ΗΥ-225: Οργάνωση Υπολογιστών
Άνοιξη 2017

Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

Σειρά Ασκήσεων 14: Συνοχή (Coherence) Κρυφών Μνημών, Προχωρημένοι Επεξεργαστές (Out-of-Order, Superscalar, Multithreading, Multicores)

Προθεσμία: έως Παρασκευή 26 Μαΐου 2017, ώρα 23:59 (βδ. F)

Βιβλίο (Ελληνικό, έκδοση 4) - διαβάστε τα εξής, και με τους εξής τρόπους:

- Συνοχή (Coherence) Κρυφών Μνημών - σελίδες 623-628 (§ 5.8): διαβάστε τις κανονικά, αποτελούν μέρος της εξεταστέας ύλης.
- Παραλληλία Επιπέδου Εντολής (ILP) - Επεξεργαστές με προχωρημένες μορφές ομοχειρίας (out-of-order execution, VLIW, superscalar) - σελίδες 456-471 (§ 4.10): διαβάστε τις κανονικά, αποτελούν μέρος της εξεταστέας ύλης.
- Πολυνημάτωση (Multithreading) - σελίδες 754-758 (§ 7.5): διαβάστε τις κανονικά, αποτελούν μέρος της εξεταστέας ύλης.
- Πολυπύρηνοι Επεξεργαστές (Multicores) (§ 1.6): σελίδες 76-79.
- Πολυεπεξεργαστές κοινόχρηστης μνήμης (shared memory) (§ 7.3): σελίδες 746-747 προσεκτικά, και σελίδες 748-749 πιό γρήγορα.

- Πολυεπεξεργαστές με μεταβίβαση μηνυμάτων (message passing) και συστοιχίες (clusters) (§ 7.4): σελίδες 749-750 επί τροχάδην, και σελίδες 751-754 εγκυκλοπαιδικά.
- Άλλα θέματα παραλληλισμού - SIMD, MIMD, Vector, GPU - § 7.6 και 7.7 (σελ. 758-773): προαιρετικά - εγκυκλοπαιδικά (σημαντικά θέματα γιά περαιτέρω προσωπική σας μελέτη, αλλά εκτός ύλης αυτού του μαθήματος).

Άσκηση 14.1: DMA και Συνοχή Κρυφής-Κύριας Μνήμης

Σ' ένα σύστημα όπου γίνονται μεταφορές DMA πρέπει να λυθεί το πρόβλημα της συνοχής κρυφής και κύριας μνήμης (πρόβλημα **Cache Coherence**). Δείξτε ποιό είναι το πρόβλημα αυτό, κάνοντας τα εξής. Σχεδιάστε (i) τον επεξεργαστή με την κρυφή του μνήμη, η οποία συνδέεται στην αρτηρία (λεωφόρο - bus) μνήμης-Ε/Ε, (ii) την κύρια μνήμη, συνδεδεμένη στην ίδια αρτηρία, και (iii) μία συσκευή Ε/Ε με μηχανισμό DMA, συνδεδεμένη στην ίδια αρτηρία.

(α) Θεωρήστε την περιφερειακή συσκευή σαν συσκευή εισόδου, και θεωρήστε ότι αυτή μεταφέρει μέσω DMA νέα δεδομένα εισόδου σε κάποια περιοχή διευθύνσεων στην κύρια μνήμη. Μετά τη λήξη της μεταφοράς, το πρόγραμμα που τρέχει στον επεξεργαστή θέλει να διαβάσει (μέσω load) τα νέα δεδομένα εισόδου από την περιοχή διευθύνσεων στην κύρια μνήμη όπου αυτά έχουν τοποθετηθεί από το DMA. Σε ποια περίπτωση θα διαβάσει τα σωστά νέα δεδομένα, και σε ποια περίπτωση θα διαβάσει λανθασμένες παλαιές τιμές;

