ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ: ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΣΥΣΤΗΜΑΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών της Πολυτεχνικής Σχολής του Πανεπιστημίου Πατρών

Δημητρακόπουλου Γεώργιου του Νικολάου

Αριθμός Μητρώου: 5953

Θέμα

«Ανάπτυξη ενσωματωμένων συστημάτων σε πολυπύρηνο ή πολυεπεξεργαστικό περιβάλλον με χρήση Real Time Java»

Επιβλέπων
Κλεάνθης Θραμπουλίδης

Αριθμός Διπλωματικής Εργασίας:

Πάτρα, ΙΟΥΛΙΟΣ 2010

ΠΙΣΤΟΠΟΙΗΣΗ

Πιστοποιείται ότι η Διπλωματική Εργασία με θέμα

«Ανάπτυξη ενσωματωμένων συστημάτων σε πολυπύρηνο ή πολυεπεξεργαστικό περιβάλλον με χρήση Real Time Java»

Του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών

Δημητρακόπουλου Γεώργιου του Νικολάου

Αριθμός Μητρώου: 5953

Παρουσιάστηκε δημόσια και εξετάστηκε στο Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών στις
6/7/2010

Ο Επιβλέπων	Ο Διευθυντής του Τομέα
Κ. Θραμπουλίδης Αναπληρωτής Καθηγητής	Ε. Χούσος Καθηγητής

Αριθμός Διπλωματικής Εργασίας:

Θέμα: «Ανάπτυξη ενσωματωμένων συστημάτων σε πολυπύρηνο ή πολυεπεξεργαστικό περιβάλλον με χρήση Real Time Java»

Φοιτητής:
Δημητρακόπουλος Γεώργιος

Επιβλέπων:
Κλεάνθης Θραμπουλίδης

Περίληψη

Ο σκοπός της παρούσας διπλωματικής εργασίας είναι να μελετηθεί με ποιον τρόπο μια εφαρμογή μπορεί να αξιοποιήσει την παρουσία πολλών επεξεργαστών σε ένα σύστημα.
Το σύστημα προς μελέτη είναι το Festo MPS, το οποίο είναι ένα κατανεμημένο ενσωματωμένο σύστημα πραγματικού χρόνου, αποτελούμενο από τρεις υπομονάδες. Το σύστημα έχει υλοποιηθεί σε Real Time Java, μια επέκταση της Java, η οποία ανταποκρίνεται σε απαιτήσεις πραγματικού χρόνου. Η εφαρμογή εκτελείται σε μια Java Virtual Machine πραγματικού χρόνου, η οποία με τη σειρά της εκτελείται σε ένα λειτουργικό σύστημα τύπου Linux.
Το κάθε επίπεδο έχει διάφορους μηχανισμούς έτσι ώστε να αξιοποιεί τους διαθέσιμους επεξεργαστές. Ερευνούνται τρόποι με τους οποίους ο προγραμματιστής μπορεί να διευκολυνθεί στο έργο του, γράφοντας αποδοτικότερο, μικρότερο και καθαρότερο παράλληλο κώδικα, καθώς και οι επιλογές, ώστε να καθορίζει ο ίδιος επακριβώς τον τρόπο εκτέλεσης, όταν αυτό απαιτείται.
Τελικός στόχος είναι να εκτελεστεί μια προσομοίωση του συστήματος, όπου κάθε υπομονάδα θα εκτελείται σε διαφορετικό επεξεργαστή. Αυτό επιτυγχάνεται με τη βοήθεια των κλήσεων του λειτουργικού συστήματος μέσω Java Native Interface.

Πρόλογος

Σε αυτή τη διπλωματική εργασία μελετάται με ποιους τρόπους μπορεί μια εφαρμογή να αξιοποιήσει την παρουσία πολλών επεξεργαστών. Ως παράδειγμα εφαρμογής έχει επιλεχθεί το Festo MPS, το οποίο είναι ένα κατανεμημένο ενσωματωμένο σύστημα πραγματικού χρόνου. Αναλύονται οι μηχανισμοί του λειτουργικού συστήματος, της Java και της Real Time Java τους οποίους μπορεί να χρησιμοποιήσει ο προγραμματιστής ώστε να αξιοποιήσει τον παραλληλισμό.

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον κ. Κλεάνθη Θραμπουλίδη για την πολύτιμη καθοδήγηση και την άψογη συνεργασία του κατά τη διάρκεια της εκπόνησης της διπλωματικής μου εργασίας.

Πίνακας περιεχομένων

Κεφάλαιο 1
Εισαγωγή
Εισαγωγή.. 11
Κεφάλαιο 2
Ανάπτυξη ενσωματωμένων συστημάτων σε πολυεπεξεργαστικό ή πολυπύρηνο περιβάλλον
2.1 Ενσωματωμένα συστήματα 13
2.2 Πολυεπεξεργαστές / πολυπύρηνοι επεξεργαστές 15
2.2.1 Γενικά 15
2.2.2 Πλεονεκτήματα πολυπύρηνων επεξεργαστών.......................... 16
2.2.3 Αξιοποίηση πολυπύρηνων επεξεργαστών 17
Κεφάλαιο 3
Υποστήριξη παραλληλισμού από το Λειτουργικό Σύστημα
3.1 Χρονοπρογραμματισμός - Scheduling 20
3.1.1 Βασικές έννοιες 20
3.1.2 Χρονοπρογραμματισμός στο λειτουργικό σύστημα Windows. 22
3.1.2 Χρονοπρογραμματισμός στο λειτουργικό σύστημα Linux 23
3.2 Συνάφεια επεξεργαστή (CPU affinity) 26
3.2.1 Ορισμός συνάφειας σε λειτουργικό σύστημα Windows 27
3.2.2 Ορισμός συνάφειας σε λειτουργικό σύστημα Linux 28
Κεφάλαιο 4
Java και παραλληλισμός
4.1 Εικονική μηχανή Java και μοντέλο νημάτων 30
4.2 Υποστήριξη παραλληλισμού από την βασική βιβλιοθήκη της Java 33
4.3 JOMP 35
4.3.1 Εισαγωγή 35
4.3.2 Η προγραμματιστική διεπαφή 36
4.3.3 Απλό παράδειγμα εφαρμογής 38
Κεφάλαιο 5
Real Time Java
5.1 Απαιτήσεις Πραγματικού Χρόνου 41
5.2 Προβλήματα της Java στις απαιτήσεις πραγματικού χρόνου..... 42
5.3 Το RTSJ.. 43
5.3.1 Κατευθυντήριες αρχές - Απαιτήσεις συμβατότητας 44
5.3.2 Προσθήκες του RTSJ σε σχέση με την κλασσική Java ... 45
5.4 Υλοποιήσεις του προτύπου RTSJ 48
5.5 Απλά παραδείγματα εφαρμογών 49
Κεφάλαιο 6
Case Study
6.1 Το σύστημα Festo MPS 55
6.2 Function Blocks 56

6.3 Αξιοποίηση παραλληλισμού από την εφαρμογή ελέγχου του συστήματος. 57
6.4 Ρητή διαχείριση νημάτων 60
Συμπεράσματα 65
Αναφορές 66

Κεφάλαιο 1

Εισαγωγή

Ο σκοπός αυτής της διπλωματικής εργασίας είναι να μελετηθεί με ποιον τρόπο μια εφαρμογή μπορεί να αξιοποιήσει την παρουσία πολλών επεξεργαστών σε ένα σύστημα. Σήμερα η τάση στον τομέα των επεξεργαστών είναι η παραγωγή πολυπύρηνων επεξεργαστών. Επίσης πολλά ενσωματωμένα συστήματα είναι κατανεμημένα, δηλαδή δεν υπάρχει κεντρικός έλεγχος, αλλά κάθε υποσύστημα είναι αυτόνομο και έχει το δικό επεξεργαστή.

Το σύστημα προς μελέτη είναι το Festo MPS, το οποίο είναι ένα κατανεμημένο ενσωματωμένο σύστημα πραγματικού χρόνου που αποτελείται από τρεις υπομονάδες. Το σύστημα έχει υλοποιηθεί σε Real Time Java, μια επέκταση της Java, η οποία ανταποκρίνεται σε απαιτήσεις πραγματικού χρόνου. Η εφαρμογή εκτελείται σε μια Java Virtual Machine πραγματικού χρόνου, η οποία με τη σειρά της εκτελείται σε ένα λειτουργικό σύστημα τύπου Linux.

Το κάθε επίπεδο έχει διάφορους μηχανισμούς έτσι ώστε να αξιοποιεί τους διαθέσιμους επεξεργαστές, οι οποίοι μηχανισμοί αναλύονται σε ξεχωριστό κεφάλαιο ανά επίπεδο. Ερευνούνται τρόποι με τους οποίους ο προγραμματιστής να μπορεί διευκολυνθεί στο έργο του, γράφοντας αποδοτικότερο, μικρότερο και καθαρότερο κώδικα, καθώς και τρόποι, ώστε να καθορίζει ο ίδιος τον τρόπο εκτέλεσης επακριβώς, όταν αυτό απαιτείται.

Τελικός στόχος είναι να εκτελεστεί μια προσομοίωση του συστήματος, όπου κάθε υπομονάδα θα εκτελείται σε διαφορετικό επεξεργαστή. Αυτό επιτυγχάνεται με κώδικα σε Java, ο οποίος όμως χρησιμοποιεί κλήσεις του λειτουργικού συστήματος μέσω Java Native Interface.

Στο κεφάλαιο 2 δίνονται μερικές βασικές έννοιες σχετικά με τα ενσωματωμένα συστήματα. Επίσης παρουσιάζονται οι πολυπύρηνοι επεξεργαστές και αναφέρονται τα πλεονεκτήματά τους και οι λόγοι που οδήγησαν στην καθιέρωσή τους. Στο τέλος αναφέρεται ο νόμος του Amdahl, ο οποίος δείχνει τον περιορισμό στην απόδοση, λόγω της ύπαρξης λειτουργιών που δεν παραλληλοποιούνται.

Στο κεφάλαιο 3 αναλύεται η λειτουργία κάποιων χρονοπρογραμματιστών λειτουργικών συστημάτων. Επίσης παρουσιάζεται η συνάφεια επεξεργαστή και ο τρόπος με τον οποίο μπορεί ο χρήστης να την τροποποιήσει μέσω κάποιων εντολών και κλήσεων συστήματος.

Στο κεφάλαιο 4 παρουσιάζονται τα μοντέλα σύμφωνα με τα οποία τα νήματα της Java αντιστοιχίζονται σε νήματα του λειτουργικού συστήματος. Μετά παρουσιάζονται οι δυνατότητες που παρέχει η βασική βιβλιοθήκη της Java στον προγραμματιστή, ώστε να διευκολυνθεί ο ταυτόχρονος προγραμματισμός. Στο τέλος παρουσιάζεται το πρότυπο Java OpenMP, το οποίο αυτοματοποιεί τη δημιουργία και τη διαχείριση νημάτων.

Στο κεφάλαιο 5 περιγράφονται οι απαιτήσεις πραγματικού χρόνου, οι λόγοι που η Java αδυνατεί να ανταποκριθεί σε αυτές και το πρότυπο RTSJ, το οποίο παρέχει συγκεκριμένες λύσεις στα παραπάνω προβλήματα.

Στο κεφάλαιο 6 μελετάται το σύστημα Festo MPS. Παρουσιάζεται το πρότυπο των Function Block, σύμφωνα με το οποίο υλοποιείται το σύστημα σε Real Time Java. Μετά αναλύονται ποιοι μηχανισμοί χρησιμοποιούνται για να εκμεταλλευτεί το σύστημα την ύπαρξη πολλών επεξεργαστών. Τέλος πραγματοποιείται μια προσομοίωση του συστήματος, εκτελώντας κάθε υπομονάδα σε έναν διαφορετικό επεξεργαστή. Αυτό επιτυγχάνεται αξιοποιώντας τις κλήσεις συστήματος, οι οποίες παρουσιάστηκαν στο κεφάλαιο 3 μέσω Java Native Interface.

Στο τέλος ακολουθεί ένα κεφάλαιο που περιγράφει συνοπτικά τα συμπεράσματα της εργασίας.

Κεφάλαιο 2

Ανάπτυξη ενσωματωμένων συστημάτων σε πολυεπεξεργαστικό ή πολυπύρηνο περιβάλλον

Σε αυτό το κεφάλαιο παρουσιάζονται αρχικά κάποιες βασικές έννοιες σχετικά με τα ενσωματωμένα συστήματα. Μετά αναφέρεται τι είναι οι πολυεπεξεργαστές και οι πολυπύρηνοι επεξεργαστές και τα πλεονεκτήματα της παράλληλης επεξεργασίας. Για την αξιοποίηση αυτών των επεξεργαστών απαιτείται και οι εφαρμογές να ακολουθούν το μοντέλο του παράλληλου προγραμματισμού, δηλαδή να είναι πολυνηματικές. Επίσης υπάρχει πάντα κάποιος περιορισμός στην απόδοση των παράλληλων επεξεργαστών, όπως δείχνει ο νόμος Amdahl επειδή δεν γίνεται να εκτελεστούν όλες οι λειτουργίες ταυτόχρονα.

2.1 Ενσωματωμένα συστήματα

Ένα ενσωματωμένο σύστημα είναι ένα υπολογιστικό σύστημα σχεδιασμένο να εκτελεί κάποιες συγκεκριμένες λειτουργίες, συχνά με απαιτήσεις πραγματικού χρόνου. Τα ενσωματωμένα συστήματα ελέγχονται από έναν ή περισσότερους μικροεπεξεργαστές ή επεξεργαστές ψηφιακού σήματος και συνήθως ενσωματώνονται ηλεκτρονικά και μηχανικά μέρη σε μια συσκευή. Το κύριο χαρακτηριστικό τους είναι ότι χειρίζονται αποκλειστικά μια συγκεκριμένη εργασία, σε αντίθεση με τους υπολογιστές γενικού σκοπού, όπως οι προσωπικοί υπολογιστές, που ικανοποιούν μια μεγάλη ποικιλία αναγκών του χρήστη.[1]

Πολύ συχνά αποτελούν μέρος ενός μεγαλύτερου συστήματος, όπως για παράδειγμα σε ένα αυτοκίνητο τα υποσυστήματα ABS, ελέγχου κατανάλωσης κλπ ή σε ένα υπολογιστή το πληκτρολόγιο, το ποντίκι, το μόντεμ και άλλα περιφερειακά. Ουσιαστικά σχεδόν όλες οι ηλεκτρονικές συσκευές που κατασκευάζονται σήμερα είναι ενσωματωμένα συστήματα, από ψηφιακά ρολόγια μέχρι συστήματα ελέγχου εργοστασίων. Υπολογίζεται ότι το 2008 κατασκευάστηκαν περίπου 10 δις. επεξεργαστές και το 98% χρησιμοποιήθηκε για ενσωματωμένα συστήματα.

Οι επεξεργαστές των ενσωματωμένων συστημάτων και οι αρχιτεκτονικές ποικίλουν, από 4-bit επεξεργαστές σε 128-bit εξειδικευμένους επεξεργαστές ψηφιακών σημάτων. Οι συσκευές αυτές μπορεί να τρέχουν από ένα μικρό πρόγραμμα σε γλώσσα μηχανής από μια ROM χωρίς λειτουργικό σύστημα (γνωστό ως firmware) έως και ένα λειτουργικό σύστημα πραγματικού χρόνου που εκτελεί πολυνηματικές εφαρμογές σε γλώσσες υψηλού επιπέδου ή ακόμα και διάφορες εκδόσεις των Windows ή Linux. Γενικά υπάρχει μια μεγάλη ποικιλία και κάποια δυσκολία στον αυστηρό ορισμό των ενσωματωμένων συστημάτων, καθώς υπάρχουν

συστήματα αυξημένης πολυπλοκότητας που περιλαμβάνουν πολλαπλές μονάδες, περιφερειακά ή και δίκτυα . [2]

Επειδή τα ενσωματωμένα συστήματα είναι ειδικού σκοπού και απαιτούν λίγους υπολογιστικούς πόρους, βελτιστοποιούνται ώστε να μειωθεί το μέγεθος και το κόστος και να αυξηθεί η αξιοπιστία τους και η απόδοση. Η διεπαφή με χρήστη ποικίλει: πολλές φορές δεν είναι καθόλου απαραίτητη, άλλοτε περιορίζεται σε μερικά κουμπιά, λυχνίες LED και LCD οθόνες ή μερικές φορές έχουν γραφική διεπαφή που θυμίζει προσωπικούς υπολογιστές. Επίσης μπορεί να χρειάζεται επικοινωνία με άλλα συστήματα, όπως για απομακρυσμένο έλεγχο, επομένως χρησιμοποιούνται διασυνδέσεις, όπως σειριακή (RS-232, USB) ή δικτύου (Ethernet).

Ο σχεδιασμός και η υλοποίηση μιας εφαρμογής για ένα ενσωματωμένο σύστημα είναι αρκετά διαφορετικός και υπάρχουν περισσότερες προκλήσεις σε σχέση με τον τυπικό προγραμματισμό για προσωπικούς υπολογιστές. Ο κώδικας πρέπει όχι μόνο να παρέχει την προβλεπόμενη λειτουργικότητα, αλλά και να εκτελείται έτσι ώστε να προλαβαίνει κάποιες προθεσμίες, να χωρά στην διαθέσιμη μνήμη και να καλύπτει τους περιορισμούς στην κατανάλωση ενέργειας. Επίσης μερικά συστήματα αναμένεται να λειτουργούν συνεχώς για μεγάλο χρονικό διάστημα χωρίς σφάλματα ή να ξεπερνούν τα σφάλματα από μόνα τους.

Οι γλώσσες υψηλού επιπέδου προγραμματισμού θεωρούνταν κάποτε ανεπαρκείς για αυτές τις ανάγκες, αλλά οι καλύτεροι μεταφραστές, ο ταχύτεροι επεξεργαστές και μνήμες και οι βελτιωμένες αρχιτεκτονικές έχουν επιτρέψει τη χρήση τέτοιων γλωσσών. Τώρα στην ανάπτυξη ενσωματωμένων συστημάτων οι πιο καθιερωμένες γλώσσες είναι οι C, C++ και Java. Αρχικά χρησιμοποιούταν γλώσσα μηχανής για λόγους απόδοσης, αλλά σήμερα χρησιμοποιείται πολύ σπάνια μόνο όταν υπάρχει κάποια πολύ κρίσιμη εργασία που δεν μπορεί να επιτευχθεί αλλιώς. [1]

Στην εργασία αυτή θα μελετηθεί ένα ενσωματωμένο σύστημα το οποίο έχει αναπτυχθεί σε Real Time Java και εκτελείται σε πολυπύρηνο επεξεργαστή ή σε πολυεπεξεργαστή. Αφού το σύστημα είναι υλοποιημένο σε Real Time Java, εκτελείται σε μια εικονική μηχανή Java πραγματικού χρόνου, η οποία με τη σειρά της εκτελείται σε ένα λειτουργικό σύστημα Linux. Επομένως θα αναλυθεί ο τρόπος με τον οποίο αξιοποιείται η παρουσία πολλών επεξεργαστών από όλα τα επίπεδα του συστήματος, όπως φαίνεται στο Σχήμα 1.

Εφαρμογή Real Time Java
Real Time Java Extensions
Java Virtual Machine
Λειτουργικό Σύστημα
Υλικό

Σχήμα 1.

2.2 Πολυεπεξεργαστές / πολυπύρηνοι επεξεργαστές

2.2.1 Γενικά

Οι επεξεργαστές αναπτύχθηκαν αρχικά με ένα μόνο πυρήνα. Ο πυρήνας είναι το μέρος του επεξεργαστή που εκτελεί πραγματικά την ανάγνωση και την εκτέλεση μιας εντολής. Οι επεξεργαστές ενός πυρήνα μπορούν να επεξεργαστούν μόνο μία εντολή κάθε φορά. Για να βελτιωθεί η αποδοτικότητα, οι επεξεργαστές χρησιμοποιούν σε εσωτερικό επίπεδο pipelines (παραλληλισμός στα διάφορα στάδια μιας εντολής), που επιτρέπουν σε μερικές εντολές να επεξεργάζονται μαζί, ωστόσο εξακολουθούν να εισέρχονται στο pipeline μια κάθε φορά.

Ένα πολυεπεξεργαστικό σύστημα αποτελείται από πολλούς ανεξάρτητους επεξεργαστές που εργάζονται μαζί. Ένας πολυπύρηνος επεξεργαστής είναι ένα σύστημα επεξεργασίας που αποτελείται από δύο ή περισσότερους επεξεργαστές στο ίδιο ολοκληρωμένο κύκλωμα.

Ένας πολυπύρηνος επεξεργαστής σχεδιάζεται συνήθως ανά ζεύγη. Για παράδειγμα, οι πυρήνες μπορεί να μοιράζονται κάποιες κρυφές μνήμες, και επικοινωνούν με μεθόδους περάσματος μηνυμάτων ή με κοινή μνήμη. Οι πολυπύρηνοι επεξεργαστές χρησιμοποιούνται ευρέως σε πολλά πεδία εφαρμογών, συμπεριλαμβανομένων των επεξεργαστών γενικής χρήσης, δικτύων, ενσωματωμένων συστημάτων, ψηφιακής επεξεργασίας σήματος (DSP) και γραφικών. Οι επεξεργαστές μπορούν να διασυνδέονται χρησιμοποιώντας δίαυλο, δακτύλιο, βρόχο δύο διαστάσεων ή μεταγωγείς (crossbars switches).

Εάν όλοι οι επεξεργαστές είναι ακριβώς ίδιοι, συνδέονται σε μια κοινή μνήμη και ελέγχονται από ένα στιγμιότυπο του λειτουργικού συστήματος, τότε έχουμε συμμετρική πολυεπεξεργασία (Symmetric Multi-Processing - SMP). Η συγκεκριμένη αρχιτεκτονική ονομάζεται και ομοιόμορφης πρόσβασης στη μνήμη (Uniform Memory Access – UMA), επειδή η κεντρική μνήμη έχει συμμετρική σχέση με όλους τους επεξεργαστές και ομοιόμορφο χρόνο προσπέλασης. Στην περίπτωση ενός πολυπύρηνου επεξεργαστή η αρχιτεκτονική SMP ισχύει για τους πυρήνες, οι οποίοι αντιμετωπίζονται σαν ξεχωριστοί επεξεργαστές. Αυτή τη στιγμή αυτή είναι η πιο δημοφιλής αρχιτεκτονική στην αγορά επεξεργαστών και η ισχύουσα τάση είναι να αυξάνεται ο αριθμός των πυρήνων (διπύρηνοι, τετραπύρηνοι κλπ). Η δεύτερη κατηγορία είναι η αρχιτεκτονική της μη ομοιόμορφης πρόσβασης στη μνήμη (Non Uniform Memory Access – NUMA). Όταν ο αριθμός των πυρήνων είναι αρκετά μεγάλος (το όριο αυτό είναι αρκετές δεκάδες πυρήνες), η μνήμη κατανέμεται στους επεξεργαστές και συνήθως απαιτείται ένα δίκτυο στο ολοκληρωμένο κύκλωμα. [3]

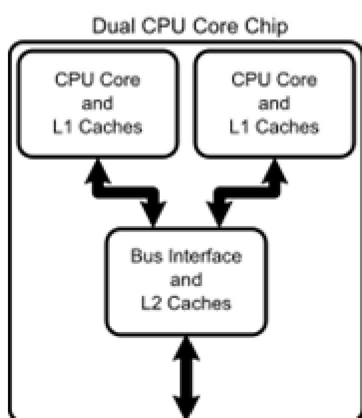
Επίσης η Intel έχει αναπτύξει την τεχνολογία Hyper-Threading, με την οποία σε κάθε φυσικό πυρήνα του επεξεργαστή αναθέτει δύο λογικούς επεξεργαστές. Η Intel εισήγαγε αυτή την τεχνολογία το 2002 και οι επεξεργαστές που την υποστηρίζουν είναι οι Atom, Core i3, Core i5, Core i7, Itanium, Pentium 4 και Xeon. Η αρχή λειτουργίας είναι η εξής: μερικά τμήματα του επεξεργαστή, αυτά που αποθηκεύουν την αρχιτεκτονική κατάσταση, υπάρχουν σε δύο αντίγραφα, ενώ οι βασικές μονάδες εκτέλεσης παραμένουν απλές. Έτσι όταν ένα νήμα δεν χρησιμοποιεί τις μονάδες εκτέλεσης επειδή βρίσκεται σε αναμονή για κάποιο λόγο (για παράδειγμα αστοχία μνήμης ή κακή πρόβλεψη διακλάδωσης), τότε ο επεξεργαστής μπορεί να εκτελέσει ένα δεύτερο νήμα. Το Hyper-Threading απαιτεί το λειτουργικό σύστημα να υποστηρίζει πολλούς επεξεργαστές, αλλά και να είναι βελτιστοποιημένο ειδικά για αυτή την τεχνολογία. Για παράδειγμα, όταν πρέπει να προγραμματιστούν δύο διεργασίες και όλοι οι επεξεργαστές είναι ελεύθεροι, αν ανατεθούν στους δύο λογικούς επεξεργαστές που βρίσκονται στον ίδιο φυσικό επεξεργαστή, θα έχουμε μειωμένη απόδοση. [4]

2.2.2 Πλεονεκτήματα πολυπύρηνων επεξεργαστών

Ο πιο σημαντικός λόγος καθιέρωσης των πολυπύρηνων επεξεργαστών είναι η βελτιωμένη απόδοση. Από τον πρώτο εμπορικό υπολογιστή το 1950, η βιομηχανία πληροφορικής έχει βελτιώσει τη σχέση κόστος-απόδοση της υπολογιστικής ισχύος περίπου 100 δισεκατομμύρια φορές. Για περισσότερα από 20 χρόνια, οι σχεδιαστές υπολογιστών βασίστηκαν στη γρήγορα αυξανόμενη ταχύτητα των τρανζίστορ, χάρη στην πρόοδο της τεχνολογίας πυριτίου και κατάφερναν να διπλασιάζουν την απόδοση περίπου κάθε 18 μήνες (νόμος Moore) . Η υπονοούμενη σύμβαση υλικού/λογισμικού ήταν ότι οι αυξήσεις στον αριθμό των τρανζίστορ και η απώλεια ισχύος ήταν αποδεκτές, εφόσον διατηρούταν το ίδιο προγραμματιστικό μοντέλο. Αυτή η σύμβαση οδήγησε σε καινοτομίες που δεν ήταν αποδοτικές σε αριθμό τρανζίστορ ούτε σε κατανάλωση ενέργειας, αλλά αύξαναν την απόδοση. Αυτή η κατάσταση λειτούργησε καλά έως ότου φτάσαμε στο όριο ισχύος που μπορούσε να αποδώσει ένα ολοκληρωμένο κύκλωμα.

