TECHNION
Israel Institute of Technology

Computer Science Department

Operating Systems – 234123

Homework Exercise 3 – Dry

Presented by:

Omer Stoler: 318471356

Adi Arbel: 207919614

Emails:

stoler.omer@campus.technion.ac.il

adi.arbel@campus.technion.ac.il

Teaching Assistant in charge:

Ido Imanuel

Assignment Subjects & Relevant Course material

Threads, Synchronization, pthread Library

Recitations 5-6, Lectures 4-5

Submission Format

1. Only **typed** submissions in **PDF** format will be accepted. Scanned handwritten submissions will not be graded.
2. The dry part submission must contain a single PDF file named with your student IDs –

   DHW3_123456789_300200100.pdf
3. The submission should contain the following:
   a. The first page should contain the details about the submitters - Name, ID number and email address.
   b. Your answers to the dry part questions.
4. Submission is done electronically via the course website, in the **HW3 – Dry** submission box.

Grading

1. **All** question answers must be supplied with a **full explanation**. Most of the weight of your grade sits on your **explanation** and **evident effort**, and not on the absolute correctness of your answer.
2. Remember – your goal is to communicate. Full credit will be given only to correct solutions which are **clearly** described. Convoluted and obtuse descriptions will receive low marks.

Questions & Answers

- The Q&A for the exercise will take place at a public forum Piazza **only**. Please **DO NOT** send questions to the private email addresses of the TAs.
- Critical updates about the HW will be published in **pinned** notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated.

A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
- Be polite, remember that course staff does this as a service for the students
- You're not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
- When posting questions regarding **hw3,** put them in the **hw3** folder

Late Days

- Please **DO NOT** send postponement requests to the TA responsible for this assignment. Only the **TA in charge** can authorize postponements. In case you need a postponement, please fill out the attached form : https://goo.gl/forms/eW76r9cRNPTw9vAW2

הנחיות בנוגע לתרגיל הבית הנוכחי:

א. שימו לב, הקוד הנתון בחלק מקטעי הקוד אינו קוד פורמלי, ולכן אין להתייחס בפתרונכם לבעיות קומפילציה כאלה או אחרות. יש לזהות את מהות השאלה ולענות לפיה.

ב. יש **להסביר כל** סעיף עליו אתם עונים. הסבר שכזה תורם לכם להבין יותר טוב את התרגיל, ותורם לנו בלהבין יותר טוב את פתרונכם. **מרבית הניקוד יינתן על סמך הסבר זה**.

ג. חלקו האחרון של תרגיל הבית היבש מיועד לפתרון **לאחר** פתרון של החלק הרטוב של תרגיל בית זה. מאידך, לאלו מכם שמסתבכים עם הרטוב, יתכן וניתן להיעזר בחלק זה עבור רמזים.

ד. התשובות אותן אנו מחפשים בשאלות "מה יודפס" לא בהכרח יתקבלו עם הרצה אמיתית של הקוד על מחשבכם, כי בשאלה אין אנו נותנים נתונים מפורשים על אורך ה-Time Slice למשל, דבר הקבוע ויחיד אצלכם במחשב האישי. קיימים כמובן שיקולים נוספים. אנו מחפשים תשובה תיאורטית בלבד, עם הסבר מבוסס.

ה. **ניקוד בונוס:** מטרת תרגיל הבית (יבש ורטוב) היא להציג בפניכם את עולם הסנכרון. זהו עולם רחב ועמוס בתובנות, ולעיתים (כמו למשל, backtracking ממבוא למדמ"ח) קשה להבין היטב את האלגוריתמיקה מאחוריו. חשוב אפוא שתתאמנו על כך, כך שחוויית הבחינה תעבור באופן חלק. מכאן, אנו כן ממליצים להתמודד עם סעיפי **כתיבת** מנגנוני הסנכרון בחלק היבש, למרות שמטלה זו קשה יותר **מההבנה** של מנגנוני סנכרון מקולקלים. בגלל הקושי היחסי, סעיפים אלו סומנו כסעיפי בונוס. הניקוד המקסימלי הניתן לצבור בחלק היבש של תרגיל בית זה הוא **115**. במידה וציונכם בתרגיל היבש עלה מעבר ל-100, גם התוספת תשוקלל במיצוע הכללי של שיעורי הבית.

חלק ראשון: זיהוי כשלי סנכרון

תזכורת: תכונות הקטע הקריטי (או תכונות מנעולים, ביחס להבטחתם על הקטע הקריטי)

תכונות הכרחיות:

1. **Mutual Exclusion – מניעה הדדית** – בכל רגע נתון, לא יכול להיות יותר מחוט אחד בתוך הקטע הקריטי (הדבר שקול לכך שהקטע הקריטי הופך לנקודת **סריאליזציה** במסלולי הביצוע).
2. **Progress – התקדמות** - אם יש חוטים שרוצים לבצע את הקטע הקריטי, לבסוף חוט **כלשהו** יצליח להיכנס. ישנה התקדמות – אין Deadlock/Livelock.

תכונות רצויות:

3. **Fairness – הוגנות –** אם יש חוט **שרוצה** לבצע את הקטע הקריטי, **הוא** לבסוף יצליח. אין הרעבה. הרחבות על הדרישה:
   - **Bounded Waiting –** הגדרת **חסם** למספר הפעמים שחוטים אחרים ייכנסו לקטע הקריטי לפני החוט הנוכחי.
   - **Order –** יש סדר ברור וידוע לזמני הכניסה של החוטים הנכנסים לקטע הקריטי. דוגמה לסדר אפשרי: FIFO.

1.

א. אילו תכונות של הקטע הקריטי מפר המימוש הבא כאשר משתמשים בו במערכת עם נפילות חוטים, ולמה? הניחו שהקוד רץ על **מעבד יחיד**.

נפילת חוטים: חוט יכול ליפול באופן פתאומי, כתוצאה מחריגה למשל.

Atomic Swap: מקבלת כתובת של תא וערך חדש, ומחליפה **באופן אטומי** את תכולת התא עם הערך החדש, ומחזירה את הערך הישן.

התכונות של הקטע הקריטי שהמימוש מפר הינן:

- מניעה הדדית – במידה ומקטע קוד מנסה לשחרר את המנעול כשאינו בבעלותו (למשל לבצע unlock ורק אחרי כן לכתוב קטע קוד קריטי עם נעילה ופתיחה) הוא עשוי לעשות זאת בהצלחה, שכן אין הגבלה של בעלות על המנעול על מנת לשחרר אותו. לכן מקרה בו חוט אחד תופס מנעול ונכנס לקטע הקריטי, ומיד לאחר מכן מתבצעת החלפת הקשר, שבה החוט הזדוני מבצע פתיחה ואז מגיע לסף קטע קריטי – הוא עשוי להצליח לתפוס את המנעול ולהיכנס גם הוא.
- Progress – במידה וחוט מסוים נופל כאשר הוא מצוי בקטע הקריטי ומחזיק במנעול, ובהנחה ולא נקרא בקוד משתמש לunlock לפני שמבצעים lock, כל חוט אחר שלא נפל וינסה לעבור קטע קריטי לא יצליח לבצע זאת – זאת מכיוון שהערך במנעול נותר 0, והלולאה תמשיך לקיים את התנאי עד ביצוע של unlock שלא יתרחש תחת ההנחה לעיל.

ב. במימוש קיימת בעיית Performance, הגורמת לחוסר יעילות של זמן המעבד. ניתן להניח שהקטע הקריטי עליו המנעול מגן הינו קטע ארוך וכבד חישובית. היכן היא? האם הבעיה עדיין קימת אם הקטע היה קצר ומהיר?

