Operating Systems – 234123

**Homework Exercise 3 – Dry**

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**

1. The submission should contain the following:
   1. The first page should contain the details about the submitters - Name, ID number and email address.
   2. Your answers to the dry part questions.
2. 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:

1. 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.
2. 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**.
3. 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.
4. 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.
5. **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 שחווית 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:**

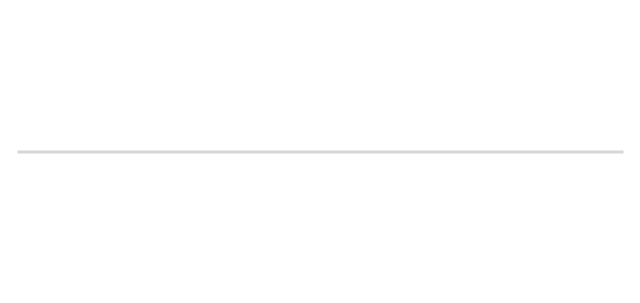
1. **Mutual Exclusion – מניעה הדדית** – בכל רגע נתון, לא יכול להיות יותר מחוט אחד בתוך הקטע הקריטי (הדבר שקול לכך שהקטע הקריטי הופך לנקודת **סריאליזציה** במסלולי הביצוע).
2. **Progress – התקדמות** - אם יש חוטים שרוצים לבצע את הקטע הקריטי, לבסוף חוט **כלשהו** יצליח להיכנס. ישנה התקדמות – אין Deadlock/Livelock.

**תכונות רצויות:**

1. **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. 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 שמופרות**:

Progress- אם חוט 1 נועל את המנעול ואז נופל לפני שהוא משחרר אותו וחוט 2 מנסה גם לנעול את המנעול במטרה להכנס לקטע הקריטי, הוא יכנס לפונקציה lock ויתקע בwhile האינסופי כי חוט 1 לא ישחרר אף פעם את המנעול כי הוא נפל (DEADLOCK).

Order- סדר בחירת החוט הבא שיכנס לקטע הקריטי תלוי בבחירת הscheduler שמבצע החלפות הקשר ואין סדר ברור ביניהם. לדוגמה: חוט 1 תפס את המנעול ואז קודם חוט 2 ולאחריו חוט 3 מחכים שהמנעול ישוחרר. כאשר חוט 1 ישחרר את המנעול לא מובטח שחוט 2 הוא החוט הבא לתפוס את המנעול אלא החוט הבא יהיה החוט שהscheduler יבחר להריץ לאחר שהמנעול ישוחרר וזה יכול להיות גם חוט 3.

Bounded Waiting- חוט 1 נועל את המנעול ואז חוט 2 מנסה לתפוס את המנעול ולהכנס לקטע הקריטי אז גם אם יגיעו N חוטים נוספים יתכן ובשל אופן בחירת החוט הבא שירוץ ע"י הScheduler שN החוטים האחרים יתפסו את המנעול לפני חוט 2 וזה נכון לכל N ולכן אין חסם.

Fairness- כפי שצוין למעלה אין חסם למספר התהליכים שיכולים לרוץ לפני חוט שמנסה לתפוס את המנעול תתכן הרעבה.

* 1. 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?

אם הקטע הקריטי מאוד ארוך, וחוט 1 נועל את המנעול ונכנס להריץ את הקטע הקריטי ואז חוט 2 מנסה לתפוס את המנעול גם הוא אז חוט 2 יכנס לbusy waiting בלולאת ה- while בפונקציה lock עד שחוט 1 ישחרר את המנעול (וזה יקרה רק כשחוט 1 יחזור להריץ את הקטע הקריטי הארוך עד סופו) וחוט 2 לא יתקדם ואף לא ישחרר את המעבד עד שיכלה את כל הtime\_slice שלו ובכך יבזבז זמן מעבד יקר.

אם הקטע הקריטי היה קצר ומהיר אז יש יתרון לשיטת הbusy wait כי כך נחסוך את התקורה של החלפות הקשר שיכולה להיות ארוכה יותר מהקטע הקריטי עצמו. אם יש מעבד יחיד, בכל מקרה נצטרך לבצע החלפת הקשר ואז אין יתרון כלשהו בשיטה זאת.

class lock {

bool lockVal;

public:

lock(bool initVal) { lockVal = initVal;}

void lock(){

while(AtomicSwap(&lockVal,0)==0){}

}

void unlock(){

lockVal = 1;

}

}

1. ממשים מנעול חדש הכולל:

* Mutexסטנדרטי
* 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

else

count=0

if ( count == MAX\_ITER )

unlock 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?

מניעה הדדית- אם יש MAX\_ITER+1 חוטים, הראשון נועל את המנעול בהצלחה נכנס לקטע הקריטי אך לא מסיים אותו ומתבצעת החלפת הקשר, ושאר MAX\_ITER חוטים מנסים לתפוס את המנעול בזה אחר זה אז MAX\_ITER החוטים יקדמו את המונה ויצאו לשינה של T מילי שניות ואז החוט הראשון שבתוך הקטע הקריטי ימשיך לרוץ ונניח שהקטע הקריטי לוקח יותר מT מילי שניות ואז שוב תתבצע החלפת הקשר כשחוט 1 בתוך הקטע הקריטי אז החוט הבא אחריו שיחזור משינה ישחרר את המנעול כי count קודם עד לערך MAX\_ITER ואז הוא יצליח לתפוס את המנעול ולהכנס לקטע הקריטי ואז גם החוט הנוסף וגם חוט 1 יהיו יחד בקטע הקריטי.

Performance- נניח שהקטע הקריטי קצר מאוד. חוט 1 תופס את המנעול ועובר נפילה. אז במקום שחוט 1 יסיים את הקטע הקריטי הקצר וישחרר את המנעול נצטרך לחכות שהמונה count יקודם MAX\_ITER פעמים עד לשחרור המנעול. מה שגורם להרעת ביצועים אם MAX\_ITER גדול מאוד ובמקום לסיים את הקטע הקריטי הקצר נצטרך לחכות עד לקידום המונה.

1. 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.

מטרת הקוד באופן כללי זה לסכום את 10 הערכים שבמערך ids.בתרחיש אופטימלי לאחר יצירת כל חוט הוא מסיים לבצע את הפונקציה thread\_workload במלואה וsum אכן סוכם את הids ללא הפרעה. אולם, תרחיש זה לא בהכרח יתקיים.

ערך מינימלי אפשרי הינו 10, ערך מקסימלי הוא 55 ויתכן גם כל ערך שהוא סכום של תת קבוצה של ערכי המערך ids המכילה לפחות את המספר 10. נפרט:

בסוף ה – main מתבצעת פקודת join עבור החוט האחרון (מספר 10), כלומר יש לחכות שיסתיים חוט זה ורק לאחר מכן יודפס הsum.

במקרה שבו כל החוטים נוצרו אך לא נכנסו לקטע הקריטי עוד, וגם חוט 10 הוא הראשון שייכנס לקטע הקריטי (בשל תזמון של החלפות הקשר), ייווצר מצב בו יסתיים חוט 10 לפני שכל שאר החוטים הספיקו לעדכן את הsum ולכן הsum יהיה שווה ל-10 כמו הid של חוט 10. לכן, יודפס 10 וזהו הערך המינימלי שיכול להיות מודפס.

תרחיש אחר הינו שכל החוטים לפני חוט 10 נכנסים לקטע הקריטי ומעדכנים את sum בהתאם לid שלהם, ואז שחוט 10 יכנס (הוא גם האחרון שנכנס) הוא יעדכן את הsum גם כן, וערכו יהיה 55 (כסכום כל ה- id של החוטים). לכן, בסיום החוט העשירי, פעולת ה- join תסתיים ויודפס הערך 55, וזהו הערך המקסימלי שיכול להיות מודפס במקרה בו כל איברי המערך ids נסכמו בהצלחה.

נשים לב שאין הבטחה שסדר ריצת החוטים יהיה כסדר היצירה שלהם ולכן כל תת קבוצה של חוטים יכולה להתסיים לפני סיום האחרון האחרון ולכן יתכן ונסכום כל סכום אפשרי של תת קבוצה של אברי המערך לפני שהחוט האחרון יסתיים ולכן בהכרח זה יהיה סכום של תת קבוצה של אברי המערך שמכילה (לפחות) את המספר 10.

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;

}

1. 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.

תחילה נשים לב שresult הינו משתנה "גלובלי" הנגיש לשני החוטים.

ערך מקסימלי אפשרי של **result**: 200

נניח כי החוט הראשון שנוצר מספיק להריץ את הפונקציה **do\_calc** במלואה ולקדם את **result** ל-100 לפני שיוצרים את החוט השני וכך גם החוט השני. במצב זה כל חוט יקדם את **result** ב-100 וההשמות בשני החוטים יהיו בלתי תלויות מהבחינה שיספיקו להתבצע כהלכה מבלי החלפות הקשר ביניהן. וכך **result** יגיע ל-200.

ערך מינימלי אפשרי: 2

נניח כי החוט הראשון שנוצר מתחיל לרוץ באיטרציה הראשונה- קורא את הערך ההתחלתי של result שהוא 0 ולפני שהוא מערדכן אותו מבתצעת החלפת הקשר לחוט השני שנוצר שמריץ את הלולאה שלו 99 פעמים ומקדם את ערכו של result ל-99 ומסיים את האיטרציה עבורה **i**=98 אצלו. נניח שעתה שוב יש החלפת הקשר לחוט הראשון שזוכר שהערך של **result** שהוא קרא הוא 0 מקדם אותו ל-1 ורושם אותו לתוך **result**. לאחר מיכן שוב יש החלפת הקשר לחוט השני, שמבצע את האיטרציה האחרונה שלו וקורא את הערך של **result** שהוא 1 ולפני שהוא מספיק לקדם אותו ב1 ולעדכן אותו מתבצעת החלפת הקשר בחזרה לחוט הראשון שמבצע את כל האיטרציות שנותרו לו ומקדם את הערך של **result** ל100 ומסיים את ריצתו ואז שוב יש החלפת הקשר לחוט השני שזוכר שהערך של **result** שהוא קרא הוא 1, ואז מקדם אותו ב-1 ורושם בו את המספר 2. ככה שני החוטים מסיימים לרוץ כשהערך של **result** שיודפס הוא 2.

נשים לב שתתכן הדפסה גם של כל מספר בין 2 ל-200 בהתאם להאם חלה החלפת הקשר בין החוטים כאשר חוט אחד קרא את הערך של **result** ולא הספיק לקדם אותו ולעדכן אותו לפני שהחוט השני חוזר לרוץ וקורא את הערך של **result** ללא השינוי מהחוט הקודם (הבעייתיות נובעת מזה שהקידום של **result** אינה אטומית).

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;

{

1. הסבירו למה אין צורך להגן על sum בעזרת משתנה סנכרון כמו Mutex או Semaphore.

הניחו ש-sum הינו משתנה גלובלי.

