Εργασία Λειτουργικά Συστήματα

Μια Απλοποιημένη Πολυνηματική Διεργασία WebServer για Windows

Στέφανος Νικολάου

Α.Μ.:4070

Έτος: 4ο

Δαλλίδης Στυλιανός

Α.Μ.:4095

Έτος: 4ο

Επιβλέπων Καθηγητής : Μηνάς Δασυγένης

Εκφώνηση Εργασίας: Να προγραμματίσετε σε περιβάλλον Windows μια απλοποιημένη διεργασία web server. Αυτή η διεργασία θα δημιουργεί έναν αριθμό από threads τα οποία είναι worker threads. Όταν έρθει μια αίτηση στην θύρα 80, τότε το master thread διαλέγει ένα thread για να προωθήσει το request το οποίο στέλνει ότι αρχεία του έχουν ζητηθεί. Π.χ. Αν γίνει το request "GET INDEX.HTML" τότε θα στέλνει το αρχείο INDEX.HTML . Τα worker threads θα πρέπει να έχουν μια στοιχειώδη κατανόηση του πρωτοκόλλου HTTP (να αγνοούνται επιλογές ή εντολές εκτός από GET) ώστε να μπορούμε να συνδεθούμε με κάποιο Browser και να δούμε μια σελίδα.

Εισαγωγή

Η εργασία αυτή υλοποιήθηκε σε γλώσσα JAVA (στο Eclipse SDK http://www.eclipse.org) λόγω της απλότητας των βιβλιοθηκών της για Threads και Sockets.
Η εφαρμογή αυτή παρέχει μόνο την δυνατότητα για λήψη σελίδων από το κατάλληλο φάκελο(root) του webserver, ενώ σε περίπτωση που δεν υπάρχει index.html επιστρέφεται η λίστα από τα αρχεία που περιέχονται στον root φάκελο.
Τα standards του πρωτοκόλλου HTML πάρθηκαν από τη σελίδα προτύπων http://www.w3.org/TR/html4/.

Περιγραφή των Κλάσεων

Αρχικά, υλοποιήθηκε ένα interface για της κλάσεις μας, με όνομα HttpStandards όπου εκεί περιέχονται όλες οι σταθερές μεταβλητές για τα standard μηνύματα επικοινωνίας με το χρήστη όπως π.χ. HTTP_OK=200,HTTP_ACCEPTED=202 κλπ. Εν συνεχεία, υλοποιήθηκε η κλάση WebServer που αποτελεί και το κυρίως πρόγραμμα αυτής της εργασίας. Αυτή η κλάση περιέχει της ακόλουθες διαδικασίες:

static void loadProperties() →
Η διαδικασία αυτή φορτώνει στη μνήμη διάφορες παραμέτρους για τη λειτουργία του webserver από το αρχείο .\params\parameters.txt. Ένα παράδειγμα αυτού του αρχείου περιέχει τις παρακάτω τρεις γραμμές:

root=./wwwroot
workers=6
timeout=5000

Η πρώτη παράμετρος είναι ο φάκελος root που θα χρησιμοποιεί ο webserver. Η παράμετρος workers είναι ο μέγιστος αριθμός των νημάτων που θα χρησιμοποιεί ο webserver για την εξυπηρέτηση των αιτήσεων. Και τέλος, η παράμετρος timeout είναι τα δευτερόλεπτα που μπορεί να είναι απασχολημένο κάθε νήμα από μια αίτηση. Τα defaults αυτών των παραμέτρων:

root = (τρέχουσα διεύθυνση χρήστη)

