struct pipe\_mutex {

int fd[2];

// YOUR CODE HERE (if necessary)

pipe\_mutex() {

int r = pipe(this->fd);

// YOUR CODE HERE (if necessary)

}

void lock() {

// YOUR CODE HERE

}

void unlock() {

// YOUR CODE HERE

}

};

1. מדוע הגישה הזאת פועלת רק בין חוטים של אותו תהליך? ואיך ה-pipe שראינו בתרגול מתגברות על בעיה זאת?

יש במימוש זה הנחה שכל החוטים יכולים לגשת לשדות של אותו הmutex\_pipe . זאת בגלל שיש שימוש בחוצץ משותף על הheap, ולחוטים שונים של אותו תהליך יש heap משותף. לעומת זאת תהליכים שונים אינם חולקים heap ולכן הפתרון בתרגיל לא יתאים (שינוי ב mutex\_pipeשל תהליך אחד לא ישנה את ה mutex\_pipe של תהליך אחר)

בגישה בראינו בתרגול, כדי לשתף מידע בין תהליכם שונים, השתמשנו בFDT שלהם, והפננו ערוצים אל קובץ שיהיה משותף. כל הקבצים הפתוחים של כל התהליכים במערכת נשמרים בטבלה גלובאלית המנוהלת ע"י הגרעין – ה-Global FDT או GFDT. וביצענו הפניה של ערוצים בFDT של התהליכים אל הקובץ.   
החוצץ כפי שנראה בתרגול - אליו נכתב בפועל הpipe, נמצא בהקצאה של הגרעין, ומיוצג ע"י הקובץ, ולכן תהליכים שונים שלא חולקים heap יכולים לגשת אליו

1. איזה מהשינויים הבאים (אם נעשה באופן נפרד) גורם להתנהגות לא מוגדרת בשימוש pipe\_mutex
2. **מחיקת שורה 4** – נקבל ערך זבל התחלתי עבור המרחק בין כותב וקורא
3. מחיקת שורה 5 – אכן מיותר
4. מחיקת " this->blen\_ < BUFSIZ ||" משורה 18. – באג אבל ההתנהגות מוגדרת: תו אחד לכל היותר
5. **מחיקת שורה 21** – לפעמים חריגה ממערך, לפעמים לא, נקרא אחכ תוכן לא נכון
6. מחיקת שורה 23 ו-24 – באג אבל התנהגות מוגדרת – לולאה אינסופית.

הסבר:

1. איזה מהשינויים הבאים (אם נעשה באופן נפרד) גורם לread::pipe\_mutex להחזיר מידע לא תקין (רצף הבתים שמוחזר מ-read לא שווה לרצף הבתים שהועבר ל-write ? (הקיפו כל האפשרויות הנכונות.
2. מחיקת שורה 4 – למשל, קריאה ללא כתיבה לפניה, וb\_len שאותחל זבל ל5
3. מחיקת שורה 5 – אכן מיותר
4. מחיקת " this->blen\_ < BUFSIZ ||" משורה 18 – כתבנו "abc" אבל הלולאה נפסקה לאחר איטרציה אחת ונכתב רק “a” ולכן לא נוכל לקרוא אחכ “abc” .
5. מחיקת שורה 21 – אם חרגנו ממערך, נכתוב את התוכן לא למקום שממנו הקריאה תקרא.
6. מחיקת שורה 23 ו-24 – לולאה אינסופית, לא נקרא שום דבר (ובפרט לא משהו לא תקין) לאחר הקריאה לכתיבה.

הסבר:

1. איזה מהשינויים הבאים (אם נעשה באופן נפרד) גורם ל-write::pipe\_mutex לא להסתיים (שבמצב של מימוש תקין, ה-write תסתיים) ? הקיפו כל האפשרויות הנכונות.
2. מחיקת שורה 4
3. מחיקת שורה 5
4. מחיקת " this->blen\_ < BUFSIZ ||" משורה 18.
5. מחיקת שורה 21 – ייתכן ונכתוב למקום אסור ונייצר חריגת segfault ועל כן הקריאה לא תסתיים בהצלחה
6. מחיקת שורה 23 ו-24 – לולאה אינסופית, כי pos לא מקודם ועל כן תנאי הלולאה ישאר נכון לעד
7. השלימו את המימוש שמצורף למעלה, וודאו שאין deadlock במימוש, ואופן ההמתנה יעיל.

