תכנות מרובה ליבות – סיכומים למבחן

סמסטר ב' תש"ע, 2010 (פרופ' ניר שביט)

פרק 1: Introduction:

מושגים:

- Safety: תכונת safety מגדירה מצבים שאסור שיקרו, למשל שני חוטים ב-CS.
- Liveness: תכונת liveness מגדירה מצבים רצויים שיקרו בסופו של דבר, למשל חוט כלשהו תמיד יצליח להכנס ל-CS (deadlock free).
- Deadlock free: מישהו תמיד מצליח להיכנס ל-CS, המערכת מתקדמת. יתכנו חוטים מורעבים, אך לפחות אחד מתקדם.
- Starvation free: כל חוט שמבקש להיכנס ל-CS יצליח בסופו של דבר.

חוק Amdahl: עבור n מספר מעבדים, p החלק מהתוכנית שניתן לבצע באופן מקבילי, ה-speedup ביחס לריצה על מעבד יחיד הוא:

$$s = \frac{1}{1-p+\frac{p}{n}} \quad 1-p: sequential\ part, \quad p: parallel\ part$$

פרק 2: Mutual Exclusion:

סימונים:

- Event: מאורע שמתרחש באופן אטומי ב-thread. למשל, קריאת משתנה, קריאה למתודה, חזרה ממתודה וכו'. חזרה k-ית של מאורע: a_i^k.
- Thread: רצף events: $a_0, \dots, a_k$ סדור כך ש-$a_0 \to a_1$ מסמן שאירוע a_0 מתרחש לפני אירוע a_1 בחוט. ניתן להתייחס לחוט כמכונת מצבים שהאירועים הם מעברים.
- Interval: זמן בין שני מאורעות מוגדר ע"י (a_i, a_j) כאשר a_i, a_j מאורעות. חזרה k-ית של אינטרוול: A_i^k.

Precedence: נאמר כי שני אינטרוולים A_0, B_0 מקיימים $A_0 \to B_0$ כאשר מאורע הסיום של A_0 קודם למאורע ההתחלה של B_0. מתקיים: $A \to B \bigwedge !B \to A$ (כלומר יש ביניהם חפיפה ולכן אין אחד קודם לשני). $\forall A.!A \to A, \quad A \to B \Rightarrow !B \to A, \quad A \to B \bigwedge B \to C \Rightarrow A \to C$ (אירפלקסיבי, אנטיסימטרי וטרנזיטיבי). יתכן $!A \to B \bigwedge !B \to A$

Mutex - הגדרה פורמלית: אם CS_i^n הוא execution של הקטע הקריטי ה-n-י של חוט i ו-CS_j^m בהתאמה, אז: $CS_i^n \to CS_j^m \bigvee CS_j^n \to CS_i^m$, כלומר אין ביניהם חפיפה.

הוכחת Mutex: מניחים בשלילה ששני חוטים נמצאים ב-CS, הולכים אחורה ברצף הפעולות שהביאו שני חוטים להיות יחד ב-CS עד שמגיעים לסתירה.

אם מגיעים למעגל בהנחות סדר האינטרוולים, אזי יתכן deadlock (כמו LockOne במצגת 2, שקופית 57).

- Fairness – first come first served: הגינות מוגדרת כך שאם $D_A^k \to D_B^j$, כאשר D הוא חלקו הראשון בעל מספר צעדים סופי של קריאת lock, אז $CS_A^k \to CS_B^j$. כלומר, סדר כניסת חוטים ל-CS מוגדר לפי סדר ביצועם את החלק הראשון (הסופי) במתודת lock.

אלגוריתמים חשובים מהפרק:

- אלגוריתם Peterson לשני חוטים: שימוש ב-flag ו-victim. מקיים deadlock free ואף starvation free.
- אלגוריתם Filter ל-n חוטים (הרחבת Peterson): שימוש ברמות (level[i]) מחזיק את הרמה לחוט ה-i) ו-victims (victim[i] מחזיק את ה-victim לרמה ה-i). בכל רמה L יש לכל היותר $n-L$ חוטים, הרמה האחרונה היא ה-CS (הוכחה באינדוקציה). אותן תכונות כמו ל-Peterson.
- אלגוריתם Bakery: שימוש במערך דגלים ו-labels. כל מי שרוצה להיכנס מסמן את הדגל שלו, ולוקח את ה-label המקסימלי כרגע + 1, וממתין כל עוד יש מישהו עם דגל מורם ו-label קטנה משלו (תור). אם ה-label אותו דבר לשני חוטים (משום מה), ההחלטה תהיה לפי ה-threadId.

משפט: פתרון deadlock-free mutex עבור n חוטים דורש לפחות n רגיסטרי MRSW או n רגיסטרי MRMW.

פרק 3: Concurrent Objects:

- **Linearizability**: אובייקט הוא לינאריזבילי אם בכל מתודה ישנה נקודה בין קריאתה לחזרתה בה השינוי שמבצעת קורה מיידית. נקודות לינאריזציה תלויות ב-execution.

היסטוריה לתיאור execution:

מחלקים כל מתודה לאינבוקציה וחזרה, כאשר:

- invocation: <Thread> <Object>.<Call>(<Args>), למשל $A\ q.enq(3)$. אם אין לה תגובה – תהיה pending.
- response: <Thread> <Object>:<Return value / exception>, למשל $A\ q: void$

Precedence: קדימות תהיה בין שתי קריאות למתודות אם ה-response של אחת קורה לפני ה-invocation של השניה.

היסטוריה: סדרת קריאות וחזרות של מתודות ע"י חוטים ואובייקטים.

היסטוריה סדרתית: היסטוריה בה קריאות וחזרות של מתודות שונות לא חופפות (כלומר לאחר כל קריאה תבוא מיד החזרה שלה).

הגדרת projection של חוט או אובייקט: המאורעות בהיסטוריה בהם מופיע אותו חוט / אובייקט. סימון: H|A, H|q.

Complete history: היסטוריה שהוסרו ממנה קריאות pending (כלומר קריאות ללא response).

Well formed history: היסטוריה בה ה-projection של כל חוט היא היסטוריה סדרתית.

Equivalent histories: היסטוריות מקבילות הן כאלו ש-projection של כל חוט זהה בין שתי ההיסטוריות (יתכן סדר שונה בין החוטים), כלומר היסטוריות H,G אקוויוולנטיות אם $\forall thread\ A: H|A = G|A$.

היסטוריה חוקית: היסטוריה תהיה חוקית אם לכל אובייקט x מתקיים כי $H|x$ היא היסטוריה שנמצאת ב-sequential spec של x.

בחזרה לסימוני פרק 2, כל שתי קריאות מתודה שיש ביניהן קדימות יסומנו $m_0 \rightarrow_H m_1$. בהיסטוריה סדרתית יחס $\rightarrow_H$ מגדיר לנו סדר מלא ובהיסטוריה מקבילית רק סדר חלקי. לינאריזביליות באה להפוך סדר חלקי זה לסדר מלא ע"י הגדרת "קדימות" בין שתי מתודות חופפות. מכאן:

Linearizability:

הגדרה נוספת: היסטוריה H תהיה לינארזבילית אם ניתן להרחיבה להיסטוריה G ע"י:

- הוספת 0 או יותר responses ל-pending invocations (כאשר הפעולה קרתה בפועל, רוצים "לסגור" אותה).
- הסרת pending invocations אחרות.

כך שמתקיימים שני תנאים:

- היסטוריה G אקוויוולנטית להיסטוריה סדרתית **חוקית** S. היסטוריה S מגדירה סדר מלא, ועליו להכיל את הסדר החלקי שמגדירה G: $\rightarrow_G \subset \rightarrow_S$.
- יחס ה-input/output של המתודות צריך להישמר זהה בין G ל-S.

בחירת S: לבחור נקודות לינאריזציה בתוך האינטרוולים של כל מתודה (בין האינבוקציה לחזרה שלה בלבד), ואם יש חפיפה ניתן לבחור את הסדר בין שתי המתודות החופפות איך שרוצים (שיהיה חוקי).

תכונות לינארזביליות:

- **Non-blocking theorem**: אם יש קריאה pending בהיסטוריה H ומשרשרים לה response לאותה קריאה, ההיסטוריה תהיה לינארזבילית. ב-linearizability לעולם לא צריך לבצע roll-back למתודות כאלה pending, כי לא יתכן שהוספת response תפגע בנכונות. ההוכחה מתבססת על כך שלמתודה pending אפשר לשים response בסוף, וכך לקבוע ש-takes effect אחרונה, ומקבלים היסטוריה סדרתית אקוויוולנטית חוקית.
- **Composability theorem**: היסטוריה H היא לינארזבילית אמ"מ לכל אובייקט x מתקיים $H|x$ לינארזבילית.

Sequential Consistency:

היסטוריה H תהיה SC אם ניתן להרחיבה להיסטוריה G ע"י:

- הוספת 0 או יותר responses ל-pending invocations (כאשר הפעולה קרתה בפועל, רוצים "לסגור" אותה).
- הסרת pending invocations אחרות.

כך שהיסטוריה G אקוויוולנטית להיסטוריה סדרתית **חוקית** S. **אין דרישה ל:** $\rightarrow_G \subset \rightarrow_S$. כלומר אין הכרח לשמר real-time order.

- לא ניתן לסדר מחדש שתי מתודות שנקראו ע"י אותו חוט.
- כן ניתן לסדר מחדש שתי מתודות גם אם אינן חופפות שנקראו ע"י שני חוטים שונים.

תכונות SC:

- לא מקיימת Composability, כלומר אם $H|x$ היא SC לכל אובייקט x זה לא אומר ש-H היא SC.

Quiescent Consistency:

- בכל נקודה בה אובייקט נעשה quiescent, כלומר אין pending methods על האובייקט באותה נקודה, אז הריצה עד כה על האובייקט שקולה לריצה סדרתית כלשהי עד כה.
- דוגמא: אם על תור נעשה enq(x), enq(y) במקביל, ולאחר סיומם נעשה enq(z), לא ידוע הסדר בין שתי ההכנסות הראשונות אך ידוע בודאות שהן קרו לפני ההכנסה השלישית.

תכונות QC: כן מקיימת Composability – הרכבת ריצות שני אובייקטים QC היא QC.

הערות:

- QC ו-SC לא מקיימים הכלה אחד בשני: קיימת ריצה QC שלא משמרת סדר ריצה ואינה SC, וריצה SC לא מושפעת משלבי quiescent ולא בהכרח תהיה QC.

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free

פרק 4: Foundations of Shared Memory:

סוגי רגיסטרים:

הבדלים בכמה חוטים יכולים לבצע פעולות: SRSW, MRSW, MRMW.

Safe register:

- MRSW
- קריאות שאינן חופפות לכתיבה מחזירות תמיד את הערך האחרון שנכתב ע"י פעולת הכתיבה האחרונה.
- קריאות חופפות לכתיבה מחזירות ערך **בטווח הערכים של הרגיסטר** (רגיסטר m-valued יחזיר תמיד ערכים בטווח $0, \dots, m-1$).

Regular register:

- MRSW
- קריאות שאינן חופפות לכתיבה – מתנהג כמו safe.
- קריאות חופפות לכתיבה – יוחזר **או הערך החדש (הנכתב עתה) או הישן**.
- כל רגיסטר לינארזבילי הוא רגולרי, אך לא להיפך!

