פרק #7 תיאום בין תהליכים
Process Synchronization

תהליך משתף-פעולה (a cooperating process) הוא תהליך שעשוי להשפיע או להיות מושפע מתהליכים אחרים במערכת.

תהליך משת"פ עשוי לחלוק מרחב כתובות (בפרט נתונים) עם אחרים, או עשוי להעביר מידע דרך אפיקי תקשורת עם עמיתים.

בפרק זה, עת אנו אומרים תהליך אנו מתכוונים גם לפתיל (עליו במיוחד חל נושא שיתוף נתונים עם פתילים אחרים).

7.1 רקע Background

נניח שברצוננו לתפעל זוג תהליכים: יצרן וצרכן המעבירים זה לזה הודעות באמצעות זיכרון משותף המנוהל כתור וממומש כמערך המתופעל מעגלית וגודלו BUFFER_SIZE תאים: היצרן מוסיף איברים לתור, והצרכן גורע.

לצד המערך, נחזיק מונה (counter) המציין כמה איברים מצויים בתור בכל נקודת זמן. היצרן יגדיל את המונה, הצרכן יקטינו.

המשתנה in יציין לאן יכניס היצרן את האיבר הבא, המשתנה out יציין מהיכן יצרוך הצרכן את האיבר הבא. הם מאותחלים לאפס.

עת התור מלא על היצרן להמתין, עת התור ריק על הצרכן להמתין.

קוד היצרן:

```
while (1) {
   while (counter == BUFFER_SIZE) ;

  buffer[in] = new_item ;
  in = (in +1) % BUFFER_SIZE ;
  counter++ ;
}
```


קוד הצרכן סימטרי:

```
while (1) {
   while (counter == 0) ;

  new_item  = buffer[out] ;
  out = (out +1) % BUFFER_SIZE ;
  counter-- ;
}
```

עתה נניח שהתור מכיל 17 פריטים. היצרן והצרכן פונים אליו במקביל, בפרט מתפעלים במקביל את counter: האחד מגדיל את ערכו, השני מקטינו.

נניח שמ.ה. מתזמנת את הפעולות באופן הבא (תוך שהמעבד מבצע לסירוגין כל תהליך):

צרכן	יצרן
	reg1 = counter
reg2 = counter	
	reg1++
reg2--	
	counter = reg1
counter = reg2	

התוצאה: ערכו של המונה הוא 16 (ולא 17 כפי שראוי).

הסיבה: שני התהליכים פנו למשתנה המשותף במקביל, ותפעלו אותו באופן לא מתואם\מסונכרן.

תקלה דומה אך שונה: עדכונו של המשתנה (על-פי ערכו של אוגר מתאים) ע"י תהליך א' לוקח יותר ממחזור זיכרון יחיד, ובין תחילת העדכון לסיומו מתעניין תהליך ב' בערכו של המשתנה, ועל כן מקבל ערך שגוי.

מצב זה כינינו מצב מרוץ (race condition): מצב בו התוצאה תלויה בתזמון *המקרי* בו המחשב הריץ את התהליכים (ועלולה על-כן להיות שגויה).

הפתרון: מנגנון שיאפשר רק לתהליך יחיד לפנות למשתנה counter בפרק זמן כלשהו.

הסוגיה באופן כללי נקראת: תיאום בין תהליכים.

פתרון אפשרי לבעיה: מנעול (lock).
הרעיון הבסיסי: רק תהליך יחיד יוכל לאחוז במנעול (לנעלו) בכל יחידת זמן.

על מנעול מוגדרות שתי פעולות:

א. נעילה\רכישה (lock/acquire): תהליך המבצע פעולה זו נחסם עד שהמנעול מתפנה, ואז הוא יכול להתקדם, תוך שהוא מובטח שהוא התהליך היחיד שרכש את המנעול (ועל כן היחיד שפונה למשתנה)
ב. פתיחה\שחרור (unlock/release): שחרור המנעול, כך שתהליך אחר יוכל לרוכשו.

הטיפול במשתנים המשותפים, אליהם נרצה לאפשר גישה בלבדית (exclusive) יחייב ראשית רכישת המנעול.
בתום הטיפול במשתנים ישוחרר המנעול.
באופן כזה לא ייווצר מצב מרוץ.

רעיון אפשרי: נגדיר משתנה בולאני שיהווה את המנעול. המשתנה עשוי להיות פתוח או סגור.

בעיה: כדי לרכוש את המנעול מבצע התהליך שתי פעולות: (א) בדיקת ערכו של המנעול (ב) עדכון ערכו, ולכן יש חשש למצב הבא (נניח שהמנעול פתוח):

תהליך א'	תהליך ב'
קרא את מצב המנעול	
	קרא את מצב המנעול
אם ערכו פתוח אזי עדכן את ערכו לסגור	
	אם ערכו פתוח אזי עדכן את ערכו לסגור
טפל במשתנים המשותפים	
	טפל במשתנים המשותפים

ולא השגנו בלבדיות בפניה למשתנים המשותפים.

מסקנה: מנעול הוא רעיון יפה, אולם צריך דרך (יותר) מעודנת, מתוחכמת לממשו מאשר משתנה בולאני פשוט שערכו נבדק ובמידת האפשר מעודכן.

נציג שלושה סוגי פתרונות לבעיה כיצד נממש מנעול, או כיצד נשיג בלבדיות בטיפול במשתנים:

א. פתרונות תכנה: פרוטוקולים (=אוסף כללים) שיקבעו כיצד על התהליכים לנהוג ע"מ שתושג הבלבדיות.
ב. פתרונות חומרה שיסייעו להשיג את התוצאה הרצויה.
ג. פתרונות תכנה מרמה גבוהה: מבני נתונים ופעולות מרמה גבוהה, שיתַמכו ע"י המהדר, ויאפשרו להשיג את הבלבדיות בנוחות (תוך שהמהדר מממש את הפעולות, בד"כ, תוך שימוש במנגנוני חומרה)

7.2 בעיית הקטע הקריטי
The Critical-Section Problem

נניח מערכת בת n תהליכים. לכל אחד מהתהליכים יש קטע קריטי בו הוא פונה למשתנים או קבצים משותפים ועליו לעשות זאת באופן בלבדי.
משמעות המונח קטע קריטי: כאשר תהליך כלשהו נמצא בקטע הקריטי שלו, אף תהליך אחר אינו יכול לבצע את הקטע הקריטי (של האחר). תהליך אחר יהיה אז בקטע השיורי (reminder section) שלו, או ימתין להיכנס לקטע הקריטי (של האחר).

בעיית הקטע הקריטי: מה יהיה הפרוטוקול (כללי ההתנהגות) באמצעותו נממש קטע קריטי.

הפרוטוקול יכלול:

א. קטע כניסה (entry section) = סדרת פעולות אותן צריך לבצע תהליך המעוניין להיכנס לק.ק. שלו.
ב. קטע יציאה (exit section): סדרת פעולות אותן יבצע התהליך בסיום ביצוע הק.ק. שלו.

כאמור, מעבר ל: קטע הכניסה, הק.ק., קטע היציאה, יבצע התהליך קטע שיורי בו הוא אינו מטפל בַמשתנים המשותפים.

שלוש הדרישות מפתרון לבעיית הק.ק:

1. בלבדיות (mutually exclusiveness): רק תהליך יחיד יוכל לבצע את הק.ק. שלו בכל נקודת זמן.
2. התקדמות (progress): אם m תהליכים כלשהם מעוניינים להיכנס לק.ק. (כל תהליך לק.ק. של עצמו), ואף תהליך אינו מצוי בק.ק. אזי תוך זמן סופי אחד המעוניינים יזכה להיכנס לק.ק = אין קיפאון.
3. המתנה חסומה (bounded waiting), אי הרעבה: יש חסם על מספר הפעמים שתהליכים אחרים רשאים להיכנס לק.ק. שלהם אחרי שתהליך P ביקש להיכנס לק.ק. שלו, ועד שבקשתו מסופקת= הוגנות.

7.2.1 פתרונות תכנה לזוג תהליכים בלבד
Two-Processes Solutions

נניח זוג תהליכים: P_0, P_1 בלבד.

נניח, הנחה *שאינה לגמרי מוצדקת* כי כל פעולת מכונה, בפרט השמה, היא אטומית, כלומר מתבצעת בשלמותה בלא שניתן יהיה לראות 'תוצאת ביניים' של הפעולה.

אעיר כי במחשבים מודרניים פעולת השמה עשויה להימשך מספר מחזורי זיכרון, ועל-כן עלול לקרות מצב בו: תהליך א' מתחיל לעדכן משתנה, תהליך ב' קורא ערך לא תקין של המשתנה, תהליך א' מסיים את עדכון ערך המשתנה.

פרוטוקול א'

התהליכים יחלקו משתנה משותף: turn . עת ערכו אפס רשאי P_0 להיכנס לק.ק., עת ערכו אחד, רשאי P_1 להיכנס לק.ק..
המשתנה יאותחל לערך כלשהו.

הקוד של P_0:

```
while (1) {
  while (turn != 0) ;        ← התהליך מבצע עסוקה.
  הקטע הקריטי

  turn = 1 ;
  הקטע השיורי
}
```

נבדוק אילו משלוש הדרישות שהצגנו הפרוטוקול משיג:

א. בלבדיות: מושגת.
ב. התקדמות: אינה בהכרח מושגת: P_0 לא יוכל להיכנס לק.ק. שלו שוב, עד ש- P_1 ייכנס לק.ק. שלו.
ג. המתנה חסומה: מושגת: אם שני התהליכים רוצים להיכנס לק.ק. אזי הם ייכנסו לסירוגין.

על כן ננסה להציע פתרון משופר:

פרוטוקול ב'

בעייתו של פרוטוקול א' הייתה שהוא לא התעניין בשאלה: האם התהליך האחר כלל מעוניין להיכנס לק.ק.: הוא העביר לו את התור בכל מקרה.

על כן עתה נציע שיפור, נגדיר את המערך:
bool want[2] = {false, false} ;
want[i] == true משמע התהליך #i מעוניין להיכנס לק.ק.

