הרצאה 12 - Concurrency - Advanced Techniques

יום ראשון 30 מאי 2021 09:08

נערך וסוכם ע"י שניר נהרי - מבוסס על סיכומיה של לינוי דוארי

12.1 Thread Scheduling

בהרצאה זו נרצה להכיר את ה-API של ספריות Java שונות שהשפה חשפה לנו. נרצה להשתמש בכלים יותר מופשטים (לעומת sleep(), wait(), join() וכו'). החל מגרסה 5.1 של Java יש ספריה שימושית הנקראת java.util.concurrent. בספריה זו ישנן כלים שמאפשרים כתיבת קוד ברמת nupayu, גבוהה יותר, ויקלו על כתיבת קוד שרץ במקביל.

לשם המחשת הרעיון, נתבונן בדוגמא הבאה. אנו רוצים ליצור תוכנית שמדפיסה בצורה עקבית את המילה 'Ping' ואחרי חצי שנייה את המילה 'Pong' .

ברמה הבסיסית ביותר, נכתוב שתי מחלקות Ping, Pong כאשר כל אחת מממשת את Runnable ודורסת את המתודה run כדי להדפיס את המידע המתאים. לאחר מכן אנו מייצרים thread לכל תוכנית, ומריצים אחד אחרי השני. כך זה יראה בקוד -

```
public class Ping implements Runnable{
 public void run(){while(true)System.out.println("ping");}
}
public class Pong implements Runnable{
 public void run(){while(true)System.out.println("pong");}
}
 public static void main(String[] args) {
  Ping ping=new Ping();
  Pong pong=new Pong();
  Thread t=new Thread(ping,"thread 1");
  Thread t1=new Thread(pong,"thread 2");
  t.start();
  t1.start();
 }
```

האם הקוד יעבוד כפי שאנו רוצים? בוודאי שלא! הקוד ירוץ כך שבכל פעם ש- thread יקבל משבצת זמן (time slice) הוא ידפיס ברצף את המידע שלו. כאשר ה-thread האחר יקבל משבצת זמן, הוא ידפיס את המידע שלו. כלומר אנו עובדים ללא סנכרון כלל.

ישנה דרך טובה יותר לממש זאת, בעזרת sleep. בכל מתודה דורסת run, נבצע sleep של שנייה וב-main בין ה-threads נבצע sleep של חצי שנייה. הקוד יראה כך -

```
public class Ping implements Runnable{
  public void run(){
   while(true){
    System.out.println("ping");
    try {Thread.sleep(1000);}
    catch (InterruptedException e) {}
   }
  }
}
// pong is the same…
```

```
public static void main(String[] args) throws InterruptedException {
  Ping ping=new Ping();
  Pong pong=new Pong();
  Thread t=new Thread(ping,"thread 1");
  Thread t1=new Thread(pong,"thread 2");
  t.start();
  Thread.sleep(500); // the main sleeps 0.5 sec
  t1.start();
}
```

לכאורה הקוד הזה אכן עונה על הנדרש - הרצה של ping , לאחר מכן pong כאשר יש הפרש של חצי שנייה ביניהם. אך זה לא מדויק, נזכיר שכאשר thread חוזר מ-sleep הוא לא חוזר לריצה מיד, אלא נכנס ל- ready queue ורק לאחר מכן נבחר לריצה וממשיך את ריצתו. כלומר גם אם התוכנית תחזור לרוץ מיד, יהיה עיכוב קל של מיקרו שניות שיצטברו וכל שהתוכנית תרוץ יותר, הסטייה תגדל והתוכנית תצא מהסנכרון הרצוי.

אם כן, אנו רוצים כלי שיאפשר לנו לתזמן את המשימות כך שלא נצטרך לעשות זאת ידנית ולדאוג לסטיות הללו. ב-Java קיים כלי כזה (נמצא במחלקה Java.util) והוא נקרא Timer.

על מנת להשתמש במנגנון זה, נצטרך לרשת את המחלקה TimerTask ולדרוס את מתודה run (שם תוגדר המשימה). בניגוד למצב בו מימשנו runnable בו היינו צריכים לממש את הלולאה שתדפיס את המידע, בעזרת timer כל שעלינו לעשות הוא לממש את מה שאנו רוצים שירוץ בתוך הלולאה וה-timer כבר ידאג לבצע זאת שוב ושוב. נראה את קטע הקוד -

```java
import java.util.Timer;
import java.util.TimerTask;
public class ThreadTest {
 private static class Ping extends TimerTask{
  public void run(){System.out.println("ping");}
 }
 private static class Pong extends TimerTask{
  public void run(){System.out.println("pong");}
 }
 public static void main(String[] args){
  Ping ping=new Ping();
  Pong pong=new Pong();
  Timer t=new Timer();
  t.scheduleAtFixedRate(ping, 0, 1000);
  t.scheduleAtFixedRate(pong, 500, 1000);
 }
}
```

Canceling tasks:

```java
int i;
while((i=System.in.read())!=13);
ping.cancel(); // canceled task
pong.cancel(); // t continues…
t.cancel(); // t is cancled
```