נבהגדרת המבנה, לאחר שורה 3, נוסיף: pthread\_mutex\_t **m**

בקריאה, נוסיף לפני שורה 10: pthread\_mutex\_lock(&**m**);

ואחרי שורה 13, נוסיף: pthread\_mutex\_unlock(&**m**);

בכתיבה, לפני שורה 22, נוסיף: pthread\_mutex\_lock(&**m**);

כמו כן, לאחר שורה 24, נוסיף: pthread\_mutex\_unlock(&**m**);

לא צריך סביב כל הפעולה, כי הקריאה צריכה אורך נוכחי חיובי, וכתיבה רק יכולה להגדיל

כתיבה דורשת אורך נוכחי שקטן מאורך החוצץ וקריאה רק מקטינה, לכן אין מקרה בו נבדק תנאי לאחר הבדיקה הוא מופר על ידי פעולה אחרת.

typedef struct mutex {

singlephore h;

} mutex;

void mutex\_init(mutex\* m) {

singlephore\_init(m->h) //h.val ==0 means open

}

void mutex\_lock(mutex\* m) {

H(m->h, -1, -2) //if h.val is smaller than -1(locked), wait.

Else, val is 0 and set it to -2

}

void mutex\_unlock(mutex\* m) {

H(m->h, -100, 2) //if h.val is smaller then -100 (never), wait.  
 so, in all cases, raise 2 –> back to 0.

}

typedef struct condvar {

mutex m;

singlephore h;

vector<pid\_t> vector;

} condvar;

void cond\_init(condvar\* c) {

mutex\_init(m)

}

void cond\_signal(condvar\* c) {

lock (vector protection)

if vector.empty:

return;

to\_be\_countinued = vector.pop()

unlock (vector protection)

lock (the regular lock)

thr\_continue(to\_be\_countinued)??? Sigcont?

return

}

void cond\_wait(condvar\* c, mutex\* m) {

lock (vector protection)

vector.push(get\_thrread\_pid())

unlock (vector protection)

thr\_suspend(self/ get\_thrread\_pid())??? Sigstop?

unlock (the regular lock)

return

}

1. מה לא תקין בפתרון של ירמיהו?  
    הראו תרחיש אפשרי בו פתרון זה לא עומד בתנאים של סעיף ב'  
   H(&c->h, INT\_MIN, 1); יוצא תנאי שתמיד מתקיים (אף ערך של h.val לא יהיה קטן מ INT MIN)

לכן, יבצע קידום ב1 של ערך h.val גם במקרה שאף אחד לא ממתין. ומקרה זה לא עומד   
בתנאי 1: אם אין חוט שמחכה על משתנה התנאי c ,אז (c(signal\_cond לא יעשה דבר.  
כמו כן, לאחר סיום wait, המנעול הושאר נעול, בניגוד לגישה מהתרגול " הפונקציה cond\_wait() מקבלת שני פרמטרים – מנעול ומשתנה תנאי – ואז משחררת את המנעול ויוצאת להמתנה באופן אטומי"

typedef struct singlephore\_waiter {

-thread pid

-the condition (maybe a pointer to a boolian func that checks the condition)

} singlephore\_waiter;

typedef struct singlephore {

mutex m;

int value;

std::vector<singlephore\_waiter\*> waiters;

// YOUR CODE HERE (if necessary)

} singlephore;

void singlephore\_init(singlephore \*h) {

h->value = 0;

// YOUR CODE HERE (if necessary)

}

void H(singlephore \*h, int bound, int delta) {

singlephore\_waiter hw;

mutex\_lock(&h->m);

while (h->value < bound) {

cond\_wait(&h->cv, &h->m);

}

h->value += delta;

if (delta > 0) {

for itr in vector:

if itr.cond == true:

thread\_signal

}

mutex\_unlock(&h->m);}