**מערכות הפעלה – תרגיל בית מספר 3 – חלק יבש**

**פתרון שאלה מספר 1:**

a.

בעיה בשימוש באותו מחסום מספר פעמים נובעת מכך ש-pthread\_cond\_signal בוחרת באופן שרירותי איזה תהליך להעיר. ניתן דוגמא בה בעיה זאת מתעוררת:

נניח שמשתמש רצה שחמישה תהליכים, נסמנם p1, p2, p3, p4, p5, יבצעו כל אחד 3 קטעי קוד זה אחר זה, נסמנם f1, f2, f3.

בקוד יש זימון בסוף כל קטע f\* לפונ' barrier\_wait. המשתמש מריץ את חמשת התהליכים, כולם מגיעים בזה אחר-זה לשורה pthread\_cond\_wait ונעצרים בה, עד תהליך p5.

p5 מדלג עליה בגלל התנאי, מאפס את הcounter ומזמן את pthread\_cond\_signal, שמעירה תהליך כלשהו.

נניח שהערנו את תהליך p1, אך p5 רץ לפניו, והוא מסיים לרוץ את f2. p5 מגיע שוב ל- pthread\_cond\_wait, היות והcounter מאותחל לאפס בכניסתו לפונקציה.

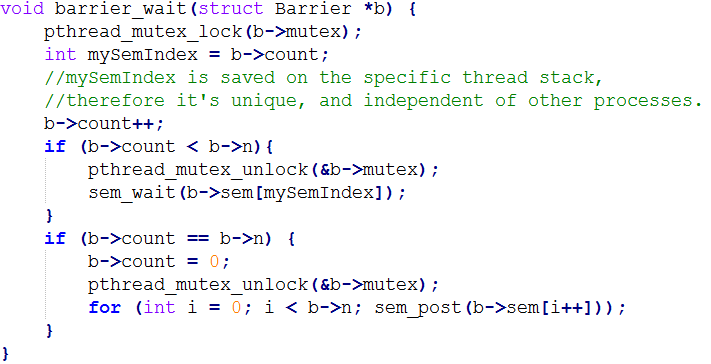
כעת מתחיל p1 את ריצתו, ומזמן pthread\_cond\_signal. היות ולא מובטחת הגינות, כל תהליך יכול להתעורר.

לכן, אם p5 יתעורר, וימשיך את ריצתו, יצא שהוא סיים את מקטע f3, על-אף שהכוונה של המתכנת היתה שכל התהליכים יסיימו את מקטע f1, ואז את f2 ורק אז f3. במקרה שלנו p5 סיים את f3, p1 התחיל את f2, והיתר ממתינים לצאת מהמתנה להתחיל את f2. כלומר p5 חצה וסיים את f3 לפני שחבריו סיימו את f2.

b.

אם במקום pthread\_cond\_signal נשתמש ב-pthread\_cond\_broadcast, ובמקום לעשות זאת בכל פעם, נעשה את זה בתוך התנאי if(b->count == b->n). כל התהליכים יתעוררו ביחד, וזה יפתור את הבעיה שלא ברור איזה תהליך קם מ-pthread\_cond\_signal. ברגע שכולם יתעוררו, פעם הבאה שתיקרא pthread\_cond\_broadcast תהיה כששוב הגיעו n תהליכים לפונקציה barrier\_wait, ולכן לא יקרה שתהליך מסויים שסיים את ריצתו השנייה (למשל) לפני שקודמיו התחילו את הראשונה, ירוץ לפניהם.

c.(הסעיף בוטל אבל אשמח אם הבודק יגיד אם הפתרון לא נכון)



d.

אם הסמפור מופיע לפני פתיחת מנעול, רק התהליך הראשון ייכנס לסמפור, והיתר יחכו על המנעול, כשרק התהליך שעל הסמפור יכול לשחרר את המנעול – Deadlock.

לכן תחילה ישתחרר המנעול, ולאחר-מכן תהיה פקודת sem\_wait.

