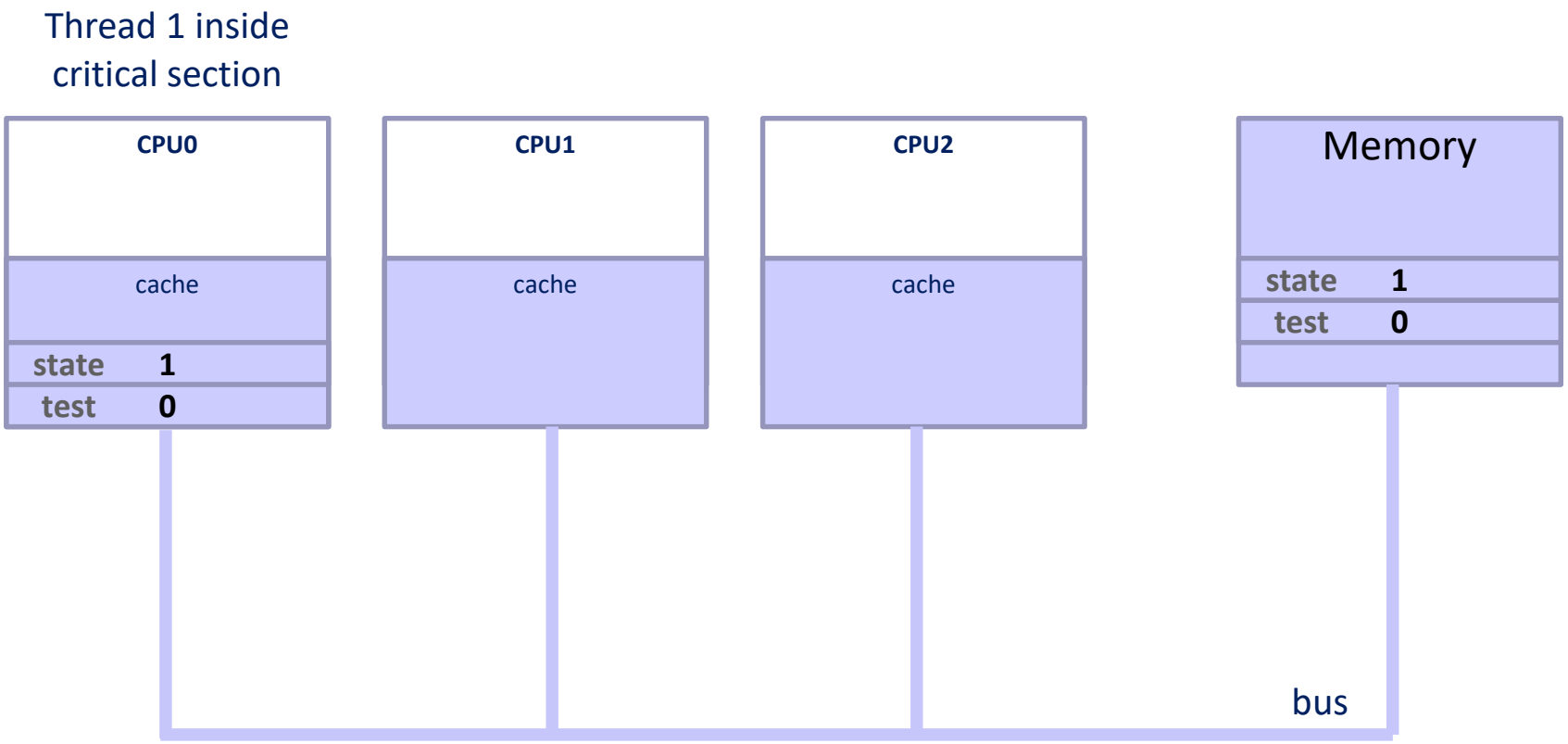
(0 = unlocked
1 = locked)

