Κεφάλαιο 4

Διεργασίες και Νήματα

Σύνοψη

Σε αυτό το κεφάλαιο παρουσιάζεται ο κρίσιμος ρόλος που παίζουν οι διεργασίες και τα νήματα στα κατανεμημένα συστήματα. Καταρχάς, γίνεται μια επισκόπηση της έννοιας της διεργασίας. Κατόπιν εισάγονται τα νήματα, ο λόγος δημιουργίας τους και η χρησιμότητά τους. Εξετάζεται η σχέση τους με τα παράλληλα συστήματα και την αποδοτικότητα των κατανεμημένων συστημάτων. Αναλύεται η διαδικασία της δημιουργίας, εκτέλεσης και καταστροφής τους στον πυρήνα ενός λειτουργικού συστήματος, ώστε να εξασφαλίζεται η ομαλή και αποδοτική εκτέλεση μιας εφαρμογής πάνω σε αυτό. Τα νήματα επιδεικνύονται στη γλώσσα προγραμματισμού Java, ενώ στην ίδια γλώσσα προγραμματισμού εξετάζονται και τα θέματα του συγχρονισμού και της επικοινωνίας μεταξύ νημάτων, παρέχοντας κατάλληλα παραδείγματα.

Προαπαιτούμενη Γνώση

1) Δρόσος, Δ., Βουγιούκας, Δ., Καλλίγερος, Ε., Κοκολάκης, Σ., Σκιάνης, Χ. 2015. Υπολογιστικά συστήματα: Δομή, αρχιτεκτονική και λειτουργικά συστήματα. [Κεφάλαιο Συγγράμματος]. Στο Δρόσος, Δ., Βουγιούκας, Δ., Καλλίγερος, Ε., Κοκολάκης, Σ., Σκιάνης, Χ. 2015. *Εισαγωγή στην επιστήμη των υπολογιστών & επικοινωνιών*. [ηλεκτρ. βιβλ.] Αθήνα: Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών. κεφ 2. Διαθέσιμο στο: http://hdl.handle.net/11419/4583

2) Μαυρίδης, Ι. 2015. Έλεγχος Πρόσβασης σε Λειτουργικά Συστήματα. [Κεφάλαιο Συγγράμματος]. Στο Μαυρίδης, Ι. 2015. Εργαστήριο ασφάλειας πληροφοριών και συστημάτων. [ηλεκτρ. βιβλ.] Αθήνα: Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών. κεφ 5. Διαθέσιμο στο: http://hdl.handle.net/11419/530

3) Σιδηρόπουλος, Α. (2015). Εισαγωγή στα λειτουργικά συστήματα. Αθήνα: Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών.

4.1 Η έννοια της διεργασίας

Η διεργασία αποτελεί τη λειτουργία ενός προγράμματος τη στιγμή την οποία αυτό εκτελείται. Δηλαδή διεργασία είναι ένα πρόγραμμα «εν εκτελέσει». Πιο συγκεκριμένα, με τον όρο διεργασία εννοούμε ένα σύνολο πόρων για μια εφαρμογή που εκτελείται, που περιλαμβάνουν την κατάσταση του επεξεργαστή, τη μνήμη και τον χώρο διευθύνσεων, την δικτυακή επικοινωνία, τα οποία αποκτήθηκαν κατά τον χρόνο εκτέλεσης της. Ένα λειτουργικό σύστημα χρειάζεται να διαχειριστεί όλες αυτές τις πληροφορίες αποτελεσματικά και αποδοτικά [2].

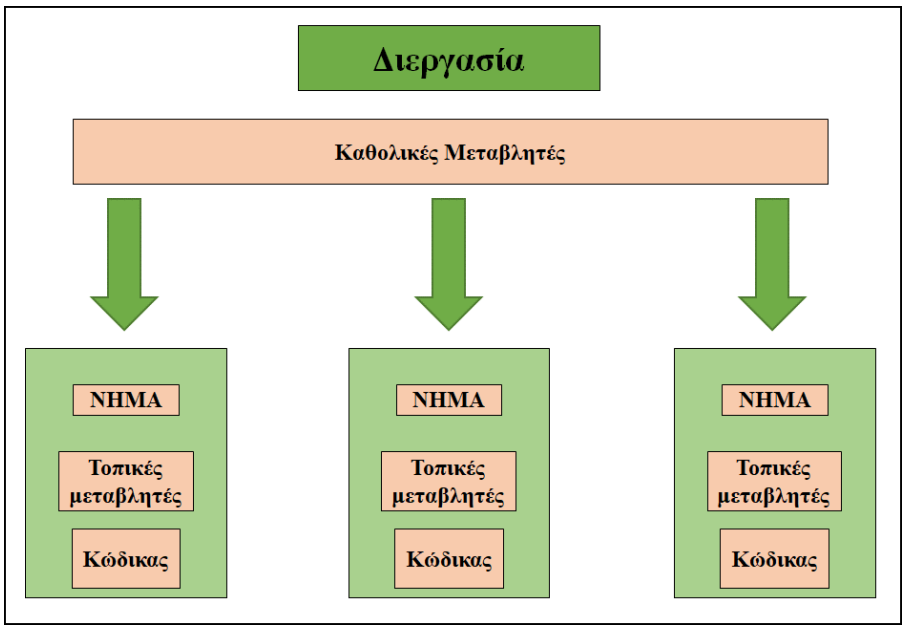
Όταν εκτελείται μια διεργασία, ο πυρήνας του λειτουργικού συστήματος φορτώνει τον κώδικα της διεργασίας στην εικονική μηχανή, δημιουργεί χώρο στη μνήμη για τις διαθέσιμες μεταβλητές και δημιουργεί τις απαραίτητες δομές δεδομένων προκειμένου να καταγραφούν διάφορες πληροφορίες σχετικά με τον αναγνωριστικό αριθμό κάθε διεργασίας, την κατάσταση τερματισμού, το αναγνωριστικό χρήστη και το αναγνωριστικό ομάδας της αντίστοιχης διεργασίας. Ο πυρήνας του λειτουργικού συστήματος διαχειρίζεται τις διεργασίες ως οντότητες, οι οποίες χρειάζεται να διαμοιραστούν τους διαθέσιμους πόρους του υπολογιστή. Με την ολοκλήρωση της εκτέλεσης κάθε διεργασίας, αποδεσμεύονται και οι αντίστοιχοι πόροι προκειμένου να επαναχρησιμοποιηθούν από άλλες διεργασίες [3]. Ο πυρήνας του λειτουργικού συστήματος αποθηκεύει τη λίστα των διεργασιών σε μια κυκλική διπλά συνδεδεμένη λίστα που ονομάζεται λίστα εργασιών του συστήματος. Κάθε στοιχείο της λίστας εργασιών είναι ένας κόμβος με πληροφορίες σχετικά με τη συγκεκριμένη διεργασία. Πιο συγκεκριμένα,

περιέχει δεδομένα σχετικά με τα αρχεία που χρησιμοποιεί η διεργασία, τον χώρο διευθύνσεων, τις εκκρεμείς ενέργειες προς εκτέλεση και την εκάστοτε κατάσταση εκτέλεσης της διεργασίας [2].