שוב, נניח שמשתמש רצה שחמישה תהליכים, נסמנם p1, p2, p3, p4, p5, יבצעו כל אחד 3 קטעי קוד זה אחר זה, נסמנם f1, f2, f3.

נניח שתהליכים p1, p2, p3 הגיעו לסמפור אחרי f1 ונכנסו לתור ההמתנה שבו.

נניח שהתהליך p4 משחרר את המנעול, ובדיוק באותו רגע מתבצעת החלפת הקשר, לפני שהספיק להיכנס לתור בסמפור. תהליך p5 (שחיכה על המנעול) נכנס, רואה שיש 4 תהליכים שכבר נכנסו (בגלל הcounter), ומזמן את פקודת sem\_post 4 פעמים. לכן יהיה בצובר בספמפור 1. במידה ושוב בוצעה החלפת הקשר, ודווקא תהליך p3 (או p1/p2) מתעורר, הוא יסיים את f2, יגיע לסמפור, ייכנס ומיד יצא (בגלל שהצובר 1 ולא 0), וימשיך ל-f3, בלי שתהליך p4 אפילו יתחיל את את f2.

לחלופין, אם השתמשנו בקריאה sem\_get\_value על-מנת לבדוק האם ארבעה תהליכים כבר נכנסו להמתנה, אז שוב תהליך p4 להיקטע על-ידי החלפת הקשר מיד אחרי שחרור המנעול, ולפני כניסתו לסמפור. אז תהליך p5 ייכנס לפונקציה, יראה בעזרת sem\_get\_value שיש רק 3 תהליכים בסמפור, ואז ייכנס גם הוא לתנאי. כך גם הוא ייכנס לסמפור, וגם p4, וכך כל התהליכים ייתקעו על הסמפור, כלומר יהיה DeadLock.

