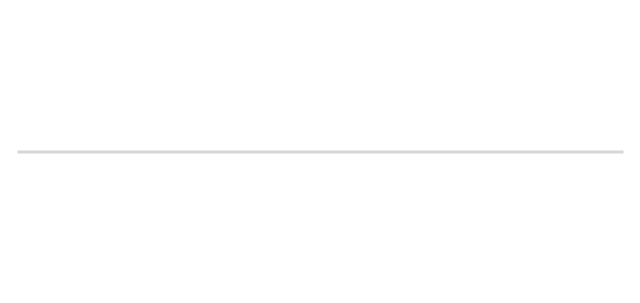
Homework 3 Dry

**אריאל ווייס 204361315  
איליה**

**חלק ראשון: זיהוי כשלי סנכרון**



* 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.   
  
עיקרון הmutual exclusion נפגע – כל חוט אשר לא נעל את המנעול (לא נמצא בלולאה) יכול לשחרר אותו תחילה (אין מניעה לכך) ולאחר מכן לנעול את עצמו – לכן יהיה יותר מחוט אחד שמריץ את הקטע קוד הקריטי.  
עיקרון ההתקדמות מופר במימוש זה. נקח לדוגמה חוט אשר נעל את המנעול (lockVal=0) ונפל באמצע הקוד הקריטי. אף אחד לא ידאג לשחרר את המנעול, כי נניח יש לנו עוד 99 חוטים שנכנסו בזמן של החוט שנפל ועשו lock (לכן הם ממתינים בלולאת while), לכן יווצר Deadlock, שכן אף אחד לא יוכל לשחרר את המנעול (נניח כי אין חוטים נוספים המגיעים אשר עושים Unlock).

* 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?  
       
     מכיוון שרק חוט אחד יכול לבצע את הקוד הקריטי, כל שאר החוטים אשר יכנסו (אין מגבלה על כמו החוטים), יווצר צוואר בקבוק גדול מאוד, כולם ימתינו למנעול. בעצם, בקטע קוד ארוך, כל שאר החוטים הללו ישהו את עבודתם (במקום לבצע משימה אחרת). עבור קטע קוד קצר – זה תלוי. אם יש הרבה מאוד תהליכים שנכנסו במקביל, אנחנו עדיין בבעיה – רק חוט אחד יכול להיכנס בלכ פעם ולכן עדיין יהיו חוטים אשר יצטרכו לחכות הרבה מאוד זמן.

1. 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?   
     
   עיקרון הMutual exclusion נפגע – נניח וישנו חוט כלשהוא שרץ הרבה מאוד זמן בקטע קוד הקריטי והגיע לסף הmax\_iter. כעת, חוט אחר ישחרר את המנעול ולפיכך יהיו 2 חוטים אשר יריצו את הקוד הקריטי.  
   עיקרון הorder נפגע, שכן הרבה מאוד חוטים יכולים לרוץ על הcount (הוא לא ממומש כקטע קוד קריטי) ולאחריו לשחרר ביחד את המנעול ולהריץ את הקוד.
2. בהנחה שהקוד מורץ על מעבד יחיד, הסבר מה ידפיס הקוד הבא, ולמה ?

כיוון שמבוצע join רק על החוט האחרון (חוט מס' 10), מובטח שחוט זה יסיים את עבודתו, אך לא מובטח לנו ששאר החוטים יסיימו את עבודתם עד להדפסת הסכום. לכן הסכום שיודפס אינו צפוי, ונמצא בטווח 10(thread\_wotkload עבור החוט ה-10) עד 55.

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 (זוהי הנחה בקורס בנוגע לכל פקודות האסמבלר למיניהן)

כיוון שאין הגנה על result, וכיוון שהעלאה של result ב1 אינה אטומית- כלומר מתפרקת למספר פקודות מכונה, אשר בינהם יכולה להתבצע החלפת קשר שתגרום להכנסת אותו ערך result בשני החוטים ולכן 'תתפספס' העלאה ערך אחת, במצב המינימלי- בכל שתי העלאות ערך מקבילות, העלאת ערכו של result תתבצע בפועל רק פעם אחת.  
לכן במצב זה יודפס: **100** (שקול לריצה אחת יחידה של do\_calc)  
במצב המקסימלי- לא יתבצעו החלפות הקשר בזמן העלאת ערכו של result ולכן הפונקציה do\_calc תתבצע במלואה פעמיים (פעם אחת מלאה בכל חוט) והערך שיודפס במצב זה: 100+100= **200**.

