# מערכות הפעלה - Operating Systems

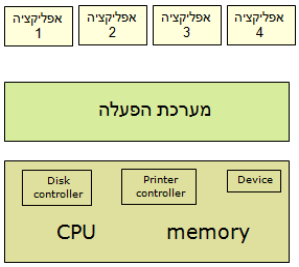
## פרק א' – הקדמה

מערכת מחשב ניתנת לחלוקה גסה של ארבעה רמות

1. חומרה – המשאבים הפיזיים של המחשב כגון מעבד, זיכרון וקלט-פלט.
2. מערכת הפעלה – מתאמת בין חומרת המחשב לבין יישומי המשתמש
3. אפליקציות – תוכניות מחשב למטרות שונות לשימוש משתמש.
4. משתמש.

### מהי מערכת הפעלה?

בעולם המחשבים אין כיום הגדרה אחת מוסכמת למושג מערכת הפעלה אך בגדול מערכת הפעלה היא תוכנה המשמשת מתווך בין משתמש במחשב לחומרה של המחשב




### סוגי מחשבים:

*PC*: מחשב בודד למשתמש בודד, דגש על נוחות שימוש וביצועים סבירים.

*Mainframe/Minicomputer*: משתמשים מרובים על אותה חומרה דרך מסופים -*terminals*. דגש על ניצול משאבים ואבטחה.

*Workstation*: מחשב עם חומרה למשתמש אחד שמחובר ומתקשר ברשת עם מחשבים נוספים. איזון בין ניצול משאבים ונוחות שימוש.

תפקיד מערכת ההפעלה בהקשר הזה הוא לנהל את השימוש בחומרה ע"י התוכנות

### מה תפקידה של מערכת הפעלה?

תפקידיה של מערכת ההפעלה הם

1. לבצע ולהריץ תוכניות של המשתמש בצורה נוחה
2. לנצל את משאבי המחשב לשימוש המשתמש
3. מערכת ההפעלה מקצה את המשאבים הנדרשים ומתמרנת בין דרישות של תוכניות שונות בעלי חשיבות שונה כמו כן היא חאגת לפקח על פעולתן של התוכניות כך שלא יגרם נזק למחשב ולתוכניות אחרות

### מי דואג לטעון את מערכת ההפעלה?

במחשב ישנה תוכנית המאוחסנת בחומרה משובצת במחשב(ואיננה ניתנת לשליפה כמו הארד-דיסק) הנקראת "קושחה" – *Firmware*. כאשר מדליקים את המחשב או מבצעים הפעלה מחדש תפקידה של הקושחה הוא לאתחל את כל שירותי המערכת לטעון את מערכת ההפעלה מהרגע שמערכת ההפעלה נטענה, נגמר תפקידה של הקושחה והיא מפסיקה את פעילותה. מעתה והלאה מערכת ההפעלה תנהל את תהליכי טעינת התוכניות התהליך של טעינת מערכת ההפעלה נקרא- *"Boot"* או אתחול.

### תמיכת חומרה

כדי שמערכת הפעלה תוכל לבצע את התפקיד שלה ולהיות מתווך בין החומרה לבין המשתמש על החומרה לספק תמיכה במספר מנגנונים שימושיים אשר מערכת ההפעלה תוכל לנצל אותם. מנגנונים אלו יכולים להיות למשל:

1. שעון חומרה
2. פעולות קלט/פלט
3. הגנת זיכרון
4. מצב עבודה מוגן
5. פקודות מוגנות
6. פסיקות.

### מצב מוגן
על מערכת חומרה בסיסית לספק לפחות שני מצבי עבודה: מצב ליבה, ומצב משתמש. מצב הליבה מיועד עבור תהליכים של מערכת ההפעלה, ויש לו סט פקודות שונה משל מצב המשתמש. במצב ליבה ניתן לבצע פעולות מסוימות שלא ניתן לבצע במצב משתמש פעולות אלו נקראות פעולות מוגנות. במצב המערכת שמור ברגיסטר מיוחד.

### פקודות מוגנות
החומרה מספקת פקודות מסוימות אשר ניתן לבצע אותן רק כאשר המערכת במצב מוגן. פעולות אלו יכולות להיות כתיבה וקריאה להתקני קלט/פלט, שינוי של נתונים המשמשים לניהול כמו תורים, מתזמנים וכ"ו, מעבר ממצב ליבה למצב משתמש ולהיפך.

### פסיקה – *Interrupt*
פסיקה היא אירוע חומרתי, המשעה את הפעילות הנוכחית של המעבד ודורש את טיפולה של מערכת ההפעלה. השימושים של פסיקות הם רבים ומגוונים וביניהם: מימוש של קריאות מערכת (*Sys Calls*), טיפול ברכיבי חומרה כגון פעולות קלט/פלט, טיפול בתקלות חומרה כגון חלוקה באפס, גישה לזיכרון לא קיים. ניהול רכיבי חומרה כגון שעון מתזמני מעבד.
ישנן שלושה סוגי פסיקות:

1. פסיקה אסינכרונית - פסיקה המתעוררת על ידי רכיב חומרה שונים, מלבד המעבד. למשל לחיצה על כפתור כלשהו במחשב, חיבור התקן אחסון חיצוני וכ"ו. נקראת אסינכרונית כי היא יכולה להתעורר בכל רגע נתון ולא כפופה ללוח זמנים או תרחיש מסוים.
2. פסיקה סינכרונית – פסיקות הנגרמות על ידי המעבד עצמו, למשל כתוצאה מהוראה לא תקינה, חלוקה באפס וכ"ו.
3. פסיקות תוכנה – סוג של פסיקה סינכרונית, הנגרמת ע"י תוכנה. למשל כאשר התוכנה מכילה הוראת מעבד מיוחדת, קריאות מערכת וכ"ו. פסיקה זו נקראת גם נקראת "טראפ - *Trap*".

### ניהול פסיקות
ישנן שתי דרכים לטפל במספר פסיקות המתעוררות ביחד:

1. *Polling*
2. *VIS - Vectored Interrupt System*

### *Polling*
בשיטה זו המערכת סוקרת ללא הפסקה בסדר מסוים וקבוע את כל ההתקנים וכאשר היא מגלה תקלת בפסיקה היא מטפלת בה.

### *VIS*
בשיטה זו כל פעם שמתעוררת פסיקה המערכת שומרת אותה בתוך מערך מיוחד המאחסן את כל הפסיקות, ואז מטפלת בהן ע"פ סדר מסוים שהמערכת תקבע.

### טיפול בפסיקה
כאשר מתרחשת פסיקה, ההתקן הרלוונטי שולח אות למעבד המורה לו על הפסיקה הייחודית לאותו התקן. המעבד מייד מפסיק את הפעולה שלו ומודיע למערכת ההפעלה על הפסיקה שהוא קיבל. מערכת ההפעלה מחזיקה טבלה מיוחדת בזיכרון, אשר משייכת לכל פסיקה את השגרה (פונקציה) המטפלת בה. כאשר מערכת ההפעלה מקבלת פסיקה, לפני שהיא מפעילה את הפונקציה לטיפול בפסיקה היא שומרת את מצב המעבד והרגיסטרים, ואז ניגשת לבצע את הפונקציה. לאחר סיום הטיפול המערכת מחזירה את המעבד לעבוד על התהליך בו הוא עצר.

