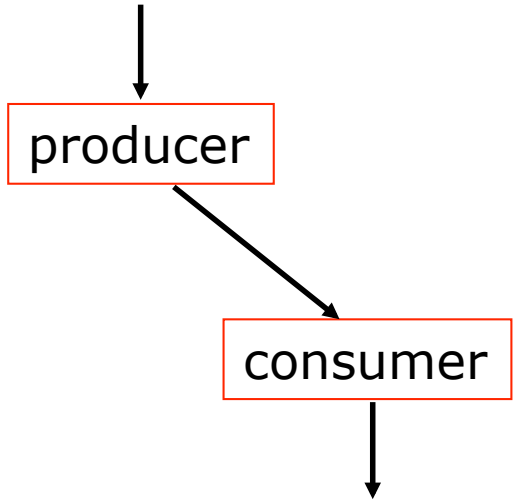
Υ-07 – Παράλληλα Συστήματα
Συνχρονισμός, κρυφές μνήμες πολλαπλών επιπέδων

Αρης Ευθυμίου

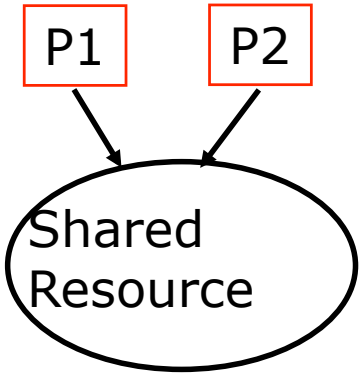
Synchronization

The need for synchronization arises whenever
there are concurrent processes in a system
(even in a uniprocessor system)

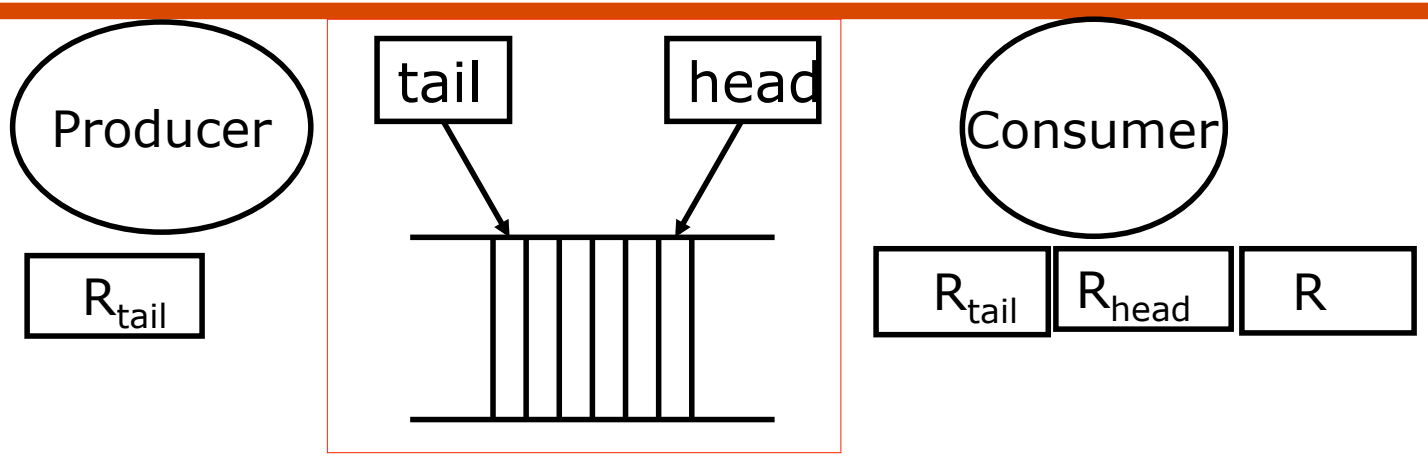


Producer-Consumer: A consumer process must wait until the producer process has produced data

Mutual Exclusion: Ensure that only one process uses a resource at a given time



A Producer-Consumer Example



Producer posting Item x:

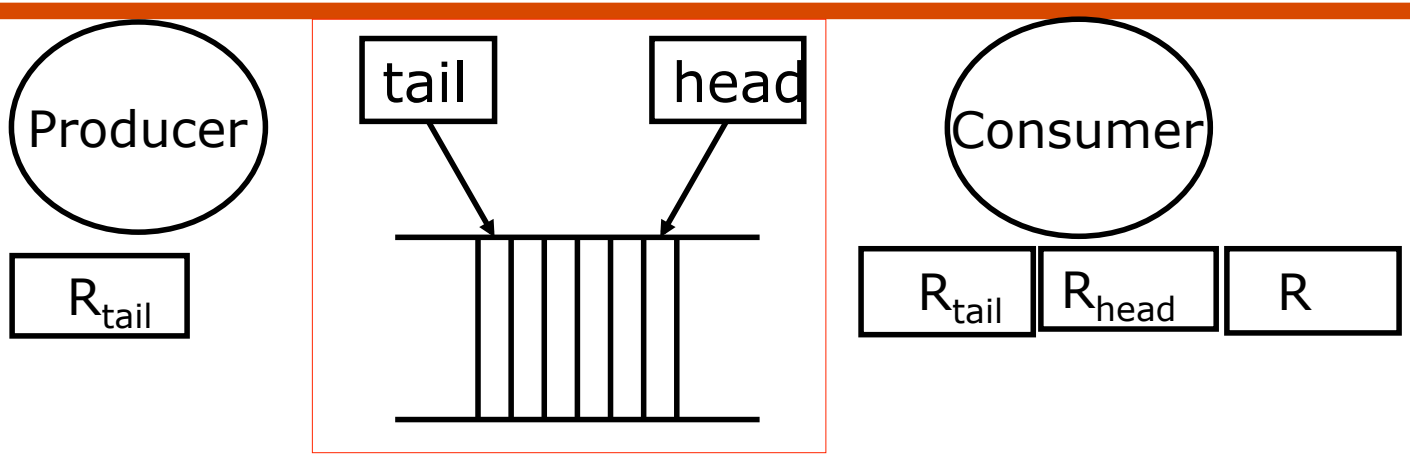
Load R_{tail}, (tail)
Store (R_{tail}), x
R_{tail}=R_{tail}+1
Store (tail), R_{tail}

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}
process(R)

The program is written assuming instructions are executed in order.

Using Memory Fences



Producer posting Item x:

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

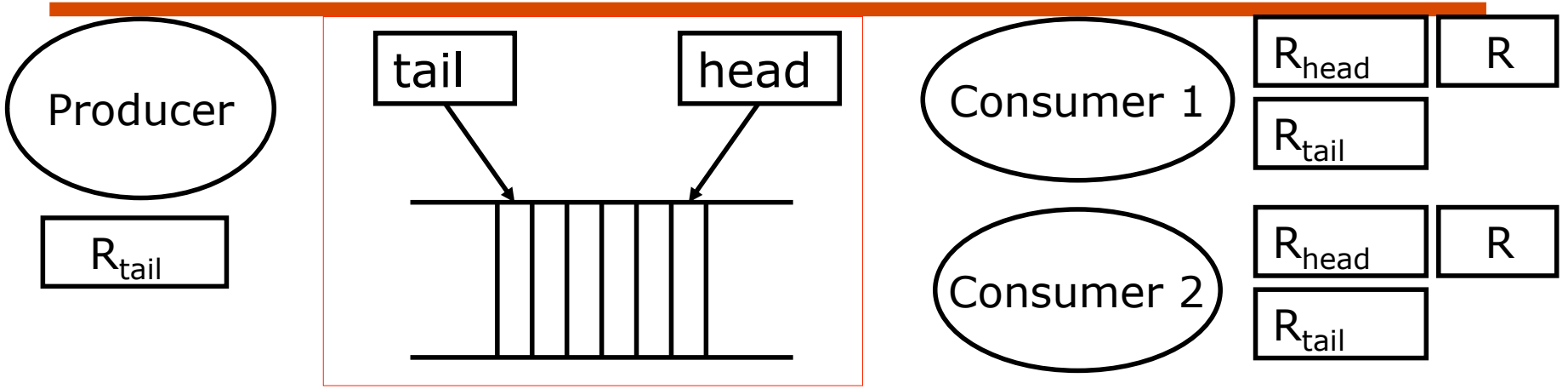
ensures that tail ptr is not updated before x has been stored

