Parallel Architectures

Memory Consistency
+
Synchronization

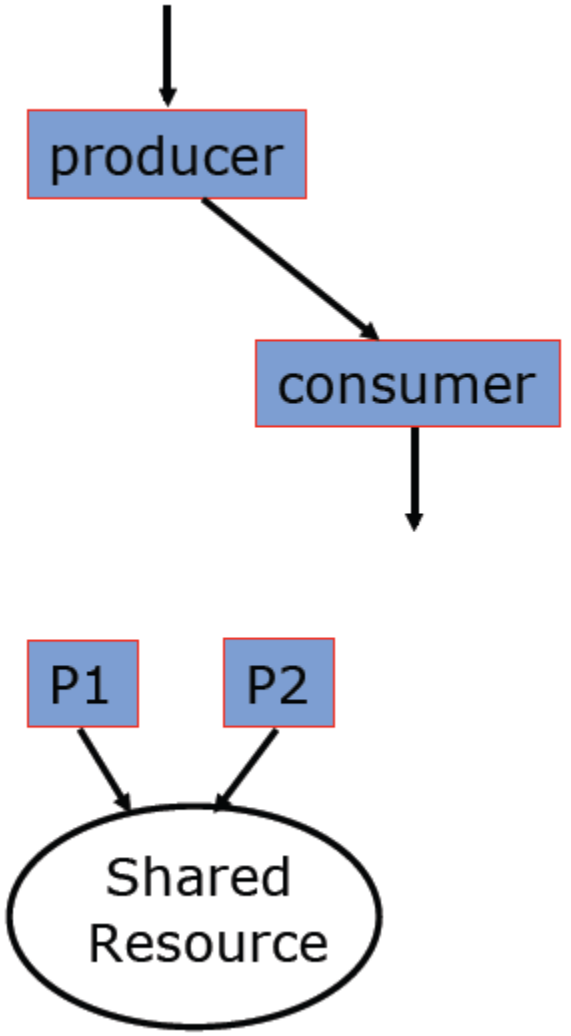
- **Figures, examples από**
  1. **Transactional Memory, D. Wood, Lecture Notes in ACACES 2009**
  2. **Krste Asanović's Lecture Notes, University of California, Berkeley.**

Παράλληλες Αρχιτεκτονικές και Εφαρμογές

- Παράλληλες αρχιτεκτονικές (CMPs, SMPs, SMTs κτλ)
- Ιδανικές για παράλληλες εφαρμογές
  - Η πλειοψηφία των εφαρμογών δημιουργεί πολλαπλά threads που δουλεύουν κάτω από κοινή μνήμη
- Κίνδυνοι με κοινή μνήμη σε παράλληλες αρχιτεκτονικές:
  - Ύπαρξη πολλαπλών αντιγράφων σε διαφορετικά επίπεδα της ιεραρχίας μνήμης → Coherence protocols = SOLVED!
  - Ανάγκη συγχρονισμού → Μεταφορά μοντέλων από uniprocessor systems

Συγχρονισμός

- Η ανάγκη για συγχρονισμό προκύπτει όποτε υπάρχουν ταυτόχρονες διεργασίες σε ένα σύστημα *(ακόμα και σε uniprocessor σύστημα)*
- ***Μοντέλο παραγωγού – καταναλωτή:*** ο καταναλωτής θα πρέπει να περιμένει μέχρι ο παραγωγός να παράξει δεδομένα
- ***Αμοιβαίος αποκλεισμός:*** εξασφαλίζει ότι μία μόνο διεργασία μπορεί να χρησιμοποιήσει έναν κοινό πόρο σε μια δεδομένη στιγμή



Παράδειγμα 1

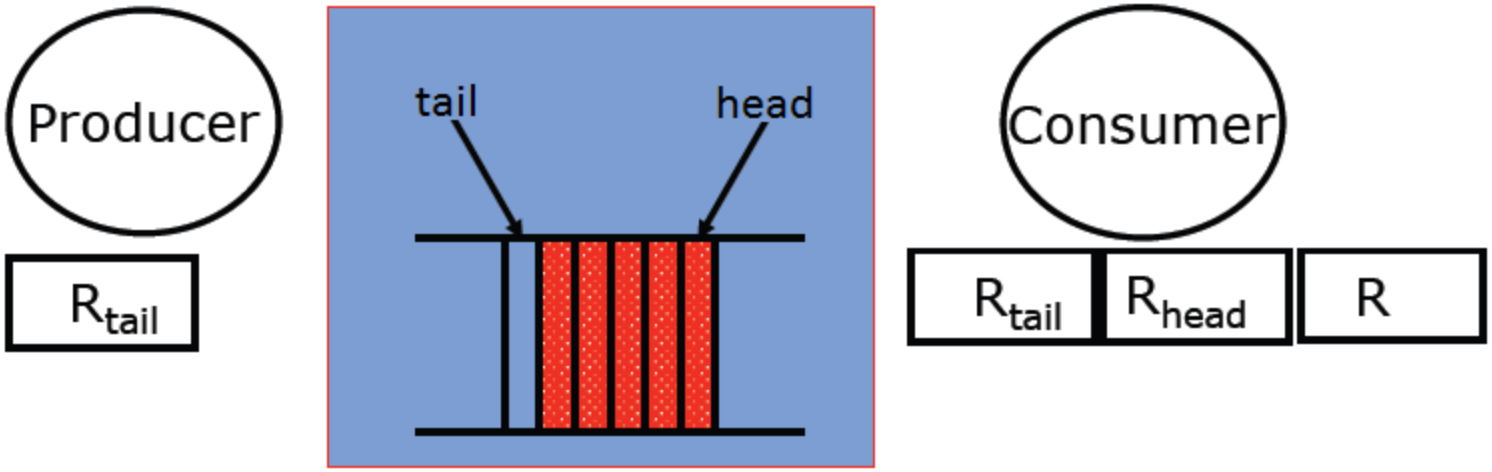
```
                    A = flag = 0;

Processor 0                 Processor 1

A = 1;                      while (!flag); //spin
flag = 1;                   print A;
```

- Τι μας λέει η *διαίσθηση*: ο P1 τυπώνει A=1
- Τι μας εγγυάται η *συνάφεια* μνήμης: απολύτως τίποτα!
  - απλά μας εξασφαλίζει ότι η καινούρια τιμή του A *κάποια στιγμή* θα γίνει ορατή από τον P1
  - ο P1 μπορεί να δει την εγγραφή του flag *πριν* την εγγραφή στο A ! Πώς;
    - » τα μηνύματα του coherence protocol για την εγγραφή του A μπορεί να καθυστερήσουν κάπου στο δίκτυο διασύνδεσης
    - » ο write buffer του P0 μπορεί να αναδιατάσσει τις εγγραφές
- Σε πραγματικά συστήματα, ο παραπάνω κώδικας μπορεί να «δουλεύει» μερικές φορές, ενώ άλλες όχι…

Παράδειγμα 2



Producer posting Item x:

```
Load R_tail,(tail)
Store (R_tail),x
R_tail=R_tail+1
Store (tail),R_tail
```

Το πρόγραμμα είναι γραμμένο με την υπόθεση ότι οι εντολές εκτελούνται σε σειρά.

