תכנות מרובה ליבות / תרגיל בית 2#

אריאל סטולרמן

(1)

בכל אחת מהריצות תוצאת הריצה היתה 200,000. נראה כי בריצת Ex2q1 (הלא מתוקנת) למרות ש**איברי** המערך אינם מוגדרים volatile עדיין הריצה בכל הפעמים נגמרה ב-200,000. יתכן שזה נובע מכך שה-victim הוגדר כ-volatile, וכאשר הובא מהזיכרון יתכן שמערכת ההפעלה הביאה איתו גם את איברי המערך, וכך הם בפועל היו גם כן volatile (זו סתם השערה).

להלן ריכוז ממוצעי תוצאות זמני הריצות הממוצעים (כפי שמופיע בקובץ ה-doc.txt):

תוכנית / פלטפורמה (מספר חוטים)	חוט בודד (על nova)	Nova	aries
Ex2q1	23723666	50121000	104221216
Ex2q1fix	35067666	68417000	146409351

ניתן לראות באופן מובהק שזמן הריצה גדל ככל שמספר החוטים במכונה עליה רצים גדל. כמו כן ניתן לראות באופן מובהק שזמני הריצה של המימוש הפשטני קצרים יותר מהמימוש באמצעות ה-Java Memory Model.

(2) **בונוס**

א. -

ב.

נראה כי דרגת הקונצנזוס של peekableStack(k) היא $k+1$. להלן פרוטוקול החלטה ל-$k+1$ חוטים:

הפרוטוקול משתמש במחסנית מסוג peekableStack(k) כדי להחזיק את הערכים המוצעים. נסמן מחסנית זו כ-proposed.

```
private void propose(Object value){
        proposed.push(value);
}

public decide(Object value){
        propose(value);           // push value to peekableStack
        Object ans = proposed.look();
        if (ans == false) do{
                ans = propose.pop();
        } while (!proposed.empty());
        return ans;
}
```

נכונות האלגוריתם:

- כל הצעה שמתבצעת נעשית באופן אטומי כיוון שהפעולות על המחסנית הנתונה הן אטומיות (עבור n חוטים), בפרט push().
- מובטח שכל חוט יחזיר את ערכו של החוט הראשון שהכניס את ערכו למחסנית:
  - אם מדובר בחוט כלשהו מ-k החוטים הראשונים שמבצעים look(), מובטח שיהיה ערך בתחתית המחסנית: כל פעולת look() שנעשית ע״י כל חוט היא אחרי שביצע push() של ערכו למחסנית, ולכן מובטח שכל חוט שיציץ יראה ערך בעת ההצצה, ולא יתכן שההצצה תהיה לפני שיש לפחות ערך אחד במחסנית. כיוון שהראשון שמצליח לבצע push() של ערכו למחסנית קובע לכל אורך חיי המחסנית את הערך שיהיה בתחתיתה, כל החוטים המבצעים look() יחזירו את אותו ערך.
  - אם מדובר בחוט ה-$k+1$ שביצע look(), והוא מקבל false, הרי שבהכרח היו k חוטים לפניו שקיבלו ערך וההחלטה נעשתה עבורם על הערך הנמצא בתחתית המחסנית, ואין סכנה אם כן שהחוט האחרון יוציא איברים מהמחסנית (לא יפגע בהחלטות האחרים). מובן כי כאשר יוציא את כל האיברים עד האחרון מהמחסנית, הערך האחרון אליו יגיע הוא בדיוק זה בתחתית המחסנית עליו כל קודמיו החליטו.

לפיכך עבור מספר הקונצנזוס של peekableStack(2) הוא 3, ולכן לא ניתן לממשו באמצעות מחסניות ורגיסטרים אטומיים, שכן כפי שיוכח בהמשך מספר הקונצנזוס של מחסנית הוא 2, וכידוע של רגיסטרים אטומיים הוא 1, ולא ניתן לממש אובייקט כלשהו באמצעות אובייקטים שמספר הקונצנזוס שלהם קטן ממש מאותו אובייקט שרוצים לממש.

שאלות מהספר:

פרק 3, שאלה 24:

<u>Quiescently consistent</u>:

כן – ניתן להסתכל על הריצה הסדרתית הבאה, המוכיחה quiescent consistency:

$$r.write(1) \to r.read(1) \to r.write(2) \to r.read(2)$$

<u>Sequentially consistent</u>:

כן – כיוון שכל היסטוריה Linearizable היא גם sequentially consistent (הוכחת לינאריזביליות בהמשך).

<u>Linearizable</u>:

כן – ההוכחה ע"י סימון נקודות הלינאריזציה בשרטוט.

