מעבדה מספר 2

ריבוי חוטים – Multi-Threading

1. מבוא

שפות התכנות אינן מאפשרות למתכנת, בדרך כלל, לבצע/לכתוב פעילויות שתתבצענה בו זמנית. לרוב, שפות תכנות מאפשרות בקרת זרימה סדרתית, כך שניתן לכתוב תוכנית בה רק פעולה אחת תתבצע בכל רגע נתון. בתוכנית זו הפעולה הבאה תתבצע רק אחרי שהפעולה הקודמת תתבצע.

באופן היסטורי, תכנות מקבילי מומש ע"י מערכת ההפעלה.

שפת Ada שפותחה ע"י משרד ההגנה האמריקאי, הפכה את הכתיבה הבסיסית של מקביליות לזמינה לספקים של משרד ההגנה שפיתחו מערכות צבאיות של שליטה ובקרה. Ada אינה בשימוש נרחב בתעשיה ובאקדמיה. שפת Java הפכה את התכנות המקבילי לזמין.

תכנות מקבילי יכול להתבצע על מחשב אחד עם מספר ליבות (ואז המשימות באמת מתבצעות במקביל) או על מחשב עם ליבה אחת, ואז יש צורך לחלק את זמן המעבד בין המשימות השונות, כי בפועל בזמן נתון המעבד יכול לבצע פקודה אחת בלבד.

תכנות מקבילי מגדיל את יעילות הביצועים של מערכת עם מעבד יחיד. אפליקציות רבות משתמשות בתכנות מקבילי. לדוגמה, כאשר טוענים קובץ גדול (של תמונה, קליפ, סרט) שמאוחסן ברשת, המשתמש לא מעוניין להמתין עד שהקובץ כולו יטען לפני תחילת ה־playback. כדי לפתור את הבעיה ניתן להפעיל מספר תהליכים/חוטים. כדי להבטיח playback תקין יש לסנכרן בין התהליכים/חוטים ולהבטיח שהתצוגה תתבצע רק אחרי שנטען חלק "מספיק" גדול מהקובץ.

תכנות מקבילי מורכב יותר מתכנות רגיל ויש לכך סיבות רבות:

א. הראש שלנו לא רגיל לחשוב "באופן מקבילי" ולכן פיתוח של מערכת/אלגוריתם מקבילי קשה יותר (נסו לדמיין את עצמכם קוראים במקביל מספר ספרים, כל פעם קטע קצר מספר אחר: עליכם לקרב ספר אחד, להיזכר מה קראתם בסבב הקודם, לקרוא קטע קצר, "לסמן" את המקום אליו הגעתם בקריאה בסבב הנוכחי ולעבור לספר הבא....).

ב. נדרשת עבודה רבה על מנת למנוע מצבים של חוסר סנכרון בין התהליכים/חוטים.

ג. הרצת תוכנית מקבילית אינה דטרמיניסטית לרוב ולכן תהליך ה־debugging מורכב במיוחד.

ד"ר מלכי גרוסמן, ליאל פרידמן, עבד אלפתאח

2. רקע

תהליך (process) במערכות הפעלה הוא יחידת ביצוע של התוכנה. תוכנה היא אוסף פסיבי של פקודות. לעומת זאת, תהליך הוא הביצוע הממשי של אוסף פקודות אלה. לכל תהליך יש את מרחב הזכרון שלו (address space) ולכן תהליכים שונים שקופים אחד לשני. היתרון הוא שקל לבצע כל תהליך בנפרד. אולם זה מקשה מאד על ההידברות בין תהליכים.

התהליכים מורצים על ידי מערכת ההפעלה. מודול של מערכת ההפעלה הנקרא scheduler מתזמן את התהליכים כאשר כל תהליך מקבל פרק זמן (time slice) לרוץ על המעבד. כאשר ה־time slice של התהליך מסתיים, התהליך מפסיק להתבצע וה-scheduler מתזמן תהליך אחר לביצוע.

קיימות שיטות רבות לקביעת גודל ה־time slice ולניהול התורים של התהליכים. בדרך כלל גודל ה־time slice קטן מספיק כדי שהמשתמש "יראה את התהליכים רצים בו זמנית". כאשר המערכת מכילה יותר מליבה (core) אחד, כמה תהליכים יכולים לרוץ במקביל על הליבות הקיימות.