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

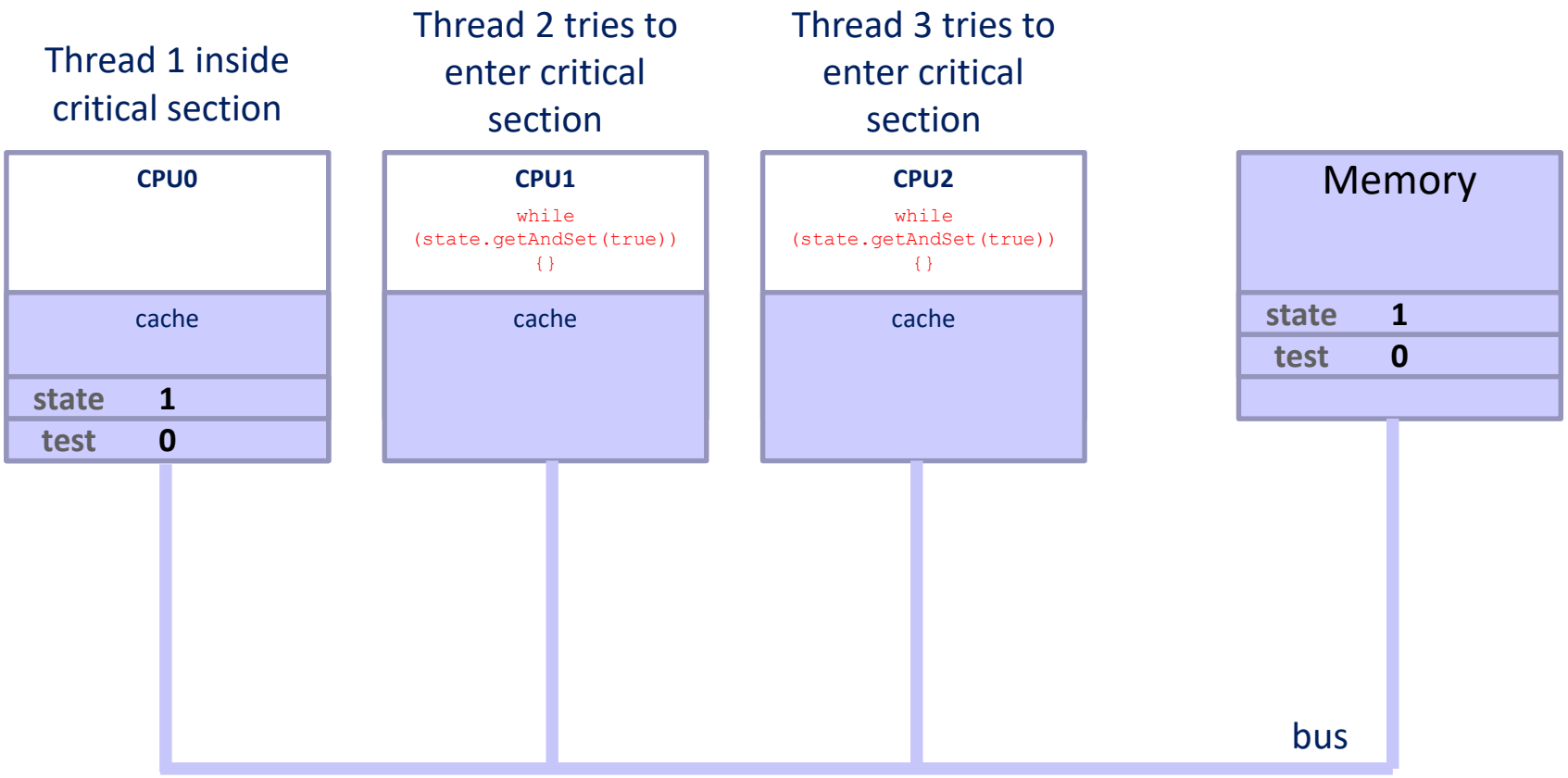
Test-and-set (TAS) lock

```
class TASlock {

 AtomicBoolean state = new AtomicBoolean(false);