Οι αρχιτέκτονες υπολογιστών αναγκάστηκαν να βρουν ένα νέο πρότυπο για να διατηρήσουν τη συνεχώς αυξανόμενη απόδοση. Η βιομηχανία αποφάσισε ότι η μόνη βιώσιμη επιλογή ήταν να αντικατασταθεί ο ένας επεξεργαστής, ο οποίος δεν ήταν αποδοτικός ενεργειακά, από αρκετούς πιο αποδοτικούς επεξεργαστές στο ίδιο ολοκληρωμένο κύκλωμα. Αυτό το είδος μικροεπεξεργαστή ονομάστηκε πολυπύρηνος επεξεργαστής Η βιομηχανία μικροεπεξεργαστών έτσι δήλωσε ότι το μέλλον ήταν στην παράλληλη επεξεργασία, με έναν διπλασιασμό του αριθμού επεξεργαστών ή πυρήνων σε κάθε γενιά τεχνολογίας, οι οποίοι εμφανίζονται περίπου κάθε δύο έτη.. Δηλαδή, το άλμα στους πολυπύρηνους επεξεργαστές δεν είναι βασισμένο σε μια σημαντική καινοτομία στον προγραμματισμό ή στην αρχιτεκτονική υπολογιστών, αλλά είναι ουσιαστικά μια υποχώρηση από τον όλο και δυσκολότερο στόχο δημιουργίας απλών επεξεργαστών που θα είναι ενεργειακά αποδοτικοί και με υψηλότερο ρολόι (συχνότητα λειτουργίας).[5]

Οι σχεδιαστές υλικού έχουν ακόμα αρκετούς λόγους που τοποθετούν τους επεξεργαστές μέσα στο ίδιο τσιπ. Ένα πλεονέκτημα είναι ότι η τοποθέτηση πολλαπλών πυρήνων σε ένα ολοκληρωμένο κύκλωμα σε ένα πακέτο καταλαμβάνουν λιγότερο χώρο στο τυπωμένο κύκλωμα της πλακέτας από ότι πολλά πακέτα των αντίστοιχων επεξεργαστών απλού πυρήνα. Οι πολυπύρηνοι επεξεργαστές έχουν επίσης τη δυνατότητα να μοιραστούν ένα δίαυλο, καθώς και τα κυκλώματα κρυφής μνήμης (cache). Ένα άλλο, λιγότερο εμφανές πλεονέκτημα είναι ότι οι πολλαπλοί πυρήνες σε ένα ενιαίο ολοκληρωμένο κύκλωμα βελτιώνουν τη συνοχή της κρυφής μνήμης, δηλαδή είναι πιθανόν κάποια δεδομένα που χρειάζεται ένας επεξεργαστής να υπάρχουν ήδη στην κρυφή μνήμη. Το σχήμα 2 δείχνει το διάγραμμα ενός διπύρηνου επεξεργαστή, όπου κάθε πυρήνας έχει τη δική του κρυφή μνήμη πρώτου επιπέδου (L1 cache) και μοιράζονται τον δίαυλο και την κρυφή μνήμη δευτέρου επιπέδου (L2 cache).

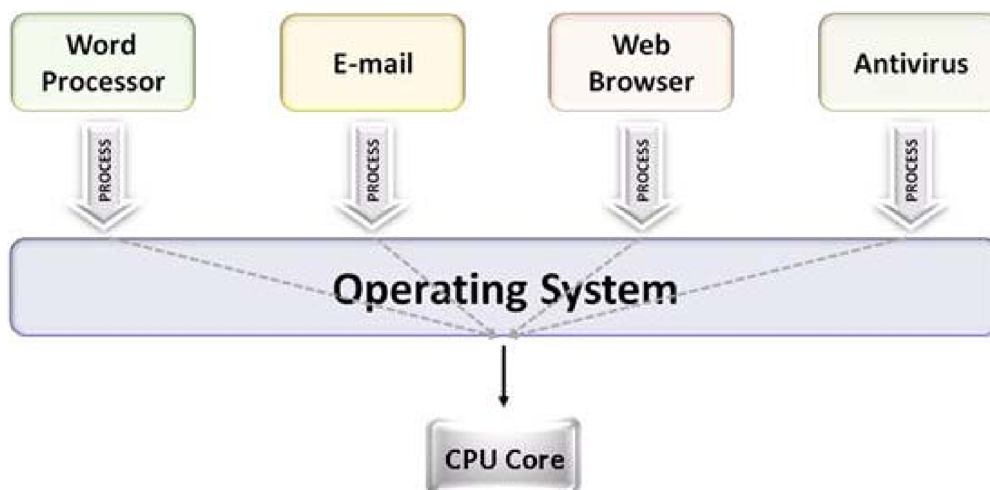


Σχήμα 2. [6]

Η εξοικονόμηση ενέργειας στους πολυπύρηνους επεξεργαστές είναι επίσης ένα σημαντικό πλεονέκτημα. Δεδομένου ότι οι πυρήνες βρίσκονται στο ίδιο τσιπ, τα σήματα μεταξύ των πυρήνων ταξιδεύουν μικρότερες αποστάσεις, άρα έχουμε λιγότερες απώλειες ενέργειας. Μια άλλη πιθανή περιοχή για την εξοικονόμηση ενέργειας είναι με την ταχύτητα ρολογιού. Οι πολυπύρηνοι επεξεργαστές μπορούν να εκτελέσουν πολλές περισσότερες πράξεις ανά δευτερόλεπτο, ενώ λειτουργούν σε χαμηλότερη συχνότητα. Για παράδειγμα, ο MIT RAW επεξεργαστής 16 πυρήνων λειτουργεί σε 425 MHz και μπορεί να εκτελέσει πάνω από 100 φορές περισσότερες πράξεις ανά δευτερόλεπτο, σε σχέση με έναν επεξεργαστή Intel Pentium 3 που λειτουργεί στα 600 MHz. Η συχνότητα λειτουργίας επηρεάζει την κατανάλωση ενέργειας του επεξεργαστή με αρκετά περίπλοκο τρόπο, αλλά ένας βασικός κανόνας είναι ότι για κάθε ένα τοις εκατό αύξηση στην ταχύτητα ρολογιού θα αυξηθεί κατά τρία τοις εκατό η κατανάλωση ενέργειας, με την προϋπόθεση ότι οι άλλοι παράγοντες που επηρεάζουν την κατανάλωση ρεύματος δεν έχουν μεταβληθεί. [6]

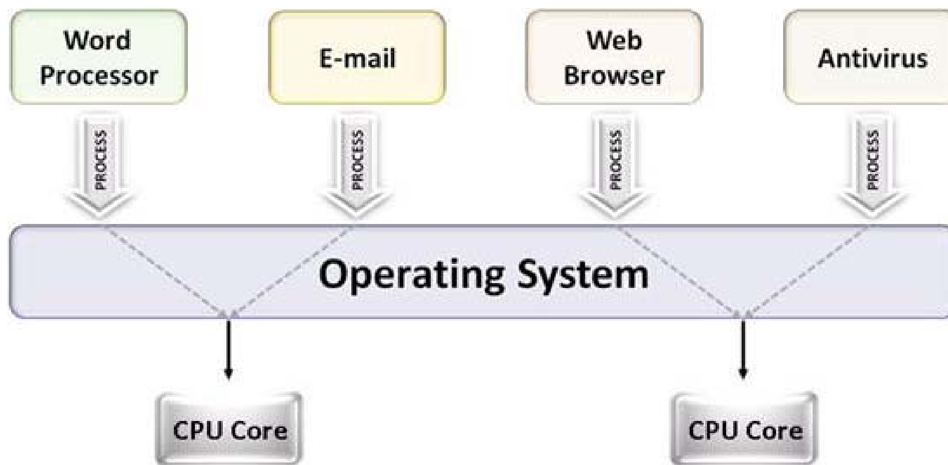
2.2.3 Αξιοποίηση πολυπύρηνων επεξεργαστών

Αρχικά στους απλούς επεξεργαστές ενός πυρήνα είχε αναπτυχθεί η πολυδιεργασία (Multitasking), όπου πολλές διεργασίες μοιράζονταν από κοινού πόρους, όπως ο επεξεργαστής. Έτσι το λειτουργικό σύστημα ήταν υπεύθυνο για την εναλλαγή των διεργασιών αρκετά γρήγορα, ώστε να δίνει την εντύπωση ότι πολλές διεργασίες εκτελούνται ταυτόχρονα. Φυσικά στον επεξεργαστή μόνο οι εντολές μιας διεργασία εκτελούνται σε κάθε στιγμή. Με την πάροδο των χρόνων οι επεξεργαστές γίνονταν όλο και πιο γρήγοροι, οπότε οι διεργασίες εκτελούνταν και εναλλάσσονταν γρηγορότερα και αυξανόταν η απόδοση συνολικά. Στο Σχήμα 3 υπάρχουν τέσσερις διεργασίες που εκτελούνται σε επεξεργαστή ενός πυρήνα. [7]



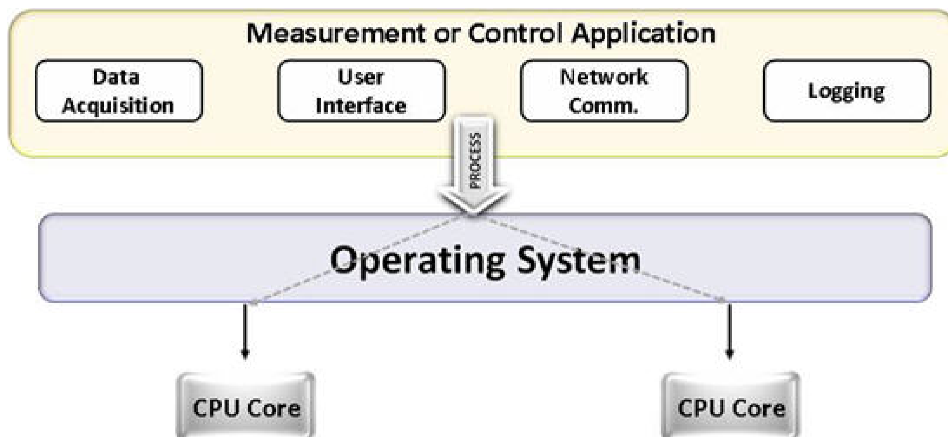
Σχήμα 3.[7]

Όταν έχουμε έναν πολυπύρηνο επεξεργαστή, το λειτουργικό σύστημα μπορεί να εκτελέσει πολλές διεργασίες πραγματικά ταυτόχρονα. Στο Σχήμα 4 τέσσερις εφαρμογές εκτελούνται σε έναν διπύρηνο επεξεργαστή. Η απόδοση αυξάνεται, αφού σε σύγκριση με πριν, το φορτίο εργασίας έχει μοιραστεί σε δύο επεξεργαστές.



Σχήμα 4.[7]

Όμως η πραγματική εκμετάλλευση ενός πολυπύρηνου επεξεργαστή επιτυγχάνεται, όταν κάθε διεργασία έχει πολλά νήματα εκτέλεσης. Ο προγραμματιστής χωρίζει το έργο της εφαρμογής σε νήματα, όπου κάθε νήμα εκτελείται ανεξάρτητα από τα υπόλοιπα και μπορεί να έχει διαφορετική προτεραιότητα. Έτσι έχουμε παραλληλισμό μέσα σε μία εφαρμογή, οπότε ο πολυπύρηνος επεξεργαστής χρησιμοποιείται αποδοτικότερα (αύξηση CPU utilization). Η δημιουργία και διαχείριση νημάτων υποστηρίζεται από τις προγραμματιστικές διεπαφές του λειτουργικού συστήματος (Win32 σε Windows, POSIX σε Linux) και από την κλάση Thread της βιβλιοθήκης της Java (η υλοποίηση της από το λειτουργικό σύστημα θα αναφερθεί σε επόμενο κεφάλαιο). Το λειτουργικό σύστημα κατανέμει το χρόνο επεξεργασίας μεταξύ των νημάτων της διεργασίας. Στο σχήμα 5 φαίνεται μια εφαρμογή ελέγχου της οποίας το έργο έχει κατανεμηθεί σε τέσσερα νήματα (διεπαφή χρήστη, αίτηση δεδομένων, επικοινωνία με το δίκτυο και τήρηση αρχείου).



Σχήμα 5.[7]

Η διάδοση των πολυπύρηνων συστημάτων όμως όχι μόνο έδωσε μεγάλη αύξηση στην υπολογιστική ισχύ, αλλά δίνει την ευκαιρία για μια ανατροπή στο σχεδιασμό των ενσωματωμένων συστημάτων. Οι σχεδιαστές θα μπορούν να έχουν ένα λειτουργικό σύστημα γενικού σκοπού και ένα λειτουργικό σύστημα πραγματικού χρόνου σε ξεχωριστούς πυρήνες ενός επεξεργαστή. Έτσι μειώνεται το κόστος, το μέγεθος και η πολυπλοκότητα στη δημιουργία συστημάτων που κάποτε απατούσαν πολλές διαφορετικές πλατφόρμες υλικού. Το σύστημα πραγματικού χρόνου, έχοντας αποκλειστικά στη διάθεσή του έναν πυρήνα, έχει όλο το χρόνο του επεξεργαστή, ώστε να ανταποκριθεί στις απαιτήσεις. [8]

Μια πολύ καλή λύση είναι να χρησιμοποιηθεί ένα λειτουργικό σύστημα πραγματικού χρόνου που να υποστηρίζει εικονικοποίηση (virtualization). Με τη δημιουργία εικονικών μηχανών, τα λειτουργικά συστήματα μπορούν να εκτελούνται σε διαφορετικούς πυρήνες απομονωμένα το ένα από το άλλο. Επίσης με αυτόν τον τρόπο είναι δυνατόν παλαιά συστήματα πραγματικού χρόνου να επεκταθούν με νέες λειτουργίες. Η συντήρηση και η επέκταση θα είναι απλούστερες, γιατί θα μπορεί να επιλεγεί μια πλατφόρμα με περισσότερους πυρήνες, η οποία θα προσφέρει επίσης νεότερα και περισσότερα υποσυστήματα εισόδου-εξόδου (π.χ. θύρες USB, Ethernet). Συνολικά θα υπάρχει κέρδος, επειδή θα υπάρχει ταχύτερη επικοινωνία και συνεργασία μεταξύ των υποσυστημάτων των διαφορετικών λειτουργικών και θα εξαλειφθεί το περιττό υλικό για επικοινωνία. Εκτός από λόγους απόδοσης, μια εφαρμογή μπορεί να απομονώνεται σε έναν πυρήνα κατά την ανάπτυξη για να δοκιμαστεί και να διευκολυνθεί η διόρθωση σφαλμάτων (debugging).

Νόμος Amdahl

Υπάρχει όμως ένας περιορισμός στο κέρδος της απόδοσης από τη χρήση ενός πολυπύρηνου επεξεργαστή, αφού εξαρτάται πάρα πολύ από τους αλγόριθμους και την υλοποίηση του λογισμικού. Πιο συγκεκριμένα, τα πιθανά κέρδη προέρχονται από το τμήμα του λογισμικού που μπορεί να παραλληλοποιηθεί και να τρέχει σε πολλούς πυρήνες ταυτόχρονα. Πάντα όμως υπάρχει κάποιο τμήμα το οποίο πρέπει να εκτελεστεί σειριακά, οπότε περιορίζεται η απόδοση. Το αποτέλεσμα αυτό περιγράφεται από το νόμο Amdahl [3]: Αν μπορεί ένα ποσοστό P μιας εργασίας να επιταχυνθεί κατά S, τότε η συνολική επιτάχυνση A είναι:

$$A = \frac{1}{(1-P) + \frac{P}{S}}$$

Το (1-P) συμβολίζει το τμήμα που είναι σειριακό και δεν επιταχύνεται. Αυτό μας δείχνει πως ανάλογα με το ποσοστό του προγράμματος που μπορεί να παραλληλοποιηθεί, η αύξηση των επεξεργαστών από κάποιο σημείο και έπειτα αναμένεται να μην επιφέρει ουσιαστική βελτίωση.

Έστω ότι μπορεί να γραφεί παράλληλα το 90% μιας εφαρμογής και κατόπιν εκτελείται σε 4 επεξεργαστές. Σε σχέση με την εκτέλεση σε έναν επεξεργαστή, από τον παραπάνω τύπο προκύπτει αύξηση της απόδοσης περίπου 3,08 αντί τετραπλασιασμού που αναμέναμε.

Κεφάλαιο 3

Υποστήριξη παραλληλισμού από το Λειτουργικό Σύστημα

Στα σύγχρονα υπολογιστικά συστήματα απαιτείται να τρέχουν ταυτόχρονα πολλά προγράμματα. Ένα πρόγραμμα έτοιμο προς εκτέλεση ονομάζεται διεργασία (process). Χρονοπρογραμματιστής – scheduler ονομάζεται ο κώδικας του πυρήνα του λειτουργικού συστήματος που αποφασίζει ποια διεργασία θα επιλεγεί για εκτέλεση και πότε θα συμβεί η εναλλαγή μεταξύ των διεργασιών. Όλα τα σύγχρονα λειτουργικά συστήματα μπορούν να αξιοποιήσουν παραπάνω από έναν επεξεργαστές. Στη συνέχεια θα εξεταστεί ο τρόπος λειτουργίας του χρονοπρογραμματιστή. Ιδιαίτερη έμφαση θα δοθεί στον τρόπο με τον οποίο μπορούμε να αναθέσουμε μια διεργασία να εκτελείται σε συγκεκριμένους επεξεργαστές.

3.1 Χρονοπρογραμματισμός - Scheduling

3.1.1 Βασικές έννοιες

Χρονοπρογραμματισμός (scheduling) ονομάζεται η αρμοδιότητα των λειτουργικών συστημάτων, υλοποιούμενη συνήθως από έναν μηχανισμό του πυρήνα ονόματι χρονοπρογραμματιστής (scheduler), με την οποία συντονίζεται η συνύπαρξη πολλαπλών εκτελούμενων διεργασιών στη μνήμη του υπολογιστή. Με τον χρονοπρογραμματισμό επιτυγχάνεται επομένως η πολυδιεργασία (multitasking), η οποία με τη σειρά της αποτελεί έναν τρόπο πρακτικής υλοποίησης ταυτοχρονισμού καθώς, είτε με κατάλληλη κατανομή του χρόνου του μοναδικού επεξεργαστή (ψευδοπαράλληλη εκτέλεση) είτε λόγω της ύπαρξης περισσοτέρων του ενός επεξεργαστών (παράλληλη εκτέλεση), είναι εφικτή η ταυτόχρονη εκτέλεση πολλαπλών διεργασιών στον ίδιο ηλεκτρονικό υπολογιστή. [9]

Στα σύγχρονα προεκτοπιστικά (preemptive) λειτουργικά συστήματα, στον επεξεργαστή συμβαίνει αυτόματη εναλλαγή διεργασιών σε τακτικά χρονικά διαστήματα (στην αρχή κάθε «χρονικού κβάντου») ώστε να επιτευχθεί η ψευδοπαράλληλη εκτέλεση πολλαπλών διεργασιών. Στην πραγματικότητα οι διεργασίες εναλλάσσονται στον επεξεργαστή με εξαιρετικά μεγάλη συχνότητα (συνήθως το κβάντο διαρκεί κάποια millisecond). Η εναλλαγή αυτή ονομάζεται context switch (εναλλαγή πλαισίου) και, προκειμένου να είναι εφικτή, πρέπει όλες οι πληροφορίες για την εκτελούμενη διεργασία, οι οποίες είναι αποθηκευμένες στην τοπική μνήμη του επεξεργαστή, δηλαδή στους καταχωρητές, να αποθηκευτούν στην κύρια μνήμη κατά την εναλλαγή πλαισίου.Όταν έρθει η σειρά της διεργασίας που μπήκε σε αναστολή

να εκτελεστεί, θα μπορούν να φορτωθούν πάλι πίσω στους καταχωρητές και η εκτέλεση να συνεχίσει από εκεί που είχε σταμάτησε.

Ο χρονοπρογραμματιστής ασχολείται κυρίως με:

•Τη χρήση επεξεργαστή (CPU utilization): προσπαθεί να κρατήσει τον επεξεργαστή όσο το δυνατόν περισσότερο απασχολημένο

•Απόδοση (Throughput): αριθμός διεργασιών που ολοκληρώνουν την εκτέλεσή τους ανά μονάδα χρόνου.

•Turnaround: το χρονικό διάστημα για να ολοκληρωθεί μια συγκεκριμένη διεργασία.

•Χρόνος αναμονής: το χρονικό διάστημα που μια διεργασία περιμένει στην ουρά προς εκτέλεση.

•Χρόνος απόκρισης: το χρονικό διάστημα από όταν ένα αίτημα υποβλήθηκε μέχρι tην πρώτη απάντηση.

•Δικαιοσύνη: ίσοι χρόνοι για κάθε νήμα

Σε περιβάλλοντα πραγματικού χρόνου, όπως για κινούμενες συσκευές αυτόματου ελέγχου στη βιομηχανία (για παράδειγμα η ρομποτική), το χρονοδιάγραμμα και οι διαδικασίες πρέπει να εξασφαλίζουν ότι μπορεί να ανταποκριθεί σε προθεσμίες. Αυτό είναι ζωτικής σημασίας για τη διατήρηση της σταθερότητας του συστήματος.

Η υλοποίηση ενός αλγόριθμου χρονοπρογραμματισμού είναι δύσκολη για δυο λόγους. Πρώτον, ένας αποδεκτός αλγόριθμος πρέπει ναι κατανέμει το χρόνο του επεξεργαστή έτσι, ώστε οι υψηλότερης προτεραιότητας εργασίες (π.χ. διαδραστικές εφαρμογές, όπως ο web browser) να προτιμούνται έναντι αυτών χαμηλής προτεραιότητας (π.χ. μη διαδραστικές διαδικασίες όπως η μεταγλώττιση ενός προγράμματος). Ταυτόχρονα, ο χρονοπρογραμματιστής πρέπει να προστατεύει τις χαμηλής προτεραιότητας διεργασίες από λιμοκτονία. Με άλλα λόγια, οι χαμηλής προτεραιότητας διεργασίες τελικά πρέπει να επιτρέπεται να εκτελεστούν, ανεξάρτητα από το πόσες διεργασίες υψηλής προτεραιότητας υπάρχουν. Επιπλέον, ο χρονοπρογραμματιστής πρέπει να είναι προσεκτικά κατασκευασμένος ώστε να μην έχει πολύ μεγάλες επιπτώσεις στην απόδοση του συστήματος και οι διεργασίες να φαίνεται ότι εκτελούνται ταυτόχρονα.

Μια διεργασία μπορεί να ανταποκριθεί στις απαιτήσεις του χρήστη μόνο όταν θα έχει τον έλεγχο του επεξεργαστή. Επειδή οι χρήστες αναμένουν οι διαδραστικές διεργασίες να παρουσιάζουν άμεση ανταπόκριση στην είσοδο, η καθυστέρηση μεταξύ της εισόδου του χρήστη και της εκτέλεσης της διεργασίας θα πρέπει να είναι ανεπαίσθητη (κάπου μεταξύ 50 και 150ms συνήθως αρκεί, λαμβάνοντας υπόψη ότι ο χρόνος αντίδρασης του ανθρώπου είναι περίπου 200ms).

Για τις μη διαδραστικές διεργασίες η κατάσταση είναι διαφορετική, καθώς η εναλλαγή μεταξύ των διεργασιών, δηλαδή η εναλλαγή πλαισίου, είναι σχετικά δαπανηρή λειτουργία. Έτσι, μεγαλύτερα διαστήματα χρόνου του επεξεργαστή, και λιγότερες διακοπές μπορούν να βελτιώσουν την απόδοση του συστήματος. Ο αλγόριθμος προγραμματισμού πρέπει να επιτύχει μια ισορροπία μεταξύ όλων αυτών των ανταγωνιστικών αναγκών [10] .

3.1.2 Χρονοπρογραμματισμός στο λειτουργικό σύστημα Windows

Τα βασισμένα σε Windows NT λειτουργικά συστήματα χρησιμοποιούν μια πολυεπίπεδη ανατροφοδοτούμενη ουρά. Ορίζονται 32 επίπεδα προτεραιότητας, 0 έως 31, με προτεραιότητες 0 έως 15 να θεωρούνται "φυσιολογικές" προτεραιότητες και τις προτεραιότητες 16 έως 31 να θεωρούνται προτεραιότητες χαλαρού πραγματικού χρόνου, απαιτώντας προνόμια για να ανατεθούν. Το 0 προορίζεται αποκλειστικά για ο λειτουργικό σύστημα. Οι χρήστες μπορούν να επιλέξουν 5 από τις προτεραιότητες αυτές για να τις αναθέσουν σε μια εφαρμογή που τρέχει από την Διαχείριση Εργασιών (Task Manager) , είτε μέσω ενός API διαχείρισης νημάτων. Οι 5 βασικές προτεραιότητες και οι αντίστοιχες τιμές είναι:[11]

Realtime	24
High	13
Above Normal	10
Normal	8
Below normal	6
Low	4

Ο πυρήνας μπορεί να αλλάξει το επίπεδο προτεραιότητας ενός νήματος ανάλογα με τις εισόδους/εξόδους (I/O) και τη χρήση του επεξεργαστή και εάν το νήμα είναι διαδραστικό (δηλαδή δέχεται και ανταποκρίνεται στις εισόδους από τον άνθρωπο). Έτσι, αυξάνει την προτεραιότητα των διαδραστικών διεργασιών και των διεργασιών δεσμευμένων σε εισόδους/εξόδους (I/O bound) και μειώνει των διεργασιών δεσμευμένων σε επεξεργαστή (CPU bound), για την αύξηση της ανταπόκρισης των διαδραστικών εφαρμογών.

