**אוניברסיטת חיפה**
החוג למדעי המחשב
מערכות הפעלה – תרגול

# תרגול מס׳ 3

# Processes and Scheduling

## 1. Threads

### 1.1. הגדרה

ה-Thread מוגדר גם כ-Light Weight Process. הוא מכיל:

- מונה תוכנית (Program Counter)
- סט רגיסטרים (Register Set)
- מחסנית (Stack)

המידע הזה **<u>ייחודי</u>** לכל Thread.

ניתן לראות שכל ה-Threads השייכים ל-Process מסוים חולקים את אותו אזור זיכרון של ה-Process.

### 1.2. שימושים

במחשב בעל CPU יחיד, לא ניתן להריץ שני תהליכים בו-זמנית.

מה הקשר ל-Threads?

נניח שיש לנו תוכנית שמבצעת חישוב על חלק של מערך ואח״כ כותבת אותו לדיסק. את הפעולה הזו היא מבצעת בלולאה מס׳ פעמים.

לכו, ניתן להסתכל על הפעולה של התוכנית באופן הבא: פעילות ב-CPU בזמן החישובים. יציאה מה-CPU ל-I/O לכתיבת הנתונים לדיסק. כך למעשה ה-CPU לא יהיה מנוצל זמן ניכר (פעולת I/O יקרה מאוד בהשוואה לזמן CPU).

את הבעיה הזו ניתן לפתור באמצעות Thread. באותה תוכנית ניצור שני Threads. הראשון יבצע את החישובים המתאימים ויאחסן אותם במערך. השני ישמור את המערך לדיסק. בזמן שה-Thread השני שומר את האינפורמציה לדיסק, ה-Thread הראשון יכול לבצע עוד פעולות חישוב.

כך התהליך ינצל את כל זמן ה-CPU שהוא יכול ובמקביל יבצע פעולות I/O.

## 1.3. Critical Sections

כפי שהוזכר לעיל, Threads חולקים את אותו אזור זיכרון. אם נפעיל מס׳ Threads במקביל, ייתכן ששני Threads ינסו לקרוא את אותו המשתנה ואז התוצאות לא צפויות.

לכן הוגדר המושג של Critical Section – אזור בתוכנית אשר **<u>אסור</u>** ששני Threads יהיו בתוכו בו-זמנית.

הפעולה שמבצעת על Critical Section נקראת לעיתים **<u>סנכרון</u>**.

### 1.3.1. Semaphores

כדי לאכוף את הדרישה הנ״ל, אנחנו נצטרך להגן על אותו אזור בתוכנית.

ניתן להסתכל על Semaphore כעל מפתח למנעול – כאשר Thread מנסה להיכנס ל-Critical Section, הוא מבקש את המפתח למנעול המסוים של אותו אזור. לאחר שהוא סיים הוא מחזיר את המפתח למקומו וכך Thread אחר יכול להיכנס ל-Critical Section. כל Thread **<u>חייב</u>** לבקש את אותו המפתח וכך למעשה אנחנו מבטיחים **פעולה ״אטומית״** של כל Thread בקטע המסוכן (**פעולה ״אטומית״** מוגדרת כפעולה המתבצעת ללא הפרעה – לא ייתכן ש-Thread נוסף יבצע את אותה הפעולה באותו זמן שהיא מתבצעת ע״י Thread אחר).

### 1.3.2. בדיקה של Semaphores

למרות שנראה ש-Semaphores נותנים פתרון לבעיה של הסנכרון, מתעוררת הבעיה הבאה:

שני Threads מגיעים ל-Critical Section ומבקשים את המפתח. ה-Thread הראשון ניגש ל״תיבת המפתחות״ ובודק אם המפתח קיים.

לאחר שהוא מקבל תשובה ״המפתח קיים״, מתבצע Context Switch וה-Thread השני גם הוא ניגש לתיבת המפתחות ורואה שיש מפתח.