Consumer:

```
        Load R_head,(head)
spin:   Load R_tail,(tail)
        if R_head==R_tail goto spin
        Load R,(R_head)
        R_head=R_head+1
        Store (head),R_head
        consume(R)
```

Παράδειγμα 2 (2)

Producer posting Item x:

```
   Load R_tail,(tail)
1  Store (R_tail),x
   R_tail=R_tail+1
2  Store (tail),R_tail
```

Ο tail pointer μπορεί να ανανεωθεί πριν την εγγραφή του x!

Consumer:

```
   Load R_head,(head)
spin: Load R_tail,(tail) 3
   if R_head==R_tail goto spin
   Load R,(R_head) 4
   R_head=R_head+1
   Store (head),R_head
   consume(R)
```

- Ο προγραμματιστής υποθέτει ότι αν η 3 πραγματοποιηθεί μετά τη 2, τότε η 4 πραγματοποιείται μετά την 1.
- Προβληματικές ακολουθίες:
  - 2, 3, 4, 1
  - 4, 1, 2, 3

- Χρειαζόμαστε ένα μοντέλο σειριοποίησης για λειτουργίες μνήμης :
  - Σε ίδιες ή διαφορετικές θέσεις μνήμης
  - Από ένα ή πολλαπλά threads/processes

➢ Μοντέλο συνέπειας μνήμης

Memory Consistency

- Συνάφεια μνήμης (coherence):
  - διασφαλίζει ότι η τιμή της τελευταίας εγγραφής σε μια θέση μνήμης θα γνωστοποιηθεί σε όλους τους τυχόν αναγνώστες
  - δημιουργεί μια καθολικά ενιαία εικόνα για *μία* συγκεκριμένη θέση μνήμης (cache line, πρακτικά)
  - δεν αρκεί:
    - » 2 cache lines A και B μπορεί να είναι μεμονωμένα συνεπείς, αλλά ασυνεπείς σε σχέση με τη σειρά που τροποποιήθηκαν στο πρόγραμμα

- Συνέπεια μνήμης (consistency):
  - καθορίζει το *πότε* θα γίνεται ορατή μια εγγραφή
  - δημιουργεί μια καθολικά ενιαία εικόνα για *όλες* τις θέσεις μνήμης, όσον αφορά τις μεταξύ τους τροποποιήσεις

- Απαραίτητη για τη σωστή λειτουργία μηχανισμών συγχρονισμού

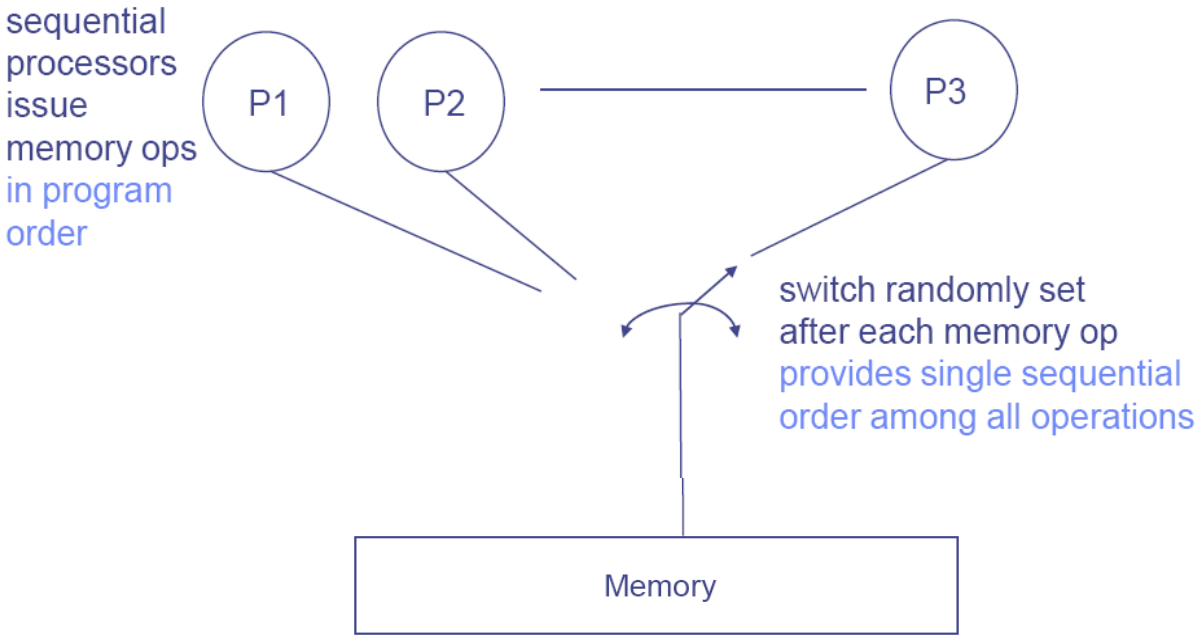
Μοντέλο Συνέπειας Μνήμης

- Περιορίζει τις πιθανές διατάξεις με τις οποίες οι λειτουργίες μνήμης μπορούν να εμφανιστούν η μια σε σχέση με την άλλη.

- Χωρίς αυτή δεν μπορούμε να πούμε τίποτα για το αποτέλεσμα της εκτέλεσης ενός προγράμματος.