(β) Θεωρήστε την περιφερειακή συσκευή σαν συσκευή εξόδου, και θεωρήστε ότι το πρόγραμμα που τρέχει στον επεξεργαστή παράγει μερικά νέα δεδομένα τα οποία γράφει (μέσω store) σε ορισμένη περιοχή διευθύνσεων μνήμης, και τα οποία στη συνέχεια θέλει να στείλει στην περιφερειακή συσκευή. Γιά το σκοπό αυτό, το λειτουργικό σύστημα ξεκινάει μία μεταφορά DMA από την παραπάνω περιοχή διευθύνσεων κύριας μνήμης προς τη συσκευή εξόδου. Έστω ότι η κρυφή μνήμη του επεξεργαστή είναι τύπου **write through**, δηλαδή, ως γνωστόν, κάθε τι που γράφει ο επεξεργαστής σε αυτήν, αυτή το γράφει αμέσως και στην κύρια μνήμη. Υπ' αυτές τις συνθήκες, υπάρχει περίπτωση να φτάσουν λάθος (παλαιά) δεδομένα στη συσκευη εξόδου; Γιατί όχι;

(γ) Έστω τώρα ότι στο σύστημα (β) η κρυφή μνήμη είναι τύπου **write back**, δηλαδή δεν γράφει αμέσως στην κύρια μνήμη κάθε αλλαγή τιμής (εγγραφή νέας τιμής) που κάνει ο επεξεργαστής, αλλά το γράφει αργότερα, όταν το block όπου έγινε η αλλαγή πρέπει να αντικατασταθεί στην κρυφή μνήμη από άλλο block. Υπ' αυτές τις συνθήκες, σε ποια περίπτωση θα καταλήξουν τα σωστά νέα δεδομένα στη συσκευή εξόδου, και σε ποια περίπτωση θα καταλήξουν εκεί λανθασμένες παλαιές τιμές;

Λύσεις στο πρόβλημα της συνοχής κρυφής και κύριας μνήμης υπάρχουν "μεσοβέζικες", με εκδίωξη (flush) σελίδων από την κρυφή μνήμη (δύσκολο ή χρονοβόρο) ή με χρήση σελίδων που η κρυφή μνήμη αναγνωρίζει και δεν κρατά (non-cacheable pages) (μειώνει την επίδοση του επεξεργαστή), ή "ριζικές", με χρήση ενός πρωτόκολλου συνοχής κρυφών μνημών σαν αυτά που χρησιμοποιούν οι πολυεπεξεργαστές κοινόχρηστης μνήμης (shared-memory multiprocessors).

14.2: Συνοχή (Coherence) Κρυφών Μνημών - Snooping

Οι πολυεπεξεργαστές κοινόχρηστης μνήμης (shared memory multiprocessors) αποτελεούνται από πολλαπλούς επεξεργαστές που όλοι "βλέπουν" μιά κοινή μνήμη, μέσω της οποίας επικοινωνούν μεταξύ τους και μπορούν να συνεργάζονται (π.χ. όταν εκτελούν ένα παράλληλο πρόγραμμα). Η επικοινωνία τους προκύπτει όταν ένας επεξεργαστής γράφει κάποια νέα αποτελέσματα σε ορισμένες θέσεις της κοινής μνήμης, και ένας ή περισσότεροι άλλοι επεξεργαστές διαβάζουν αυτές τις νέες τιμές από εκείνες τις θέσεις μνήμης –κάτι που θυμίζει αρκετά και την επικοινωνία με τις συσκευές εισόδου/εξόδου. Οι πιό συχνοί τέτοιοι πολυεπεξεργαστές σήμερα είναι οι πολυπύρηνοι επεξεργαστές (multicore processors), που έχουν πολλαπλούς πυρήνες (cores) –δηλαδή επεξεργαστές– όλους μέσα στο ίδιο chip.

