Vorlesung „Java I“ SS 04 10. Übung (28. 6. 2004)

http://www.coli.uni-sb.de/cl/courses/java-04/

---

1 Tic-Tac-Toe, Teil 3 (3 Punkte)

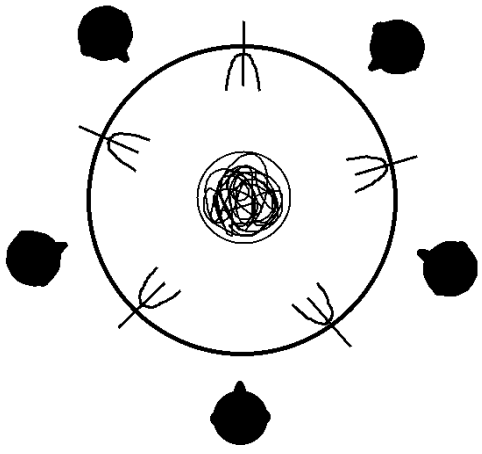
Im dritten Teil der Tic-Tac-Toe-Aufgaben geht es jetzt darum, den *Server* für das Spiel zu programmieren. Die Aufgabe des Servers ist es, für jedes Paar von Clients das Spielbrett zu verwalten, Züge auszuführen, das Spielende zu erkennen und alle Clients über Netzwerk-Nachrichten über den Spielablauf zu informieren. Das Protokoll für die Kommunikation zwischen Client und Server bleibt wie in der 9. Übung beschrieben – nur dass Ihr jetzt die Server-Seite statt der Client-Seite implementiert.

Der Server soll beliebig viele Clients bedienen können. Dafür soll er, sobald er zwei Verbindungen von Clients akzeptiert hat, einen neuen Thread starten, in dem das Spiel für diese beiden Clients verwaltet wird. Sobald dieses Spiel endet, soll sich der Thread beenden.

Um Verbindungen von den Clients zu akzeptieren, legt der Server am Anfang ein einziges Objekt der Klasse `java.net.ServerSocket` für einen Port an, der auf der Kommandozeile übergeben wird. Er kann dann die `accept`-Methode dieses Objekts aufrufen, um eine Verbindung anzunehmen; die Methode gibt ein `Socket`-Objekt zurück, das die Verbindung darstellt.

2 Dining Philosophers (2+2 Punkte)

Ein berühmtes Problem zur Illustration von Deadlocks ist das der “speisenden Philosophen” *(dining philosophers problem)*. In einer vereinfachten Form sitzen dabei n Philosophen um einen runden Tisch, auf dem ein Teller mit Spaghetti steht. Ein Philosoph braucht zwei Gabeln, um damit ordentlich Spaghetti zu essen; leider gibt es auch nur n Gabeln. Ein naiver Algorithmus könnte so aussehen, dass jeder Philosoph zuerst versucht, die Gabel zu seiner linken Hand aufzunahmen und dann die zu seiner rechten. Wenn die Philosophen eine Gabel erst dann wieder ablegen, wenn sie gegessen haben, führt dieser Algorithmus in ein Deadlock.



Schreibe ein Java-Programm, das dieses Deadlock illustriert. Dabei soll jeder Philosoph als ein eigener Thread implementiert sein und die Gabeln als Monitore von Objekten. Achtung: Es kann sein, dass der erste Philosoph sich beide Gabeln greift, bevor der zweite überhaupt die erste nimmt. In diesem Fall bleibt das Deadlock aus (warum?). Du kannst das Deadlock zuverlässig provozieren, indem Du (mit `Thread.sleep()`) eine Verzögerung von 10ms zwischen dem Ergreifen der beiden Gabeln einbaust.

Für die Bonuspunkte denke Dir eine Strategie aus, mit der das Deadlock vermieden werden kann. Erkläre, warum sie funktioniert, implementiere sie und probiere sie aus.

3 Web-Crawler, Teil 3 (2+2 Punkte)

Der dominierende Faktor bei der Laufzeit des Web-Crawlers aus Übung 9 ist die Zeit, die das Programm braucht, um Webseiten herunterzuladen. Ändere das Programm so ab, dass jeder Download einer URL in einem eigenen Thread abläuft, also mehrere HTML-Requests gleichzeitig passieren. Das HTML-Parsing und die Verwaltung der noch zu lesenden URLs können entweder in diesen Threads oder in einem eigenen Thread stattfinden. Zunächst soll jeder Thread die URL ausgeben, die er gleich besuchen wird; die komplette Ausgabe des Programms stellt dann die Menge der besuchten URLs dar.

Bei dieser Aufgabe ist es unbedingt notwendig, die Zugriffe auf die Collections (z.B. zur Verwaltung der schon besuchten Seiten) zu synchronisieren. Dazu kann man entweder die entsprechenden Hilfsmethoden aus `java.util` verwenden oder eigene `synchronized`-Methoden.

Für den ersten Bonuspunkt soll das Hauptprogramm die Menge der besuchten URLs ausgeben, sobald alle Download-Threads abgeschlossen sind. Dazu ist es nötig, dass das Hauptprogramm erkennt, dass alle anderen Threads fertig sind. Hierzu gibt es verschiedene Möglichkeiten, z.B. dass jeder Thread sich bei einem zentralen Objekt `obj` anmeldet, wenn er gestartet wird, und wieder abmeldet, wenn er fertig ist; man kann dann in `obj` mitzählen, wie viele Threads noch am Arbeiten sind.

Für den zweiten Bonuspunkt implementiere einen Timeout für die Download-Threads: Wenn der Thread mehr als eine gegebene Anzahl Millisekunden braucht, um eine Webseite herunterzuladen, soll der Thread beendet werden. Hierzu könnten `Thread.interrupt()` und `java.io.InterruptedIOException` nützlich sein.

Abgabe bis 5. 7. 2004, 9 Uhr
(java-uebungen@coli.uni-sb.de)