Concurrent Programming Problems

OPERATING SYSTEMS COURSE

THE HEBREW UNIVERSITY

SPRING 2023

Outline

- Concurrency
  - Classical problems of synchronization

Classical Problems of Synchronization

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Dining-Philosophers Problem
- Readers and Writers Problem

Bounded-Buffer Problem

- One cyclic buffer that can hold up to N items
- **Producers** and **consumers** use the buffer

- The problem:
  - Make sure that producer can't add data if the buffer is full
  - Make sure that consumer can't consume data if the buffer is empty.

- The buffer is shared, so protection is required.
- We use **counting semaphores**:
  - the number in the semaphore represents the number of resources of some type

Bounded-Buffer Problem

- Semaphore **mutex** initialized to the value 1
  - Protects the buffer (one access at a time)
- Semaphore **fillCount** initialized to the value 0
  - Indicates how many items in the buffer are available to be read.
- Semaphore **emptyCount** initialized to the value N
  - Indicates how many items in the buffer are available to be write.

Bounded-Buffer Problem – Cont.

Producer:

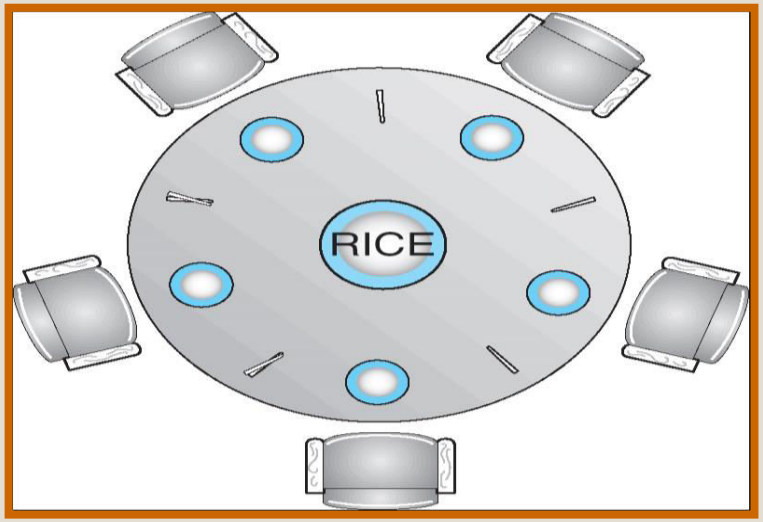
```
while (true)  {
    produce an item
    down (emptyCount);
      down (mutex);
        add the item to the
        buffer
      up (mutex);
    up (fillCount);
}
```

Consumer:

```
while (true) {
   down (fillCount);
      down (mutex);
         remove an item from
         buffer
      up (mutex);
   up (emptyCount);
   consume the item
}
```

Dining-Philosophers Problem

- **The problem:**
  - Five philosophers sit in a round table with a rice bowl in the center.
  - Between each pair there is single chopstick.
  - They only eat or think.
  - When a philosopher has two chopsticks he can start eating.



Dining-Philosophers Problem – Cont.

Shared data:

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

```
While (true)  {
down ( chopstick[i] );
down ( chopstick[ (i + 1) % 5] );
eat
up (chopstick[ (i + 1) % 5] );
up ( chopstick[i] );
think
}
```

Dining-Philosophers Problem – Cont.

Shared data:

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

```
While (true)  {
down ( chopstick[i] );
down ( chopstick[ (i + 1) % 5] );
eat
up (chopstick[ (i + 1) % 5] );
up ( chopstick[i] );
think
}
```

This may cause **deadlocks**.

Dining Philosophers Problem

- This abstract problem demonstrates some fundamental limitations of deadlock-free synchronization.

- There is no symmetric solution:
  - Any symmetric algorithm leads to either starvation or deadlock

Possible Solutions

- Use a waiter (as an arbiter).
- Asymmetric solutions:
  - Execute different code for odd/even.
  - Randomized (Lehmann-Rabin).