### מנגנון קלט-פלט
גישה א' - סינכרונית: כאשר יש תעבורת נתונים במסלול הקלט-פלט, השליטה ניטלת מתוכנית המשתמש והיא תוחזר רק כאשר תעבורת הנתונים תסתיים והוראת המתנה גורמת למעבד לנוח עד לקבלת הפסיקה הבאה. כאשר יש התנגשות בקשות לגישה לזיכרון מופעלת לולאת המתנה. רוב רק בקשה אחת מתבצעת בכל רגע נתון – אין עיבוד מקבילי.
גישה ב' – א-סינכרונית: ברגע שיש פסיקה היא מפעילה קריאה למערכת לתת למשתמש להמתין עד שהפסיקה תחזור. הפסיקה נרשמת בתור הפסיקות משם היא מטופלת עוד לפני שהסתיים הטיפול בה. השליטה חוזרת למשתמש עד שהפסיקה תחזור מטיפול.

### טבלת מצבי התקנים

המערכת מחזיקה טבלה בה רשומים כל ההתקנים המחוברים למחשב, מצבם הנוכחי והבקשות שהם שלחו. דרך הטבלה המערכת מוציאה את הבקשות ומסמנת אותם בוקטור הפסיקות.

### מנגנון *Direct Memory Access – DMA*

לעיתים אנו נדרשים להעביר נתונים בין הזיכרון להתקן חיצוני בכמויות גדולות ובמהירות. כמו כן אנחנו לא מעוניינים להחזיק את המעבד מועסק רק במלאכת ההעתקה. לשם כך ישנו מנגנון אשר משמש לתעבורת נתונים במהירות וללא צורך בהתערבות המעבד. מנגנון זה נקרא מנגנון גישה ישירה בשלב ראשון ההתקן שולח פסיקה ראשונית אשר מודיעה למערכת שההתקן מבקש גישה. לאחר שההתקן מקבל את הגישה הוא ניגש לזיכרון ומעתיק נתונים ביחידות של **בלוקים** בפסיקה אחת ולא ביטים בודדים. ולכן אם נרצה להעתיק קובץ של *4kb* בלבד להתקן ללא *DMA* הדבר ידרוש 32,000 פסיקות! ואילו בהתקן בעל *DMA* התהליך יהיה מהיר פי מאות ואלפים

### מנגנון *System Calls*

קריאות מערכת הן אוסף של פונקצות אשר פונות למערכת ההפעלה בבקשה לבצע משימה מסוימת. הקריאה למערכת הפעלה מתבצעת בעיקר דרך ממשק שנקרא *API – Application Program Interface* אשר שייך לשפות עיליות כגון *C* או *C++*. ה*API* הנפוצים ביותר הם *Win32* של ווינדוס, או *POSIX* של כל מערכות ההפעלה מבוססות *UNIX*.

כאשר אנו קוראים לפונקצית מערכת לעיתים נידרש לספק פרמטרים לאותה הפונקציה. אבל כיצד נעביר את הפרמטרים? ישנן שלושה דרכים

1. להעביר את הפרמטרים ישירות לרגיסטרים. שיטה זו מאוד מוגבלת מכיוון שמספר הרגיסטרים מוגבל.
2. להעביר את הפרמטרים בצורה של בלוק נתונים בזיכרון אשר כתובת הבלוק מועברת לפונקציה
3. ליצור מחסנית בזיכרון אשר בה יאוחסנו הפרמטרים. הנתונים נדחפים למחסנית ע"י התוכנית הקוראת לפונקציה, ומוצאים מהמחסנית ע"י מערכת ההפעלה שמשתמשת בהם

ישנו תהליך בסיסי במערכת הפעלה שנקרא *Command Interpreter* שתפקידו לפרש את הפקודות המגיעות לקריאות המערכת ולספק להם את הכתובות והאינדקסים של הנתונים והפרמטרים המועברים אליהם

### סוגי קריאות מערכת

א. בקרה על תהליכים
ב. ניהול קבצים
ג. ניהול התקנים
ד. תחזוקת מידע
ה. תקשורת– קבלת\שליחת מידע.

לכל מערכת הפעלה יש בד"כ כלי פיתוח וסביבת פיתוח התואמים לאותה המערכת ומאפשרים בנייה של אפליקציות חדשות שיוכלו לרוץ על אותה המערכת. ככל שכלי הפיתוח של מערכת הפעלה הם יותר נוחים וידידותיים כך יותר ויותר מפתחים ומתכנתים יכתבו יישומים תואמים לאותה המערכת. תוגמא לכלי פיתוח כזה הוא *Visual Studio* של מייקרוסופט.

## פרק 2 – מבנה מערכת הפעלה

### גישות בבניית מערכת הפעלה

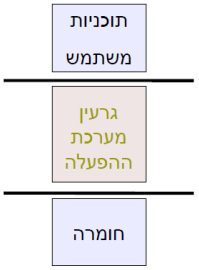
ישנן גישות שונות באשר למבנה הפנימי של מערכת הפעלה. לכל גישה יש את המאפיינים הייחודיים לה. היתרונות שלה והחסרונות שלה. למשל אחת ממערכות ההפעלה הבסיסיות ביותר היא *MS-DOS*. למרות שיש לה מבנה כללי מסוים, קשה לאפיין אותה בקווים ברורים ולמצוא בה יחידות שונות. במערכת הפעלה זו ישנן שכבות מסוימות החל מהחומרה ועד ליישומי המשתמש, אבל המעבר בשכבות הביניים אינו מחייב כך שלעיתים שכבת היישומים נושקת לשכבת החומרה והמתכנת צריך לדעת לכתוב יישומים המותאמים ספציפית לחומרה בה הוא פועל.

### א. גישת השכבות

בגישה זו מערכת המחשב מחולקת למספר שכבות אחת על גב חברתה, כאשר התחתונה ביותר היא שכבת החומרה והעליונה ביותר היא ממשק המשתמש. שכבה מספקת שירותים ופעולות לשכבה שמתחתיה. החיסרון הגדול ביותר של שיטה זו היא שאם ישום מסוים רוצה לגשת לחומרה או להשתמש בהתקני קלט-פלט הוא חייב לעבור בכל השכבות שמתחתיו, דבר הגוזל זמן רב.

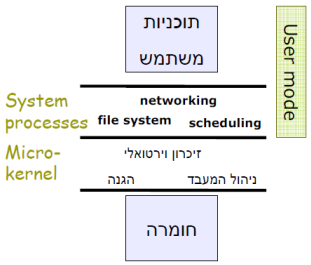
### ב. גישת המודולים

בגישת המודולים המערכת מחולקת למספר חלקים עצמאיים (מודולים) האחראים על פעולות שונות. המודולים מקושרים ביניהם כך שיוכלו לתפקד כמערכת אחת גדולה. היתרון הגדול של גישה זו היא העובדה שהמודולים אינם צריכים לעבור שכבות בדרך והם יכולים לגשת ישירות אחד לשני ולחומרה. כמו כן, ניתן לעדכן בקלות את מערכת ההפעלה על ידי הוספת מודולים חדשים לתמיכה נוספת. אבל גם החיסרון שלה נעוץ בעובדה זו שהיא "מתנפחת במהירות" בהרבה מודולים חדשים והיא הופכת כבדה יותר ויותר עם הזמן.



