

# Δοσοληψίες

Κατανεμημένα Συστήματα  
2017-2018

<http://www.cslab.ece.ntua.gr/courses/distrib>

# Στο προηγούμενο μάθημα...

---

- Group communication
  - Multicast για FIFO διάταξη
  - Multicast για ολική διάταξη
    - Sequencer
    - ISIS
  - Multicast για αιτιώδη διάταξη
    - Χρησιμοποιεί vector timestamps
- Αμοιβαίος αποκλεισμός
  - Κεντρικός έλεγχος
  - Δακτύλιος με σκυτάλη
  - Αλγόριθμος Ricart and Agrawala
- Ο συντονισμός σε καταναμημένα συστήματα απαιτεί συχνά εκλογή αρχηγού
  - Αλγόριθμος δακτυλίου
  - Αλγόριθμος τροποποιημένου δακτυλίου
  - Αλγόριθμος bully

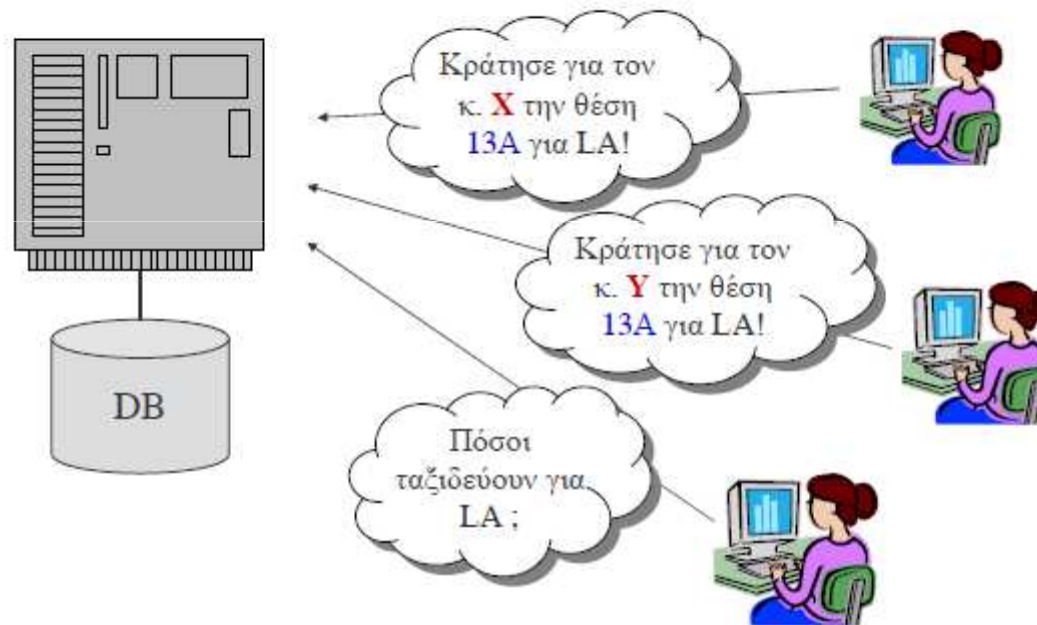
# Σε αυτό το μάθημα

---

- Απλές δοσοληψίες (transactions)
- Ιδιότητες ACID
  - Και κυρίως atomicity και durability
- Σειριοποιησιμότητα (serializability)
- Έλεγχος ταυτοχρονισμού
  - Κλειδώματα (Locking)
  - Αισιόδοξος έλεγχος ταυτοχρονισμού (optimistic concurrency control)
  - Διάταξη χρονοσφραγίδων (Timestamp ordering)
- Αδιέξοδα (deadlocks)

# Παράδειγμα 1

---



# Παράδειγμα 2

---

- Συναλλαγή πελάτη με την τράπεζα (σε ATM ή browser)
  - Μεταφορά \$100 από λογαριασμό ταμιευτηρίου σε τρεχούμενο λογαριασμό
  - Μεταφορά \$200 από προθεσμιακό σε τρεχούμενο λογαριασμό
  - Ανάληψη \$400 από τρεχούμενο
- Συναλλαγή
  - ταμιευτηρίου.αφαίρεση(100)
  - τρεχούμενο.πρόσθεση(100)
  - προθεσμιακό.αφαίρεση(200)
  - τρεχούμενο.πρόσθεση(200)
  - τρεχούμενο.αφαίρεση (400)
  - ανάληψη(400)

# Γιατί δε μας αρέσει

Αποτυχία σε αυτά τα σημεία: ο πελάτης χάνει χρήματα. χρειαζόμαστε τρόπο να ανακτήσουμε προηγούμενη κατάσταση

1. ταμειυτηρίου.αφαίρεση(100)
2. τρεχούμενο.πρόσθεση(100)
3. προθεσμιακό.αφαίρεση(200)
4. τρεχούμενο.πρόσθεση(200)
5. τρεχούμενο.αφαίρεση (400)
6. ανάληψη(400)

Αποτυχία σε αυτά τα σημεία: δεν χάνονται χρήματα. Αλλα δεν μπορούν να επαναληφθούν τα προηγούμενα βήματα

# Κάτι μου θυμίζει...

---

- Τι μπορεί να πάει στραβά;
  - Πολλοί clients
  - Πολλοί servers
- Πώς το λύνεις;
  - Ολόκληρη η συναλλαγή σε ένα βήμα
- Πού το έχω ξαναδεί;
  - Mutual exclusion
- Άρα τελειώσαμε;

# Ταυτόχρονες συναλλαγές

---

- Process 1

```
lock(mutex);  
ταμιευτηρίου.αφαίρεση(100)  
τρεχούμενο.πρόσθεση(100)  
προθεσμιακό.αφαίρεση(200)  
τρεχούμενο.πρόσθεση(200)  
τρεχούμενο.αφαίρεση (400)  
ανάληψη(400)  
unlock(mutex);
```

- Process 2

```
lock(mutex);  
ταμιευτηρίου.αφαίρεση(200);  
τρεχούμενο.πρόσθεση(200);  
unlock(mutex);
```



# Θυμήσου

---

- mutual exclusion = one big lock.
  - Όλοι οι υπόλοιποι πρέπει να περιμένουν
  - Δεν αντιμετωπίζει τα σφάλματα του συστήματος
- Απόδοση
  - Μπορούμε να συνδυάσουμε λειτουργίες από διαφορετικές διεργασίες
- Σφάλματα
  - Τι γίνεται αν κρασάρει μια διεργασία που έχει το lock;

# Η έννοια του transaction

---

- Ένα διατεταγμένο σύνολο από ενέργειες
- Ένα transaction είναι ατομικό (atomic) δηλ αδιαίρετο σε σχέση με άλλα transactions
  - Δεν υπάρχει πρόσβαση σε ενδιάμεσα αποτελέσματα/καταστάσεις
  - Δεν υπάρχει παρεμβολή από άλλες ενέργειες
- Primitives
  - `openTransaction()` -> `trans`;
  - `closeTransaction(trans)` -> (`commit`, `abort`);
  - `abortTransaction(trans)`;
- Υλοποιώντας transactions
  - Απόδοση (Performance): ποιες ενέργειες μπορούμε να τρέξουμε ταυτόχρονα
  - Σφάλματα (Failure): πώς χειριζόμαστε σφάλματα, κάνοντας roll-back αλλαγές αν χρειαστεί

# Ιδιότητες ACID

---

- **Ατομικότητα (Atomicity):** όλα ή τίποτα, δηλαδή, είτε όλες οι πράξεις μια δοσοληψία επιτυγχάνουν, είτε όλες αποτυγχάνουν
- **Συνέπεια (Consistency):** αν ο server ξεκινά από συνεπή κατάσταση, το transaction τον αφήνει πάλι σε συνεπή κατάσταση
- **Απομόνωση (Isolation):** Κάθε δοσοληψία τρέχει χωρίς παρεμβολές, δηλαδή ακόμα κι αν τρέχουν πολλές δοσοληψίες ταυτόχρονα, κάθε δοσοληψία είναι σαν να τρέχει μόνη της
- **Μονιμότητα (Durability):** Αν μια δοσοληψία επιτύχει, το αποτέλεσμα της είναι μόνιμο και επιβιώνει ακόμα κι αν αποτύχει το σύστημα, δηλαδή γράφεται στον δίσκο

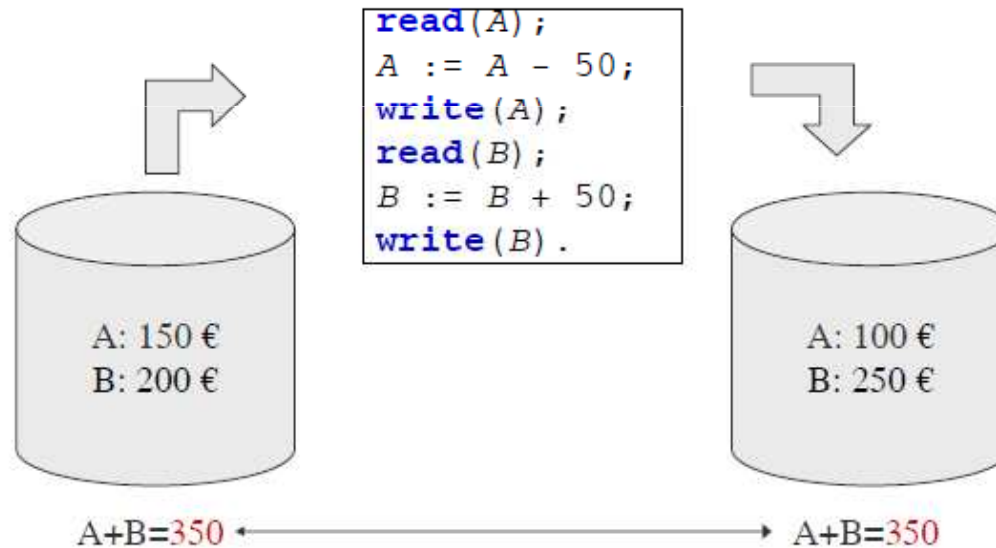
# Παράδειγμα: Μεταφορά χρημάτων

---

- Δοσοληψία που μεταφέρει \$50 από τον A στον B:
  1. read(A)
  2.  $A := A - 50$
  3. write(A)
  4. read(B)
  5.  $B := B + 50$
  6. write(B)