אם כן ב-main אנו יוצרים את המחלקות Ping, Pong , וכן ניצור מופע של Timer. נגדיר ל-timer לתזמן את Ping בקצב קבוע, החל מעכשיו (לכן הפרמטר השני הוא 0), ולקרוא למתודה run בכל שנייה (לכן הפרמטר השלישי הוא 1,000) . מאחורי הקלעים, ייווצר thread שיריץ את המתודה run של Ping כל שנייה. באופן דומה נגדיר את Pong רק שהפעם נגדיר שהפרמטר השני יהיה חצי שנייה, כלומר החל מעוד חצי שנייה הרץ את המתודה run של Pong. מכיוון שזהו FixedRate, ה- timer אחראי לפצות על הסטיות, במידה והוא מזהה אותן. אם נרצה לעצור את הטיימר, ניתן להכניס את שורות הקוד מימין - ביטול של כל משימה ולאחר מכן ביטול של הtimer כולו. בצורה זו נוכל לבטל רק חלק מהמשימות ולא את כולן. כמו כן, כל עוד לא ביטלנו את הtimer , הוא ימשיך לרוץ ולהריץ משימות חדשות שיוכנסו לו בהמשך.

12.2 Replacing Synchronized (Atomic Variables)

בחלק זה נרצה להחליף את מנגנון ה-synchronized בהדרגה (מנגנון איטי כזכור). נזכיר שהמנגנון נועל מנעול על האובייקט עליו הוא מוגדר וכך מבטיח שלא יתבצעו פעולות "במקביל" על האובייקט. אם thread מסוים מפעיל מתודה שהיא synchronized, אז אף thread אחר לא יוכל להפעיל אף מתודה שהיא synchronized באותו האובייקט. אם thread אחר כן ינסה להפעיל מתודה שהיא synchronized , הוא יעבור למצב של blocked עד שה-thread הראשון יסיים את פעולתו וישחרר את המנעול של ה-synchronized .

נרצה להשתמש במנעולים מתקדמים, מנעולים שלא יובילו threads למצב של blocked, שעשוי לגרום לבעיות (דוגמת deadlock) .

נמחיש את הרעיון בעזרת הדוגמא הבאה. נתנה המחלקה Count שמכילה משתנה int בשם count . בנוסף, מוגדרת המתודה update אשר מוגדרת כ- synchronized, ומגדילה את ערך ה-count ב-1. אנו כבר יודעים שפעולת ++ (אשר על פניו נראית כפעולה אחת), מתורגמת ל-3 פקודות ברמת המערכת (שליפה של ערך והכנסתו לזיכרון, הגדלת הערך בזיכרון, כתיבה מחדש במקום המתאים). כעת, אם threads שונים ינסו להגדיל את ערכו של count, הערך הסופי עלול להיות לא צפוי בעקבות דריסות הדדיות של הפעולות. לכן עלינו להגדיר את המתודה הזו כ-synchronized . כאשר נגדיר את המתודה כ-synchronized , כל מי שנכנס למתודה הזו חייב לסיים אותה מתחילתה ועד סופה.

כעת, נגדיר את המחלקה CountUpdater שמממשת את המחלקה Runnable. המחלקה מכילה משתנה מסוג Count c. במתודה run מתבצעת לולאה למשך 100 מיליון פעמים המעדכנת את המשתנה c בעזרת c.update. ב-main נייצר אובייקט מסוג Count ונקבע את ערכו ל- 0. נייצר 2 threads שיקבלו כ-runnable את אותו אובייקט CountUpdater. קטע הקוד יראה כך -

```java
public class Count {
 private int count;
 public void setCount(int x){count=x;}
 public int getCount(){return count;}
 public synchronized void update(){count++;}
}
```

```java
public class CountUpdater implements Runnable{
 Count c;
 public CountUpdater(Count c){this.c=c;}
 public void run(){
  for(int i=0;i<100000000; c.update(),i++);
 }
}
```

```java
public static void main(String[] args) {
  Count c=new Count();
  c.setCount(0);
  CountUpdater ca=new CountUpdater(c);
  Thread t=new Thread(ca);
  Thread t1=new Thread(ca);
  long time=System.currentTimeMillis();
  t.start();
  t1.start();
  t.join();
  t1.join();
  System.out.println(c.getCount());
  long duration=(System.currentTimeMillis()-time)/1000;
  System.out.println(duration);
 }
```

46 seconds!!!

נדפיס את התוצאה לאחר 2 ההרצאות ואת הזמן שלקח להריץ. נקבל שהתוצאה היא אכן כצפוי, 200 מיליון וזמן ההרצה (נכון ללפני 10 שנים שהמרצה הריץ את הקוד) לקח כ-46 שניות. למה לוקח לתוכנית כל כך הרבה זמן לרוץ? שני ה- threads מתחרים על המתודה שמוגדרת כ-synchronized , במשך 200 מיליון ניסיונות, ובכל ניסיון רק אחד מהם יצליח להיכנס לקטע הקוד ולהגדיל את ערך ה-count. תחרות זו גוזלת זמן רב. מנגד, אם נוותר על מנגנון ה-synchronized , התכנית תסתיים מהר יותר אך התוצאה תהיה לא צפויה. אם ננסה להעביר את מנגנון ה-synchronized מהמתודה update() למתודה run(), אזי 2 ה- threads הללו לא ירוצו במקביל – ה- thread הראשון יריץ את run(), וה- thread השני היה נכנס למצב של blocked כיוון ששניהם מריצים את אותו משתנה. ה- thread הראשון היה מבצע את העדכון של c 100 מיליון פעם מהר מאוד, ואח"כ ה- thread השני היה מבצע זאת. במקרה זה התכנית הייתה רצה מהר יותר והתוצאה הייתה כצפוי, אך לא הייתה מקביליות.