Επειδή οι μνήμες είναι συνήθως μονόπορτες (γιά να μην κοστίζουν πολύ), θα ήταν πολύ αργό εάν όλοι οι επεξεργαστές σ' ένα τέτοιο σύστημα εργάζονταν συνεχώς με την (μοναδική πόρτα της) μία(ς) κοινόχρηστη(ς) μνήμη(ς). Αντ' αυτού, λοιπόν, ο κάθε επεξεργαστής έχει την δική του "ιδιωτική" (private) κρυφή μνήμη, και μόνον οι αστοχίες αυτών των κρυφών μνημών (που είναι και σπανιώτερες) πηγαίνουν στην (μοναδική πόρτα της) κοινόχρηστη(ς) μνήμη(ς). Αυτήν την (μοναδική) πόρτα της κοινόχρηστης μνήμης την βλέπουμε συνήθως σαν μιά αρτηρία (bus) που ενώνει όλες τις κρυφές μνήμες όλων των πυρήνων (επεξεργαστών) καθώς και τη μνήμη. Όλες οι αστοχίες (misses) των κρυφών μνημών περνάνε από αυτή την αρτηρία (και άρα, όποιος θέλει και κοιτάζει (snoop) μπορεί να τις βλέπει).

Όποτε δημιουργούμε πολλαπλά αντίγραφα της ίδιας πληροφορίας και κάποιος αλλάζει ένα από τα αντίγραφα (ή το πρωτότυπο) ενώ άλλοι έχουν και βλέπουν άλλα αντίγραφα, τότε είναι σα να "πηγαίνουμε γυρεύοντας γιά μπελάδες"! Αυτό ισχύει και στο υλικό (κρυφές μνήμες), και στο λογισμικό (π.χ. web browser caches), και στην καθημερινή μας ζωή (π.χ. πολλοί φίλοι έχουν αντίγραφο του αριθμού τηλεφώνου μου στα κινητά τους, κι εγώ αλλάζω αριθμό τηλεφώνου...). Όπως και με τα I/O DMA's και την κρυφή μνήμη, παραπάνω στην άσκηση 14.1, ανάλογα προβλήματα εμφανίζονται και με τις (ιδιωτικές) κρυφές μνήμες στους πολυεπεξεργαστές κοινόχρηστης μνήμης.

Την επιθυμητή συνοχή (coherence) τέτοιων κρυφών μνημών, δηλαδή το να βλέπουν όλοι οι επεξεργαστές την ίδια τιμή στην ίδια διεύθυνση (σχεδόν) ανά πάσα στιγμή, την εξασφαλίζουν ειδικά πρωτόκολα συνοχής, συνήθως υλοποιημένα σε υλικό (χωρίς να αποκλείονται και πρωτόκολα σε λογισμικό), τα απλούστερα από τα οποία είναι τα πρωτόκολα "snooping" (κρυφοκοίταγμα, ή κατασκοπία), όταν όλες οι κρυφές μνήμες συνδέονται πάνω στην ίδια αρτηρία (bus) και παρακολουθούν εκεί "όλα τα νέα της γειτονιάς".

Συνήθως τα πρωτόκολα αυτά ακολουθούν την πολιτική "**write-invalidate**": όποτε ένας από τους επεξεργαστές γράφει σε μιά διεύθυνση μνήμης (που αντίγραφό της είχε ή φέρνει στην κρυφή του μνήμη), τότε πρέπει να σιγουρευτούμε ότι αυτό θα είναι το

μοναδικό αντίγραφο αυτής της διεύθυνσης, μεταξύ όλων των κρυφών μνημών του συστήματος. Εάν λοιπόν ο επεξεργαστής αυτός δεν είναι σίγουρος ότι κανείς άλλος δεν έχει αντίγραφο σε άλλη κρυφή μνήμη, τότε είναι υποχρεωμένος να "βροντοφωνάξει" (broadcast) πάνω στην κοινή αρτηρία ότι όλοι οι άλλοι πρέπει να **ακυρώσουν** (invalidate) τα αντίγραφα που τυχόν είχαν (αφού περιέχουν την παλαιά, άκυρη πλέον, τιμή).