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

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

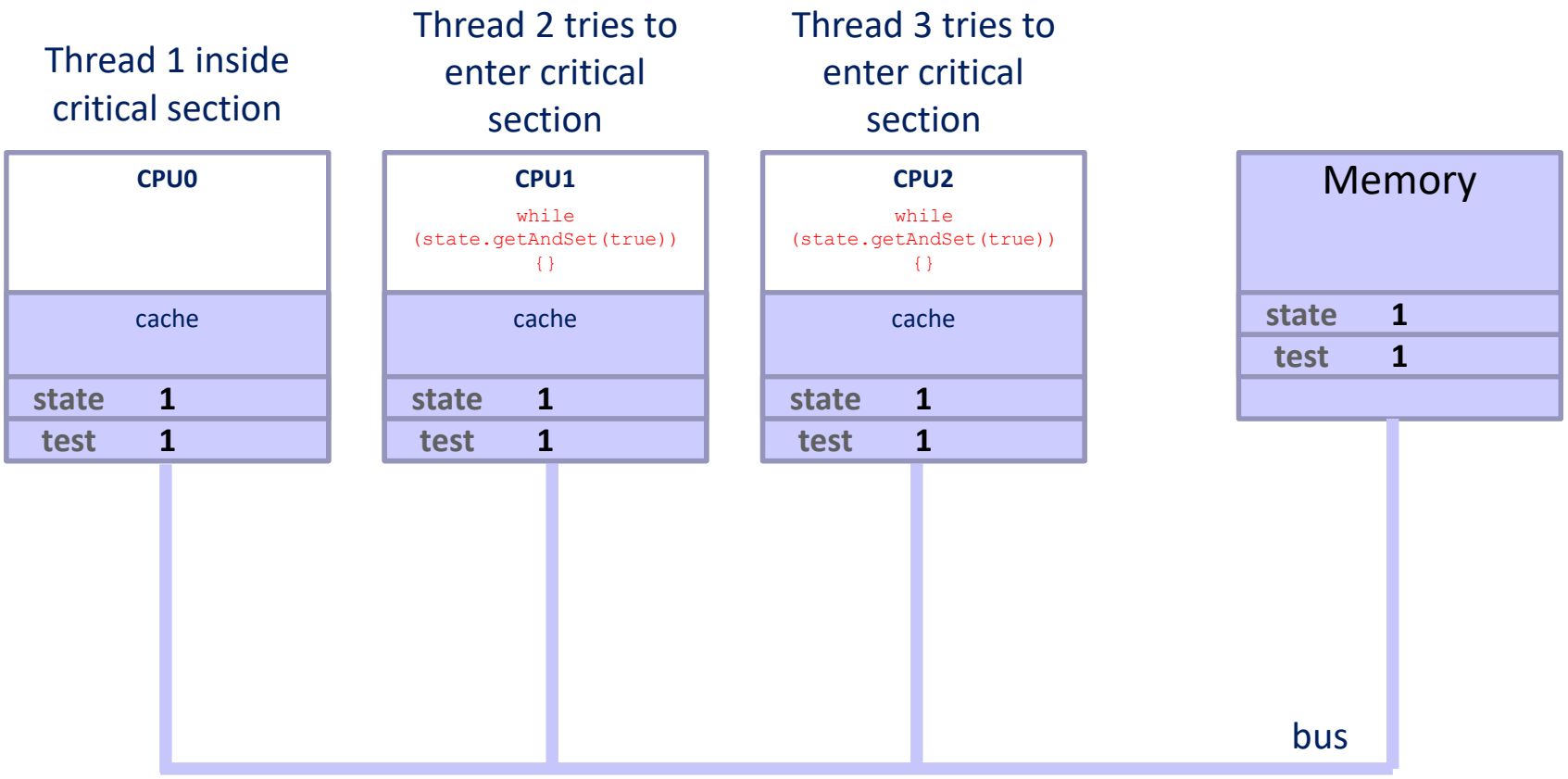
TAS lock in a bus-based multiprocessor