Atomic register:

- רגיסטר לינארזבילי לרגיסטר sequential safe. אטומיות = לינארזביליות במונחי רגיסטרים.

Wait free: אובייקט יהיה ממומש באופן wait-free אם כל קריאה למתודה שלו מסתיימת במספר סופי של צעדים.

היררכית בניות:

סוג רגיסטר	בנייה באמצעות רגיסטרים מסוג השלב הקודם
SRSW safe bool (or multi valued)	
MRSW safe bool (or multi valued)	מחזיקים מערך של SRSW, פעולת הכתיבה היא לכתוב לכל אחד ממקומות המערך, פעולת הקריאה היא לקרוא מהמקום במערך המתאים לחוט הקורא. שימור התכונות הנדרשות ברור. כנ"ל ל-multi valued.
MRSW regular bool (לא יעבוד עבור multi valued)	• עבור bool, אם נכתב ערך שונה מהקודם, התנהגות safe זהה להתנהגות regular כי טווח הערכים של הרגיסטר {0,1} זהה לקבוצת הערך הישן והחדש של הרגיסטר. • אם נעשה ניסיון כתיבה של ערך שזהה לערך הנוכחי של הרגיסטר, כדי למנוע למשל שכתיבת 0 ל-0

	תחזיר בטעות 1, הכותב יזכור את הערך הישן, ואם הוא זהה הוא יוותר על פעולת הכתיבה עצמה.
MRSW regular multi valued	מחזיקים מערך של MRW-RB לייצוג אונארי של הערך (1 באינדקס שמייצג הערך, 0 בשאר). • כתיבה: כותבים 1 למקום הרצוי, ומאפסים את כל הביטים באינדקסים נמוכים ממנו. רק אחרי כן לאפס את הביט של הערך הקודם. • קריאה: רצים על המערך מתחילתו, כשנתקלים ב-1 הראשון מחזירים את האינדקס אליו הגיעו.
SRSW atomic	יש בעיה אם הקורא היחיד מבצע כמה קריאות חופפות לכתיבה כלשהי. הפתרון הוא שהכותב יכתוב גם ערך וגם timestamp, והקורא יזכור כל קריאה את הערך האחרון והחתימה שקרא. אם קרא חתימה מאוחרת יותר ממה שיש לו – יחזיר את הערך החדש שקרא. אחרת יחזיר את הישן. כך כל קריאה ראשונה מכלל הקריאות החופפות לכתיבה מסויימת אחרי נקודת הלינאריזציה קובעת את ערך הקריאות שאחריה.
MRSW atomic	כעקרון מחזיקים ערך+חתימה לכל חוט קורא, אבל אם הכותב איטי, יתכן שיספיק לכתוב לקורא אחד שיקרא ערך חדש, וטרם יספיק לכתוב לקורא שני שיבוא אחרי הקורא הראשון אך בכל זאת יקרא ישן. • מחזיקים עמודה לכל חוט – כותב וקוראים. מספר שורות כמספר הקוראים. • הכותב כותב לעמודה שלו, כל שורה שמגיע = הודיע ישירות לקורא של אותה שורה. • כל קורא שמבצע קריאה, קורא את השורה שלו בכל העמודות (כל החוטים) ולוקח את הערך עם החתימה המאוחרת ביותר. לאחר מכן כותב את הערך לעמודה שלו (מודיע לכל הקוראים). כך פעולת קריאה לא הסתיימה עד שהקורא הודיע לכולם, ואז בהכרח קורא מאוחר יותר יקרא בשורה שלו את הערך האחרון, גם אם הכותב המקורי איטי וטרם הגיע אליו.
MRMW atomic	בדומה לאלגוריתם של Bakery, כל כותב קורא את החתימות של כל שאר הכותבים, וכותב את ערכו עם חתימה גדולה ב-1 מהמקסימום שראה. בקריאה הקורא עובר על כל החתימות ובוחר את הערך עם החתימה הגבוהה ביותר. אין כאן בעיה ששני קוראים לא חופפים יקראו ערכים כך שהאחרון יקרא ערך ישן יותר מהראשון, כי חתימה מאוחרת יותר בטוח נכתבה כאשר הקורא השני התחיל (הראשון הרי קרא אותה).
Atomic snapshot מערך של MRMW atomic (נתחיל ממערך של MRSW)	scan – Snapshot של תמונת הזיכרון בזמן שיש updates במקביל. המנשק מספק שתי פעולות: scan שמחזיר את המערך ו-update(int v) שמעדכן את המקום במערך של החוט הקורא לערך v. • Update: כל חוט כותב עם חתימה, ומעלה את החתימה בכל כתיבה. • Scan: דוגם פעמיים את המערך. אם החתימות נשארו זהות, מחזיר (בין שתי הדגימות – !snapshot).

Wait-free snapshot:

- Update: בתחילת הפעולה מבצעת scan, ואז מעדכנת את המקום שלה שיכיל: חתימה חדשה, ערך חדש, תוצאת ה-scan לפני עדכון הערך.
- Scan: אם אחרי שני ניסיונות collect רואים שערך כלשהו זז פעמיים, אזי אותו חוט בעל הערך (שב-update שלו קרא ל-scan) סיים update בזמן שה-scan הנוכחי עוד בתהליך. לכן בטוח להחזיר את ה-scan של אותו ערך. בודקים האם חוט זז פעמיים לכל חוט קיים.

הבחנה: כל scan יכול היה להיות מופרע בדרך ע"י update של חוט אחר, אך לכל היותר זה היה קורה $n-1$ פעמים כי חוט לא יכול להפריע פעמיים באותו מהלך.

פרק 5: The Relative Power Of Synchronization Operations:

קונצנזוס:

בעיית החלטה של ריצת כמה חוטים כך שכולם יחליטו על אותו ערך, והוא יהיה ההצעה (קלט) של אחד החוטים הרצים.

משפט:

לא קיים מימוש wait-free לבעיית קונצנזוס ל-n חוטים המתבססת על רגיסטרי RW (מכאן שאם בעיה היא חשיבה במובן Turing, היא לא בהכרח חשיבה במובן אסינכרוני).

הוכחה:

ב-wait free computation ניתן לשרטט את עץ המצבים, כאשר כל תנועה קדימה היא ע"י חוט מסויים המבצע קריאה / כתיבה לרגיסטר. בהוכחה מסתכלים על עץ עבור שני חוטים, כאשר:

- צומת bivalent: צומת שניתן להגיע ממנו לשתי תוצאות אפשריות (0 או 1)
- צומת univalent: צומת ממנו ניתן להגיע לתוצאה יחידה. ניתן לסמן כ- x-valent עבור צומת ממנו ניתן להגיע רק לערך x.
- צומת critical: צומת ביוולנטי בו הצעד הבא של אחד החוטים לוקח ל-1 והצעד הבא של השני לוקח ל-0.

<u>טענה</u>: קיים צומת התחלתי ביוולנטי.

<u>הוכחה</u>: מספיק שיש ריצת סולו של חוט אחד אז מוחלט ערכו. מכאן שחייב להיות צומת ביוולנטי.

תמיד ניתן להגיע לצומת קריטי: יורדים עם חוט A עד שמגיעים לצומת בו הצעד הבא הוא 0-valent. אם זה לא צומת קריטי, ממשיכים על B עד שמגיעים לצומת בו הצעד הבא של B הוא 1-valent. כיוון שהריצה היא wait-free, תוך מספר סופי של צעדים נסיים, ולכן בדרך חייב להיות צומת קריטי.

ההוכחה ממשיכה בכך שמראים על טבלת האינטראקציות האפשריות בין שני החוטים שכל פעולת כתיבה/קריאה לרגיסטר x/y ע"י שני החוטים מובילה לסתירה, כלומר אין פתרון קונצנזוס. מתחילים מצומת קריטי:

- אם A ממשיך, בסופו של דבר מגיע ל-0. אם B קורא (עוברים למצב 1-valent) ואז A ממשיך, בסוף מגיע ל-1. אבל A לא יכול להבחין בין המצב בו B לא קרא למצב בו כן קרא, ובכל זאת יש שתי תוצאות שונות – סתירה.
- אם A כותב ל-x ו-B כותב ל-y: לא ניתן להבחין בין המצב $A \to x, B \to y$ למצב 0-valent שהוא $B \to y, A \to x$, שהוא 1-valent, סתירה.
- אם A ו-B כותבים ל-x: לא ניתן להבחין בין $A \to x$ ובין 0-valent שהוא $B \to x, A \to x$, שהוא 1-valent, גם כאן סתירה.

<u>מימוש קונצנזוס לשני חוטים ע"י Two-dequeuer wait free Queue</u>:

מאתחלים את התור להחזיק כדור אדום ראשון וכדור שחור שני. מי שמוציא את הכדור האדום מחזיר את הערך שלו (הראשון שהגיע), מי שמוציא את השחור מחזיר את הערך של האחר. נכונות:

- אם אחד מוציא את האדום, השני בהכרח מוציא את השחור.
- אם השני מוציא את השחור, זה בהכרח אחרי שהראשון הוציא את האדום, וזה אחרי שהוא כתב את ערכו למערך ה-proposed, מכאן שערכו שם.

מכאן הוכחה שלא ניתן לבנות כסזה תור מרגיסטרים אטומיים, אחרת ניתן היה לפתור קונצנזוס ל-2 ע"י רגיסטרים אטומיים.

<u>מספר קונצנזוס</u>:

מספר קונצנזוס n של אובייקט x הוא מספר החוטים עבורם ניתן לפתור קונצנזוס באמצעות אובייקטי x ורגיסטרי RW אטומיים, ולא ניתן ל-n+1.

משפט:

- אם ניתן לממש X מ-Y ומספר הקונצנזוס של X הוא c, אזי מספר הקונצנזוס של Y הוא לפחות c. כלומר במימוש עלולים "לאבד" דרגות קונצנזוס, אך לא להרוויח.
- הפוך: אם מספר הקונצנזוס של X הוא c ומספר הקונצנזוס של Y הוא d<c, אז לא ניתן לממש X באמצעות Y.

(יש למשפט זה סייגים).

Multiple Assignment theorem:

רגיסטרים אטומיים לא יכולים לממש השמה מרובה באופן wait-free.

הוכחה: מתבסס על כך שאם ניתן לכתוב ל-$\frac{2}{3}$ מאיברי מערך, פותרים קונצנזוס לשני חוטים – שכמובן אי אפשר.

נניח מערך בו A כותב אטומית ל-0,1 ו-B כותב אטומית ל-1,2, וכולם יכולים לקרוא הכל. פעולת ה-decide: תחילה החוט כותב למקומות שלו במערך (אחד מהמקומות משותף עם האחר). אז החוט קורא את המשותף וזה של האחר.

- אם האחר ריק, אז הוא טרם כתב, כלומר אני ראשון – מחזיר את הערך של עצמי. אם האחר שווה למשותף, השני כתב רק אחרי (דרס את המשותף שאני כתבתי), גם כן הגעתי ראשון – מחזיר את הערך של עצמי.
- אם לא, השני ניצח – מחזיר את הערך שלו.