**פתרון שאלה מספר 2:**

1. הפתרון אכן מבטיח שסטודנט לא ישאל שאלה כאשר מתרגל עונה על שאלה של סטודנט אחר שכן על מנת להגיע לקטע בקוד בו הסטודנט שואל שאלה (שורה 34) על הסטודנט לקבל את המנעול ולאחר מכן שלא יתקיים התנאי בלולאת הwhile, כלומר המתרגל לא יהיה במהלך מענה על שאלה. בנוסף שינוי הערך של answering מתבצע תמיד בקטע קוד בו המנעול תפוס ועל כן לא ייתכן ששורה 34 תרוץ למרות שהערך של answering הוא TRUE. שורה 11 בה המתרגל עונה על שאלה מתבצעת רק כאשר ערך answering הוא TRUE ועל כן לעולם לא יווצר מצב בו נשאלת שאלה בזמן שניתנת תשובה לשאלה אחרת.
2. הפתרון אכן מבטיח שסטודנטים לא ישאלו שאלה באותו זמן. שורה 34 מתבצעת בקטע קוד קריטי – כלומר באיזור שבו המנעול תפוס ועל כן רק סטודנט אחד יכול לבצע את שורה 34 באותו פרק זמן.
3. לא, הפתרון אינו מבטיח את קיום תנאי 2 בשאלה. ייתכן מצב שבו 2 סטודנטים מעוניינים לשאול שאלה ושניהם נמצאים בשורה 30, כאשר אחד מהסטודנטים תפס את המנעול והשני ממתין לשחרור המנעול. כמו כן המתרגל ממתין על שתנה תנאי בשורה 7 (TA\_cond). כאשר הסטודנט בעל המנעול ימשיך, ישאל את השאלה (34) וישלח סיגנל למתרגל (35) (תתבצע החלפת הקשר לפני 36) והמתרגל יעבור להמתין למנעול (בתוך הפונקציה cond wait), הסטודנט השני ימשיך להמתין למנעול והסטודנט הראשון ימשיך בריצתו. הסטודנט הראשון ישחרר את המנעול (ותתבצע החלפת הקשר), הסטודנט השני יקבל את המנעול (מכיוון שלא מובטחת הגינות בהמתנה למנעול!) וישאל גם הוא שאלה עוד לפני שהמתרגל הספיק לענות על השאלה הראשונה.
4. כלום. אפילו אם תתבצע החלפת הקשר מיד לאחר ביצוע שורה 35 והמתרגל ימשיך בריצתו והמתרגל ימשיך בריצתו או שאחד הסטודנטים ימשיכו בריצתם כך או כך כל קריאה/כתיבה של ערך questions מתבצעת רק כאשר המנעול בידי התהליך ועל כן ההשפעה של ההיפוך לא תשנה דבר. המתרגל ימתין למנעול וכך גם שאר הסטודנטים וכאשר שורה 36 תתבצע והמנעול ישוחרר מובטח מצב תקין של ערך questions והמשך תפקוד זהה של הקוד.
5. ההבדל הוא שבמקום שסטודנט אחד שממתין על משתנה התנאי "יתעורר" (כלומר יצא ממצב המתנה ויעבור להמתין למנעול) כל הסטודנטים שממתינים על משתנה התנאי "יתעוררו". לאחר "התעוררות" ידרוש כל סטודנט מנעול. כל הסטודנטים ימתינו על המנעול ורק סטודנט אחד יקבל את המנעול בכל פעם. הבדל נוסף הוא שאם כל הסטודנטים שרצו לשאול שאלה עד כה (מלבד הראשון שהגיע) ניסו לשאול כאשר הערך של answering היה TRUE אז רק סטודנט אחד מבין הממתינים (על student\_cond) היה שואל שאלות בכל פעם ומתרגל היה עונה וכעט ישנה אפשרות שסטודנטים רבים ישאלו שאלות במקביל, אך התניה זהו מקרה ספציפי ובאופן כללי מצב בו מספר סטודנטים שואלים שאלות בו זמינת היה אפשרי גם בלי השינוי ועל כן באופן כללי התפקוד של הקוד לא משתנה.
6. יווצר באג. מכיוון שהפעולה students--; אינה פעולה אטומית ייתכן שמספר סטודנטים יערכו את ערך students במקביל – לכל היותר אחד שמבצע students++; (שורה 26) ועד 4 שמבצעים student--; במקביל ולמשך אם הערך הוא 4 ושני סטודנטים ביחד מנסים לבצע את הפקודה בשורה 39 אך בין שלב החישוב של הערך החדש לבין שלב ההשמה מתבצעת החלפת הקשר לסטודנט השני (או שני סטודטנים ש"עובדים" בו זמנית אם מדובר במכונה מרובת מעבדים) ייתכן שהסטדנט השני יעדכן את הערך ל3 ואז כשנחזור לסטודנט הראשון גם הוא יבצע השמה עם הערך 3 למרות שכעט כאשר 2 סטודנטים עזבו הערך הנכון צריך היה להיות 2.

**פתרון שאלה מספר 3:**

1.סוגיית ההוגנות של מנעולים עוסקת בשאלה האם כל חוט מקבל את האפשרות לנעול מנעול כאשר הוא חופשי (ניתן לנעילה) ובסדר מתאים. בהרצאה דנו בנושא של הוגנות והצגנו אותו כדרך למניעת הרעבה (במקרה קיצון), כלומר שחוט יבצע את הקטע הקריטי בסופו של דבר. בנוסף, רצינו שההמתנה תהיה מוגבלת וכן העלנו אפשרות לסדר של FIFO. בשורה התחתונה, ניתן להגדיר הוגנות של מנעול כאשר הוא מאפשר לבצע נעילה שלו לפי סדר הבקשות לנעילה. למשל, בתרגולים ראינו את מנעול ה-mutex אשר לא מבטיח כי סדר הנעילה שיתקיים הוא בהתאם לסדר בקשת הנעילה. לעומתו, semaphore מחזיק תור ממתינים הוגן בשיטת FIFO לפי עדיפויות.

