Software Agent

Καμπυλαυκάς Ιωάννης
sdi9500781@di.uoa.gr

Σούλης Αθανάσιος
sdi0900155@di.uoa.gr

13 Νοεμβρίου 2015

1 Η συγχρονισμένη ουρά

1.1 Το interface `IBlockingQueue<E>`

```
public interface IBlockingQueue<E> {
    public void put(E item) throws InterruptedException;
    public E get() throws InterruptedException;
    public E peek();
    public int size();
    boolean hasMaxSize();
}
```

Το interface `IBlockingQueue<E>` αποτελεί μια γενικευμένη αφαίρεση για μια συγχρονισμένη ουρά αντικειμένων τύπου `E`, με προαιρετικό μέγιστο μέγεθος. Η μέθοδος `put()` πέφτει σε αναμονή στην περίπτωση που το τρέχον μέγεθος της ουράς είναι ίσο με το προαιρετικό μέγιστο. Ομοίως η κλήση `get()` πέφτει σε αναμονή στην περίπτωση που η ουρά είναι κενή. Η μέθοδος `peek()` επιστρέφει το πρώτο αντικείμενο της ουράς όπως και η `get()`, χωρίς όμως να το αφαιρέσει από την ουρά. Η μέθοδος `size()` επιστρέφει το τρέχον μέγεθος της ουράς. Οι κλάσεις που υλοποιούν το παραπάνω interface πρέπει επίσης να επιστρέφουν αν έχει τεθεί μέγιστο μέγεθος για την ουρά, με τη μέθοδο `hasMaxSize()`.

Στο παραδοτέο υπάρχουν δύο υλοποιήσεις του `IBlockingQueue<E>` με τελείως ισοδύναμη συμπεριφορά αλλά διαφορετική μέθοδο συγχρονισμού. Στο τελικό παραδοτέο στο οποίο υπάρχουν δύο ουρές (μία για τις εργασίες που εκτελούνται μία φορά και μία για τα αποτελέσματα όλων των εργασιών) χρησιμοποιείται από μία φορά η κάθε υλοποίηση.

1.2 Η κλάση `WaitNotifyQueue<E>`

Η κλάση `WaitNotifyQueue<E>` περιέχει μια εσωτερική μη συγχρονισμένη ουρά τύπου `Queue<E>`. Ο έλεγχος της πρόσβασης στη δομή αυτή γίνεται με τη χρήση κλήσεων `wait()` και `notify()` μέσα σε `synchronized` blocks των μεθόδων `put()` και `get()` της κλάσης.

1.3 Η κλάση `SemaphoreQueue<E>`

Η κλάση `SemaphoreQueue<E>` περιέχει κι αυτή μια μη συγχρονισμένη εσωτερική ουρά, η σωστή όμως πρόσβαση σε αυτή γίνεται χρησιμοποιώντας σημαφόρους. Ο σημαφόρος `occupied` αντιστοιχεί στις κατειλημμένες θέσεις της ουράς και χρησιμοποιείται πάντα ενώ ο σημαφόρος `available` αντιστοιχεί στις διαθέσιμες θέσεις της ουράς και χρησιμοποιείται μόνο αν έχει προσδιοριστεί μέγιστο μέγεθος για την ουρά. Μετά τον έλεγχο για κενή/γεμάτη ουρά με τον αντίστοιχο σημαφόρο, το νήμα θα πρέπει να

μπει σε synchronized block για να έχει πρόσβαση στην εσωτερική ουρά, όπως και στην περίπτωση της WaitNotifyQueue<E>.