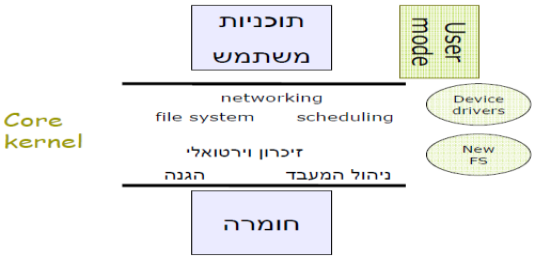

### היסטוריה – מערכת מונוליתית (הדינוזאורים מהספר)

בתחילת הדרך מערכות הפעלה היו פשוט "גוש" תכנה אחיד ללא אבחנה בין חלקים מודולאריים בעלי תפקידים שונים. דוגמא למערכת כזאת היא ה-*MS-DOS*. אין מעבר ברור בין תהליכי המשתמש לתהליכי מערכת על המתכנת להתעסק בחומרה בעצמו לפעמים.




### מיקרו-ליבה – *Micro Kernel*

לאחר מכן התפתחו מערכות הפעלה בעלות שכבת "גרעין" קטן המנהלת את החומרה עצמה, ושכבת תהליכי מערכת המספקים שירותי כגון ניהול קבצים, רשתות וכ"ו. מעל שכבות אלו באה שכבת יישומי המשתמש בה רצו התוכניות שהמשתמש מפעיל.
יתרונות שיטה זו הם האמינות הגבוהה יחסית, והיכולת לבצע שינויים ועדכונים בקלות יחסית למערכת. לעומת זאת, מערכות אלו לוקות בחסר בכל מה שקשור לביצועים בגלל המעבר בין מצב ליבה למצב משתמש.

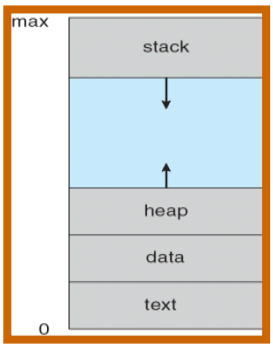



### ליבה מודולארית

כיום הגישה הרווחת היא מערכת הפעלה בעלת ליבה מודולארית, כלומר מורכבת ממספר רכיבים עצמאיים הניתנים לטעינה והסרה בכל שלב מהזיכרון. גישה זו מאפשרת הרחבה ועדכונים בקלות ללא סכנה של "התנפחות הליבה" כמו במיקרו-ליבה, כמו כן היא מאפשרת טעינה של רכיבי קוד ונתונים לפי דרישה מה שחוסך במשאבי מערכת. גם הביצועים של מערכות אלו הם טובים מאוד יחסית לשיטות האחרות. בגישה זו משתמשות כל מערכות ההפעלה המודרניות כגון *Windows, MAC OS X, Solaris, Linux*.

## פרק 3 – תהליכים

### מהו תהליך




תהליך הוא למעשה מחלקה המממשת את יחידת הביצוע של מערכת ההפעלה. כלומר כל משימה שהמשתמש רוצה לבצע, יוצרת מופע של המחלקה "תהליך". לעיתים משתמשים במילה "עבודה".
כל תהליך חייב לכלול:

א. קטע הקוד עצמו – התכנית.
ב. מונה התוכנית – *PC*.
ג. מחסנית.
ד. אזור הנתונים השייכים לאותו התהליך.

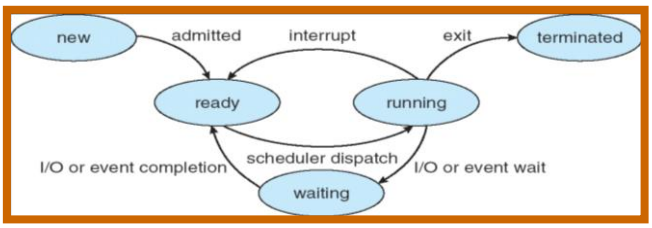
### התהליך בזיכרון – מרחב הכתובות של תהליך

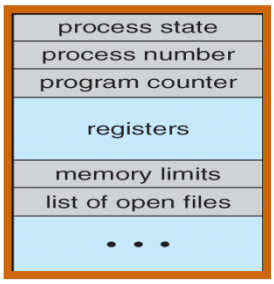
האזור המוקצה בזיכרון לכל תהליך מחולק לכמה אזורים: המחסנית, הקוד עצמו, ערימה (להקצאות דינמיות) והנתונים. נדבר עליו בהרחבה בפרק 8.

### שלבי הרצה:

אנו מבחינים בשלבים שונים בביצוע תהליכים

1. יצירת התהליך והקצאת משאבים.
2. תחילת הביצוע בפועל.
3. המתנה – לאירוע מסוים (קלט וכ"ו).
4. מוכן להרצה – חזרה ממצב המתנה.
5. סיום – התהליך סיים פעולתו והחזיר את כל המשאבים.




### בלוק בקרת התהליך:

עבור כל תהליך הקיים כרגע במערכת, מערכת ההפעלה מחזיקה מבנה נתונים חשוב מאוד בזיכרון שלה ובו נשמר כל המידע החיוני על התהליך, כדי לאפשר למערכת ההפעלה לנהל את התהליכים ולבצע את הפעולה שלהם. מבנה נתונים זה נקרא *Process – PCB Control Block* והוא מכיל מידע כגון מצב התהליך, מונה התוכנית, עדיפות וכ"ו. בעזרת בלוקים אלו מערכת ההפעלה יודעת לנהל את חלוקת המשאבים בין התהליכים, הקצאת זמן המעבד ועוד.

### החלפת הקשר – *Context Switch*

כאשר תהליך שמתבצע מקבל פסיקה על סיום הזמן המוקצה לו, או שהוא מסיים לרוץ והגיע תור של תהליך חדש, מתבצעות כמה פעולות:

1. המידע על התהליך נשמר בתוך ה-*PCB* שלו (כגון נקודת העצירה – *P.C* וכן כל הרגיסטרים).
2. המידע על התהליך החדש נטען מתוך ה-*PCB* שלו.
3. פסיקה להתחלת ריצת התהליך החדש
4. ריצת התהליך עד שמתקבלת פסיקה על סיום הזמן המוקצה לתהליך וחוזר חלילה.

מנגנון זה של החלפת תהליכים נקרא *Context Switch*. בזמן זה המערכת למעשה נמצאת במצב מבוזבז שבו איננה מבצעת שום דבר "שימושי".

**שאלה:** כיצד מערכת ההפעלה יודעת לעבור מתהליך מסוים לתהליך אחר?

**תשובה:** לשם כך היא מחזיקה מספר תורים המשמשים לקביעה איזה תהליך יבוא אחרי מה:

**תור העבודות** – מכיל את כל התהליכים הקיימים במערכת בכל המצבים

**תור התהליכים המוכנים** – מכיל את כל התהליכים שכבר נמצאים בזיכרון הראשי במצב מוכן

**תור ההתקנים** – מכיל את כל התהליכים המחכים להתקן מסוים. (ייחודי לכל התקן) .

התהליכים נודדים בין התורים במהלך הביצוע שלהם

### מתזמנים - *Schedulers*

מתזמנים הם מנגנונים המנהלים רשימת תהליכים

**יומן לטווח ארוך או מתזמן העבודות** – קובע איזה תהליך יכנס לתור התהליכים המוכנים לביצוע. יומן זה משתנה בתדירות גבוהה יחסית (שניות, דקות). יומן זה קובע בעצם את מידת הרב-תוכניתיות *(Multiprogramming)* של המערכת

**יומן לטווח קצר או מתזמן ה-CPU** – קובע איזה תהליך מתור התהליכים המוכנים יכנס לביצוע במעבד. יומן זה משתנה בתדירות גבוהה מאוד (מילישניות). כמו כן הוא קובע את מידת הרב-משימתיות *(Multitasking)* של המערכת

בחלק ממערכות ההפעלה ישנו יומן נוסף הנקרא **יומן לטווח בינוני**. יומן זה מבצע העתקה מהירה של ה-*PCB* של תהליך שזה עתה סיים את זמן המעבד שלו, בחזרה אל תחילת תור התהליכים המוכנים ובכך חוסך זמן מנוחה למערכת

בחלוקה גסה ניתן לסווג את התהליכים לשתי קטגוריות עיקריות:

1. תהליכים צמודי מעבד – תהליכים המבצעים רוב הזמן פעולות חישוביות בעזרת המעבד.
2. תהליכים צמודי קלט-פלט – עסוקים רוב הזמן בפעולות של קלט פלט ופחות בפעולות של מעבד.