2. הרעבה בהקשר של מנעולי סנכרון מתייחסת למצב הבא: כאשר אנו מבצעים נעילה של קטע קוד קריטי מסוים אנו לא בהכרח מקבלים ערובה כלשהי בנוגע לסדר בו החוטים ימתינו על מנת להיכנס לקטע הקוד הזה. לפיכך, קיימת באופן תיאורטי האפשרות שחוט יישאר חסום לתמיד כאשר הוא מנסה להיכנס אל קטע הקוד הנעול וזאת מכיוון שחוטים אחרים מנסים מקבלים גישה קודמת לאותו חוט באופן מתמיד. בשורה התחתונה, הרעבה שכזו כוללת את המצב בו חוט מסוים מנסה לנעול את המנעול אך אף פעם לא יצליח מכיוון שהמנעול כבר נעול. דוגמא בסיסית ביותר לכך היא המקרה בו תהליך מסוים נעל קטע קוד קריטי וכעת מחזיק במנעול בצורה לא מוגבלת ויתר התהליכים לא מקבלים את האפשרות לרוץ בקטע קוד זה ולכן מקיימים את הגדרת ההרעבה.

3. כפי שציינו בהקשר של הגינות מנעולים אנו מעוניינים שלכל חוט/תהליך תהיה הזדמנות להחזיק במשאב מסוים ואף רצוי כי בהתאם לסדר עדיפויות הוגן מסוים. כאשר מתבצעת הרעבה במנעולי סנכרון, אנו מבצעים (כפי שנאמר בשאלה מספר 2) נעילה של קטע קוד קריטי אך לא מבטיחים שיהיה סדר בו החוטים יוכלו להיכנס לקטע קוד זה ועלול להיווצר מצב כי ישנם חוטים שלא מקבלים בכלל את הכניסה לקטע הקוד הזה (וזה בדיוק המצב של הרעבה). כאשר מנעול הוגן הוא ינסה למנוע את מצב זה של הרעבה ע"י קביעת סדר מסוים בין החוטים שיבטיח כי בסופו של דבר לכלל החוטים תהיה האופציה להחזיק במשאב. לדוגמא, במנעול מסוג semaphore שנלמד בתרגולים,קיים תור המתנה הפועל לפי שיטת FIFO ומבטיח כי החוטים ייוכלו להחזיק במשאב לפי סדר מסוים והוגן של עדיפויות. אולם, במידה ומדובר במנעול שאינו הוגן כדוגמאת mutex אנו עלולים להיגרר למצב בו סדר הנעילה של הקטע הקריטי לא מוגדר בצורה מסוימת (בהתאם ל-POSIX) ולפיכך חוט מסוים ביצע נעילה של הקטע ויכול להיוותר מצב נעילה זה "לתמיד" וזאת מכיוון שהפעולה pthread\_mutex\_lock() עלולה לבצע נעילה לפרק זמן שאינו מוגבל. יתר החוטים במקרה זה יגיעו למצב של הרעבה בשל חוסר ההוגנות של מנעול ה-mutex וזאת מכיוון שלא יקבלו את אפשרות הריצה ותתקיים הגדרת ההרעבה.

4. קיפאון או deadlock מתייחס למצב בו תהליך/חוט נכנס למצב המתנה מכיוון שמשאב מסוים מוחזק ע"י תהליך/חוט ממתין אחר אשר גם הוא מחכה למשאב אחר המוחזק ע"י חוט/תהליך אחר וכו'. במידה ותהליך/חוט לא מסוגל לשנות את מצבו אף פעם בגלל שאותם משאבים להם מחכה מוחזקים ע"י תהליך/חוט אחר שגם הוא מחכה לשחרור משאב אחר אז נאמר כי אנו במצב של deadlock. באופן פורמלי, אנו יכולים להגדיר deadlock בתור מצב בו קבוצת תהליכים/חוטים נמצאים כאשר כל אחד מחכה למשאב המוחזק ע"י תהליך/חוט אחר במסגרת קבוצה זו.

