Υ-07 – Παράλληλα Συστήματα
Transactional memory

Αρης Ευθυμίου

Παρ. προγρ/μός με κλειδιά

- Χαμηλού επιπέδου
  - πολύ κοντά στα μέσα και τις δομές του υλικού
  - πολλές λεπτομέρειες, εύκολα γίνεται λάθος
  - χαμηλή παραγωγικότητα
- Παράλληλος προγραμματισμός για «όλους»
  - απαιτεί υψηλότερου επιπέδου λύσεις
- Transactional memory
  - νέος τρόπος παράλληλου προγραμματισμού
  - μπορεί να έχει καλύτερη απόδοση

Εμπλοκή (deadlock)

- Προγραμματίζοντας με κλειδιά είναι σχετικά εύκολο να προκληθεί εμπλοκή

```
Lock(a);          Lock(b);
Lock(b);          Lock(a);
….                ….
Unlock(b);        Unlock(a);
Unlock(a);        Unlock(b);
```

- Ο προγραμματιστής είναι ελεύθερος να καλέσει τις ρουτίνες όπως θέλει
  - αντί να υπαρχει κάποιο πλαίσιο που εμποδίζει τα λάθη

Συναλλαγές (transactions)

- Εστιάζουν στην αδιαιρετότητα (atomicity) αντί για τα θεμελιακά στοιχεία των κλειδωμάτων (lock primitives)
- Το «σύστημα» εξασφαλίζει ότι μια συναλλαγή θα πραγματοποιηθεί ολόκληρη από μόνη της
  - σαν το υπόλοιπο σύστημα να μη κάνει τίποτα μέχρι να τελειώσει
  - συνήθως οι συναλλαγές εκτελούνται υποθέτοντας ότι δεν υπάρχουν αντιμαχόμενες συναλλαγές που εκτελούνται ταυτόχρονα (optimistically)
  - Αν το σύστημα ανιχνεύσει κάποια σύγκρουση (δλδ προσπέλαση μνήμης στις ίδιες διευθύνσεις), η συναλλαγή ακυρώνεται, χωρίς να έχει προκαλέσει οποιαδήποτε αλλαγή στα δεδομένα

Ιδιότητες συναλλαγών

- Αδιαίρετη εκτέλεση
  - όλες οι εντολές της συναλλαγής φαίνονται να εκτελούνται είτε όλες ή καμία (failure atomicity)
  - η συναλλαγή είτε ολοκληρώνεται (commit) ή ματαιώνεται (abort)
- Εκτελούνται μεμονωμένα (in isolation)
  - άλλες πράξεις δεν μπορούν να «δουν» ενδιάμεσα αποτελέσματα/δεδομένα της συναλλαγής
  - (serialisation) το αποτέλεσμα ταυτόχρονης εκτέλεσης συναλλαγών είναι το ίδιο σαν να είχαν εκτελεστεί με (κάποια) σειρά

Παράδειγμα 1

```
lock (lock1)
    counter = counter + 1;
 unlock (lock1)
```

```
transaction begin
    counter = counter + 1;
 transaction end
```

- Η διαφορά είναι μικρή

Παράδειγμα 2

- Παραγωγός – καταναλωτής με συνδεδεμένη λίστα
  - παραγωγός προσθέτει στην ουρά
  - καταναλωτής αφαιρεί από την κεφαλή

```
Enqueue
transaction begin
    if (tail == NULL)
      update head and tail
    else
      update tail
transaction end
```

```
Dequeue
transaction begin
    if (head->next == NULL)
      update head and tail
    else
      update head
transaction end
```

- Υλοποίηση με κλειδιά: μόνο μία πράξη στην λίστα μπορεί να γίνει σε κάθε στιγμή
- Με συναλλαγές οι πράξεις μπορούν να γίνουν ταυτόχρονα, αν δεν είναι (σχεδόν) άδεια η λίστα