Consumer:

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

ensures that R is not loaded before x has been stored

Multiple Consumer Example



Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
R_{tail}=R_{tail}+1
Store (tail), R_{tail}

Consumer:

spin: Load R_{head}, (head)
Load R_{tail}, (tail)
if R_{head}==R_{tail} goto spin
Load R, (R_{head})
R_{head}=R_{head}+1
Store (head), R_{head}
process(R)

Critical section:
Needs to be executed atomically by one consumer ⇒ locks

What is wrong with this code?

Locks or Semaphores

E. W. Dijkstra, 1965

A *semaphore* is a non-negative integer, with the following operations:

P(s): *if s>0, decrement s by 1, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P's and V's must be executed atomically, i.e., without

- *interruptions* or
- *interleaved accesses to s* by other processors

```
Process i
    P(s)
        <critical section>
    V(s)
```

initial value of s determines the maximum no. of processes in the critical section

Blocking vs spinning

- Τα semaphores είναι χρήσιμα σε επίπεδο διεργασίας του Λ.Σ. αλλά όχι για νήματα
- Το «πάγωμα» (blocking) δουλεύει χρησιμοποιώντας το σύστημα των διακοπών (interrupts)
  - Αναστέλεται η εκτέλεση της διεργασίας και ξαναξεκινά να είναι διαθέσιμη για εκτέλεση όταν ο μετρητής είναι > 0
- Οι κλήσεις Λ.Σ. και οι διακοπές είναι αργές
- Για νήματα, γενικά χρησιμοποιούμε busy-waiting (spinning)
  - συνεχής έλεγχος μιας μεταβλητής (στη μνήμη) μέχρι να έχει την κατάλληλη τιμή

Υλοποίηση lock – 1η απόπειρα

- Κλείδωμα:

```
lock: ld r, location
      cmp r, #0
      bnz lock
      st location, #1
```

- Ξεκλείδωμα:

```
      st location, #0
```

- Προβλήματα;

Υλοποίηση Locks

- Θεωρητικά μπορούν να υλοποιηθούν με απλές load, store αλλά η υλοποίηση είναι δύσκολη
- Οι περισσότερες αρχιτεκτονικές διαθέτουν ειδικές εντολές που εκτελούν **αδιαίρετα** ανάγνωση και εγγραφή (atomic read-modify-write)
  - δεν παρεμβάλεται καμία άλλη προσπέλαση στη μνήμη μέχρι να ολοκληρωθεί η εντολή

```
Test&Set (m), R:
    R ← M[m];
    if R==0 then
        M[m] ← 1;
```

```
Fetch&Add (m), R_V, R:
    R ← M[m];
    M[m] ← R + R_V;
```

```
Swap (m), R:
    R_t ← M[m];
    M[m] ← R;
    R ← R_t;
```

m θέση μνήμης, R καταχωρητής

Multiple Consumers Example

using the Test&Set Instruction

```
P:      Test&Set (mutex),Rtemp
        if (Rtemp!=0) goto P
        Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rhead=Rhead+1
        Store (head), Rhead
V:      Store (mutex),0
        process(R)
```

Critical Section

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P's and V's

What if the process stops or is swapped out while in the critical section?

Απόδοση κλειδώματος

- Κάθε εντολή test&set προκαλεί μια εγγραφή στη μνήμη
  - επιπλέον της ανάγνωσης
- Κάθε έλεγχος είτε το κλειδί είναι ελεύθερο είτε όχι, προκαλεί εγγραφή
- Οι εγγραφές κοστίζουν πολύ
  - χρειάζονται προσπέλαση στο δίαυλο
- Το κλειδί θα βρίσκεται στην cache του τελευταίου πυρήνα που το πήρε. Για κάθε test&set, σε κάθε πυρήνα:
  - αστοχία στην τοπική cache,
  - προσπέλαση στο δίαυλο
  - ακύρωση του αντίγραφου στην άλλη cache
  - επανάληψη αν δεν πήραμε το κλειδί