כעת שני ה-Threads מקבלים את המפתח כי הם רואים שהוא קיים; ולכן שניהם יכולים לחיות בתוך ה-Critical Section.

כדי לפתור את הבעיה הזו המציאו פתרון בחומרה שנקרא Test-and-Set: אני בודק אם קיים מפתח ואם הוא קיים אני מסמן שהוא תפוס, כל זה ב**פעולה אטומית**.

# 2. Scheduling

כאשר תהליכים מגיעים לזיכרון (ע״י איזה Scheduler?) הם נכנסים לתור. עכשיו ה-Short Term Scheduler צריך לבחור איזה תהליך להכניס למעבד.

ישנם מס׳ אלגוריתמים של תזמון, חלקם פשוטים וחלקם מורכבים.

לפני שנכיר את האלגוריתמים, נכיר מושג בסיסי בתזמון תהליכים: Preemption. כאשר אנחנו אומרים שאלגוריתם מסוים הוא Non-Preemptive, אנחנו מתכוונים לכך שמהרגע שהתהליך נכנס למעבד ועד שהוא עוזב אותו מרצונו (למשל לפעולת I/O) הוא נשאר במעבד.

אלגוריתם Preemptive עשוי להפסיק את פעולתו של תהליך מסוים שנמצא במעבד בעקבות החלטות שונות, אפילו עם התהליך לא ויתר על המעבד מעצמו.

## 2.1. FCFS – First Come First Served

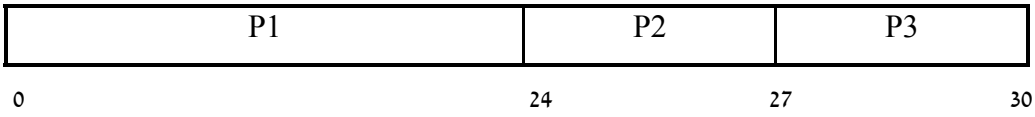
האלגוריתם הזה למעשה מדמה תור (Queue) – התהליך הראשון שמגיע הוא הראשון שמקבל את המעבד. זהו אלגוריתם Non-Preemptive כי הוא לא מפריע לאף תהליך בזמן הריצה.

למשל, נניח ומגיעים 3 תהליכים למעבד:

| Process | Burst Time |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

### 2.1.1. סדר הגעה 1

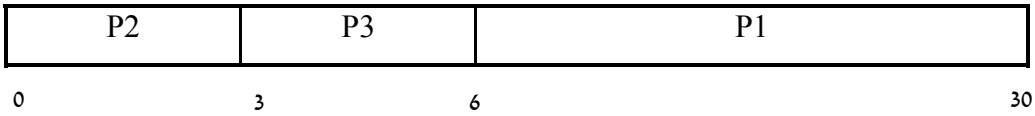
התהליכים מגיעים בסדר הבא: p1, p2, p3. סדר הריצה יהיה:




**זמן ממוצע:** $(0+24+30)/3=17$

### 2.1.2. סדר הגעה 2

התהליכים מגיעים בסדר הבא: p2, p3, p1. סדר הריצה יהיה:




**זמן ממוצע:** $(6+0+3)/3=3$

**<u>מסקנה</u>:** באלגוריתם הזה, סדר ההגעה של התהליכים משפיע מאוד על הביצועים של המעבד.

## 2.2. SJF – Shortest Job First

האלגוריתם הזה בוחר את התהליך הבא לריצה לפי ה-CPU Burst שלו. האלגוריתם יעדיף את זה בעל הזמן הבא הקצר ביותר, ללא קשר לסדר ההגעה שלהם.
זהו אלגוריתם אופטימלי לממוצע זמן המתנה במידה וכל התהליכים זמינים לאלגוריתם. זאת משום שאם אנו מעבירים תהליך קצר לפני תהליך ארוך, זמן ההמתנה שלו מתקצר יותר מאשר הזמן שנוסף לתהליך הארוך.
האלגוריתם הוא Non-Preemptive.
לדוגמה, נניח והתהליכים הבאים זמינים לאלגוריתם:

| Process | Burst Time |
|---|---|
| P1 | 6 |
| P2 | 8 |
| P3 | 7 |
| P4 | 3 |

### 2.2.1. סדר ההגעה זהה – כולם הגיעו בזמן t=0

סדר הריצה שלהם יהיה:


**זמן ממוצע:** $(3+16+9+0)/4=7$

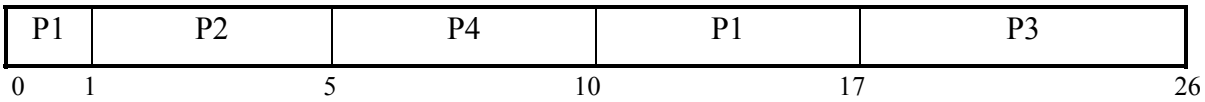
## 2.3. Preemptive SJF

האלגוריתם הזה בוחר את התהליך הבא לריצה לפי הזמן שעוד נותר לתהליך לרוץ. כלומר, התהליך שנותר לו הכי מעט זמן לרוץ, נבחר.
ה-Preemption בא לידי ביטוי בכך שאם תהליך מסוים רץ במעבד ומגיע תהליך שזמן הריצה שנותר לו קטן יותר מזה של התהליך שכרגע רץ במעבד, האלגוריתם יפסיק את עבודתו של התהליך שרץ ויקצה את המעבד לתהליך החדש.

לדוגמה, נניח שהתהליכים הבאים יגיעו ל-CPU עם זמני הריצה המפורטים:

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 1 | 4 |
| P3 | 2 | 9 |
| P4 | 3 | 5 |

שימו לב: זמן ההגעה של התהליכים שונה וכן גם זמן הריצה שנותר לכל תהליך ככל שהזמן מתקדם.
סדר הריצה שלהם יהיה:




**זמן ממוצע:** $[(10-1)+(1-1)+(17-2)+(5-3)]/4=6.5$

ישנו שיפור יחסית ל-SJF הרגיל.

## 2.4. Priority Queue

האלגוריתם הזה מנהל תור עדיפויות. לכל תהליך ניתנת עדיפות. האלגוריתם בוחר את התהליכים לפי העדיפות שלהם.

האלגוריתם הזה יכול להיות Preemptive וגם Non-Preemptive.

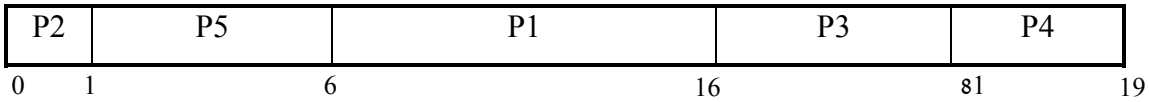
### 2.4.1. דוגמה – תזמון עם עדיפות

נניח שמגיעים התהליכים הבאים (כולם ב-**t=0**):

| Process | Burst | Priority |
|---|---|---|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 3 |
| P4 | 1 | 4 |
| P5 | 5 | 2 |

- ככל שהמס' ב-Priority יותר גבוה, זה אומר שהעדיפות יותר נמוכה.

סדר הריצה שלהם יהיה:




**זמן ממוצע:** $(6+0+16+18+1)/5 = 8.2$

## 2.5. Round Robin

זהו אלגוריתם מתוחכם שמכיל מס׳ וריאציות. בעיקרון האלגוריתם עובד עם Time Slice שהיא יחידת זמן שכל תהליך מקבל עבור שימוש במעבד. בסיום יחידת הזמן, התהליך עובר לסוף התור ואח״כ מקבל שוב את המעבד למשך אותה יחידת זמן וחוזר חלילה.

במידה ותהליך מסיים את העבודה במעבד לפני שנגמר לו ה-Time Slice, הוא מפנה את המעבד.