- בעיית הביצועים של הקוד בבזבוז זמן מעבד של המימוש שלהלן, מתבטאת בלולאת הwhile שמתפקדת כbusy-wait בדומה למימוש spinlock שהכרנו. ההמתנה לשחרור המנעול מכלה את פיסת הזמן שהוקצתה לתהליך, על אף שעשוי לא לעשות דבר מלבד להמתין. אם ידוע שהקטע הקריטי כבד וארוך חישובית אזי המנעול עשוי להיתפס לזמן רב ובכך לגרום לתהליכים הממתינים למנעול לכלות פיסות זמן רבות במקום לתת לחוט שבקטע הקריטי לסיים מהר ככל הניתן ולשחרר את המנעול לשימושם. אילו הקוד הקריטי היה קצר ומהיר היינו מצפים לשיפור ביצועים – ישנו טריידאוף בין כילוי פיסת זמן על ידי תהליך שמבצע המתנה פעילה למנעול, לבין עלות היציאה לתור המתנה והחזרה בעת התפנות המשאב. במידה וקטע הקוד קצר המעבר מתור ריצה להמתנה עשוי להיות ארוך מהזמן שבפועל מתכלה מפיסת הזמן ולכן במקרה והקוד קצר שווה להמתין באופן פעיל אם יש סבירות גבוהה להשתחררות המנעול בזמן קצר – בדומה למניעים לשימוש בspinlock בגרעין כפי שהוצג בתרגולים.

```
class lock {
        bool lockVal;
public:
        lock(bool initVal) { lockVal = initVal;}
        void lock(){
                while(AtomicSwap(&lockVal,0)==0){}
        }
        void unlock(){
                lockVal = 1;
        }
}
```

2. ממשים מנעול חדש הכולל:
   - Mutex סטנדרטי
   - מונה count
   - סף איטרציות MAX_ITER
   - קבוע שלם T

תיאור מנגנון הנעילה: בזמן ניסיון נעילה (דהיינו, קריאה לפונקציה ()lock), המנעול תומך ב-Timeout אותו ממשים על ידי מונה בצורה הבאה: במידה והחוטים במערכת מנסים לתפוס את המנעול MAX_ITER פעמים, אך המנעול אינו שוחרר במהלך ניסיונות אלו, המנעול ישוחרר. שימו לב ש-MAX_ITER הינו define גלובלי הידוע לכל החוטים. תיאור בפסודו קוד של מימוש הפונקציה ()lock נתון בקטע הקוד הבא:

```
while ( mutex is locked ) {
        if ( mutex wasn't released yet )
                count++
                sleep T miliseconds
                if ( count == MAX_ITER )
                        unlock mutex

        else
                count=0
                lock mutex
}
```

הניחו מערכת עם **מעבד יחיד** ואפשרות לנפילת חוטים פתאומית. הניחו שה-Mutex מומש **בעזרת תור** ושומר על סדר הכניסות אליו (FIFO). זהו אגב, נקרא מנעול "הוגן". אילו תכונות של הקטע הקריטי מופרות פה?

התכונות של הקטע הקריטי שהמימוש מפר הינן:

- הערה: נניח וקיימת גם נעילה עבור החוט הראשון שמגיע – כיוון שזה לא רשום במפורש קוד, שכן הוא לא ייכנס ללולאה אך עליו לנעול כדי לאפשר מניעה הדדית, אחרת עקרון זה יופר. במידה והקוד תקין בנעילה הראשונה הבעיות יוצגו בנקודות הבאות.
- מניעה הדדית – נניח תסריט בו יש מקטע קוד ארוך בקטע הקריטי, עשויים להגיע חוטים רבים ולבקש בעלות על המנעול, אשר נמצא בבעלותו של חוט יחיד. הבקשות הרבות על המנעול עשויות להתבצע כך שההגעה לסף כמות האיטרציות הגיע והחוט המקורי עודנו בקוד הקטע הקריטי, אך המנעול ישוחרר. בכך נוכל לאפשר כניסתם של יותר מתהליך אחד לקטע הקריטי – מצב כזה עשוי למשל להתרחש בעת עומס רב של חוטים.
- Progress – במידה וחוט מסוים נופל כאשר הוא מצוי בקטע הקריטי ומחזיק במנעול, נתאר את התסריט הבא – במצב בו המונה קרוב לסף וערכו MAXITER-1 וקיימים שני חוטים נוספים במערכת. כעת נניח שאחד מהם נכנס לאיטרציה שאמורה להגדיל את המונה להיות בדיוק הסף, וגם החוט השני באותה נקודה. אזי תסריט בעייתי הוא שהחוט הראשון יגדיל את המונה, תתבצע החלפת הקשר מיד לחוט השני, וכעת גם הוא יבצע את ההגדלה של המונה. במצב זה המונה יכיל MAXITER+1 ועל כן לא החוט השני ולא החוט הראשון ישחררו את המנעול. מנקודה זו ואילך המונה רק יגדל יותר ויותר ולא תהיה אפשרות לשחרור המנעול שנתפס על ידי החוט שנפל בפנים.
- Fairness – הקידום של המונה אמנם מאפשר מכסה מסוימת של בקשות שיביאו לשחרור המשאב, אבל במידה לא בהכרח באופן שבו התכוון המתכנן המקורי. ההגדלות של המונה עשויות להתבצע כך שמתבצעת הגדלה לערך בחוט אחד, אחרי כן מתבצעת החלפת הקשר ומתבצעת הגדלה של הערך הלא מעודכן בחוט אחר, ורק אז השמה בשני , החלפת הקשר והשמה בראשון. בתסריט כזה הערך יגדל רק ב1 ולא בשניים כמו שציפינו, ובמקרה בו יש יותר חוטים יגדל רק ב1 במקום במספר החוטים. על כן המכסה הרשומה לא בהכרח תהיה מספר האיטרציות בפועל שנעשו וזה יפגום בfairness שהמתכנן המקורי ביקש ליצור.

3. בהנחה שהקוד מורץ על **מעבד יחיד**, הסבר מה ידפיס הקוד הבא, ולמה ? התשובה צריכה להיות מורכבת מערך מקסימלי אפשרי וערך מינימלי אפשרי, עם תרחיש אפשרי לכל אחד. ניתן להניח שפעולות store ו-load מתבצעות באופן אטומי.

```
int sum=0;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
int ids[10]={1,2,3,4,5,6,7,8,9,10};

void* thread_workload(void *threadID){
        int* p_val = (int*) threadID;
        pthread_mutex_lock(&mutex);
        sum += *p_val;
        pthread_mutex_unlock(&mutex);
}

int main(){
        pthread_t t;
        int i;
        for(i=0;i<10;++i)
                pthread_create(&t,NULL, thread_workload,(void*)(ids+i));
        pthread_join(t,(void**(&i));
        printf("%d\n", sum);
        return 0;
}
```

- הקוד שלעיל עשוי להניב אוסף של ערכים בקטע $[10,55]$ ונסביר את התסריטים עבור הערך המקסימלי והמינימלי
- 10 – עשוי להתקבל עבור תסריט בו החוט הראשי רץ ומייצר את החוטים האחרים, כאשר לא מתבצעת החלפת הקשר לאף חוט אחר באמצע. בהגעתו לפקודה join החוט הראשי בלית ברירה מבצע החלפת הקשר. במידה והחוט ה10 הוא הראשון שמקבל את המעבד לריצה מבין שאר החוטים (כלומר מיד לאחר שהחוט הראשי ויתר עליו) ויבצע את הוספתו למונה הגלובלי ויגדיל אותו באופן בטוח ללא החלפות בדרך, הערך יהיה 10. לאחר סיומו אם החלפת ההקשר תחזיר את המעבד לחוט הראשי שהתעורר כתוצאה מסיום חוט 10, הוא יוכל להדפיס את ערכו הקיים והוא 10. במידה ויתבצע מעבר אחר – מחוט ראשי לחוט שאינו 10 והוא יגדיל את המונה, יגרום לכך שהמונה שידפיס החוט הראשי לבסוף יהיה רק גדול או שווה לו.
- 55 – זהו הסכום של כל ערכי המערך. תסריט אפשרי הוא כזה שבו באופי הריצה הוא כך שברגע יצירת חוט מתבצעת החלפת הקשר ועוברים לביצוע מקטע הקוד שלו. כך כל חוט שנוצר מיד יצליח לבצע את העלאת המונה ואז נחזור לחוט הראשי ליצירת החוט הבא. על ידי כך נקבל העלאה של המונה עד לסכום המקסימלי. לבסוף נחזור לחוט הראשי ונבצע join על החוט ה10 אלא שהוא כבר סיים ולכן נמשיך מיד להדפסת הערך המקסימלי במונה 55.
- שאר ערכי הביניים יתקבלו מכל אפשרות להחלפת הקשר בה חוט 10 מצוי במקום ביניים במעברים השונים, כך שהחוט הראשי יהיה בעל האפשרות להדפיס, ולקבל ערך שמורכב מ10 ומספר תאים נוספים במערך שזכו למעבד בדרך.