**על אף שהמשתנה sum הוא גלובלי fork() משכפלת את מרחב הזכרון ולמעשה לתהליך האב ולתהליך הבן יהיו שני עותקים שונים של המשתנה sum. לכן אין צורך להגן על המשתנה כי שני התהליכים כותבים לשני משתנים שונים בזכרון כל אחד ל-sum שלו.**

int sum = 0;

if( fork() ) {

sum = sum+5;

} else {

sum = sum +1;

}

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.

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

**רעיון: כאשר ה-value של הsinglephore יהיה 0 המנעול יהיה "פנוי" ושהvalue יהיה 1- המנעול יהיה "נעול"**

typedef struct mutex {

singlephore h;

} mutex;

void mutex\_init(mutex\* m) {

singlephore\_init(&(m->h));

}

איתחול שדה הvalue לערך 0 ע"י פונקציית איתחול singlephore (נתחיל ממנעול "פתוח")

void mutex\_lock(mutex\* m) {

H(&(m->h),0,-1);

}

בפעם הראשונה שיתבצע lock נדלג על הwhile כי הvalue הוא 0 שלא קטן מbound שהוא 0. ולכן נוריד 1 לvalue שיהיה עכשיו 1-. בפעם הבאה שנעשה lock כשהמנעול תפוס כן ניכנס לwhlie כי כל עוד המנעול תפוס הvalue הוא 1- והbound הוא 0 ולכן גדול ממנו. נצא מהwhile רק עקב פעולת unlock שתחזיר חזרה את value לערך 0.

Void mutex\_unlock(mutex\* m) {

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

}

הנחה: נניח שקוראים לunlock רק כאשר המנעול נעול.

אם לא נוכל להניח את זה נכתוב:

Void mutex\_unlock(mutex\* m) {

If((m->h).value == 0) // already open

return;

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

}

כל עוד המנעול נעול ערך הvalue הוא 1-. ברגע שנרצה לשחרר את המעול הbound יהיה 1- ולכן לא נכנס לwhile ונגדיל את הvalue ב-1 והוא יהיה 0 ונחזור למצב ההתחלתי בו המנעול משוחרר. המשך התרחיש כפי שתואר למעלה.

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

typedef struct condvar {

mutex m;

singlephore h;

**int** waiting;

} condvar;

נחזיק מונה למספר התהליכים שמחכים על משתנה התנאי.

// Initilize the condition variable

void cond\_init(condvar\* c) {

singlephore\_init(&c->h);  
mutex\_init(&c->m);  
c->waiting=0;

}

איתחול השדות ע"י פונקציות האתחול.

// Signal the condition variable

void cond\_signal(condvar\* c) {

mutex\_lock(&c->m);  
**if**(c->waiting>0) {  
 H(&c->h, INT\_MIN, 1);

c->waiting--;  
}  
mutex\_unlock(&c->m);

}

אם אין חוטים שמחכים אין טעם לשלוח סיגנל ולא נבצע כלל את גוף הif. אם יש חוטים שמחכים נשלח סיגנל שבהכרח יעיר אחד מהם ולכן נוריד את מספר הממתינים ב-1. ע"י מנעול נגן על החלק של הבדיקה שאינו אטומי ויכול להיות משותף לכמה חוטים.

// 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->waiting++;   
mutex\_unlock(&c->m);  
H(&c->h,0,-1);   
mutex\_lock(&m);

}

נעלה ב-1 את מספר הממתינים על משתנה התנאי (נגן על חלק לא אטומי זה ע"י מנעול). ואז נשלח להמתנה את החוט הנ"ל ע"י קריאה ל-H.

**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.
5. 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'.

בפונקציה cond\_signal הקריאה לH תשלח ערך Bound שתמיד יגרום לכך שלא נכנס לwhile ורק נגדיל את value ב-1. בכך למעשה אנחנו סופרים ושומרים במשתנה value כמה סיגנלים נשלחו וזאת בניגוד לדרישה ממשתנה תנאי שאם אין אף חוט שממתין באותו רגע על משתנה התנאי אז הפעולה צריכה להיות חסרה השפעה.

התרחיש הבא אפשרי:

עושים signal כשעוד אף תהליך לא תפס את המנעול ולכן נגדיל את הערך של value למספר חיובי 1. ואז כשנעשה wait כיוון שvalue גדל להיות חיובי נדלג על הwhile לא נשלח את החוט להמתנה וישר נוריד את הvalue ל-0 כלומר בפועל לא גרמנו לחוט לצאת להמתנה ולכן ישר נתפוס שוב את המנעול. כלומר חוט המשיך בריצה למרות שלאחר שביצע cond\_wait לא בוצע cond\_signal על ידי אף חוט אחר.

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:

****

****

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

**תשובה**:

המנגנון היה אמור לעבוד כך:

עבור כל דור, בשלב 4 הproducer ידחוף לתור את כל המשימות הנחוצות לטיפול באותו דור ואז ע"י wait נחכה שהconsumers יסיימו את כל המשימות לפני שנעבור לדור הבא. נצפה שעבור כל working++ יתבצע working-- ככה שנדע שכשנסיים את כל דחיפת המשימות והערך של working יחזור ל-0 שסיימנו את הטיפול בדור הנוכחי שאנו עובדים עליו.

כלומר נצפה קודם להזין לתור את כל המשימות ואז לחכות עד שכולן יתבצעו (יתכן ותוך כדי דחיפת המשימות הconsumers כבר יתחילו לבצע אותם) ו-working ישמור את מספר המשימות שטרם הסתיים ביצוען.

נשים לב שאין שימוש במנעולים במימוש של פיראס וזאת בניגוד למימוש של הsemaphore שראינו בכיתה. פעולת הdecrease יכולה להיקרא ע"י מספר חוטים שיכולים כולם יחד לגשת לאותו משתנה waiting ולהחסירו מבלי שיהיה מנעול להגן עליו. בנוסף, ההבדל העיקרי בין מנגנון זה לבין ה-semaphore עליו למדנו הוא שהמנגנון של Barricade משתמש ב busy-wait ומחכה בלולאת while עד לקיום התנאי בעוד הסמפור מונע לגמרי את ה-busy wait ובמקום זאת מוציא תהליך להמתנה.

1. 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.

**תשובה**: נניח שישנם שני חוטים שמשמשים כconsumers שמבצעים את שלב הdecrease

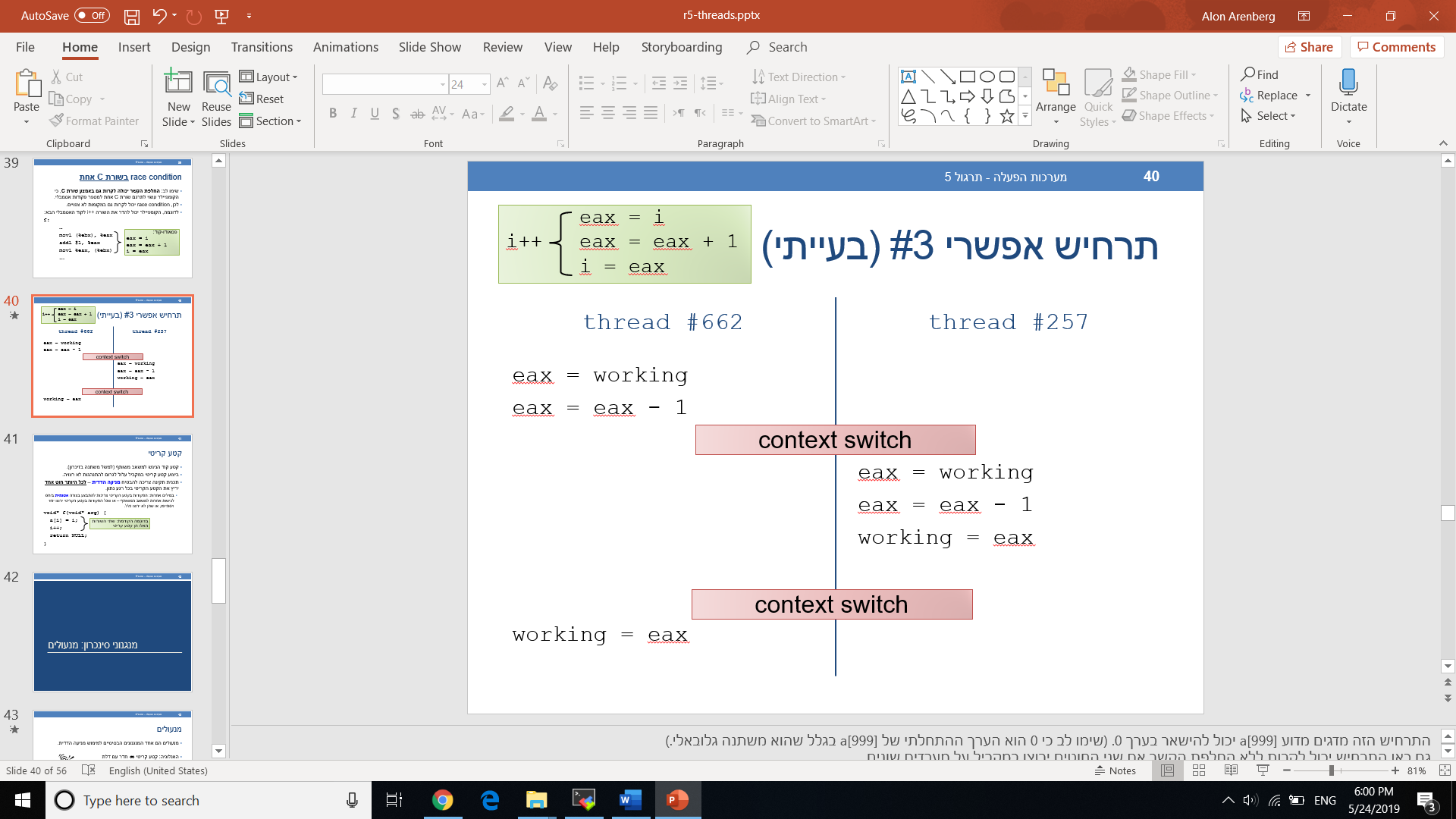
נניח כי הקוד האסמבלי שנפרש הינו

eax = working

working-- eax = eax - 1

working = eax

ונניח כי סדר החלפות ההקשר הינו כדלקמן



Consumer #2

Consumer #1

כלומר כיוון שקוד האסמבלי אינו אטומי יתכן והחלפות ההקשר יגרמו לשיבוש הלוגיקה. בסך הכול התקבל מצב שבו 2 חוטים "סיימו 2 משימות שונות", אך ערכו של working משקף מצב שבו רק משימה אחת הסתיימה, והדבר שובר מבחינה לוגית את המשך הריצה התקינה של הקוד ויכול לגרום לproducer להתקע בwhile אינסופי בפעולת הwhile.

