Τεχνολογικό Εκπαιδευτικό Ίδρυμα Κρήτης

Σχολή Τεχνολογικών Εφαρμογών Τμήμα Εφαρμοσμένης Πληροφορικής & Πολυμέσων

Πτυχιακή Εργασία

Τίτλος:

"Μελέτη και αξιολόγηση διαφορετικών τρόπων επικοινωνίας και συγχρονισμού σε παράλληλα και κατανεμημένα συστήματα"

Δημήτρης Ντόντος (ΑΜ: 976)

Επιβλέπων καθηγητής : Δρ. Γραμματικάκης Μίλτος
Επιτροπή Αξιολόγησης : Δρ. Φραγκοπούλου Βιβή
Δρ. Παπαδάκης Νίκος

Ημερομηνία παρουσίασης: Τρίτη, 17 Μάη 2011

ΕΥΧΑΡΙΣΤΙΕΣ

Με την ολοκλήρωση αυτής της εργασίας, θεωρώ καθήκον μου να ευχαριστήσω τους ανθρώπους που συνέβαλλαν στην ολοκλήρωση της, καθώς και στην ολοκλήρωση των σπουδών μου γενικότερα. Κατ' αρχήν θα ήθελα να εκφράσω την ευγνωμοσύνη μου στους γονείς μου για την συνεχή συμπαράσταση και ηθική υποστήριξη που μου παρείχαν καθ' όλη την διάρκεια των σπουδών μου. Ιδιαιτέρως θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κ. Γραμματικάκη Μίλτο για την υποστήριξη και καθοδήγηση του καθ' όλη την διάρκεια της υλοποίησης αυτής της πτυχιακής εργασίας.

Τέλος ευχαριστώ πολύ όλο το διδακτικό προσωπικό το Τ.Ε.Ι. Ηρακλείου που με δίδαξε τόσα χρόνια και αποκόμισα την υπέροχη αυτή ακαδημαϊκή εμπειρία ολοκληρώνοντας το πρόγραμμα σπουδών της σχολής 'Εφαρμοσμένης Πληροφορικής και Πολυμέσων', δίνοντάς μου την ευκαιρία να εξοπλίσω τον εαυτό μου με τις απαραίτητες γνώσεις που απαιτούνται στις μέρες μας για την επαγγελματική μου καταξίωση.

ΣΥΝΟΨΗ

Το θεματικό αντικείμενο της παρούσας πτυχιακής εργασίας η μελέτη διαφορετικών παράλληλων και κατανεμημένων προγραμματιστικών μοντέλων, όπως pthread, multithreading με shared memory και network programming (διεργασίες κοινής μνήμης με νήματα) και message passing (επικοινωνία με μηνύματα). Η μελέτη δίνει τη δυνατότητα στον ενδιαφερόμενο να δει και να καταλάβει πως λειτουργούν σε βάθος τα μοντέλα αυτά. Η περιγραφή και η ανάλυση τους γίνεται με γνώμονα την κατανόηση των μοντέλων αυτών προκειμένου ο αναλυτής να μπορεί να μάθει και να χρησιμοποιεί με όσο το δυνατόν πιο απλό και κατανοητό τρόπο τα προγραμματιστικά μοντέλα αυτά.

Για την επίτευξη του παραπάνω στόχου ακολουθήθηκαν επιμέρους βήματα τα οποία συνοψίζονται παρακάτω και τα οποία συνιστούν και τη μεθοδολογία που χρησιμοποιήθηκε στην παρούσα εργασία.

Πρώτα από όλα, στο μεθοδολογικό σκέλος της πτυχιακής μελετήθηκαν θέματα που αφορούν τις απαραίτητες θεωρητικές έννοιες ώστε να μπορέσουμε να προγραμματίσουμε απλές εφαρμογές με μοντέλα ανταλαγής μηνυμάτων και προγραμματισμού κοινής μνήμης με βάση νήματα.

Μετά έχοντας σα βάση σύγκρισης διάφορα υπολογιστικά προβλήματα που απαιτούν αρκετή επικοινωνία, όπως εσωτερικό γινόμενο, πολλαπλασιασμό πινάκων, πολλαπλασιασμό πίνακα με διάνυσμα και mergesort, κωδικοποιούμε αντίστοιχους αλγορίθμους με βάση διάφορες πρότυπες βιβλιοθήκες (MPI, pthreads standards) και εξετάζουμε σε περιβάλλον C/C++ και Linux πώς διάφορα μοντέλα επικοινωνίας και συγχρονισμού (και οι αντίστοιχες παράμετροι λειτουργίας τους) επηρεάζουν την απόδοση της εφαρμογής (application latency).

Με βάση την μεθοδολογία αυτή, η πτυχιακή δίνει τη δυνατότητα να εξάγουμε σημαντικά συμπεράσματα ως τον πιο αποδοτικό τρόπο υλοποίησης αυτών των προβλημάτων.

Abstract

The topic of this thesis is to study different parallel and distributed programming models, including message passing and shared memory multithreading. The study of parallel and distributed programming models at a theoretical level enables us to become familiar with these technologies. Moreover, our algorithm design and implementation for a set of scientific computation problems, including dot product, matrix multiplication, matrix-vector multiplication and mergesort has enabled understanding the underlying standard message passing and shared memory multithreading libraries.The analysis has provided a glimpse on the efficiency of Pthreads, Shared Memory multithreading and MPI libraries using C/C++ on a Linux platform, providing conclusions on performance benefits and limitations from these models for the above set of scientific computation problems.

ΠΕΡΙΕΧΟΜΕΝΑ

1. Εισαγωγή 8
2. Μεθοδολογία και Τεχνολογίες Υλοποίησης 10
3. Σχέδιο Δράσης 11
4. Προγραμματιστικά Μοντέλα 12
5. Βιβλιοθήκες 30
6. Αποτελέσματα Προγραμμάτων και Συμπεράσματα 62
7. Βιβλιογραφία 69
8. Εικόνες 70

Πίνακας Εικόνων:

Εικόνα 1-1: Πρωτόκολλο με Synchronous Communication.. 13
Εικόνα 1-2: Πρωτόκολλο με Synchronous Communication 14
Εικόνα 1-3: Πρωτόκολλο με αναστέλλουσα ασύγχρονη επικοινωνία (Blocking Asynchronous)........ 15
Εικόνα 1-4: Πρωτόκολλο με αναστέλλουσα ασύγχρονη επικοινωνία (Non-Blocking Asynchronous) 16
Εικόνα 2-1: Τυπικό προγραμματιστικό μοντέλο κοινής μνήμης 17
Εικόνα 2-3: Μοντέλο Ιεραρχικής Μνήμη.. 18
Εικόνα 2-4: Κοινή Μνήμη μέσω Διαύλου 19
Εικόνα 3-5: Dance-Hall 19
Εικόνα 3-6: Distributed-Memory 20
Εικόνα 3-5: Cache Coherence Problem 24
Εικόνα 4-1: Requirements of event synchronization through flags.. 26
Εικόνα 4-2: Orders among accesses without synchronization 27
Εικόνα 4-3: Σειρά εκτέλεσης προσπελάσεων μνήμης σε πρόγραμμα χωρίς συγχρονισμό 27
Εικόνα 4-4: Παράδειγμα ακολουθιακής συνέπειας.. 28
Εικόνα 5-4: Unix Process vs. Thread 32
Εικόνα 5-5: Concurrent Pthreads 34
Εικόνα 5-6: Κοινή Μνήμη με Threads 35
Εικόνα 5-7: Thread Safeness.. 36
Εικόνα 5-8: PThreads 37
Εικόνα 5-9: Join PThreads.. 38
Εικόνα 5-10: Interleaved memory cycles with two threads 40
Εικόνα 5-11: Two threads synchronizing memory access.. 41
Εικόνα 5-12: Two unsynchronized threads incrementing the same variable.. 41
Εικόνα 5-13: Αριστερά παρουσιάζεται η λειτουργία Reduce και δεξιά η λειτουργία AllReduce. 59
Εικόνα 5-14: Λειτουργία Broadcast 60
Εικόνα 5-15: Λειτουργίες Scatter και Gather 61
Εικόνα 6-1: Pthread MatrixProduct.. 62
Εικόνα 6-2: Pthread DotProduct 62
Εικόνα 6-3: Pthread MergeSort.. 63
Εικόνα 6-4: Pthread VectorProduct.. 63
Εικόνα 6-5: MPI MatrixProduct.. 64
Εικόνα 6-6: MPI DotProduct 64

Εικόνα 6-7: MPI MergeSort 65
Εικόνα 6-8: MPI MatrixVector 65
Εικόνα 6-9: Linux MatrixProduct 66
Εικόνα 6-10: Linux DotProduct 66
Εικόνα 6-11: Linux MergeSort 67
Εικόνα 6-12: Linux MatrixVector 67

1. Εισαγωγή

Στο κεφάλαιο που ακολουθεί θα παρουσιάσουμε γενικές πληροφορίες για τα εργαλεία που χρησιμοποιήσαμε για την ολοκλήρωση της πτυχιακής εργασίας και τους λόγους για τους οποίους τα επιλέξαμε.

1.1 Περίληψη

Η εργασία αυτή επικεντρώνεται σε τεχνικές παράλληλης και κατανεμημένης επεξεργασίας με βάση προγραμματιστικά μοντέλλα όπως network programming (διεργασίες κοινής μνήμης με νήματα), shared memory multithreading και message passing (επικοινωνία με μηνύματα). Θα χρησιμοποιηθεί C/C++ σε πλατφόρμα linux με διάφορες πρότυπες βιβλιοθήκες (pthreads, linux threads και MPI standards).

Έχοντας σα βάση σύγκρισης διάφορα υπολογιστικά προβλήματα που απαιτούν αρκετή επικοινωνία, εξετάζουμε σ' αυτό το περιβάλλον πώς διάφορα μοντέλα επικοινωνίας και συγχρονισμού (και οι αντίστοιχες παράμετροι λειτουργίας τους) επηρεάζουν την απόδοση της εφαρμογής (application latency) με στόχο να εξάγουμε συμπεράσματα ως τον πιο αποδοτικό τρόπο επεξεργασίας αυτών των προβλημάτων. Πιο συγκεκριμένα μέσα στο πλαίσιο αυτό θα μελετηθούν απλά υπολογιστικά προβλήματα.

1.2 Κίνητρο για τη διεξαγωγή της εργασίας

Το ενδιαφέρον για την παράλληλη αρχιτεκτονική είναι προφανές, δεδομένης της αυξανόμενης σημασίας των πολυεπεξεργαστών. Οι σχεδιαστές τσιπ πρέπει να καταλάβουν ότι αποτελεί ένα ζωτικό στοιχείο για την οικοδόμηση πολυεπεξεργαστικών συστήματων, ενώ οι σχεδιαστές των συστημάτων πληροφορικής πρέπει να καταλάβουν τον καλύτερο τρόπο αξιοποίησης ενός σύγχρονου μικροεπεξεργαστή και της τεχνολογίας μνήμης προκειμένου να φτιάξουν πολυεπεξεργαστικά συστήματα. Τομείς εφαρμογών, όπως γραφικά υπολογιστών και πολυμέσων, επεξεργασία συναλλαγών, συστήματα υποστήριξης λήψης αποφάσεων, επιστημονικοί υπολογισμοί και σχεδιασμός υπολογιστών είναι πιθανόν να δούν σημαντικές μεταβολές ως αποτέλεσμα της τεράστιας υπολογιστικής δύναμης διαθέσιμης σε χαμηλό κόστος μέσω των παράλληλων υπολογιστών. Ωστόσο, η ανάπτυξη παράλληλων εφαρμογών που είναι καλοφτιαγμένη και παρέχει καλή ταχύτητα σε τρέχοντα και μελλοντικά πολυεπεξεργαστικά συστήματα είναι ένα δύσκολο έργο, και απαιτεί μια βαθιά κατανόηση των κινητήριων δυνάμεων των παράλληλων υπολογιστών.

Η πτυχιακή επιδιώκει να παράσχει μια μερική κατανόηση αυτών των πραγμάτων ώστε να κάνει το έργο του προγραμματιστή πιο εύκολο και να βοηθήσει στην κατανόηση του πιο αποδοτικού τρόπου υλοποίησης.

1.3 Σκοπός και Στόχοι της Εργασίας

Στόχος της εργασίας είναι η μελέτη των κυριότερων προγραμματιστικών μοντέλων των παράλληλων υπολογιστώ, δίνοντας τη δυνατότητα σε οποιονδήποτε μηχανικό λογισμικού ή τελειόφοιτο φοιτητή πληροφορικής να μελετήσει την λειτουργία αυτών των μοντέλων. Η κατανόηση αυτών των μοντέλων γίνεται και με κωδικοποιώντας απλές εφαρμογές.

Σκοπός της εργασίας αυτής είνα να μελετήσουμε το σχεδιασμό και προγραμματισμό διαφόρων απλών εφαρμογών όπως εσωτερικό γινόμενο, πολλαπλασιασμό πινάκων, πολλαπλασιασμό πίνακα με διάνυσμα και mergesort σε περιβάλλον C/C++ και Linux με βάση μοντέλα Pthreads, shared memory multithreading και MPI.

1.4 Δομή εργασίας

Η δομή της εργασίας έχει ως εξής:

(α) Στο δεύτερο κεφάλαιο αναφερόμαστε στη μεθοδολογία που χρησιμοποιήσαμε για την υλοποίηση της πτυχιακής.

(β) Στο τρίτο κεφάλαιο επικεντρωνόμαστε στην έρευνα και στην εξέλιξη των παράλληλων συστημάτων.

(γ) Στο τέταρτο κεφάλαιο εξετάζουμε τα διάφορα προγραμματιστικά μοντέλα που χρησιμοποιούμε και πώς αυτά λειτουργούν.

(ε) Στο έκτο κεφάλαιο αναφερόμαστε στά API’s που χρησιμοποιήσαμε για τη διεξαγωγή του προγραμματιστικού μέρους της πτυχιακής

(ζ) Στο έβδομο κεφάλαιο αναφερόμαστε στα διαγράμματα από τα διάφορα προγράμματα που μελετήσαμε χρησιμοποιώντας διαφορετικά προγραμματιστικά μοντέλα. Επίσης αναγράφονται τα συμπεράσματά μας από την πτυχιακή.

2. Μεθοδολογία και Τεχνολογίες Υλοποίησης

- Linux: Η ονομασία Linux, που στα ελληνικά προφέρεται Λίνουξ, είναι ένας γενικός όρος αναφοράς σε λειτουργικά συστήματα που βασίζονται στον πυρήνα Linux. Η αρχιτεκτονική του Linux είναι παρόμοια με αυτή του λειτουργικού Unix αλλά έχει αναπτυχθεί εκ του μηδενός και δεν περιλαμβάνει κώδικα από το Unix. Η ανάπτυξη του Linux είναι χαρακτηριστικό παράδειγμα εθελοντικής συνεργασίας από διαδικτυακές κοινότητες, ενώ όλο το έργο είναι ανοικτού κώδικα και ελεύθερα προσβάσιμο από όλους για αντιγραφή, τροποποίηση ή αναδιανομή χωρίς περιορισμό. Το Linux είναι διαθέσιμο υπό άδειες όπως η GNU General Public License.Το Linux μπορεί να εγκατασταθεί και να λειτουργήσει σε μεγάλη ποικιλία υπολογιστικών συστημάτων, από μικρές συσκευές όπως κινητά τηλέφωνα μέχρι μεγάλα υπολογιστικά συστήματα και υπερυπολογιστές

- MPICH: είναι μια ελεύθερα διαθέσιμη, φορητή βιβλιοθήκη ανταλλαγής μηνυμάτων (με βάση το MPI), ένα πρότυπο για την ανταλλαγή μηνυμάτων μιας κατανεμημένης εφαρμογής μνήμης που χρησιμοποιούνται σε παράλληλα συστήματα. Το MPICH είναι Ελεύθερο Λογισμικό και είναι διαθέσιμο για τις περισσότερες διανομές του Unix (Linux και Mac OS X) και για τα Microsoft Windows. Επιπλέον, το MPICH είναι μια αναπτυγμένη βιβλιοθήκη προγραμμάτων. MPICH σημαίνει Message Passing Interface Chameleon. Η αρχική υλοποίηση του MPICH ονομάζεται MPICH1 και υλοποιεί το MPI-1.1 πρότυπο. Από το 2006, η τελευταία εφαρμογή ονομάζεται MPICH2 και υλοποιεί το MPI-2.0 πρότυπο που περιλαμβάνει και παράλληλο I/O.

- Τεχνολογία Νήματος: Στην επιστήμη υπολογιστών, ένα νήμα εκτέλεσης προκύπτει από ένα “fork” ενός προγράμματος σε δύο ή περισσότερα παράλληλα εκτελούμενα έργα. Η υλοποίηση των νημάτων και διεργασιών διαφέρει από το ένα λειτουργικό σύστημα στο άλλο, αλλά στις περισσότερες περιπτώσεις ένα νήμα περιέχεται σε μια διεργασία. Πολλαπλά νήματα μπορούν να υπάρχουν μέσα στην ίδια διεργασία, και μοιράζονται τους πόρους του συστήματος όπως π.χ. η μνήμη, ενώ άλλες διεργασίες δεν μοιράζονται τους πόρους αυτούς. (Εκτενέστερη ανάλυση γίνεται στο κεφάλαιο Βιβλιοθήκες Νημάτων όπου εξετάζονται και οι αντίστοιχες βιβλιοθήκες Pthreads και Linux threads)

3. Σχέδιο Δράσης

3.1 State of the art

Ο παράλληλος προγραμματισμός ήταν ένα κρίσιμο συστατικό της τεχνολογίας πληροφορικής τη δεκαετία του '90 και παραμένει ακόμα, και είναι πιθανό να έχει τον ίδιο αντίκτυπο τα επόμενα είκοσι χρόνια, όπως οι μικροεπεξεργαστές είχαν κατά τα τελευταία είκοσι χρόνια. Η εξέλιξη του υψηλού βαθμού ολοκλήρωσης μικροεπεξεργαστών και των τσιπ μνήμης καθιστά τα συστήματα των πολυεπεξεργαστών όλο και πιο ελκυστικά. Για να αποκτήσουμε ένα σημαντικό εύρος απόδοσης, η απλούστερη προσέγγιση είναι να αυξήσουμε τον αριθμό των επεξεργαστών, από οικονομικής άποψης αυτό το κάνει εξαιρετικά ελκυστικό. Πολύ σύντομα, δεκάδες επεξεργαστές θα χωρούν σε ένα μόνο chip.

Παρά το γεγονός ότι τα παράλληλα συστήματα έχουν μια μακρά και πλούσια ακαδημαϊκή ιστορία, η στενή σύνδεση με τα ενοποιημένα εμπορικά προϊόντα τεχνολογίας έχει αλλάξει ριζικά την κατάσταση. Η έμφαση σε ριζικές αρχιτεκτονικές και τεχνολογίες έχει δώσει τη θέση σε ποσοτική ανάλυση και προσεκτικές ανταλλαγές μηχανικής. Στόχος μας στο γράψιμο αυτης εργασίας είναι η κατανόηση θεμελιωδών θεμάτων αρχιτεκτονικής και διαθέσιμων προγραμματιστικών τεχνικών ώστε να αναπτύξουμε ένα κοινό πλαίσιο για την κατανόηση και αξιολόγηση προγραμματιστικών λύσεων.

Το παράλληλο λογισμικό έχει ωριμάσει σε σημείο όπου τα δημοφιλή μοντέλα παράλληλου προγραμματισμού είναι διαθέσιμα σε μια ευρεία γκάμα μηχανών και υπάρχει ουσιαστικά ένα κρήσιμο σημείο αναφοράς επιδόσεων. Αυτή η ωρίμανση του τομέα επιτρέπει να προβούμε τόσο σε ποσοτική, καθώς και ποιοτική μελέτη των αλληλεπιδράσεων υλικού / λογισμικού. Αν και στη παρούσα εργασία δε θα επικεντρωθούμε τόσο σε θέματα αρχιτεκτονικής, οι συγκρίσεις διαφορετικά προγραμματιστικά μοντέλα βοηθούν στην εξαγωγή αρχικών συμπερασμάτων που αφορούν την υλοποίηση των βιβλιοθηκών σε συγκεκριμένα συστήματα Linux.

4. Προγραμματιστικά Μοντέλα

4.1 Μοντέλο Ανταλλαγής Μηνυμάτων (Message Passing Model)

Με τον όρο ανταλλαγή μηνυμάτων (Message Passing) στην επιστήμη της πληροφορικής εννοούμε μια μορφή επικοινωνίας η οποία χρησιμοποιείται στον παράλληλο προγραμματισμό, στον αντικειμενοστραφή προγραμματισμό και στην δια-διεργασιακή επικοινωνία (Interprocess Communication, IPC). Στο μοντέλο αυτό διεργασίες ή αντικείμενα μπορούν να λαμβάνουν ή και να στέλνουν μηνύματα από/σε άλλες διεργασίες. Τα μηνύματα αυτά μπορούν να αποτελούνται από ένα ή περισσότερα byte, πολύπλοκες δομές δεδομένων ή ακόμα και να εμπεριέχουν τμήματα κώδικα. Επίσης τα μηνύματα διεργασιών μπορούν να συγχρονιστούν μεταξύ τους με το να περιμένουν για ένα συγκεκριμένο χρονικό διάστημα ή για ένα διακριτό γεγονός (discrete event).

4.1.1 Επισκόπηση Μοντέλου

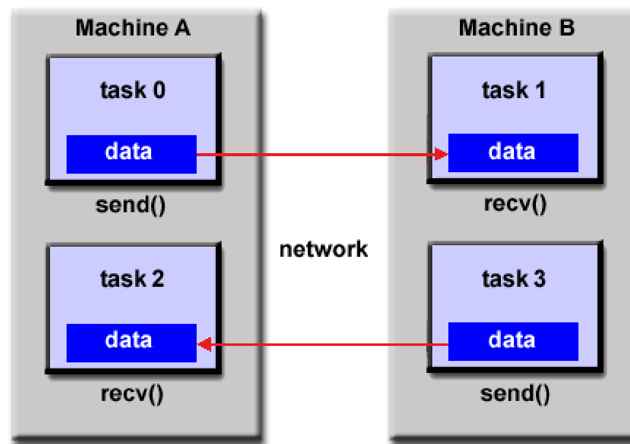
Το μοντέλο ανταλλαγής μηνυμάτων είναι μια μέθοδος επικοινωνίας όπου τα μηνύματα στέλνονται από τον αποστολέα σε ένα ή περισσότερους αποδέκτες. Τύποι μηνυμάτων περιλαμβάνουν (απομακρυσμένες) κλήσεις μεθόδων, σημάτων, και πακέτων δεδομένων. Όταν σχεδιάζουμε ένα σύστημα ανταλλαγής μηνυμάτων πολλές επιλογές εξαρτώνται από διάφορες συνθήκες και περιορισμούς, όπως παρατίθενται παρακάτω.

- Αν τα μηνύματα ταξιδεύουν αξιόπιστα, δηλαδή στέλνονται με αξιόπιστο πρωτόκολλο, το οποίο μας παρέχει ιδιότητες αξιόπιστες και σεβασμό παράδοσης των δεδομένων στους παραλήπτες.
- Αν τα μηνύματα από ένα ή πολλούς αποστολείς είναι εγγυημένα ότι θα παραδοθούν σε κάποια σειρά.
- Αν τα μηνύματα αποστέλλονται ένα-ένα, ένα προς πολλά (multicasting or broadcasting) ή πολλά προς ένα (client-server).
- Αν η επικοινωνία είναι σύγχρονη ή ασύγχρονη (synchronous, asynchronous).
- Αν η επικοινωνία είναι … (blocking, nonblocking).

4.1.2 Χαρακτηριστικά Μοντέλου

Το μοντέλο ανταλλαγής μηνυμάτων έχει να μας παρουσιάσει τα παρακάτω χαρακτηριστικά:

- Ένα πλήθος καθηκόντων (tasks) τα οποία χρησιμοποιούν τη δική τους τοπική μνήμη κατά την διάρκεια των υπολογισμών. Πολλαπλά καθήκοντα μπορούν να συνυπάρχουν στο ίδιο φυσικό μηχάνημα καθώς και σε ένα αυθαίρετο αριθμό μηχανών, ενώ κάθε καθήκον έχει δικά του δεδομένα.
- Τα διάφορα καθήκοντα επικοινωνούν και ανταλλάσουν δεδομένα με το να στέλνουν και να λαμβάνουν μηνύματα.
- Η μεταφορά δεδομένων συνήθως απαιτεί συνεργάσιμες λειτουργίες να λαμβάνουν μέρος από κάθε διεργασία. Για παράδειγμα μια λειτουργία αποστολής πρέπει να έχει μια αντίστοιχη λειτουργία αποδοχής.



Εικόνα 1-1: Μοντέλο Message Passing

Επιπλέον το μοντέλο ανταλλαγής μηνυμάτων έχει αποκτήσει ευρεία χρήση στο παράλληλο προγραμματισμό εξαιτίας των πλεονεκτημάτων που προσφέρει:

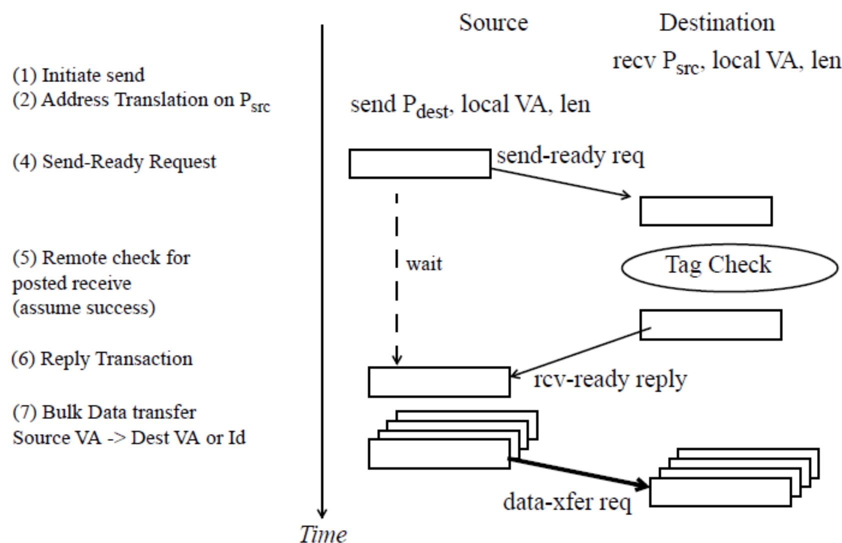
- Καθολικότητα
  Το μοντέλο ανταλλαγής μηνυμάτων προσαρμόζεται καλά σε ξεχωριστούς επεξεργαστές που συνδέονται από ένα (αργό ή γρήγορο) δίκτυο επικοινωνίας. Επιπλέον ταιριάζει με το hardware των περισσοτέρων σημερινών παράλληλων υπερυπολογιστών (supercomputers) καθώς επίσης και των σταθμών δικτύων και των συμπλεγμάτων PC τα οποία συναγωνίζονται τους υπερυπολογιστές. Εκεί που το μηχάνημα παρέχει επιπλέον hardware για να υποστηρίξει ένα μοντέλο μοιραζόμενης-μνήμης (shared-memory model) το μοντέλο ανταλλαγής μηνυμάτων παίρνει ως πλεονέκτημα την επιπλέον μνήμη για να αυξήσει τη μεταφορά δεδομένων.

- Εκφραστικότητα
  Το μοντέλο ανταλλαγής μηνυμάτων έχει αποδειχθεί ένα χρήσιμο και ολοκληρωμένο μοντέλο για παράλληλους αλγορίθμους. Παρέχει έλεγχο που απουσιάζει από τα παράλληλα-δεδομένα και τα μεταγλωττισμένα μοντέλα σε σχέση με τη τοποθεσία των δεδομένων. Κάποιοι βρίσκουν την ανθρωπόμορφη του φύση χρήσιμη για τη διαμόρφωση ενός παράλληλου αλγορίθμου. Είναι κατάλληλο για να προσαρμόζεται, να αυτο-προγραμματίζει αλγορίθμους καθώς και ιδανικό σε προγράμματα τα οποία μπορούν να είναι ανεκτικά σε έλλειψη ισορροπίας της ταχύτητας των διαδικασιών οι οποίες χρησιμοποιούν κοινά δίκτυα.