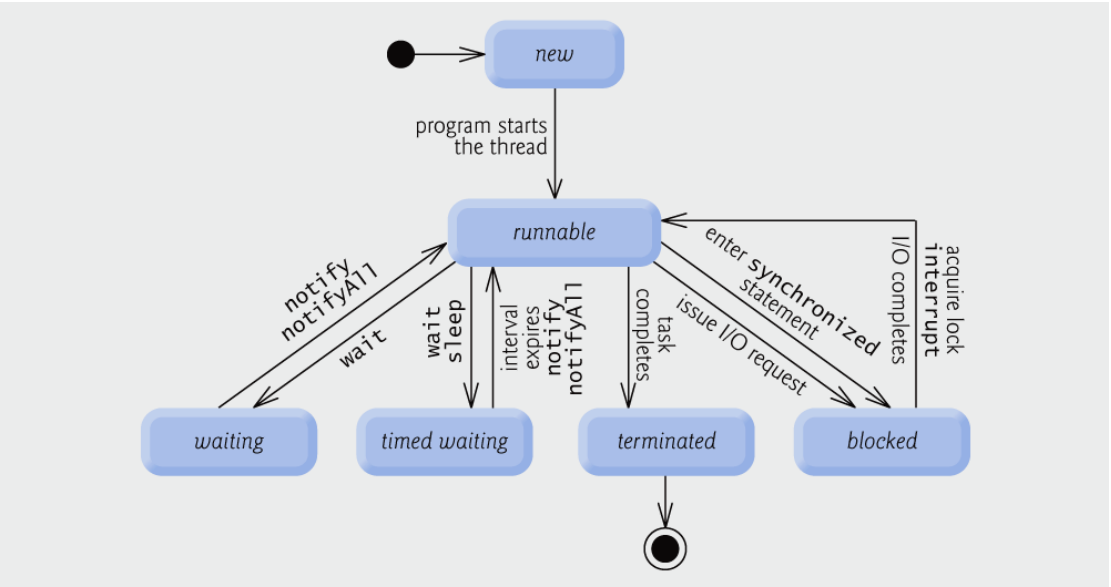
ההחלפה בין תהליכים נקראת החלפת הקשר (context switch). תוכנת התהליך אינה מודעת ואינה משפיעה על התזמון של התהליכים. התזמון יכול להתרחש בין כל שתי פקודות מכונה של התוכנה. פעולה זו דורשת כמובן התערבות של מערכת ההפעלה וכוללת לא מעט פקודות.

חוטים ב־JAVA (threads) הם תהליכים "קלים" (lightweight processes). כל תהליך יכול להכיל מספר רב של חוטים. כל תהליך מכיל תמיד חוט אחד הנקרא החוט הראשי. חוט זה יכול ליצור חוטים חדשים שיהיו שייכים אליו.

לכל חוט ב־JAVA יש מחסנית משלו, PC (Program Counter) משלו והם חולקים את הערימה (heap) בה מקצים את האובייקטים, דבר ההופך את שיתוף הפעולה בין חוטים לקל יותר, אבל יש להקפיד שהחוטים לא יפריעו זה לזה. העובדה שהחוטים חולקים ביניהם את הערימה הופכת את פעולת החלפת ההקשר למהירה יותר בהשוואה להחלפת הקשר של תהליכים, אך עדיין יש להתחשב בה בהערכת הביצועים של המערכת.

3. חוטים ב־Java

בכל רגע נתון, חוט יכול להיות באחד מכמה מצבי החוט המודגמים בעזרת תרשים המצבים של UML הבא:



New and Runnable Status

חוט חדש מתחיל את מחזור החיים שלו במצב new. הוא נשאר במצב זה עד שהתוכנית מתחילה את החוט ע"י העברתו למצב runnable. חוט הנמצא במצב runnable נחשב למבצע את המשימה שלו.

Waiting State

לעיתים חוט הנמצא במצב Runnable מועבר למצב Waiting בזמן שהוא ממתין לביצוע מטלה של חוט אחר. הוא יחזור למצב Runnable רק לאחר שהחוט האחר יודיע לו שהוא יכול להמשיך בביצוע.

Timed Waiting State

חוט במצב Runnable נכנס למצב Timed Waiting לפרק זמן מוגדר. המעבר חזרה למצב Runnable נעשה כאשר פרק הזמן המוגדר פוקע או האירוע לו ממתין החוט מתרחש. מקרה נוסף בו חוט מועבר למצב של Timed Waiting הוא כאשר "משכיבים" את החוט לישון (ע"י המתודה sleep של המחלקה Thread) לפרק זמן הנקרא sleep interval במקרים בהם אין לחוט מה לבצע. בחלוף ה־sleep interval החוט חוזר למצב Runnable. לדוגמה: מעבד מילים יכול להכיל חוט שמבצע גיבוי בצורה מחזורית של המסמך הנוכחי למטרת התאוששות במקרה של תקלה (נפילת מתח למשל). אם החוט לא יישן בין שני גיבויים, הוא יבצע לולאה בה ייבדק התנאי אם צריך לעשות גיבוי. לולאה זו צורכת זמן מעבד מבלי לבצע דבר מה משמעותי ולכן יעיל יותר ל"השכיב לישון" את החוט.

חוטים הנמצאים במצב Waiting או במצב Timed Waiting לא יכולים להשתמש עם המעבד אפילו אם הוא זמין.

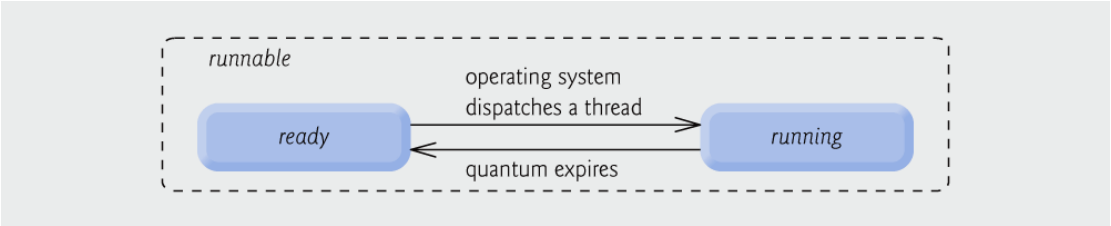
Blocked State

חוט במצב Runnable יעבור למצב blocked כאשר הוא מנסה לבצע משימה שאינה יכולה להסתיים מיד והיא תלויה בביצוע של דבר מה על ידי חוט אחר. במקרה זה, החוט ימתין עד לביצוע אותם דברים בהם המשימה תלויה. לדוגמה, כאשר חוט מבקש לבצע פעולת קלט/פלט, מערכת ההפעלה חוסמת אותו עד שבקשת הקלט/פלט (I/O) requestתסתיים ולאחר מכן הוא יחזור למצב Runnable. חוט חסום לא יכול להשתמש במעבד.