2 Η υπηρεσία εκτέλεσης εργασιών

2.1 Το interface IExecutorService

```
public interface IExecutorService {
    boolean submit(Runnable r);
    void shutdown();
    void awaitTermination() throws InterruptedException;
}
```

Το interface αυτό αποτελεί μια βασική αφαίρεση για μια υπηρεσία εκτέλεσης Runnable αντικειμένων. Με τη μέθοδο submit() γίνεται υποβολή του Runnable για εκτέλεση και η κλήση επιστρέφει true μετά από επιτυχή υποβολή και false σε αντίθετη περίπτωση. Με τη μέθοδο shutdown() ειδοποιούνται οι εργασίες που έχουν ήδη ξεκινήσει να τερματίσουν και γίνεται αδύνατη η υποβολή περαιτέρω εργασιών. Κάθε κλήση της μεθόδου submit() μετά απο κλήση shutdown() επιστρέφει false. Με τη μέθοδο awaitTermination() γίνεται αναμονή για την ολοκλήρωση των εργασιών υπό τερματισμό. Η μέθοδος δεν επιστρέφει παρά μόνο όταν ολοκληρωθούν όλες οι εργασίες οι οποίες είχαν προλάβει να ξεκινήσουν όταν ζητήθηκε τερματισμός. Η μοναδική υλοποίηση του IExecutorService στο παραδοτέο είναι η κλάση ThreadPool.

2.2 Η κλάση ThreadPool

Η κλάση ThreadPool υλοποιεί το interface IExecutorService χρησιμοποιώντας ένα σύνολο νημάτων τύπου WorkerThread, το πλήθος των οποίων προσδιορίζεται στον constructor της κλάσης. Εσωτερικά περιέχει μια ουρά IBlockingQueue<Runnable> (χρησιμοποιήθηκε η SemaphoreQueue<Runnable>) από την οποία τα νήματα εξάγουν Runnables και τα εκτελούν. Αν η ουρά είναι κενή και κάποιο νήμα προσπαθήσει να εξάγει ένα Runnable από αυτή, τότε το νήμα περνάει σε αναμονή. Αυτό γίνεται στην κλήση της μεθόδου get() του IBlockingQueue<E> interface που υλοποιείται από την ουρά.

2.3 Η κλάση WorkerThread

Η κλάση WorkerThread είναι υποκλάση της Thread και χρησιμοποιείται από την κλάση ThreadPool. Όσο ένα WorkerThread είναι ενεργό, προσπαθεί να πάρει κάποιο Runnable από την ουρά που του έχει περαστεί ως παράμετρος κατά την κατασκευή του. Ακολούθως εκτελεί τη μέθοδο run() του Runnable. Ένα WorkerThread μπορεί να τερματίσει είτε σε χρονική στιγμή που περιμένει στη μέθοδο get() του IBlockingQueue<Runnable> ή κατά τη διάρκεια της εκτέλεσης του run() του Runnable που έχει αναλάβει. Στη δεύτερη περίπτωση, και αν το Runnable χειριστεί εσωτερικά κάποιο InterruptedException, πρέπει να φροντίσει να θέσει ξανά το interrupted status σε true πριν την ολοκλήρωση της run() μεθόδου του, έτσι ώστε να τερματίσει το WorkerThread που το έχει αναλάβει.

3 Ο πάροχος εργασιών

3.1 Το interface `IJobProvider`

```
public interface IJobProvider {
    public boolean hasMoreJobs();
    public Job[] getNextJobs();
}
```

Πρόκειται για ένα απλό interface για την λήψη εργασιών προς εκτέλεση από κάποια πηγή. Στο παραδοτέο υπάρχουν δύο υλοποιήσεις του παραπάνω interface. Η κλάση `Job` περιέχει όλες τις πληροφορίες μιας εργασίας καθώς και την έξοδο της, αν η εργασία έχει ολοκληρωθεί. Η κλάση `Job` υλοποιεί το interface `Runnable` και η μέθοδος `run()` αναλαμβάνει να εκτελέσει την nmap διεργασία και να παραλάβει την έξοδο.

