תכנות מרובה ליבות – תרגול 6#

בכיתה ראינו שאם יש אובייקט שמממש קונצנזוס אפשר לבנות כל מיני אובייקטים בצורה wait-free (אמנם בצורה לא יעילה). כיום נעבור על בנייה ב-lock-free ו-wait-free. לבסוף נדבר על cache ופרוטוקול MESI שישמש אותנו בהמשך הקורס.

משפט: אוניברסליות:

קונצנזוס הוא אוניברסלי: באמצעות אובייקט קונצנזוס אפשר לבנות כל אובייקט בצורה wait-free, שיהיה לינאריזבילי...

הבנייה משתמשת במתודת apply שמקבלת כקלט invocation ומחזירה response. נרצה לינארזבליות, כלומר סדר פעולות כלשהו. כל קריאה תהיה node, ותהיה רשימה של nodes המייצגת את סדר הפעולות. הפרוטוקול אומר: כל עוד אני לא מסוגל להוסיף את עצמי לרשימה, אני אקח את המקסימלי ואנסה להוסיף אותו לרשימה. ההחלטה על כך באמצעות קונצנזוס. באמצעות הקונ' מחליטים כולם על אותו node שיהיה הבא. החוט מנסה שוב ושוב עד שמצליח לשים את ה-node שלו ברשימה.

זה lock-free: המערכת תמיד מתקדמת. יתכן שחוט כלשהו לעולם לא יקבל את תורו אך חוט כלשהו יצליח.

לאחר ההחלטה על הרצף, נרצה לדעת מה תהיה תוצאת ההפעלה של מתודה של חוט מסויים – החוט מתחיל לרוץ לפי הלוג על אובייקט מקומי אצלו, ועליו מבצע את הפעולה עצמה. כאשר מגיע לתוצאה הרצויה מחזירה אותה – response.

<u>סיכום</u>: יוצרים היסטוריה משותפת של כל החוטים הרצים יחד. אחרי שהוחלטה ההיסטוריה, כולם רצים באופן פרטי לפי אותה היסטוריה ויחזירו את הערך כפי שאמור להיות.

<u>איך עושים את זה wait-free</u>: עזרה הדדית;

כל אחד מפרסם את הערך שלו. אז עובר לעזור למישהו אחר. בגלל שדואגים לאינטרס של חוט אחר, מובטח שתוך זמן חסום כולם יעזרו לכולם וכולם יתקדמו. לפי פרוטוקול זה, כל חוט לוקח את ה-head (מי שהוא חושב שהוא האחרון). אם הוא צריך עזרה, מסייע לו. אם לא, החוט מבצע את העבודה שלו.

<u>שאלה 1</u>:

נניח שה-tail מאותחל ל-0 ולא ל-1. האם זה יוצר בעיה?

<u>תשובה</u>:

Lock-free: לא ישפיע, אף אחד לא מסתכל עליו.

Wait-free: מפריע, ה-announce בהתחלה מאותחל להיות ה-tail של כולם, והוא 1. אם משנים אותו ל-0, כולם ינסו לעזור לו ולהכניס אותו לרשימה.

<u>שאלה 2</u>:

אם ב-wait-free נחליף את הסדר כך שקודם חוט ינסה להכניס את עצמו, ואם לא רק אז יעזור למישהו אחר:

זה כבר לא יהיה wait-free, אלא רק lock-free.

<u>שאלה 3</u>:

צריך לבנות קוד לאובייקט לא דטרמיניסטי, כך שבצורה הסריאלית שלו יש גם אלמנט ראנדומיזציה. רוצים בנייה אוניברסלית שתהיה ל-n חוטים, wait-free ולינארזבילית.

<u>פתרון</u>:

הפתרון הדטרמיניסטי לא מתאים כאן, כיוון שהפעלת אותה היסטוריה על אותו אובייקט על ידי שני חוטים לא דטרמיניסטים לא יתנו בהכרח את אותו מצב פלט. הפתרון יהיה כך: נקח את אותו אובייקט דטרמיניסטי ונקח את ההיסטוריה הדטרמיניסטית שלו. לבסוף נצטרך לעשות קונצנזוס על התוצאה. ברגע שהחוטים מגיעים לתוצאה אחרי שכולם הפעילו את אותו רצף הפעלות (באופן לא דטרמיניסטי), כולם יתחרו בקונצנזוס על התוצאה הפרטית של כל חוט.

שאלה 4:

ב-lock free הרגיל משתמשים ב-distributed head. אי אפשר להשתמש ב-head אחד מרכזי כי כל הזמן יידרס. לכן מחזיקים מערך בו לכל חוט יש פוויינטר לאיפה הוא חושב שה-head נמצא. כיצד נמנעים מה-ditributed head?

תשובה:

ה-head לא באמת נדרש, הוא רק חוסך פעולות. ב-wait free הוא נדרש, אך ב-lock free לא: רצים על כל ה-nodes עד שמגיעים ל-null או עם seq==0, ואותו מחזירים.

שאלה 5:

להציע פרוטוקול אוניברסלי שעובד עם bounded memory.

פתרון:

להוסיף GC: אם כולם מצביעים על אותו node, לנסות להעיף את ה-nodes הקודמים.

פרוטוקול MESI:

פרוטוקול cache coherency. שימוש ב-cache נעשה בכדי לחסוך זמן גישה לזיכרון, אך יש צורך בסנכרון זיכרון פרטי של כל הליבות (החוטים). לפרוטוקול 4 שלבים.

- כאשר חוט הולך למשוך מהזכרון, כאשר הוא חוזר, הוא אמור להודיע לכולם אם זה משתנה משותף. אם זה משתנה אקסקלוסיבי שלו, כלומר שאף אחד לא משתמש בו, הוא יעבוד עם ה-cache שלו. במצב זה הרגיסטר במצב E באותו חוט.
- כאשר החוט כותב ומשנה אותו, הוא ישתנה למצב M – modified. כאשר חוט אחר צריך אותו, הוא ימשוך אותו לאחר השינוי ובשני העותקים של שני החוטים המצב ישתנה ל-S – shared.
- במצב shared כל מעבד יודיע לאחרים על כל שינוי – כי זהו ערך משותף. כאשר חוט רוצה לעדכן, הוא מודיע לכולם שהוא שינה – הופך אצלו את הערך ל-E ואצל כל האחרים ל-I – invalid), והוא יכול להמשיך לקרוא ולעדכן לוקאלית.

אם נגמר לחוט כלשהו זיכרון ב-cache? יתפנה מקום מה-cahce לטובת ערך אחר מהזיכרון.

שאלה 1:

למה צריך להבדיל בין E ל-M?

תשובה:

מצב M אומר שערך שונה אך אחרים לא יכולים לראות אותו עדיין. מצב E אומר שאצל אותו חוט יש ערך מעודכן, ואף אחד אחר לא יכול לראות אותו. כשיש את שניהם, כאשר הערך ב-E, ידוע שהערך של הליבה הוא הערך בזיכרון. אם הערך ב-M, אז ידוע שהיינו ב-E קודם. ב-E הכתיבה היא לוקאלית ומאוד מהירה. אם אין הרבה ערכים משותפים, העבודה של אותו חוט תהיה הרבה מאוד עם ה-cache ולכן מהיר.

שאלה 2:

למה צריך E ו-S ואי אפשר להסתפק ב-E?

תשובה:

מצב S הוא שכמה חוטים קוראים את אותו ערך. אם לא היה אותו, אז כל הזמן חוט שקורא תופס E, והאחרים I והיה race על ה-data.