Ανάλογα με το πόσο λεπτομερή πληροφορία θέλει να θυμάται η κάθε κρυφή μνήμη γιά την κάθε γραμμή δεδομένων της, μπορεί, σε διάφορες περιπτώσεις, να είναι ή να μην είναι σίγουρη κατά πόσον μπορεί ή δεν μπορεί να υπάρχουν αντίγραφα της ίδιας γραμμής και σε άλλες κρυφές μνήμες. Όλα τα πρωτόκολα ξεχωρίζουν γιά την κάθε γραμμή σε ποιάν από τις εξής τρείς τουλάχιστο περιπτώσεις ("καταστάσεις") αυτή βρίσκεται:

- **M** (Modified): έχω αλλάξει (modified) το περιεχόμενο αυτής της γραμμής, και άρα είμαι η μόνη κρυφή μνήμη που έχω αντίγραφό της (dirty).
- **S** (Shared): έχω έγκυρο αντίγραφο αυτής της γραμμής, αλλά ενδέχεται και άλλες κρυφές μνήμες να έχουν αντίγραφό της, και άρα, βάσει της συμφωνίας μας "write-invalidate", το αντίγραφό μου είναι "καθαρό" (clean), δηλαδή όχι dirty.
- **I** (Invalid): αυτή η γραμμή της κρυφής μου μνήμης είναι άκυρη, δηλαδή περιέχει σκουπίδια.

Εκτός από τις τρείς παραπάνω περιπτώσεις (καταστάσεις) γιά την κάθε γραμμή, που υπάρχουν σε όλα τα πρωτόκολα συνοχής, μερικά πρωτόκολα κρατάνε λεπτομερέστερες πληροφορίες –ξεχωρίζουν μία ή δύο περισσότερες περιπτώσεις (καταστάσεις):

- **E** (Exclusive): έχω έγκυρο και καθαρό (clean) (not dirty) αντίγραφο αυτής της γραμμής, και είμαι σίγουρος ότι καμία άλλη κρυφή μνήμη δεν έχει αντίγραφό της (τα πρωτόκολα που δεν έχουν αυτή την κατάσταση, περιλαμβάνουν αυτή την περίπτωση μέσα στην κατάσταση "S").
- **O** (Owned): είχα αλλάξει το περιεχόμενο αυτής της γραμμής (dirty), και γι' αυτό οφείλω (κάποτε μελλοντικά) να το γράψω πίσω (write back) στη μνήμη, αλλά εν τω μεταξύ υπάρχουν κι άλλες κρυφές μνήμες που ζήτησαν αντίγραφό της, και τους έχω δώσει, άρα ΔΕΝ είμαι πιά η μόνη που έχω αντίγραφο (τα πρωτόκολα που δεν έχουν αυτή την κατάσταση, περιλαμβάνουν αυτή την περίπτωση μέσα στην κατάσταση "S", αλλά υποχρεούνται να έκαναν το write back με το που η πρώτη "άλλη" κρυφή μνήμη μου ζήτησε αντίγραφο).

Όταν μιά κρυφή μνήμη ζητάει να διαβάσει μιά γραμμή, προφανώς λόγω αστοχίας, τότε, εκτός από τη κεντρική μνήμη, και όλες οι άλλες κρυφές μνήμες ακούν το αίτημα. Εάν κάποια άλλη κρυφή μνήμη έχει τη γραμμή, σπεύδει αυτή πρώτη να την δώσει, πριν το κάνει η κεντρική μνήμη: στην περίπτωση Shared (ή Exclusive) αυτό αποτελεί απλή επιτάχυνση, αλλά στην περίπτωση Modified (ή Owned) αυτό αποτελεί απαραίτητη προϋπόθεση ορθότητας, αφού η κεντρική μνήμη ΔΕΝ έχει την σωστή (πλέον πρόσφατη) τιμή!