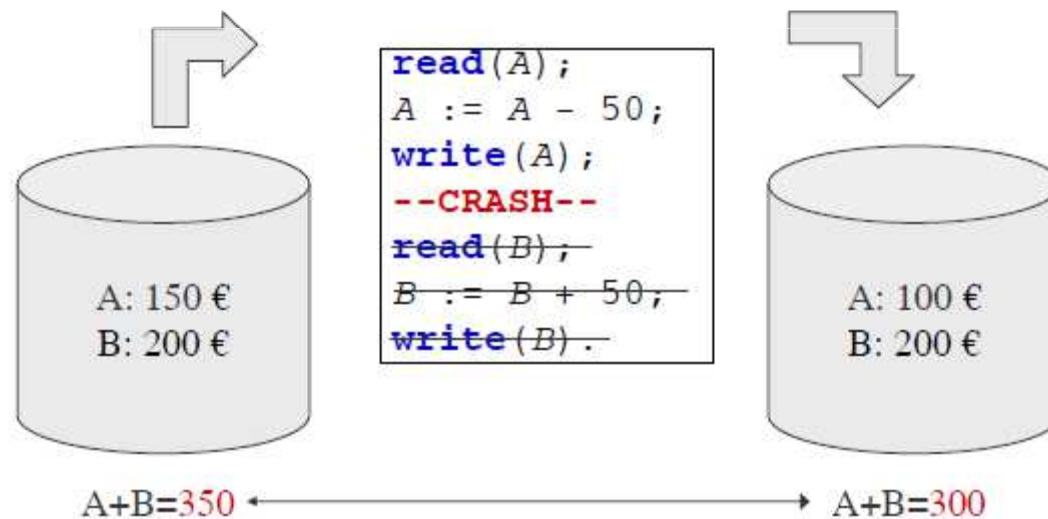
# ΣΥΝΕΠΕΙΑ

- Το άθροισμα του A και του B παραμένει αμετάβλητο από την εκτέλεση του transaction



# Ατομικότητα

- Αν το transaction αποτύχει μετά το βήμα step 3 και πριν το βήμα 6, το σύστημα πρέπει να εξασφαλίσει ότι θα αναιρεθούν τα βήματα 1-3, αλλιώς θα υπάρχει ασυνέπεια

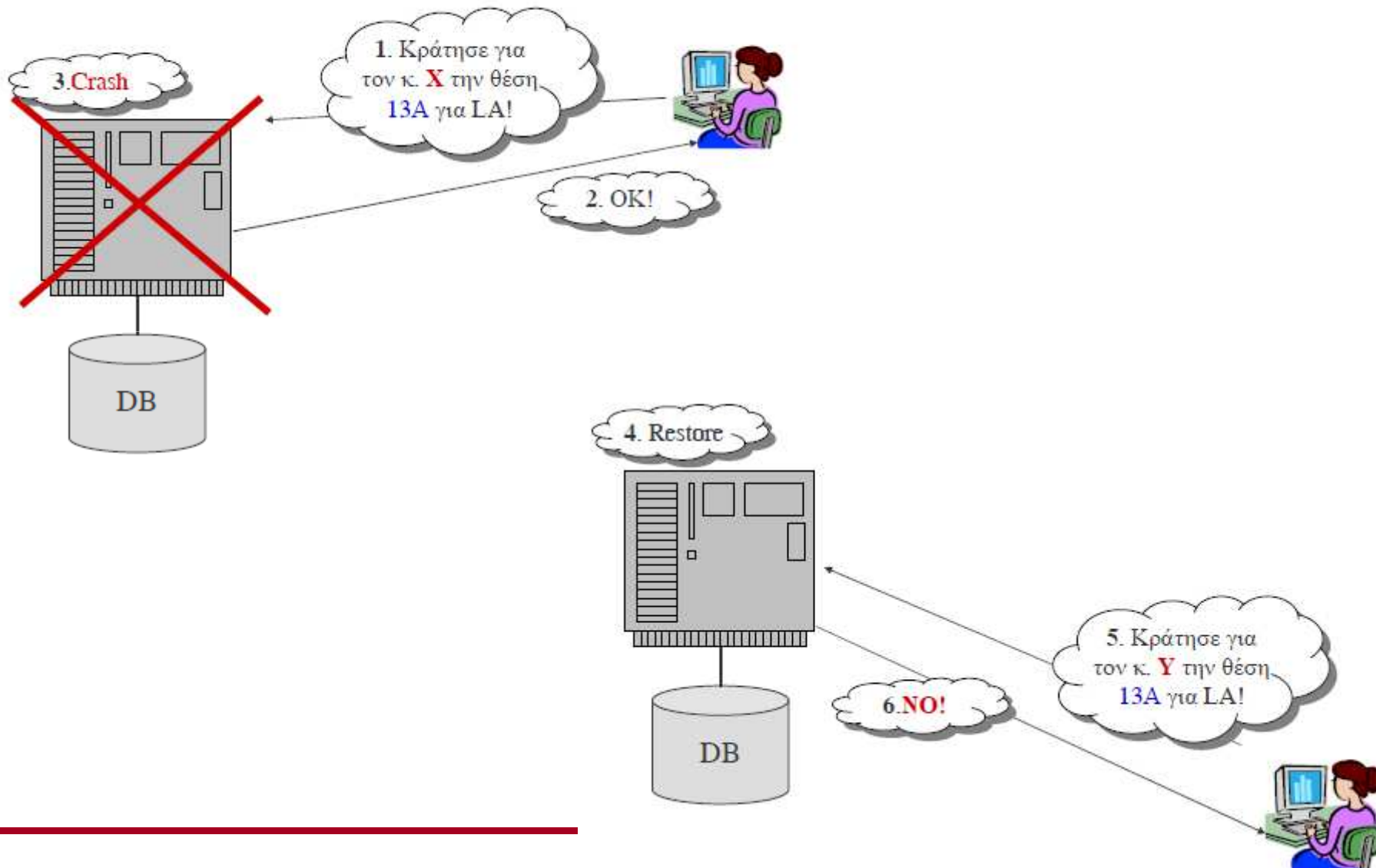


# Μονιμότητα (1)

---

- Αν το transaction επιτύχει, δηλδ ο χρήστης ενημερωθεί ότι έγινε η μεταφορά χρημάτων επιτυχώς, αυτό δε θα αλλάξει από κανένα σφάλμα.

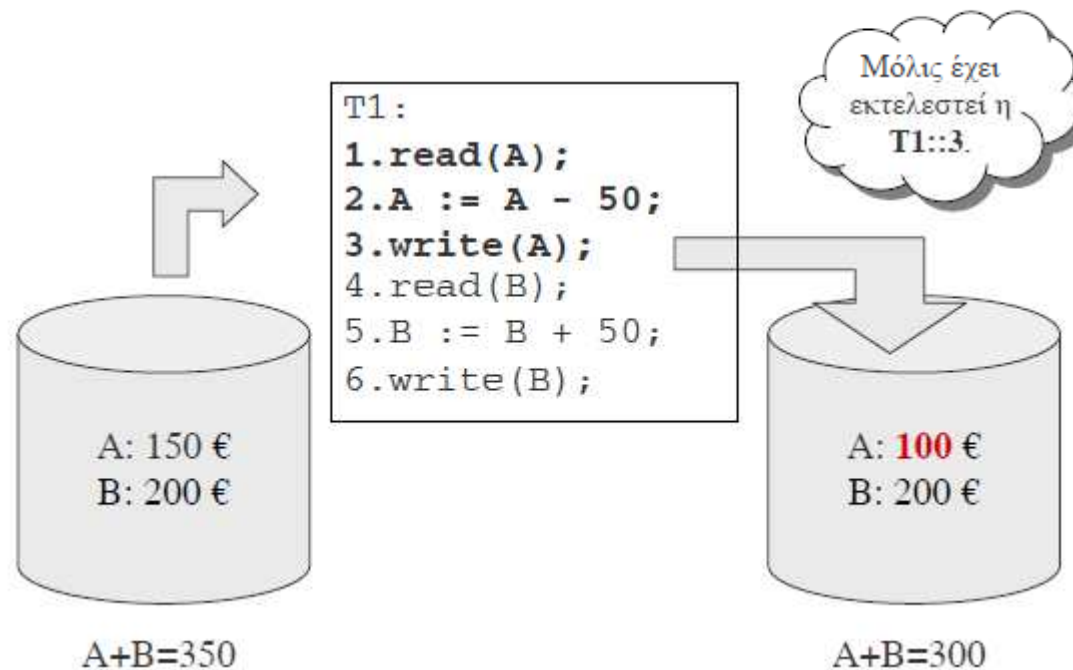
# Μονιμότητα (2)





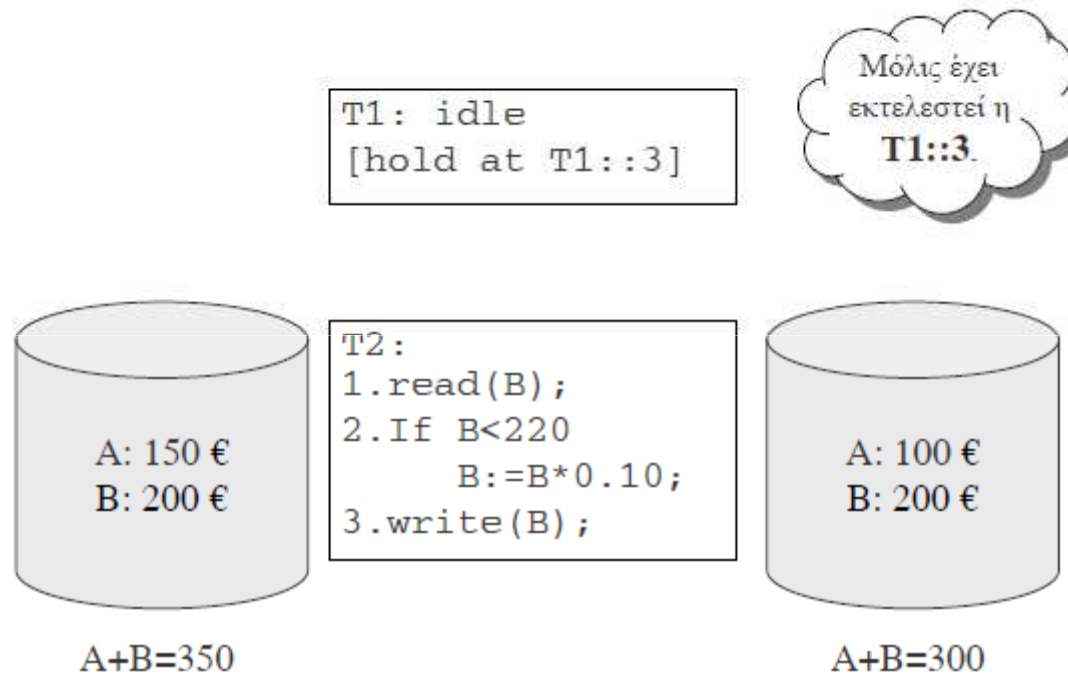
# Απομόνωση (1)

- Αν στο σύστημα τρέχουν παραπάνω από μια δοσοληψίες ταυτόχρονα και η μια μπορεί να δει τα ενδιάμεσα αποτελέσματα της άλλης (π.χ. ανάμεσα στο βήμα 3 και 6) μπορεί να έχουμε ασυνέπεια.  
Η πιο απλή ιδανική λύση είναι η **σειριακή** εκτέλεση transactions , είναι όμως κακή από πλευράς απόδοσης