timeout = 5000
workers = 5

static void PrintProperties() →
Η διαδικασία αυτή εκτυπώνει στην κονσόλα της παραμέτρους που αναφέραμε παραπάνω(root, workers, timeout).
public static void main(String[] args) →
Είναι η αρχική διαδικασίας εκτέλεσης του προγράμματος αυτού.
Όταν καλούμε τον webserver, μπορούμε να έχουμε ως όρισμα εισόδου την πόρτα στην οποία θέλει ο χρήστης να τρέχει ο webserver. Επίσης, εδώ εκκινούνται τα νήματα που θα εξυπηρετούν τα αιτήματα προς τον webserver. Συγκεκριμένα, σε αυτή τη διαδικασία παραμένει ο webserver μέχρι να τερματιστεί αναθέτοντας σε κάθε αίτηση ένα worker από αυτούς που υπάρχουν. Όταν, ένας worker χρησιμοποιηθεί καταστρέφεται. Έτσι, αν γίνουν workers σε πλήθος αιτήσεις την πρώτη φορά (π.χ. για workers=6 όπως φαίνεται στο παραπάνω αρχείο παραμέτρων), τότε καταστρέφονται όλοι οι προϋπάρχοντες workers, και σε κάθε νέα αίτηση δημιουργείται ένας νέος.
Η δεύτερη κλάση ονομάζεται worker. Αυτή η κλάση περιέχει της ακόλουθες διαδικασίες:
Worker() → Αυτή η διαδικασία είναι ο constructor της κλάσης worker.
synchronized void setSocket(Socket s) →
Αυτή η διαδικασία θέτει σε μια μεταβλητή(με εμβέλεια σε όλη τη κλάση) το socket το οποίο εξυπηρετεί ο συγκεκριμένος worker. Παρατηρούμε ότι αυτή η διαδικασία έχει οριστεί ως **synchronized** πράμα που σημαίνει ότι άλλο instances σε άλλο thread δε μπορεί να καλέσει αυτή τη διαδικασία μέχρι να τελειώσει στο ήδη τρέχον instance.
public synchronized void run()→
Αυτή η διαδικασία είναι απαραίτητη σε όσες κλάσεις θέλουν να λειτουργούν σε νήματα και στην ουσία από εδώ ξεκινάει να εκτελεί ο κάθε worker. Εδώ, περιμένει το νήμα του worker έως ότου του ανατεθεί ένα socket. Μόλις, του ανατεθεί προχωράει στην εξυπηρέτηση των αιτήσεων που καταφθάνουν μέσω του socket.
void handleClient() →
Σε αυτή τη διαδικασία περιέχονται οι διαδικασίες που πρέπει να ακολουθήσει ο worker ώστε να εξυπηρετήσει τις αιτήσεις των browsers. Πιο αναλυτικά, αρχικά διαβάζουμε από το socket τα εισερχόμενα δεδομένα ψάχνοντας για τη λέξη get. Εν συνεχεία παίρνουμε το όρισμα που ακολουθεί τη λέξη get που λογικά είναι το αρχείο που θέλει ο χρήστης. Ελέγχουμε αν υπάρχει αυτό το αρχείο εντός του φακέλου root(που έχει εισαχθεί μέσω του αρχείου παραμέτρων ή από τα defaults της συνάρτησης WebServer.loadProperties()). Εάν πράγματι υπάρχει τότε ανοίγεται και αποστέλλεται μέσω του socket. Αν είναι directory (αντί για αρχείο) τότε ψάχνουμε να βρούμε το αρχείο index.html μέσα σε αυτό το directory και το αποστέλλουμε με τον ίδιο τρόπο, αν το αρχείο δε βρεθεί αποστέλλεται η λίστα των αρχείων που βρίσκονται εντός του. Εδώ βρίσκονται οι έλεγχοι για το αν υπάρχουν τα αρχεία και εδώ επιλέγονται πότε θα αποσταλούν τα κατάλληλα μηνύματα λάθους μέσω του socket.
boolean printHeaders(File targ, PrintStream ps) →
Αυτή είναι μια βοηθητική διαδικασία η οποία διαβάζει τα αρχεία και τους καταλόγους που του θέτονται ως ορίσματα και τα επιστρέφει μέσω του PrintStream.
void send404(File targ, PrintStream ps) →
Και αυτή είναι μια βοηθητική διαδικασία που καλείται όταν δεν βρίσκεται ένα αρχείο και επιστρέφει μέσω του PrintStream μήνυμα λάθους με κωδικό 404.
void sendFile(File targ, PrintStream ps) →
Βοηθητική διαδικασία που αποστέλει το αρχείο targ στο PrintStream ps.
static void setSuffix(String k, String v) →

Βοηθητική διαδικασία που δημιουργεί την αντιστοιχία μεταξύ των String k και String v μέσα σε ένα HashTable. Αυτό το HashTable χρησιμοποιείται για να εμφανίζονται οι τύποι των αρχείων σύμφωνα με τη προέκταση τους κατά το listing ενός directory(π.χ. για την προέκταση ".html" θα επιστραφεί το string "text/html").

Περιγραφή της λειτουργίας των sockets στη JAVA

Το γενικότερο χαρακτηριστικό της JAVA που αποτελεί και ιδιαίτερο πλεονέκτημα της πολλές φορές έναντι πολλών άλλων γλωσσών είναι το επίπεδο αφαίρεσης που χρησιμοποιεί, με το οποίο αποκρύπτει από το χρήστη πολύπλοκη, προγραμματιστικά χαμηλότερη δομή. Το ίδιο συμβαίνει και με τα sockets.

Τα URLs παρέχουν ένα σχετικά υψηλού επιπέδου μηχανισμό για προσπέλαση δεδομένων στο διαδίκτυο. Πολλές φορές χρειαζόμαστε χαμηλότερου επιπέδου επικοινωνία, για παράδειγμα, όταν χρειάζεται να γράψουμε ένα πρόγραμμα client-server.

Σε εφαρμογές της μορφής client-server(πελάτης-εξυπηρετητής), ο server παρέχει κάποιες υπηρεσίες όπως π.χ. επεξεργασία database queries ή αποστολή αρχείων κ.α. Το πρόγραμμα πελάτης χρησιμοποιεί την υπηρεσία που παρέχει ο server, είτε εμφανίζοντας τα αποτελέσματα που του έστειλε ο server είτε σώζοντας το απεσταλμένο αρχείο στον τοπικό σκληρό δίσκο. Η επικοινωνία μεταξύ του προγράμματος πελάτη και προγράμματος εξυπηρετητή πρέπει να είναι, όπως αντιλαμβανόμαστε, ιδιαιτέρως αξιόπιστη.

Το πρωτόκολλο TCP εγγυάται αυτή την απαιτούμενη αξιοπιστία σε μια point-to-point επικοινωνία όπως δηλαδή απαιτείται από διαδικτυακές εφαρμογές τύπου client-server. Για να επικοινωνήσουν τα δύο σημεία μέσω TCP, δημιουργούν μια σύνδεση το ένα με το άλλο πρόγραμμα. Έτσι, κάθε πρόγραμμα αποκτά ένα socket μέχρι να έλθει το πέρας της επικοινωνίας με το άλλο πρόγραμμα. Κατά τη διάρκεια αυτής της επικοινωνίας τα προγράμματα γράφουν και διαβάζουν από το socket(περίπου σαν να είναι σειριακό αρχείο).

Συνήθως, λοιπόν, ο server εκτελείται σε έναν συγκεκριμένο υπολογιστή και έχει δεσμευμένο ένα socket σε μια συγκεκριμένη θύρα και αναμένει κάποια αίτηση για σύνδεση να καταφθάσει στη θύρα αυτή(listening). Το πρόγραμμα client γνωρίζει το συγκεκριμένο υπολογιστή(hostname,ip) και τη θύρα που τρέχει ο server. Έτσι, ο client κάνει μια αίτηση για σύνδεση. Σε αυτό το σημείο, ο υπολογιστής που τρέχει το πρόγραμμα client πρέπει να αναγνωριστεί από τον server, και, εν συνεχεία, να ανοιχτεί θύρα στον client(η θύρα αυτή αναθέτεται αυτόματα από το σύστημα).