- Ευκολία στο Debugging
  Το Debugging παράλληλων προγραμμάτων είναι σύνθετη διαδικασία και αποτελεί μια περιοχή έρευνας με πολλές προκλήσεις. Ενώ οι debuggers για παράλληλα προγράμματα είναι ευκολότερο να γραφτούν για το μοντέλο της κοινής μνήμης (shared-memory model) είναι κοινώς αποδεκτό ότι η διαδικασία του debugging είναι ευκολότερη για το χρήστη στο μοντέλο ανταλλαγής μηνυμάτων. Αυτό γιατί μια από τις κυριότερες αιτίες λάθους είναι η απρόσμενη επικάλυψη της μνήμης. Αντίθετα το μοντέλο ανταλλαγής μηνυμάτων με το να ελέγχει τις αναφορές της μνήμης με περισσότερη σαφήνεια από κάθε άλλο μοντέλο, αφού μόνο μια διεργασία έχει άμεση πρόσβαση σε κάθε τοποθεσία στη μνήμη, είναι ευκολότερο να εντοπίσει λανθασμένα reads και writes.

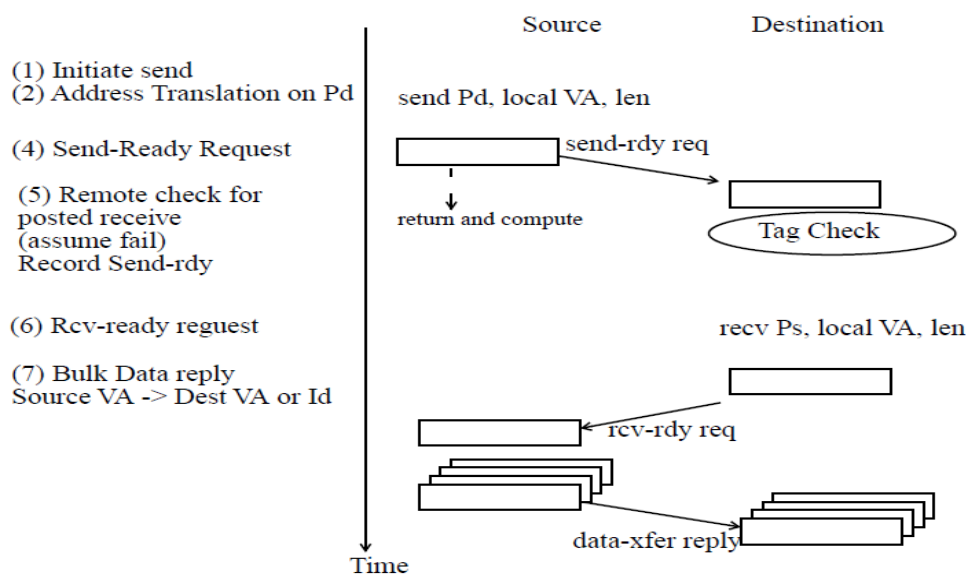
- Απόδοση
  Το μοντέλο ανταλλαγής μηνυμάτων παρέχει ένα τρόπο για το προγραμματιστή να συνδέει με σαφήνεια συγκεκριμένα δεδομένα με διεργασίες και συνεπώς να επιτρέψει στον μεταγλωττιστή και τη διαχείριση της μνήμης cache να λειτουργήσουν καλύτερα. Πράγματι ένα πλεονέκτημα των υπολογιστικών συστημάτων με κατανεμημένη μνήμη είναι ότι παρέχουν περισσότερη μνήμη και cache ακόμη και από τους μεγαλύτερους υπολογιστές με μόνο ένα επεξεργαστή (uniprocessors). Εφαρμογές που βασίζονται στη μνήμη μπορούν έτσι να επιδείξουν υπεργραμμικές επιταχύνσεις (superliner speedup), όταν κωδικοποιηθούν σε τέτοιες μηχανές. Ακόμα και σε συστήματα με κοινή μνήμη, η χρήση του μοντέλου ανταλλαγής μηνυμάτων μπορεί να βελτιώσει την απόδοση με το να παρέχει καλύτερο έλεγχο της τοποθεσίας των δεδομένων στην ιεραρχία της μνήμης. Αυτή η ανάλυση εξηγεί γιατί το μοντέλο ανταλλαγής έχει αναδυθεί ως ένα από τα πιο χρησιμοποιούμενα προγραμματιστικά μοντέλα για να εκφράσει παράλληλους αλγορίθμους. Παρόλο που έχει τα ελαττώματα του, το μοντέλο ανταλλαγής μηνυμάτων έρχεται κοντύτερα από κάθε άλλο μοντέλο στο να είναι μια συχνή και σταθερή προσέγγιση για την υλοποίηση παράλληλων αλγορίθμων.

- Συγχρονισμένη Επικοινωνία (Synchronous Communication)
  Τα συστήματα ανταλλαγής μηνυμάτων συγχρονισμένης επικοινωνίας (ssend) απαιτούν τον αποστολέα και τον παραλήπτη να περιμένει ο ένας τον άλλο για τη μεταφορά του μηνύματος. Δηλαδή, ο αποστολέας δεν θα συνεχίσει έως ότου ο δέκτης έχει λάβει το μήνυμα. Η συγχρονισμένη επικοινωνία έχει δύο πλεονεκτήματα. Το πρώτο πλεονέκτημα είναι ότι η λογική σχετικά με το πρόγραμμα μπορεί να απλοποιηθεί κατά το ότι υπάρχει ένα σημείο συγχρονισμού μεταξύ του αποστολέα και του παραλήπτη στη μεταφορά μηνύματος. Το δεύτερο πλεονέκτημα είναι ότι δεν απαιτείται ενδιάμεση μνήμη (buffering). Το μήνυμα μπορεί πάντα να αποθηκευτεί στη μεριά του αποστολέα, επειδή ο αποστολέας θα συνεχίσει την αποστολή μόνο αν ο παραλήπτης είναι έτοιμος.



Εικόνα 1-2: Πρωτόκολλο με Synchronous Communication

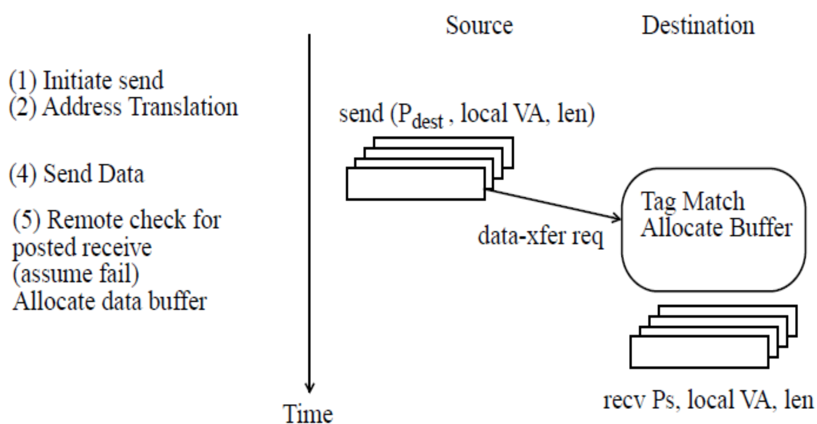
- Αναστέλλουσα Ασύγχρονη Επικοινωνία (Blocking Asynchronous)
  Η αναστέλλουσα ασύγχρονη διεργασία/κλήση αποστολής (blocking asend) επιστρέφει τον έλεγχο στην καλούσα διεργασία όταν το μήνυμα έχει ληφθεί από τη δομή δεδομένων της αποστέλλουσας εφαρμογής και είναι συνεπώς στη φροντίδα του τοπικού συστήματος αποστολής μηνυμάτων (network interface). Αυτό σημαίνει ότι το σύστημα μπορεί να χρησιμοποιήσει εκ νέου τη δομή προέλευσης των δεδομένων, χωρίς να επηρεαστεί το μήνυμα. Σε σύγκριση με τη διεργασία αποστολής/κλήσης στη συγχρονισμένη επικοινωνία (ssend), επιτρέπεται η αποστολή του μηνύματος να γίνει συντομότερα, αλλά δεν εγγυάται ότι το μήνυμα θα φτάσει στον δέκτη. Λαμβάνοντας υπόψη μία τέτοια διαβεβαίωση θα απαιτούσαμε μια επιπλέον εγγύηση. Η διεργασία «κλήση δέκτης» μίας αναστέλλουσας ασύγχρονης επικοινωνίας είναι η εγγύηση που περιγράψαμε πριν, μόνο που τα δεδομένα/μηνύματα που λαμβάνει έχουν αφαιρεθεί από την προσωρινή μνήμη του συστήματος και έχουν τοποθετηθεί σε μία καθορισμένη διεύθυνση της εφαρμογής. Μόλις ο δέκτης έχει αυτό το μήνυμα, η εφαρμογή μπορεί άμεσα να χρησιμοποιήσει τα δεδομένα στην προκαθορισμένη προσωρινή μνήμη της εφαρμογής. Αντίθετα με το δέκτη της συγχρονισμένης επικοινωνίας, ο δέκτης δεν αποστέλλει ειδοποίηση στον παραλήπτη ότι έλαβε το μήνυμα.



Εικόνα 1-3: Πρωτόκολλο με αναστέλλουσα ασύγχρονη επικοινωνία (Blocking Asynchronous)

- Μη-Αναστέλλουσα Ασύγχρονη Επικοινωνία (Non-Blocking Asynchronous)
  Οι μη-αναστέλλουσες ασύγχρονες διεργασίες/κλήσεις, αποστολή (nonblocking asend) και λήψη (nonblocking arecv), επιτρέπουν τη μεγαλύτερη επικάλυψη μεταξύ ανταλλαγής μηνυμάτων και υπολογισμών με το να επιστρέφουν τον έλεγχο στην καλούσα διεργασία. Η μη-αναστέλλουσα διεργασία «κλήση αποστολέας» επιστρέφει τον έλεγχο άμεσα. Ο μη-αναστέλλων δέκτης επιστρέφει τον έλεγχο αφού απλά εκδηλώσει το ενδιαφέρον να ελέγξει την ακριβή παραλαβή του μηνύματος και τοποθέτηση του σε μια συγκεκριμένη δομή εφαρμογής από δεδομένα ή μια

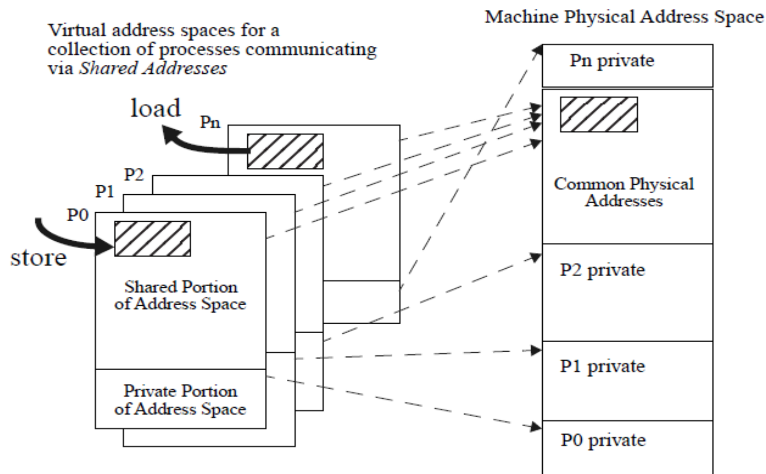
προσωρινή μνήμη (buffer). Οι διαδικασίες αυτές εκτελούνται ασύγχρονα σε ακαθόριστο χρόνο στο σύστημα, με βάση τον αναρτημένο δέκτη. Και στις δύο μη-αναστέλλουσες περιπτώσεις του δέκτη και του παραλήπτη, η επιστροφή του ελέγχου δεν υποδηλώνει τίποτα για την κατάσταση του μηνύματος ή για την εφαρμογή της δομής δεδομένων που χρησιμοποιεί, γι' αυτό είναι ευθύνη του χρήστη να καθορίσει την κατάσταση όταν είναι απαραίτητο. Η κατάσταση μπορεί να προσδιοριστεί με χωριστές κλήσεις σε αρχέτυπα που εξετάζουν (ερωτούν) για την κατάσταση. Για αυτό το λόγο τα μη-αναστέλλοντα μηνύματα τυπικά χρησιμοποιούνται με δύο τρόπους: πρώτα ως λειτουργίες send/receive από μόνα τους και μετά ως αρχέτυπα. Τα αρχέτυπα τα οποία πρέπει να δοθούν από τη βιβλιοθήκη του μοντέλου ανταλλαγής μηνυμάτων μπορεί να ανασταλούν μέχρι να παρατηρηθεί η απαιτούμενη κατάσταση, ή να επιστρέψουν τον έλεγχο άμεσα και απλά να αναφέρουν ποιά κατάσταση παρατηρήθηκε.



Εικόνα 1-4: Πρωτόκολλο με αναστέλλουσα ασύγχρονη επικοινωνία (Non-Blocking Asynchronous)

4.2 Μοντέλο Κοινής Μνήμης(Shared-Memory)

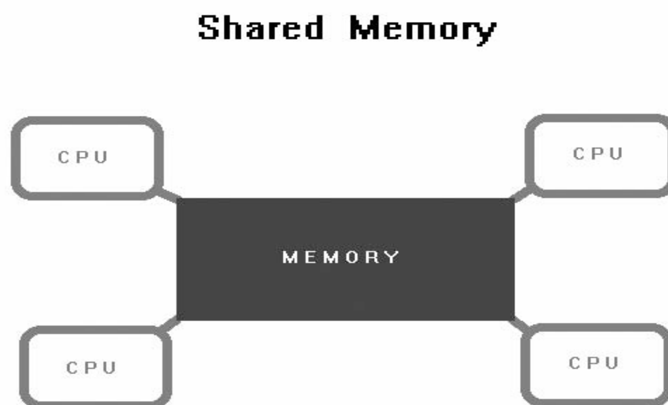
Στην πληροφορική, κοινόχρηστη μνήμη είναι η μνήμη στην οποία μπορεί να υπάρχει πρόσβαση ταυτόχρονα από πολλά προγράμματα, με πρόθεση να παρέχει επικοινωνία μεταξύ τους ή να αποφευχθούν περιττά αντίγραφα. Ανάλογα με την περίπτωση, τα προγράμματα μπορούν να τρέξουν σε έναν επεξεργαστή ή σε πολλαπλούς χωριστούς επεξεργαστές. Η χρήση της μνήμης για την επικοινωνία των πολλαπλών νημάτων (threads) ενός απλού προγράμματος επίσης αναφέρεται ως κοινόχρηστη μνήμη.



Εικόνα 2-1: Τυπικό προγραμματιστικό μοντέλο κοινής μνήμης

Στην Εικόνα 2-1 απεικονίζεται συλλογή διεργασιών έχουν μία κοινή περιοχή φυσικών διευθύνσεων που αντιστοιχίζεται στον εικονικό χώρο διευθύνσεων τους, καθώς και την ιδιωτική τους περιοχή, η οποία και τυπικά περιέχει τη στοίβα και τα ιδιωτικά δεδομένα. Όπως παρατηρούμε από το παραπάνω σχήμα ενώ η κοινόχρηστη μνήμη μπορεί να χρησιμοποιηθεί για επικοινωνία ανάμεσα σε συλλογές από αυθαίρετους επεξεργαστές, τα περισσότερα παράλληλα προγράμματα είναι αρκετά δομημένα στο πως διαχειρίζονται την ιδεατή κοινόχρηστη μνήμη. Τυπικά η πρόσβαση στη μνήμη γίνεται μέσα από κοινή εικόνα του κώδικα, τμήματα τοπικής μνήμης με ατομικά δεδομένα και κοινόχρηστα τμήματα τα οποία είναι στην ίδια περιοχή της ιδεατής περιοχής διευθύνσεων κάθε διεργασίας ή νήματος (thread) του προγράμματος. Αυτή η απλή δομή συνεπάγεται ότι οι ατομικές μεταβλητές στο πρόγραμμα είναι παρούσες σε κάθε διεργασία και ότι οι κοινόχρηστες μεταβλητές έχουν την ίδια διεύθυνση και σημασία σε κάθε διεργασία ή νήμα του προγράμματος.

4.2.1 Επισκόπηση Μοντέλου Κοινής Μνήμης



Εικόνα 2-2: Μοντέλο Κοινής Μνήμης

- Στο μοντέλο κοινόχρηστης μνήμης, τα καθήκοντα μοιράζονται τον ίδιο χώρο διευθύνσεων, από τον οποίο διαβάζουν και γράφουν ασύγχρονα. Πολλαπλοί επεξεργαστές λειτουργούν ανεξάρτητα αλλά μοιράζονται τους ίδιους πόρους μνήμης.

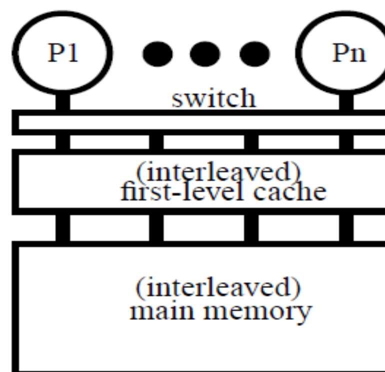
- Πολλαπλοί μηχανισμοί όπως locks (δέσμευση), semaphores (σηματοφορείς) και condition variables μπορούν να χρησιμοποιηθούν προκειμένου να ελέγξουμε τη πρόσβαση στη μνήμη. Μόνο ένας επεξεργαστής μπορεί να έχει πρόσβαση στη περιοχή της μνήμης κάθε φορά.
- Ένα πλεονέκτημα αυτού του μοντέλου είναι ότι από την άποψη του προγραμματιστή η ιδέα της "ιδιοκτησίας" των δεδομένων δεν υπάρχει, για αυτό υπάρχει η ανάγκη να προσδιορίσουμε σαφώς την επικοινωνία των δεδομένων μεταξύ των καθηκόντων. Προγράμματα ανάπτυξης παράλληλου κώδικα μπορούν συχνά έτσι να απλοποιηθούν.
- Ένα σημαντικό μειονέκτημα όσον αναφορά τις επιδόσεις είναι ότι γίνεται πιο δύσκολο να καταλάβουμε και να διαχειριστούμε τη τοποθεσία των δεδομένων.
    - ✓ Η διατήρηση των δεδομένων τοπικά στον επεξεργαστή ο οποίος εργάζεται εκείνη τη στιγμή διατηρεί και προσβάσεις στη μνήμη, η cache ανανεώνεται και δημιουργείται κυκλοφοριακό πρόβλημα όταν πολλοί επεξεργαστές χρησιμοποιούν τα ίδια δεδομένα.
    - ✓ Δυστυχώς, ο έλεγχος της τοποθεσίας των δεδομένων είναι δύσκολο να κατανοηθεί και ακόμη περισσότερο ο έλεγχος του μέσου χρήστη.

4.2.2 Χαρακτηριστικά Μοντέλου

Το μοντέλο κοινόχρηστης μνήμης ή κοινής μνήμης (shared memory) παρουσιάζει τα παρακάτω χαρακτηριστικά:

Οι ιεραρχίες μνήμης στους πολυεπεξεργαστές χωρίζονται σε τέσσερις κατηγορίες οι οποίες ανταποκρίνονται στην υπάρχουσα κατηγοριοποίηση των πολυεπεξεργαστών. Αυτές είναι:

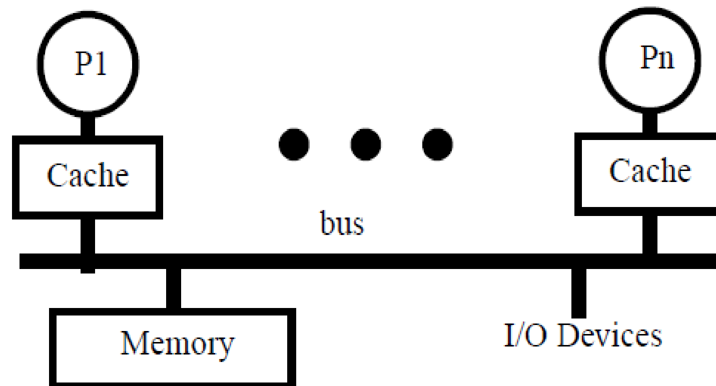
- Η πρώτη κατηγορία είναι η προσέγγιση της "Shared Cache". Το δίκτυο διασύνδεσης τοποθετείται μεταξύ των επεξεργαστών και του πρώτου επίπεδου κοινόχρηστης κρυφής μνήμης, το οποίο με τη σειρά του συνδέεται με το υποσύστημα της κοινόχρηστης κύριας μνήμης. Τόσο η κρυφή μνήμη όσο και η κύρια μνήμη ενδέχεται να διασυνδέονται επίσης για να αυξήσουν το διαθέσιμο εύρος ζώνης. Αυτή η προσέγγιση έχει χρησιμοποιηθεί για να συνδέει πολύ μικρούς αριθμούς επεξεργαστών (2-8). Ωστόσο εφαρμόζεται σε πολύ μικρή κλίμακα, τόσο γιατί η αλληλοσύνδεση μεταξύ των επεξεργαστών και της κοινόχρηστης κρυφής μνήμης πρώτου επιπέδου είναι στο κρίσιμο μονοπάτι που καθορίζει την καθυστέρηση πρόσβασης της κρυφής μνήμης καθώς και επειδή η κοινόχρηστη κρυφή μνήμη πρέπει να παραδώσει τεράστιο εύρος ζώνης στους επεξεργαστές που συνδέονται σε αυτό.



Εικόνα 2-3: Μοντέλο Ιεραρχικής Μνήμη

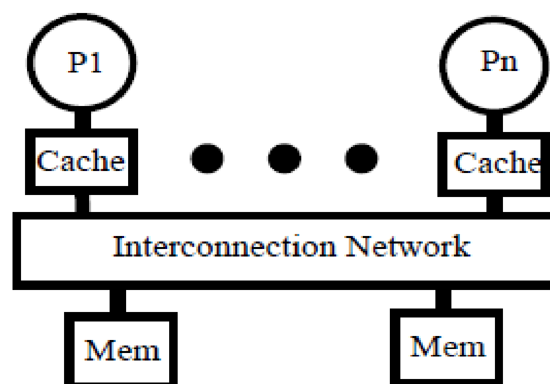
- Στη δεύτερη κατηγορία "Bus-Based Shared Memory" η διασύνδεση μοιράζεται μεταξύ του επεξεργαστή και της ιδιωτικής κρυφής μνήμης και του υποσυστήματος της κοινόχρηστης μνήμης . Αυτή η προσέγγιση έχει ευρέως χρησιμοποιηθεί για μικρές έως μεσαίας κλίμακας

πολυεπεξεργαστών απαρτίζοντας έως 20 με 30 επεξεργαστές. Είναι η κυρίαρχη μορφή παράλληλων μηχανών που πωλούνται σήμερα, και ουσιαστικά μετά από σημαντική προσπάθεια σχεδιασμού όλα οι σύγχρονοι μικροεπεξεργαστές είναι σε θέση να στηρίξουν τις διαμορφώσεις της συναφούς (ή συνεκτικής) κοινόχρηστης κρυφής μνήμης "cache coherence". Η επεκτασιμότητα έχει όριο για αυτές τις μηχανές κυρίως λόγω των περιορισμών του εύρου ζώνης (bandwidth) του κοινόχρηστου διαύλου.



Εικόνα 2-4: Κοινή Μνήμη μέσω Διαύλου

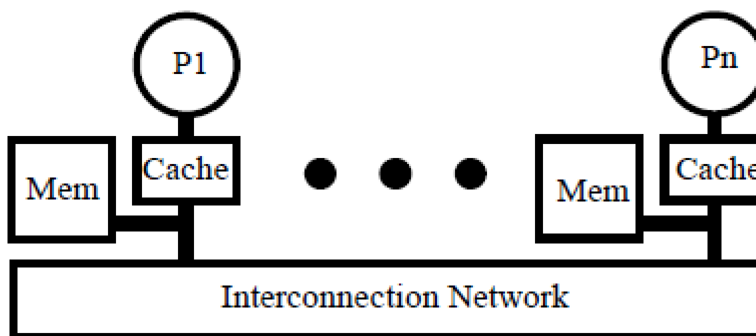
- Οι δύο τελευταίες κατηγορίες είναι επεκτάσιμες σε πολλούς κόμβους. Η τρίτη κατηγορία, "Dance-Hall" τοποθετεί επίσης την διασύνδεση ανάμεσα στις κρυφές μνήμες και τη κύρια μνήμη, αλλά με τη διαφορά ότι η διασύνδεση είναι ένα επεκτάσιμο δίκτυο με βάση τοπολογίες σημείο-σε-σημείο (point-to-point) και η μνήμη διαιρείται σε πολλά λογικά υποσύνολα που συνδέονται σε διαφορετικά λογικά σημεία διασύνδεσης. Η προσέγγιση αυτή είναι συμμετρική, όλη η κύρια μνήμη είναι ομοιόμορφα μακριά από όλους τους επεξεργαστές, αλλά ο περιορισμός της, είναι ότι όλη η μνήμη είναι πράγματι πολύ μακριά από όλους τους επεξεργαστές: Ειδικά σε μεγάλα συστήματα, θα πρέπει να διέλθει κανείς στη διασύνδεση πολλά "hops" ή διακόπτες για να καταλήξει σε μια μονάδα μνήμης.



Εικόνα 3-5: Dance-Hall

- Στη τέταρτη κατηγορία βρίσκεται η προσέγγιση της κατανεμημένης μνήμης (distributed memory) η οποία δεν είναι συμμετρική. Μια επεκτάσιμη διασύνδεση τοποθετείται μεταξύ των κόμβων επεξεργασίας με τη διαφορά ότι κάθε κόμβος έχει το δικό του τοπικό μερίδιο

της κύριας συνολικής μνήμης στο οποίο έχει και ταχύτερη πρόσβαση. Με την αξιοποίηση της τοπικής μνήμης της κατανομής των δεδομένων, προσδοκούμε ότι οι περισσότερες προσβάσεις θα είναι ικανοποιημένες με την τοπική μνήμη και δεν θα χρειαστούν να διασχίσουν το δίκτυο.



Εικόνα 3-6: Distributed-Memory

Σε όλες τις περιπτώσεις, οι κρυφές μνήμες διαδραματίζουν ουσιαστικό ρόλο α) στη μείωση του μέσου χρόνου προσπέλασης μνήμης, όπως φαίνεται από τον επεξεργαστή και β) στη μείωση της απαίτησης του εύρους ζώνης που κάθε επεξεργαστής διαθέτει στην κοινόχρηστη διασύνδεση και του συστήματος μνήμης. Η απαίτηση του εύρους ζώνης μειώνεται επειδή οι προσβάσεις στα δεδομένα που εκδίδονται από έναν επεξεργαστή πληρούνται στη μνήμη cache και δεν χρειάζεται να εμφανιστούν στο δίαυλο επικοινωνίας, διαφορετικά, κάθε πρόσβαση από κάθε επεξεργαστή θα φαίνεται στο δίαυλο επικοινωνίας που γρήγορα θα κορεσθεί.

Σε όλα τα μοντέλα αλλά κυρίως στη προσέγγιση της κοινόχρηστης cache, ο κάθε επεξεργαστής έχει τουλάχιστον ένα επίπεδο ιεραρχίας cache που είναι ιδιωτική και αυτό δημιουργεί μια κρίσιμη πρόκληση, της συνάφειας της μνήμης (cache coherence). Το πρόβλημα δημιουργείται όταν ένα μπλοκ μνήμης είναι στο παρόν στις cache ενός ή περισσοτέρων επεξεργαστών και ένας άλλος επεξεργαστής τροποποιεί το ίδιο μπλοκ μνήμης. Αν δεν ληφθούν μέτρα, οι πρώτοι επεξεργαστές θα εξακολουθήσουν να έχουν πρόσβαση στο παλιό αντίγραφο του μπλοκ που βρίσκεται στις cache τους.