Ο χρονοπρογραμματιστής τροποποιήθηκε στα Windows Vista για να χρησιμοποιεί τους καταχωρητές μέτρησης απόδοσης (Performance Counters) των σύγχρονων επεξεργαστών για να παρακολουθεί ακριβώς πόσους κύκλους μηχανής έχει εκτελεστεί ένα νήμα, αντί απλώς να χρησιμοποιεί το διάστημα του χρονομέτρου μιας ρουτίνας διακοπής. Τα Vista χρησιμοποιούν επίσης χρονοπρογραμματιστή προτεραιότητας για την ουρά εισόδου/εξόδου έτσι ώστε προγράμματα όπως η ανασυγκρότηση δίσκου να μην διαπλέκονται με τις λειτουργίες προσκηνίου. [9]

3.1.2 Χρονοπρογραμματισμός στο λειτουργικό σύστημα Linux

Η πρώτη επίσημη έκδοση 1.0 του Linux υποστήριζε το 1994 υπολογιστές με έναν μόνο επεξεργαστή i386. Η επόμενη έκδοση 1.2, το 1995, υποστήριζε πολλές πλατφόρμες υλικού (Alpha, Sparc, Mips), αλλά είχε πάλι υποστήριξη για έναν μόνο επεξεργαστή. Η έκδοση 2.0 τον Ιούνιο του 1996 υποστήριζε περισσότερες αρχιτεκτονικές και κυρίως πολυεπεξεργαστικές μηχανές (Symmetric Multiprossecors - SMP). Σήμερα το Linux υποστηρίζει πολλές αρχιτεκτονικές επεξεργαστών και υπάρχουν εκδόσεις τόσο για μεγάλους εξυπηρετητές (servers) με πολλούς επεξεργαστές όσο για διάφορες μικρές φορητές συσκευές και ενσωματωμένα συστήματα.

Από την έκδοση 2.5 του πυρήνα και μετά, το Linux χρησιμοποιεί μια πολυεπίπεδη ουρά ανατροφοδότησης με επίπεδα προτεραιοτήτων που κυμαίνονται από 0 ως 140. Τα επίπεδα 0-99 είναι δεσμευμένα για διεργασίες πραγματικού χρόνου, ενώ τα 100-140 για διεργασίες του χρήστη. Σε κάθε διεργασία έχει δοθεί ένα κβάντο χρόνου με βάση μια στατική προτεραιότητα. Το κβάντο χρόνου καθορίζει το χρόνο που μπορεί η διεργασία να εκτελείται στον επεξεργαστής πριν διακοπεί, ώστε κάποια άλλη διεργασία να εκτελεστεί. Για εργασίες πραγματικού χρόνου, το κβάντο χρόνου για εναλλαγή διεργασιών είναι περίπου 200 ms, ενώ

για τις απλές 10 ms. Ο χρονοπρογραμματιστής διατηρεί δύο λίστες, μια λίστα ενεργών διεργασιών και μια ανενεργών (expired, αυτών που έχουν λήξει). Όταν δημιουργηθεί μια νέα διεργασία, ξεκινά από την ενεργή λίστα. Ο χρονοπρογραμματιστής θα διατρέξει τη λίστα των έτοιμων προς εκτέλεση διεργασιών (ready queue) και θα αφήσει τις υψηλότερες να εκτελεστούν πρώτα μέσα στο κβάντο χρόνου τους και μετά θα τις τοποθετήσει στην ανενεργή ουρά (expired queue). Όταν η ενεργή ουρά αδειάσει, τότε εναλλάσσονται οι δύο ουρές και η ουρά αυτών που έληξαν γίνεται ενεργή. Από την έκδοση 2.6 ως την 2.6.23, ο πυρήνας χρησιμοποιεί τον O(1) χρονοπρογραμματιστή. Από την έκδοση 2.6.23 αυτή η μέθοδος αντικαταστάθηκε από τον Completely Fair Scheduler – CFS (απολύτως δίκαιος χρονοπρογραμματιστής) [9] .

Ο O(1) χρονοπρογραμματιστής

Ένα θεμελιώδες χαρακτηριστικό αυτού του χρονοπρογραμματιστή είναι ο χρόνος εκτέλεσης του αλγορίθμου. Ένας O(1) αλγόριθμος χρειάζεται το ίδιο χρόνο για να εκτελεστεί ανεξάρτητα από το μέγεθος των δεδομένων εισόδου (στην περίπτωση αυτή, ο αριθμός των διεργασιών που λειτουργούν με το σύστημα). Ένα σημαντικό πρόβλημα των παλιότερων εκδόσεων χρονοπρογραμματιστή, ειδικά σε υψηλά φορτία, όταν υπήρχαν πολλές διεργασίες, ήταν ότι έπαιρναν αρκετό χρόνο να εκτελεστούν, αφού ο αλγόριθμος ήταν πολυπλοκότητας O(n). Επίσης χρησιμοποιούταν μόνο μια ουρά αναμονής για όλους επεξεργαστές, οπότε μόνο ένας επεξεργαστής μπορούσε να έχει πρόσβαση σε αυτή κάθε χρονική στιγμή. Μια διεργασία μπορούσε να εκτελεστεί σε οποιονδήποτε επεξεργαστή, το οποίο είναι θετικό για την κατανομή φορτίου, αλλά κακό για την απόδοση των κρυφών μνημών (cache memory). Επίσης η εικονική μηχανή Java, η οποία χρησιμοποιεί πολλά νήματα εκτέλεσης, δεν μπορούσε να εκτελεστεί αποδοτικά και αυτό ήταν ένας λόγος για τη δημιουργία του O(1) χρονοπρογραμματιστή.

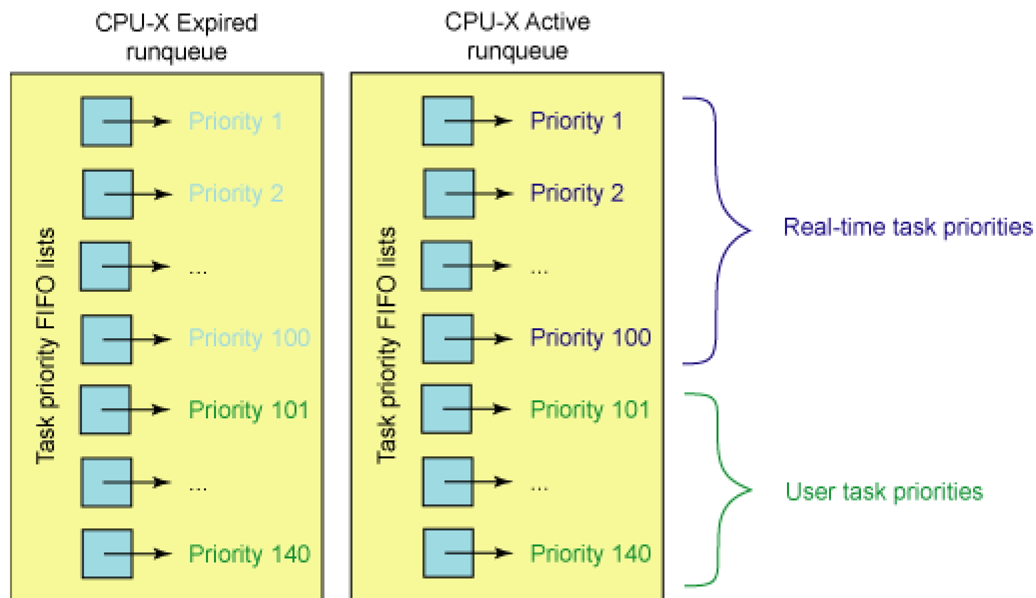
Στον O(1) χρονοπρογραμματιστή κάθε επεξεργαστής έχει μια δική του ουρά εκτέλεσης με 140 επίπεδα προτεραιοτήτων καθώς και μια ανενεργή ουρά όπως φαίνεται στο σχήμα 1, και περιγράφηκε παραπάνω. Αυτό έχει ως αποτέλεσμα την αποφυγή του προβλήματος κλειδώματος της ουράς και τη βελτίωση της συνάφειας επεξεργαστή. Επίσης επειδή δεν είναι γνωστό εκ των προτέρων πόσο χρόνο θα χρειαστεί μια διεργασία για να ολοκληρωθεί, η αρχική κατανομή στους επεξεργαστές πρέπει να τροποποιηθεί για να υπάρξει μια εξισορρόπηση φορτίου. Κάθε 200ms εξετάζεται αν κάποιος επεξεργαστής έχει μεγαλύτερο φορτίο, οπότε κάποιες διεργασίες αναδιανέμονται στις ουρές άλλων επεξεργαστών. Το αρνητικό αυτής της λειτουργίας είναι ότι η τοπική κρυφή μνήμη είναι «κρύα», δηλαδή δεν περιέχει τα δεδομένα της διεργασίας.

Επίσης μια σημαντική προσθήκη είναι ότι επιτρέπεται η διακοπή μιας διεργασίας για να εκτελεστεί μια διεργασία υψηλότερου επιπέδου.

Ένα άλλο προηγμένο χαρακτηριστικό του O(1) χρονοπρογραμματιστή είναι η δυναμική αλλαγή προτεραιοτήτων των διεργασιών. Ο χρονοπρογραμματιστής τιμωρεί τις διεργασίες που χρησιμοποιούν πολύ τον επεξεργαστή (δεσμευμένες στον επεξεργαστή) και επιβραβεύει όσες χρησιμοποιούν περισσότερο είσοδο/έξοδο (δεσμευμένες σε είσοδο/έξοδο), αυξάνοντας ή μειώνοντας την προτεραιότητα ως και πέντε επίπεδα. Μια διεργασία συνήθως χρειάζεται να περιμένει αρκετά για να ολοκληρωθεί η είσοδος/έξοδος, οπότε μεταβαίνει σε κατάσταση ύπνου και παραχωρεί τη θέση της σε άλλες διεργασίες. Ο αλγόριθμος προσπαθεί να προσδιορίσει τις διαδραστικές διεργασίες μέσω κάποιων ευριστικών μεθόδων, αναλύοντας το μέσο χρόνο ύπνου. Διεργασίες που βρίσκονται σε κατάσταση ύπνου για μεγάλα χρονικά διαστήματα είναι

πιθανόν να περιμένουν το χρήστη, οπότε ο χρονοπρογραμματιστής χαρακτηρίζει την διεργασία ως διαδραστική.

Αξίζει να σημειωθεί ότι η αλλαγή στις προτεραιότητες συμβαίνει μόνο στις διεργασίες του επιπέδου του χρήστη (επίπεδα 100-140) και όχι στις διεργασίες πραγματικού χρόνου (0-99). [12]



Σχήμα 1. [12]

Ο CFS χρονοπρογραμματιστής

Από την έκδοση 2.6.23 ο χρονοπρογραμματιστής O(1) αντικαταστάθηκε από τον Completely Fair Scheduler – CFS (απολύτως δίκαιος χρονοπρογραμματιστής), ο οποίος χρησιμοποιεί κόκκινα - μαύρα δένδρα αντί για ουρές. Το πρόβλημα με τον O(1) ήταν ο μεγάλος και πολύπλοκος κώδικας των ευριστικών μεθόδων για την αναγνώριση των διαδραστικών διεργασιών (με βάση το χρόνο σε κατάσταση ύπνου). Ο CFS χρησιμοποιεί έναν απλό αλγόριθμο χρονοπρογραμματισμού σε σχέση με τον O(1), αφού δεν χρησιμοποιεί πια αυτές τις ευριστικές μεθόδους. Με τον αλγόριθμο RSDL του Con Kolivas, στον οποίο βασίστηκε ο CFS, έχει αποδειχτεί ότι ο δίκαιος χρονοπρογραμματισμός μπορεί να επιτευχθεί χωρίς να οδηγεί σε καθυστέρηση των διαδραστικών διεργασιών. [13]

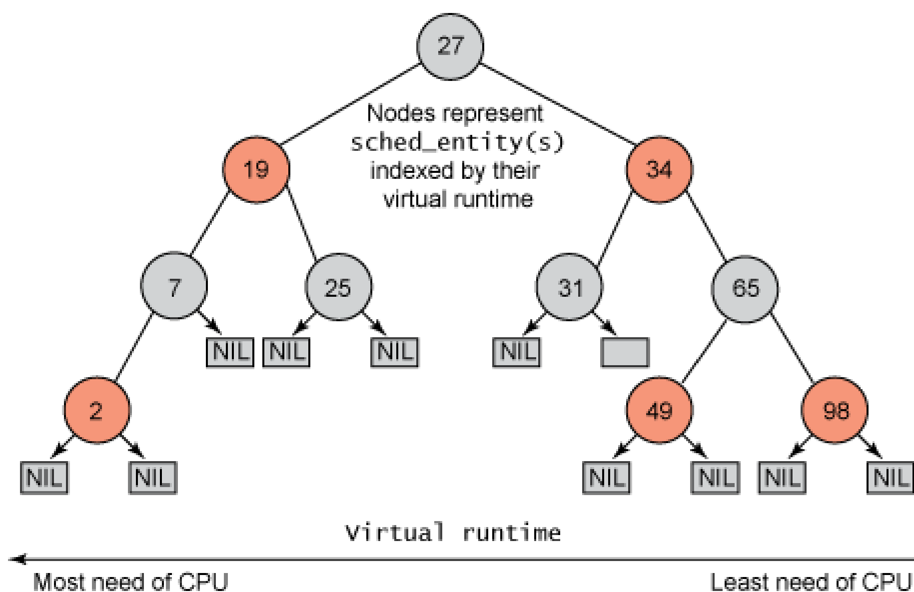
Η βασική ιδέα αυτού του χρονοπρογραμματιστή είναι να διατηρήσει την ισορροπία (δικαιοσύνη) στην κατανομή του χρόνου μεταξύ των εργασιών. Αυτό σημαίνει ότι οι διεργασίες πρέπει να έχουν ένα δίκαιο μερίδιο χρόνου του επεξεργαστή και όταν κάποιες δεν έχουν αρκετό χρόνο σε σχέση με τις υπόλοιπες, πρέπει να τους δοθεί χρόνος προς εκτέλεση. Ο συγγραφέας Ingo Molnar περιγράφει τον CFS [2] ως «μοντελοποίηση ενός ιδανικού παράλληλου επεξεργαστή». Ένας τέτοιος επεξεργαστής πρέπει να τρέχει παράλληλα πολλές διεργασίες την ίδια στιγμή δίνοντας στην καθεμία ένα ίσο μερίδιο επεξεργαστικής ισχύος. Για να καθοριστεί η ισορροπία, ο CFS παρακολουθεί το χρόνο που παρέχεται σε μια συγκεκριμένη εργασία να έχει πρόσβαση στον επεξεργαστή, το οποίο ονομάζεται εικονικός χρόνος

εκτέλεσης. Όσο μικρότερος είναι ο εικονικός χρόνος εκτέλεσης, τόσο μεγαλύτερη είναι η ανάγκη για χρόνο επεξεργασίας. Ακόμα, ο CFS συμπεριλαμβάνει την δικαιοσύνη σε κατάσταση ύπνου, δηλαδή προσπαθεί να εξασφαλίσει ότι εργασίες που δεν εκτελούνται επειδή είναι σε κατάσταση ύπνου (π.χ. περιμένοντας είσοδο/έξοδο), θα λάβουν ένα μερίδιο χρόνου στον επεξεργαστή, όταν το χρειαστούν.

Αντί να διατηρεί τις διεργασίες σε ουρές εκτέλεσης, όπως όλοι οι προηγούμενοι χρονοπρογραμματιστές, ο CFS διατηρεί ένα ταξινομημένο ως προς το χρόνο κόκκινο-μαύρο δένδρο όπως φαίνεται στο σχήμα 2. Ένα κόκκινο-μαύρο δένδρο είναι μια δομή δεδομένων με μερικές πολύ χρήσιμες ιδιότητες. Πρώτον, κρατά από μόνο του μια ισορροπία, αφού κανένα μονοπάτι στο δένδρο δεν μπορεί να είναι το πολύ διπλάσιο από οποιοδήποτε άλλο. Δεύτερον, όλες οι λειτουργίες εκτελούνται σε O(log n) χρόνο, όπου n ο αριθμός των κόμβων στο δένδρο, το οποίο σημαίνει ότι η εισαγωγή ή η διαγραφή μιας διεργασίας από το δένδρο μπορεί να γίνει αρκετά γρήγορα και αποδοτικά. [14]

Οι διεργασίες (αναπαρίστανται από αντικείμενα τύπου sched_entity) αποθηκεύονται στο δένδρο ταξινομημένες με βάση το χρόνο και αυτές με σοβαρότερη ανάγκη για χρόνο εκτέλεσης βρίσκονται στα αριστερά του δένδρου.

Ο χρονοπρογραμματιστής για να είναι δίκαιος επιλέγει τη διεργασία στον πιο αριστερό κόμβο για εκτέλεση. Η διεργασία προσθέτει τον χρόνο που κατείχε τον επεξεργαστή στον εικονικό χρόνο εκτέλεσης και τοποθετείται πάλι πίσω στο δένδρο αν είναι έτοιμη προς εκτέλεση. Με αυτόν τον τρόπο, οι διεργασίες στην αριστερή πλευρά του δένδρου παίρνουν χρόνο προς εκτέλεση και μετά μετακινούνται προς τα δεξιά του δένδρου για να διατηρηθεί η δικαιοσύνη.



Σχήμα 2. [13]

Ο CFS δεν χρησιμοποιεί τις προτεραιότητες άμεσα, αλλά τις χρησιμοποιεί σαν ένα παράγοντα εξασθένησης (decay factor) για τον χρόνο που δικαιούται η διεργασία. Οι διεργασίες χαμηλής προτεραιότητας έχουν υψηλότερους παράγοντες εξασθένισης και οι υψηλής προτεραιότητας χαμηλούς. Άρα μια διεργασία χαμηλής προτεραιότητας έχει λιγότερο

χρόνο στη διάθεσή της σε σχέση με μια υψηλής. Αυτή είναι μια λύση που επιτρέπει να μην χρησιμοποιηθούν ουρές εκτέλεσης με βάση την προτεραιότητα.

Άλλη μια ενδιαφέρουσα λειτουργία του CFS είναι ότι επιτρέπει τον χρονοπρογραμματισμό σε ομάδες. Αυτός είναι ένας τρόπος να επιφέρει δικαιοσύνη στον χρονοπρογραμματισμό, όταν κάποιες διεργασίες δημιουργούν πολλές νέες διεργασίες. Αντί όλες οι διεργασίες να αντιμετωπίζονται ισότιμα, μια ομάδα διεργασιών μοιράζεται τον εικονικό χρόνο εκτέλεσης, ενώ οι απλές ανεξάρτητες διεργασίες διατηρούν τον δικό τους εικονικό χρόνο εκτέλεσης. Επομένως μια διεργασία μόνη της θα λάβει περίπου τον ίδιο χρόνο με μια ομάδα. Για παράδειγμα, έστω ότι έχουμε δύο χρήστες A και B, όπου ο A έχει 2 εργασίες και ο B 48 για να εκτελέσουν στο ίδιο μηχάνημα. Ο ομαδικός χρονοπρογραμματισμός επιτρέπει στον CFS να είναι δίκαιος, αφού και ο A και ο B θα πάρουν από 50% του χρόνου του επεξεργαστή για να εκτελέσουν τις εργασίες τους, αντί για 4% και 96%.

3.2 Συνάφεια επεξεργαστή (CPU affinity)

Η ικανότητα του λειτουργικού συστήματος να επιτρέπει σε μία ή περισσότερες διεργασίες να εκτελούνται μόνο σε έναν ή περισσότερους συγκεκριμένους επεξεργαστές καλείται συνάφεια επεξεργαστή (CPU affinity) [15].

Υπάρχουν δύο τύποι συνάφειας. Η πρώτη, χαλαρή συνάφεια ή φυσική, είναι η τάση του χρονοπρογραμματιστή να προσπαθεί να κρατήσει τις διεργασίες στον ίδιο επεξεργαστή όσο το δυνατόν περισσότερο. Είναι περισσότερο μια προσπάθεια, γιατί αν αυτό δεν είναι εφικτό, οι διεργασίες μετακινούνται προς άλλον επεξεργαστή. Ο O(1) παρουσιάζει πολύ καλή φυσική συνάφεια. Η αυστηρή συνάφεια παρέχεται από μια κλήση συστήματος και είναι μια προϋπόθεση για τις διεργασίες να τρέξουν στον καθορισμένο επεξεργαστή.

Αυτό το χαρακτηριστικό είναι αρκετά χρήσιμο σε ορισμένες περιπτώσεις. Το πρώτο πλεονέκτημα είναι ότι βελτιστοποιείται η απόδοση της κρυφής μνήμης. Οι πολυεπεξεργαστικοί υπολογιστές έχουν πολλά προβλήματα στο να κρατούν έγκυρες τις εγγραφές της κρυφής μνήμης, καθώς τα δεδομένα για να είναι έγκυρα πρέπει να βρίσκονται μόνο σε μια κρυφή μνήμη τη φορά. Έτσι. όταν ένας επεξεργαστής προσθέτει μια γραμμή στα δεδομένα του, όλοι οι άλλοι πρέπει να ακυρώσουν τα δικά τους αντίγραφα των δεδομένων, το οποίο μειώνει την απόδοση του συστήματος. Επίσης σημαντικό πρόβλημα προκύπτει όταν μια διεργασία εναλλάσσεται στους επεξεργαστές, καθώς αυξάνονται οι αστοχίες στην κρυφή μνήμη. Η συνάφεια βοηθάει στην βελτίωση της ευστοχίας στην κρυφή μνήμη.

Επιπρόσθετα από το παραπάνω, αν κάποια νήματα δουλεύουν στα ίδια δεδομένα, έχει νόημα να βρίσκονται στον ίδιο επεξεργαστή, αν και αυτό ακυρώνει τα πλεονεκτήματα του ταυτόχρονου προγραμματισμού.

Ο ορισμός της συνάφειας επεξεργαστή είναι χρήσιμος σε εφαρμογές πραγματικού χρόνου. Σε αυτή την περίπτωση όλες οι διεργασίες του συστήματος προσδένονται σε ένα υποσύνολο των διαθέσιμων επεξεργαστών και η κρίσιμη εφαρμογή στους απομένοντες. Αυτό εξασφαλίζει ότι η εφαρμογή θα λάβει ολοκληρωτικά την προσοχή του επεξεργαστή.

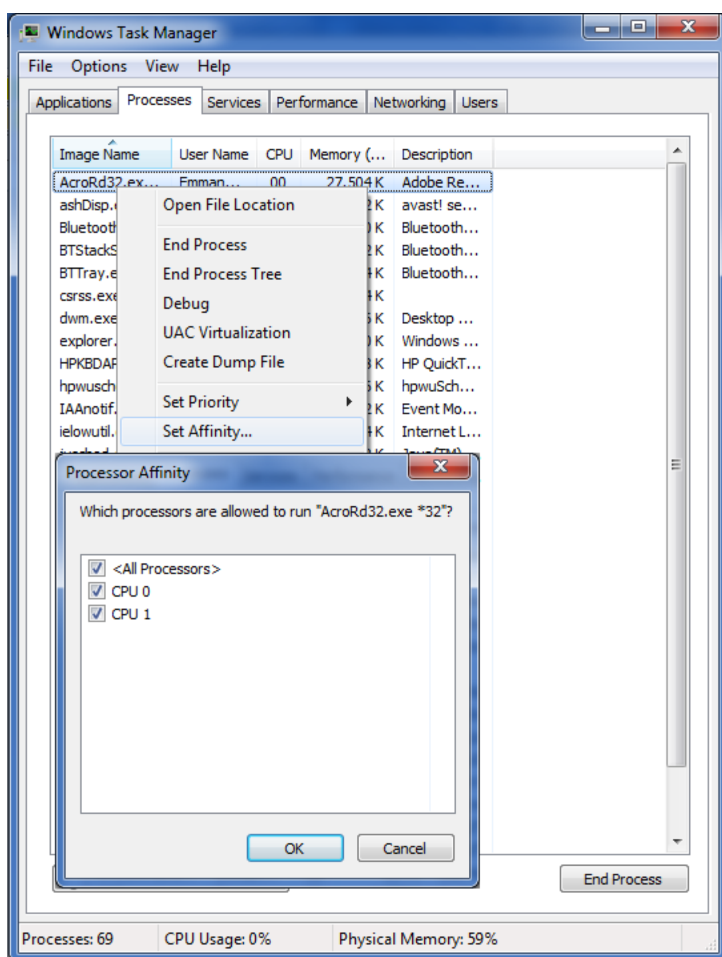
3.2.1 Ορισμός συνάφειας σε λειτουργικό σύστημα Windows

Μέσω της εφαρμογής «Διαχείριση Εργασιών» - “Task Manager” μπορούμε να θέτουμε τη συνάφεια επεξεργαστή μιας διεργασίας, όπως παρουσιάζεται στο σχήμα 3. Επίσης στον προγραμματιστή παρέχονται αρκετές συναρτήσεις σχετικές με διαχείριση διεργασιών και νημάτων. [16]

Οι πιο σημαντικές είναι:

GetProcessAffinityMask
SetProcessAffinityMask
GetThreadGroupAffinity
SetThreadGroupAffinity
SetThreadAffinityMask
GetCurrentProcessorNumber

Η λειτουργία τους είναι φανερή από το όνομά τους. Η τελευταία επιστρέφει τον αριθμό του επεξεργαστή στον οποίο εκτελείται το τρέχον νήμα κατά τη διάρκεια της κλήσης αυτής της συνάρτησης.