נפתור את בעיה בעזרת שימוש בספרייה Java.concurrent.atomic . נגדיר משתנה מסוג AtomicInteger (ישנם טיפוסים אטומיים נוספים) . כל פעולה על משתנה זה היא אטומית, דהיינו לא ניתנת לפיצול ולכן הוא thread safe. המשתנים האטומיים הללו משתמשים במנעולים מתקדמים ומהירים יותר וכן במנגנונים שונים פנימיים על מנת לבצע את הנעילה בצורה מהירה. נתבונן בקטע הקוד הבא -

```java
import java.util.concurrent.atomic.AtomicInteger;
public class Count {
 AtomicInteger count = new AtomicInteger(0);
 public  void setCount(int x){count.set(x);}
 public int getCount(){return count.get();}
 public void update(){
  count.incrementAndGet();// ++count
 }
}
```

```java
public class CountUpdater implements Runnable{
 Count c;
 public CountUpdater(Count c){this.c=c;}
 public void run(){
  for(int i=0;i<100000000; c.update(),i++);
 }
}
```

```java
public static void main(String[] args) {
  Count c=new Count();
  c.setCount(0);
  CountUpdater ca=new CountUpdater(c);
  Thread t=new Thread(ca);
  Thread t1=new Thread(ca);
  long time=System.currentTimeMillis();
  t.start();
  t1.start();
  t.join();
  t1.join();
  System.out.println(c.getCount());
  long duration=(System.currentTimeMillis()-time)/1000;
  System.out.println(duration);
 }
```

6 seconds!!!

אנו רואים שביצוע אותו מספר פעולות, על משתנה אטומי וללא מתודה שמוגדרת כ- synchronized לקח 6 שניות (על אותו מחשב מלפני 10 שנים)! שיפור משמעותי מאד ביחס לדוגמא הקודמת. בהמשך נראה מנגנונים נוספים שפותרים בעיות אחרות של synchronized – כמו הכניסה ל-blocked.

12.3 DeadLock Example (tryLock)

ראינו לעיל שכדאי להמעיט את השימוש במנגנון ה-synchronized וזאת מהסיבות הבאות-

- מנגנון איטי
- אם thread מסוים מנסה להפעיל מתודה synchronized באובייקט שהוא כבר נעול, הוא עובר למצב blocked. מצד אחד, זה עוזר כיוון שבעזרת ה-blokcked מתאפשרת מניעה הדדית. מצד שני, תכנות לא נכון עשוי לגרום ל- deadlock .

Deadlock מתרחש כאשר יש מעגליות מסוימת, לדוגמא כל thread או תהליך תפס משאב מסוים וכעת משאב זה לא ישוחרר עד שיסתיים החישוב שלו. בתוך אותו חישוב ה- thread או התהליך ממתינים למשאב נוסף על מנת לסיים את החישוב. אם המשאב הנוסף תפוס, התהליך יחכה עד שהוא ישתחרר ורק לאחר מכן יוכל להמשיך את ריצתו וחישובו.

אם ישנו רצף של תהליכים שממתינים זה לזה (לא במעגל), זהו מצב תקין, כיוון שכאשר התהליך האחרון שנמצא ברצף ישחרר את המשאב ,בהדרגה התהליכים יוכלו להתקדם וכל אחד בתורו יסיים את החישוב וישחרר את המשאבים שברשותו. אולם אם יש רצף תהליכים בו התהליך האחרון ממתין למשאב המוחזק ע"י התהליך הראשון , בסבירות גבוהה ייווצר deadlokc , והתכנית תתקע. נצרף דוגמת קוד ל- deadlock פשוט יחסית -

```
Object R=new Object();// readers lock
Object W=new Object();// writers lock
```

```
new Thread(new Runnable() {
   @Override
   public void run() {

       synchronized (W) {
          // do the writing...
          synchronized (R) {
             // do some reading...
          }
          // do more writing...
       }
   }
}).start();
```

```
new Thread(new Runnable() {
   @Override
   public void run() {

       synchronized (R) {
          // do the reading...
          synchronized (W) {
             // do some writing...
          }
          // do more reading...
       }
   }
}).start();
```