5. קיפאון או לחילופין deadlock אכן גורם להרעבה. בעצם deadlock הוא מקרה קיצוני של הרעבה וזאת מכיוון שהוא מתאר מצב בו שניים או יותר חוטים/תהליכים נמצאים במצב המתנה שלא ניתן לספקו. לרוב, מצב deadlock מתייחס למצב בו שני תהליכים/חוטים מחכים כל אחד לפעולה מצד החוט/תהליך האחר ובמצב כזה אכן מתרחשת הגדרת ההרעבה לפיה ישנם חוטים/תהליכים שלא יקבלו את האפשרות להחזיק במעבד שכן החוטים/התהליכים שרצים מחזיקים בו ולא ניתן לשחררם ממצב זה בגלל שיש המתנה הדדית בהתאם להגדרת deadlock כפי שהוצגה בשאלה 4.

6. הפונקציה pthread\_cond\_wait מקבלת כפרמטר mutex מהסיבה הבאה: במידה ולא היינו משתמשים ב-mutex ייתכן והיינו נגררים למצב של "מירוץ" (race condition) כפי שלמדנו במסגרת ההרצאות והתרגולים. כידוע, variable condition או בעברית משתנה תנאי עובד כך שנבדק תנאי מסוים וכל עוד התנאי אינו מתקיים, חוט/תהליך הולך להמתנה עד אשר התנאי נהיה בעל ערך אמת. כאשר אנחנו מדברים על רעיון של מערכת הפועלת במקביל, קיימת האפשרות כי בשלב שבין בדיקת התנאי לבין ההכנסה להמתנה יהפוך ערך התנאי לערך אמת. על מנת למנוע מצב כזה של "מירוץ" עלינו לגרום לצמד השלבים של בדיקת התנאי וההכנסה להמתנה לתפקד כפעולה אטומית וזאת באמצעות אותו mutex על מנת למנוע את המירוץ. בדוגמא הקלאסית שנלמדה בתרגול בנוגע לתור, הרי שבין בדיקת התנאי אם התור ריק לבין שלב ההכנסה להמתנה, יכול להיווצר מצב שאחד החוטים הכניס איבר לתור במידה ואין מנעול הגורם לצמד הפקודות לתפקד כפעולה אטומית.

7. משתנה תנאי מקושר ל-mutex מהסיבה הבאה: ה-mutex המקושר למשתנה התנאי נועד להגן על משתנה התנאי עצמו. על המנעול להיות במצב נעילה לפני הקריאה ל-wait. באמצעות פעולה ה-wait אנו בעצם מעין באופן אטומי מסירים את נעילת ה-mutex. ואז נוכל לספק גישה לחוטים נוספים למשתנה התנאי וכאשר נבצע signal נוכל להעיר חוטים מתוך רשימת ההמתנה. בשורה התחתונה, מנעול ה-mutex המצורף ל-condition variable מספק מנגנון למניעת "מירוץ" (בהתאם להגדרת מירוץ מההרצאות והתרגולים) כאשר חוט אחד מתכונן לבצע wait על משתנה התנאי לבין חוט אחר אשר עלול לקרוא ל-signal בדיוק לפני שהחוט הראשון ביצע את ה-wait הרצוי.

נבחן את המקרה בו לא היה מקושר mutex למשתנה התנאי במסגרת הדוגמא מהתרגול של מנעול קוראים כותבים. לפיכך, בפונקציה read\_lock המוגדרת בשקף מספר 30 של תרגול 7 (סינכרוניזציה) היינו משמיטים את שורת נעילת ה-mutex (השורה הראשונה). היה עלול להיווצר מצב למשל בו בכניסה לפונקציה מספר הכותבים הוא 1 אך גם נוצר מירוץ עם הקוד בפונקציה write\_unlock ולכן בדיוק ירד מספר הכותבים ב-1 ולכן לא בוצעה כניסה לתוך לולאת ה-while שאליה היינו אמורים להיכנס ולכן לא בוצע wait על משתנה התנאי אף על פי שהגדלנו את מספר הקוראים. כלומר, הפעולה לא באמת תתבצע באופן מעין אטומי כפי שציפינו מה שעלול לגרום לשיבוש המנגנון.