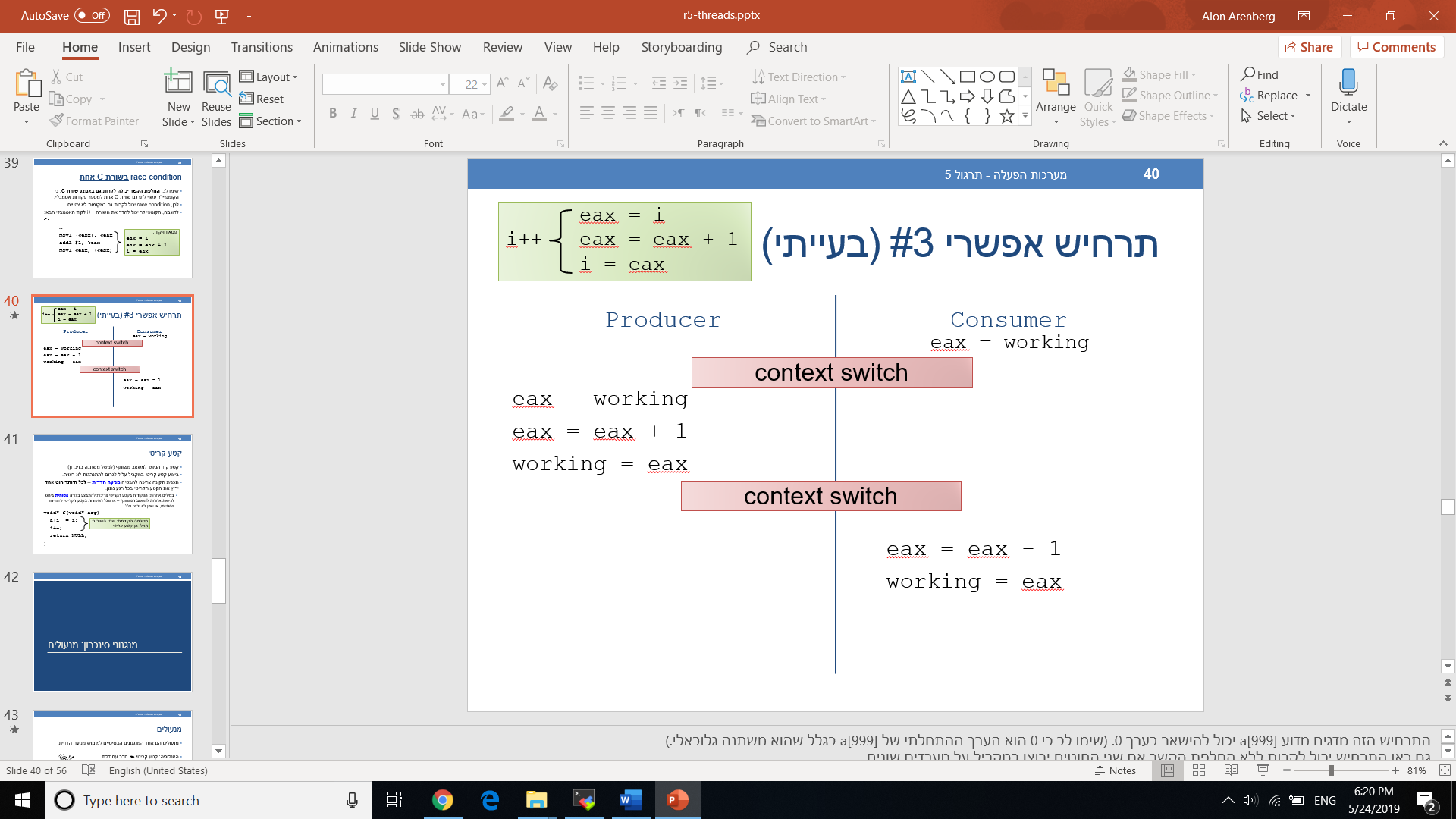
* 1. תארו תרחיש שבו מופר ה-**Mutual** **Exclusion**. דהיינו, חישוב הלוח curr טרם הסתיים, וה-Producer מבצע למרות זאת את ה-swap של הלוחות.

**תשובה**: נניח כי N=1 (כלומר היצרן דוחף עברו כל דור שתי משימות). התרחיש הוא: היצרן דחף משימה ולפני שהספיק לבצע increase מתבצעת החלפת הקשר, הצרכן שולף את המשימה, מבצע אותה והdecrease שלו "מתערבב" בסדרת החלפות הקשר עם הincrease של היצרן ככה שבסוף working יקטן ב-1 ויהיה 1-. בעת דחיפת המשימה השנייה היצרן ידחוף את המשימה לתור, יסיים לבצע insrease ואז working יגדל מ-1 ל-0 ולכן היצרן יעבור את הwait ויבצע swap למרות שהמשימה השנייה עדיין לא בוצעה ועדיין נמצאת בתור המשימות לביצוע.

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

**תשובה**:

תרחיש 1: ישנם שני חוטים אחד יצרן והשני צרכן. נניח כי N=0 כלומר מתבצעת רק איטרציה אחת בשלב 4. היצרן דחף משימה לתור, החלפת הקשר, הצרכן מוציא משימה מהתור ולא נחסם כי היצרן כבר הכניס משימה, הצרכן מסיים לבצע את המשימה הצרכן עושה decrease והיצרן עושה increase ובפועל התוצאה של שניהם ביחד תהיה working--, כלומר נתחיל מworking=0 התרחיש יהיה:



בסיום working יכיל -1 ואז כשהיצרן יעבור לwait הוא יתקע בwhile אינסופי כי הוא לא יוכל להתקדם ולהגדיל את הערך של working כי רק הוא יכול לעשות את זה והוא תקוע. לכן הוא גם לא דוחף משימות לתור וכתוצאה מכך לא יהיו consumers שאולי היו יכולים לשנות את הערך של working כי הם יתקעו בנסיון להוציא איבר מתור ריק שאף אחד לא ידחוף אליו משימות.

תרחיש 2:

נניח כי ישנו יצרן אחד ושני צרכנים וN=1. היצרן דוחף שתי משימות לתור ומגדיל את working ל-2 אך נניח כי שני הצרכנים מבצעים כל אחד משימה אחת וכשהם מגיעים לשלב של הdecrease מתבצע רצף של פעולות החלפת הקשר (בדומה לסעיף ב שאלה 1 )כך שבפועל הערך של working יופחת רק ב-1 אז היצרן יתקע בwait בwhile אינסופי כי working ישאר שווה ל-1 כלומר לא 0 ולא היצרן יוכל להתקדם ולהגדיל ערך זה וגם לא יהיו צרכנים שיעשו זאת כי התור כבר ריק ויתקעו כשינסו להוציא ממנו משימות.

1. 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.

**תשובה**:

class {

private:

int working;

pthread\_mutex\_t m;

public:

Barricade(){

working =0;

pthread\_mutex\_init(&m,NULL);

}

increase(){

pthread\_mutex\_lock(&m);

working++;

pthread\_mutex\_unlock(&m);

decrease(){

pthread\_mutex\_lock(&m);

working--;

pthread\_mutex\_unlock(&m);

}

wait(){

pthread\_mutex\_lock(&m);

while(working!=0){

pthread\_mutex\_unlock(&m);

pthread\_mutex\_lock(&m);

}

pthread\_mutex\_unlock(&m);

}

} ;

נגן על החלקים הקריטיים (כתיבה למשתנים משותפים) שיכולים להיות נגישים ממספר חוטים ע"י מנעולים. ככה נבטיח אטומיות בעידכון של כל חוט את אותו משתנה.

כמו כן נדאג לקריאה אטומית. נגן על הקריאה עם מנעולים כשבתוך גוף הWHILE נותיר שחרור ונעילה משוחררת של המנעול כדי שאם יש חוטים שממתינים על המנעול לבצע DECREASE או INCREASE נותיר להם לבצע אותם למשל אם בין שתי הפעולות בגוף הWHILE תתבצע החלפת הקשר.

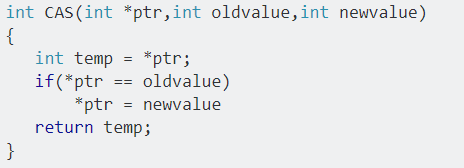
* + - * 1. מנגנון הסנכרון שפיראס ניסה ליצור, ואותו אתם השלמתם בסעיף 1' נקרא "מונה משותף", ואינו מוצלח במיוחד ממבט של בביצועים - Performance. הסבירו מדוע. התייחסו לחסרון המנגנון כאשר N, מספר החוטים האפקטיבי גבוה מאוד.

**תשובה**: החוסר יעילות נובע מהbusy wait שמתבצע בסוף שלב 4 בפקודה wait. נניח כי N גדול מאוד וכל המשימות נדחפו ע"י היצרן לפני שהצרכנים הספיקו לבצע אותם והיצרן הגדיל את working ב-N, אז היצרן יצטרך להמתין זמן רק מאוד עד שכל אותן N משימות יתבצעו וworking יקטן חזרה ל-0. היצרן מחכה בbusy wait ומבזבז זמן מעבד.

בנוסף, המנגנון אינו מוצלח משום שעל מנת לבצע פעולות קצרות מאוד כמו working++ מתבצעת בכל פעם נעילה של מנעול, שהיא פעולה כבדה הכוללת יציאה להמתנה, ולכן התקורה בתוכניות שמשתמשות במנגנון הזה גבוה.

הדבר מורגש במיוחד כאשר מספר החוטים גבוה, משום שנקבל שבצוואר הבקבוק הנוצר כתוצאה מהמתנות של הרבה חוטים למנעול בודד גורמת לכך נעצרים יותר חוטים, ולכן תהיה פגיעה בביצועים.

* + - * 1. הראו פתרון נוסף בעזרת שימוש בפעולות אטומיות. הפעולות האטומיות שעומדות לרשותכם:
    - atomicAdd(int \* ptr,int val)אשר מוסיפה **באופן אטומי** val לערך התא ptr
    - atomicCAS, עם תיאור הפעולה בקוד:



* + - הניחו שפקודות אלו ממומשת **בחומרה**, ולא בתוכנה, כיאה לפעולות אטומיות.
    - **אין** 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)){}

}

} ;