בדוגמא אנו רוצים ליצור כמה threads שכותבים וקוראים. כדי ליצור מניעה הדדית ביניהם ,נשתמש באובייקטים R, W שיוגנו ב-synchronized בהתאמה. נניח שה- thread הראשון מתחיל בניסיון לקחת את המשאב W, במידה והצליח מבצע קוד מסוים ומנסה לאחר מכן לקחת גם את המשאב R. במקביל ה- thread הנוסף מבצע את אותו תהליך אך הפוך - ראשית מתחיל בניסיון לקחת את המשאב R, מבצע קוד מסוים, ולאחר מכן מנסה לקחת את המשאב W. בקוד זה, בהסתברות גבוהה יחסית ניכנס ל- deadlock. בזמן שה- thread הראשון נעל את W, ה- thread השני הספיק לנעול את R. כשה- thread הראשון רוצה לקחת את R, R תפוס ונעול לטובת הthread - השני ולכן ה- thread הראשון נכנס למצב של blocked. לכאורה, אם ה- thread השני ישחרר את ה- synchronized של R, ה- thread הראשון יקבל את R ויוכל להתקדם בקוד ולסיים אותו. אולם ה- thread השני לא יכול לשחרר את R, כיוון שהוא במצב blocked ב-synchronized על W, שתפוס ע"י ה- thread הראשון. באופן זה, שני ה - thread במצב של blocked כאשר הם מחכים למשאבים אחד של השני, זהו deadlock.

ה-synchronized הוא זה שהכניס אותנו למצב blocked (אם המנעול היה נעול כאשר ניסינו לקבל את המשאב). כעת נתבונן במנעול חדש, המאפשר לנו להימנע ממצב שכזה. מנעול זה נקרא בשם ReentreantLock (נמצא ב- java.util.concurent.locks). בעזרת מנעול זה אנו מנסים לנעול (try lock) את כל המשאבים הדרושים לנו לצורך תהליך מסוים (הכל או כלום). אם הצלחנו לנעול נקבל TRUE ונוכל להמשיך. אם אחד או יותר מהמשאבים תפוסים, נקבל FALSE ונוכל להמשיך בקוד ללא המשאבים (לא ניכנס למצב של blocked).

המתודה בה נשתמש על מנת לנסות לנעול את המשאבים נקראת tryLock. מתודה זו היא thread-safe, גם כאשר מספר threads מנסים להפעיל אותה בו זמנית, רק אחד יצליח לנעול ולקבל TRUE בחזרה. כל שאר ה- threads שלא הצליחו לנעול וקבלו FALSE לא יכנסו למצב של blocked ולכן בינתיים יוכלו לבצע קוד אחר שלא מצריך את המשאבים. על מנת להתחיל נגדיר משתנים מסוג ReentrantLock -

```
ReentrantLock W=new ReentrantLock();
ReentrantLock R=new ReentrantLock();
```

ולאחר מכן נבצע את קטע הקוד הבא -

```
public void run() {
   boolean w=W.tryLock();
   boolean r=R.tryLock();
   try{
      if(w && r){
         // do the writing...
         // do some reading...
         // do more writing...
      } else{
         // try again later...
      }
   }finally{
       if(w) W.unlock();
       if(r) R.unlock();
   }
}
```

כלומר רק אם שתי הנעילות הצליחו, ניכנס אל קטע הקוד, אחרת נבצע דבר מה. לבסוף נשחר את כל המנעולים שנעלנו.

Thread Safe Containers 12.4

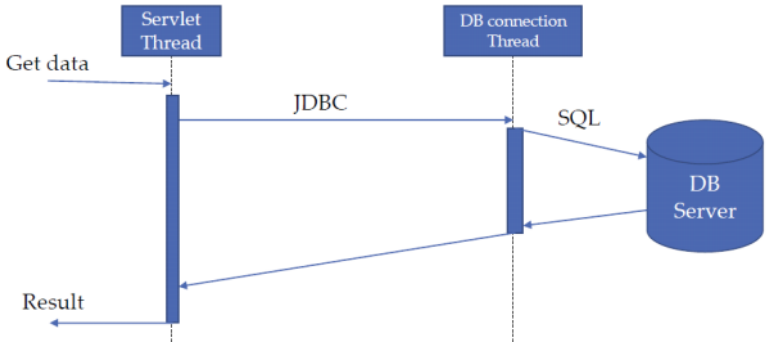
רוב מבני נתונים שהכרנו עד כה ב-java , דוגמת ArrayList, LinkedList, HashMap וכד' הם אינם thread-safe , הפעולות אינם אטומיות. לדוגמא , הוספת key,value ל-HashMap משני threads שונים תגרום לדריסה של מידע. נרצה ללמוד כיצד להשתמש במבני נתונים Thread-Safe.

בגרסאות המוקדמות של Java , אם היינו רוצים לדוגמא ליצור HashMap שהוא Thread-safe, היה ניתן להשתמש