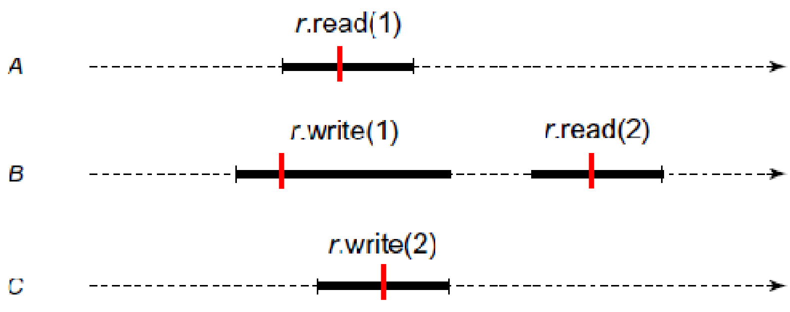

Figure 3.13 First history for Exercise 24.

<u>Quiescently consistent</u>:

כן – ניתן להסתכל על הריצה הסדרתית הבאה, המוכיחה quiescent consistency:

$$r.write(2) \to r.write(1) \to r.read(1) \to r.read(1)$$

<u>Sequentially consistent</u>:

כן – כיוון שכל היסטוריה Linearizable היא גם sequentially consistent (הוכחת לינאריזביליות בהמשך).

<u>Linearizable</u>:

כן – ההוכחה ע"י סימון נקודות הלינאריזציה בשרטוט.

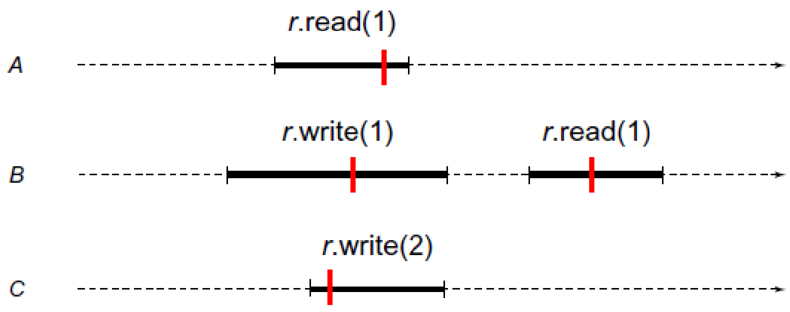

Figure 3.14 Second history for Exercise 24.

פרק 3, שאלה 31:

המתודה m המוגדרת כך שבכל היסטוריה, בפעם ה-i בה thread קורא ל-m, היא חוזרת לאחר 2^i צעדים, היא wait-free אך אינה bounded-wait free. הסיבה שהיא wait-free היא שבכל קריאה, בהנחה ש-m בנויה כך שאין נעילות, המתודה חוזרת תוך מספר סופי של צעדים, לכל פעם $i \in \mathbb{N}$ בה היא נקראת. הסיבה לכך שאינה bounded wait-free היא שאין חסם עליון למספר הצעדים שהמתודה יכולה לרוץ, גם כתלות במספר החוטים, קיימת תלות במספר הקריאות למתודה ע"י כל חוט. באופן תיאורטי, חוט שרץ אינסוף זמן וקורא קריאות חוזרות ונשנות ל-m יגרור כי m אסימפטוטית תרוץ אינסוף זמן, ולעומת זאת אם m היתה bounded wait-free, גם בזמן אינסוף קריאה ל-m היתה נגמרת תוך זמן סופי.

פרק 4, שאלה 40: **בונוס**

אלגוריתם ה-Mutex של Peterson עבור שני חוטים היה עובד אם היינו מחליפים את ה-shared atomic registers ב-regular registers. נניח כי שני חוטים A ו-B משתמשים במנעול כמתואר בשאלה ונראה כי לא יתכן ששניהם ב-CS. יתכנו שני מצבים:

- חוט A קרא $flag[B] = false$ ולכן נכנס ל-CS. במקרה זה הפעולה המאוחרת ביותר שיכול היה B לעשות בזמן קריאת A את $flag[B]$ (בדרכו לתפוס את המנעול) היא כתיבת true ל-$flag[B]$, וכיוון שהרגיסטר רגולרי אפשרי ש-A קרא בכל זאת false. במקרה זה A נמצא כבר אחרי כתיבת true ל-$flag[A]$ ואחרי כתיבת עצמו ל-victim. כעת כאשר B יגיע לבדיקת תנאי ה-while אחרי שכתב עצמו ל-victim, ואז יהיה זה אחרי ש-A כתב כבר true ל-$flag[A]$ והכתיבה האחרונה שהיתה ל-victim היא B – תנאי ה-while מתקיימים. לכן B לא יכנס ל-CS.
- חוט A קרא $flag[B] = true$ אך $victim = B$ ולכן נכנס ל-CS. במקרה זה B היה תו"כ כתיבת B ל-victim או אחרי שלב זה. כאשר B יגיע לשלב בדיקת תנאי ה-while, יהיה זה אחרי ש-A כתב כבר true ל-$flag[A]$ והכתיבה האחרונה ל-victim היא B. לכן B לא יכנס ל-CS.