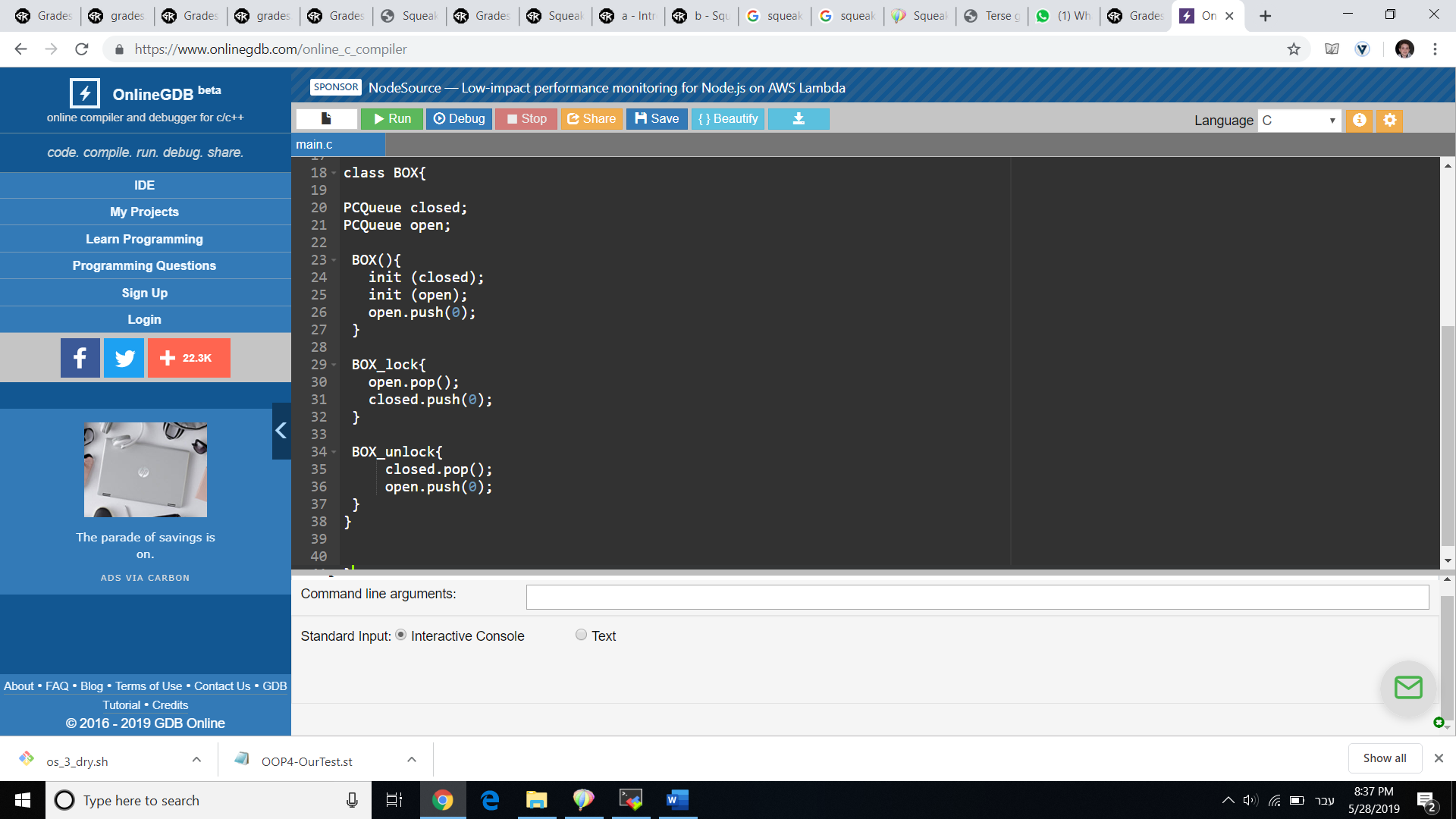
הלוגיקה של increase, decrease זהה למקרה הקודם, אלא שבמקום להשתמש במנעול להבטחת האטומיות, משתמשים ב־atomicAdd.

עבור wait, האטומיות של בדיקת הערך working מובטחת גם היא על ידי CAS, ומשום ש־CAS מחזירה את הערך הישן של המשתנה, הביטוי בתמונה מחזיר ערך זהה (מבחינה בוליאנית) לשל working!=0. כלומר נבצע קריאה אטומית של המשתנה working ובכך נממש מעין "מנגנון קוראים-כותבים" ונמנע קריאת משתנה תוך כדי שחוט אחר כותב אליו. למשל אם המספר מורכב מהרבה בתים ונתחיל לקרוא אותו תוך כדי שחוט אחר כותב אליו, ניקרא ערכי ביניים ובתים שחלקם שונו וחלקם לא שונו ונקבל תוצאה לא מוגדרת.

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

**תשובה**: פתרון זה עדיף על שימוש במנעולים כיוון שכאשר חוט מנסה לתפוס מנעול נעול יש תקורה של החלפת הקשר כי הוא יוצא להמתנה. לעומת זאת אם הפקודה תהיה אטומית נמנע מהתקורה של החלפת הקשר ויציאה להמתנה מה שישפר את הביצועים. כאשר נבצע increase ו-decrease לא נותר על המעבד עקב הצורך לחכות למנעול שיתפנה.7

1. **בונוס (10 נקודות):** הציגו מנגנון סנכרון המנצל **שני** PCQueues כקופסא שחורה לטובת הסנכרון ההדדי בין Producer ל-Consumer. הסבירו בפירוט את פתרונכם.
   * + 1. **אין** להשתמש בכל מנגנון סנכרון נוסף (בפרט mutex,semaphore, atomics, ו-busy wait).
       2. מומלץ להחליף קטעים רלוונטיים בפסודו קוד לשם הצגת פתרונכם.
       3. 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 **ומנימלי**, אשר אינו מבצע פעולות אלו. ניתן להשתמש ב-mutex יחיד, condvar יחיד ו-PCQueue יחיד.



פתרון:

נממש את מנגנון הסינכרון ע"י שני PCQueues שידמו מנגנון דמוי mutex.

כאשר התור Open אינו ריק "המנעול" המדומה יהיה פתוח וכאשר Closed

אינו ריק המנעול יהיה "נעול". נשים לב שבכל רגע (למעט לצמב בו פותחים

או סוגרים מנעול) בדיוק אחד משני התורים ריק (מצב המנעול יהיה חוקי

מחוץ לפונקציות).

בפונקציה BOX\_init נאתחל את שני התורים ולאחר מכן נדחוף איבר לOpen

כלומר ברירת המחדל היא מנעול פתוח.

בפונקציה BOX\_lock אנו נרצה לנעול את המנעול ולכן נעשה open.pop(),

כלומר נוציא את האיבר שיש בו- נשים לב שפעולה זו תחסם עד אשר התור לא

יהיה ריק (מאיך שממומש הPCQueue) ולכן לא יתכן שיהייו שני חוטים באותו

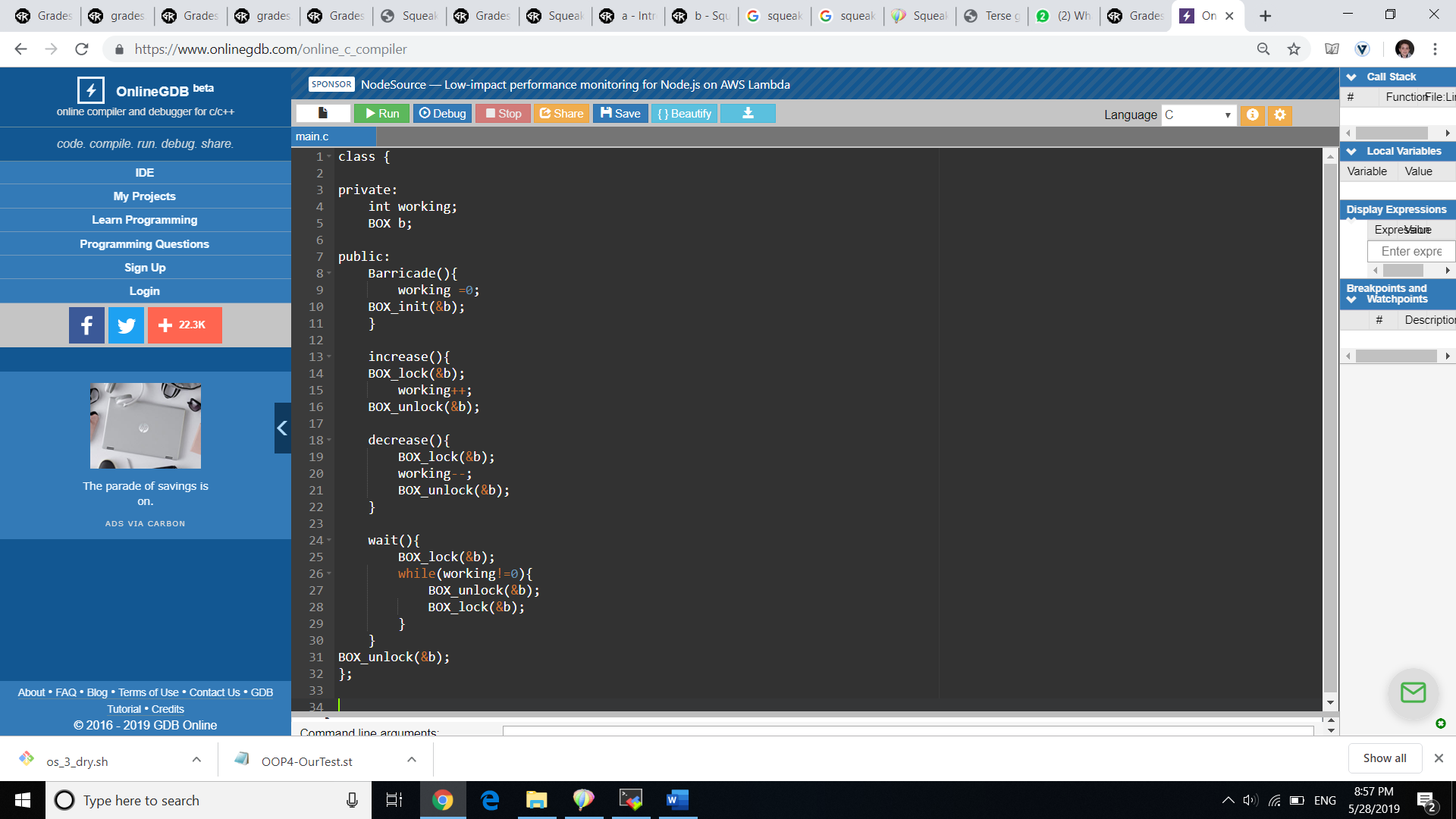
קטע קריטי עליו המנעול מגן. ככה יחסם נסיון לנעול מעון נעול ע"י חוט אחר.