Αν όλα πάνε καλά, ο server αποδέχεται τη σύνδεση. Κατά την αποδοχή της σύνδεσης, ο server δημιουργεί ένα socket πάνω στην ίδια τοπική θύρα που κάνει listening και στο ip και port του client. Τώρα, πλέον τα δύο προγράμματα(client και server) μπορούν να επικοινωνήσουν γράφοντας και διαβάζοντας πάνω στο ανοικτό socket που τα συνδέει.

Σε αυτό το σημείο θα επιχειρήσουμε να δώσουμε ένα ορισμό για το socket. Έτσι, λοιπόν, ένα socket είναι ένας σύνδεσμος επικοινωνίας μεταξύ δύο προγραμμάτων που τρέχουν σε δίκτυο. Ένα socket αποτελείται από τις παρακάτω πέντε παραμέτρους: πρωτόκολλο(TCP,UDP),τοπική ip διεύθυνση, τοπική θύρα, απομακρυσμένη ip διεύθυνση, απομακρυσμένη θύρα. Δηλαδή κάθε socket σε TCP προσδιορίζεται από την ακόλουθη τετράδα (τοπική ip διεύθυνση, τοπική θύρα, απομακρυσμένη ip διεύθυνση)(ή, αλλιώς, αυτή η τετράδα είναι μοναδική για κάθε socket TCP). Αντιλαμβανόμαστε, λοιπόν, ότι μπορούν να υπάρχουν πολλές συνδέσεις μεταξύ του client και του server.

Πιο συγκεκριμένα, στη JAVA, το πακέτο(package) java.net παρέχει μία κλάση με όνομα Socket, που υλοποιεί την σύνδεση μεταξύ του προγράμματος JAVA και ενός άλλου προγράμματος στο δίκτυο, μέσω socket. Η κλάση Socket «κάθεται» πάνω στην υλοποίηση των socket για την συγκεκριμένη πλατφόρμα που χρησιμοποιούμε και ως εκ τούτου κρύβει τις λεπτομέρειες της. Πιο απλά, χρησιμοποιώντας την κλάση java.net.Socket αντί για κώδικα εξαρτώμενο από τη πλατφόρμα, δημιουργούμε προγράμματα που επικοινωνούν μεταξύ τους ανεξαρτήτως πλατφόρμας.

Επιπλέον, το πακέτο(package) java.net περιέχει και μία ακόμη πολύ χρήσιμη κλάση, την ServerSocket, που μπορεί να χρησιμοποιηθεί από προγράμματα server που περιμένουν και αποδέχονται αιτήσεις σε διάφορες θύρες τους.

Ας μελετήσουμε ένα σύντομο παράδειγμα, στο οποίο παρουσιάζεται πολύ εύγλωττα πως μπορούμε να χρησιμοποιήσουμε την κλάση java.net.Socket, για να συνδεθούμε σε ένα server μέσω socket και πως χρησιμοποιούμε το αυτό το socket για αποστολή και λήψη δεδομένων.

Το παρακάτω πρόγραμμα είναι ένας client για έναν Echo Server(εξυπηρετητής ηχούς). Αυτό που κάνει ο Echo Server είναι απλά να στέλνει πίσω στο client ότι δεδομένα του αποστέλλει. Ο Echo Server είναι μια πολύ διαδεδομένη εφαρμογή και συνήθως βρίσκεται στη θύρα 7 του υπολογιστή μας.

Το παρακάτω πρόγραμμα συνδέεται στον Echo Server στην θύρα 7 και δημιουργεί ένα socket. Επίσης, δέχεται δεδομένα από το χρήστη μέσω του πληκτρολογίου και τα προωθεί στον Echo Server γράφοντας τα στο socket. Ο server αποστέλλει ξανά πίσω τα δεδομένα μέσω του socket στον client και ο client τα διαβάζει από το socket.

```
import java.io.*;
import java.net.*;

public class EchoClient {
   public static void main(String[] args)
            throws IOException {

                                    Socket echoSocket = null;
                                    PrintWriter out = null;
                                    BufferedReader in = null;

                                    try {
                                    echoSocket = new Socket("taranis", 7);
                                    out = new PrintWriter(echoSocket.getOutputStream(), true);
```

```
                in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
                                } catch (UnknownHostException e) {
                        System.err.println("Don't know about host: taranis.");
                        System.exit(1);
                        } catch (IOException e) {
             System.err.println("Couldn't get I/O for "  + "the connection to: taranis.");
             System.exit(1);
    }

            BufferedReader stdIn = new BufferedReader(
                        new InputStreamReader(System.in));
            String userInput;

            while ((userInput = stdIn.readLine()) != null) {
               out.println(userInput);
               System.out.println("echo: " + in.readLine());
            }

            out.close();
            in.close();
            stdIn.close();
            echoSocket.close();
   }
}
```

Ας προχωρήσουμε βήμα βήμα στον παραπάνω κώδικα εξηγώντας τα ουσιαστικά σημεία του. Οι πιο σημαντικές γραμμές κώδικα βρίσκονται εντός των try blocks. Οι παρακάτω γραμμές κώδικα δημιουργούν την socket επικοινωνία του προγράμματος μας με τον Echo Server και ανοίγουν ένα αντικείμενο PrintWriter(για την εγγραφή δεδομένων πάνω στο socket) και BufferedReader(για την ανάγνωση δεδομένων από το socket):

```
echoSocket = new Socket("taranis", 7);
out = new PrintWriter(echoSocket.getOutputStream(), true);
 in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
```

Η πρώτη, από τις παραπάνω τρεις γραμμές, δημιουργεί ένα αντικείμενο Socket. Ο constructor αυτού του αντικειμένου χρησιμοποιεί το όνομα του υπολογιστή(σε αυτό το παράδειγμα taranis) και η θύρα(7) που πρέπει να συνδεθεί το socket.