הקוד של P_0:

```
while (1) {
  want[0] = true ;    // I want to enter
  while (want[1]) ;   // I wait to my pal

  הקטע הקריטי
  want[0] = false ;
  הקטע השיורי
}
```

נבדוק את תכונות הפרוטוקול:

א. בלבדיות נשמרת: אם חברי רוצה להיכנס אני ממתין.

ב. התקדמות: עלולה שלא להתקיים. נראה דוגמה:

P_0	P_1
want[0] = true	
	want[1] = true
while(want[1]) מתקיים	
	while (want[0]) מתקיים

<u>מסקנה</u>: אנו בחסימה הדדית.

על כן ננסה את פרוטוקול ג':

<u>פרוטוקול ג'</u>

המשתנים:

```
bool want[2] = {false, false} ;
int turn = 0 ;
```

P_0 יבצע:

```
while (1) {
  want[0] = true ;    // I want to enter
  turn = 1 ;          // as a gentlemen
                      // I allow my pal 2 b 1st
  while(want[1] && turn == 1) ;
                      // I wait while it is his
                      // turn AND he also
                      // wants to enter
  הקטע הקריטי
  want[0] = false ;
  הקטע השיורי
}
```


תכונות הפרוטוקול:

א. בלבדיות + התקדמות: נניח ששני התהליכים מעוניינים להיכנס, על-כן want[i] == true עבור שניהם. השני מביניהם שיבצע: turn = other-process יזין אחרון ערך ל- turn לפני לולאת ההמתנה, וזה יהיה הערך שיוותר במשתנה, ועל כן תהליך זה לא יכנס לק.ק., ומשנהו כן ייכנס, וייכנס לבד. (אם רק אחד מעוניין להיכנס, קל לראות שהוא יוכל להיכנס, וברור שהוא נכנס לבד.)

ב. הוגנות:
עתה נניח שהשני שהזין ערך למשתנה הינו איטי. עמיתו המהיר ייכנס לק.ק., יצא, יסמן שאינו מעוניין להיכנס, ונניח שאף יספיק לסמן שהוא שוב מעוניין להיכנס, אך אז הזריז יעניק את התור לַאיטי, ובכך ייתְקע את עצמו, עד שהאיטי ייכנס, יצֵא, ויסמן שאינו מעוניין; או לחילופין, אם האיטי פתאום הפך נורא זריז אזי הוא: יסמן שאינו מעוניין, שהוא שוב מעוניין, ושעתה תורו של חברו. <u>בקיצור</u>: מי שיצא בהכרח מעניק התור למשנהו.

<u>7.2.2 פתרונות תכנה לתהליכים רבים</u>

<u>Multi Process Solutions</u>

בסעיף זה נציג:

א. אלג'\פרוטוקול המאפיה.

ב. מימוש מנעולים באמצעות תכנה

<u>אלג' המאפיה The bakery algorithm</u>

מקור השם: מדוכני מזון מהיר בהם מופעל האלג'.

האלג' מניח:

א. לכל תהליך קיים מזהה ייחודי בתחום 0..n-1.

ב. האלג' עשוי להיות מורץ במערכת מקבילית (בת כמה מעבדים).

<u>סימונים</u>:

א. (a, b) < (c, d) על-פי יחס הסדר הלקסיקוגרפי 'הרגיל'.

ב. $\max(a_0, \ldots, a_{n-1})$ מחזירה ערך הגדול או שווה מכל ה- a_i (i=0, …, n-1).

<u>מבני נתונים</u>:

bool choosing[n] = { false } ;

מציין האם התהליך בוחר עתה מספר בתור (לק.ק.)

int number[n] = {0} ;

המספר בתור של התהליך.

האלג':

```
while (1) {
  choosing[me] = true ;  // I want a number
  number[me] = max(number[0], …,
                   number[n-1]) + 1 ;
                המספר אינו בהכרח ייחודי,
          שכן תהליך אחר במעבד אחר עשוי לבקש
                מספר במקביל 'אלי'
  choosing[me] = false ;

                עבור על כל האחרים:
  for (other = 0; other < n; other++) {
    while (choosing[other]) ;

    while (number[other] != 0 &&
           (number[other], other) <
            number[me], me) ) ;
                כל עוד האחר רוצה להיכנס,
              והוא בעדיפות על-פני אני ממתין
   }
    הקטע הקריטי
  number[me] = false ;
  קטע שיורי
}
```


פתרון שני לבעיית התיאום באמצעות תכנה יניח שהתכנית שלנו יכולה לחסום פסיקות, ועל ידי כך להבטיח שהמעבד לא ייגזל ממנה בין בדיקת תנאי להשמה. (אם הקוד מבוצע ע"י מ.ה., למשל כק.מ. אזי ההנחה אינה בלתי סבירה).

על-סמך ההנחה נוכל לממש מנעולים, כפי שתוארו קודם לכן, באופן הבא:


```
lock_acquire( L ) {
  disable_interrupts ;
  while (L != free) {
    enable_interrupts ;
    diablae_interrupts ;
   }
   L = busy ;
   enable_interrupts ;
}
```

נבדוק:

א. אם המנעול פנוי אזי: אנו חוסמים הפסיקות, בודקים ומגלים שהמנעול פנוי, לא נכנסים ללולאה, נועלים את המנעול, ומאפשרים פסיקות.
כלומר בין הבדיקה לנעילה הפסיקות חסומות ולכן המעבד לא ייגזל מאיתנו.

ב. אם המנעול אינו פנוי: שוב ושוב נכנס ללולאה, בסוף כל סיבוב, לפני בדיקת התנאי (בכניסה לסיבוב הבא) אנו חוסמים הפסיקות; לכן עת נבדוק התנאי אחרי שהמנעול שוחרר (ע"י מי שאחז בו) הפסיקות תהיינה חסומות. בשלב זה לא נכנס ללולאה, ננעל את המנעול, ורק עתה נאפשר את הפסיקות. שוב בין בדיקת ערך המנעול, לנעילתו הפסיקות היו חסומות.

שחרור המנעול:

```
lock_release(L) {
  L = free ;
}
```

מגבלות השיטה:

א. אם התכנית מועפת עת הפסיקות חסומות יש לדעת לאפשרן.

ב. חסימת פסיקות משמעותה עיכוב הטיפול בהן. לכל הפחות, החסימה כאן היא לפרק זמן קצר.

ג. הפתרון אינו ישים למערכת בת מספר מעבדים, שכן שם מעבד אחר עלול לגשת למנעול בין בדיקתו לנעילתו, לגלות שהוא פנוי, ולנעלו גם עבור תהליך אחר.

7.3 חומרת סינכרון
Synchronization Hardware

עד כה ראינו פתרונות לבעיית התיאום באמצעות כלי תכנה. בסעיף זה נציג פתרונות המסתמכים על פקודות הממומשות בחומרה.

הרעיון דומה למה שראינו קודם עת הנחנו חסימה של הפסיקות: החומרה תספק לנו פקודה אשר תתבצע באופן אטומי (כלומר לעולם לא תקטע באמצעה) ואשר תבדוק ערך של משתנה ותשים לו ערך חדש, או תבצע פעולה שקולה אחרת.

הפקודה המוכרת ביותר בנושא זה נקראת TestAndSet והיא מוגדרת באופן הבא:

```
bool TestAndSet( bool &var) {
  bool old_val = var ;
  var = true ;            ← למשתנה מוכנס בכל
                            מקרה הערך 'נעול'
  return old_val ;        ← ומוחזר מצבו הקודם
}
```

כזכור, הפקודה מבוצעת באופן אטומי.

אופן השימוש בה ליצירת ק.ק.:


```
while (1) {
  while (TestAndSet(lock)) ;
        כל עוד מוחזר לי שהמנעול כבר היה נעול
        אני ממתין (המתנה עסוקה)
  קטע קריטי
  lock = false ;

  קטע שיורי
}
        (המנעול יאותחל לערך false = לא נעול)
```

נשים לב שהפתרון מבטיח בלבדיות, והתקדמות, אך אינו מבטיח הוגנות או מניעת הרעבה.

פתרון דומה, בעל תכונות דומות, מציע פקודת swap אטומית:

```
void swap(bool &v1, bool &v2) {
  bool temp = v1 ;
  v1 = v2 ;
  v2 = temp ;
}
```

מימוש ק.ק. באמצעות הפקודה:

נחזיק משתנה גלובלי המוכר לכל התהליכים:

```
bool lock = false ;
```

לכל תהליך יהיה משתנה לוקלי שלו:

```
bool key ;
```

תיאום הגישה לקטע הקריטי:

```
while (1) {
  key= true ;   ← I want to lock
                  (to assign T to the lock)
  while (key == true)
    swap(key, lock) ;  ←
                 swap מכניס ל- key את ערכו של
                 lock, לכן עת ערכו של key הוא
                 false משמע 'תפסנו' את המנעול
                 פתוח, וה- swap שהחזיר לנו
                 איתות על כך, גם נעל את המנעול
                 עבורנו.
  הקטע הקריטי
  lock = false ;
  הקטע השיורי
}
```


כפי שציינו, הפקודות הנ"ל אינן מבטיחות חופש מהרעבה. על כן עתה נציע פרוטוקול שישתמש ב- TestAndSet ויממש את כל שלוש הדרישות מפרוטוקול תיאום.

