תרגול 7

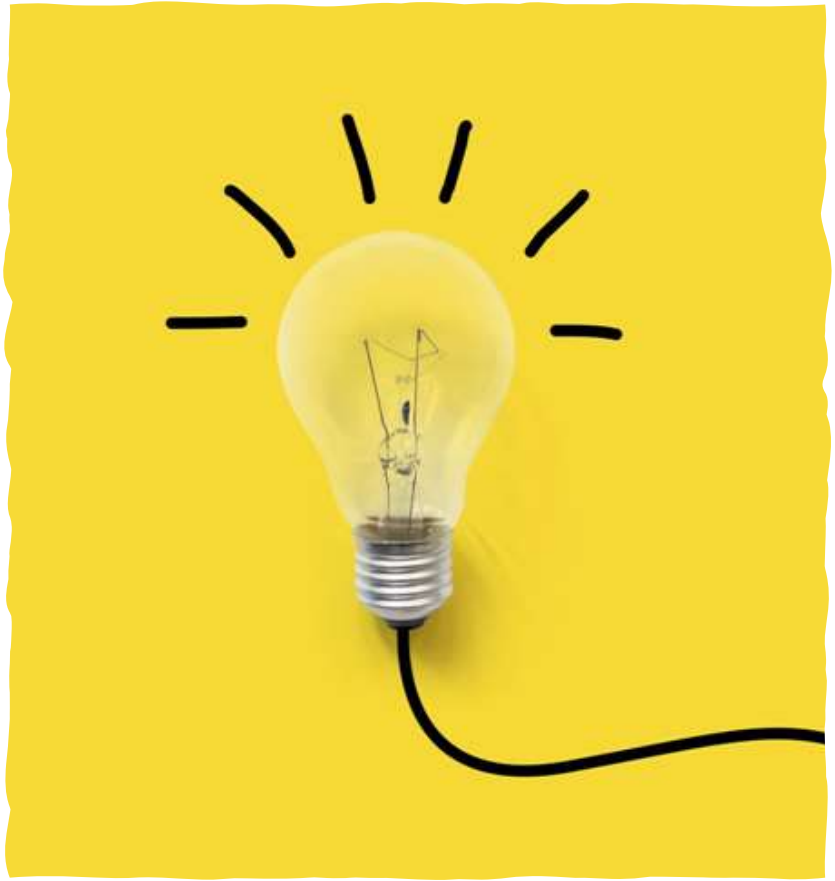
מערכות הפעלה

תודה למר ברק גונן על הבסיס לתרגיל.

על הפרק היום

- בעית הסנכרון
- RACE CONDITION
- סנכרון בין תהליכים ובין תהליכונים
- מימוש מנגנון מנעול MUTEX וסמפור SEMAPHORE
- בעית הפילוסופים הסועדים
- פתרון בעית הפילוסופים הסועדים בסביבת UNIX

60 שניות של מושג בקצרה – **סנכרון תהליכים**

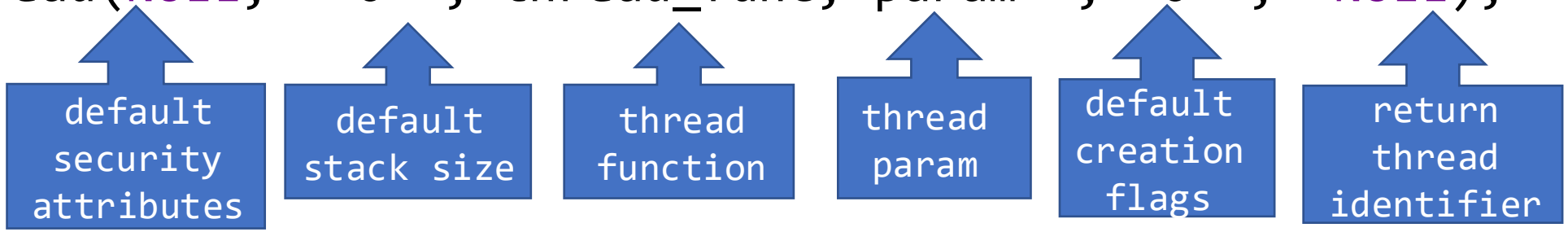


- כפי שכבר למדתם בהרצאה יש צורך לסנכרן בין התהליכים המתבצעים במערכת
  - סינכרון = בו זמניות, תיאום זמנים, הפעלה בעת ובעונה אחת. התאמת התרחשויות שונות, כך שיעבדו באחדות בהתייחס לאלמנט הזמן.
- עלולה להיווצר בעיה כאשר שני תהליכים משתמשים באותו מידע מהזיכרון. גישה בו-זמנית לנתונים משותפים עלולה ליצור מצב של חוסר עקביות במידע.
- בעיה זו נוצרת בעת שימוש ב multi-programing. כאשר שני התהליכים משתמשים במידע לסירוגין, המידע עלול להיות שגוי עבור אחד מן התהליכים, או שניהם.

עבודה עם תהליכונים בסביבת windows

- חלק מהתרגיל שלהלן נדגים בסביבת visual studio. לכן נכיר את ההגדרות הרלוונטיות ליצירת תהליכונים בסביבה זו (והן שונות ממה שלמדנו על סביבת Linux):
- יש להכליל את הספריה המתאימה
- `#include <windows.system.threading.h>`
- טיפוס הנתונים המתאים לשמש כמצביע לתהליכון הוא `HANDLE`
- המתודה ליצירת תהליך נראית כך

```
CreateThread(NULL,   0   , thread_func, param   ,   0   ,   NULL);
```



שקופית זו לקוחה
מספר מערכות הפעלה מאת
ברק גונן

הדגמה – מרוץ תהליכים (race condition)

```
#include <windows.system.threading.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 2
#define MAX 10000
VOID inc(PINT param) {
        INT temp = *param;
        temp++;
        *param = temp;
}
DWORD WINAPI thread_func(LPVOID param
        for (INT i = 0; i < MAX; i++)
                inc((PINT)param);
        }
        return 1;
}
```