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

למדנו שבfork מוקצה לבן זכרון שונה משל האב אשר מועתקים אליו המשתנים הקיימים אצל האב, לכן(ונניח שsum לוקאלי) כתיבה אל sum בבן לא תשפיע על הsum של האב ולהיפך (ללא קשר למנעולים).

**חלק שני: Singlephore**

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

typedef struct mutex {

singlephore h;

} mutex;

void mutex\_init(mutex\* m) {

*//TODO*

singlephore\_init(m->h);

}

void mutex\_lock(mutex\* m) {

*//TODO*

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

}

void mutex\_unlock(mutex\* m) {

*//TODO*

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

**{**

1. **סעיף בונוס** (**7 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;

*//TODO*

int waiting;//number of waiting threads

int signals;//number of pending signals

} condvar;

// Initilize the condition variable

void cond\_init(condvar\* c) {

*//TODO*

waiting = 0;

signals = 0;

singlephore\_init(&c->h);

mutex\_init(&c->m);

}

// Signal the condition variable

void cond\_signal(condvar\* c) {

*//TODO*

H(&c->h, INT\_MIN, 1);

mutex\_lock(&c->m);

waiting--;

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) {

*//TODO*

if(signals != 0 && waiting == 0){//init to 0 if no one is waiting:

singlephore\_init(&c->h);

signals = 0;

}

signals++;

waiting++;

mutex\_unlock(m);

H(&c->h, signals, 0);

mutex\_lock(m);

}

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

ברגע שc->h=0 החוט אשר יבצע cond\_wait() לא ילך לישון שכן תנאי הwhile בsinglephore לא יתבצע, לכן הc->h=0 ירד (בצורה תקינה), אבל רק החוט הבא באמת יכנס להמתנה. לדוגמה: חוט 1 מבצע cond\_wait() , c->h=-1 מתעדכן כראוי אבל החוט לא נכנס להמתנה. חוט 2 , אם מבצע cond\_wait() יכנס להמתנה שכן c->h<bound(=0) . במידה שחוט 2 לא יבצע cond\_wait(), חוט 3 יתפספס גם כן, כלומר בכל מצב שבו c->h=0 החוט שיקרא לcond\_wait() לא יכנס להמתנה.  
בעיה נוספת: במידה שהחוטים הראשונים יבצעו cond\_signal בלבד, נשים לב כי c->h יעלה בערכו על אף שאין אף חוט שממתין! במימוש הסטנדרטי, לא קורה כלום, כלומר הסיגנל נעלם- ואילו במקרה של ירמיהו, הסיגנל "נשמר" למרות שאין חוטים שמקבלים אותו. זוהי טעות בקונספט של משתנה תנאי.