Terminated State

חוט במצב Runnable נכנס למצב Terminate (שנקרא גם dead state) כאשר הוא מסיים את המטלה שלו או כאשר הוא מסתיים מסיבה אחרת (כמו שגיאה).

Runnable State – מבט של מערכת ההפעלה



ברמה של מערכת ההפעלה, מצב Runnable של Java מורכב משני מצבים נפרדים.

כאשר חוט מועבר לראשונה למצב Runnable ממצב new, הוא עובר למצב ready. חוט במצב ready מועבר למצב running (כלומר מצב ביצוע) כאשר מערכת ההפעלה מציבה עבורו מעבד. הפעולה של הצבת מעבד לחוט נקראת dispatching the thread – שיגור החוט. ברוב מערכות ההפעלה, לכל חוט ניתן זמן קצר של המעבד הנקרא quantum או time slice בו הוא מבצע קוד. מעברים בין המצבים ready ו־running מטופלים על ידי מערכת ההפעלה בלבד. ה-JVM אינה "רואה" את המעברים.

Priority

לכל חוט ב־Java יש עדיפות (priority) שעוזרת לקבוע את הסדר בו מקצים חוטים. העדיפות מוגדרת בטווח של MIN_PRIORITY (קבוע 1) ועדMAX_PRIORITY (קבוע 10). ברירת המחדל – כל חוט שנוצר מקבל עדיפות של NORM_PRIORITY שערכה 5.

כל חוט יורש את העדיפות של החוט שיצר אותו. לכן, אם לא נקבע אחרת, החוט הראשי שנוצר בתוכנית יהיה עם עדיפות 5, וכך גם כל החוטים שיווצרו בו יהיו עם עדיפות 5.

באופן לא רשמי, חוטים עם עדיפות גבוהה יותר חשובים יותר למערכת, ועליהם לקבל זמן מעבד לפני חוטים עם עדיפות נמוכה. אבל עדיפות לא מבטיחה את הסדר בו הם יתבצעו. הקבועים MAX_PRIORITY, MIN_PRIORITY ... NORM_PRIORITY, מוכרזים במחלקה Thread.

הקצאת חוטים היא תלוית פלטפורמה: ההתנהגות של תוכנית מרובת חוטים יכולה להשתנות מאוד בהתאם למערכת עליה מורצת התוכנית (בתלות במימוש של Java).

יצירת חוטים ב־Java

כמו כל דבר ב־Java, חוטים הם אובייקטים (כל אובייקט ב־Java יורש ממחלקת Object). כל חוט הוא מופע של מחלקת java.lang.Thread או תת מחלקה שלה. למחלקה Thread מתודה בשם run()והקוד של מתודה זו יתבצע כאשר החוט יתבצע.

ניתן לייצר חוט בכמה דרכים.

דרך 1 – יצירת מופע של המחלקה Thread

ניתן ליצור מופע של המחלקה Thread באופן הבא:

```
Thread myThread = new Thread();
```

יש לשים לב כי יצירת אובייקט מטיפוס Thread לא מפעילה שום חוט במערכת ההפעלה. על מנת להתחיל את הריצה של החוט החדש, צריך להפעיל את מתודת start() של המחלקה:

```
Thread myThread = new Thread();
myThread.start();
```

דרך 2 – הגדרת תת-מחלקה של המחלקה Thread ויצירת מופע שלה

```
public class MyThread extends Thread { /* Code here */ }

MyThread myThread = new MyThread();
myThread.start();
```

דרך 3 – יצירת מופע של מחלקה המממשת את הממשק Runnable הבא:

```
public interface Runnable {
    void run();
}
```

יצירת החוט נעשית ע"י העברת המופע של המחלקה המממשת את Runnable לבנאי של המחלקה Thread באופן הבא:

```
public class ThreadLab implements Runnable {
    public void run() { /* Code here */}
}

ThreadLab threadLab = new ThreadLab();
Thread thread1 = new Thread(threadLab);
```

בשלוש הדוגמאות הנ"ל לא התייחסנו לקוד עצמו אותו יבצע החוט בזמן ביצועו. הקוד שיתבצע הוא קוד המתודה ()run. מתודה זו יש לכתוב.

אם יוצרים את החוט בדרך 2 יש לדרוס את המתודה ()run של מחלקת העל (Thread). לדוגמה:

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("MyThread running");
    }
}
```

אם יוצרים את החוט בדרך 3 יש לממש את המתודה ()run המוגדרת בממשק Runnable. לדוגמה:

```
public class ThreadLab implements Runnable {

    @Override

    public void run() {

        System.out.println("MyThread running");

    }

}

ThreadLab myThread = new ThreadLab();