Σχήμα 3

3.2.2 Ορισμός συνάφειας σε λειτουργικό σύστημα Linux

Στο Linux υπάρχει η εντολή taskset που επιτρέπει στον προγραμματιστή να ορίζει τη συνάφεια μιας διεργασίας. Επιπλέον υπάρχουν και ορισμένες κλήσεις συστήματος που επιτρέπουν στον προγραμματιστή να ορίζει τη συνάφεια επεξεργαστή. [15]

Taskset

Η taskset είναι μια εντολή που επιτρέπει να θέτουμε ή να λαμβάνουμε τη συνάφεια επεξεργαστή μιας διεργασίας, όταν δοθεί το PID της (Process ID: ο μοναδικός αριθμός κάθε διεργασίας) ή να εκτελέσουμε μια νέα διεργασία με μια συγκεκριμένη συνάφεια. Ο χρονοπρογραμματιστής του Linux θα σεβαστεί τη δεδομένη συνάφεια και η διεργασία δεν θα τρέξει σε κανένα άλλο επεξεργαστή. Βέβαια και ο χρονοπρογραμματιστής προσπαθεί να κρατάει τις διεργασίες στον ίδιο επεξεργαστή όσο το δυνατόν περισσότερο για λόγους απόδοσης.

Η συνάφεια επεξεργαστή αναπαρίσταται με μια μάσκα από μπιτ (bitmask) με το bit χαμηλότερης τάξης να αντιστοιχεί στον πρώτο λογικό επεξεργαστή και το υψηλότερης τάξης στον τελευταίο. Μπορεί να μην υπάρχουν όλοι οι επεξεργαστές σε ένα σύστημα, αλλά η μάσκα μπορεί να ορίζει περισσότερους επεξεργαστές από όσους είναι φυσικά παρόντες. Μια ληφθείσα μάσκα αντιστοιχεί σε όσους επεξεργαστές είναι παρόντες. Αν δοθεί μια μη έγκυρη μάσκα, θα επιστραφεί λάθος. Οι μάσκες δίνονται τυπικά σε δεκαεξαδική μορφή. [17]

Για παράδειγμα:

0x00000001 προσδιορίζει τον επεξεργαστή 0
0x00000003 προσδιορίζει τους επεξεργαστές 0 και 1
0xFFFFFFFF προσδιορίζει όλους τους επεξεργαστές, από 0 ως 31

Όταν η εντολή εκτελεστεί χωρίς μήνυμα λάθους είναι εγγυημένο ότι το πρόγραμμα έχει χρονοπρογραμματιστεί στους επιτρεπόμενους επεξεργαστές.

Χρήση

taskset [mask] [command] [arguments]
Η προεπιλεγμένη χρήση είναι να εκτελεστεί η εντολή σε μια νέα διεργασία με τη δοθείσα μάσκα.

Παράμετροι:
-p, --pid
Επιδρά σε μια υπάρχουσα διεργασία με το συγκεκριμένο PID και δεν δημιουργείται νέα διεργασία.
-c, --cpu-list
Καθορίζει μια αριθμητική λίστα επεξεργαστών αντί μάσκας. Η λίστα μπορεί να περιέχει πολλαπλά στοιχεία, χωρισμένα με κόμμα, και εύρος τιμών, πχ 0,5,7,9-11.

Λήψη συνάφειας επεξεργαστή υπάρχουσας διεργασίας:
taskset -p [pid]

Ορισμός συνάφειας επεξεργαστή υπάρχουσας διεργασίας:
taskset -p [mask] [pid]

Κλήσεις συστήματος

Από την έκδοση 2.5 του πυρήνα εισάγονται ένα σύνολο από κλήσεις συστήματος για ρύθμιση και ανάκτηση της συνάφειας μιας διεργασίας. Ο χρονοπρογραμματιστής τότε υπακούει σε αυτές τις ρυθμίσεις και η διεργασία τρέχει μόνο στους επιτρεπόμενους πυρήνες. [15]

Για να θέσουμε τη συνάφεια, χρησιμοποιούμε μια μάσκα από bit. Στα συστήματα 32-bit η μάσκα είναι μια σειρά 32 bit, όπου το καθένα αντιπροσωπεύει αν η συγκεκριμένη διεργασία μπορεί να εκτελεστεί στον αντίστοιχο επεξεργαστή. Η προκαθορισμένη μάσκα είναι η 11111111111111111111111111111111, όπου όλοι οι επεξεργαστές είναι διαθέσιμοι.

Προϋποθέτοντας νέα έκδοση πυρήνα και της βιβλιοθήκης glibc, σε c μπορούμε να χρησιμοποιήσουμε τις παρακάτω κλήσεις:.

```
#include <sched.h>

long sched_setaffinity(pid_t pid, unsigned int len, cpu_set_t *user_mask_ptr);

long sched_getaffinity(pid_t pid, unsigned int len,  cpu_set_t *user_mask_ptr);
```

Η πρώτη από αυτές μας επιτρέπει να θέσουμε και η δεύτερη να πάρουμε την συνάφεια μιας διεργασίας. Το πρώτο όρισμα είναι το pid της επιθυμητής διεργασίας ή 0 για την τρέχουσα, το δεύτερο το μήκος σε bytes της μάσκας και το τρίτο δείκτης προς τη μάσκα. Επίσης μόνο ο ιδιοκτήτης της διεργασίας ή ο root μπορεί να αλλάξει τη συνάφεια.

Επίσης υπάρχουν οι ακόλουθες μακροεντολές η οποίες διευκολύνουν τη διαχείριση της μάσκας bit.

void CPU_CLR(int cpu, cpu_set_t *set):
αφαιρεί τον επεξεργαστή με αριθμό cpu από τη μάσκα set.

int CPU_ISSET(int cpu, cpu_set_t *set):
ελέγχει αν ο επεξεργαστής με αριθμό cpu έχει τεθεί στη μάσκα set.

void CPU_SET(int cpu, cpu_set_t *set):
προσθέτει τον επεξεργαστή με αριθμό cpu στη μάσκα set.

void CPU_ZERO(cpu_set_t *set):
μηδενίζει τη μάσκα set (αφαιρεί όλους τους επεξεργαστές).

Κεφάλαιο 4

Java και παραλληλισμός

Η Java είναι μια γλώσσα που υποστηρίζει εγγενώς τον παράλληλο προγραμματισμό, αφού η βασική βιβλιοθήκη υποστηρίζει τη δημιουργία νημάτων. Όμως η υποστήριξη των νημάτων εξαρτάται από το λειτουργικό σύστημα, οπότε θα αναλυθούν τα διάφορα μοντέλα υλοποίησης που υπάρχουν. Επίσης θα αναλυθεί ένα πακέτο που προστέθηκε πρόσφατα στη βασική βιβλιοθήκη και το οποίο διευκολύνει τον παράλληλο προγραμματισμό. Τέλος θα αναφερθεί το πρότυπο JOMP (Java Open MultiProcessing) που προσφέρει μια προγραμματιστική διεπαφή για αυτοματοποιημένο παραλληλισμό, καθώς ο προγραμματιστής δεν χρειάζεται να ασχοληθεί με τη διαχείριση των νημάτων.

4.1 Εικονική μηχανή Java και μοντέλο νημάτων

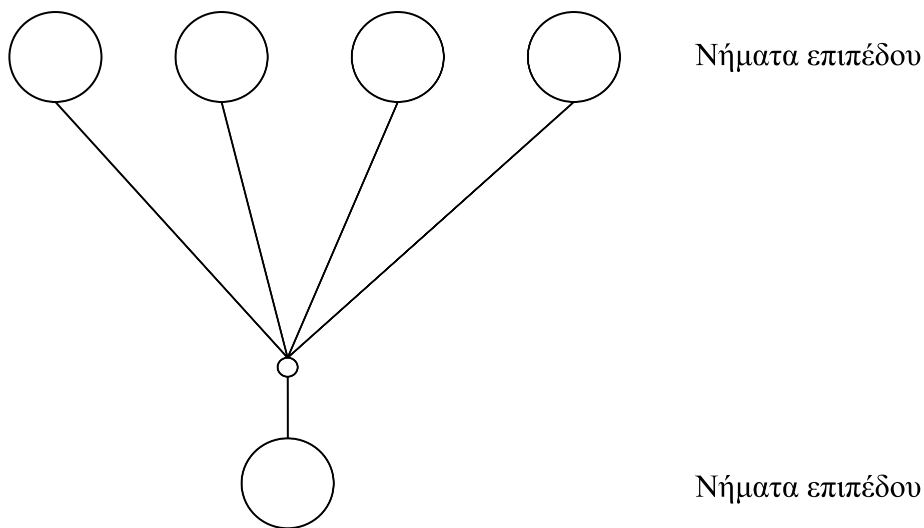
Τα προγράμματα σε γλώσσα Java εκτελούνται από την εικονική μηχανή Java (Java Virtual Machine - JVM) και όχι απευθείας από το λειτουργικό σύστημα. Η εικονική μηχανή αποκρύπτει επομένως τις λεπτομέρειες υλοποίησης του λειτουργικού συστήματος. Αυτό το χαρακτηριστικό κάνει τα προγράμματα γραμμένα σε Java μεταφέρσιμα σε διάφορες πλατφόρμες (όπως Windows, Linux), δηλαδή μπορούν να εκτελούνται χωρίς καμία μετατροπή στον κώδικα. Η βασική βιβλιοθήκη επιτρέπει την άμεση δημιουργία και διαχείριση νημάτων, αλλά αυτά υλοποιούνται από το λειτουργικό σύστημα που φιλοξενεί την εικονική μηχανή. Για παράδειγμα, σε Windows τα νήματα υλοποιούνται με τη χρήση του Win32 API (Application Programming Interface) και σε Linux από το Pthreads API.

Σε ένα λειτουργικό σύστημα υπάρχουν δύο επίπεδα νημάτων: τα νήματα χρήστη (user threads) και τα νήματα πυρήνα (kernel threads).Η διαχείριση των νημάτων χρήστη γίνεται από τον προγραμματιστή και η εκτέλεσή τους υποστηρίζεται από τον πυρήνα, ενώ τα νήματα πυρήνα τα διαχειρίζεται απευθείας το λειτουργικό σύστημα. Υπάρχουν τρεις τρόποι με τους οποίους σχετίζονται τα νήματα χρήστη με τα νήματα πυρήνα [11].

1. Μοντέλο Πολλά προς Ένα

Το μοντέλο πολλά προς ένα αντιστοιχίζει όλα τα νήματα επιπέδου χρήστη της εφαρμογής σε ένα νήμα επιπέδου πυρήνα (σχήμα 1). Αλλά μόνο ένα νήμα μπορεί να έχει πρόσβαση στον επεξεργαστή κάθε φορά και έτσι είναι αδύνατη η εκμετάλλευση της παρουσίας πολλών επεξεργαστών από την εφαρμογή. Επίσης ολόκληρη η διεργασία θα μπλοκαριστεί σε περίπτωση που ένα νήμα κάνει κλήση συστήματος με αναμονή. Κατά κάποιο τρόπο είναι προσομοίωση της εκτέλεσης πολλών νημάτων από την εικονική μηχανή, αφού στην

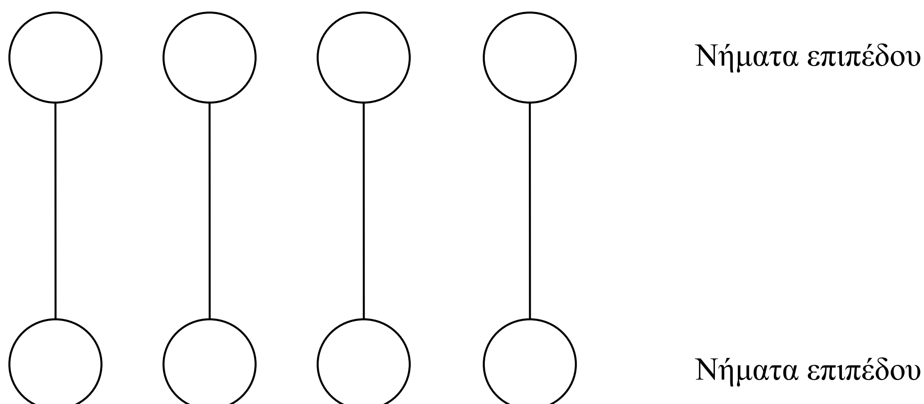
πραγματικότητα υπάρχει μόνο ένα νήμα. Άλλωστε, εμφανίστηκε και χρησιμοποιήθηκε στο παρελθόν, όταν επικρατούσαν οι επεξεργαστές ενός πυρήνα, οπότε δεν μπορούσε να υπάρξει παραλληλισμός ούτως ή άλλως. Αυτό το μοντέλο χρησιμοποιείται στο Solaris από τη βιβλιοθήκη Green Threads και από την βιβλιοθήκη Portable Threads του GNU.



Σχήμα 1. Πολλά προς Ένα

2. Μοντέλο Ένα προς Ένα

Το μοντέλο ένα προς ένα, όπως φαίνεται στο σχήμα 2, αντιστοιχίζει κάθε νήμα χρήστη σε ένα νήμα πυρήνα. Παρέχει παραλληλισμό, αφού είναι δυνατή η εκτέλεση πολλών νημάτων σε πολλούς επεξεργαστές. Όταν ένα νήμα πραγματοποιήσει μια κλήση συστήματος με αναμονή, τότε επιτρέπεται να εκτελεστεί κάποιο άλλο. Το μόνο αρνητικό σε αυτό το μοντέλο είναι ότι επιβραδύνεται η εφαρμογή, αφού η δημιουργία ενός νήματος χρήστη απαιτεί τη δημιουργία ενός νήματος πυρήνα. Για αυτό κάποιες υλοποιήσεις περιορίζουν το πλήθος των νημάτων που μπορεί να υποστηρίξει το σύστημα. Τα λειτουργικά συστήματα Windows και Linux υλοποιούν αυτό το μοντέλο.

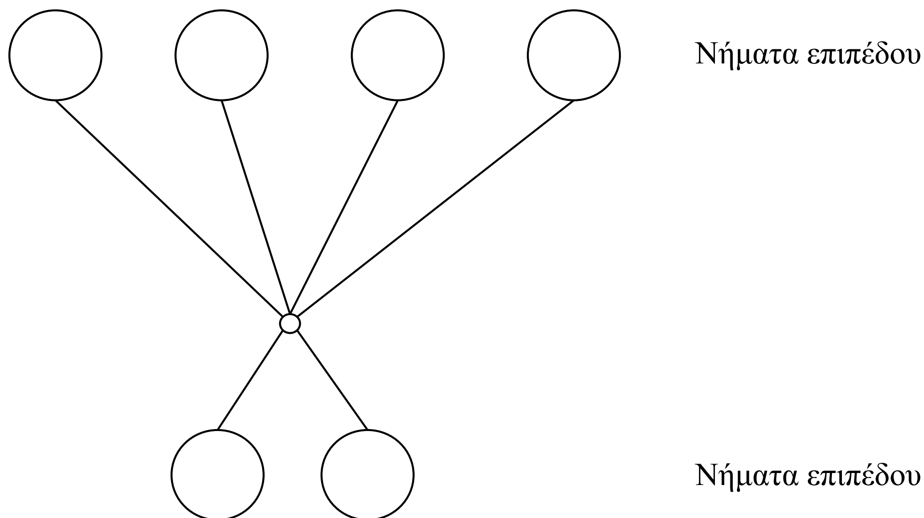


Σχήμα 2. Ένα προς Ένα

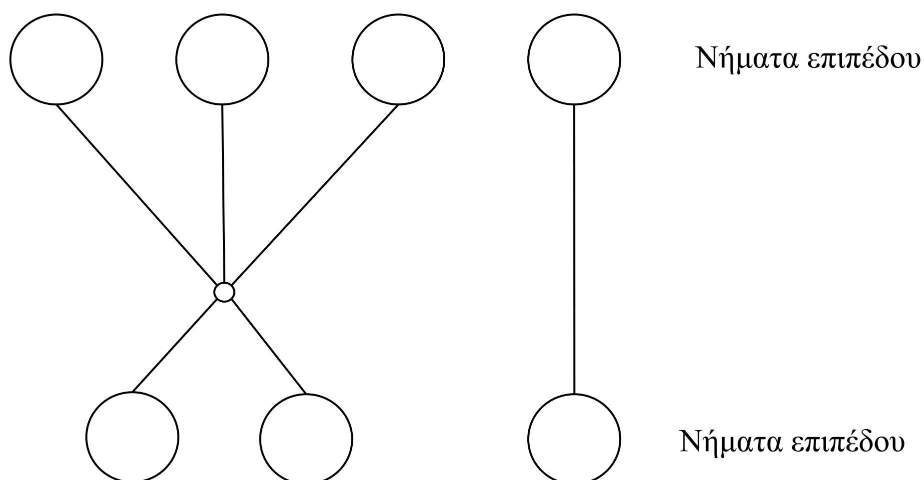
3. Μοντέλο Πολλά προς Πολλά

Το μοντέλο πολλά προς πολλά περιπλέκει πολλά νήματα επιπέδου χρήστη σε ένα μικρότερο ή ίσο αριθμό νημάτων επιπέδου πυρήνα, όπως φαίνεται στο σχήμα 3. Το πλήθος των νημάτων πυρήνα μπορεί να είναι καθορισμένο για μια συγκεκριμένη εφαρμογή ή για ένα συγκεκριμένο σύστημα (ανάλογα με τον αριθμό των διαθέσιμων επεξεργαστών). Αν και σε αυτό το μοντέλο δεν μπορεί να επιτευχθεί πραγματικός παραλληλισμός, ο προγραμματιστής μπορεί να δημιουργήσει όσα νήματα χρήστη χρειάζεται και τα αντίστοιχα νήματα πυρήνα μπορούν να εκτελεστούν παράλληλα σε ένα πολυεπεξεργαστικό σύστημα. Επίσης, όταν ένα νήμα πραγματοποιεί μια κλήση συστήματος με αναμονή, ο πυρήνας μπορεί να χρονοπρογραμματίσει ένα άλλο νήμα για εκτέλεση.

Μια παραλλαγή του μοντέλου πολλά προς πολλά επίσης πολυπλέκει πολλά νήματα επιπέδου χρήστη σε μικρότερο ή ίσο αριθμό νημάτων πυρήνα, αλλά επιπλέον επιτρέπει σε ένα νήμα επιπέδου χρήστη να δεσμεύει ένα νήμα πυρήνα όπως φαίνεται στο σχήμα 4. Η παραλλαγή αυτή αναφέρεται και ως μοντέλο δύο επιπέδων. Το Solaris έως πριν την έκδοση 9 υποστήριζε το μοντέλο των δύο επιπέδων. Με αυτό το μοντέλο μπορούσε να υπάρξει μια απομίμηση του μοντέλου ένα προς ένα, αν προσδεθεί κάθε νήμα χρήστη σε ένα νήμα πυρήνα. Από το Solaris 9 και μετά υποστηρίζεται το μοντέλο ένα προς ένα.



Σχήμα 3. Πολλά προς Πολλά



Σχήμα 4. Μοντέλο δύο επιπέδων

Η προδιαγραφή της εικονικής μηχανής Java δεν υποδεικνύει τον τρόπο που τα νήματα της Java θα αντιστοιχίζονται στο λειτουργικό σύστημα, αφήνοντας την απόφαση στους σχεδιαστές της εκάστοτε υλοποίησης. Το Solaris αρχικά υλοποιούσε τα νήματα Java με το μοντέλο πολλά προς ένα (Green Threads). Αργότερα υιοθετήθηκε το μοντέλο πολλά προς πολλά, ενώ από την έκδοση 9 το ένα προς ένα. Το μοντέλο ένα προς ένα χρησιμοποιείται από Windows και Linux

4.2 Υποστήριξη παραλληλισμού από την βασική βιβλιοθήκη της Java

Η Java είναι μια γλώσσα προγραμματισμού στην οποία ο παράλληλος προγραμματισμός είναι αρκετά εύκολος, αφού είναι ενσωματωμένη η υποστήριξη νημάτων (κλάση Thread και διεπαφή Runnable). Όμως οι βασικές παροχές, όπως συγχρονισμός σε επίπεδο μπλοκ, με τη λέξη κλειδί synchronized, και οι μέθοδοι Object.wait() και Object.notify() είναι ανεπαρκείς για πολύπλοκες εφαρμογές. Για αυτό το λόγο από την έκδοση J2SE 5.0 της Java παρέχεται το πακέτο java.util.concurrent το οποίο προσφέρει ένα σύνολο κλάσεων και διεπαφών για να διευκολυνθεί το έργο της ανάπτυξης πολυνηματικών εφαρμογών. Το πακέτο αυτό ακολουθεί το πρότυπο JSR 166: Concurrency Utilities [], όπως ορίστηκε από την Java Community Process. Η παρεχόμενη λειτουργικότητα μπορούσε να επιτευχθεί και στο παρελθόν, αλλά ο σκοπός είναι να δοθούν υψηλής ποιότητας υλοποιήσεις που διευκολύνουν τη λύση στο πρόβλημα του συγχρονισμού. Επομένως ο προγραμματιστής χρησιμοποιώντας την βασική βιβλιοθήκη γίνεται παραγωγικότερος και αποφεύγει πιθανόν λανθασμένες και αναποτελεσματικές υλοποιήσεις. Η χρήση του πακέτου αυτού βοηθάει ώστε ο κώδικας να γίνει πιο σύντομος και σαφής, αξιόπιστος, αποδοτικός, επαναχρησιμοποιήσιμος και ευκολότερα συντηρήσιμος [18].

Το πακέτο αυτό προσφέρει:

- *Παράλληλες συλλογές δεδομένων:* Έχουν προστεθεί νέες παράλληλες υλοποιήσεις κλάσεων και διεπαφών, όπως ουρών, λιστών και χαρτών.

- *Ατομικές μεταβλητές:* Το πακέτο java.util.concurrent.atomic παρέχει κλάσεις για διαχείριση απλών μεταβλητών που μπορεί να είναι πρωτογενείς τύποι ή αναφορές (πχ AtomicInteger, AtomicBoolean, AtomicReference). Οι κλάσεις αυτές επιτρέπουν σε πολλές λειτουργίες να εκτελεστούν ατομικά (δηλαδή χωρίς να διακοπούν ενδιάμεσα) με αυτόματο τρόπο, δηλαδή χωρίς τη χρήση κάποιου κλειδώματος συγχρονισμού. Χαρακτηριστικά παραδείγματα μεθόδων που διαθέτουν αυτές οι κλάσεις είναι η `getAndSet(int`

newValue), η οποία θέτει στη μεταβλητή μια τιμή και επιστρέφει την προηγούμενη τιμή. Επίσης η μέθοδος compareAndSet(int expect, int update) συγκρίνει την τιμή της μεταβλητής και αν είναι ίση με την αναμενόμενη, τότε θέτει την νέα τιμή. Ομοίως υπάρχουν και άλλες μέθοδοι, ανάλογα με τον τύπο της κλάσης, όπως για την AtomicInteger η addAndGet(int delta) προσθέτει στη μεταβλητή μια τιμή και επιστρέφει τη νέα τιμή, ενώ η getAndAdd(int delta) προσθέτει την τιμή και επιστρέφει την παλαιά τιμή.

- *Συγχρονισμός* : Παρέχονται γενικού επιπέδου κλάσεις για συγχρονισμό, οι οποίες διευκολύνουν το συντονισμό μεταξύ νημάτων. Οι κλάσεις αυτές είναι οι Semaphore, CyclicBarrier , CountDownLatch και Exchanger.

- *Κλειδώματα (Locks)*: Το πακέτο java.util.concurrent.locks παρέχει υλοποιήσεις κλειδωμάτων με κάποιες προηγμένες λειτουργίες που παρέχουν περισσότερη ευελιξία. Μια σημαντική προσθήκη είναι τα κλειδώματα ανάγνωσης/εγγραφής, τα οποία βασίζονται στο γεγονός ότι όταν ένα νήμα διαβάζει την τιμή μιας μεταβλητής, δεν χρειάζεται να απαγορεύεται η πρόσβαση σε άλλα νήματα που θέλουν να διαβάσουν αυτή την τιμή χωρίς να την τροποποιήσουν. Για παράδειγμα, η διεπαφή ReadWriteLock έχει δύο μεθόδους, τις readLock() και writeLock(), όπου η πρώτη επιτρέπει πολλαπλή πρόσβαση ταυτόχρονα, ενώ η δεύτερη απαιτεί αποκλειστική πρόσβαση. Μια άλλη προσθήκη είναι ότι παρέχεται η δυνατότητα υπαναχώρησης από μια προσπάθεια απόκτησης του κλειδώματος. Η μέθοδος tryLock αποκτά το κλείδωμα αν είναι διαθέσιμο και επιστρέφει true, αλλιώς επιστρέφει false. Επίσης μπορεί να καθοριστεί ένα χρονικό διάστημα μέσα στο οποίο θα προσπαθήσει να αποκτήσει το κλείδωμα.

- *Ένα πλαίσιο (framework) εκτέλεσης εργασιών*: με το πλαίσιο Executor προσφέρεται ένας τυποποιημένος τρόπος κλήσης, χρονοπρογραμματισμού, εκτέλεσης και ελέγχου μιας ασύγχρονης εργασίας σύμφωνα με ένα σύνολο πολιτικών εκτέλεσης.