Υποχώρηση (back-off)

- Το κύριο πρόβλημα είναι ότι το νήμα προσπαθεί αμέσως μετά την αποτυχία να πάρει το κλειδί
  - όλα τα νήματα κάνουν το ίδιο και υπάρχουν πολλές προσπελάσεις στο δίαυλο
- Αν προσθέσουμε μια καθυστέρηση, υπάρχει λιγότερος συναγωνισμός
  - και καλύτερη απόδοση
- Καλή μέθοδος είναι η *exponential backoff*
  - η καθυστέρηση μετά τη πρώτη προσπάθεια είναι *k,*
  - αυξάνεται εκθετικά σε κάθε επανάληψη (i): $k*c^i$

Ανάγνωση πριν τον έλεγχο

- Το άλλο πρόβλημα είναι ότι κάθε έλεγχος προκαλεί εγγραφή
- Μπορεί ο πυρήνας να διαβάζει μόνο το κλειδί και όταν το βρεί ξεκλείδωτο, τότε να εκτελέσει την test&set
  - λέγεται test & test & set
- Οσο το κλειδί είναι 1, περιμένει διαβάζοντας τοπικό αντίγραφο στη cache
  - εκτός από την πρώτη φορά που προκαλεί miss
- Οταν ο άλλος πυρήνας ξεκλειδώσει, κάνει εγγραφή
  - θα προκαλέσει invalidation
  - και νέο miss στην τοπική cache
  - μεγάλος συναγωνισμός για το κλειδί

Αλλες εντολές συνχρονισμού

- Αντί για ξεχωριστές πράξεις στο «κλειδί» και στα δεδομένα, μπορεί κανείς να συνδιάσει το κλείδωμα με την πράξη
  - ένα νήμα ολοκληρώνει το critical section
  - λέγονται non-blocking
- Εντολές αρχιτεκτονικής
  - compare and swap
  - load-reserve (load-link, load-lock) – store-conditional

Nonblocking Synchronization

```
Compare&Swap(m), Rt, Rs:
  if (Rt==M[m]) then
        M[m]=Rs;
        Rs=Rt ;
        status ← success;
  else
        status ← fail;
```

status is an *implicit argument*

```
try:    Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rnewhead = Rhead+1
        Compare&Swap(head), Rhead, Rnewhead
        if (status==fail) goto try
        process(R)
```