Thread thread1 = new Thread(myThread);
```

הבחירה בין הורשת Thread לבין מימוש Runnable היא בחירה של המתכנת. היתרון במימוש Runnable שרק כך אפשר ליצור מחלקה שהיא Thread ובנוסף היא יכולה להרחיב מחלקה אחרת (מכיוון ש Java לא מאפשרת הורשה מרובה).

מתודות בקרה של חוטים

להלן רשימה חלקית של מתודות שימושיות של המחלקה Thread.

את הרשימה המלאה ניתן לראות בקישור הבא:

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Thread.html

מתודה	הסבר
void setName(String name)	ניתן לתת שם לכל thread ב־Constructor, או דרך המתודה setName. את השם ניתן לקרוא באמצעות מתודת getName בזמן ריצת החוט. מתן השמות יכול לעזור ל־debugging ותורם לתוכנה מובנת יותר למתכנת.
static void sleep(long millis)	מתודה סטטית המעבירה את החוט למצב Timed Waiting לפרק זמן (הפרמטר millis).

static Thread currentThread()	מתודה סטטית המחזירה את החוט הנוכחי.
void join()	מתודה המחכה עד שהחוט הנתון יסיים את הריצה שלו. הפקודה thread.start()אומנם מריצה את החוט החדש אבל לא מחכה עד שהוא יסתיים (אנחנו גם לא רוצים שהיא תעשה זאת כי אז נאבד את המקביליות). לכן אם חוט אחד בשלב מסוים צריך לחכות עד שחוט אחר (thread) יסתיים, הוא צריך לקרוא למתודה thread.join(). מתודה זו היא מתודה חוסמת (blocking).המתודה לא חוזרת ישר אלא רק אחרי תנאי מסוים (במקרה שלנו התנאי הוא סיום ה־thread).

הערה: המתודות sleep ו־join יכולות לזרוק InterruptedException. המשמעות של הדבר היא שהחוט שהמתין (לחלוף זמן או לסיום של חוט אחר למשל) יצא מההמתנה "לפני הזמן" (לפני שחלף פרק הזמן או לפני שהחוט הסתיים).

שאלה 1

נניח ש־thread הוא מופע מטיפוס המחלקה Thread.

מה ההבדל בין הפעלת המתודה thread.start() לבין הפעלת המתודה thread.run()?

4. התוכנית המקבילית הראשונה

עכשיו אנחנו מוכנים לכתוב את התוכנה המקבילית הראשונה שלנו. החוט הראשי מפעיל 10 חוטים, מ־1 עד 10. כל

```
public class HelloWorld implements Runnable {
    public void run() {
        System.out.println("Hello world from thread number "
                           + Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        Thread[] threads = new Thread[10]; // create an array of threads
        for (int i = 0; i < 10; i++) {
            String threadName = Integer.toString(i);
            // create threads
            threads[i] = new Thread(new HelloWorld(), threadName);
        }
        for (Thread thread : threads) {
            thread.start(); // start the threads
        }
        for (Thread thread : threads) {
            try {
                thread.join(); // wait for the threads to terminate
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("That's all, folks");
    }
}
```

אחד מהחוטים מדפיס את השם שלו למסך.

הסבר:

מחלקת HelloWorld מממשת את הממשק Runnable.

שורות 5-2: מימוש של המתודה ()run, הניגשת ל־currentThread ומדפיסה את השם שלו.

המתודה main: זהו הקוד של החוט הראשי המבצע את הדברים הבאים:

- מייצר 10 מופעים של המחלקה Thread (שורה 12). כל מופע מקבל בבנאי שלו מופע מטיפוס המחלקה HelloWorld ושם ייחודי.
- מפעיל את מתודת ()start של כל Thread.
- ממתין עד שכל החוטים יסתיימו באמצעות מתודת ()join.

שאלה 2

1. הריצו את התוכנית. מהו הפלט שלה?
2. מה קורה אם מורידים את הלולאה השלישית (שורות 16 – 24)? תריצו את התוכנית בגרסה ללא הלולאה והראו את הפלט. הסבירו את התוצאה.
3. מה קורה אם היינו קוראים ל־thread.join ישר אחרי thread.start בתוך הלולאה השנייה אחרי שורה 14? הריצו את התוכנית בגרסה של join אחרי start והראו את הפלט. הסבירו את התוצאה.

<u>כתיבת תכנית מרובת חוטים:</u>

יש לשים לב לכך שהחתימה של מתודת ()void run, המתודה בה נכלל הקוד של החוט, אינה מקבלת פרמטרים ואינה מחזירה כל ערך מוחזר. לכן צריך למצוא דרך כדי להעביר לחוט פרמטרים ולקבל ממנו תוצאות עיבוד. לשם כך ניתן להגדיר משתנה או משתנים ולהעביר אותם לחוט באמצעות הבנאי.

לצורך תרגול יש לפתור את הבעיה הבאה:

מעוניינים לכתוב תכנית קטנה המחשבת ומדפיסה את כל המספרים הראשוניים מעשר ועד מיליון ולמדוד את משך הזמן שהחישוב ארך.

יש לבצע את החישוב בשתי דרכים:

א. בדרך הרגילה. כלומר באמצעות תכנית Prime עם חוט ראשי (החוט הכולל את מתודת ה־main).

ב. באמצעות ריבוי חוטים: יש לכתוב שתי תכניות בשם PrimeThreads1 ו-PrimeThread2 לחישוב הסכום הנ"ל בעזרת 10 חוטים. PrimeThread1 – כל חוט יבדוק תחום של מספרים וידפיס את המספרים הראשוניים שימצא. PrimeThread2 - כל חוט יבדוק תחום של מספרים וישמור את המספרים הראשונים במערך. בסיום פעולת החוטים תדפיס התכנית את המספרים הראשוניים בצורה ממוינת. האם יש הבדל בסדר המספרים הראשוניים בין שתי התוכניות?

כדי לחשב את משך זמן החישוב יש לבדוק את השעה לפני תחילת החישוב ובסיומו. ניתן לעשות זאת עם הקוד המובנה בשפה:

```
import java.util.concurrent.TimeUnit;

public class CalcTime {
    public static void main (String[] args){
        long startTime = System.nanoTime(); // Computation start time
        /* Do computation Here */
        // The difference between the start time and the end time
        long difference = System.nanoTime() - startTime;
        // Print it out
        long minutesInDifference = TimeUnit.NANOSECONDS.toMinutes(difference);
        long secondsInDifference =
            TimeUnit.NANOSECONDS.toSeconds(difference) -
TimeUnit.MINUTES.toSeconds(minutesInDifference);
        System.out.format(
            "Total execution time: %d min, %d sec\n",
            minutesInDifference,
            secondsInDifference
        );
    }
}
```

שנו את הקוד שלכם כך ששלושת התוכניות ידפיסו את המספרים הראשוניים ואת משך זמן ביצוע התכנית.

שאלה 3

הריצו את התוכניות Prime, PrimeThreads1 ו־PrimeThreads2 שכתבתם מספר פעמים.

1. האם יש יתרון לביצוע החישוב באמצעות מספר חוטים? הסבירו.
2. האם משך זמן החישוב באמצעות חוטים הוא קבוע? מדוע?

5. מנגנוני סנכרון

לעיתים חוטים שונים צריכים לגשת לנתונים משותפים. נסתכל על הדוגמה הבאה של UnsafeCounter:

פעולת addValue() אינה פעולה אטומית אלא מורכבת משלוש פעולות:

```
public class UnsafeCounter {
    private int counter = 0;

    public void addValue(int val) {
        counter = counter + val;
    }

    public int getCounter() {
        return counter;
    }
}
```

1. קריאת הערך הנוכחי של ה־counter.
2. פעולת החישוב של הערך החדש.
3. כתיבת הערך החדש לתוך ה־counter

התבנית הנ"ל נפוצה מאד בתכנות ונקראת Read-Modify-Write. נתונה התוכנית הבאה:

```
public class ReadModifyWrite implements Runnable {
    private UnsafeCounter counter;
    private int val;

    public ReadModifyWrite(UnsafeCounter counterRef, int ival) {
        counter = counterRef;
        val = ival;
    }

    public void run() {
        counter.addValue(val);
    }
}

public class CounterApp {
    public static void main(String[] args) {
        UnsafeCounter sharedCounter = new UnsafeCounter();
        Thread[] threads = new Thread[10];
        // create 10 threads
        for (int i = 0; i < 10; i++)
            threads[i] = new Thread((new ReadModifyWrite(sharedCounter,
2)), "Thread " + i);
        // start the threads
        for (Thread thread : threads) {
            thread.start();
        }
        for (Thread thread : threads) {
            try {
                thread.join(); // wait for the threads to terminate
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Counter =" + sharedCounter.getCounter());
    }
}
```

בתוכנית הנ"ל מריצים עשרה חוטים שהם מופעים של המחלקה `ReadModifyWrite`. כל חוט מקדם Counter (מטיפוס המחלקה `UnsafeCounter`) בערך val (המועברים אליו כפרמטרים באמצעות הבנאי).

במתודה main היוצרת ומריצה את החוטים, מעבירים לכל החוטים אותו Counter (בשורות 18 ו־20 מועבר האובייקט sharedCounter שנוצר בשורה 14 כפרמטר של הבנאי).

לאחר סיום פעולת כל החוטים מדפיסים את ערך המונה.

התוצאה אמורה להיות 20.

בהרצות חוזרות של התוכנית לא תמיד מתקבל ערך 20. נסו להריץ את התוכנית מספר פעמים.

ההסבר לכך הוא שהפעולה של קידום ה־Counter (שורה 5 בקוד המחלקה `UnsafeCounter`) אינה פעולה אטומית. היא מתורגמת למספר פקודות מכונה ופעולת החוט המבצע אותה עלולה להיפסק במהלכה (במידה וחלף ה־time sliceשל החוט והמעבד ילקח ממנו לטובת חוט אחר).

נסתכל על התזמון הבא:

החוט הראשון קורא ערך 0 מהמונה ואחר כך נלקח ממנו המעבד. המונה עדיין לא קודם.	
	חוט אחר קיבל את המעבד וקורא 0 מהמונה. גם חוט זה לא מסיים את הפעולה ונלקח ממנו המעבד.
החוט הראשון חוזר להתבצע ומחשב את הערך החדש 2 ורושם אותו בחזרה.	
	החוט האחר חוזר להתבצע וגם הוא מחשב את הערך החדש 2 ורושם אותו בחזרה.

התזמון שתואר אפשרי מכיוון שלתוכנה אין שום שליטה על ה־scheduling של החוטים במערכת ההפעלה.

מגדירים מבנה נתונים מסוים כ־thread-safe אם הפעלת המתודות שלו בכמה חוטים במקביל אינה תלויה בתזמון של מערכת ההפעלה (זו הגדרה לא פורמלית). מחלקת UnsafeCounter אינה thread-safe , מכיוון שהתוצאה של הפעלת מתודת addValue אינה דטרמיניסטית.

קטע הקוד של write-modify-read בדוגמה של UnsafeCounter מהווה דוגמה קלאסית ל־critical section - קטע קוד שצריך להתבצע בצורה אטומית (כפעולה אחת) . הדרך הפשוטה ביותר להגן על critical section היא מניעה הדדית (mutual exclusion): אם לא נאפשר ליותר מחוט אחד לבצע את הפעולות של ה־critical section אז מובטח לנו שהחוטים לא יפריעו אחד לשני. הדרך הפשוטה ביותר להשיג מניעה הדדית ב־Java היא באמצעות synchronized statements.