מבני הנתונים המוכרים לכל התהליכים:

```
bool waiting[n] = { false } ;
```

מציין מי רוצה להיכנס לק.ק. בעזרתו נעביר את הזכות להיכנס בין המעוניינים בזה אחר זה: כל מי שיצא יאתר את הבא אחריו שמעוניין להיכנס ו- 'יזמין' אותו.

```
bool lock = false ;
```

משתנה זה יציין האם אין אף תהליך שמעוניין להיכנס לק.ק. (ואז הראשון שרוצה יכול להיכנס חופשי, חופשי, ורק לנעול אחריו את הדלת)

לכל תהליך יוגדר משתנה לוקלי key כמו קודם.

```
while (1) {
  waiting[me] = true ;  ← (גם) אני רוצה
                          להיכנס
  key = true ;  ← ישמש כמו קודם עם ה-
                  TestAndSet
  while (waiting[me] &&
                  משתנה זה עשוי לשנות את ערכו
                  עת 'חבר' 'יזמין' אותי להיכנס אחריו,
                  כאשר הוא יצא וישנה את ערך המשתנה
                  עבורי
          key
                  key ישנה את ערכו אם רק אני
                  רוצה להיכנס לק.ק.
                  (ואז lock == false)
          )
          key = TestAndSet(lock) ;
  waiting[me] = false ;
  קטע קריטי
```


קוד היציאה:

```
  other = (me +1) % n ;
  while (other != me && !waiting[other])
    other = (other +1) % n ;
```

כאשר אני יוצא אני מחפש אם יש מישהו שהכריז שברצונו להיכנס.

```
  if (other != me)  אם מצאתי כזה
    waiting[other] = false ; יאללה, חבר, כנס
  else
    lock = false ;   אף אחד לא מעונין להכנס
  הקטע השיורי
}
```


ב- IA32 קיימת הפקודה הבאה:

```
compare&swap(mem, R1, R2) {
  if (mem == R1) {
    mem = R2 ;
    return true ;
  }
  return false ;
}
```

<u>תרגיל</u>: כיצד נממש בעזרתה את TestAndSet?

7.4. סמפורים Semaphores

<u>סמפור</u> = שיטת תקשורת המבוססת על שימוש בדגלים לאיתות (לכל אות קיימת תנועת דגל מיוחדת). בהקשר שלנו: התהליכים יאותתו זה לזה האם ניתן\אסור להיכנס לק.ק.

הסמפור ישמש אותנו, למשל, כדי להשיג מנעול.

סמפור הוא משתנה שלם (int), הניתן לאיתחול (לַערך אחד [או יותר]).
מעבר לאיתחולו מוגדרות עליו שתי פעולות <u>אטומיות</u> המתנה\נעילה ואיתות\שחרור:

```
wait(s) {
  while (s <= 0) ;
  s-- ;
}

signal(s) {
  s++ ;
}
```

וכיצד ישמש אותנו הסמפור לצורך הגנה על ק.ק.:

נגדיר משתנה שיוכר לכל התהליכים: mutex = 1;

כל תהליך יבצע:

```
while (1) {
  wait(mutex) ;
```
פעולה זו עוצרת את התהליך כל עוד ערכו של המשתנה אינו חיובי.
```
  הקטע הקריטי
  signal(mutex) ;
```
עת סיימת שחרר הסמפור לאחרים: הגדל את ערכו באחד.
```
  קטע שיורי
}
```

סמפור יכול לסייע לנו גם במשימות סנכרון אחרות, לא רק בק.ק..
לדוגמה: נניח ש- P2 יכול לבצע קטע קוד רק אחרי ש- P1 'הכין' לו נתונים, כלומר סיים לבצע קטע קוד.
נוכל לכתוב: ראשית synch = 0;
ועתה P1 יבצע:
קוד ההכנה
signal(synch);

P2 יבצע:
wait(synch);
קוד השימוש

7.4.2 מימוש סמפורים

הפתרונות שהצגנו עד כה בין בתכנה ובין בחמרה חייבו המתנה עסוקה (busy waiting): בקוד הכניסה שלו, התהליך מבצע לולאה ריקה .
המתנה עסוקה היא בגדר בזבוז זמן מעבד, במיוחד אם הק.ק. יחסית ארוך, ויש להמתין זמן רב עד שמי שנמצא בו יפנה אותו.

סמפור עם המתנה עסוקה נקרא גם: spinlock שכן: process spins while waiting for a lock.

במערכת בת כמה מעבדים, עת הק.ק. הינו קצר, לעתים נעדיף המתנה עסוקה על-פני שתי החלפות הקשר; אך במערכת של מעבד יחיד, או עת הק.ק. ארוך נעדיף שהתהליך ילך לשון עד שהוא יוכל להיכנס לק.ק., ואז הוא יוער.

נראה כיצד ניתן להשיג את התוצאה בעזרת סמפור:
נגדיר:
```
struct Sleepy-Semaphore {
  int value ;          כמו בעבר
  struct Process *waiting ;   רשימת תהליכים הממתינים עליו
}
```


פעולת ה- wait תוגדר:
```
void wait(Sleepy-Semaphore s) {
  s.value-- ;
  if (s.value < 0) {
    add yourself to s.waiting
    go to sleep
  }
}
```
ובהתאמה, signal:
```
void signal(Sleepy-Semaphore s) {
  s.value++ ;
  if (s.value <= 0) {
    remove a process from s.waiting
    and wake it up
  }
}
```
מכיוון שכל אחת משתי הפעולות גם משנה ערך של משתנה, וגם בודקת את ערכו אזי יש לדאוג שהן תבוצענה באופן אטומי.

אם עומד לרשותנו spinlock נוכל להשתמש בו: נגדיר כל פעולה בתוך ק.ק. (בין wait ל- signal על ה- spinlock) ואז היא ודאי תבוצע אטומית.

אבל אז מה הועילו חכמים בתקנתם?
אם בכל מקרה אנו משתמשים ב- spinlock בשביל מה לנו הסמפור הנוכחי (הישנוני)?

נסביר:
נניח שהק.ק. עליו ברצוננו להגן כולל אלף פקודות, ולעומת זאת פעולת ה- wait/signal על הסמפור הישנוני כוללת עשר פקודות.

עת אנו משתמשים רק ב- spinlock תהליך שממתין לפתחו של הק.ק ממתין המתנה עסוקה למישהו 'שצועד' אלף פקודות, משמע ממתין זמן רב (ומבזבז זמן מעבד רב).

עת אנו משתמשים בסמפור הישנוני: תהליך שרוצה להיכנס לק.ק. ראשית צריך להיכנס לקטעון קריטי (בן עשר פקודות). זאת הוא עושה תוך שימוש בהמתנה עסוקה, אולם היא, סביר להניח שתהיה קצרה. עתה הוא נכנס לקטעון הקריטי, ואז או שנכנס לק.ק. 'האמיתי' או שהולך לשון.

תהליך שיוצא מהק.ק. הארוך:
עת משתמשים ב- spinlock צריך לבצע מעט מאוד (signal של spinlock): להגדיל ערך הסמפור.
לעומת זאת, עת משתמשים בסמפור ישנוני יהיה עליו ראשית לבצע קטע כניסה לסמפור spinlock, עתה הוא יבצע קטע קוד קצר: סיגנל של הסמפור הישנוני, ואז signal של spinlock.

אם הק.ק. המרכזי ארוך אזי הרווחנו.

7.4.3 חסימות הדדיות והרעבה
Deadlocks and Starvation

נניח שאנו משתמשים בזוג סמופרים: s1, s2.
נניח ששני תהליכים המורצים במערכת מבצעים את סדרת הפקודות הבאה:

P2	P1
	wait(s1) – can proceed
wait(s2) – can proceed	
	wait(s1) – has to wait
wait(s2) – has to wait	

התוצאה: כל אחד מהתהליכים ממתין ל- signal שישלח לו עמיתו (התקוע) ולכן המערכת מצויה במצב של חסימה הדדית.

הגדרת חסימה הדדית (ח.ה.): קבוצת תהליכים מצויה בח.ה. אם כל תהליך בקבוצה ממתין לאירוע אשר יכול להיגרם רק ע"י תהליך אחר בקבוצה.

בעיה דומה, אך שונה מח.ה. היא החסימה הבלתי מוגבלת (indefinite blocking) או הרעבה (starvation): התהליך ממתין עד בלי די בלי לקבל את המשאב\שרות לו הוא זקוק.

פתרונות החומרה שמנינו עלולים לצור הרעבה, כך גם סמפור אשר ההמתנה עליו אינה בתור.

7.4.4 סמפורים בינאריים Binary Semaphore

את הסמפורים שתיארנו עד כה ניתן היה לאתחל לכל ערך טבעי, ועל כן הם נקראים סמפורים מונים (counting semaphore).

בניגוד להם, סמפור בינארי עשוי לקבל את הערך אפס או אחד בלבד.

לעתים, החומרה או התכנה שעומדות לרשותנו (למשל, ספריית POSIX) מספקות לנו רק סמפור בינארי.

סעיף זה מציג כיצד נממש סמפור מונה באמצעות זוג סמפורים בינאריים (ומשתנה שלם).

נגדיר:

binary-semaphore sem_for_counter=1,
sem_for_cs=0 ;

int counter = … ערכו של הסמפור המונה הדרוש לנו

פעולות ההמתנה והשחרור על הסמפור המונה:

המתנה:

```
wait(sem_for_counter) ;
counter-- ;
if (counter < 0) {        לא ניתן להיכנס לק.ק.
   signal(sem_for_counter) ;
                          שחרר הטיפול במונה
                          והמתן הפתח הק.ק
   wait(sem_for_cs) ;
}
signal(sem_for_counter) ;
```

נשים לב שאם הק.ק. היה פנוי אנו עושים signal יחיד ל: sem_for_counter, ואחרת (אם המתנו) אנו עושים שניים. נראה מוזר...

שחרור:

```
wait(sem_for_counter) ;
counter++ ;
if (counter <= 0)      אם יש ממתינים לפתחו של
                       הק.ק. אזי הער אחד מהם:
          signal(sem_for_counter)  (ואל תעשה
       כי המוער יעשה זאת במקומך עת הוא יוקץ)
   signal(sem_for_cs) ;
else
   signal(sem_for_counter) ;
```


7.5 בעיות סינכרון קלאסיות
Classic Problems of Synchronization

סעיף זה חורג ממ.ה. ומציג מספר בעיות קלאסיות של בקרה מקבילית (concurrency control), בעיות עליהן נהוג לבחון מנגנונים ופרוטוקולים של סנכרון (כדוגמת אלה שהוצגו).