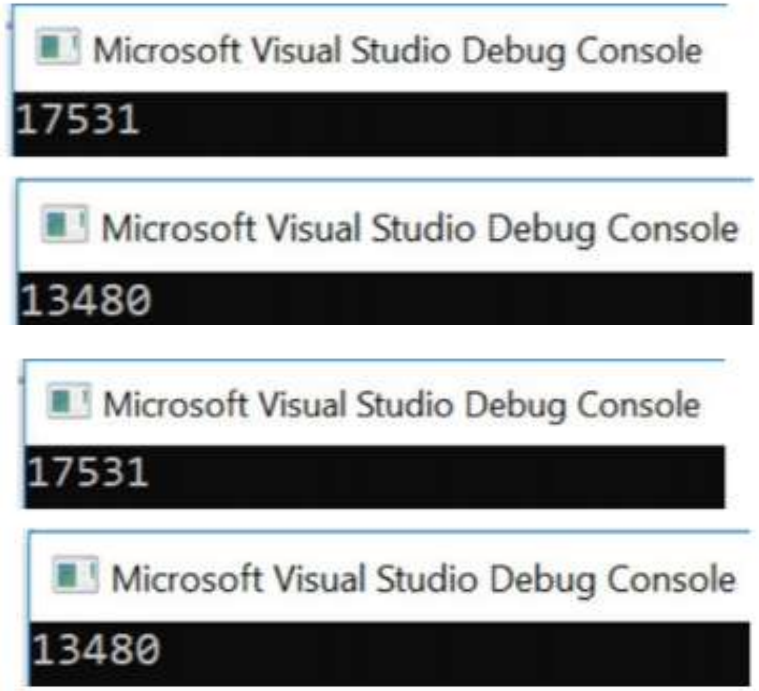
המתנה
של
תהליכון
האב
לסיום כל
תהליכוני
הבנים

```
int main() {
        INT val = 0;
        LPVOID param = &val;
        HANDLE hThread[NUM_THREADS];
        for (INT i = 0; i < NUM_THREADS; i++)
        {
                hThread[i] = CreateThread(
                        NULL,   0,
                        thread_func,
                        param,
                        0, NULL         );
        }
        WaitForMultipleObjects(NUM_THREADS,
                hThread, TRUE, INFINITE);
        printf("Param value: %d\n",
                        *(PINT)param);
}
```

שקופית זו לקוחה
מספר מערכות הפעלה מאת
ברק גונן

תוצאה

- הקוד יוצר שני תהליכונים, שכל אחד מהם מקבל כתובת של אותו משתנה. כל תהליכון אמור לקדם ערכו של המשתנה 10000 פעם ע"י פעולת inc.
- באופן טבעי, היינו מצפים שבסיום הריצה ערכו של param יהיה שווה ל-20000 .
- הנה תוצאות של ארבע הרצות- באף אחת מהן התוצאה אינה קרובה לתוצאה הצפויה:
  - לא רק שהתוצאות אינן 20000 ,יש גם שונות רבה מאד בין התוצאות. מחשבים לא אמורים לתת תוצאה שונה בכל פעם שמריצים את אותה תוכנה.
- זאת מאחר והגישה למשתנה param מהווה **קטע קריטי** עבור התהליכונים שהוצגו בקוד.

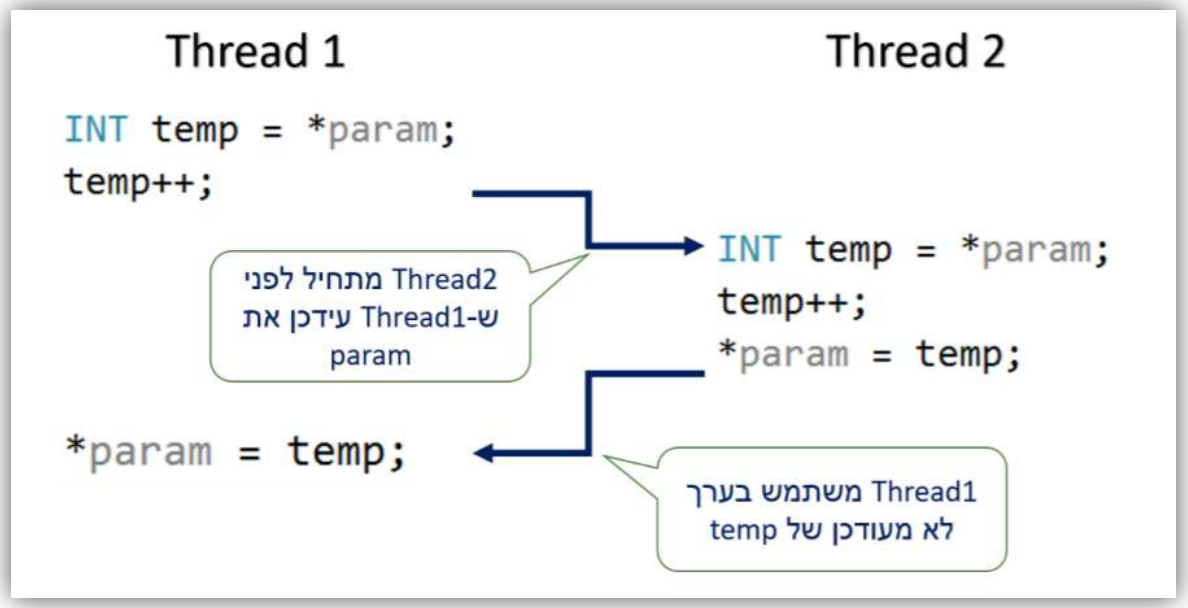


שקופית זו לקוחה
מספר מערכות הפעלה מאת
ברק גונן

מה מתרחש כאן?

- שני התהליכונים קוראים וכותבים לאותה כתובת בזיכרון.
- הבעיות מתרחשות כאשר מתבצעת החלפת הקשר בזמן לא טוב.
- כזכור – החלפת הקשר, היא ארוע שמתוזמן על ידי הסדרן של מערכת ההפעלה, ובעקבותיו המעבד עובר מהרצה של קוד של תהליכון אחד, להרצה של קוד של תהליכון אחר.

דוגמא למצב שעלול לגרום לבעיית סינכרון.
(החלפות הקשר מסומנות בחץ כחול)
כמובן, ישנן החלפות הקשר נוספות באמצע ביצוע התהליכונים, חלקן בעיתיות וחלקן אינן יוצרות בעיה



שאלה

- לפנייך פסודו קוד למימוש פעולות push ו pop על מחסנית המבוססת על מערך. בהנחה שאלגוריתם זה יכול לפעול בסביבה מקבילה, ענה על השאלות הבאות:
- עבור אילו נתונים יש מצב מרוץ?
- כיצד ניתן לתקן את מצב המרוץ?

```
push(item) {

   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR

}

pop() {

   if (!is_empty()) {
      top--;
      item = stack[top];

      return item;
   }
   else
      ERROR

}

is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

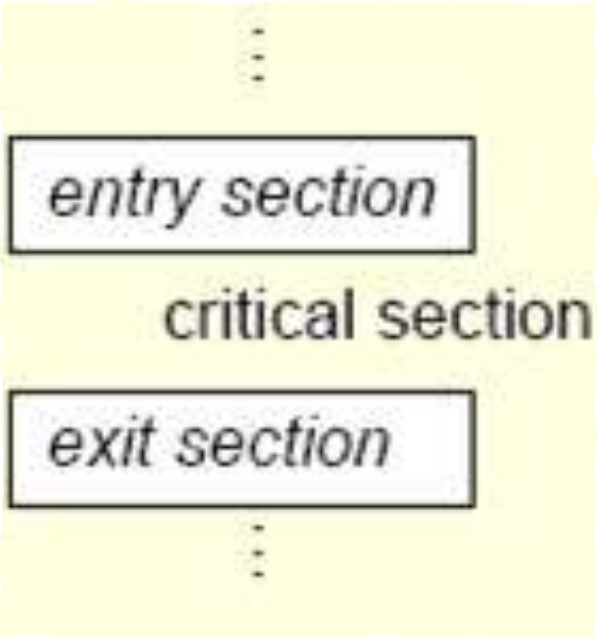
```
push(item) {
   acquire();
   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR
   release();
}