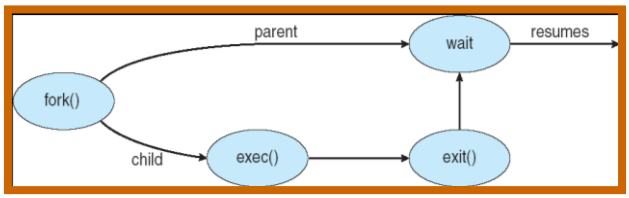
## יצירת תהליכים

תהליך-אב יכול ליצור תהליך-בן, אשר בתורו יוכל לייצר תהליכי משנה כך שייווצר עץ תהליכים הנגזרים מתהליך האב.

כל התהליכים תחת אותו העץ חולקים את המשאבים ממשאבי האב והם אינם מקבלים משאבים חדשים מהמערכת. כל הילדים של אותו האב חולקים משאבים מסויימים בינם לבין עצמם אך הם אינם חוקים משאבים עם תהליך האב

תהליך האב מתבצע יחד עם תהליכי הבנים תהליך האב אינו מסתיים עד לסיום כל תהליכי הבנים

במערכת הפעלה *UNIX* הפקודה ליצירת תהליך בן חדש נקראת *fork*. מיד אחרי *fork* תבוא פקודת *exec* שתפקידה להחליף את הקוד של תהליך האב המשוכפל לתוך הבן בקוד אחר אותו אנו רוצים להריץ תרשים:




דוגמא לקוד היוצר בן:

```
int main()
{
Pid_t pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

## סיום תהליכים

כאשר תהליך מסיים לבצע את הפקודה האחרונה שלו הוא פונה בבקשה למערכת ההפעלה למחוק אותו

# פרק 4 - תהליכונים

## ריבוי תהליכים ותהליכים עם בעיות השמנה

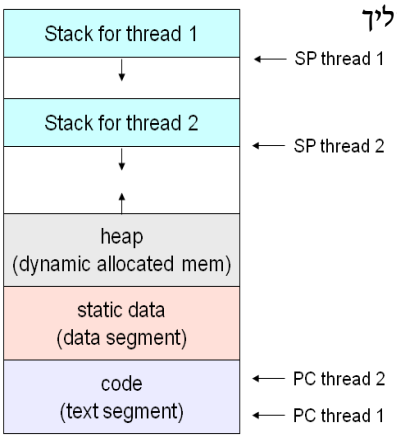
לעיתים רבות משימות שונות יכולות להיפתר ע"י ריבוי תהליכים בקלות ובפשטות, בעיקר כאשר אנו רוצים לבצע כמה משימות ללא תלות אחת בשנייה.
אך מנגד, פתרון זה גוזל זמן ומשאבים רבים כי הפעולות הכרוכות ביצירה וניהול של תהליכים גוזלות המון זמן מעבד ומקום בזיכרון.
אז מה עושים? דיאטה!

המטרה: ליצור תהליכים קלי משקל (*Light Weight Processes –LWP*).
התהליכים קלי המשקל משתפים ביניהם את אותו מרחב הכתובות, אותן הרשאות ואותם המשאבים וכך חוסכים את העלות הגבוהה של ריבוי תהליכים.

כאמור, תהליך תכיל לעיתים מספר תהליכי משנה קטנים הנקראים גם "חוטים" (*Threads*). כמו לתהליך רגיל, גם לחוט דרושים משאבים כמו *PC*, מחסנית וכ"ו. וכמו לתהליך רגיל גם לחוט יש בלוק מידע המכיל את כל הנתונים על החוט בלוק זה נקרא *TCB – Thread Control Block*.

## מרחב הכתובות של תהליך מרובה חוטים

בתהליך מרובה חוטים כל החוטים מקבלים חלק ממרחב הכתובות של תהליך האב לצרכיהם המקומיים ואין להם מרחב כתובות משלהם, אך לחוטים אין ערימה משלהם והם חולקים כולם את אותה הערימה של האב.




## מימוש ריבוי משימות באמצעות ריבוי תהליכונים

- חוט מנהל
  1. מקבל בקשה.
  2. מייצר תהליכון עבודה ומעביר הלאה לביצוע
- תהליכון עבודה
  1. מבצע את הבקשה.
  2. מחזיר תשובה.

יתרונות:

- יצירת חוט יעילה – אין צורך בשכפול רב של נתונים, בלוק בקרה קטן, החלפה מהירה בין תהליכונים.
- ניצול משאבים טוב יותר – אם חוט אחד נחסם החוטים האחרים ממשיכים לרוץ ללא קשר, מקביליות אמיתית במערכות מרובות מעבדים (או מעבדים מרובי ליבות).
- תקשורת נוחה יותר בין חוטים השייכים לאותו תהליך אב.

חיסרון:

- חוסר הגנה בין חוטים באותו תהליך – חוט אחד עלול לדרוס את המחסנית של חוט אחר, הגישה למשתנים המשותפים לכל התהליכונים אינה מתואמת ומבוקרת.

## תהליכוני משתמש ותהליכוני מערכת

- חוטי משתמש – (*User Threads*)
  חוטים אשר למשתמש יש שליטה עליהם, מוגדרים על ידי סביבת התכנות, אינם דורשים קריאות מערכת, זימון בשיתוף פעולה ע"י פקודת *yield*, אין החלפת קשר בגרעין.
- חוטי מערכת – (*Kernel Threads*).
  חוטים קלי משקל המוכרים למערכת ההפעלה.

## כמה חוטים שווה ליצר בכל תהליך?

נניח שמעבד מסוים מקצה $P$ זמן לכל תהליך בממוצע ו $C$ זמן לכל חוט. אזי ברור שלא שווה לייצר יותר מ $\frac{P}{C}$ תהליכונים לכל תהליך כי המעבד לא יספיק לעבור על כל התהליכונים, וסכן ריבוי תהליכונים כאלה מעלה את התקורה עבור פעולות של החלפת קשר וסנכרון, ומקטין את השטח המוקצה בזיכרון לכל תהליכון. וכן לא שווה לייצר פחות כי המעבד לא ינצל את מירב הזמן המוקצה לאותו תהליך.

### *Thread Pooling*

דרך יעילה לחסוך בעלויות של יצירת תהליכונים נקראת"בריכת תהליכונים" היא פועלת באופן כזה שאנו יוצרים תהליכונים חדשים אם נדרשאבל כאשר הם מסיימים את פעולתם הם אינם מתים אלא פשוט מפסיקים לפעול (כך שהזיכרון שלהם נשמר). ואז, כאשר נדרש שוב ליצור תהליכון, לא ניצור תהליכון חדש מאפס, אלא פשוט נקצה את המשימה לתהליכון שכבר קיים בבריכה שלנו

## פרק 5 – ניהול ותזמון זמן מעבד

הניצול המקסימאלי של זמן המעבד מתממש בשיטת*Multi Programming*. בשיטה זו כל תהליך המתבצע ברגע נתון רץ במחזוריות של"פרצי מעבד/*IO*" מיד עם טעינת התהליך המעבד עובד בתפוקה גבוהה יחסית כדי להפעיל לראשונה את התהליך לאחר זמן קצר, התהליך דורש פעולת קלט/פלט ולכן המעבד כמעט ולא יפעל במצב זה.

כזכור, יש במערכת יומן לטווח קצר או להחליט איזה תהליך ירוץ של יומן המעבד

1. כשתהליך עבר ממצב ריצה למצב המתנה.
2. כשתהליך עובר ממצב ריצה למצב מוכן.
3. כשתהליך עובר ממצב המתנה למצב מוכן.
4. כשתהליך מסיים את פעולתו ומחליט להתאבד.

במצבים 2,3 הפעולה הנדרשת מהיומן היא "הקדמת תרופה למכה".

"יומן המעבד" שאחראי נדרשות בארבעה מצבים:


### מנגנון השילוח -*Dispatcher*

מנגנון זה אחראי להעביר את השליטה על המעבד לתהליך הנמצא בראש התור שיוצר יומן המעבד.דבר זה כרוך במספר פעולות בסיסיות

- החלפת ההקשר– להחליף את ה*PCB* של התהליך הישן בזה של התהליך הנכנס
- החלפה למצב משתמש (באם נדרשת).
- טעינת הפקודה הראשונה האמורה להתבצע מ*P6*.

### זמן תגובה–*Dispatcher Latency*

זמן התגובה הוא הזמן הלוקח למנגנון השילוח לעצור תהליך אחד ולטעון תהליך אחר

### קריטריונים לניהול יומן המעבד

כאשר היומן נדרש להחליט איזה תהליך יעמוד בראש תור הכניסה למעבד עליו לקחת בחשבון מספר גורמים חשובים:

- ייעול *CPU* – לשמור על המעבד בתפוקה גבוהה ככל שנית(ללא זמן מנוחה).
- הספק תהליכים גבוה – השלמת ביצוע של כמה שיותר תהליכים בזמן כמה שיותר קצר
- זמן ביצוע – הזמן הלוקח לתהליך מסוים לבצע את משימתו
- משך ההמתנה – כמה זמן המתין התהליך בתוך המוכנים
- זמן תגובה – הזמן שלוקח משליחת בקשה למחשב ועד לתחילת העבודה עליה

המצב האידיאלי הוא המצב בו פעילות המעבד וההספק מקסימאליים, ואילו זמן ביצוע,משך ההמתנה וזמן התגובה מינימאליים.

## שיטות לניהול יומן

### 1. *First Come, First Served (FCFS)*

בשיטה זו התהליך הראשון שנכנס לתור המוכנים הוא הראשון שיתבצע, ללא התערבות נוספת מצד היומן. בעצם תור קלאסי ללא עדיפויות

יתרון: מימוש קל (וואו. ביג דיל)

חסרון: הספק לא אופטימאלי. מדוע? נניח לדוגמא שיש לנו שלושה תהליכים מוכנים $P_1,P_2,P_3$. משך הזמן הדרוש לביצוע שלהם הוא 24,3,3 מילישניות בהתאמה משמאל לימין כעת נחשב את הזמן הממוצע שכל התהליכים המתינו בתור:

התהליך הראשון לא המתין כלל וביצועו נמשך 24 מילישניות התהליך השני המתין 24 מ"ש וביצועו נמשך 3 מ"ש. התהליך השלישי המתין 27 מ"ש (עד ששני הקודמים יסיימו) ונמשך 3 מ"ש.

זמן ההמתנה הכללי: $0 + 24 + 27 = 51$. כעת נחשב את זמן ההמתנה הממוצע: $\frac{51}{3} = 17$ מילישניות.

לעומת זאת, אם היינו מסדרים אחרת את סדר ביצוע התהליכים ואת התהליך הארוך היינו משאירים לסוף היינו מקבלים תוצאה אחרת לגמרי:

התהליך הראשון לא המתין כלל ונמשך 3 מ"ש. התהליך השני המתין 3 מ"ש ונמשך 3 מ"ש. התהליך שלישי והאחרון המתין 6 מ"ש ונמשך 24 מ"ש. כעת נחשב: משך ההמתנה הכללי: $0 + 3 + 6 = 9$. ולכן זמן ההמתנה הממוצע במערכת הוא: $\frac{9}{3} = 3$ מ"ש בלבד!

### 2. *Shortest Job First (SJF)*

שיטה זו מיישמת את המסקנות שלמדנו מחסרונות השיטה הקלאסית. בשיטה זו העדיפות הגבוהה ביתר ניתנת לתהליכים שבהם משך פרץ ה-*CPU* הבא שלהם הוא הקצר ביותר. כך אנו מבצעים כמה שיותר תהליכים בזמן קצר ככל שניתן. החיסרון של שיטה זו הוא פשוט תהליכים ארוכים עשויים להמתין זמן רב מדי עד לביצוע כי כל הזמן עשויים להיכנס לתור תהליכים קצרים יותר.

בשיטה זו יש שני דרכי פעולה:

- השיטה הבלתי הפיכה (*Non Preemptive*) - ברגע שתהליך התחיל לתבצע במעבד, גם אם נכנס לתור תהליך בעל משך פרץ *CPU* יותר קצר הוא לא יחליף את התהליך המתבצע עד לסיום פרץ ה *CPU* הנוכחי שלו.
- השיטה ההפיכה (*preemptive*) – גם כאשר תהליך מתבצע אם יכנס לתור תהליך אשר משך פרץ ה *CPU* שלו קצר יותר ממשך **הזמן הנותר** לתהליך הנוכחי, התהליך החדש יחליף את התהליך המתבצע. שיטה זו ידועה גם בשם *Shortest Remaining Time First*.

בעיה: שיטה זו מסתמכת על כך שאנו יודעים מראש מה יהיה משך פרץ המעבד הבא של כל תהליך. אבל איך ניתן לדעת מראש דבר כזה? פתרון: אנו מעריכים על סמך פרצי המעבד הקודמים של כל תהליך את משך הזמן המשוער של הפרץ הבא. הנוסחה היא כזאת:

$$t_n = Duration\ of\ the\ n_{th}\ CPU\ burst.$$

$$t_{n+1} = Approximate\ duration\ of\ the\ next\ burst.$$

$$0 < \alpha < 1$$

$$t_{n+1} = \alpha * t_n + (1 - \alpha) * t_n$$

## תורי עדיפויות (על אפליה ועל פז"ם...)

בשיטה זו לכל תהליך משויך מספר שלם המבטא את העדיפות של התהליך בתור. מספר קטן יותר מבטא עדיפות גבוהה יותר. המעבד מוקצה לתהליך עם העדיפות הגבוהה ביותר בתור. שיטה זו יכולה להיות הפיכה או בלתי הפיכה. בשיטה הפיכה המעבד מוקצה **בכל רגע נתון** לתהליך עם העדיפות הגבוהה ביותר. למשל *SJF* היא דוגמא לתור עדיפויות אשר קריטריון העדיפות בה הוא משך פרץ *CPU* הבא של כל תהליך.

הבעיה בתורי עדיפויות היא "הרעבה". קרי, תהליכים עם עדיפות נמוכה עלולים לעולם לא להתבצע (בעיקר בשיטה ההפיכה) כי עלולים להיכנס לתור תהליכים בעלי עדיפות גבוהה יותר כל הזמן.

הפתרון לבעיה זו הוא לתת גיל או פ"ם לכל תהליך. הפז"ם עולה כלל שהתהליך שוהה יותר זמן בתור. הפז"ם משמש מדד בנוסף לעדיפות כדי לקבוע איזה תהליך יכנס לביצוע

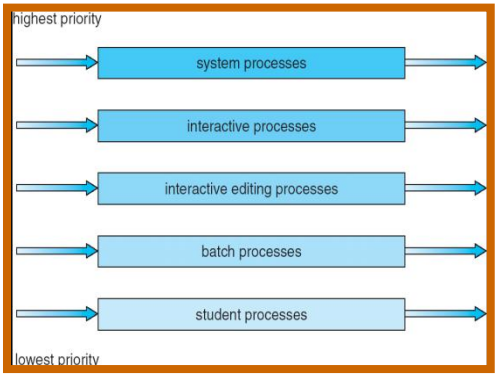
### Round Robin (RR) (תורת הקוואנטים)

בשיטה זו כל תהליך מקבל פרק זמן מעבד קצרצר וקבוע (קוואנט) בד"כ בין 10 ל 100 מ"ש. ברגע שהזמן עבר התהליך מוחלף בתהליך הבא בתור והוא עצמו עובר לסוף התנתניח שישנם $n$ תהליכים בתור והקוואנט שלנו הוא $q$, אזי כל תהליך יקבל $\frac{1}{n}$ מזמן המעבד הכולל, ואף תהליך לא יחכה אף פעם יותר מ $(n-1)q$ מילי שניות.

לכאורה, זהו בעצם תור קלאסי ($FCFS$) עם זמן מוגבל לכל תהליך(בעצם כמו תור למתקן בלונה פארק אבל באירופה ולא בארץ...) וזהו אכן המצב כאשר אנו בוחרים $q$ גבוה יותר מדי, כי ככה כל תהליך מתבצע עד לסיומו לפי סדר ההגעה. לעומת זאת, אם נבחר $q$ קטן מדי יחסית לזמן התגובה של המשלח נקבל מערכת לא יעילה שהרבה זמן מתבזבז בה על תקורהלכן חשוב לבחור $q$ כל שייתן ביצועים מיטביים לזמן תגובת מערכת קצר ככל שניתן, ובו בזמן תפוקת תהליכים גבוהה ככל שניתן

### תורים רבי רמות – Multi Level Queues

בגישה זו ה $Ready\ Queue$ מחולק לכמה תורים בעלי אלגוריתמי ניהול נפרדים




התור הראשון– התור החזית; מנוהל בשיטת $RR$.

התור השני– התור ה"אחורי". מנוהל בשיטת $FCFS$.

כמו כן נדרשת מדיניות לניהול התור של התורים כלומר מי מבין שני התורים יהיה התור הפעיל כרגע

המדיניות יכולה להיות מבוססת $RR$, כלומר הקצאת פרק זמן מסוים לכל תור להריץ את תהליכיו, לדוגמא $80\%$ לתור קדמי ו $20\%$ לתור האחורי; או מבוססת עדיפות, לדוגמא כל עוד ישנם תהליכים בתור הקדמי הוא יהיה התור הפעיל (סבירות להרעבה).

כמובן שניתן לחלק את ה $Ready\ Queue$ ליותר משני תורים בסיסיים, ולתת לכל תור עדיפות משלו

### תרשים גאנט– Gantt Chart

תרשים גאנט הוא צורת הצגה גרפית לתהליך ביצוע התהליכים במעבד. למעשה תרשים גאנט הוא טבלה בעלת ארבעה עמודות, ומספר לא קבוע של שורות כל שורה מציינת תהליך אשר התבצע או עתיד להתבצע בזמן מסוים. בעמודה השמאלית ביותר נמצא זמן התחלה וזמן סיום של ביצוע התהליך. בעמודה השנייה נמצא ה $PID$ של התהליך המתבצע באותה שורה. בעמודה השלישית נמצא זמן ההגעה של התהליך לתוך התהליכים המוכנים. בעמודה הרביעית נמצא משך זמן המעבד הקרוב אותו התהליך דורש. כאשר רוצים לייצג בעזרת תרשים גאנט את שיטת התזמון על פי עדיפות ($Priority$) נידרש להוסיף עמודה נוספת לעדיפות של כל תהליך

כאשר בונים תרשים גאנט יש לקחת בחשבון את שיטת התזמון. למשל אם אנו עובדים ב $FCFS$ הרי שנמיין את התהליכים ע"פ רגע ההגעה שלהם, ונכניס אותם לטבלה ע"פ הסדר, ואז נמלא את השדה של זמן ההתחלה וזמן הסיום של כל תהליך, על פי זמן ההתחלה שלו ומשך הזמן בו הוא רץ

דוגמא: ארבעה תהליכים –

$$P_1 = \{PID = 1, CPU\ Burst = 10, Arrival\ Time = 37\}$$
$$P_2 = \{PID = 2, CPU\ Burst = 23, Arrival\ Time = 24\}$$