4. בהנחה שהקוד מורץ על **מעבד יחיד**, הסבר מה ידפיס הקוד הבא, ולמה ? התשובה צריכה להיות מורכבת מערך מקסימלי אפשרי וערך מינימלי אפשרי, עם תרחיש אפשרי לכל אחד. ניתן להניח שפעולות store ו-load מתבצעות באופן אטומי.

```
int result;
void* do_calc();
        int i;
        for(i=0; i<100 ; ++i)
                result=result+1;
int main(){
        pthread_t threads[2];
        int i;
        result =0;
        for(i=0;i<2;++i)
                pthread_create(&threads[i],NULL,do_calc,NULL);
        for(i=0;i<2;++i)
                pthread_join(threads[i],NULL);
        printf("%d\n", result); return 0;
}
```

- הקוד שלעיל עשוי להניב אוסף של ערכים בקטע [2,200] ונסביר את התסריטים עבור הערך המקסימלי והמינימלי
- 200 – עשוי להתקבל עבור תסריט בו החוט הראשי רץ ומייצר את החוטים האחרים, כאשר לא מתבצעת החלפת הקשר לאף חוט אחר באמצע. בהגעתו לפקודה join החוט הראשי בלית ברירה מבצע החלפת הקשר. במידה והחוט 1 הוא הראשון שמקבל את המעבד לריצה מבין שאר החוטים (כלומר מיד לאחר שהחוט הראשי ויתר עליו) ויבצע את הוספתו למונה הגלובלי ויגדיל אותו באופן סדרתי ללא החלפות בדרך, הערך יהיה 100. לאחר סיומו אם החלפת ההקשר תחזיר את המעבד לחוט הראשי שהתעורר כתוצאה מסיום חוט 1, הוא יגיע לפקודת join השנייה ויעביר את המעבד לרשות החוט 2. בתורו הוא יגדיל את המונה באופן סדרתי עד שיגיע ל200 ויסיים את הריצה. לאחר מכן בחזרה לחוט הראשי יודפס הערך המקסימלי 200.
- 2 – מצב זה עשוי להתקבל עבור תסריט בו חוט אחד מחשב את החישוב הראשוני ונעצר בשלב ההשמה, החלפת הקשר מביאה לחוט השני את המעבד ומבצע את כל פעולתו כמעט עד הסוף – ולפני האיטרציה ה100 חוזרים לחוט הראשון. בחזור לחוט הראשון הוא מבצע השמה לערך 1. נניח שבשלב זה חוזרים שוב לחוט השני שבאיטרציה האחרונה שלו לוקח את הערך 1 ומגדיל אותו ל2 ולפני ההשמה שוב מתבצעת החלפת הקשר. משם ימשיך וימלא הראשון את המונה עד סוף ריצתו. לבסוף יגיע החוט השני ויסיים את ריצתו על ידי השמה של הערך 2 למונה.
- שאר ערכי הביניים יתקבלו מכל אפשרות אחרת להחלפת הקשר במהלך ההרצה, איטרציות בהן התהליך המתואר יקרה ואיטרציות שתבוצע עבורן הגדלה סדרתית ייתנו ערכים בתחום שלעיל.

5. הסבירו למה אין צורך להגן על sum בעזרת משתנה סנכרון כמו Mutex או Semaphore.
הניחו ש-sum הינו משתנה גלובלי.

```
int sum = 0;

if( fork() ) {
   sum = sum+5;
} else {
   sum = sum +1;
}
```

- המצב המתואר אינו יצירת חוט אלא יצירת **תהליך** בן חדש. במצב זה מרחבי הזיכרון מועתקים לגמרי ואין זה כלל אותו משתנה sum עבור שני התהליכים. ברגע הכניסה של כל אחד לבלוק המתאים לו, יעודכן הערך השמור בזיכרון כל אחד מהתהליכים לערך המתאים, ולא יהיה חשש לדריסה – כזכור תהליכים לא משתפים זיכרון (ולכן אין חשש לגישה לא מסונכרנת למשאב משותף), ובעת השכפול fork משכפלת מרחבי זיכרון שרק בעת כתיבה מועתקים ממש וערכם משתנה – מנגנון COW.

חלק שני: Singlephore

לרוב מנגנוני הסנכרון עליהם למדתם, קיימים לפחות שתי פעולות. מנעולים פשוטים תומכים ב-lock ו-unlock.
משתני תנאי תומכים ב-wait ו-signal, וסמפורים ב-up ו-down או בשמם המקורי בספרות, P ו-V.
בתרגיל זה תעבדו עם מנגנון סנכרון שלו **תמיכה רק בפעולה אחת ויחידה**, ונקרא – **singlephore**.

הגדרת פעולות של המנגנון:

```
typedef struct singlephore {
   int value;
} singlephore;

// Initialize the singlephore to value 0.
void singlephore_init(singlephore * h) {
   h->value = 0;
}

// Block until the singlephore has value >= bound, then atomically increment its value by delta.
void H(singlephore * h, int bound, int delta) {
// This is pseudocode; a real singlephore implementation would block, not
// spin, and would ensure that the test and the increment happen in one
// atomic step.
   while (h->value < bound) {
      sched_yield();
   }
   h->value += delta;
}
```

ברגע שה-singlephore אותחל, קוד אפליקציה יגש אליו רק דרך הפעולה H.

א. ממש מנעול למניעה הדדית בעזרת singlephore. מלא את תבניות הקוד הבאות:

```
typedef struct mutex {
   singlephore h;
} mutex;

void mutex_init(mutex* m) {
   singlephore_init(&(m->h));
}

void mutex_lock(mutex* m) {
   H(&(m->h),-1,-1);
}

void mutex_unlock(mutex* m) {
   H(&(m->h),-2,1);
}
```

הערך 2- הוא שרירותי כיוון שהוא מאפשר לבצע העלאה חזרה ל0 ללא כניסה ללולאה – כלומר אנו נקבל תוצאה זהה עבור כל ערך bound בקריאה זו שקטן מ1-. נציין שכאן ניתן לבצע שחרור של המנעול על ידי חוט שלא מחזיק בו בבעלותו – למשל קריאה כלשהי לunlock בטרם ביצוע lock בדומה למימוש של מנעול על ידי סמפור בינארי.

<u>סעיף בונוס</u> (**5 נקודות**): ממש משתנה תנאי בעזרת singlephore ו-mutex (שכבר מימשתם). מלא את תבניות הקוד הבאות: (שימו לב, הסעיף הבא **<u>אינו</u>** סעיף בונוס, אך יכול לעזור לפתרון סעיף זה).

```
typedef struct condvar {
   mutex m;
   singlephore h;
   int num_wait;
} condvar;

// Initilize the condition variable
void cond_init(condvar* c) {
            singlephore_init(&(m->h));
            num_wait=0;
}

// Signal the condition variable
void cond_signal(condvar* c) {
            //Only when there are waiting threads
            mutex_lock(&c->m);
            if(c->num_wait>0)
                        H(&c->h, INT_MIN, 1);
            mutex_unlock(&c->m);
}
```

```
// Block until the condition variable is signaled. The mutex m must be locked by the
// current thread. It is unlocked before the wait begins and re-locked after the wait
// ends. There are no sleep-wakeup race conditions: if thread 1 has m locked and
// executes cond_wait(c,m), no other thread is waiting on c, and thread 2 executes
// mutex_lock(m); cond_signal(c); mutex_unlock(m), then thread 1 will always recieve the
// signal (i.e., wake up).
void cond_wait(condvar* c, mutex* m) {
        mutex_unlock(m);

        mutex_lock(&c->m);
        c->num_wait++;
        mutex_unlock(&c->m);

        H(&c->h, 1, -1);

        mutex_lock(&c->m);
        c->num_wait--;
        mutex_unlock(&c->m);

        mutex_lock(m);
}
```

רמזים:

1. אם אין חוט שמחכה על משתנה התנאי c, אז cond_signal(c) לא יעשה דבר.
2. הנח ש-N חוטים ממתינים על משתנה התנאי c. אז N קריאות ל- cond_signal(c) הם תנאי הכרחי ומספיק על מנת להעיר את כולם.
3. יתכן ותוכל להיעזר בסעיף הבא כדי למצוא את הפתרון הנכון
4. ניתן ורצוי להשתמש בקבוע INT_MIN, הערך הנמוך ביותר ש-integer יכול לקבל.

ב. ג'ון סנו החרוץ מתלמידי הקורס, סיפק את הפתרון הבא לסעיף ב':

```
typedef struct condvar {
   singlephore h;
} condvar;

void cond_init(condvar* c) {
   singlephore_init(&c->h);
}

void cond_signal(condvar* c) {
   H(&c->h, INT_MIN, 1);
}

void cond_wait(condvar* c, mutex* m) {
   mutex_unlock(m);
   H(&c->h, 0, -1);
   mutex_lock(m);
}
```

מה לא תקין בפתרון? הראו תרחיש אפשרי בו פתרון זה לא עומד בתנאים של סעיף ב'.

הבעיה העיקרית נובעת מהעובדה שsignal כן משנה את המונה הפנימי של הסינגלפור – למרות שלא בהכרח צריך לצבור אותם. זה ישפיע על התקינות עבור הbound שנקבע בפונקציה cond_wait.