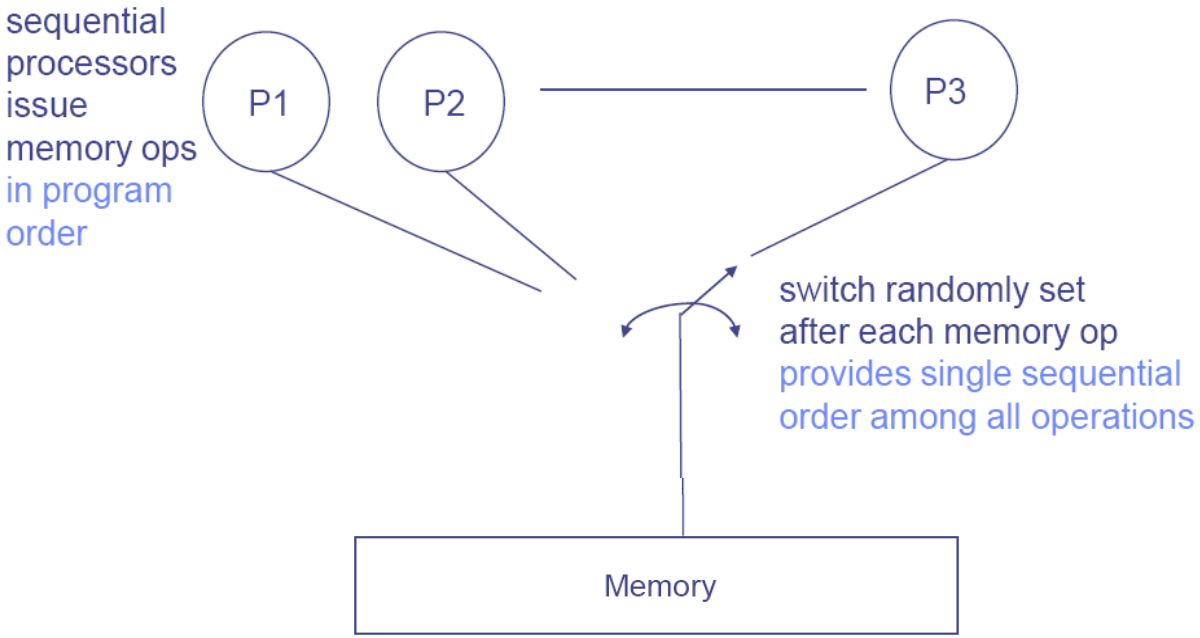
- Συνέπειες :
  - Ο *προγραμματιστής* αποφασίζει για την ορθότητα και τα πιθανά αποτελέσματα
  - Ο *σχεδιαστής του συστήματος* περιορίζει πόσο μπορούν να αναδιατάσσονται οι λειτουργίες μνήμης από τον compiler ή το hardware.

Sequential Consistency



- *"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program."* [Lamport, 1979]
- SC = τυχαία μίξη των (εν σειρά) αναφορών των σειριακών προγραμμάτων στους επεξεργαστές

Sequential Consistency (2)



- Σαν να μην υπήρχαν κρυφές μνήμες, παρά μόνο μια μνήμη.
- Κάθε επεξεργαστής δρομολογεί και ολοκληρώνει μια λειτουργία μνήμης σύμφωνα με τη σειρά του προγράμματος
- Η μνήμη ικανοποιεί τις προσπελάσεις σύμφωνα με *κάποια* σειρά.
- Κάθε λειτουργία σε αυτή τη σειρά φαίνεται σα να εκτελείται και να ολοκληρώνεται ατομικά (πριν ξεκινήσουν οι επόμενες).

Sequential Consistency (3)

- Ορισμός *program order*
  - Διαισθητικά η σειρά με την οποία εμφανίζονται οι εντολές στον πηγαίο κώδικα.
  - Η σειρά με την οποία εμφανίζονται οι λειτουργίες μνήμης στην assembly που προκύπτει από απευθείας μετατροπή του πηγαίου κώδικα.
- *Δεν* είναι υποχρεωτικά η ίδια σειρά με αυτή που παράγει ο compiler και εκτελείται στο hardware
  - Ένας optimizing compiler μπορεί να αναδιατάξει τις εντολές του πηγαίου κώδικα.
- Άρα η σειρά του προγράμματος εξαρτάται από το επίπεδο που κοιτάζουμε.
  - Εμείς υποθέτουμε τη σειρά έτσι όπως τη βλέπει ο προγραμματιστής.

Sequential Consistency (4)

```
P1                              P2
/* Assume initial values of A and B are 0 */
(1a) A = 1;                     (2a) print B;
(1b) B = 2;                     (2b) print A;
```

- Παράδειγμα
- Πιθανά αποτελέσματα για (A,B) :
  - (0,0), (1,0), (1,2) √
- Επιτρέπει η SC το (0,2);
  - Σύμφωνα με SC πρέπει 1a → 1b και 2a → 2b (program order).
  - Αν A = 0, τότε 2b → 1a.
  - Άρα και 2a → 1a.
  - Όμως B = 2, μόνο αν 1b → 2a !
  - Επομένως μη επιτρεπτό αποτέλεσμα.

Sequential Consistency (5)

- Ανακεφαλαιώνοντας, για την SC έχουμε 2 απαιτήσεις.

- Program Order
  - Οι λειτουργίες μνήμης ενός thread πρέπει να γίνονται ορατές (στους άλλους και στον εαυτό του) με τη σειρά που υπαγορεύει το πρόγραμμα.

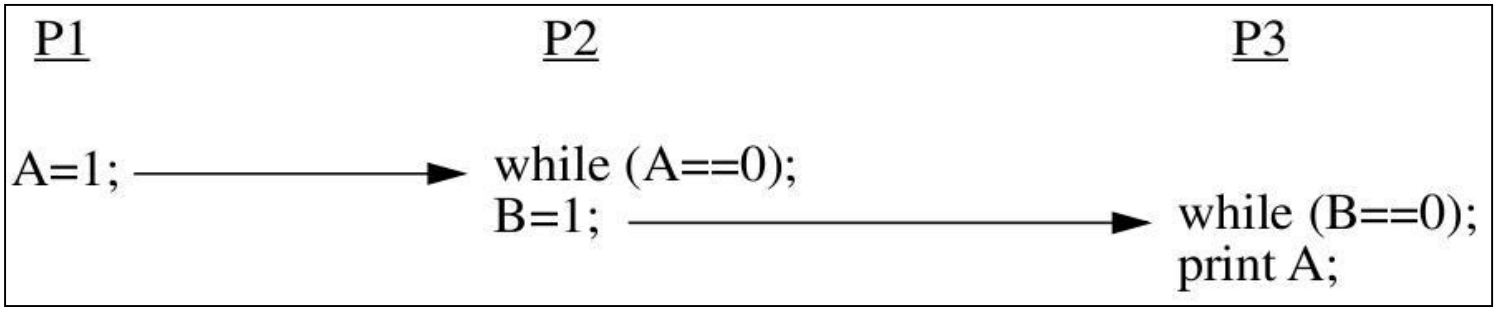
- Atomicity
  - Μια λειτουργία μνήμης ολοκληρώνεται προτού κληθεί η επόμενη σύμφωνα με την καθολική σειρά (ανεξάρτητα από το σε ποιο thead ανήκει η επόμενη λειτουργία).