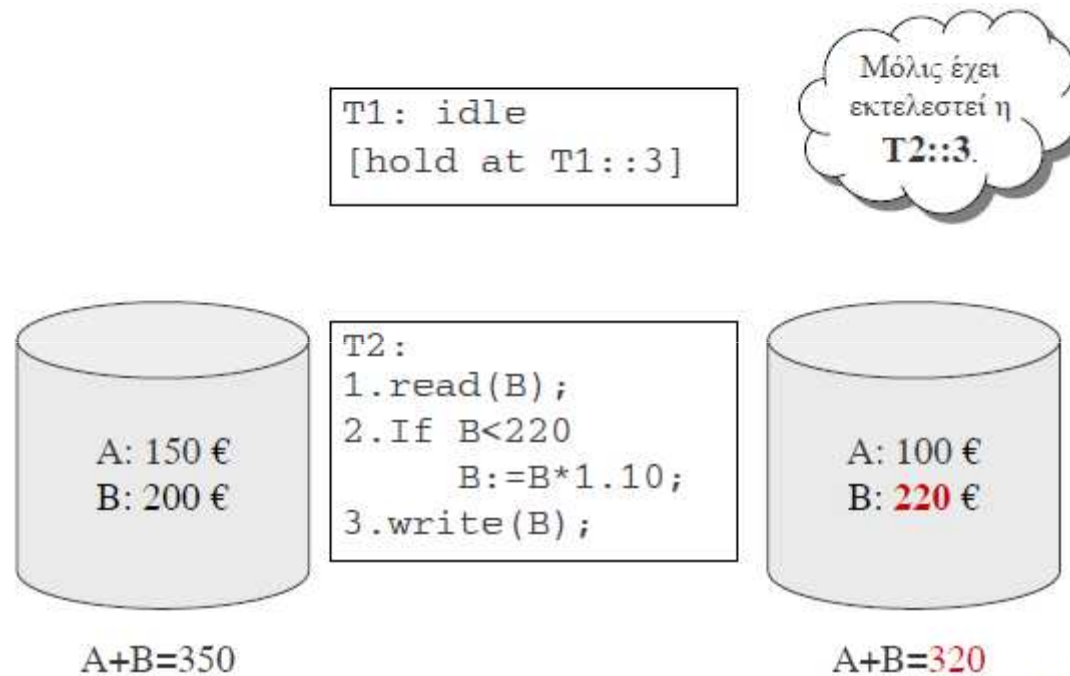
# Απομόνωση (2)

---



# Απομόνωση (3)

---



# Ταυτόχρονη Εκτέλεση

---

- Και γιατί να μην τρέχουμε σειριακά τις δοσοληψίες, τη μία μετά την άλλη;
  - **Αποδοτικότερη χρήση CPU και I/O:** μεγαλύτερο transaction *throughput*: ένα transaction μπορεί να χρησιμοποιεί CPU ενώ άλλο να διαβάζει ή να γράφει στον δίσκο
  - **Μειωμένος μέσος χρόνος απόκρισης** για τα transactions: Οι σύντομες δοσοληψίες, δεν έχουν λόγο να αναμένουν την ολοκλήρωση των πιο χρονοβόρων
- *Αλγόριθμοι διαπλοκής (interleaving) των δοσοληψιών – μηχανισμοί για isolation*

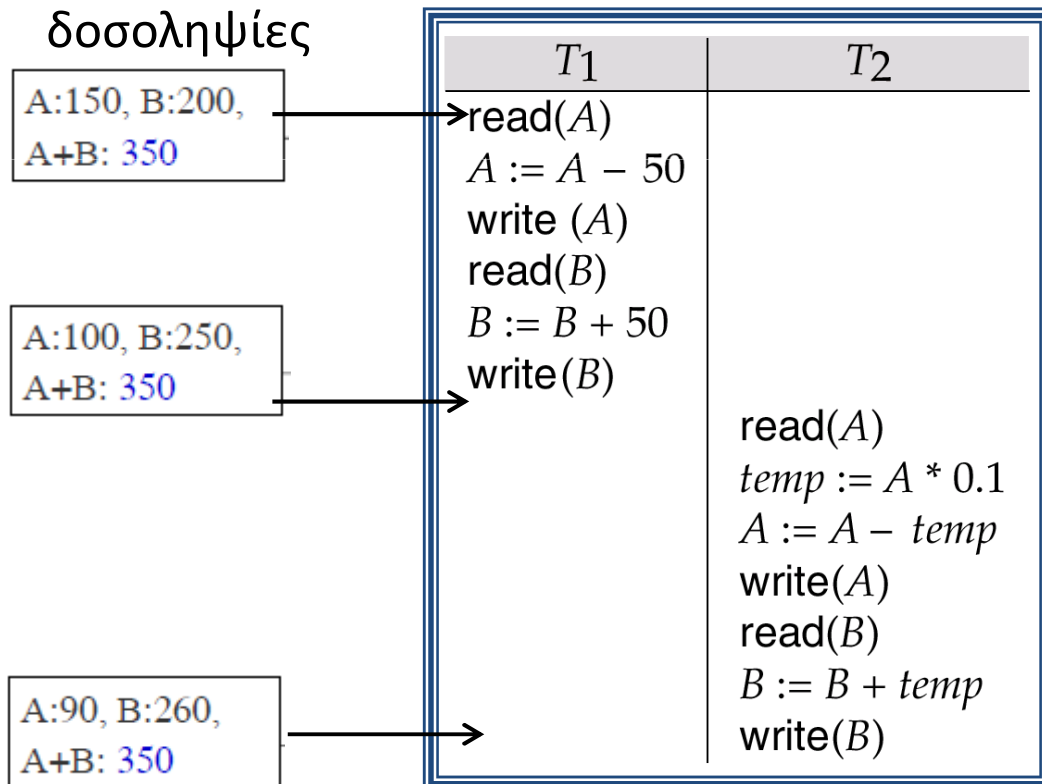
# Χρονοπρογράμματα - Schedules

---

- *Schedules* – Σειρά από ενέργειες που δείχνουν τη χρονολογική σειρά με την οποία τελικά εκτελέστηκαν ομάδες από ταυτόχρονες δοσοληψίες
  - Περιέχει όλες τις ενέργειες των εμπλεκόμενων δοσοληψιών
  - Διατηρεί τη σειρά με την οποία εμφανίζονται οι ενέργειες σε κάθε δοσοληψία

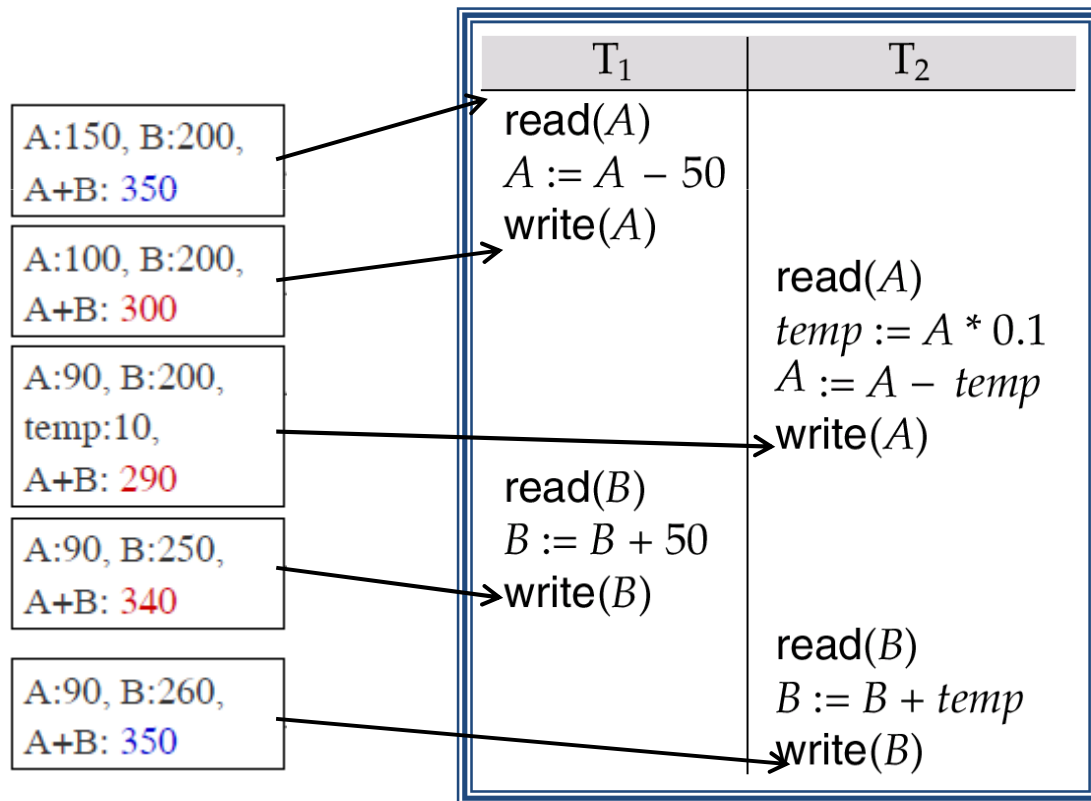
# Παράδειγμα

- Το  $T_1$  μεταφέρει \$50 από το  $A$  στο  $B$
- Το  $T_2$  μεταφέρει 10% του υπολοίπου του  $A$  στο  $B$ .
- **Σειριακό** χρονοπρόγραμμα: Πρώτα όλο το  $T_1$  και μετά όλο το  $T_2$  ή πρώτα όλο το  $T_2$  και μετά το  $T_1$  (n! δυνατά σειριακά χρονοπρογράμματα για n δοσοληψίες