7.5.1 בעיית החוצץ המוגבל
The Bounded-Buffer Problem

בעיה זו כבר ראינו: בהינתן מאגר של n תאים (כל אחד מכיל פריט בודד) יש לדאוג לסינכרון מילויו וריקונו ע"י יצרנים וצרכנים (אחד או יותר מכל סוג).

פתרון *אפשרי* של הבעיה באמצעות סמפורים ישתמש בשלושה סמפורים שמאותחלים כמתואר:

mutex = 1 מבקר את הגישה למאגר.
empty = n מונה כמה מקומות פנויים קיימים במאגר.
full = 0 מונה כמה פריטים מצויים במאגר.

קוד יצרן:

```
while (1) {
   nextp יצר איבר נוסף במשתנה
   wait(empty) ;          המתן כל עוד אין מקום
                          והקטן מספר הפנויים\פנויות
   wait(mutex) ;          המתן לשם פניה למאגר עצמו
   הוסף את האיבר שייצרת למאגר
   signal(mutex) ;
   signal(full) ;         אותת לצרכנים שנוסף איבר
 }
```

קוד צרכן:

```
while (1) {
   wait(full) ;           המתן כל עוד יש אפס מלאים
                          והקטן באחד את מספר המלאים
   wait(mutex) ;
   הסר איבר מהמאגר
   signal(mutex) ;
   signal(empty) ;        אותת ליצרנים שנוסף מקום
                          פנוי
}
```


7.5.2 בעיית הקוראים-כותבים
The readers-Writers Problem

נניח אובייקט אליו פונים שני סוגי תהליכים:

א. קוראים מעוניינים רק לשלוף נתונים מהאובייקט.

ב. כותבים מעוניינים (גם) לעדכן את ערכו של האובייקט.

קוראים רבים עשויים לפנות לאובייקט במקביל, כותב חייב לפנות לאובייקט לבד (כדי שלא יקרה מצב בו קוראים קיבלו נתוני עדכון חלקי, וכך שגוי, של האובייקט, או ששני כותבים פגעו זה בנתונים של זה).

לבעיה שני וריאנטים הנבדלים בקדימות שתינתן לתהליכים השונים:

א. בעיית הק\כ הראשונה: קורא יידרש להמתין רק אם כבר קיים כותב שזכה באפשרות גישה לאובייקט. (במילים אחרות: אם יש כבר קורא שניגש למ"נ, ועל-כן יש כותבים שממתינים, הקורא החדש יוכל להצטרף לקוראים הקיימים).

ב. בעיית הק\כ השנייה: עת כותב מעוניין לכתוב הוא יוכל לעשות זאת מהר ככל האפשר, כלומר שום קורא לא יוכל לקבל אפשרות קריאה.

בשני הוריאנטים תיתכן הרעבה: בראשון של כותב, בשני של קורא.

נציג פתרון לווריאנט הראשון:

```
semaphore rc_mutex = 1,  // רק לק'
          wrt = 1,       // לק' ול-כ'
int readers_count =0 ;   // מונה ק'
                         // מוגן ע"י rc_mutex
```

כותב:

```
wait(wrt) ;          ← המתן עד שאין לא ק' ולא כ'
כתוב
signal(wrt) ;
```

קורא:

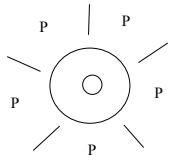
```
wait(rc_mutex) ;          ← המתן בפניה למונה
readers_count++ ;
if (readers_count == 1) ← אתה הקורא הרשון
   wait(wrt) ;            ← המתן לכ' פעיל
                            וחסום כ' נוספים
signal(rc_mutex) ;        ← סיימת על המונה
```

קרא

```
wait(rc_mutex) ;
readers_count-- ;
if (readers_count == 0) ← אתה הק' האחרון
   signal(wrt) ;          ← הזמן כותבים
signal(rc_mutex) ;
```


7.5.3 בעיית הפילוסופים הסועדים
The Dining-Philosophers Problem

בעיית הפילוסופים הסועדים מנוסחת באופן הבא:
סביב שולחן עגול יושבים חמישה פילוסופים.
הפילוסופים עשויים לחשוב או לאכול.
עת פילוסוף שקוע בהגיגיו אין לו אינטראקציה עם עמיתיו.
עת הפילוסוף נתקף רעב, ברצונו לאכול מסיר האורז המונח במרכז השולחן. לשם כך הוא נזקק לשני מקלות אכילה.
ע"ג השולחן, בין כל שני פילוסופים, מונח מקל אכילה יחיד.



הפעולה האטומית שמבצע פילוסוף (רעב): הרמת מקל אכילה יחיד.
עת הפילוסוף זכה בשני מקלות הוא יכול לאכול; ועת הוא מסיים לאכול הוא מניח בנחת את המקלות, וחוזר לסרעפיו.

המטרה: להביא לסיפוקם הרוחני, וחשוב מכך הגשמי, של הפילוסופים, בלי להיקלע לח.ה. חו"ח, או להרעבה.

נציג מספר פתרונות אפשריים, ונבחנם:

<u>פתרון א'</u>

עבור כל מקל אכילה נחזיק סמפור שיאפשר להמתין כל עוד הוא אינו פנוי:

```
semaphore chopstick[5] = {1, 1, 1, 1, 1} ;
```

כל פילוסופית (שמספרה i) תבצע:

```
while (1) {
  wait(chopstick[i]) ;      ← המתיני\הרימי המקל שמימינך
  wait(chopstick[(i+1) % 5]) ;  ← כנ"ל לזה שמשמאלך
  איכלי
  signal(chopstick[i]) ;
  signal(chopstick[(i+1) % 5]) ;
}
```

נדון בפתרון:

אם כל הפילוסופיות תתקפנה ברעב במקביל, אזי כ"א מהן תרים את המקל שמימינה, ותמתין עד בלי די למקל שמשמאלה. כלומר: חסימה הדדית.

שיפורים אפשריים:

א. פתרון קל: נוסיף מקל אכילה שישי: כך לפחות אחת הפ' תוכל לאכול, לשחרר את מקלותיה, ואז לפי תור גם כל האחרות תוכלנה לשבוע.

ב. הרמת זוג מקלות האכילה תבוצע בק.ק., ואם רק אחד מהם פנוי יש לשחרר את זה שנרכש. פתרנו את בעיית החסימה ההדדית, אך נקלענו לבעיה חלופית: חשש להרעבה.

ג. נמסְפֵר את הפ'. פ' פרדית תתחיל בהרמת המקל שלשמאלה, זוגית תתחיל בהרמת המקל שמימינה. בדקו שגם כאן נחלצנו מהח.ה. אך לא מחשש להרעבה (פ' איטית יושבת לצד פ' מהירה וזללנית שאוכלת לעתים מזומנות).

נוותר כרגע רעבים לפתרון. נחזור ונציג פתרון (לא שלם) לבעיה בסעיף 7.7.

7.6 תחומים קריטיים Critical region

סמפורים הם כלי שימושי ומקובל, וכפי שנראה בהמשך, הן ביוניקס והן ב- POSIX קיימים סמפורים.

בעייתם של הסמפורים שהם כלים יחסית מרמה נמוכה, וככאלה מועדים לתקלות (עת תישמט פעולת wait/signal או תוחלף פעולה אחת באחרת). על כן בשפות תכנות שונות הוצעו כלים מרמה גבוהה יותר שיסייעו להשיג בלבדיות בצורה 'נקייה' יותר. כמובן שאת המבנים הללו צריך אחר הקומפיילר לממש עבור המתכנת תוך שימוש בסמפורים או בכלים אחרים מרמה נמוכה יותר.

נכיר עתה שתי הצעות בתחום זה: תחום קריטי ו- מוניטור.

ראשית נדון בתחום הקריטי, הקרוי גם <u>תחום קריטי מותנה</u> (conditional critical region).

נניח כי מספר תהליכים\פתילים חולקים משתנה משותף v בו עליהם לטפל באופן מתואם (בלבדי). מעבר לכך לכל תהליך יש משתנים פרטיים משלו.

כדי לגשת ל- v באופן בלבדי נכלול בקוד כל תהליך את הפקודה:

```
region v when(<condition>) <statement>;
```

פקודת התחום הקריטי מבטיחה לנו שהתהליך שלנו יבצע את הפקודה <statement> רק עת:

א. התנאי <condition> מסופק, וכן:

ב. אף תהליך אחר אינו מצוי בתחום הקריטי של v.

על כן, אם שני תהליכים מבצעים במקביל:

```
region v when(true) s1 ;
```

וכן:

```
region v when(true) s2 ;
```

אזי אנו מובטחים ש- s1 ו- s2 לא תבוצענה במקביל (בד"כ s1, s2 תטפלנה ב- v). ראשית תתבצע אחת מהן, ובינתיים התהליך השני ימתין, ורק בתום ביצוע האחת יוכל התהליך השני לבצע את השנייה.

דוגמה לשימוש בקטע קריטי: בעיית החוצץ המוגבל.

נניח כי הגדרנו את המשתנה המשותף:

```
struct Buffer {
   item pool[N] ;
   int count,        ←מספר האיברים במאגר
   int in, out ;     ←מ\היכן יוצא\יוכנס האיבר
} buffer ;
```

עתה היצרנ(ים) יבצע(ו):

```
region buffer when(count < N) {
   pool[in] = next_item ;
   in = (in +1) % N ;
   count++ ;
}
```

צרכנים יבצעו:

```
region buffer when(count >0) {
   next_item = pool[out];
   out = (out +1) % N ;
   count-- ;
}
```

ובאחריות הקומפיילר יהיה לדאוג שרק יצרן\צרכן אחד לכל היותר יהיה בקטע הקריטי בכל נקודת זמן.
(בספר תוכלו למצוא כיצד ימומש ק.ק. באמצעות סמפורים).

7.7 נוטרים\שומרים Monitors

פתרון אחר, מרמה גבוהה, הקיים, למשל בג'אווה, הוא אובייקט מסוג monitor המבטיח גם הוא בלבדיות בטיפול במשתנים משותפים.