Ο σκοπός της δημιουργίας των Executors είναι να διαχωριστεί η δημιουργία νημάτων από την διαχείρισή τους. Υπάρχει η κλάση Executor και οι διεπαφές Executor, ExecutorService, ScheduledExecutorService που υποστηρίζουν την εκτέλεση νέων εργασιών.[19] Έτσι αν έχουμε ένα αντικείμενο r το οποίο είναι Runnable και ένα αντικείμενο e που υλοποιεί κάποια από τις παραπάνω διεπαφές, τότε η έκφραση

(new Thread(r)).start();

μπορεί να αντικατασταθεί με την

e.execute(r);

Ο τρόπος εκτέλεσης εξαρτάται από την υλοποίηση του αντικειμένου e. Οι εργασίες που καθορίζονται από το r μπορούν να εκτελεστούν εντός του νήματος που τις υποβάλλει, σε ένα νήμα παρασκηνίου, σε ένα νέο νήμα ή σε ένα νήμα από μια δεξαμενή νημάτων (Thread Pool). Οι περισσότερες υλοποιήσεις χρησιμοποιούν δεξαμενές νημάτων, οι οποίες ορίζονται επίσης σε αυτό το πακέτο.

Μια δεξαμενή νημάτων απαρτίζεται από πολλά νήματα –εργάτες. Αυτά τα νήματα υπάρχουν ανεξάρτητα από τις εργασίες Runnable και Callable (μια διεπαφή παρόμοια με την Runnable, αλλά επιστρέφει ένα αποτέλεσμα) και συνήθως εκτελούν πολλαπλές εργασίες. Η χρήση τους ελαχιστοποιεί τις καθυστερήσεις από τη δημιουργία νέων νημάτων και επίσης μειώνεται η χρήση μνήμης, αφού αποφεύγεται η δημιουργία αντικειμένων τύπου Thread.

Ένας συνηθισμένος τύπος δεξαμενής νημάτων είναι η σταθερή δεξαμενή νημάτων (fixed thread pool). Αυτός ο τύπος έχει πάντα τον ίδιο, συγκεκριμένο αριθμό νημάτων και αν κάποιο τερματιστεί, αντικαθίσταται με ένα νέο. Οι εργασίες υποβάλλονται στη δεξαμενή σε μια εσωτερική ουρά, η οποία διατηρεί τις επιπλέον εργασίες, όταν είναι περισσότερες από τον αριθμό των νημάτων. Ένα πλεονέκτημα αυτού του μηχανισμού είναι ότι η εφαρμογή δεν φτάνει σε κορεσμό των πόρων. Έστω ότι για παράδειγμα έχουμε έναν διαδικτυακό εξυπηρετητή, όπου κάθε αίτηση των χρηστών την χειρίζεται ένα νέο νήμα. Αν δημιουργούνται συνεχώς νήματα και το σύστημα λάβει περισσότερες αιτήσεις από όσες μπορεί να εξυπηρετήσει, τότε θα σταματήσει εντελώς να ανταποκρίνεται. Ενώ με ένα όριο στον αριθμό νημάτων, η εφαρμογή θα ανταποκρίνεται με το ρυθμό που μπορεί να αντέξει το σύστημα και όχι με το ρυθμό που φτάνουν οι εισερχόμενες εργασίες.

Η δημιουργία μιας σταθερής δεξαμενής νημάτων γίνεται μέσω της κλάσης Executors, η οποία παρέχει αυτή τη λειτουργικότητα με τη μέθοδο newFixedThreadPool. Επίσης παρέχει και άλλα μοντέλα. Η μέθοδος newCachedThreadPool δημιουργεί μια επεκτάσιμη δεξαμενή νημάτων και είναι κατάλληλη για εφαρμογές με πολλά νήματα μικρής διάρκειας ζωής. Η μέθοδος newSingleThreadExecutor δημιουργεί έναν executor, ο οποίος έχει μόνο ένα νήμα-εργάτη, άρα εκτελεί μόνο μια εργασία τη φορά.

Επιπλέον έχει προστεθεί η μέθοδος *java.lang.System.nanotime( )* που παρέχει υψηλότερη ακρίβεια, αν και η υλοποίησή της εξαρτάται από την πλατφόρμα

4.3 JOMP

4.3.1 Εισαγωγή

Το OpenMP είναι ένα πρότυπο για παράλληλο προγραμματισμό κοινόχρηστης μνήμης . Το πρότυπο αυτό ορίζει ένα σύνολο οδηγιών για τη C/C++ και τη Fortran και παρέχει μια υψηλού επιπέδου αφαιρετικότητα στον προγραμματιστή. Ο προγραμματιστής απλά καθορίζει ποιες περιοχές θα εκτελεστούν παράλληλα, χωρίς όμως να ασχολείται απευθείας με τον προγραμματισμό νημάτων, όπως τη δημιουργία τους, το συγχρονισμό τους ή την επικοινωνία μεταξύ τους. Ουσιαστικά ο κώδικας γράφεται σειριακά και οι οδηγίες (directives) προστίθενται υπό μορφή σχολίων. Κατά τη μεταγλώττιση από έναν συμβατό μεταγλωττιστή, τα σχόλια διαβάζονται και ο κώδικας παράγεται έτσι, ώστε να χρησιμοποιήσει νήματα σύμφωνα με τις οδηγίες.

Το JOMP είναι ένα αντίστοιχο πρότυπο για την Java [20]. Φυσικά είναι δυνατό να γραφεί ένα παράλληλο πρόγραμμα κοινόχρηστης μνήμης χρησιμοποιώντας τις ενσωματωμένες δυνατότητα της Java. Ωστόσο, σε σχέση με αυτή την προσέγγιση, το σύστημα με τις οδηγίες παρουσιάζει ένα σύνολο πλεονεκτημάτων. Καταρχάς, ο κώδικας είναι πολύ πιο κοντά στη σειριακή έκδοση του ίδιου προγράμματος, το οποίο κάνει την ανάπτυξη και τη συντήρηση ευκολότερη. Επίσης, ένα πρόγραμμα μπορεί να εκτελεστεί σειριακά σωστά, όταν οι οδηγίες αγνοηθούν, κατά την μεταγλώττιση από έναν παραδοσιακό μεταγλωττιστή. Άρα το πρότυπο αυτό είναι κατάλληλο για μετατροπή σε παράλληλο ήδη υπάρχοντα σειριακού κώδικα. Ένα άλλο πλεονέκτημα είναι ότι για μέγιστη απόδοση πρέπει να χρησιμοποιηθεί ένα νήμα ανά επεξεργαστή και να διατηρηθούν αυτά τα νήματα σε εκτέλεση για όλη τη διάρκεια ζωής του προγράμματος. Ο προκαθορισμένος αριθμός νημάτων εκτέλεσης στις παράλληλες περιοχές ισούται με τον αριθμό των επεξεργαστών του συστήματος. Μια επιπρόσθετη σημαντική λειτουργικότητα, την οποία παρέχει το σύστημα των οδηγιών, είναι η αυτόματη παράλληλη εκτέλεση βρόχων. Δηλαδή οι επαναλήψεις ενός βρόχου κατανέμονται αυτόματα σε νήματα και εκτελούνται παράλληλα.

Η υλοποίηση του JOMP παρέχει μια βιβλιοθήκη κλάσεων και έναν μεταγλωττιστή, ο ο οποίος είναι γραμμένος σε Java. Ο κώδικας γράφεται σε αρχείο με κατάληξη .jomp και ο μεταγλωττιστής παράγει ένα αρχείο .java, το οποίο μεταγλωττίζεται και εκτελείται κατά τον παραδοσιακό τρόπο.

4.3.2 Η προγραμματιστική διεπαφή

Η προγραμματιστική διεπαφή του JOMP βασίζεται στο προϋπάρχον πρότυπο OpenMP για τις C/C++. Στις C/C++ οι οδηγίες απευθύνονται στον προεπεξεργαστή, αλλά αφού η Java δεν διαθέτει προεπεξεργαστή, οι οδηγίες δίνονται σαν σχόλια της μορφής //omp <οδηγία>.

Οι διαθέσιμες οδηγίες είναι:

1.Only

Η περιοχή του κώδικα που σηματοδοτείται με την οδηγία only εκτελείται μόνο αν μεταγλωττιστεί από έναν συμβατό με το JOMP μεταγλωττιστή

```
//omp only <statement>
```

2.Parallel

Όταν ένα νήμα συναντήσει αυτή την εντολή, δημιουργείται μια νέα ομάδα νημάτων, αν η συνθήκη είναι αληθής (η έκφραση if είναι προαιρετική). Κάθε νήμα της ομάδας εκτελεί το μπλοκ κώδικα. Οι εκφράσεις default, shared, private, firstprivate, reduction έχουν σχέση με την ορατότητα των μεταβλητών.

```
//omp parallel [if(<condition>)]
//omp [default (shared|none)][shared(<vars>)]
//omp [private(<vars>)][firstprivate(<vars>)]
//omp [reduction(<operation>:<vars>)]
<Java code block>
```

3. for

Η οδηγία αυτή επιτρέπει στις επαναλήψεις ενός βρόχου να μοιραστούν σε νήματα και να εκτελεστούν παράλληλα. Οι επιλογές για την πρόταση schedule είναι μια από τις static, dynamic, guided, runtime. Η επιλογή ordered εξασφαλίζει ότι το μπλοκ που ακολουθεί θα εκτελεστεί σε κάθε επανάληψη με τη σειρά που θα εκτελούταν κατά τη διάρκεια της σειριακής εκτέλεσης.

```
//omp for [nowait] [reduction (<operator>:<vars>)]
//omp [schedule(<mode>,[chunk-size])] [ordered]
<for-loop>
```

4. sections

Ο κώδικας χωρίζεται σε περιοχές (sections) που μπορούν να εκτελεστούν παράλληλα. Κάθε περιοχή ανατίθεται σε ένα νήμα με βάση την πολιτική first-come-first-served.

```
//omp sections [nowait]
{
//omp section
<code block 1>
```

```
//omp section
<code block 2>
…
}
```

5. single

Το τμήμα κώδικα που σηματοδοτείται από την οδηγία single εκτελείται μόνο από ένα νήμα της ομάδας. Δηλαδή αν το αυτό το τμήμα κώδικα βρίσκεται μέσα σε μια παράλληλη περιοχή, θα εκτελεστεί από το πρώτο νήμα που θα συναντήσει αυτή την οδηγία.

```
//omp single [nowait]
<code block>
```

6. master

Το τμήμα κώδικα εκτελείται μόνο από το κυρίως νήμα μιας ομάδας (το νήμα με αριθμό 0).

```
//omp master
<code block>
```

7. critical

Η οδηγία αυτή επιτρέπει σε ένα τμήμα κώδικα, στο οποίο δίνεται ένα όνομα, να εκτελείται μόνο από ένα νήμα κάθε στιγμή. Οι περιοχές χωρίς όνομα θεωρούνται ότι έχουν το ίδιο όνομα, null. Έτσι όταν ένα νήμα φτάσει σε αυτή την οδηγία περιμένει να αποκτήσει το κλείδωμα με βάση το όνομα.

```
//omp critical [name]
<code block>
```

8. barrier

Η οδηγία αυτή αναγκάζει κάθε νήμα να περιμένει έως ότου όλα τα νήματα της τρέχουσας ομάδας φτάσουν σε αυτό το σημείο. Για αποφυγή αδιεξόδου πρέπει ή όλα τα νήματα ή κανένα να φτάσουν σε αυτό το σημείο.

```
//omp barrier
```

Για συντομία, μπορεί να χρησιμοποιηθεί η οδηγία //omp parallel for για μια παράλληλη περιοχή που περιέχει μόνο μια οδηγία για έναν παράλληλο βρόχο. Ομοίως η οδηγία //omp parallel section ισοδυναμεί με μια παράλληλη περιοχή που περιέχει μόνο μια οδηγία section.

Η βιβλιοθήκη του JOMP

Η βιβλιοθήκη του JOMP παρέχει στο χρήστη πολλές μεθόδους οι οποίες παρέχονται ως στατικές μέθοδοι της κλάσης jomp.runtime.OMP και έχουν σχέση με τη διαχείριση των νημάτων.

Αυτές είναι:

getNumThreads():επιστρέφεται ο αριθμός των νημάτων της ομάδας που εκτελούνται στην τρέχουσα παράλληλη περιοχή ή 1 αν κληθεί από σειριακή περιοχή.

setNumThreads(n):θέτει σε n τον αριθμό των νημάτων που μπορούν να χρησιμοποιηθούν σε παράλληλη περιοχή. Για να έχει αποτέλεσμα πρέπει να κληθεί από σειριακή περιοχή.

getMaxThreads():επιστρέφεται ο μέγιστος αριθμός νημάτων που θα χρησιμοποιηθεί σε μια παράλληλη περιοχή, υποθέτοντας ότι δεν παρεμβάλλεται κλήση στην setNumThreads().

getThreadNum():επιστρέφει τον αριθμό εντός της ομάδας του καλούντος νήματος. Αν είναι το κύριο νήμα είναι 0. Επίσης από σειριακή περιοχή επιστρέφει 0.

getNumProcs() :αριθμός διεργασιών που έχουν ανατεθεί στο πρόγραμμα.

inParallel():επιστρέφει true αν κληθεί από παράλληλη περιοχή, ακόμα και να υπάρχει ένα νήμα και false αν κληθεί από σειριακή περιοχή.

setDynamic():ενεργοποιεί/απενεργοποιεί την αυτόματη προσαρμογή του αριθμού των νημάτων. Αν έχει ενεργοποιηθεί η *getDynamic()* επιστρέφει true, αλλιώς false .

setNested():ενεργοποιεί/απενεργοποιεί την εμφώλευση πολλών παράλληλων περιοχών. Αν έχει ενεργοποιηθεί, η *getNested()* επιστρέφει true, αλλιώς false.

Επίσης παρέχονται οι κλάσεις jomp.runtime.Lock και jomp.runtime.NestLock οι οποίες υλοποιούν ένα κλείδωμα. Η διαφορά τους έγκειται στη συμπεριφορά τους όταν το νήμα που κατέχει το κλείδωμα το ζητήσει πάλι. Η Lock προκαλεί αδιέξοδο, ενώ η NestedLock θα επιτύχει την ανάκτηση του κλειδώματος. Κάθε κλάση έχει τρεις μεθόδους:

set():προσπαθεί να αποκτήσει το κλείδωμα. Αν το κατέχει άλλο νήμα, το καλόν νήμα μπλοκάρεται μέχρι να απελευθερωθεί το κλείδωμα.

unset(): απελευθερώνει το κλείδωμα.

test(): ελέγχει αν είναι δυνατόν να αποκτηθεί το κλείδωμα αμέσως χωρίς μπλοκάρισμα. Αν είναι διαθέσιμο, αποκτάται και επιστρέφει true, αλλιώς επιστρέφει false.

Τέλος παρέχονται κάποιες επιλογές στη μορφή ιδιοτήτων συστήματος της Java.

jomp.schedule: ορίζει τη στρατηγική χρονοπρογραμματισμού και παίρνει τις ίδιες τιμές με την πρόταση schedule της οδηγίας for, όπως αναφέρθηκε παραπάνω

jomp.threads:ορίζει τον αριθμό νημάτων για παράλληλη εκτέλεση

jomp.dynamic: έχει τιμή true/false ανάλογα με τον έχει ενεργοποιηθεί ή απενεργοποιηθεί αντίστοιχα η αυτόματη προσαρμογή του αριθμού των νημάτων

jomp.nested: έχει τιμή true/false ανάλογα με τον έχει ενεργοποιηθεί ή απενεργοποιηθεί αντίστοιχα ο εμφωλευμένος παραλληλισμός.

4.3.3 Απλό παράδειγμα εφαρμογής

Παρατίθεται ο κώδικας μια εφαρμογής που χρησιμοποιεί το JOMP, η οποία υπολογίζει το π. Το πρόγραμμα δέχεται ως όρισμα τον αριθμό των νημάτων που θα χρησιμοποιηθούν. Ο κώδικας βρίσκεται στο αρχείο pi.jomp και αρχικά μεταγλωττίζεται από τον μεταγλωττιστή του JOMP για να παραχθεί το αρχείο pi.java ως εξής:

java -classpath jomp1.0b.jar jomp.compiler.Jomp pi

Ύστερα μεταγλωττίζεται με τον παραδοσιακό τρόπο, ώστε να παραχθεί το αρχείο pi.class:

javac -classpath jomp1.0b.jar pi.java

Για να εκτελεστεί για παράδειγμα με 4 νήματα δίνουμε:

java pi 4

```
import jomp.runtime.*;

public class pi {

	public static void main(String args[]){

		int i, num_steps = 100000;
		double pi, x, step, sum=0.0;
		step = 1.0/(double) num_steps;
		int n=Integer.parseInt(args[0]);
		OMP.setNumThreads(n);
		System.out.println("Number of Threads:"+n);
		int [] counter=new int[n];

		for(i=0;i<n;i++){
			counter[i]=0;
		}


		//omp parallel for private(x) reduction(+:sum)
		for(i=0;i<num_steps;i++){
			x = (i+0.5)*step;
			sum = sum + 4.0/(1.0+x*x);
			counter[OMP.getThreadNum()]++;
		}


		pi = step * sum;
		System.out.println("PI = "+pi);

		for(i=0;i<n;i++){
			System.out.println(i+" "+counter[i]);
		}

	}

}
```

Παρατηρούμε ότι ο κώδικας είναι πολύ απλός, αφού δεν χρειάζεται ο προγραμματιστής να ασχοληθεί ρητά με τη δημιουργία και διαχείριση των νημάτων. Ο καταμερισμός των επαναλήψεων σε νήματα γίνεται αυτόματα, όπως αυτόματα γίνεται ο συγχρονισμός κατά τη χρήση κοινών μεταβλητών, όπως εδώ η sum. Ο αριθμός των νημάτων, ο οποίος δίνεται ως παράμετρος κατά την εκτέλεση, και οι μετρητές (πίνακας counter) υπάρχουν μόνο για διευκόλυνση κατά τη δοκιμή του κώδικα. Φαίνεται επίσης ότι το πρόγραμμα μπορεί να τρέξει σειριακά σε περίπτωση μη ύπαρξης συμβατού μεταγλωττιστή με το JOMP, αφού οι οδηγίες είναι σχόλια που θα αγνοηθούν.

Επομένως η χρήση του JOMP διευκολύνει πάρα πολύ τον παράλληλο προγραμματισμό, προπαντός όταν απαιτείται η δημιουργία πολλών νημάτων ίδιου τύπου, δηλαδή όταν εκτελούν τον ίδιο κώδικα, με σκοπό την ταχύτερη διεκπεραίωση μιας εργασίας. Μια κατηγορία εφαρμογών που ευνοούνται αρκετά από τη χρήση του OpenMP είναι επιστημονικές εφαρμογές που περιλαμβάνουν πολλούς αριθμητικούς υπολογισμούς, οι οποίοι συνήθως έχουν μεγάλους βρόχους επανάληψης. Επίσης το πρότυπο αυτό είναι κατάλληλο για εύκολη παραλληλοποίηση ήδη υπάρχοντος σειριακού κώδικα, χωρίς να χρειαστούν μεγάλες αλλαγές. [21]

Για ανάπτυξη μεγάλων εφαρμογών που έχουν πολλά νήματα διαφορετικού τύπου, όπως το παράδειγμα του Festo MPS, που εξετάζεται στο κεφάλαιο 6, η χρησιμότητα του κρίνεται περιορισμένη.

Κεφάλαιο 5

Real Time Java

Η χρήση της Java σε συστήματα πραγματικού χρόνου (real-time) δεν είναι διαδεδομένη για διάφορους σημαντικούς λόγους. Αυτοί περιλαμβάνουν την απρόβλεπτη συμπεριφορά που οφείλεται στο σχεδιασμό της γλώσσας, όπως η δυναμική φόρτωση κλάσεων, ο garbage collector, η μεταγλώττιση σε native κώδικα . Το Real-time Specification for Java - RTSJ (Προδιαγραφές Πραγματικού Χρόνου για την Java) είναι ένα ανοιχτό πρότυπο που ενισχύει τη Java, ώστε να γίνει κατάλληλη για ανάπτυξη συστημάτων πραγματικού χρόνου.[22] Στο κεφάλαιο αυτό αρχικά προσδιορίζεται η έννοια των απαιτήσεων πραγματικού χρόνου και στη συνέχεια παρουσιάζονται τα προβλήματα της κλασσικής Εικονικής Μηχανής Java και πώς αντιμετωπίζονται από το πρότυπο RTSJ.

5.1 Απαιτήσεις Πραγματικού Χρόνου

Ο πραγματικός χρόνος είναι ένας όρος που περιγράφει εφαρμογές που έχουν απαιτήσεις στον χρόνο του πραγματικού κόσμου. Για παράδειγμα μια διεπαφή χρήστη έχει κάποιες χαλαρές απαιτήσεις πραγματικού χρόνου (soft RT application), αφού μπορούμε να πούμε ότι η εφαρμογή πρέπει να ανταποκρίνεται σε ένα κλικ του ποντικιού το πολύ σε 0,1 δευτερόλεπτα. Βέβαια αν αυτό δεν συμβεί, έχουμε μια μικρή αποτυχία, καθώς η εφαρμογή θα συνεχίσει να λειτουργεί, απλά ο χρήστης πιθανόν να είναι δυσαρεστημένος. Αντιθέτως, οι εφαρμογές που πρέπει να ικανοποιούν τις προθεσμίες τους οπωσδήποτε με αυστηρότητα λέγονται αυστηρού πραγματικού χρόνου (hard RT applications). Μια εφαρμογή ελέγχου σε αεροπλάνο δεν πρέπει να καθυστερεί για κανένα λόγο, αλλιώς το αποτέλεσμα μπορεί να είναι καταστροφικό. Επομένως μια εφαρμογή χαρακτηρίζεται ως πραγματικού χρόνου ανάλογα με το αν μπορεί να ανεχθεί απώλεια των προθεσμιών της. [22]

Σε ένα σύστημα πραγματικού χρόνου ο σκοπός είναι η εφαρμογή να εκτελέσει κάποιες εργασίες μέχρι μια προθεσμία. Δηλαδή αυτό που είναι σημαντικό είναι να υπάρχει προβλέψιμη συμπεριφορά (deterministic behaviour) και όχι απαραίτητα μεγάλη ταχύτητα στην εκτέλεση, όπως λανθασμένα θεωρείται μερικές φορές .

Είναι πολύ σημαντικό για τους προγραμματιστές που γράφουν εφαρμογές είτε αυστηρού είτε χαλαρού πραγματικού χρόνου να καταλάβουν τον περιορισμό που έχουν στον χρόνο απόκρισης. Οι τεχνικές που απαιτούνται για απόκριση σε 1 ms είναι αρκετά διαφορετικές από αυτές για απαιτήσεις 100 ms. Στην πράξη, ικανοποίηση απαιτήσεων της τάξης μερικών δεκάδων μs απαιτούν ένα συνδυασμό υλικού και λογισμικού ειδικού σκοπού, πιθανόν φτιαγμένου κατά παραγγελία (custom) και χωρίς την παρεμβολή ενός λειτουργικού συστήματος.

Τέλος, οι σχεδιαστές σθεναρών εφαρμογών πραγματικού χρόνου συνήθως χρειάζονται κάποιο βαθμό προβλέψιμης συμπεριφοράς για να σχεδιάσουν την εφαρμογή ώστε να ικανοποιεί τους χρόνους απόκρισης. Εάν παρουσιάζονται σημαντικές διακυμάνσεις στην απόδοση, έτσι ώστε να επηρεάζεται η ικανότητα του συστήματος να ικανοποιήσει την προθεσμία του, ο σχεδιασμός εφαρμογών πραγματικού χρόνου γίνεται υπερβολικά δύσκολος έως αδύνατος. . Οι σχεδιαστές των περισσότερων περιβαλλόντων εκτέλεσης τέτοιων εφαρμογών καταβάλλουν σημαντικές προσπάθειες ώστε να μειώσουν τέτοια φαινόμενα, για να ικανοποιήσουν το χρόνο απόκρισης σε όσο το δυνατόν ευρύτερο φάσμα προβλημάτων πραγματικού χρόνου.

5.2 Προβλήματα της Java στις απαιτήσεις πραγματικού χρόνου

Οι παραδοσιακές εφαρμογές java τρέχουν σε μια γενικού σκοπού Java Virtual Machine – JVM (εικονική μηχανή Java) σε γενικού σκοπού λειτουργικά συστήματα και μπορούν να ικανοποιήσουν χαλαρές απαιτήσεις μερικών εκατοντάδων ms, αλλά δεν είναι κατάλληλες για αυστηρές απαιτήσεις πραγματικού χρόνου . Πολλά θεμελιώδη χαρακτηριστικά της γλώσσας είναι υπεύθυνα για την απόδοση της εικονικής μηχανής: διαχείριση νημάτων (thread management), φόρτωση κλάσεων (class loading), Just-in-time (JIT) compilation , και συλλογή απορριμμάτων (garbage collection - GC) [22]. Στη συνέχεια γίνεται μια αναφορά στα σημαντικότερα από αυτά:

Διαχείριση νημάτων

Η κλασική Java δεν παρέχει εγγυήσεις για τον χρονοπρογραμματισμό των νημάτων με σεβασμό στην προτεραιότητά τους. Μια εφαρμογή που πρέπει να αποκρίνεται σε γεγονότα σε ένα καλώς καθορισμένο χρόνο δεν έχει τρόπο να εξασφαλίσει ότι ένα άλλο νήμα χαμηλότερης προτεραιότητας δεν θα προγραμματιστεί για εκτέλεση πριν από ένα υψηλότερης προτεραιότητας.