```
Object o = new Object();
synchronized(o) { /* Do stuff */ }
```

בדוגמה אנחנו רואים סינכרוניזציה על משתנה o מטיפוס Object. ב־Java ניתן לעשות סינכרוניזציה על כל אובייקט. למעשה, לכל אובייקט ב־Java קיים מנעול מובנה. הכללים לשימוש במנעול הזה פשוטים מאוד:

- בתחילת הריצה המנעול פתוח. אם חוט A קורא ל־synchronized(o) והמנעול פתוח, אז החוט מצליח "לרכוש" את המנעול ולנעול אותו.
- אם גם חוט B מנסה לקרוא ל־synchronized(o) והמנעול סגור (כלומר חוט אחר כבר נעל אותו) B יחכה על המנעול של o עד שהוא ייפתח. כלומר ריצת חוט B תיפסק והוא יועבר למצב Blocked State. הריצה שלו תתחדש רק לאחר שיצליח "לרכוש" את המנעול.
- כאשר חוט A מסיים לבצע את את ה־synchronized block הוא משחרר את המנעול.
- כפי צוין לעיל, לכל אובייקט ב־Java יש מנעול מובנה ולכן למחלקתObject קיימות מתודות ()lock ו־ ()unlock. כתיבת הבלוק עם המשפטים ()o.lock בתחילתו ו־()o.unlock בסופו שקולה לקטע synchronized(o) {}.
- יש לשים לב לכך שכאשר כמה חוטים מחכים לאותו מנעול, רק אחד מהם יצליח "לרכוש" אותו כאשר הוא יפתח (אין שום הבטחה לגבי החוט המנצח).

כעת, נראה איך ניתן להגן על ה־counter באמצעות synchronized statement:

```
public void addValue(int val) {
    synchronized(this) {
        counter = counter + val;
    }
}
```

אם שני החוטים ינסו להפעיל את המתודה addValue, הראשון שיגיע לשורה synchronized(this) "ירכוש" את המנעול של ה־counter והשני יצטרך לחכות עד שהראשון יסיים את הפעולה שלו. השגנו מניעה הדדית!

במקרה שלנו, ה־critical section של מתודת addValue הוא המתודה כולה. במקרים כאלה ניתן להצהיר על המתודה בתור synchronized באופן הבא:

```
public synchronized void addValue(int val) {
    counter = counter + val;
}
```

ה־compiler של Java יפעיל את הבלוק של synchronized(this) פרוס על כל המתודה.

במקרה של מתודה סטטית, הסנכרון יתבצע על האובייקט שמחזיק את ה־class של המתודה.

יש להבין כמה דברים חשובים על מנעולים:

1. מנעול הוא per object.אם חוט אחד מבצע synchronized(o1) וחוט שני מבצע synchronized(o2) אז החוטים לא יפריעו אחד לשני.
2. אם מגנים על אובייקט מסוים באמצעות מנעול, אז כל הגישות לאובייקט צריכות להיות מוגנות (לדוגמה, אם ישנן שתי מתודות שונות שניגשות לאובייקט שדורש הגנה, אז בשתי המתודות צריכה להיות סינכרוניזציה על אותו המנעול).

3. כדאי לנעול את ה־critical section הקטן ככל האפשר. חשוב להבין שמניעה הדדית בעצם מבטלת את המקביליות ואם ה־critical section יהיה גדול מדי, הביצועים יפגעו.

להלן מספר דוגמאות:

דוגמה 1

```
public class WrongSynchronization {
    // WRONG SYNCHRONIZATION - DON'T DO THAT!!!
    private int counter = 0;

    public synchronized void addValue(int val) {
        counter = counter + val;
    }

    public void removeValue(int val) {
        counter = counter - val;
    }
}
```

בדוגמה לעיל יש שתי מתודות addValue ו־removeValue. שתיהן ניגשות לאותו אובייקט משותף. מתודת addValue היא synchronized אבל מתודת removeValue אינה synchronized. נניח שחוט A מתחיל את המתודה addValue ונועל את המנעול של this, כאשר חוט B יתחיל את המתודה removeValue הוא לא ינסה לנעול אף מנעול. התוצאה היא שהחוט יגש ל־counter בלי שום בעיה – אין מניעה הדדית והתוצאות עלולות להיות שגויות.

דוגמה 2

```
public class WrongSynchronization {
    // WRONG SYNCHRONIZATION - DON'T DO THAT!!!
    private int counter = 0;
    private Object o1 = new Object();
    private Object o2 = new Object();

    public void addValue(int val) {
        synchronized (o1) {
            counter = counter + val;
        }
    }

    public void removeValue(int val) {
        synchronized (o2) {
            counter = counter - val;
        }
    }
}
```

בדוגמה לעיל מתודות addValue ו־removeValue מנסות לתפוס מנעולים שונים, ולכן חוט A שמפעיל את addValue לא יפריע לחוט B שמפעיל את removeValue. שוב, אין מניעה הדדית בקוד והתוצאות עלולות להיות שגויות.

דוגמה 3