4.2.3 Cache Coherence (Συνάφεια Κρυφής Μνήμης)

Σκεφτείτε για μια στιγμή διαισθητικά το μοντέλο για το τι μια μνήμη πρέπει να κάνει. Πρέπει να παρέχει το σύνολο των θέσεων που κρατά τις τιμές, και όταν μια θέση διαβαστεί θα πρέπει να επιστρέψει τις τελευταίες γραπτές τιμές σε αυτήν τη θέση. Αυτή είναι η θεμελιώδης ιδιότητα του αφηρημένου μοντέλου μνήμης που στηριζόμαστε στα διαδοχικά προγράμματα, όταν χρησιμοποιούμε τη μνήμη για να μεταδώσουμε μια τιμή από ένα σημείο σε ένα πρόγραμμα όπου υπολογίζεται σε άλλα σημεία όπου χρησιμοποιείται. Βασιζόμαστε δηλαδή στην ίδια ιδιότητα της μνήμης του συστήματος όταν χρησιμοποιούμε έναν κοινόχρηστο χώρο διευθύνσεων για να κοινοποιηθούν τα στοιχεία μεταξύ νημάτων ή διαδικασιών που εκτελούνται σε έναν επεξεργαστή. Μια ανάγνωση επιστρέφει τις τελευταίες τιμές που γράφεται στη θέση, ανεξάρτητα από το ποιός επεξεργαστής το έγραψε. Η προσωρινή αποθήκευση δεν ανακατεύεται με τη χρήση των πολλαπλών διεργασιών σε έναν επεξεργαστή, επειδή όλοι βλέπουν τη μνήμη μέσα από την ίδια ιεραρχία της κρυφής μνήμης.

Θα θέλαμε επίσης να βασιστούμε στην ίδια ιδιότητα, όταν δύο διεργασίες εκτελούνται σε διαφορετικούς επεξεργαστές που μοιράζονται μια μνήμη. Δηλαδή, θα θέλαμε τα αποτελέσματα

ενός προγράμματος που χρησιμοποιεί πολλαπλές διεργασίες να μην διαφέρουν όταν οι διαδικασίες τρέχουν σε διαφορετικούς φυσικούς επεξεργαστές από ό, τι όταν τρέχουν (ανακατεμένα ή πολυπρογραμματισμένα) για το ίδιο φυσικό επεξεργαστή. Ωστόσο, όταν δύο διεργασίες δουν τη κοινόχρηστη μνήμη μέσω διαφορετικών κρυφών μνημών, υπάρχει κίνδυνος ότι ένας μπορεί να βλέπει τη νέα τιμή στη κρυφή του μνήμη, ενώ ο άλλος να εξακολουθεί να βλέπει τη παλιά.

Οι πολιτικές ανάγνωσης κυριαρχούν στις cache του επεξεργαστή, καθώς οι περισσότερες εντολές διαβάζονται μόνο και τα δεδομένα συχνά δεν γράφονται στη μνήμη. Οι πολιτικές ανάγνωσης είναι:

- **Read Through -** ανάγνωση μιας λέξης από την κύρια μνήμη στην CPU
- **No Read Through -** ανάγνωση ενός μπλοκ από την κύρια μνήμη στη μνήμη προσωρινής αποθήκευσης και στη συνέχεια από την προσωρινή μνήμη στην CPU.

Η αμεσότητα πρόσβασης δεν συμβαίνει στην περίπτωση της εγγραφής. Η τροποποίηση ενός μπλοκ δεν μπορεί να αρχίσει μέχρι η ταμπέλα επισήμανσης να ελεγχθεί για να διαπιστωθεί αν η διεύθυνση είναι αυτή που ζητείται (write hit). Επίσης, ο επεξεργαστής καθορίζει το μέγεθος της εγγραφής, συνήθως μεταξύ 1 και 8 bytes μόνο για το τμήμα του μπλοκ που μπορεί να αλλάξει. Αντίθετα, οι αναγνώσεις έχουν πρόσβαση σε περισσότερα bytes χωρίς πρόβλημα. Οι πολιτικές εγγραφής σε ευστοχία της εγγραφής συχνά διαχωρίζουν τα πρότυπα των cache:

- **Write Through -** Οι πληροφορίες γράφονται τόσο στο μπλοκ της κρυφής μνήμης όσο και στο μπλοκ της μνήμης του χαμηλότερου επιπέδου.

  Πλεονεκτήματα:

  - Αστοχία στην ανάγνωση ποτέ δεν καταλήγει σε εγγραφές στη κύρια μνήμη.
  - Εύκολη υλοποίηση.
  - Η κύρια μνήμη πάντα έχει το πιο πρόσφατο κομμάτι των δεδομένων (συνέπεια).

  Μειονεκτήματα:

  - Η εγγραφή είναι πιο αργή.
  - Κάθε εγγραφή χρειάζεται πρόσβαση στη κύρια μνήμη.
  - Σαν αποτέλεσμα χρησιμοποιεί περισσότερο εύρος ζώνης.

- **Write Back -** οι πληροφορίες γράφονται μόνο στο μπλοκ της κρυφής μνήμης. Το τροποποιημένο block cache γράφεται στην κύρια μνήμη μόνο όταν αντικαθίσταται. Για να μειωθεί η συχνότητα της γραφής πίσω στο μπλοκ όσον αναφορά την αντικατάσταση, ένα “βρώμικο bit” (dirty bit) συνήθως χρησιμοποιείται. Αυτή η κατάσταση του bit υποδεικνύει αν το block είναι “βρώμικο” (είναι δηλαδή τροποποιημένο όσο βρίσκεται στη cache) ή “καθαρό” (δηλαδή μη τροποποιημένο). Αν είναι “καθαρό” το μπλοκ δεν εγγράφεται σε μια αστοχία.

  Πλεονεκτήματα:

  - Οι εγγραφές γίνονται με την ταχύτητα της μνήμης cache.

- Πολλαπλές εγγραφές μέσα σε ένα μπλοκ απαιτούν μόνο μία εγγραφή στην κύρια μνήμη.
- Ως αποτέλεσμα χρησιμοποιεί λιγότερο εύρος ζώνης της μνήμης.

Μειονεκτήματα:

- Δύσκολη υλοποίηση.
- Η κύρια μνήμη δεν είναι πάντα σύμφωνη με cache.
- Οι αναγνώσεις ως αποτέλεσμα της αντικατάστασης μπορεί να προκαλέσουν εγγραφές, "βρώμικων" μπλοκ στην κύρια μνήμη.

Υπάρχουν δύο κοινές επιλογές εγγραφών αστοχίας (write miss):

- **Write Allocate –** Το μπλοκ φορτώνεται από τη μνήμη στη cache σε μια εγγραφή αστοχίας, ακολουθούμενη από μια ενέργεια ευστοχίας-εγγραφής.
- **No Write Allocate -** Το μπλοκ τροποποιείται στην κύρια μνήμη και δεν φορτώνεται στην μνήμη cache.

Ευστοχία/Αστοχία Εγγραφής (Read hit / miss)

- **Read hit**: ανάγνωση των δεδομένων από την cache.
- **Read miss**: μεταφορά ολόκληρου του block που περιέχει τα δεδομένα που αναζητάμε στην cache και στη συνέχεια όπως στο read hit.

Παρά το γεγονός ότι, η πολιτική της ευστοχίας εγγραφής μπορεί να χρησιμοποιηθεί με write through ή write-back cache, οι write-back caches γενικά χρησιμοποιούν write allocate (ελπίζοντας ότι διαδοχικές εγγραφές στο μπλοκ θα "πιαστούν" από τη μνήμη cache) και οι write-through caches χρησιμοποιούν συχνά no write allocate (αφού διαδοχικές εγγραφές σε αυτό το block θα πρέπει εν τούτοις να γραφούν στη μνήμη).

Write Through with No Write Allocate:

- Στις ευστοχίες (**write hit**) γράφει στη μνήμη cache και στη κύρια μνήμη.
- Στις αστοχίες (**write miss**) ενημερώνει το μπλόκ στη κύρια μνήμη, χωρίς να φέρνει αυτό το μπλόκ στη cache.
- Μεταγενέστερες εγγραφές στο μπλόκ θα ενημερώσουν τη κύρια μνήμη επειδή η πολιτική της Write Through είναι απασχολημένη. Έτσι, για ορισμένο χρόνο φυλάσσεται χωρίς να φέρνει το μπλόκ στη cache σε μια αστοχία επειδή θα εμφανίζεται άχρηστη η εγγραφή ούτως ή άλλως.

Write Back with Write Allocate:

- Στις επιτυχίες/ευστοχίες (**write hit**) ενημερώνει τη cache κάνοντας το bit του μπλόκ σε "βρώμικο" , η κύρια μνήμη δεν ενημερώνεται.
- Στις αστοχίες (**write miss**) ενημερώνει το μπλόκ στη κύρια μνήμη και φέρνει το μπλόκ στη cache.

- Μεταγενέστερες εγγραφές στο ίδιο μπλόκ, αν το μπλόκ αρχικά προκάλεσε μια αστοχία, θα ευστοχήσει την επόμενη φορά, θέτοντας ένα "βρώμικο" bit για το μπλόκ. Αυτό θα εξαλείψει περαιτέρω προσβάσεις στη μνήμη και θα καταλήξει σε μια πολύ αποτελεσματική εκτέλεση σε σύγκριση με τον συνδυασμό Write Through με Write Allocate.

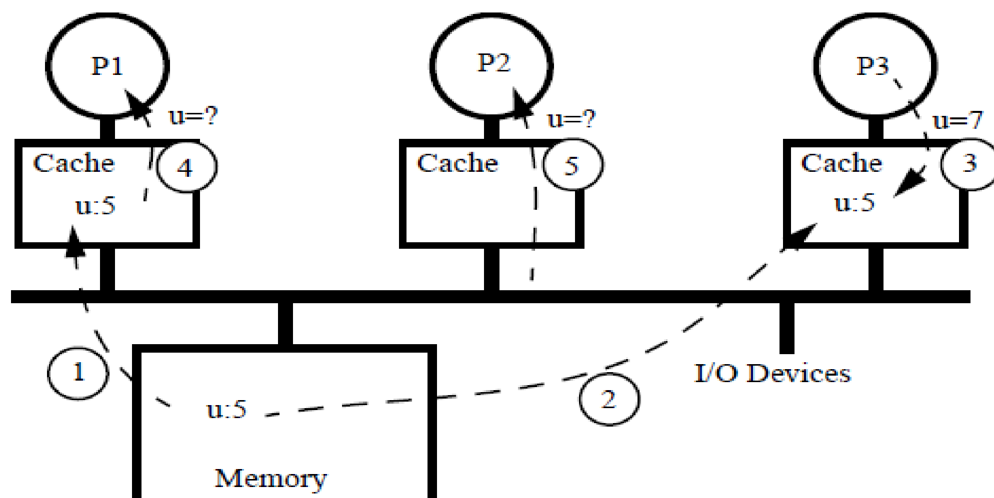
Το Πρόβλημα της Συνάφειας της Κρυφής Μνήμης (Cache Coherence)

Το πρόβλημα της κοινόχρηστης συνάφειας της μνήμης σε πολυεπεξεργαστικά συστήματα είναι τόσο αποδοτικά κρίσιμο όσο και διαδεδομένο. Το πρόβλημα απεικονίζεται καλύτερα με το ακόλουθο παράδειγμα.

Το Σχήμα 3-5 απεικονίζει τρεις επεξεργαστές με κρυφές μνήμες συνδεδεμένες μέσω **δίαυλου** επικοινωνίας (bus) στη κοινόχρηστη βασική μνήμη. Μια ακολουθία προσβάσεων στην τοποθεσία u γίνεται από τους επεξεργαστές. Πρώτον, ο επεξεργαστής P1 διαβάζει τη u από την κύρια μνήμη, φέρνοντας ένα αντίγραφο στην προσωρινή μνήμη του (cache). Στη συνέχεια ο P3 επεξεργαστής διαβάζει τη u από την κύρια μνήμη, φέρνοντας ένα αντίγραφο στην προσωρινή γρήγορη μνήμη του (cache). Στη συνέχεια ο P3 επεξεργαστής γράφει στη μεταβλητή u αλλάζοντας την αξία του από 5 σε 7. Σε μια εγγραφή-μέσω cache, αυτό θα προκαλέσει την κύρια θέση μνήμης να ενημερωθεί, ωστόσο, όταν ο επεξεργαστής P1 διαβάσει πάλι την τοποθεσία u (κίνηση 4), θα διαβάσει, δυστυχώς, τη παλιά τιμή 5 από την cache του αντί της σωστής τιμής που είναι η 7 από την κύρια μνήμη. Τι θα συμβεί λοιπόν αν οι cache επιστρέψουν την εγγραφή, αντί να γράψουν κατευθείαν στη μνήμη;

Απάντηση

Η κατάσταση είναι ακόμη χειρότερη με τις cache που επιστρέφουν την εγγραφή. Η εγγραφή του P3 θα ρυθμίσει μερικά το "βρώμικο" (ή τροποποιημένο) bit που συνδέεται με το μπλοκ της cache που κρατάει τη μεταβλητή u χωρίς να ενημερώσει την κύρια μνήμη αμέσως. Μόνο όταν αυτό το μπλοκ της μνήμης cache αντικατασταθεί από το περιεχόμενο της μνήμης cache του P3 θα γραφτεί, στη συνέχεια πίσω στην κύρια μνήμη. Όχι μόνο ο P1 θα διαβάσει τη παλιά τιμή, αλλά όταν ο P2 επεξεργαστής διαβάσει τη u τοποθεσία (κίνηση 5) θα αστοχήσει στη μνήμη cache του και θα διαβάσει τη παλιά τιμή 5 από την κύρια μνήμη, αντί για το 7. Τέλος, αν πολλαπλοί επεξεργαστές γράψουν διαφορετικές τιμές στη θέση u στις caches που επιστρέφουν την εγγραφή, η τελική τιμή που θα φτάσει στη κύρια μνήμη θα καθορίζεται από τη σειρά με την οποία τα μπλοκ μνήμης cache που περιέχουν τη u αντικαθίστανται και δεν θα έχουν σε τίποτα να κάνουν με τη σειρά με την οποία οι εγγραφές στη u συμβαίνουν.



Εικόνα 3-5: Cache Coherence Problem

Το σχήμα δείχνει τρεις επεξεργαστές με cache που συνδέεται με ένα δίαυλο στην κύρια μνήμη. Η u είναι μια θέση στη μνήμη της οποίας το περιεχόμενο διαβάζεται και γράφεται από τους επεξεργαστές. Η σειρά με την οποία οι αναγνώσεις και οι εγγραφές γίνονται υποδεικνύονται από τον αριθμό που αναγράφεται στο εσωτερικό του κύκλου δίπλα στο τόξο. Είναι εύκολο να δούμε ότι εάν ειδική δράση δεν ληφθεί όταν ο P3 ενημερώσει την τιμή του u σε 7, ο P1 στη συνέχεια θα συνεχίσει να διαβάζει τη παλιά τιμή από την προσωρινή μνήμη του (cache) και επίσης ο P2 θα διαβάζει την παλιά τιμή από την κύρια μνήμη.

Παρόμοια προβλήματα αντιμετωπίζουμε ακόμη και σε μονο-επεξεργαστικά συστήματα, όπου πρόσβαση στη μνήμη έχουν λειτουργίες εισόδου / εξόδου (I/O operations) ή συσκευές άμεσης προσπέλασης της μνήμης (DMA devices).

<u>Προβλήματα με τη Συμπεριφορά της Μνήμης</u>

Η τιμή που επιστρέφεται από μια λειτουργία ανάγνωσης πρέπει να είναι η “*τελευταία*” που γράφτηκε. Αλλά η “*τελευταία*” δεν έχει οριστεί επαρκώς. Ακόμη και στην περίπτωση σειριακού προγράμματος, η “τελευταία” ορίζεται σύμφωνα με τη **σειρά που επιβάλλεται από το πρόγραμμα, και όχι από το χρόνο εκτέλεσης.** Στην περίπτωση παράλληλου προγράμματος, η σειρά του προγράμματος ορίζεται εντός της διεργασίας, αλλά πρέπει να οριστεί και μια σειρά που να αφορά όλες τις διεργασίες.

<u>Εξειδίκευση της Συμπεριφοράς της Μνήμης</u>

Αν υπάρχει μία μόνο μοιραζόμενη μνήμη και καθόλου μονάδες κρυφής μνήμης, κάθε λειτουργία ανάγνωσης και εγγραφής προσπελαύνει την ίδια φυσική θέση μνήμης που επιβάλλει μια **καθολική σειρά** στις λειτουργίες αυτές. Άρα

- οι λειτουργίες στη θέση αυτή από έναν επεξεργαστή γίνονται με τη σειρά που επιβάλλει το πρόγραμμα και
- η σειρά των λειτουργιών από διαφορετικούς επεξεργαστές είναι μια παρεμβολή (interleaving) που διατηρεί τις σειρές που επιβάλλονται από τις ανεξάρτητες διαδικασίες του παράλληλου προγράμματος.

“*Τελευταία*” τιμή τότε σημαίνει η πιο πρόσφατη σε μια υποθετική ακολουθιακή σειρά που ικανοποιεί τις παραπάνω ιδιότητες, ενώ

- οι επεξεργαστές βλέπουν τις εγγραφές στην ίδια θέση μνήμης με την ίδια σειρά,

- η καθολική σειρά δεν κατασκευάζεται ποτέ σε πραγματικά συστήματα, αλλά οι διαδικασίες του παράλληλου προγράμματος συμπεριφέρονται σαν να υπάρχει αυτή η καθολική σειρά.

Ορισμός της Συνάφειας Μνήμης

Ένα σύστημα μνήμης είναι ***συναφές*** αν τα αποτελέσματα κάθε εκτέλεσης ενός προγράμματος είναι έτσι ώστε για κάθε θέση μνήμης να μπορούμε να κατασκευάσουμε μια υποθετική ακολουθιακή σειρά όλων των λειτουργιών που αναφέρονται σ' αυτή, που είναι συνεπής με τα αποτελέσματα της εκτέλεσης και στην οποία:

1. Οι λειτουργίες που καλούνται από κάθε διεργασία συμβαίνουν με τη σειρά στην οποία κλήθηκαν από τη διεργασία αυτή**.**
2. Η τιμή που επιστρέφεται από μια λειτουργία ανάγνωσης είναι η τιμή της τελευταίας εγγραφής στη συγκεκριμένη θέση σύμφωνα με την καθολική σειρά.

Προσδιορισμός Cache Coherence

Η συνάφεια καθορίζει τη συμπεριφορά ενός επεξεργαστή που διαβάζει και γράφει στην ίδια θέση μνήμης. Η συνάφεια των κρυφών μνημών επιτυγχάνεται εφόσον πληρούνται οι ακόλουθες προϋποθέσεις:

- Σε μια ανάγνωση που πραγματοποιείται από ένα επεξεργαστή P σε μια θέση x και ακολουθείται από μια εγγραφή από τον ίδιο επεξεργαστή P στη x, δίχως εγγραφές στη x από άλλον επεξεργαστή να λαμβάνουν χώρα μεταξύ των εντολών της εγγραφής και της ανάγνωσης του P, η x πρέπει να επιστρέφει πάντα την τελευταία εγγραμμένη τιμή του P. Η προϋπόθεση αυτή συνδέεται με τη διατήρηση της σειράς του προγράμματος και αυτό πρέπει να επιτυγχάνεται ακόμα και σε αρχιτεκτονικές με μονούς επεξεργαστές.
- Διάδοση εγγραφών, η τιμή που γράφεται πρέπει να γίνεται αμέσως ορατή στους άλλους. Μια ανάγνωση που πραγματοποιείται από έναν επεξεργαστή P1 στη θέση x και ακολουθεί μια εγγραφή από έναν άλλο επεξεργαστή P2 στη x πρέπει να επιστρέψει την εγεγραμμένη τιμή από τον P2 εάν δεν υπάρχουν άλλες εγγραφές στη x να πραγματοποιηθούν μεταξύ των δύο προσβάσεων από άλλο επεξεργαστή. Η προϋπόθεση αυτή ορίζει την έννοια της συνάφειας της μνήμης. Εάν οι επεξεργαστές μπορούν να διαβάσουν την ίδια παλιά τιμή, μετά την εγγραφή που έχει πραγματοποιηθεί από τον P2, μπορούμε να πούμε ότι η μνήμη είναι ασαφής.
- Σειριοποίηση εγγραφών, οι εγγραφές σε μια θέση φαίνονται στην ίδια σειρά για όλους. Εγγραφές στην ίδια θέση πρέπει να είναι διαδοχικές. Με άλλα λόγια, εάν η τοποθεσία x λάβει δύο διαφορετικές τιμές A και B, σε αυτή τη σειρά, από οποιουσδήποτε δύο επεξεργαστές, οι επεξεργαστές δεν θα μπορέσουν ποτέ να διαβάσουν τη τοποθεσία x ως B και στη συνέχεια να τη διαβάσουν ως A. Η θέση του x θα πρέπει να αναζητηθεί με τις τιμές A και B σε αυτή τη σειρά. Δεν χρειάζεται όμοια σειριοποίηση των λειτουργιών ανάγνωσης, αφού οι λειτουργίες ανάγνωσης δεν είναι ορατές από τους υπόλοιπους.

Οι προϋποθέσεις αυτές καθορίζονται υποθέτοντας ότι οι λειτουργίες ανάγνωσης και εγγραφής γίνονται στιγμιαία. Ωστόσο, αυτό δεν συμβαίνει στον υπολογιστή δεδομένου του υλικού της λανθάνουσας μνήμης και άλλων πτυχών της αρχιτεκτονικής. Μια εγγραφή από τον επεξεργαστή P1 μπορεί να μην διαβαστεί από μια ανάγνωση του επεξεργαστή P2 αν η ανάγνωση γίνει μέσα σε πολύ μικρό χρονικό διάστημα αφού η ανάγνωση έχει πραγματοποιηθεί. Το μοντέλο συνέπειας μνήμης καθορίζει πότε μια γραπτή τιμή, πρέπει να εξεταστεί από μια εντολή ανάγνωσης που έκανε ένας άλλος επεξεργαστής.

4.2.4 Συνέπεια Μνήμης (Memory Consistency)

Για σωστά και αποδοτικά προγράμματα κοινόχρηστης μνήμης, οι προγραμματιστές χρειάζονται να γνωρίζουν πώς η μνήμη συμπεριφέρεται όσον αφορά τις διαδικασίες διαβάσματος και γραψίματος από πολλαπλούς επεξεργαστές. Παραδείγματος χάριν, εξετάστε το πρόγραμμα στο σχήμα 4-1. Η εικόνα παρουσιάζει τον επεξεργαστή P_1 να αναθέτει επανειλημμένα μια εργασία, εισάγοντας την κατάλληλα σε μια σειρά αναμονής καθηκόντων. Όσο υπάρχουν νέες εργασίες, ο επεξεργαστής P_1 ενημερώνει έναν δείκτη (Head), για να εισάγει την εργασία στην σειρά αναμονής καθηκόντων. Εν τω μεταξύ, οι άλλοι επεξεργαστές περιμένουν η κεφαλή/Head να έχει μια μη-μηδενική τιμή, αποδεσμεύουν την εργασία που δείχνεται από το δείκτη Head μέσα στο κρίσιμο τμήμα (critical section) και τελικά την εκτελούν. Ωστόσο, τι αναμένει ο προγραμματιστής από το σύστημα μνήμης για να εξασφαλίσει τη σωστή εκτέλεση αυτού του τμήματος ενός προγράμματος; Μια σημαντική απαίτηση είναι ότι η τιμή που διαβάστηκε από τους επεξεργαστές P_2-P_n να είναι η ίδια με αυτό που γράφτηκε από τον P_1 σε εκείνο το αρχείο. Παρόλα αυτά, σε πολλά εμπορικά συστήματα κοινόχρηστης μνήμης, είναι δυνατό για τους επεξεργαστές να παρατηρήσουν την παλαιά τιμή του τομέα στοιχείων (δηλαδή τη τιμή πριν ο P_1 να γράψει στη περιοχή εγγραφής), οδηγώντας σε συμπεριφορά τελείως διαφορετική από τις προσδοκίες του προγραμματιστή.

```
Initially all pointers = null, all integers = 0.

P1                                    P2, P3, ..., Pn

while (there are more tasks) {        while (MyTask == null) {
  Task = GetFromFreeList();             Begin Critical Section
  Task → Data = ...;                    if (Head != null) {
  insert Task in task queue               MyTask = Head;
}                                         Head = Head → Next;
Head = head of task queue;              }
                                        End Critical Section
                                      }
                                      ...  = MyTask → Data;
```

Εικόνα 4-1: Requirements of event synchronization through flags

Είναι σαφές ότι περιμένουμε περισσότερα από ένα σύστημα μνήμης από την "επιστροφή την τελευταία τιμής που γράφεται" για κάθε θέση. Για παράδειγμα, σε σχέση με τη σειρά μεταξύ των προσβάσεων προς την κάθε θέση (Task->Data), μερικές φορές προσδοκούμε το σύστημα μνήμης να τηρήσει τη σειρά που διαβάζει και γράφει μια δεδομένη διεργασία σε διαφορετικές τοποθεσίες (Task->Data και Head). Η συνάφεια δεν λέει τίποτα για τη σειρά που οι εγγραφές εκδίδονται από τον P_1 και γίνονται ορατές στο P_2, δεδομένου ότι αυτές οι λειτουργίες βρίσκονται σε διαφορετικές τοποθεσίες και μεσολαβούν οι διεπαφές επεξεργαστή/δικτύου (processor/network interfaces) και το δίκτυο. Ομοίως, η συνάφεια δεν λέει τίποτα για την σειρά με την οποία οι αναγνώσεις που έχουν εκδοθεί σε διαφορετικές θέσεις από τον P_2 εκτελούνται σε σχέση με τον P_1.

Σε άλλες περιπτώσεις, η πρόθεση του προγραμματιστή μπορεί να μην είναι τόσο σαφής, π.χ. στο σχήμα 4-2. Η πρόσβαση γίνεται από τη διεργασία P_1 με απλές εγγραφές, ενώ οι A και B δεν είναι μεταβλητές συγχρονισμού.

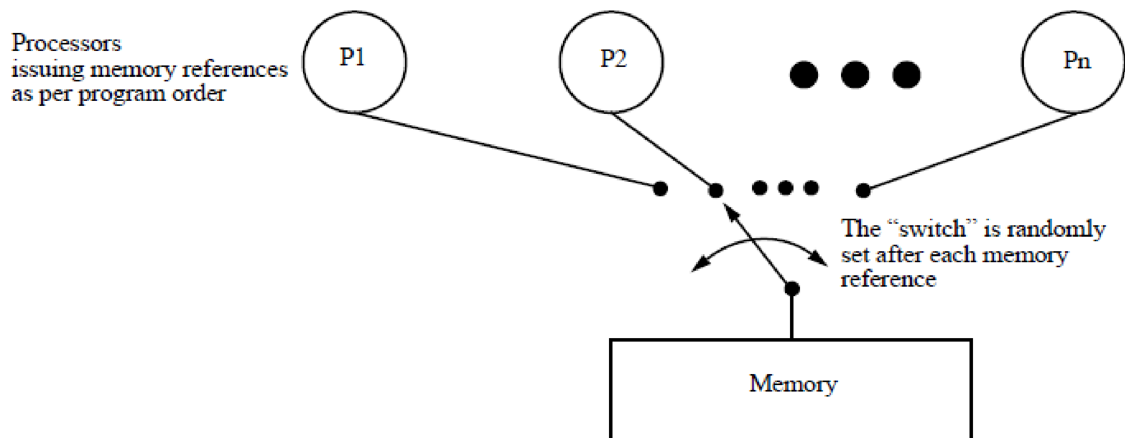
```
P1                      P2
/* Assume initial values of A and B are 0 */
(1a) A = 1;             (2a) print B;
(1b) B = 2;             (2b) print A;
```

Εικόνα 4-2: Orders among accesses without synchronization

Μπορούμε διαισθητικά να αναμένουμε ότι αν η τιμή που τυπώνεται για τη B είναι 2 τότε η τιμή που τυπώνεται για την A είναι 1. Ωστόσο, οι δύο δηλώσεις εκτύπωσης αφορούν διαφορετικές θέσεις, και η συνάφεια δεν καθορίζει πότε η εγγραφή από το P_1 γίνεται ορατή από τον P_2. Είναι σαφές ότι χρειαζόμαστε ένα μοντέλο διάταξης που θα δώσει σ' ένα κοινό χώρο διευθύνσεων μια σαφή σημασιολογία, ώστε οι προγραμματιστές να μπορούν να αιτιολογήσουν τα πιθανά αποτελέσματα και ως εκ τούτου την ορθότητα των προγραμμάτων τους. Το μοντέλο συνέπειας μνήμης για έναν κοινόχρηστο χώρο διευθύνσεων καθορίζει αυτούς τους περιορισμούς σχετικά με τη σειρά με την οποία οι δραστηριότητες μνήμης γίνονται ορατές στους επεξεργαστές. Αυτό αφορά τόσο προσπελάσεις για την ίδια τοποθεσία ή διαφορετικές τοποθεσίες μνήμης, όσο και από την ίδια διαδικασία ή από διαφορετικές διαδικασίες.

