תכנות מרובה ליבות / תרגיל בית #3

אריאל סטולרמן

שאלה 1 : שאלת תכנות :

להלן תוצאות זמני הריצה ב-nanosecs על aries בממוצע על 3 ריצות עבור ריצת 4 חוטים (בסוגריים : אחוז הכישלונות הממוצע) :

PriorityQueue1 results:

Initial priorities \ timeout (millisecs)	20	100	500
10	197341935 (0%)	98200494 (0%)	93412748 (0%)
20	109619707 (0.0025%)	111023403 (0%)	89679058 (0%)
30	105993396 (0%)	105827654 (0%)	106098608 (0%)

PriorityQueue2 results:

Initial priorities \ timeout (millisecs)	20	100	500
10	924568470 (0%)	308878594 (0%)	347110842 (0%)
20	420370770 (0.00125%)	307867096 (0%)	296487961 (0%)
30	347929057 (0%)	301942038 (0%)	307197750 (0%)

פירוט מלא של כל הריצות ב-doc.txt.

פרק 6, שאלה 77 :

להלן הצעה לתיקון הבנייה האוניברסלית כך שתתאים לאובייקטים עם non-deterministic sequential specification :

הרעיון הוא להחליף את האובייקט לאובייקט דטרמיניסטי אקוויוולנטי בכדי לשמר את אופן העבודה עם הקונצנזוס כאשר מחליטים על סדר הפעולות שיבוצעו על האובייקט ע״י כל חוט בעותק המקומי שלו, וכמו כן להוסיף קונצנזוס נוסף לכל מצב לאחר ביצוע כל פעולה ע״י כל חוט (בעותק המקומי) כך שכולם יחליטו על מהו המצב הבא באופן אחיד. בשונה מהבנייה האוניברסלית המוכרת, לאחר כל פעולה על האובייקט כל חוט מציע את מצב האובייקט כפי שהוא רואה אותו והמצב המשותף הבא מוחלט ע״י קונצנזוס. באופן זה מובטח שכל החוטים רואים בדיוק את אותו מצב של האובייקט לאחר כל פעולה שנעשית עליו.

החלק הראשון נדרש בכדי להשאיר את אופן העבודה עם האובייקט כפי שהוא, כך שהתחרות בין החוטים בקונצנזוס תהיה על מצב דטרמיניסטי בודד שניתן לתארו בפשטות בכל שלב. החלק השני נדרש כיוון שלא מובטח שמצב האובייקט יהיה זהה בין כל החוטים, גם אם כולם החלו עליו את אותה פעולה כאשר היה באותו מצב, וזאת כיוון שהתנהגותו לא דטרמיניסטית. לשם כך אחרי כל ביצוע פעולה בעותק מקומי, צריך להכריע בקונצנזוס איזה מהמצבים שנוצרו בקרב כל חוט יהיה זה שיחליטו עליו שהוא המצב הבא של האובייקט, וכך החלטה זו תהיה משותפת, אחידה ו-wait-free כנדרש.

פרק 7, שאלה 85 :

הבעיה במימוש הנתון עבור BadCLHLock תחילה היא שעבור החוט הראשון הרץ ייזרק NullPointerException, שכן tail לא מאותחל ולכן לאחר השמת tail (כרגע null) ל-pred, כאשר ינסה לבדוק את תנאי הלולאה pred.locked ייזרק החריג.

גם אם נתקן את הבעיה (הקטנה) הזו, המימוש עדיין בעייתי כיוון שיכול להיווצר dead-lock. יתכן מצב בו חוט A תופס שוב ושוב את המנעול וכך מונע מחוט B וכל החוטים שאחריו לתפוס את המנעול : חוט A תופס את המנעול כך ש-A.myNode.locked == true, וכאשר B בודק אותו הוא ממשיך להמתין. לאחר מכן A עוזב את הקטע הקריטי, משנה את A.myNode.locked == false ומיד תופס אותו שוב כך ששוב A.myNode.locked == true, ורק אז B בודק שוב את A.myNode.locked – וממשיך להסתובב. במקרה אחר חוט A נכנס ל-dead-lock לבדו ונתקע לעד (וכמובן כל חוט שבא אחריו) : חוט A תופס את המנעול כך ש-A.myNode.locked == true, ומשנה את tail שיצביע על ה-QNode שלו. לאחר מכן חוט A יוצא מהקטע

הקריטי, ומכוון את A.myNode.locked == false. כעת חוט A שוב תופס את המנעול: מכוון את A.myNode.locked == true וגם את pred = tail.getAndSet(qnode) – אבל tail כבר מצביע ל-QNode של A, ולכן בבדיקת pred.locked שהרגע כיוון ל-true הוא יתקע לעד בלולאה.