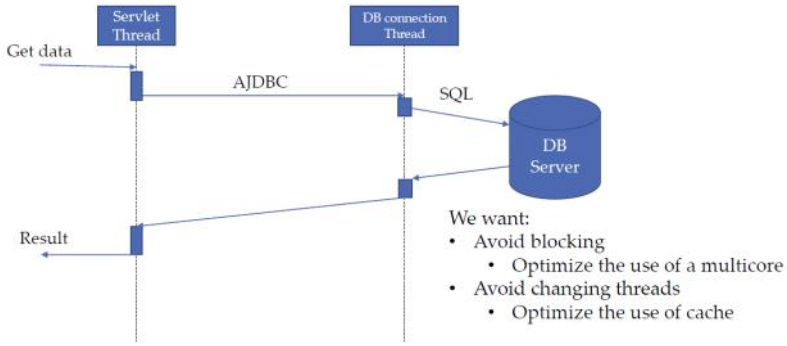
בתבנית העיצוב קשטן (Decorator) ולעטוף את ה-HashMap ב-Map אחר הנקרא synchronizedMap ותרומתו היחידה היא שהוא עוטף את המתודות במנגנון ה- synchronized.
לדוגמא, במימוש של המתודה put, ה-synchronizedMap יעטוף את המתודה הזו ב -synchronized ולכן רק thread אחד יוכל לגשת למתודה בכל פעם.במימוש זה, משלמים את "המחיר" של synchronized רק אם באמת משתמשים בו.
החל מגרסה 1.6 בספרייה java.util.concurrent יש מבני נתונים שהם Thread-safe . מבנים אלו משתמשים במנגנוני הגנה מתקדמים יותר מאשר ה-synchronized. חשוב לזכור שגם אם משתמשים במבנ"ת שמוגדר thread-safe, רצף הפעולות שהוא קורא להן לא בהכרח thread-safe. לדוגמא - אם קוראים מה-main 10 פעמים למתודה put לאובייקט מסוג concurrentHashMap. כל פעולה בודדת של המתודה put היא thread-safe, אך רצף ההפעלות הוא לא thread-safe, ומשבצת הזמן של התהליך על המעבד יכולה להסתיים לפני שסיימנו לבצע את רצף ההכנסות. אם נרצה שגם רצף ההכנסות יהיה thread-safe, נקיף את קטע הקוד הזה ב-synchronized. כלומר אנו משתמשים ב- synchronized רק כאשר אין ברירה אחרת.

Fork Join Pool 12.5

החל מגרסה 8 של Java הצטרפו מס' גדול של ספריות קוד, וחלקן קשורות לתכנות מקבילי. נרצה לגעת בשני אלמנטים עיקריים - Fork join pool , Completable Future (בחלק הבא). נתבונן בדוגמא הבאה -



תכנית הרצה על במקביל ב-2 threads. ה- thread הראשון מבצע בקשה לקבלת מידע מתוך בסיס הנתונים. הבקשה נשלחת אל ה-thread של בסיס הנתונים, ולאחר כמה זמן, חוזרת התשובה. למרות שהקוד הוא א-סינכרוני, כלומר רץ בשני threads נפרדים, אנו נמצאים במצב של blocking שכן ה-thread הראשון לא יכול להמשיך את ריצתו כל עוד הוא לא קיבל את המידע בחזרה מבסיס הנתונים. אנו נרצה להגיע למצב א-סינכרוני מלא, בו ניתן לשלוח בקשה לקבלת מידע, בינתיים להריץ קוד אחר שיבצע חישובים אחרים, וכאשר נקבל את התשובה נפעל בהתאם. זה יראה כך -

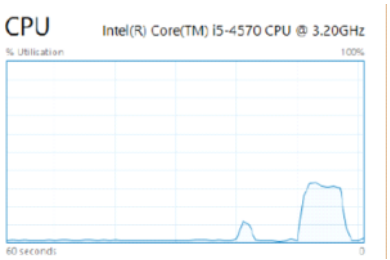


החל מגרסא 7 של Java התווסף thread-pool מסוג fork-join pool . ל-fork - join pool יש את אותן המתודות כמו שראינו בסוגים האחרים של thread-pool (כלומר execute ל- runnable ו-submit ל- callable וכד'), אך הוא בעל מספר מתודות נוספות אשר מסייעות לו לבצע משימות במקביל ואידיאליות למשימות רקורסיביות.
היתרון העיקרי של fork-join pool – הוא work stealing. מה הכוונה ב-work stealing? נניח שמשימה רצה על ליבה מסוימת במעבד. יש זיקה מסוימת בין התהליך שאחראי על המשימה הזו לבין הליבה שעליה הוא רץ, שכן חלק מהמידע שרלוונטי למשימה נמצא בזיכרון המטמון של ליבה זו. נרצה שמשבצת הזמן הבאה של אותו תהליך, תוקצה על אותו מעבד (אותה ליבה). בצורה זו נוכל לנצל את זיכרון המטמון ששמור בליבה וכך נצטרך לבצע פחות עדכונים והקוד ירוץ מהר יותר.
עם זאת, כאשר יש משימות רבות שמצטברות בתור לליבה ספציפית, זה מצב לא אידאלי וגורם לניצולת נמוכה של המעבדים. למרות שהמשימות האלו הורצו בעבר על הליבה הזו, לא נרצה שתהיה ליבה אחת עמוסה במשימות שמחכות בתור וליבה אחרת פנויה לגמרי. הרעיון ב- work stealing היא "לגנוב" משימות מסוף התור של הליבה העמוסה, ולהעבירן לתור של הליבה הפנויה. נעביר דווקא משימות מסוף התור, כיוון שסביר להניח שאם יש משימות רבות שיורצו לפני משימה זו הנמצאת בסוף התור, המשימות הקודמות ימחקו את רוב המידע שנשמר בזיכרון המטמון ורלוונטי למשימה שמחכה בסוף התור.