פרק 4, שאלה 41:

אם קריאות ל-write() לעולם אינן חופפות, אז:

- מימוש זה רגולרי. נניח מתבצעת כתיבה לרגיסטר, ובאותו זמן מתבצעת קריאה בחוט P_i. הקריאה מחזירה את הערך המקומי של הרגיסטר ב-P_i. אם הערך הנוכחי הוא לפני שבסבב הכתיבה הגיע אליו, הקריאה תחזיר את הערך הישן. אם סבב הקריאה כבר עבר את P_i אז הערך שיוחזר יהיה הערך החדש. כיוון שמובטח שאין פעולות כתיבה חופפות, אזי בזמן פעולת כתיבה בודדת, כמתואר לעיל, יוחזר בכל חוט או הערך הישן או החדש. לפיכך המימוש הוא רגולרי.
- מימוש זה אינו אטומי, נתאר דוגמא נגדית: נניח כי P_i מתחיל כתיבה, P_{i+1} מעדכן בעקבות הכתיבה את הערך המקומי שלו. כעת נניח כי מתבצעות שתי קריאות אחת אחרי השניה ללא חפיפה: P_{i+1} קורא את הערך, והוא יהיה הערך החדש; P_{i+2} קורא את הערך והוא יהיה הערך הישן. בגלל שקריאת P_{i+2} התבצעה ללא חפיפה אחרי קריאת P_{i+1}, אז P_{i+2} היה צריך לקרוא את הערך החדש. לכן ההיסטוריה הנ"ל לא לינארזבילית ולכן המימוש אינו אטומי.

אם יתכנו קריאות חופפות ל-write() אז כמובן שהמימוש אינו אטומי, כיוון שהוכח לעיל שאינו אטומי אף עבור קריאה בודדת ל-write().

פרק 5, שאלה 53:

להלן הוכחה לכך שה-consensus number של המחלקה Stack הוא בדיוק 2:

תחילה נוכיח כי ה-consensus number הוא לפחות 2, ע"י תיאור אלגוריתם פתרון קונצנזוס לשני חוטים באמצעות אובייקט מסוג Stack. נגדיר את הפרוטוקול באופן הבא, הדומה לשימוש בתור כפי שהוצג בהרצאה:

- אתחול המחסנית: שני כדורים, אחד שחור ואחד אדום, כאשר האדום בראש המחסנית (הראשון שיצא ב-pop()).
- מערך רגיסטרים אטומים בגודל 2 – כאשר הראשון מוקצה לכתיבת ערכו של החוט הראשון, והשני עבור ערכו של החוט השני.
- כאשר חוט קורא למתודת decide() הוא תחילה כותב את ההצעה שלו למקום שלו במערך, ואז מבצע pop(). אם קיבל כדור אדום, מחליט על הערך שלו. אחרת (קיבל כדור שחור), מחליט על הערך השני במערך (של החוט השני).

נכונות הפרוטוקול נובעת מכך שמהגדרת המחסנית, אם הראשון קיבל את הכדור האדום, השני בהכרח קיבל את הכדור השחור (פעולת ה-pop() אטומית). כמו כן, כאשר המפסיד ניגש לערך המנצח במערך, הוא בהכרח כתוב שם: כיוון שהמפסיד הוציא את הכדור השחור, בהכרח קודם לכן הוצא הכדור האדום ע"י המנצח, ולכן בהכרח המנצח כבר כתב את ערכו למערך.