3.2 Η κλάση `JobGenerator`

Η συγκεκριμένη κλάση έχει τη χρησιμότητα του να παράγει ένα σχετικά μεγάλο πλήθος εργασιών από ένα μικρό πραγματικό αρχείο και χρησιμοποιήθηκε κατά τον έλεγχο του προγράμματος. Δημιουργεί πολλές εργασίες με αυξανόμενα `id`, όμως οι παράμετροι της κάθε μίας έχουν απλά επιλεγεί τυχαία από τις ενδεχομένως λίγες διαθέσιμες εργασίες του αρχείου.

3.3 Η κλάση `JobReader`

Η κλάση `JobReader` λειτουργεί πιο συμβατικά, διαβάζοντας ένα αρχείο εισόδου με τη σειρά και επιστρέφοντας όλες τις εργασίες έως ότου φθάσει στο τέλος του. Προορίζεται για χρήση με μεγάλα αρχεία εισόδου τα οποία διαβάζει σταδιακά. Επειδή διαβάζει σταδιακά, υπάρχει μια μικρή ασυνέπεια στην υλοποίηση του interface υπό την έννοια ότι μπορεί να επιστρέψει `true` σε μια κλήση `hasMoreJobs()` και ακολούθως να επιστρέψει 0 εργασίες. Αυτό μπορεί να συμβεί το πολύ μια φορά, στην περίπτωση που έχουμε φτάσει ακριβώς στο τέλος του αρχείου με την προηγούμενη ανάγνωση. Κάθε επόμενη κλήση της `hasMoreJobs()` επιστρέφει `false`.

4 Εκτέλεση του προγράμματος

4.1 Μεταγλώττιση

Στον φάκελο k23b/sa όπου βρίσκεται το αρχείο `pom.xml` του Maven project εκτελούμε:

```
mvn clean package
```