Write atomicity (1)

- Write atomicity
  - Η θέση όπου εμφανίζεται να εκτελείται ένα write σύμφωνα με την καθολική σειρά, πρέπει να είναι ίδια για όλα τα threads.

- Τίποτα από αυτά που κάνει ένα thread αφού δει την καινούρια τιμή που παράγει μια εγγραφή W δεν πρέπει να γίνει ορατό από τα υπόλοιπα threads πριν δουν και αυτά τη W.
- Ουσιαστικά, επεκτείνουμε την σειριοποίηση εγγραφών που απαιτεί η συνάφεια.
  - Write serialization : Όλες οι εγγραφές σε μια τοποθεσία θα πρέπει να εμφανίζονται σε όλα τα threads με την ίδια σειρά
  - Write atomicity : Όλες οι εγγραφές σε κάθε τοποθεσία θα πρέπει να εμφανίζονται σε όλα τα threads με την ίδια σειρά

Write atomicity (2)



- Έστω ότι επιτρέπουμε στον P2 να προχωρήσει στο B=1 πριν η εγγραφή του A γίνει ορατή από τον P3.
- Μπορεί ο P3 να διαβάσει την παλιά τιμή του A και την καινούρια του B;
- Παραβίαση της SC!

Προβλήματα SC: Hardware

- Πολύ λίγα συστήματα υλοποιούν SC
  - Ούτε οι x86 ούτε οι ARM
- Μειωμένη απόδοση
  - Αναμονή για ολοκλήρωση ενός store πριν την δρομολόγηση του επόμενου
- Πολύπλοκο hardware
  - π.χ. MIPS R10K: “speculatively issue loads but squash if memory inconsistency with later-issued store is discovered”

Προβλήματα SC: Software

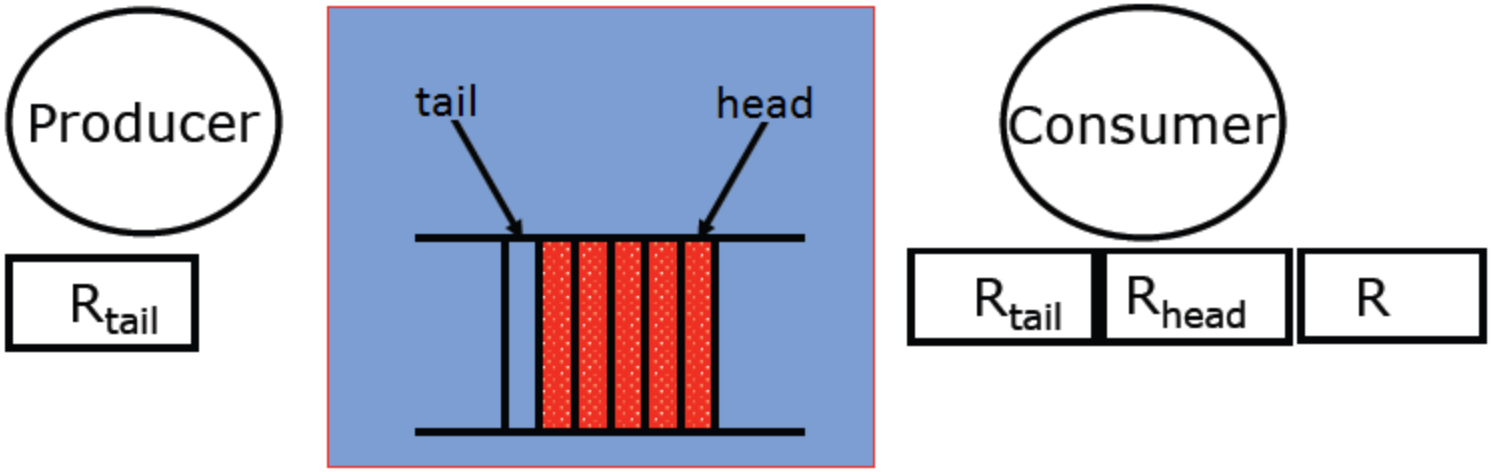
- Ο compiler μπορεί να **αναδιατάξει** εντολές πρόσβασης στη μνήμη
  - Instruction scheduling: Μεταφορά ενός Load πριν από Store αν πρόκειται για διαφορετικές διευθύνσεις
- Ο compiler μπορεί να **διαγράψει** εντολές πρόσβασης στη μνήμη
  - Register allocation: Αν η τιμή είναι cached σε ένα καταχωρητή δεν χρειάζεται να ελεγθεί η μνήμη
- Τα compiler optimizations είναι αναγκαία για λόγους απόδοσης!
- Λύση → **Relaxed Memory Models**

Relaxed Memory Models

- Η SC είναι πολύ αυστηρή → Τα relaxed models δεν υποστηρίζουν όλες τις απαιτήσεις της SC.

- Ο προγραμματιστής πρέπει να εισάγει στον κώδικα του ότι απαιτήσεις/εξαρτήσεις χρειάζονται και δεν υποστηρίζονται από το μοντέλο μνήμης.
  - Fence Instructions (sync or memory barriers)
  - Atomic memory instructions

Παράδειγμα : Relaxed Consistency με Fences



Producer posting Item x:

```
Load R_tail,(tail)
Store (R_tail),x
Fence_ss
R_tail=R_tail+1
Store (tail),R_tail
```

εγγυάται ότι ο tail pointer δε θα ανανεωθεί πριν την εγγραφή του x

Consumer:

```
Load R_head,(head)
spin: Load R_tail,(tail)
if R_head==R_tail goto spin
Fence_LL
Load R,(R_head)
R_head=R_head+1
Store (head),R_head
consume(R)
```

εγγυάται ότι ο R δε θα φορτωθεί πριν την εγγραφή του x

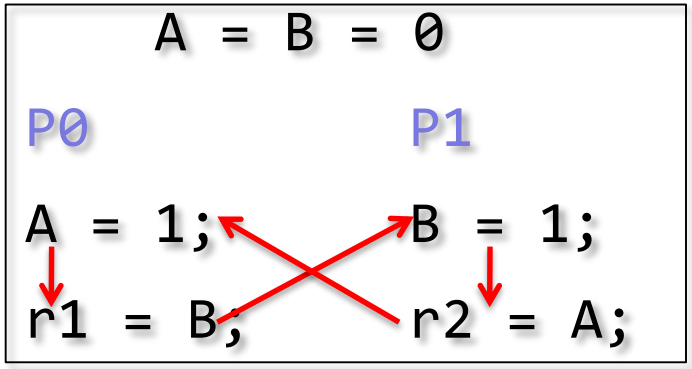
Relaxed Consistency Models (2)