Load-locked & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-locked R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs' reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-locked 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-conditional (head), R_head
        if (status==fail) goto try
        process(R)
```

load-locked, store-conditional

- Η εγγραφή γίνεται μόνο όταν η συνθήκη ισχύει
  - δεν γίνονται εγγραφές αν δεν πάρει το κλειδί
  - σύγκριση με test & test & set
- Μπορεί να χρησιμοποιηθεί για να υλοποιήσει σε λογισμικό διάφορες πράξεις κλειδώματος
  - π.χ. fetch&add
- και lock-free δομές δεδομένων
  - π.χ. κοινόχρηστος μετρητής/accumulator
- Οι εντολές μεταξύ ll, sc δεν εκτελούνται αδιαίρετα
  - το ll δεν είναι lock και το sc unlock!
  - Αν το sc πετύχει, ολόκληρη η ακολουθία θεωρείται αδιαίρετη μόνο για εγγραφές στη θέση μνήμης m

Υλοποίηση ll-sc σε υλικό

- Ο ορισμός των ll,sc προϋποθέτει ένα επιπλέον bit σε κάθε λέξη στη μνήμη
  - τεράστιο κόστος
- Στην πραγματικότητα κάθε πυρήνας έχει καταχωρητές lock-flag και lock-address
  - για κάθε ll που μπορεί εκρεμεί
- Οταν εκτελεστεί η ll, θέτει τους καταχωρητές
- Κάθε ακύρωση (invalidation) εξετάζει τη lock-address
  - και θέτει το flag στο 0 αν η διεύθυνση ταιριάζει
  - επίσης το flag γίνεται 0 αν η εκτέλεση του νήματος ανασταλεί
- Αν όταν εκτελεστεί η sc το flag είναι 0, η sc αποτυχαίνει

ticket-lock

- Δουλεύει παρόμοια με το σύστημα ουράς μιας τράπεζας
- Κάθε νήμα παίρνει έναν αριθμό-εισητήριο
  - χρησιμοποιώντας αδιαίρετο fetch&increment
- και περιμένει κοιτώντας το πίνακα που αναφέρει τον αριθμό που εξυπηρετείται
  - κάνοντας απλά load σε μια διεύθυνση
- Η αδιαίρετη πράξη γίνεται μόνο στην αρχή
  - συνήθως τα νήματα «φτάνουν» σε διαφορετικές χρονικές στιγμές
- Το ξεκλείδωμα είναι μια εγγραφή
  - τα τοπικά αντίγραφα ακυρώνονται και όλα τα νήματα διαβάζουν την καινούρια τιμή

Array-based lock

- Κάθε νήμα χρησιμοποιεί αδιαίρετο fetch&increment για να πάρει μια διεύθυνση μνήμης για να κάνει spin
  - κάθε νήμα περιμένει σε διαφορετική διεύθυνση
  - σε χωριστές γραμμές cache για να μην έχουμε false-sharing
- Ξεκλείδωμα: εγγραφή ειδικής τιμής στην επόμενη διεύθυνση
- Πλεονέκτημα: μόνο μία cache ακυρώνει τη γραμμή κλειδί κατά το ξεκλείδωμα
- Πρακτικά θέματα:
  - wrap-around στις διευθύνσεις κλειδιών
  - αν υπάρχουν περισσότερα νήματα από κλειδιά, συναγωνισμός

Συγχρονισμός νημάτων

Δύο είδη:

- συγχρονισμός ένα-προς-ένα (point to point)
  - χρησιμοποιεί locks απευθείας, π.χ. multiple consumers
- καθολικός συγχρονισμός (global)
  - χρησιμοποιεί barrier
  - δεν πρέπει να συγχεόνται με memory fences για επιβολή σειράς μεταξύ προσπελάσεων μνήμης
- Υλοποίηση barrier:
  - μετρητής αυξάνεται από κάθε νήμα που φτάνει στο barrier
  - κάθε νήμα περιμένει μέχρι ο μετρητής να φτάσει τον αριθμό νημάτων που συμμετέχουν

barrier – απόπειρα 1

```
struct bar_type {
   int counter;
   struct lock_type lock;
   int flag = 0;
} bar_name;

BARRIER (bar_name, p) {
  LOCK(bar_name.lock);
  if (bar_name.counter == 0)
    bar_name.flag = 0;  /* reset flag if first to reach*/
  mycount = bar_name.counter++; /* mycount is a private */
  UNLOCK(bar_name.lock);
  if (mycount == p) {/* last to arrive */
    bar_name.counter = 0;/* reset counter for next barrier */
    bar_name.flag = 1;/* release waiting processes */
  } else
    while (bar_name.flag == 0)
      ;  /* busy wait for release */
}
```

corner case bug

Αν το barrier χρησιμοποιείται όπως παρακάτω:

```
BARRIER(bar1, p); … BARRIER(bar1, p);
```

- Ενα νήμα μπορεί να κολήσει στο loop περιμένοντας το flag να γίνει 1 και το Λ.Σ. μπορεί να αναστέλει την εκτέλεσή του
- Αν τα υπόλοιπα συνεχίσουν και μπουν στο 2ο barrier
  - το πρώτο θα κάνει το flag 0
- και το κολημένο νήμα δεν θα βγεί ποτέ από το πρώτο barrier
  - και τα υπόλοιπα θα κολήσουν στο 2ο περιμένοντας αυτό να νήμα να φτάσει

barrier with sense reversal

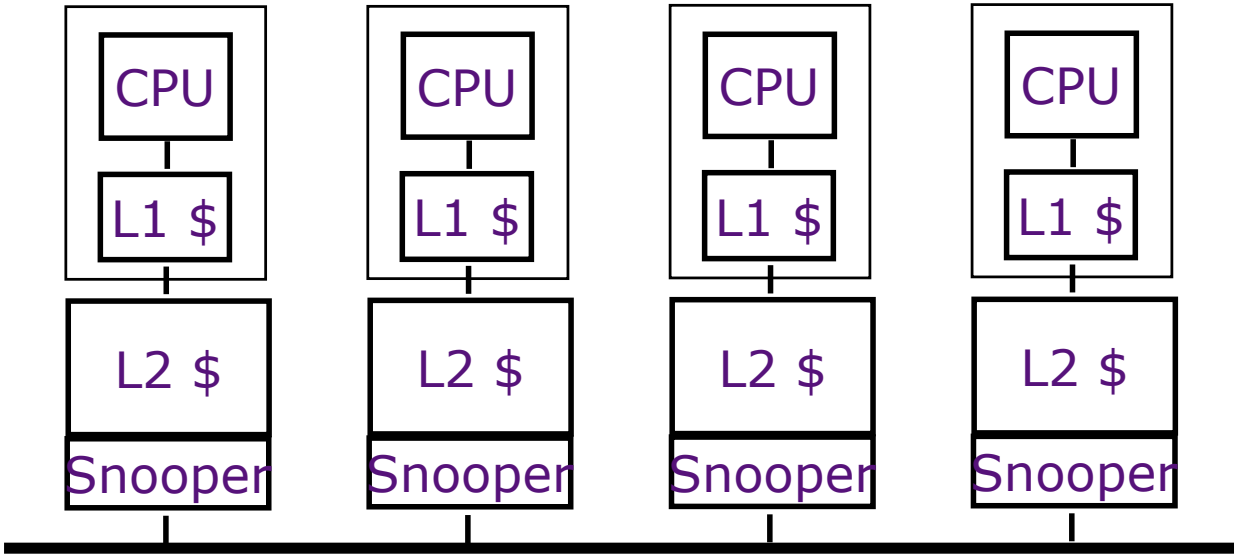
- Χρησιμοποιούμε και τις δύο τιμές για το flag και τα νήματα περιμένουν διαφορετικές τιμές σε συνεχόμενα barrier