# Παράδειγμα (2)

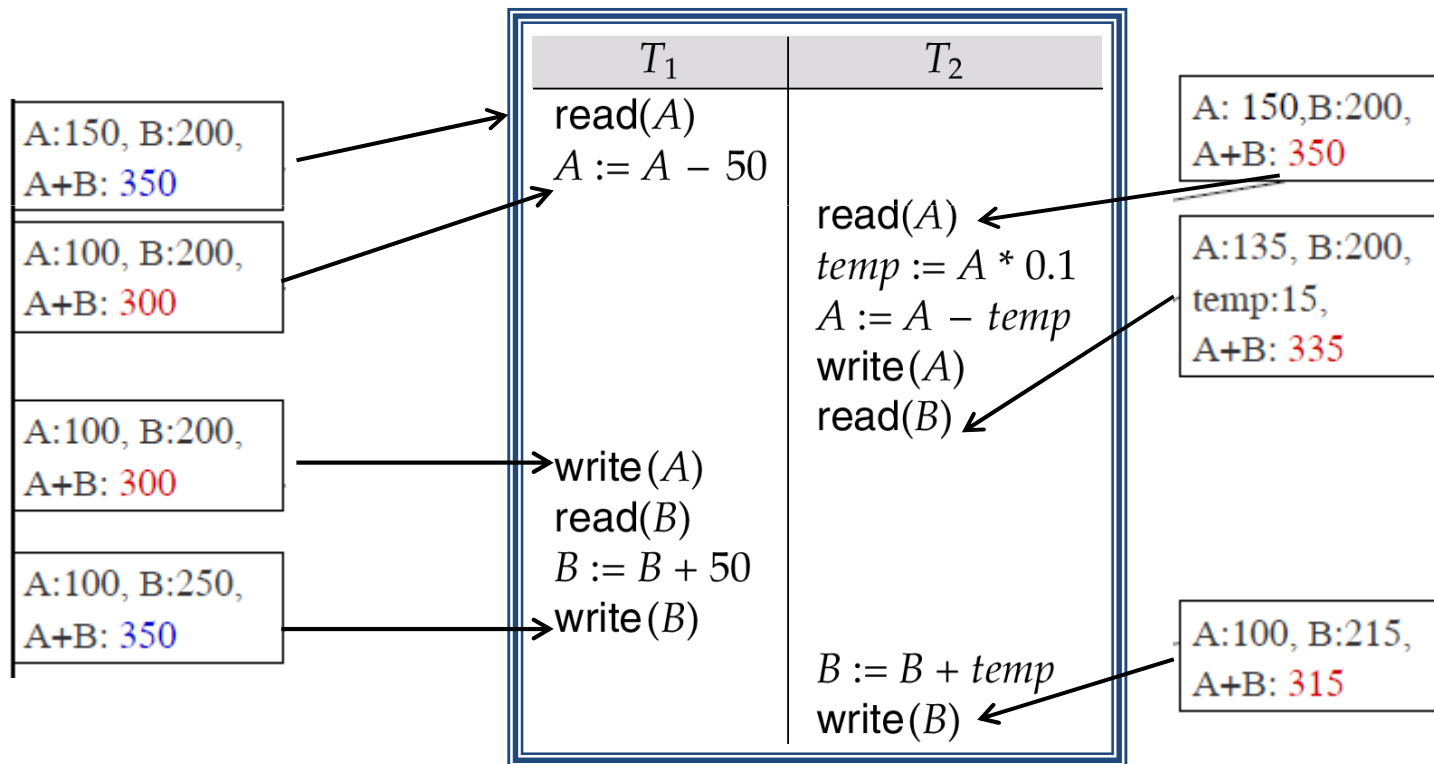
- Δεν είναι σειριακό χρονοπρόγραμμα αλλά είναι ισοδύναμο με ένα σειριακό



Διατηρείται το άθροισμα  
A + B

# Παράδειγμα (3)

- Αυτό το χρονοπρόγραμμα δεν είναι ισοδύναμο με το σειριακό, γιατί δε διατηρεί την τιμή  $sum\ A + B$ .





# Προβλήματα σε χρονοπρογράμματα

---

- Ασυνεπείς αναγνώσεις (dirty reads)
- Διαδοχικές αναιρέσεις (cascading aborts)
- Απώλειες ενημερώσεων (lost updates)

# Ασυνεπής ανάγνωση (dirty read)

- Το μερικό αποτέλεσμα ενός transaction διαβάζεται από κάποιο άλλο
- a:200, b:200

**Transaction V:**

*a.withdraw(100)*  
*b.deposit(100)*

**Transaction W:**

*branchTotal()*

*a.withdraw(100);*                      \$100

*total = a.getBalance()*                      \$100

*total = total+b.getBalance()*                      \$300

*total = total+c.getBalance()*

*b.deposit(100)*                      \$300

⋮

- Το μερικό αποτέλεσμα του V χρησιμοποιείται από το W και τελικά δίνει λάθος αποτέλεσμα

# Διαδοχικές αναιρέσεις (cascading aborts)

Transaction T:	Transaction U:
<i>a.getBalance()</i> <i>a.setBalance(balance + 10)</i>	<i>a.getBalance()</i> <i>a.setBalance(balance + 20)</i>
<i>balance = a.getBalance()</i> \$100 <i>a.setBalance(balance + 10)</i> \$110	<i>balance = a.getBalance()</i> \$110 <i>a.setBalance(balance + 20)</i> \$130 <i>commit transaction</i>
<i>abort transaction</i>	

- Το abort του T απαιτεί και abort του U

# Απώλεια ενημέρωσης (lost update)

---

- Ένα transaction προκαλεί απώλεια πληροφορίας για ένα άλλο

## Transaction *T*:

```
balance = b.getBalance();  
b.setBalance(balance*1.1);
```

```
balance = b.getBalance(); $200
```

```
b.setBalance(balance*1.1); $220
```

## Transaction *U*:

```
balance = b.getBalance();  
b.setBalance(balance*1.1);
```

```
balance = b.getBalance(); $200
```

```
b.setBalance(balance*1.1); $220
```

---

T/U' s update on the shared object, "b", is lost

# Γιατί συμβαίνει αυτό

---

- Γιατί τα προηγούμενα χρονοπρογράμματα δεν είναι *σειριοποιήσιμα* (serializable)
  - Δλδ δεν έχουν το ίδιο αποτέλεσμα με μια σειριακή εκτέλεση
- Χρονοπρογράμματα με ταυτόχρονες δοσοληψίες πρέπει να έχουν το ίδιο αποτέλεσμα σε όλες τις μεταβλητές όπως ένα σειριακό χρονοπρόγραμμα
  - Δλδ πρέπει να έχουν πρόσβαση σε αντικείμενα (για read/write) με τρόπο ανάλογο της σειριακής εκτέλεσης

# Σειριοποιησιμότητα

---

- Ένα χρονοπρόγραμμα είναι σειριοποιήσιμο (serializable) αν είναι ισοδύναμο με ένα σειριακό χρονοπρόγραμμα.
- 2 τεχνικές
  - Σειριοποιησιμότητα συγκρούσεων (conflict serializability)
  - Σειριοποιησιμότητα όψεως (view serializability)
- Ποιες λειτουργίες λαμβάνουμε υπόψιν;
  - Read/write
- Ποιες λειτουργίες έχουν σημασία για την ορθότητα;
  - write

# Συγκρούσεις *read* and *write*

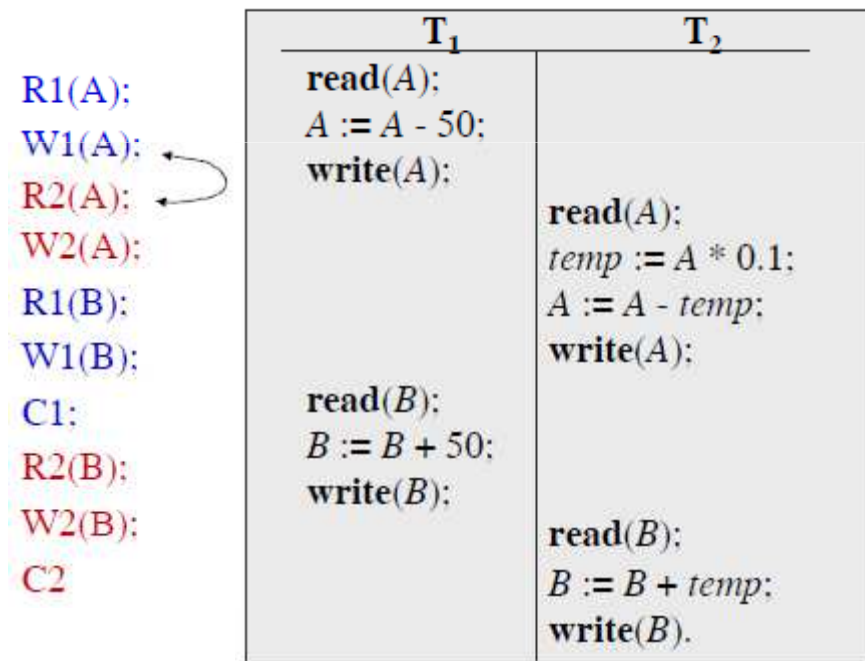
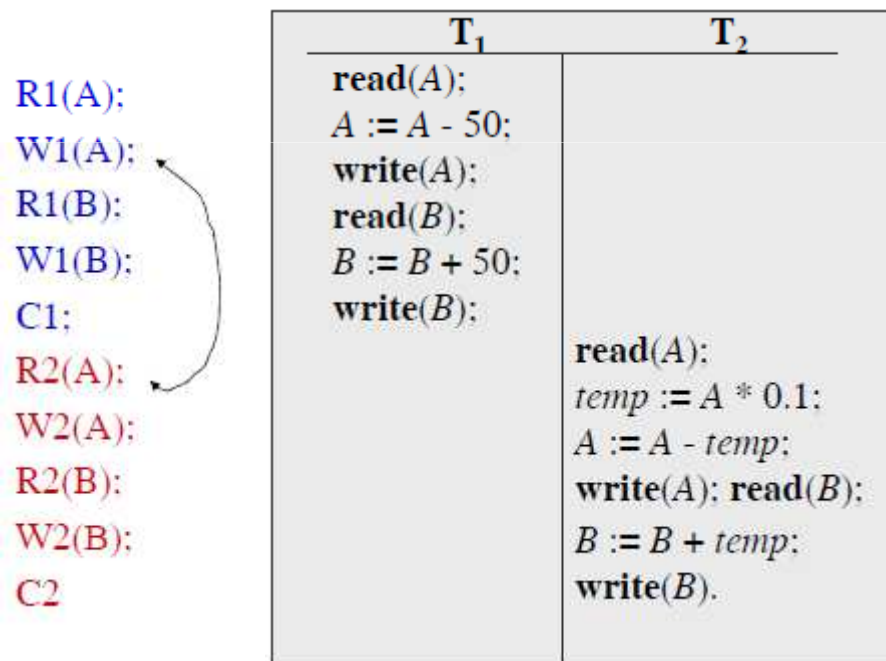
---

<i>Operations of different transactions</i>		<i>Conflict</i>	<i>Reason</i>
<i>read</i>	<i>read</i>	No	Because the effect of a pair of <i>read</i> operations does not depend on the order in which they are executed
<i>read</i>	<i>write</i>	Yes	Because the effect of a <i>read</i> and a <i>write</i> operation depends on the order of their execution
<i>write</i>	<i>write</i>	Yes	Because the effect of a pair of <i>write</i> operations depends on the order of their execution