Η εκτέλεση κάθε διεργασίας πραγματοποιείται στον χώρο χρήστη (user space) του λειτουργικού συστήματος. Όταν ένα πρόγραμμα εκτελεί μια κλήση συστήματος (system call) οδηγείται στον χώρο του πυρήνα (kernel space) για να διεκπεραιώσει την αντίστοιχη λειτουργία. Ο χώρος του πυρήνα αποτελεί τον συνδετικό κρίκο μεταξύ των λειτουργιών που χρειάζεται να πραγματοποιηθούν κατά την εκτέλεση μιας διεργασίας σε σχέση με αυτές του λειτουργικού συστήματος. Με την έξοδο από τον χώρο του πυρήνα, η διεργασία συνεχίζει την εκτέλεσή της στον χώρο χρήστη, εκτός και εάν μια διεργασία υψηλότερης προτεραιότητας έχει καταφέρει να εκτελεστεί πρώτα. Οι κλήσεις συστήματος αποτελούν τις διεπαφές μέσω τον οποίων δίνεται η δυνατότητα σε μια διεργασία να εκτελεστεί στον πυρήνα του λειτουργικού συστήματος [2].

4.2 Εισαγωγή στα νήματα και η σχέση τους με τη διεργασία

Τα νήματα είναι ένας μηχανισμός που επιτρέπει σε μια διεργασία να εκτελέσει πολλαπλές εργασίες ταυτόχρονα. Όλα τα νήματα εκτελούν ανεξάρτητα τον ίδιο κώδικα και μοιράζονται μεταξύ τους την ίδια μνήμη, συμπεριλαμβανομένων των αρχικοποιημένων δεδομένων, μη αρχικοποιημένων δεδομένων και τμήματα σωρού [3]. Η εκτέλεση ενός προγράμματος προϋποθέτει τη δημιουργία ενός αριθμού από εικονικούς επεξεργαστές, όπου ο καθένας θα είναι υπεύθυνος για την εκτέλεση ενός διαφορετικού κομματιού του προγράμματος. Για την παρακολούθηση των παραπάνω εικονικών επεξεργαστών, το σύστημα διαθέτει ένα πίνακα διεργασιών, ο οποίος περιέχει πληροφορίες σχετικά με την αποθήκευση τιμών στους καταχωρητές της CPU (Central Processing Unit), τη διαθέσιμη μνήμη, τα ανοιχτά αρχεία, καθώς και τα δικαιώματα που έχει το κάθε νήμα [1]. Έτσι, ο κύριος ρόλος του λειτουργικού συστήματος είναι η διασφάλιση των ανεξάρτητων λειτουργιών που πραγματοποιούνται από ένα σύνολο νημάτων, τα οποία δεν πρέπει να επηρεάζουν την αποτελεσματική εκτέλεση του προγράμματος [1].



Σχήμα 4.1 Η σχέση της διεργασίας με τα νήματα

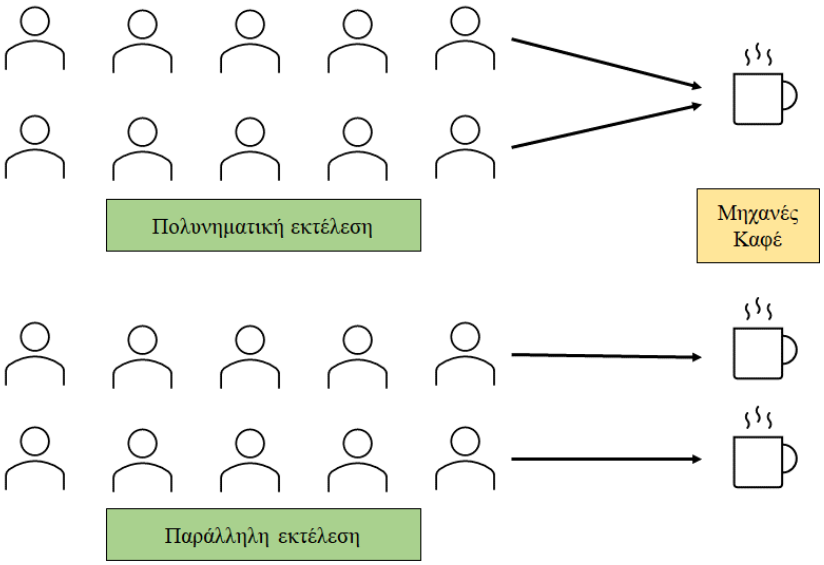
Αν και οι διεργασίες αποτελούν ένα δομικό στοιχείο των λειτουργικών συστημάτων, η ανάγκη για αυξημένη απόδοση και δημιουργία κατανεμημένων εφαρμογών καθιστά τη

χρήση νημάτων απαραίτητη. Το νήμα είναι η μικρότερη μονάδα εκτέλεσης σε μια διεργασία. Ένα νήμα απλά εκτελεί σειριακές οδηγίες. Μια διεργασία μπορεί να έχει αρκετά νήματα που να εκτελούνται ως κομμάτια αυτής. Κάθε νήμα περιλαμβάνει έναν μοναδικό μετρητή προγράμματος, μια στοίβα διεργασιών και ένα σύνολο καταχωρητών του επεξεργαστή [2]. Ο πυρήνας του λειτουργικού συστήματος είναι υπεύθυνος να συντονίζει τα νήματα και όχι οι διεργασίες. Έχοντας μια λεπτομερή εικόνα με την μορφή πολλαπλών νημάτων ελέγχου ανά διεργασία, είναι αρκετά πιο εύκολο να δημιουργήσουμε κατανεμημένες εφαρμογές και να επιτύχουμε υψηλή απόδοση [1]. Η σχήμα 4.1. επιδεικνύει τη σχέση μεταξύ διεργασίας και νημάτων.

4.3 Διαφορές πολυνηματικού και παράλληλου προγραμματισμού

Ο πολυνηματικός και ο παράλληλος προγραμματισμός πολύ συχνά συγχέονται ως προς την ικανότητα του συστήματος να εκτελέσει πολλαπλά και διαφορετικά προγράμματα ταυτόχρονα. Ένα πρόγραμμα το οποίο μπορεί να εκτελεστεί πολυνηματικά είναι αυτό το οποίο μπορεί να αποσυντεθεί σε μικρότερα συστατικά μέρη, χωρίς να επηρεάσει το τελικό αποτέλεσμα. Ένα σύστημα ικανό να εκτελεί πολλά διακριτά προγράμματα ή μονάδες αυτού σε αλληλεπικαλυπτόμενα χρονικά διαστήματα ονομάζεται ταυτόχρονο (concurrent) πρόγραμμα. Ένα σύστημα με πολυνηματικό προγραμματισμό έχει ως κύριο στόχο να μεγιστοποιήσει την απόδοση και να ελαχιστοποιήσει την καθυστέρηση στην απάντηση προς τον χρήστη της εφαρμογής.

Ένα παράλληλο πρόγραμμα έχει την δυνατότητα εκτέλεσης πολλαπλών προγραμμάτων ταυτόχρονα. Συνήθως, αυτό υποστηρίζεται από επεξεργαστές πολλών πυρήνων σε μεμονωμένα μηχανήματα ή από υπολογιστικές μονάδες, όπου πολλές μηχανές ενώνονται για να λύσουν ανεξάρτητα κομμάτια ενός προβλήματος παράλληλα. Στα παράλληλα συστήματα δίνεται έμφαση στην αύξηση της απόδοσης και της βελτιστοποίησης της χρήσης των πόρων του υλικού. Ο στόχος είναι να επιτευχθεί όσο το δυνατόν μεγαλύτερη ταχύτητα υπολογισμού.