נניח ויש משתנה מצב שמחזיק בתוכו דגל על קיומו של ערך חדש לקריאה – 1 ברגע עדכון, וברגע שקוראים אותו יש לשנות את הדגל ל0.

נניח תסריט ובו יש חוטים רבים שכותבים לאותו מקום ובכל פעם יעדכנו את ערך משתנה המצב ל1, ויבצעו signal. כעת נניח שיש לנו חוט שקורא לאחר הרבה כתיבות. בפעם הראשונה שינסה לקרוא יהיה 1 והתנאי יתקיים ולכן יוכל להתקדם בהצלחה וישנה את ערך הדגל ל0. בפעם השנייה שיקרא (נניח לא היו כתיבות בדרך) התנאי לא יתקיים ויצטרך ללכת להמתנה. מה שיקרה בפועל זה שהקריאה לwait לא תשלח אותו להמתנה, אלא תגרום לו לבצע את הפתיחה החסרת 1 ונעילה כמות פעמים ששווה לכמות הכתיבות שנעשו, זאת לאור העובדה שהמונה הפנימי גדולה ממש מ0, ועל כן התנאי על היות המונה קטן מbound לא יתקיים – ותתבצע הקטנה מיידית בלי ויתור על המעבד. במצב כזה אנו במקום להמתין עושים busy-wait .

חלק שלישי: ניתוח של החלק הרטוב וחוק אמדל

חלק זה מבוסס על חלקו הרטוב של תרגיל בית 3, ומיועד לפתרון לאחר סיום חלק זה. במידה והסתבכתם, ניתן גם להיעזר בחלק זה לשם פתרון החלק הרטוב.

שאלה 1: ניתוח החלק הרטוב

פיראס החרוץ מתלמידי הקורס ביסס מנגנון סנכרון בין Producer-Consumer שלו הוא קרא "Barricade":

```
class {
private:
        int working;
public:
        Barricade(){
                working =0;
        }
        increase(){
                working++;
        decrease(){
                working--;
        }
        wait(){
                while(working!=0){}
        }
};
```

השימוש במנגנון היה כדלקמן:

Producer:

1. Init Barricade b
2. Init PCQueue p
3. Init fields $curr, next$
4. for t=0 $\rightarrow$ t=n_generations

    $for\ i = 0 \rightarrow i = N$

        p.push(job);

        b.increase();

    b.wait();

    swap($curr, next$);

Consumer (One of N)

while(1)

    $job\ \ j =$ p.pop() $//\ blocked\ here\ if\ queue\ is\ empty$

    execute j

    b.decrease();

הנחות:

1. חלק מהפסודו קוד שניתן לכם במסגרת התרגיל הרטוב הושמט. השאלה מתייחסת רק למנגנון הסנכרון.
2. התור מעלה הינו אותו תור יצרן-צרכן שהתבקשתם לממש בתרגיל הרטוב.
3. job הינו struct אשר מתאר לחוט כלשהו את גבולות הגזרה עליהם עליו לרוץ.

א. הסבירו את כוונותיו של פיראס – איך היה אמור המנגנון לעבוד? מה ההבדל העיקרי בין מנגנון זה לבין ה-semaphore עליו למדתם בביתה?

- פיראס התכוון לממש מונה שממתינים להתאפסותו, כך שבכל סיום עבודה של חוט הוא יקטין את המונה עד שיגיע ל0 וברגע שיגיע לאפס הלולאה בקריאה לפונקציה wait תסתיים, וחוט הראשי יוכל להמשיך בקוד ולהחליף בין הלוחות שכן עודכנו לגמרי. ההבדל העיקרי בין מנגנון זה לבין סמפור הינו שאנו לא מגבילים את מספר החוטים בקטע הקריטי לסף מסוים, אלא אנו מגבילים חוטים מסוימים מלהיות בקטע הקריטי עד שכל החוטים שרצים מסיימים ומאפשרים לו להיכנס – על כן המשמעות היא שתחת מכסה של חוטים בקטע הקריטי (נניח n) לא מספיק שיתפנה מקום אחד כדי שחוט הראשי יוכל להיכנס כמו בסמפור אלא יש לדרוש שכולם יפנו את הקטע הקריטי כדי שיוכל להמשיך בריצתו. זוהי אינה המתנה לקטע הקריטי בתור המתנה אלא זו המתנה ב-busy wait כך שהיא מבזבזת זמן מעבד.

ב. במימוש זה מספר בעיות **Correctness**.

1. מצאו בעיה אחת של **Race Condition** בפתרון. הסבירו. בכלליות – RC מתרחש כאשר 2 או יותר חוטים בעלי משאב משותף רצים במקביל, ושקיים סדר תזמונים (scheduling) ביניהם **המשבש את לוגיקת הקוד** ביחס למשאב המשותף.

- הבעיה העיקרית של race-condition שעשויה להתרחש בקוד זה הוא בהקטנת המונה לאחר סיום ריצתו של חוט על עבודה מסוימת. נניח ובסיום העבודה מתחיל החוט בפועל ההקטנה שנקראת ולוקחת את הערך הקיים מחסירה ממנו 1, ורגע לפני ההשמה מתבצעת החלפת הקשר. כעת המעבד עובר לחוט אחר המבצע את עבודתו וגם הוא בא להקטין את המונה בסופה. הוא לוקח את הערך הקיים שעוד לא עודכן ומקטין אותו ב1 ולאחר מכן מבצע השמה. אחרי כן אם תתבצע החלפת הקשר ונחזור לחוט המקורי ההשמה שלו תתבצע והחוט יקטן ב1 במקום ב2 . מצב זה יוביל לכך שהמונה לא יקטן מספיק למרות שכל העבודות יסתיימו – מצב זה יוביל להמתנה של החוט הראשי לעד כלומר הrace-condition יגרום גם לdeadlock

2. תארו תרחיש שבו מופר ה-**Mutual Exclusion**. דהיינו, חישוב הלוח curr טרם הסתיים, וה-Producer מבצע למרות זאת את ה-swap של הלוחות.

- תסריט אפשרי שיביא להחלפה לא תקינה הינו הבא- נניח והחוט הראשי רץ עד הדחיפה של המשימה הראשונה לתור המשימות. נניח כי בתום ההכנסה והגדלת המונה בדיוק הגיעה החלפת הקשר ועברנו לחוט שיבצע לה POP ויעשה אותה. נזכור כי כעת המונה הוגדל פעם אחת ועל כן הוא ב1, כעת נניח שנעשה החישוב לערך שצריך להיות מוקטן כלומר מתקבל 0 ומתקבלת החלפת הקשר בדיוק בהשמה. כעת נניח שהחוט הראשי ידחוף ויגדיל ללא הפרעות וללא מעבר לחוטים אחרים עד סוף הלולאה. כעת נניח שבסוף הלולאה תבוא החלפת הקשר ותחזיר אותנו בדיוק לחוט הראשון לשלב ההשמה, אזי למונה יוכנס 0 ותסתיים ריצתו. בהחלפת ההקשר שלאחר מכן נניח שחזרנו לחוט הראשי וביצענו wait בדיוק כשהמונה 0 – כך נפסח על ההמתנה למרות ששאר החוטים לא התחילו אפילו את ביצוע העבודה, אך אנחנו כבר נבצע swap על לוח לא תקין – כך פגענו במניעה ההדדית.

3. תארו שני תרחישים שונים בהם יתכן **Deadlock** בפתרון.

- התסריט שהוזכר בסעיף 1 מביא גם לdeadlock כפי שהוזכר
- תסריט נוסף המתבסס על עקרונות דומים הוא תסריט בו כל העבודות נדחפות ברצף ומוגדל המונה ללא החלפת הקשר. מגיעים להמתנה ומתבצעת החלפת הקשר. נניח כי החוט הראשון אליו עוברים מבצע את עבודתו ובא להקטין את המונה, מחשב את ערך המונה החדש כמספר העבודות פחות אחד ולפני ההשמה מתבצעת החלפת הקשר לחוט אחר. נניח כי מעתה כל חוט מסיים את עבודתו באופן סדרתי ומקטין את המונה. המונה הסופי שיתקבל הוא 1. בחזרה לחוט הראשון נקבל שהערך שירצה הוא מספר העבודות המקורי פחות 1 ונגיע למצב בו נגמרו העבודות וערך המונה הוא רק קטן באחד מהערך המקורי ומכאן שנמתין לעד בחוט הראשי כיוון שלא יוכנסו עוד עבודות.