Παράδειγμα 3

Παραγωγός – καταναλωτής με hash table, λίστα

```
transaction begin
    index = hash(key);
    head = bucket[index];
    traverse linked list until key matches
    perform operations
transaction end
```

- Σπάνια θα υπάρχει σύγκρουση σε ένα bucket, άρα οι συναλλαγές θα γίνονται παράλληλα χωρίς ματαιώσεις
- Υλοποίησεις με κλειδιά:
  - χονδρόκοκκη (coarse grain): όλες οι πράξεις γίνονται σειριακά
  - λεπτόκοκκη (fine grain): ένα κλειδί ανά bucket – περισσότερος χώρος αποθήκευσης, πολύπλοκος κώδικας, ...

Κλειδιά - συναλλαγές

- Απλά αλλάζοντας τα κλειδιά σε συναλλαγές δεν έχει πάντα το αναμενώμενο αποτέλεσμα!
- Κάθε συναλλαγή εκτελείται αδιαίρετα ως προς **κάθε άλλη** συναλλαγή
- Με τα κλειδιά υπάρχει ευελιξία:
  - διαφορετικά κλειδιά για ομάδες νημάτων που εκτελούνται ταυτόχρονα αλλά μοιράζονται διαφορετικά δεδομένα

Επανάληψη συναλλαγής

```
Enqueue
transaction begin
     if (queue is full)
          retry
     if (tail == NULL)
       update head and tail
     else
       update tail
  transaction end
```

- Ματαίωση της συναλλαγής και επανάληψη
  - η υλοποίηση μπορεί να προσπαθεί αμέσως ή να περιμένει μέχρι να δεί κάποιες αλλαγές στη μνήμη

Επιλογή

```
atomic {
    x = q0.dequeue()
orElse {
    x = q1.dequeue()
}
```

- Αν η πρώτη εντολή ματαιωθεί (ή προκαλέσει retry)
  - δοκίμασε την δεύτερη
- Αν και οι δύο ματαιωθούν
  - επανέλαβε ολόκληρο το atomic

Πλεονεκτήματα συναλλαγών

- Ευκολος προγραμματισμός
  - παρόμοια δυσκολία με coarse-grain locks
- Καλές επιδόσεις
  - παρόμοιες με fine-grain locks
- Παραλληλισμός με εικασία
  - ότι δεν υπάρχουν συγκρούσεις
- Δεν μπορεί να υπάρξει εμπλοκή (deadlock)
- Μπορεί να αντέξει σφάλματα
  - προσωρινά προβλήματα του συστήματος (π.χ. λάθος κατά τη μεταφορά)

Εγγραφές στη μνήμη

- Οι εγγραφές δεν μπορούν να αλλάξουν πραγματικά τη μνήμη
  - πριν ολοκληρωθεί η συναλλαγή
- Είτε οι νέες τιμές κρατούνται σε ένα χώρο που λειτουργεί παρόμοια με cache (lazy versioning)
  - και γράφονται στη μνήμη όταν επιτύχει η συναλλαγή
- ή οι παλιές τιμές κρατούνται σε ένα χώρο αποθήκευσης undo log (eager versioning)
  - ώστε να μπορούν να αποκατασταθούν αν η συναλλαγή ματαιωθεί

Ανίχνευση συγκρούσεων

- Το βασικό πρόβλημα υλοποίησης συστήματος συναλλαγών μνήμης είναι η ανίχνευση συγκρούσεων
  - απαιτεί γνώση των διευθύνσεων μνήμης που διαβάστηκαν και γράφτηκαν
- Κρατούνται δύο set
  - write-set – το σύνολο των διευθύνσεων που γράφονται
  - read-set – το σύνολο των διευθύνσεων που διαβάζονται
- Οταν μια άλλη συναλλαγή τελειώνει, σύγκριση του write set με το δικό μας read-set
  - αν υπάρχει επικάλυψη, ματαίωση της συναλλαγής (ποιάς?)
- Οταν η συναλλαγή τελειώνει, αποστολή του write-set σε όλους για έλεγχο