מתכנת המוניטור יגדיר:

א. משתנים\מבני-נתונים שיכללו במוניטור (בדומה ל- data members באובייקט רגיל).
ב. קטע אתחול שיבוצע למבני הנתונים (בדומה ל- constructor).
ג. סדרת פעולות\שיטות שתכללנה במוניטור, ותבוצענה באופן בלבדי (בניגוד לשיטות באובייקט רגיל). כלומר רק תהליך אחד לכל היותר יוכל להריץ שיטות אלה בכל נקודת זמן.

כדי להוסיף כוח למוניטור, מוגדר בו גם הטיפוס:
condition v1, v2 ;
נדון בטיפוס זה בהרחבה בהמשך (ב- POSIX)
אך נציגו כבר עתה:
עת תהליך מבצע: v1.wait(); הוא הולך לשון, עד שתהליך אחר מבצע: v1.signal();.
אם כמה תהליכים בצעו v1.wait() אזי עת תהליך מבצע v1.signal() יוער *אחד* הישנים.
אם תהליך מבצע v1.signal() ואין אף תהליך ישן אזי לפעולה אין כל משמעות, היא אובדת (בשונה לגמרי מ- signal על סמפור): אין את מי להעיר.

כלומר תפקידו של משתנה התנאי הוא לאפשר לתהליך ללכת לשון עד שהוא יוכל לפעול (להיכנס לק.ק).

נשים לב לנקודה מעט עדינה ביחס למשתנה תנאי במוניטור: נניח ש- P1 הלך לשון על משתנה תנאי, עתה P2 מעיר אותו. התוצאה: לכאורה יש שני תהליכים פעילים במוניטור, בניגוד לכללי המוניטור.
פתרונות אפשריים:
א. המעיר יושהה עד שהמוער יסיים את הרצת השיטה שהוא מריץ.
ב. המוער ימתין עד שהמעיר יסיים את הרצת השיטה שהמעיר מריץ.
פתרון ב' הינו טבעי יותר: המעיר כבר רץ, נעדיף שלא להשהותו ואז להעירו שוב.
מנגד יש חשש שעד שהמוער יזכה לרוץ התנאים ישתנו והוא שוב לא יוכל לפעול.

נדגים שימוש במוניטור לשם הצגת פתרון חופשי מח.ה. (אך לא מהרעבה) לבעיית הפילוסופים הסועדים:
(קוד המוניטור יופיע בגופן מיוחד כדי להבדיל בינו לבין ההסברים המרובים שסובבים אותו)


```
monitor dining_philosophers {
   // data members:
   enum {thinking, hungry, eating}
         state[5] ;
   condition sleep_wakeup[5] ;
```

המערך state מייצג את מצבה של כל פילו'.
כדי שפילו' תוכל לאכול צריך ששתי שכנותיה לא תאכלנה:
state[(i+1)%5] != eating &&
state[(i+4)%5] != eating
אם פילו' רעבה גילתה שהנ"ל אינו מתקיים, והיא אינה יכולה לספק את צרכיה הגופניים, אזי בעזרת sleep_wakeup[i] היא תלך לשון, ותוער ע"י שכנתה, עת האחרונה תגמור לאכול (ואז הפילו' תוכל *לבדוק* האם עתה יש באפשרותה לזלול).

```
  // initialization/constructor:
  void init() {
      for (int i=0; i< 5; i++)
         state[i] = thinking ;
  }
```

מעבר לאיתחולים תכיל המחלקה\המוניטור שלוש שיטות\פעולות:

א. pickup תבוצע ע"י פילו' החפצה לאכול, עשויה להשהותה (בעזרת משתנה התנאי). מנגד, אם תושלם בהצלחה, תעביר את הפילו' למצב אכילה.

ב. putdown תבוצע ע"י פילו' שסיימה להגשים את צרכיה הגופניים, ומעוניינת לפנות למנוחת צהריים, מדיטציה, או פעילות רוחנית אחרת. תשלח סיגנל הערה לשכנותיה (שאולי מושהות).

ג. test תקרא ע"י שתי השיטות האחרות. בודקת האם פילו' יכולה לאכול, ואם כן *מעבירה אותה למצב אכילה*.

כל פילו' i תבצע:

```
while (true) {
  קטע שיורי
  dining_philosophers.pickup(i) ;
  אכילה \ קטע-קריטי בו משתמשים במשאב:
                    זוג המקלות
  dining_philosophers.putdown(i) ;
}
```

כלומר זה הקוד *שישתמש* במוניטור (ולא חלק מהגדרתו)

נפנה עתה להגדרת השיטות\פעולות במוניטור:


```
void pickup(int philo) {
   state[philo] = hungry ;
   test(philo) ;   ←        נסי להעביר את
                            מצבך ל-'אוכלת'
   if (state[philo] != eating)
                            אם לא הצלחת, לכי לשון:
      sleep_wakeup[philo].wait() ;
                            תוערי ע"י שכנתך עת תוכלי לסעוד
}

void putdown(int philo) {
                            העבירי עצמך למצב חשיבה
   state[philo] = thinking ;
                            ובדקי האם ניתן\צריך להעיר אחת משכנותיך:
   test( (philo +1) % 5) ;
   test( (philo +4) % 5) ;
}
```



```
void test(int philo) {
   if (state[philo] == hungry &&
                            הנ"ל תנאי טיפשי עת test נקראת ע"י פילו'
                            עבור עצמה; אך דרוש עת השיטה נקראת
                            עבור פילו' ע"י שכנתה שסיימה לאכול
      state[(philo +1) %5]!=eating &&
      state[(philo +4) %5]!=eating )
   {
      state[philo] = eating ;
      self[philo].signal() ;
                            גם לפעולה זאת יש משמעות רק אם test
                            נקראת עבור פילו' ע"י שכנתה, שכך מעירה
                            אותה (אם הפילו' מבצעת את הפעולה עבור
                            עצמה, משמע היא ערה, אין לפעולה משמעות)
   }
}
} // end of monitor definition
```


7.8 אמצעים לסנכרון תהליכים ביוניקס

שניים מהכלים שראינו בדיוננו התיאורטי ממומשים ביוניקס:

א. סמפורים.

ב. משתני תנאי.

7.8.1 סמפורים

כזכור(?) הרעיון הבסיסי בסמפור הוא שהסמפור מבטיח לנו ששתי הפעולות המבוצעות בו:

א. בדיקת ערכו, ובמידת הצורך המתנה עליו.

ב. שינוי ערכו.

תבוצענה באופן אטומי כלומר בלי שהמעבד 'ייחטף' מידי התהליך שמבצע אותן.

רעיון זה ממומש בסמפור של יוניקס.

עת יוניקס מעמידה לרשותנו סמפור אנו מקבלים למעשה קבוצת סמפורים שתכיל מספר סמפורים כרצוננו.

הטיפול בסמפורים יהיה בדומה לטיפול בזיכרון משותף או בתורי הודעות שגם מקורם ב- System V והם ממשפחת XSI IPC. (כולם ממשיכים להתקיים גלובלית במערכת עד שמשחררים אותם מפורשות [ipcs, ipcrm])

א. <u>יצירת סמפור בעזרת ק.מ. semget()</u>

כדי לקבל קבוצת סמפורים נתחיל בכך שנייצר מפתח לקבוצה:

```
key_t key = ftok("~yoramb", 'y') ;
if (key == -1) …
```

תוך שימוש במפתח נבקש לצור קבוצת סמפורים, או להיקשר לקבוצה קיימת:

```
int sem_set_id =
        semget(key,
               1,
               IPC_CREAT | 0600) ;
if (sem_set_id == -1) …
```

המפתח החיצוני
כמה סמפורים בקבוצה

עת אנו מעוניינים להצטרף לסמפור קיים ניתן להעביר אפס בארגומנט השני (בדומה לז"מ).

קיבלנו קבוצת סמפורים *בלתי מאותחלת* המונה מספר סמפורים כפי שביקשנו.

עתה נאתחלם:

ב. <u>איתחול הסמפור</u>

לשם האיתחול נזדקק ל- union הבא *שנגדיר* בתכניתנו:

```
union semun {
  int val ;
  struct semid_ds *buf ;
  unsigned short *array ;
} ;
```

un = union

אנו נתעניין רק בחבר val ב- union.

נגדיר משתנה:

union semun <u>sem_val ;</u>

ונאתחלו לערך התחילי הרצוי של הסמפור:

sem_val.val = 1 ;

ועתה נאתחל את הסמפור #0 (היחיד) שיש לנו בקבוצה:

```
status = semctl(sem_set_id,
                0,
                SETVAL,
                sem_val) ;
if (status == -1) …
```

פעל על הקבוצה
על הסמפור #
בצע את הפעולה
תוך שימוש ב:

semctl() מאפשרת לנו לבצע מגוון פעולות על קבוצת הסמפורים (מחיקתה, בדיקת ערך של סמפור רצוי בקבוצה). בפעולות השונות נספק קוד פעולה שונה, ובמידת הצורך ערכים שונים ב- union.

ג. <u>קוד הכניסה (פעולת ה- wait)</u>

נגדיר משתנה:

struct sembuf sem_op ;

נאתחלו:

```
sem_op.sem_num = 0 ;
sem_op.sem_op = -1 ;
sem_op.flg = 0 ;
```

פעל על סמפור מספר:
הקטן את ערכו באחד
דגלים שלא יעניינו אותנו (למשל אם איננו רוצים להמתין)

ונבצע את הפעולה:

```
if (semop(sem_set_id,
          &sem_op,
          1) == -1) …
```

מזהה הקבוצה
מתאר הפעולה
מספר הסמפורים בקבוצה

עת חלפנו על-פני פעולה זאת בהצלחה אנו בקטע הקריטי.