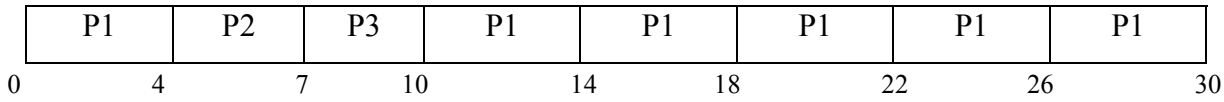
לאלגוריתם יש מס׳ וריאציות שמשלבות שימוש בתורי עדיפויות.

### 2.5.1. דוגמה לתזמון עם יחידות זמן

נניח שה-Time Slice במעבד הוא 4 – כלומר כל תהליך מקבל את המעבד ל-4 יחידות זמן. וכמו כן, נניח שמגיעים התהליכים הבאים:

| Process | Burst Time |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

סדר הריצה שלהם יהיה:




**זמן ממוצע:** $(6+4+7)/3 = 5.67$

הביצועים מושפעים מגודל ה-Time Slice. מונע בעיה של Starvation.

- שימו לב כי למרות שה-Time Slice הוא 4, התהליכים p2 ו-p3 ויתרו על המעבד לאחר 3 יחידות זמן משום שהם סיימו את עבודתם.

# 3. אלגוריתם הבנקאים (Bankers Algorithm)

לאחר שדיברנו על Threads ו-Scheduling, אנחנו עומדים בפני בעיה חדשה – Deadlock.

מהו **Deadlock**?

ובכן, בואו נחשוב על המצב הבא:

יש לנו שני תהליכים במערכת P1, P2. שניהם מבצעים פעולות חישוב מסוימות ומגיע מצב שבו שניהם מגיעים לקטע הקוד הקריטי שלהם:

Process P1

```
Lock (Resource A) {
   Lock (Resource B) {
      Continue Execution …
   }
}
```

Process P2

```
Lock (Resource B) {
   Lock (Resource A) {
      Continue Execution …
   }
}
```

נניח שתהליך P1 קיבל את מנעול A ותהליך P2 קיבל את מנעול B. כעת שני התהליכים רוצים להמשיך בפעולה שלהם אולם הם לא יכולים: תהליך P1 ממתין למנעול B (שמוחזק אצל P2) ותהליך P2 ממתין למנעול A (שמוחזק אצל P1). שני התהליכים "תקועים" כי אין ביכולתם להמשיך ואף אחד לא ישחרר את המנעול שברשותו.

מצב כזה נקרא **Deadlock.**

כדי למונע מצב כזה, נכיר את אלגוריתם הבנקאים אשר נועד להתמודד עם הקצאות משאבים כאלו.

באופן כללי (תיכף נראה דוגמאות), האלגוריתם יבדוק האם המערכת תהיה במצב בטוח (**Safe State**) לאחר הקצאת המשאבים ואם כן, הוא יקצה אותם.

- מצב בטוח (Safe State) – מצב שבו המערכת אינה יכולה להיכנס ל-Deadlock.

## 3.1. האלגוריתם

נניח שבמערכת קיימים כרגע n תהליכים ו-m סוגים של משאבים. מבני הנתונים הדרושים למימוש האלגוריתם הם:

- **Available:** Vector of length m. If available[j] = k, there are k instances of resource type $R_j$ available.
- **Max:** n x m matrix. If Max[i, j] = k, then process $P_i$ may request at most k instances of resource $R_j$.
- **Allocation:** n x m matrix. If Allocation[i, j] = k then $P_i$ is currently allocated k instances of $R_j$.
- **Need:** n x m matrix. If Need[i, j] = k, then $P_i$ may need k more instances of $R_j$ to complete its task.

כמו כן, נשים לב שמתקיים הקשר הבא:

$$Need [i, j] = Max[i, j] – Allocation [i, j].$$

נגדיר את היחס $\leq$ בין שני מבני נתונים באופן הבא:

אם x ו-y הם מבני נתונים, אזי $x \leq y$ אם מתקיים לכל i: $x[i] \leq y[i]$.