<u>אובייקטי Read-Modify-Write</u>:

מתודת RMW עם פונקציה mumble(x) היא <u>לא טרוויאלית</u> אם קיים ערך v כך ש-$v \neq mumble(v)$, וזאת כאשר הערך mumble(v) הוא זה שמושם כערך החדש במקום v. למשל $identity(x) = x$ היא טרוויאלית, $getAndIncrement(x) = x + 1$ היא לא.

משפט: כל אובייקט RMW לא טרוויאלי הוא בעל דרגת קונצנזוס לפחות 2 (מכאן שאין מימוש RMW מרגיסטרים אטומיים).

הוכחה: פשוט לבנות קונצנזוס; r אובייקט כזה מאותחל ל-v. חוט שבא להחליט מבצע getAndMumble() – אם הוחזר v, הוא ראשון ולכן מנצח ומחזיר את הערך שלו. אחרת יחזיר את הערך של האחר.

הגדרות:

- Commuting functions: f, g יהיו commute אם $f(g(v)) = g(f(v))$.
- Overwriting functions: f, g יהיו overwrite אם $f(g(v)) = f(v)$.

טענה: כל קבוצה של אובייקטי RMW ש-commute או overwrite היא בעלת מספר קונצנזוס 2 בדיוק.

טבלת דרגות קונצנזוס:

דרגה	אובייקט (פעולות אובייקט)
1	Atomic RW registers, snapshots…
2	Test-and-set (getAndSet(1)), swap (getAndSet(x)), fetch-and-inc (getAndIncrement())…
2k-2	Atomic k-assignment
∞	compareAndSet(expected, update),…

לכל מספר קונצנזוס זוגי יש אובייקט (ניתן להרחיב גם למספרים אי זוגיים).

Lock-freedom:

- בריצה אינסופית, קריאה כלשהי תסיים במספר סופי של צעדים.
- מישהו במערכת מתקדם. כל ריצה wait free היא בודאי lock-free (כולם מתקדמים, לא רק אחד לפחות).
- אם הריצה היא סופית, lock free הוא כמו wait free.

פרק 6: Universality of Consensus:

Consensus Universality Theorem:

מאובייקטי קונצנזוס ל-n חוטים ורגיסטרים אטומיים ניתן לבנות מימוש ל-n חוטים שהוא לינארזבילי ו-wait free לכל אובייקט סדרתי.

בנייה גנרית לאובייקט סדרתי:

מתודת apply המקבלת Invocation – תיאור פעולה על האובייקט והארגומנטים שלה, ומחזירה Response – פלט המתודה על הארגומנטים.

Universal concurrent object: לינארזבילי לאובייקט הגנרי הסדרתי.

הרעיון הנאיבי:

- להחזיק מערך אובייקטי קונצנזוס, בו הצעות הן המצב הבא של האובייקט (למשל חוט המציע deq() יציע למעשה את מצב התור לאחר deq(), חוט המציע enq(5) יציע את מצב התור לאחר enq(4)). כך כולם מחליטים יחד מה המצב הבא.
- **לא טוב**: כיוון שאובייקט קונצנזוס טוב לפעם אחת בלבד (כל חוט יכול להציע הצעה אחת).

מימוש מבוסס רשימה:

- מחזיקים רשימת הפעולות שנבחרו להתבצע על האובייקט (enq, deq וכו').
- בכל צומת ברשימה יש אובייקט קונצנזוס טרי כמתואר במימוש הנאיבי להחלטה על המצב הבא. כך לא צריך reuse לאובייקטי קונצנזוס.

כאן אובייקט מיוצג ע"י מצבו ההתחלתי ורשימת קריאות מתודות עם הארגומנטים שלהם לפי הסדר שנקבע.

- מתודה חדשה: מגיעים לסוף הרשימה, מוסיפים את הקריאה באופן אטומי לסוף ומחשבים את התוצאה ע"י ביצוע הפעולות ביומן הפעולות על אובייקט מקומי עם מצב התחלתי כמו של תיאור האובייקט.

החלק הראשון של apply – הכנסת ה-node:

את הבא מבצע חוט הקורא ל-apply על הצומת שהוא רוצה להציע כל עוד seq==0, כלומר כל עוד הוא לא הצליח להכניס אותו לרשימה.

- מחזיקים רשימת head כך שלכל חוט יש head במקום שלו במערך.
- החוט הקורא יוצר צומת חדש אותו רוצה להציע, מושך את ה-head עם החתימה (seq number) הגדולה ביותר ממערך ה-heads ע"י max, ולוקח אותו להיות מבחינתו ראש הרשימה (האחרון שהוכנס).
- הוא מציע על הקונצנזוס של ה-head הזה את הצומת שלו, וכתשובה קובע את ה-next של head להיות תוצאת הקונצנזוס, וקובע את ה- seq number שלו להיות זה של הקודם + 1. יתכן שדורס את ה-next, אך עם אותו ערך מהקונצנזוס, כך שזה בסדר.
- בסוף הוא מעדכן את המקום שלו במערך ה-head להיות ה-head החדש.

החלק השני של apply – חישוב ה-response:

- החוט יוצר אובייקט מקומי מסוג seqObject, עליו מבצע באופן איטרטיבי את כל הפעולות ביומן עד שמגיע לצומת שהוא הכניס.
- כאשר מגיע אליו, מחזיר את תוצאת הפעלת הפעולה בצומת זה.

נכונות: בנייה זו היא מימוש לינארזבילי לאובייקט הסדרתי כי יומן הפעולות בלתי ניתן לשינוי ונקודת הלינאריזציה של כל apply היא כאשר מוחלט בקונצנזוס על הצומת שמוצע בקריאה זו של apply.

Lock-freedom:

הצומת הבא עליו הוחלט תמיד יוכנס תוך זמן סופי מרגע ההחלטה למערך ה-head, כלומר יעודכן להיות ה-head המקסימלי (ומשם יעבור לחלק השני של apply שלוקח זמן סופי ויסתיים תוך זמן סופי):

- הצומת הקודם שלו בטוח נמצא במערך ה-head.
- כל חוט שמנסה להכניס צומת חדש, יראה את אותו צומת קודם, ומהקונצנזוס עליו יראה את הצומת הבא עליו הוחלט ויכתוב אותו (עם seq גדול ב-1 מקודמו) במערך ה-head, כך שבסופו של דבר תוך זמן סופי הוא מוכנס למערך ה-head (ויהיה תוצאת ה-max הבא).
- הדרך היחידה שחוט יתקע לעד הוא אם חוטים אחרים מקדימים אותו ומצליחים להכניס עצמם לפניו, כלומר מעדכנים את ה-head הבא כל הזמן.

הפיכת הבניה ל-wait free:

- מוסיפים מערך announce בו חוט i שלא מצליח מכריז על הצומת שרוצה להוסיף ע"י השמתו ב-Announce[i].
- אחרים מסייעים לו להיכנס, וכך כל קריאה תסתיים בסופו של דבר בזמן סופי.

השינויים בבניה:

- באיתחול מאותחלים כל המקומות ב-Announce להצביע ל-tail (ה-dummy node הראשוני).
- בתחילת apply מכריז החוט במקום שלו ב-Announce על הצומת שמנסה להכניס.
- לולאת ה-while רצה כעת על Announce[i].seq == 0 (כמו ריצה על prefer.seq == 0 קודם).
- בתוך הלולאה בוחרים את help – הצומת לו עוזרים ע"י Announce[before.seq+1%n] שמתקדם סדרתית. אם ה-seq שלו 0, כלומר טרם הוכנס, באמת עוזרים לו. אחרת אני עוזר לעצמי – מנסה להכניס את הצומת שלו.

הנקודה כאן היא שאחרי שצומת A מכריז על צורך בעזרה, לא יותר מ-n קריאות אחרות יכולות להקרא ולהסתיים מבלי לעזור ל-A בדרך (מישהו בטוח יתפוס אותו בין n הקריאות הללו ויעזור ל-poor bastard).

פרק 6 (תוספת): Multiprocessor Architecture Basics:

סוגי ארכיטקטורות:

- SMP (Symmetric multiprocessor): תקשורת בין המעבדים לזיכרון נעשית מעל bus. טוב למערכות בקנה מידה קטן.
- NUMA (non-uniform memory access): לכל מעסד יש פיסת זיכרון משלו, כך שגישה לזיכרון שלו מהירה יותר מ-SMP אך גישה לזיכרון של מעבד אחר איטית יותר.
- CC-NUMA (cache-coherent...): מכונות NUMA עם caches.

סוגי interconnect:

- BUS: כמו מיני Ethernet העובדת בשיטת broadcast, לתקשורת בין מעבדים לזיכרון ובין מעבד למעבד.
- Network: מערכות גדולות משתמשות בתקשורת point-to-point מעל רשת, כמו LAN בין מעבדים.

ה-interconnect הוא משאב סופי, ולכן יש לשאוף להנמכת traffic באלגוריתמים בשביל יעילות.

Cache:

כדי לחסוך גישות לזיכרון מחזיקים זיכרון קרוב למעבד ומהיר, כך שכל קריאה וכתיבה מתבצעת עליו. כל עוד ערך לא משתנה, הוא נקרא מה-cache.

Cache-line: בלוק מילים שכנות, היחידה המינימלית שנקראת / נכתבת בעדכון cache מהזיכרון.

רמות cache: בפרקטיקה ישנן כמה רמות cache – L1, L2 כך ש-L1 קרוב יותר למעבד (ומהיר יותר). ככל שהרמה רחוקה יותר כך היא מכילה יותר מקום אך התקשורת עמה איטית יותר. L1: לרוב יושבת על אותו שבב של המעבד, גישה עולה 1-2 cycles. L2: עשרות cycles, cache line גדול יותר.

מדיניות החלפה: כאשר ה-cache מלא לרוב משתמשים במדיניות LRU להחליט איזה מקום יפונה לטובת החדש.

סוגי caches:

- Fully associative: כל line יכול להיות בכל מקום ב-cache. יתרון: גמישות בהחלפה; חיסרון: קשה למצוא lines.
- Direct mapped cache: כל line מתמפה בדיוק למקום אחד. יתרון: קל למצוא. חסרון: חייבים להחליף שורה ידועה מראש, פחות LRU.
- K-way set associative cache: כל line יכול להתמפות ל-k מקומות. שילוב של השניים לעיל.

Cache coherence:

סינכרון מידע בין מעבדים.

פרוטוקול MESI: פרוטוקול בו cache line יכול להיות באחד מ-4 מצבים:

- Modified: ה-line עודכן ב-cache אך טרם עודכן בזיכרון, לכן צריך להכתב לזיכרון לפני שמישהו יכול להשתמש בו.
- Exclusive: ה-line הזה נמצא רק אצלי ב-cache, כך שאם אעדכן אשנה אותו לא צריך להודיע לאף אחד.
- Shared: ה-line אצלי לא שונה מאז העדכון האחרון, והוא משותף. אם יהיה שינוי, צריך להודיע לאחרים לבצע invalidate.
- Invalid: ה-line חסר משמעות, ערכו מיושן (אולי כי מעבד אחר שינה אותו).

סיווג נוסף:

- Write through: כל שינוי מיד משודר, סינכרון מיידי אך יוצר עומס על ה-bus.
- Write back: צוברים שינויים ב-line. הוא נשלח לעדכון רק כשמפנים את ה-line או אם מעבד אחר רוצה אותו (ואז נשלח invalid).