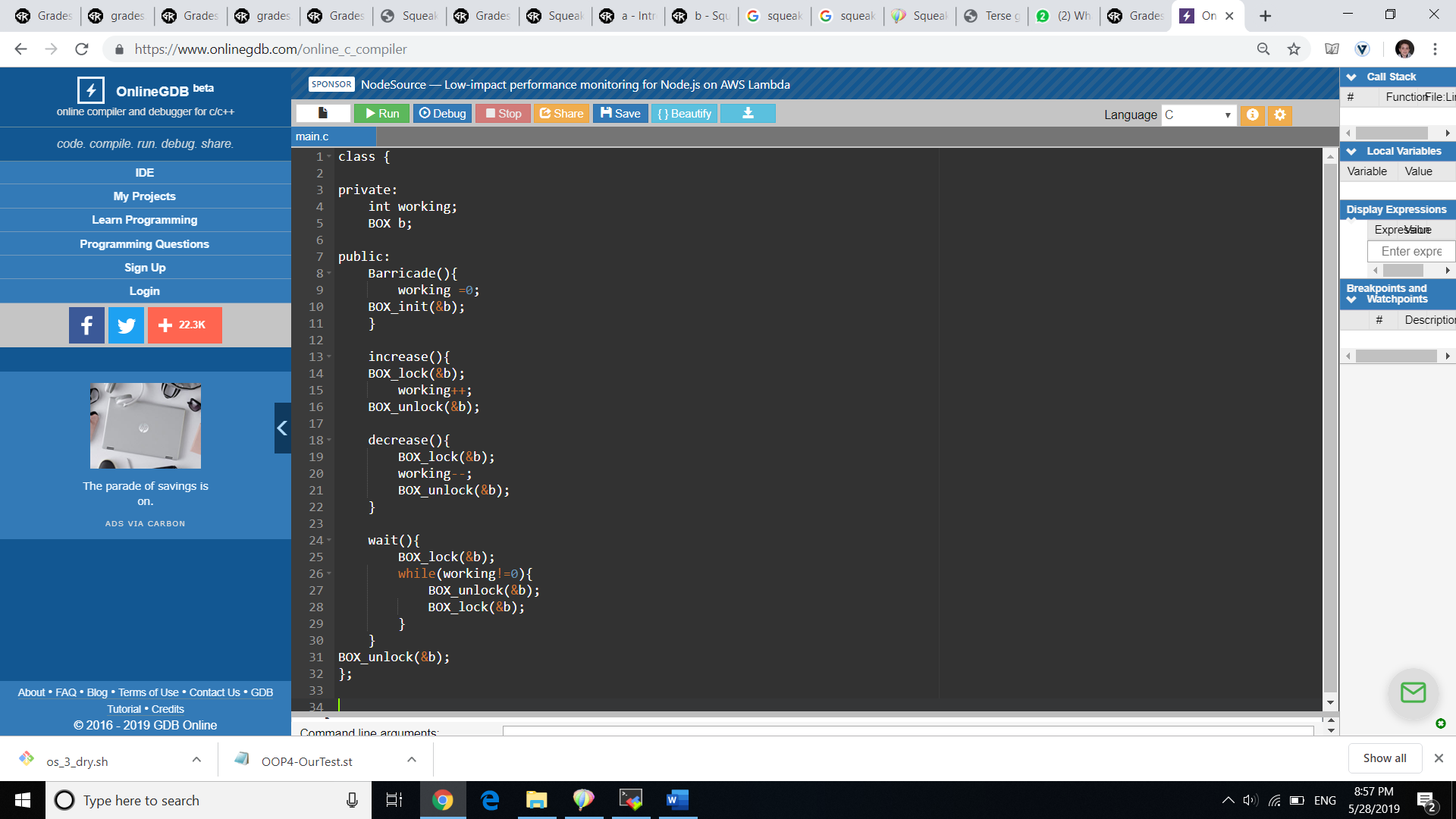
לאחר מכן נעשה closed.push שיסמן שהמנעול נעול.

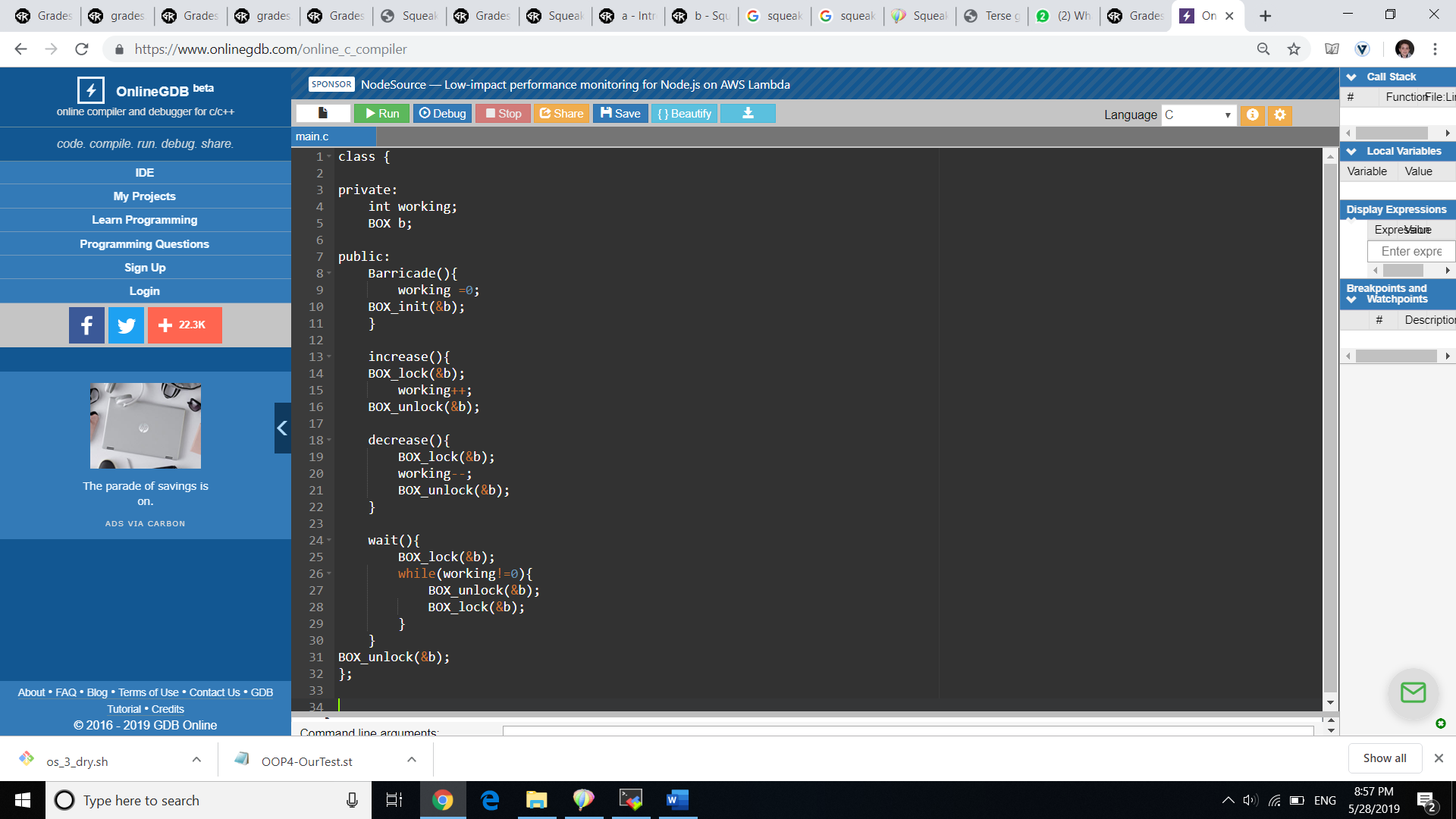
בפונקציה BOX\_unlock נרצה לפתוח את המנעול לכן נעשה closed.pop()

כלומר נוציא איבר שיש בו ואז נעשה open.push(0)כדי לסמן שהמנעול פתוח.

ככה למעשה בכל מקום בו משתמשים בmutex נוכל להשתמש במקום בBOX.







Producer:

1. Init Barricade b

2. Init PCQueue p

5. Init fields curr,next

6. for t=0 t=n\_generations

For i=0 i=N

p.push(job);

b.increase();

b wait();

swap(curr,next);

Consumer(One of N)

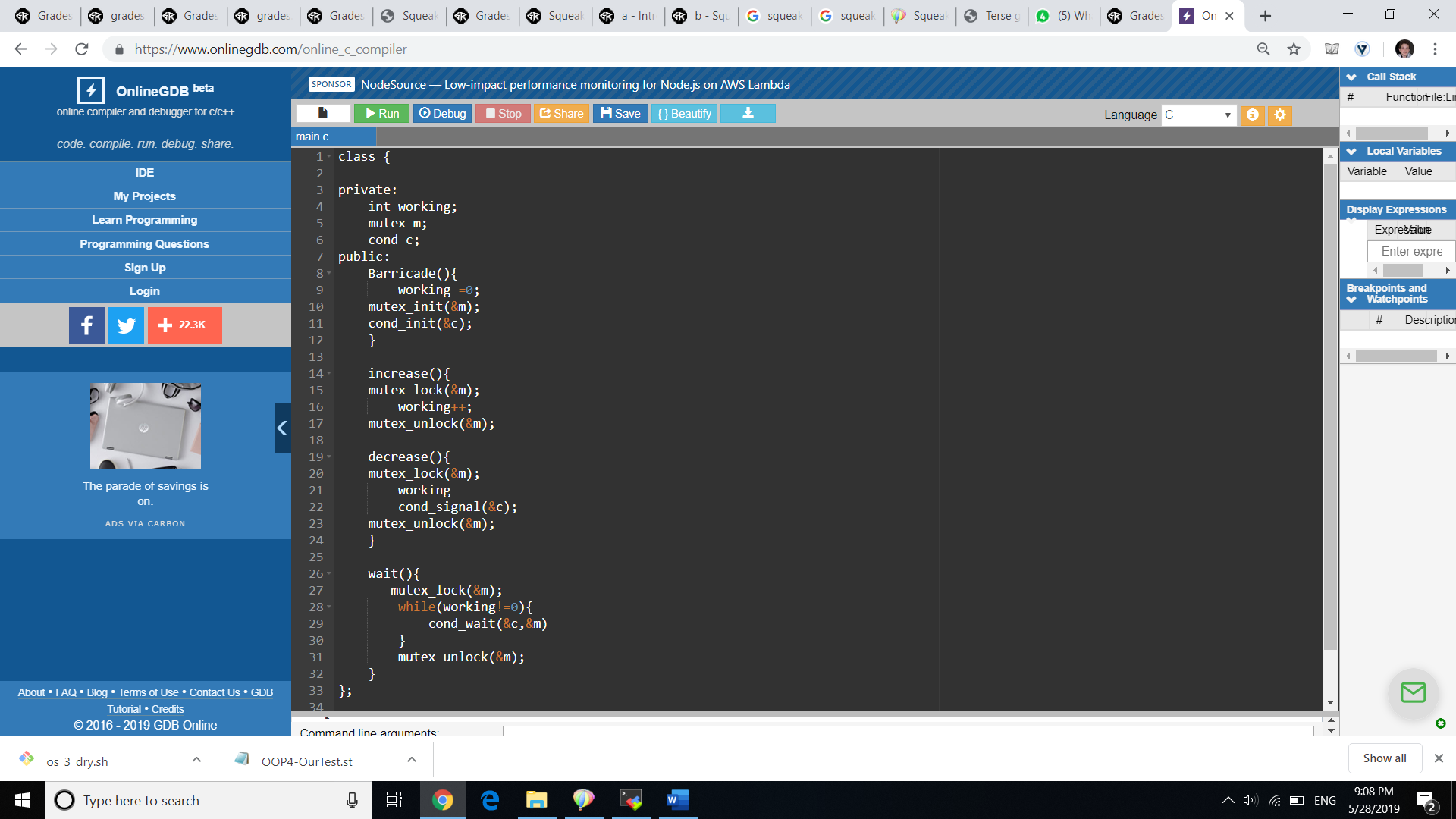
While(1)

Job j= p.pop();

Execute j;

b.decrease();

**מימוש מינימלי**

המימוש הבא שקול למימוש שהצגנו קודם כיוון שהפונקציונליות

של הBOX שקולה לזו של הMUTEX ולכן נוכל להחליף ביניהם.