Ακολουθιακή Συνέπεια (Sequential Consistency)

Ένα σύστημα πολυεπεξεργαστών είναι **ακολουθιακά συνεπές** εάν το αποτέλεσμα οποιασδήποτε εκτέλεσης είναι το ίδιο όταν οι ακολουθίες όλων των επεξεργαστών εκτελεσθούν σε κάποια ακολουθιακή σειρά, και οι διαδικασίες κάθε μεμονωμένου επεξεργαστή εμφανίζονται σε αυτήν την ακολουθία με τη σειρά που ορίζεται από τα επιμέρους προγράμματα. Το μοντέλο κρύβει τελείως τη θεμελιώδη συνεργασία του υλικού μνήμης του συστήματος από τον προγραμματιστή, για παράδειγμα την ενδεχόμενη παρουσία κατανεμημένων μνημών, caches ή buffers.

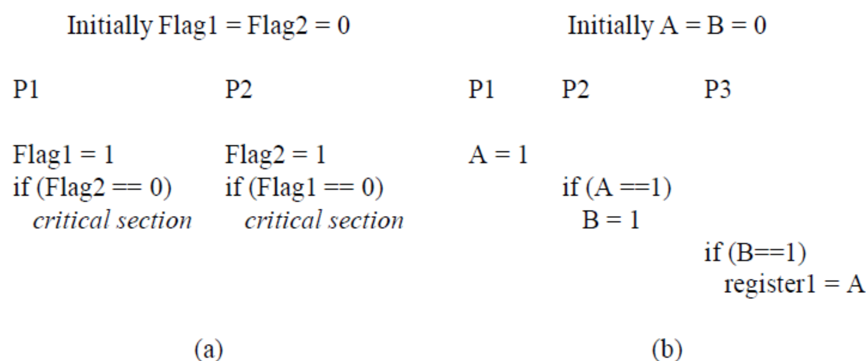


Εικόνα 4-3: Σειρά εκτέλεσης προσπελάσεων μνήμης σε πρόγραμμα χωρίς συγχρονισμό

Υπάρχουν δύο πτυχές στη ακολουθιακή συνέπεια: (1) διατήρηση της σειράς προγράμματος μεταξύ των προσπελάσεων μνήμης που εκτελούνται από τους μεμονωμένους επεξεργαστές, και (2) η διατήρηση μιας ενιαίας διαδοχικής σειράς μεταξύ των προσπελάσεων μνήμης από όλους τους επεξεργαστές. Ο απώτερος σκοπός είναι να εμφανίζεται η κάθε προσπέλαση μνήμης σαν να εκτελείται ατομικά ή στιγμιαία και ανεξάρτητα από τις άλλες προσπελάσεις μνήμης.

Η ακολουθιακή συνέπεια παρέχει μια απλή απεικόνιση του συστήματος στους προγραμματιστές, όπως παρουσιάζεται στην εικόνα 4-3. Εννοιολογικά, υπάρχει μια μοναδική ενιαία μνήμη και ένας διακόπτης ο οποίος συνδέει ένα αυθαίρετο επεξεργαστή με τη μνήμη οποιαδήποτε στιγμή. Κάθε επεξεργαστής διανέμει τις προσπελάσεις μνήμης του με βάση τις διαδικασίες που εκτελεί και ο διακόπτης παρέχει ενιαία σειριακή διάταξη για όλες τις λειτουργίες της μνήμης.

Το σχήμα 4-4 παρέχει δύο παραδείγματα που επεξηγούν τη σημασιολογία της ακολουθιακής συνέπειας.



Εικόνα 4-4: Παράδειγμα ακολουθιακής συνέπειας

Το σχήμα 4-4 (α) επεξηγεί τη σημασία της σειράς του προγράμματος ανάμεσα στις λειτουργίες από ένα ανεξάρτητο επεξεργαστή. Το τμήμα του κώδικα απεικονίζει υλοποίηση του αλγορίθμου Dekker για τα κρίσιμα σημεία, συμπεριλαμβάνοντας δυο επεξεργαστές (P_1 και P_2) με 2 μεταβλητές (Flag1 και Flag2) οι οποίες παίρνουν αρχική τιμή 0. Όταν ο P_1 προσπαθεί να εισέλθει στο κρίσιμο τμήμα, ενημερώνει τη Flag1 σε 1, και ελέγχει την τιμή της Flag2. Η τιμή 0 για τη Flag2 δείχνει ότι ο P_2 δεν έχει προσπαθήσει ακόμα να εισέλθει στο κρίσιμο τμήμα επομένως, είναι ασφαλές για τον P_1 να εισέλθει. Αυτός ο αλγόριθμος στηρίζεται στην υπόθεση ότι η τιμή 0 επιστρέφεται από την ανάγνωση του P_1 υπονοώντας ότι η διαδικασία γραψίματος του P_1 έχει λάβει χώρα πριν από τις λειτουργίες γραψίματος και ανάγνωσης του P_2. Συνεπώς η ανάγνωση της μεταβλητής του P_2 θα επιστρέψει την τιμή 1,απαγορεύοντας έτσι στον P_2 να εισέλθει στο κρίσιμο σημείο. Η ακολουθιακή συνέπεια εξασφαλίζει τα ανωτέρω με την απαίτηση ότι η σειρά του προγράμματος μεταξύ των διαδικασιών μνήμης P_1 και P_2 διατηρείται, αποκλείοντας κατά συνέπεια τη δυνατότητα και των δύο επεξεργαστών να διαβάσουν την τιμή 0 και να εισαχθούν μαζί στο κρίσιμο τμήμα.

Το σχήμα 4-4 (β) επεξηγεί τη σημασία της ατομικής εκτέλεσης των διαδικασιών μνήμης. Η εικόνα 4-3 δείχνει επεξεργαστές να μοιράζονται τις μεταβλητές A και B, με αρχική τιμή 0. Υποθέστε ότι ο επεξεργαστής P_1 επιστρέφει τη τιμή 1 (που έχει γραφτεί από τον P_1) για την ανάγνωση της A, ο P_2 γράφει στη μεταβλητή B και ο P_3 επιστρέφει την τιμή 1 (που έχει γραφτεί από τον P_2) για την B. Έτσι η ακολουθιακή συνέπεια μας επιτρέπει να συμπεράνουμε ότι το αποτέλεσμα της γραφής του P_1 φαίνεται από όλο το σύστημα την ίδια χρονική στιγμή.

Implementing Sequential Consistency

Η ενότητα αυτή περιγράφει πως μπορεί να πρακτικά υλοποιηθεί η ακολουθιακή συνέπεια που φαίνεται στην εικόνα 4-3. Σε αντίθεση με τους μονό-επεξεργαστές, η διατήρηση της σειρά των πράξεων με βάση τη θέση προσπέλασης μνήμης δεν αρκεί για τη διατήρηση της

ακολουθιακής συνέπειας στους πολυεπεξεργαστές εφόσον η ακολουθιακή συνέπειας αλληλεπιδρά με απλές βελτιστοποιήσεις υλικού, όπως αρχιτεκτονικές χωρίς μνήμες, bypass buffer και άλλες.

<u>Συνάφεια Μνήμης Και Ακολουθιακή Συνέπεια</u>

Πολλοί προσδιορισμοί για την συνάφεια μνήμης (συχνά αναφερόμενη ως συνεκτικότητα μνήμης) υπάρχουν στη βιβλιογραφία. Οι ισχυρότεροι προσδιορισμοί μεταχειρίζονται τον όρο ως συνώνυμο με την ακολουθιακή συνέπεια.

Η αλληλεπίδραση της ακολουθιακής συνέπειας από άποψη διάταξης προγράμματος με το μεταγλωττιστή είναι ανάλογη με αυτήν του υλικού. Συγκεκριμένα, για όλα τα αποσπάσματα προσραμμάτων που έχουν συζητηθεί μέχρι τώρα, αναδιατάξεις παραγμένες από το μεταγλωττιστή σε λειτουργίες της κοινόχρηστης μνήμης θα οδηγήσουν σε παραβιάσεις της ακολουθιακής συνέπειας παρόμοιες με τις αναδιατάξεις παραγμένες από το υλικό. Ως εκ τούτου, ελλείψει πιο σύνθετης ανάλυσης, βασική προϋπόθεση για το μεταγλωττιστή είναι να διατηρήσει τη σειρά στο πρόγραμμα μεταξύ των λειτουργιών κοινόχρηστης μνήμης. Η απαίτηση αυτή περιορίζει άμεσα κάθε βελτιστοποίηση που μπορεί να οδηγήσει σε πράξεις αναδιάταξης της μνήμης.

5. Βιβλιοθήκες Νημάτων (API Threads)

Τα **POSIX** (**P**ortable **O**perating **S**ystem **I**nterface for Uni**x**) είναι το όνομα μιας σειράς πρότυπων που καθορίζουν τις βασικές υπηρεσίες που παρέχει ένα λειτουργικό σύστημα, και κυρίως το API των διαθέσιμων κλήσεων συστήματος. Δημιουργήθηκε με στόχο την εξασφάλιση της συμβατότητας μεταξύ των ποικίλλων λειτουργικών συστημάτων UNIX, έτσι ώστε τα προγράμματα που τρέχουν σε έναν υπολογιστή με UNIX να μπορούν να μεταγλωττιστούν σε οποιονδήποτε άλλον υπολογιστή με UNIX χωρίς να χρειάζεται να τροποποιηθεί ο πηγαίος κώδικας (ή μέρος αυτού). Η συμμόρφωση με το POSIX αποτελεί προϋπόθεση για να μπορεί ένα λειτουργικό σύστημα να ονομάζεται UNIX. Τα POSIX είναι μια οικογένεια προτύπων που αναπτύχθηκαν από το IEEE (Institute of Electrical and Electronics Engineers). Αρχικά, το όνομα "POSIX" αναφερόταν ως IEEE Std 1003.1-1988, κυκλοφορώντας, όπως υποδηλώνει το όνομα, το 1988. Η οικογένεια των POSIX πρότυπων έχουν ορισθεί επισήμως ως **IEEE 1003** και το διεθνές πρότυπο όνομα είναι το ISO / IEC 9945. Το POSIX αποτελείται από διάφορες προδιαγραφές, οι κύριες που μας απασχολούν είναι:

A. POSIX.1b

POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993)

- Priority Scheduling
- Real-Time Signals
- Clocks and Timers
- Semaphores
- Message Passing
- Shared Memory
- Synch and Synch I/O
- Memory Locking Interface

B. POSIX.1c

POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)

- Thread Creation, Control, and Cleanup
- Thread Scheduling
- Thread Synchronization
- Signal Handling

5.1 Ορισμός Νήματος (Thread Basics)

Τεχνικά, ένα νήμα ορίζεται ως μια ανεξάρτητη ροή εντολών που μπορεί να προγραμματιστεί να λειτουργεί με τέτοιο τρόπο από το λειτουργικό σύστημα. Για παράδειγμα, ας εξετάσουμε το ακόλουθο τμήμα κώδικα που υπολογίζει το γινόμενο δύο πινάκων A μέγεθος.

```
1   for (row = 0; row < n; row++)
2       for (column = 0; column < n; column++)
3           c [row][column] =
4               dot_product(get_row(a, row),
5                           get_col(b, col));
```

Ο for βρόγχος σε αυτό το κομμάτι κώδικα έχει n^2 επαναλήψεις, καθεμία από τα οποίες μπορούν να εκτελεστούν ανεξάρτητα. Μια τέτοια ανεξάρτητη ακολουθία των εντολών αναφέρεται ως ένα νήμα. Στο παράδειγμα που παρουσιάζεται πιο πάνω, υπάρχουν n^2 νήματα, ένα για κάθε επανάληψη του βρόγχου for. Δεδομένου ότι κάθε ένα από αυτά τα νήματα μπορούν να

εκτελεστούν ανεξάρτητα από τα άλλα, μπορούν να προγραμματιστούν ταυτόχρονα σε πολλαπλούς επεξεργαστές. Μπορούμε να μετατρέψουμε το παραπάνω τμήμα κώδικα ως εξής:

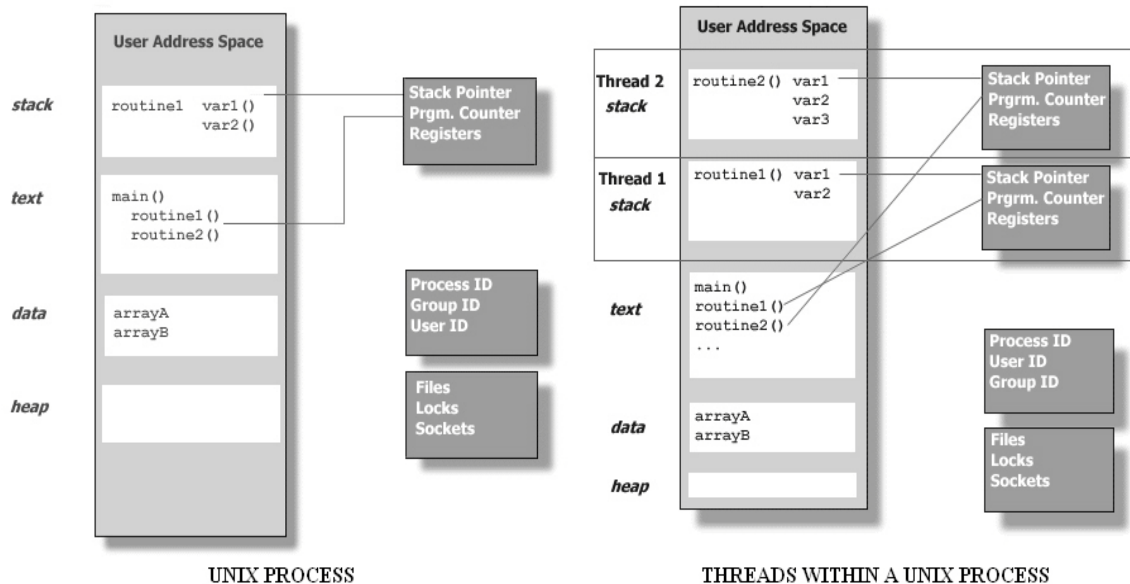
```
1   for (row = 0; row < n; row++)
2       for (column = 0; column < n; column++)
3           c[row][column] =
4            create_thread(dot_product(get_row(a, row),
5                                     get_col(b, col)));
```

Εδώ, χρησιμοποιούμε μια συνάρτηση, create_thread, που προσφέρει ένα μηχανισμό για τον προσδιορισμό μιας συνάρτησης C ως νήμα. Το υποκείμενο σύστημα μπορεί προγραμματίσει αυτά τα νήματα σε πολλαπλούς επεξεργαστές. Φανταστείτε έτσι ένα βασικό εκτελέσιμο πρόγραμμα (a.out) που περιέχει μια σειρά διαδικασιών που είναι σε θέση να προγραμματιστούν για να τρέξουν ταυτόχρονα και ανεξάρτητα από το λειτουργικό σύστημα. Από την πλευρά του προγραμματιστή η έννοια της "διαδικασίας" που τρέχει ανεξάρτητα από το κύριο πρόγραμμα μπορεί να περιγράψει καλύτερα ένα νήμα.

5.1.1 Τα Νήματα σε σχέση με τις Διεργασίες (Thread vs. Process)

Πριν κατανοήσουμε τι είναι το νήμα, πρέπει πρώτα να καταλάβουμε τι είναι μια διεργασία UNIX. Μια διεργασία δημιουργείται από το λειτουργικό σύστημα, και απαιτεί ένα δίκαιο μερίδιο του "φόρτου" του συστήματος. Διεργασίες περιέχουν πληροφορίες σχετικά με τους πόρους του προγράμματος και τη κατάσταση εκτέλεσης των εντολών του προγράμματος. Συμπεριλαμβάνονται:

- Process ID, process group ID, user ID, and group ID (κωδικός ταυτότητας διεργασίας, ομάδας διεργασίας, χρήστη και ομάδας)
- Environment (Μεταβλητές διεργασίας)
- Working directory (Κατάλογος εργασίας)
- Program instructions (Εντολές προγράμματος)
- Registers (Καταχωρητές)
- Stack (Στοίβες)
- Heap (Σωροί)
- File descriptors (Περιγραφείς αρχείων)
- Signal actions (Σήματα)
- Shared libraries (Κοινόχρηστες βιβλιοθήκες)
- Εργαλεία διαδιεργασιακής επικοινωνίας (inter-process communication tools), όπως ουρές αναμονής μηνυμάτων, αλυσιδωτές ροές διεργασιών, σηματοφόροι ή κοινόχρηστη μνήμη (message queues, pipes, semaphores ή shared memory).



Εικόνα 5-4: Unix Process vs. Thread

Τα νήματα υπάρχουν μέσα στους πόρους της διεργασίας και τους χρησιμοποιούν, αλλά είναι σε θέση να προγραμματιστούν από το λειτουργικό σύστημα και να λειτουργήσουν ως ανεξάρτητες οντότητες σε μεγάλο βαθμό, διότι χρησιμοποιούν εις διπλούν μόνο τους απολύτως απαραίτητους πόρους που τους επιτρέπουν να υφίστανται ως εκτελέσιμος κώδικας.

Η ανεξάρτητη ροή ελέγχου επιτυγχάνεται επειδή το κάθε νήμα διατηρεί χωριστά:

- Δείκτη στοίβας (stack pointer)
- Σύνολο καταχωρητών (registers)
- Πρωτόκολλο τακτικής (priority scheduling)
- Εκκρεμή και αναστέλλοντα σήματα (pending/blocked signals)
- Ιδιωτικά δεδομένα

Έτσι, συνοπτικά, στο περιβάλλον UNIX ένα νήμα κάνει τα εξής:

- Υπάρχει μέσα σε μια διεργασία και χρησιμοποιεί τους πόρους της.
- Έχει δική του ανεξάρτητη ροή ελέγχου όσο η διεργασία της μητρικής υπάρχει και το λειτουργικό σύστημα την υποστηρίζει.
- Αντιγράφει μόνο τους απαραίτητους πόρους που χρειάζεται για να είναι ανεξάρτητο.
- Μπορεί να μοιράζεται τους πόρους της διεργασίας με άλλα νήματα που ενεργούν εξ' ίσου ανεξάρτητα (και εξαρτώμενα)
- Τερματίζει αν η διαδικασία γονέας τερματίσει - ή κάτι παρόμοιο
- Είναι σχετικά "ελαφρύ", επειδή ο περισσότερος φόρτος έχει ήδη εκπληρωθεί από δημιουργία της διεργασίας του.

Επειδή:

- Οι αλλαγές που γίνονται από ένα νήμα σε κοινόχρηστους πόρους του συστήματος (όπως το κλείσιμο ενός αρχείου) θα είναι ορατές και από τα άλλα νήματα.
- Δύο δείκτες που έχουν την ίδια τιμή δείχνουν στα ίδια δεδομένα.
- Ανάγνωση και γραφή από τις ίδιες θέσεις μνήμης είναι πιθανή περίπτωση, και ως εκ τούτου απαιτείται ρητός συγχρονισμός από τον προγραμματιστή.

5.2 POSIX Threads

POSIX Threads ή Threads, είναι ένα πρότυπο του POSIX για τα νήματα/threads. Το πρότυπο POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), προσδιορίζει ένα API για τη δημιουργία και το χειρισμό των νημάτων. Το πρότυπο POSIX συνέχισε να εξελίσσεται και να υπόκειται σε αναθεωρήσεις, καθώς και οι προδιαγραφές των προτύπων Pthreads. Η τελευταία γνωστή έκδοση είναι το πρότυπο IEEE Std 1003,1, 2004 Edition. Τα Pthreads ορίζονται ως ένα σύνολο γλώσσας προγραμματισμού τύπου C και κλήσεων διεργασιών, που υλοποιείται με τη κεφαλίδα του pthread.h αρχείου και βιβλιοθήκης του νήματος - αν και αυτή η βιβλιοθήκη μπορεί να είναι μέρος μιας άλλης βιβλιοθήκης, όπως η libc, σε ορισμένες εφαρμογές.

5.2.1 Επισκόπηση των PThread

Τα Pthreads μας προσφέρουν τα παρακάτω χαρακτηριστικά:

Φορητότητα λογισμικού. Οι νηματικές εφαρμογές μπορεί να αναπτυχθούν σε σειριακές μηχανές και να τρέχουν σε παράλληλες μηχανές χωρίς καμία αλλαγή. Αυτή η ικανότητα να μεταναστεύουν προγράμματα μεταξύ διαφορετικών αρχιτεκτονικών πλατφόρμων είναι ένα πολύ σημαντικό πλεονέκτημα των νηματικών API, με επιπτώσεις όχι μόνο στη χρησιμοποίηση του λογισμικού αλλά και στην ανάπτυξη εφαρμογών.

Απόκρυψη Λανθάνουσας Κατάστασης (Latency). Ένα από τους σημαντικότερους "φόρτους" σε προγράμματα (σειριακά και παράλληλα) είναι η καθυστέρηση της πρόσβασης στη μνήμη, σε I / O και στην επικοινωνία. Επιτρέποντας πολλαπλά threads να εκτελεστούν από τον ίδιο επεξεργαστή, τα νηματικά API, μπορούν αυτή τη λανθάνουσα κατάσταση να τη κρύψουν. Στην πραγματικότητα, ενώ ένα νήμα περιμένει για μια λειτουργία επικοινωνίας, άλλα νήματα μποροούν να αξιοποιήσουν την CPU, έτσι ώστε να συγκαλύψουν το "φόρτο".

Προγραμματισμός και Load Balancing. Γράφοντας στο χώρο διευθύνσεων κοινόχρηστων παράλληλων προγραμμάτων, ένας προγραμματιστής πρέπει να εκφράσει παραλληλισμό με τρόπο που να ελαχιστοποιεί το "φόρτο" των απομακρυσμένων αλληλεπιδράσεων και της αδράνειας. Ενώ σε πολλές δομημένες εφαρμογές το καθήκον της κατανομής της ίσης εργασίας στους επεξεργαστές γίνεται εύκολα, σε αδόμητες και δυναμικές εφαρμογές (όπως παιχνίδια) το έργο αυτό είναι πιο δύσκολο. Τα νηματικά APIs επιτρέπουν στον προγραμματιστή να καθορίσει ένα μεγάλο αριθμό ταυτόχρονων καθηκόντων και την υποστήριξη σε επίπεδο συστήματος της δυναμικής χαρτογράφησης των καθηκόντων στους επεξεργαστές, με σκοπό την ελαχιστοποίηση του "φόρτου". Με την παροχή αυτής της υποστήριξης σε επίπεδο συστήματος, τα νηματικά API, απαλλάσσουν τον προγραμματιστή από το βάρος του ρητού προγραμματισμού και της εξισορρόπησης φορτίου.

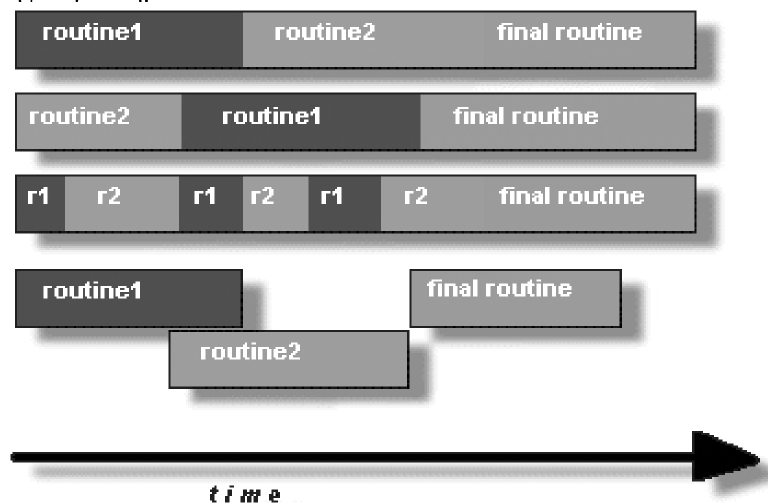
Ευκολία προγραμματισμού, εκτεταμένη χρήση Λόγω των παραπάνω πλεονεκτημάτων, με νηματικά προγράμματα είναι πολύ πιο εύκολο να γράψουμε από τα αντίστοιχα προγράμματα που χρησιμοποιούν το API ανταλλαγής μηνυμάτων. Η επίτευξη ίδιων επίπεδων απόδοσης για τα δύο μοντέλα μπορεί να απαιτεί επιπλέον προσπάθειες, ωστόσο. Με την ευρεία αποδοχή του API POSIX προτύπου, τα εργαλεία ανάπτυξης για τα νήματα POSIX είναι πιο ευρέως διαθέσιμα και σταθερά. Τα νήματα αυτά είναι σημαντικά από άποψη ανάπτυξης του προγραμματισμού και από άποψης μηχανικής λογισμικού.

Σχεδιάζοντας Παράλληλα Προγράμματα με Νήματα

- **Παράλληλος Προγραμματισμός:**

  - Στα σύγχρονα, μηχανήματα multi-cpu (πολυεπεξεργαστών), τα pthreads είναι ιδανικά προσαρμοσμένα για παράλληλο προγραμματισμό. Ότι ισχύει στο παράλληλο προγραμματισμό γενικά εφαρμόζεται στα παράλληλα προγράμματα με pthreads.

- Υπάρχουν πολλά θέματα που λαμβάνονται υπόψη για το σχεδιασμό παράλληλων προγραμμάτων, όπως:
  - ✓ μοντέλο παράλληλου προγραμματισμού
  - ✓ προβλήματα στεγανοποίησης (memory leakage)
  - ✓ εξισορρόπηση φορτίου
  - ✓ εξάρτηση δεδομένων
  - ✓ συγχρονισμός και ζητήματα ταχύτητας και απόδοσης επικοινωνιών
  - ✓ θέματα μνήμης
  - ✓ θέματα I/O
  - ✓ Πολυπλοκότητα προγράμματος
  - ✓ Προσπάθεια προγραμματιστή, κόστος και χρόνος
- Σε γενικές γραμμές όμως, ένα πρόγραμμα για να επωφεληθεί από τα pthreads πρέπει να είναι σε θέση να οργανωθεί σε διακριτά, ανεξάρτητα καθήκοντα τα οποία μπορούν να εκτελέσουν ταυτόχρονα. Για παράδειγμα, αν τα routine1 και routine2 μπορούν να εναλλάσσονται, να διαστρωματώνονται ή / και να επικαλύπτονται σε πραγματικό χρόνο, τότε είναι υποψήφια για νήματα.

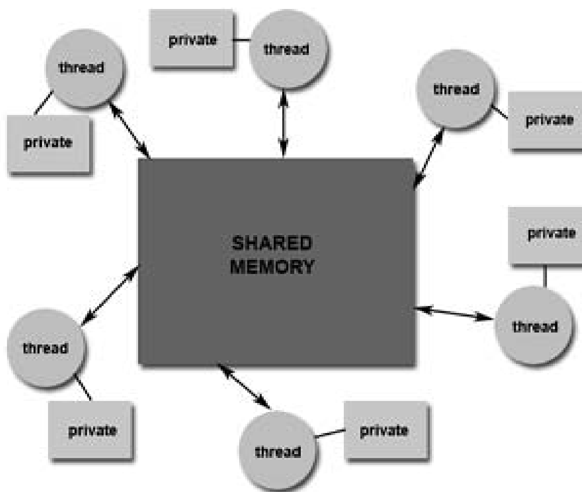


Εικόνα 5-5: Concurrent Pthreads

- Προγράμματα που έχουν τα ακόλουθα χαρακτηριστικά μπορεί να είναι κατάλληλα για pthreads:
  - ✓ Πολλαπλές ταυτόχρονες εργασίες που χρησιμοποιούν πολλούς κύκλους επεξεργαστή
  - ✓ Δυνητικά μεγάλες αναμονές I / O
  - ✓ Ανταπόκριση σε ασύγχρονα γεγονότα
  - ✓ Κάποια εργασία είναι πιο σημαντική από άλλες εργασίες (διακοπές προτεραιότητας - priority inversion)
- Τα Pthreads μπορούν επίσης να χρησιμοποιηθούν και σε σειριακές εφαρμογές, για να μιμηθούν μια παράλληλη εκτέλεση. Ένα τέτοιο παράδειγμα είναι ο web browser ο οποίος για τους περισσότερους ανθρώπους, τρέχει σε μια ενιαία επιφάνεια εργασίας μιας cpu /ή μηχανής, laptop. Πολλά πράγματα μπορούν να “φαίνεται” να συμβαίνουν την ίδια στιγμή.
- Αρκετά κοινά μοντέλα για νηματικά προγράμματα υπάρχουν:
  - ✓ Διαχειριστής/ Εργάτης: ένα ενιαίο νήμα, ο διαχειριστής αναθέτει εργασίες σε άλλα νήματα, που λέγονται εργαζόμενοι. Χαρακτηριστικά, ο διευθυντής χειρίζεται όλες τις εισροές και διαμοιράζει τις εργασίες στα άλλα καθήκοντα. Τουλάχιστον δύο μορφές του

μοντέλου διαχειριστή/εργάτη είναι συχνές: στατικό δείγμα εργάτη και δυναμικό δείγμα εργάτη.
- ✓ Pipeline: ένα έργο έχει χωριστεί σε μια σειρά από υπολειτουργίες, καθεμία από τις οποίες διακινούνται σε σειρά, αλλά ταυτόχρονα, με ένα διαφορετικό νήμα. Για παράδειγμα μια γραμμή συναρμολόγησης αυτοκινήτων περιγράφεται καλά με αυτό το μοντέλο.
- ✓ Peer: παρόμοιο με το μοντέλο διαχειριστή/εργάτη, αλλά μετά αφού το κύριο νήμα δημιουργήσει άλλα νήματα, συμμετέχει στο έργο.