Πολιτικές ανίχνευσης

- Η προηγούμενη μέθοδος ανίχνευσης είναι αισιόδοξη
  - δουλεύει καλά αν σπάνια συμβαίνουν συγκρούσεις
  - όλη η δουλειά της συναλλαγής ακυρώνεται αν υπάρχει σύγκρουση
  - το κόστος είναι μεγάλο αν οι συγκρούσεις είναι συχνές
- Εναλλακτικά, κάθε εγγραφή μπορεί να ελέγχεται
  - μεγάλο κόστος ανα εγγραφή
  - γρήγορη ματαίωση σε περίπτωση σύγκρουσης
- Conflict detections: eager, lazy

Εμπειρικοί κανόνες

- Οι συναλλαγές συνήθως αφορούν λίγα δεδομένα
  - πάνω από 95% των προσπελάσεων χωρούν στη κρυφή μνήμη
- Οι συναλλαγές συνήθως ολοκληρώνονται με επιτυχία
  - Λιγότερες από 10% ματαιώνονται
- 99.9% των συναλλαγών δεν κάνουν I/O
- Νόμος του Amdahl: βελτιστοποίηση του πιο συχνού φαινομένου

Υλοποίηση TM

- Με λογισμικό (software transactional memory – STM)
  - αρκετά αργό χωρίς υποστήριξη από υλικό
- Με υλικό (hardware TM – HTM)
  - έμφαση της διάλεξης

TCC (lazy TM)

- Transactional Coherence and Consistency (TCC)
  - Kozyrakis, Olukotun (Stanford U.)
  - Lazy system: ανίχνευση συγκρούσεων, data versioning
  - Υποθέτουμε μικρό σύστημα με snoop πρωτόκολλο
- Βασίζεται σε μικρές αλλαγές του πρωτόκολλου συνοχής
- 2 επιπλέον bit κατάστασης ανά γραμμή κρυφής μνήμης
  - rd-bit - 1 όταν μια γραμμή διαβάζεται από μια συναλλαγή
  - wr-bit – 1 όταν μια γραμμή γράφεται από μια συναλλαγή

Λειτουργία TCC

- Αναγνώσεις:
  - η γραμμή cache, προσκομίζεται ως read-only, αν δεν υπάρχει ήδη
  - το rd-bit τίθεται (για το read-set)
- Εγγραφές:
  - το wr-bit τίθεται (για το write-set)
  - η γραμμή προσκομίζεται πάλι ως **read-only**, ως προς το πρωτόκολλο coherence, αν δεν υπάρχει ήδη
  - έτσι οι αλλαγές δεν φαίνονται εξωτερικά, πριν το τέλος της συναλλαγής
- Αντικατάσταση γραμμών
  - αν η γραμμή έχει το wr-bit η συναλλαγή απορίπτεται
  - ή υπάρχει κάποιος μηχανισμός σε λογισμικό

Διαδικασία commit

- Οταν η συναλλαγή τελειώσει οι αλλαγές πρέπει να γίνουν γνωστές στο σύστημα (commit)
- Ο διαιτητής του διαύλου (bus arbiter), αποφασίζει ποια συναλλαγή θα κάνει commit
  - η συναλλαγή κρατάει το δίαυλο μέχρι να κάνει όλες τις εγγραφές
  - Ολες οι γραμμές με wr-bit ίσο με 1, ακυρώνουν πιθανά αντίγραφα σε άλλες caches (busUpgrade αφού ήταν read-only)
  - δεν χρειάζεται πραγματική αντιγραφή των γραμμών μέχρι να αντικατασταθούν
- Οι άλλες συναλλαγές, αν έχουν ακύρωση μιας γραμμής που έχει το rd-bit, πρέπει να ματαιωθούν
  - καθαρίζουν τα rd- wr-bits και ξανατρέχουν (ή εκτελούν το orElse)