בנוסף, נחליף את המימוש הקודם בו בוצע

BUSY WAIT בפונקציה WAIT בלולאת

הWHILE כאשר בדקנו שוב ושוב מתי התנאי יפסיק להתקיים

מבלי לותר על המעבד. עתה, נוציא את החוט להמתנה על

משתנה התנאי ונתעורר רק כאשר יתקבל סיגנל.

**שאלה 2: ביצועים וחוק אמדל**

נניח אלגוריתם A המורכב ממספר רב של עבודות 

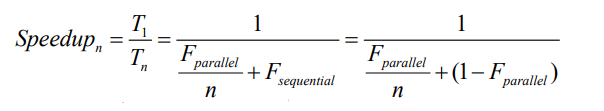
לנוחיותכם, מושגים נפוצים לניתוח Performance של האלגוריתם:

* + 1. **Latency(A)** – זמן החישוב הכולל של האלגוריתם A
    2. **Latency(j)** – זמן החישוב הכולל של עבודה j
    3. **Turnaround Time(j)** – זמן החישוב + זמן ההמתנה בתור של עבודה j
    4. **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.**

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

נסמן Latency(A) במצב בו אין מיקבול כלל כS. חוק אמדל הינו:

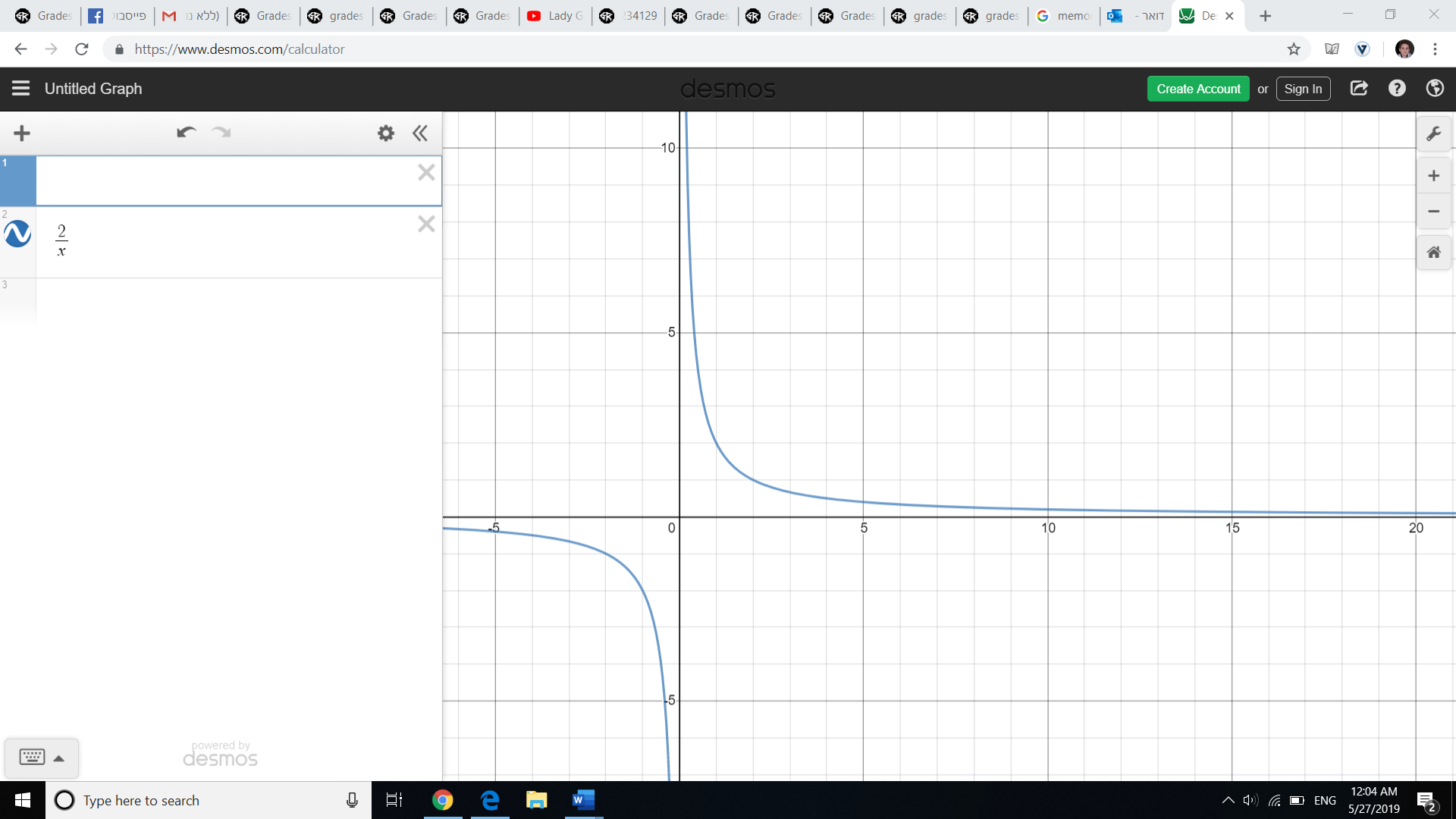


כאשר הוא הlatency של תהליך הרץ על n מעבדים, ו הוא החלק היחסי של התהליך שרץ במקביל. כאשר תהליך ניתן למקבול מלא, . לכן,

כלומר

כלומר כאשר תהליך ניתן למקבול ע"י n חוטים שרצים במקביל הגיויני שאם ללא מיקבול לוקח לו לרוץ S אז עתה המשימה מתחלקת שווה בשווה בין n החוטים שכל אחד רץ במשך ולכן זה סך זמן הריצה.

סה"כ בגרף נקבל היפרבולה מהצורה



N

S

1. הוסיפו לגרף בסעיף **א'** מעלה עקומים המתארים גם A סריאלי לחלוטין, ו- A המורכב מחלק של  שניתן למקבול.

סעיף זה נועד לספק לכם מדד איכותי לגרפים התיאורטיים שיתקבלו בתנאי "מעבדה", בהתאם לכמה האלגוריתם סריאלי/מקבילי.

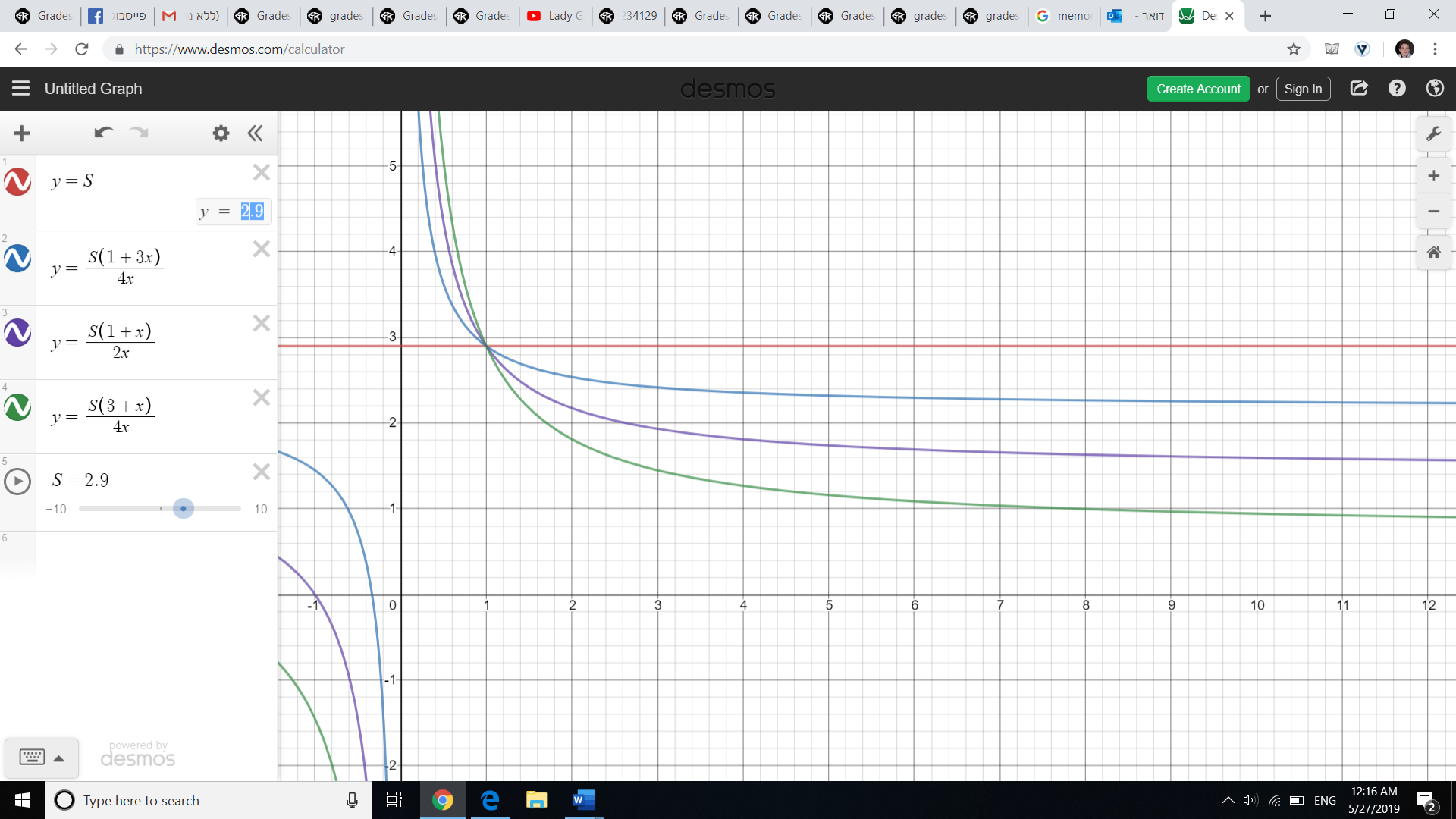
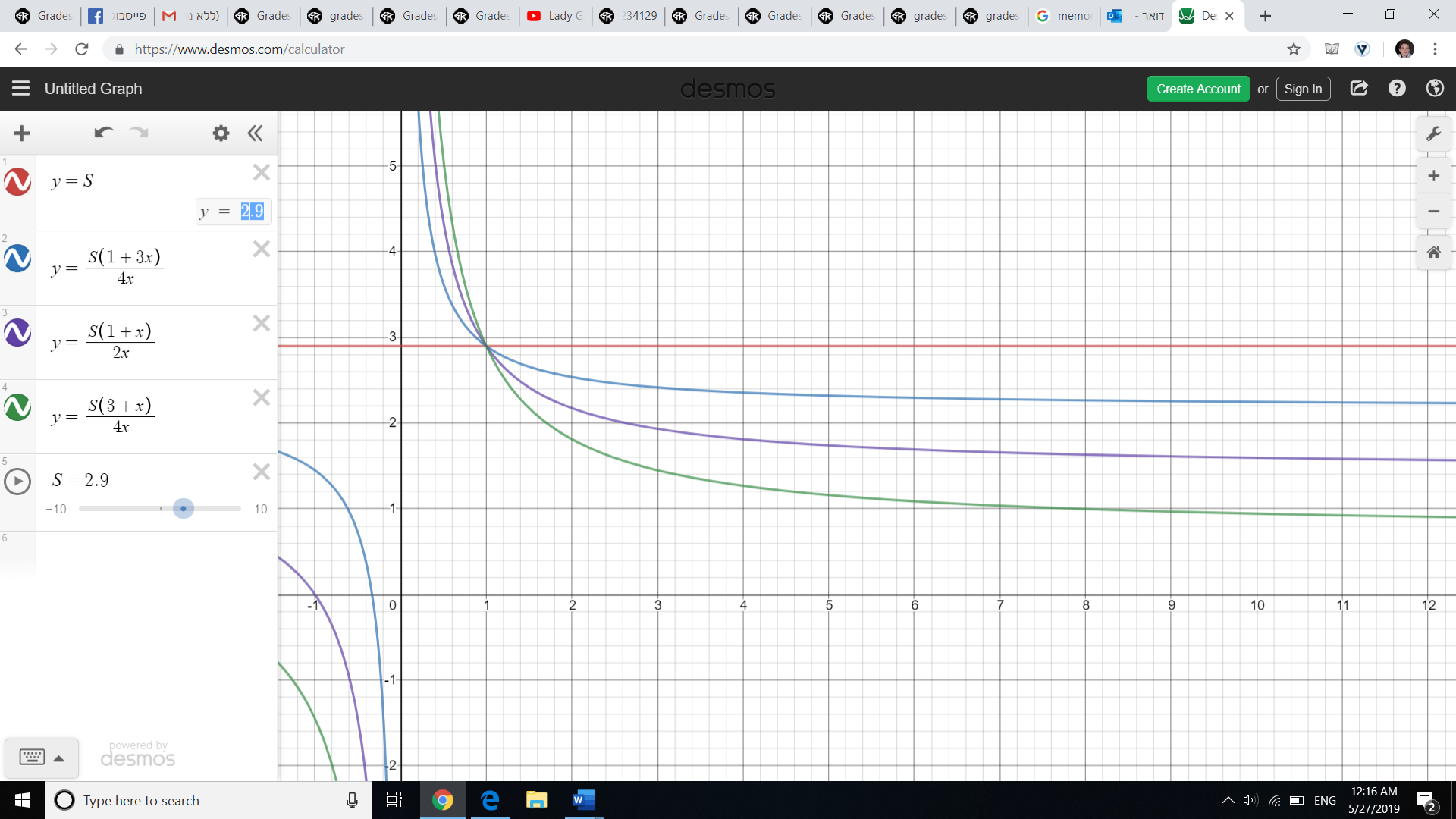
ניתן לקבע את זמן הביצוע הכולל אם A היה סריאלי לחלוטין כרצונכם (למשל – שניה).