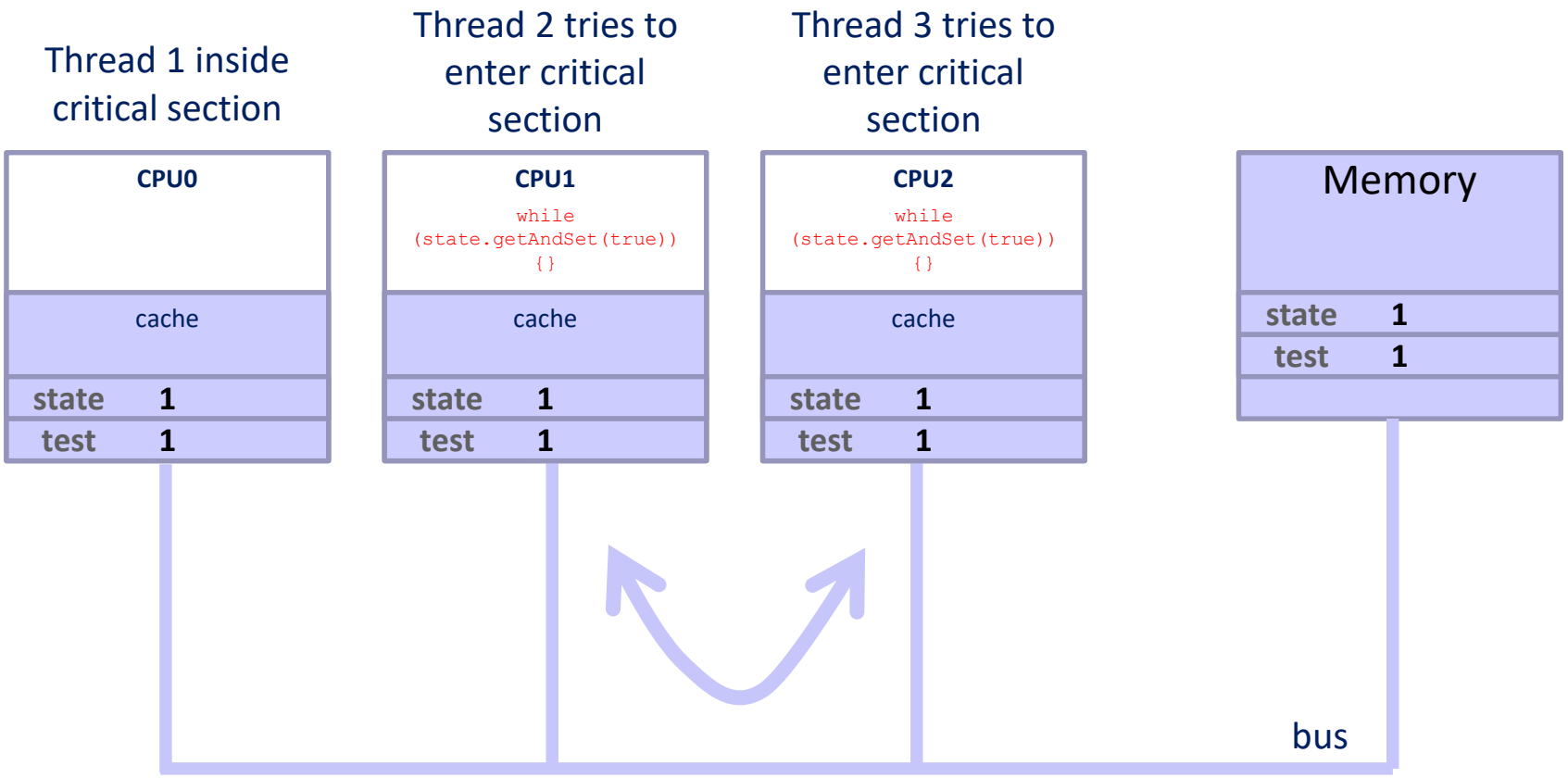
TAS lock in a bus-based multiprocessor



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

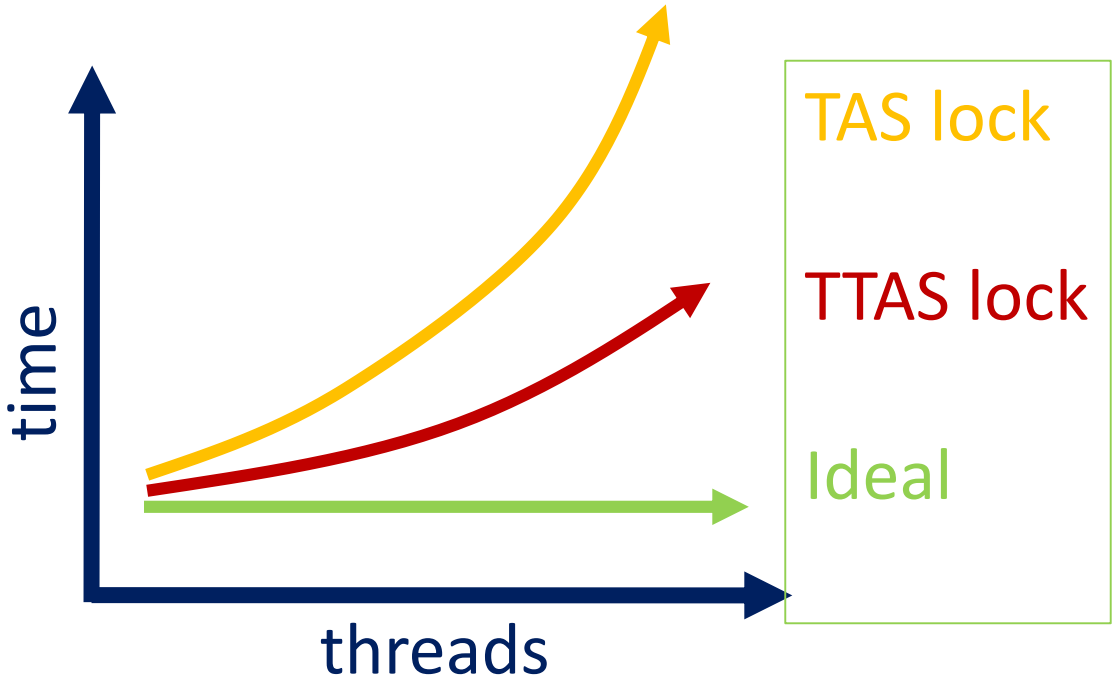
Test and test and set (TTAS) lock