---

# Σειριοποιησιμότητα Συγκρούσεων

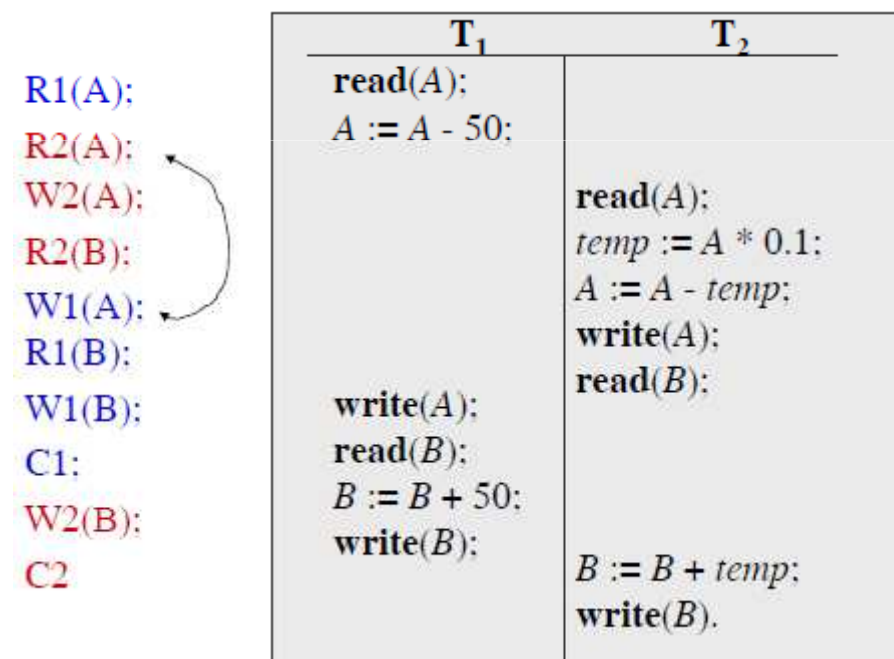
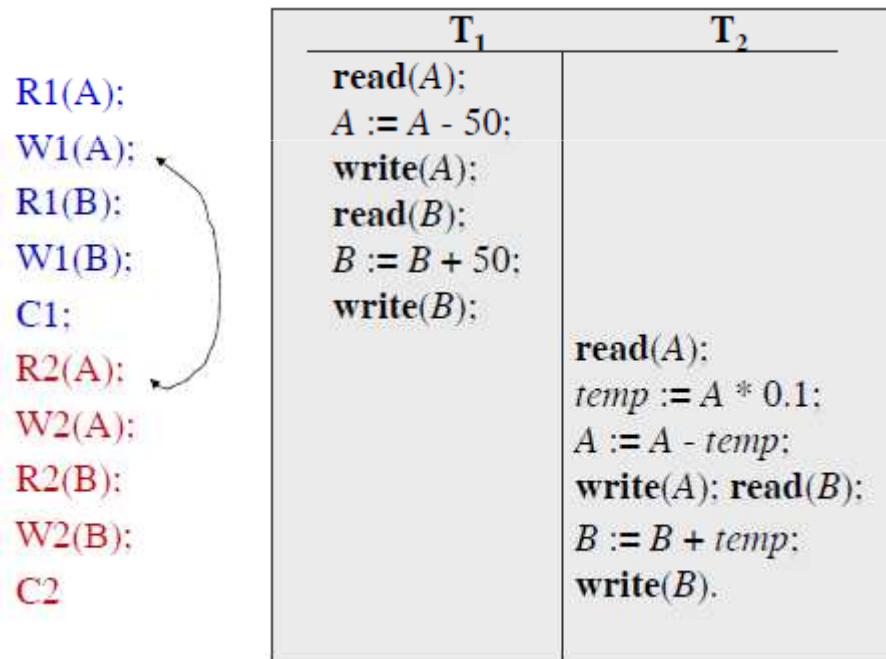
- Δύο χρονοπρογράμματα είναι ισοδύναμα συγκρούσεων αν για κάθε σύγκρουση, οι συγκρουόμενες πράξεις έχουν την ίδια σειρά στα 2 προγράμματα





# Σειριοποιησιμότητα Συγκρούσεων (2)

- Προβληματικό χρονοπρόγραμμα



# Τυπικός και Άτυπος Ορισμός

- Ένα πρόγραμμα είναι σειριοποιήσιμο συγκρούσεων (conflict serializable) αν είναι ισοδύναμο συγκρούσεων με ένα σειριακό
- Στο σειριακό χρονοπρόγραμμα κάθε δοσοληψία «ξεμπερδώνει» ξεχωριστά με κάθε αντικείμενο και μετά το αναλαμβάνει μια άλλη
- Σε ένα σειριοποιήσιμο, το να μην υπάρχει σύγκρουση σημαίνει ότι το πρόγραμμα «ξεμπερδώνει» με τα αντικείμενα με την ίδια σειρά ανά δοσοληψία όπως κι ένα σειριακό

T <sub>1</sub>	T <sub>2</sub>
read(A);	
write(A);	
	read(B);
	write(B);
read(B);	
write(B);	
	read(A);
	write(A);

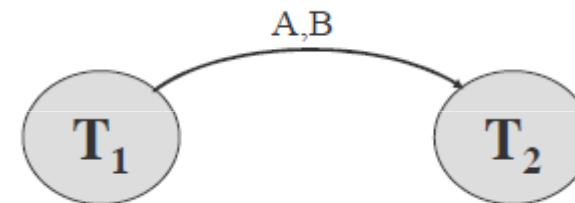
# Έλεγχος σειριοποιησιμότητας

---

- Γράφος σειριοποιησιμότητας
  - Κάθε δοσοληψία ένας κόμβος
  - Κατευθυνόμενη ακμή από την  $T_i$  στην  $T_j$  αν μια ενέργεια της  $T_i$  συγκρούεται με μια επακόλουθή της, της  $T_j$
  - Πάνω στην ακμή σημειώνουμε το αντικείμενο για το οποίο συγκρούονται οι δοσοληψίες
- Γράφος με κύκλο: μη σειριοποιήσιμος σε σχέση με συγκρούσεις
- Γράφος χωρίς κύκλο: σειριοποιήσιμος σε σχέση με συγκρούσεις

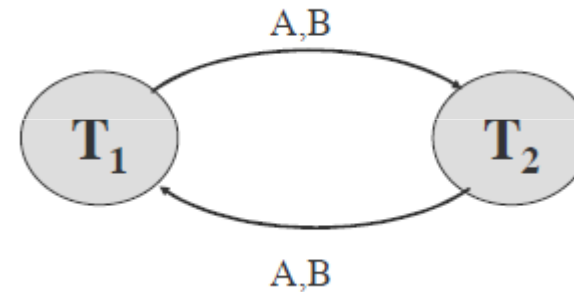
# Παράδειγμα

$T_1$	$T_2$
<code>read(A); A := A - 50; write(A);</code>	<code>read(A); temp := A * 0.1; A := A - temp; write(A);</code>
<code>read(B); B := B + 50; write(B);</code>	<code>read(B); B := B + temp; write(B);</code>



# Παράδειγμα

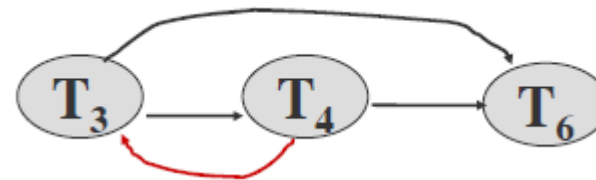
$T_1$	$T_2$
<code>read(A); A := A - 50;</code>	<code>read(A); temp := A * 0.1; A := A - temp; write(A); read(B);</code>
<code>write(A); read(B); B := B + 50; write(B);</code>	<code>B := B + temp; write(B).</code>



# Σειριοποιησιμότητα όψεως

- Δεν είναι conflict serializable....
- Τι μας νοιάζει, αφού τελικά σημασία έχει τι γράφει η  $T_6$

$T_3$	$T_4$	$T_6$
read( $Q$ )	write( $Q$ )	
write( $Q$ )		write( $Q$ )



# Σειριοποιησιμότητα όψεως

---

- Κάθε conflict serializable χρονοπρόγραμμα είναι και view serializable.
- Δεν ισχύει το αντίστροφο
- Κάθε view serializable χρονοπρόγραμμα που δεν είναι conflict serializable έχει τυφλές εγγραφές (blind writes), δηλ writes για τα οποία δεν έχουν προηγηθεί reads στη δοσοληψία τους
- Σειριοποιησιμότητα όψεως: NP-complete αλγόριθμος

# Στην πράξη

---

- Αλγόριθμοι ελέγχου σειριοποιησιμότητας πολύ ακριβοί
- Έλεγχος αφού φτιαχτεί το χρονοπρόγραμμα... λίγο αργά
- Online πρωτόκολλα ελέγχου ταυτοχρονισμού δοσοληψιών
  - Locking
  - Optimistic concurrency control
  - Timestamp ordering
- Τι συμβαίνει με τα σφάλματα;
  - Πρέπει να μπορούμε να κάνουμε roll-back στην κατάσταση πριν τα transactions



# Αυστηρή εκτέλεση Transactions

---

- Τα transactions πρέπει να καθυστερούν τα reads και writes σε ένα αντικείμενο μέχρι όλα τα υπόλοιπα transactions που έγραψαν το ίδιο αντικείμενο να κάνουν είτε commit είτε abort