Φόρτωση κλάσεων

Η Εικονική Μηχανή Java κανονικά φορτώνει μια κλάσης όταν βρεθεί η πρώτη αναφορά σε αυτή στο πρόγραμμα. Η φόρτωση της κλάσης διαρκεί μεταβλητό χρόνο που εξαρτάται κυρίως από το αποθηκευτικό μέσο που φορτώνεται (συνήθως ο σκληρός δίσκος) και από το μέγεθός της. Η καθυστέρηση αυτή μπορεί να φτάσει στις περισσότερες περιπτώσεις ως και τα 10 ms []. Αν χρειάζεται να φορτωθούν πολλές κλάσεις, ο χρόνος αυτός είναι μια σημαντική και απρόβλεπτη καθυστέρηση. Με προσεκτικό σχεδιασμό της εφαρμογής είναι δυνατόν όλες οι κλάσεις να φορτωθούν κατά την εκκίνηση, αλλά αυτό πρέπει να γίνει χειροκίνητα, αφού το πρότυπο της Java δεν επιτρέπει στην Εικονική Μηχανή να εκτελέσει αυτό το βήμα πιο νωρίς,

Συλλογέας απορριμμάτων (Garbage Collector -GC)

Τα πλεονεκτήματα του GC στην ανάπτυξη εφαρμογών είναι πολύ σημαντικά, όπως αποφυγή διαρροής μνήμης, ασφάλεια δεικτών, απελευθέρωση του προγραμματιστή από την ανάγκη να απελευθερώνει μόνος του τη μνήμη. Όμως, ο GC είναι μια ακόμα πηγή προβλημάτων για τους προγραμματιστές εφαρμογών αυστηρού πραγματικού χρόνου που χρησιμοποιούν την Java. Η συλλογή απορριμμάτων συμβαίνει αυτόματα όταν η μνήμη heap εξαντληθεί και δεν μπορεί να γίνει άλλη ανάθεση. Επίσης η ίδια η εφαρμογή μπορεί να ενεργοποιήσει την εκτέλεση του GC.

Από την μια πλευρά, ο GC είναι πολύ χρήσιμος, γιατί παρέχει προστασία από δύσκολα σε διάγνωση λάθη που προέκυπταν από τη διαχείριση μνήμης από τον προγραμματιστή σε γλώσσες όπως C και C++ . Τη μνήμη διαχειρίζεται η Εικονική Μηχανή Java και όχι ο προγραμματιστής και αυτό είναι ένα μεγάλο πλεονέκτημα του μοντέλου της Java.

Από την άλλη πλευρά, ο παραδοσιακός garbage collector εισάγει μεγάλες καθυστερήσεις σε σημεία που είναι αδύνατον να προβλεφθούν. Καθυστερήσεις μερικών εκατοντάδων ms είναι κάτι αναμενόμενο. Ο μόνος τρόπος να λυθεί αυτό το πρόβλημα στο επίπεδο της εφαρμογής είναι να αποτρέψουμε την συλλογή, δημιουργώντας αντικείμενα που επαναχρησιμοποιούνται, οπότε εξασφαλίζουμε ότι η heap μνήμη ποτέ δεν θα εξαντληθεί. Με άλλα λόγια οι προγραμματιστές αντιμετώπισαν το πρόβλημα, παρακάμπτοντας την αυτόματη διαχείριση μνήμης και των πλεονεκτημάτων της, εφαρμόζοντας μια ρητή διαχείριση μνήμης . Στην πράξη αυτή η μεθοδολογία αποτυγχάνει, γιατί αποτρέπει τους προγραμματιστές από το να χρησιμοποιήσουν πολλές κλάσεις της βιβλιοθήκης του JDK ή τρίτων, οι οποίες πιθανόν να δημιουργούν προσωρινά αντικείμενα, τα οποία στο τέλος θα γεμίσουν τη μνήμη.

Συνήθως οι συλλογές απορριμμάτων πραγματοποιούνται ενώ η εφαρμογή του προγράμματος έχει σταματήσει, μια διαδικασία ονομαζόμενη «Σταματήστε τον κόσμο» (Stop The World). Με έναν τέτοιο συλλέκτη σκουπιδιών, ένα πρόγραμμα αισθάνεται την συλλογή απορριμμάτων ως μια παύση στη λειτουργία του. Αυτές οι παύσεις δεν έχουν περιορισμό στη διάρκεια και κατά κανόνα είναι πολύ ενοχλητικές, αφού κυμαίνονται από εκατοντάδες χιλιοστά του δευτερολέπτου ως μερικά δευτερόλεπτα . Το μήκος της παύσης εξαρτάται από το μέγεθος του σωρού, τον αριθμό των ζωντανών δεδομένων στο σωρό και πόσο επιθετικά προσπαθεί ο συλλέκτης να διεκδικήσει ελεύθερη μνήμη. Οι παύσεις μπορούν ακόμη να εμφανιστούν σε απροσδιόριστο χρόνο με απεριόριστη διάρκεια.

Μεταγλώττιση

Η μεταγλώττιση κώδικα Java σε native κώδικα εισάγει ένα παρόμοιο πρόβλημα με τη φόρτωση κλάσεων. Οι περισσότερες μοντέρνες JVM αρχικά μεταφράζουν (interpret) τις μεθόδους και μόνο αυτές οι μέθοδοι που εκτελούνται αρκετά συχνά μεταγλωττίζονται σε native κώδικα. Η καθυστερημένη μεταγλώττιση έχει ως αποτέλεσμα γρηγορότερη εκκίνηση και μειώνει το ποσοστό της μεταγλώττισης που εκτελείται κατά τον χρόνο εκτέλεσης . Αλλά η εκτέλεση μιας εργασίας με interpreted κώδικα και με μεταγλωττισμένο (compiled) μπορεί να χρειαστεί σημαντικά διαφορετικό χρόνο. Για μια αυστηρού πραγματικού χρόνου εφαρμογή , η ανικανότητα να προβλεφθεί πότε θα συμβεί η μεταγλώττιση εισάγει τόσο απροσδιοριστία (non-determinism) που κάνει αδύνατο τον προγραμματισμό των ενεργειών της διαδικασίας αποτελεσματικά. Όπως και με τη φόρτωση κλάσεων το πρόβλημα μπορεί να μετριαστεί χρησιμοποιώντας την κλάση Compiler για να μεταγλωττίσει τις μεθόδους κατά την εκκίνηση, αλλά η συντήρηση μιας τέτοιας λίστας μεθόδων είναι κουραστική και επιρρεπής σε λάθη.

5.3 Το RTSJ

Το Real-time Specification for Java – RTSJ, δηλαδή η προδιαγραφή πραγματικού χρόνου για Java, δημιουργήθηκε για να αντιμετωπίσει ορισμένους περιορισμούς της Java που απέτρεπαν την ευρεία χρήση της σε συστήματα πραγματικού χρόνου. Αντιμετωπίζει διάφορες προβληματικές καταστάσεις, στις οποίες περιλαμβάνεται ο χρονοπρογραμματισμός (scheduling), η διαχείριση μνήμης, τα νήματα, ο συγχρονισμός, ο χρόνος, τα ρολόγια και ο χειρισμός ασύγχρονων γεγονότων.

5.3.1 Κατευθυντήριες αρχές - Απαιτήσεις συμβατότητας

Η ομάδα εμπειρογνωμόνων για την Java πραγματικού χρόνου (Real-Time for Java Expert Group - RTJEG) ανέλαβε να σχεδιάσει ένα πρότυπο για την επέκταση της γλώσσας προγραμματισμού Java και της εικονικής μηχανής Java και να παρέχει μια προγραμματιστική διεπαφή, ώστε να είναι δυνατή η διαχείριση νημάτων, των οποίων η ορθότητα περιλαμβάνει χρονικούς περιορισμούς (νήματα πραγματικού χρόνου). Παρακάτω περιγράφονται οι κατευθυντήριες αρχές που ακολουθήθηκαν κατά τη δημιουργία αυτού του προτύπου.[23]

Εφαρμογή σε συγκεκριμένα περιβάλλοντα Java: Το RTSJ δεν περιλαμβάνει προδιαγραφές που θα περιορίζουν τη χρήση τους σε συγκεκριμένα περιβάλλοντα Java, όπως μια συγκεκριμένη έκδοση του Java Development Kit, του Embedded Java Application Environment, ή του Java 2 Micro Edition.

Συμβατότητα με παλαιότερες εκδόσεις: Το RTSJ δεν εμποδίζει τα υφιστάμενα, σωστά γραμμένα, μη πραγματικού χρόνου προγράμματα Java από την εκτέλεσή τους σε υλοποιήσεις του RTSJ.

Γράψτε μια φορά, εκτελέστε οπουδήποτε (Write Once, Run Anywhere): Το RTSJ θα πρέπει να αναγνωρίσει τη σημασία του "Write Once, Run Anywhere", αλλά θα πρέπει επίσης να αναγνωρίσει και τη δυσκολία της επίτευξής του για προγράμματα πραγματικού χρόνου και να μην επιχειρήσει να αυξήσει ή να διατηρήσει τη φορητότητα των εκτελέσιμων αρχείων (binary portability) σε βάρος της προβλεψιμότητας.

Τρέχουσα πρακτική σε σχέση με προηγμένα χαρακτηριστικά: Το RTSJ θα πρέπει να ασχοληθεί με τις τρέχουσες πρακτικές συστημάτων πραγματικού χρόνου καθώς και να επιτρέψει στις μελλοντικές υλοποιήσεις να περιλαμβάνουν παραπάνω δυνατότητες.

Προβλέψιμη εκτέλεση: Το RTSJ θέτει την προβλέψιμη εκτέλεση ως πρώτη προτεραιότητα σε όλες τις περιπτώσεις. Αυτό μπορεί μερικές φορές να είναι σε βάρος των τυπικών, γενικής χρήσης μέτρων υπολογιστικής απόδοσης.

Καμία επέκταση συντακτικού: Για να διευκολυνθεί το έργο των εργαλείων προγραμματισμού και να αυξήσει την πιθανότητα για υλοποιήσεις σύντομα, το RTSJ δεν πρέπει να εισάγει νέες λέξεις-κλειδιά ή να κάνει άλλες συντακτικές επεκτάσεις στη γλώσσα Java.

ποικιλία στις αποφάσεις υλοποίησης: αναγνωρίζεται ότι οι υλοποιήσεις του RTSJ μπορεί να διαφέρουν σε μια σειρά από αποφάσεις, όπως η χρήση αποτελεσματικών ή όχι αλγορίθμων , η αντιστάθμιση μεταξύ του χρόνου και του χώρου της απόδοσης, η συμπερίληψη αλγορίθμων χρονοπρογραμματισμού που δεν απαιτούνται στην ελάχιστη υλοποίηση και η διακύμανση στο μήκος της διαδρομής κώδικα για την εκτέλεση των bytecodes. Το RTSJ δεν θα πρέπει να κάνει υποχρεωτικούς αλγορίθμους ή συγκεκριμένες χρονικές σταθερές, αλλά απαιτεί να πληρείται η σημασιολογία της υλοποίησης. Το RTSJ προσφέρει την ευελιξία να δημιουργηθούν προσαρμοσμένες υλοποιήσεις για να ανταποκριθούν στις απαιτήσεις των πελατών.

5.3.2 Προσθήκες του RTSJ σε σχέση με την κλασσική Java

Η ομάδα εμπειρογνωμόνων που δημιούργησε το RTSJ έκρινε ότι υπάρχουν επτά περιοχές στις οποίες η Java χρειάζεται ενίσχυση για να επιτυγχάνει αποδόσεις πραγματικού χρόνου. Αυτές είναι [24]:

1. Χρονοπρογραμματισμός (Scheduling) και εκτέλεση νημάτων
2. Διαχείριση μνήμης
3. Συγχρονισμός
4. Χειρισμός ασύγχρονων γεγονότων (Asynchronous event handling)
5. Ασύγχρονη μεταφορά ελέγχου
6. Τερματισμός Ασύγχρονων Νημάτων
7. Πρόσβαση φυσικής μνήμης

1. Χρονοπρογραμματισμός (Scheduling) και εκτέλεση νημάτων

Τα συστήματα πραγματικού χρόνου πρέπει να ελέγχουν αυστηρά το πώς τα νήματα προγραμματίζονται χρονικά για εκτέλεση και να εγγυώνται ότι προγραμματίζονται με προβλέψιμο τρόπο, (ντετερμινιστικά) δηλαδή τα νήματα προγραμματίζονται με τον ίδιο τρόπο αν υπάρχουν οι ίδιες συνθήκες. Παρόλο που στη βιβλιοθήκη κλάσεων ορίζονται προτεραιότητες στα νήματα, δεν απαιτείται από τις παραδοσιακές εικονικές μηχανές να εφαρμόσουν αυστηρά αυτές τις προτεραιότητες. Επίσης οι υλοποιήσεις μη πραγματικού χρόνου τυπικά χρησιμοποιούν round-robin preemptive (με δυνατότητα προεκτόπισης) scheduler με απρόβλεπτη σειρά προγραμματισμού. Με το RTSJ, εφαρμόζονται αυστηρά οι προτεραιότητες και ένας fixed-priority preemptive scheduler με υποστήριξη κληρονομικότητας προτεραιότητας (priority-inheritance) για τα νήματα πραγματικού χρόνου. Αυτή η προσέγγιση εξασφαλίζει ότι το ενεργό νήμα με την υψηλότερη προτεραιότητα θα εκτελείται πάντα και θα συνεχίσει να εκτελείται ώσπου να αφήσει εθελοντικά τον επεξεργαστή ή θα διακοπεί από νήμα υψηλότερης προτεραιότητας. Η κληρονομικότητα προτεραιότητας εξασφαλίζει ότι θα αποφευχθεί η αντιστροφή προτεραιοτήτων (priority inversion), όταν ένα υψηλής προτεραιότητας νήμα χρειάζεται έναν πόρο που έχει δεσμευτεί από ένα χαμηλής προτεραιότητας νήμα. Η αντιστροφή προτεραιότητας είναι ένα από τα σημαντικά προβλήματα των συστημάτων πραγματικού χρόνου.

Λαμβάνοντας υπόψη την ποικιλομορφία των μοντέλων χρονοπρογραμματισμού και αναγνωρίζοντας ότι κάθε μοντέλο έχει ευρεία εφαρμογή σε ποικίλα συστήματα πραγματικού χρόνου, καταλήγουμε στο συμπέρασμα ότι η κατεύθυνση για μια προδιαγραφή χρονοπρογραμματισμού θα επιτρέπει σε έναν μηχανισμό χρονοπρογραμματισμού να χρησιμοποιηθεί από τα νήματα πραγματικού χρόνου, αλλά δεν θα πρέπει να προσδιοριστεί εκ των προτέρων η φύση των δυνατών μηχανισμών χρονοπρογραμματισμού. Οι προδιαγραφές είναι κατασκευασμένες έτσι ώστε να καταστεί δυνατή η υλοποίηση των αναμενόμενων αλγορίθμων χρονοπρογραμματισμού. Οι υλοποιήσεις θα επιτρέπουν την εκχώρηση κατάλληλων παραμέτρων στο μηχανισμό χρονοπρογραμματισμού, καθώς και θα παρέχουν τις απαραίτητες μεθόδους για τη δημιουργία, τη διαχείριση, την αποδοχή, και τον τερματισμό των νημάτων πραγματικού χρόνου. Ωστόσο, παρέχεται αρκετή ευελιξία στο πλαίσιο του χρονοπρογραμματισμού νημάτων για να επιτρέψουμε στις μελλοντικές εκδόσεις των προδιαγραφών να βασιστούν σε αυτή την έκδοση και να επιτρέψουν τη δυναμική φόρτωση πολιτικών χρονοπρογραμματισμού.

Για να ληφθεί υπόψη η υπάρχουσα πρακτική το RTSJ απαιτεί έναν βασικό χρονοπρογραμματιστή (Scheduler) σε όλες τις υλοποιήσεις. Ο απαιτούμενος

χρονοπρογραμματιστής θα είναι οικείος στους προγραμματιστές συστημάτων πραγματικού χρόνου. Είναι βασισμένος σε προτεραιότητες, μπορεί να διακοπεί (preemptive), και πρέπει να έχει τουλάχιστον 28 μοναδικές προτεραιότητες.

Το RTSJ προσθέτει υποστήριξη για δύο νέες κατηγορίες νημάτων που παρέχουν τη βάση για συμπεριφορά πραγματικού χρόνου:

•RealtimeThread και

•NoHeapRealtimeThread (NHRT).

•

Οι κατηγορίες αυτές παρέχουν:

•την υποστήριξη για τις προτεραιότητες,

•περιοδική συμπεριφορά,

•προθεσμίες με χειριστές που μπορούν να προκληθούν όταν ξεπερνιέται η προθεσμία, και

•χρήση των περιοχών μνήμης εκτός από το σωρό.

Τα NHRTs δεν μπορούν να έχουν πρόσβαση στο σωρό και έτσι, αντίθετα από άλλους τύπους νημάτων, τα NHRTs συνήθως δεν διακόπτονται από την συλλογή απορριμμάτων. Τα συστήματα πραγματικού χρόνου χρησιμοποιούν NHRTs με υψηλές προτεραιότητες για τα έργα με αυστηρότερες απαιτήσεις καθυστέρησης, RealtimeThreads για τους στόχους με τις απαιτήσεις καθυστέρησης που μπορούν να συμβιβαστούν με έναν συλλέκτη απορριμμάτων και τα κανονικά νήματα της Java για όλα τα άλλα. Επειδή τα NHRTs δεν μπορεί να έχουν πρόσβαση στο σωρό, η χρήση αυτών των νημάτων απαιτεί μεγάλη προσοχή, ώστε καμία κλάση που χρησιμοποιούν να μην δημιουργεί ακούσια προσωρινά ή εσωτερικά αντικείμενα στο σωρό.

2. Διαχείριση μνήμης

Αναγνωρίζεται ότι η αυτόματη διαχείριση μνήμης είναι ένα ιδιαίτερα σημαντικό χαρακτηριστικό του περιβάλλοντος προγραμματισμού Java, οπότε το έργο της διαχείρισης μνήμης θα πρέπει να υλοποιείται αυτόματα από το σύστημα και δεν θα εμπλέκεται με το έργο του προγραμματιστή. Επιπλέον, γνωρίζουμε ότι υπάρχουν πολλοί αυτόματοι αλγόριθμοι διαχείρισης μνήμης, γνωστοί και ως συλλογείς απορριμμάτων, αρκετοί από τους οποίους είναι κατάλληλοι για ορισμένες κατηγορίες συστημάτων πραγματικού χρόνου και στυλ προγραμματισμού. Επομένως η προδιαγραφή εκχώρησης και ανάκτησης μνήμης πρέπει να

a) είναι ανεξάρτητη από οποιονδήποτε συγκεκριμένο αλγόριθμο συλλογής απορριμμάτων,

b) επιτρέπει στο πρόγραμμα να αξιολογεί ακριβώς την επίδραση μιας υλοποίησης του αλγορίθμου συλλογής απορριμμάτων στο χρόνο εκτέλεσης, διακοπής, και αποστολής για εκτέλεση των νημάτων πραγματικού χρόνου

c) επιτρέπει την ανάθεση και ανάκτηση των αντικειμένων χωρίς οποιαδήποτε παρέμβαση του συλλογέα απορριμμάτων

Παρόλο που μερικά συστήματα πραγματικού χρόνου μπορούν να ανεχτούν τις καθυστερήσεις που προκύπτουν από τον συλλογέα απορριμμάτων, σε πολλές περιπτώσεις οι καθυστερήσεις είναι απαράδεκτες. Για να υποστηρίξει το RTSJ έργα που δεν μπορούν να

ανεχτούν διακοπές για συλλογή απορριμμάτων, όρισε την immortal και scoped μνήμη συμπληρωματικά από τη μνήμη σωρού (heap). Αυτές οι περιοχές μνήμης μπορούν να χρησιμοποιηθούν χωρίς να απαιτείται block εάν ενεργοποιηθεί ο συλλογέας απορριμμάτων για να ελευθερώσει μνήμη στο heap. Τα αντικείμενα που ανατίθενται στην immortal μνήμη είναι προσβάσιμα σε όλα τα νήματα και δεν συλλέγονται ποτέ. Επειδή δεν συλλέγονται, η immortal μνήμη είναι ένας περιορισμένος πόρος που πρέπει να χρησιμοποιείται με προσοχή. Η περιοχή μνήμης τύπου scope δημιουργείται και καταστρέφεται από τον προγραμματιστή. Κάθε τέτοια περιοχή μνήμης έχει ένα μέγιστο μέγεθος που μπορεί να χρησιμοποιηθεί για ανάθεση αντικειμένων. Για να εξασφαλιστεί η ακεραιότητα των αναφορών σε αντικείμενα, το RTSJ καθορίζει κανόνες για το πώς μπορεί τα αντικείμενα μιας περιοχής να αναφέρονται σε αντικείμενα άλλων περιοχών. Άλλοι κανόνες καθορίζουν πότε τα αντικείμενα της scope μνήμης μπορούν να καταστραφούν και πότε η μνήμη μπορεί να ελευθερωθεί και να χρησιμοποιηθεί ξανά. Εξαιτίας αυτής της πολυπλοκότητας η χρήση αυτών των περιοχών συνίσταται μόνο για στοιχεία που δεν ανέχονται διακοπές για συλλογή απορριμμάτων.

3. Συγχρονισμός

Η λογική του προγράμματος συχνά απαιτεί σειριακή πρόσβαση σε πόρους. Τα συστήματα πραγματικού χρόνου εισάγουν μια πρόσθετη πολυπλοκότητα: έλεγχος αναστροφής προτεραιότητας, μια κατάσταση όπου νήματα υψηλής προτεραιότητας είναι στην αναμονή περιμένοντας την ολοκλήρωση της εκτέλεσης νημάτων χαμηλής προτεραιότητας. Αποφασίστηκε ότι η λιγότερο ενοχλητική προδιαγραφή που θα επιτρέπει ασφαλή συγχρονισμό σε πραγματικό χρόνο είναι η απαίτηση σύμφωνα με την οποία οι υλοποιήσεις της λέξης κλειδί της Java synchronized να περιλαμβάνουν ένα ή περισσότερους αλγορίθμους που εμποδίζουν την αναστροφή προτεραιότητα μεταξύ των νημάτων πραγματικού χρόνου που μοιράζονται τους σειριακούς πόρους. Πρέπει επίσης να σημειωθεί ότι σε ορισμένες περιπτώσεις, η χρήση της λέξης-κλειδί synchronized που υλοποιεί την αναστροφή προτεραιότητας δεν είναι αρκετή για να αποτρέψει τόσο την αναστροφή προτεραιότητας όσο και να επιτρέπει σε ένα νήμα να είναι υψηλότερης επιλεξιμότητας για εκτέλεση από το συλλέκτη απορριμμάτων.

Το RTSJ παρέχει τη δυνατότητα στα νήματα να επικοινωνούν χωρίς συγχρονισμό μέσω κλάσεων «ουρών χωρίς αναμονή» εγγραφής και ανάγνωσης (wait-free read and write queue) .

4. Χειρισμός ασύγχρονων γεγονότων (Asynchronous event handling)

Τα συστήματα πραγματικού χρόνου συνήθως αλληλεπιδρούν στενά με τον πραγματικό κόσμο. Επειδή ο πραγματικός κόσμος είναι ασύγχρονος, πρέπει επομένως να συμπεριληφθούν αποτελεσματικοί ασύγχρονοι μηχανισμοί. Το RTSJ γενικεύει το μηχανισμό χειρισμού ασύγχρονων γεγονότων της Java. Οι νέες κλάσεις είναι η AsyncEvent, η οποία αντιπροσωπεύει γεγονότα (πράγματα μπορεί να συμβούν) και οι AsyncEventHandler, BoundAsyncEventHandler που αντιπροσωπεύει τη λογική που εκτελείται όταν συμβαίνουν αυτά τα γεγονότα. Υποστηρίζεται ο χειρισμός των ασύγχρονων γεγονότων που πυροδοτούνται από διάφορες πηγές συμπεριλαμβανομένων των χρονομετρητών, σήματα του λειτουργικού συστήματος, χαμένες προθεσμίες, και άλλα γεγονότα καθορισμένα από την εφαρμογή.

5. Ασύγχρονη μεταφορά ελέγχου

Μερικές φορές, ο πραγματικός κόσμος αλλάζει τόσο δραστικά (και ασύγχρονα) που το τρέχον σημείο της λογικής εκτέλεσης θα πρέπει να μεταφερθεί άμεσα και αποτελεσματικά σε άλλη θέση. Το RTSJ επεκτείνει το χειρισμό εξαιρέσεων της Java, για να επιτρέψει στις εφαρμογές να αλλάξουν προγραμματιστικά τη θέση ελέγχου ενός άλλου νήματος με την προσθήκη της εξαίρεσης AsynchronouslyInterruptedException. Η ασύγχρονη μεταφορά ελέγχου πραγματοποιείται με τη διακοπή ενός νήματος πραγματικού χρόνου, οπότε και δημιουργείται αυτή η εξαίρεση. Είναι σημαντικό να σημειωθεί ότι το RTSJ περιορίζει την ασύγχρονη μεταφορά του ελέγχου σε ειδικά γραμμένη λογική με την υπόθεση ότι η θέση ελέγχου μπορεί να αλλάξει ασύγχρονα.

6. Ασύγχρονος Τερματισμός Νημάτων

Και πάλι, λόγω μιας δραστικής και ασύγχρονης αλλαγής στον πραγματικό κόσμο, η λογική της εφαρμογής μπορεί να χρειαστεί να αναγκάσει ένα νήμα πραγματικού χρόνου να μεταβιβάσει γρήγορα και με ασφάλεια τον έλεγχο στο πιο εξωτερικό πεδίο και να τερματίσει με κανονικό τρόπο. Αυτό επιτυγχάνεται με την κατάλληλη αντιμετώπιση της εξαίρεσης AsynchronouslyInterruptedException. Σημειώνεται ότι σε αντίθεση με τον παραδοσιακό, ανασφαλή και αποδοκιμασμένο μηχανισμό της Java για την διακοπή των νημάτων, ο μηχανισμός του RTSJ για το χειρισμό ασύγχρονων γεγονότων και ασύγχρονης μεταβίβασης ελέγχου είναι ασφαλής.