נראה כעת כי דרגת הקונצנזוס של Stack לא גדול מ-2. נניח בשלילה כי ניתן לפתור קונצנזוס באמצעות מחסנית עבור 3 חוטים. יהיו A,B,C שלושה חוטים. לפי הנחה כי מתקיים קונצנזוס, קיים מצב קריטי בו (בה"כ) אם A מבצע את הצעד הבא הערך המוחלט יהיה 0, ואם B מבצע את הצעד הבא הערך המוחלט יהיה 1. נבחן את המצבים במכפלה הקרטזית בין pop(), push() ע"י A ובין פעולות אלו ע"י B:

- אם A,B שניהם מבצעים pop(): אם A רץ ראשון ואחריו B, יוחלט 0. אם B רץ ראשון ואחריו A יוחלט 1. אם C רץ אחרי pop() של שניהם, הוא לא מסוגל להבחין בין שני המצבים (כי מצב המחסנית זהה), אך באחד מחליט 0 ובשני 1. לכן זו סתירה.
- אם אחד מבצע push() והשני pop(): אם במחסנית לפחות איבר אחד, C לא יוכל להבחין בין מצב שהוא 0-valent בו A מבצע push() ו-B pop() ובין מצב שהוא 1-valent בו B מבצע pop() ו-A push() (או הפוך מבחינת מי מבצע איזו פעולה), ומכאן סתירה. אם המחסנית ריקה, אז

C לא יכול להבחין בין המצב ה-0-valent בו A מבצע ()push ו-B לא עושה כלום, ובין המצב ה-1-valent בו B מבצע ()pop (שלא עושה כלום) ו-A מבצע ()push, ומכאן סתירה.

- אם שניהם מבצעים ()push : אם בה"כ A מבצע ()push ראשון ולאחריו B, עוברים למצב שהוא 0-valent. כדי ש-B "יכיר" זאת, הוא חייב לעשות מתישהו ()pop לאיבר ש-A הכניס, אחרת לא יוכל להבחין בין מצב זה ולבין המצב הבי-ולנטי בו היה לפני ש-A ביצע ()push. באופן דומה יתכן מצב בו B מבצע ()push ראשון וכן הלאה. אם לאחר ה-()pop של האיבר של זה שביצע ()push ראשון ירוץ בריצת סולו C, הרי שלא יוכל להבחין בין שני המצבים, כאשר אחד הוא 0-valent והשני הוא 1-valent, ומכאן סתירה.

מכאן שלא ניתן לממש קונצנזוס עבור 3 חוטים באמצעות Stack, ולפיכך ולפי החלק הראשון נובע כי מספר הקונצנזוס של Stack הוא 2 בדיוק.

פרק 5, שאלה 59 : -

פרק 5, שאלה 70 :

מימוש consensus עבור n חוטים זהה לזה שראינו בהרצאה :

```
public class RMWConsensus extends ConsensusProtocol {
	private AtomicInteger r = new AtomicInteger(-1);

	public T decide(T value) {
		propose(value);
		r.compareAndSet(-1,i);
		return proposed[r.get()];
	}
}
```

הנכונות נובעת מהנכונות שהוכחה בהרצאה : כיוון שפעולת ה-compareAndSet אטומית ומתבצעת ע"י כל חוט רק אחרי שהציע כבר ערך, אזי החוט הראשון שיבצע compareAndSet יחזיר את ערכו כי עבורו ה-compareAndSet הצליח עם 1-, וכל חוט אחר שיקרא ל-compareAndSet בהכרח יחזיר את ערך החוט הראשון, שכאמור כבר הוצע וקריא לכל שאר החוטים. אם נריץ פרוטוקול זה עבור n חוטים, כיוון שמובטח שעבור כל n הקריאות הראשונות פעולת ה-compareAndSet מתבצעת באופן תקין, וכל חוט יקרא ל-compareAndSet פעם אחת בלבד, הרי שמספר הקונצנזוס של compareAndSet עבור n חוטים הוא אכן n.

נראה כי לא ניתן לממש קונצנזוס באמצעות compareAndSet ל-n ריצות עבור n+1 חוטים : יהיו A,B,C שלושה חוטים מתוך n+1 החוטים. נניח כי המערכת במצב קריטי, כך שאם חוט A מתקדם אז מגיעים למצב שהוא 0-valent, ואם חוט B מתקדם מגיעים למצב שהוא 1-valent. אם A ביצע את הצעד ראשון ואחריו רצו כל החוטים כאשר C היה האחרון, אז קריאת C מחזירה ⊥ והוא רץ ריצת סולו ולכן בהכרח יחזיר 0. באופן דומה אם B היצא את הצעד ראשון ואחריו רצו כל החוטים כאשר C היה האחרון, אז קריאת C מחזירה גם כאן ⊥ והוא רץ ריצת סולו ולכן בהכרח יחזיר 1. שני המצבים השונים מהם מתחיל לרוץ C לא ניתן להבחנה, כאשר אחד מסתיים ב-0 והשני ב-1, ומכאן סתירה. לפיכך לא ניתן לממש באמצעות compareAndSet ל-n ריצות קונצנזוס ל-n+1 חוטים, ולכן מספר הקונצנזוס של אובייקט זה הוא בדיוק n.