כאשר A סריאלי לחלוטין, . לכן כלומר

כאשר 0.25 מA ניתן למקבול, . לכן כלומר

כאשר 0.5 מA ניתן למקבול, . לכן כלומר

כאשר 0.75 מA ניתן למקבול, . לכן כלומר



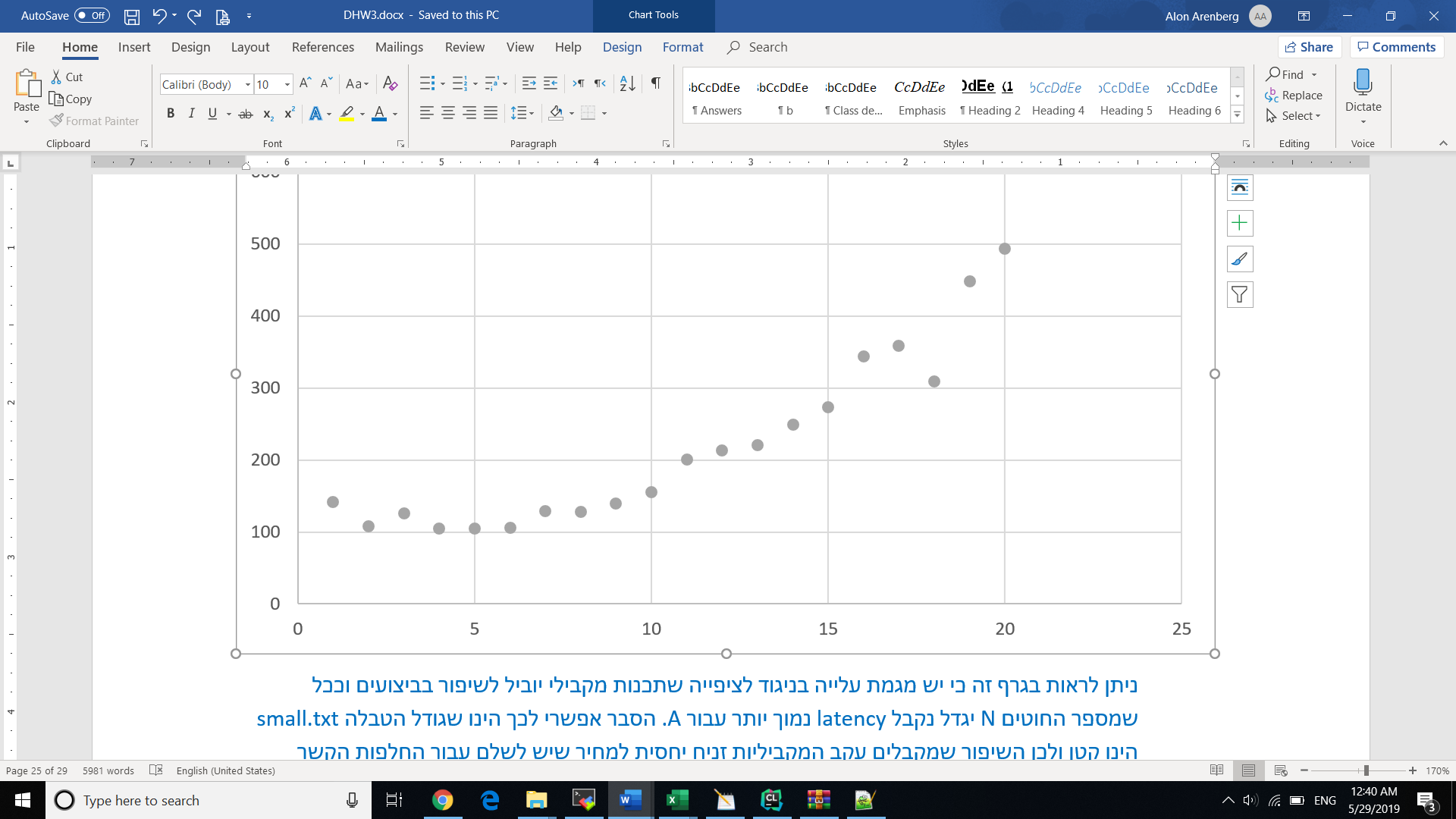
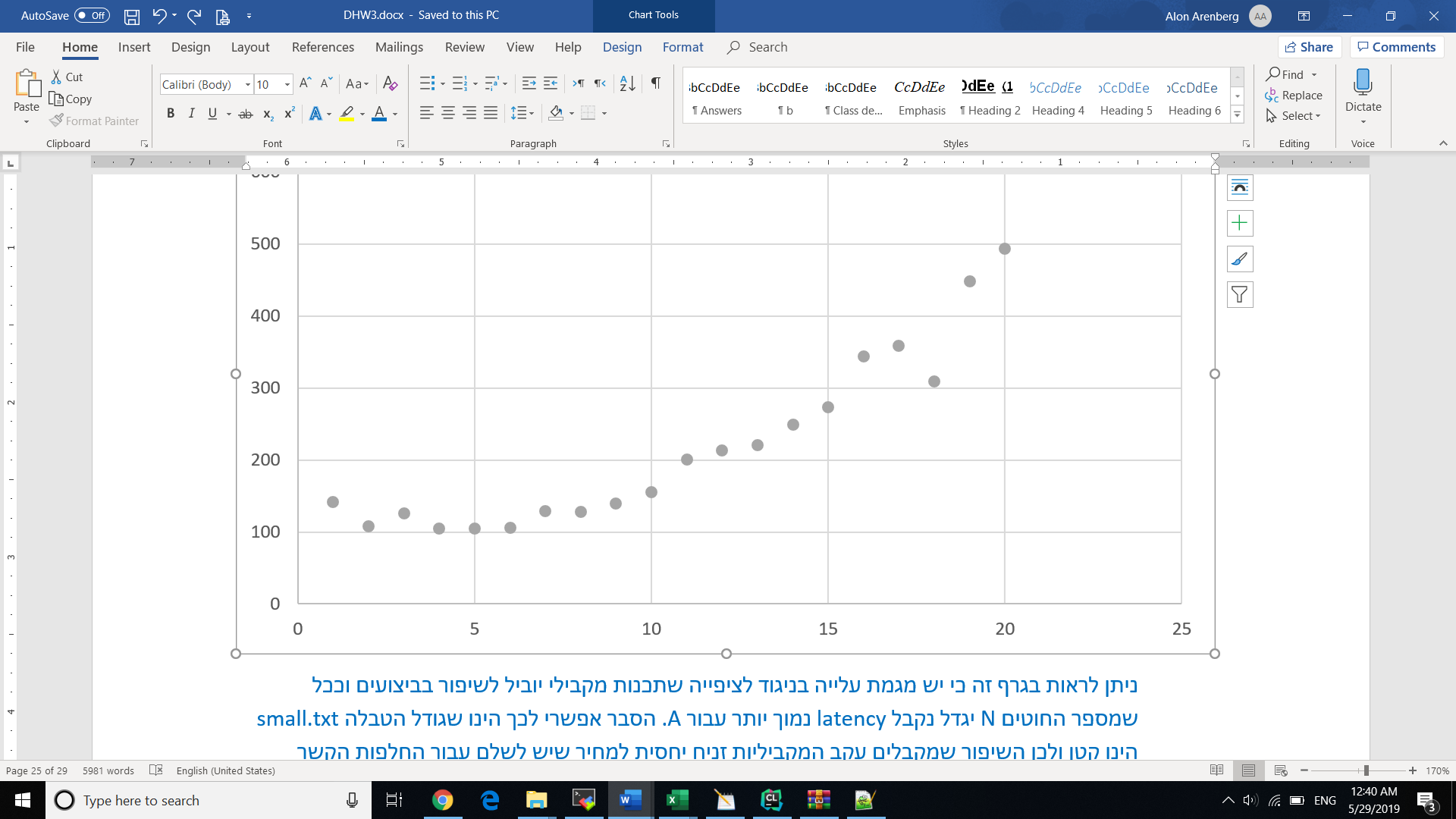
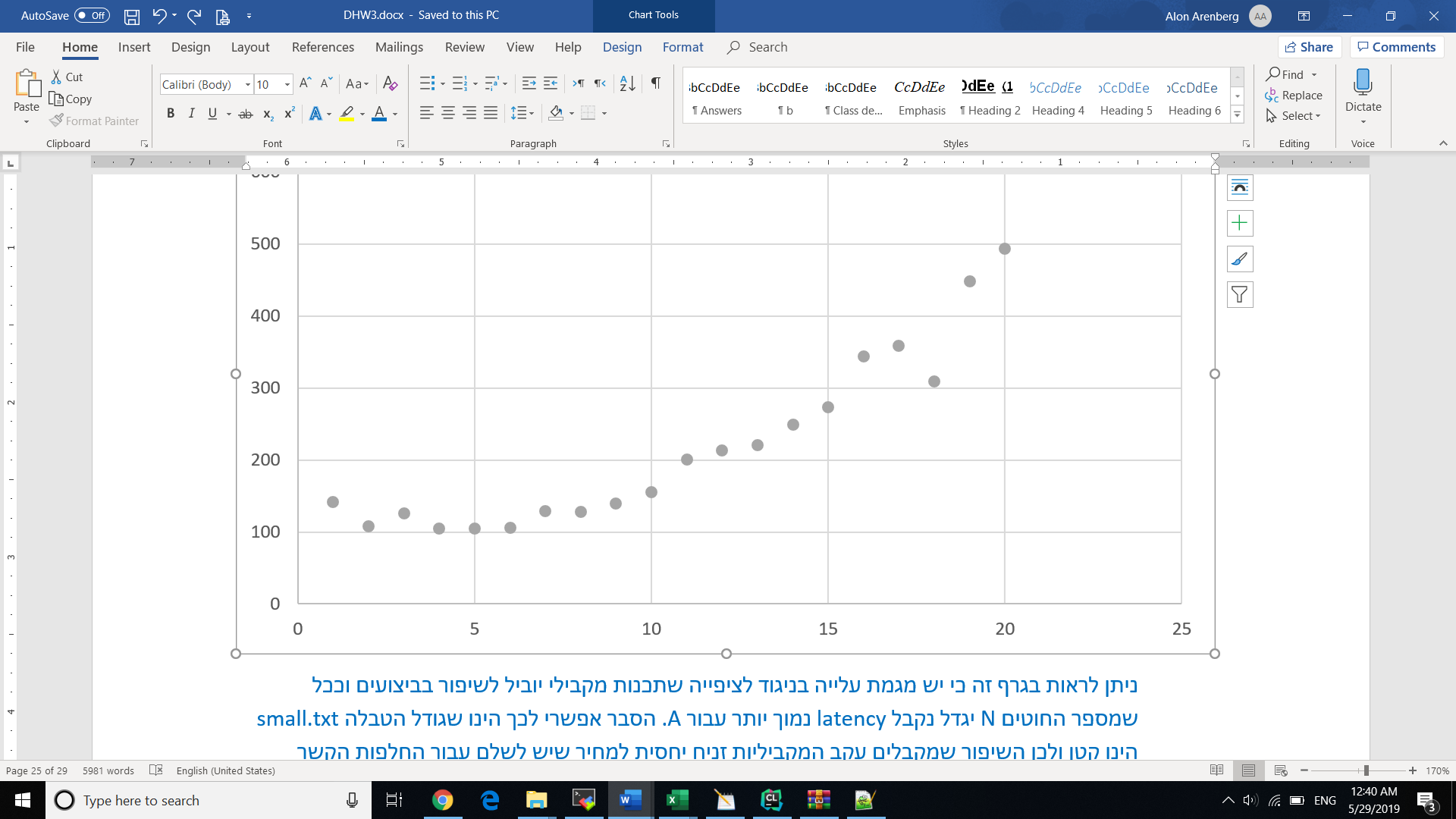
Latency(A)