$$P_3 = \{PID = 3, CPU\ Burst = 17, Arrival\ Time = 3\}$$
$$P_4 = \{PID = 4, CPU\ Burst = 43, Arrival\ Time = 15\}$$

תרשים גאנט של תהליכים אלו בשיטת $FCFS$ יראה כך:

| Time | PID | Arrival Time | CPU Burst Time |
|---|---|---|---|
| 3-20 | 3 | 3 | 17 |
| 20-63 | 4 | 15 | 43 |
| 63- 87 | 2 | 24 | 23 |
| 87 - 97 | 1 | 37 | 10 |

### תורים רבי רמות עם משוב– *Feedback MLQ's*

תורים בעלי משוב הם פיתוח של תורים רבי רמות, אשר תהליך יכול לנדוד במהלך חייו בין הרמות השונות

תור משוב חייב לקיים את כל התנאים הבאים

1. הוא תור רב רמות
2. מכיל אלגוריתם ניהול לכל רמה
3. מכיל אלגוריתם לניהול התור הכללי, כלומר איזו רמה תשלח תהליך לביצוע
4. מכיל אלגוריתם הקובע מתי תור עולה רמה ומתי תור יורד רמה

דוגמא: תור תלת רמתי-

רמה א' – בשיטת *RR* עם קוואנטום של 8 מ"ש.

רמה ב' – בשיטת *RR* עם קוואנטום של 16 מ"ש.

רמה ג' - בשיטת *FCFS*.

מדיניות – תור חדש נכנס לרמה הראשנה ומחכה לתורו בתורו הוא נכנס למעבד ל 8 מ"ש. אם הוא עדיין לא סיים הוא עובר לרמה השנייה גם שם הוא ממתין לתורו ומקבל 16 מ"ש נוספות אם הוא עדיין לא סיים הוא עובר לרמה השלישית בין הרמות ישנה עדיפות כלומר כל רמה א' מתבצעת לפני רמה ב' וכן הלאה.

### ניהול תהליכים בסביבה רבת מעבדים

ניהול תורי תהליכים בסביבה רבת מעבדים הוא מעט יותר מסובך מאשר בסביבה עם מעבד אחד, דבר זה נובע מהעובדה שכעת נדרש גם לתזמן את ביצוע העבודות בין המעבדים השונים (כאשר יש תלות בין תהליכים שונים) וכן לנהל את חלוקת המשאבים והמשימות בין כל המעבדים.

מבחינים בין שני סוגי סביבות:

1. סביבת מעבדים הומוגנית, בה כל המעבדים זהים בתפקודם ותפקידם
2. סביבה אסימטרית, בה יש מעבד מרכזי אחד בעל גישה לזיכרון ולנתונים, הוא מנהל ומחלק את העבודה בית תתי המעבדים האחרים

### ניהול זמן אמת - *Real Time Scheduling*

זמן אמת היא מושג המתאר מערכת המבצעת פעולות כמעט ללא השהייה, אפילו לא של מילי שניות בודדות

ישנן שתי סוגי מערכות זמן אמת:

1. זמן אמת נוקשה – *Hard RT* – מבטיח ביצוע משימות קריטיות תוך פרק זמן מיידי
2. זמן אמת גמיש – *Soft RT* - מבטיח שמשימות קריטיות יבוצעו בעדיפות גבוהה על חשבון שאר המשימות

# פרק 6 – סינכרוניזציה של תהליכים

## רקע

גישה בו זמנית של תהליכים שונים למידע משותף עלולה להוביל לאי אמינות המידע עקב שינוי ע"י תהליכים אחרים. לכן עלינו למצוא דרך שתבטיח ביצוע של תהליכים במקביל יחד עם שמירה על אמינות המידע בכל רגע נתון בביצוע כל אחד מהתהליכים

## מקרה הצרכן – יצרן, ובעיות קטעים קריטיים

ישנם שני תהליכים הרצים במקביל תהליך המייצג יצרן המייצר "משהו", ותהליך המייצג צרכן הצורך את אותו "משהו"

כל המוצרים נמצאים במערך הנקרא "חוצץ" בעל מספר מקומות ידוע וקבוע

תהליך היצרן:

בצע תמיד:

{

כל עוד (מונה == חוצץ) // בדיקה האם החוצץ מלא במוצרים

{ // ממתין ולא עושה כלום

}

חוצץ במקום [$in$] = עוד מוצר. // יצור מוצר נוסף

$In = (in + 1)\% \, Buffer\ Size$. // קידום המקום הבא לייצור.

מונה++. // מספר המוצרים בחוצץ

}

תהליך הצרכן:

בצע תמיד:

{

כל עוד (מונה == 0) // בדיקה אם אין מוצרים לצרוך

{ // בצע כלום והמתן

}

צורך את המוצר בחוצץ במקום [$out$].

$out = (out + 1)\% \, Buffer\ Size$. // קידום האינדקס למוצר הבא לצריכה

מונה --. // הורדת מספר המוצרים בחוצץ ב 1

}

נשים לב כי לשני התהליכים משתנה משותף – מונה. כעת ננסה להבין כיצד מתבצעות פעולות קידום וחיסור המונה בתוך המעבד בצורה סימבולית

קידום:

*register1 = count*

*register1 = register1 + 1*

*count = register1*

והחיסור:

*register2 = count*

*register2 = register2 - 1*

*count = register2*

כעת נבין שאם המעבד מקצה רק זמן מסוים לכל תהליך הרי יכול להיות שהוא יבצע רק את פעולת הקידום או החיסור ללא הצריכה או היצור בפועל, ומיד יעבור לתהליך השני, וכך יקרה מצב בו המונה מכיל נתון שקרי, שהרי לא יוצר או נצרך שום דבר למרות שהמונה השתנה

### תנאים הכרחיים לפתרון

על כל פתרון לבעיה זו לקיים כמה תנאים

1. מניעה הדדית - כאשר תהליך כלשהוא מבצע"קטע קריטי", אף תהליך אחר לא יכול להתבצע עד לסיום הקטע הקריטי. *Mutual Exclusion*
2. התקדמות - אם אין אף תהליך באמצע קטע קריטי אבל יש תהליכים הממתינים להיכנס לקטע כזה, אזי צריך לדאוג שאותם התהליכים יתבצעו כמה שיותר מהר. *Progress*
3. המתנה חסומה - *Bounded Waiting*. אם תהליך מבקש משאב, רק מספר סופי של תהליכים יוכלו לקבל את המשאב(קטע קריטי) לפניו.

### אלגוריתם פיטרסון–*Peterson*

אלגוריתם זה מוגבל לשני תהליכים אשר מתחלפים כל פעם בביצוע הקטע הקריטי שלהם.
התהליכים חולקים שני משתנים לפחות:

א. מערך בוליאני *Flag* בגודל 2 תאים כאשר הערך בכל תא מציין אם תהליך בעל מספר התא הזה עומד ומוכן להיכנס לקטע קריטי.
ב. משתנה מסוג *int* הנקרא *Turn* שהוא או 1 או 2 ומציין את מספר התהליך שתורו להיכנס לקטע הקריטי.

האלגוריתם לתהליך מספר *i*:

בצע:
{
*Flag* במקום [*i*] = *true*.
*j* = *Turn*.
כל עוד (*Turn* == *j* וגם *Flag* במקום [*j*] == *true*)
{ לא תורך, פשוט חכה בשקט}
// ביצוע הקטע הקריטי
*Flag* במקום [*i*] = *False*.
// שאר הקוד
}

מדוע זה מבטיח את שלושת התנאים?

1. לא יתכן מצב בו שני התהליכים נמצאים בקטע הקריטי: כי כיצד יתכן מצב בו שני התהליכים יבצעו את הקטע הקריטי ולא יתקעו בלולאת המתנה? אם בשני התאים במערך יהיה ערך "שקר". אך מצב כזה אומר ששניהם לא מוכנים להיכנס לקטע הקריטי כי אחרת הדגל שלהם היה "אמת"! לכן אין בעיה.
2. לא יתכן מצב בו שני התהליכים תקועים בלולאת ההמתנה, כי המשתנה "תור" (*Turn*) יכול להיות או 0 או 1 ולא שניהם בו זמנית לכן רק עבור אחד מהתהליכים תנאי ההמתנה יכול להתקיים בכל רגע נתון.