- Μπορούμε να οργανώσουμε τα μοντέλα σε 3 διαφορετικές κατηγορίες

1. Επιτρέπεται σε ένα Load να ολοκληρωθεί προτού ολοκληρωθεί κάποιο προηγούμενο (σύμφωνα με το program order) Store.
   - Total Store Order (TSO), Processor Consistency (PC) (e.g., IBM 370, Sparc TSO, Intel IA-32)
2. Επιτρέπεται και σε ένα Store να προσπεράσει κάποιο προηγούμενο (σύμφωνα με το program order) Store.
   - Partial Store Order (PSO) (e.g., Sparc PSO)
3. Επιτρέπεται σε ένα Load ή ένα Store να προσπεράσει κάποια προηγούμενη (σύμφωνα με το program order) λειτουργία μνήμης, είτε Load είτε Store.
   - Weak Ordering (WO), Release Consistency (RC), Relaxed Memory Order (RMO) (e.g., Sparc RMO, Alpha, PowerPC)

Relaxing Write-to-Read Order

- Αντιμετώπιση της καθυστέρησης για writes που κάνουν miss στην L1 cache.
- Όσο το write είναι στο write buffer, ο επεξεργαστής ολοκληρώνει reads που κάνουν hit στην cache.
- Καταργείται η συνθήκη της SC ότι όλες οι αναφορές πραγματοποιούνται στη σειρά προγράμματος



- επιτρέπει r1 == r2 == 0 (δεν το επιτρέπει η SC)

Relaxing Write-to-Read & Write-to-Write Order

- Κίνητρο : Περαιτέρω μείωση της καθυστέρησης εξαιτίας ενός write miss και βελτίωση της επικοινωνίας μεταξύ των επεξεργαστών κάνοντας ορατές τις καινούριες τιμές νωρίτερα.
- Επιτρέπεται το merging πολλαπλών writes που βρίσκονται στο write buffer.
- Τα writes μπορεί να γίνουν ορατά εκτός σειράς!

Γιατί να μην χαλαρώσουμε όλους τους περιορισμούς;

```
/* initially all 0 */
  P1                          P2
A = 1;                  while (flag == 0); /* spin */
B = 1;                        r1 = A;
flag = 1;                     r2 = B;
```

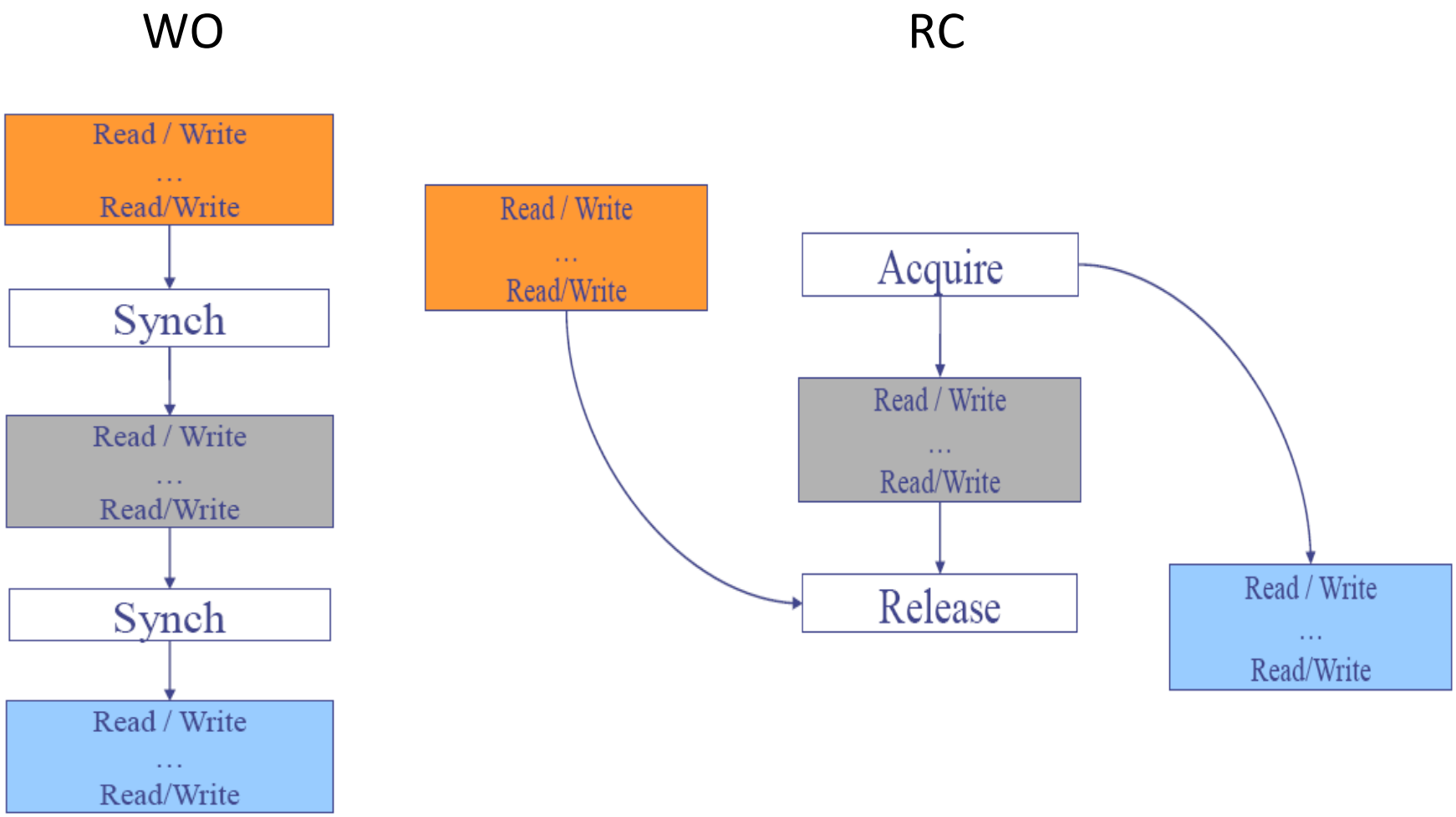
- OK αν μπορούμε να αναδιατάξουμε τα ζεύγη “A=1/B=1” ή “r1=A/r2=B”
  - μέσω OoO επεξεργαστών, non-FIFO write buffers, καθυστερήσεις στο δίκτυο διασύνδεσης, κ.λπ.
- Όμως ο προγραμματιστής στηρίζεται στα εξής για την ορθότητα του προγράμματος:
  - “A=1/B=1” προτού “flag=1”
  - “flag!=0” προτού “r1=A/r2=B”