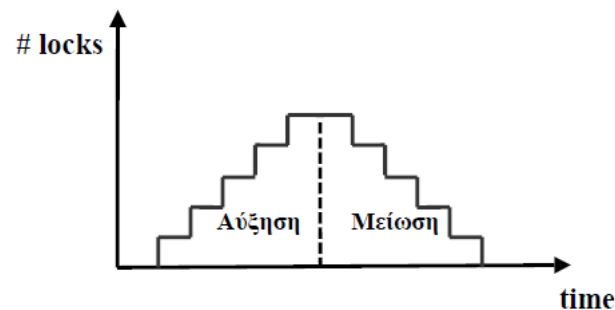
# Πρωτόκολλα κλειδώματος

---

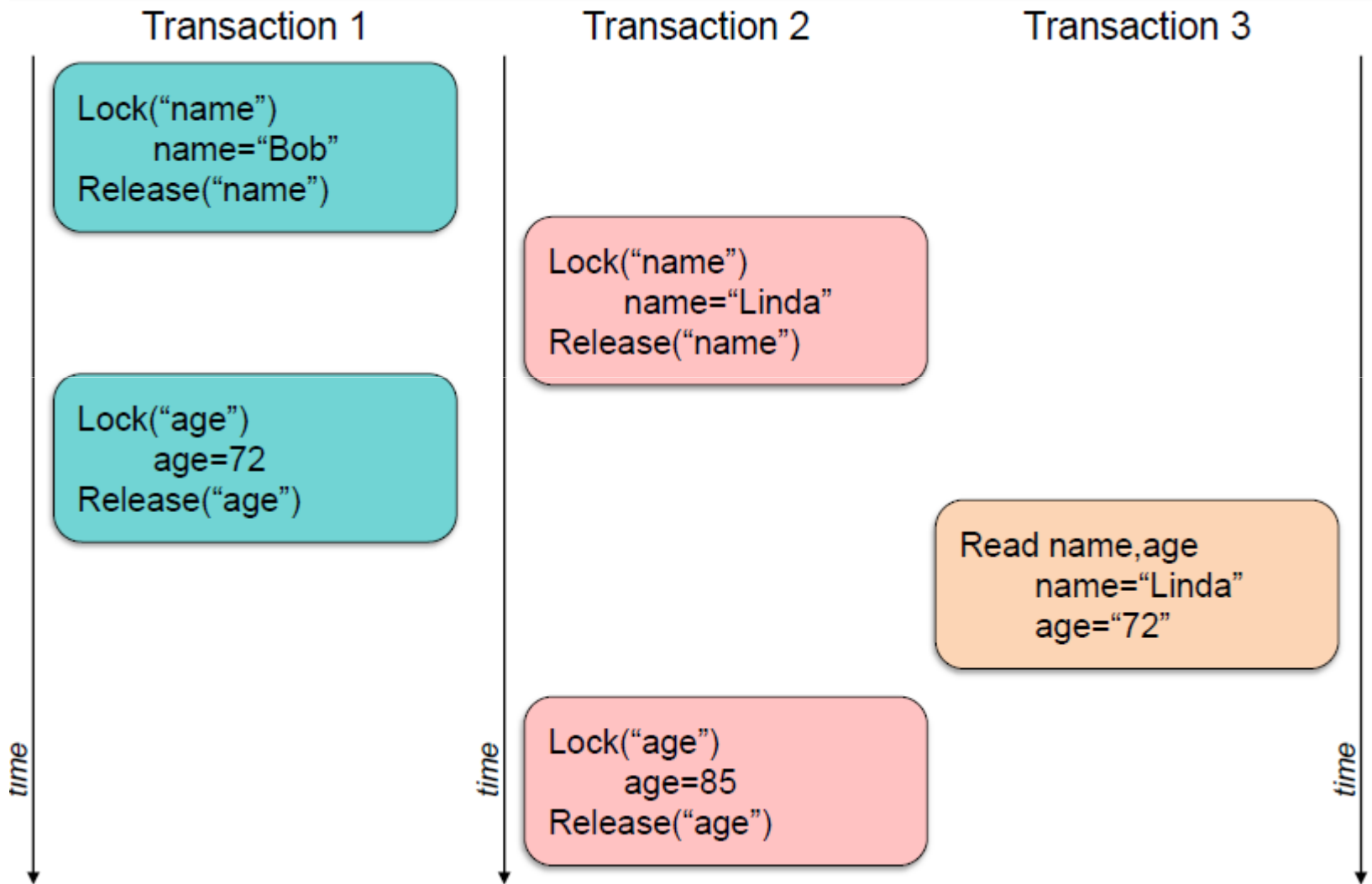
- Διασφαλίζουν την σειριοποιησιμότητα μιας σειράς ενεργειών με κανόνες για το τι μπορεί να προσπελάσει μια δοσοληψία.
- Βασική έννοια του κλειδώματος (lock)
  - μεταβλητή που σχετίζεται με ένα στοιχειώδες δεδομένο και περιγράφει την κατάστασή του, σε σχέση με πιθανές πράξεις που μπορούν να εφαρμοσθούν σε αυτό.

# 2 phase locking (2PL)

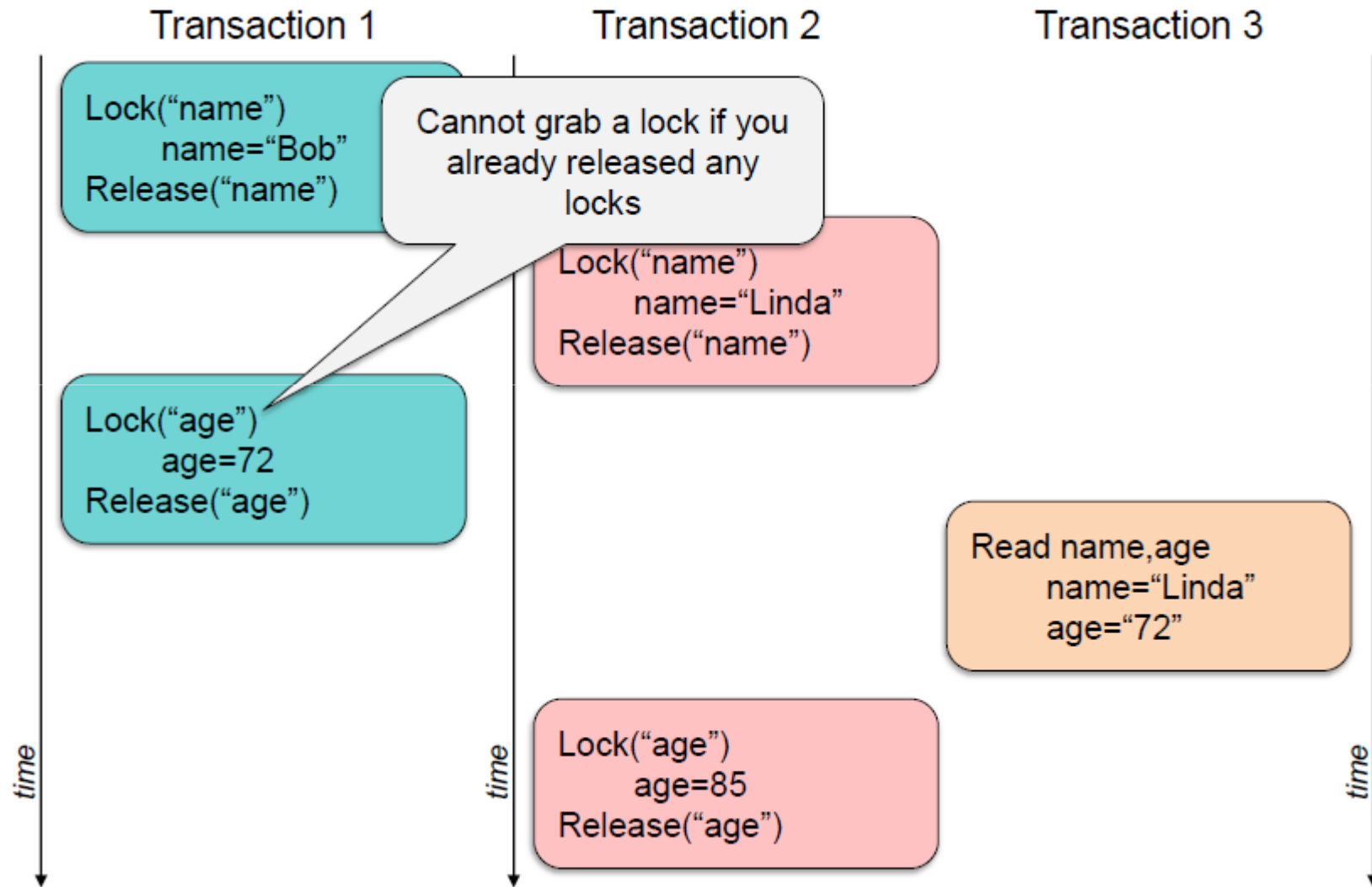
- Αλγόριθμος Κλειδώματος Δύο Φάσεων
- Εξασφαλίζει τη σειριοποιησιμότητα επιτρέποντας στις δοσοληψίες να προσπελάσουν αντικείμενα, αν καταφέρουν να αποκτήσουν αποκλειστικό (exclusive) lock γι' αυτά.
- Αντικείμενο: μεταβλητές, εγγραφές βάσης, σελίδες...
- Πρώτη φάση (growing phase): παίρνω νέα locks
- Δεύτερη φάση (shrinking phase): απελευθερώνω τα locks
- Ένα transaction δεν παίρνει νέο lock, αν έχει απελευθερώσει έστω κι ένα



