תכנות מרובה ליבות / תרגיל בית #4

אריאל סטולרמן

שאלה 1.1:

להלן תוצאות זמני הריצה **בשניות** עבור בדיקת LockBasedHashTable על 8 חוטים כאשר כל אחד מהם מבצע 1,000,000 הכנסות ראנדומליות (מסומנים זמני הריצה המהירים ביותר עבור אותה שורה):

Load factor \ #locks	1	2	8	32	256	1024	4096
10	24.77767	30.98138	32.03905	37.97644	26.8108	29.2175	27.45146
20	36.6123	24.08551	27.53221	29.8674	24.88119	25.1246	28.01137
50	48.42842	31.68005	34.32379	25.31122	24.31806	23.23993	31.53346
100	78.92194	47.99365	28.95626	27.17742	24.32194	23.25888	23.13957

שאלה 1.2:

פרק 8, שאלה 96:

- מימוש באמצעות locks ו-conditions נמצא בקובץ **exercise96a.java**. להלן הסבר מדוע נשמר mutex וחוסר הרעבה:

- ה-mutex נשמר תחילה כיוון שכל פעולה מוגנת באמצעות מנעול יחיד שרק החוט שתופס אותו יכול לנסות להיכנס או להשתחרר מהקטע הקריטי. החלק העיקרי הוא שאם מין כלשהו נמצא בפנים, דהיינו המונה שלו גדול מ-0, המין השני לא נכנס אלא ממתין עד שיתריעו לו. רק כאשר האחרון מהמין שהיה עד כה בפנים יוצא, ישנה התרעה למין השני ורק אז יוכל לנסות להיכנס.
- המימוש שומר על חוסר הרעבה כיוון שבכל רגע נתון (פרט להתחלה), אם יש ממתינים משני המינים, תמיד מוגדר מין אחד להיות הבא בתור להיכנס, כך שאם מין א' בפנים, מין ב' ממתין ומגיעים עוד ממין א' שממתינים, כאשר האחרון שבפנים ממין א' יצא, מובטח שמין ב' יותרע ויכנס. במקרה זה מין ב' לפני כניסתו דואג לכוון את המין הבא שיכנס להיות שוב מין א', כך שההגינות תשמר לכל אורך חיי התוכנית.

- מימוש באמצעות synchronized, wait, notify, notifyAll נמצא בקובץ **exercise96b.java**. להלן הסבר מדוע נשמר mutex וחוסר הרעבה:

הלוגיקה במימוש זה זהה ללוגיקה במימוש הקודם, ולכן נכונה. המימוש נכון בכך שאת המנעול הראשי במימוש הקודם מחליף סינכרון מעל אובייקט השירותים המשותפים (ע"י הכרזת מתודות הכניסה והיציאה כ-synchronized), ולכן מובטח סינכרון מעל כל חוט מכל מין שיסתכל עליו. כמו כן כל מין ממתין על אובייקט נוסף שקשור למין שלו בלבד, ולכן כל המתנה של מין אחד ושחרור כולם מאותו מין אינה משפיעה על המתנת המין האחר ושחרורו מהמתנה.

פרק 8, שאלה 97:

קוד הפתרון נמצא בקובץ **exercise97.java**.

פרק 9, שאלה 106:

מימוש list ה-optimistic לא יהיה נכון אם נחליף את סדר הנעילה של pred ו-curr בפעולת ה-add, כיוון שאם הפעולה תרוץ במקביל לחוט אחר המבצע remove ומנסה לתפוס את המנעולים על אותם צמתים אך בסדר הרגיל, יתכן deadlock – החוט המריץ את add יתפוס את המנעול של curr והחוט המריץ את remove יתפוס את המנעול של pred, ושניהם ימתינו שהשני ישחרר את המנעול הבא שהם צריכים לתפוס לנצח.

פרק 9, שאלה 109:

האלטרנטיבה המוצעת היא אכן לינאריזבילית. אם שתי קריאות contains רצות במקביל (יש ביניהן חפיפה כלשהי), הריצה לינאריזבילית. נבחן את המקרים בהם שתי ריצות contains רצות באופן סדרתי אחת אחרי השניה, בעוד נעשית קריאה למתודה המשנה את מבנה הנתונים המקבילה לשתי ריצות ה-contains הנ"ל. נניח כי מתודה זו היא insert. אם קריאת ה-contains הראשונה מחזירה false, אין בעיה שכן בין אם ה-contains השניה תחזיר false ובין אם תחזיר true, הריצה לינאריזבילית. אם הקריאה הראשונה של contains מחזירה true, אזי נרצה שהשניה תחזיר בהכרח true.

