מרוצים וסינכרונים

עד עכשיו, השתדלנו מאד שלכל חוט יהיו משתנים נפרדים - לא איפשרנו לחוטים שונים לעדכן את אותו משתנה. בשיעור זה נראה אילו בעיות קורות כשחוטים שונים מנסים לעדכן את אותו משתנה, ואיך אפשר למנוע אותן.

מרוץ הגורם לתוצאה שגויה

ניזכר בתרגיל הכיתה מתחילת השיעור הקודם - חישוב סכום של מערך. נניח שהיינו משתמשים, במקום במערך בגודל 2, במערך בגודל 1 המכיל את הסכום, באופן הבא (ראו בקובץ ParallelSumRaceCondition):

```
int[] sums = new int[1];
Thread t1 = new Thread(() -> {
     for (int i=0; i<array.length/2; ++i)
          sums[0] += Math.pow(array[i], 3);
});
t1.start();
Thread t2 = new Thread(() -> {
     for (int i=array.length/2; i<array.length; ++i)
          sums[0] += Math.pow(array[i], 3);
});
t2.start();
```

הסיבה שאנחנו משתמשים במערך בגודל 1 היא שהקומפיילר לא מרשה להשתמש ב-int מחוץ לחוט ולשנות אותו בתוך החוט. הקומפיילר מנסה להגן עלינו משגיאה, אולם אנחנו יכולים לעקוף את ההגנה הזאת ע"י שימוש במערך בגודל 1.
הריצו את התוכנית כמה פעמים על מערך בגודל 100 מיליון המלא ב-2. מה קורה?
--- אצלי, התוצאה כמעט תמיד קטנה יותר מהערך הצפוי (800 מיליון)!

למה זה קורה?

הסיבה היא, שפעולת ההוספה של איבר במערך לסכום:

```
sums[0] += Math.pow(array[i],3);
```

לא מתבצעת בצורה **אטומית**. למעשה מתבצעות כאן כמה פעולות: המחשב טוען את הערך הנוכחי של sums[0] לתוך רגיסטר, מוסיף לרגיסטר את הערך שצריך להוסיף, ואז שומר את ערך הרגיסטר החדש בתוך sums[0]. מתישהו, באמצע התהליך הזה, מערכת-ההפעלה מעבירה את הביצוע לחוט אחר, שגם הוא מוסיף ערכים ל-sums[0]. ואז אנחנו חוזרים לחוט הראשון, והוא ממשיך בדיוק מאותה נקודה שהפסיק - הוא שומר את ערך הרגיסטר שלו לתוך sums[0]. הבעיה היא, שערך הרגיסטר שלו לא כולל את כל הערכים שנוספו ע"י החוט השני! כל ההוספות שבוצעו ע"י החוט השני נמחקו! המצב הזה נקרא "מרוץ" - **race condition**.

יש כמה פתרונות לבעיה.

1. הפתרון הטוב ביותר הוא **לא להשתמש בכלל במשתנים משותפים לכמה חוטים**. בשיעור הקודם ראינו שאפשר לעשות זאת ע"י שימוש בזרמים בשיטת מיפוי-צמצום; זו השיטה היעילה והבטוחה ביותר לבצע חישוב כמו בתרגיל הנוכחי.

2. אבל, לפעמים אנחנו חייבים להשתמש במשתנים משותפים. לדוגמה, חישבו על אפליקציה כמו headstart, המקבלת תרומות מאנשים שונים וצריכה להציג לכל אחד מהם את הסכום המעודכן. כאן, אנחנו צריכים להחזיק משתנה אחד המציין את הסכום, ולאפשר לכמה חוטים לעדכן אותו.
לשם כך אפשר להשתמש בפעולת **סינכרון** - מילת המפתח synchronized. נשנה את החוט הראשון באופן הבא (ואת השני באופן דומה):

```
for (int i=0; i<array.length/2; ++i) {
     double toAdd = Math.pow(array[i], 3);
     synchronized(sums) {
          sums[0] += toAdd;
     }
}
```

אחרי המילה "synchronized" צריך לבוא בסוגריים עגולים שם של אובייקט כלשהו בג'אבה, ואחר-כך בלוק של קוד מוקף בסוגריים מסולסלים. לכל אובייקט בג'אבה יש **מנעול** פנימי; משמעות הפעולה synchronized(obj) היא "קח את המנעול הפנימי על האובייקט obj, ורק אחר-כך תיכנס לבלוק הקוד". אם המנעול תפוס, החוט יחכה עד שהמנעול יתפנה. מכאן, שבלוק-הקוד המוקף ב-synchronized יכול להיות מבוצע ע"י חוט אחד בכל פעם - לא ייתכן ששני חוטים שונים יבצעו אותו באותו זמן. כך נמנע המרוץ (race) ומתקבלת תוצאה נכונה.
כדי להבין איך עובד synchronized תיזכרו בטיול השנתי בתיכון - כל עוד הולכים בשטח פתוח, אפשר ללכת במקביל, אבל כשמגיעים למנהרה צרה, חייבים לעבור בה אחד אחד. קטע המוקף ב-synchronized הוא כמו מנהרה צרה, שהחוטים חייבים לעבור בה אחד אחד. מובן שהדבר מאט מאד את התהליכים; במקרה שלנו החישוב איטי כמעט כמו ביצוע סדרתי, אולי אפילו יותר.
לכן כדאי להשתמש בפתרון זה רק כשממש אין ברירה אחרת.