Στον φάκελο `target` θα αντιγραφούν τα αρχεία `jobList.txt` και `sa.properties` που απαιτούνται για την εκτέλεση του προγράμματος. Επίσης δημιουργούνται δύο `.jar` αρχεία, ένα που περιλαμβάνει τις εξαρτήσεις του project και ένα χωρίς αυτές. Εκτελούμε το πρόγραμμα με:

```
java -jar sa-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

Η μοναδική εξωτερική εξάρτηση του προγράμματος είναι η βιβλιοθήκη logging log4j. Κατά την εκτέλεση του προγράμματος δημιουργείται φάκελος `log/` που περιέχει το αρχείο `sa.log`. Στο αρχείο `log4j.properties` που επίσης αντιγράφεται από το φάκελο του project στον φάκελο `target` κατά τη μεταγλώττιση, μπορεί να επιλεγεί το επιθυμητό επίπεδο logging (`DEBUG`, `INFO`, `ERROR`).

4.2 Ρυθμίσεις

Στο παραδοτέο περιλαμβάνεται το αρχείο `sa.properties` με τις εξής ρυθμίσεις:

`jobFileName=jobfile.txt`	(όνομα του αρχείου εργασιών, στον ίδιο κατάλογο)
`generateJobs=true`	(`true`: χρήση `JobGenerator`, `false`: χρήση `JobReader`)
`jobRequestInterval=60`	(χρονικό διάστημα μεταξύ δυο αιτημάτων για νέες εργασίες)
`maxNextJobs=15`	(μέγιστο πλήθος εργασιών που επιστρέφονται σε ένα αίτημα)
`threadPoolSize=5`	(πλήθος των worker threads στο thread pool)
`runNmapAsRoot=true`	(εκτέλεση διεργασιών με `"sudo nmap ..."` αντί για `"nmap ..."`)
`resultsQueueMax=10`	(μέγιστο μέγεθος ουράς αποτελεσμάτων)
`senderThreadInterval=15`	(χρονικό διάστημα μεταξύ δύο αποστολών αποτελεσμάτων)

4.3 Λειτουργία

Το κυρίως νήμα ξεκινά διαβάζοντας τις ρυθμίσεις χρησιμοποιώντας τη βοηθητική κλάση `Settings`. Ακολούθως δημιουργεί μια λίστα που θα κρατήσει τα νήματα που εκτελούν τις περιοδικές εργασίες (αντικείμενα τύπου `PeriodicThread`), καθώς και ένα `ThreadPool` για τις μη περιοδικές εργασίες. Το `ThreadPool` αρχικοποιεί την εσωτερική του ουρά από `Runnables` (αρχικά κενή) και ξεκινά τα `WorkerThreads` του. Αμέσως μετά, το κυρίως νήμα δημιουργεί μια ουρά αποτελεσμάτων, τύπου `IBlockingQueue<Result>` (εδώ χρησιμοποιήθηκε η κλάση `WaitNotifyQueue<Result>`), και ξεκινά ένα `SenderThread` το οποίο αναλαμβάνει περιοδικά το άδειασμα της ουράς και την αποστολή των αποτελεσμάτων που αυτή περιέχει στον Aggregator Manager. Το κυρίως νήμα χρησιμοποιώντας τη μέθοδο `addShutdownHook()` της κλάσης `Runtime` και περνώντας της ως παράμετρο ένα αντικείμενο τύπου `ShutdownThread` εξασφαλίζει τον σωστό τερματισμό των νημάτων όταν ληφθεί το σήμα `SIGINT`. Αμέσως μετά δημιουργείται ένας `IJobProvider` (`JobGenerator` ή `JobReader` ανάλογα με τη σχετική ρύθμιση) και το κυρίως νήμα ξεκινά περιοδικά να παίρνει εργασίες από αυτόν και ανάλογα με το αν είναι περιοδικές η όχι να τις υποβάλλει στο `ThreadPool` ή να δημιουργεί ένα νέο `PeriodicThread` το οποίο προσθέτει και στη σχετική λίστα που διατηρεί. Έχει γίνει η επιλογή να υπάρχει ένα μέγιστο μέγεθος της ουράς αποτελεσμάτων έτσι ώστε να αποφεύγεται τυχόν συμφόρηση του δικτύου ή του Aggregator Manager λόγω μεγάλου πλήθους αποτελεσμάτων σε μια αποστολή. Αυτό έχει σαν αποτέλεσμα το ενδεχόμενο τα `WorkerThreads` και `PeriodicThreads` να πέσουν σε αναμονή καλώντας τη μέθοδο `put()` της ουράς αποτελεσμάτων. Για την ουρά από `Runnables` του `ThreadPool` δεν υπάρχει κάποιος περιορισμός μεγέθους έτσι ώστε κάθε φορά που έρχονται νέες εργασίες να είναι δυνατή η υποβολή τους χωρίς να χαθούν.

4.4 Τερματισμός

Κατά τον τερματισμό, το `ShutdownThread` αρχικά ειδοποιεί το κυρίως νήμα να τερματίσει, έτσι ώστε να μην υποβάλλονται νέες εργασίες στο σύστημα. Ακολούθως ειδοποιούνται από το νήμα τερματισμού τα `WorkerThreads` (εμμέσως μέσω της `ThreadPool`) και `PeriodicThreads` (άμεσα) να τερματίσουν. Σε αυτή την φάση, αν κάποιο νήμα βρίσκεται στο στάδιο εκτέλεσης μιας εργασίας, δεν τερματίζει άμεσα αλλά μόνο όταν αυτή ολοκληρωθεί και τοποθετήσει τα αποτελέσματα της στη σχετική ουρά. Το νήμα τερματισμού, αφού περιμένει για τον τερματισμό των `PeriodicThreads` και του `ThreadPool`, ειδοποιεί τελευταίο το `SenderThread` το οποίο έχει την ευκαιρία να αδειάσει την ουρά αποτελεσμάτων από τα τελευταία αποτελέσματα και να τα στείλει στον Aggregator Manager. (Η παραπάνω συμπεριφορά γίνεται εμφανής με χρήση του kill -SIGINT <java pid>, αφού με Ctrl-C από το τερματικό οι nmap διεργασίες τερματίζουν βίαια χωρίς το πρόγραμμα να πάρει τα αποτελέσματα).