ג. תקנו את class Barricade ואת הפסודו קוד של היצרן-צרכן כך שכל בעיות ה-Correctness יפתרו. אין לשנות את מתווה הפתרון של פיראס באופן מהותי.

1. הראו פתרון אחד עם הוספה של mutex יחיד, וקריאות מתאימות שלו בתוך ה-Class והפסודו קוד.

```
class {
private:
        int working;
        pthread_mutex_t m_lock;
public:
        Barricade(){
                working =0;
                pthread_mutex_init(&m_lock);

        }
        increase(){
                pthread_mutex_lock(&m_lock);
                working++;
                pthread_mutex_unlock(&m_lock);

        decrease(){
                pthread_mutex_lock(&m_lock);
                working--;
                pthread_mutex_unlock(&m_lock);

        }
        wait(){

                pthread_mutex_lock(&m);
                while(working!=0)
                {
                        pthread_mutex_unlock(&m);
                        // Window for context switches to happen to enable jobs to be done
                        pthread_mutex_lock(&m);
                }
                pthread_mutex_unlock(&m);

        }
};
```

אין תוספת של קוד לאלגוריתם שלעיל רק למחלקה עצמה.נציין כי ההגנה על קריאת waiting בצורה זו מדמה משתנה תנאי בצורה קצת פחות חסכונית אבל נאמנה לדרישות, לצורך אפשרות קריאה בטוחה מצד אחד ומצד שני אפשרות להתקדמות העבודות במקביל להמתנה זו

2. מנגנון הסנכרון שפיראס ניסה ליצור, ואותו אתם השלמתם בסעיף 1' נקרא "מונה משותף", ואינו מוצלח במיוחד ממבט של בביצועים - Performance. הסבירו מדוע. התייחסו לחסרון המנגנון כאשר N, מספר החוטים האפקטיבי גבוה מאוד.

- המנגנון בעייתי כאשר מבצעים wait בחוט הראשי. המעבד מריץ לולאת המתנה פעילה ולא מאפשר לחוטים אחרים להתקדם כל עוד היא רצה. זהו בזבוז זמן רב למעבד שמכלה את זמן הריצה של החוט הראשון במקום ללכת להמתין עד שחוט יבוא ויעדכן את הערך, כך שהבדיקה תעשה רק בעת עדכונים. כאשר עומס גבוה במערכת חוטים רבים ימתינו בכל לולאה שכזו כך שנוצר צוואר בקבוק בהעלאת המונה שגורם להם להמתין פיסות זמן שלמות לריצה לאור בדיקה פעילה שכמובן תחזור על עצמה ללא שינוי במידה ולא תתבצע החלפת הקשר בדרך ונישאר בחוט הראשי.

3. הראו פתרון נוסף בעזרת שימוש בפעולות אטומיות. הפעולות האטומיות שעומדות לרשותכם:
   - atomicAdd(int * ptr,int val) אשר מוסיפה **<u>באופן אטומי</u>** val לערך התא ptr
   - atomicCAS, עם תיאור הפעולה בקוד:

```
int CAS(int *ptr,int oldvalue,int newvalue)
{
   int temp = *ptr;
   if(*ptr == oldvalue)
       *ptr = newvalue
   return temp;
}
```

   - הניחו שפקודות אלו ממומשת **<u>בחומרה</u>**, ולא בתוכנה, כיאה לפעולות אטומיות.
   - **<u>אין</u>** להשתמש במנגנוני סנכרון נוספים, למשל mutex או semaphore, אך ניתן לנצל busy wait.

```
class {
private:
        int working;
public:
        Barricade(){
                working =0;
        }
        increase(){
                atomicAdd(&working,1);

        decrease(){
                atomicAdd(&working,-1);
        }
        wait(){
                while(atomicCAS(&working,0,0)){}
        }
};
```

- atomicCAS(&working,0,0) מהווה בדיקת השוואה של הערך הקיים במונה ל0, ומחזירה את המספר אם לא מה שיגרום לריצת הלולאה ואם 0 היא תעצר וההמתנה תיגמר. התנאי בכך הופך להיות ממומש באופן אטומי

4. הסבירו למה הפתרון ב-(3) עדיף על פתרון ב-(1).

- היתרון העיקרי של שימוש בפעולות אטומיות למימוש הוא בזמן שנחסך במעבר לתור המתנה למנעול על ידי כל אחד מן החוטים שמחכה להעלות את המונה. מעבר לתור המתנה לוקח זמן ולחזור לתור הריצה גם כן. על כן מימוש הנמנע מתורי המתנה וגם מנצל busy-wait עבור ההמתנה לסיום יהיה מהיר יותר מאשר המתנה למנעולים בתורי המתנה כיוון שאף תהליך לא יצא מתור הריצה – הקטעים הקריטיים הארוכים מנצלים את הפקודות האטומיות, והקטע הקצר שבא לאחר wait של החלפת הלוחות מבצע התנהגות של spinlock מכיוון שSWAP היא פקודה .

ד. **בונוס (10 נקודות):** הציגו מנגנון סנכרון המנצל **שני** PCQueues כקופסא שחורה לטובת הסנכרון ההדדי בין Producer ל-Consumer. הסבירו בפירוט את פתרונכם.

- **אין** להשתמש בכל מנגנון סנכרון נוסף (בפרט semaphore,mutex, atomics, ו-busy wait).
- מומלץ להחליף קטעים רלוונטיים בפסודו קוד לשם הצגת פתרונכם.
- פתרון זה אינו "מינימלי" – דהיינו, הוא מבצע פעולות מיותרות בצד ה-Producer אשר אינן דרושות לנכונות המנגנון. בסביבה בה ביצועי המנגנון הינו דבר קריטי, דבר זה מהווה בזבוז מהותי. הציגו מנגנון שקול **ומינימלי**, אשר אינו מבצע פעולות אלו. ניתן להשתמש ב-mutex יחיד, condvar יחיד ו-PCQueue יחיד.

המימוש הראשוני עם שני תורים הינו

```
Producer:

Init PCQueue p, c
Init fields curr, next
for t=0:n_generations
      for i=0:N
            p.push(job)
      for i=0:N
            c.pop()
      swap(curr, next)

Consumer:

while(1)
      job j = p.pop()
      execute j
      c.push(j)
```

בסוף כל עבודה הצרכן דוחף את העבודה שסיים וביצרן הוא מוודא שכל העבודות הגיעו לסיומן. כשאין עבודות הוא ימתין בPOP וכך יגיע בבטחה לסוף הדור כשהלוח מעודכן באופן תקין. המימוש המינימלי הינו

```
Producer:

Init int count
Init cond_t cond_count
Init mutex_t m_lock
Init PCQueue p
Init fields curr, next
for t=0:n_generations
      count = N
      for i=0:N
            p.push(job)
      lock(m_lock)
      while(count > 0)
            cond_wait(cond_count,lock)
      unlock(m_lock)
      swap(curr, next)

Consumer:
while(1)
      job j = p.pop()
      execute j
      lock(m_lock)
      count--
      cond_signal(cond_count)
      unlock(m_lock)
```

שאלה 2: ביצועים וחוק אמדל

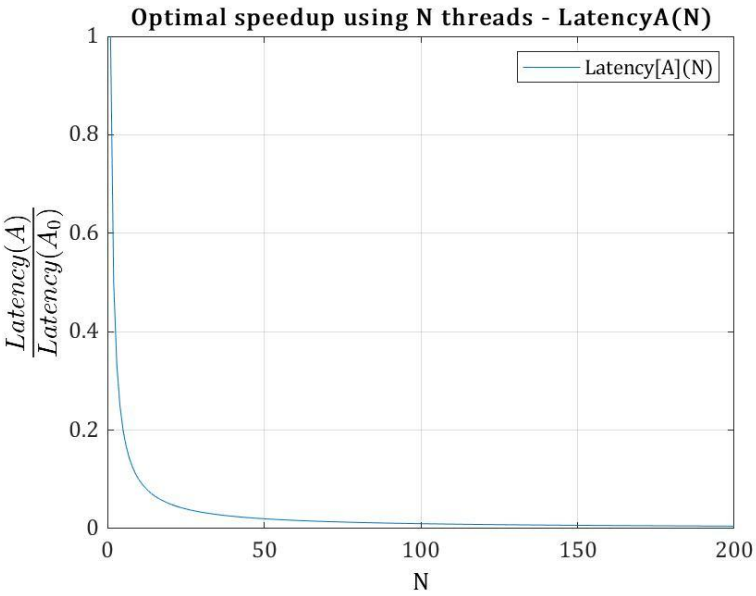
נניח אלגוריתם A המורכב ממספר רב של עבודות $j_1, j_2, \ldots, j_M$

לנוחיותכם, מושגים נפוצים לניתוח Performance של האלגוריתם:

i. **Latency(A)** – זמן החישוב הכולל של האלגוריתם A
ii. **Latency(j)** – זמן החישוב הכולל של עבודה j
iii. **Turnaround Time(j)** – זמן החישוב + זמן ההמתנה בתור של עבודה j
iv. **Throughput – תפוקה** – מספר העבודות המסתיימות ביחידת זמן

בשאלה זו יש לצייר מספר גרפים, ולהסבירם. על הגרפים לכלול כותרת, מקרא, שמות צירים והסבר קצר על מה התקבל בגרף ולמה. ניתן להשתמש בכל תוכנה ליצירת גרפים שתרצו, למשל Desmos, Excel, Python או Matlab.