Σχήμα 4.2 Διαφορά πολυνηματικού με τον παράλληλο προγραμματισμό.

Από το σχήμα 4.2 είναι εμφανές ότι ένα ταυτόχρονο σύστημα δεν χρειάζεται να είναι παράλληλο, ενώ ένα παράλληλο σύστημα είναι ταυτόχρονο. Επίσης, ένα σύστημα θα μπορούσε να είναι αφενός παράλληλο και ταυτόχρονο (πολυνηματικό), όπως π.χ. ένα λειτουργικό σύστημα πολλαπλών εργασιών που εκτελείται σε ένα μηχάνημα πολλαπλών πυρήνων.

4.3.1 Η αναγκαιότητα των νημάτων στην αποδοτικότητα των κατανεμημένων συστημάτων

Τα νήματα προσφέρουν πλεονεκτήματα έναντι των διεργασιών κατά την εκτέλεση αρκετών εφαρμογών. Κύριο παράδειγμα αποτελεί η ταυτόχρονη δημιουργία πολλαπλών διεργασιών στο UNIX. Ένα σχετικό παράδειγμα, είναι ο σχεδιασμός ενός διακομιστή δικτύου στον οποίο μια γονική διεργασία δέχεται εισερχόμενες συνδέσεις από πελάτες και στη συνέχεια δημιουργεί μια ξεχωριστή θυγατρική διεργασία με την χρήση του fork(), ώστε να διαχειριστεί την επικοινωνία με κάθε πελάτη. Το σενάριο που περιεγράφηκε παραπάνω έχει τους εξής περιορισμούς [3]:

- Είναι δύσκολο να μοιράσουμε πληροφορίες μεταξύ των διαδικασιών. Δεδομένου ότι η διεργασία γονέας και η διεργασία παιδί δεν έχουν κοινή πρόσβαση στη διαθέσιμη μνήμη, χρειάζεται να χρησιμοποιηθεί κάποια μορφή επικοινωνίας μεταξύ των διεργασιών για την ανταλλαγή πληροφορίας.
- Η διαδικασία δημιουργίας μιας διεργασίας με την χρήση του fork() είναι ακριβή υπολογιστικά και χρονοβόρα.

Τα νήματα αντιμετωπίζουν και τα δύο αυτά προβλήματα με τους εξής τρόπους:

- Η κοινή χρήση πληροφοριών μεταξύ νημάτων είναι εύκολη και γρήγορη. Τα νήματα κάνουν χρήση της κοινής διαμοιραζόμενης μνήμης τους, με αποτέλεσμα η επικοινωνία μεταξύ τους να συνοψίζεται στην απλή αντιγραφή δεδομένων σε κοινόχρηστες μεταβλητές ή δομές δεδομένων.
- Η δημιουργία νήματος είναι ταχύτερη από αυτή της διεργασίας. Πολλά από τα χαρακτηριστικά που χρειάζεται να αναπαραχθούν σε μια διεργασία κατά την δημιουργία της, διαμοιράζονται στην αντίστοιχη περίπτωση των νημάτων [3].

Γενικότερα, με την χρήση πολυνηματικού προγραμματισμού μας δίνεται η δυνατότητα να εκμεταλλευτούμε την εκτέλεση του προγράμματος και σε ένα σύστημα πολλαπλών επεξεργαστών. Πιο συγκεκριμένα, κάθε νήμα αντιστοιχεί σε ξεχωριστό πυρήνα της CPU, ενώ τα κοινόχρηστα δεδομένα αποθηκεύονται στη διαμοιραζόμενη κοινή μνήμη. Επομένως, ο σωστός σχεδιασμός ενός τέτοιου προγράμματος κάνει χρήση των δυνατοτήτων που υπάρχουν τόσο στο κομμάτι της πολυνηματικής, όσο και της παράλληλης εκτέλεσης του [3].

4.4 Νήματα στη Java

Σε αυτό το κεφάλαιο, παρέχονται όλες η απαραίτητη πληροφορία, προκειμένου να δημιουργήσει κάποιος βασικά νήματα. Κάθε πρόγραμμα έχει τουλάχιστον ένα νήμα το οποίο εκτελείται σειριακά. Η γλώσσα προγραμματισμού Java μάς δίνει την δυνατότητα να αναπτύξουμε εφαρμογές που να υποστηρίζουν την ταυτόχρονη εκτέλεση των εντολών τους για την αποτελεσματική και αποδοτική λειτουργία τους. Τα νήματα μπορούν να δημιουργηθούν είτε χρησιμοποιώντας την κλάση Thread είτε με την χρήση της διεπαφής Runnable [4].

Το προτιμώμενο μέσο για τη δημιουργία πολυνηματικών (multithreaded) εφαρμογών στη Java είναι με τη διεπαφή (interface) Runnable. Ένα αντικείμενο Runnable αντιπροσωπεύει μια "διεργασία" που μπορεί να εκτελεστεί ταυτόχρονα με άλλες διεργασίες. Η διεπαφή Runnable δηλώνει μια ενιαία μέθοδο, τη run, η οποία περιέχει θα πρέπει να εκτελέσει τον κώδικα που το αντικείμενο Runnable . Όταν εκτελείται ένα thread, δημιουργείται και ξεκινάει ένα Runnable, το νήμα καλεί την μέθοδο run του αντικειμένου Runnable, η οποία εκτελείται στο νέο νήμα [4].

Παρακάτω παρουσιάζονται δύο παραδείγματα, όπου προσδιορίζονται οι κύριοι τρόποι δημιουργίας νημάτων στη γλώσσα προγραμματισμού Java. Στην πρώτη περίπτωση (κώδικας 4.1) δηλώνουμε την κλάση, η οποία κληρονομεί από την Thread, υλοποιείται η μέθοδος run() και το νήμα ξεκινά με την μέθοδο start(). Η εντολή object.start() οδηγεί το νήμα σε κατάσταση εκτέλεσης και εκτελείται το κομμάτι του κώδικα της μεθόδου run(). Στη δεύτερη περίπτωση (κώδικας 4.2), μια κλάση η οποία υλοποιεί το interface Runnable, δεν είναι νήμα. Αντιθέτως, πρέπει να δημιουργηθεί μια οντότητα του νήματος και να περασθεί ως όρισμα [4].

```
class MultithreadingDemo extends Thread {

      public void run() {

            try {

                  // Εμφανίζει τον νήμα που «τρέχει»
                  System.out.println("Thread " + Thread.cur
rentThread().getId() + " is running");

            }

            catch(Exception e)

            {

                  //όταν «πετάξει» μία εξαίρεση
                  System.out.println ("Exception is caught");

            }
      }
}

public class Multithread {

      public static void main(String[] args)

      {

            int n = 10; //Αριθμός των νημάτων;

            for(int i=0; i<10; i++)

            {

                  MultithreasDemo object = new MultithreadDemo();

                  object.start();
            }
      }
}
```

