תכנות מרובה ליבות / תרגיל בית #5

אריאל סטולרמן

הערות על השאלה התכנותית:

המימוש לא משתמש ב-Callable והחזקת מערך של Future בכל שלב ברקורסיה, אלא משתמש ב-Runnable ללא המתנה של שלבי הרקורסיה לקריאות שיצר. הסיבה לכך היא שאין צורך ביותר מפתרון אחד (לפי תשובת המתרגל בפורום), ולכן המימוש דואג שהקריאה הראשונה המגיעה לפתרון מעדכנת את המבנה כך שעוצר את כל המשימות שרצות באותו זמן (יתכן שבדרך חזרו פתרונות אחרים ודרסו את הקודם, אחד מהם ינצח).

ניתן לממש באופן דומה יותר לדוגמא אם היינו נדרשים להחזרת כל הפתרונות: המימוש יהיה דומה למימוש כעת, רק שייעשה שימוש ב-Callable. כל שלב יחכה לקבל רשימות של פתרונות מלמטה (כאשר פתרון בשלב הקצה מחזיר מעלה רשימה בגודל 1), יאחד את כולם לרשימה אחת ויעבירה מעלה. את ניהול הקריאות הרקורסיביות במקרה זה ניתן לעשות באמצעות מערך Future והמתנה לחזרת המשימות שפותח כל שלב.

שאלה 185:

נניח כי פעולת merge לוקחת $\Theta(n)$ פעולות, אז:

- העבודה T_1: ישנן שתי קריאות רקורסיביות על מחצית מגודל הקלט הראשוני ולכן מתקיים:

$$T_1(n) = \Theta(n) + 2 \cdot T_1\left(\frac{n}{2}\right) = \Theta(n) + 2\left(\Theta\left(\frac{n}{2}\right) + 2 \cdot T_1\left(\frac{n}{4}\right)\right) = \cdots \Rightarrow T_1(n) = \Theta(n \cdot \log n)$$

- אורך המסלול הקריטי T_∞: עץ המסלולים מאוזן ולכן ניקח את אחד המסלולים. אורכו הוא:

$$T_\infty(n) = \Theta(n) + T_1\left(\frac{n}{2}\right) = \Theta(n)$$

- מקבילות T_1/T_∞:

$$\frac{T_1(n)}{T_\infty(n)} = \Theta(n \cdot \log n)/\Theta(n) = \Theta(\log n)$$

שאלה 186:

האלגוריתם השני נחשב משופר במובן שזמן הריצה הכולל שלו על מעבד יחיד, כלומר לריצתו הסדרתית, קצר יותר מאשר האלגוריתם הראשון, הפשוט: עבור הפשוט זמן הריצה הוא $2048/1 + 1 = 2049$, ועבור המשופר זמן הריצה הוא $1024/1 + 8 = 1032$, כמעט פי שניים מהיר יותר.

כאשר מריצים את האלגוריתמים הנ"ל על מכונה עם 512 מעבדים, זמן הריצה של האלגוריתם המשופר לכאורה ארוך יותר מאשר הפשוט. עבור הפשוט זמן הריצה הוא $\frac{2048}{512} + 1 = 4 + 1 = 5$ ועבור ה"משופר" הוא $\frac{1024}{512} + 8 = 2 + 8 = 10$. נשים לב שככל שמספר המעבדים יגדל, כך ביצועי האלגוריתם ה"משופר" לכאורה יורעו ביחס לאלגוריתם הפשוט, וזאת כיוון שזמן הריצה שואף לאורך המסלול הקריטי, והוא ארוך יותר ב"משופר".

שאלה 188:

$$T_4 = 80, T_{64} = 10$$

$$T_p \geq \frac{T_1}{P}, \quad T_p \geq T_\infty, \quad T_p \leq \frac{T_1 - T_\infty}{P} + T_\infty$$

$$\begin{cases} 80 \leq \dfrac{T_1 - T_\infty}{4} + T_\infty \\ 10 \leq \dfrac{T_1 - T_\infty}{64} + T_\infty \end{cases} \Rightarrow \begin{cases} 320 \leq T_1 - T_\infty + 4T_\infty \\ 640 \leq T_1 - T_\infty + 64T_\infty \end{cases} \xRightarrow[\text{חיסור}]{} 320 \leq 60T_\infty \Rightarrow T_\infty \leq 5\frac{1}{3}$$

$$\Rightarrow 320 \leq T_1 + 3T_\infty \leq T_1 + 16 \Rightarrow T_1 \geq 304$$

$$\Rightarrow T_{10} \geq \frac{T_1}{10} \geq \frac{304}{10} = 30.4 \text{ seconds}$$