pop() {
   acquire();
   if (!is_empty()) {
      top--;
      item = stack[top];
      release();
      return item;
   }
   else
      ERROR
   release();
}

is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

תשובה

- יש מצב מרוץ לגישה למשתנה top.
- נניח שפעולת הדחיפה והמשיכה נקראות במקביל:
  - ייתכן מצב שבו לפני ביצוע ההשמה לראש המחסנית = stack [top] מתבצע שינוי של המשתנה ע"י top--.
  - במקרה כזה - הערך שנכנס למחסנית יחליף את מה שיש בראש המחסנית, וזו תהיה אז התוצאה (השגויה) של stack[top]
- ניתן לתקן את מצב המירוץ ע"י סנכרון נתונים. כלומר – הגדרה של הגישה לנתון top כקטע קריטי, וביצוע נעילה שלו.

פתרון בעיית הקטע הקריטי

- יש לוודא כי כאשר תהליך אחד מריץ את הקטע הקריטי שלו, אף תהליך אחר לא יהיה רשאי להריץ את הקטע הקריטי שלו.
- אפשרות א:
  - יצירת קטע קריטי קצר מאד, כך שלא תתבצע החלפת הקשר בזמן ביצוע שלו – **פעולה אטומית**
- אפשרות ב:
  - אם מתבצעת החלפת הקשר בזמן שעלול לפגוע בסינכרון הנתונים, התהליך שנכנס למעבד לא יוכל לגשת לנתונים – **מנעול, סמפור**
- אפשרות ג:
  - אם מגיע הזמן להחלפת הקשר עבור תהליך כך שהוא עלול לפגוע בסנכרון הנתונים, מערכת ההפעלה תרדים אותו עד שהתהליך שמשתמש באותו סמן בקוד הקריטי, יסיים – **סמפור (מסוים)**



פתרון ראשון – פעולה אטומית

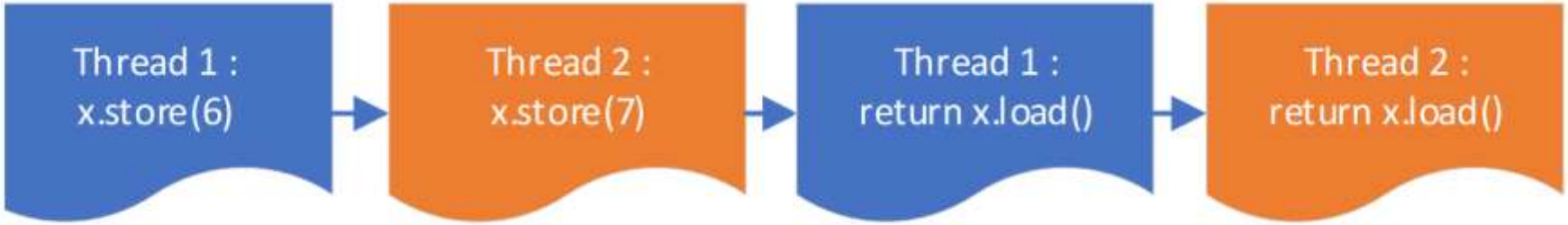
- משתנה אטומי הוא משתנה שהמעבד מבצע עליו כל פעולה, כך שלא ניתן לבצע באמצעה החלפת הקשר, ואף רכיב במחשב לא יכול להבחין / לשנות משהו בשלבי ביניים – אלא רק במצב ההתחלתי והסופי של הפעולה.

- לפניכם קטע קוד בשפת c++.
  - בקטע קוד זה מוגדר משתנה אטומי, ושתי מתודות בהן מתבצע שינוי של הערך האטומי, והחזרה של הערך האטומי מהמתודה.
- הסבירו בעזרת קטע קוד זה, מדוע השימוש במשתנה אטומי אינו יעיל תמיד

```
#include <iostream>

std::atomic<int> x =5;

int foo() {
      x.store(6);
      return x.load();
}
int bar() {
      x.store(7);
      return x.load();
}
```

תשובה:

- בהנחה ש- foo מורצת על ידי thread אחד ו- bar מורצת על יד thread אחר, אף על פי שהשתמשנו במשתנים אטומיים אין שום הבטחה שה thread שמריץ את foo יחזיר 6 מהפונקציה שלו ושה- thread שמריץ את bar יחזיר 7 מהפונקציה שלו, זאת מכיוון שהקוד שלנו היה יכול להתבצע, למשל, באופן הבא:

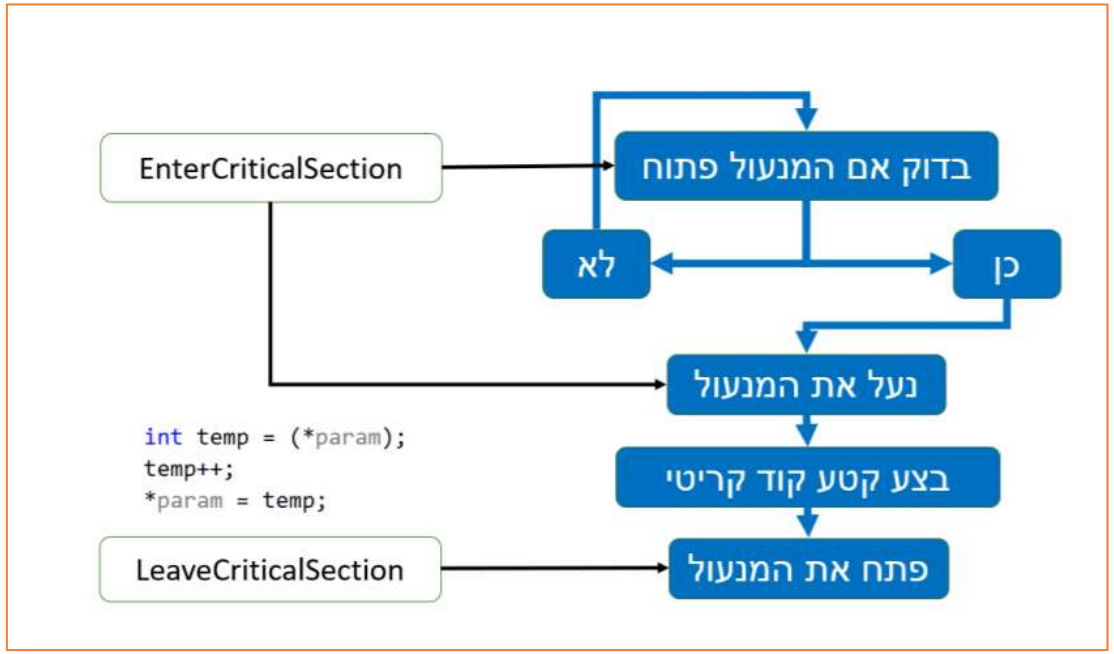


- כפי שניתן לראות אפילו בפעולות אטומיות יכול להיות מרוץ תהליכים, שכן פעולה אחת על משתנה אטומי היא אכן אטומית, אך רצף של פעולות על משתנה כזה איננו אטומי.