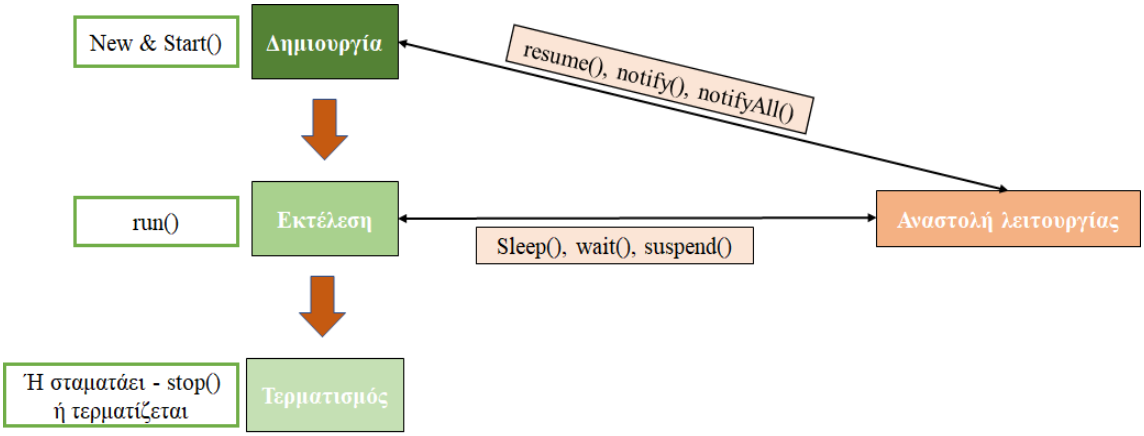
Κώδικας 4.1: Δημιουργία νήματος

```
class MultithreadingDemo implements Runnable {

      public void run () {

            try{

                  // Εμφανίζει το νήμα που «τρέχει»
                  System.out.println("Thread " + Thread.cur
rentThread().getId() + " is running");
            }

            catch (Exception e)

            {

                   // Όταν «πετάξει» μία εξαίρεση
                  System.out.println ("Exception is caught");

            }
      }
}

class Miltithread {

      public static void main (String[] args)

      {

            int n = 10; // Αριθμός των νημάτων

            for (int i=0; i<10; i++)

            {

                Thread object = new Thread (new Multithreading
Demo());

                  object.start();
            }
      }
}
```

Κώδικας 4.2: Δημιουργία νήματος στη JAVA (δεύτερος τρόπος)

4.4.1 Ο κύκλος ζωής ενός νήματος

Στην συνέχεια παρουσιάζεται ο κύκλος ζωής ενός νήματος και των αντίστοιχων μεθόδων που εκτελούνται καθ' όλη τη διάρκεια της εκτέλεσής του (σχήμα 4.3). Αναλύονται τα βασικά στάδια από την στιγμή που δημιουργείται μέχρι να τερματίσει.



Σχήμα 4.3: Απεικόνιση του κύκλου ζωής του νήματος

Αναλυτικότερα, τα στάδια του κύκλου ζωής του νήματος έχουν ως εξής:

1) Δημιουργία νήματος

Η πρώτη φάση του κύκλου ζωής κάθε νήματος είναι η δημιουργία του. Κάθε νήμα αποτελεί ένα instance της κλάσης Thread. Κάθε νήμα (thread) διαθέτει το δικό του όνομα βάσει του οποίου η εικονική μηχανή (Java Virtual Machine - JVM) μπορεί να το προσδιορίσει μοναδικά. Το πιο σύνηθες είναι να αποτελείται [4] από πληροφορίες σχετικά με την προτεραιότητα (priority) του και την ομάδα του νήματος (thread group) [4]. Η προτεραιότητα κάθε νήματος είναι ένας ακέραιος αριθμός που προσδιορίζει την σειρά εκτέλεσης με την οποία κάθε νήμα θα εκτελέσει την λειτουργία του.

2) Έναρξη νήματος

Ένα νήμα το οποίο έχει μόλις κατασκευαστεί δεν σημαίνει πως εκτελείται. Αντιθέτως, βρίσκεται σε κατάσταση αναμονής. Αυτό σημαίνει πως πληροφορίες σχετικά με το νήμα, όπως η προτεραιότητα και το όνομά του είναι δυνατόν να τροποποιηθούν ύστερα από επικοινωνία με τα υπόλοιπα νήματα της εφαρμογής. Η μέθοδος start() είναι αυτή που πραγματοποιεί την έναρξη της εκτέλεσης του κώδικα του νήματος. Η μέθοδος αυτή έχει ως αποτέλεσμα να μεταβεί το νήμα σε κατάσταση εκτέλεσης. Πιο συγκεκριμένα, η κλάση Thread έχει διαθέσιμη τη μέθοδο isAlive(), βάσει της οποίας μπορούμε να επαληθεύσουμε εάν το νήμα είναι σε κατάσταση εκτέλεσης, σε κατάσταση αναμονής ή έχει τερματίσει [4].

3) Εκτέλεση νήματος

Μόλις ξεκινήσει την εκτέλεσή του ένα νήμα, πραγματοποιεί τις ενέργειες που περιγράφονται βάσει των αντίστοιχων εντολών στο κομμάτι της μεθόδου run().

4) Τερματισμός νήματος

Με την ολοκλήρωση της μεθόδου run(), το νήμα τελειώνει την εκτέλεσή του.

Η τεκμηρίωση (documentation) της κλάσης Thread παρέχει τη χρήση της μεθόδου stop() για τον τερματισμό ενός νήματος. Παρά ταύτα, καθώς υπάρχει ένα εγγενές πρόβλημα (race condition) με τη χρήση της συνάρτησης και ως εκ τούτου προτείνεται να αποφεύγεται [4].

Ο κώδικας 4.3 δείχνει τις συναρτήσεις που χρησιμοποιούνται για την υλοποίηση του κύκλου ζωής του νήματος.

```
package java.lang;

public class Thread implements Runnable {

      public void start();
      public void run();
      public void stop();
      public void resume();
      public void suspend();
      public static void sleep(long millis);
      public static void sleep(long millis, int nanos);
      public boolean isAlive();
      public void interrupt();
      public boolean interrupted();
      public void join() throws InterruptedException;
}
```

Κώδικας 4.3: Τεχνικές συναρτήσεις που υλοποιούν τον κύκλο ζωής του νήματος.

5) Παύση, ανατολή και επανέναρξη νημάτων

Κάθε νήμα το οποίο έχει αρχίσει να εκτελείται ενδέχεται να παύσει την εκτέλεσή του για κάποιο χρονικό διάστημα και να συνεχίσει τη λειτουργία του αργότερα. Δηλαδή, το νήμα ενώ εκτελείται, μπορεί να πέσει σε κατάσταση λήθαργου ή να μπλοκαριστεί για κάποιους λόγους ή να περιμένει την «κλειδαριά» ενός κοινού πόρου. Συνεχίζει την εκτέλεσή του μετά από το ξεμπλοκάρισμά του. Η κλάση Thread παρέχει τις μεθόδους suspend() και resume(), οι οποίες λόγω του προβλήματος του «race condition» που αντιμετωπίζουν, δεν ενδείκνυται να χρησιμοποιούνται. Πιο συγκεκριμένα, η μέθοδος sleep() αποτελεί έναν τρόπο κατά τον οποίο ένα νήμα μπορεί να αναστείλει την λειτουργία του για ένα καθορισμένο χρονικό διάστημα. Με το πέρας του αντίστοιχου χρονικού διαστήματος, το νήμα τίθεται ξανά σε κατάσταση εκτέλεσης. Τα νήματα μπορούν να χρησιμοποιήσουν τις μεθόδους wait(), notify() ή signal() προκειμένου να επιτύχουν την λειτουργία της αναστολής και επανέναρξής τους [4]. Ο τρόπος με τον οποίο μπορούν να χρησιμοποιηθούν περιγράφεται στην συνέχεια του κεφαλαίου.