Άσκηση 14.3: Πράξεις στην Αρτηρία, σε Snooping Coherence

Θεωρήστε δύο επεξεργαστές, τους A και B, και τις αντίστοιχες ιδιωτικές τους κρυφές μνήμες, που επικοινωνούν μεταξύ τους και με την κεντρική τους μνήμη με μιάν αρτηρία (bus). Οι A και B χρησιμοποιούν και οι δύο τη μεταβλητή sharedVar, και συμβαίνουν τα εξής κατά χρονική σειρά:

i. Αρχικά, η μεταβλητή sharedVar βρίσκεται μόνο στη μνήμη και έχει τιμή μηδέν (0).
ii. Ο επεξεργαστής A διαβάζει την sharedVar και κάνει υπολογισμούς με αυτήν.
iii. Ο επεξεργαστής B διαβάζει κι αυτός την sharedVar, και κάνει κι αυτός υπολογισμούς με αυτήν.
iv. Ο επεξεργαστής A εκτελεί την εκχώρηση sharedVar = 11;
v. Ο επεξεργαστής B ξαναδιαβάζει την sharedVar, γιά να κάνει κι άλλους

υπολογισμούς με αυτήν.

(α) Εάν ΔΕΝ υπάρχει πρωτόκολο συνοχής στην αρτηρία και στις κρυφές μνήμες, πόσες και ποιές πράξεις θα γίνουν πάνω στην αρτηρία, και σε ποιά από τα παραπάνω βήματα; (Δηλαδή, ποιά κρυφή μνήμη θα ζητήσει τι από την κεντρική μνήμη, γιατί, και πότε;) Στο βήμα (v), ο επεξεργαστής B, με τι τιμή της μεταβλητής sharedVar θα κάνει τους νέους του υπολογισμούς, και γιατί; Υποθέστε ότι οι κρυφές μνήμες είναι write-back, αλλά, προαιρετικά, σκεφτείτε (ή και γράψτε και στην απάντησή σας) τι θα άλλαζε εάν οι κρυφές μνήμες ήταν write-through.

(β) Έστω τώρα ότι οι A και B επικοινωνούν μεταξύ τους και με την *κοινόχρηστη* κεντρική τους μνήμη με πρωτόκολο snooping coherence πάνω στην αρτηρία. Απαντήστε πάλι πόσες και ποιές πράξεις θα γίνουν πάνω στην αρτηρία, και σε ποιά από τα παραπάνω βήματα; Στο βήμα (v), ο επεξεργαστής B, με τι τιμή της μεταβλητής sharedVar θα κάνει τους νέους του υπολογισμούς, και γιατί; Υποθέστε ότι το πρωτόκολο snooping coherence έχει τις βασικές καταστάσεις M, S, και I που είδαμε παραπάνω, αλλά, προαιρετικά, σκεφτείτε (ή και γράψτε και στην απάντησή σας) τι θα άλλαζε εάν το πρωτόκολο είχε επιπλέον και την κατάσταση E.

(γ) Έστω τώρα ότι το παραπάνω σενάριο αλλάζει, και περνάμε κατευθείαν από το βήμα (ii) στο βήμα (iv), χωρίς να συμβεί ενδιάμεσα η ανάγνωση (iii) της sharedVar από τον B. Σε αυτή την περίπτωση, πώς θα διαφέρουν μεταξύ τους (1) το πρωτόκολο συνοχής "MSI" που έχει μόνο τις καταστάσεις M, S, και I, από (2) το πρωτόκολο συνοχής "MESI" που έχει τις καταστάσεις M, E, S, και I;

Άσκηση 14.4: Instruction Scheduling: Στατικό (Compiler) και Δυναμικό (Out-of-order Execution)

(α) Θεωρήστε το εξής πρόγραμμα σε C (δύο εκχωρήσεις όλες κι όλες): "x = a+2; y = b-1;" όπου οι ακέραιες μεταβλητές a, b, x, και y βρίσκονται στη μνήμη, στις διευθύνσεις 120, 124, 128, και 132 αντίστοιχα (δεκαδικό). Μεταφράστε το πρόγραμμα αυτό σε Assembly του MIPS χωρίς "instruction scheduling", δηλαδή χωρίς να αναδιατάσετε την θέση των εντολών που αυτές έχουν "by default", βάσει του προγράμματος.