## 3.2. דוגמה

האלגוריתם הבא בודק האם המערכת נמצאת במצב בטוח (בנקודה מסוימת):

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize:
   Work = Available
   Finish[i] = false for i = 1, 2, …, n.
2. Find an i such that both:
   (a) Finish[i] = false
   (b) Needi ≤ Work
   If no such i exists, go to step 4.
3. Work = Work + Allocationi
   Finish[ i] := true
   go to step 2.
4. If Finish[i] = true for all i, then the system is in a safe state.

לדוגמה: השתמש באלגוריתם כדי לקבוע האם המצב הבא הוא Safe.

5 processes: P0 through P4;
3 resource types: A (10 instances),
B (5 instances),
C (7 instances).
Snapshot at time T0:

| | Allocation | | | Max | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 |
| P1 | 2 | 0 | 0 | 3 | 2 | 2 | | | |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | | | |

**Solution:**

Need [i, j] = Max[i, j] – Allocation [I, j]

| | A | B | C |
|---|---|---|---|
| P0 | 7 | 4 | 3 |
| P1 | 1 | 2 | 2 |
| P2 | 6 | 0 | 0 |
| P3 | 0 | 1 | 1 |
| P4 | 4 | 3 | 1 |

The system is in a safe state since the sequence < P1, P3, P4, P2, P0 > satisfies safety criteria. Is this the only possible sequence?

## 3.3. דוגמה 2

האלגוריתם הבא בודק האם ניתן להקצות משאבים לתהליך מסוים בנקודה מסוימת:

Requesti = request vector for process Pi. If $Request_i$ [j] = k then process Pi wants k instances of resource type Rj.

1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available.
3. Pretend to allocate requested resources to Pi by modifying the state as follows:
   Available = Available - Requesti;
   Allocationi = Allocationi + Requesti;
   Needi = Needi – Requesti;
4. If the new state is safe - the resources are allocated to Pi.
   If the new state is unsafe - Pi must wait, and the old resource-allocation state is restored.

# אוניברסיטת חיפה

החוג למדעי המחשב

מערכות הפעלה – תרגול

לדוגמה, השתמש באלגוריתם כדי לקבוע האם ניתן להיענות לבקשתו של תהליך P1 המבקש את המשאבים (1, 0, 2):

ראשית, נשים לב כי Request ≤ Available משום ש: (1, 0, 2) ≤ (3, 3, 2) ולכן ניתן באופן עקרוני להקצות את המשאבים ל-P1.

כעת נבדוק האם המערכת תהיה עדיין במצב Safe לאחר ההקצאה.

ובכן, לאחר ההקצאה זה יהיה מצב המערכת:

| | Allocation | | |
|---|---|---|---|
| | A | B | C |
| P0 | 0 | 1 | 0 |
| P1 | 2 | 0 | 0 |
| P2 | 3 | 0 | 2 |
| P3 | 2 | 1 | 1 |
| P4 | 0 | 0 | 2 |

| Max | | |
|---|---|---|
| A | B | C |
| 7 | 5 | 3 |
| 3 | 2 | 2 |
| 9 | 0 | 2 |
| 2 | 2 | 2 |
| 4 | 3 | 3 |

| Available | | |
|---|---|---|
| A | B | C |
| 3 | 3 | 2 |
| | | |
| | | |
| | | |
| | | |

לאחר הרצת אלגוריתם ה-Safety, נראה שהסדרה הבאה: <P1, P3, P4, P0, P2> חוקית (כלומר תספק את כל התהליכים) ותשאיר את המערכת במצב Safe.

לכן ניתן להקצות את המשאבים לתהליך P1.

האם ניתן לבצע את ההקצאות הבאות:

- תהליך P4 מבקש את (3, 3, 0)
- תהליך P0 מבקש את (0, 2, 0)