N

1. 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:

* + 1. גרף של Latency(A) Average כתלות במספר החוטים N.
    2. גרף של Average Latency(j)כתלות במספר החוטים N.
    3. גרף scatter plot של Tile Index אל מול Thread Id, כאשר Tile Index 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: , 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.



ניתן לראות בגרף זה כי יש מגמת עלייה בניגוד לציפייה שתכנות מקבילי יוביל לשיפור בביצועים וככל שמספר החוטים N יגדל נקבל latency נמוך יותר עבור A. הסבר אפשרי לכך הינו שגודל הטבלה small.txt הינו קטן ולכן השיפור שמקבלים עקב המקביליות זניח יחסית למחיר שיש לשלם עבור החלפות הקשר הרבות וההמתנה במנעולים. יתכן שאלו יגרמו לעליה בזמן הביצוע אילו היינו עושים את הכול ללא מקביליות. בנוסף, יש לזכור שבכל קוד קיימים חלקים שלא ניתנים למקבול וככל שמספר החוטים גדל חלקים אלו גדלים גם כן.

ניתן לראות בגרף זה כי יש מגמת ירידה ב – latency ככל שמספר החוטים גדל. הסבר לכך הינו שככל שיש יותר חוטים גודל כל tile קטן כלומר כל משימה קטנה יותר ולכן הזמן הדרוש לביצועו קטן בהתאם לציפיות. ניתן לראות כי ככל שמספר החוטים גדל השיפור היחסי קטן (שיפוע הגרף גדל)- זאת מאחר ועדיין יש חלקים שלא ניתנים למקבול, וגם מפני שיש יותר החלפות הקשר. סה"כ הגרף תואם את חוק אמדל.

בדומה לטבלת הsmall, ניתן לראות כי גם בגרף זה יש מגמת עלייה בניגוד לציפייה שתכנות מקבילי יוביל לשיפור בביצועים ככל שמספר החוטים N יגדל נקבל latency נמוך יותר עבור A. הסבר אפשרי לכך הינו שגודל הטבלה mid קטן יחסית, ולכן השיפור שמקבלים עקב המקביליות זניח יחסית למחיר שיש לשלם עבור החלפות הקשר הרבות וההמתנה במנעולים. בנוסף, יש לזכור שבכל קוד קיימים חלקים שלא ניתנים למקבול וככל שמספר החוטים גדל חלקים אלו גדלים גם כן.

בדומה לגרף התואם עבור small.txt גם עתה גודל כל משימה הינו ביחס הפוך למספר החוטים ולכן ככל שיש יותר חוטים כל חוט מבצע משימה קטנה יותר. (ניתן לראות שיפור של כמעט פי 2 כשעוברים מחוט 1 ל-2).

בניגוד לטבלאות הקודמות, כעת ניתן לראות מגמת ירידה התואמת לציפיות שתכנות מקבילי יוביל לשיפור בביצועים ככל שמספר החוטים N יגדל נקבל latency נמוך יותר עבור A. הסבר אפשרי לכך הינו שגודל הטבלה גדול יחסית, ולכן השיפור שמקבלים עקב המקביליות משמעותי יחסית למחיר שיש לשלם עבור החלפות הקשר הרבות וההמתנה במנעולים. בהתאם לחוק אמדל כי הגרף מתייצב על ערך קבוע בשל החלק הסריאלי שלו. לכן, ככל שמספר החוטים גדל החלק הסריאלי גדל, והחלק המקבילי לא מביא לשיפורים משמעותיים יותר והגרף מתייצב.

ניתן לראות בגרף זה כי יש מגמת ירידה ב – latency ככל שמספר החוטים גדל. הסבר לכך הינו שככל שיש יותר חוטים גודל כל tile קטן ולכן הזמן הדרוש לביצועו קטן בהתאם לציפיות. ניתן לראות כי ככל שמספר החוטים גדל השיפור היחסי קטן זאת מאחר ועדיין יש חלקים שלא ניתנים למקבול, וגם מפני שיש יותר החלפות הקשר. סה"כ הגרף תואם את חוק אמדל.

קיבלנו מעין סריג- מתוך 100 משימות ניתן לראות שכל חוט מבצע בערך 10 מישמות- סה"כ חלוקה יחסית שווינית בין החוטים. הסבר לפיזור יחסית אחיד הוא שכל משימה לוקח יותר מTIME SLICE אחד שמוקצה לחוט מסוים, לא נוצר מצב בו חוט "גונב" משימה מחוט כי הוא סיים לבצע את שלו לפני שנגמר לו הTime Slice

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

שאלות מכווינות אליהם התייחסו בניתוח:

* 1. האם בחלקים מסוימים ניתן לראות מגמה כזאת או אחרת? עליה/ירידה/קו שטוח ? ממה הדבר נובע לדעתכם?

בנוסף להסבר לצד כל גרף בפירוט, ראינו בגרפים בהם יש מגמת ירידה שהשיפור המושג על ידי התכנות המקבילי עולה על המגרעות של הזמן הדרוש עבור החלפות ההקשר, המתנה על המנעולים והקוד הסריאלי. ככל שהטבלה גדולה יותר, החלק היחסי שניתן למקבל גדל ולכן ניתן לראות מגמת ירידה בתחילת הגרף. לעומת זאת, בגרפים עם מגמת עלייה או התייצבות, השיפור היחסי של התכנות המקבילי זניח יחסית להחלפות ההקשר והקוד הסריאלי וכך נקבל דווקא מגמת עלייה ככל שעולה מספר החוטים. ניתן לראות זאת בטבלאות קטנות, בהן השיפור של התכנות המקבילי לא עולה על הזמן הדרוש עבור הקוד הסריאלי ולכן התקבל מגמת עליה. הסבר נוסף להתייצבות (או עלייה) של הגרף הוא שמנקודה מסוימת החוטים הראשונים שנוצרו יסיימו את עבודתם לפני שהחוטים האחרונים התחילו אותה ולכן אין באמת צורך ביצירתם. דבר זה מוביל להגדלת הקוד הסריאלי באופן מיותר.

* 1. מהו מספר החוטים האידיאלי לכל עומס ? הסבירו סיבות לכך.

בכל גרף נחפש את ערך מספר החוטים שיוביל למינימום בגרף. על פי הגרפים שקיבלנו, מספר החוטים האידיאלי עבור הטבלה small הינו 1, בהתאם להסבר בגרף זה שהשיפור של התכנות המקבילי לא יעלה על הזמן הדרוש עבור החלפות ההקשר, המתנה על המנעולים והקוד הסריאלי ולכן לא נקבל שיפור. מספר החוטים האידיאלי עבור הטבלה mid הינו 10. הסבר אפשרי לכך הינו שעבור 10 חוטים השיפור שנובע מהקוד המקבילי הוא הכי גדול ביחס לזמן המנוצל להחלפות ההקשר והקוד הסריאלי. עבור הטבלה הגדולה (big), מספר החוטים האידאלי הינו 30 (בקירוב). הסבר לכך הינו שבטבלה גדולה כזאת חילוק של העבודה למספר חוטים שכזה יוביל לניצול הטוב ביותר של הקוד המקבילי.

* 1. האם בהכרח זמן החישוב היה משתפר אם היינו מוסיפים **מעבדים** נוספים?

המסקנה הנובעת מ-2 היא שלא בכל המקרים הוספת מעבדים נוספים הייתה משפרת את זמן החישוב.

בלוח הקטן והבינוני הוספת מעבדים נוספים לא תעזור. ראינו שעבור הלוח הקטן המקרה של חוט אחד הוא אידאלי. ובמקרה של הלוח הבינוני

* 1. 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.
  2. השוו בין הגרפים של העומס הקטן, גדול ובינוני. במידה ויש שוני, ממה נובע השוני בין הגרפים של העומסים השונים?
  3. 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.