```
BARRIER (bar_name, p) {
  local_sense = !(local_sense);
  LOCK(bar_name.lock);
  bar_name.counter++;
if (bar_name.counter == p) {/* last to arrive */
    UNLOCK(bar_name.lock);
    bar_name.counter = 0;/* reset counter for next barrier */
    bar_name.flag = local_sense;/* release waiting processes */
  } else
    UNLOCK(bar_name.lock);
    while (bar_name.flag != local_sense)
      ;  /* busy wait for release */
}
```

Συνχρονισμός - περίληψη

- Αμοιβαίος αποκλεισμός (mutual exclusion)
- Καθολικός συγχρονισμός (Barriers)
  - χρησιμοποιεί locks όπως και το mutex
- Απόδοση:
  - πόσο γρήγορα μπορεί ένα νήμα να πάρει το κλειδί (όταν υπάρχει συναγωνισμός, όταν δεν υπάρχει)
- Αποφυγή deadlock, livelock
- Fairness, αποφυγή starvation
- Αποφυγή
  - lock-free δομές δεδομένων

Πολλαπλά επίπεδα cache



- Οι σύνχρονοι πυρήνες έχουν πολλαπλά επίπεδα κρυφής μνήμης
  - Συνήθως 2-3 επίπεδα ανά πυρήνα μέσα στο Ο.Κ. του επεξεργαστή
- Πώς δουλεύει το snooping με πολλαπλά επίπεδα cache;
  - Αλλαγές από τον πυρήνα στο πρώτο επίπεδο δεν γίνονται ορατές από το ελεγκτή του τελευταίου επιπέδου που είναι υπεύθυνος για το snooping

Cache inclusion

Ιδιότητα της κρυφής μνήμης που διευκολύνει το πρόβλημα snooping σε caches πολλαπλών επιπέδων

- Κάθε γραμμή της cache επιπέδου i, πρέπει να βρίσκεται και στην cache επιπέδου i+1
- Η διατήρηση της ιδιότητας δεν είναι τόσο απλή...
- Το κόστος δεν είναι αμελητέο: η πραγματική χωρητικότητα των cache υψηλού επιπέδου μειώνεται, άρα έχουν χαμηλότερο hit rate

Διατήρηση cache inclusion

Προβληματική για τους ακόλουθους λόγους:

- Προσπελάσεις από τον πυρήνα αλλάζουν το 1ο επίπεδο και προκαλούν αντικαταστάσεις γραμμών
- Προσπελάσεις στο δίαυλο προκαλούν αλλαγή κατάστασης ή flush γραμμών του τελευταίου επιπέδου
- Αν μια γραμμή είναι σε κατάσταση modified στο επίπεδο i, πρέπει να είναι σε κατάσταση modified (ή παρόμοια) στο επίπεδο i+1

Παράδειγμα: 2 επίπεδα, ίδιο μέγεθος γραμμής, LRU πολιτική αντικατάστασης, L1 2way, L2 4way

Υλοποίηση inclusion

- Υποθέτουμε 2 επίπεδα cache
- Για κάθε γραμμή της L2 που αντικαθίσταται, ο ελεγκτής της L2 στέλνει μήνυμα στην L1 για ακύρωση ή flush (αν είναι αλλαγμένη) των αντίστοιχων γραμμών της L1
  - Η L2 μπορεί να έχει διαφορετικό (μεγαλύτερο) μέγεθος γραμμής από την L1
  - Παρόμοια για ακυρώσεις γραμμών της L2
- Αν υπάρχουν πολλές L1 πάνω από μία L2;
  - π.χ. χωριστές L1 caches για εντολές, δεδομένα
  - Μηνύματα σε όλες (αν δεν εχουν τη γραμμή, δεν κάνουν τίποτα)
  - Επιπλέον πληροφορία στην L2 για τις L1 που έχουν αντίγραφο της γραμμής

Υλοποίηση inclusion – write hits

- Σε ένα write hit στην L1, η πληροφορία ότι η γραμμή είναι modified πρέπει να περάσει στην L2
- Ορίζεται μια νέα κατάσταση modified-but-stale για τις ενδιάμεσες caches
- Συμπεριφέρονται ως modified για μηνύματα συνοχής (coherence) από το δίαυλο αλλά επιστρέφει δεδομένα από την L1 όταν ζητείται flush

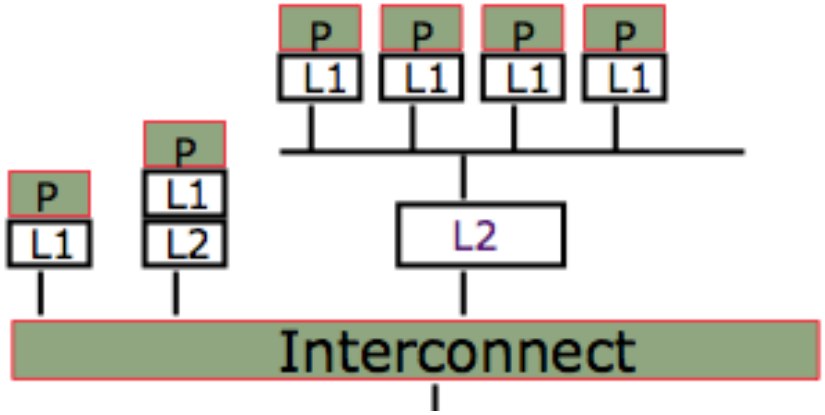
Λειτουργία inclusive caches