א. הניחו ש-A ניתן למקבול באופן מלא. ציירו גרף של ה-Latency(A) כתלות במספר החוטים N, עפ"י חזונו של אמדל (Amdahl).



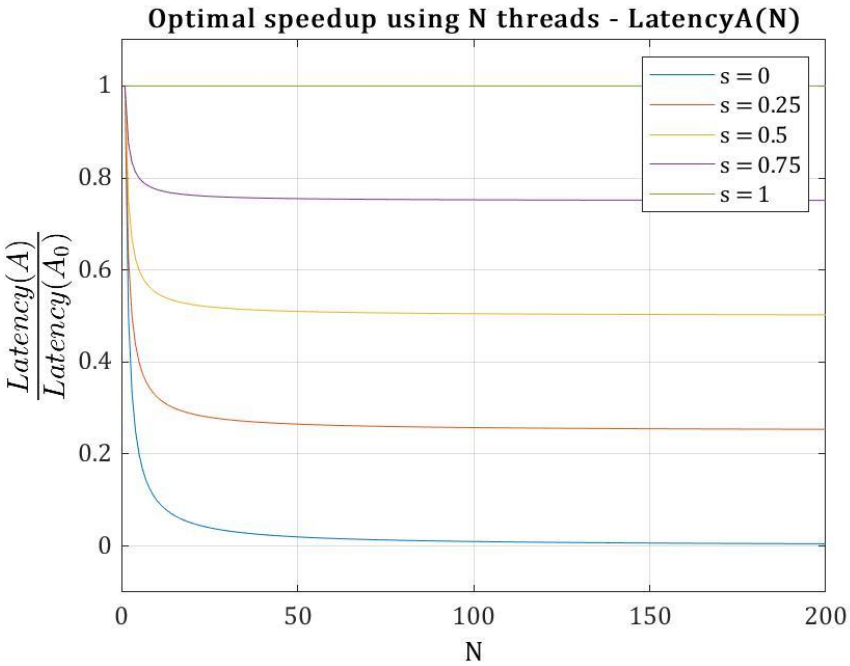
גרף זה נובע ישירות מחוק אמדל. אם האלגוריתם מקבילי לגמרי כל זמן ריצה שיהיה יתחלק בין מספר חוטים והזמן הכולל לריצה היא מנת חלקו של כל חוט בהרצת מקטע הקוד שלו כלומר

$$T_{tot} = \frac{T_{serial}}{N}$$

נציין כי הנקודות על הגרף הן בדידות, על מספרים טבעיים של חוטים.

ב. הוסיפו לגרף בסעיף **א'** מעלה עקומים המתארים גם A סריאלי לחלוטין, ו- A המורכב מחלק של $s = \{0.25, 0.5, 0.75\}$ שניתן למקבול.

- סעיף זה נועד לספק לכם מדד איכותי לגרפים התיאורטיים שיתקבלו בתנאי "מעבדה", בהתאם לכמה האלגוריתם סריאלי/מקבילי.
- ניתן לקבע את זמן הביצוע הכולל אם A היה סריאלי לחלוטין כרצונכם (למשל – שניה).



חוק אמדל אומר כי בעבור קוד שאינו ניתן לחלוקה באופן מקבילי לגמרי, יש תלות באורך הקטע שיש חובה להריץ באופן סדרתי – קטע קריטי. s מייצג את הנתח של הקוד הקריטי מסך הקוד השלם ונותן חסם על מדד השיפור בביצועי הקוד. ההתכנסות לזמני הריצה השונים נובעת מחוסר היכולת לבטל מקטעי קוד קריטיים שירוצו באופן סדרתי, ועל כן בהגדלת ערכי N הזמן שיישאר לאחר כל השיפור הוא רק הזמן שייקח לאותם מקטעי קוד לרוץ אחד אחרי השני באופן סדרתי בקוד הכולל.

ג. עתה נתפנה לנתח את התוצאות המתקבלות מהאלגוריתם שכתבתם בחלקו הרטוב של התרגיל.

נגדיר את A כחישוב לוח משחק יחיד, פעולה הנעשת ע"י N חוטים במקביל. נגדיר חישוב כל Tile כעבודה j. הריצו שלושה עומסים שונים על המערכת: small.txt, mid.txt, big.txt וציירו לכל אחד שני גרפים:

i. גרף של Average Latency(A) כתלות במספר החוטים N.

ii. גרף של Average Latency(j) כתלות במספר החוטים N.

iii. גרף scatter plot של Tile Index אל מול Thread Id, כאשר Index Tile הינו מספור של כל ה-Tiles השונים לפי סדר ההוספה שלהם להיסטוגרמה m_tile_hist, ו-Thread Id הוא המספר הסידורי של החוט שביצע את העבודה על ה-Tile. שימו לב – עליכם לאפשר הדפסה של וקטור זה ע"י שינוי main.cpp.

- יש תחילה לכבות את דגל ההדפסה print_on על מנת שהזמנים השונים לא יושפעו מתהליך ההדפסה. יש להעביר לארגומנט האחרון ל-main N במקום Y.
- יש להריץ כל קונפיגורציה ל-100 Generations, עם מספר חוטים משתנה: $N = 1, 2, 3, \ldots, \min\left(100, N_{effective}\right)$, ולהשתמש ב-Avg Gen Time ו-Avg Tile Time המתקבל ב-results.csv. ניתן לכתוב סקריפט bash קצר לביצוע דבר זה.
- ניתן לטעון קבצי CSV (Comma Separated File) ישירות לאקסל באופן פשוט וקל ע"י פקודת Import.

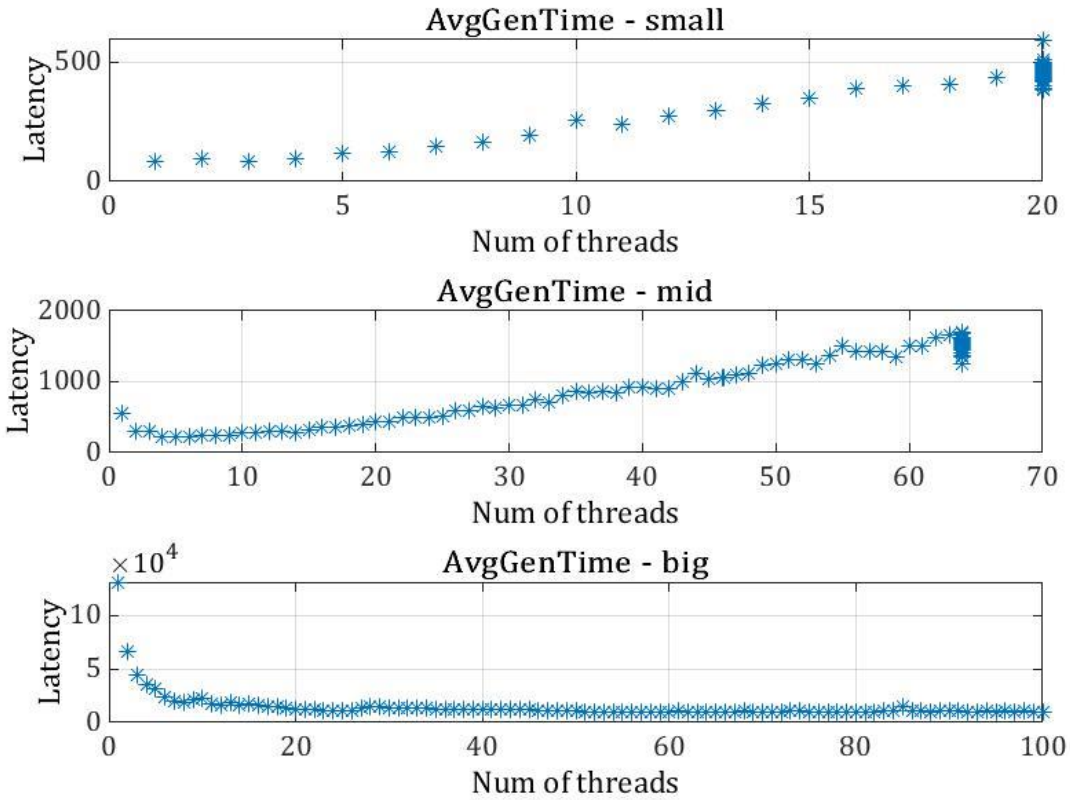
ד. נתחו את הגרפים שהתקבלו באופן **מעמיק:**