4.5 Συγχρονισμός Νημάτων

Οι προγραμματιστές κάθε εφαρμογής θα πρέπει να διασφαλίζουν ότι οι κοινόχρηστοι πόροι θα πρέπει να προστατεύονται από την ταυτόχρονη πρόσβαση σε αυτούς. Το ενδεχόμενο δύο ή περισσότερα νήματα να έχουν ταυτόχρονη πρόσβαση στους κοινόχρηστους πόρους του συστήματος μπορεί να οδηγήσει την εφαρμογή σε αναξιόπιστο, ασταθή και αβέβαιο τρόπο λειτουργίας. Η προστασία λοιπόν των κοινόχρηστων και διαμοιραζόμενων σημείων του συστήματος αποτελεί καθοριστικό ρόλο στην αποτελεσματική και αποδοτική λειτουργία της εφαρμογής [2]. Το πρόβλημα αυτό ονομάζεται race condition και ο συγχρονισμός των νημάτων έχει ως στόχο την επίλυσή του. Παρακάτω παρουσιάζονται δύο τρόποι συγχρονισμού των νημάτων κατά την εκτέλεσή τους. Αυτοί είναι τα mutexes και τα monitors.

Το mutex, όπως υποδηλώνει και το όνομά του, σχετίζεται άμεσα με τον αμοιβαίο αποκλεισμό (mutual exclusion). Ένα mutex χρησιμοποιείται για τη φύλαξη των κοινόχρηστων δεδομένων μιας εφαρμογής. Πιο συγκεκριμένα, μόλις ένα νήμα αποκτήσει ένα mutex, διασφαλίζεται ότι κανένα άλλο νήμα δεν μπορεί να εκτελέσει το κοινόχρηστο κομμάτι του κώδικα χωρίς να έχει προηγηθεί η απελευθέρωση του mutex από το νήμα που αρχικά το δέσμευσε [2].

Το monitor, από την άλλη μεριά, αποτελεί έναν μηχανισμό συγχρονισμού μεταξύ των νημάτων της εφαρμογής σε υψηλότερο επίπεδο. Ενώ τα mutexes αποτελούν κομμάτι του λειτουργικού συστήματος και παρέχονται στην εφαρμογή με σκοπό τον συγχρονισμό των νημάτων, τα monitors, που θα μελετήσουμε παρακάτω, είναι συστατικό στοιχείο της γλώσσας προγραμματισμού Java για την αποτελεσματική εκτέλεση πολυνηματικών εφαρμογών [2].

Συνήθως, σε εφαρμογές πολυνηματικού προγραμματισμού, είναι αρκετά συχνό φαινόμενο ένα νήμα να περιμένει την εκτέλεση κάποιου άλλου νήματος. Ένα τέτοιο παράδειγμα είναι αυτό του παραγωγού/καταναλωτή (producer/consumer). Εάν ο παραγωγός δεν έχει παραγάγει κάτι, ο καταναλωτής χρειάζεται να περιμένει έως ότου υπάρχουν δεδομένα τα οποία μπορεί να αξιοποιήσει. Το νήμα του καταναλωτή λοιπόν ελέγχει επαναλαμβανόμενα έναν βρόχο μέχρι να γίνει η συνθήκη αληθής, όπως παρουσιάζεται παρακάτω [2].

```
void busyWaitFunction() {

	// acquire mutex
	while(predicate is false) {

		// release mutex
		// acquire mutex
	}

	// do something useful

	// release mutex
}
```

Κώδικας 4.4. Η χρήση της acquire mutex και release mutex στο συγχρονισμό των νημάτων.

Στον κώδικα 4.4 παρατηρούμε ότι μέσα στην βρόχο αποδεσμεύεται το mutex, δίνοντας έτσι την ευκαιρία στα υπόλοιπα νήματα να εκτελέσουν τις εντολές τους, καθώς, επίσης, δεσμεύεται το mutex για να πραγματοποιηθεί ο έλεγχος του βρόχου. Είναι προφανές ότι η λειτουργία της παραπάνω συνάρτησης ολοκληρώνεται με την εκτέλεση του κρίσιμου κομματιού της (critical section) και την αποδέσμευση του mutex. Το πρόβλημα όμως που δημιουργείται είναι αυτό του “spin waiting” ή “busy waiting”, όπου σπαταλούνται αρκετοί κύκλοι της CPU πραγματοποιώντας τον άκομψο επαναλαμβανόμενο έλεγχο του βρόχου. Οι μεταβλητές συνθήκης (condition variables) επιλύουν αποτελεσματικά την κατάσταση κατά την οποία ένα νήμα αναμένει να δεσμεύσει το mutex και να εκτέλεσει τη λειτουργία του ενώ κάποιο άλλο νήμα ήδη εκτελείται. Κάθε μεταβλητή συνθήκης διαθέτει τις μεθόδους wait() και signal().

Η μέθοδος wait() προκαλεί την αποδέσμευση του αντίστοιχου mutex και το νήμα τοποθετείται σε μια ουρά αναμονής (waiting queue). Προφανώς, θα μπορούσαν να υπάρχουν και άλλα νήματα τα οποία περιμένουν στην ουρά αναμονής. Έτσι δίνεται η δυνατότητα σε άλλα νήματα της εφαρμογής να αποκτήσουν το διαθέσιμο mutex και να εκτελέσουν τη λειτουργία τους, ενώ η εφαρμογή παραμένει συγχρονισμένη.

Η μέθοδος signal() μιας μεταβλητής συνθήκης έχει ως αποτέλεσμα να ετοιμάσει ένα από τα νήματα στην ουρά αναμονής, ώστε να συνεχίσει την εκτέλεσή του. Πιο συγκεκριμένα, το νήμα το οποίο έχει ειδοποιηθεί από την ουρά αναμονής δεν έχει αρχίσει να εκτελείται, αφού δεν έχει αποδεσμευτεί το αντίστοιχο mutex από το νήμα που έδωσε το σήμα [2]. Στην συνέχεια, όμως, δεσμεύεται το mutex και εκτελείται κανονικά (κώδικας 4.5).

```
void effecientWaitingFunction () {

      mutex.acquire()
      while (predicate == false) {

            condVar.wait()
      }
      // do something useful
      mutex.release()

}

void changePredicate () {

      mutex.acquire();
      set predicate = true
      condVar.signal()
      mutex.release()
}
```

Κώδικας 4.5: Συνάρτηση που επιλύει το πρόβλημα “spin waiting”