מרוץ הגורם ללולאה אינסופית

מרוץ בין חוטים יכול לגרום לא רק לתוצאה שגויה אלא גם לצרות הרבה יותר גדולות. בקובץ HashMapRaceCondition יש דוגמה לשימוש שגוי ב-HashSet. יוצרים HashSet ריקה, ושני חוטים שמכניסים לתוכה ערכים במקביל. כשמכניסים מספר קטן של ערכים (נניח מיליון), מקבלים "רק" תוצאה שגויה - למרות שהחוטים מכניסים ערכים שונים, מספר הערכים הכולל בקבוצה קטן ממיליון - חלק מהערכים הולכים לאיבוד. אבל כשמכניסים יותר ערכים (נניח 9 מיליון) המחשב נתקע! למה זה קורה? - כי HashSet זה מבנה מורכב. כפי שלמדתם בקורס מבנה-נתונים, טבלת עירבול ממומשת כמערך שהאינדקס שלו הוא ערך-העירבול של המפתח. אם יש שני מפתחות שונים עם אותו ערך-עירבול, נוצרת התנגשות, צריך להעביר את המפתח החדש למקום אחר וליצור קישור בין המקומות. אבל אם, בזמן שחוט עסוק בלשנות את הקישורים הפנימיים, מערכת ההפעלה מכניסה חוט אחר - עלול להיווצר מעגל של קישורים, שיגרום למחשב להיתקע בלולאה אינסופית.
הלולאה הזאת מאד קשה לגילוי - הרי בקוד רואים רק לולאות סופיות. מאד קשה להבין למה המחשב נתקע! לכן צריך להיזהר מאד מאד ממשתנים משותפים.

ואם בכל-זאת צריך אותם? - אז יש שני פתרונות:

- משתמשים ב-synchronized כמו שראינו קודם;
- משתמשים במחלקה ConcurrentHashMap. המחלקה הזאת משתמשת באופן חכם בכמה מנעולים (במקום במנעול יחיד כמו synchronized) וכך מאפשרת לחוטים שונים לעדכן חלקים שונים של המפה באופן בטוח, בלי להפריע אחד לשני.

בחבילה java.util.concurrent ניתן למצוא עוד סוגים רבים של אוספים בגירסת concurrent - גירסה בטוחה לשימוש במקביל. מומלץ להשתמש באוספים אלה בכל מקרה שצריכים לעדכן אוספים מכמה חוטים שונים. זה הרבה יותר פשוט ובטוח מלהשתמש ב-synchronized!

אפליקציה מרובת-חוטים עם אוסף משותף מסונכרן

לסיום נראה דוגמה לשימוש באוספים משותפים מסונכרנים.

המחלקה למדעי המחשב החליטה לממן את המעבר לבניין החדש ע"י מכירה פומבית של עטים. לצורך המכירה נכתבה אפליקציית רשת. השרת נמצא בקובץ AuctionServer. הוא דומה לשרתים שראינו בשיעורים הקודמים, אלא שהפעם יש לו משתנה מסוג Map<Integer, Set<String>>. המשתנה הזה הוא מפה שהמפתח שלה הוא מספר שלם - המחיר שהוצע עבור העט (באגורות שלמות), והערך שלה הוא קבוצה של מחרוזות - קבוצת המשתמשים שהציעו את המחיר הזה. כדי לוודא שהמפה מסודרת בסדר יורד של המחיר, נשתמש ב-TreeMap ונעביר לבנאי משווה הפוך, באופן הבא:

```
Map<Integer, Set<String>> bids = new TreeMap<>(
          (x,y) -> Integer.compare(y, x)
);
```

הבקשה מגיעה בצורת מחרוזת מופרדת בפסיק, החלק הראשון הוא שם המשתמש והחלק השני הוא ההצעה באגורות שלמות. הטיפול בכל בקשה מתבצע בחוט נפרד. הקוד שבתוך החוט מתחיל כך:

```
final String input = request.getRequestURI().getQuery();
String[] inputs = input.split(",");
String name = inputs[0];
int bid = Integer.valueOf(inputs[1]);

synchronized(bids) {
     bids.putIfAbsent(bid, new LinkedHashSet<>());
     bids.get(bid).add(name);
     StringBuilder outputBuilder = new StringBuilder();
     for (int b: bids.keySet())
          outputBuilder.append("<div>"+b+": "+bids.get(b)
+"</div>");
     output = outputBuilder.toString();
}
```

(בנוסף יש שם גם טיפול בחריגות וכתיבת הפלט - דיברנו על זה בשיעורים קודמים).

שימו לב - אנחנו מקיפים ב-synchronized רק את החלק שבו ניגשים למשתנה המשותף bids.get.

חידה: במקום להשתמש ב-synchronized יכולנו גם להשתמש ב-ConcurrentHashMap, אבל התוצאות היו שונות במקרים מסויימים. מדוע?

צד-הלקוח של האפליקציה הזאת נמצא ב-lesson7/auction.html. הוא מדגים עוד תכונה נחמדה של html - קל מאד לכתוב ממשק-משתמש בעברית. כל מה שצריך לעשות הוא לכתוב בתג html בראש הקובץ "dir='rtl' lang='he'".

מקורות

- Cay Horstmann, “Core Java for the Impatient”, chapter 10.
- Udemy, Java Multithreading Course, https://www.udemy.com/java-multithreading/

סיכם: אראל סגל-הלוי.