Lehmann-Rabin Algorithm


```
repeat
if coinflip() == 0 then              // randomly decide on a first chopstick
        first = left
else
        first = right
end if
wait until chopstick[first] == false
chopstick[first] = true              // wait until it is available
if chopstick[~first] == false then   // if second chopstick is available
        chopstick[~first] = true     // take it
        break
else
        chopstick[first] = false     // otherwise drop first chopstick
end if
end repeat
eat
chopstick[left] = false
chopstick[right] = false
```

Readers-Writers Problem

- A data structure is shared among a number of concurrent processes:
  - Readers – Only read the data; They do not perform updates.
  - Writers – Can both read and write.

- The problem:
  - Allow multiple readers to read at the same time.
  - Only one writer can access the shared data at the same time.
  - While a writer is writing to the data structure, reader should be blocked from reading.

Readers-Writers Problem: First Solution

- Shared Data:
  - Integer **readCount** initialized to 0.
    - Number of readers
  - Semaphore **readCount_mutex** initialized to 1.
    - Protects readCount
  - Semaphore **write** initialized to 1.
    - Makes sure the writer doesn't use data at the same time as any readers

Readers-Writers Problem: First solution

The structure of a writer thread

```
while (true) {
  down (write) ;
        writing is performed
  up (write) ;
}
```

Readers-Writers Problem: First solution

The structure of a reader thread

```
while (true) {
   down (readCount_mutex );
      readCount ++;
      if (readCount == 1)
         down (write); // lock from writers
   up (readCount_mutex )

   reading is performed // CS

   down (readCount_mutex );
      readCount - -;
      if (readCount  == 0)
         up (write);
   up (readCount_mutex );
}
```

Readers-Writers Problem: First solution

The structure of a reader thread

```
while (true) {
   down (readCount_mutex );
     readCount ++;
     if (readCount == 1)
        down (write); // lock from writers
   up (readCount_mutex )

   reading is performed // CS

   down (readCount_mutex );
     readCount - -;
     if (readCount  == 0)
        up (write);
   up (readCount_mutex );
}
```

This solution is not perfect:

What if a writer is waiting to write but there are readers that read all the time?

Writers are subject to starvation!

Second Solution: Writer Priority

- Extra semaphores and variables:
  - Semaphore **read** initialized to 1
    - Inhibits readers when a writer wants to write.
  - Integer **writeCount** initialized to 0
    - Counts waiting writers.
  - Semaphore **writeCount_mutex** initialized to 1
    - Controls the updating of writecount.
  - **Queue** semaphore used only in the reader (initialized to 1).

Second solution: Writer Priority

The writer:

```
while (true) {
  down (writeCount_mutex )
    writeCount++;  //counts number of waiting writers
    if (writeCount ==1)
      down (read)
  up(writeCount_mutex)

  down (write) ;
    // writing is performed – one writer at a time
  up (write) ;

  down (writeCount_mutex)
    writeCount--;
    if (writeCount ==0)
      up (read)
  up (writeCount_mutex)
}
```

Second Solution: Writer Priority (cont.)

Queue semaphore, initialized to 1:

Since we don't want to allow more than one reader at a time in this section (otherwise the writer will be blocked by multiple readers when doing down (read).)

The reader:

```
while (true) {
  down (queue)
    down (read)
      down (readCount_mutex) ;
        readCount ++ ;
        if (readCount == 1)
          down (write) ;
      up (readCount_mutex)
    up (read)
  up (queue)

  reading is performed

  down (readCount_mutex) ;
    readCount - - ;
    if (readCount  == 0)
      up (write) ;
  up (readCount_mutex) ;
}
```

שאלה ממבחן)מועד ב(2012 '

3. כאשר מספר תהליכים חולקים גישה למבני נתונים משותפים, עלולות לצוץ בעיות.

a) להלן פתרון לבעיית הקטע הקריטי:

```
shared boolean flag[2] = {false};
shared int turn = 0;
do
{
        flag[i] = true;
        turn = i;
        while (flag[1-i]  &&  turn==1-i);
        critical section
        flag[i] = false;
        remainder section
} while (true);
{}
```

מה דעתך על המימוש הזה?

א) הוא מבטיח את תכונת המניעה ההדדית במערכת עם שני תהליכים.

ב) הוא לא מבטיח פתרון בעיית הקטע הקריטי במערכת עם שני תהליכים.