(β) Όταν το πρόγραμμα (α) εκτελείται στην "κλασσική pipeline" των 5 βαθμίδων, πόσοι κύκλοι ρολογιού χάνονται λόγω αναμονών (αλληλεξαρτήσεων) επόμενων εντολών από εντολές load; Κάντε τώρα instruction scheduling όπως θα έκανε ένας σύγχρονος compiler, δηλαδή αναδιατάξτε τις εντολές –χωρίς να αλλάξει η σημασιολογία του προγράμματος, φυσικά– ούτως ώστε να μην χάνονται αυτοί οι κύκλοι ρολογιού, και εξηγήστε εν συντομία γιατί επιτρέπεται αυτή η αναδιάταξη και γιατί δεν χάνονται οι κύκλοι ρολογιού.

(γ) Έστω τώρα η εξής παραλλαγή του προγράμματος (α): "(*px) = (*pa)+2; (*py) = (*pb)-1;" όπου οι μεταβλητές pa, pb, px, και py βρίσκονται στους καταχωρητές $12, $13, $14, και $15 αντίστοιχα, και είναι όλες τύπου pointers σε ακεραίους (άρα οι αυξήσεις και εκχωρήσεις του προγράμματος (α) τώρα γίνονται στις (ακέραιες) λέξεις μνήμης όπου δείχνουν αυτοί οι pointers). Μεταφράστε το πρόγραμμα αυτό σε Assembly του MIPS χωρίς instruction scheduling, και στη συνέχεια απαντήστε εάν ο compiler μπορεί και τώρα να κάνει instruction scheduling όπως στο ερώτημα (β) ή όχι. Εξετάστε χωριστά την περίπτωση που οι pointers px και pb έχουν διαφορετικές τιμές, και την περίπτωση που έχουν την ίδια τιμή: τι υπολογίζει το πρόγραμμα στην μία περίπτωση και τι στην άλλη; Αφού ο compiler δεν ξέρει την τιμή που θα έχουν οι pointers όταν θα εκτελείται το πρόγραμμα, επιτρέπεται να κάνει instruction scheduling όπως στο (β);

(δ) Έστω ότι εκτελούμε το πρόγραμμα (γ), που έχει γίνει compiled αναγκαστικά χωρίς instruction scheduling όπως είπαμε στο (γ), σε έναν επεξεργαστή pipelined, με **out-of-order (ooo)** execution, και ότι στο 1% των περιπτώσεων που το εκτελούμε οι pointers px και pb έχουν ίση τιμή, ενώ στις υπόλοιπες 99% των περιπτώσεων έχουν διαφορετική τιμή. Μετά

την πρώτη εντολή load, η εντολή addi πρέπει να περιμένει μέχρι να έλθουν τα δεδομένα που χρειάζεται από τη μνήμη, και η εντολή store πρέπει επίσης να περιμένει μέχρι να υπολογιστεί το (*pa)+2. Όμως, η επόμενη εντολή load, πότε υποχρεούται να περιμένει και πότε μπορεί να εκτελεστεί ανεξάρτητα; Θυμηθείτε ότι σ' έναν επεξεργαστή με εκτέλεση οοο, η κάθε εντολή περιμένει *μόνον* όταν εξαρτάται από κάποια προηγούμενή της που δεν έχει βγάλει ή κάνει ακόμα το αποτέλεσμα που χρειαζόμαστε. Εδώ, η επόμενη load από ποιάν προηγούμενη μπορεί να εξαρτάται, και υπό ποιά προϋπόθεση θα υπάρχει όντως αυτή η εξάρτηση; Πώς το hardware θα πετύχει να κάνει το ισοδύναμο του instruction scheduling που ο compiler δεν μπορούσε να κάνει; Πόσο συχνά θα χάνεται χρόνος λόγω αλληλεξαρτήσεων και πόσο συχνά δεν θα χάνεται;