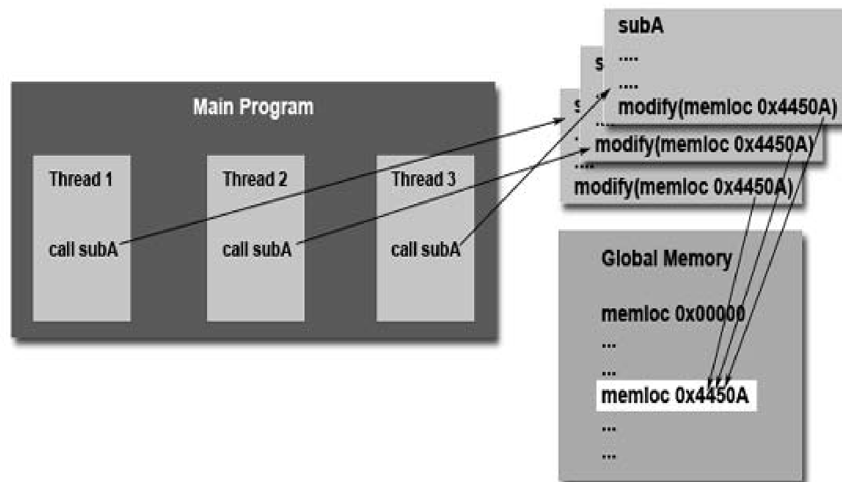
- Κοινή μνήμη:
  - ✓ Όλα τα νήματα έχουν πρόσβαση στην ίδια γενική, κοινόχρηστη μνήμη
  - ✓ Νήματα διαθέτουν επίσης ιδιωτικά τους δεδομένα
  - ✓ Οι προγραμματιστές είναι υπεύθυνοι για το συγχρονισμό πρόσβασης (προστασία) σε κοινά δεδομένα.



Εικόνα 5-6: Κοινή Μνήμη με Threads

Thread-safeness:

- Με Thread-safeness αναφέρεται η ικανότητα μιας εφαρμογής να εκτελέσει πολλαπλά threads ταυτόχρονα χωρίς να αλλάζει αντικανονικά κοινά δεδομένα ή να δημιουργεί συνθήκες ανταγωνισμού.
- Για παράδειγμα, ας υποθέσουμε ότι μια αίτηση δημιουργεί διάφορα νήματα, καθένα από τα οποία πραγματοποιεί μια κλήση στην ίδια ρουτίνα βιβλιοθήκης:
  - ✓ Αυτή η ρουτίνα πιθανά τροποποιεί μια ενιαία δομή ή μια θέση στη μνήμη.
  - ✓ Δεδομένου ότι κάθε νήμα καλεί την ίδια ρουτίνα είναι πιθανό ότι μπορεί να προσπαθήσουν να τροποποιήσουν τη δομή στη θέση μνήμης την ίδια στιγμή.
  - ✓ Αν η ρουτίνα δεν χρησιμοποιεί κάποιες δομές συγχρονισμού για την πρόληψη της φθοράς των δεδομένων, τότε δεν είναι νηματικά-ασφαλής.



Εικόνα 5-7: Thread Safeness

- ✓ Οι συνέπειες για τους χρήστες με εξωτερικές ρουτίνες βιβλιοθήκης είναι ότι εάν δεν είναι 100% βέβαιοι ότι η ρουτίνα είναι νηματικά-ασφαλής, τότε θα πρέπει να επιλύσουν τα προβλήματα που θα μπορούσαν να προκύψουν.
- ✓ Σύσταση: Καλό είναι να είμαστε προσεκτικοί αν η αίτησή μας χρησιμοποιεί βιβλιοθήκες ή άλλα αντικείμενα που δεν εγγυώνται ρητά νηματική ασφάλεια. Σε περίπτωση αμφιβολίας, είναι καλύτερο να υποθέσουμε ότι δεν είμαστε νηματικά-ασφαλείς μέχρις αποδείξεως του εναντίου. Αυτό μπορεί να γίνει με “σει κλπ.

5.2.2 API Pthreads

Διαχείριση Thread: Ρουτίνες που εργάζονται απευθείας για τα θέματα όπως είναι η δημιουργία, απόσπαση, ένωση νημάτων, κ.λπ. Επίσης, περιλαμβάνονται λειτουργίες που θέτουν ή εξετάζουν ιδιότητες στο νήμα (joinable, προγραμματισμός κ.λπ.)

Κυριότερες Βιβλιοθήκες Pthreads

Δημιουργία Thread

```
#include <pthread.h>

int  pthread_create(pthread_t *restrict tidp,
                    const pthread_attr_t *restrict attr,
                    void *(*start_rtn)(void), void *restrict arg);
```

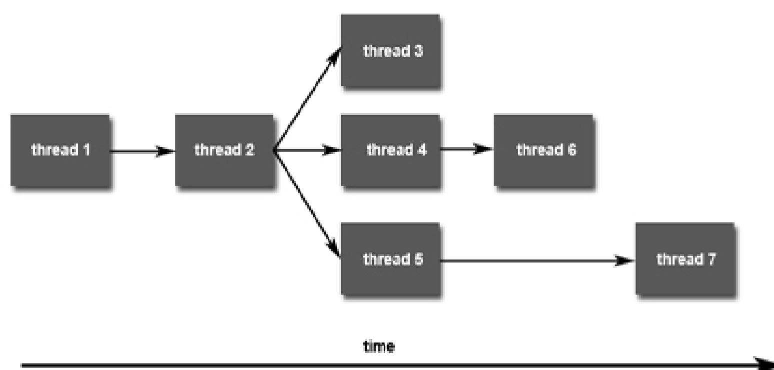
Αρχικά, στη main() το πρόγραμμα περιλαμβάνει ένα και μόνο, νήμα προεπιλογή. Όλα τα άλλα νήματα πρέπει να συσταθούν ρητά από τον προγραμματιστή. Η pthread_create δημιουργεί ένα νέο νήμα και το καθιστά εκτελέσιμο. Αυτή η ρουτίνα μπορεί να κληθεί όσες φορές χρειάζεται από οπουδήποτε μέσα στον κώδικά σας.

Pthread_create ορίσματα:

- Pthread_t: Επιστρέφει το id της διεργασίας.

- Pthread_attr_t: Μπορεί να χρησιμοποιηθεί για να ορίσουμε τα χαρακτηριστικά του νήματος, ή να επιλέξουμε NULL για τις προεπιλεγμένες τιμές.
- void *(*start_rtn): Η ρουτίνα C που θα εκτελέσει το νήμα τη στιγμή που δημιουργείται.
- arg: Ένα μόνο όρισμα που μπορεί να περάσει στη void *(*start_rtn) . Θα πρέπει να περαστεί ως αναφορά σε τύπου void. NULL μπορεί να γίνει εάν κανένα όρισμα δεν χρειάζεται.

Όταν ένα νήμα δημιουργηθεί, δεν υπάρχει καμία εγγύηση ποιό τρέχει πρώτα: το νεοσύστατο νήμα ή το αρχικό νήμα. Το νεοσύστατο νήμα έχει πρόσβαση στο χώρο διευθύνσεων της διεργασίας και κληρονομεί το περιβάλλον του καλούντος νήματος ωστόσο, το σύνολο σημάτων εν αναμονή για το νήμα είναι απενεργοποιημένο.



Εικόνα 5-8: PThreads

```
#include <pthread.h>

int  pthread_attr_init(pthread_attr_t *attr);
```

Η ρουτίνα pthread_attr_init αρχικοποιεί τα χαρακτηριστικά ενός νήματος (attr) με προεπιλεγμένη τιμή που δίνεται από την εκάστοτε δεδομένη εφαρμογή.

.

<u>Τερματισμός Thread</u>

```
#include <pthread.h>

void  pthread_exit(void *rval_ptr);
```

pthread_exit χρησιμοποιείται για έξοδο ρητά από ένα νήμα. Συνήθως, η pthread_exit () ρουτίνα καλείται μετά αφού ένα νήμα έχει ολοκληρώσει το έργο του και δεν απαιτείται πλέον να υπάρχει.

Αν το main() τελειώσει πριν από τα νήματα που έχει δημιουργήσει, και κάνει έξοδο με pthread_exit() , τα άλλα νήματα θα συνεχίσουν να εκτελούνται. Σε αντίθετη περίπτωση, θα τερματιστούν αυτόματα όταν λήξει η main().

Ο προγραμματιστής μπορεί προαιρετικά να ορίσει ένα καθεστώς τερματισμού, το οποίο αποθηκεύεται ως δείκτης void για κάθε νήμα που θέλει να συμμετέχει καλώντας το νήμα.

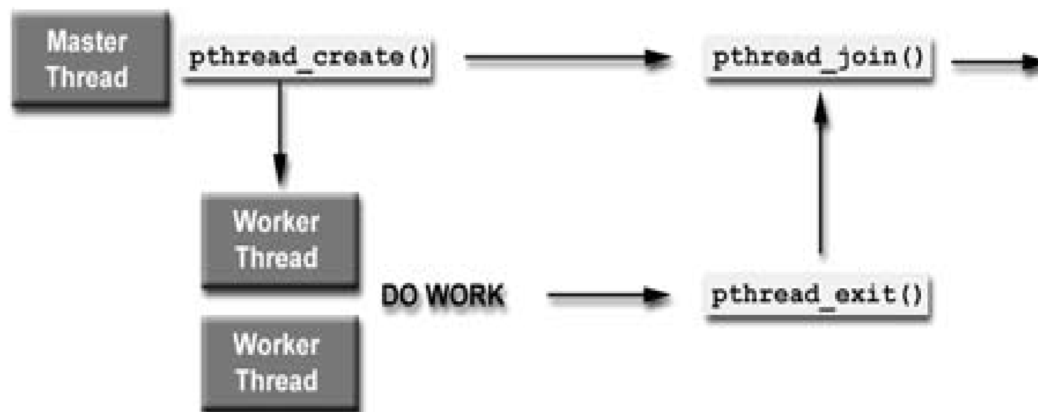
Στη main(), υπάρχει σαφές πρόβλημα, αν η main() ολοκληρωθεί πριν από τα νήματα που γέννησε. Αν δεν καλέσουμε τη pthread_exit() ρητά, και η main() τελειώσει, η διεργασία (και όλα τα νήματα), θα τερματιστούν. Καλώντας τη pthread_exit() στην main(), η διεργασία και όλα τα νήματα της θα διατηρηθούν, ακόμη κι αν όλος ο κώδικας της main() έχει εκτελεστεί.

Συνένωση των Thread

#include <pthread.h>

int pthread_join(pthread_t thread, void **rval_ptr);

Η ένωση ή συνένωση είναι ένας τρόπος για να επιτευχθεί ο συγχρονισμός μεταξύ των νημάτων. Για παράδειγμα:



Εικόνα 5-9: Join PThreads

Η pthread_join() μπλοκάρει το νήμα που καλεί μέχρι το συγκεκριμένο νήμα να τερματιστεί.

Ο προγραμματιστής είναι σε θέση να πάρει τη κατάσταση επιστροφής τερματισμού για το νήμα εάν έχει οριστεί κλήση του νήματος προς τη pthread_exit ().

Η συνένωση ενός νήματος μπορεί να είναι ισότιμη με μια pthread_join() κλήση. Είναι ένα λογικό σφάλμα να επιχειρήσει πολλές ενώσεις στο ίδιο νήμα.

Όταν ένα νήμα έχει δημιουργηθεί, ένα από τα χαρακτηριστικά του, καθορίζει αν μπορεί να συνενωθεί με άλλα ή αποσπαστεί. Μόνο τα νήματα που δημιουργούνται ως “joinable” μπορούν να ενωθούν. Εάν ένα νήμα δημιουργείται ως ανεξάρτητο, δεν μπορεί ποτέ να ενωθεί.

Παράδειγμα

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS      4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
```

```
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR;return code from pthread_create¨%d\n", rc);
         exit(-1);
         }
      }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
         }
      printf("main:completed join with thread:%ld status
:%ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Αποτέλεσμα

```
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Thread 1 starting...
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 1 done. Result = -3.153838e+06
Thread 0 done. Result = -3.153838e+06
```

```
Main: completed join with thread 0 having a status of 0
Main: completed join with thread 1 having a status of 1
Thread 3 done. Result = -3.153838e+06
Thread 2 done. Result = -3.153838e+06
Main: completed join with thread 2 having a status of 2
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
```

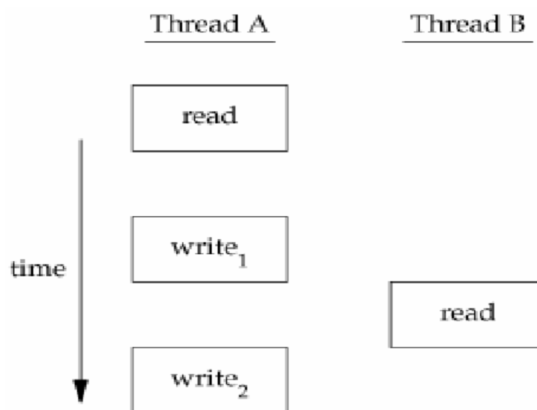
Συχρονισμός Thread (Thread Synchronization)

Όταν πολλαπλά threads μετέχουν στον έλεγχο της ίδιας μνήμης, πρέπει να βεβαιωθούμε ότι κάθε νήμα βλέπει μια συνεκτική εικόνα των δεδομένων του. Εάν κάθε νήμα χρησιμοποιεί μεταβλητές που άλλα νήματα δεν έχουν τροποποιήσει, δεν υπάρχουν προβλήματα συνέπειας. Ομοίως, εάν μια μεταβλητή είναι μόνο για ανάγνωση, δεν υπάρχει πρόβλημα συνέπειας με περισσότερα από ένα νήματα να διαβάζουν την τιμή του την ίδια στιγμή. Ωστόσο, όταν ένα νήμα να τροποποιήσει μια μεταβλητή όπου άλλα νήματα μπορούν να διαβάσουν ή να τροποποιήσουν, τα νήματα θα πρέπει να συγχρονιστούν ώστε να εξασφαλίζεται ότι δεν χρησιμοποιούν κάποια άκυρη τιμή κατά την πρόσβαση τους στα περιεχόμενα της μνήμης.

Όταν ένα νήμα τροποποιεί μια μεταβλητή, τα άλλα νήματα μπορεί να δουν πιθανές διαφορές κατά την ανάγνωση της τιμής της μεταβλητής. Στις αρχιτεκτονικές επεξεργαστή όπου η τροποποίηση διαρκεί περισσότερο από ένα κύκλο μνήμης, αυτό μπορεί να συμβεί όταν μια ανάγνωση μνήμης αναμιγνύεται μεταξύ των κύκλων εγγραφής της μνήμης. Φυσικά, η συμπεριφορά αυτή είναι αρχιτεκτονικά ανεξάρτητη, αλλά τα προγράμματα δεν μπορούν να κάνουν παραδοχές σχετικά με το τι είδους επεξεργαστή ή αρχιτεκτονική χρησιμοποιείται.

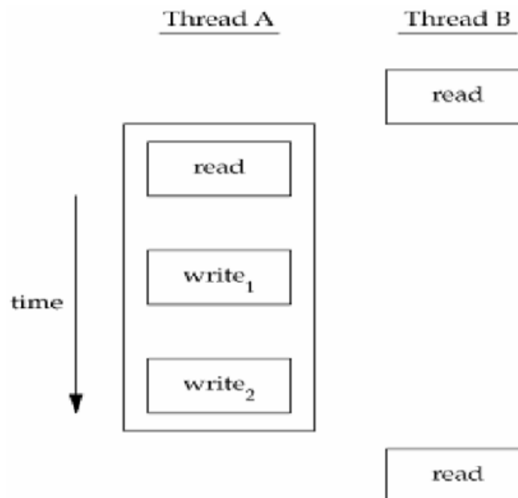
Παράδειγμα

Ένα νήμα διαβάζει τη μεταβλητή και, στη συνέχεια γράφει μια νέα τιμή σε αυτή, αλλά η λειτουργία της εγγραφής διαρκεί δύο κύκλους μνήμης. Αν το νήμα B διαβάσει την ίδια μεταβλητή μεταξύ των δύο κύκλων εγγραφών, θα δούμε μια ασυνεπή τιμή.



Εικόνα 5-10: Interleaved memory cycles with two threads

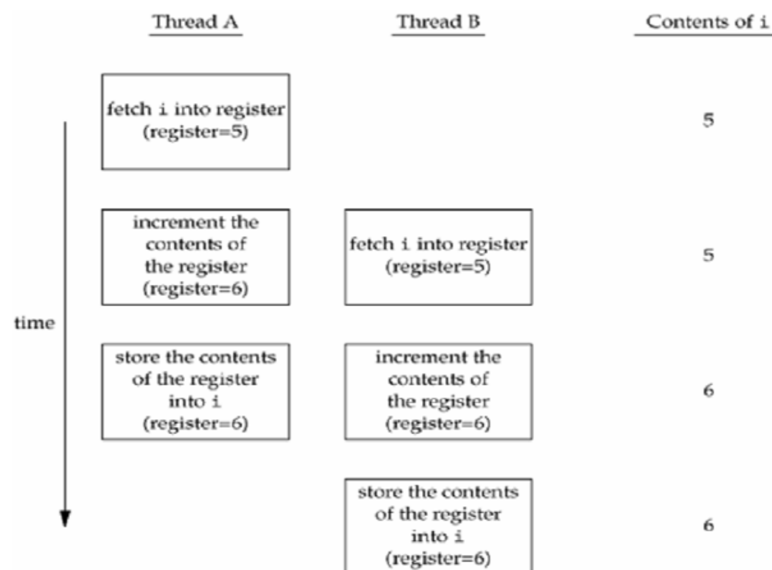
Για την επίλυση αυτού του προβλήματος, τα νήματα πρέπει να χρησιμοποιήσουν lock που επιτρέπει σε ένα μόνο νήμα να αποκτήσει πρόσβαση στη μεταβλητή σε κάποιο χρόνο. Αν θέλει να διαβάσει τη μεταβλητή, το νήμα B εξασφαλίζει ένα lock. Ομοίως, όταν το νήμα ενημερώνει τη μεταβλητή A, εξασφαλίζει το Ίδιο lock. Έτσι, το νήμα B δεν θα είναι σε θέση να διαβάσει τη μεταβλητή μέχρι το νήμα A απελευθερώσει το lock.



Εικόνα 5-11: Two threads synchronizing memory access

Μπορούμε επίσης να συγχρονίσουμε δύο ή περισσότερα νήματα που ενδέχεται να προσπαθήσουν να τροποποιήσουν την ίδια μεταβλητή ταυτόχρονα. Εξετάστε την περίπτωση κατά την οποία προσαυξάνουμε μια μεταβλητή. Η λειτουργία της προσαύξησης συνήθως αποτελείται σε τρεις φάσεις.

1. Ανάγνωση της θέσης μνήμης σε ένα καταχωρητή.
2. Αύξηση της τιμή του σε ένα καταχωρητή.
3. Εγγραφή της νέας τιμής πίσω στη θέση μνήμης.



Εικόνα 5-12: Two unsynchronized threads incrementing the same variable

Αν δύο νήματα προσπαθήσουν να αυξήσουν την ίδια μεταβλητή σχεδόν την ίδια στιγμή χωρίς συγχρονισμό μεταξύ τους, τα αποτελέσματα μπορεί να είναι ασυνεπή. Μπορεί να καταλήξετε με μια τιμή που είναι μια ή δύο φορές μεγαλύτερη από πριν, ανάλογα με την τιμή που παρατηρήθηκε όταν το δεύτερο νήμα ξεκινά τη λειτουργία του. Εάν το δεύτερο νήμα εκτελέσει

το βήμα 1 πριντο πρώτο νήμα εκτελέσει το βήμα 3, το δεύτερο νήμα θα διαβάσει την ίδια αρχική τιμή όπως το πρώτο νήμα, την αυξάνει, και ξαναγράφει, που δεν είναι σωστό.

Εάν η τροποποίηση είναι ατομική, τότε δεν υπάρχει ανταγωνισμός. Εάν τα δεδομένα μας φαίνεται πάντοτε να είναι διαδοχικά συνεπή, τότε δεν χρειάζονται επιπλέον συγχρονισμό. Οι δραστηριότητές μας είναι διαδοχικά συνεπείς όταν πολλαπλά threads δεν μπορεί να παρατηρήσουν ασυνέπειες στα στοιχεία μας. Στα σύγχρονα συστήματα πληροφορικής, οι προσπελάσεις μνήμης λαμβάνουν πολλαπλούς κύκλους, και οι πολυεπεξεργαστές παρεμβάλουν γενικά τους κύκλους μεταξύ των πολλαπλών επεξεργαστών, έτσι δεν είναι εγγυημένο ότι τα δεδομένα μας είναι διαδοχικά συνεπή.

Σε διαδοχικά συνεκτικό περιβάλλον, μπορούμε να εξηγήσουμε τροποποιήσεις στα δεδομένα μας ως μια διαδοχική φάση στις ενέργειες που λαμβάνονται από τη λειτουργία των νημάτων. Μπορούμε να αναφέρουμε εκφράσεις όπως “Το νήμα Α αυξάνει τη μεταβλητή, κατόπιν το νήμα Β αυξάνει η μεταβλητή, έτσι ώστε η τιμή του είναι δύο φορές μεγαλύτερες από πριν” ή ”Το νήμα Β αυξάνει τη μεταβλητή, τότε το νήμα Α αυξάνει η μεταβλητή, έτσι ώστε η τιμή του είναι δύο φορές μεγαλύτερες από πριν”. Καμία πιθανή διάταξη των νημάτων δεν μπορεί να οδηγήσει σε οποιαδήποτε άλλη τιμή της μεταβλητής.

Εκτός από τις αρχιτεκτονικές υπολογιστή, ανταγωνισμοί (data races) μπορούν να προκύψουν από τον τρόπο με τον οποίο τα προγράμματά μας χρησιμοποιούν μεταβλητές, δημιουργώντας χώρους όπου είναι δυνατή η προβολή σε ασυνέπειες. Για παράδειγμα, θα μπορούσαμε να προσαυξήσουμε μια μεταβλητή και στη συνέχεια να πάρουμε μια απόφαση με βάση την τιμή του. Ο συνδυασμός του βήματος αύξησης και του βήματος λήψης αποφάσεων δεν είναι ατομικές, έτσι αυτό ανοίγει ένα παράθυρο όπου μπορούν να προκύψουν ασυνέπειες.

Mutexes (Αμοιβαίος Αποκλεισμός)

Μπορούμε να προστατεύσουμε τα δεδομένα μας και να εξασφαλίσουμε την πρόσβαση με ένα μόνο νήμα κάθε φορά, χρησιμοποιώντας pthreads διασυνδέσεις αμοιβαίου αποκλεισμού. Το mutex είναι βασικά μια “κλειδαριά” που θέτουμε (κλείδωμα) πριν από την πρόσβαση σε ένα κοινό πόρο και απελευθερώνουμε (ξεκλείδωμα) όταν τελειώσουμε. Όντας ορισμένο, κάθε άλλο νήμα που προσπαθεί να καθορίσει τον κοινό πόρο θα μπλοκάρει μέχρι να τον απελευθερώσουμε. Εάν περισσότερα από ένα νήματα είναι αποκλεισμένα όταν θα ξεκλειδώσουμε το mutex, τότε όλα τα νήματα που έχουν μπλοκαριστεί στο κλείδωμα θα γίνουν εκτελέσιμα, και ο πρώτος που θα τρέξει, θα είναι σε θέση να ρυθμίσει το κλείδωμα. Οι άλλοι θα δουν ότι το mutex εξακολουθεί να είναι κλειδωμένο και θα επιστρέψουν στην αναμονή να περιμένουν να γίνει και πάλι διαθέσιμο. Με αυτόν τον τρόπο, μόνο ένα νήμα θα προχωρήσει σε κάποιο χρόνο.

Πολύ συχνά η ενέργεια που εκτελείται από ένα νήμα που του ανήκει ένα mutex είναι η επικαιροποίηση των global μεταβλητών. Αυτός είναι ένας ασφαλής τρόπος για να εξασφαλίσουμε ότι, όταν πολλά νήματα ενημερώνουν την ίδια μεταβλητή, η τελική τιμή είναι η ίδια με εκείνη που θα ήταν αν ένα μόνο νήμα πραγματοποιούσε την ενημέρωση. Οι μεταβλητές που ενημερώνεται ανήκουν στη “κρίσιμη περιοχή”.

Μια τυπική σειρά για τη χρήση ενός mutex έχει ως εξής:
- Δημιουργία και αρχικοποίηση μιας μεταβλητή mutex.
- Πολλά νήματα προσπαθούν να κλειδώσουν το mutex.
- Μόνο ένα πετυχαίνει και αυτό το νήμα κατέχει το mutex.
- Το thread που έχει το mute κάνει τις ενέργειες του.
- Ο κάτοχος ξεκλειδώνει το mute.
- Ένα άλλο νήμα αποκτά το user1 και επαναλαμβάνει τη διαδικασία.

- Τέλος, το user1 καταστρέφεται.