ג) אפשר לפתור את בעיית הקטע הקריטי לשני תהליכים בהוספת תווים בודדים (יש להוסיף אותם אם לדעתך אפשר). (2 נק')

ד) אפשר לפתור את בעיית הקטע הקריטי לכל מספר של תהליכים בשנוי פקודה אחת בתכנית. (2 נק')

Taken from: http://www4.huji.ac.il/exams/67808_2012_2_2_1.pdf

שאלה ממבחן)מועד ב(2012 '

3. כאשר מספר תהליכים חולקים גישה למבני נתונים משותפים, עלולות לצוץ בעיות.

a) להלן פתרון לבעיית הקטע הקריטי:

```
shared boolean flag[2] = {false};
shared int turn = 0;
do
{
        flag[i] = true;
        turn = i;
        while (flag[1-i]  &&  turn==1-i);
        critical section
        flag[i] = false;
        remainder section
} while (true);
{}
```

מה דעתך על המימוש הזה?

א) הוא מבטיח את תכונת המניעה ההדדית במערכת עם שני תהליכים.

ב) הוא לא מבטיח פתרון בעיית הקטע הקריטי במערכת עם שני תהליכים.

ג) אפשר לפתור את בעיית הקטע הקריטי לשני תהליכים בהוספת תווים בודדים (יש להוסיף אותם אם לדעתך אפשר). (2 נק')

ד) אפשר לפתור את בעיית הקטע הקריטי לכל מספר של תהליכים בשנוי פקודה אחת בתכנית. (2 נק')

Taken from: http://www4.huji.ac.il/exams/67808_2012_2_2_1.pdf

שאלה ממבחן)מועד ב(2012 '

3. כאשר מספר תהליכים חולקים גישה למבני נתונים משותפים, עלולות לצוץ בעיות.

a) להלן פתרון לבעיית הקטע הקריטי:

```
shared boolean flag[2] = {false};
shared int turn = 0;
do
{
        flag[i] = true;
        turn = i;
        while (flag[1-i]  &&  turn==1-i);
        critical section
        flag[i] = false;
        remainder section
} while (true);
{}
```

מה דעתך על המימוש הזה?

א) הוא מבטיח את תכונת המניעה ההדדית במערכת עם שני תהליכים.

ב) הוא לא מבטיח פתרון בעיית הקטע הקריטי במערכת עם שני תהליכים.

ג) אפשר לפתור את בעיית הקטע הקריטי לשני תהליכים בהוספת תווים בודדים (יש להוסיף אותם אם לדעתך אפשר). (2 נק')

ד) אפשר לפתור את בעיית הקטע הקריטי לכל מספר של תהליכים בשנוי פקודה אחת בתכנית. (2 נק')

Taken from: http://www4.huji.ac.il/exams/67808_2012_2_2_1.pdf

מבחן 2013 – שאלה נוספת

b) להלן פתרון לבעיית קוראים כותבים. מה דעתך על המימוש הזה?

```
semaphore wrt_lock = 1;
int rd_count = 0;

WRITER:
down(wrt_lock);
do_write();
up(wrt_lock);
```

```
READER:
rd_count++;
if (rd_count==1)
    down (wrt_lock);
do_read();
rd_count--;
if (rd_count==0)
    up (wrt_lock);
```

(3 נק') א) הוא מבטיח את תכונת המניעה ההדדית בין קבוצת קוראים לקבוצת כותבים.

(2 נק') ב) הוא מבטיח את תכונת המניעה ההדדית בין הכותבים.

Taken from: http://www4.huji.ac.il/exams/67808_2013_2_1_1.pdf

Question 3 - Moed A 2019

- נניח שמערכת הפעלה תומכת בעד N=1000 תהליכים. כאשר לכל תהליך מזהה ייחודי המסומן בערך id שהוא מספר שלם גדול מ-0. בשאלה זו נתעסק ב-entry code של בעיית המניעה ההדדית ונניח כי ה-exit code עובד כנדרש (כאשר תהליך יוצא מהקטע הקריטי הוא משחרר את הנעילה ומאפשר לתהליכים אחרים להיכנס לקטע הקוד הקריטי).
- בשאלה זו אנחנו נתייחס לשתיים מהתכונות אותן נרצה להבטיח בעת שימוש במשאב משותף: מניעה הדדית (mutual exclusion) והתקדמות (progress).
- **שימו לב:** בסעיפים א-ג תתבקשו לכתוב דוגמאות נגדיות **מפורטות** בהן נדרש להדגים על תהליכים עם מזהים ספציפיים, להסביר על אופן הרצתם ומיקומם בקוד (ביחס למספרי השורות).
- **מומלץ** לקרוא את כל סעיפי השאלה לפני שמתחילים לענות עליה.