פעולת היציאה סימטרית לגמרי לפעולת הכניסה רק בה נקבע:

sem_op.sem_op = 1 ;

ד. <u>דוגמה</u>

תכנית המבצעת את המשימה הבאה:

1. הגדר קבוצת סמפורים בת סמפור יחיד.
2. אתחלו לערך 1.
3. הולד חמישה ילדים.
4. כל ילד בלולאה מנסה להוסיף נתונים לקובץ משותף על כן:
   - א) ממתין על הסמפור.
   - ב) פותח את הקובץ.
   - ג) כותב עליו
   - ד) סוגרו
   - ה) משחרר הסמפור


```
// file: semaphore1_protect_file.c
// Adopted from Guy Keren (actcom)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>

#define NUM_PROCS  5
#define SEM_ID    3879
#define CHILD_PROC    0
#define FILE_NAME "sem_mutex.txt"
// ======structs etc. ======================
union semun {
   int val ;
   struct semid_ds *buf ;
   unsigned short *array ;
} ;
//=====PROTOTYPES ====================
void do_child_loop(int sem_set_id, char* file_name) ;
void update_file(int sem_set_id, char* file_path, int
number) ;
//=====================================
```



```
int main()
{
   int sem_set_id;
   union semun sem_val;
   int child_pid;
   int i;
   int rc;

   sem_set_id = semget(SEM_ID,
                       1,
                       IPC_CREAT | 0600);
   if (sem_set_id == -1) {
      perror("main: semget failed");
      exit(EXIT_FAILURE);
   }

   sem_val.val = 1;
   rc = semctl(sem_set_id,
               0,
               SETVAL,
               sem_val);
   if (rc == -1) {
      perror("main: semctl failed");
      exit(EXIT_FAILURE);
   }
```

יצירת קבוצת הסמפורים (ללא ftok)

איתחול הסמפור #0 בקבוצה לערך 1.


```
   for (i=0; i<NUM_PROCS; i++) {
      child_pid = fork();
      switch (child_pid) {
         case -1:
            perror("fork failed");
            exit(EXIT_FAILURE);
         case CHILD_PROC:
            do_child_loop(sem_set_id, FILE_NAME);
            exit(EXIT_SUCCESS);
         default:
            break;
      }
   }

   /* wait for all children to finish running */
   for (i=0; i<NUM_PROCS; i++)
      wait(NULL);

   puts("main: we're done\n");
   fflush(stdout);
   return EXIT_SUCCESS ;
}
```

```
//==========================================
// repeat:
// wait on the sem sem_set_id (on #0 in it)
// then open file_name, write on it, and close it

void do_child_loop(int sem_set_id, char* file_name)
{
   pid_t pid = getpid();
   int i;

   for (i=0; i<7; i++)
      update_file(sem_set_id, file_name, pid);
}
//==========================================
void update_file(int sem_set_id,
                 char* file_path,
                 int number)
{
   /* structure for semaphore operations.   */
   struct sembuf sem_op;
   FILE* file;

   // Entry section:
   /* wait on the semaphore,
      unless it's value is non-negative. */
   sem_op.sem_num = 0;          // num of sem in set
   sem_op.sem_op = -1;          // wait operation
   sem_op.sem_flg = 0;          // flags we leave 0
   semop(sem_set_id, &sem_op, 1);
```



```
   // The critical section
   file = fopen(file_path, "a");
   if (file) {
      setbuf(file, NULL) ;
      fputs("I ", file) ;
      fputs("am ", file) ;
      fputs("process ", file) ;
      fputs("#", file) ;
      fprintf(file, "%d\n", number);
      fclose(file);
   }

   // Exit section
   /* finally, signal the semaphore –
      increase its value by one. */
   sem_op.sem_num = 0;
   sem_op.sem_op = 1;          // signal operation
   sem_op.sem_flg = 0;
   semop(sem_set_id, &sem_op, 1);
}
```

כל פעולת כתיבה נשלחת מיידית לקובץ, ללא חציצה

כמה פעולות כתיבה אינן אטומיות


```
<205|0>yoramb@inferno-05:~/os$ less sem_mutex.txt
I am process #20087
I am process #20088
I am process #20089
I am process #20090
I am process #20091
I am process #20087
I am process #20088
I am process #20089
I am process #20090
I am process #20091
I am process #20087
I am process #20088
I am process #20089
I am process #20090
I am process #20091
I am process #20087
I am process #20088
I am process #20089
I am process #20090
I am process #20091
I am process #20087
I am process #20088
I am process #20089
I am process #20090
I am process #20091
I am process #20087
I am process #20088
I am process #20089
I am process #20090
I am process #20091
I am process #20087
I am process #20088
I am process #20089
I am process #20090
I am process #20091
<206|0>yoramb@inferno-05:~/os$
```

הפלט מסודר ויפה.

לשם השוואה נראה מה קורה ללא סמפורים (השימוש בסמפורים נסגר בהערת תיעוד):


```
<206|0>yoramb@inferno-05:~/os$ less
sem_mutex.txt.no_sem
I am process #20020
I am process #20021
I am process #I am process 20020
I I #I am I am 20022
am process #20021
process am process #I process #20023
I am #20024
am process I 20020
process #am #I 20022
process I 20021
am #I am process 20023
am process #I process #20020
am I #20024
process am I 20022
#process #20023
I am process #20021
am I 20020
process I am I am process #20021
#20024
am process process ##I 20023
20022
I am I am I process am process am #I process #process 20020
am #20021
#process 20024
20023
I am process #20021
#I I 20022
I am am process #am I process 20024
process am ##I process 20020
20023
am #process 20022
#20024
I am I am process process ##I 20022
20023
am process #20024
<207|0>yoramb@inferno-05:~/os$
```

7.9 סמפורים בממשק POSIX

הסמפורים כפי שראינו בסעיף הקודם מציעים ממשק מעט מסורבל לסמפורים. על כן POSIX קבעו אופנות שימוש שונה בסמפורים המתאימה לתכנית הכוללת כמה פתילים (אך לא למספר תהליכים):

א. במקום לקבל קבוצת סמפורים אנו מקבלים סמפור בודד.
ב. הסמפור הוא בהכרח בינארי.
ג. הוא מאותחל לערך 1 (=פתוח).
ד. הוא ממוחזר אוטומטית עת כל התהליכים שהשתמשו בו מסיימים.

הטיפוס המתאים: pthread_mutex_t.

נראה דוגמה:


```
//file:  pthread_mutex.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>

pthread_mutex_t mtx ;
void* my_func(void *) ;
//-------------------------------------------------------

int main() {
  pthread_t thread_data[2] ;
  int status,
     i ;

  srand((unsigned)time(NULL)) ;

  if (pthread_mutex_init(&mtx, NULL) != 0) {
    fputs("pthread_mutex_init() failed\n", stder
    exit(EXIT_FAILURE) ;
  }

  for (i=0; i < 2; i++) {
    status = pthread_create(thread_data+i,
                                  NULL, my_func, NULL) ;
    if (status != 0) {
          fputs("pthread_create failed in main", stderr) ;
          exit(EXIT_FAILURE) ;
    }
  }
  for (i= 0; i< 2; i++)
    status = pthread_join(thread_data[i], NULL) ;

  pthread_mutex_destroy(&mtx) ;
  return(EXIT_SUCCESS) ;
}
```

הסמפור הגלובלי ישמש את כל הפתילים

איתחול הסמפור. בהכרח לכדי פתוח. NULL = תכונות מחדליות.

הריסת הסמפור.


```
//-------------------------------------------------------
void* my_func(void * args)  {
  int i;

  setbuf(stdout, NULL) ;
  for (i= 0; i< 5; i++) {
    pthread_mutex_lock( &mtx) ;
    printf("I ") ;
    sleep(rand() %3) ;
    printf("am ") ;
    sleep(rand() %3) ;
    printf("thread ") ;
    sleep(rand() %3) ;
    printf("#") ;
    sleep(rand() %3) ;
    printf("%u\n",  (unsigned int) pthread_self()) ;
    pthread_mutex_unlock( &mtx) ;
    sleep(rand() %5) ;
  }

  pthread_exit(NULL) ;
}
//-------------------------------------------------------
/* a run with the mtx:
  ===================
<269|0>yoramb@inferno-05:~/os$ !a
a.out
I am thread #3086117776
I am thread #3075627920
I am thread #3086117776
I am thread #3086117776
I am thread #3075627920
I am thread #3075627920
I am thread #3086117776
I am thread #3075627920
I am thread #3086117776
I am thread #3075627920
```

נעילת הסמפור:

שחרור הסמפור:


```
<270|0>yoramb@inferno-05:~/os$

  a run without the mtx:
  ===================
<272|0>yoramb@inferno-05:~/os$ !a
a.out
I I am am thread #3075566480
I thread am #thread #3086056336
3075566480
I I am am thread thread ##3075566480
3086056336
I am thread #3086056336
I am I thread am thread ##3086056336
3075566480
I am thread #I am thread 3086056336
#3075566480
<273|0>yoramb@inferno-05:~/os$
*/
```

הערות:

א. אם הסמפור הינו סטטי (או גלובלי) ניתן לאתחלו גם באופן:

```
pthread_mutex_t    mutex =
                    PTHREAD_MUTEX_INITIALIZER;
```

ב. הפונ':

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

מנסה לנעול את הסמפור. מחזירה אפס אם מצליחה, ערך אחר (EBUSY) אם הסמפור כבר נעול.

ג. סמפור כפי שייצרנו עד כה, מחדלית, נחשב ל-'מהיר' או 'נורמלי'. לחילופין סמפור עשוי להיות 'רקורסיבי': ואז אם הוא ננעל n פעמים ע"י הפתיל *היחיד* שאוחז בו יש גם לשחררו n פעמים כדי שהוא יהיה פתוח; או: 'בטוח' ואז נבדק גם חשש לחסימה הדדית (ליתר דיוק: עצמית) בשל נעילה חוזרת ע"י הפתיל האוחז (בסמפור מהיר נעילה חוזרת תביא לח.ה..)

ד. ד

7.10 משתני תנאי Condition Variables

משתני תנאי (מ.ת.) הם כלי עזר נוסף ש-POSIX מעמידה לרשותנו לשם סינכרון *פתילים*. מ.ת. יתווספו לפתילים (ולא יחליפו אותם) כדי להשיג תיאום בין פתילים.

מטרתם: לאפשר לפתיל(ים) להמתין בלי לצרוך זמן מעבד, עד שיחול אירוע לו הפתיל(ים) אמור(ים) להגיב.