```
class TTASlock {

 AtomicBoolean state = new Atomi

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

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

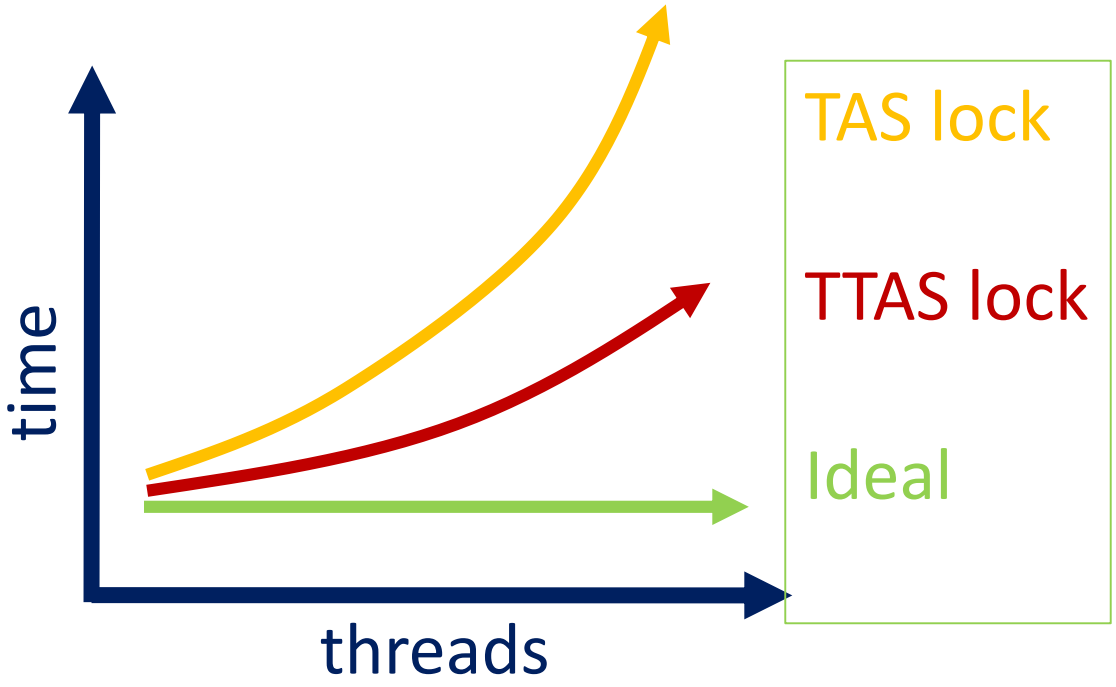
«Επίμονο» διάβασμα της κατάστασης του lock