Volatile:

- כאשר משתנה מוכרז volatile ב-Java, זה מונע מפעולות להיות מסודרות מחדש ע"י הקומפיילר.
- ה-write buffer תמיד נשפך לזיכרון לפני שמורשית כתיבה.

פרק 7: Spin Locks and Contention:

גורמים המשפיעים על תפקוד:

- Contention on memory: לא ניתן לגשת לאותו מקום בזיכרון ע"י כמה מעבדים, ואם ינסו יכנסו לתור המתנה.
- Contention on communication medium: צורך בהמתנה אם רוצים ליצור קשר בו"ז או לאותו מעבד.
- Communication latency: הזמן שלוקח למעבד לתקשר עם הזיכרון או מעבד אחר.

סוגי spin locks:

Basic spin lock:

- חוט אחד תופס את המנעול, שאר החוטים מסתובבים על המנעול עד שמשתחרר ואחד מהם תופס אותו.
- Contention: קורה כאשר הרבה מנסים לתפוס את המנעול. High / low – בהתאם לכמה מנסים לתפוס אותו.

TAS (test and set) lock:

- מאותחל ל-false. תפיסה: כל עוד מוחזר true, תחליף את ערך המנעול ל-true (כאשר מוחזר הערך הנוכחי). כאשר מוחזר false – אזי המנעול היה משוחרר וזה שכתב אליו עכשיו true תפס אותו. שחרור: פשוט לכתוב אליו false.
- עצם השימוש באובייקט RMW (AtomicBoolean) מאפשר לנו להפטר מחסם ה-$\Omega(n)$ ב-Peterson, Bakery ולהשתמש ב-$O(1)$ זיכרון בלבד גם עבור n חוטים.

TTAS (test and test and set) lock:

- תפיסה: קוראים את ערך המנעול פעם אחת ומסתובבים עליו בקריאה בלבד. כאשר המנעול "נראה" פנוי והקריאה מחזירה false, רק אז עושים TAS לתפיסת המנעול. אם מפסיד – חוזרים לשלב הקודם.

הסבר לביצועים:

ביצועי TAS ו-TTAS אינם אידיאלים, ו-TTAS טוב יותר מ-TAS בגלל ביצועי חומרה: גישות ל-cache מקומי של מעבד לוקחות 1-2 cycles ולזיכרון 50-100. ככל שיש יותר cache-hits כך הביצועים טובים יותר. זאת כיוון שעובדים מעל shared BUS. לכן ככל שיש יותר קריאות מעל ה-BUS כך הביצועים נמוכים יותר.

- TAS: כל קריאת TAS גם גורמת invalidation לערך בכולם, גם אם הערך לא השתנה, וגם שולחת בקשת קריאה ל-BUS. עומס רב מאוד. אפילו החוט שרוצה לשחרר מעוכב ע"י עומס על ה-BUS בניסיון כתיבת false לצורך שחרור.
- TTAS: אחרי משיכה ראשונה ל-cache החוטים מסתובבים על עותק מקומי ולא מעמיסים על ה-BUS, ואלא מעכבים את המשחרר כשירצה לשחרר. כאשר המנעול משתחרר ויש invalidation, יש storm על ה-BUS. בכל מקרה פחות עמוס מ-TAS.

Write back caches:

- כתיבה נעשית רק כשצריך את המקום ב-cache או כשמעבד אחר רוצה את הערך.
- כשמעבד רוצה לשנות ערך ב-cache, הוא שולח invalidate לכל שאר המעבדים, וכשסיים יכול לשנות כאוות נפשו. כשיתבקש, יכתוב את הערכים ה-dirty מה-cache שלו לאן שצריך.
- מצבי cache entry: invalid – כתובים שטויות; valid – כתוב ערך שניתן להשתמש בו אך אסור לשנותו; dirty – הערך שונה, צריך לשלוח למעבדים אחרים אם צריכים ולכתוב לזיכרון אם רוצים לפנות את ה-cache לערכים אחרים.
- כתיבה לזיכרון תהיה רק בפינוי cache, כל מעבד שירצה יקבל את הערך הנוכחי ממעבד שמחזיק בערך המעודכן (תקשורת בין מעבדים מהירה יחסית לתקשורת מול זיכרון).

Write through: לא יעיל כל כך, פחות נפוץ.

Exponential backoff lock:

- כל ניסיון תפיסת מנעול שנכשל גורם להמתנה. בכל ניסיון נוסף שנכשל זמן ההמתנה מוכפל פי שניים, עד מקסימום קבוע.
- מוריד contention על ה-BUS. שאר ההתנהגות כמו TTAS.
- יותר טוב מ-TTAS אך תלוי פרמטרים, ולכן הגדרה יחידה לא פורטבילית בין מכונות שונות (צריך להתנסות על כל מכונה כדי לכייל את הפרמ').

מנעולי תור:

כל אחד ממתין בתור עד שקודמו משחרר את המנעול. מנעול זה גם הוגן (first come first served) וגם חוסך invalidations כי כל אחד מסתובב על ערך משלו – האם קודמו שחרר.

Anderson Queue lock:

- מחזיקים מערך של בוליאנים שכולם F חוץ מהמקום הראשון שהוא T, ומצביע אטומי עליו.
- תפיסה: מבצעים getAndIncrement על המצביע. אם המערך במקום הערך שהתקבל (מודולו גודל המערך) T, אזי תפסנו את המנעול. אם לא, מסתובבים על הערך הזה – מקומית כמובן. כשתופסים את המנעול מיד משנים את ערך המקום במערך ל-F, אתחולו לקראת הסיבוב הבא.
- שחרור: משימים במערך במקום הבא אחרי ה-slot שאני תפסתי קודם את הערך T, כהודעה לבא בתור שהוא יכול להיכנס.
- בניגוד לקודמים, בגלל עקרון ה-FCFS אלגוריתם זה הוא starvation free.

בעית false sharing: מקומות עליהם מסתובבים ממתינים מקבלים invalidation כשהמחזיק הנוכחי משחרר את מנעול הבא אחריו כיוון שחולקים את אותו cache line, גם אם ערכם לא השתנה.

פתרון padding: מרפדים את המקומות בין הביטים כדי שכל אחד מהם יהיה ב-cache line נפרד.

חסרונות: צריך לדעת את מספר החוטים, ותופס המון זיכרון – $O(N)$ פר מנעול, כאשר N הוא חסם מספר החוטים שיכולים לרוץ במקביל (cache line שלם לכל חוט). כמו כן האלגוריתם לא יעבוד טוב על ארכיטקטורה uncached. עבור L מנעולים: $O(LN)$ מקום (ב-wc שכל חוט ניגש למנעול אחד).

CLH lock:

- מחזיקים מצביע אטומי tail המצביע ל-QNode ראשוני עם ערך false. ערך זה אומר שהמנעול חופשי. ערך true אומר שמישהו ב-CS או ממתין למנעול לפניך.
- תפיסה: חוט יוצר QNode עם הערך true ומבצע getAndSet על tail כך ש-tail יבציע על ה-QNode שלו והוא יצביע על מה ש-tail הסתכל עליו, הוא הקודם. כל עוד הקודם true, תסתובב עליו – מקומית ב-cache של החוט. ברגע שנהיה false – אפשר להיכנס לקטע הקריטי.
- שחרור: החוט משנה את ערך ה-QNode המקורי שיצר (זה שמי שאחריו ממתין עליו) ל-false.

יעילות זיכרון:

- כאשר חוט תופס את המנעול, הוא יכול לעשות שימוש ב-QNode של קודמו ששחרר המנעול, כי הוא לא צריך אותו יותר, לנעילות עתידיות.

- עבור L מנעולים: O(L+N) מנעולים כאשר כל חוט ניגש למנעול אחד.

חסרונות: למרות שהוא טוב מבחינת זיכרון ו-invalidation נעשית לממתין הבא למנעול בלבד, הוא לא עובד טוב ל-uncached NUMA. ב-NUMA ללא cache, יכול חוט להסתובב על זיכרון של חוט אחר (קודמו) שעלול להיות מרוחק ויקר להסתובב עליו.

יתרונות: לא צריך לדעת את מספר החוטים מראש, הוגנות.

MCS lock:

- דומה ל-CLH רק דואג שהסתובבות תהיה במקום ידוע וקרוב, טוב גם לארכיטקטורות cachless. כאן רשימת ה-QNode אמיתית, כלומר לכל QNode יש מצביע (ולא implicit list כמו ב-CLH).
- תפיסה: החוט התופס יוצר QNode מצביע ל-null עם הערך T. אחר כך מבצע getAndSet עם tail, וכשמקבל אותו, אם יש לו קודם הוא גורם למצביע שלו להצביע ל-QNode המקורי שיצר (רשימה אמיתית). עכשיו (בהנחה שיש לו קודם), הוא מסתובב על הערך ב-QNode המקורי שיצר עד שקודמו ישחרר (במקור ה-QNode שיוצר החוט שמנסה לתפוס עושה אותו עם ערך false ורק אם יש לו קודם הוא יעשה אותו true).
- שחרור: דרך המצביע מה-QNode שלי אני מגיע ל-QNode של הבא אחרי ומשחרר אותו – מכוון את ערכו ל-false. כאן הגישה הרחוקה שלי היא יחידה, לעומת הסתובבות ארוכה של הממתין לי, שהיא מקומית אצלו. לשים לב בקוד שחוט שבא לשחרר מחכה עד שהמצביע של ה-QNode שלו שונה מ-null, כלומר ממתין עד שיורשו מסיים להכין עצמו להמתנה, ורק אז משחרר אותו.

Abortable locks:

עזיבת back-off lock:

- פשוט חוזרים ממתודת ה-lock אם ההמתנה ארוכה מדי. הביטול הוא wait-free (מספר צעדים סופי וקטן). לא משפיע על מצב המנעול.

Abortable CLH lock – Timeout lock:

- כל QNode מצביע לקודם או ל-NULL ואם מצביע ל-A אזי זה אומר שהמנעול חופשי.
- תפיסה: כמו קודם החוט יוצר QNode אך עכשיו יש לו מצביע ל-NULL (קודם לא חופשי). מבצע getAndSet על tail ומסתכל האם קודמו מצביע על A. אם כן (כמו במצב ההתחלתי) – המנעול חופשי, תפס את המנעול.
- עזיבה: חוט שרוצה לעזוב מכוון את המצביע של ה-QNode שלו, עליו מסתובב יורשו, להצביע לקודמו. יורשו מזהה שהוא כבר לא NULL, אז הוא עובר להמתין על קודמו של זה שעזב (ה-QNode של זה שעזב חופשי לשימוש אח"כ).
- שחרור: אם יש לי יורש, אסמן את המצביע שלי ל-A.