<u>פרק 7, שאלה 86</u>:

במימוש החסם הראשון בו יש counter המוגן על ידי מנעול TTAS, התנהגות החסם ב-cache coherent מבוסס bus ל-n חוטים תהיה כך ב-w.c.:

- בכל תפיסת המנעול ע"י אחד מהחוטים כל אחד מהחוטים יעודכן בערך החדש, סה"כ n^2 קריאות.
- בכל שינוי ערכו של המונה מתעדכן כל אחד מהחוטים בערך החדש, סה"כ n^2 קריאות.
- בכל שחרור המנעול ע"י אחד מהחוטים כל אחד מהחוטים יעודכן בערך החדש, סה"כ n^2 קריאות.

במימוש החסם השני בו יש מערך Booleans בו כל חוט מסתובב על הערך של קודמו המציין האם עבר כבר את foo() או לא, התנהגות החסם ב- cache coherent מבוסס bus תהיה חסכונית יותר. אם נניח כי גודל המערך לכל היותר כגודל cache-line (ב-w.c., כך בכל שינוי כל החוטים יצטרכו עדכון), בכל שינוי של ערך במערך, כל החוטים יתעדכנו (גם אלו המסתכלים על ערך במערך שלא שונה), אך בכל מקרה נמשך כל ה-cache-line עליו נמצא המערך. סה"כ כל חוט יעדכן פעם אחת ערך במערך, ולכן סה"כ הקריאות על ה-cache יהיו n^2. בניגוד למימוש הקודם, כאן נחסכים עדכוני תפיסת ונעילת מנעול, ולכן העומס על ה-bus במימוש זה נמוך יותר (אבל, אם נשים לב, המימוש השני כפי שמתואר בספר כלל לא פותר את הבעיה שרצו לפתור מלכתחילה...).

<u>פרק 7, שאלה 91</u>:

להלן מימוש מתודת isLocked() הבודקת האם חוט כלשהו מחזיק במנעול מבלי לתפוס אותו:

<u>Any test-and-set spin-lock</u>:

```
return state.get();
```

פשוט מחזירים את ערכו של הרגיסטר של המנעול – אם הוא true אזי מישהו תפס אותו, ואם הוא false אזי הוא פנוי.

<u>The CLH queue lock</u>:

```
return (tail != null && tail.locked == true);
```

אם המנעול לא נתפס כלל ע"י אף חוט, אז tail יהיה null (אלא אם נניח שהמנעול תמיד יחזיק לכל הפחות QNode עם ערך false, ואז רק הביטוי הימני רלוונטי). אם tail אינו null, הוא מצביע ל-QNode של האחרון ברשימת החוטים הממתינים למנעול או תפסו אותו. רק אם tail.locked == false אז אין אף חוט שממתין למנעול, והחוט האחרון שתפס את המנעול כבר שחרר אותו.

<u>The MCS queue lock</u>:

```
return tail != null;
```

נסתכל מתי המנעול לא תפוס: או לאחר אתחול כאשר אף חוט לא ניגש אליו עדיין, שם tail == null, או כאשר החוט האחרון שממתין למנעול משחרר אותו - כאשר בתהליך השחרור נעשה ניסיון לשים ערך null ל-tail במידה ואין חוט אחר שממתין למנעול אחרי החוט המשחרר. מכאן שכאשר אין אף חוט שממתין למנעול והוא משוחרר, יתקיים tail == null. לפיכך בכל מצב אחר, כאשר tail != null, יהיה המנעול תפוס.

<u>פרק 7, שאלה 92</u>:

בניתוח סיבוכיות המקום בפרק 2 אנו מניחים שאם $n-1$ חוטים נמצאים במצב בו אף אחד מהם לא בקטע הקריטי, וכולם עוצרים בנקודה בה בצעד הבא חוט i כלשהו כותב שהוא נכנס לקטע הקריטי ונכנס אליו, אז ניתן להריץ חוט n כך שיכתוב לרגיסטר אליו כותב חוט i לפני כתיבתו, שייכנס לקטע הקריטי. כאשר החוט i יכנס לקטע הקריטי, הוא יידרוס את כתיבת החוט n ושני חוטים יהיו בקטע הקריטי.

הנחה זו אינה רלוונטית כאשר ניתן להשתמש ברגיסטרי RMW, שכן ניתן לבצע קריאה מרגיסטרים אלו ולאחריהם כתיבה כפעולה אטומית אחת, ובכך למנוע הפרדה בין הקריאה והכתיבה שביניהם יכול חוט אחר לכתוב ומיד להידרס ללא עדות לכך שהוא בקטע הקריטי.