Άσκηση 14.5: Πολυνημάτωση (Multithreading)

Έστω ένας επεξεργαστής με multithreading που έχει δύο νήματα στο υλικό του, το A και το B. Δηλαδή, (σχεδόν) όλα στον επεξεργαστή αυτόν είναι όπως και σ' έναν κανονικό, εκτός από το ότι έχει **δύο** καταχωρητές PC, τον PCA και τον PCB, και **δύο** αρχεία καταχωρητών, το RFA και το RFB. Ο PCA δείχνει στην επόμενη εντολή που πρέπει να εκτελεστεί από το πρόγραμμα A, και το RFA περιέχει την τρέχουσα τιμή των καταχωρητών του προγράμματος A. Ο PCB δείχνει στην επόμενη εντολή που πρέπει να εκτελεστεί από το πρόγραμμα B, και το RFB περιέχει την τρέχουσα τιμή των καταχωρητών του προγράμματος B.

Σε κάποια στιγμή, και ενώ ο επεξεργαστής μας εκτελεί εντολές του προγράμματος A (χρησιμοποιώντας τον PCA και το RFA), μιά εντολή load προκαλεί δυστυχώς αστοχία κρυφής μνήμης, και πρέπει να πάμε στην κεντρική μνήμη, πράγμα που θα μας κοστίσει π.χ. 80 κύκλους ρολογιού (σε αυτή την άσκηση). Χάρις στο instruction scheduling από τον compiler, και χάρις στην out-of-order-execution pipeline, ο επεξεργαστής βρίσκει χρήσιμες εντολές να εκτελέσει (οι οποίες προφανώς ΔΕΝ χρειάζονται τα data της load που καθυστερεί), και οι οποίες θα κρατήσουν (παραγωγικά) απασχολημένο τον επεξεργαστή γιά 8 από αυτούς τους 80 κύκλους αναμονής.

(α) Εάν ο επεξεργαστής δεν είχε multihtreading, πόσοι κύκλοι ρολογιού θα χάνονταν σε αυτό το σενάριο;

Επειδή όμως ο επεξεργαστής μας έχει mulithreading, με το που διαπιστώνει ότι η εντολή load του νήματος A προκαλεί αστοχία, αρχίζει να φέρνει (fetch) εντολές μέσω του PCB – άρα του προγράμματος B– και να τις εκτελεί χρησιμοποιώντας τους καταχωρητές RFB, μέχρι να απαντήσει η κεντρική μνήμη στην αστοχία της load του A, οπότε και ξαναγυρίζει ο επεξεργαστής στην εκτέλεση εντολών από το νήμα A.

(β) Το νήμα (πρόγραμμα) A τρέχει πιό γρήγορα (σε λιγότερο χρόνο) τώρα που υπάρχει multithreading απ' ό,τι πριν που δεν υπήρχε;

Όμως, **(γ)** τώρα που έχουμε multithreading, πόσοι κύκλοι ρολογιού θα χαθούν "συνολικά" στο παραπάνω σενάριο; "Συνολικά χαμένοι" κύκλοι σημαίνει κύκλοι που τίποτα χρήσιμο δεν προχωρά –ούτε εντολές του νήματος A, ούτε εντολές του νήματος B. Υποθέστε ότι το νήμα B είναι τυχερό, και τρέχει γιά 80 (τουλάχιστο) κύκλους χωρίς να του συμβεί καμία αστοχία της κρυφής μνήμης.

Τρόπος Παράδοσης

Παραδώστε τις απαντήσεις σας on-line, σε μορφή **PDF** (μόνον) (μπορεί να είναι κείμενο μηχανογραφημένο ή/και "σκαναρισμένο" χειρόγραφο, αλλά *μόνον* σε μορφή PDF). Παραδώστε μέσω **`turnin ex14@hy225 [directoryName]`** ένα αρχείο ονόματι **`ex14.pdf`** που θα περιέχει τις απαντήσεις σας.