Relaxing All Memory Orders

```
/* initially all 0 */
  P1                        P2
A = 1;                      while (SYNCH  flag == 0);
B = 1;                      r1 = A;
SYNCH flag = 1;             r2 = B;
```

- Κίνητρο : Όταν μας ενδιαφέρει η σειρά των λειτουργιών, τα παράλληλα προγράμματα χρησιμοποιούν συγχρονισμό. Επομένως, μπορούμε χαλαρώσουμε όλους τους περιορισμούς για τις λειτουργίες πριν και μετά τις λειτουργίες συγχρονισμού.
- Weak Ordering (WO) : Ξεχωρίζει μεταξύ “κανονικών” λειτουργιών και λειτουργιών συγχρονισμού.
- Release Consistency (RC) : Επεκτείνει το WO, διακρίνοντας τις λειτουργίες συγχρονισμού σε *acquire* και *release* (επιτρέποντας καλύτερη επικάλυψη των λειτουργιών)

Weak Ordering vs Release Consistency



Παράδειγμα: Sparc V9 memory fences

- `#LoadLoad`
- `#StoreLoad`
- `#LoadStore`
- `#StoreStore`
- Logical or-ed combinations possible
- #XY = "***All X operations*** that appear before the memory fence in program order ***complete before any Y operations*** that follow after the memory fence in program order."
- (+) Ευελιξία όσον αφορά την βέλτιστη εκμετάλλευση του εκάστοτε relaxed consistency model για μέγιστη απόδοση
- (-) Προγραμματιστικά δύσκολη + ζητήματα μεταφερσιμότητας ανάμεσα σε διαφορετικά models

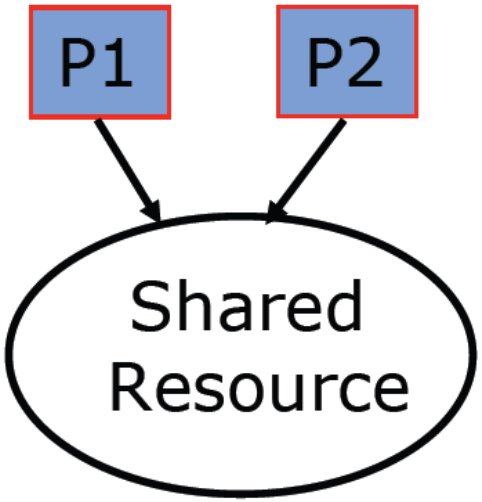
Relaxed Consistency Models (3)

- Hardware optimizations
  - Η χρήση buffers επιτρέπει στον επεξεργαστή να μη κάνει stall εφόσον οι τοπικές εξαρτήσεις δεδομένων διατηρούνται.
  - Το hardware πρέπει να μπορεί να ξεχωρίσει ποιες προσβάσεις στη μνήμη γίνονται για συγχρονισμό και ποιες όχι.
- Software optimizations
  - Αναδιάταξη των εντολών μεταξύ των σημείων συγχρονισμού.
  - Επιτρέπονται compiler optimizations όπως register allocation.
  - Το πρόγραμμα πρέπει να περιέχει τα κατάλληλα annotations και τυχόν memory races να επιλύονται με χρήση συγχρονισμού.
- Ποιο είναι το πρόβλημα με την RC;
  - Τα παράλληλα προγράμματα θα έχουν απροσδιόριστη συμπεριφορά αν εκτελεστούν σε ένα consistency model διαφορετικό από αυτό για το οποίο έχουν γραφτεί

Consistency Models

- Sequential Consistency
  - Οι λειτουργίες μνήμης πρέπει να ολοκληρώνονται με βάση τη σειρά του προγράμματος.
  - Η καθυστέρηση μπορεί να μειωθεί ίσως με χρήση speculation.
  - Δεν επιτρέπονται τα περισσότερα compiler optimizations
  - x86, ARM : not SC.
- TSO & PC
  - Επιτρέπεται η χρήση write buffers.
  - Δεν επιτρέπονται και πάλι τα περισσότερα compiler optimizations.
- WO & RC
  - Επιτρέπεται η χρήση read και write buffers.
  - Επιτρέπονται compiler optimizations μεταξύ των σημείων συγχρονισμού.
  - Το πρόγραμμα πρέπει να περιέχει τα κατάλληλα annotations για να εκτελεστεί σωστά.
  - Η RC έχει υιοθετηθεί στο memory model των C/C++ και Java. Αν όλα τα data accesses προστατεύονται από συγχρονισμό συμπερίφέρεται σαν SC.

Η ανάγκη για συγχρονισμό

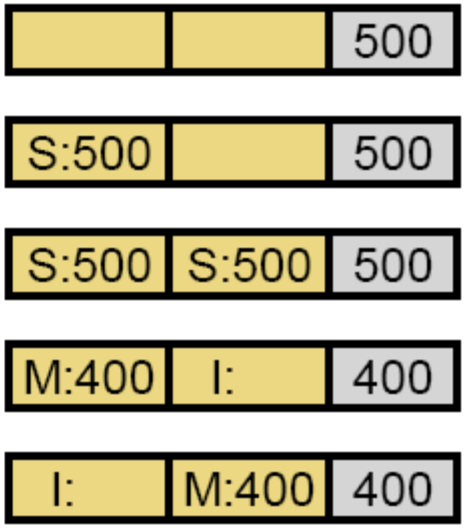


- Αρκούν τα coherence protocols και τα consistency models για να μας εξασφαλίσουν σωστή σημασιολογία στην παράλληλη εκτέλεση ενός προγράμματος;

Η ανάγκη για συγχρονισμό (2)

```
Processor 0
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

```
Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```



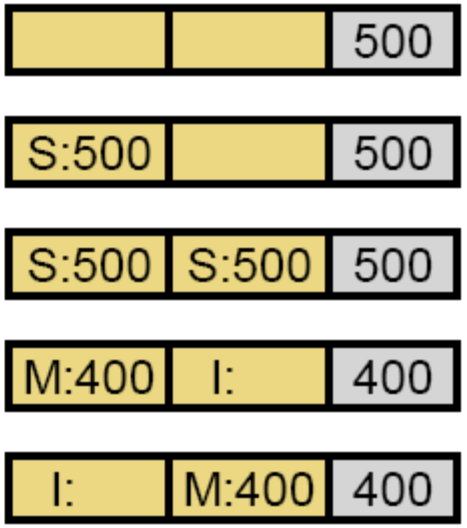
- Υπόθεση: write-back caches (θα μπορούσε όμως και write-through…) + MSI protocol