Composite lock (מהתרגול):

- מחזיקים מערך בגודל קבוע עם צמתי מנעולים.
- תפיסה: מנסה לתפוס מנעול ראנדומי, אם נכשל נכנס ל-back off. כשמצליח מכניס את הצומת לתור ומסתובב על צומת קודם בתור. המתנה על צומת קודם תהיה אם הוא במצב W, עד שישתנה ל-F (חופשי לתפיסה).
- עזיבה: חוט שעוזב משנה את המצביע שלו לקודמו ואת מצבו ל-A. אם היה ממתין על זה שעזב, הוא משנה את מצבו מ-A ל-F ועובר להמתין על הקודם של זה שעזב (אליו שינה העוזב את המצביע לפני עזיבתו). אם היה ממתין להיכנס לתור שהמתין באופן ראנדומי על אותו אחד שעזב, אחרי שיורשו שינה את מצבו ל-F הוא חופשי כעת להתפס – התופס משנה את מצבו ל-W וה-pred שלו יהיה היורש של זה שעזב (זה ששינה קודם את מצב העוזב ל-A).

פרק 9: Linked Lists: Locking, Lock-Free, and Beyond...:

הדגמת שיטות שונות על Linked Lists:

- ההדגמה תהיה על מימוש מנשק Set ע"י רשימה עם פעולות add, remove, contains.
- לאיבר ברשימה יש ערך item, מפתח key ומצביע לצומת אחר – next.
- הרשימה ממויינת עם sentinel nodes להתחלתה וסופה.
- הוכחת נכונות אובייקט: משמר תכונות (invariants) ביצירה ובכל צעד בכל מתודה.

Coarse-grained synchronization:

- כל מי שבא לבצע משהו על הרשימה נועל את הרשימה כולה.
- פשוט אבל יוצר bottleneck ו-hotspot (כולם ממתינים על המנעול של הרשימה). גם אם נשתמש ב-Queue lock עדיין contention גבוה.

Fine-grained synchronization:

- לכל entry ברשימה יהיה מנעול משלה, וכך חוט שרץ על הרשימה נועל בדרך את ה-enties שמגיע אליהן ומשחרר עם עזיבתו, במקום כל הרשימה.
- מתודות העובדות על מקומות מרוחקים יכולות לעבוד במקביל.
- Hand-over-hand: חוט עובר על הרשימה כך שנועל תחילה את ה-head sentinel והצומת הראשון, לאחר מכן משחרר את ה-head ונועל את השני, וכך הלאה.
- Remove: צריך לנעול גם את הצומת שרוצים למחוק וגם את קודמו, כדי שלא יהיה אפשר למחוק מתחת לידיים את קודמו ולגרום לקודם קודמו להצביע לזה שרוצים למחוק. נקודת הלינאריזציה: אם נמצא ה-item, היא תהיה כאשר מבצעים pred.next = curr.next – שינוי מצביע הקודם. אחרת, היא בקידום curr ב-curr = curr.next.
- Add: להוספה צריך לנעול את הקודם ואת היורש כך שאף אחד מהם לא יכול להימחק. אז מכניסים ביניהם.

חסרונות: הרבה פעולות תפיסה ושחרור מנעולים – לא יעיל.

Optimistic synchronization:

- בעת פעולה החוט רץ על הרשימה מבלי לנעול. כאשר מגיע לאן שרוצה, נועל את אותם צמתים כמו במימוש הקודם, ומוודא:
  - הראשון מביניהם עדיין נגיש מ-head (לא נמחק בדרך).
  - הראשון עדיין מצביע לשני (לא הוכנס ביניהם מישהו).
- אם אחרי הנעילה התנאים נשמרים, אפשר לעבוד.
- נקודת לינאריזציה: בהוספה נקודת הלינאריזציה היא בשינוי המצביע מהראשון אל החדש (אחרי שהחדש הוצבע אל השני).

חסרונות: יעיל רק אם סריקה כפולה (לפני נעילה ואחרי נעילה לוידוא) עולה פחות מסריקה אחת עם נעילות, ונועלים גם ב-contains, 90% מהפעולות.

Lazy list:

- כמו optimistic רק עם סריקה בודדת ו-contains לא נועלת.
- Remove: עד שלב הנעילות הכל אותו דבר. בעת מחיקה מסמנים את הצומת שמוחקים כמחוק, ורק אז עושים מחיקה פיסית (שינוי מצביע).
- חוסך סריקה נוספת: פשוט בודקים שהצומת והקודם לא מסומנים כמחוקים ושהקודם מצביע לנוכחי (זה ה-validation כאן).
- Contains: פשוט עוברים על הרשימה עד שעוברים את ה-key של הערך שמחפשים, ואז מחזירים האם הערך נמצא והוא לא מסומן כמחוק.
- פעולות add, remove – lazy; פעולת contains – wait-free.

חסרונות: קריאות add, remove כן re-traverse, ואם חוט מתעכב ב-CS נוצר עיכוב (traffic jam).

Lock-free List:

- נעשה שימוש ב-AtomicMarkableReference כך שה-mark הוא bit המסמן האם הצומת נמחק לוגית והמצביע הוא המצביע אותו משנים.
- Remove: שינוי ה-mark ב-next ואז לשנות את המצביע של הקודם ע"י CAS ליורש של זה שרוצים למחוק, רק אם ה-mark הוא זה שסימנו.
- בעת traversal (add / remove) אם נמצא צומת מסומן (מחוק לוגית בלבד), מבצעים עליו מחיקה פיסית ע"י CAS וממשיכים.

ReadWriteLocks (תרגול):

- לא ניתן לתפוס מנעול read כשה-write תפוס.
- לא ניתן לתפוס מנעול write כאשר ה-read ו/או ה-write תפוס.

במימוש SimpleReadWriteLock בכל פעולה תחילה תופסים מנעול מקומי (כמו בתרגיל עם הנשים/גברים).

ReadLock:

- תפיסה: כל עוד ה-writer תפוס, condition.await – ממתינים ומשחררים בינתיים את המנעול המקומי. כאשר מתפנה, מעלים את מספר הקוראים ב-1. בסוף משחררים את המנעול המקומי.

- שחרור: תופסים את המקומי, מורידים את מספר הקוראים ב-1, ואם הורדנו ל-0 מבצעים signalAll על ה-condition כדי להעיר את כל הממתינים ומשחררים את המקומי.

WriteLock:

- תפיסה: תופסים את המקומי. כל עוד יש קוראים או כותב, תמתין. ביציאה מההמתנה כוון את writer להיות true – סימון שיש כותב. משחררים את המקומי.
- שחרור: כותבים false ל-writer וקוראים ל-signalAll להעיר את כולם. שחרור ה-writer היא היחידה שלא צריכה להיות תחת נעילה כי הוא היחיד שפועל כרגע.

חסרונות: לא הוגן, ניתן לפתור ע"י FIFO (כמו בתרגיל הבית).

פרק 10,11: Concurrent Queues and Stacks:

Bounded Queue:

- תור מבוסס רשימה עם מצביעי head, tail. באיתחול שניהם מצביעים ל-sentinel: בעל ערך חסר משמעות ומצביע null. לתור יש שני מנעולים: deqLock ו-enqLock לכל אחד מקצוות התור. מחזיקים שדה size ומגבילים את גודלו ל-8.
- Enq: תפיסת מנעול enqLock, בדיקה שה-size קטן מ-8 ודחיפת צומת הערך החדש: שינוי tail ושינוי הצומת האחרון כרגע (ש-tailמצביע עליו) להצביע לצומת החדש. אז מעלה את size ב-1 ורק אז משחרר את המנעול. אם היה משחרר קודם אולי enq אחר היה נכנס ולא רואה size מעודכן. במימוש בו deq הרואה תור ריק מושהה, יצטרך enq שהכניס ערך לתור ריק בסוף לתפוס את מנעול ה-deqLock ולהודיע דרכו ל-deqs על כך.
  אם התור מלא:
  - אם משתמשים בשיטת הודעות, ה-enq יושהה וימתין ל-deq שיודיע שהתור התפנה.
  - אם משתמשים ב-spin, כעקרון מסוכן להסתובב כשהמנעול עדיין תפוס, אבל אם ה-deq גם מסתובבים כשהתור ריק זה בסדר. ה-spin כאן הוא על עותק מקומי ב-cache כך שאין עומס.
- Deq: תפיסת מנעול ה-deqLock, ואז קורא את ה-next של ה-sentinel. אם יהיה לא ריק, ישאר כך עד שישחרר את המנעול כי הוא ה-deq היחיד וגודל התור יכול רק לעלות (ע"י enqים). אז שומרים את ערך הצומת הראשון (עליו מצביע ה-sentinel) מקומית, ומשנה את מצביע head להצביע על צומת זה והופך אותו בעצם ל-sentinel החדש. הישן נזרק. אז משחררים את המנעול ומורידים את size ב-1. כאן לא חשוב להחזיק את המנעול בשינוי size כי deq אחר יכול לראות אם התור ריק ע"י הסתכלות על מצביע ה-sentinel ולא צריך להסתכל על size בשביל זה. אם ריק:
  - אם משתמשים בהודעות, ה-deq מושהה עד ש-enq יודיע שהתור התמלא.
  - אם משתמשים ב-spin זה בסדר. אחת משתי השיטות טובה, אך צריכה להיות זהה בין ה-enq ל-deq אחרת יהיה deadlock.

הערות Java:

- כדי להודיע לממתינים על מנעול צריך לתפוס אותו קודם.
- **Monitors**: אובייקטי synchronized ו-ReentrantLocks כלומר מאפשרים ליצור condition עליו מושהים במקום spin.
- **Condition**: אובייקט מקושר למנעול, נוצר ע"י קריאה ל-lock.newCondition. מספק await (עם או בלי timeout) שמשחרר את המנעול וממתין על התנאי. מספק גם signal, signalAll המעיר חוט אחד הממתין על התנאי / את כל מי שממתין עליו.
- **Synchronized blocks**: מספקים מנגנון דומה עם wait() ו-notifyAll() / notify().
- Lost wakeup: כמו lockout על ידי תנאי. כדי להמנע רצוי להשתמש תמיד ב-notifyAll / signalAll.

המשך Bounded Queue:

אופטימיזציה – פיצול מונים:

כדי למנוע race על ה-size שהוא משותף, מפצלים את המונה. מחזיקים שני מונים: מונה ל-enq שכל פעם יורד בכל enq. כאשר מגיע ל-0 נעשה סינכרון: תופסים את deqLock ומחשבים מחדש כמה מקום יש (אולי התפנו). באופן דומה עבור deq. חשוב להחזיק גם את המנעול של השני כדי שלא יעשה שינוי באחד השדות עליהם מתבסס החישוב תוך כדי.

Lock-Free Queue:

- מבנה דומה לקודם – רשימה עם מצביעי head, tail וצומת sentinel, רק ללא size כי התור לא חסום.
- Enq: יוצר צומת חדש עם הערך, משנה ע"י CAS את מצביע ה-tail הנוכחי להצביע מ-null אל הצומת החדש ואז משנה ע"י CAS את tail להצביע מהקודם אל הצומת החדש. כיוון שיש פה שתי פעולות, יתכן שיגיע נוסף כשמצביע ה-tail לא מעודכן עדיין. פתרון: מזהים שהוא לא מעודכן ע"י בדיקה שהמצביע של הצומת ש-tail מצביע עליו אינו null, ואם כך אז מתקנים את tail ל-tail.next ע"י CAS. אם CAS נכשלים:
  - אם הראשון נכשל, שום דבר לא קרה – פשוט מנסים שוב מהתחלה (חוט אחר הקדים אותנו).
  - אם השני נכשל, זה בסדר – חוט אחר הקדים אותנו בתיקון tail לצומת החדש שהוספנו.
- Deq: קורא את ערך הצומת עליו מצביע ה-sentinel ומחזיק אותו מקומית. ע"י CAS מחליף את ה-head להצביע על צומת זה והופך אותו ל-sentinel החדש.