Question 3 - Moed A 2019

א. (5 נק') עבור קטע הקוד הבא (עבור תהליך עם מזהה id, כל תהליך יריץ את הקוד עם ערך ה- id שלו). ציינו איזו מהתכונות הנ"ל לא ניתן להבטיח, והראו דוגמת הרצה נגדית בה תכונה זו אינה מתקיימת. המשתנים door ו-race הם משתנים גלובליים. door מאותחל לערך true ו-race מאותחל לערך 1-.

Question 3 - Moed A 2019

א. (5 נק') עבור קטע הקוד הבא (עבור תהליך עם מזהה id, כל תהליך יריץ את הקוד עם ערך ה- id שלו). ציינו איזו מהתכונות הנ"ל לא ניתן להבטיח, והראו דוגמת הרצה נגדית בה תכונה זו אינה מתקיימת. המשתנים door ו-race הם משתנים גלובליים. door מאותחל לערך true ו-race מאותחל לערך 1-.

```
while (true)  {
    race = id;
    if door == false
            continue;
   else {
        door = false;
        break;
    }
  }
<critical section>
```

Question 3 - Moed A 2019

ב. (5 נק') הוצע השינוי הבא בקטע הקוד. ציינו איזו מהתכונות הנ"ל לא ניתן להבטיח, והראו דוגמת הרצה נגדית בה תכונה זו אינה מתקיימת.

```
while (true)  {
  race = id;
  if door == false
          continue;
  else   {
      door = false;
      if race == id
          break;
      else
          continue;
    }
  }
<critical section>
```

Question 3 - Moed A 2019

ג. (5 נק') הוצע שינוי נוסף בקוד. ציינו איזו מהתכונות הנ"ל לא ניתן להבטיח, והראו דוגמת הרצה נגדית בה תכונה זו אינה מתקיימת.

המשתנה win הוא משתנה לוקאלי!

```
1. race = id;
2. if door == false
3.     win=false;
4. else {
5.     door = false;
6.     if race == id
7.         win=true;
8.     else
9.         win=false;
   }

10. if (win==false) {
11.     Run entry code for the bakery algorithm (as we learned in class)
   }

<critical section>
```

Question 3 - Moed A 2019

ד. (7 נק') הוצע שינוי נוסף בקוד. האם אלגוריתם זה מבטיח את 2 התכונות הרצויות בעת גישה למשאב משותף? (הסבירו בקצרה למה לאחר שינוי זה האלגוריתם מקיים את תכונות בעיית המניעה ההדדית).

Question 3 - Moed A 2019


```
1. race = id;
2. if door == false
3.     win=false;
4. else {
5.     door = false;
6.     if race == id
7.         win=true;
8.     else
9.         win=false;
   }

10. if (win==false) {
11.         Run entry code for the bakery algorithm (as we learned in class)
12.         xid = 0;
    }
13. else {
14.         xid = 1;
    }
15. Run entry code for Peterson's algorithm (as we learned in class)
with
    xid value as the process id

<critical section>
```

המשתנה xid הוא משתנה לוקאלי!

Question 3 - Moed A 2019

ה. (4 נק') הסבירו באיזה מקרים עדיף להשתמש באלגוריתם שהוצג בסעיף ד' ולא ב-bakery algorithm שהוצג בכיתה.

Question 3 - Moed A 2019

ה. (4 נק') הסבירו באיזה מקרים עדיף להשתמש באלגוריתם שהוצג בסעיף ד' ולא ב-bakery algorithm שהוצג בכיתה.