- Τι συνέβη;
  - *…πάντως το coherence protocol λειτούργησε σωστά*

Η ανάγκη για συγχρονισμό (3)

```
Processor 0
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

```
Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

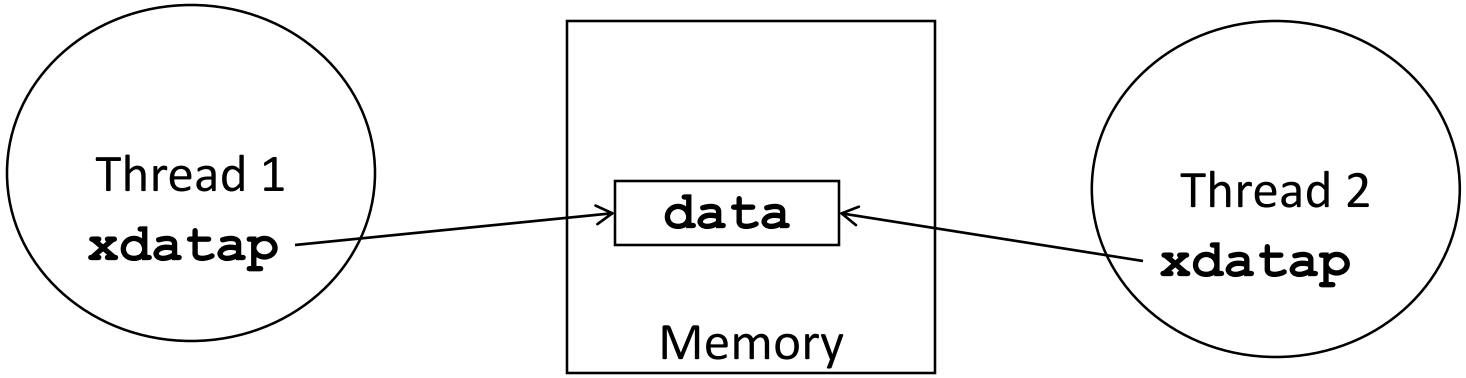


- Τι συνέβη στην πραγματικότητα;
  - διαισθητικά, η πρόσβαση (=read + modify + write) στον accts[241].bal θα πρέπει να γίνεται **ατομικά**
    - » οι δοσοληψίες δε θα πρέπει να επικαλύπτονται μεταξύ τους
    - » …όμως αυτό ακριβώς έγινε!
  - λύση: **συγχρονισμός** προσβάσεων στον accts[241].bal

Συγχρονισμός

- Ρυθμίζει την πρόσβαση σε μοιραζόμενα δεδομένα
- **Κρίσιμη περιοχή** (critical section): όλες οι λειτουργίες εντός αυτής θα πρέπει να γίνονται *ατομικά*, σαν μία ενιαία και αδιαίρετη λειτουργία
- Πώς;
  - αμοιβαίος αποκλεισμός
    - » locks, semaphores, monitors
  - atomic instructions
  - non-blocking μηχανισμοί

Παράδειγμα : Αμοιβαίος Αποκλεισμός




```
ld  xdata, (xdatap)
add xdata, 1
sd  xdata, (xdatap)
```

Τι χρειάζεται για να εκτελεστεί σωστά ο κώδικας;

Αμοιβαίος Αποκλεισμός με Load/Store (1)

- Χρήση 2 διαμοιραζόμενων μεταβλητών.

```
    c1 = 1;
L:  if c2 = 1 then go to L;
    <critical section>
    c1 = 0;
```

```
    c2 = 1;
L:  if c1 = 1 then go to L;
    <critical section>
    c2 = 0;
```

Πρόβλημα; → **Deadlock!**

Αμοβαίος Αποκλεισμός με Load/Store (2)

- Χρήση 2 διαμοιραζόμενων μεταβλητών.

```
    c1 = 1;
L: if c2 = 1 {
       c1 = 0;
       go to L;
   }
   <critical section>
   c1 = 0;
```

```
    c2 = 1;
L: if c1 = 1 {
       c2 = 0;
       go to L;
   }
   <critical section>
   c2 = 0;
```

- Αποφυγή deadlock.
- Μικρή πιθανότητα *livelock*.
- Πιθανό *starvation*!

Αμοιβαίος Αποκλεισμός: Peterson (1981)

- Τροποποίηση του αλγορίθμου του Dekker (1986)
- Χρήση 3 διαμοιραζόμενων μεταβλητών.

```
    c1 = 1;
    turn = 1;
L:  if c2 = 1 & turn = 1
        go to L;
    <critical section>
    c1 = 0;
```

```
    c2 = 1;
    turn = 2;
L:  if c1 = 1 & turn = 2
        go to L;
    <critical section>
    c2 = 0;
```

- turn = i → μόνο το thread i μπορεί να περιμένει
- Οι c1, c2 εγγυόνται το mutual exclusion
- Πολύπλοκη λύση για n! (Dijkstra)

Peterson (1981) vs Dekker (1966)

Peterson's:

"I want to enter."	`flag[0]=true;`
"You can enter next."	`turn=1;`
"If you want to enter and	`while(flag[1]==true&&turn==1){`
it's your turn I'll wait."	`}`
Else: Enter CS!	`// CS`
"I don't want to enter any more."	`flag[0]=false;`

Dekker's:

"I want to enter."	`flag[0]=true;`
"If you want to enter	`while(flag[1]==true){`
and if it's your turn	`  if(turn!=0){`
I don't want to enter any more."	`  flag[0]=false;`
"If it's your turn	`  while(turn!=0){`
I'll wait."	`  }`
"I want to enter."	`  flag[0]=true;`
	`  }`
	`}`
Enter CS!	`// CS`
"You can enter next."	`turn=1;`
"I don't want to enter any more."	`flag[0]=false;`

Πηγή: http://cs.stackexchange.com/questions/12621/contrasting-peterson-s-and-dekker-s-algorithms

Αμοιβαίος αποκλεισμός

- Επιτρέπει την είσοδο στην κρίσιμη περιοχή μίας διεργασίας κάθε φορά
- Locks
  - acquire(lock) , release(lock)
  - δεύτερη προσπάθεια για acquire (από άλλη ή και την ίδια διεργασία) ενώ δεν έχει γίνει release, θα μπλοκάρει την εκτέλεση