אופטימיזציה – recycling:

- אפשר להחזיק רשימת צמתים לכל חוט, שלא דורשת סינכרון. כאשר חוט רוצה להכניס לתור הוא לוקח מהרשימה צומת (או יוצר אם היא ריקה), וכשרוצה להוציא מהתור הוא שם את ה-old sentinel ברשימה שלו. אפשר להשתמש ב-pool משותף.

בעיה: יתכן שחוט שרוצה לבצע deq ילך לישון וכשיתעורר ה-sentinel וזה שאחריו (שרצה להוציא) כבר ב-pool (כי מישהו הוציא אותם בזמן שישן). כעקרון לא תהיה בעיה אם הם ב-pull, כי ה-CAS שיבצע על ה-head יכשל (ה-head מצביע כעת לצומת אחר כ-sentinel) אבל:

ABA fail: אם ה-Sentinel המקורי הספיק להתמחזר ולהיות sentinel שוב, כשיתעורר החוט ה-CAS על head יצליח ויצביע על צומת ב-pool.

פתרון: מוסיפים לכל מצביע גם counter – AtomicStampedReference. כך ניתן לזהות ב-CAS האם הוא של הסיבוב הזה או הספיק להתמחזר.

Lock-Free Stack:

- משתמשים במצביע top המצביע תחילה לצומת sentinel.
- Push: יוצרים צומת חדש עם הערך, מצביעים איתו על ה-top הנוכחי ומבצעים CAS ל-top שיצביע עליו במקום על ה-top הקודם.
- Pop: קוראים את הצומת ב-top, ומבצעים CAS על top שיבציע על ה-next שלו.

Elimination-Backoff Stack:

משתמשים ב-lock free stack בתוספת elimination array: מנסים להיכנס ל-stack. אם מצליחים, מבצעים את הפעולה על ה-Stack. אם לא מצליחים נסוגים ל-elimination array, שם מחכים על טווח מסויים במשך זמן מסויים בהמתנה ל-collision בין push ל-pop ופשוט מבצעים את ההחלפה שם. הטווח והזמן צריכים להיות פונקציה של ה-contention. אם עבר הזמן – מנסים שוב את ה-stack (זה סוג של backoff אקטיבי).

Linearizability:

- Un-eliminated calls: כמו קודם.
- Eliminated calls: עבור push ו-pop מתאימים, נשים את ה-pop מיד אחרי ה-push שאת ערכו לקח.

מימוש:

- מחזיקים מערך elimination של Exchanger, לו יש מצבים: EMPTY – עוד ריק; WAITING – מישהו ממתין לאחר; BUSY – מתבצעת החלפה.
- EMPTY: ע"י CAS מנסים לשים ב-slot (exchanger מסויים) את ה-item שלי ולהחליף את המצב ל-WAITING. אם נכשל אז מישה ושינה את המצב כבר ל-WAITING, מנסים שוב. מסתובבים בלולאה עד שרואים שהמצב השתנה ל-BUSY – כלומר נלקח, או שנגמר הזמן. אז מכוונים את ה-SLOT להיות שוב EMPTY, ומחזירים את ה-item מהשני. אם נגמר הזמן, ע"י CAS מכוונים את ה-slot להיות null, EMPTY. אם האתחול הזה נכשל – מישהו בכל זאת הגיע אז פועלים כמו קודם (אתחול ולקיחה).
- WAITING: מבצעים CAS בו מצפים לראות את ה-item של האחר ולשים במקומו את שלי, כשהמצב WAITING והופכים אותו ל-BUSY. מחזירים את ה-item של השני. אם ה-CAS נכשל מישהו לקח לפני, לנסות שוב.
- BUSY: חוטים אחרים משתמשים ב-slot, לנסות שוב מהתחלה את הלולאה.
- **במחסנית**: כל visit ב-elimination array כולל לבחור slot באופן ראנדומלי ולהחזיר את תוצאתו – או שמצליחים או timeout.
  - ב-push: אם הגענו ל-visit והערך המוחזר הוא null אז סימן שבא מולנו pop – הצלחנו. אחרת מנסים מהתחלה גישה למחסנית...
  - ב-pop: אותו דבר רק מצפים ל-non-null value לחזור – סימן שבא מולנו pusher.

Skip List (מהתרגול):

בונים Set מבוססת skip list באופן הבא:

- ב-skip list מחזיקים את הצמתים מקושרים אחד לשני באופן ממויין בכמה רמות כך שכל רמה היא תת רשימה של הרמה מתחתיה
- בנייה: בוחרים גבהים באופן ראנדומי כדי שהתכונה תשמר באופן פרובביליסטי: כל מצביע בגובה i "מדלג" מתחתיו על $\sim 2^i$ פרטים. חיפוש ב-log(N).
- Find: בודקים בכמה רמות מלמטה מוצאים מצביעים אל הערך שמחפשים, ובכמה מצביעים מהערך שמחפשים. אם מספר זה שווה – הערך נמצא. אחרת – הוא לא נמצא.

Lazy Skip List:

- מוסיפים mark לסימון logic delete. Add, remove – ישתמשו בנעילות על צמתים, contains – wait-free.
- Add: יוצרים צומת עם הערך שרוצים להכניס בגובה ראנדומי; מוצאים מלמעלה למטה את כל ה-pred שלו, נועלים ומוודאים שנשארו. מכניסים את הצומת החדש למקום ומשחררים את הנעילות. מוסיפים ביט לצומת המסמן האם הוא fully linked (מוצבע ע"י כל ה-pred שלו).
  - נקודת לינאריזציה: כאשר הצומת מסומן כ-fully linked.
- Remove: מוצאים את ה-pred של זה שרוצים למחוק, נועלים אותו בלבד, מסמנים אותו כמחוק לוגית. אז נועלים את ה-pred בסדר עולה (מ-0 והלאה) ומבצעים מחיקה פיסית על ידי הצבעתם לבא אחרי המחוק. אז משחררים את המנעולים של כולם, והמחוק נעלם.
  - נקודת לינאריזציה: כאשר הצומת מסומן במחיקה לוגית.
- Contains: חיפוש כמו קודם, בתוספת בדיקה שהצומת לא מסומן מחוק. נקודת הלינאריזציה: כאשר נמצא צומת fully linked unmarked.

פרק 12: Shared Counters and Parallelism:

Shared pool:

- קבוצה לא סדורה של אובייקטים, עם מתודות add, remove שהן blocking (כאשר ה-pool מלאה או ריקה בהתאמה).
- אם משתמשים במנעול אחד (או בלוק synchronized) לשמור על ה-pool נקבל bottleneck על המנעול והתנהגות סדרתית.

Counting implementation:

- ה-pool הוא מערך של אלמנטים שכל אחד מכיל אובייקט או null.
- Put: מבצע getAndIncrement לקבלת אינדקס במערך אליו רוצים לשים. אם מלא, ממתינים עליו שיתרוקן.
- Remove: מבצע getAndIncrement על מונה אחר לקבלת אינדקס אותו רוצים להוציא. אם ריק, ממתינים עליו שיתמלא.
- שיטה זו הופכת bottleneck יחיד לשניים ומקלה קצת על העומס.

Software combining tree:

- עץ בו כל צומת מוקצה לחוט או לכל היותר לשניים. בשורש העץ מוחזק מונה.
- Increment: חוט מתחיל מהצומת שלו ומתחיל לעלות מעלה בעץ. אם מגיע לצומת יחד עם חוט אחר, משלבים את הערכים ששניהם מביאים, ואז אחד מהם מעלה אותו מעלה כדי להוסיף אותו למונה והשני ממתין עד שהוא יחזור עם הערך שהיה במונה בעת שהוסיף לו. כשחוזר זה שהלך, שני החוטים ממשיכים לרדת מטה תוך שהם זכרו מה הערך שמבחינתם ראו במונה – זה שהלך בפועל למונה הוא הראשון, והוא יראה את ערך המונה שקרא. השני (שהמתין לו) יראה את ערך המונה + ערך הראשון בעת מפגשו עם השני, כך כאילו ראה השני לבדו את המונה אחרי שביקר בו הראשון.

מצבים:

חוט מתקדם בתוך צומת בעץ ע"י ריצה על לולאת node.precombine() וכל פעם מקבל את ערך ה-parent שלו. Precombine: מתודה סינכרונית לזמן קצר יחסית, בתחילתה החוט יכנס ל-wait אם הוא נעול (מישהו משתמש בו ל-combine מוקדם יותר). שלבי ה-navigation:

מתודת precombine:

- IDLE: הוא הראשון שמגיע, משנה את המצב ל-FIRST, יחזור לצומת לבדוק קלט של חוט שני בצומת. מחזיר true – עולה במעלה העץ.
- FIRST: הוא השני שמגיע, לכן **נועל** את הצומת כדי שהראשון לא יעזוב בלעדיו. משנה את המצב ל-SECOND ומחזיר false – לא מתקדם מעלה.
- ROOT: מחזיר false – לא מתקדם במעלה העץ (אין לאן).

ה-combining navigation: ראשית מתחיל מערך 1 (ההוספה שלו), אז מטפס למעלה, תוך שמחשב על הדרך ערך משותף לצמתים בהם עובר, אם יש שם מה לחשב, שם אותם במחסנית (כדי להוציא בסדר הפוך לעליה בשביל עדכון אחר כך) ומתקדם למעלה עד ה-root.

מתודת combine:

תחילה ממתין עד שמנעול פנוי. הוא יהיה נעול עד שהחוט השני יפקיד את הערך שלו בצומת. אז מיד יתפוס את המנעול (בין לבין לא יתפס המנעול ע"י חוט אחר כי המנעול נקרא מתוך מתודות סינכרוניות). כשתופס – נועל בחוץ נסיונות מאוחרים של חוטים בשלב ה-precombine שלהם.

- FIRST: החוט לבד, מחזיר את הערך ההתחלתי שלו ללא תוספות.
- SECOND: לא לבד, מחבר את הערך שלו עם ערך השני ומחזיר את זה.

קריאה ל-(stop.op(combined – מתודת op:

כאן רק ROOT או SECOND אפשריים כי הראשון היה והמשיך במעלה העץ ולא עצר בצומת הזה.

- ROOT: כאן מוסיפים את הערך combined (מה שצבר החוט המגיע עד כה) לערך ב-root (המונה) ומחזיר את הערך הקודם שלו.
- SECOND: מפקיד את הערך שצבר עד כה ב-secondValue שייצבר ע"י מי שהוכתר כראשון בצומת זה (הוא שיחבר את הערכים של השניים). אז משחרר את המנעול ועושה notifyAll כדי לשחרר את הראשון. אז ממתין (נכנס ל-wait) עד שהמצב עובר ל-RESULT ע"י החוט הראשון – מחזיר לו את הערך שאמור לקבל, משחרר את המנעול (שתפס החוט הראשון בשלב ה-combining), מודיע לכולם, מעביר את המצב ל-IDLE וחוזר.