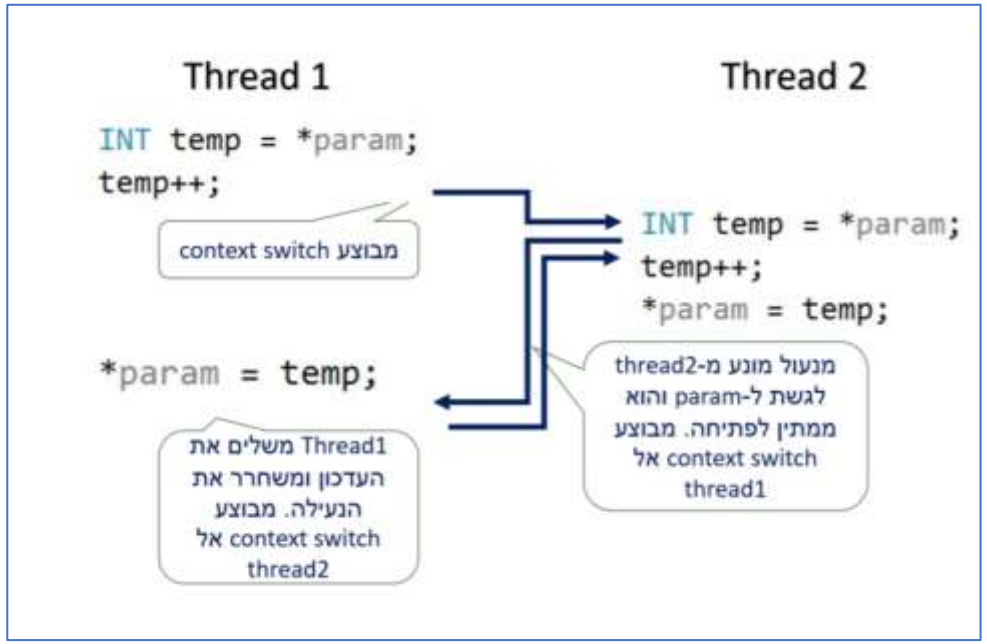
שקופית זו לקוחה
מספר מערכות הפעלה מאת
ברק גונן

פתרון שני - נעילה

מבנה של מנגנון הנעילה



וכעת, הכל תקין, גם במקרה של החלפת הקשר בזמן "לא מתאים"



הדגמת מנעול ב C#

```
sealed internal class DalList : IDal
{

    private static DalList? instance;

    private static readonly object key = new();

    public static DalList? Instance
    {
        get
        {
            if (instance == null) //Lazy Initialization
            {
                lock (key)
                {
                    if (instance == null)
                        instance = new DalList();
                }
            }

            return instance;
        }
```

עצם עבור מנעול

פונקציית מערכת של נעילה

מדוע בודקים שוב ?instance==null

Learn / .NET / C# guide / Language reference /

lock statement - ensure exclusive access to a shared resource

Article • 03/15/2023 • 3 contributors

Feedback

The `lock` statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released. The `lock` statement ensures that a single thread has exclusive access to that object.

The `lock` statement is of the form

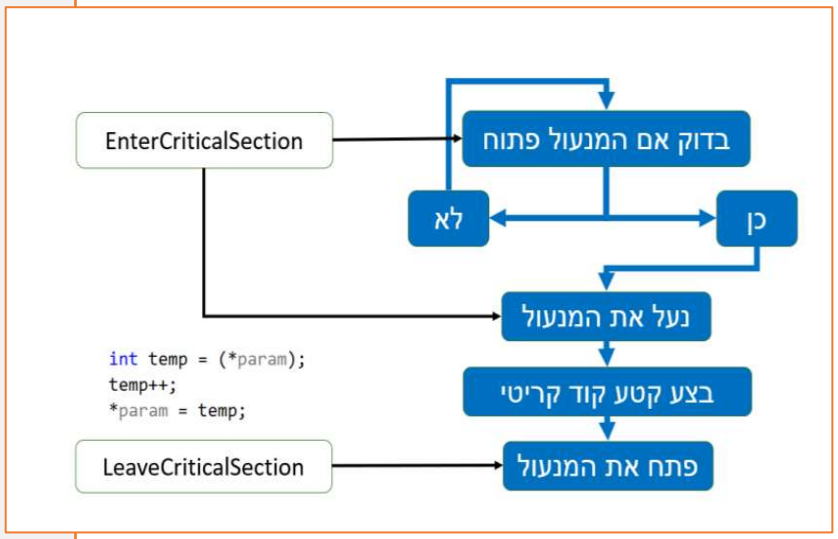
C# Copy

```
lock (x)
{
    // Your code...
}
```

בקטע קוד זה מוודאים שהעצם instance הוא סינגלטון (נוצר כמופע יחידאי), כדי שלא תהיה גישה כפולה לשכבת הנתונים DalList.

שימוש במנעול לצורך פתרון – בשפת C++

```
#include <minwinbase.h>
#include <synchapi.h>
CRITICAL_SECTION ghCriticalSection;
int main(void)
{
        InitializeCriticalSection(&ghCriticalSection);
        //…
        DeleteCriticalSection(&ghCriticalSection);
}

DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
        EnterCriticalSection(&ghCriticalSection);

        // Place critical code here

        LeaveCriticalSection(&ghCriticalSection);
        return 1;
}
```



נעילה באמצעות סמפור

- סֶמָפוֹר (semaphore) הוא מנגנון סנכרון, הכולל משתנה, שניתן לבצע עליו את הפעולות Signal(S) ו- wait(S) כאשר כל אחת מהם נעשית בצורה אטומית.
- כל תהליך מבצע wait לפני הקטע הקריטי ו signal לאחריו

```
wait(s);
// הקטע הקריטי
signal(s);
```