Ρουτίνες

```
#include <pthread.h>

int  pthread_mutex_init(pthread_mutex_t *restrict mutex,
                                  const pthread_mutexattr_t *restrict attr);

int  pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Δημιουργία και Καταστροφή Mutexes

Οι Mutex μεταβλητές πρέπει να δηλώνονται με τον τύπο pthread_mutex_t , και πρέπει να αρχικοποιηθούν πριν να χρησιμοποιηθούν. Υπάρχουν δύο τρόποι για να αρχικοποιηθεί μια mutex μεταβλητή:

1. Στατικά, όταν έχει δηλωθεί. Για παράδειγμα: pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER
2. Δυναμικά, με την pthread_mutex_init() ρουτίνα. Η μέθοδος αυτή επιτρέπει τον καθορισμό των ιδιοτήτων των αντικειμένων mutex, *attr*.

Το *attr* αντικείμενο χρησιμοποιείται για να καθορίσει τις ιδιότητες για το αντικείμενο mutex, και πρέπει να είναι του τύπου pthread_mutexattr_t , εφόσον χρησιμοποιείται (μπορεί να οριστεί ως NULL δηλαδή να το παραλείψουμε). Το πρότυπο pthreads ορίζει τρία προαιρετικά χαρακτηριστικά mutex:

- Πρωτόκολλο: Καθορίζει το πρωτόκολλο που χρησιμοποιείται για την πρόληψη σε αναστροφές προτεραιότητας για ένα mutex.
- Προτεραιότητα: Καθορίζει το ανώτατο όριο προτεραιότητας ενός mutex.
- Διαδικασία κοινόχρηστη: Καθορίζει τη διαδικασία μοιρασιάς ενός mutex.

Locking and Unlocking Mutexes

Η pthread_mutex_lock() ρουτίνα χρησιμοποιείται από ένα νήμα για να αποκτήσει ένα “κλείδωμα” στην καθορισμένη *mutex* μεταβλητή. Αν το mutex είναι ήδη κλειδωμένο από ένα άλλο νήμα, η παρούσα κλήση θα μπλοκάρει τον καλόν νήμα μέχρι το mutex να ξεκλειδώσει.

pthread_mutex_trylock() θα επιχειρήσει να κλειδώσει ένα mutex. Ωστόσο, εάν η mutex είναι ήδη κλειδωμένη, η ρουτίνα θα επιστρέψει αμέσως με ένα "απασχολημένος" κωδικό σφάλματος. Αυτή η τακτική μπορεί να είναι χρήσιμη στην πρόληψη συνθηκών αδιεξόδου, σε μια-αναστροφή της κατάστασης προτεραιότητας (priority inversion).

Η pthread_mutex_unlock() θα ξεκλειδώσει ένα mutex εάν κληθεί από το νήμα που το έχει. Η κλήση σε αυτή τη ρουτίνα απαιτείται μετά αφού ένα νήμα έχει ολοκληρώσει τη χρήση των

προστατευόμενων δεδομένων, αν άλλα νήματα χρειάζεται να αποκτήσουν το mutex για να κάνουν την εργασία τους με τα προστατευόμενα δεδομένα. Ένα σφάλμα θα επιστραφεί αν:

- Αν το mutex ήταν ήδη ξεκλείδωτο.
- Αν το mutex ανήκει σε άλλο νήμα.

Παράδειγμα

```
pthread_mutex_t minimum_value_lock;
...
main() {
....
pthread_mutex_init(&minimum_value_lock, NULL);
....
}
void *find_min(void *list_ptr) {
....
pthread_mutex_lock(&minimum_value_lock);
if (my_min < minimum_value)
minimum_value = my_min;
/* and unlock the mutex */
pthread_mutex_unlock(&minimum_value_lock);
}
```

Deadlock Avoidance of Mutexes

Ένα νήμα θα βρεθεί σε αδιέξοδο αν προσπαθήσει να κλειδώσει το ίδιο mutex δύο φορές, αλλά υπάρχουν λιγότερο εμφανείς τρόποι για τη δημιουργία αδιέξοδων με mutexes. Για παράδειγμα, όταν χρησιμοποιούμε περισσότερα από ένα mutex στα προγράμματά μας, το αδιέξοδο μπορεί να προκύψει, εάν αφήσουμε ένα νήμα να κρατήσει ένα mutex και να κάνει μπλοκ, ενώ προσπαθεί να κλειδώσει ένα δεύτερο mutex την ίδια στιγμή που ένα άλλο νήμα το έχει ενώ το νήμα που έχει το δεύτερο mutex προσπαθεί να κλειδώσει το πρώτο mutex. Κανένα νήμα δε μπορεί να προχωρήσει, γιατί κάθε ένα χρειάζεται ένα πόρο που κατέχεται από το άλλο, έτσι έχουμε ένα αδιέξοδο.

Αδιέξοδα μπορεί να αποφευχθούν με τον προσεκτικό έλεγχο της σειράς με την οποία mutexes είναι κλειδωμένα. Για παράδειγμα, ας υποθέσουμε ότι έχετε δύο mutexes, A και B, που χρειάζεται για να κλειδώσουν την ίδια στιγμή. Εάν όλα τα νήματα κλειδώνουν το mutex A πριν το mutex B, κανένα αδιέξοδο δεν μπορεί να προκύψει από τη χρήση των δύο mutexes (αλλά μπορεί να συμβεί ακόμη αδιέξοδο σε άλλους πόρους). Ομοίως, εάν όλα τα νήματα κλειδώνουν πάντα το mutex B πριν το mutex A, δεν θα υπάρξει αδιέξοδο. Θα έχουμε τη δυνατότητα για πιθανό αδιέξοδο μόνον όταν ένα νήμα επιχειρεί να κλειδώσει τα mutexes με την αντίστροφη σειρά από άλλο νήμα.

Types of Mutexes

- Τα Pthreads υποστηρίζουν τρεις τύπους mutexes - κανονικά, αναδρομικά, και ελέγχου λάθους.
  - ✓ Τα κανονικά mutex καταλήγουν σε αδιέξοδο αν ένα νήμα που έχει ήδη ένα lock επιχειρεί δεύτεροο lock σε αυτό.

- ✓ Τα αναδρομικά mutex επιτρέπουν σε ένα μόνο νήμα να κλειδώσει ένα mutex όσες φορές θέλει. Απλά κάνει μια καταμέτρηση του αριθμού των lock. Ένα lock εγκαταλείπεται από ένα νήμα, όταν η μέτρηση γίνεται μηδέν.
- ✓ Ένα mutex ελέγχου λάθους αναφέρει ένα σφάλμα όταν ένα νήμα με ήδη ένα lock προσπαθεί να το ξανακλειδώσει (σε αντίθεση με το deadlocking στην πρώτη περίπτωση, ή τη χορήγηση της κλειδαριάς, στη δεύτερη περίπτωση).

- Ο τύπος του mutex μπορεί να ρυθμιστεί στο αντικείμενο των χαρακτηριστικών πριν την αρχικοποίηση.

Condition Variables

- Μεταβλητές κατάστασης παρέχουν έναν ακόμη τρόπο προκειμένου να συγχρονιστούν τα νήματα. Ενώ τα mutexes υλοποιούν συγχρονισμό με τον έλεγχο πρόσβασης του νήματος στα δεδομένα, μεταβλητές καταστάσεων επιτρέπουν στα νήματα να συγχρονιστούν με βάση την πραγματική αξία των δεδομένων.
- Μια κατάσταση μεταβλητής επιτρέπει σε ένα νήμα να μπλοκάρει τον εαυτό του μέχρι να φθάσουν τα συγκεκριμένα δεδομένα σε μια προκαθορισμένη κατάσταση.
- Μια κατάσταση μεταβλητής σχετίζεται με κάποιο όριο. Όταν το όριο γίνεται πραγματικότητα, η μεταβλητή κατάστασης χρησιμοποιείται για να σηματοδοτήσει ένα ή περισσότερα νήματα που περιμένουν τη συγκεκριμένη συνθήκη.
- Μια απλή μεταβλητή κατάστασης μπορεί να σχετίζεται με περισσότερα από ένα κατηγορήματα.
- Μια μεταβλητή κατάστασης έχει πάντα ένα mutex που συνδέεται με αυτό. Ένα νήμα κλειδώνει αυτό το mutex και δοκιμάζει το όριο που ορίζεται από τη κοινή μεταβλητή.
- Αν το όριο δεν αληθεύει, το νήμα περιμένει την μεταβλητή κατάστασης που σχετίζεται με το όριο χρησιμοποιώντας τη pthread_cond_wait συνάρτηση.

Η ακολουθία για τη χρήση μεταβλητών κατάστασης φαίνεται στο πίνακα παρακάτω::

<u>Κυρίως Νήμα</u>

- Δηλώνει και αρχικοποιεί τα ενιαία δεδομένα / μεταβλητές που απαιτούν συγχρονισμό (όπως η "καταμέτρηση").
- Αναγνωρίζει και αρχικοποιεί ένα αντικείμενο μεταβλητής κατάστασης.
- Δηλώνει και προετοιμάζει ένα σχετικό mutex.
- Δημιουργεί τα νήματα Α και Β.

Νήμα Α

- Εργάζεται μέχρι το σημείο όπου μια συγκεκριμένη κατάσταση πρέπει να γίνει (π.χ. μια "καταμέτρηση" πρέπει να καταλήξει σε συγκεκριμένη τιμή).
- Κλειδώνει το σχετικό mutex και ελέγχει τη καθολική μεταβλητή.
- Καλεί τη pthread_cond_wait() για να εκτελέσει μια αναστέλλουσα αναμονή για ένα σήμα από το νήμα-Β. Σημειώστε ότι μια κλήση στη pthread_cond_wait() ξεκλειδώνει αυτόματα και ατομικά τη σχετική μεταβλητή mutex έτσι ώστε να μπορεί να χρησιμοποιηθεί από νήμα-Β.
- Όταν σηματοδοτηθεί, ξυπνά. Το mutex είναι και αυτόματα και ατομικά κλειδωμένο.
- Αποκλειστικά ξεκλειδώνει το mutex
- Συνέχεια

Νήμα Β

- Εργάζεται
- Κλειδώνει το σχετικό mutex
- Αλλάζει τη τιμή της ενιαίας μεταβλητής στην οποία περιμένει το νήμα-Α.
- Ελέγχει τη τιμή της ενιαίας μεταβλητής του νήματος-Α το οποίο περιμένει. Εάν πληροί την επιθυμητή συνθήκη, σηματοδοτεί το νήμα-Α.
- Ξεκλειδώνεται το mutex.
- Συνέχεια

Κυρίως Νήμα

Εγγραφή / Συνέχεια

Ρουτίνες

```
#include <pthread.h>

int  pthread_cond_wait(pthread_cond_t *cond,  pthread_mutex_t *mutex);

int  pthread_cond_signal(pthread_cond_t *cond);

int  pthread_cond_broadcast(pthread_cond_t *cond);

int  pthread_cond_init(pthread_cond_t *cond,
                       const pthread_condattr_t *attr);

int  pthread_cond_destroy(pthread_cond_t *cond);
```

Δημιουργία και Καταστροφή Μεταβλητών Κατάστασης

Οι μεταβλητές κατάστασης πρέπει να δηλώνονται με τον τύπο pthread_cond_t , και πρέπει να αρχικοποιηθούν πριν να χρησιμοποιηθούν. Υπάρχουν δύο τρόποι για να αρχικοποιηθεί μια mutex μεταβλητή:

1. Στατικά, όταν έχει δηλωθεί. Για παράδειγμα: pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
2. Δυναμικά, με την pthread_ cond _init() ρουτίνα. Το id της κατάστασης μεταβλητής επιστρέφεται στο thread μέσω της παραμέτρου *cond*. Η μέθοδος αυτή επιτρέπει τον καθορισμό των ιδιοτήτων της μεταβλητής κατάστασης αντικειμένων, *attr*.

Το *attr* αντικείμενο χρησιμοποιείται για να καθορίσει τις ιδιότητες του αντικειμένου. Υπάρχει μόνο ένα γνώρισμα που ορίζεται για τις μεταβλητές κατάστασης: κοινόχρηστη-διεργασία που επιτρέπει στη μεταβλητή κατάστασης να παρακολουθείται από άλλα threads, άλλων διεργασιών. Το αντικείμενο γνωρίσματος πρέπει να είναι του τύπου pthread_condattr_t, εφόσον χρησιμοποιείται (μπορεί να οριστεί ως NULL δηλαδή να το παραλείψουμε).

Σημειώστε ότι όλες οι εφαρμογές μπορεί να μη χρησιμοποιήσουν το γνώρισμα της κοινόχρηστης-διεργασίας.

Τέλος η pthread_cond_destroy () θα πρέπει να χρησιμοποιηθεί για την ελευθέρωση μιας μεταβλητής καταστάσεων υπό τον όρο ότι δεν είναι πλέον απαραίτητη.

Αναμονή και σηματοδότηση για τις Μεταβλητές Καταστάσεων

- Η pthread_cond_wait() μπλοκάρει το καλούν νήμα μέχρι η συγκεκριμένη *κατάσταση* σηματοδοτηθεί. Αυτή η ρουτίνα θα πρέπει να καλείται όταν το *mutex* είναι κλειδωμένο, και απελευθερώνεται αυτόματα ενώ το mutex περιμένει. Αφού το σήμα ληφθεί και το νήμα ξυπνήσει, το *mutex* κλειδώνει αυτόματα για χρησιμοποιηθεί από το νήμα. Ο προγραμματιστής είναι τότε υπεύθυνος για το ξεκλείδωμα του *mutex*, όταν το νήμα έχει τελειώσει με αυτό.

- Η pthread_cond_signal() ρουτίνα χρησιμοποιείται για να σηματοδοτεί (ή να ξυπνήσει) ένα άλλο νήμα που είναι σε αναμονή για τη κατάσταση μεταβλητής. Θα πρέπει να κληθεί αφότου το *mutex* είναι κλειδωμένο και πρέπει να ξεκλειδώσει το *mutex* προκειμένου η pthread_cond_wait() ρουτίνα να ολοκληρωθεί.

- Η pthread_cond_broadcast() ρουτίνα θα πρέπει να χρησιμοποιηθεί αντί του pthread_cond_signal() εάν περισσότερα από ένα νήμα είναι σε αναστέλλουσα κατάσταση αναμονής.

- Είναι ένα λογικό σφάλμα να καλέσουμε τη pthread_cond_signal() πριν από την κλήση της pthread_cond_wait().

  Η σωστή χρήση κλειδώματος και ξεκλειδώματος των μεταβλητών mutex είναι απαραίτητη όταν χρησιμοποιούμε αυτές τις ρουτίνες. Για παράδειγμα:

  - Η αποτυχία να κλειδώσουμε το mutex πριν από την κλήση pthread_cond_wait() μπορεί να μην ανασταλεί

- Η αποτυχία να ξεκλειδώσουμε το mutex μετά την κλήση της pthread_cond_signal() μπορεί να αποτρέψει μια αντίστοιχη pthread_cond_wait() ρουτίνα να ολοκληρωθεί (θα παραμείνει δεσμευμένη).

<u>Παράδειγμα</u>

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when
condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %ld, count = %d  Threshold
reached.\n",
             my_id, count);
      }
    printf("inc_count(): thread %ld, count = %d, unlocking
mutex\n",
      my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some "work" so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);
```

```
  /*
  Lock mutex and wait for signal.  Note that the
pthread_cond_wait
  routine will automatically and atomically unlock mutex while it
waits.
  Also, note that if COUNT_LIMIT is reached before this routine
is run by
  the waiting thread, the loop will be skipped to prevent
pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal
received.\n", my_id);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id,
count);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable
state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);

}
```

Αποτέλεσμα

```
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 going into wait...
inc_count(): thread 3, count = 3, unlocking mutex
inc_count(): thread 2, count = 4, unlocking mutex
inc_count(): thread 3, count = 5, unlocking mutex
inc_count(): thread 2, count = 6, unlocking mutex
inc_count(): thread 3, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 3, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
inc_count(): thread 3, count = 11, unlocking mutex
inc_count(): thread 2, count = 12  Threshold reached. Just sent
signal.
inc_count(): thread 2, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received.
watch_count(): thread 1 count now = 137.
inc_count(): thread 3, count = 138, unlocking mutex
inc_count(): thread 2, count = 139, unlocking mutex
inc_count(): thread 3, count = 140, unlocking mutex
inc_count(): thread 2, count = 141, unlocking mutex
inc_count(): thread 3, count = 142, unlocking mutex
inc_count(): thread 2, count = 143, unlocking mutex
inc_count(): thread 3, count = 144, unlocking mutex
inc_count(): thread 2, count = 145, unlocking mutex
Main(): Waited on 3 threads. Final value of count = 145. Done.
```

Barriers (Φραγμός)

- Όπως και στο MPI, τα barrier συγκρατούν ένα νήμα μέχρι όλα τα νήματα που συμμετέχουν στο barrier να το προλάβουν.
- Τα barriers μπορούν να εφαρμοστούν χρησιμοποιώντας ένα μετρητή, ένα mutex και μία μεταβλητή κατάστασης.
- Ένας ενιαίος ακέραιος χρησιμοποιείται για να παρακολουθεί τον αριθμό των threads που έχουν φθάσει στο barrier.
- Αν ο αριθμός είναι μικρότερος από τον συνολικό αριθμό των νημάτων, τα νήματα που εκτελούν μια κατάσταση αναμονής.
- Το τελευταίο νήμα που εισέρχεται (τοποθετεί έναν μετρητή για αριθμό των νημάτων) ξυπνάει όλα τα νήματα που χρησιμοποιούν μια κατάσταση εκπομπής.

Παράδειγμα

```
typedef struct {
pthread_mutex_t count_lock;
pthread_cond_t ok_to_proceed;
int count;
} mylib_barrier_t;
void mylib_init_barrier(mylib_barrier_t *b) {
b -> count = 0;
pthread_mutex_init(&(b -> count_lock), NULL);
pthread_cond_init(&(b -> ok_to_proceed), NULL);
}
```

```
void mylib_barrier (mylib_barrier_t *b, int num_threads) {
pthread_mutex_lock(&(b -> count_lock));
b -> count ++;
if (b -> count == num_threads) {
b -> count = 0;
pthread_cond_broadcast(&(b -> ok_to_proceed));
}
else
while (pthread_cond_wait(&(b -> ok_to_proceed),
&(b -> count_lock)) != 0);
pthread_mutex_unlock(&(b -> count_lock));
}
```

5.3 IPC API Linux

Διαχείριση Thread: Η διαχείριση ενός παράλληλου προγράμματος γίνεται όπως προαναφέρθηκε παραπάνω (ενότητα 4.1.1) με τη βοήθεια των διεργασιών που είναι είτε ανεξάρτητα προγράμματα (διαφορετικά προγράμματα ή πιο συνηθισμένα ανεξάρτητα αντίγραφα του ίδιου προγράμματος) είτε διαδικασίες (procedures).

Κυριότερες Βιβλιοθήκες IPC

Δημιουργία Διεργασίας

#include <unistd.h>

pid_t fork(void);

Μια υφιστάμενη διαδικασία μπορεί να δημιουργήσει μια νέα καλώντας τη συνάρτηση fork. Η νέα διεργασία που δημιουργείται από τη fork λέγεται "παιδί". Η λειτουργία αυτή καλείται μια φορά αλλά επιστρέφει δύο φορές. Η μόνη διαφορά στην επιστροφή είναι ότι η τιμή που επιστρέφεται στο παιδί είναι 0, ενώ η τιμή επιστροφής της μητρικής διεργασίας είναι το ID της διαδικασίας του νέου παιδιού. Ο λόγος που η διεργασία παιδί επιστρέφει την ID της στη μητρική διεργασία είναι ότι μια διεργασία μπορεί να έχει περισσότερα από ένα παιδιά, και δεν υπάρχει συνάρτηση που να επιτρέπει σε μια διεργασία να δέχεται το ID των παιδιών της. Ο λόγος που η fork επιστρέφει 0 στο παιδί είναι ότι το παιδί μπορεί να έχει μόνο μια μητρική διεργασία.

Τερματισμός Διεργασίας

#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);

Η μητρική διεργασία περιμένει για μια διεργασία παιδί να τερματίσει. Όπως αναφέρεται, η μόνη νόμιμη αναμονή είναι όταν ένας γονέας περιμένει για ένα παιδί. Ένα παιδί δεν μπορεί να περιμένει για ένα γονέα ή αδελφό, μια διεργασία δεν μπορεί να περιμένει για ένα "εγγόνι". Η λειτουργία wait() θα μπλοκάρει (αν χρειάζεται), μέχρι μια διεργασία παιδί να τελειώσει, επιστρέφοντας τη pid του παιδιού και αποθηκεύοντας τη κατάσταση τερματισμού στο όρισμα *status, όταν η κατάσταση δεν είναι NULL. Η waitpid λειτουργία μπορεί να περιμένει για ένα συγκεκριμένο pid παιδί ή για κάθε παιδί (όταν η παράμετρος είναι pid == -1). Μια κοινή χρήση της wait είναι για μια αρχική διεργασία να περιμένει όλα τα παιδιά της

πριν η ίδια κάνει exit. Εάν όλες οι διεργασίες της εφαρμογής ακολουθούν το καθεστώς αυτό, στη συνέχεια, όταν μια κλήση κελύφους επανεμφανιστεί, ξέρουμε, ότι δεν υπάρχουν μακρόχρονες διεργασίες που εξακολουθούν να λειτουργούν.

Κάθε διεργασία κατά τον τερματισμό της επιστρέφει έναν *κωδικό εξόδου* (exit code) ώστε να μπορεί να διαπιστωθεί ο λόγος και ο τρόπος τερματισμού της. Ο οικειοθελής τερματισμός μιας διεργασίας μπορεί να γίνει με την κλήση exit():

```
#include <unistd.h>

void _exit (int status);
```

Η κατάσταση τερματισμού status είναι κατά σύμβαση 0 όταν η διεργασία τερματίζει κανονικά, ή διάφορη του 0 όταν προκύψει κάποιο λάθος. Η κλήση exit() κλείνει αυτόματα όλα τα ανοικτά αρχεία της διεργασίας και απελευθερώνει όλα τα τμήματα μνήμης που είχε δεσμεύσει κατά τη διάρκεια εκτέλεσής της. Σημειώνουμε ωστόσο ότι όταν μια διεργασία τερματιστεί, δεν καταστρέφεται ολοκληρωτικά αλλά παραμένει στην τερματισμένη κατάσταση (zombie state). Όταν ο πατέρας εκτελέσει την κλήση wait() και συλλέξει τον κωδικό εξόδου, η διεργασία καταστρέφεται ολοκληρωτικά, ελευθερώνοντας και την εγγραφή του πίνακα διεργασιών που κατείχε.

System V Shared Memory

Η κοινόχρηστη μνήμη επιτρέπει σε δύο ή περισσότερες διεργασίες να μοιράζονται μια δεδομένη περιοχή της μνήμης. Αυτή είναι η ταχύτερη μορφή του IPC, επειδή τα δεδομένα δεν χρειάζεται να αντιγραφούν μεταξύ του πελάτη και του διακομιστή. Το μόνο που χρειάζεται η κοινόχρηστη μνήμη είναι ο συγχρονισμός πρόσβασης σε μια συγκεκριμένη περιοχή μεταξύ πολλών διαδικασιών. Εάν ο διακομιστής έχει πρόσβαση εκείνη τη στιγμή σε μια κοινόχρηστη περιοχή μνήμης, ο πελάτης δεν θα πρέπει να προσπαθήσει να έχει πρόσβαση στα δεδομένα μέχρι ο διακομιστής να τελειώσει.

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int flag);
```

Ένα κοινόχρηστο τμήμα μνήμης δημιουργείται, ή ένα υπάρχον προσεγγίζεται, από την shmget συνάρτηση. Η τιμή επιστροφής είναι ένας ακέραιος αριθμός ονομάζεται αναγνωριστικό κοινόχρηστης μνήμης και χρησιμοποιείται από άλλες λειτουργίες shmXXX που αναφέρονται σε αυτό το τμήμα. Η παράμετρος size είναι το μέγεθος του κοινόχρηστου τμήματος μνήμης σε bytes. Όταν μια νέα κοινόχρηστη μνήμη δημιουργείται, μια τιμή διαφορετική από το μηδέν για το μέγεθος της μνήμης πρέπει να προσδιορίζεται. Εάν γίνεται αναφορά σ' ένα ήδη υπάρχον τμήμα κοινόχρηστης μνήμης, το μέγεθος θα πρέπει να είναι ίσο με το μηδέν. Όταν ένα νέο τμήμα δημιουργηθεί, το περιεχόμενο του τμήματος αρχικοποιείται με μηδενικά. Το όρισμα shmflg διευκρινίζει τις αρχικές άδειες πρόσβασης και τα flags ελέγχου, π.χ IPC-CREAT or IPC-CREAT I IPC-EXCL. Η shmget δημιουργεί ή ανοίγει ένα κοινόχρηστο τμήμα μνήμης, αλλά δεν παρέχει πρόσβαση στο τμήμα για την κλήση μιας διαδικασίας. Αυτό είναι σκοπός της λειτουργίας shmat, την οποία περιγράφουμε στη συνέχεια.

Παράδειγμα

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
...
key_t key; /* key to be passed to shmget() */
int shmflg; /* shmflg to be passed to shmget() */
int shmid; /* return value from shmget() */
int size; /* size to be passed to shmget() */
...
key = ...
size = ...
shmflg = ...
if ((shmid = shmget (key, size, shmflg)) == -1)
{
 perror("shmget: shmget failed"); exit(1);
}
else
{
 (void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
 exit(0);
}
```

```
#include <sys/shm.h>

void *shmat(int shmid, const void *addr, int flag);
```

Όταν ένα κοινόχρηστο τμήμα μνήμης έχει δημιουργηθεί ή ανοιχθεί από τη shmget, προσαρτούμε ,το χώρο των διευθύνσεων μας καλώντας τη shmat. Η shmid είναι μια τιμή αναγνώρισης που επιστρέφεται από τη shmget. Η τιμή που επιστρέφεται από τη shmat είναι η αρχική διεύθυνση του κοινόχρηστου τμήματος μνήμης εντός της διεργασίας που καλείται.

```
#include  <sys/shm.h>

int  shmdt  (const  void *shmaddr)  ;
```

Όταν μια διεργασία έχει τελειώσει με ένα κοινό τμήμα μνήμης, το τμήμα μνήμης ελευθερώνεται καλώντας τη shmdt. Όταν μια διεργασία τερματίζει, όλα τα κοινά τμήματα μνήμης που έχουν προς το παρόν αποδοθεί στη διεργασία ανεξαρτητοποιούνται. Αυτή η κλήση δεν διαγράφει το κοινόχρηστο τμήμα μνήμης. Η διαγραφή αυτή συνοδεύεται καλώντας την shmctl εντολή που θα περιγράψουμε στην επόμενη ενότητα.

Παράδειγμα

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
static struct state {
/* Internal record of attached segments. */
 int shmid; // shmid of attached segment
 char *shmaddr; // attach point
 int shmflg; // flags used on attach
 } ap[MAXnap]; // State of current attached segments.
int nap; // Number of currently attached segments.
...
char *addr; /* address work variable */
register int i; /* work area */
register struct state *p; /* ptr to current state entry */
```

```
...
p = &ap[nap++];
p->shmid = ...
p->shmaddr = ...
p->shmflg = ...
p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
if(p->shmaddr == (char *)-1)
{
 perror("shmop: shmat failed");
 nap--;
}
else
 (void) fprintf(stderr, "shmop: shmat returned %#8.8x\n",
p->shmaddr);
...
i = shmdt(addr);
if(i == -1)
{
 perror("shmop: shmdt failed");
}
else
{
 (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);
 for (p = ap, i = nap; i--; p++)
   if (p->shmaddr == addr) *p = ap[--nap];}
```

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Η shmctl() χρησιμοποιείται για να αλλάξει τις άδειες χρήσης και άλλα χαρακτηριστικά ενός κοινού τμήματος μνήμης. Η διεργασία πρέπει να έχει το κατάλληλο shmid (ID) του ιδιοκτήτη ή του δημιουργού του κοινού τμήματος για να εκτελέσει αυτήν την εντολή. Το όρισμα cmd είναι μία από τις παρακάτω εντολές ελέγχου:

SHM_LOCK

Κλειδώνει το συγκεκριμένο κοινό τμήμα μνήμης στη μνήμη.

SHM_UNLOCK

Ξεκλειδώνει το κοινό τμήμα μνήμης.

IPC_STAT

Επιστρέφει τις πληροφορίες κατάστασης που περιλαμβάνονται στη δομή ελέγχου και τις τοποθετεί στον buffer με δείκτη buf. Η διεργασία πρέπει να έχει άδεια ανάγνωσης στο τμήμα για να εκτελέσει αυτή η εντολή.

IPC_SET

Αρχικοποίηση των αδειών χρήσης και πρόσβασης του κοινού τμήματος μνήμης.

IPC_RMID

Αφαιρεί το κοινό τμήμα μνήμης. Ο buffer με δείκτη buf είναι μία δομή του τύπου struct shmid_ds που ορίζεται στο <sys/shm.h> Ο ακόλουθος κώδικας επεξηγεί την χρήση της shmctl ():

Παράδειγμα

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
...
int cmd; /* command code for shmctl() */
int shmid; /* segment ID */
struct shmid_ds shmid_ds; /* shared memory data structure to
... hold results */
shmid = ...
cmd = ...
if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1)
{
 perror("shmctl: shmctl failed");
 exit(1);
}
```

5.4 API Message Passing Interface

Η έννοια της διεργασίας

Μια εφαρμογή MPI μπορεί να θεωρηθεί ως μια συλλογή από ταυτόχρονες διεργασίες που επικοινωνούν. Ένα πρόγραμμα περιέχει κώδικα, γραμμένο από τον προγραμματιστή της εφαρμογής, ο οποίος χρησιμοποιεί μια βιβλιοθήκη από ρουτίνες επικοινωνίας ανάμεσα στις διεργασίες που παρέχει η υλοποίηση MPI. Σε κάθε διεργασία δίνεται ένας μοναδικός ακέραιος (rank), από 0 ως n-1, αν n είναι οι διεργασίες που αποτελούν την εφαρμογή. Το αναγνωριστικό αυτό χρησιμοποιείται από τις διεργασίες του MPI για να αναγνωρίζουν η μία την άλλη όταν στέλνουν ή λαμβάνουν μηνύματα, όταν εκτελούν συλλογικές λειτουργίες και γενικά όταν συνεργάζονται. Οι διεργασίες του MPI μπορούν να εκτελούνται στον ίδιο ή σε διαφορετικούς επεξεργαστές ταυτόχρονα.