שלב ה-distribution – מתודת distribute:

עוברים על ה-stack כל עוד יש בה ערכים, נותנים את הערך לחוט השני בצמתים שאוחסנו במחסנית ובסוף מחזירים את הערך שלנו לקורא. בחלוקה:

- FIRST: אין חוט שני שהמתין. מאפסים את הצומת – מעבירים ל-IDLE, משחררים את המנעול, מודיעים (notifyAll).
- SECOND: שמים ב-result את הערך הקודם + הערך שלי (של הראשון), משנים את המצב ל-RESULT ומודיעים. השני ידע לקחת את הערך ולשחרר את המנעול.

יעילות:

ה-latency גבוה כי גובה העץ הוא log(N). המרווחים בהם מגיעים החוטים גם משפיעים – אם כולם באים יחד, ונעזרים אחד בשני (combine) – ל-n חוטים יעלה log n, אבל אם באים לא ממש ביחד הם נועלים אחד את השני בחוץ, וכיוון שכל אחד מטפס log n העלות תהיה $n \cdot \log n$.

ה-latency של ה-combining tree מתחיל גבוה מאשר spin lock, אבל ככל שיש יותר מעבדים כך הוא נהיה יותר טוב מ-spin. ה-throughput שלו מתחיל כמו spin lock אבל גדל בהרבה ככל שיש יותר מעבדים.

כשיש הרבה עבודה, אם מספר המעבדים גבוה, עדיף המתנה לא מוגבלת כדי לייעל latency – למנוע כמה שיותר נעילות של חוטים הבאים בעיכוב ע"י חוט שהגיע ונעל צומת בזמן שמטייל על העץ.

סיכום: עובד טוב תחת contention גבוה, רגיש לשינויים, ניתן להשתמש בהם לאטומיות של פעולות getAndMumble, לינארזבילי.

Balancer:

מקבל כקלט אובייקטים ומוציא אותם ליציאות באופן מאוזן. מאזן בינארי: כל אובייקט i שנכנס דרך כניסה כלשהי יצא ביציאה $i \bmod 2$.

Counting Network:

מערך בעומק $\log_2 n$ של balancers עבור n חוטים משמש כמונה מדורג. המבנה מבטיח שלא יקרה מצב בו ילקחו מספרים גבוהים לפני שילקחו נמוכים. ה-contention נמוך ללא bottleneck סדרתי, throughput גבוה.

Bitonik[k] Network:

סוג של counting network. מבנה זה לא לינארזבילי אבל כן Quiescently consistent.

מימוש אינדוקטיבי:

- עבור Bitonik[2k] משתמשים בשני Bitonik[k] ו-Merger[2k].
- Bitonik[k] יהיה ברוחב k ועומק $\frac{\log_2(k)(\log_2 k+1)}{2}$.

ההוכחה שתכונת ה-step נשמרת היא באינדוקציה. עבור הנחה על bitonik[k], שני כאלה משמרים את תכונת המדרגה לפני הכניסה ל-merger[2k]. מכאן שחלק השורות הזוגיות בפלט (והאי זוגיות) גם משמר את תכונת המדרגה. ה-merger מצליב $even(0) + odd(1), odd(0) + even(1)$ כך שהפלט נבדל לכל היותר ב-1. אז אולי פלט התחתון יהיה גדול ב-1 מהעליון, אבל בסוף נעביר אותו בשכבה אחרונה שתעביר את ה-1 הזה לעליון.

מבנה מחזורי:

מימוש נוסף בו כל block[k] בעומק $\log_2 k$, צריך $\log_2 k$ בלוקים. סה"כ: $(\log_2 k)^2$ בלוקים.

משפט חסם תחתון לעומק:

עומק על counting network ברוחב w הוא לפחות $\Omega(\log w)$. קיימת counting network שחסם זה צמוד (Θ).

Sequential Theorem:

אם רשת סופרת באופן סדרתי, כלומר tokens מתקדמים אחד אחרי השני, היא סופרת גם אם הם עוברים באופן מקבילי.

יעילות:

- טובה במקביליות גבוהה, בדומה ל-combining trees.
- הכי יעיל כאשר הרשת saturated, כלומר כל ה-balancers מלאים. עדיף על פחות או על יותר מדי (כלומר עדיף $P \cong w \cdot \log w$).
- בהרבה מעבדים, ככל שה-k גדל כך גם ה-throughput.
- אז אם נשתמש ב-bitonik network ב-pool ה-latency יהיה $(\log_2 w)^2$ אבל נרוויח במקומות אחרים.

Anti-Tokens:

ניתן להשתמש ב-anti token כדי לבטל – להוריד ערך וכך למשל לממש decrement. הוא יעבוד כמו token רגיל רק במקום לקרוא, להפוך ולהמשיך, הוא יהפוך יקרא וימשיך – כך יגיע למקום בו נמצא token ויבטל אותו.

רשת מניה היא כמו מכונת מצבים שמתישהו חוזרת למצב התחלתי אחרי Ω כניסות ולכן מחזורית, anti-tokens עוזר לחזור מצבים אחורה.

- לכל anti-token על כניסה i יש אותו אפקט כמו $\Omega - 1$ tokens. לכן הרשת עדיין תהיה במצב חוקי.

בקיצור זו הדרך לממש getAndDecrement.

Counting Tree:

- שילוב – יוצרים counting network בצורת עץ שיש כניסה אחת והעלים הם היציאות.
- Diffracting tree: זה יוצר contention גבוה על ה-root כי זו כניסה אחת. ניתן לפתור זאת באמצעות prism array – כיוון שכל מעבר של מספר זוגי ב-balancer נראה כאילו לא השתנה מצבו, אפשר להעביר מראש חלק מיד ימינה וחלק מיד שמאלה ללא contention גבוה.

פרק 13: Hashing and Natural Parallelism:

Hash: open vs. closed addressing:

- Closed: לכל item יש bucket קבוע בטבלה. כל bucket מכיל כמה items.
- Open: כל item יכול להמצא בכל bucket בטבלה, כל bucket מכיל לכל היותר item אחד.

Sequential closed hash-map: המימוש אליו מתייחסים. כל item ממופה ל-bucket בטבלה במקום item.hashCode() % table.size. כאשר המספר הממוצע ב-buckets של ה-items עובר סף מסויים, מגדילים את גודל הטבלה פי 2 ומבצעים rehash.

Simple Hash-Set:

מבוססת מערך של lock-free list, מספקת מתודות add, remove, contains.

Read/Write locks:

מנעולים לכתיבות ולקריאות, תכונות safety שצריכות להישמר:

- אם readers > 0 אז writer = false (אין יותר מכותב אחד לכן מספיק בוליאני, יכולים להיות כמה קוראים).
- אם writer = true אז readers = 0.

Fifo r/w lock: כאשר w מבקש את המנעול, לא נכנסים יותר r. מחכים שיצאו הנוכחיים ואז יתפוס w את המנעול.

Lock free resizing:

מחזיקים את כל האובייקטים ברשימה אחת מתמשכת, ובכל resizing (הכפלה פי 2 של מספר ה-buckets) ה-buckets החדשים מצביעים למקומות ברשימה המסמנים את תחילת אותו Bucket.

Split order:

עבור טבלה בגודל 2^i, ה-bucket b מכיל את האיברים שהמפתח שלהם הוא $k = b(mod\ 2^i)$, והאינדקס של ה-bucket הוא i ה-lsb של המפתח.

כאשר יש פיצול, חלק מהמפתחות נשארים במקום $b = k\ mod(2^{i+1})$ וחלק עוברים ל-bucket $b + 2^i = k\ mod(2^{i+1})$, כאשר זה נקבע לפי הביט ה-lsb ה-i+1.

- בעצם המפתח יימצא באינדקס של ה-reverse bits שלו. אם ברשימה 8 מקומות כרגע כלומר 2^3 אז המפתח 1 שהוא 001 לפיכך ישב במקום 100 כלומר במקום מספר 4 (המקום החמישי ברשימה).
- עבור שני מפתחות a, b אז a יהיה לפני b אם ה-reverse bits של a הוא מספר קטן מאשר b.

בעיה: כדי למחוק ערך צריך CAS כפול כי עלולים להיות שני מצביעים – זה של הרשימה וזה של ה-bucket.

פתרון – sentinel nodes:

- מכניסים sentinels בתחילת כל bucket (בתוך הרשימה, כך שכל bucket יצביע אליו.
- הוספת bucket: CAS על הוספת sentinel לרשימה ועוד CAS כדי להצביע מה-bucket ל-sentinel.
- אתחול רק בעת צורך: ב-resize רק מגדילים את גודל מערך ה-buckets בלי הכנסת ה-sentinels כדי שזה יהיה מהיר. אז, רק כשצריך את ה-bucket שלו אין עדיין sentinel יוצרים אותו ועושים את השינוי הנדרש. יתכן מצב בו נגדיל את הטבלה ונצטרך כדי לגשת לאיבר לאתחל עד log n סנטינלים, אבל תוחלת האתחולים הנדרשים פר פעולה היא עדיין **קבועה**.
- יצירת מפתח: מפתח רגיל הוא reverse של המפתח המקורי עם 1 בביט הגבוה, ומפתח sentinel הוא רק reverse.
- Add: לוקחים את ה-hash של האובייקט, ובוחרים לו מפתח על ידי hash % table-size. אז יוצרים לו מפתח regular (רוורס עם 1 בביט הגבוה) כי הוא לא sentinel. אז מושכים את הרשימה שה-bucket של ערך זה, ואם צריך מאתחלים אותה (יוצרים לה sentinel), ומנסים להוסיף לה את האובייקט עם ה-reversed key. אם לא מצליחים (קיים אובייקט עם מפתח כזה) – חוזרים. אחרת בודקים האם צריך עכשיו resize וחוזרים.
- סה"כ: בתוחלת פעולות יעלו $O(1)$ - גישה דרך קיצור ויצירה של sentinels.

ביצועים:

זמני גישה נשארים יחסית קבועים גם כשהצפיפות של הנתונים עולה (פחות resizing), אבל תפיסת זיכרון דינאמית בהגדלות ויצירות איברים יכול להביע לחוסר לוקאליות ב-cache – ביצועי cache רעים.

Linear Probing: ...

פרק 16: Futures, Scheduling, and Work Distribution:

Thread pools:

- החזקת מאגר חוטים בעלי זמן חיים ארוך, אליהם מצמידים משימות כשיש, ובין לבין ממתינים למשימה הבאה. חוסך overhead של יצירת חוט והריגתו כל פעם מחדש פר משימה.
- יתרון נוסף הוא הפרדת האפליקציה מספסיפיקציות של החומרה.

ExecuterService:

- Runnable: משימות שלא מחזירות תוצאת חישוב. קריאה למתודת ()run.
- Callable<T>: משימות שמחזירות תוצאה מסוג T. קריאה למתודת ()call.
- חישוב אסינכרוני באמצעות Future<T>: Future<T> f = executor.submit(task) כאשר task היא Callable<T>. ההמתנה לערך המשימה: f.get() – חוסם עד שהמשימה בוצעה והחזירה ערך. בשביל Runnable: Future<?> f ופשוט f.get() לא מחזירה ערך.
- העברת משימות אינה מחייבת את ה-scheduler.