Συμπεράσματα TCC

- Lazy versioning:
  - οι αλλαγές είναι τοπικές (στην cache)
  - η μνήμη ενημερώνεται στο τέλος της συναλλαγής
- Lazy conflict detection:
  - στο τέλος (commit) γίνεται ο έλεγχος
- Οι ματαιώσεις γίνονται γρήγορα
  - Απαιτείται καθαρισμός των bit στη cache, άδειασμα pipeline και αντιγραφή καταχωρητών από checkpoint
- Η ολοκλήρωση της συναλλαγής είναι αργή
  - έλεγχος συγκρούσεων, ακυρώσεις μέσω πρωτοκόλου coherence
- Δεν συμβαίνουν αδιέξοδα (deadlock)
  - η συναλλαγή που θα κερδίσει τη διαιτησία ολοκληρώνεται πάντα

LogTM (eager TM)

- Log based TM
  - Un. of Wisconsin
- Εγγραφές γίνονται αμέσως, πριν τελειώσει η συναλλαγή
  - eager versioning
  - Αν γίνει ανάγνωση από άλλη συναλλαγή, η τελευταία τιμή επιστρέφεται και μπορεί να αλλάξει την κύρια μνήμη
  - η προηγούμενη τιμή κρατιέται σε log γιατί μπορεί η συναλλαγή να αποτύχει
  - το log κρατιέται στη μνήμη της διεργασίας και μπορεί να είναι στη κρυφή μνήμη
    - υπάρχει βοήθεια από το υλικό

Eager versioning

- Εγραφές απαιτούν ανάγνωση και εγγραφή της αρχικής τιμής στο log
  - τμήμα της εικονικής μνήμης του νήματος
- Μια γραμμή που ήδη καταγράφεται (ανήκει write-set) δεν χρειάζεται άλλες καταχωρήσεις στο log
  - μόνο η τιμή πριν την συναλλαγή χρειάζεται, όχι όλες οι ενδιάμεσες τιμές
- Το log μπορεί να είναι οργανωμένο ως στοίβα
  - υποστήριζει φωλιασμένες συναλλαγές

Ανίχνευση συγκρούσεων

- Αφού οι αλλαγές μιας συναλλαγής, TrA γίνονται άμεσα, ανάγνωση/εγγραφή από άλλη συναλλαγή, TrB, επικοινωνεί με το αντίγραφο της TrA
- Ανίχνευση σύγκρουσης: αν μία τουλαχιστον από τις δύο προσπελάσεις είναι εγγραφή
- Μία λύση (requester retries) είναι να καθυστερήσει η TrB:
  - η TrA στέλνει ένα NACK στην TrB
  - η TrB περιμένει λίγο και ξαναπροσπαθεί
  - με λίγη τύχη η TrA θα έχει ολοκληρωθεί και μπορεί να δώσει τη γραμμή στην TrB, χωρίς να χρειάζεται να ματαιωθεί καμία συναλλαγή

Εμπλοκή

- Η λύση requester replies μπορεί να προκαλέσει εμπλοκή:

Tr-A	Tr-B
write X	write Y
…	…
read Y	read X

- Μπορεί να ανιχνευθεί και να ματαιωθεί μία συναλλαγή, χρησιμοποιώντας έναν contention manager (σε υλικό/ λογισμικό)
- Εναλλακτικά κάθε συναλλαγή έχει μια ηλικία και η νεώτερη ματαιώνεται όταν υπάρχει σύγκρουση

eager versioning = απαισιοδοξία!

- Αφού οι συναλλαγές δεν ελέγχουν συγκρούσεις στο τέλος, δεν ξέρουμε με ποιά σειρά θα ολοκληρωθούν
  - αν T1 τελειώσει πριν την T2, δεν υπάρχει σύγκρουση
  - Αλλά επειδή δεν ξέρουμε τη σειρά υποθέτουμε το χειρότερο: T2 -> T1, δλδ. σύγκρουση