נמחיש את הרעיון בעזרת דוגמא פשוטה יחסית – סדרת פיבונאצ'י. נתבונן בקוד הבא שמחשב את הסדרה עד 45 -

```
public class Fib {

 int num;
 public Fib(int num) {
  this.num=num;
 }

 public int compute(){
  if(num<=1)
   return num;
  Fib fib1= new Fib(num-1);
  Fib fib2= new Fib(num-2);
  return fib2.compute()+fib1.compute();
 }
 public static void main(String[] args) {
  System.out.println(new Fib(45).compute());
 }
}
```

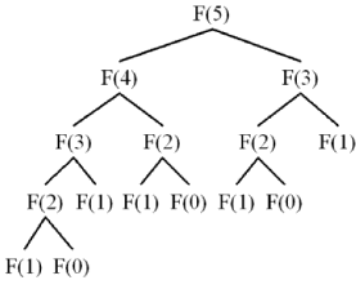
נרצף צילום מסך של הביצועים של הקוד ביחס לניצול המעבד והליבות השונות -



אנו מקבלים ניצולת נמוכה יחסית של המעבד. נרצה לייעל את המשימה ע"י טכניקה שרלוונטית מאד במשימות רקורסיביות - פיצול לעצים בלתי תלויים כך שכל עץ מחושב ב-thread נפרד (במקביל).

Dynamic Programming

נשתמש בתכנות דינמי. בדוגמא שלנו, ניתן להשתמש בתוצאות שכבר חושבו בעבר. נתבונן בעץ הרקורסיה של החישוב עד 5 -



ישנם ערכים רבים שחוזרים על עצמם ולכן נוכל לפתור זאת ע"י שמירת הערכים במבנה נתונים ורק אם לא קיים ערך לחשב. נתבונן בקוד הבא -

```
public class Fib_DP { // without concurrency
                      // but with dynamic programming
 static HashMap<Integer,Integer> fibs=new HashMap<>();

 int num;
 public Fib_DP(int num) { this.num=num;}

 public int compute(){ // a recursive task
  if(num<=1)
   return num;
  if(fibs.get(num)!=null)
   return fibs.get(num);
  Fib_DP fib1= new Fib_DP(num-1);
  Fib_DP fib2= new Fib_DP(num-2);
  int result=fib2.compute()+fib1.compute();
  fibs.put(num,result);
  return result;
 }
 public static void main(String[] args) {
  System.out.println(new Fib_DP(2048).compute());
 }
}
```

קוד זה יוביל אותנו לשיפור בביצועים אך מכיוון שזוהי לא מטרת חלק זה לא נתמקד בזאת. נעבור להתבונן בשיפור הביצועים כאשר מחלקים את החישובים הבלתי תלויים ל- threads שונים.

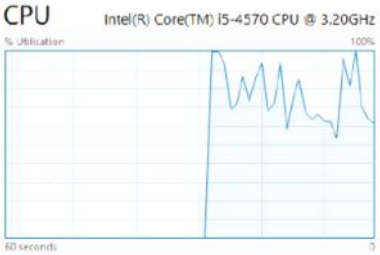
Cached Thread Pool

נשתמש ב-thread pool מסוג cached thread pool (יצירה של threads עד שמערכת ההפעלה עוצרת את

התוכנית). נתבונן בקטע הקוד הבא -

```
public class Fib_TP implements Callable<Integer>{

 static ExecutorService es=Executors.newCachedThreadPool();

 int num;
 public Fib_TP(int num) {this.num=num;}

 @Override
 public Integer call() throws Exception {
  if(num<=1)
   return num;
  Future<Integer> fib1 = es.submit(new Fib_TP(num-1));
  Future<Integer> fib2 = es.submit(new Fib_TP(num-2));
  return fib2.get()+fib1.get();
 }

 public static void main(String[] args) throws InterruptedExcep
  Future<Integer> f=es.submit(new Fib_TP(45));
  System.out.println(f.get());
 }
}
```

אנו ממשים את המחלקה callable ויוצרים thread pool מסוג cached . מכניסים אל התור את תת י העץ fib1,fib2 ומחכים לתוצאות שיחזרו. בהרצה זו נקבל את הביצועים הבאים -



אמנם ניצולת המעבד עלתה בעקבות שימוש ב-threas רבים ככל שהניתן אך זמן הריצה התארך מאד (למעשה התוכנית לא סיימה את הריצה אלא נעצרה ידנית בגלל זמן ריצה ארוך).

הסיבה לזמן הריצה הארוך היא בשל השימוש ב-cached thread pool. התקורה של יצירת threads חדשים היא שגורמת לפעולות I/O שמורידות את ביצועי המעבד . כלומר cached thread pool הוא לא תמיד הפתרון האידיאלי. בדוגמא שלנו היה עדיף להגביל את כמות ה- threads או לא להשתמש בהם כלל, וכך התוצאה הייתה מתקבלת מהר יותר.

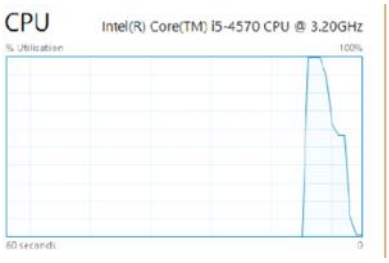
כמו כן נשים לב שיש כאן בזבוז משווע שכן ה- thread שמריץ את המתודה call, תקוע בהמתנה לחישוב שני הערכים (בשורה של return) . היה עדיף לקרוא ל-fib2.call() שהיא קריאה סינכרונית, ואז לתוצאה להוסיף את fib1.get(). כך לפחות thread אחד היה ממשיך לחשב, והיינו חוסכים את כמות ה- threads שצריך ליצור.