שאלות מכוונות אליהם התייחסו בניתוח:

1. האם בחלקים מסוימים ניתן לראות מגמה כזאת או אחרת? עליה/ירידה/קו שטוח ? ממה הדבר נובע לדעתכם?
2. מהו מספר החוטים האידיאלי לכל עומס ? הסבירו סיבות לכך.
3. האם בהכרח זמן החישוב היה משתפר אם היינו מוסיפים **מעבדים** נוספים?
4. כיצד ניתן להסיק מהגרף השלישי **(tid vs tile id)** את מספר הליבות? מה גרף זה מספר לנו על יעילות הקוד שכתבתם? נסו לספק שערוך מספרי למספר הליבות הקיימות בסביבת ההרצה (למשל, שרתי CSL3 או LUX. אם הרצתם על מחשבכם האישי – מספר הליבות הניתן להסקה **מהגרף**). נסו לספק שערוך לגודל ה-time slice של כל חוט מתצורת הגרף.
5. השוו בין הגרפים של העומס הקטן, גדול ובינוני. במידה ויש שוני, ממה נובע השוני בין הגרפים של העומסים השונים?
6. האם הגרף מתנהג כמו אחד הגרפים שהתקבלו בסעיפים א',ב' באופן גס? אם כן, כמה מקבילי אתם מעריכים שהקוד שלכם?

שימו לב: הגרפים שיתקבלו בסעיף זה יכולים להיות שונים ומגוונים. לא בהכרח שהגרפים יסתדרו עם הציפיות שלכם. במידה ומתקבלים גרפים המתארים התנהגות לא "מקבילית" – בדקו את מימושכם עד שהשתכנעתם שהוא סביר. בכל מצב, הצדיקו את הגרפים שהתקבלו עם טיעונים איכותיים. הניקוד בחלק זה ינתן עבור הסברים משכנעים של התוצאות בגרף, המראים הבנה של החומר ושיקולי המערכת.

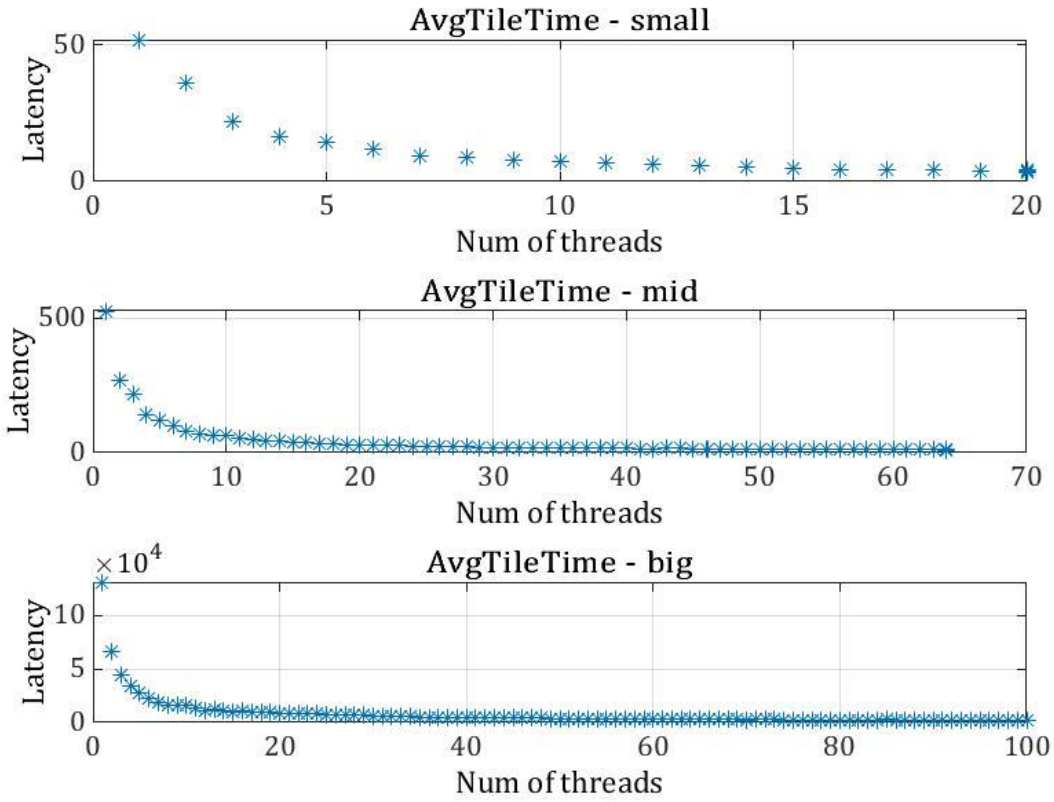
במידה והייתם רוצים לספק גרפים על מטריצות אחרות או הגדרות אחרות – ניתן ורצוי, במידה והם מסייעים בהסבר שלכם לשאלה מעלה.



ננתח את תוצאות הגרף המשווה בין זמני הדורות בין קלטים שונים :

- קטן:
  - כיוון שכל משבצת מבצעת החלפות הקשר הזמן שלוקח בפועל לתוכנית ולכל דור בדרך להתבצע עולה. תקורת החלפות ההקשר עולה יותר מלבצע הכל באופן סדרתי.
- בינוני
  - עליה באורך של דור – באופן דומה לקצר
- גדול:
  - ירידה באורך של דור – אורך של דור יורד לאור העובדה שבכמות גדולה של מידע כל משבצת של חוט היא גדולה מאוד, ועל כן עלות החלפות ההקשר ביחס לזמן הריצה של משבצת הוא די קטן ולכן המקביליות היא גורם שיפור משמעותי לעומת התקורה שזניחה ביחס למידע העצום שכל חוט מחשב בעצמו

אנו רואים כי הגרף היחידי שממתנהג כמו חוק אמדל הוא הגרף הגדול, בעוד הקטנים לא משפרים את יכולתם. זאת לאור העובדה שהחוק לא מתחשב בתקורת החלפות ההקשר ולא מוסיף אותה כפקטור למידת השיפור האפשרית בקוד המקבילי שמורץ.



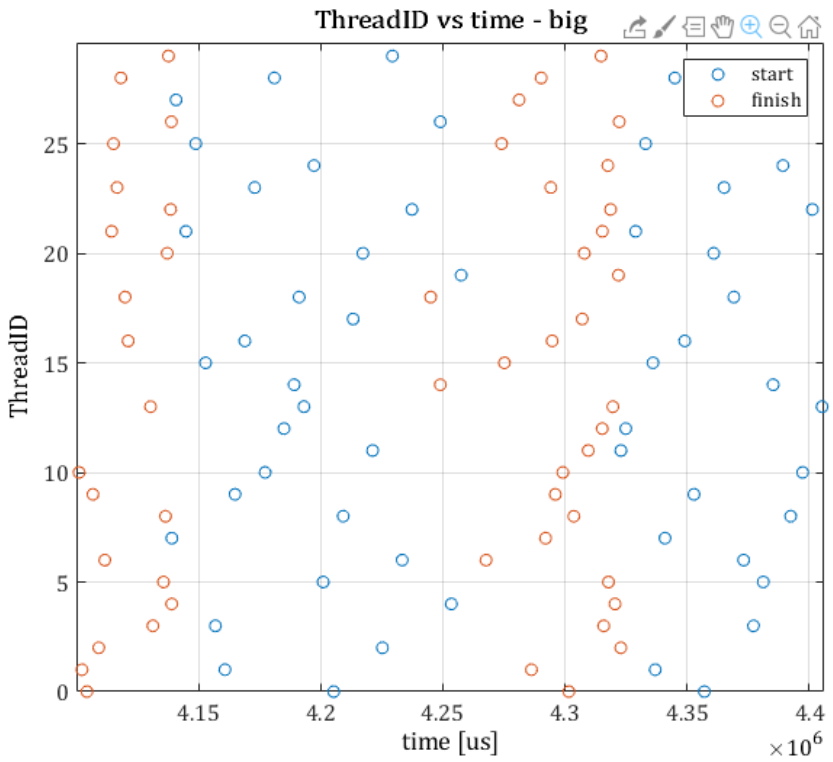
ננתח את תוצאות הגרף המשווה בין זמני המשבצות בין קלטים שונים:

- קטן בינוני וקטן:
  - ירידה באורך של משבצת – משבצת הופכת לקטנה וקטנה יותר עד שנגיע לגודל של שורה וברור שבעת ביצועה ייקח פחות זמן לגודל זיכרון קטן יותר להתעדכן, אבל יהיה מספר גדול יותר שלהם שיגדיל את הזמן הכולל שהתוכנית תרוץ

לאור האמור לעיל ניתן להסיק את מספר החוטים האידיאלי לכל עומס. האופטימום ייקבע לפי הטריידאוף בין זמן משבצת לבין זמן דור. עבור עומס גדול נראה שככל שמספר החוטים גדול (לפחות עד 100 חוטים שהרצנו) עדיף כמה שיותר חוטים כדי להקטין את זמן הריצה. בשונה מכך אם נתבונן בעומס בינוני או קטן, נראה שהחלפות הקשר עולות הרבה יותר מהתרומה של המקביליות וזמן דור בשלב מסוים מתחיל לעלות מה שמגדיל את זמן הריצה הכולל. עבורם בגרפים ניקח אופטימום בין זמנים אלו על ידי בחירת מספר החוטים הבא

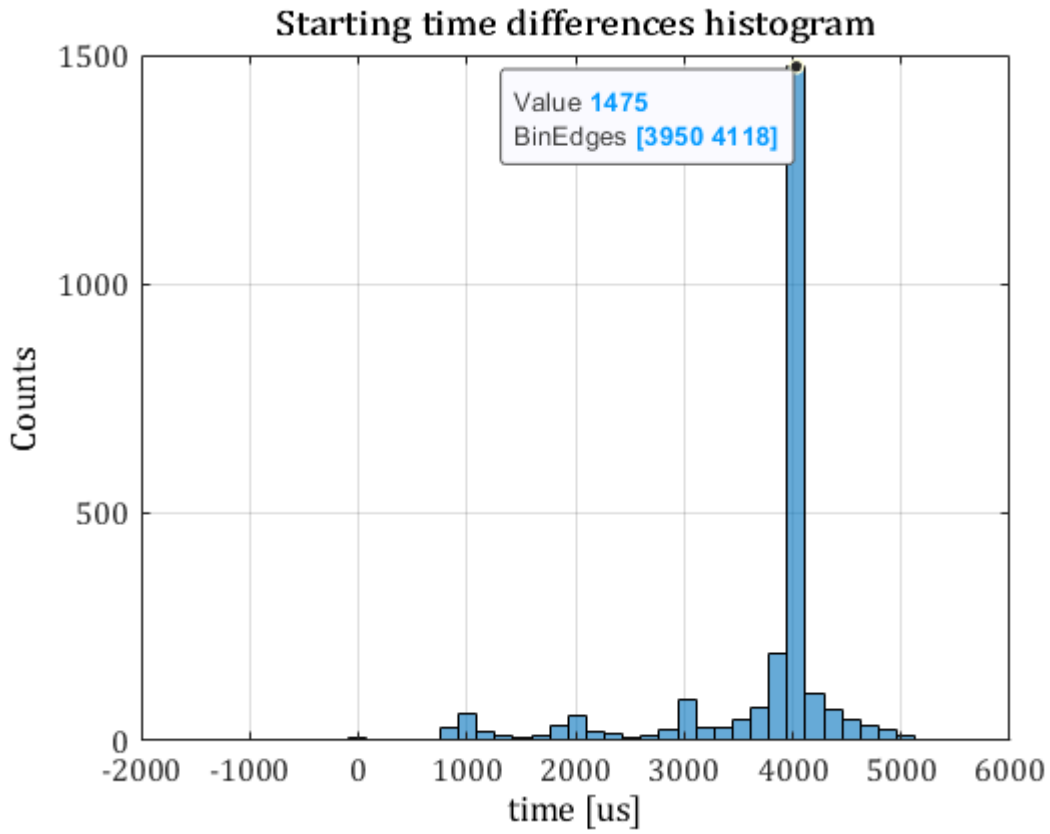
- עומס בינוני – 10 חוטים זו נקודת המפנה בשני הגרפים – תחילת ההתייצבות על גודל משבצת מה שאומר שלא נרוויח מעוד הגדלה של מספר חוטים כל כך הרבה, בעוד מנקודה זו ואילך משך הזמן שלוקח לדור מתחיל לעלות.
- עומס נמוך – בין 5-10 חוטים מקבלים את ההתייצבות בין כל אחד מהזמנים שוב מן השיקולים שהוצגו עבור העומס הממוצע

הוספת מעבדים לא בהכרח תסייע – שכן יש מחיר להחלפה של חוט בין מעבדים שונים. מחיר החלפת ההקשר נותר זהה עבור העומס הקטן והבינוני גם בהוספת עוד מעבדים ולא שווה להם התקורה על החלפות הקשר ביחס לכמות המידע הקטן שהם מנתחים בכל משבצת. מחיר החלפת הקשר עשוי להיות אף גרוע יותר אם יש מעבר של החוט מן מעבד למעבד אחר (אולי עדיף שימתין בתור הריצה ולא יועתק לתור ריצה של מעבד אחר למשל). בעומס גדול ראינו שככל שנרבה במשאבים כך התהליך ייגמר מהר יותר ובמצב זה ריבוי מעבדים נוספים עשוי לעזור.



הגרף שלעיל המתאר את זמני ההתחלה והסיום של עבודות מחולקות לפי ID כאשר רצים 30 חוטים, מאפשר לנו להתבונן במידת המקביליות של הקוד שכתבנו. המקביליות תימדד ביכולת של מספר משימות להתבצע בהפרשי זמן קטנים , תוך התקדמות בציר הזמן, ומתן מקום לכל אחד מהחוטים לבצע דבר במקטע הזמן שיוקצה לו. במידה ונראה שיש פרקי זמן ארוכים בהם חוט יחיד מבצע עבודה והשאר פסיבים לא נוכל להגיד שהוא מקבילי, אך במידה ונראה כי בסקלת הזמן שקרובה לכמות כלשהי של פסיקות שעון יש חילופים בין החוטים, נוכל להגיד שיש ריצה במקביל על ליבות שונות או שיתוף משאב המעבד בין החוטים השונים. הגרף שמוצג לעיל מראה את ריצת הקוד עם מספר חוטים באזור תחילת ההרצה. ניתן לראות שמתוך 30 חוטים שרצים, בערך 2 התחילו לרוץ יחדיו בכל נקודת זמן בה מתחילה ריצה. . מידת המקביליות של הקוד נראית טובה, כיוון שבזמנים מאוד קרובים רואים שיתוף של 2 חוטים בריצה – ולכן זה מתאים למספר ליבות של 2 על המחשבים האישיים שלנו. נראה שהקוד די יעיל אם הוא מאפשר מקביליות כזו של התחלת פעולות יחד.

סקלות הזמן שבציר מתאימות לעשרות מילי שניות ועל כן מתאימות לסקלות של פסיקות שעון שפגשנו בקורס אך לצורך חישוב פיסת זמן השתמשנו בשיטה הבאה – לקחנו את זמני התחלת ריצה של tile וחישבנו את וקטור ההפרשים – זאת לאור העובדה כי ניתן לראות בדיאגרמת הנקודות שפעולות מתחילות יחד אבל לרוב לא מסתיימות מיד אלא יש תחום בו מתחילות כל העבודות ואז הן מסתיימות בזמן שלהן. עבורו ביצענו היסטוגרמה לכמות הפעמים שמופיע כל הפרש לפי bins מתאימים. קיבלנו את ההיסטוגרמה הבאה:



קיבלנו שהשיא של ההיסטוגרמה הוא בbin [3950,4118] כלומר רוב ההפרשים הם בתחום הזה. כיוון שזמנים ארוכים יותר קיימים, וכיוון שיש 2999 הפרשים (יש 3000 זמני התחלה עבור 100 דורות ו30 חוטים) קיבלנו שבערך חצי מן ההפרשים הם במקטע זה. זה מתאים לעובדה שמעבר דורות או כל פעולה אחרת הן פעולות פחות תדירות מאשר החלפת הקשר ומכאן שהשיא מתאים לזמן האופייני להחלפת הקשר. קיבלנו 4000 מיקרו שניות כלומר בערך 4 מילישניות שהן זמן סביר (בערך חצי ממה שפגשנו בקרנל של שיעורי הבית.