אם הראשונה החזירה true, אז היא בהכרח ראתה את השינוי של insert של המצביע שבפועל הכניס את האיבר לרשימה. כיוון שהקריאה השניה של contains קרתה אחרי שהראשונה הסתיימה, בהכרח גם היא תחזיר בתרחיש זה true, ולכן הריצה לינאריזבילית. באופן דומה ניתן לתאר מקרה בו ישנה ריצת remove מקבילה לשתי קריאות contains סדרתיות: כאן אם הראשונה מחזירה false, השניה בהכרח תחזיר false.

כיוון שגם ב-insert וגם ב-contains נקודת הלינאריזציה היא בשינוי המצביע המכניס / מוציא איבר מהרשימה, ולפני השינוי ואחריו האובייקט קונסיסטנטי, אזי כל שתי ריצות סדרתיות ולא חופפות של contains יהיו עקביות אחת עם השניה כמתואר במקרה לעיל עבור insert, ובאופן מתאים עבור remove.

<u>פרק 9, שאלה 112</u>:

הבעיה בפונקציית ה-validate הפשוטה המוצעת היא שאם יוכנסו צמתים חדשים בין ה-pred ל-curr ע"י חוט כלשהו בזמן שחוט אחר מנסה לבצע add או remove (ההכנסה תתבצע בזמן בין יציאה מלולאת ההמתנה של החוט האחר ועד תפיסת המנעולים של pred ו-curr על ידו), אז כל הצמתים שהוכנסו ביניהם יימחקו. פעולת ה-validate תעבור כי אף אחד מ-pred, curr לא נמחק, ואילו שינוי המצביעים בהתאם לפעולה יגרום לצמתים שהוכנסו בינתיים להעלם.

<u>פרק 10, שאלה 120</u>:

במימוש הנתון כן נדרש memory barrier. להלן דוגמא לתרחיש ללא memory barrier בו הריצה נתקעת:

- ה-deq מנסה לבצע הוצאה, ומתחיל להסתובב על עותק מקומי של head, tail כיוון שהתור ריק.
- ה-enq מכניס איברים עד שהתור מתמלא. בדרך הוא מעדכן את items ואת tail מקומית.
- ה-deq לא רואה את עדכון tail ולכן חושב ש-0 == tail – head, ותקוע בלולאה לעד. בינתיים גם ה-enq תקוע כי התור מלא.

כדי לשמר נכונות יש צורך להוסיף memory barrier, כלומר volatile, על המשתנים head, tail. כך, בסוף כל ריצה של כל פעולת enq / deq, כל ה-cache מתחילת הריצה יתעדכן בשני החוטים (האחד הוא המכניס והשני הוא המוציא), הכוללת גם את עדכון המערך items על תוכנו וכמובן גם את עדכון המונים head, tail. כיוון שכל אחד מהחוטים מעדכן ערך אחד מ-head, tail וקורא את השני מספיק להשתמש ב-volatile ואין צורך ב-RMW.

אם נהפוך את head, tail ל-volatile, מובטח שבסוף כל פעולה יעודכן החוט השני בערכים החדשים. נשים לב שעד הרגע בו החוט השני, שכרגע ממתין, מתעדכן, נכונות מבנה הנתונים נשמרת כיוון שהוא רק מבצע spin ע"י קריאה בלבד. סיום פעולת החוט המשנה את מבנה הנתונים תשחרר את החוט הממתין, וברגע שחרורו מובטח גם שהתור אצלו יהיה מעודכן, כך שאם למשל מכניס ביצע הכנסה של איבר לתור ועדכן את tail, מוציא בהכרח יראה אותו ולא יהיה מצב בו המוציא ינסה להוציא איבר מאינדקס במערך שלא מכיל כלום. בכיוון השני המצב עוד יותר פשוט: המוציא רק מבצע עדכון של ה-head ולא משנה את התור עצמו, ובעדכון ה-head הוא משחרר את החוט המכניס.

ה-memory barrier הכרחי כיוון שהוא מבטיח שאף אחד מהחוטים לא יסתובב לעד כאשר בפועל מבנה הנתונים מאפשר לו זאת (enq לא יסתובב לעד אם התור לא מלא, deq לא יסתובב לעד אם התור לא ריק), ומבטיח שכאשר החוטים ניגשים למבנה הנתונים או משנים אותו, נכונותו נשמרת.