Fork Join Pool

נעבור אל הפתרון בעזרת fork-join pool . כאמור ה-fork-join pool אידיאלי למשימות רקורסיביות. לקריאה הרקורסיבית נקרא fork, ולהמתנה לתוצאה קוראים join. נתבונן בקטע הקוד הבא -

```
public class Fib_FJ extends RecursiveTask<Integer>{
 // with fork-join pool
 int num;
 public Fib_FJ(int num) { this.num=num; }

 @Override
 public Integer compute(){ // a recursive task
  if(num<=1)
   return num;
  Fib_FJ fib1= new Fib_FJ(num-1);
  fib1.fork();
  Fib_FJ fib2= new Fib_FJ(num-2);
  return fib2.compute()+fib1.join();
 }

 public static void main(String[] args) {
  Fib_FJ fib=new Fib_FJ(45);
  ForkJoinPool pool = new ForkJoinPool();
  System.out.println(pool.invoke(fib));
 }
}
```

הקריאה ל-fork במתודה compute לוקחת את המשימה compute של Fib(num-1) ומכניסה אותה לתוך התור, ולכן היא מיד חוזרת. בינתיים, ניצור את fib2 וננצל את ה- thread הנוכחי לבצע compute. ה- thread הנוכחי יחשב את החישוב הסינכרוני של fib2, ויחזיר את התוצאה מתקבלת עם fib1.join. הביצועים המתקבלים הם -



ניצולת גבוהה של המעבד וכן זמן ריצה קצר יחסית.
לסיכום התהליך, לקחנו את fib1, הרצנו את המשימה שלו ברקע ע"י הזרקה ל-fork join pool , חישבנו את ה-compute של fib2 בצורה סינכרונית. כשסיימנו לחשב אותו, המתנו לתוצאה של חישוב fib1. כך ביצענו את המשימה בצורה מיטיבית ושימוש טוב יותר ב- threads .

נעיר שבדוגמא הספציפית הזו היה עדיף להשתמש בתכנות דינמי, אך לשם הלמידה הראינו כיצד לממש זאת בעזרת fork join pool .
ל- fork join pool יש את המתודות (...)submit , (...)execute. ניזכר שלמדנו על אובייקט פעיל (הרצאה 11) בו ניתן להשתמש בתור, אזי במקום להשתמש ב-blocking queue ולהזריק לו runnable, נוכל להשתמש ב fork. join pool-

Completable Future 12.6

לפני שנצלול לעומק של חלק זה נזכיר את האובייקט Future. אובייקט זה ממש כ- container למשתנה גנרי V אשר יכיל בעתיד את ערך ההחזרה V , שיחזור מ- thread שמתבצע ברקע. המתודה get במידה ונקראה, תמתין עד שהפלט החישובי מוכן. ההמתנה למודה get גורמת לבזבוז משאבים.
כעת נכיר טכניקה חדשה בעזרת אובייקט הנקרא completable future. מטרת אובייקט זה היא להגדיר משימות שיתבצעו ברגע שערך ההחזרה V יגיע. כך ה- thread שהגדיר את המשימות הללו מסיים את עבודתו ומשחרר את המשאבים שלו. כאשר ה-thread ירוץ וסיים את החישוב הנדרש, וערך ההחזרה יגיע, כבר הוגדר לו מראש מה הפעולות שעליו לבצע כעת.

Callable and Future

נמחיש את הרעיון בעזרת דוגמא. נגדיר מתודה בשם deepThought (רפרנס למדריך לטרמפיסט בגלקסיה) שלאחר הרבה מאד זמן תחזיר את המחרוזת "42". בנוסף, נגדיר thread pool רגיל ע"י ExecutorService. נזריק ל- executor אובייקט מסוג callable שמחזיר מחרוזת, שאותה הוא מקבל מהפעלת המתודה deepThought .
נשים לב שהמתודה deepThought לא תתחיל לרוץ כעת, אלא בסה"כ הכנסנו אותה אל התור של ה- ExecutorService. מכיוון שהזרקנו callable המתודה submit חזרה מיד.
לפני שמתקבל ערך ההחזרה נמשיך לבציע חישובים במקביל, עד לשלב שבו אנו זקוקים לערך ההחזרה. נקרא למתודה get שתאלץ אותנו להמתין לערך החזרה. כך זה יראה בקוד -

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
ExecutorService executor=Executors.newCachedThreadPool();

Future<String> f = executor.submit(new Callable<String>() {
        @Override
        public String call() throws Exception {
                return deepThought();
        }
});
```

```
//...
System.out.println(f.get()); // blocks until an answer is given
```

Lambda Expression

נקצר את הקוד ובמקום מופע אנונימי של callable, נשתמש בביטוי למבדה. הפונקציה call היא ביטוי למבדה, פונק' חסרת שם שלא קיבלה פרמטרים ומחזירה את deepThought . זהו איננו שיפור משמעותי אלא קיצור בתחביר בלבד -