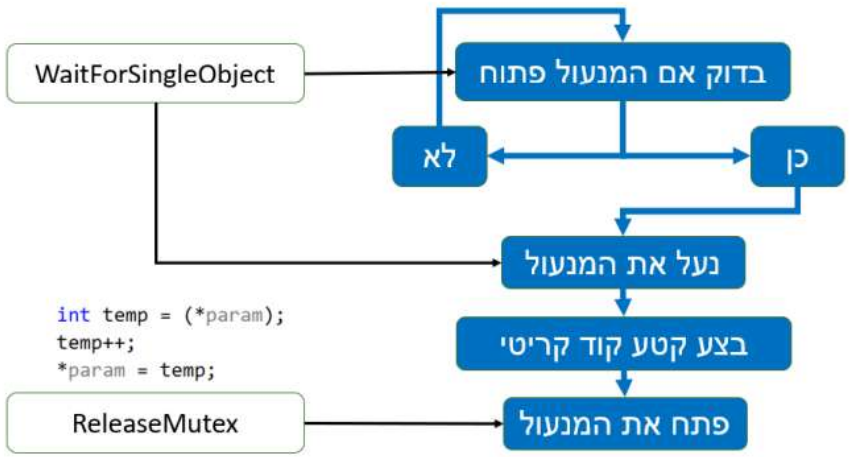
- סמפור אינו מנעול – הוא מנגנון איתות.
- הסמפור שייך למערכת ההפעלה, ולא ל thread מסוים.
- מאחר ואין לסמפור אף תהליכון שהוא הבעלים שלו – כל תהליכון במערכת יכול להעלות ולהוריד את הערך שלו.

סוגי הסמפורים

- **סמפור מונה** (מכונה גם סמפור S) מיועד לבקרת תהליכים (נתייחס לכך בהמשך הקורס)
  - לסמפור זה קיים לרוב, תור המתנה של תהליכים
  - הסמפור מאפשר ל - n ישויות להחזיק בו ברגע נתון.
    - למשל: סמפור המאפשר לn חלונות להיות פתוחים במקביל

- סמפור בו ערכו של המשתנה יכול להיות רק 0/1 , משמש כדי לפתור את בעיית המניעה ההדדית בדומה למנעול.
  - **סמפור בינארי** הוא מקרה פרטי של סמפור מונה, שבו מספר הישויות המחזיקות בסמפור הוא 1, ויש רק שני תהליכים בהם הוא מטפל.
  - **סמפור מוטקס – mutex**, דומה לסמפור בינארי, אך ההבדל הוא שמערכת ההפעלה שומרת אותו בזיכרון העבודה של אחד התהליכונים, ותהליכון זה נחשב הבעלים שלו

שימוש בסמפור Mutex



- נועד לשם מניעה הדדית בין n תהליכים
- כל התהליכים חולקים סמפור מוטקס, מאותחל ל 1.
- כל תהליך מאורגן באופן הבא:

```
do {
   wait(mutex);

      /* critical section */

   signal(mutex);

      /* remainder section */
} while (true);
```

נעבור לסביבת Linux

- שוב, קוד תוכנית המקדמת ערכו של משתנה 2000000 פעם, בשני תהליכונים, בשפת C.
- שוב – תוצאות הריצה אינן עקביות ואינן כמצופה

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NITER 1000000000

long count = 0;
void * ThreadAdd(void * a) {
        long i, tmp;
        for(i = 0; i < NITER; i++)
                count++;
}
```

```
int main(int argc, char * argv[]) {
        pthread_t tid1, tid2;
        pthread_create(&tid1, NULL, ThreadAdd, NULL);
        pthread_create(&tid2, NULL, ThreadAdd, NULL);
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
          printf("count is [%ld] Sould be %ld\n", count, 2*NITER);
        pthread_exit(NULL);
}
```

מימוש נעילה ע"י MUTEX

```
#include <pthread.h>

pthread_mutex_t mutex;
if (pthread_mutex_init(&mutex, NULL) < 0) {
        perror("pthread_mutex_init");
        exit(1);
}
```

או לחילופין איתחול כך-

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
```

הקצאת מנעול mutex

מתבצע כחלק ממנגנון הנעילה

בדוק האם מנעול פתוח

לא

כן

```
pthread_mutex_lock(&mutex);
```

נעל את המנעול

בצע קטע קוד קריטי

```
pthread_mutex_unlock(&mutex);
```

פתח את המנעול

```
pthread_mutex_destroy(&mutex);
```

שחרור הקצאת מנעול

שילוב המנעול Mutex בקוד

- הפעם תוצאות הריצה תקינות.

```
pthread_mutex_t mutex =
        PTHREAD_MUTEX_INITIALIZER;
long count = 0;
void * ThreadAdd(void * a)    {
    for(long i = 0; i < NITER; i++) {
        pthread_mutex_lock( &mutex );
        count++;
        pthread_mutex_unlock( &mutex );
    }  }
```

```
int main(int argc, char * argv[]) {
        pthread_mutex_init(&mutex, NULL);
        pthread_t tid1, tid2;
        pthread_create(&tid1, NULL, ThreadAdd, NULL);
        pthread_create(&tid2, NULL, ThreadAdd, NULL);
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
        printf("count is [%ld] ", count);
        pthread_mutex_destroy(&mutex);
        pthread_exit(NULL);
        }
```

שימוש בסמפור במערכת Linux

- כמו שאמרנו, בסביבת לינוקס, mutex ממומש ע"י מערכת ההפעלה, אך באזור הזיכרון של תהליך מסויים (user space), ולכן יש תהליך שהוא הבעלים שלו.
- לעומת זאת – סמפורים ממומשים באזור המערכת (kernal space) ולכן הם שייכים באותה מידה לכל התהליכים (אע"פ שניתן לשייך אותו לתהליך מסוים).
- כדי לפתור את בעיית הסינכרון, נזדקק לסמפור בינארי – כלומר, סמפור שמאפשר בו זמנית, רק לתהליכון אחד להיכנס לקטע הקריטי – ואז הסמפור מחזיק את הערך 0, וכן לסמן שהקטע פנוי – ואז ערכו של הסמפור הוא 1.
- בלינוקס אין סמפורים בינאריים, הסמפורים אינם מוגבלים ל-1. **במקום זאת – משתמשים בסמפור מונה עם ערך התחלתי 1. כל עוד המתכנת שומר על הבינאריות שלו הוא יפעל בצורה תקינה.**

מימוש נעילה ע"י סמפור

```
#include <pthread.h>
#include < semaphore.h>
sem_t sema;
sem_init(&sema,0,1);
```
הערך של הסמפור

הקצאת סמפור

בדוק האם מנעול פתוח

לא

כן

יש לבדוק אם ערכו של הסמפור הוא 1 (ולא 0)
מתבצע ע"י מנגנון ה wait

```
sem_wait( &sema);
```

נעל את המנעול

בצע קטע קוד קריטי

```
sem_post( &sema);
```

פתח את המנעול

```
sem_destroy(&sema);
```

שחרור הסמפור

שילוב נעילה ע"י סמפור בקוד

- גם הפעם תוצאות הריצה תקינות.