Στον κώδικα 4.6 παρουσιάζεται ο τρόπος συγχρονισμού δύο νημάτων με τη χρήση monitor. Είναι εμφανές ότι κάθε monitor αποτελείται από ένα mutex και μία ή περισσότερες condition variables. Σε αυτή την κατεύθυνση, η γλώσσα προγραμματισμού Java παρέχει ορισμένους μηχανισμούς με τους οποίους μπορούν να συγχρονιστούν δύο η περισσότερα νήματα. Με τη χρήση της λέξης-κλειδί synchronized επιτυγχάνεται η πρόσβαση ενός μόνο νήματος κάθε φορά για την εκτέλεση του διαμοιραζόμενου κώδικα και του κοινού πόρου. Ο τρόπος με τον οποίο λειτουργεί η λέξη-κλειδί synchronized είναι πανομοιότυπος με τη χρήση του τρόπου που περιγράφηκε παραπάνω. Μόλις αποκτήσει ένα νήμα το mutex του αντικειμένου, έχει τη δυνατότητα να εκτελέσει τις μεθόδους που δηλώνονται ως synchronized. Κανένα άλλο νήμα δεν μπορεί την ίδια στιγμή να έχει πρόσβαση σε αυτές τις διαμοιραζόμενες εντολές χωρίς να έχει αποδεσμευτεί προηγουμένως το mutex του νήματος που αρχικά αποκτήθηκε. Επομένως, όταν ένα νήμα εκτελεί μια μέθοδο η οποία έχει δηλωθεί ως synchronized, τα υπόλοιπα νήματα που προσπαθούν να εκτελέσουν την ίδια μέθοδο χρειάζεται να περιμένουν μέχρι να το αρχικό νήμα να ολοκληρώσει την εκτέλεσή του [2].

```
class SyncThread {
	synchronized public void getSyncThread() {
		for (int i=0; i<10; i++) {
			System.out.println(i);

			try {
				Thread.sleep(1000);

			}
			catch (Exception e) {

				System.out.println(e);
			}
		}
	}
}

class MyThread extends Thread {

	SyncThread syncthread;

	MyThread(SyncThread syncthread) {

		this. syncthread = syncthread;
	}

	@override
	public void run(){
		syncthread.getSyncthread();
	}
}
```

```
public class SyncEx1 {

	public static void main (String[] args) {

		SyncThread obj = new SyncThread();

// Αντικείμενο της SyncThread class το οποίο διαμοιράζεται μεταξύ των
νημάτων

		MyThread mythread1 = new MyThread(obj);
		MyThread mythread 2 = new MyThread(obj);

		mythread1.start();
		mythread2.start();
	}
```

Κώδικας 4.6. Συνάρτηση συγχρονισμού νημάτων σε Java με την χρήση της λέξης κλειδιού synchronized.

Έτσι στον κώδικα 4.6 παρουσιάζεται ο τρόπος με τον οποίο συγχρονίζονται τα νήματα στη γλώσσα προγραμματισμού Java με τη χρήση της λέξη-κλειδίsynchronized. Η μέθοδος `getSyncThread()` η οποία έχει δηλωθεί ως synchronized επιτρέπει σε ένα μόνο νήμα κάθε φορά να αποκτά πρόσβαση σε αυτή. Άρα, δεν θα υπάρχει ασυνέπεια ως προς την τιμή της μεταβλητής i που θα εκτυπώνει η μέθοδος, καθώς) έχει επιλυθεί το πρόβλημα της συνθήκης ανταγωνισμού (race condition [4].

Ένας άλλος μηχανισμός για τον συγχρονισμό μεταξύ των νημάτων στη Java πραγματώνεται με τη χρήση της λέξης volatile. Κάθε μεταβλητή η οποία έχει δηλωθεί ως volatile και χρειάζεται να ανακτηθεί από την κύρια μνήμη ή να αποθηκευτεί τιμή σε αυτή, παύει να προκαλεί το πρόβλημα του race condition μεταξύ των νημάτων της εφαρμογής. Πιο συγκεκριμένα, ο συγχρονισμός μεταξύ των νημάτων επιτυγχάνεται στην περίπτωση που οι ενέργειες που πραγματοποιούνται στην αντίστοιχη μεταβλητή είναι ατομικές. Με τον όρο ατομικές εννοούμε ότι οι μοναδικές πράξεις που μπορεί να εφαρμοστούν στις μεταβλητές αυτές είναι η ανάκτηση (load) και η αποθήκευση (store) στην κύρια μνήμη. Π.χ., λειτουργίες όπως η αύξηση και η μείωση μιας volatile μεταβλητής δεν επιτρέπονται, καθώς η ενέργεια τόσο της πρόσθεσης όσο και της αφαίρεσης εμπεριέχουν τις εξής τρείς ενέργειες [4]:

1. Την ανάκτηση της μεταβλητής από την κύρια μνήμη
2. Την αύξηση ή μείωση της τιμής της
3. Την αποθήκευση της τελικής τιμής στην μνήμη

Είναι εμφανές πως η χρήση των *volatile* μεταβλητών αποτελεί έναν κύριο τρόπο συγχρονισμού μεταξύ των νημάτων. Εάν, για παράδειγμα, ένα νήμα αναλαμβάνει την αποθήκευση της μεταβλητής στη μνήμη και τα υπόλοιπα νήματα κάνουν φόρτωση της μεταβλητής από τη μνήμη, τότε η δήλωση της μεταβλητής ως volatile αρκεί για να συγχρονίσουμε την εκτέλεση των νημάτων [4]. Ωστόσο, εάν στο παραπάνω σενάριο υπήρχε η δυνατότητα πολλών νημάτων να αποθηκεύουν την τιμή της μεταβλητής τους στην μνήμη, ο συγχρονισμός των νημάτων δεν θα μπορούσε να επιτευχθεί με τη χρήση του volatile λόγω της απροσδιοριστίας της αποθηκευμένης τιμής της μεταβλητής.

4.5.1 Παράδειγμα συγχρονισμού νημάτων

Στον κώδικα 4.7 παρουσιάζεται ένα παράδειγμα που τονίζει την προσοχή που πρέπει να δίνεται όταν υπάρχει πρόσβαση σε κοινόχρηστα δεδομένα σε ένα πολυνηματικό περιβάλλον εκτέλεσης μιας εφαρμογής. Ο ανακριβής συγχρονισμός μεταξύ των νημάτων μπορεί να οδηγήσει σε τελείως διαφορετικό αποτέλεσμα εξόδου της εφαρμογής ανάλογα με την σειρά εκτέλεσης των νημάτων [3].

```
1. int counter = 0;
2.
3. void incrementCounter() {
4.     counter ++;
}
```

Κώδικας 4.7 Προβληματική εκτέλεση ελλείψει συγχρονισμού των νημάτων.

Η εντολή που πραγματοποιείται στη γραμμή 4 μπορεί να αποσυντεθεί στα εξής βήματα:

- Διάβασε την τιμή της μεταβλητής από τον καταχωρητή της κύριας μνήμης στον οποίο βρίσκεται.
- Πρόσθεσε την τιμή ένα στην μεταβλητή που μόλις διαβάστηκε.
- Αποθήκευσε το αποτέλεσμα της νέας τιμής στον καταχωρητή της κύριας μνήμης.

Υποθέτουμε ότι έχουμε δύο νήματα που προσπαθούν να εκτελέσουν την ίδια λειτουργία incrementCounter(). Το πιθανό σενάριο εκτέλεσης των δύο νημάτων φαίνεται παρακάτω.

Θεωρούμε το πρώτο νήμα T1 και το δεύτερο T2, καθώς και την τιμή του μετρητή ίσο με επτά.