```
ExecutorService executor=Executors.newCachedThreadPool();

Future<String> f = executor.submit( ()-> {
        return deepThought();
});
```

Completable Future

הזכרנו לעיל את האובייקט completable future . אובייקט זה מכיל בתוכו פונקציה שנקראת supplyAsync.
פונקציה זו מקבלת כפרמטר ראשון את ה- callable שיחזיר את deepThought . הפרמטר השני הוא

ה-ExecutorService של ה-thread pool. כך זה יראה -

```
ExecutorService executor=Executors.newCachedThreadPool();

// an asynchronous call
CompletableFuture.supplyAsync( ()->{
        return deepThought();
},executor);
```

ניתן לקצר את הקוד ע"י כך שלא נכניס את הפרמטר השני. כאשר לא מספקים את ה-ExecutorService, המתודה תשתמש ב- ExecutorService ברירת המחדל שהוא join pool fork .

מה נקבל בחזרה מהמתודה supplyAsync ?
המתודה תחזיר אובייקט מסוג Completable Future. בדומה לאובייקט future, אובייקט זה הולך להכיל את ערך ההחזרה שלו (בדוגמא שלנו, מחרוזת) לאחר שה-thread שאחראי על המתודה שהוזרקה יצא מהתור יבצע את הנדרש ויחזיר את הפלט המבוקש. בינתיים נקבל את משתנה מסוג Completable Future .
במשתנה מסוג זה נוכל להפעיל מתודות שונות דוגמת fc.thenAccept. זוהי מתודה שמקבלת פונקציה כפרמטר. בדוגמא זו בהינתן מחרוזת, נדפיס אותה. ראינו בעבר שכל ביטוי למבדה הוא פונקציה לממשק. בדוגמא זו הפונקציה לממשק היא consumer – כיוון שבהינתן ערך כל שהוא, הפונקציה צורכת אותו ולא מחזירה ערך.
ה- thread שהריץ את הקוד יכול לשחרר את משאביו ולסיים, וכשה- thread שהריץ את deepThought יסיים, תחזור המחרוזת "42" והפעולות שהוגדרו מראש ע"י ה- Completable Future יופעלו . נתבונן בקוד -

```
CompletableFuture<String> fc = CompletableFuture.supplyAsync( ()->{
        return deepThought();
});

fc.thenAccept( (String answer)->{System.out.println("answer: "+answer);});
```

בצורה זו אנו ממשים תבנית שנקראת Reactive Pattern (מלשון ריאקציה, תגובה). כלומר בהינתן ערך/אובייקט כיצד להגיב, אלו פעולות לבצע. בנוסף אנו מאפשרים תכנות שוטף (Fluent Programming) ע"י שרשור של מתודות לאובייקטים החוזרים בכל קריאה. לדוגמא supplyAsync מחזירה Completable Future שניתן להפעיל עליו את המתודה thenAccept.

ניתן לשפר אף יותר את הקוד. מכיוון שהמתודה ()supplyAsync מחזירה אובייקט מסוג CompletableFuture, ניתן להפעיל עליו מתודות גם מבלי לתת לו שם. כך זה יראה -

```
CompletableFuture.supplyAsync( ()->{return deepThought();})
        .thenApply(answer->Integer.parseInt(answer))
        .thenApply(x->x*2)
        .thenAccept(answer->System.out.println("answer: "+answer));
```

נשים לב שבדוגמא שהבאנו כעת, בניגוד למתודה thenAccept, המתודה thenApply מקבלת כפרמטר פונקציה כך שבהינתן מחרוזת תשובה, היא תחזיר אותו כ-int. גם כאן זוהי פונקציה לממשק, הפעם מטיפוס function (כי היא מחזירה פלט). לאחר הפונקציה thenApply הראשונה יוחזר אובייקט מסוג CallableFuture<int>. הפונק' תשוב מיד אף על פי שהערך שלה יוחזר בשלב מתקדם יותר . לאחר מכן על ה-int שהוחזר נבצע מניפולציה (כפל ב-2) ונחזיר זאת. לבסוף בפונקציה thenAccept נזריק את הערך האחרון (לאחר הכפל) ונדפיס אותו.

ה-main שקרא לכל הקוד הזה יסיים וישחרר את משאביו טרם קבלת ערך החזרה כל שהוא. בינתיים, המתודה deepThought תיקח זמן מה. לאחר שהמתודה תסיים את חישוביה, סדר הפעולות שהוגדר ב- Completable Future יכנס לפעולה.

בדוגמא שראינו מתקיים עיקרון Inversion of control . המתודה deepThought מחזירה מחרוזת. על מחרוזת זו נוכל להגדיר מראש את הפעולות המבוקשות. כאשר ה- thread שיריץ את deepThought יסיים, הוא יריץ את הפעולות המוגדרות מראש עבור ערך ההחזרה הנ"ל, עד לקבלת הפלט הרצוי. כך נחסוך משאבים רבים.
לסיום נעיר שה-API של Completable Future מכיל מתודות רבות – זוהי רק דוגמא קטנה. בשקופית האחרונה יש לינקים לקריאה בהמשך הנושאים האלו.