Η δεύτερη γραμμή ανοίγει πάνω στο socket ένα αντικείμενο PrintWriter για εγγραφή δεδομένων στο socket, ενώ αντίστοιχα η τρίτη γραμμή δημιουργεί το αντικείμενο BufferedReader για ανάγνωση δεδομένων από το socket. Να υπενθυμίσουμε ότι τα αντικείμενα PrintWriter και BufferedReader είναι κλάσεις που βρίσκονται εντός του package java.io (Για περισσότερες λεπτομέρειες σχετικά με τις συγκεκριμένες κλάσεις i/o της JAVA μεταβείτε στην ιστοσελίδα http://java.sun.com/docs/books/tutorial/essential/io/index.html).

Το επόμενο πολύ ενδιαφέρον σημείο του παραπάνω κώδικα είναι ο βρόγχος while. Στο βρόγχο αυτό διαβάζουμε μία μία γραμμή από το πληκτρολόγιο και, εν συνεχεία, το αποστέλλουμε στον server γράφοντας την κάθε γραμμή στον αντικείμενο PrintWriter:

```
String userInput
while ((userInput = stdIn.readLine()) != null) {
```

```
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

Η τελευταία γραμμή εντός του παραπάνω βρόγχου διαβάζει μία γραμμή από το αντικείμενο BufferedReader(που είναι συνδεδεμένο στο socket επικοινωνίας) και την εκτυπώνει στην κονσόλα μας. Η ανάγνωση των δεδομένων από τον BufferedReader γίνεται με την μέθοδο readline(). Αυτή η μέθοδος αναμένει μέχρι ο server να αποστείλει πίσω τα δεδομένα στον client.

Έτσι, αυτός ο βρόγχος while συνεχίζεται μέχρι ο χρήστης να εισάγει από το πληκτρολόγιο τον χαρακτήρα τερματισμού εισόδου(end-of-input character). Συνοπτικότερα, δηλαδή, σε αυτόν το βρόγχο διαβάζονται τα δεδομένα από το πληκτρολόγιο, αποστέλλονται στον Echo Server, λαμβάνονται πίσω από τον server και εκτυπώνονται στην οθόνη έως ότου ο χρήστης πατήσει τον χαρακτήρα τερματισμού εισόδου. Μόλις, λοιπόν το πρόγραμμα βγει από το βρόγχο εκτελεί τις παρακάτω τέσσερις γραμμές:

```
out.close();
in.close();
stdIn.close();
echoSocket.close();
```

Αυτές οι εντολές κλείνουν τα αντικείμενα ανάγνωσης και εγγραφής στο socket καθώς και τη σύνδεση με τον server. Η σειρά που μπαίνουν εδώ οι εντολές είναι σημαντική διότι πρέπει πρώτα να κλείνονται τα αντικείμενα που είναι συνδεδεμένα με το socket και μετά το ίδιο το socket.

Το συγκεκριμένο παράδειγμα είναι αρκετά απλό και παρουσιάζει με πολύ ξεκάθαρο τρόπο τα παρακάτω πέντε βασικά σημεία που χρειάζονται για την επικοινωνία προγραμμάτων με JAVA Sockets:

1. Δημιουργία ενός socket
2. Δημιουργία αντικειμένων εγγραφής και ανάγνωσης από το socket
3. Ανάγνωση από και εγγραφή στα αντικείμενα εγγραφής και ανάγνωσης σύμφωνα με το πρωτόκολλο του server
4. Κλείσιμο των αντικειμένων εγγραφής και ανάγνωσης από το socket
5. Κλείσιμο του socket

Διαδικασίες και Νήματα(Processes and Threads)

Στον ταυτόχρονο προγραμματισμό(concurrent programming), υπάρχουν δύο βασικές μονάδες της εκτέλεσης: οι *διαδικασίες* και τα *νήματα*. Στη γλώσσα προγραμματισμού της Java, ο ταυτόχρονος προγραμματισμός αναφέρεται συνήθως στα νήματα. Εντούτοις, οι διαδικασίες είναι επίσης σημαντικές.

Ένα σύστημα ηλεκτρονικών υπολογιστών έχει κανονικά πολλές ενεργές διαδικασίες αλλά και νήματα. Αυτό ισχύει ακόμη και στα συστήματα που έχουν μόνο έναν ενιαίο πυρήνα εκτέλεσης, και έτσι έχουν μόνο ένα νήμα που εκτελεί πραγματικά σε οποιαδήποτε δεδομένη χρονική στιγμή. Ο χρόνος επεξεργασίας για έναν ενιαίο πυρήνα μοιράζεται μεταξύ των διαδικασιών και των νημάτων μέσω ενός χαρακτηριστικού γνωρίσματος OS, ο οποίος αποκαλείται χρονικός τεμαχισμός.

Γίνεται όλο και περισσότερο σύνηθες για τα συστήματα ηλεκτρονικών υπολογιστών να έχουν τους πολλαπλούς επεξεργαστές ή τους επεξεργαστές με τους πολλαπλούς πυρήνες εκτέλεσης. Αυτό ενισχύει πολύ την ικανότητα ενός συστήματος για την ταυτόχρονη εκτέλεση των διαδικασιών και των νημάτων.

Διαδικασίες

Μια διαδικασία έχει ένα ανεξάρτητο περιβάλλον εκτέλεσης. Μια διαδικασία έχει γενικά ένα πλήρες, ιδιωτικό σύνολο βασικών πόρων χρόνου εκτέλεσης ειδικότερα, κάθε διαδικασία έχει το διάστημα μνήμης της.

Οι διαδικασίες θεωρούνται συχνά συνώνυμες με τα προγράμματα ή τις εφαρμογές. Εντούτοις, το τι θεωρεί ο χρήστης ως ενιαία εφαρμογή μπορεί να είναι στην πραγματικότητα ένα σύνολο διαδικασιών συνεργασίας. Για να διευκολύνουν την επικοινωνία μεταξύ των διαδικασιών, τα περισσότερα λειτουργικά συστήματα υποστηρίζουν *τις αλληλοδιαδικασίες επικοινωνίας*(IPC), όπως τα pipes και τα sockets. Η IPC χρησιμοποιείται όχι μόνο για την επικοινωνία μεταξύ των διαδικασιών του ίδιου συστήματος, αλλά και για την επικοινωνία μεταξύ των διαδικασιών διαφορετικών συστημάτων.
Οι περισσότερες εφαρμογές της εικονικής μηχανής της Java «τρέχουν» ως ενιαία διαδικασία. Μια εφαρμογή της Java μπορεί να δημιουργήσει πρόσθετες διαδικασίες χρησιμοποιώντας έναν `ProcessBuilder`.

Νήματα

Τα νήματα καλούνται μερικές φορές *ελαφριές διαδικασίες*. Και οι διαδικασίες και τα νήματα παρέχουν ένα περιβάλλον εκτέλεσης, αλλά η δημιουργία ενός νέου νήματος απαιτεί λιγότερους πόρους από την δημιουργία μιας νέας διαδικασίας.

Τα νήματα υπάρχουν μέσα σε κάθε διαδικασία — κάθε διαδικασία έχει τουλάχιστον ένα. Τα νήματα μοιράζονται τους πόρους της διαδικασίας, συμπεριλαμβανομένης της μνήμης και των ανοικτών αρχείων. Αυτό κάνει αποδοτική την επεξεργασία δεδομένων, αλλά ενδεχομένως προβληματική, την επικοινωνία μεταξύ τους.
Η πολυνηματική εκτέλεση είναι ένα ουσιαστικό χαρακτηριστικό γνώρισμα της πλατφόρμας της Java. Κάθε εφαρμογή έχει τουλάχιστον ένα νήμα ή περισσότερα, εάν μετρήσουμε τα νήματα "συστημάτων" που κάνουν πράγματα, όπως η διαχείριση της μνήμης και ο χειρισμός των σημάτων. Αλλά από την άποψη του προγραμματιστή εφαρμογής, όλα εκκινούνται μόνο από ένα νήμα, αποκαλούμενο *κύριο νήμα*. Αυτό το νήμα έχει τη δυνατότητα να δημιουργήσει τα πρόσθετα νήματα.

Αντικείμενα Νημάτων και κλάσεις Νημάτων της Java

Υπάρχουν δύο βασικές στρατηγικές για τα αντικείμενα νημάτων προκριμένου να δημιουργήσουν μια πολυνηματική εφαρμογή.

- Για τον άμεσο έλεγχο της δημιουργίας και της διαχείρισης των νημάτων, απλά ελέγχουμε το νήμα κάθε φορά που πρέπει να αρχίσει η εφαρμογή μία ασύγχρονη εργασία.
- Για την αφαίρεση της διαχείρισης των νημάτων από το υπόλοιπο της εφαρμογής σας, περάστε τις εργασίες της εφαρμογής σε έναν *εκτελεστή*.

Μια εφαρμογή που δημιουργεί ένα νήμα πρέπει να παρέχει τον κώδικα που θα τρέξει σε αυτό το νήμα. Υπάρχουν δύο τρόποι για να γίνει αυτό:

Με ένα runnable αντικείμενο(εκτελέσιμο αντικείμενο). Το runnable αντικείμενο καθορίζει μια ενιαία μέθοδο, η οποία προορίζεται για να περιέχει τον κώδικα που εκτελείται από το νήμα. Το runnable αντικείμενο συντάσσεται όπως στο HelloRunnable παράδειγμα:

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

Με Υποκλάσεις νημάτων. Μια εφαρμογή μπορεί να κάνει μια υποκλάση νημάτων και να παρέχει η ίδια την εκτέλεση μια εργασίας, όπως το επόμενο παράδειγμα HelloThread :

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Σημείωση: Και τα δύο παραδείγματα επικαλούνται το thread.start προκειμένου να εκκινήσει το νέο νήμα.

Ποιον από αυτούς τους δύο τρόπους θα πρέπει να χρησιμοποιείται; Ο πρώτος τρόπος, που χρησιμοποιεί ένα runnable αντικείμενο , είναι ο πιο γενικός, επειδή το runnable αντικείμενο μπορεί να δημιουργήσει μια υποκλάση μιας κλάσης εκτός από την κλάση νήματος. Ο δεύτερος τρόπος είναι ευκολότερος στην χρήση στις απλές εφαρμογές, αλλά περιορίζεται από το γεγονός ότι η κλάση θα πρέπει να είναι «απόγονος» του νήματος.

Η κλάση νημάτων καθορίζει διάφορες μεθόδους χρήσιμες για τη διαχείριση νημάτων.

Η διαδικασία Thread.sleep() αναγκάζει το τρέχον νήμα να αναστείλει την εκτέλεση για μια καθορισμένη περίοδο. Αυτός είναι ένας αποδοτικός τρόπος έτσι ώστε ο

χρόνος επεξεργασίας να είναι διαθέσιμος και στα άλλα νήματα μιας εφαρμογής ή άλλων εφαρμογών που ανήκουν σε ένα σύστημα ηλεκτρονικών υπολογιστών. Η μέθοδος Thread.sleep() μπορεί επίσης να χρησιμοποιηθεί, όπως φαίνεται στο παράδειγμα που ακολουθεί, και για την αναμονή ενός άλλου νήματος με καθήκοντα που «καταλαβαίνουν» πότε να έχουν χρονικές απαιτήσεις, όπως με το παράδειγμα SimpleThreads παρακάτω.

Δύο υπερφορτωμένες εκδόσεις του Thread.sleep() παρέχονται: μία που καθορίζει το χρόνο ύπνου στα χιλιοστά του δευτερολέπτου και μία που καθορίζει το χρόνο ύπνου στο νανοδευτερόλεπτα. Εντούτοις, αυτοί οι χρόνοι αδράνειας του νήματος δεν είναι εγγυημένα ακριβείς. Επίσης, η περίοδος αδράνειας μπορεί να διακοπεί εξαιτίας διακοπών(interrupts), όπως θα δούμε παρακάτω. Εν πάση περιπτώσει, δεν μπορούμε να υποθέσουμε ότι η κλήση του Thread.sleep() θα αναστείλει την λειτουργία του νήματος για καθορισμένο χρονικό διάστημα.

Το παρακάτω παράδειγμα SleepMessages χρησιμοποιεί την Thread.sleep() προκειμένου να τυπώνει μηνύματα ανά διαστήματα τεσσάρων δευτερολέπτων:

```
public class SleepMessages {
   public static void main(String args[]) throws InterruptedException {
      String importantInfo[] = {
         "Mares eat oats",
         "Does eat oats",
         "Little lambs eat ivy",
         "A kid will eat ivy too"
      };

      for (int i = 0; i < importantInfo.length; i++) {
         //Pause for 4 seconds
         Thread.sleep(4000);
         //Print a message
         System.out.println(importantInfo[i]);
      }
   }
}
```

Σημείωση: Η συνάρτηση *main* δηλώνεται ως *"throws InterruptedException"*. Αυτό είναι μια εξαίρεση(Exception) που το Thread.sleep() δημιουργεί, όταν ένα άλλο νήμα διακόπτει το τρέχον νήμα, ενώ το νήμα βρίσκεται σε αδράνεια. Εφόσον αυτή η εφαρμογή δεν έχει προσδιορίσει ένα άλλο νήμα, ώστε να προκαλέσει τέτοιου είδους διακοπή, δεν επηρεάζεται από *InterruptedException.*

Διακοπές(Interrupts)

Διακοπή είναι μια ένδειξη σε ένα νήμα ότι πρέπει να σταματήσει αυτό που κάνει εκείνη την στιγμή και κάνει κάτι άλλο. Εξαρτάται από τον προγραμματιστή να αποφασιστεί ακριβώς πώς ένα νήμα αποκρίνεται στην διακοπή, αλλά είναι πολύ σύνηθες το νήμα να ολοκληρώνεται. Προκειμένου ο μηχανισμός διακοπής να λειτουργεί σωστά θα πρέπει το κάθε νήμα να υποστηρίζει την διακοπή του.

Υποστήριξη Διακοπής στο Νήμα

Πώς ένα νήμα υποστηρίζει τη διακοπή του; Εάν το νήμα επικαλείται συχνά τις μεθόδους που στέλνουν InterruptedException, επιστρέφει απλά από τη μέθοδο «τρεξίματος». Παραδείγματος χάριν, υποθέστε ότι ο κεντρικός βρόχος μηνυμάτων στο παράδειγμα SleepMessages βρισκόταν στη μέθοδο τρεξίματος του runnable αντικείμενο ενός νήματος. Κατόπιν τροποποιείται έτσι ώστε να υποστηρίξει την διακοπή:

```
for (int i = 0; i < importantInfo.length; i++) {
   try {
      Thread.sleep(4000);
   } catch (InterruptedException e) {
            return;
   }
      System.out.println(importantInfo[i]);
}
```

Πολλές μέθοδοι που προκαλούν InterruptedException, όπως η Thread.sleep() , έχουν ως σκοπό να ακυρώσουν την τρέχουσα λειτουργία τους και να επιστρέψουν αμέσως σ' αυτήν όταν παραληφθεί η διακοπή.

Τι γίνεται εάν ένα νήμα αφήσει να περάσει ένα μεγάλο χρονικό διάστημα χωρίς να κάνει κλήση μιας μεθόδου που προκαλεί InterruptedException; Τότε, πρέπει να επικαλείται περιοδικά η εντολή *Thread.interrupted* , η οποία επιστρέφει TRUE εάν μια διακοπή έχει παραληφθεί. Για παράδειγμα