<u>פרק 10, שאלה 125</u>:

1. פעולת ה-enq היא wait-free (ולכן גם lock-free) כיוון שהיא מסתיימת במספר פעולות חסום תמיד, ובפרט סופי. לעומת זאת, פעולת ה-deq אינה wait-free שכן יתכן מצב בו התור יהיה ריק, והמתודה תרוץ בלולאה לנצח. כמו כן המתודה אינה lock-free כיוון שאם היא רצה לבד על תור ריק (כלומר רצה לנצח), המערכת אינה מתקדמת - אף חוט לא מצליח לבצע התקדמות.
2. נקודות הליאריזציה ב-enq ו-deq אכן תלויות בריצה. נבנה ריצה סדרתית תקינה מתוך הריצה המקבילה, וכך נבצע לינאריזציה. נסמן את מצב התור $x_1, \dots, x_k$ שהם האיברים המאחסנים את האינדקסים $1, \dots, k$, ונשים לב לשינויים במהלך הריצה והיסטוריית הריצה הסדרתית המתאימה, בה נזהה את נקודות הלינאריזציה. בתחילת הריצה כמובן שהתור ריק כמו גם היסטוריית הריצה הסדרתית.

- <u>enq</u>: כאשר נתקלים בפעולת items[i].set(x); בתוך פעולת ה-enq, מצב התור לפני ההכנסה הוא $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_k$ בעת הפעולה משתנה מצב התור ל-$x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_k$ –פעולה זו היא נקודת הלינאריזציה, וניתן בנקודה זו להוסיף את enq(x) להיסטוריית הריצה הסדרתית.
- <u>deq</u>: כאשר נתקלים בפעולת T value = items[i].getAndSet(null); בתוך פעולת ה-deq, כאשר פעולת ה-getAndSet מחזירה אובייקט x, נסמן את מצב התור $X, x, \dots, x_k$ כאשר X היא קבוצת האיברים עם אינדקס קטן משל x. אם X ריקה, אזי x הוא ראש התור, ולכן ניתן לקבוע

את נקודת הלינאריזציה בנקודה זו ולהסיר את x מהתור. אם X לא ריקה, אזי כל ההכנסות של האיברים ב-X קרו באופן מקבילי להכנסה של x, כיוון שבפעולת ה-deq ישנו מעבר על כל האינדקסים הקטנים מזה של x שלא נראו בהם איברים (אך הם שם – X לא ריק). יתכן גם ש-deq אחר אסף את אותם איברים, אך מצב זה הוא כמתואר קודם. כיוון שהכנסת x היתה לאינדקס גדול יותר מכל האיברים ב-X, אזי הכנסתו חפפה לכל ההכנסות של האיברים ב-X, ולפיכך ניתן להביא את x להיות בראש התור, להביא את enq(x) להיות בהיסטוריה לפני הכנסות כל האיברים ב-X ולשים בהיסטוריה את deq(x) מיד לאחר enq(x). באופן זה נקודת הלינאריזציה זהה למקרה ש-X ריקה, לאחר שהראינו ריצה סדרתית חוקית מתאימה לריצה המקבילה. לפיכך פעולת ה-getAndSet היא נקודת הלינאריזציה.

<u>פרק 11, שאלה 132</u>:

קוד הפתרון נמצא בקובץ **exercise132.java**. להלן הסבר מדוע הקוד הנתון לא עובד, ומדוע הקוד המוצע מתקן זאת:

המימוש הנתון לא יעבוד במקרה שפעולות push ו-pop רצות במקביל. במקרה כזה יתכן למשל מצב בו פעולת push מעלה את top ב-1, ולפני שמכניסה את הערך, פעולת pop מורידה את top ב-1 ומחזירה את הערך שרואה ב-top, אך ערך זה סתמי ולא אמור להיות מוחזר. הפתרון המוצע מתקן זאת ע"י סינכרון תוך שימוש בחדרים. המימוש מנצל את העובדה שבכל רגע נתון רק חדר אחד יכול להיות מלא במספר כלשהו של חוטים, ולכן משתמשים בשני ערכים – אחד ל-push ואחד ל-pop, כאשר כל חוט הרוצה לבצע את אחת הפעולות צריך תחילה "להיכנס" לחדר המייצג את אותה פעולה. כך לא יתכנו פעולות push ו-pop מקביליות, אלא רק פעולות מאותו סוג באופן מקביל. כמובן שאין בעיה שפעולות מאותו סוג ירוצו באופן מקביל, כיוון שנקודת הלינאריזציה היא ב-top.getAndDecrement / top.getAndIncrement כך שלא תיווצר בעיה בין החוטים המבצעים את אותה פעולה.