מוטיבציה: נניח שתכניתנו כוללת אוסף של פתילים אשר אמורים לטפל בבקשות המוספות למבנה נתונים כלשהו (לדוגמה: בקשות הדפסה). באופן בסיסי הפתילים ממתינים, ישנים; עת נוספת בקשה\איבר למבנה הנתונים אחד הפתילים מתעורר, מטפל בבקשה (חלק מהטפול מתבצע בקטע קריטי בו הפתיל, ורק הוא, מסיר את האיבר ממבנה הנתונים), וחוזר לישון.

על כן מ.ת. מספק את הפעולות:

א. המתנה עליו (תוך שינה).
ב. הערת אחד הממתינים = שליחת סיגנל (על מ.ת., לא סיגנל 'רגיל' של מ.ה.)
ג. הערת כלל הממתינים.

הערות:

א. כאשר נשלחת הודעת הערה ואין אף ממתין (ישן) ההודעה אובדת (כאילו היא לא בוצעה). אם *אח"כ* פתיל כלשהו יבצע פעולת המתנה אזי הוא לא יוער מיידית—הוא יוער רק אם וכאשר תשלח הודעת הערה *אחרי* לכתו לשון.

ב. משתנה תנאי אינו מבטיח בלבדיות, ועל כן נשתמש בו בשילוב עם סמפור. משתנה התנאי יאפשר להמתין על הסמפור המתנה שאינה עסוקה. עת הפתיל הממתין מוער (למשל אחרי הגעת בקשה להדפסה), ההערה גם נועלת את הסמפור.

ג. עת מספר תהליכים ממתינים על משתנה התנאי, ונשלחת הודעת הערה (סיגנל), יוער *אקראית* (מבחינתנו) אחד מהם, בלי התחייבות איזה מהממתינים מוער.

ד. ד

אופן השימוש במשתנה

א. הגדרה ואיתחול:

```
pthread_cond_t cv =
        PTHREAD_COND_INITIALIZER ;
```

איתחול זה (באמצעות מאקרו)ניתן לבצע רק בהגדרה, אחרת
נשתמש בפונ': pthread_con_init()

מכיוון שנזדקק גם לסמפור נגדיר ונאתחל גם אותו:

```
pthread_mutex_t mutex =
        PTHREAD_MUTEX_INITIALIZER;
```

ב. המתנה:
פעולת ההמתנה מבצעת שני דברים באופן אטומי: ראשית משחררת את הסמפור (כדי לתת לאחרים אפשרות גישה למבנה הנתונים), ושנית שולחת את הפתיל לשון (עד הוספת איבר למבנה הנתונים, והערתו, תוך שבמקביל הסמפור ננעל).

לכן, ראשית, הסמפור ננעל:
rc = pthread_mutex_lock(&mutex) ;
if (rc) …
ושנית, נפתח, תוך שליחת הפתיל לשון:
rc = pthread_cond_wait(&cv, &mutex) ;

עת הפתיל מגיע לנקודה זאת הוא יכול לבצע את הק.ק. שלו: שכן ההערה מההמתנה גם נועלת את הסמפור.

ג. הערת אחד הממתינים = שליחת סיגנל:
עד כאן ראינו כיצד פתיל ממתין על מ.ת.. עתה נראה כיצד מעירים פתיל (למשל, אחרי הוספת איבר למבנה הנתונים הרלוונטי), המוסיף יבצע את הקוד הבא:
rc = pthread_mutex_lock(&mutex) ;
if (rc) …
הוסף איבר למבנה הנתונים, בק.ק.
rc = pthread_mutex_unlock(&mutex) ;
if (rc) …

rc = pthread_cond_signal(&cv) ;
if (rc) …

הערת כל הממתינים:
rc = pthread_cond_broadcast(&cv) ;

ד. שחרור משתנה התנאי:
rc = pthread_cond_destroy(&cv) ;
if (rc) …

הערה:
ניתן דעתנו לבעיה הבאה:
א. n פתילים ממתינים על מ.ת.
ב. n איברים מוסִפים למ.נ., נשלחים n סיגנלים, כל n הפתילים מעוררים ומטפלים בבקשות.
ג. במקביל: נוסף איבר חדש למ.נ. ונשלח סיגנל *שאובד* (אין אף פתיל ישן).
ד. n הפתילים מסיימים לטפל ב- n הבקשות וחוזרים לְשון שנת ישרים (בלי לתת את דעתם לבקשה שנוספה)

מסקנה: צרה!

פתרון אפשרי: מונה בקשות ממתינות. כל פתיל לפני שחוזר לשון יבדוק את ערכו, ורק אם ערכו אפס הפתיל יחזור לשון. (כמובן שהטיפול במונה יעשה בק.ק..)

דוגמה:
התכנית שנציג תנהל רשימה מקושרת ('רגילה') של 'בקשות'. כל איבר ברשימה יכיל מספר שלם.

הפתיל הראשי (בפונ' עזר add_request) יוסיף איברים לרשימה.

שלושה פתילי משנה שהתכנית תייצר (ושמריצים את הפונ': handle_requests_loop) יסירו ו- 'יטפלו' באברי הרשימה.

פתילי המשנה ימתינו על משתנה תנאי, על הטיפול ברשימה יגן סמפור.

סיום התכנית: הפתיל הראשי מוסיף שלושה אפסים למ.נ.. כ"א מהפתילים עת קורא ערך זה מסיים.

```
// file: thread_pool_server_mutex_cv2.c
// Compile: cc -Wall –lpthread
//                    thread_pool_server_mutex_cv2.c

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>     // for sleep()

#define NUM_HANDLER_THREADS     3

//-------------------- struct  ------------------------------------------
struct request {
   int number;
   struct request* next;
};

//-------------------- global vars ---------------------------------------
struct request* requests_head = NULL;
struct request* requests_tail = NULL;

int requests_counter = 0;                // size of linked list

pthread_mutex_t request_mutex =
            PTHREAD_MUTEX_INITIALIZER; ;
pthread_cond_t  got_request_cv   =
            PTHREAD_COND_INITIALIZER;

//-------------------- prototypes ----------------------------------------
void add_request(int request_num,
         pthread_mutex_t* p_mutex,
         pthread_cond_t*  p_cond_var) ;

void* handle_requests_loop(void* data) ;
```

המבנה עבור הרשימה המקושרת

מצביעים לראש וזנב הרשימה (גלובליים לטובת כלל הפתילים)

הסמפור ומ.ת. (מאותחלים)


```
struct request* get_request(pthread_mutex_t* p_mutex) ;

void handle_request(struct request* a_request,
                         int thread_id) ;
//-----------------------------------------------------------------------
int main(int argc, char* argv[])
{
   int       i;
   int       rc;
                                   // id of thread = 0, 1, 2,...
   int       thread_id[NUM_HANDLER_THREADS];
   pthread_t  p_threads[NUM_HANDLER_THREADS];

   srand(time(NULL));

   for (i = 0; i < NUM_HANDLER_THREADS; i++) {
      thread_id[i] = i;
      rc = pthread_create(&p_threads[i], NULL,
                              handle_requests_loop,
                              (void*)&thread_id[i]);
      if (rc != 0) {
        fputs("pthread_create failed in main", stderr) ;
        exit(EXIT_FAILURE) ;
      }
   }

   for (i = 1; i <= 21; i++) {
      add_request(i, &request_mutex,
                        &got_request_cv);
      sleep(rand()%3) ;
   }
```

יצור הפתילים. כ"א מקבל את מזההו = מספרו

הפתיל הראשי מוסיף 'הודעות' לרשימה


```
   // sign threads to terminate
   for (i=0; i < NUM_HANDLER_THREADS; i++)
      add_request(0, &request_mutex, &got_request_cv);

   for (i=0; i < NUM_HANDLER_THREADS; i++)
       pthread_join( p_threads[i], NULL) ;

   pthread_cond_destroy(&got_request_cv) ;

   puts("main thread exits") ;

   return EXIT_SUCCESS;
}
```

הוספת 0 ==> הנחיה לפתילים להסתיים, והמתנה להם


```
//-----------------------------------------------------------------------
void add_request(int request_num,
         pthread_mutex_t* p_mutex,
         pthread_cond_t*  p_cond_var)
{
   int rc;
   struct request* a_request;

   a_request = (struct request*)
                  malloc(sizeof(struct request));
   if (!a_request) {
      fputs("add_request: out of memory\n", stderr);
      exit(EXIT_FAILURE);
   }
   a_request->number = request_num;
   a_request->next = NULL;

   rc = pthread_mutex_lock(p_mutex);

   if (requests_counter == 0) {
      requests_head = a_request;
      requests_tail = a_request;
   }
   else {
      requests_tail->next = a_request;
      requests_tail = a_request;
   }
   requests_counter++;

   rc = pthread_mutex_unlock(p_mutex);

   rc = pthread_cond_signal(p_cond_var);
}
```

הוספת הודעה בודדת לתור (ע"י הפתיל הראשי. פונ' זו נקראת בלולאה)

הקצאת איבר נוסף, והזנתו בנתונים הדרושים= קטע שיורי

קוד הכניסה לק.ק. = נעילת הסמפור

הקטע הקריטי

שחרור הסמפור והערת אחד הממתינים

```
//-----------------------------------------------------------------------
void* handle_requests_loop(void* data)
{
   int rc;
   struct request* a_request;
   int thread_id = *((int*)data);


   rc = pthread_mutex_lock(&request_mutex);

   while (1) {
      if (requests_counter > 0) {
         a_request = get_request(&request_mutex);
         if (a_request) {
            rc = pthread_mutex_unlock(&request_mutex);


            handle_request(a_request, thread_id);
            free(a_request);

            rc = pthread_mutex_lock(&request_mutex);
         }
      }

      else {
         rc = pthread_cond_wait(&got_request_cv,
                                &request_mutex);
      }
   }
}
```

הפונ' הראשית של הפתילים (בלולאה מסירה הודעות מהרשימה).

קוד הכניסה לק.ק

הק.ק.