```
public class WrongSynchronization {
    // WRONG SYNCHRONIZATION - DON'T DO THAT!!!
    private int counter = 0;

    public void addValue(int val) {
        synchronized (this) {
            counter = counter + val;
            someLongComputation();
        }
    }
}
```

בדוגמה לעיל מתודת addValue מבצעת חישוב ארוך בנוסף לגישה ל־counter. החישוב אינו מהווה חלק מה־critical section. אם שני חוטים יפעילו את מתודת addValue רק אחד מהם יוכל לרוץ, והחישוב הארוך יתבצע בצורה סדרתית ללא צורך. אנחנו שואפים לבצע דברים במקביל עד כמה שאפשר. לכן במקרה כזה צריך לשים את הבלוק של synchronized רק סביב ה־critical section.

6. יחסי Consumer – Producer

ביחסים של יצרן-צרכן היצרן מייצר מידע, מאחסן אותו באובייקט משותף והצרכן קורא את המידע מהאובייקט המשותף. יחסים אלה שכיחים. לדוגמה: אפליקציה המורידה סרט ממאגר זכרון ומציגה אותו על מסך המערכת. אם אפליקציה זו תמומש בחוט אחד בלבד, המשתמש יצטרך להמתין זמן רב עד להורדת הסרט כולו לפני שיוכל להתחיל לצפות בו. לכן לרוב מממשים אפליקציה כזו באמצעות שני חוטים: החוט היצרני (producer thread) יעתיק קטע מידע בגודל קבוע ממאגר הזכרון לתוך אובייקט משותף הנקרא buffer. החוט הצרכני (consumer thread) קורא את המידע מה־buffer ויכול להתחיל להציג את הסרט (למרות שרק מקטע ראשון שלו הורד). יחסים אלה דורשים סנכרון מורכב יותר. צריך להבטיח שהמידע שנקרא נכון ומוצג נכון. כלומר יש להבטיח שהחוט היצרני מביא מידע

חדש ל־buffer רק כאשר הוא ריק (כלומר החוט הצרכני צרך כבר את המידע הקודם שהיה ב־buffer. ולהיפך – צריך לוודא שהחוט הצרכני יציג מידע מה־buffer רק פעם אחת, כלומר רק כאשר יש מידע חדש ב־buffer).

כלומר הפעולות על ה־buffer המשותף צריכות להתבצע רק אם הוא נמצא במצב נכון: ניתן לכתוב לתוכו רק אם הוא ריק וניתן לקרוא ממנו רק אם הוא מלא במידע חדש.

דוגמה:

נחשוב על הדוגמה הקלאסית של producer/consumer. נניח שיש תור משותף והיצרן כותב איברים לתור והצרכן קורא ממנו איברים. כמובן שהיצרן יכול להוסיף איברים לתור רק אם התור אינו מלא. באופן דומה הצרכן יכול לקרוא רק אם יש איברים בתור.

הקוד הבא מממש תור ללא מגבלה בגודל:

```
public class ProducerConsumer1 {
    // BUSY WAIT - DON'T DO THAT!!!
    Queue<Integer> workingQueue = new LinkedList<Integer>();

    public synchronized void produce(int num) {
        workingQueue.add(num);
    }

    public Integer consume() {
        while (true) {
            synchronized (this) {
                if (!workingQueue.isEmpty()) {
                    return workingQueue.poll();
                }
            }
        }
    }
}
```

בדוגמה זו ה־consumer נועל את התור ובודק שהתור אינו ריק. אם התור ריק, אז ה־consumer משחרר את המנעול (על מנת לתת ל־producer את ההזדמנות לגשת לתור) וחוזר על הבדיקה בלולאה אינסופית. הגישה הזאת נקראת לפעמים busy-wait. היא בודקת את התנאי בצורה "אקטיבית" ומבזבזת מחזורי CPU לשווא. לכן אינה מקובלת.

הפתרון שמונע שימוש ב־busy-wait פותח ע"י Hoare ו־Hansen בשנת 1972 ונקרא מוניטור. בנוסף לפעולות של נעילת האובייקט, משתמשים במנגנון ה־monitor המציע פעולות של wait ו־notify והמאפשר המתנה לתנאי מסוים. מנגנון זה ממומש ב־Java בעזרת המתודות wait, notify ו־notifyAll של המחלקה Object.

- כדי לבצע פעולה על אובייקט המותנית בתנאי מסוים, על החוט קודם להחזיק את ה־monitor של האובייקט. אם התנאי מתקיים הוא יכול לבצע את הפעולה. אם התנאי אינו מתקיים, החוט קורא למתודה () waitהמשחררת את ה־monitor בצורה אוטומטית ומכניסה את החוט לתור של ה"ממתינים" על ה־monitor. בזמן ההמתנה מצב האובייקט יכול להשתנות כתוצאה מפעולה כלשהי שתתבצע על האובייקט ע"י חוט אחר.
- אם חוט מסוים משנה את מצב האובייקט (וגורם בכך לקיום תנאי שנדרש ע"י חוט אחר) הוא קורא למתודה ()notifyAll כדי להעיר את כל החוטים הממתינים ל־monitor של האובייקט. חוט שהעירו אותו צריך "לרכוש" שוב את ה־monitor כדי שיוכל להמשיך את הריצה.

הקוד הבא מממש תור עם שימוש במתודות wait() ו־notify():