- Ενα αίτημα από τον πυρήνα προχωράει «προς τα κάτω» (μακριά από τον πυρήνα) μέχρι να βρει μια γραμμή στη cache σε κατάλληλη κατάσταση
  - αναγνώσεις – οτιδήποτε εκτός από invalid
  - εγγραφές, read-exclusive – modified ή exclusive
- Αν δεν ικανοποιηθεί το αίτημα μέχρι να φτάσει στην τελευταία cache, προσπέλαση στο δίαυλο
- Απαντήσεις προχωρούν προς τα πάνω αλλάζοντας τις cache που συναντούν μέχρι να φτάσουν στον πυρήνα
  - απαντήσεις shared – κατάσταση shared παντού
  - απαντήσεις read-exclusive – κατάσταση modified-but-stale παντού εκτός από την L1, κατάσταση modified (ή exclusive για MESI)

Λειτουργία inclusive caches

- Αιτήματα από τον δίαυλο: invalidation, flush, copy-back προχωρούν προς τα πάνω, αλλάζοντας την κατάσταση των γραμμών
  - invalid για αιτήματα invalidation, flush
  - shared για αιτήματα copy-back
  - Απαντήσεις για αιτήματα flush, copy-back δίνονται από την cache που έχει τη γραμμή σε κατάσταση modified
- Εκδιώξεις (evictions)
  - στέλνουν αίτημα προς τα πάνω ώστε να διατηρηθεί η inclusion (μπορεί να είναι και σε κατάσταση modified)
  - Αν η γραμμή δεν είναι αλλαγμένη (clean), δεν χρειάζεται να ενημερωθεί η ιεραρχία προς τα κάτω
  - Αλλιώς η γραμμή γράφεται στην κάτω cache και αλλάζει η κατάστασή σε modified

Μοιραζόμενες caches



- Πιο συνηθισμένη ιεραρχία μνήμης σε CMPs σήμερα
- Πολλαπλά επίπεδα συνοχής μνήμης
- Η μοιρασμένη L2 κρατά σε συνοχή τις L1 που βρίσκονται ψηλότερα
- Αλλο επίπεδο συνοχής για τις L2, κτλ

Πλεονεκτήματα

- Η μοιρασμένη cache διευκολύνει την ανταλλαγή δεδομένων μεταξύ νημάτων που τρέχουν στους πυρήνες που τη μοιράζονται
  - Σαν κοινόχρηστη κύρια μνήμη, αλλά ταχύτερη
- Αν οι πυρήνες τρέχουν ανεξάρτητα προγράμματα (multi-programming) και ένα από αυτά χρειάζεται περισσότερη cache απ'ότι του αναλογεί, μπορεί να το κάνει
  - σε συναγωνισμό με τους υπόλοιπους φυσικά

Διατήρηση συνοχής

- Κάθε γραμμή κρατά πληροφορίες για τα «αδέλφια» της και τα «παιδιά» της (προς τους πυρήνες)
- Πληροφορίες για τα αδέλφια: έχει κανείς αντίγραφο;
  - shared – μπορεί να υπάρχουν και άλλα αντίγραφα
  - exclusive – έχω το μοναδικό αντίγραφο (ίσως και modified)
  - όπως οι caches πάνω από δίαυλο
- Πληροφορίες για τα παιδιά: έχω δώσει αυτή τη γραμμή σε κάποιο παιδί;
  - W(core_id) – η μοναδική cache που έχει αντίγραφο. Το δικό μου αντίγραφο θεωρείται stale (modified-but-stale)
  - R(list of core_ids) – κατάλογος cache που έχει αντίγραφο

Ασκηση 3 – Αλγόριθμοι αντικατάστασης

- Ο αλγόριθμος LRU έχει προβλήματα σε επιπέδα cache > 1
- Υλοποίηση αλγορίθμου SRRIP (static re-reference interval prediction)
  - άρθρο Jaleel *et al* ISCA'10
- Re-reference interval – χρόνος μεταξύ προσπελάσεων γραμμής κρυφής μνήμης
  - όσο μικρότερος τόσο πιο σημαντική είναι η γραμμή για την cache
  - δεν είναι γνωστός από πρίν, πρέπει να τον μαντέψουμε
  - ο LRU υποθέτει ότι κάθε καινούρια γραμμή στην cache (και μετά από κάθε προσπέλαση) το RRI της γραμμής θα είναι το μικρότερο του set
  - Ο SRRIP επιτρέπει διαχωρισμό του RRI για καινούριες γραμμές και προσπελάσεις παλιώτερων