Communicator

Μια σημαντική απαίτηση σε όλα τα συστήματα ανταλλαγής μηνυμάτων είναι να εγγυώνται έναν ασφαλή χώρο επικοινωνίας όπου μη σχετιζόμενα μεταξύ τους μηνύματα να είναι διαχωρισμένα το ένα από το άλλο. Για παράδειγμα, τα μηνύματα βιβλιοθήκης μπορούν να αποσταλούν και να ληφθούν χωρίς παρεμβολές από άλλα μηνύματα που παράγονται στο σύστημα. Στο MPI, όπου δεν υπάρχει εικονική μηχανή, η χρήση μιας απλής διεύθυνσης σε κάθε μήνυμα δεν είναι αρκετή για τον ασφαλή διαχωρισμό των μηνυμάτων βιβλιοθήκης από τα μηνύματα του χρήστη. Για την επίτευξη της απαίτησης αυτής, εισάγεται στο MPI η έννοια του communicator. Ο communicator μπορεί να θεωρηθεί ως μια συσχέτιση ανάμεσα σε μια ομάδα διεργασιών στο πλαίσιο μιας επικείμενης επικοινωνίας. Ο communicator είναι ένα αντικείμενο που μπορεί να προσπελαστεί μέσω ενός χειριστή τύπου MPI_COMM.

Βαθμός Διεργασίας

Στις διεργασίες που ανήκουν σε κάποιον communicator ανατίθενται συνεχόμενοι ακέραιοι από μηδέν ως n, όπου n το μέγεθος της ομάδας του communicator πλην ένα, ως

αναγνωριστικά. Τα αναγνωριστικά αυτά, που ονομάζονται βαθμοί (ranks), χρησιμοποιούνται για το διαχωρισμό των διεργασιών μέσα στην ίδια ομάδα. Για παράδειγμα, σε διεργασίες με διαφορετικό βαθμό μπορούν να ανατεθούν διαφορετικές εργασίες για εκτέλεση. Μια διεργασία μπορεί να βρει το βαθμό της μέσα σε έναν communicator καλώντας τη συνάρτηση MPI_Comm_Rank ως εξής:

```
MPI_Comm communicator;          /* communicator handle */
int my_rank;                    /* ο βαθμός της καλούσας διεργασίας */
MPI_Comm_rank(communicator, &my_rank);
```

<u>Μέγεθος μιας ομάδας</u>
Το μέγεθος μιας ομάδας συσχετιζόμενης με έναν communicator μπορεί να βρεθεί καλώντας τη συνάρτηση MPI_Comm_size(). Η συνάρτηση αυτή δέχεται έναν communicator και επιστρέφει το μέγεθος της αντίστοιχης ομάδας ως εξής:

```
MPI_Comm communicator;        /*communicator handle */
int number_of_tasks;
MPI_Comm_size(communicator, &number_of_tasks);
```

<u>Επικοινωνία</u>
Η επικοινωνία ανάμεσα στις διεργασίες του MPI βασίζεται στην τεχνική της ανταλλαγής μηνυμάτων. Το MPI χρησιμοποιεί ένα πλούσιο σύνολο συναρτήσεων για την αποστολή και τη λήψη μηνυμάτων. Η επικοινωνία μεταξύ δύο διεργασιών περιλαμβάνει τα εξής στοιχεία:
1. Αποστολέας, ο οποίος προσδιορίζεται συνήθως από το βαθμό του.
2. Παραλήπτης, ο οποίος προσδιορίζεται επίσης από το βαθμό του.
3. Τα δεδομένα του μηνύματος.
4. Η ετικέτα του μηνύματος, η οποία χρησιμοποιείται για την ταξινόμηση των μηνυμάτων που ανταλλάσσονται μεταξύ δύο διεργασιών.
5. O Communicator, ο οποίος παρέχει το γενικό πλαίσιο για την επικοινωνία.

<u>Blocking Επικοινωνία</u>
Οι βασικές συναρτήσεις για την αποστολή και τη λήψη μηνυμάτων στο MPI είναι η blocking send και η blocking receive. Υπάρχουν διάφορες παραλλαγές των συναρτήσεων αυτών που εξυπηρετούν διαφορετικά είδη επικοινωνίας μεταξύ διεργασιών. Το MPI υποστηρίζει τις παρακάτω μεθόδους:

Standard Send
Με τη μέθοδο αυτή, ο αποστολέας θα «μπλοκάρει» μέχρι το μήνυμά του να αντιγραφεί με ασφάλεια είτε στον αντίστοιχο χώρο αποθήκευσης του παραλήπτη είτε σε έναν προσωρινό χώρο αποθήκευσης του συστήματος. Αν το μήνυμα πρέπει να αποθηκευτεί ή όχι εξαρτάται από την εκάστοτε υλοποίηση του MPI. Μόλις γίνει επιστροφή της κλήσης send, η προσωρινή μνήμη του αποστολέα μπορεί να επαναχρησιμοποιηθεί για άλλους σκοπούς από τον αποστολέα. Η συνάρτηση αυτή ορίζεται ως εξής:

```
MPI_Recv(buf, count, data_type, from_whom, tag, communicator,&status);
```

Η συνάρτηση αυτή θα επιλέξει ένα μήνυμα από τον αποστολέα με rank from_whom με ετικέτα tag και θα το αποθηκεύσει στη διεύθυνση που ξεκινά από buf. Πρόσθετη πληροφορία για την κατάσταση της λειτουργίας θα επιστραφεί στη μεταβλητή status. Η μεταβλητή αυτή είναι συνήθως μια δομή που αποτελείται από δύο πεδία MPI_SOURCE και MPI_TAG που

αντιστοιχούν στο βαθμό του αποστολέα και στην ετικέτα του μηνύματος. Και εδώ ο αποστολέας και ο παραλήπτης πρέπει να ανήκουν στον communicator.

Non-Blocking Επικοινωνία

Το MPI υποστηρίζει non-blocking επικοινωνία κατά την οποία μια διεργασία μπορεί να ξεκινήσει μια λειτουργία αποστολής ή παραλαβής, να εκτελέσει στη συνέχεια κάποια άλλη εργασία και έπειτα να επιστρέψει για να ελέγξει την κατάσταση της λειτουργίας ανταλλαγής μηνύματος. Αυτές οι δυνατότητες μπορούν να χρησιμοποιηθούν σε συνδυασμό με τις standard λειτουργίες και δεν είναι ανάγκη να απαντώνται σε ζεύγη. Η χρήση μιας non-blocking λειτουργίας αποστολής (λήψης) μπορεί να επιτευχθεί σε τρία βήματα:

1. Έναρξη λειτουργίας αποστολής (λήψης) καλώντας τη συνάρτηση MPI_I send() (MPI_Irecv()).
2. Εκτέλεση κάποιων υπολογισμών κατά τη διάρκεια του χρόνου επικοινωνίας.
3. Ολοκλήρωση της επικοινωνίας καλώντας την MPI_Wait() ή την MPI_Test().

Έναρξη της Nonblocking Επικοινωνίας

Η συνάρτηση που χρησιμοποιείται για μια nonblocking αποστολή είναι η εξής:

MPI_Isend(buf, count, data_type, to_whom, tag, communicator,&request)

Μια κλήση στη συνάρτηση αυτή θα επιστρέψει πριν αντιγραφεί το μήνυμα από την προσωρινή μνήμη του αποστολέα. Όλα τα ορίσματα της συνάρτησης αυτής, εκτός από ένα, έχουν το ίδιο νόημα όπως στις υπόλοιπες συναρτήσεις αποστολής. Το τελευταίο όρισμα, request, είναι ένα αντικείμενο του συστήματος που επιστρέφεται όταν καλείται η συνάρτηση και μπορεί να προσπελαστεί μέσω ενός χειριστή. Χρησιμοποιείται για την αναγνώριση και το συνδυασμό διάφορων λειτουργιών επικοινωνίας. Ομοίως, για nonblocking λήψη ενός μηνύματος χρησιμοποιείται η εξής συνάρτηση:

MPI_Irecv(buf, count, data_type, from_whom, tag, communicator,&request)

Μια κλήση στη συνάρτηση αυτή θα ξεκινήσει τη λειτουργία λήψης. Θα επιστρέψει αμέσως χωρίς να χρειάζεται να περιμένει να αποθηκευτεί κάποιο μήνυμα στην προσωρινή μνήμη λήψης. Όπως και στην περίπτωση της αποστολής, το όρισμα request μπορεί να χρησιμοποιηθεί αργότερα για την εξακρίβωση της κατάστασης της λειτουργίας.

Ολοκλήρωση της Non-blocking Επικοινωνίας

Το MPI παρέχει ρουτίνες για τον έλεγχο της ολοκλήρωσης των λειτουργιών επικοινωνίας που έχουν ξεκινήσει. Η ολοκλήρωση μιας non-blocking λειτουργίας αποστολής υποδηλώνει ότι τα δεδομένα έχουν αντιγραφεί από τη μνήμη του αποστολέα και η μνήμη αυτή μπορεί να χρησιμοποιηθεί από άλλες λειτουργίες. Αντίστοιχα, η ολοκλήρωση μιας non-blocking λειτουργίας λήψης υποδηλώνει ότι τα δεδομένα έχουν ήδη τοποθετηθεί στον κατάλληλο χώρο αποθήκευσης ώστε ο παραλήπτης να είναι σε θέση να τα διαβάσει. Η ακόλουθη συνάρτηση χρησιμοποιείται για τον έλεγχο της ολοκλήρωσης μιας nonblocking λειτουργίας:

MPI_Test(request, &flag, &status)

Μια κλήση στη συνάρτηση αυτή επιστρέφει την τρέχουσα κατάσταση της λειτουργίας που δηλώνεται μέσω της μεταβλητής request. Το όρισμα flag θα τεθεί true αν η επικοινωνία έχει ολοκληρωθεί, αλλιώς θα πάρει την τιμή false. Το πεδίο status θα επιστρέψει πρόσθετη πληροφορία για την κατάσταση της λειτουργίας. Η ακόλουθη συνάρτηση θα περιμένει μέχρι η επικοινωνία να ολοκληρωθεί:

MPI_Wait(request, &status)

Μια κλήση στην MPI_Wait() θα επιστρέψει όταν ολοκληρωθεί η λειτουργία που ορίζεται από τη μεταβλητή request.

Συγχρονισμός

Οι δομές συγχρονισμού χρησιμοποιούνται για την επιβολή μιας συγκεκριμένης σειράς εκτέλεσης ανάμεσα στις λειτουργίες των παράλληλων διεργασιών. Σε μερικές περιπτώσεις, οι παράλληλες διεργασίες απαιτείται να συγχρονιστούν μεταξύ τους σε κάποιο σημείο κατά τη διάρκεια της εκτέλεσης. Τα μέλη μιας ομάδας ίσως χρειαστεί να περιμένουν σε κάποιο σημείο συγχρονισμού μέχρι όλες οι διεργασίες να φτάσουν στο ίδιο σημείο. Ο συγχρονισμός στο MPI μπορεί να επιτευχθεί είτε με την ανταλλαγή μηνυμάτων, χρησιμοποιώντας blocking μεθόδους, είτε με άλλες τεχνικές, όπως με τη χρήση barriers.

Barriers

Οι διεργασίες που ανήκουν σε μία ομάδα μπορούν να συγχρονιστούν σε ένα σημείο χρησιμοποιώντας ένα barrier. Καμία διεργασία δεν μπορεί να προχωρήσειπέραν του σημείου μέχρι ότου όλες οι διεργασίες να φτάσουν στο barrier. Ανάλογα με τον communicator, μπορούμε να συγχρονίσουμε όλες τις διεργασίες μιας ομάδας ή ένα υποσύνολο αυτών. Η συνάρτηση που εκτελεί την παραπάνω λειτουργία ορίζεται ως εξής:

MPI_Barrier(communicator)

Ο συγχρονισμός επιτυγχάνεται αν όλες οι διεργασίες στην ομάδα του communicator καλέσουν τη συνάρτηση αυτή. Μία κλήση στη συνάρτηση αυτή επιστρέφει όταν όλα τα μέλη της ομάδας του communicator έχουν εκτελέσει την δική τους κλήση στην MPI_Barrier().

Collective Operations

Συλλογικές λειτουργίες (collective operations) στο MPI είναι εκείνες που εφαρμόζονται σε όλα τα μέλη μιας ομάδας ενός communicator. Μια συλλογική λειτουργία ορίζεται συνήθως στα πλαίσια μιας ομάδας διεργασιών. Η λειτουργία εκτελείται όταν όλες οι διεργασίες της ομάδας καλέσουν την αντίστοιχη ρουτίνα με τις κατάλληλες παραμέτρους. Υπάρχουν τρεις τύποι συλλογικών λειτουργιών: ελέγχου διεργασιών, καθολικού υπολογισμού και μεταφοράς δεδομένων. Η συνάρτηση MPI_Barrier() που περιγράφηκε προηγουμένως μπορεί να θεωρηθεί ως συλλογική λειτουργία ελέγχου διεργασιών.

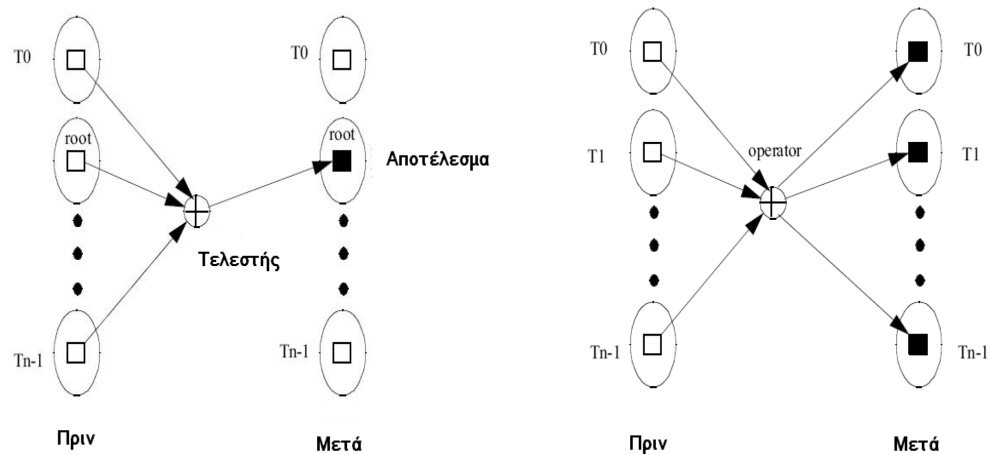
Καθολικός Υπολογισμός (Global Computation)

Στην κατηγορία αυτή ανήκουν λειτουργίες αναγωγής (reduce) και σάρωσης (scan). Μια λειτουργία αναγωγής είναι μια συσχετιστική, τροποποιητική λειτουργία που εφαρμόζεται πάνω σε δεδομένα που παρέχονται από διεργασίες μιας ομάδας. Μια λειτουργία αναγωγής μπορεί να είναι προκαθορισμένη από το MPI, όπως άθροισμα, εύρεση ελάχιστου-μέγιστου κ.ά. ή μια ορισμένη από το χρήστη συνάρτηση. Το αποτέλεσμα της αναγωγής μπορεί να σταλεί σε κάθε διεργασία της ομάδας ή σε μία μόνο η οποία ονομάζεται ρίζα (root).

Το MPI παρέχει την παρακάτω συνάρτηση αναγωγής με την οποία το αποτέλεσμα επιστρέφεται μόνο στη διεργασία-ρίζα:
MPI_Reduce(sbuf, rbuf, n, data_type, op, rt,communicator)

Μια παραλλαγή της παραπάνω συνάρτησης είναι η λειτουργία αναγωγής κατά την οποία το αποτέλεσμα αποστέλλεται σε όλες τις διεργασίες που είναι μέλη της ομάδας. Η συνάρτηση ορίζεται ως εξής:
MPI_Allreduce(sbuf, rbuf, n, data_type, op, communicator)

Τα ορίσματα έχουν την ίδια σημασία όπως στην MPI_Reduce(). Οι δύο παραπάνω λειτουργίες παρουσιάζονται σχηματικά στο Σχήμα 3.1.



Εικόνα 5-13: Αριστερά παρουσιάζεται η λειτουργία Reduce και δεξιά η λειτουργία AllReduce.

Μεταφορά Δεδομένων

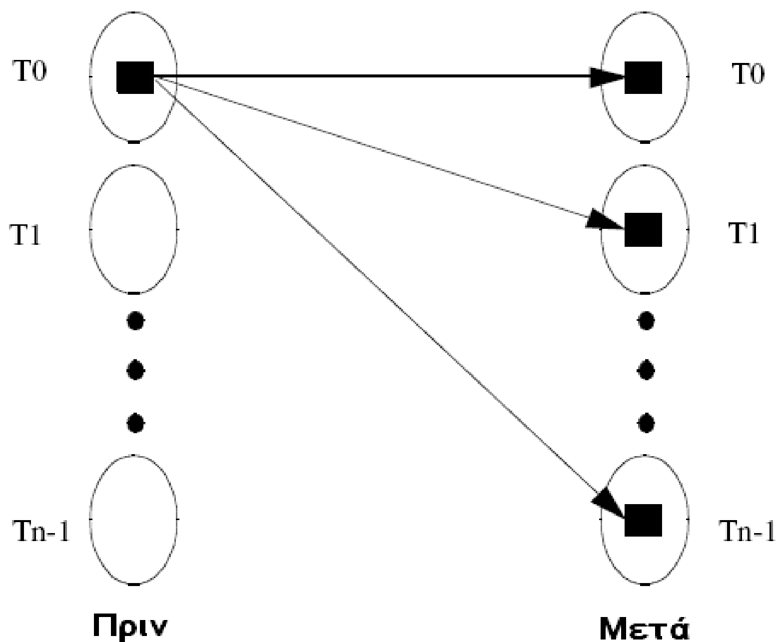
Το MPI υποστηρίζει μεγάλη ποικιλία συμμετρικών συλλογικών συναρτήσεων για τη μεταφορά δεδομένων. Οι βασικές λειτουργίες που καλύπτονται είναι οι *broadcast, scatter* και *gather*. Με τη λειτουργία *broadcast*, μια διεργασία στέλνει το ίδιο μήνυμα σε κάθε μέλος μιας ομάδας. Μια λειτουργία *scatter* επιτρέπει σε μια διεργασία να στείλει διαφορετικό μήνυμα σε κάθε μέλος, ενώ η λειτουργία *gather* είναι η δυική της *scatter*, κατά την οποία μία διεργασία θα λάβει ένα μήνυμα από κάθε μέλος μιας ομάδας διεργασιών.

Broadcast

Πιο συγκεκριμένα, το MPI παρέχει την ακόλουθη συνάρτηση για την αποστολή ενός μηνύματος από τη root-διεργασία σε όλες τις υπόλοιπες στην ομάδα ενός communicator:

MPI_Bcast(buffer, n, data_type, root, communicator)

Η συνάρτηση αυτή πρέπει να κληθεί από όλες τις διεργασίες μιας ομάδας χρησιμοποιώντας τα ίδια ορίσματα για το βαθμό της root-διεργασίας και τον communicator. Τα περιεχόμενα της προσωρινής μνήμης αποστολής της root-διεργασίας θα αντιγραφούν στις μνήμες όλων των υπόλοιπων διεργασιών (βλέπε Σχήμα 3.2).



Εικόνα 5-14: Λειτουργία Broadcast

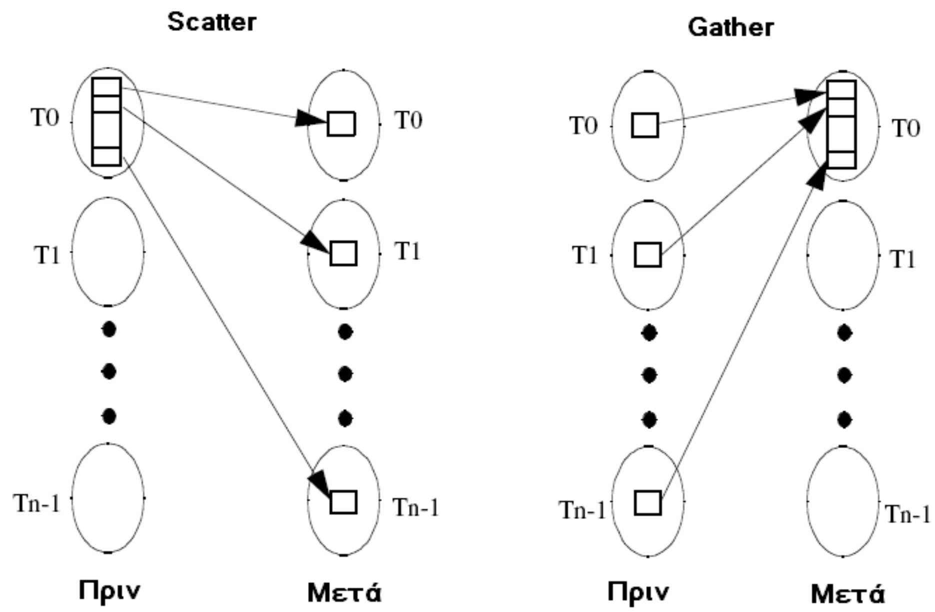
Scatter – Gather

Η συνάρτηση *scatter* επιτρέπει σε μια διεργασία να κατανείμει τα δεδομένα της μνήμης της σε κάθε μέλος μιας ομάδας διεργασιών ενώ η συνάρτηση *gather* επιτρέπει σε μια διεργασία να συλλέξει κομμάτια δεδομένων από τις άλλες διεργασίες για να χτίσει το περιεχόμενο της μνήμης της. Οι δύο συναρτήσεις ορίζονται ως εξής:

MPI_Scatter(sbuf, n, stype, rbuf, m, rtype, rt, communicator)

MPI_Gather(sbuf, n, stype, rbuf, m, rtype, rt, communicator)

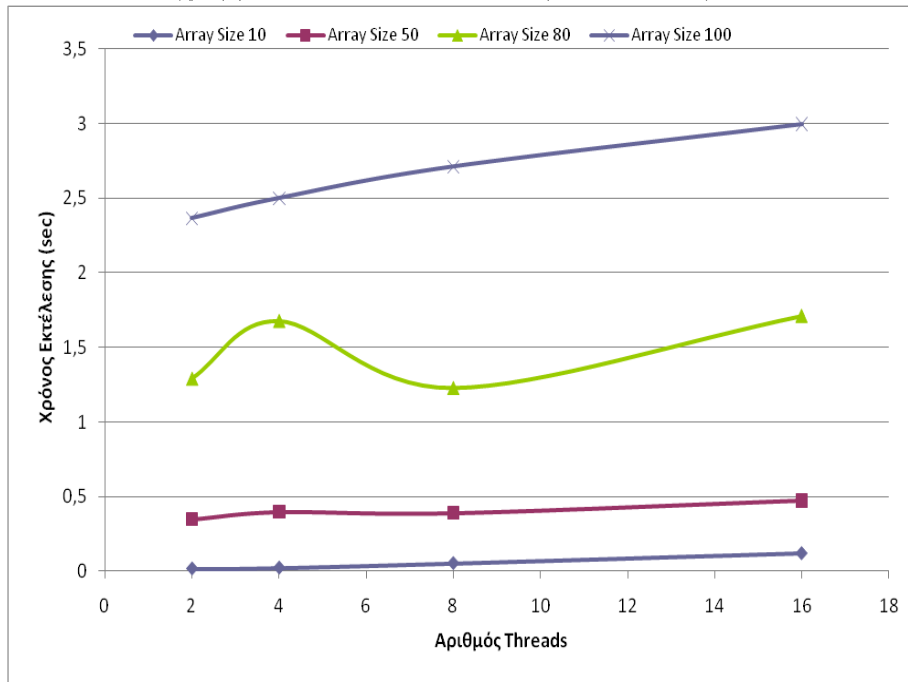
Στη *scatter* η προσωρινή μνήμη αποστολής της root-διεργασίας χωρίζεται σε τμήματα μεγέθους n. Τα πρώτα n στοιχεία στη μνήμη της root-διεργασίας αντιγράφονται στο χώρο αποθήκευσης του πρώτου μέλους της ομάδας, τα επόμενα n στοιχεία στο χώρο του δεύτερου μέλους κλπ. Στη *gather,* κάθε διεργασία (συμπεριλαμβανομένης της root) στέλνει τα περιεχόμενα της προσωρινής της μνήμης στη root-διεργασία. Η root-διεργασία λαμβάνει τα μηνύματα και τα αποθηκεύει στη σειρά με βάση το βαθμό της κάθε διεργασίας. Οι λειτουργίες αυτές παρουσιάζονται στο Σχήμα 3.3.