7. Πρόσβαση φυσικής μνήμης

Αν και δεν είναι άμεσα ένα θέμα πραγματικού χρόνου, η πρόσβαση στη φυσική μνήμη είναι επιθυμητή για πολλές από τις εφαρμογές που θα μπορούσαν να κάνουν πιο παραγωγική την χρήση μιας υλοποίησης του RTSJ. Πρέπει, επομένως, να επιτρέπει στους προγραμματιστές πρόσβαση στη φυσική μνήμη σε επίπεδο byte, και κατασκευή αντικειμένων στη φυσική μνήμη. Οι κλάσεις που παρέχουν αυτή τη λειτουργία είναι οι RawMemoryAccess, RawMemoryFloatAccess, ImmortalPhysicalMemory, LTPhysicalMemory,

Επιπρόσθετα, τα συστήματα πραγματικού χρόνου χρειάζονται ρολόγια υψηλότερης ανάλυσης από εκείνα που παρέχονται από τον καθιερωμένο κώδικα της Java. Οι νέες κλάσεις HighResolutionTime και Clock προσφέρουν αυτές τις χρονικές υπηρεσίες.

5.4 Υλοποιήσεις του προτύπου RTSJ

Σήμερα υπάρχουν αρκετές εικονικές μηχανές Java, οι οποίες υλοποιούν το πρότυπο αυτό, οι οποίες παρατίθενται παρακάτω. Στην παρούσα εργασία χρησιμοποιήθηκε το πρώτο, το Sun JRTS, λόγω της καλύτερης συνεργασίας με το περιβάλλον ανάπτυξης, το Eclipse IDE.

Sun Java Real-Time System: http://java.sun.com/javase/technologies/realtime.jsp

IBM Websphere Realtime: http://www-03.ibm.com/linux/realtime.html

RTSJ Reference Implementation (RI): http://www.timesys.com/java/

jRate: http://jrate.sourceforge.net

Jamaica: http://www.aicas.com/jamaica.html

RTJRE:http://www.apogee.com/products/rtjre

5.5 Απλά παραδείγματα εφαρμογών

Παρακάτω δίνεται η υλοποίηση μερικών απλών εφαρμογών, με σκοπό την επίδειξη χρήσης και την εξοικείωση με τους νέους τύπου νημάτων RealtimeThread και NoHeapRealtimeThread, καθώς και με τον ασύγχρονο χειρισμό γεγονότων. Για την χρήση των κλάσεων που ορίζονται από το RTSJ, απαιτείται σε όλα τα αρχεία η εισαγωγή του πακέτου realtime:

```
import javax.realtime.*;
```

Αρχικά δίνεται η υλοποίηση του προβλήματος συγχρονισμού παραγωγού - καταναλωτή με χρήση Realtime Threads. Ουσιαστικά δεν υπάρχει σημαντική διαφορά από την παραδοσιακή υλοποίηση σε Java.

Η κλάση του παραγωγού:

```
public class Producer extends RealtimeThread{
        static int num=0;
        int time,id;
        Buffer buf;

        public Producer(int time, Buffer buf){
                this.time=time;
                this.buf=buf;
                id=++num;
        }

        public void run(){
                for(int i=0;i<10;i++){
                      try{
                            buf.put(id);
                            System.out.println("Produced: "+id);
                            sleep(time);
                      }
                      catch(InterruptedException e){
                            System.out.println("Error");
                      }
                }
        }

}
```

Η κλάση του καταναλωτή:

```
public class Consumer extends RealtimeThread{
        int time,id;
        static int num=0;
        Buffer buf;

public Consumer(int time,Buffer buf){
        this.time=time;
        this.buf=buf;
        id=++num;
}

public void run(){
        int product;

        for(int i=0;i<10;i++){
             try{
                  product=buf.get();
                  System.out.println("Consumed :"+product+" by "+id);
                  sleep(time);
             }
             catch(InterruptedException e){
                  System.out.println("Error");
             }
        }
}

}
```

Η κλάση του προσωρινού χώρου αποθήκευσης:

```
public class Buffer {
        int [] b;
        int pos;

        public Buffer(int i){
             b= new int[i];
             pos=0;
        }

        synchronized public int get(){
             while(pos<=0){
                  try{
                       wait();
                  }
                  catch(InterruptedException e){
                       System.out.println("Error");}
             }

             notifyAll();
             return b[--pos];
        }

        synchronized public void put(int k){
             while(pos>=b.length){
                  try{
                       wait();
                  }
                  catch(InterruptedException e) {
                       System.out.println("Error");
                  }
```

```
            }

            b[pos++]=k;
            notifyAll();
        }

}
```

Η κύρια κλάση του προγράμματος:

```
public class MyMain {
        public static void main(String[] args) {
             Buffer buf=new Buffer(2); //προσωρινός χώρος 2 θέσεων
             Consumer c=new Consumer(150,buf);
             Producer p=new Producer(100,buf);

             c.start();
             p.start();

        }

}
```

Σημαντική διαφοροποίηση έχουμε στην παρακάτω υλοποίηση όπου χρησιμοποιούνται νήματα τύπου NoHeapRealtimeThread. Αυτός ο τύπος νημάτων δεν επιτρέπεται να έχει αναφορές στη μνήμη τύπου heap, για να μη διακόπτεται από τον garbage collector. Επομένως όλα τα αντικείμενα πρέπει να δημιουργηθούν στην μνήμη τύπου Immortal, όπως φαίνεται στην κύρια κλάση του προγράμματος.

Η κλάση του παραγωγού:

```
public class Producer extends NoHeapRealtimeThread{
      static int num=0;
      int time,id;
      Buffer buf;

public Producer(SchedulingParameters priority, int time, Buffer buf){
           super(priority,ImmortalMemory.instance());

           this.time=time;
           this.buf=buf;
           id=++num;
}

public void run(){ …//χωρίς αλλαγή}

}
```

Η κλάση του καταναλωτή:

```
public class Consumer extends NoHeapRealtimeThread{
      int time,id;
      static int num=0;
      Buffer buf;

public Consumer(SchedulingParameters priority, int time,Buffer buf){
      super(priority,ImmortalMemory.instance());
```

```
        this.time=time;
        this.buf=buf;
        id=++num;
}

public void run(){…// χωρίς αλλαγή }

}
```

Η κλάση Buffer δεν παρουσιάζει καμία αλλαγή

Η κύρια κλάση του προγράμματος:

```
public class MyMain {

public static void main(String[] args) {

        MemoryArea mem=ImmortalMemory.instance();


        Runnable logic=new Runnable(){
              public void run(){
                    Buffer buf=new Buffer();
                    PriorityParameters priority=new
PriorityParameters(PriorityScheduler.instance().getNormPriority());
                    Consumer c=new Consumer(priority,150,buf);
                    Producer p=new Producer(priority,100,buf);
                    c.start();
                    p.start();
              }
        };

        mem.executeInArea(logic);
}
}
```

Στην επόμενη εφαρμογή σκοπός είναι η παρουσίαση της χρήσης των κλάσεων AsyncEvent και AsyncEventHandler. Υποθέτουμε ότι έχουμε μια «πηγή» γεγονότων και τρία νήματα - «στόχους». Όταν η πηγή παράγει ένα γεγονός, ενεργοποιούνται μέσω του Handler τα τρία νήματα-στόχοι, τα οποία βρίσκονται σε κατάσταση αναμονής.

η κλάση της πηγής γεγονότων

```
public class Source extends RealtimeThread{
        AsyncEvent ev;

        public Source(AsyncEvent ev){
             this.ev=ev;
        }

        public void run(){
             for(int i=0;i<10;i++){
                  try{
                       // το Event πυροδοτείται κάθε δευτερόλεπτο
                       sleep(1000);
                       ev.fire();
                  }
```

```
                catch(InterruptedException e){
                     System.out.println("Error");
                }
           }
      }
}
```

η κλάση του νήματος – «στόχου»

```
public class Target extends RealtimeThread{
      int id;
      public Object o;

      public Target(int id){
           this.id=id;
           o=new Object();
      }

      public void run(){
           int i=0;
           while(i<10){

                try{
                     synchronized (o){
                          o.wait();
                          accept();
                     }
                     i++;
}
                catch(InterruptedException e){
                     System.out.println("error!");
                }
           }
      }

      public synchronized void accept(){
System.out.println(
RealtimeThread.currentRealtimeThread().getName()+"  "+id);
      }
}
```

// η κλάση του Asynceventhandler

```
public class myAEH extends AsyncEventHandler{
      Target[] t;

      public myAEH(Target[] t){
           this.t=t;
      }

      public void handleAsyncEvent(){
/*H μέθοδος αυτή περιέχει τις ενέργειες που γίνονται όταν συμβεί το
γεγονός*/

           System.out.print("AEH start ");
           for(int i=0;i<t.length;i++){
                synchronized(t[i].o){
                t[i].o.notify();
                }
           }
```

```
            System.out.println("AEH end ");
        }
}
```

//η κύρια κλάση του προγράμματος

```
public class MyMain {
      public static void main(String[] args) {

            PriorityScheduler ps=PriorityScheduler.instance();
            System.out.println(ps.getMaxPriority());
            System.out.println(ps.getMinPriority());
            System.out.println(ps.getNormPriority());

            AsyncEvent ev=new AsyncEvent();
            Source s=new Source(ev);
            Target[] t=new Target[3];

            myAEH AEH=new myAEH(t);
            ev.setHandler(AEH); //σύνδεση Event με Handler

            for(int i=0;i<3;i++){
                 t[i]=new Target(i);
                 t[i].start();
            }

            //Αλλαγή προτεραιοτήτων
            //Target 0 : Default
            //Target 1 : Default+10
            //Target 1 : Default-10
            PriorityParameters pp=
(PriorityParameters) t[0].getSchedulingParameters();
            pp.setPriority(pp.getPriority()+10);
            t[1].setSchedulingParameters(pp);
            pp.setPriority(pp.getPriority()-20);
            t[2].setSchedulingParameters(pp);

            s.start();
      }

}
```

Κεφάλαιο 6
Case Study

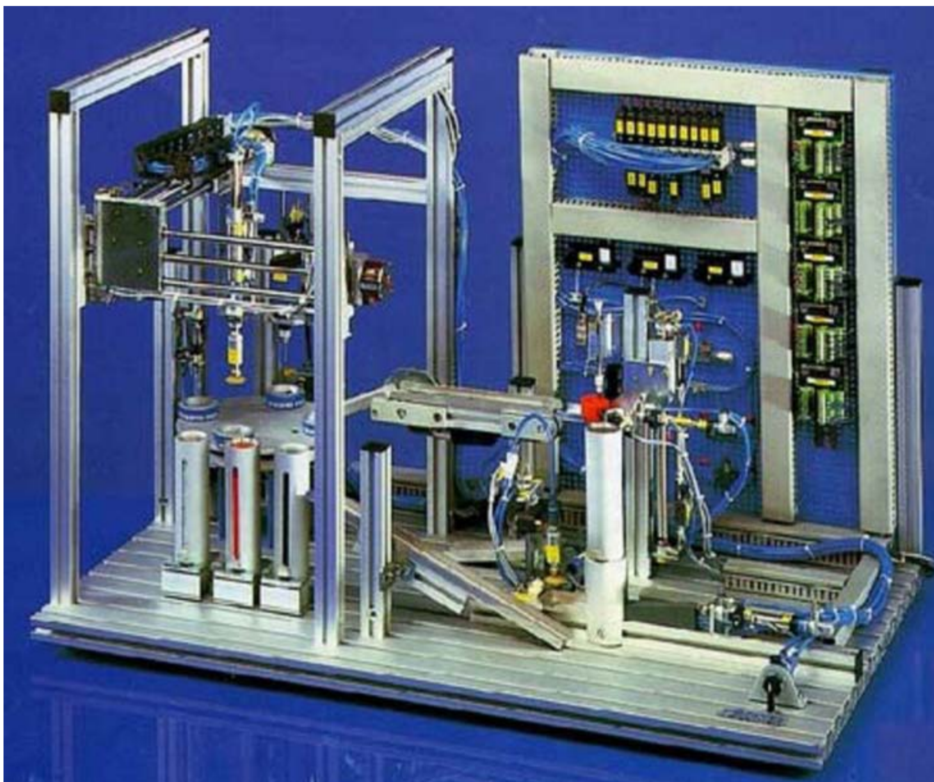
Για την ανάλυση και την υλοποίηση όσων αναφέρθηκαν στα προηγούμενα κεφάλαια για τον παράλληλο προγραμματισμό επιλέχθηκε ως παράδειγμα εφαρμογής προς έρευνα το σύστημα Festo MPS, η εφαρμογή ελέγχου του οποίου είναι υλοποιημένη σε Real Time Java βάση το πρότυπο των Function Block. Αρχικά παρουσιάζεται το σύστημα και μετά οι βασικές αρχές του προηγούμενου προτύπου. Ύστερα αναλύεται πως το σύστημα αξιοποιεί την ύπαρξη πολλών επεξεργαστών. Στο τέλος παρουσιάζεται ο τρόπος με τον οποίο ο προγραμματιστής ορίζει σε ποιον επεξεργαστή θα τρέξει κάθε νήμα.

6.1 Το σύστημα Festo MPS

Το Festo MPS σύστημα (Modular Production System) είναι ένα σύστημα γραμμής παραγωγής που αναπτύχθηκε από την εταιρεία Festo για εκπαιδευτικούς και ερευνητικούς σκοπούς. Το σύστημα απαρτίζεται από τις ακόλουθες μονάδες:

- Μονάδα Διανομής (Distribution Unit)
- Μονάδα Ελέγχου (Test Station)
- Μονάδα Επεξεργασίας (Processing Station)

Κυλινδρικά κομμάτια χρησιμοποιούνται σαν δείγματα (workpieces), τα οποία διαφέρουν στο χρώμα τους (κόκκινο, μαύρο, ασημί), στο υλικό τους (αλουμίνιο, πλαστικό) και στο ύψος τους. Το σύστημα περιλαμβάνει πνευματικές συσκευές παραγωγής, ευθύγραμμης και περιστροφικής κίνησης, όπως επίσης και ηλεκτρικές συσκευές (actuators). Οι αισθητήρες (sensors) για τον έλεγχο ύπαρξης αντικειμένου, την αναγνώριση χρώματος, υλικού και τη μέτρηση ύψους βασίζονται σε μηχανικές, οπτικές, επαγωγικές και χωρητικές διαδικασίες μέτρησης. [25]

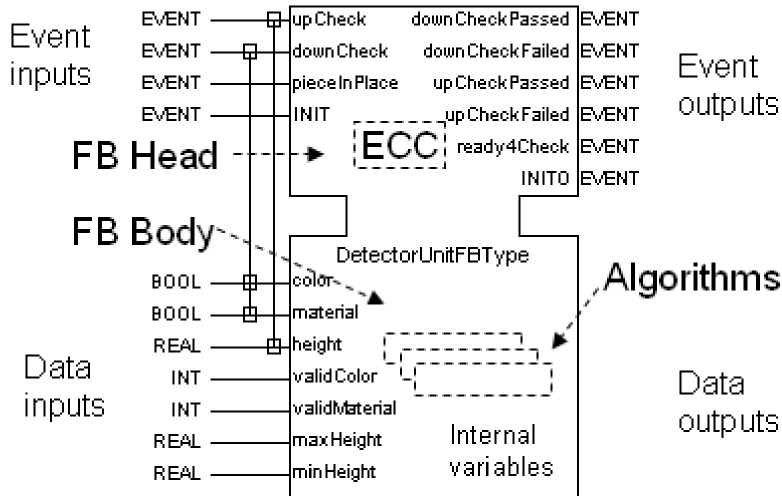


Σχήμα 1. Το πραγματικό σύστημα Festo MPS

6.2 Function Blocks

Τα Function blocks – FB είναι μια δομή που χρησιμοποιείται ευρέως από τους μηχανικούς συστημάτων ελέγχου. Αρχικά περιγράφηκε από το πρότυπο IEC1131 για γλώσσες προγραμματισμού για προγραμματιζόμενους λογικούς ελεγκτές (Programmable Logic Controllers). Ωστόσο αυτό το πρότυπο δεν ανταποκρινόταν στην αυξανόμενη ζήτηση για μια πιο ευέλικτη διαδικασία ανάπτυξης στον τομέα ελέγχου και αυτοματισμού. Για αυτό το λόγο η IEC όρισε στο πρότυπο IEC61499 τις βασικές έννοιες για το σχεδιασμό κατανεμημένων συστημάτων ελέγχου (distributed control Systems - DCS). Αυτό το πρότυπο επεκτείνει την έννοια των FB για να μοιραστεί μερικά από τα καλώς ορισμένα και ήδη ευρέως αποδεκτά πλεονεκτήματα της αντικειμενοστραφούς τεχνολογίας. Σύμφωνα με αυτό το πρότυπο, το FB αποτελείται από μια κεφαλή (head) και ένα σώμα (body) όπως φαίνεται στη γραφική αναπαράσταση στο σχήμα 2. [26]

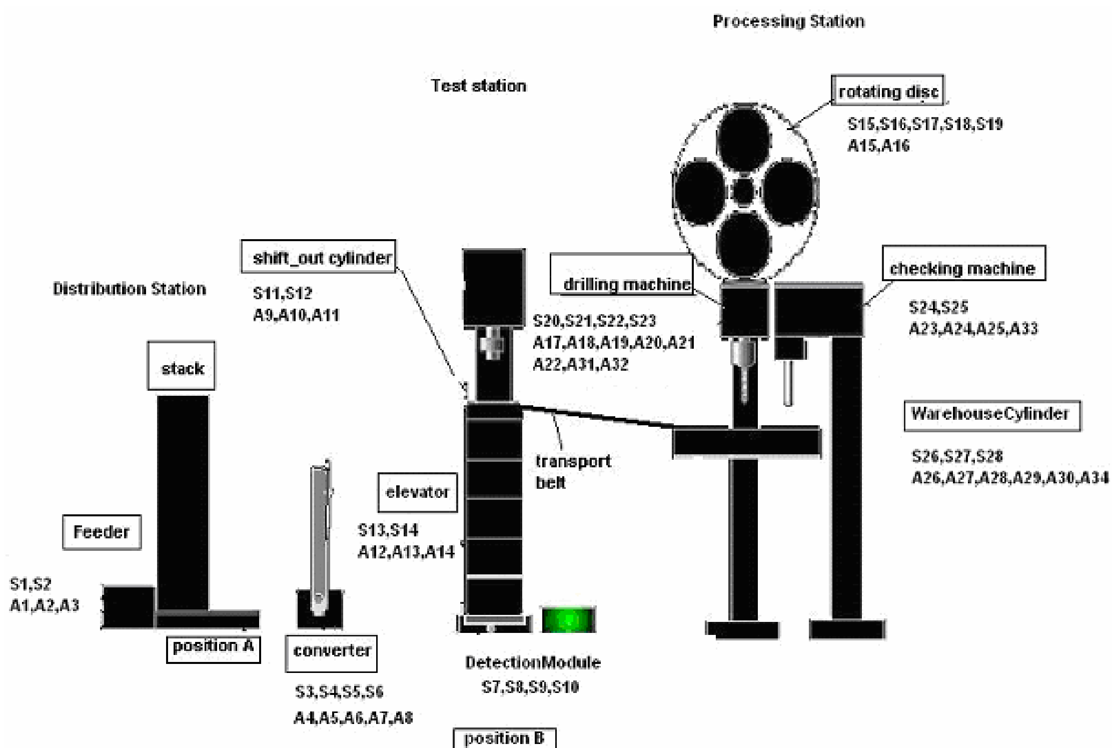
Οι αλγόριθμοι, υπό τη μορφή εσωτερικών συναρτήσεων, ορίζουν τη λειτουργικότητα του FB. Αυτοί επεξεργάζονται τα δεδομένα εισόδου και τις εσωτερικές μεταβλητές και παράγουν δεδομένα εξόδου. Η σειρά που καλούνται οι αλγόριθμοι ορίζεται στο head, το οποίο αποδέχεται τα γεγονότα εισόδου και παράγει τα γεγονότα εξόδου. Το Execution Control Chart – ECC (διάγραμμα ελέγχου εκτέλεσης), μια παραλλαγή του διαγράμματος καταστάσεων (statechart), αποτελείται από τις EC states, EC transitions and EC actions (καταστάσεις, μεταβάσεις και ενέργειες του Ελέγχου Εκτέλεσης) και χρησιμοποιείται για να ορίζεται η συμπεριφορά του head. Ένα FB μπορεί να είναι πολύ απλό, όπως για παράδειγμα αυτά που χρησιμοποιούνται για τον έλεγχο ενός απλού μηχανικού μέρους, όπως μια βαλβίδα, ή πολύ περίπλοκα, όπως στην περίπτωση ενός ρομποτικού βραχίονα. Η εφαρμογή ελέγχου ορίζεται ως ένα δίκτυο στιγμιότυπων FB που διασυνδέονται μέσω σύνδεσης γεγονότων και δεδομένων.



Σχήμα 2. Δομή Function block [26]

6.3 Αξιοποίηση παραλληλισμού από την εφαρμογή ελέγχου του συστήματος

Για τον έλεγχο του συστήματος έχει αναπτυχθεί μια εφαρμογή σε Real Time Java, η οποία αξιοποιεί το πρότυπο των Function Blocks. Η εφαρμογή επικοινωνεί με το πραγματικό σύστημα μέσα από τα μηνύματα των αισθητήρων και των ενεργοποιητών. Δέχεται μηνύματα από το πραγματικό σύστημα μέσω των αισθητήρων και σε απόκριση αυτών των μηνυμάτων στέλνει μηνύματα σε αυτό μέσω των ενεργοποιητών. Στην γραφική αναπαράσταση του συστήματος στο σχήμα 3 φαίνονται οι συσκευές που το αποτελούν και απαριθμούνται οι ενεργοποιητές (γράμμα S) και οι αισθητήρες (γράμμα A).



Σχήμα 3. Γραφική αναπαράσταση του συστήματος Festo MPS [27]

Στην παρούσα εργασία επειδή το πραγματικό σύστημα δεν είναι διαθέσιμο, η εφαρμογή ελέγχου επικοινωνεί με μια εφαρμογή εξομοίωσης του φυσικού συστήματος.

Χρήση νημάτων πραγματικού χρόνου

Για κάθε «έξυπνη» συσκευή που απαρτίζει το σύστημα αντιστοιχίζεται ένα Function Block: Feeder, Converter, Detection Module, Shift out cylinder (αντιστοιχεί στα FB SOC_UP και SOC_DOWN), Elevator, Rotating Disc, Drilling Machine, Checking Machine, Warehouse cylinder. Για τη σωστή λειτουργία και το συγχρονισμό του συστήματος χρησιμοποιούνται επιπλέον τύποι FB που δεν αντιστοιχούν σε κάποια φυσική οντότητα: HeightFailed, HeightPassed, Tester2Converter. Η επικοινωνία της εφαρμογής με τους αισθητήρες και τους ενεργοποιητές υλοποιείται μέσω των κλάσεων SensorListener και OutSocket.

Τα σήματα εισόδου των FB είναι τα σήματα που λαμβάνονται από τους αισθητήρες, ενώ τα σήματα εξόδου είναι αυτά που στέλνονται στους ενεργοποιητές (actuators). Επίσης τα FB μπορούν να δεχθούν σαν γεγονότα εισόδου σήματα από άλλα FB και να στείλουν σήματα εξόδου προς άλλα FB.

Η εφαρμογή αναπτύχθηκε έτσι ώστε το φυσικό σύστημα να επεξεργάζεται περισσότερους από έναν κυλίνδρους ταυτόχρονα, αφού το κάθε μηχανικό υποσύστημα μπορεί να λειτουργεί ανεξάρτητα από τα υπόλοιπα. Βέβαια απαιτείται συγχρονισμός μεταξύ τους, δηλαδή για παράδειγμα, ο Converter για να μεταφέρει ένα κύλινδρο από τη θέση A του Σχήματος 2 στη θέση B, πρέπει η θέση B να είναι ελεύθερη. Κάθε FB έχει ένα ECC, το οποίο παρέχει όλη αυτή τη λειτουργικότητα. Το ECC αναγνωρίζει τα γεγονότα εισόδου και ελέγχει αν πρέπει να υπάρξει μετάβαση σε κάποια άλλη κατάσταση, οπότε πραγματοποιείται η μετάβαση και εκτελούνται οι καθορισμένες ενέργειες. Το ECC κληρονομεί την κλάση RealtimeThread, άρα είναι ένα νήμα πραγματικού χρόνου, οπότε εκτελείται ανεξάρτητα από τα υπόλοιπα ECC.

Επομένως η εφαρμογή ελέγχου του συστήματος έχει 14 νήματα πραγματικού χρόνου, ισάριθμα με τα FB που αναφέρθηκαν παραπάνω. Επίσης υπάρχει ένα νήμα κανονικής προτεραιότητας που εκτελεί την main, η οποία δημιουργεί ένα νήμα πραγματικού χρόνου, το οποίο αναλαμβάνει την δημιουργία και την αρχικοποίηση όλων των αντικειμένων του προγράμματος και μετά τερματίζεται.

Χρήση AsyncEventHandler

Επιπλέον υπάρχουν μερικά νήματα που προκύπτουν από τους AsyncEventHandlers. Καταρχάς, η σχέση γεγονότων και χειριστών μπορεί να είναι πολλά προς πολλά, δηλαδή ένα AsyncEvent μπορεί να έχει προσδεθεί σε έναν ή παραπάνω AsyncEventHandler, όπως και ένας AsyncEventHandler μπορεί να εξυπηρετεί ένα ή παραπάνω AsyncEvent. Στη συγκεκριμένη εφαρμογή σε κάθε AsyncEvent έχει ανατεθεί ένας AsyncEventHandler.