**חלק שלישי: ניתוח של החלק הרטוב**

**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.**

1. הסבירו את כוונותיו של פיראס – איך היה אמור המנגנון לעבוד?   
   תחילה היצרן יכניס את כל המשימות (המשימות שלנו במקרה זה) ונעלה את המונה ככמות המשימות שנרצה לבצע. בסיום הכנסת כל המשימות, אם עדיין קיימות משימו שלא בוצעו, היצרן יכנס להמתנה עד שלא ישאר כאלו, כדי שנוכל לעדכן את הgrid של המשחק ולהתחיל לעבוד על הgrid הבא. כל צרכן יבצע עבודה ובסיומה יוריד את המונה באחת. כך שכל עוד לא הגענו ל0 במונה, לא נבצע את ההחלפה בין הgridים, כלומר היצרן לא יתעורר.
2. במימוש זה מספר בעיות Correctness.
   1. מצאו בעיה אחת של Race Condition בפתרון. הסבירו.   
      הdecrease לא אטומי, זה בעייתי מכיוון שיכול להיווצר מצב בו מס' חוטים ניגשים במקביל לפונ' הDecrease כך שבפועל הערך של working ירד בכמות נמוכה יותר מהמתוכנן (שתי קריאות decrease ידרסו אחת את השנייה).
   2. תארו תרחיש שבו מופר ה-Mutual Exclusion. דהיינו, חישוב הלוח curr טרם הסתיים, וה-Producer מבצע למרות זאת את ה-swap של הלוחות.   
      נניח כי יש לנו 3 עבודות. נכניס את העבודה הראשונה לעבודה והיא נכנסת לעבוד ועושה Decrease בזמן שהיצרן עושה Increase. כעת, מפני שהפקודות אינן אטומיות, נניח כי Decrease מנצח כלומר working=-1 . כעת היצרן מכניס שתי עובודות וworking=1. כעת יש לנו בתור 2 עבודות. נניח כי אחת מסתיימת וworking=0. B.wait() לא ימתין וההחלפה תבוצע, למרות שהמשימה האחרונה טרם הסתיימה.
   3. תארו שני תרחישים שונים בהם יתכן Deadlock בפתרון.   
      תרחיש שבו בRace Condition שני חוטים יעשו Decrease ולכן הערך יעודכן רק פעם אחת, כלומר ייתכן מצב שבו working לעולם לא יגיע ל0 שכן לא כל המשימות ביצעו decrease. במצב זה working יהיה תמיד גדול מ0.  
      תרחיש שבו Decrease ינצח את Increase הרבה פעמים ולכן נגיע לערך שלילי בWorking, כלומר working ירד מתחת ל0 וישאר שלילי.  
      בשני המקרים working לא יהיה 0 לעולם כשנגיע לwait(), לכן לולאת הWhile לא תפסיק לעולם.
3. תקנו את class Barrier ואת הפסודו קוד של היצרן-צרכן כך שכל בעיות ה-Correctness יפתרו. אין לשנות את מתווה הפתרון של פיראס באופן מהותי ואין להשתמש בפעולות אטומיות.

class Barrier {

private:

pthread\_mutext\_t mutex;

int cond;

int working;

public:

Barrier(){

working =0;

cond=1;  
 pthread\_mutex\_init(&mutex,NULL);

}

increase(){  
 pthread\_mutex\_lock(&mutex);

working++;   
 pthread\_mutex\_unlock(&mutex);

decrease(){  
 pthread\_mutex\_lock(&mutex);

working--;  
 pthread\_mutex\_unlock(&mutex);

}

wait(){

while(cond){  
 pthread\_mutex\_lock(&mutex);

if(working==0){

cond=0;

}  
 pthread\_mutex\_unlock(&mutex);

}

}

} ;

1. בהתייחסות לסעיף ג: מנגנון סנכרון זה נקרא "מונה משותף", ואינו מוצלח במיוחד ממבט של בביצועים – Performance. הסבר מדוע.   
   מכיוון שבwait, יש לנו מנעול בלולאת הwhile - בכל פעם שאנחנו רוצים לבדוק האם כל החוטים סיימו את עבודתם, נצטרך לנעול אותו, כלומר להרעיב חוטים אחרים באותו רגע: אם יש לנו כמות מאוד גדולה של חוטים המחכים להשתמש במנעול)מנסים לעשות Decrease או Incress), הם יאלצו לנסות "לתפוס" את המנעול בין האיטרציות של wait, והיעילות של הסנכרון תיפגע.
2. **בונוס (7 נקודות):** בנו Barrier אחר, המבוסס על שני מערכים בינריים של באורך N, הפותרים יחדיו את בעיית הסנכרון. על המימוש **לא** להשתמש במנעולים מכל סוג, אך יכול לנצל את הפעולה **האטומית** CompareAndSwap, הנתונה לכם מטה. הניחו שפעולה זו ממומשת **בחומרה**, ולא בתוכנה, כיאה לפעולות אטומיות אחרות. הסבירו למה פתרון זה עדיף מבחינת Performance.