- Το Τ1 είναι προγραμματισμένο στην CPU και εισέρχεται στη συνάρτηση. Εκτελεί το βήμα Α, δηλαδή διαβάζει την τιμή της μεταβλητής από το μητρώο που είναι επτά.
- Το λειτουργικό σύστημα αποφασίζει να κάνει context switch και να φέρει στο προσκήνιο το νήμα Τ2.
- Το νήμα Τ2 ολοκληρώνει τα τρία βήματα που περιγράφονται παραπάνω και στη συνέχεια γίνεται context switch από το Τ1. Επομένως, έχει αποθηκεύσει την τιμή οκτώ στην διαμοιραζόμενη μεταβλητή.
- Το νήμα Τ1 επιστρέφει στο προσκήνιο, ώστε να εκτελέσει το υπόλοιπο κομμάτι του, εξακολουθώντας να έχει την παλιά τιμή της μεταβλητής που είναι επτά. Δεν γνωρίζει ότι έχει πραγματοποιηθεί αλλαγή της μεταβλητής. Έτσι λοιπόν, αποθηκεύει την νέα τιμή αντικαθιστώντας την προϋπάρχουσα τιμή, αφού δεν γνωρίζει ότι είχε προηγηθεί context switch.

Με τον όρο context switch εννοούμε τη διαδικασία της αποθήκευσης της κατάστασης μιας διεργασίας ή ενός νήματος προκειμένου να έχει τη δυνατότητα να συνεχίσει την εκτέλεση της σε μεταγενέστερο χρονικό σημείο. Αυτό αποτελεί ένα βασικό χαρακτηριστικό ενός λειτουργικού συστήματος, το οποίο επιτρέπει σε πολλές διεργασίες να μοιράζονται τους πόρους της CPU [3].

Εάν τα νήματα εκτελούνταν σειριακά, η τελική τιμή θα ήταν εννέα. Το πρόβλημα που δημιουργήθηκε είναι ξεκάθαρο. Είναι αναγκαία η προστασία των διαμοιραζόμενων μεταβλητών ή των δομών δεδομένων μεταξύ των νημάτων, προκειμένου να μην δημιουργούνται σφάλματα κατά την εκτέλεση της εφαρμογής. Δεδομένου ότι η εκτέλεση των νημάτων εξαρτάται αποκλειστικά από το λειτουργικό σύστημα, δεν μπορούμε να προβλέψουμε τον τρόπο εκτέλεσης του προγράμματος σε σχέση με την σειρά εκτέλεσης των νημάτων [3].

4.6 Επικοινωνία Νημάτων

Στο προηγούμενο κεφάλαιο, παρουσιάστηκαν η σημασία και οι μηχανισμοί συγχρονισμού μέσω των οποίων μπορούμε να πετύχουμε τη σωστή και αποδοτική εκτέλεση του προγράμματος με την χρήση νημάτων. Πιο συγκεκριμένα, η ανάγκη να διασφαλίσουμε τον ασφαλή διαμοιρασμό των κοινών δεδομένων μεταξύ των νημάτων καθιστά τον συγχρονισμό μεταξύ των νημάτων καθοριστικό στοιχείο. Ωστόσο, η αποδοτική και αποτελεσματική εκτέλεση των νημάτων προϋποθέτει εκτός από το κομμάτι του συγχρονισμού και την ασφαλή επικοινωνία μεταξύ τους αναγκαία.

Έστω ότι ένα νήμα χρειάζεται μια συγκεκριμένη συνθήκη προκειμένου να ξεκινήσει ή να συνεχίσει την εκτέλεσή του. Ένα άλλο νήμα είναι αυτό που θα δημιουργήσει αυτή την κατάσταση, ώστε να επιτευχθεί ο συντονισμός και κατ' επέκταση η σωστή επικοινωνία μεταξύ τους. Αυτή είναι η βασική ιδέα μέσω της οποίας θα επιδιώξουν να επικοινωνήσουν δύο ή περισσότερα νήματα, χρησιμοποιώντας τις μεθόδους που παρουσιάζονται παρακάτω [2]:

- void wait()
- void wait(long timeout)
- void notify()
- void notifyAll()

Η μέθοδος wait() είναι υπεύθυνη να αναστείλει την εκτέλεση του νήματος μέχρι να δοθεί σήμα από κάποιο άλλο νήμα (notify()) που σηματοδοτεί την υλοποίηση μιας απαραίτητης ενέργειας. Είναι σημαντικό να εκτελεστεί σε μια συγχρονισμένη μέθοδο, ώστε να αποφευχθεί ενδεχομένως μια κατάσταση race condition. Με την εκτέλεση της wait(), το αντίστοιχο νήμα τοποθετείται σε ουρά αναμονής μεταβαίνοντας σε μη ενεργή κατάσταση εκτέλεσης. Το νήμα

θα συνεχίσει να βρίσκεται σε κατάσταση αναστολής έως ότου του ξαναδοθεί η δυνατότητα να συνεχίσει την εκτέλεσή του. Η συνάρτηση wait(long timeout) είναι παρόμοια με την wait() με τη μόνη διαφορά ότι θα επανέλθει σε κατάσταση εκτέλεσης ύστερα από ένα δεδομένο χρονικό όριο το οποίο παίρνει ως παράμετρο.

Κύριος ρόλος της μεθόδου notify() είναι να ειδοποιήσει ένα νήμα το οποίο βρίσκεται σε κατάσταση αναμονής, ώστε να συνεχίσει την λειτουργία του. Η συνάρτηση αυτή είναι απαραίτητο να βρίσκεται σε ένα συγχρονισμένο κομμάτι κώδικα προκειμένου να αποτραπούν καταστάσεις race condition μεταξύ των νημάτων. Έτσι, με τη χρήση της δίνεται η δυνατότητα να ενεργοποιηθεί κάποιο από τα νήματα που βρίσκονται στην ουρά αναμονής. Επιπρόσθετα, η συνάρτηση notifyAll() έχει ακριβώς την ίδια λειτουργικότητα με την notify() με τη μόνη προσθήκη ότι ειδοποιεί όλα τα νήματα που βρίσκονται σε κατάσταση αναμονής. Πιο συγκεκριμένα, αδειάζει τη διαθέσιμη ουρά αναμονής, ώστε κάθε νήμα να συνεχίσει κανονικά την εκτέλεσή του.

Ο σκοπός της χρήσης των συναρτήσεων που περιγράφηκαν παραπάνω είναι η συγχρονισμένη επικοινωνία μεταξύ των νημάτων. Το πρόβλημα του συγχρονισμού μεταξύ των νημάτων δεν μπορεί να επιλυθεί με τις συναρτήσεις αυτές. Στην πραγματικότητα, η ασφαλής επικοινωνία μεταξύ των νημάτων πρέπει να γίνεται σε ένα συγχρονισμένο περιβάλλον εκτέλεσης. Δεν είναι δυνατή η επίλυση μιας κατάστασης race condition μεταξύ των νημάτων με τη χρήση των συναρτήσεων wait() και notify().