```
for (int i = 0; i < inputs.length; i++) {
   heavyCrunch(inputs[i]);
   if (Thread.interrupted()) {
      return;
   }
}
```

Η σημαία κατάστασης διακοπής

Ο μηχανισμός διακοπής εφαρμόζεται χρησιμοποιώντας μια εσωτερική σημαία γνωστή ως *κατάσταση διακοπής*. Η κλήση της εντολής Thread.interrupt θέτει αυτήν την σημαία. Όταν ένα νήμα ελέγχει την διακοπή με την επίκληση της στατικής μεθόδου Thread.interrupted, η κατάσταση διακοπής «καθαρίζεται». Τα μη-στατικά Thread.isInterrupted, που χρησιμοποιούνται από ένα νήμα για να ρωτήσουν την θέση διακοπής ενός άλλου, δεν αλλάζουν την σημαία της κατάστασης διακοπής.

Συμβατικά, οποιαδήποτε μέθοδος που προκύπτει από την αποστολή ενός InterruptedException «καθαρίζει» την κατάσταση διακοπής όταν το κάνει αυτό. Ωστόσο, είναι πάντα δυνατό που η κατάσταση διακοπής να τεθεί πάλι αμέσως, στην περίπτωση που ένα άλλο νήμα επικαλεστεί διακοπή.

Ενώσεις

Η μέθοδος ένωσης επιτρέπει ένα νήμα να περιμένει για την ολοκλήρωση ενός άλλου. Εάν t είναι ένα αντικείμενο νήματος(thread αντικείμενο) του οποίου το νήμα εκτελείται την παρούσα χρονική στιγμή, η εντολή t.join() σταματάει το τρέχον νήμα μέχρι το νήμα του t να τερματίσει. Υπερφορτώσεις της μεθόδου join επιτρέπουν να ορίσουμε χρόνο αναμονής του νήματος. Ωστόσο, όπως και με τη μέθοδο sleep, η μέθοδος join εξαρτάται από τους χρονισμούς του λειτουργικού συστήματος(OS timing) και έτσι δεν μπορούμε να αναμένουμε η εντολή join να λειτουργεί με πολύ μεγάλη χρονική ακρίβεια.

Το παράδειγμα SimpleThreads

Το ακόλουθο παράδειγμα συγκεντρώνει μερικές από τις έννοιες που αναφέραμε παραπάνω. Το SimpleThreads αποτελείται από δύο νήματα.Το πρώτο είναι το κύριο νήμα που υπάρχει σε κάθε εφαρμογή της Java.. Το κύριο νήμα δημιουργεί ένα νέο νήμα από το runnable αντικείμενο, MessageLoop, και το περιμένει να τελειώσει. Εάν το νήμα MessageLoop παίρνει πάρα πολύ χρόνο για να τελειώσει, το κύριο νήμα το διακόπτει.

Το νήμα MessageLoop εκτυπώνει μια σειρά μηνυμάτων. Εάν διακόπτεται προτού να τυπώσει όλα τα μηνύματά του, το νήμα MessageLoop τυπώνει ένα μήνυμα κάνει exit.

```
public class SimpleThreads {

    static void threadMessage(String message) {
    String threadName = Thread.currentThread().getName();
    System.out.format("%s: %s%n", threadName, message);
}

private static class MessageLoop implements Runnable {
    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        try {
            for (int i = 0; i < importantInfo.length; i++) {
                Thread.sleep(4000);
                 threadMessage(importantInfo[i]);
            }
        } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
        }
    }
}

public static void main(String args[]) throws InterruptedException {
  long patience = 1000 * 60 * 60;
  if (args.length > 0) {
      try {
          patience = Long.parseLong(args[0]) * 1000;
```

```
            } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
                System.exit(1);
            }

        }

        threadMessage("Starting MessageLoop thread");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new MessageLoop());
        t.start();

        threadMessage("Waiting for MessageLoop thread to finish");
        while (t.isAlive()) {
            threadMessage("Still waiting...");
            t.join(1000);
            if (((System.currentTimeMillis() - startTime) > patience) &&
                  t.isAlive()) {
                threadMessage("Tired of waiting!");
                t.interrupt();
                t.join();
            }

        }
        threadMessage("Finally!");
    }
}
```

Συγχρονισμός

Στο συγχρονισμό των νημάτων υπάρχουν διάφορα προβλήματα όπως: λάθη *παρεμβολής νήματος(thread interference)* και λάθη *συνέπειας μνήμης(memory consistency errors)*. Το εργαλείο που απαιτείται προκειμένου να αποτρέψει αυτά τα λάθη είναι ο *συγχρονισμός*.

Συγχρονισμένες μέθοδοι

Η γλώσσα προγραμματισμού της Java παρέχει δύο βασικούς τρόπους συγχρονισμού: *συγχρονισμένες μέθοδοι* και *συγχρονισμένες δηλώσεις*. Ο πιο σύνθετος των δύο, είναι οι συγχρονισμένες δηλώσεις, και περιγράφεται στην επόμενη παράγραφο. Σε αυτή την παράγραφο περιγράφονται οι *συγχρονισμένες μέθοδοι.*

Για να καταστήσουμε μια μέθοδο συγχρονισμένη, προσθέτουμε απλά τη λέξη-κλειδί synchronized στη δήλωσή της:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
```

```
    }
}
```

Εάν ο count είναι ένα instance του SynchronizedCounter, τότε το να καταστήσεις αυτές τις μεθόδους συγχρονισμένες έχει δύο αποτελέσματα:

- Κατ' αρχάς, δεν είναι δυνατό για δύο κλήσεις των συγχρονισμένων μεθόδων στο ίδιο αντικείμενο να συνυπάρχουν. Όταν ένα νήμα εκτελεί μια συγχρονισμένη μέθοδο για ένα αντικείμενο , όλα τα άλλα νήματα που καλούν συγχρονισμένες μεθόδους για το ίδιο block αντικείμενο (αναστέλλουν την εκτέλεση) μέχρι το πρώτο νήμα να ολοκληρώσει την εργασία του.
- Δεύτερον, όταν μια συγχρονισμένη μέθοδος τερματίζει, καθιερώνεται αυτόματα μία happens-before σχέση με οποιαδήποτε επόμενη κλήση μιας συγχρονισμένης μεθόδου για το ίδιο αντικείμενο. Αυτό εγγυάται ότι οι αλλαγές στην κατάσταση του αντικειμένου είναι ορατές σε όλα τα νήματα.