מודל דינאמי: DAG: תוכנית מרובת חוטים ניתנת לתיאור פורמאלי ע"י DAG בה כל צומת בגרף הוא צעד בתוכנית

מושגים:

- T_1 – Work: העבודה הוא הזמן הכולל על מעבד יחיד.
- T_∞ - critical path length: ה-dependency path הארוכה ביותר, כך שלא משנה על כמה מעבדים נריץ את התוכנית, לא נוכל לקצר דרך זו.
- T_p - time on P processors: זמן הריצה על P מעבדים.

<u>תכונות</u>:

- $T_p \geq \frac{T_1}{P}$
- $T_p \geq T_\infty$

<u>Speedup</u>: היחס בין זמן הריצה על מעבד אחד לעומת על P מעבדים.

- Linear speedup: $T_1/T_p = \Theta(P)$ - כאשר ה-speedup לינארי במספר המעבדים.
- Max (average) speedup: T_1/T_∞ (ידוע גם בכינוי parallelism).

<u>חישובים</u>:

- לחשב את $T_1(n)$ העבודה לרוב ע"י Master Theorem, למשל: $T_1(n) = 2T_1\left(\frac{n}{2}\right) + \Theta(1) = \cdots = \Theta(n \cdot \log n)$
- לחשב את $T_\infty(n)$ ה-critical path לרוב ע"י חישוב מסלול אחד ב-Master theorem, למשל: $T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\log n)$

<u>תזכורת: master theorem</u>:

עבור $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ כאשר $a \geq 1, b > 1$:

- אם $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ אז: $T(n) = \Theta\left(n^{\log_b a}\right)$
- אם $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ אז $T(n) = \Theta\left(n^{\log_b a} \cdot \log^{k+1} n\right)$
- אם $f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right)$ וגם $af\left(\frac{n}{b}\right) \leq cf(n)$ ל-$c < 1$ אז $T(n) = \Theta\left(f(n)\right)$

<u>עוד חישובים</u>:

- זמן הריצה על greedy scheduler מקיים: $T \leq \frac{T_1}{P_A} + T_\infty \cdot \frac{(P-1)}{P_A}$ כאשר P_A הוא מספר המעבדים הממוצע לרשותינו. ככל שממוצע המעבדים גדל, כך הריצה יותר טובה (בא לידי ביטוי בשני הגורמים בחסם).

<u>Lock Free Work Stealing</u>:

- כל תהליך שאין לו משימות מנסה "לגנוב" מחוט אחר.
- לכל תהליך יש work pool שהיא double-ended queue:
  - ה-owner מכניס משימות ולוקח ב-popBottom, ואז עובד על החוט עד שמסיים.
  - ה-owner מכניס ע"י pushBottom משימות חדשות.
- כשנגמרת לו העבודה, הוא בוחר <u>באקראי</u> תור של חוט אחר כדי לגנוב ממנו משימות ע"י popTop.
- popTop ייכשל אם אחר הצליח או popBottom לקח את האחרון בתור (שם עלול להתנגש עם גנב).

<u>ABA problem</u>:

ה-top הוא אינדקס עליו גנבים עושים CAS כדי להורידו ולמשוך **אינדקס** של משימה בתור אותה יעשו. יתכן גנב שיעשה CAS וימשוך משימה, יקח את המשימה, ואז ה-owner יקח את כל המשימות, ויכניס מספיק חדשות כך שכשהגנב מתעורר הוא מבצע שוב CAS ל-top כדי להוריד את המצביע למשימה הבאה, אבל בפועל מתפספסת משימה שעוד לא עשו.

<u>פתרון</u>:

שימוש ב-stamp עבור מצביע ה-top שבכל שינוי עושים לו העלאה ב-1. כך שלא יתכן בדוגמא לעיל שגנב יצליח ב-CAS השני (נעזוב stamp overflow).

<u>Bounded DEQueue</u>:

- מחזיק AtomicStampedReference<Integer> ל-top, volatile int ל-bottom (רק ה-owner משנה אותו) ומערך []Runnable למשימות.
- <u>pushBottom</u>: שמים את הערך במערך במקום ה-bottom הנוכחי ומקדמים את bottom ב-1 (תמיד מצביע למקום הריק הבא בתחתית).
- <u>popTop</u>: קורא את הערך וה-stamp של top, מעלה את הערכים שלהם ב-1. אם oldTop => bottom – התור ריק, מחזירים null. אחרת לוקחים את המשימה ב-oldTop ומנסים לעשות CAS לשנות את top וה-stamp שלו. אם מצליחים, מחזירים את המשימה. אחרת היה קונפליקט עם גנב אחר או ה-owner, מחזירים null.

- popBottom: אם התור ריק (bottom == 0), מחזירים null. אחרת, קודם כל מורידים את bottom ב-1 ולוקחים את המשימה במקום הזה (שכן bottom מצביע תמיד על המקום הריק הבא). אז לוקחים את oldTop ומכינים את הערכים החדשים, הפעם top החדש יהיה 0 (איפוס top) – למקרה שמדובר במשימה האחרונה. אם bottom גדול מה-top הישן, אין קונפליקט, ישר מחזירים את המשימה. אם יש שוויון, נותר לכל היותר איבר אחד, ואז: מאפסים את bottom, אם ה-CAS מצליח – מחזירים את המשימה (האחרונה) ובדרך איפסנו את האינדקסים (אם ה-CAS נכשל, בכל מקרה המשימה האחרונה נלקחה, אז נאפס בכל מקרה את bottom). אם גנב ניצח (ה-CAS נכשל) ולא מוחזרת המשימה – בכל מקרה נאפס את top (ע"י set).

אופטימיזציות:

Work balancing:

- גניבה עם CAS היא פעולה יקרה, ולכן אפשר לגנוב יותר ממשימה אחת – randomly balance loads.
- עובד טוב כאשר יש תנודה במספר המעבדים.

פרק 17: Barrier Synchronization:

מימושים של barriers:

- Cache coherence: הסתובבות על משתנה מקומי אל מול מקום סטטי.
- Latency: מספר הצעדים שלוקחים בהנחה שכולם הגיעו.

בעיה במימוש הנאיבי: יתכן שמישהו ממתין לעדכון שהוא יכול להמשיך, נרדם, בינתיים בא האחרון ומודיע לכולם שאפשר להמשיך, ונעשה שימוש חוזר במחסום. אז יתעורר הישן מבלי שהתעדכן, וכולם יתקעו – כי אלו שהתקדמו ממתינים לאחרון מה-barrier הקודם שיעבור, והוא ממתין שישחררו אותו.

פתרון: Sense-Reversing Barriers:

- משתמשים ב-sense המציין האם נמצאים בשימוש בשלב זוגי או אי זוגי. כל חוט שומר מקומית את ה-sense לשלב הבא, נכנס, וממתין כל עוד ה-sense שונה מה-sense שיש לו לשלב הבא. האחרון שבא (מוריד את המונה שבודק כמה עברו כבר ל-0) מעדכן את המונה חזרה ל-n ומשנה את sense להיות mySense – שישחרר את כולם. בכל מקרה כשיוצאים מהמחסום משנים את ה-sense שלו להיות ההופכי.

Combining Tree Barriers:

- לכל Node יש מונה, גודל (אליו מאופס המונה בעת איפוס ה-barrier), Node הורה ו-sense.
- Await: אם אני אחרון להיכנס למחסום, וההורה שלי לא null, אני קורא ל-await של ההורה, וכשמסיים (מגיע לצומת ללא הורה) מעדכן את המונה חזרה ל-size ומשנה את ה-Sense של אותו שלב (וכל השלבים חזרה ברקורסיה). אחרת, ממתין על ה-Sense של השלב אליו הגיע עד שישתנה.

תכונות:

- אין sequential bottleneck: קריאות getAndDecrement מתבצעות במקביל במקומות שונים בכל ברמה בעץ.
- Low memory contention: מאותה סיבה.
- Cache: טוב על bus-based, כיוון שהחוטים מסתובבים מקומית. לא טוב ל-NUMA שם החוטים יכולים להסתובב רחוק.

Tournament Barrier:

- כל TBarrier מחזיק דגל בוליאני, ושני TBarrier: partner ו-parent. בנוסף בוליאני המסמן האם אני השורש.
- Await: מקבלת כפרמטר את ה-sense הנוכחי. אם אני השורש, מיד חוזר. אחרת, אם ה-parent לא null אזי אני מנצח שלב זה, ואז: מסתובב כל עוד הדגל שלי שונה מ-mySense – עד שיבוא הבן זוג (המפסיד) לשנות לי את הדגל, וכשיגיע תהיה קריאה ל-await(mySense) של ה-parent שלי ובסוף (אחרי שחזר מלמעלה) שם לבן זוג (המפסיד) את mySense להיות הדגל שלו (מודיע לו). אם אני המפסיד – מודיע ל-partner המנצח שהגעתי (הוא מסתובב על תנאי זה), ואז מסתובב כל עוד הדגל שלי שונה מ-mySense – עד שיבוא המנצח מלמעלה וישנה אותו.

תכונות:

- כל חוט מסתובב על מקום סטטי – טוב גם ל-bus based וגם ל-NUMA.

Dissemination Barrier:

- יש log n סיבובים כאשר בכל סיבוב i מודיע כל חוט A לחוט במקום ה-$A + 2^i (mod\ n)$.
- תכונות: לא טוב מבחינת cache.

Static Tree Barrier:

- לכל חוט מוקצה צומת בעץ, בו מחזיק את מספר הבנים שממתין להם. יש דגל sense המסמן את המצב.
- כל חוט שבא תחילה מעדכן את ההורה שלו שהגיע ע"י שמוריד לו את המונה ב-1. אז בודק את המונה שלו ומסתובב עליו עד שיתאפס, אם הוא 0 – עובר להמתין על הדגל, אלא אם הוא השורש – אז משנה את הדגל ומסיים.
- כאשר כולם משוחררים ע"י שינוי הדגל ע"י השורש, הם משחזרים את המונה שלהם להיות מלא ובעצם יכולים לעבור את המחסום.

תכונות:

- מעט עומס על ה-cache.
- מעט זיכרון

The Nature of Progress:

הגדרה חדשה: obstruction freedom: כל מתודה חוזרת אם היא רצה לבד למשך מספיק זמן.

אלגוריתם ה-simple snapshot מקיים:

- Update היא wait-free
- Scan היא obstruction free – מסיימת אם ניתן לה מספיק זמן לרוץ לבד (אז אין updates מקבילים אליה).

Maximal vs. Minimal:

- Maximal progress: קריאה כלשהי מתישהו חוזרת – אין נקודה בהיסטוריה בה כל הקריאות למתודות לוקחות זמן אינסופי מבלי לחזור.
- Minimal progress: כל קריאה מתישהו חוזרת – אין נקודה בהיסטוריה בה קריאה כלשהי לוקחת זמן אינסופי מבלי לחזור.

עוד מושגים:

- Fair history: כל חוט מבצע מספר אינסופי של צעדים. בפועל: כל חוט מתישהו עוזב את ה-CS אליו נכנס (וכך נמנע מחוט אחר להיות בעל מספר סופי של צעדים בהיסטוריה).
- Uniformly isolating history: אם כל חוט בסופו של דבר מקבל זמן לרוץ לבדו למשך מספיק זמן.

Clash freedom: מתודה תהיה clash-free אם היא מבטיחה minimal progress בכל היסטוריה uniformly isolating.