**בונוס:**

class Barrier {

private:

int N;

int \*inc\_arr;

int \*dec\_arr;

public:

Barrier(int N){

int i = 0;

while(i < N){

inc\_arr[i] = 0;

dec\_arr[i] = 0;

}

}

increase(){

int i = 0;

while(i < N){

if(CAS(&inc\_arr[i],0,1)==0) break;

i++;

}

decrease(){

int i = 0;

while(i < N){

if(CAS(&dec\_arr[i],0,1)==0) break;

i++;

}

}

wait(){

int i = 0;

while(i < N){

if(dec\_arr[i] != inc\_arr[i]){

i = 0;

//arrays ar’nt equal, start again.

}

}

//If we get here, the arrays are the equals.

}

} ;

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

**הגרף מופיע בסעיף ב' (s=0)  
הסבר עבור s=0, מצב בו יש מקביליות מלאה: כלומר ככל שנשתמש ביותר חוטים כך נשיג קצב עבודה גדול יותר (ללא תקורה). ניתן לראות כי מכל העקומים, העקום שבו s=0 נקבל את הירידה החדה ביותר של העקום, שכן כתלות במס' החוטים-קצב סיום כל עבודה גדל מאוד ולפיכך זמן הסיום יורד בצורה חדה, עד שמגיעים לנקודה ממנה השיפורים נעשים לעדינים יותר.**

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

**הנחנו שרירותית כי הזמן עבור חוט אחד לוקח 4 שניות (כלומר T1=4), ומספר החוטים המקסימלי הוא 100:**

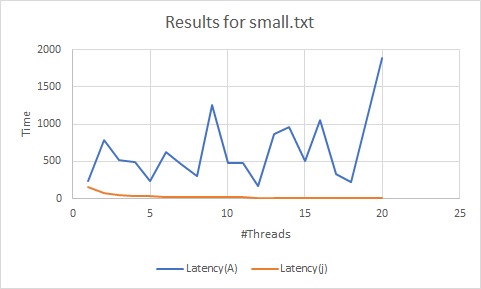
**נשים לב כי עתה, בנוסף לs=0, במצב שבו יש מקביליות מלאה, ישנם גם עקומים נוספים, בהם ערך s יורד ברבע בכל פעם עד שמגיעים לs=1 מצב בו אין מקביליות כלל. ניתן לראות שככל שאנחנו מורידים את ערכו של s, כלומר מורידים את החלק הניתן למיקבול, זמן ביצוע כל עבודה גדל- והLatency יותר גדול. בs=1 נקבל גרף קבוע על זמן עבור חוט 1, שכן אין מקביליות כלל. עבור s=0.75, כלומר 25% ניתן למקבול, נראה כי כבר עבור רבע מכמות החוטים המקסימלית מתקבלת התייצבות מינמילית של הLatency, ועבור כמות חוטים גדולה יותר הזמן התייצב(כלומר הושג שיפור מקסימלי), כל זאת בהתאם לחוק אמדל. בנוסף-  
עבור s=0.5 נקבל כי כבר עבור חצי מהחוטים נקבל כי ישנה התייצבות על הLatency מינימלי, ואילו עבור s=0.25 זה יקרה עבור ¾ מן החוטים.**

ביצירת הגרף השתמשנו בנוסחת אמדל שניתנה בהרצאה.

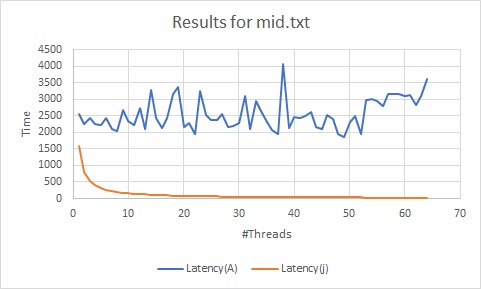
1. הגרפים בהמשך
2. נתחו את הגרפים שהתקבלו באופן **מעמיק.** הניחו בניתוח שהחישוב מבוצע על מעבד יחיד**.**

שאלות מכווינות אליהם התייחסו בניתוח:

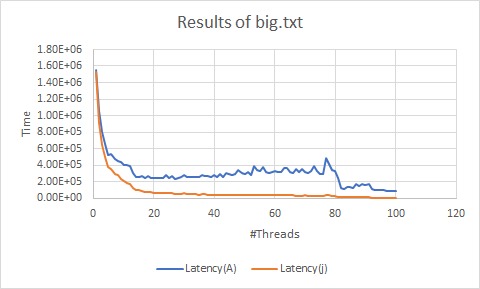
* 1. האם בחלקים מסוימים ניתן לראות מגמה כזאת או אחרת? עליה/ירידה/קו שטוח ? ממה הדבר נובע לדעתכם?
  2. מהו מספר החוטים האידיאלי לכל עומס ? הסבירו סיבות לכך.
  3. האם בהכרח זמן החישוב היה משתפר אם היינו מוסיפים **מעבדים** נוספים?
  4. השוו בין הגרפים של העומס הקטן, גדול ובינוני. במידה ויש שוני, ממה נובע השוני בין הגרפים של העומסים השונים?
  5. האם הגרף מתנהג כמו אחד הגרפים שהתקבלו בסעיפים א',ב' באופן גס? אם כן, כמה מקבילי אתם מעריכים שהקוד שלכם?

**Small:**  
  


1. עבור Latency(j) ניתן לראות מגמה ירידה כתלות במס' החוטים ואילו עבור Latency(A) ניתן לראות כי ישנן עליות וירידות חדות אבל בסה"כ ישנה מגמה עולה ממוצעת (זה קורה מכיוון שכל החלפת הקשר לוקחת יותר זמן מאשר ביצוע זמן המשימה בסה"כ).
2. ניתן לראות כי החל מהחוט הראשון יש לנו עלייה בזמן הביצוע, לפיכך מס' החוטים האידיאלי הינו חוט אחד. עם זאת, ישנן נקודות בהן ישנו ערך זמן נמוך יותר מנק' ההתחלה, אבל לא בערך משמעותי מאוד.
3. נתייחס לכך בסוף
4. נתייחס לכך בסוף
5. נתייחס לכך בסוף

**Medium:**  
  


1. עבור Latency(j) ניתן לראות מגמה ירידה כתלות במס' החוטים ואילו עבור Latency(A) ניתן לראות כי הוא במגמת ירידה עד N=52 והחל מנקודה זאת הוא במגמת עלייה, שכן החל ממס' חוטים זה, החלפות ההקשר יקחו יותר זמן מאשר ביצוע המשימה עצמה.
2. מסעיף קודם, נשים לב כי המס' האופטימלי הינו 52.
3. נתייחס לכך בסוף
4. נתייחס לכך בסוף
5. נתייחס לכך בסוף

**Big:**  
  


1. עבור Latency(j) ניתן לראות מגמה ירידה כתלות במס' החוטים ואילו עבור Latency(A) ניתן לראות כי הוא במגמת ירידה עד N=100 וזה קורה מפני שהחלפות ההקשר כבר פחות משמעותיות לעומת זמן ביצוע העבודה.
2. מסעיף קודם, נשים לב כי המס' האופטימלי הינו 100 (נשים לב כי זהו מס' החוטים המקסימלי האפקטיבי).
3. נתייחס לכך בסוף
4. נתייחס לכך בסוף
5. נתייחס לכך בסוף

**סעיפים c,e,d:**

c) באופן כללי, התשובה היא כן. הוספת מעבדים נוספים לא יכולה להזיק לזמן הכולל שיקח לסיים את המשימה, שכן או שהמעבדים יעזרו להוסיף עוד כוח עיבוד ולתרום למקבליות או שלא יהיו רלוונטיים, כלומר לא ינוצלו.

d) נשים לב כי ככל שהגדלנו את הלוח, מס' החוטים האידיאלי גדל גם כן כפי שהסקנו מהמגמות בגרפים. בנוסף, ככל שהגדלנו את העומס, המקבול תרם יותר לביצועים- כפי שניתן לראות במיוחד מגרף הbig.  
נשים לב שהצגנו בגרפים את הזמן כתלות במס' החוטים האפקטיבי- כלומר מספר החוטים המקסימלי הוא מספר השורות בקובץ העומס בכל גרף.  
e) עבור הגרף של big.txt, ניתן לראות כי באופן כללי ההתנהגות שלו דומה להתנהגות של הגרף מסעיף א- בפרט מגמת הירידה החדה דומה.