Σημειώστε ότι οι constructors δεν μπορούν να συγχρονιστούν — χρησιμοποιώντας τη λέξη-κλειδί synchronized με έναν constructor είναι λάθος σύνταξης. Ο συγχρονισμός των constructors δεν έχει νόημα, επειδή μόνο το νήμα που δημιουργεί ένα αντικείμενο πρέπει να έχει πρόσβαση σε αυτό ενώ αυτό κατασκευάζεται.

Οι συγχρονισμένες μέθοδοι επιτρέπουν μια απλή στρατηγική για τα λάθη παρεμβολής νήματος και συνέπειας μνήμης: εάν ένα αντικείμενο είναι ορατό σε περισσότερα από ένα νήματα, όλος διαβάζει ή γράφει στις μεταβλητές εκείνου του αντικείμενο που γίνεται μέσω των συγχρονισμένων μεθόδων. (Μια σημαντική εξαίρεση: οι τελικοί τομείς, που δεν μπορούν να τροποποιηθούν αφότου κατασκευάζεται το, μπορούν να διαβαστούν ακίνδυνα μέσω των μη-συγχρονισμένων μεθόδων, μόλις κατασκευαστεί το αντικείμενο) αυτή η στρατηγική είναι αποτελεσματική.

Αμετάβλητα Αντικείμενα(Immutable Objects)

Ένα αντικείμενο θεωρείται *αμετάβλητο* εάν η κατάσταση του δεν μπορεί να αλλάξει αφότου έχει κατασκευαστεί. Η χρησιμοποίηση αμετάβλητων αντικειμένων για την δημιουργία νημάτων και για την εκτέλεση από νήματα θεωρείτε μια από της καλύτερες στρατηγικές για τη δημιουργία απλού και αξιόπιστου κώδικα.

Τα αμετάβλητα αντικείμενα είναι ιδιαίτερα χρήσιμα στις πολυνηματικές εφαρμογές. Δεδομένου ότι δεν μπορούν να αλλάξουν την κατάσταση τους, δεν μπορούν να αλλοιωθούν από την παρεμβολή νημάτων ή να βρεθούν σε λανθασμένη κατάσταση.

Οι προγραμματιστές είναι συχνά απρόθυμοι να χρησιμοποιήσουν τα αμετάβλητα αντικείμενα, επειδή ανησυχούν για το επεξεργαστικό κόστος της δημιουργίας ενός νέου αντικείμενο(συγκριτικά με το επεξεργαστικό κόστος της αλλαγής ενός ήδη υπάρχοντος αντικειμένου). Αυτό το επεξεργαστικό κόστος(της δημιουργίας αντικειμένου) υπερεκτιμάται πολλές φορές, αφού τελικά αντισταθμίζεται από τα άλλα πλεονεκτήματα των αμετάβλητων αντικείμενο. Τέτοια πλεονεκτήματα είναι η μειωμένη επιβάρυνση λόγω της συλλογής απορριμμάτων(garbage collection) της Java και η αποφυγή κώδικα που τα μεταβλητά αντικείμενα(mutable objects) χρειάζονται οπωσδήποτε για να μη «πέσουν» σε λανθασμένη κατάσταση.

Παράδειγμα συγχρονισμένης κλάσης

Η κλάση, SynchronizedRGB, ορίζει αντικείμενα που αντιπροσωπεύουν τα χρώματα. Κάθε αντικείμενο αντιπροσωπεύει το χρώμα με τρεις ακέραιους αριθμούς, οι οποίοι αντιπροσωπεύουν τα τρία βασικά χρώματα και ένα αλφαριθμητικό(string) που είναι το όνομα του χρώματος.

```
public class SynchronizedRGB {
   private int red;
   private int green;
   private int blue;
   private String name;
   private void check(int red, int green, int blue) {
      if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
         throw new IllegalArgumentException();
      }
   }
   public SynchronizedRGB(int red, int green, int blue, String name) {
      check(red, green, blue);
      this.red = red;
      this.green = green;
      this.blue = blue;
      this.name = name;
   }
   public void set(int red, int green, int blue, String name) {
      check(red, green, blue);
      synchronized (this) {
         this.red = red;
         this.green = green;
         this.blue = blue;
         this.name = name;
      }
   }
   public synchronized int getRGB() {
      return ((red << 16) | (green << 8) | blue);
   }
   public synchronized String getName() {
      return name;
   }
   public synchronized void invert() {
      red = 255 - red;
      green = 255 - green;
      blue = 255 - blue;
      name = "Inverse of " + name;
   }
}
```

Το SynchronizedRGB πρέπει να χρησιμοποιηθεί προσεκτικά προκειμένου να αποφευχθεί ο κίνδυνος να βρεθεί σε μία ασυμβίβαστη κατάσταση. Υποθέστε, παραδείγματος χάριν, ένα νήμα που εκτελεί τον ακόλουθο κώδικα:

```
SynchronizedRGB color = new SynchronizedRGB(0, 0, 0, "Pitch Black");
```

...
int myColorInt = color.getRGB(); //Δήλωση 1
String myColorName = color.getName(); //Δήλωση 2
Εάν ένα άλλο νήμα καλέσει την color.set(), μετά από τη δήλωση 1 και πριν από τη δήλωση 2, η τιμή myColorInt δεν θα ταιριάξει με την τιμή myColorName. Για να αποφύγει αυτήν την έκβαση, οι δύο δηλώσεις πρέπει να δεσμευθούν μαζί:

```
synchronized (color) {
   int myColorInt = color.getRGB();
   String myColorName = color.getName();
}
```

Βιβλιογραφία

1. http://www.eclipse.org
2. http://www.w3.org/TR/html4/
3. http://java.sun.com/developer/technicalArticles/Networking/Webserver/
4. http://java.sun.com/developer/technicalArticles/Networking/Webserver/
5. http://java.sun.com/docs/books/tutorial/networking/sockets/index.html