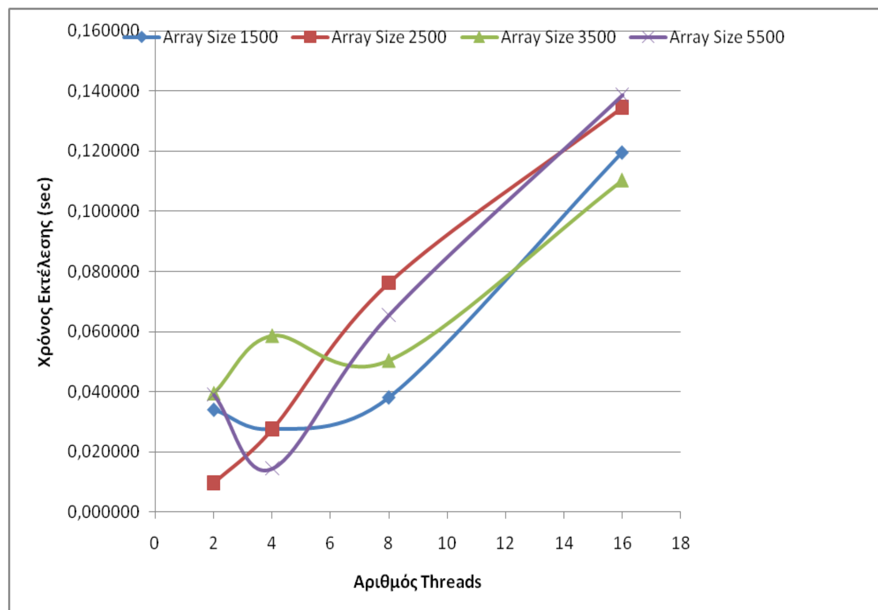
Εικόνα 5-15: Λειτουργίες Scatter και Gather

6. Αποτελέσματα Προγραμμάτων και Συμπεράσματα

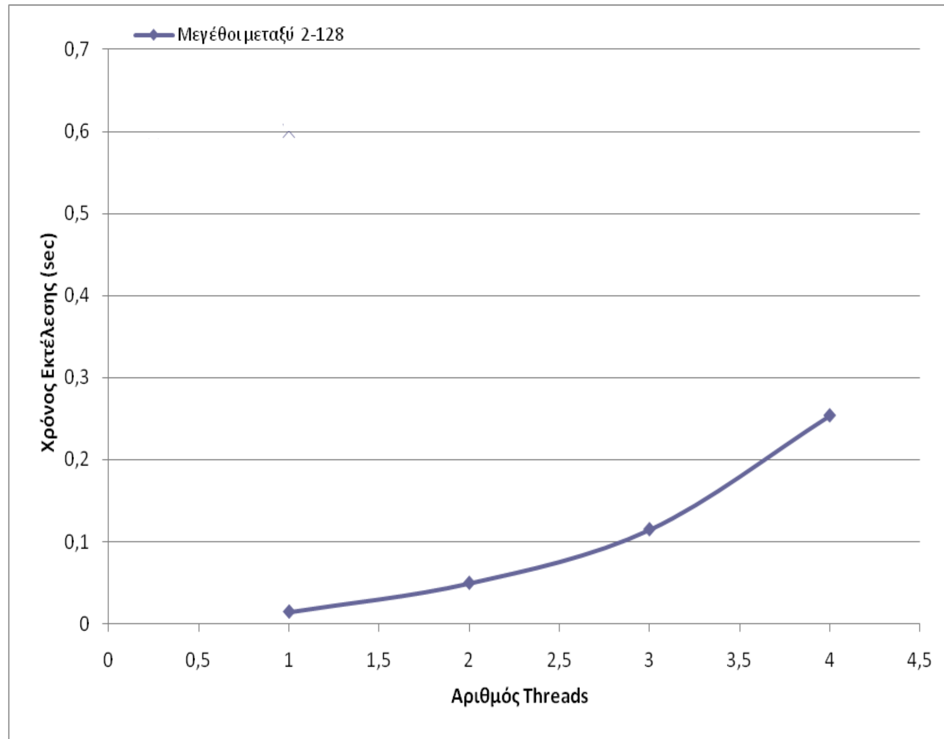
Διαγράμματα Για Πολλαπλασιασμό Πινάκων με Pthreads



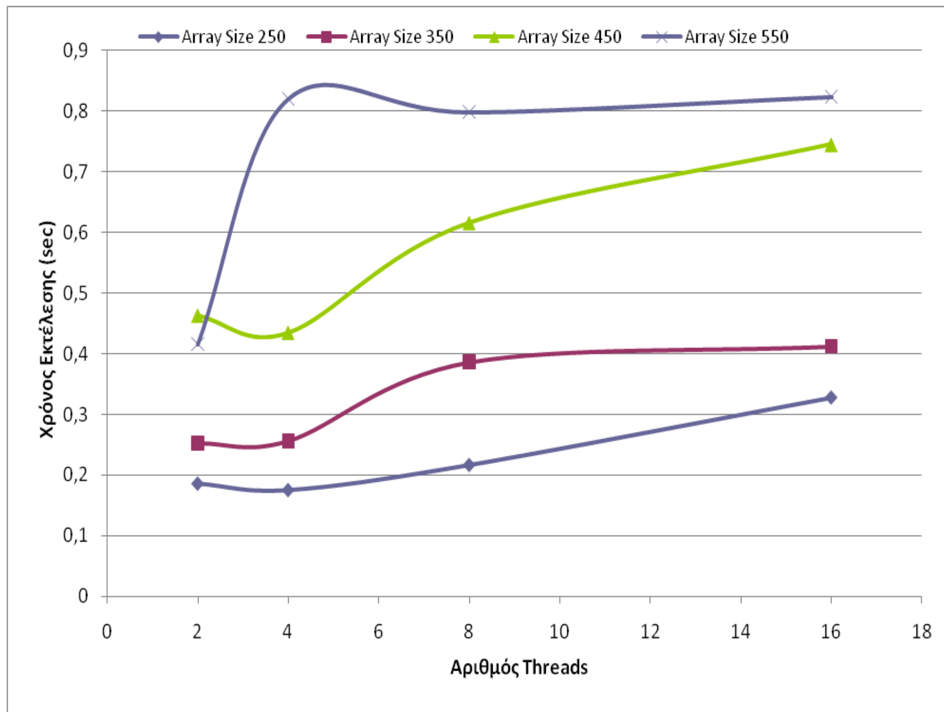
Εικόνα 6-1: Pthread MatrixProduct



Εικόνα 6-2: Pthread DotProduct

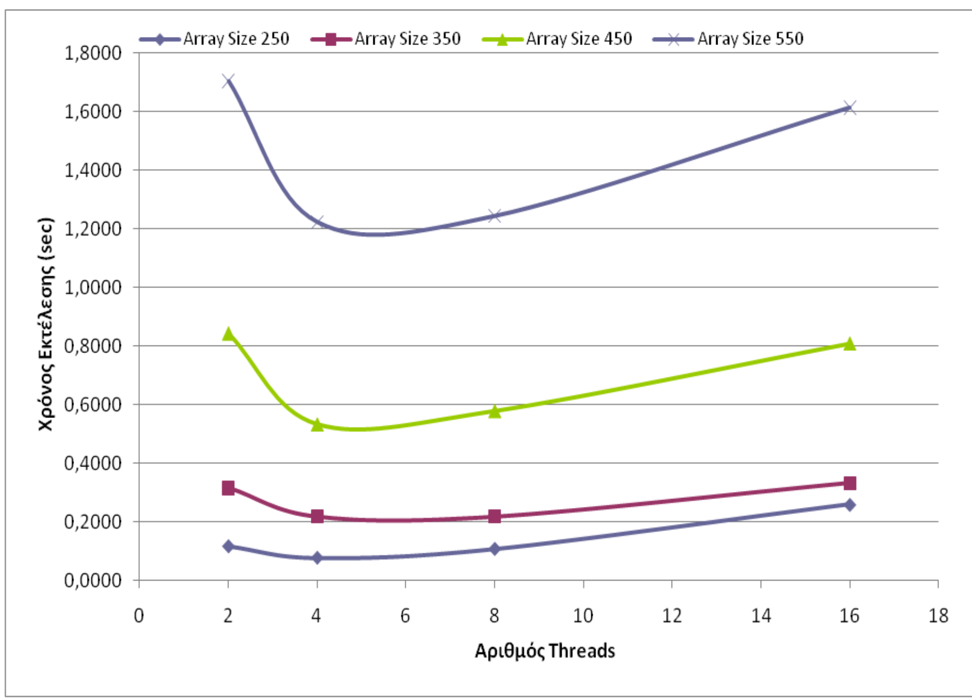


Εικόνα 6-3: Pthread MergeSort

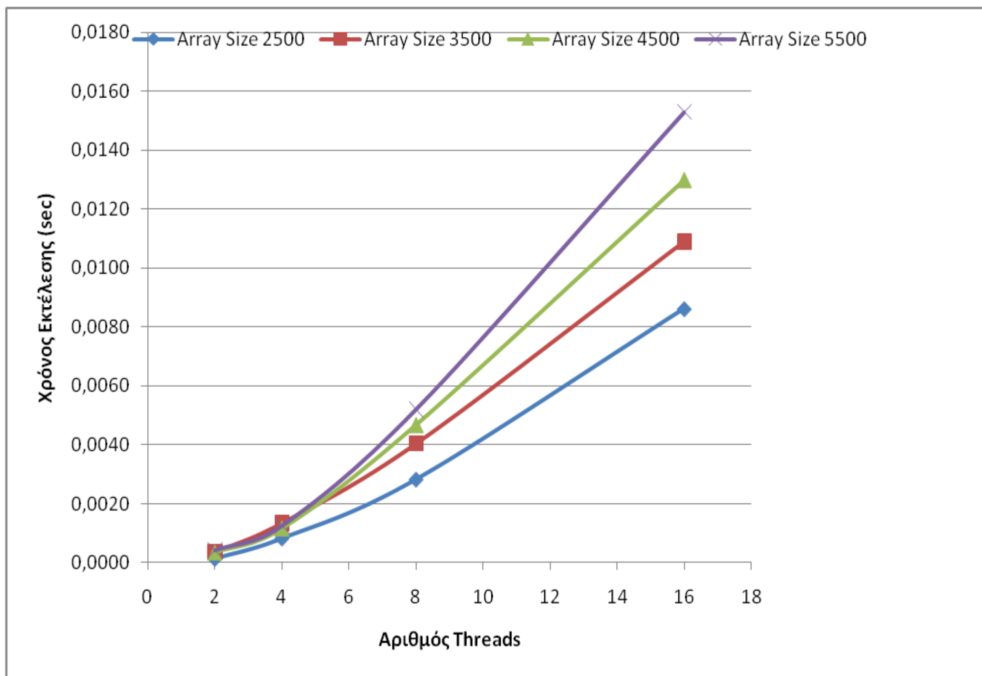


Εικόνα 6-4: Pthread VectorProduct

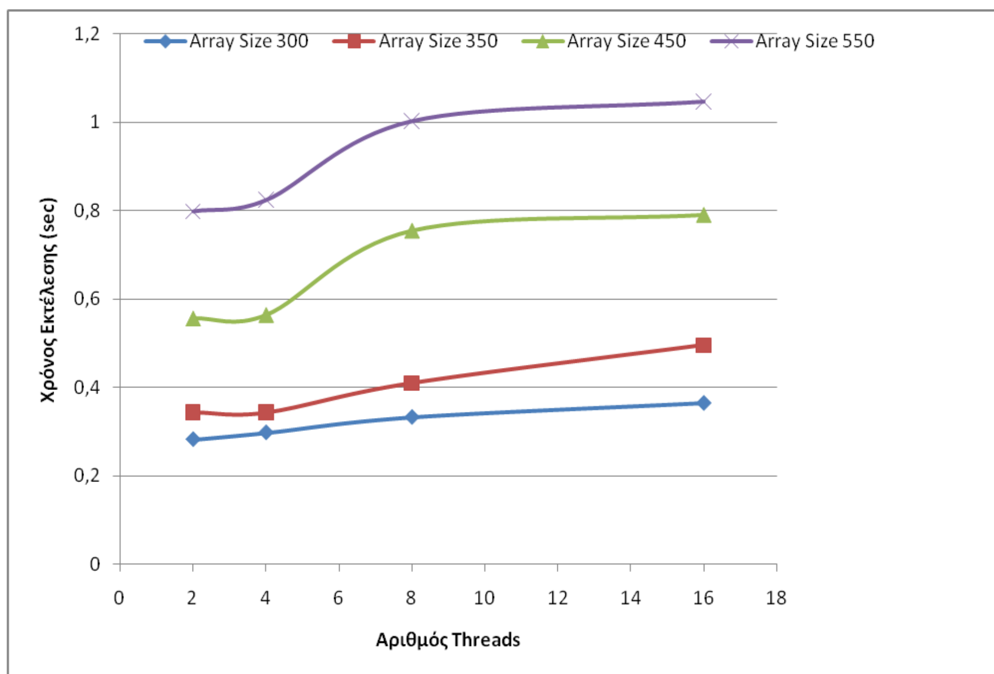
<u>Διαγράμματα Για Πολλαπλασιασμό Πίνακα με Διάνυσμα με MPI Threads</u>



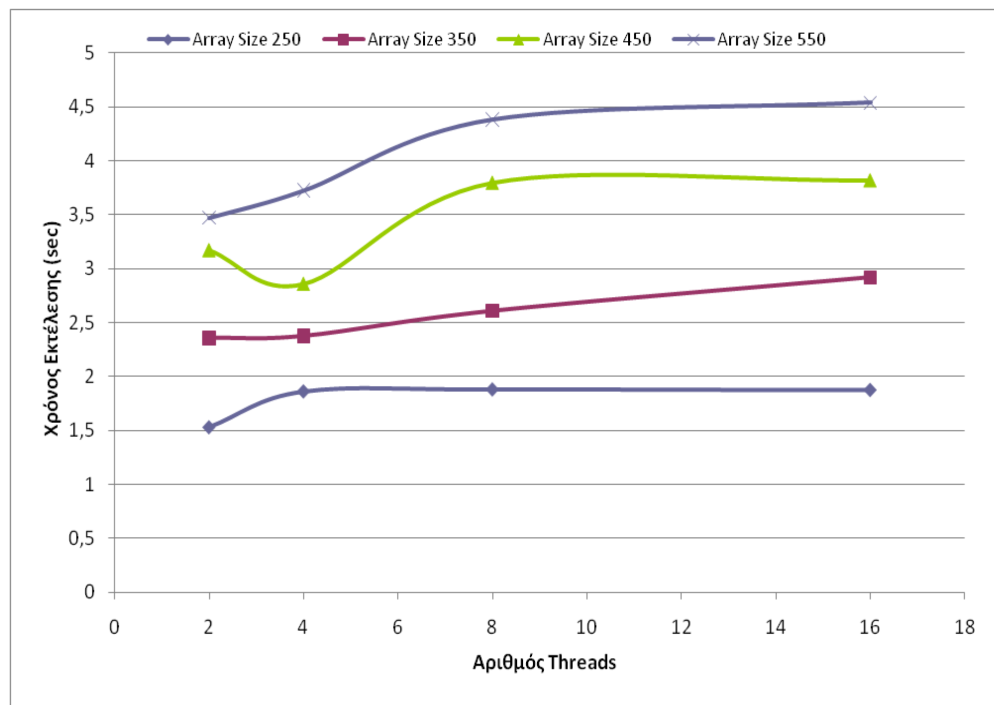
Εικόνα 6-5: MPI MatrixProduct



Εικόνα 6-6: MPI DotProduct

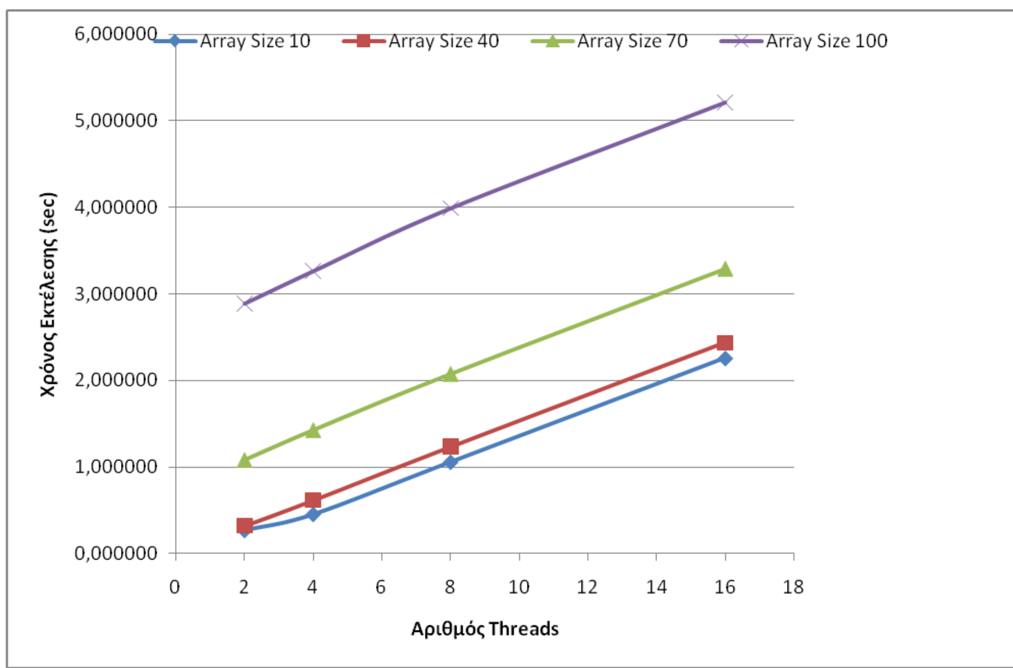


Εικόνα 6-7: MPI MergeSort

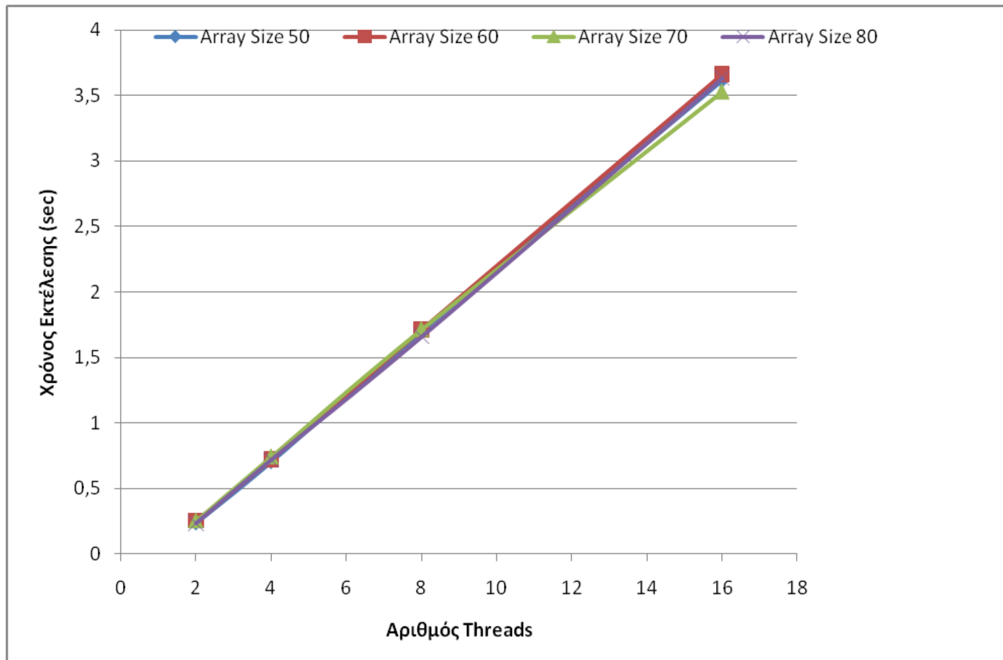


Εικόνα 6-8: MPI MatrixVector

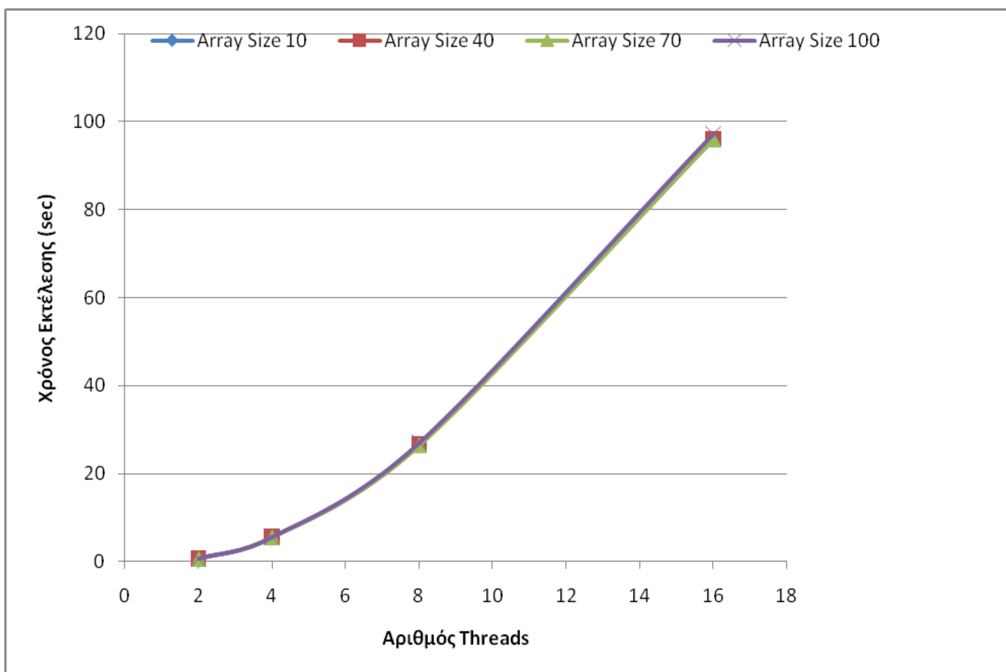
Διαγράμματα Για Πολλαπλασιασμό Πινάκων με Linux Threads



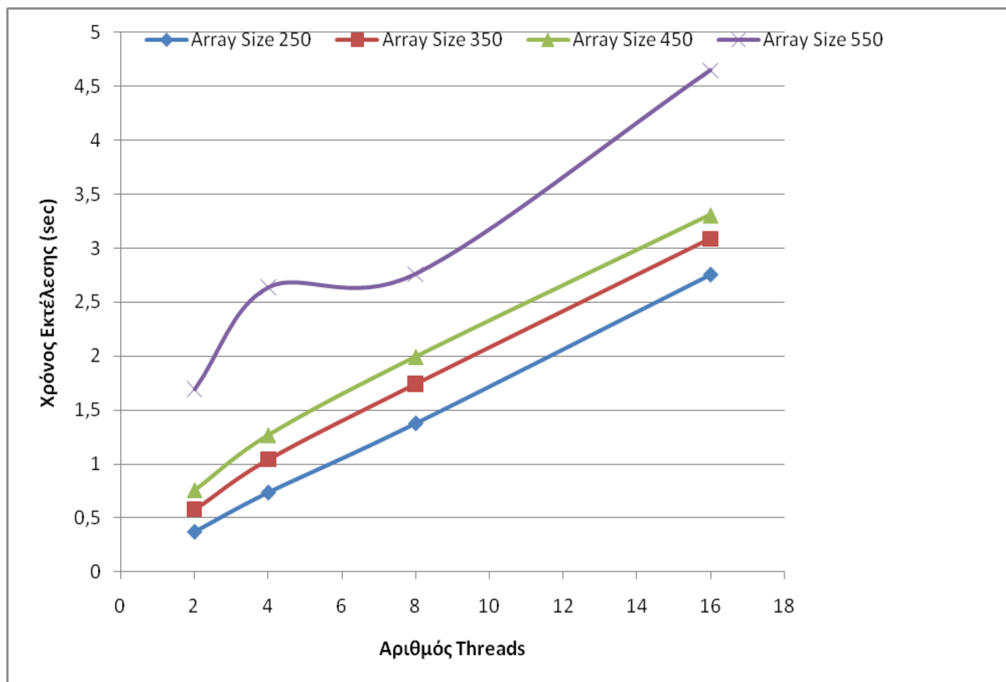
Εικόνα 6-9: Linux MatrixProduct



Εικόνα 6-10: Linux DotProduct



Εικόνα 6-11: Linux MergeSort



Εικόνα 6-12: Linux MatrixVector

Συμπεράσματα

Από τα διαγράμματα παρατηρούμε ότι η πιο γρήγορη βιβλιοθήκη είναι το Pthread, αμέσως μετά το MPI και τέλος Linux Threads. Το πόρισμα αυτό βγαίνει κυρίως παρατηρώντας τους τελικούς χρόνους εκτέλεσης των παραπάνω διαγραμμάτων για τα διάφορα προγραμματάκια (εσωτερικό γινόμενο, πολλαπλασιασμό πινάκων, πολλαπλασιασμό πίνακα με διάνυσμα και mergesort) που τρέξαμε.

Επίσης από τις μετρήσεις των χρόνων των διαγραμμάτων για 2 έως 4 και 4 έως 8 threads, πλήν ορισμένων εξαιρέσεων για ορισμένα μεγέθη στοιχείων πινάκων δεδομένων, παρατηρούμε ότι από 2 ή 4 threads οι καμπύλες αυξάνονται απότομα. Άρα οι βιβλιοθήκες που εξετάσαμε παρέχουν την καλύτερη απόδοση για 2 ως 4 threads, αναλόγως με το πρόβλημα.

Βιβλιογραφία

[1] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[2] S. V. Adve, K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial", *Technical Report*, Digital Western Research Laboratory, 1995.

[3] J.D. Kubiatowicz, "Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor, *PhD Thesis*, MIT, 1998.

[4] A. Grama, A. Gupta, G. Karypis, and Vipin Kumar. "*Introduction to Parallel Computing*", Addison Wesley, 2003.

[5] W. Richard Stevens, Stephen A. Rago. "*Advanced Programming in the UNIX Environment*", Second Edition, 2005. Διαθέσιμο από την ηλεκτρονική διεύθυνση URL:http://book.chinaunix.net/special/ebook/addisonWesley/APUE2/0201433079/toc.html

[6] K.A. Robbins and Steven Robbins. "*Unix Systems Programming: Communication, Concurrency, and Threads*", Prentice Hall, 2003.

[7] Βασιλική Α. Καλαβρή. "Παραλληλοποίηση Αλγόριθμων Γραμμικής Άλγεβρας για Αρχιτεκτονικές Υψηλής Επίδοσης", *Διπλωματική Εργασία*, Ε.Μ.Π., Αθήνα, 2010.

[8] G.V. Wilson, "*Practical Parallel Programming"*, MIT Press, Cambridge, MA, 1995

[9] A. Burns and G. Davies, "*Concurrent Programming*", Addison-Wesley, Reading, Massachusetts, 1994

[10] S. Brawer, "Introduction to Parallel Programming", Academic Press, San Diego, CA, 1989.

[11] R. Stevens, "*Advanced Programming in the Unix Environment*", Addison-Wesley, Reading, Massachusetts, 1992.

[12] Sequent Computer Systems Inc., "*Sequent Guide to Parallel Programming"*, Report No. 1003-44459, 1987

[13] D. R. Butenhof, "*Programming with POSIX Threads*", Addison-Wesley, Reading, Massachusetts, 1997

[14] A. D. Marshall, "*Programming in C: Unix System Calls and Subroutines"*, διαθέσιμο στην ηλεκτρονική διεύθυνση URL: http://docs.linux.cz/programming/c/marshall/CE.html

[15] http://en.wikipedia.org/wiki/Linux - Linux

[16] http://en.wikipedia.org/wiki/MPICH - MPICH

[17] http://en.wikipedia.org/wiki/Threads_%28computer_science%29 - Thread

[18] http://en.wikipedia.org/wiki/Message_passing - Message Passing

[19] https://computing.llnl.gov/tutorials/mpi/ - Message Passing Interface

[20] http://www.hku.hk/cc/sp2/workshop/html/message_passing/message_passing.html#message1 – SP Parallel Programming Workshop Message Passing Overview

[21] https://computing.llnl.gov/tutorials/parallel_comp/#ModelsMessage – Introduction To Parallel Computing

[22] http://www.scribd.com/doc/54101857/10/The-Message-Passing-Model - Message Passing Model

[23] http://en.wikipedia.org/wiki/Shared_memory - Shared Memory

[24] http://www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html#MP%20Definition – Introduction Parallel Programming

[25] http://www.cs.iastate.edu/~prabhu/Tutorial/CACHE/interac.html - Interaction Policies with Main Memory

[26] http://en.wikipedia.org/wiki/Posix - Posix

[27] https://computing.llnl.gov/tutorials/pthreads/ - POSIX Threads Programming

[28] http://en.wikipedia.org/wiki/Pthread - Posix Threads

[29] http://parallelcomp.atw.hu/ch07lev1sec10.html - Introduction to Parallel Computing, Second Edition

Εικόνες

[1-1] https://computing.llnl.gov/tutorials/parallel_comp/#ModelsMessage

[1-2] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[1-3] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[1-4] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[2-1] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[2-2] http://www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html#shared%20memory

[2-4] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[3-1] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[3-2] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[3-3] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[3-4] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[3-5] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[3-6] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[3-7] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[4-1] S. V. Adve, K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial", *Technical Report*, Digital Western Research Laboratory, 1995.

[4-2] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[4-3] D. Culler, B. Jaswinder, P. Singh and A. Gupta. *Parallel Computer Architecture A Hardware / Software Approach*, 1999.

[4-4] S. V. Adve, K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial", *Technical Report*, Digital Western Research Laboratory, 1995.

[5-5] https://computing.llnl.gov/tutorials/pthreads/

[5-6] https://computing.llnl.gov/tutorials/pthreads/

[5-7] https://computing.llnl.gov/tutorials/pthreads/

[5-8] https://computing.llnl.gov/tutorials/pthreads/

[5-9] https://computing.llnl.gov/tutorials/pthreads/

[5-10] W. Richard Stevens, Stephen A. Rago. "*Advanced Programming in the UNIX Environment*", Second Edition, 2005. Διαθέσιμο από την ηλεκτρονική διεύθυνση URL:http://book.chinaunix.net/special/ebook/addisonWesley/APUE2/0201433079/toc.html

[5-11] W. Richard Stevens, Stephen A. Rago. "*Advanced Programming in the UNIX Environment*", Second Edition, 2005. Διαθέσιμο από την ηλεκτρονική διεύθυνση URL:http://book.chinaunix.net/special/ebook/addisonWesley/APUE2/0201433079/toc.html

[5-12] W. Richard Stevens, Stephen A. Rago. "*Advanced Programming in the UNIX Environment*", Second Edition, 2005. Διαθέσιμο από την ηλεκτρονική διεύθυνση URL:http://book.chinaunix.net/special/ebook/addisonWesley/APUE2/0201433079/toc.html

[5-13] Βασιλική Α. Καλαβρή. “Παραλληλοποίηση Αλγόριθμων Γραμμικής Άλγεβρας για Αρχιτεκτονικές Υψηλής Επίδοσης”, *Διπλωματική Εργασία*, Ε.Μ.Π., Αθήνα, 2010.
[5-14] Βασιλική Α. Καλαβρή. “Παραλληλοποίηση Αλγόριθμων Γραμμικής Άλγεβρας για Αρχιτεκτονικές Υψηλής Επίδοσης”, *Διπλωματική Εργασία*, Ε.Μ.Π., Αθήνα, 2010.
[5-15] Βασιλική Α. Καλαβρή. “Παραλληλοποίηση Αλγόριθμων Γραμμικής Άλγεβρας για Αρχιτεκτονικές Υψηλής Επίδοσης”, *Διπλωματική Εργασία*, Ε.Μ.Π., Αθήνα, 2010.