<u>שאלה 193</u>:

(1) אם משתנה *bottom* לא היה *volatile*, ניתן היה להגיע למצב בו מערך משימות נתון לא תקין. להלן תיאור ריצה המביאה את מערך המשימות למצב כזה:

- נניח כי נותרה משימה אחת במערך המשימות והאינדקסים במצב הבא: bottom = 3, top = 2, כלומר נותרה משימה אחת r במקום tasks[2].
- מתחילה ריצת popBottom() המורידה את bottom ל-2, אך רק בקרב בעל מערך ה-tasks כיוון ש-bottom אינו volatile.
- מתחילה ריצת popTop() שרואה את ערכי top הנוכחי ו-bottom הלא מועדכן, ועל כן עוברת את תנאי הבדיקה bottom <= top ומצליחה לקחת את המשימה r תוך שינוי ע"י compareAndSet של top להיות 3 (ושינוי ה-stamp שאינו משנה לתיאור). הריצה מסתיימת עם החזרת r, כלומר הגנב הצליח לקחת את המשימה.
- ממשיכה ריצת ה-popBottom(), כאשר bottom = 2 כיוון ששונה מקומית ולכן השינוי נראה, ו-oldTop = top = 3 כיוון שנלקח לאחר סיום פעולת הגנב, שעדכן את top בצורה אטומית הנראית לכל החוטים. בשל נתונים אלו, לא מתקיימים גם bottom > oldTop וגם bottom == oldTop, ולכן נעשה top.set(newTop,newStamp), כאשר newTop = 0.
- כעת מצב האובייקט הוא ש-bottom = 2, top = 0 – ונראה כאילו יש משימות חדשות למרות שאין – ולכן האובייקט אינו קונסיסטנטי והמצב אינו תקין.

(2)

נרצה לאפס את bottom ל-0 מוקדם ככל האפשר בכדי לחסוך בפעולת popTop ניסיונות compareAndSet שייכשלו (ע"י האיפוס יוחזר מיד null ולא ימשיך לניסיון ה-compareAndSet בשורה 8). ככל שהאיפוס יהיה מוקדם יותר, כך הסיכוי להקטין את מספר ה-CAS גדל. להלן תיאור סיטואציה בה מצב זה מתרחש, כאשר במצב ההתחלתי ישנם שני איברים בתור:

- גנב א' קורא ל-popTop ונעצר לפני בדיקת התנאי bottom <= oldTop.
- גנב ב' קורא ל-popTop ושולף את האיבר (יגרום להמשך גנב א' להכשל אם יגיע לבצע CAS).
- בעל התור קורא ל-popBottom והתור מתרוקן. כיוון שהוא מוציא את האיבר האחרון, הוא מעדכן את bottom ל-0, וכיוון ש-bottom הוא volatile, אם גנב א' יתעורר מאוחר מספיק הוא יראה את שינוי הערך ל-0.

כאשר גנב א' מתעורר ובודק את התנאי bottom <= oldTop, אם נקדים את איפוס bottom ב-popBottom ככל שניתן, הסיכוי שיראה את שינוי ערך bottom ל-0 גדל, ולכן הסיכוי שיחזיר null במקום להמשיך ל-CAS שבודאות ייכשל גם כן גדל. כיוון ש-CAS היא פעולה יקרה, נרצה לחסוך אותה כמה שאפשר, ולכן הקדמת איפוס bottom היא דבר רצוי.

השורה המוקדמת ביותר שניתן לבצע בה את איפוס bottom היא שורה 19, כלומר להכניס אותה בין שורות 18, 19 הנוכחיות, ויש להכניסה בהתניית bottom == oldTop. בין שורה זו למיקום הנוכחי של איפוס bottom לא נעשה כל שינוי ב-bottom או ב-oldTop, ואלו משתנים מקומיים / רק חוט אחד (בעל התור) מבצע בהם כתיבות, ולכן בטוח להקדים את האיפוס לשורה זו.

מימוש זה עדיין יכול להגיע ל-overflow באופן הבא: בעל התור רק מכניס משימות, כאשר בכל הכנסה הוא מעלה את bottom ב-1, והיחידים שיבצעו את המשימות יהיו גנבים. באופן זה לעולם לא יקרא popBottom ולכן לעולם לא יורד ערכו של bottom. כיוון ש-tasks נקבע בגודל קבוע בעת יצירתו (ל-capacity), מתישהו קריאה ל-pushBottom תגרום ל-overflow.