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;
 }
}
```

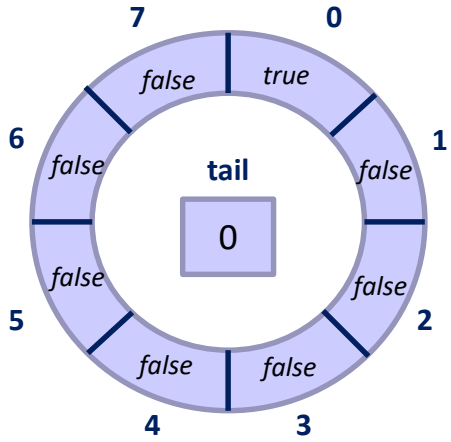
Τοπική μεταβλητή ανά thread

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;
 }
}
```

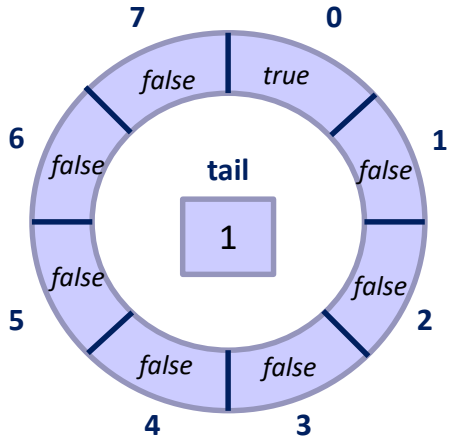



```
Alock mylock = new Alock(8);
```

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

Queue Locks: Array-based lock

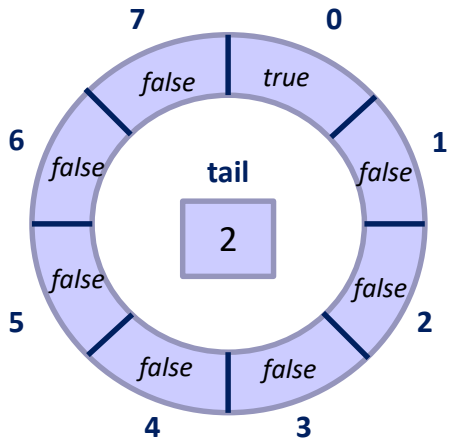
```
class ALock {

 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;
 }
}
```

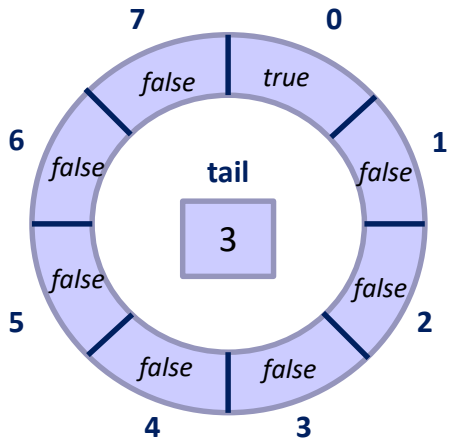


Thread A
in CS

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;
 }
}
```



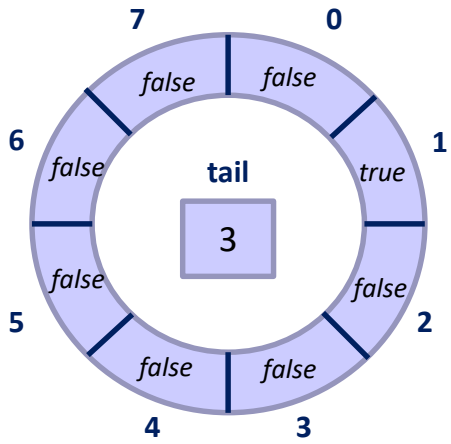
Thread A
in CS

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;
 }
}
```



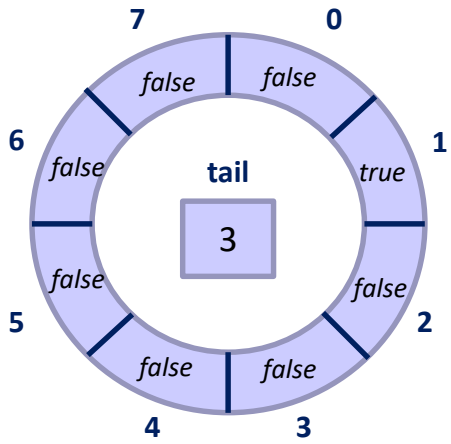
Thread A
mylock.unlock()

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.

Ερωτήσεις;