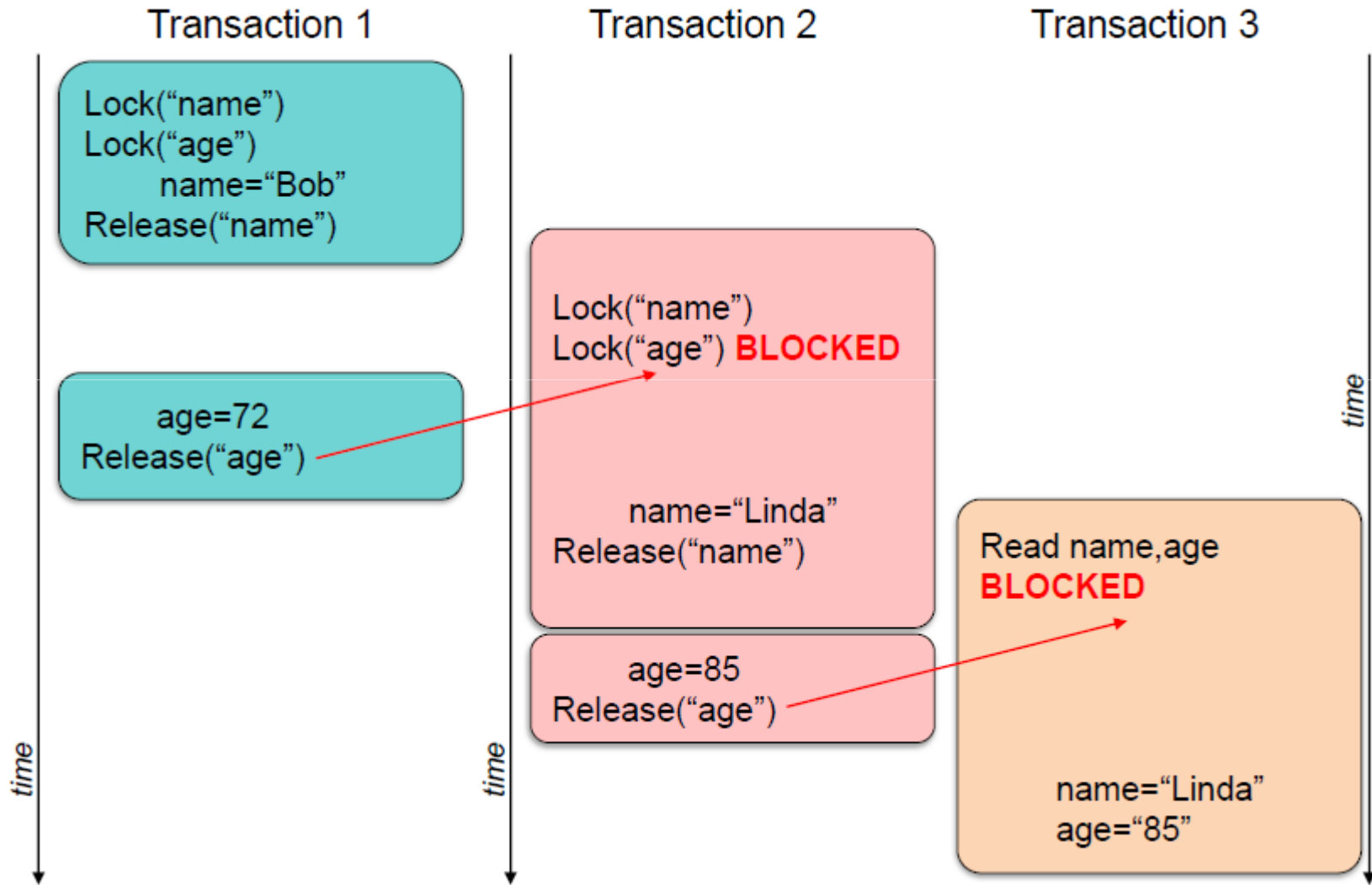
# Παράδειγμα - χωρίς 2PL



# Παράδειγμα χωρίς 2PL



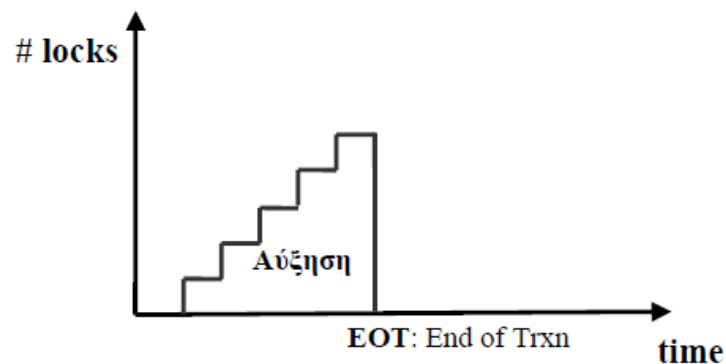
# Παράδειγμα 2PL



# Πρόβλημα με 2PL

---

- Cascading aborts!
  - Αν ένα transaction κάνει abort τότε οποιοδήποτε άλλο transactions έχει προσπελάσει δεδομένα από released locks (uncommitted data) πρέπει να κάνει κι αυτό abort
- Λύση: Αυστηρό (Strict) 2PL
  - Ένα transaction κρατάει όλα τα locks μέχρι να κάνει commit ή abort



# Μπορούμε να κάνουμε κάτι καλύτερο;

---

- Είδαμε “exclusive” locks
- Βελτιώνουμε ταυτοχρονισμό υποστηρίζοντας πολλούς αναγνώστες
  - Δεν υπάρχει πρόβλημα όταν διαβάζουν ταυτόχρονα το ίδιο αντικείμενο πολλά transactions
  - Μόνο ένα transaction πρέπει να μπορεί να γράφει ένα αντικείμενο και κανένα άλλο δεν πρέπει να μπορεί να το διαβάσει
- Δύο είδη locks: read locks και write locks
  - read lock πριν διαβάσουμε ένα αντικείμενο
    - Ένα read lock απαγορεύει το γράψιμο
  - write lock πριν γράψουμε ένα αντικείμενο
    - Ένα write lock απαγορεύει διάβασμα και γράψιμο
  - Ένα transaction μπλοκάρει αν δεν μπορεί να πάρει lock



# Non-Exclusive Locks

<i>For one object</i>		<i>Lock requested</i>	
		<i>read</i>	<i>write</i>
<i>Lock already set</i>	<i>none</i>	OK	OK
	<i>read</i>	OK	wait
	<i>write</i>	wait	wait

- Αν ένα transaction
  - Δεν έχει lock για κάποιο αντικείμενο:
    - Οποιοδήποτε άλλο transaction μπορεί να πάρει read ή write lock
  - Έχει read lock για κάποιο αντικείμενο :
    - Οποιοδήποτε άλλο transaction μπορεί να πάρει read lock αλλά να περιμένει για write lock
  - Έχει write lock για κάποιο αντικείμενο :
    - Οποιοδήποτε άλλο transaction πρέπει να περιμένει για read ή write lock

# Αναβάθμιση κλειδώματος

---

- Μέχρι τώρα υποθέταμε ότι κάθε δοσοληψία γνωρίζει εκ προοιμίου όλα τα κλειδώματα που θα ζητήσει.
- Τι γίνεται όμως αν δεν τα γνωρίζει;
- Η λύση έγκειται στην αναβάθμιση κλειδώματος (αντίστοιχα, υποβάθμιση)

# Κανόνες αναβάθμισης

---

- Αναβαθμίσεις επιτρέπονται μόνο στη φάση αύξησης
- Υποβαθμίσεις επιτρέπονται μόνο στη φάση μείωσης
- Ένα read lock αναβαθμίζεται σε write lock όταν ένα transaction θέλει να γράψει στο ίδιο αντικείμενο.
- Ένα read lock που μοιράζονται περισσότερα του ενός transactions read lock(s) δεν μπορεί να αναβαθμιστεί. Για αναβάθμιση θα πρέπει το transaction να περιμένει να απελευθερωθούν τα υπόλοιπα read locks

# Αδιέξοδο (Deadlock)

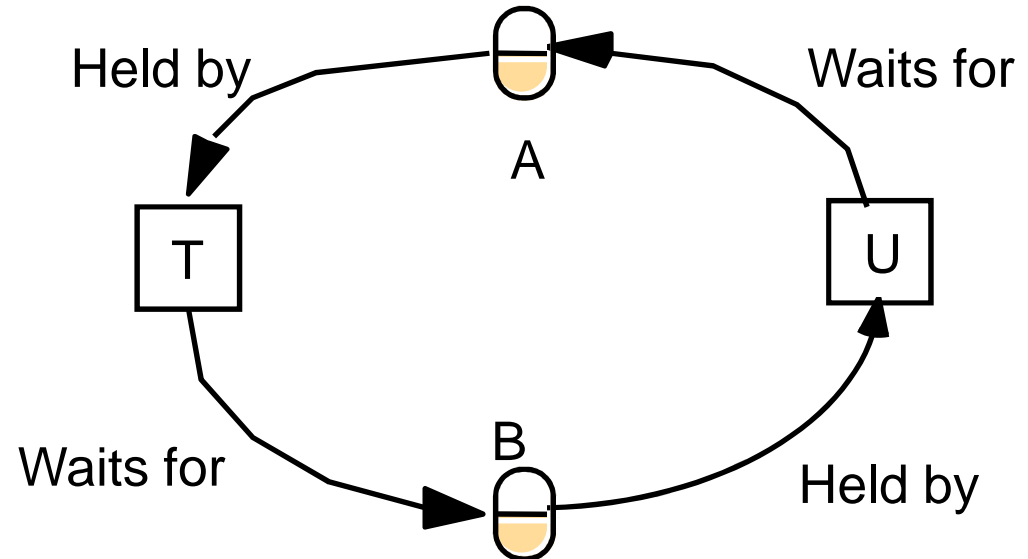
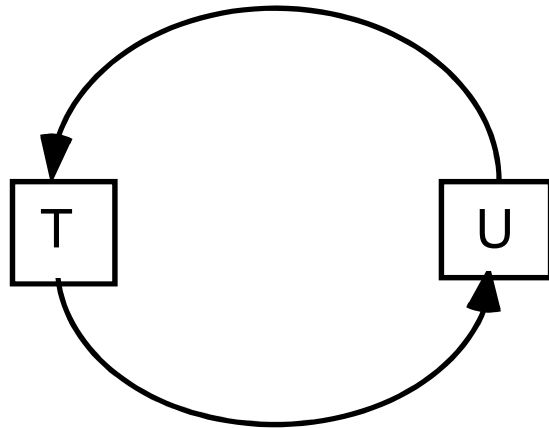
Transaction $T$		Transaction $U$	
Operations	Locks	Operations	Locks
$a.deposit(100);$	write lock $A$		
		$b.deposit(200)$	write lock $B$
$b.withdraw(100)$			
•••	waits for $U$ 's lock on $B$	$a.withdraw(200);$	waits for $T$ 's lock on $A$
•••		•••	
•••		•••	

# Deadlock

---

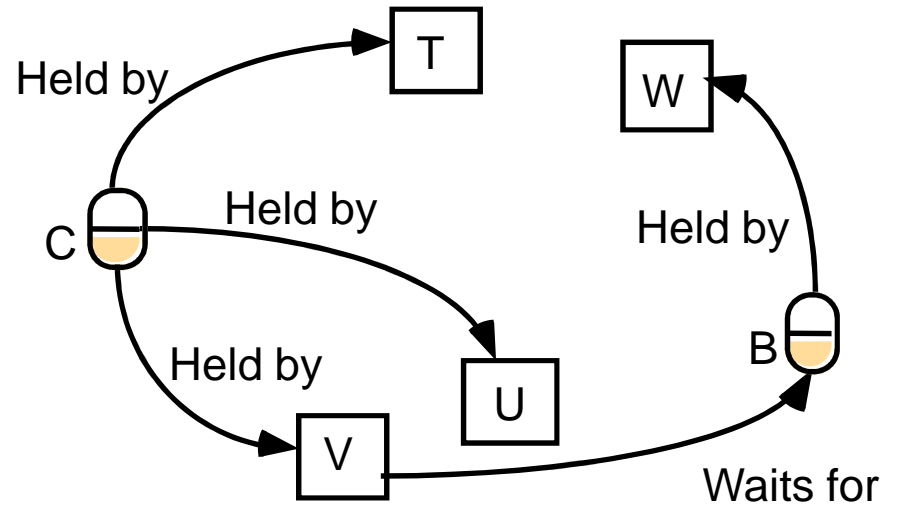
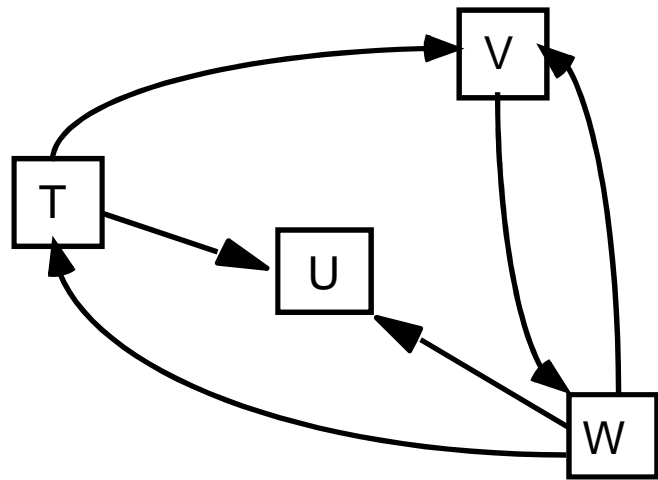
- Η κατάσταση κατά την οποία δύο δοσοληψίες  $T$  και  $T'$  αναμένουν η μία την άλλη για την απελευθέρωση κάποιων κλειδωμάτων
- Προφανώς, ο παραπάνω ορισμός γενικεύεται για περισσότερες από δύο δοσοληψίες

# Γράφος αναμονής (wait-for)



- Θεώρημα: Υπάρχει αδιέξοδος αν και μόνο αν ο γράφος αναμονής έχει κύκλο
- Στην πράξη το σύστημα ελέγχει τον γράφο αναμονής περιοδικά
- Εναλλακτικά, αντί για γράφο, μπορεί να μετρά πόση ώρα μια δοσοληψία αναμένει για ένα κλείδωμα και να την τερματίζει αν περάσει κάποιο όριο...