<u>שאלה 198</u>:

1. אם מספר החוטים n יהיה גדול יותר מ-m (תחת הנחה שיש barriers במקומות המתאימים), אז יתכנו אחד משני המקרים הבאים:

- אם ל-$n-m$ החוטים הנותרים ינתנו i שנעשה בהם כבר שימוש (כאשר i הוא בין 0 ל-$m-1$, גודל $a[]$ הוא m), אזי יהיה i אחד לפחות עבורו יתבצע ע"י שני חוטים שונים $a[i]+=sum$ עבור sum מסויים, ותיתכן ריצה בה יתווסף ל-$a[i]$ למעשה $2*sum$ והסכום החלקי יהיה שגוי.
- אם ל-$n-m$ ינתנו מספרים שאינם בטווח ה-i של m החוטים הראשונים, יזרק מתישהו ArrayOutOfBoundException כאשר יעשה ניסיון גישה ל-$a[i]$ בניסיון ההשמה בשורה 14.

2. תחת הנחה שיש מספיק חוטים לביצוע המשימה עבור m נתון, יש להוסיף barriers בשני מקומות בקוד:

- בין שורות 12 ל-13, כלומר אחרי ביצוע $sum = a[i-d]$; : אם לא יושם שם מחסום, יתכן שבשלב אחרי כן חוט כלשהו יבצע $a[i] += sum$ כאשר אותו sum שהוא $a[i-d]$ כבר עודכן ע"י ה"בעלים" של מקום $i-d$ במערך, ולכן ה-sum שיתווסף ל-$a[i]$ יהיה שגוי.
- בין שורות 14 ל-15, כלומר אחרי ביצוע $a[i] += sum$; : אם לא יושם שם מחסום אז יתכן שבאיטרציה הבאה של חוט מסויים כאשר יבצע $sum = a[i-d]$, ה"בעלים" של מקום $i-d$ במערך לא הספיק לבצע באיטרציה הקודמת הוספה של sum (שלו), ואז החוט הנוכחי יקרא $a[i-d]$ לא מועדכן והסכום יהיה שגוי.

לפיכך יש צורך בשני מחסומים לפחות, שכן יש צורך בהפרדה בין שלב הקריאה של כל חוט לשלב הכתיבה של כל חוט (הוספת הערך שקרא לסכום החלקי עליו אחראי), ובין שלב הכתיבה הזה לאיטרציה הבאה – שלב הקריאה הבא.

<u>שאלה 199</u>:

להלן שינוי קוד ה-sense-reversing barrier בהם החוטים הממתינים קוראים ל-wait במקום להסתובב:

```
public synchronized void await() {
        boolean mySense = threadSense.get();
        int position = count.getAndDecrement();
        if (position == 1) {
                count.set(size);
                sense = mySense;
                notifyAll();
        } else {
                while (sense != mySense) {
                        try {
                                wait();
                        } catch (InterruptedException e) {}
                }
        }
        threadSense.set(!mySense);
}
```

המימושים השונים יהיו טובים אחד מהשני כתלות במרווחים בין הקריאות למתודת ה-await ע"י החוטים השונים:

- אם המרווחים בין הקריאות יהיו <u>גדולים</u> יחסית (מספיק עד כדי שהחוט הקורא יספיק לבצע את wait במימוש החדש) אז המימוש עם ה-wait עדיף, שכן בגלל המרווחים לא יהיה contention גדול על המתודה (שכן היא synchronized), ולעומת המימוש המקורי אנו חוסכים את ה-spin על תנאי ה-mySense =! sense כיוון שהחוט נכנס להמתנה עד שיקבל הודעה (notifyAll). לעומת זאת במימוש המקורי לא יחסכו פעולות CPU על ה-spin של כל החוטים, וכיוון שהחוטים לא יישנו, בכל getAndDecrement יהיה יותר עומס על ה-bus בעדכון המשתנים המשותפים לעומת המימוש עם ה-wait.
- אם המרווחים בין הקריאות יהיו <u>קטנים</u> יחסית, במימוש עם ה-wait יהיה contention גבוה על הבלוק ה-synchronized, שלא יהיה במקרה של המימוש המקורי. במימוש המקורי גם לא תהיה המתנה ארוכה (spin) של החוטים הראשונים שמגיעים, כיוון שבמקרה זה "כולם מגיעים יחד" למתודה (ולכן תנאי שחרור המחסום יגיע יחסית מהר). לפיכך במקרה זה המימוש המקורי עדיף.