שאלה 4 מועד א 2020

בשל מגבלות הקורונה, לצורך עמידה בתו הסגול, מבוצעת מכירה רק ב-take away וכן יכולים רק 20 אנשים להמתין במסעדה. נתון כי למסעדה שף אחד.

השף משרת את הלקוחות אחד אחרי השני, כך שבכל פעם הוא מכין מנה עבור לקוח אחד בדיוק. כשהלקוח מגיע למסעדה והשף עסוק בהכנת מנה אחרת, הוא ימתין אם במסעדה לכל היותר 19 לקוחות ממתינים (לא כולל הוא עצמו). אם יש כבר 20 לקוחות ממתינים, הוא יעזוב את המסעדה ולא יקבל מנה.

פעילות המסעדה ממומשת באופן הבא: השף ממומש ע"י תהליך יחיד שמריץ את הפונקציה chef וכל לקוח ממומש ע"י תהליך, כך שכל אחד מריץ את הפונקציה customer. כאשר השף מסיים להכין מנה של לקוח (דהיינו מסיים את הפונקציה prepareMeal()), הלקוח לוקח את המנה (הפונקציה getMeal()) והשף בוחר לקוח ממתין כדי לטפל בהזמנתו. הפונקציות prepareMeal ו-getMeal ממומשות ע"י כתיבה למקום מוסכם בזיכרון, כאשר נתון שהפונקציה getMeal מסתיימת אך ורק לאחר שהקריאה המתאימה ל-prepareMeal הסתיימה.

שאלה 4 מועד א 2020

נבחן את הפתרון בהיבטים הבאים:

1. מניעה הדדית: רק לקוח אחד מקבל מנה (מריץ את הקוד של getMeal) בו זמנית.
2. התקדמות: אם יש לקוח שממתין למנה, אז לקוח כלשהו יקבל מנה תוך זמן סופי.
3. יעילות: השף לא מבזבז משאבי CPU אם אין לקוח ממתין למנה או מקבל מנה.
4. עמידה בתנאי התו הסגול: לכל היותר 20 לקוחות ממתינים במסעדה. לקוח מוגדר כממתין במסעדה אם הוא בביצוע של אחד משורות הקוד המסומנות בין start of waiting block ל-end of waiting block.

תזכורת:

int sem_init(sem_t **sem*, int *pshared*, unsigned int *value*);

מאתחלת את הסמפור sem בערך value. אם pshared גדול מ-0 אז הסמפור משותף לתהליכים. אם pshared הוא 0 אז הסמפור משותף בין ת'רדים שונים של אותו תהליך אבל לא בין תהליכים. ערך ההחזרה הוא 0 אם הפעולה הצליחה ואחרת 1-.

int sem_post(sem_t **sem*);

שקולה לפעולה up שנלמדה בכיתה

int sem_wait(sem_t **sem*);

שקולה לפעולת down שנלמדה בכיתה

הסמפורים אינם הוגנים אלא אם נכתב במפורש שהם הוגנים.

שאלה 4מועד א 2020

א. (6 נק') מוצע הפתרון הבא:

```
Initialization of shared variables (איתחול משתנים משותפים):
sem_init(customers, 1, 0); // initialized to 0, shared among processes
sem_init(chef, 1, 20); // initialized to 20, shared among processes
```

```
void chef (void) {
  while(TRUE) {
        sem_wait (&customers);
        prepareMeal();
        sem_post (&chef);
    }
  }

void customer (void) {
    \\ start of waiting block - תחילת קטע הקוד בו תהליכים ממתינים לשירות
    sem_post (&customers);
    sem_wait (&chef);
    \\ end of waiting block - סיום - קטע הקוד בו תהליכים ממתינים לשירות
    getMeal();
}
```

לכל אחת מהתכונות הבאות סמן האם היא מתקיימת או לא:

- מניעה הדדית כן/לא
- התקדמות כן/לא
- יעילות כן/לא
- עמידה בתנאי תו סגול כן/לא

שאלה 4 מועד א 2020

לכל אחת מהתכונות הבאות סמן האם היא מתקיימת או לא:

- מניעה הדדית כן/לא
- התקדמות כן/לא
- יעילות כן/לא
- עמידה בתנאי תו סגול כן/לא

תשובה:

- אין מניעה הדדית מאחר ש-20 לקוחות יכולים להכנס, עושים post ל-customers, לא ממתינים על ה-chef (כי הוא מאותחל ל-20) ומבצעים getMeal בו זמנית.
- יש התקדמות מאחר שבכל פעם שמגיע לקוח הוא יעשה post ל-customers, השף יעשה post ל-chef ויאפשר ללקוחות נוספים לקבל שירות.
- הקוד יעיל כי כשאין לקוחות customers הוא 0 והשף לא פועל.
- כמו כן, אין עמידה בתנאי התו הסגול כי למשל הלקוח ה-21 יכול להכנס ולחכות על chef (במקום שיצא מהמסעדה).

שאלה 4 מועד א 2020

ב. (6 נק') לקוד התווספו שורות הקוד הבאות (בהדגשה):

```
Initialization of shared variables (איתחול משתנים משותפים):
sem_init(customers, 1, 0); // initialized to 0, shared among processes
sem_init(chef, 1, 0); // initialized to 0, shared among processes
int cnt_customers=0;
```

```
void chef (void) {
  while(TRUE) {
        sem_wait (&customers);
        prepareMeal();
        sem_post (&chef);
```

```
  }
}

void customer (void) {
        if (cnt_customers>=20) {
                // leave shop - הלקוח עוזב את החנות בלי לקבל שירות
                return;
          }
        cnt_customers = cnt_customers+1;
        // start of waiting block - תחילת קטע הקוד בו תהליכים ממתינים לשירות
        sem_post (&customers);
        sem_wait (&chef);
        // end of waiting block - סיום - קטע הקוד בו תהליכים ממתינים לשירות
        getMeal();
        cnt_customers = cnt_customers-1;
}
```

לכל אחת מהתכונות הבאות סמן האם היא מתקיימת או לא:

- מניעה הדדית כן/לא
- התקדמות כן/לא
- יעילות כן/לא
- עמידה בתנאי תו סגול כן/לא

לכל אחת מהתכונות הבאות סמן האם היא מתקיימת או לא:

- מניעה הדדית כן/לא
- התקדמות כן/לא
- יעילות כן/לא
- עמידה בתנאי תו סגול כן/לא

תשובה:

- אין מניעה הדדית כי מספר לקוחות יכולים לחכות על ה-wait של ה-chef. השף ישחרר אותם אחד אחד אבל בין השחרור לבין אחד לשני הם יכולים לא לסיים לבצע את getMeal – כלומר יריצו אותה במקביל.
- התקדמות ויעילות – כמו ב-א'.
- עדיין אין עמידה בתנאי התו הסגול כי מספר לא מוגבל של לקוחות יכול לעדכן את cnt_customers בו זמנית, וכך למשל 100 לקוחות יכולים ביחד לראות שה-cnt_customers הוא 19 ואז לעבור את התנאי ה-if ולהכנס למסעדה.

ג. (6 נק') לקוד התווספו שורות הקוד הבאות (בהדגשה):

```
Initialization of shared variables (איתחול משתנים משותפים):
sem_init(customers, 1, 0); // initialized to 0, shared among processes
sem_init(chef, 1, 0); // initialized to 0, shared among processes
sem_init(barrier, 1, 0); // initialized to 0, shared among processes
int cnt_customers=0;
```

```
void chef (void) {
  while(TRUE) {
        sem_wait (&customers);
        prepareMeal();
        sem_wait (&barrier);
        sem_post (&chef);

  }
}

void customer (void) {
        if (cnt_customers>=20) {
                // leave shop - הלקוח עוזב את החנות בלי לקבל שירות
                return;
```

```
        }
        cnt_customers = cnt_customers+1;
        // start of waiting block - תחילת קטע הקוד בו תהליכים ממתינים לשירות
        sem_post (&customers);
        sem_wait (&chef);
        //   end of waiting block - סיום - קטע הקוד בו תהליכים ממתינים לשירות
        getMeal();
        sem_post(&barrier);
        cnt_customers = cnt_customers-1;
}
```

לכל אחת מהתכונות הבאות סמן האם היא מתקיימת או לא:

- מניעה הדדית כן/לא
- התקדמות כן/לא
- יעילות כן/לא
- עמידה בתנאי תו סגול כן/לא

לכל אחת מהתכונות הבאות סמן האם היא מתקיימת או לא:

- מניעה הדדית כן/לא
- התקדמות כן/לא
- יעילות כן/לא
- עמידה בתנאי תו סגול כן/לא

תשובה:

- אין התקדמות מאחר שיש deadlock. סמפור barrier מאותחל ל-0, הלקוחות מחכים לשף שיעשה up ל-chef ורק אחר כך עושים up ל-barrier , כאשר השף, מצידו עושה up ל-chef רק אחרי שהוא עובר את ה-barrier.
- מאחר שיש deadlock לפני ביצוע getMeal, אף לקוח לא יריץ getMeal, בפרט לא יהיה יותר מאחד שיריצו בו זמנית.
- כמו כן, עדיין אין עמידה בתנאי התו הסגול כי מספר לא מוגבל של לקוחות יכול לעדכן את cnt_customers בו זמנית.
- עדיין יש יעילות מאחר שכשאין לקוחות השף לא פועל.

ד. (6 נק') בקוד השתנה סדר שתי השורות המודגשות בפונקציה chef (מודגש):

```
Initialization of shared variables (איתחול משתנים משותפים):
sem_init(customers, 1, 0); // initialized to 0, shared among processes
sem_init(chef, 1, 0); // initialized to 0, shared among processes
sem_init(barrier, 1, 0); // initialized to 0, shared among processes
int cnt_customers=0;
```

```
void chef (void) {
  while(TRUE) {
        sem_wait (&customers);
        prepareMeal();
        sem_post (&chef);
        sem_wait (&barrier);

  }
}

void customer (void) {
        if (cnt_customers>=20) {
                // leave shop - הלקוח עוזב את החנות בלי לקבל שירות
                return;
         }
        cnt_customers = cnt_customers+1;
        // start of waiting block - תחילת קטע הקוד בו תהליכים ממתינים לשירות
        sem_post (&customers);
        sem_wait (&chef);
        // end of waiting block - סיום - קטע הקוד בו תהליכים ממתינים לשירות
        getMeal();
        sem_post(&barrier);
        cnt_customers = cnt_customers-1;
}
```

לכל אחת מהתכונות הבאות סמן האם היא מתקיימת או לא:

- מניעה הדדית כן/לא
- התקדמות כן/לא
- יעילות כן/לא
- עמידה בתנאי תו סגול כן/לא

לכל אחת מהתכונות הבאות סמן האם היא מתקיימת או לא:

- מניעה הדדית כן/לא
- התקדמות כן/לא
- יעילות כן/לא
- עמידה בתנאי תו סגול כן/לא

תשובה:

- בקוד זה יש התקדמות (שכן בעיית ה-deadlock נפתרת בשל שינוי סדר השורות שמונעת את ההמתנה המעגלית).
- עם זאת, עדיין מספר לקוחות יכולים להגיע לתנאי ה-if ביחד ולראות שיש רק 19 לקוחות ואז להכנס למסעדה למרות שרק אחד מהם אמור לעשות זאת על פי דרישות התו הסגול.

Ex 3 – MapReduce – cont.

- MapReduce is used to parallelise tasks of a specific structure, defined by two functions, *map* and *reduce*:
  - The input is given as a sequence of input elements.
  - Map phase - The *map* function is applied to each input element, producing a sequence of intermediary elements.
  - Sort/Shuffle phase - The intermediary elements are sorted into new sequences (more on this later).
  - Reduce phase - The *reduce* function is applied to each of the sorted sequences of intermediary elements, producing a sequence of output elements.
- The output is a concatenation of all sequences of output elements.

Ex 3 – MapReduce - example

Counting character frequency in strings

- The input is a sequence of strings.
- **Map phase** - in each string we count how many times each character appears and then produce a sequence of the results.
- **Sort/Shuffle phases** - we sort the counts according to the character, creating new sequences in the process. Now for every character we have a sequence of all counts of this character from all strings.
- **Reduce phase** - For each character we sum over its respective sequence and produce the sum as a single output.
- The output is a sequence of the sums.

Ex 3 – MapReduce – example – cont.