Με την εκτέλεση του νήματος καταναλωτή, εκτυπώνεται το μήνυμα «Consumer is now waiting on the key» και στη συνέχεια μπαίνει σε κατάσταση αναμονής με την κλήση της μεθόδου wait(). Ο ρόλος του νήματος του παραγωγού είναι να ειδοποιήσει το νήμα που περιμένει, δηλαδή το νήμα καταναλωτή, με τη χρήση της μεθόδου notify(). Ειδικότερα, το νήμα του παραγωγού πραγματοποιεί τη μέθοδο sleep(), προκειμένου να προσομοιώσει μια λειτουργία προς εκπλήρωση και στην συνέχεια ειδοποιεί το νήμα που αναμένει στην ουρά αναμονής. Η μέθοδος notify() δεν θα δώσει ακαριαία τον έλεγχο στο νήμα του καταναλωτή, αλλά θα συνεχίσει με την ολοκλήρωσή του και την εκτύπωση του μηνύματος «Done notifying the consumer.». Στην συνέχεια, θα ακολουθήσει η εκτύπωση του μηνύματος «Consumer just got notified!» και η ολοκλήρωση του νήματος του καταναλωτή.

```
public class WaitNotifyExample {

  public static void main(String[] args) {
            Integer key = new Integer(0);
            Thread consumer = new Thread(new Consumer(key));
            consumer.start();
            Thread producer = new Thread(new Producer(key));
            producer.start();
      }

}

class Consumer implements Runnable {

      public Integer keyInteger;

      public Consumer(Integer key) {
            this.keyInteger = key;
      }

      public void run() {
            synchronized (key) {
                  try {
                        System.out.println("Consumer is now waiting on the
key");
                        key.wait();
                        System.out.println("Consumer just got notified!");
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}

class Producer implements Runnable {

      public Integer keyInteger;

      public Producer(Integer key) {
            this.keyInteger = key;
      }

      public void run() {
            synchronized (key) {
                  try {
                        // sleep!
                        Thread.sleep(1000);
                        System.out.println("About to notify the
Consumer.");
                        key.notify();
                        System.out.println("Done notifying the
Consumer.");
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
```

Κώδικας 4.11 Επικοινωνία μεταξύ των νημάτων σε συγχρονισμένο περιβάλλον

Βιβλιογραφικές αναφορές

[1] Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed systems: principles and paradigms*. Prentice-Hall.

[2] Love, R., Are, S. H. W., Linus, A. C., Kernels, L. V. C. U., & Begin, B. W. (2005). *Linux kernel development second edition*. Novell Press.

[3] Kerrisk, M. (2010). *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press.

[4] Oaks, S. & Wong, H. (2004). *Java Threads*, 3rd edition. USA: O'Reilly Media.

Κριτήρια αξιολόγησης

Ερώτηση 1

Ποιο είναι το όνομα της μεθόδου που χρησιμοποιείται για την έναρξη της εκτέλεσης ενός νήματος στην Java;

A. init();
B. start();
Γ. run();
Δ. resume();

Ερώτηση 2

Ποιο από τα παρακάτω θα σταματήσει άμεσα την εκτέλεση ενός νήματος;

A. wait()
B. notify()
Γ. notifyall()
Δ. Κανένα από τα παραπάνω

Ερώτηση 3

Τι από τα παρακάτω δεν ισχύει για τα νήματα σε σχέση με τις διεργασίες;

Α. Καταναλώνουν λιγότερους πόρους του συστήματος.
B. Ανήκουν σε περισσότερες από μια διεργασίες.
Γ. Η επικοινωνία μεταξύ τους είναι σημαντικά πιο γρήγορη.
Δ. Όλα τα παραπάνω

Ερώτηση 4

Ποια από τις παρακάτω μεθόδους περιμένει να τερματίσει το νήμα την εκτέλεση του;

A. sleep()
B. isAlive()
Γ. join()
Δ. stop()

Ερώτηση 5

Τι είναι πολυνηματικός προγραμματισμός**;**

Α. Μια διεργασία στην οποία δύο διεργασίες τρέχουν ταυτόχρονα
B. Μια διεργασία στην οποία δύο ή περισσότερα κομμάτια αυτής εκτελούνται ταυτόχρονα
Γ. Μια διεργασία στην οποία αρκετές διαφορετικές διεργασίες έχουν πρόσβαση στα ίδια δεδομένα
Δ. Μια διεργασία στην οποία μια κάποια διεργασία μπορεί να έχει πρόσβαση σε πληροφορίες από αρκετούς πόρους

Ερώτηση 6

Ποιο από τα παρακάτω δεν θα προκαλέσει αμέσως τη διακοπή του νήματος;

A. notify()
B. wait()

Γ. sleep()
Δ. isAlive()

Ερώτηση 7
Τι είναι το race condition στα νήματα ;
Α. Όταν δύο ή περισσότερα νήματα έχουν πρόσβαση σε μη κοινά διαμοιραζόμενα δεδομένα
Β. Όταν δύο ή περισσότερα νήματα έχουν πρόσβαση σε κοινά διαμοιραζόμενα δεδομένα σε διαφορετικές χρονικές στιγμές
Γ. Όταν δύο ή περισσότερα νήματα έχουν πρόσβαση σε κοινά διαμοιραζόμενα δεδομένα την ίδια ακριβώς στιγμή
Δ. Κανένα από τα παραπάνω

Ερώτηση 8
Με ποια μέθοδο μπορώ να αναστείλω την εκτέλεση ενός νήματος για κάποιο καθορισμένο χρόνο;
Α. sleep()
Β. exit()
Γ. stop()
Δ. wait()

Ερώτηση 9
Ποιος από τους παρακάτω δεν αποτελεί τρόπο δημιουργίας ενός νήματος;
Α. Να κάνουμε extend την κλάση Thread και να δημιουργήσουμε το νήμα
Β. Να κάνουμε implement το runnable interface και να δημιουργήσουμε το νήμα
Γ. Να κάνουμε implement το Serializable interface.
Δ. Κανένα από τα παραπάνω

Ερώτηση 10
Η προτεραιότητα το νήματος (thread priority) στην Java είναι :
Α. Integer
Β. Float
Γ. double
Δ. long

Ερώτηση 11
Ποιος είναι ο κύριος ρόλος του mutex στην Java:
Α. Η διαφύλαξη των κοινόχρηστων διαμοιραζόμουν δεδομένων μεταξύ των νημάτων της εφαρμογής .
Β. Η επικοινωνία μεταξύ των νημάτων της εφαρμογής
Γ. Η διαφύλαξη της σωστής εκτέλεσης κάθε νήματος σε καθορισμένο χρόνο.
Δ. Όλα τα παραπάνω

Ερώτηση 12
Ποια μέθοδος δεν ανήκει στην κλάση Thread;
Α. run()
Β. isAlive()
Γ. wait()
Δ. join()

Ερώτηση 13
Ποιο πρόβλημα επιλύουν τα monitors στην Java σε σχέση με τα απλά mutexes;
Α. Συγχρονισμού (Synchronization)
Β. Επίδοσης (Performance issues)
Γ. Αναμονής (Busy waiting)

Δ. Ποιότητας κώδικα (Code quality)

Ερώτηση 14

Τι επιτυγχάνει η εφαρμογή της λέξης synchronized σε μια καθορισμένη μέθοδο στην Java;
Α. Πραγματοποιεί αμοιβαίο αποκλεισμό, διασφαλίζοντας ότι ένα μόνο νήμα θα εκτελέσει τον αντίστοιχο κώδικα της μεθόδου.
Β. Δίνει την δυνατότητα σε κάποιο άλλο νήμα να εκτελέσει τις λειτουργίες του.
Γ. Επιτρέπει σε πάνω από δύο νήματα να εκτελέσουν κοινά διαμοιραζόμενα κομμάτια τους.
Δ. Κανένα από τα παραπάνω

Ερώτηση 15

Ποιος είναι ο σκοπός της λέξης volatile στην Java;
Α. Ο συγχρονισμός κοινόχρηστων δεδομένων μεταξύ των νημάτων
Β. Η ανταλλαγή μηνυμάτων μεταξύ των νημάτων
Γ. Η ταυτόχρονη εκτέλεση των νημάτων χωρίς την πρόσβαση σε συγχρονισμένα κοινά δεδομένα
Δ. Κανένα από τα παραπάνω.