קוד היציאה מהק.ק

הקטע השיורי. בינתיים היצרן יוסיף

קוד הכניסה לק.ק (לקראת סיבוב נוסף).

אם הרשימה ריקה, לך לשון, (תוך שחרור הסמפור)


```
//-----------------------------------------------------------------------
struct request* get_request(pthread_mutex_t* p_mutex)
{
   struct request* a_request;

   if (requests_counter > 0) {
      a_request = requests_head;
      requests_head = a_request->next;
      if (requests_head == NULL) {
         requests_tail = NULL;
      }
      requests_counter--;
   }
   else {
      a_request = NULL;
   }
   return a_request;
}
//-----------------------------------------------------------------------
void    handle_request(struct    request*    a_request,    int
thread_id)
{
   if (a_request) {
      printf("Thread '%d' handled request '%d'\n",
            thread_id, a_request->number);
      fflush(stdout);

      if (a_request->number == 0) {
         printf("Thead %d exits\n",  thread_id) ;
         pthread_exit(NULL) ;
      }
   }
}
//-----------------------------------------------------------------------
```

הק.ק. של כל פתיל (בין נעילת הסמפור לשחרורו): הסרת איבר מהרשימה. (הקוד עצמו מאוד סטנדרטי)

הקוד השיורי של כל פתיל: טיפול באיבר שכבר הוסר מהרשימה

אם האיבר מכיל אפס, אזי הפתיל מסיים


```
/* a run:
   ======
<243|1>yoramb@inferno-05:~/os$ !g
gcc -Wall -lpthread thread_pool_server_mutex_cv2.c
<244|0>yoramb@inferno-05:~/os$ !a
a.out
Thread '0' handled request '1'
Thread '0' handled request '2'
Thread '0' handled request '3'
Thread '1' handled request '4'
Thread '2' handled request '5'
Thread '0' handled request '6'
Thread '1' handled request '7'
Thread '2' handled request '8'
Thread '0' handled request '9'
Thread '1' handled request '10'
Thread '2' handled request '11'
Thread '0' handled request '12'
Thread '1' handled request '13'
Thread '2' handled request '14'
Thread '0' handled request '15'
Thread '1' handled request '16'
Thread '2' handled request '17'
Thread '1' handled request '19'
Thread '0' handled request '18'
Thread '2' handled request '20'
Thread '1' handled request '21'
Thread '1' handled request '0'
Thead 1 exits
Thread '0' handled request '0'
Thead 0 exits
Thread '2' handled request '0'
Thead 2 exits
main thread exits
<245|0>yoramb@inferno-05:~/os$
*/
```


7.11 מנעולי קריאה\כתיבה Read/Write Locks

סמפור עשוי להיות פתוח או נעול, ורק תהליך\פתיל יחיד (או בסמפור מונה n תהליכים\פתילים) יכולים לנעלו בכל נקודת זמן.

עת התהליכים\פתילים מעוניינים לעדכן מבנה נתונים העיקרון הנ"ל מתאים.

במצב בו קיימים שוני סוגי תהליכים:

א. כאלה המעוניינים לעדכן את מ.נ. (באופן בלבדי).

ב. כאלה המעוניינים לקרוא (לשלוף) נתונים ממ.נ. (לצד אחרים שעושים זאת, אך עת לא קיים תהליך הכותב על מ.נ.).

נרצה כלי אחר.

הפתרון: מנעולים.

מנעול עשוי להינעל באופן בלבדי (exclusive) ע"י כותב יחיד, או באופן משותף (shared) במקביל, ע"י קוראים רבים.

מנעול שנרכש ישוחרר בכל מקרה ע"י פעולת פתיחה\שחרור (unlock).

עת תהליך\פתיל מבקש מנעול, ולא ניתן לספק לו אותו (התהליך ביקש מנעול בלבדי וכבר קיימים קוראים, או ביקש מנעול משותף וכבר קיים כותב) הוא נחסם (נתקע), ואינו יכול לעשות פעולה אחרת.

שאלה רלוונטית אחרת היא הבאה: נניח שמנעול l נרכש ע"י קורא(ים). נניח שעתה מגיע כותב, ולכן נחסם. נניח שאחרי הכתב מגיעים קוראים נוספים. האם הם יורשו להצטרף לקוראים הקיימים, או שהם יחסמו?

<u>התשובה המקובלת</u>: הם יחסמו כדי למנוע הרעבה של הכותב, או כדי למנוע את השארת מ.נ. במצב לא מעודכן.

תהליך\פתיל עשוי גם רק <u>לנסות</u> לנעול את המנעול (באופנות רצויה). במידה והניסיון מצליח המנעול נרכש על-ידו, אחרת (המנעול כבר נעול באופנות *מתנגשת* ע"י תהליך אחר): יוחזר לו כישלון.

נציג מימוש של מנעולי POSIX לסינכרון בין פתילים (אך לא בין תהליכים).

המשתנה הדרוש:

```
pthread_rwlock_t lock ;
```

<u>איתחולו:</u>

```
if (pthread_rwlock_init(&lock, NULL) == -1)
  …
```

NULL בארגומנט השני == תכונות מחדליות (התכונה האפשרית היחידה, כמו בסמפורים: שימוש במשתנה גם בין תהליכים. לא נדון בה).

<u>נעילה ושחרור:</u>

```
pthread_rwlock_wrlock(&lock);

if (pthread_rwlock_rdlock(&lock) != 0)
  failure
```

כישלון בנעילה: עשוי לנבוע מכך שיש מגבלה על כמות הפתילים שרשאים לאחוז במנעול משותף במקביל

```
pthread_rwlock_unlock(&lock);
```

<u>ניסיון נעילה:</u>

```
if (pthread_rwlock_tryrdlock(&lock) != 0)
   failure

if (pthread_rwlock_trywrlock(&lock) != 0)
  failure
```


<u>הריסה (בסיום השימוש):</u>

```
pthread_rwlock_destroy(&lock) ;
```

<u>נראה דוגמה:</u>

בתכנית נחזיק struct שיעודכן ע"י שני פתילים, ויקרא (בלבד) ע"י שני פתילים אחרים.
במבנה יהיו שני חברים: a, b. הפתילים המעדכנים יעבירו ערך אקראי n (חיובי או שלילי) מ- a, ל- b (שמאותחלים לאפס).
המעדכנים ירכשו, כמובן, מנעול בלבדי על 'מבנה הנתונים', הקוראים יסתפקו במנעול משותף.


```
// file: pthread_rw_locks.c
/* A program that uses locks.
   The program maintains 'a data structure': struct ab.
   Two threads update it, other two read it.
   The writers need exclusive access to the dast,
   the readers need a shared access.
   All lock the variable before using it.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>

#define N      4          // num of threads

struct {
   int a, b ;
} ab = {0, 0 } ;
```

מ.נ., עליו נגן באמצעות המנעול. מאותחל ל: 0,0

```
pthread_rwlock_t lock ;
```

המנעול:

```
void* my_func(void *) ;
//-----------------------------------------------------
int main() {
   pthread_t thread_data[N] ;
   int arr[] = {0, 1} ;       // is thread reader or writer
   int status,
       i ;

   srand((unsigned)time(NULL)) ;

   if (pthread_rwlock_init(&lock, NULL) != 0) {
      puts("pthread_rwlock_init(() failed\n") ;
      exit(EXIT_FAILURE) ;
   }
```

איתחול המנעול:

```
  for (i=0; i < N; i++) {
    status = pthread_create(thread_data+i, NULL,
                            my_func,
                            (void *) (arr+ i%2)) ;
    if (status != 0) {
          puts("pthread_create failed in main") ;
          exit(EXIT_FAILURE) ;
    }
  }

  for (i= 0; i< N; i++)
    status = pthread_join(thread_data[i], NULL) ;

  pthread_rwlock_destroy(&lock) ;

  return(EXIT_SUCCESS) ;
}
```

כל פתיל מקבל את 'תפקידו': ק' או כ'

הרס המנעול לפני סיום


```
//-------------------------------------------------------
void* my_func(void * args)  {
  int i, n,
      me =  *((int *) args) ;

  for (i= 0; i< 5; i++) {
    if (me  == 0) {                    // a reader thread
      pthread_rwlock_rdlock( &lock) ;
      printf("(%d %d)\n", ab.a, ab.b ) ;
      pthread_rwlock_unlock( &lock) ;

      sleep(rand() %10) ;
    }
    else {                             // a writer thread
      n= rand() % 10 -5 ;    // move n from ab.a to ab.b
      pthread_rwlock_wrlock( &lock) ;
      ab.a -= n ;
      sleep(rand() %5) ;
      ab.b += n ;
      pthread_rwlock_unlock( &lock) ;
      sleep(rand() %5) ;
    }
  }
  pthread_exit(NULL) ;
}
//-------------------------------------------------------
```

האם אני ק' (=0), או כ' (=1):

נעילה במנעול משותף, קריאת 'מ.נ.', שחרור המנעול

נעילה במנעול בלבדי, עדכון 'מ.נ.', שחרור המנעול (בין לבין שינה שלכאורה נותנת לק' הזדמנות לק' את מ.נ. במצב 'לא עקבי').


```
/* a run with the lock:
   ===================
<216|0>yoramb@inferno-05:~/os$
                gcc -Wall pthread_rw_locks.c -lpthread
<217|0>yoramb@inferno-05:~/os$ a.out
(0 0)
(-9 9)
(-9 9)
(-14 14)
(-13 13)
(-13 13)
(-13 13)
(-7 7)
(-9 9)
(-9 9)
a run without the lock:
=======================
<224|0>yoramb@inferno-05:~/os$ a.out
(0 0)
(0 0)
(0 0)
(0 0)
(5 0)
(4 -5)
(7 -11)
(7 -11)
(14 -12)
(14 -14)

*/
```

בהרצה עם מנעולים תמיד a+b = 0

בהרצה בה לא נעשה שימוש במנעולים. לא תמיד a+b = 0: הק' ניגשו למ.נ. עת הוא היה במצב ביניים, לא עקבי.