Όταν υπάρξει ένα γεγονός (δηλαδή γίνει κλήση της μεθόδου fire() ενός αντικειμένου τύπου AsyncEvent), οι AsyncEventHandlers που έχουν συνδεθεί με το γεγονός αυτό προγραμματίζονται για εκτέλεση. Ο AsyncEventHandler είναι αντικείμενο τύπου Schedulable,

άρα έχει τα ίδια χαρακτηριστικά με ένα RealtimeThread, αλλά η πρόθεση του προτύπου RTSJ είναι να έχουν διαφορετικές ρυθμίσεις για καλύτερη απόδοση. Παρόλο που ένα RealtimeThread μπορεί να χρησιμοποιηθεί για να εκτελεστεί ένας σύντομος κώδικας, αυτή δεν είναι καλή επιλογή για λόγους απόδοσης και κατανάλωσης μνήμης. Στις υλοποιήσεις εικονικών μηχανών Java πραγματικού χρόνου συνήθως επιλέγεται να μην αφιερώνεται ένα νήμα ανά AsyncEventHandler, αλλά να χρησιμοποιείται μια δεξαμενή νημάτων. Έτσι όταν απαιτείται να εκτελεστεί ένας AsyncEventHandler, προσδένεται σε κάποιο διαθέσιμο νήμα πραγματικού για να εκτελεστεί το έργο του και η πρόσδεση αυτή συμβαίνει δυναμικά κατά το χρόνο εκτέλεσης.

Αν η καθυστέρηση ώσπου να συμβεί η πρόσδεση σε ένα νήμα και η εκτέλεση θεωρείται μη αποδεκτή, μπορούν να χρησιμοποιηθούν αντικείμενα τύπου BoundAsyncEventHandler. Ένας τέτοιος χειριστής γεγονότων προσδένεται κατά τη δημιουργία του αποκλειστικά σε ένα νήμα. Η χρήση αυτού του τύπου χειριστών καλό είναι να περιορίζεται μόνο όταν είναι αναγκαία, καθώς η ύπαρξη ενός νήματος ανά χειριστή δεν εκμεταλλεύεται το μηχανισμό των χειριστών, αλλά καταλήγει απλά στη χρήση νημάτων. [28]

Επιλογές της εικονικής μηχανής σχετικά με τη διαχείριση νημάτων

Στην υλοποίηση Sun Java RTS υπάρχουν μερικές επιλογές σχετικές με την δεξαμενή νημάτων για την εξυπηρέτηση των AsyncEventHandler και μερικές σχετικές με τη συνάφεια επεξεργαστή. Οι επιλογές αυτές δίνονται ως ορίσματα γραμμής εντολών στην εικονική μηχανή με την εξής σύνταξη:

–XX:+<flag> για να τεθεί το flag αληθές

–XX: –<flag> για να τεθεί το flag ψευδές

–XX: <param>=<value> για να τεθεί τιμή σε μια παράμετρο

Οι παράμετροι σχετικά με τη δεξαμενή νημάτων είναι:

RTSJInitialNumberOfRealtimeServerThread : ο αρχικός αριθμός νημάτων για την εξυπηρέτηση στιγμιότυπων AsyncEventHandler που χρησιμοποιούν τη μνήμη τύπου heap. Έχει προκαθορισμένη τιμή 2.

RTSJMaxNumberOfRealtimeServerThread: Ο μέγιστος αριθμός νημάτων που προορίζονται για την εξυπηρέτηση στιγμιότυπων AsyncEventHandler που χρησιμοποιούν τη μνήμη τύπου heap. Έχει προκαθορισμένη τιμή 100.

RTSJInitialNumberOfNoHeapRealtimeServerThread : ο αρχικός αριθμός νημάτων για την εξυπηρέτηση στιγμιοτύπων AsyncEventHandler που δεν χρησιμοποιούν τη μνήμη τύπου heap. Έχει προκαθορισμένη τιμή 2.

RTSJMaxNumberOfNoHeapRealtimeServerThread: Ο μέγιστος αριθμός νημάτων που προορίζονται για την εξυπηρέτηση στιγμιότυπων AsyncEventHandler που δεν χρησιμοποιούν τη μνήμη τύπου heap. Έχει προκαθορισμένη τιμή 100.

Οι παράμετροι σχετικά με τη συνάφεια επεξεργαστή είναι:

RTSJBindNHRTToProcessors: προσδένει τα νήματα τύπου NoHeapRealtimeThread με την καθοριζόμενη λίστα επεξεργαστών (χωριζόμενων από κόμματα) ή με το καθοριζόμενο cpuset. Η εντολή αυτή ισχύει για το λειτουργικό σύστημα Linux, ενώ για Solaris η αντίστοιχη εντολή είναι RTSJBindNHRTToProcessorSet

RTSJBindRTTToProcessors: προσδένει τα νήματα τύπου RealtimeThread με την καθοριζόμενη λίστα επεξεργαστών (χωριζόμενων από κόμματα) ή με το καθοριζόμενο cpuset. Η εντολή αυτή ισχύει για το λειτουργικό σύστημα Linux, ενώ για Solaris η αντίστοιχη εντολή είναι RTSJBindRTTToProcessorSet

RTSJTraceThreading: Παρακολουθεί και εκτυπώνει γεγονότα σχετικά με τον χρονοπρογραμματισμό νημάτων πραγματικού χρόνου, όπως δημιουργία, καταστροφή, αλλαγή προτεραιότητας, μπλοκάρισμα σε κλείδωμα, αναμονή. Η προκαθορισμένη τιμή είναι false.

RTSJTraceThreadBinding: παρακολουθεί την πρόσδεση νημάτων σε επεξεργαστές.

6.4 Ρητή διαχείριση νημάτων

Το σύστημα Festo MPS είναι ένα σύστημα που λειτουργεί κατανεμημένα. Δηλαδή δεν υπάρχει κεντρικός έλεγχος, αλλά κάθε μονάδα μόνη της είναι υπεύθυνη για το συντονισμό με τις υπόλοιπες. Υποθέτουμε ότι έχουμε έναν πολυπύρηνο επεξεργαστή και για λόγους προσομοίωσης και δοκιμής θέλουμε κάθε μονάδα του συστήματος (Διανομής, Ελέγχου, Επεξεργασίας) να εκτελείται σε διαφορετικό πυρήνα.

Καταρχάς πρέπει να λάβουμε υπόψη ότι στο σύστημα εκτελούνται και άλλες διεργασίες. Για να απελευθερώσουμε τους επεξεργαστές που θα τρέξει η εφαρμογή μας, πρέπει να ορίσουμε ότι όλες οι υπόλοιπες διεργασίες του συστήματος δεν θα τρέχουν σε αυτούς τους επεξεργαστές. Ένας τρόπος να πραγματοποιηθεί αυτό είναι να δημιουργηθεί ένα script που λαμβάνει τα PID από όλες τις διεργασίες και τους αλλάζει τη συνάφεια. Επειδή όμως μια διεργασία κληρονομεί τη συνάφεια από τη μητρική της, ένας πιο απλός τρόπος να συμβεί αυτό είναι να θέσουμε κατά την εκκίνηση του συστήματος την διεργασία init (με PID=1) και την τρέχουσα διεργασία εκείνη τη στιγμή να εκτελούνται πχ στον επεξεργαστή με αριθμό 0, οπότε όλες οι διεργασίες του συστήματος θα εκτελεστούν στον ίδιο επεξεργαστή, αφήνοντας τους υπόλοιπους ελεύθερους.[15]

Μια απλή λύση για να επιτύχουμε την εκτέλεση κάθε μονάδας σε έναν συγκεκριμένο επεξεργαστή, θα μπορούσε να είναι το «σπάσιμο» της εφαρμογής σε τρεις εφαρμογές που θα τρέχουν ανεξάρτητα. Έτσι είναι πολύ απλό να αναθέσουμε την εκτέλεση της καθεμίας σε έναν συγκεκριμένο πυρήνα, με την εντολή taskset που περιγράφεται στο κεφάλαιο 3. Όμως προκύπτουν αρκετά προβλήματα κυρίως στην επικοινωνία μεταξύ τους και επίσης υπάρχουν στην εφαρμογή αντικείμενα που μοιράζονται και οι τρεις μονάδες.

Εδώ ο σκοπός είναι να βρεθεί ένας τρόπος ώστε κατά τον χρόνο εκτέλεσης να μπορεί ένα νήμα μιας εφαρμογής να προσδεθεί σε έναν συγκεκριμένο επεξεργαστή. Όπως φαίνεται από τα παραπάνω, η εικονική μηχανή πραγματικού χρόνου προσφέρει κάποιες επιπλέον επιλογές, όπως να εκτελεστούν όλα τα νήματα ενός τύπου σε κάποιον επεξεργαστή. Όμως δεν παρέχεται τρόπος ρητής διαχείρισης της συνάφειας από τον προγραμματιστή. Ο βασικός λόγος είναι ότι αυτή η λειτουργία εξαρτάται από το λειτουργικό σύστημα και η Java είναι μια γλώσσα με βασικό πλεονέκτημα την ανεξαρτησία πλατφόρμας. Εκτός από πολύ ειδικές περιπτώσεις όπου χρειάζεται ένα πολύ συγκεκριμένο αποτέλεσμα, γενικά είναι καλό να αφήνεται η διαχείριση

στην εικονική μηχανή και στο λειτουργικό σύστημα, καθώς υπάρχει βελτιστοποιημένη απόδοση, λόγω της εξισορρόπηση φορτίου,.

Επομένως ο στόχος θα επιτευχθεί με τη βοήθεια μιας κλήσης του λειτουργικού συστήματος. Στη Java ο συνηθισμένος τρόπος να εκτελεστεί μια εντολή του λειτουργικού συστήματος είναι:

Runtime.getRuntime().exec("command");

To String "command" είναι μια εντολή του λειτουργικού συστήματος η οποία εκτελείται σε νέα διεργασία. Αυτός ο τρόπος είναι ακατάλληλος στην περίπτωση αυτή, γιατί η εντολή εκτελείται σε νέα διεργασία, ενώ για να αλλάξει η συνάφεια επεξεργαστή θέλουμε η εντολή να εκτελεστεί από το καλόν νήμα.

Η λύση είναι να χρησιμοποιηθεί το Java Native Interface – JNI. Το JNI επιτρέπει σε ένα πρόγραμμα να καλέσει μια μέθοδο γραμμένη σε C/C++. Έτσι θα γραφεί μια κλάση σε Java η οποία θα χρησιμοποιεί μια βιβλιοθήκη γραμμένη σε C, η οποία θα εκμεταλλεύεται τις κλήσεις συστήματος που παρέχει το Linux για νε τίθεται η επιθυμητή συνάφεια επεξεργαστή.

Java Native Interface

Το JNI χρησιμοποιείται ως εξής:

•Αρχικά ο κώδικας γράφεται σε java (στο συγκεκριμένο παράδειγμα Affinity.java) και δηλώνονται οι native μέθοδοι. Η κλάση μεταγλωττίζεται κανονικά (παράγεται το αρχείο Affinity.class). Σε συστήματα Linux στον φάκελο εγκατάστασης του JDK στο αρχείο include/jni.h συνήθως πρέπει να υπάρξει η διόρθωση #include "jni_md.h" σε #include "linux/jni_md.h".

•Από την κλάση παράγεται ένα αρχείο επικεφαλίδας (.h), το οποίο περιέχει τα πρωτότυπα των συναρτήσεων που θα υλοποιηθούν σε C/C++. Για την Affinity.class έχουμε:

javah -jni Affinity

•Γράφεται ο κώδικας σε C/C++, περιλαμβάνοντας το παραπάνω αρχείο επικεφαλίδας. Οι συναρτήσεις απαιτούν δύο ορίσματα τύπου JNIEnv και Jobject τα οποία χρειάζονται για την επικοινωνία της εικονικής μηχανής με τη native συνάρτηση. Μετά ακολουθούν τα ορίσματα που επιθυμεί ο προγραμματιστής. Οι primitive τύποι μεταβλητών της Java μπορούν να χρησιμοποιηθούν απευθείας πχ ο jint αντιστοιχεί στον int, ενώ οι υπόλοιποι τύποι περνούν ως αναφορές και απαιτείται χειρισμός από συγκεκριμένες συναρτήσεις.

•Το αρχείο μεταγλωττίζεται για τη δημιουργία μιας στατικής βιβλιοθήκης ως εξής (από το αρχείο Affinity.c παράγεται το αρχείο libAffinity.so):

```
gcc -o libAffinity.so -shared -Wl,-soname,libAffinity.so
-I/usr/java/jdk1.6.0_18/include
-I/usr/java/jdk1.6.0_18/include/linux
Affinity.c -static –lc
```

•Εκτελούμε το πρόγραμμα συμπεριλαμβάνοντας στα ορίσματα της εικονικής μηχανής την επιλογή Djava.library.path=PATH, όπου PATH ο φάκελος στον οποίο βρίσκεται η βιβλιοθήκη που δημιουργήσαμε προηγουμένως.

Π.χ. αν η βιβλιοθήκη βρίσκεται στον τρέχοντα φάκελο

java -Djava.library.path=. Affinity

Παρακάτω δίνεται η υλοποίηση της κλάσης Affinity, η οποία παρέχει λειτουργικότητα αντίστοιχη των κλήσεων συστήματος του Linux, οι οποίες ορίζονται στο sched.h και η λειτουργία τους περιγράφηκε στο κεφάλαιο 3.

Έχει τις παρακάτω μεθόδους:

`void setAffinity(int n)`: επιτρέπει στο νήμα να εκτελεστεί μόνο στον επεξεργαστή με αριθμό n

`int checkAffinity(int n)`: ελέγχει αν το νήμα επιτρέπεται να εκτελεστεί στον επεξεργαστή με αριθμό n

`void addAffinity(int n)`: προσθέτει τον επεξεργαστή n στο σύνολο των επεξεργαστών που επιτρέπεται να εκτελεστεί το νήμα

`void clearAffinity(int n)`: αφαιρεί τον επεξεργαστή n από το σύνολο των επεξεργαστών που επιτρέπεται να εκτελεστεί το νήμα

```
public class Affinity {
//native μέθοδοι που θα υλοποιηθούν σε C
private native void linuxSetAffinity(int n);
private native int linuxCheckAffinity(int n);
private native void linuxAddAffinity(int n);
private native void linuxClearAffinity(int n);

    static {
//φόρτωση της βιβλιοθήκης
        System.loadLibrary("Affinity");
    }

    public void setAffinity(int n){
    linuxSetAffinity(n);
    }

    public boolean checkAffinity(int n){
    int check=linuxCheckAffinity(n);
    if(check!=0){
    return true;
    }
    else{
```

```
        return false;
        }
        }

        public void addAffinity(int n){
        linuxAddAffinity(n);
        }

        public void clearAffinity(int n){
        linuxClearAffinity(n);
        }
    }
```

Παρακάτω δίνεται η υλοποίηση της βιβλιοθήκης libAffinity.so σε C. Ουσιαστικά ενθυλακώνονται οι κατάλληλες κλήσεις συστήματος.

```
#define _GNU_SOURCE
#include <jni.h>
#include <stdio.h>
#include <sched.h>
#include "Affinity.h"

JNIEXPORT void JNICALL Java_Affinity_linuxSetAffinity(JNIEnv
*env, jobject obj, jint n){

int i=n;
cpu_set_t cpuSet;
CPU_ZERO(&cpuSet);
CPU_SET(i,&cpuSet);
sched_setaffinity(0, sizeof(cpu_set_t), &cpuSet);
}


JNIEXPORT jint JNICALL Java_Affinity_linuxCheckAffinity(JNIEnv
*env, jobject obj, jint n){

int i=n;
cpu_set_t cpuSet;
sched_getaffinity(0, sizeof(cpu_set_t), &cpuSet);
return CPU_ISSET(i,&cpuSet);
}

JNIEXPORT void JNICALL Java_Affinity_linuxAddAffinity(JNIEnv
*env, jobject obj, jint n){

int i=n;
cpu_set_t cpuSet;
```

```
  sched_getaffinity(0, sizeof(cpu_set_t), &cpuSet);
  CPU_SET(i,&cpuSet);
  sched_setaffinity(0, sizeof(cpu_set_t), &cpuSet);
  }

  JNIEXPORT void JNICALL Java_Affinity_linuxClearAffinity(JNIEnv
*env, jobject obj, jint n){

  int i=n;
  cpu_set_t cpuSet;
  sched_getaffinity(0, sizeof(cpu_set_t), &cpuSet);
  CPU_CLR(i,&cpuSet);
  sched_setaffinity(0, sizeof(cpu_set_t), &cpuSet);
  }
```

Για να χρησιμοποιηθεί η παραπάνω κλάση, δημιουργείται ένα στιγμιότυπο τύπου Affinity στην κλάση Main και το κάθε νήμα καλεί τη μέθοδο setAffinity αυτού του αντικειμένου. Για μεγαλύτερη ευκολία έχει προστεθεί μια μεταβλητή int cpu στην κλάση BasicFB, οπότε όταν δημιουργούμε ένα νέο FB δίνουμε στη μεταβλητή αυτή τον αριθμό του επεξεργαστή που επιθυμούμε να εκτελείται αυτό το νήμα και στη μέθοδο run του ECC η πρώτη εντολή που εκτελείται είναι η

Main.affinity.setAffinity(cpu);

Σε τετραπύρηνο επεξεργαστή μπορούμε να υποθέσουμε ότι η μονάδα διανομής θα εκτελείται στον επεξεργαστή 1, η μονάδα δοκιμής στον επεξεργαστή 2 και η μονάδα επεξεργασίας στον επεξεργαστή 3, οπότε θέτουμε τις εξής τιμές στις κλάσεις που αποτελούν το σύστημά μας.

cpu=1 για τις Feeder, Converter

cpu=2 για τις Detection Module, SOC_UP, SOC_DOWN, Elevator

cpu=3 για τις Rotating Disc, Drilling Machine, Checking Machine, Warehouse cylinder

cpu=0 για τις υπόλοιπες βοηθητικές κλάσεις.

Συμπεράσματα

Σε αυτή την εργασία εξετάστηκε ο τρόπος αξιοποίησης των πολυπύρηνων επεξεργαστών από ένα ενσωματωμένο σύστημα υλοποιημένου σε Real Time Java. Διαπιστώθηκε ότι κάθε επίπεδο του συστήματος προσφέρει αρκετούς τρόπους ώστε να υποστηρίξει τον παράλληλο προγραμματισμό.

Το λειτουργικό σύστημα Linux μπορεί να διαχειριστεί αποδοτικά μια πολυνηματική εφαρμογή αυτόματα με τον χρονοπρογραμματιστή, αλλά παρέχει και υποστήριξη για τη διαχείριση των διεργασιών και των νημάτων από τον χρήστη. Για αυτό το σκοπό διαθέτει μια εντολή για χρήση από τη γραμμή εντολών και ένα σύνολο κλήσεων συστήματος, το οποίο μπορεί να χρησιμοποιήσει ένας προγραμματιστής σε μια εφαρμογή.

Η βασική βιβλιοθήκη της Java παρέχει ένα πλήθος κλάσεων και διεπαφών που διευκολύνουν την ανάπτυξη σύνθετων παράλληλων εφαρμογών. Όμως δεν προσφέρει κάποιο τρόπο ρητής ανάθεσης ενός νήματος σε έναν συγκεκριμένο επεξεργαστή, με βασικό επιχείρημα ότι η λειτουργία αυτή περιορίζει την μεταφερσιμότητα του προγράμματος, αφού εξαρτάται από το λειτουργικό σύστημα.

Επίσης εξετάστηκε το πρότυπο Java OpenMP, το οποίο διευκολύνει την ανάπτυξη παράλληλων εφαρμογών, αφού ο προγραμματιστής δεν έχει την ευθύνη της δημιουργίας και διαχείρισης των νημάτων, αλλά οι διαδικασίες αυτές αυτοματοποιούνται. Τα μεγάλα πλεονεκτήματα του προτύπου αυτού εντοπίζονται στον αυτόματο παραλληλισμό βρόχων και στην εύκολη παραλληλοποίηση ήδη υπάρχοντος σειριακού κώδικα, αλλά κρίνεται ότι έχει περιορισμένη χρησιμότητα στην ανάπτυξη πολύπλοκων εφαρμογών, με πολλούς διαφορετικούς τύπους νημάτων, όπως το Festo MPS.

Η Real Time Java είναι μια επέκταση της Java με στόχο να ανταποκρίνεται στις απαιτήσεις συστημάτων πραγματικού χρόνου. Αυτό επιτυγχάνεται εισάγοντας νέους μηχανισμούς και τροποποιώντας τους ήδη υπάρχοντες, ώστε κατά την εκτέλεση η απόδοση να είναι εγγυημένη και προβλέψιμη. Επίσης η εικονική μηχανή Java παρέχει κάποιες επιπλέον επιλογές για τη διαχείριση νημάτων, χωρίς όμως να καταστεί δυνατή η πλήρης διαχείριση από το χρήστη.

Για αυτό το σκοπό δημιουργήθηκε μια κλάση σε Java, η οποία προσφέρει αυτή τη λειτουργικότητα, χρησιμοποιώντας μέσω του Java Native Interface τις κλήσεις που παρέχει το λειτουργικό σύστημα.

Επιπλέον βγαίνει το συμπέρασμα ότι το πρότυπο των Function Blocks είναι κατάλληλο και αρκετά αποτελεσματικό στο σχεδιασμό ενός πολύπλοκου ενσωματωμένου συστήματος, όπως αυτό που εξετάστηκε.

Μια μελλοντική επέκταση είναι να δημιουργηθούν και άλλες κλάσεις οι οποίες θα προσφέρουν περισσότερες λειτουργίες, οι οποίες είναι δεν είναι διαθέσιμες στη βασική βιβλιοθήκη, αλλά όμως παρέχονται από το λειτουργικό σύστημα. Έτσι προγράμματα σε Java θα μπορούν να αξιοποιούν το API του λειτουργικού συστήματος, αν εμφανιστεί αυτή η ανάγκη. Επίσης δεν έχει ελεγχθεί κατά πόσο το σύστημα ανταποκρίνεται στις απαιτούμενες χρονικές προθεσμίες. Σε περίπτωση που χρειάζεται ταχύτερη απόκριση μπορούν να χρησιμοποιηθούν NoHeapRealimeThreads αντί απλών RealtimeThreads και BoundAsyncEventHandlers αντί AsyncEventHandlers.

Αναφορές

[1] Michael Barr, Anthony J. Massa, Programming embedded systems: with C and GNU development tools, O'Reilly, 2006

[2] Barr, Michael. "Embedded Systems Glossary." Online at http://www.netrino.com/Embedded-Systems/Glossary

[3] John L. Hennessy & David A. Patterson, Αρχιτεκτονική Υπολογιστών -3η έκδοση, Εκδόσεις Τζιόλα,

[4] Performance Insights to Intel Hyper-Threading Technology, http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/

[5] The Multicore Challenge, David A. Patterson, http://www.cccblog.org/2008/08/26/the-multicore-challenge

[6] Multi-Core CPUs, Russell Hitchcock, http://www.windowsnetworking.com/articles_tutorials/Multi-Core-CPUs.html

[7] Multicore programming fundamental whitepaper series, http://zone.ni.com/devzone/cda/tut/p/id/6424

[8] Multicore Processors Revolutionize Real-Time Embedded Systems, Paul Fischer, http://electronicdesign.com/article/digital/multicore-processors-revolutionize-real-time-embed.aspx

[9] http://el.wikipedia.org/wiki/Χρονοπρογραμματισμός

[10] http://immike.net/blog/2007/08/01/what-is-the-completely-fair-scheduler/

[11] Silberschatz,Galvin,Gagne, Λειτουργικά συστήματα, Εκδόσεις Ίων

[12] Inside the Linux scheduler, M. Tim Jones http://www.ibm.com/developerworks/linux/library/l-scheduler/

[13] Inside the Linux 2.6 Completely Fair Scheduler, M. Tim Jones http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/index.html

[14] Multiprocessing with the Completely Fair Scheduler, Avinesh Kumar, http://www.ibm.com/developerworks/linux/library/l-cfs/

[15] CPU Affinity, Robert Love, http://www.linuxjournal.com/article/6799

[16] http://msdn.microsoft.com/en-us/library/ms684847(v=VS.85).aspx

[17] taskset(1) - Linux man page http://linux.die.net/man/1/taskset

[18] Concurrent Programming with J2SE 5.0 , *Qusay H. Mahmoud* http://java.sun.com/developer/technicalArticles/J2SE/concurrency/index.html

[19] http://java.sun.com/docs/books/tutorial/essential/concurrency/executors.html

[20] J.M. Bull, M.E. Kambites, JOMP: an OpenMP-like interface for Java

[21] L. Dagum, R. Menon, OpenMP: An Industry-Standard API for Shared-Memory Programming

[23] http://www.rtsj.org/specjavadoc/book_index.html…..

[22] http://www.ibm.com/developerworks/java/library/j-rtj1/index.html

[24] Andy Wellings Concurrent and Real-Time Programming in Java

[25] Μπογιατζή E., "Using IEC61499 in control and automation: The FESTO MPS case study" Διπλωματική εργασία, Απρίλιος 2006

[26] G. Doukas, K. Thramboulidis, "A Real-Time Linux Based Framework for Model-Driven Engineering in Control and Automation", IEEE Transaction on Industrial Electronics

[27] http://seg.ee.upatras.gr/seg/dev/FestoMPS.htm

[28] MinSeong Kim, Andy Wellings, Asynchronous Event Handling in the RTSJ