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#define NITER 1000000000
sem_t sema;
long count = 0;

void * ThreadAdd(void  * a) {
    for(long i = 0; i < NITER; i++)
   {       sem_wait( &sema);
                count++;
                sem_post( &sema);
   }
}
```

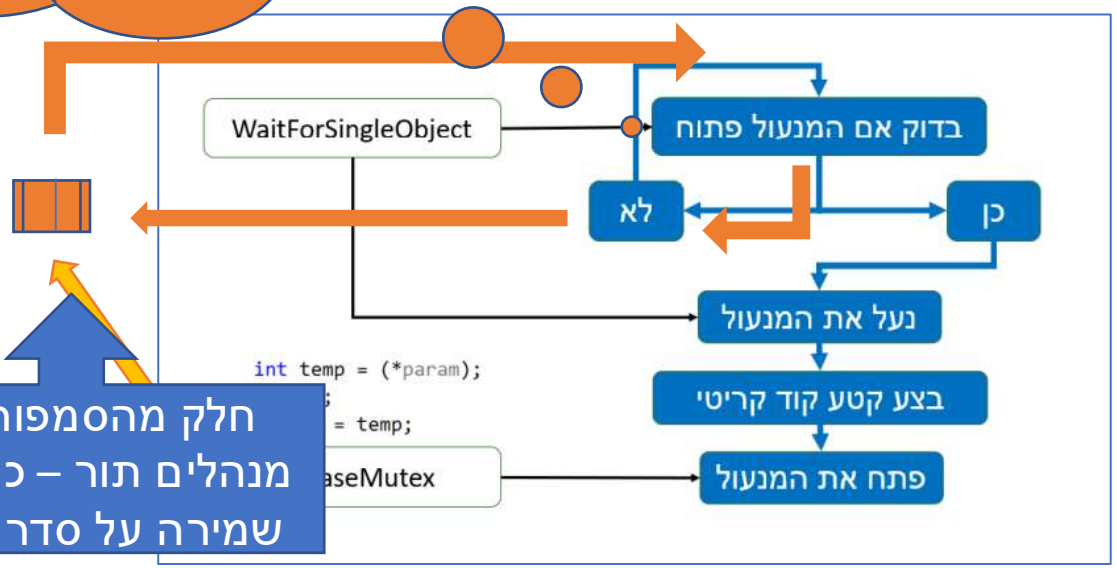
```
int main(int argc, char * argv[]) {
            pthread_t tid1, tid2;
            sem_init(&sema,0,1);
            pthread_create(&tid1, NULL, ThreadAdd, NULL);
            pthread_create(&tid2, NULL, ThreadAdd, NULL);
            pthread_join(tid1, NULL);
            pthread_join(tid2, NULL);
            printf("count is [%ld]\n", count);
            sem_destroy(&sema);
            pthread_exit(NULL);
}
```

פתרון שלישי – נעילה + הרדמה

- אם מגיע הזמן להחלפת הקשר עבור תהליך כך שהוא עלול לפגוע בסנכרון הנתונים מערכת ההפעלה תרדים אותו עד שהקטע הקריטי יתפנה.

במידה והקטע הקריטי לא פנוי, התהליך יועבר לתור המתנה. שחרור מהתור יתבצע כאשר הקטע הקריטי משתחרר

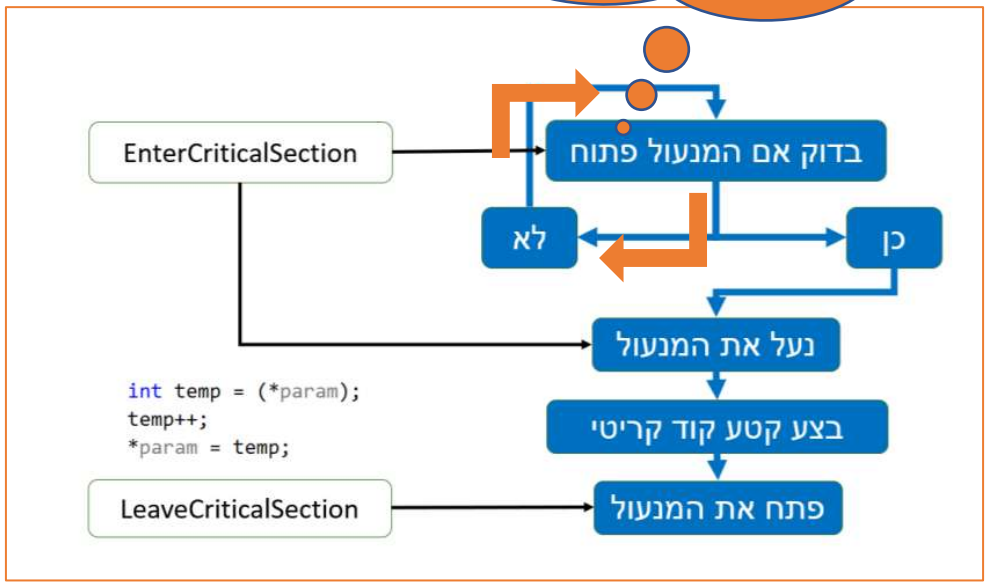
מבנה של מנגנ



חלק מהסמפורים לא מנהלים תור – כלומר אין שמירה על סדר ההגעה

הנעילה

התהליך מנצל את זמן המעבד לבדיקה חוזרת ונשנית (לולאה) האם המנעול פתוח



Spinlock VS Mutex

- מנגנון המוטקס "מרדים" את התהליכים הממתינים. אבל ללא שימוש בתור.
- לעומת זאת - ספינלוק הוא מנעול אשר מנגנון הנעילה שלו ממתין בלולאה תוך שהוא בודק שוב ושוב אם הנעילה זמינה. כלומר: מבצע המתנה עסוקה.
- בד"כ ספינלוק לא ישחרר את המעבד עצמאית (אם כי - בחלק מהיישומים הם עשויים להשתחרר אוטומטית אם התהליכון שהם ממתינים לו נחסם).
- ככל שהקטע הקריטי ארוך יותר, כך גדל הסיכון שהביצוע יקטע על ידי מתזמן מערכת ההפעלה בזמן שהוא מחזיק את הנעילה.
  - אם זה קורה, תהליכונים אחרים יבצעו המתנה פעילה (מנסים שוב ושוב לרכוש את המנעול), ואילו התהליכון שאוחז במנעול אינו מתקדם לשחרורו. התוצאה היא דחייה ממושכת עד שהתהליכון האוחז במנעול יתוזמן מחדש ויוכל לסיים ולשחרר אותו).

מתי כדאי להשתמש בSpinlock ?

- כאשר החסימה היא לזמן קצר בלבד
  - ← חוסך הרדמה, הערה, תור
- כאשר מדובר על תהליכון שיזמה מערכת ההפעלה
  - ← בד"כ אין החלפת הקשר באמצע הביצוע של תהליכון כזה.
- במערכות מרובות מעבדים, ולא במערכות עם מעבד יחיד.
  - ← מכיוון שביצוע התנאי שיוציא תהליך מהספינלוק, מתרחש רק על ידי ביצוע תהליך אחר. במערכת מרובת מעבדים, תהליכים אחרים מבוצעים על מעבדים אחרים, ולכן יכולים לשנות את מצב התוכנית על מנת לשחרר את התהליך הראשון מהספינלוק.

שאלת הבנה

- נניח כי למערכת יש ליבות עיבוד מרובות.
- עבור כל אחד מהתרחישים הבאים, תאר מהו מנגנון נעילה טוב יותר - ספינלוק או נעילת mutex (שבו התהליכים הממתינים ישנים בזמן ההמתנה), כדי שהמנעול יהיה זמין:
  - יש להחזיק את המנעול למשך זמן קצר.
    - ספינלוק
  - יש לנעול את המנעול לאורך זמן.
    - נעילת מוטקס
  - תהליכון יכול להירדם תוך כדי החזקת המנעול.
    - נעילת מוטקס

מה דעתכם?

- למחשבים המכילים ליבת לינוקס יש מדיניות שתהליך אינו יכול להחזיק ספינלוק בזמן ניסיון לרכוש סמפור.
- מדוע קיימת מדיניות זו?
- מכיוון שרכישת סמפור עשויה להרדים את התהליך בזמן שהוא מחכה שהסמפור יתפנה.
- ספינלוקס יכול לתפוס את הקטע הקריטי רק למשך זמן קצר, ותהליך שישן עשוי להחזיק את הספינלוק זמן רב מדי.

שאלת הבנה

- שרת אינטרנט שבו יש תהליכונים רבים מעוניין לעקוב אחר מספר ההפניות (hits) הקיימות באתרים אחרים אל האתר.
- מוצעות שתי אסטרטגיות, כדי למנוע תנאי מירוץ.
- הסבר איזו משתי האסטרטגיות הללו יעילה יותר.
- אסטרטגיה ראשונה:
  - שימוש במספר שלם אטומי (שמעודכן בפעולה אטומית):