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
shared int lock;
int id,amt;
acquire(lock);
if (accts[id].bal >= amt) {
   accts[id].bal -= amt;                    // critical section
   spew_cash(); }
release(lock);
```

  - Ζητούμενο: το acquire να γίνεται κι αυτό ατομικά!

Αμοιβαίος αποκλεισμός : ISA support

- Μπορεί να υλοποιηθεί με χρήση Load & Stores σε SC σύστημα ή και με την προσθήκη fences σε relaxed memory model.
  - Πολύπλοκος κώδικας
  - Μη αποδοτικός κώδικας

- Επέκταση ISA με atomic read-modify-write εντολές
  - Test and Set
  - Swap
  - Compare and Swap
  - Fetch and Increment, …

Spin-lock ("Test-And-Set")

- Πολλές αρχιτεκτονικές παρέχουν εντολές για atomic lock acquisition
  - Παράδειγμα: test-and-set
    - » t&s r1, 0(&lock)
  - Εκτελεί ατομικά:

```
mov r1,r2
ld r1,0(&lock)
st r2,0(&lock)
```

    - » Αν το lock ήταν ελεύθερο (=0), το δεσμεύει (το θέτει σε 1)
    - » Αν το lock ήταν δεσμευμένο (=1), δεν το αλλάζει
  - Acquire sequence:

```
A0: t&s r1,0(&lock)
A1: bnez r1,A0
```

  - Παρόμοιες εντολές: swap, compare & swap, exchange, fetch-and-add

Ορθότητα “Test-and-Set” Lock

Processor 0	Processor 1
A0: t&s r1,0(&lock)	
A1: bnez r1,#A0	A0: t&s r1,0(&lock)
CRITICAL_SECTION	A1: bnez r1,#A0
	A0: t&s r1,0(&lock)
	A1: bnez r1,#A0

- Ο P1 συνεχίζει να κάνει spin πάνω στο lock…

Απόδοση “Test-and-Set” Lock

Processor 1	Processor 2
A0: t&s r1,0(&lock)	
A1: bnez r1,#A0	A0: t&s r1,0(&lock)
A0: t&s r1,0(&lock)	A1: bnez r1,#A0
A1: bnez r1,#A0	A0: t&s r1,0(&lock)
	A1: bnez r1,#A0

M:1	I:	1
I:	M:1	1
M:1	I:	1
I:	M:1	1
M:1	I:	1

- …επηρεάζοντας αρνητικά όμως την απόδοση
  - έστω ότι έχουμε 3 επεξεργαστές αντί για 2
  - ο P0 έχει το lock και είναι εντός της κρίσιμης περιοχής
  - τι κάνουν οι P1, P2 στο ενδιάμεσο;
    - » εκτέλεση πολλών επαναλήψεων του t&s loop, κάθε μία εκ των οποίων περιλαμβάνει και ένα store
    - » ο ένας ακυρώνει συνεχώς την cache line του άλλου, παράγοντας ιδιαίτερα αυξημένη (και μη χρήσιμη) κίνηση στο δίαυλο

"Test-and-Test-and-Set" Locks

- Σκεπτικό: αντί μία διεργασία να γράφει «τυφλά» στο lock μέσω t&s, να παρακολουθεί απλά την τιμή του και μόνο όταν *φαίνεται* να είναι ελεύθερο να επιχειρεί να το δεσμεύσει
- Acquire sequence:

```
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1,r1
A3: t&s r1,0(&lock)
A4: bnez r1,A0
```

- Μέσα σε κάθε επανάληψη του loop, πριν γίνει ένα t&s:
  - επαναληπτικά ελέγχουμε (load) να δούμε αν η τιμή του lock έχει αλλάξει
  - εκτελούμε το t&s (store) όταν το lock (φαίνεται να) είναι ελεύθερο
- Οι επεξεργαστές κάνουν spinning τοπικά στην cache τους
- Λιγότερη άχρηστη κίνηση στο δίαυλο

Απόδοση "Test-and-Test-and-Set" Lock

```
Processor 1                 Processor 2
A0: ld r1,0(&lock)
A1: bnez r1,A0              A0: ld r1,0(&lock)
A0: ld r1,0(&lock)          A1: bnez r1,A0
        // lock released by processor 0
A0: ld r1,0(&lock)          A1: bnez r1,A0
A1: bnez r1,A0              A0: ld r1,0(&lock)
A2: addi r1,1,r1            A1: bnez r1,A0
A3: t&s r1,(&lock)          A2: addi r1,1,r1
A4: bnez r1,A0              A3: t&s r1,(&lock)
CRITICAL_SECTION            A4: bnez r1,A0
                            A0: ld r1,0(&lock)
                            A1: bnez r1,A0
```

S:1	I:	1
S:1	S:1	1
S:1	S:1	1
I:	I:	0
S:0	I:	0
S:0	S:0	0
S:0	S:0	0
M:1	I:	1
I:	M:1	1
I:	M:1	1
I:	M:1	1
I:	M:1	1

- Ο P0 κάνει release το lock, και κάνει invalidate την αντίστοιχη cache line στους P1, P2
- Οι P1, P2 ανταγωνίζονται για την απόκτηση του lock, ο P1 κερδίζει
- **CONSISTENCY???**

Atomic Instructions

- Πολλές αρχιτεκτονικές παρέχουν τη δυνατότητα ατομικής εκτέλεσης για συγκεκριμένες εντολές
  - όλες οι επιμέρους λειτουργίες που κάθε τέτοια εντολή περιλαμβάνει εκτελούνται σαν ένα ενιαίο σύνολο
  - συνήθως πρόκειται για Read-Modify-Write λειτουργίες
- π.χ. x86
  - INC, DEC, NOT, ADD, SUB, AND, OR, XOR,… (με LOCK prefix)
  - XCHG, CMPXCHG,…

```
int cmpxchg(int *p, int v1, int v2)
{ //atomically
     int oldval = *p;
     if (oldval == v1) *p = v2;
     return oldval;
}
```

  - στα παλιότερα μοντέλα υλοποιούνται με κεντρικό κλείδωμα του bus (#LOCK signal)
  - στα νεότερα μοντέλα, κλειδώνονται μόνο οι caches που περιέχουν τα αντίστοιχα δεδομένα (αν τα περιέχουν)
- (+) Γενικά, πολύ πιο αποδοτικές σε σχέση με τα locks
- (-) Κατάλληλες για απλές λειτουργίες (π.χ. RMW), αλλά όχι για πιο σύνθετες