# Γράφος wait-for



# Ανίχνευση ή αποτροπή;

---

- Αντί να περιμένουμε να συμβεί το αδιέξοδο, μπορούμε να το αποτρέψουμε προληπτικά
- Συντηρητικός 2PL: μια δοσοληψία αποκτά όλα τα κλειδιά που χρειάζεται στο ξεκίνημα της.
- Αν δεν μπορεί να τα πάρει ΟΛΑ, δεν ξεκινά, αλλά αναμένει!
- Εν γένει, προτιμώνται τα αδιέξοδα...



# Αυξάνοντας τον ταυτοχρονισμό

---

- 2-version locking
- Ένα transaction μπορεί να γράφει προσωρινές εκδοχές ενός αντικειμένου
  - Τα υπόλοιπα διαβάζουν από την αρχική εκδοχή
- Τα reads περιμένουν αν άλλα transactions κάνουν commit το ίδιο αντικείμενο

# 2 version locking

---

- 3 τύποι locks: read lock, write lock, commit lock
  - Ένα transaction δεν μπορεί να πάρει read ή write lock αν υπάρχει commit lock
- Όταν ο transaction coordinator πάρει αίτημα για commit
  - Μετατρέπει όλα τα write locks ενός transaction σε commit locks
  - Αν οποιοδήποτε αντικείμενο έχει read locks, το transaction πρέπει να περιμένει μέχρι τα υπόλοιπα transaction που έχουν τα read locks να τα απελευθερώσουν
- Σύγκριση με read/write locks:
  - Τα reads καθυστερούν μόνο όταν transactions γίνονται commit
  - Το read ενός transaction μπορεί να καθυστερήσει το commit ενός άλλου

# Προβλήματα με locks

---

- Οι τεχνικές κλειδώματος είναι συντηρητικές (αποφεύγονται οι συγκρούσεις)
- Επιβάρυνση (overhead) χειρισμού κλειδώματος
- Έλεγχος προς ανίχνευση αδιεξόδων

# Αισιόδοξη εκτέλεση δοσοληψιών

---

- Σε πολλές εφαρμογές η πιθανότητα 2 transactions να προσπελάζουν το ίδιο αντικείμενο είναι μικρή
- Επιτρέπουμε στα transactions να δρουν χωρίς locks
- Τσεκάρουμε για συγκρούσεις (conflicts)
- 3 φάσεις
  - **Read phase:** (ή working phase) η δοσοληψία διαβάζει, αλλά τροποποιεί τοπικά αντίγραφα των αντικειμένων σε δικό της χώρο
  - **Validation phase:** έλεγχος για συγκρούσεις – αν υπάρχει σύγκρουση, abort & restart
  - **Write phase:** (ή update phase) γράφει τα τοπικά αντίγραφα στο δίσκο



# Πώς δουλεύει;

---

- Timestamp δοσοληψίας: η στιγμή που ξεκινά η φάση επικύρωσής της
- Έστω ότι έχουμε μια δοσοληψία  $T_j$  που θέλει να κάνει Επικύρωση
  - Θα την ελέγξουμε σε σχέση με όλες τις δοσοληψίες  $T_i$  με timestamp μικρότερο της  $T_j$ .
  - Για όλες τις δοσοληψίες  $T_i$ , τ.ω.  $T_i < T_j$ , αρκεί να ισχύει μία από τις 3 παρακάτω συνθήκες ελέγχου
  - Αν δεν ισχύει καμία εξ αυτών, η  $T_j$  πρέπει να κάνει ABORT και να επανεκκινηθεί.

# 1<sup>η</sup> συνθήκη ελέγχου

---

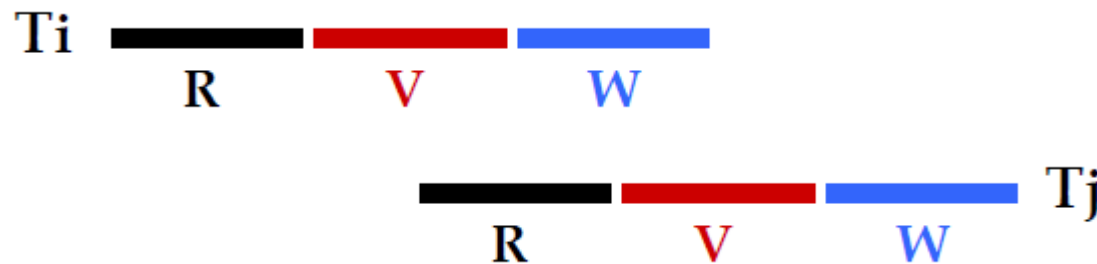
- Η  $T_i$  τελειώνει πριν αρχίσει η  $T_j$ 
  - Σειριακή εκτέλεση των δοσοληψιών



## 2<sup>η</sup> συνθήκη ελέγχου

---

- Η  $T_i$  τελειώνει πριν αρχίσει η φάση εγγραφής της  $T_j$
- $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \emptyset$

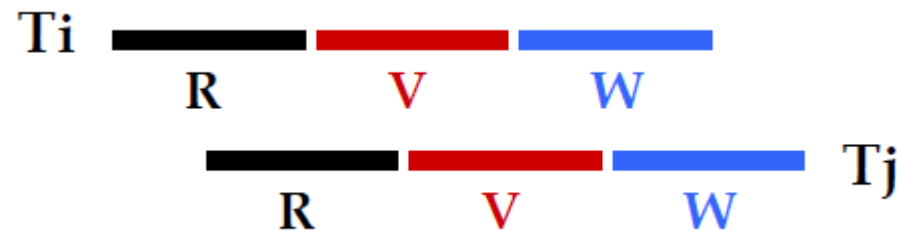


- *Η  $T_j$  δεν διαβάζει κανένα αντικείμενο που τροποποιεί η  $T_i$*
- *Όλες οι εγγραφές της  $T_i$  προηγούνται αυτών της  $T_j$*

# 3<sup>η</sup> συνθήκη ελέγχου

---

- Η  $T_i$  τελειώνει τη φάση ανάγνωσης πριν τελειώσει η φάση ανάγνωσης της  $T_j$
- $WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$
- $WriteSet(T_i) \cap WriteSet(T_j) = \emptyset$



- *Η  $T_j$  και η  $T_i$  γράφουν αντικείμενα ταυτόχρονα, μεν, ... αλλά, οι εγγραφές τους ΔΕΝ επικαλύπτονται!*



# Προσέξτε ότι...

---

- ΠΕΡΙΟΡΙΣΜΟΣ: Όταν μια δοσοληψία βρίσκεται σε φάση επικύρωσης, καμιά άλλη δοσοληψία δεν μπορεί να κάνει COMMIT!!!
- Ο αισιόδοξος έλεγχος ταυτοχρονισμού, ουσιαστικά υλοποιεί το γράφο σειριοποιησιμότητας (πρακτικά ελέγχει αν οι συγκρούσεις γίνονται με την ίδια σειρά)

# Σχόλια

---

- Overhead διατήρησης λίστας προσπελασθέντων αντικειμένων
- Αν οι συγκρούσεις είναι σπάνιες έχουμε μεγαλύτερη απόδοση.
- Αν οι συγκρούσεις είναι συχνές, το overhead του abort & restart είναι μεγαλύτερο.

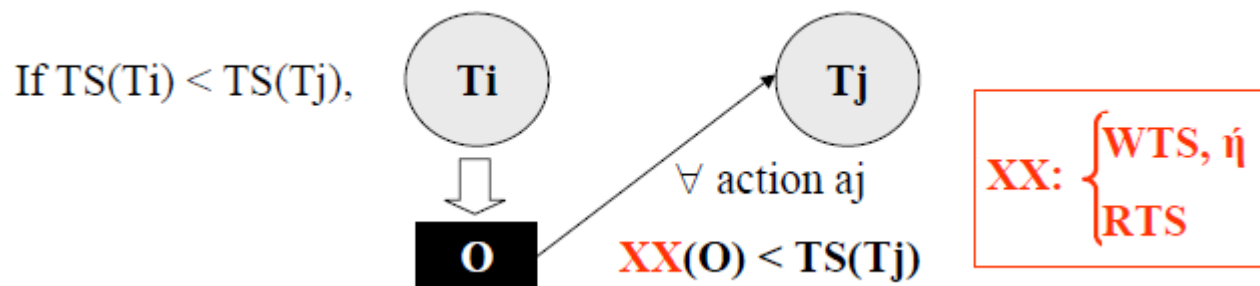
# Διάταξη χρονοσφραγίδων

---

- timestamp δοσοληψίας  $TS(T)$ : η χρονική στιγμή που ξεκινά η δοσοληψία  $T$
- timestamp ανάγνωσης αντικειμένου  $RTS(O)$ : η χρονική στιγμή της τελευταίας ανάγνωσης του αντικειμένου  $O$  (ασχέτως δοσοληψίας)
- timestamp εγγραφής αντικειμένου  $WTS(O)$ : η χρονική στιγμή της τελευταίας εγγραφής του αντικειμένου  $O$  (ασχέτως δοσοληψίας)
- Αντί να συγκρίνω δοσοληψίες, θεωρώ ότι η  $T_i$  διαβάζει/γράφει το αντικείμενο  $O$
- Κάθε φορά, μια δοσοληψία  $T_j$  συγκρίνει το  $TS(T_j)$  με τα timestamps των αντικειμένων

# Κανόνες

- Σωστή διάταξη:
  - Τα *read* και *write timestamps* του αντικειμένου πρέπει να είναι παλιότερα του *timestamp* του τρέχοντος *transaction* αν αυτό θέλει να γράψει
  - Τα *write timestamps* του αντικειμένου πρέπει να είναι παλιότερα του *timestamp* του τρέχοντος *transaction* αν αυτό θέλει να διαβάσει
- Abort και restart transaction με μεγαλύτερο timestamp αν η σειρά είναι λάθος



# Πραγματικά Παραδείγματα

---

- Dropbox
  - Optimistic concurrency control
  - File granularity
  - Version history
- Google Docs
  - Character granularity
  - Conflict resolution (rare)
- Wikipedia
  - Optimistic concurrency control

# Ανακεφαλαίωση

---

- Απλές δοσοληψίες (transactions)
- Ιδιότητες ACID
  - Και κυρίως atomicity και durability
- Σειριοποιησιμότητα (serializability)
- Έλεγχος ταυτοχρονισμού
  - Κλειδώματα (Locking)
  - Αισιόδοξος έλεγχος ταυτοχρονισμού (optimistic concurrency control)
  - Διάταξη χρονοσφραγίδων (Timestamp ordering)
- Αδιέξοδα (deadlocks)

\*Χρησιμοποιήθηκε υλικό από τις διαφάνειες του Π. Βασιλειάδη