```
atomic_t hits;
atomic_inc(&hits);
```

המשך

- אסטרטגיה שניה:
  - שימוש במוטקס בסיסי בעת עדכון ההפניות (באמצעות אחת מהפעולות האטומיות)

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

```
// initialization
mutex->available = 0;

// acquire using compare_and_swap()
void acquire(lock *mutex) {
    while (compare_and_swap(&mutex->available, 0, 1) != 0)
        ;

    return;
}


// acquire using test_and_set()
void acquire(lock *mutex) {
    while (test_and_set(&mutex->available) != 0)
        ;

    return;
}


void release(lock *mutex) {
    mutex->available = 0;

    return;
}
```

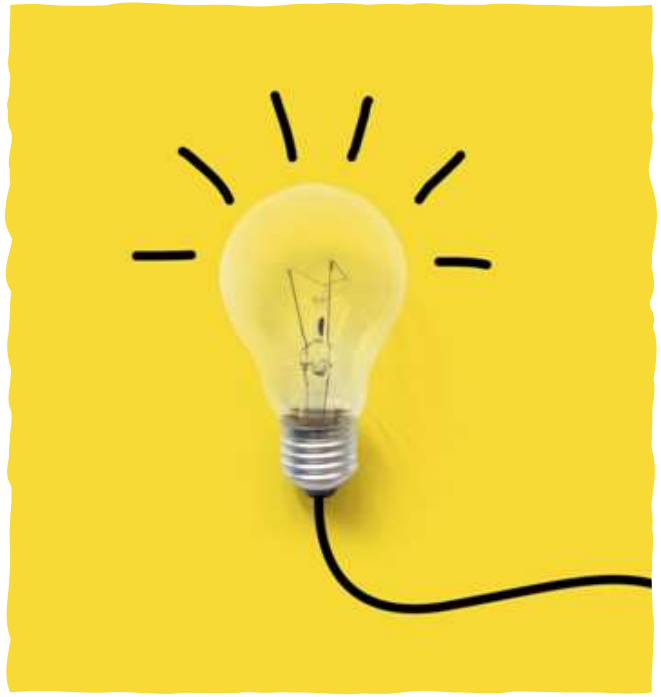
תשובה

- כמובן - השימוש במנעולים הוא מיותר במצב זה.
- נעילה בדרך כלל דורשת פסיקת מערכת ועשויה לדרוש הרדמת תהליך (שתגרום לביצוע מיתוג הקשר) אם הנעילה אינה זמינה. התעוררות התהליך תדרוש מיתוג הקשר נוסף.
- מצד שני, המספר השלם אטומי, מספק עדכון אטומי של משתנה מספר ההפניות ומבטיח שלא יהיו תנאי מרוץ, ללא התערבות של הגרעין.
- לכן, הגישה הראשונה יעילה יותר.

מדוע יש ריבוי מנגנוני נעילה?

- מערכות הפעלה (כגון: Windowsו- Linux) מיישמות כמה סוגים של מנגנוני נעילה. הסיבה לכך היא, שכל סוג של מנגנון מתאים למצב אחר.
- מערכות הפעלה אלה מספקות מנגנוני נעילה שונים בהתאם לצרכי מפתחי האפליקציות.
- ספינלוקס שימושיים עבור מערכות מרובות מעבדים בהן תהליכון יכול לרוץ בלולאה של המתנה עסוקה (לזמן קצר) כדי להימנע מהתקורה של הרדמת תהליך.
- Mutexes שימושיים לקטע קריטי של נעילת משאבים. (אגב – יש מערכות הפעלה בהם המוטקס מיושם ע"י ספינלוק)
- סמפורים ומשתני מצב מתאימים לסנכרון כאשר יש להחזיק משאב לפרק זמן ארוך (מכיוון שהספינלוק אינו יעיל למשך זמן ארוך).

60 שניות של מושג בקצרה – deadlock **קפאון**



- בהרצאה נקדיש לכך שיעור, אך בשלב זה לא נכנס לכל הפרטים
- קיפאון (נעילות, תיקו) הוא מצב בו שתי פעולות מחכות כל אחת לסיומה של האחרת, ומכיוון שכך, אף אחת מהן אינה מסתיימת.
- כלומר - מצב בו שני תהליכים מחכים האחד לאחר לשחרורו של משאב, או למצב בו יותר משני תהליכים מחכים למשאבים בשרשרת מעגלית.
  - דוגמה: תהליך א' נועל את משאב A וממתין לשחרורו של משאב B, משום שהוא זקוק לשני משאבים אלה יחדיו לשם השלמת פעולתו.
  - באותו זמן תהליך ב' נועל את משאב Bוממתין לשחרורו של משאב A, משום שגם הוא זקוק לשני משאבים אלה יחדיו לשם השלמת פעולתו.
  - בקשת הנעילה היא פעולה חוסמת ועל כן שני התהליכים הפעילו פעולות שלא יחזרו עד שישוחרר המשאב שביקשו לנעול. אך כיוון שהתהליך שמחזיק במשאב נמצא במצב דומה - הוא לא יכול לשחרר את המשאב ומכאן ששני התהליכים תקועים

בעיית הפילוסופים הסועדים

- פילוסופים (מעמי הארץ...) יושבים במעגל סביב שולחן ובו צלחות וביניהן מקלות אכילה.
- הם מקדישים את חייהם לחשיבה ואכילה בלבד, ולא מתקשרים עם שכניהם.
- מדי פעם מנסה פילוסוף להעביר לעצמו אוכל מהקערה המרכזית באמצעות שני מקלות אכילה.
  - כל פילוסוף יכול להרים רק את מקלות האכילה הקרובים אליו.
  - לא ניתן להרים את שני המקלות בו-זמנית (אלא, זה לאחר זה)
  - יש לשחרר את המקלות עם סיום האכילה.
- להלן מקרה של 5 פילוסופים סועדים. המשאבים המשותפים:
  - קערת אורז (אוסף הנתונים המשותפים)
  - מערך סמפורים [5]chopstick מאותחל ל-1

מה הבעיה שלהם?

- בעיית הפילוסופים הסועדים היא המחשה פשוטה לבעיית תזמון ותיאום של עיבוד מקבילי.
- הפילוסופים מייצגים תהליכים שרצים במקביל על מעבדים שונים. מקלות האכילה מייצגים משאבים משותפים - זיכרון מחשב, קבצים וכדומה.
- האלגוריתם המתאר את פילוסוף i:

```
do {
        wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );

        // eat

        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

        // think

} while (TRUE);
```

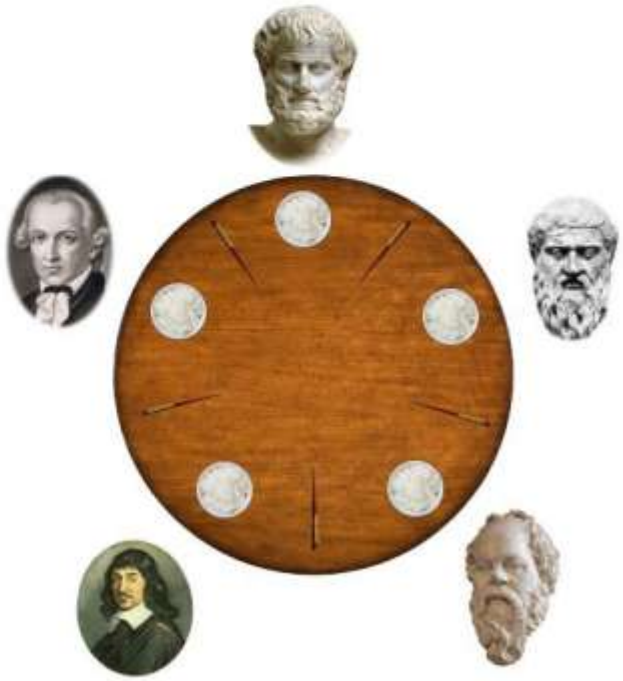
הבעיות הבסיסיות שעלולות להתהוות הן:

- **קיפאון**
  - אם הכלל שעל פיו הפילוסופים פועלים אומר כי כאשר הם רוצים לאכול עליהם להרים את המזלג שמימינם ולחכות עד שהמזלג משמאלם יתפנה, עלול להיווצר קיפאון אם כל הפילוסופים ירצו לאכול בו-זמנית - כולם ירימו את המזלג שמימינם ולאחר מכן יחכו לנצח למזלג שמצד שמאל
- **הרעבה**
  - אם דרך הפעולה של הפילוסופים היא להרים את המזלג שמצד ימין, לחכות חמש דקות למזלג שמצד שמאל, ואם הוא טרם התפנה - להניח את המזלג שבצד ימין ולחכות עוד חמש דקות לפני הניסיון הבא, המצב שמתקבל הוא הרעבה כי אף פילוסוף אינו מצליח לאכול. (לא קיפאון, שכן הפילוסופים כל הזמן מורידים ומרימים את המזלגות, ולא נתקעים במצב בודד)
  - אם מוסיפים למערכת אלמנט של עדיפות. פילוסוף האוחז מזלג ומחכה למזלג אחר, ישחרר את המזלג שבידו אם שכנו מבוגר ממנו וגם הוא מחזיק מזלג. ייתכן שהפילוסוף הצעיר ביותר אינו מספיק אף פעם לאכול, בעוד הפילוסופים משני צדדיו אוכלים לסרוגין.

פתרונות אפשריים לבעיית הקיפאון

- לאפשר לכל היותר ל-4 פילוסופים לשבת ליד השולחן
- לאפשר לפילוסוף להרים מקל אכילה, רק אם שניהם זמינים (הרמת המקל עצמה חייבת להיעשות בקטע הקריטי)
- שימוש בפתרון א-סימטרי. פילוסוף בעל מספר זוגי מרים ראשית את המקל השמאלי, ואז את המקל הימני. פילוסוף בעל מספר אי-זוגי מרים קודם את המקל הימני ואז את השמאלי.
  - הפתרון הפשוט: הפילוסוף הראשון פועל בצורה הפוכה מכולם (כולם מרימים את הימני ואז את השמאלי, והוא מרים קודם שמאלי ואז ימני)

בעית הפילוסופים הסועדים - תרגיל



- עליכם לכתוב תוכנה המממשת את הפתרון הפשוט לבעיית הפילוסופים הסועדים, תוך הימנעות מ deadlock.
- התוכנית תדפיס הודעה על מספר הפילוסוף שאוכל, בסיום האכילה הפילוסוף ישחרר את מקלות האכילה לסועד הבא.

- היה עדיף למנוע גם הרעבה , אבל בLINUX סמפרים ומנעולים לא מבטיחים הוגנות (fairness) ולא המתנה מוגבלת (bonuded waitnig) לכן הפתרון הבסיסי הפשוט לא מבטיח מניעת הרעבה

לסיכום

- התייחסנו לבעיית הסנכרון
- התייחסנו לבעיית הקפאון

- כמעבדת כיתה – יש לפתור את בעיית הפילוסופים הסועדים
- לבית – יש לתקן את בעיית הסוכנים בתכנות מקבילי תוך שימוש בסנכרון באמצעות סמפורים בינאריים

- בהצלחה!