```
public class ProducerConsumer2 {
    Queue<Integer> workingQueue = new LinkedList<Integer>();

    public synchronized void produce(int num) {
        workingQueue.add(num);
        notifyAll();
    }

    public synchronized Integer consume() throws InterruptedException {
        while (workingQueue.isEmpty()) {
            wait();
        }
        return workingQueue.poll();
    }
}
```

בקוד היצרן מוסיף איבר לתור ומיידע את כל החוטים האחרים (שממתינים שהתור יכיל איברים) על שינוי במצב התור באמצעות המתודה notify().

הצרכן נכנס להמתנה אם התור ריק (שורות 10-12).

7. Concurrent Collections

ב־Java קיים אוסף מגוון של מחלקות המממשות collections מסוגים שונים (כמו מחלקות שמממשותSet, List).

לצערנו, לא ניתן להשתמש ב־collections אלה בצורה ישירה בסביבה מרובת חוטים כי המחלקות האלה אינן thread-safe. הפתרון ניתן בחבילה בשם java.util.concurrent. חבילה זו כוללת סוגים רבים של collections שנכתבו כ־thread-safe. יש לזכור שתמיד עדיף להשתמש ב־collections הקיימים ולא להמציא את הגלגל מחדש.

להלן כמה דוגמאות של מחלקות כאלה:

- BlockingQueue - בנוסף לפעולות Queue הרגילות תומכת המחלקה בפעולות חוסמות של הוצאת איבר (מחכים כל עוד התור ריק) והכנסת איבר (מחכים כל עוד כמות האיברים מעל סף מסוים).
- ConcurrentHashMap - מחלקה שמממשת HashMap עם אפשרות לעבודה מקבילית של כמות לא מוגבלת של חוטים.

העבודה עם thread-safe collections של Java מקלה מאד על התכנות המקבילי (אך לא פותרת את כל הבעיות).

שאלה 4

כתבו תוכנית ב Java-המיישמת מערכת של Producer ו- Consumer בה יש מחסנית מוגבלת בגודל של N איברים. במערכת ישנם **שני חוטים שמכניסים איברים** למחסנית (Producers) ושני חוטים שמוציאים איברים מהמחסנית(Consumers).

תנאים:

1. **המחסנית מוגבלת בגודלה** N איברים. אם היא מלאה, ה -Producers צריכים להמתין עד שיתפנה מקום.
2. אם המחסנית **ריקה** ,ה -Consumers צריכים להמתין עד שיוכנסו איברים חדשים.
3. החוטים המייצרים (Producers) יגרילו מספרים אקראיים בטווח 0 עד 100 ויכניסו אותם למחסנית. כל Producerיכניס בסך הכול 100 **מספרים** למחסנית.
4. החוטים הצורכים (Consumers) יוציאו את המספרים מהמחסנית, כאשר כל אחד ימשיך לקרוא עד שסך כל 100 המספרים הוכנסו והוצאו.
5. יש להדפיס בסוף את **סכום המספרים שהוכנסו למחסנית** על ידי ה -Producers ואת **סכום המספרים שהוצאו** על ידי ה -Consumers ולהשוות בין שני הסכומים.
6. יש לבדוק מה קורה כאשר נגדיל את מספר החוטים שמכניסים וקוראים, לדוגמה ל-4 Producers ו-4 Consumers. מה עשוי לקרות במצב כזה?

שאלות:

1. כתבו את המחלקה BoundedStack המנהלת את המחסנית בצורה מסונכרנתכתבו את המחלקה BoundedStack המנהלת את המחסנית בצורה מסונכרנת
2. כתבו את המחלקות Producer ו -Consumer שמבצעות את פעולות הכנסת והוצאת האיברים מהמחסנית, בהתחשב בכך שכל Producer יגריל מספרים ויכניס למחסנית 100 מספרים.

3. הריצו את התוכנית ובדקו את תפקודה כאשר גודל המחסנית הוא N=5 עם שני Producers ושני Consumers.
4. נסו להגדיל את מספר החוטים ל-4 Producers ו-4 Consumers ותארו מה קורה כשיש יותר חוטים מכותבים וקוראים. מה לדעתכם עשוי לקרות?

הוראות הגשה לדוח מעבדה 2:

1. ההגשה בזוגות בלבד באמצעות הגשה אלקטרונית. ניתן להגיש מספר פעמים. ניתן להגיש באיחור, כל יום איחור יגרור הורדת נקודות לפי נוסחה:

$$LabAssignmentGrade = LabGrade - 2^{LateDays}$$

2. יש להגיש קובץ zip אחד. שם קובץ ה־zip הוא מספרי תעודות הזהות של הסטודנטים מופרדים בגרש תחתון. הקובץ כולל שני קבצים:
   א. קובץ pdf ובו תשובות לשאלות 1-4.
   ב. קובץ JAR שניתן להרצה עם קבצי המקור (.java) וקבצי המחלקות (.class). לצור קובץ JAR לפי קובץ CreatingAJar יש שתי דרכים (הראשונה היא נוחה יותר), כדי לכלול קבצי המקור נא להסתכל על הדף האחרון בקובץ CreatingAJar.
3. אנא הקפידו על הוראות ההגשה.

ע ב ו ד ה נ ע י מ ה!