ובזמן שתהליך אחד ממתין, התהליך השני מבצע את הקטע הקריטי אשר בסופו הוא ישחרר את התהליך הממתין, כך שהוא לא יחכה לנצח. וכך פתרנו את שני התנאים האחרונים

### מה הקטע שלך??

פתרון המבוסס על רעיון זה נקרא"פתרון מבוסס מנעול" כלומר, על תהליך המבקש להיכנס לביצוע קטע קריטי להשיג "נעילה" מפני שאר התהליכים, ולשחרר אותה מיד כאשר הוא מסיים את ביצוע הקטע שלו.

### סנכרון בעזרת חומרה

הרבה מערכות מספקות פתרון חומרתי לבעיית קטע קריטי: במערכות עם מעבד אחד ניתן למשל למנוע פסיקות במהלך הביצוע וכך להבטיח שאף תהליך לא יכנס באמצעגישה זו ממומשת במערכות הפעלה בעלות ליבה ללא אפשרות"חטיפה" (*Preemption*). במערכות חדישותיש תמיכה בהוראות מורכבות כך שיתבצעו בפקודת מעבד אחת בלבד או בכמה פקודות ללא אפשרות הפרעה ביניהם

### סמאפור – *Semaphore*

מכיוון שהגדרה ידנית של מנגנוני נעילה ע"י המתכנת בכל פעם שיש קטע קריטי היא מסורבלת למה, ניתן להשתמש במנגנון תזמון הנקרא"סמאפור". סמאפור *S* הוא משתנה מסוג *int* אשר למעט פעולת האתחול שלו, ניתן לשנות אותו ע"י שתי מתודות"אטומיות": *Wait(S)* ו *Signal(S)*.

מימוש *WaitS)*:

{

כל עוד (*S* >= 0) המתן בשקט.

*S--;*

{

מימוש *Signal (S)*.

}

*S++;*

{

אנו מניחים ששני פונקציות אלה מתבצעות בצורה אטומית, כלומר בלתי ניתנת לחלוקה או לחטיפה מבחינים בין שני סוגי סמאפור שונים:

1. סמאפור בינארי – *S* מקבל ערך שהוא או 0 או 1. סמאפור זה נקרא גם *Mutex Lock*. הוא מאפשר להבטיח שרק תהליך אחד ישתמש במשתנה משותף ברגע נתון – *Mutual Exclusion*.
2. סמאפור מנייה – מקבל טווח רחב של ערכים משמש לניהול תהליכים הניגשים למספר סופי של משאבים. הוא מאותחל עם מספר המשאבים הזמינים במערכת

### אופן שימוש בסמאפור בינארי

נגדיר סמאפור בשם *Mutex* המאותחל ל 1.

כל תהליך המשתמש בו יראה כך:

{

*Wait (Mutex);* // השגת נעילה.

//ביצוע הקטע הקריטי

*Signal (Mutex);* // שחרור נעילה.

// ביצוע שאר הקוד

{

### אופן השימוש בסמאפור מנייה

כל תהליך הרוצה להשתמש במשאב חייב קודם לבצע פעולת *Wait*. פעולה זו תוריד את מספר המשאבים הזמינים ב 1 כי התהליך צורך משאב אחד. אחרי סיום הפעולה שלו התהליך ישחרר את המשאב באמצעות *Signal*. כאשר כל המשאבים תפוסים ערך הסמאפור יהיה 0 וכך תהליך שיבקש משאב יתקע בלולאת ההמתנה בפונקציה ה *Wait*. (כל עוד סמאפור <=0).

{

*Wait (Mutex);* // השגת משאב.

//שימוש במשאב

*Signal (Mutex);* // שחרור משאב.

// ביצוע שאר הקוד

{

### חיסרון

החיסרון הגדול של שיטת הסמאפור(בעיקר הבינארי) הוא שכאשר תהליך מסוים נמצא בתוך קטע קריטי, כל תהליך אחר המבקש להיכנס לא יידחה על הסף, אלא פשוט ירוץ בלולאת המתנה בכל משך הזמן המוקצה לפרץ הנוכחי שלו(בכל משך הקוואנטום של). דבר זה גורם לבזבוז רב של זמן מעבד יקר, וכאשר מדובר בהמון תהליכים המשתמשים באותו סמאפור, מתרחשת למעשה תופעה בה המעבד עסוקרוב הזמן בלהריץ לולאות המתנה במקום לבצע תהליכים(אמנם בכך חסכנו את משך הזמן הדרוש להחלפת תהליכים – *Context Switch* אב את הזמן שחסכנו בזבזנו בהמתנה..)

**פיתרון**

את בעיית ההמתנה ניתן לפתור אם נוכל לגרום שכאשר תהליך נתקל בלולאת ההמתנהוא לא יבצע אותה אינספור פעמים, אלא יעביר את זכות השימוש במעבד לתהליך אחר

ניתן לבצע את הרעיון הזה ע"י שינוי קל במבנה הסמאפור: מעתה הסמאפור יהיה מבנה המכיל שני ערכים

א. משתנה מסוג *int* שהוא בעצם הסמאפור הישן

ב. רשימה של תהליכים בהמתנה לאותו הסמאפור

הגדרת הסמאפור החדש:

*Class Semaphore*

*}*

*Int value;*

*List<process> queue;*

*{*

כעת נגדיר גם שתי פעולות נוספות המבצעות שינויים בסמאפור: פעולת *Block()* ופעולת *Wakeup(process p)*. פעולות אלו משפיעות על הרשימה של התהליכים הממתינים: הפונקציה *Block* נקראת מתוך הפונקציה *Wait* ותפקידה להכניס את התהליך הקורא אל התור של אותו סמאפור ולהעביר אותו ממצב "ריצה" למצב "המתנה", ואז להעביר את הפיקוד אל ה *CPU Scheduler* שיקרא לתהליך הבא הפונקציה *Wakeup* נקראת מתוך הפונקציה *Signal* ותפקידה "לעורר" תהליך שהיה בהמתנה לסמאפור ולהוציא אותו ממצב ההמתנה למצב "מוכן".

הפונקציות החדשות יראו כך:

*Wait (Semaphore *S)*

*}*

**S.Value--;*

אם *((*S).Value < 0 )*

*}*

העבר את התהליך אל *(*S).queue;*

*Block () ;* // מעביר את התהליך למצב ממתין

*{*

*{*

ופעולת *Signal*:

*Signal (Semaphore *S)*

*}*

*(*S).Value ++;*

אם *((*S).Value <=0)*

*}*

הוצא את תהליך *P* מהתור *(*S).queue;*

*Wakeup (P);*

*{*

*{*

עדיין חשוב לזכור שעלינו להבטיח ששתי הפעולות הבסיסיות של הסמאפור תהינה אטומיות, כלומר בלתי ניתנות לעצירה ברגע שהתחילו

# פרק 7 - מבוי סתום / קיפאון (Deadlock)

## הקדמה

במערכות רב-תוכניתיות (Multi Programming) יתכן מצב בו תהליך אחד או יותר מתחרים על מספר משאבים מסוים. כאשר תהליך מבקש משאב אך המשאב אינו זמין כרגע, התור עובר ממצב ריצה למצב המתנה עד שישתחרר המשאב הדרוש לו. לעיתים יתכן שהתהליך הנ"ל לעולם לא ייצא מתור ההמתנה, כי המשאב הדרוש לו מוחזק ע"י תהליך אחר, אשר גם הוא בהמתנה.

מצב כזה נקרא "מבוי סתום" או באנגלית Deadlock.

דוגמא מציאותית: (חוק שעבר בקנזס בתחילת המאה שעברה) *"אם שתי רכבות מתקרבות לצומת מכיוונים שונים, על שתיהן להאט מייד עד לעצירה מוחלטת, ולהמתין עד שהרכבת השנייה תפנה את הצומת"*

לכל מערכת יש מספר מוגבל של משאבים, המתחלקים לסוגים שונים. יתכנו כמה משאבים זהים מאותו סוג, למשל שני מעבדים, שני כוננים קשיחים, 4 יציאות USB וכ"ו. כל "הופעה" אחת של משאב מסוג מסוים נקראת "עותק".

תהליך משתמש במשאבי מערכת באופן הבא:

א. בקשה. התהליך מודיע למערכת על נחיצותם של כמה משאבים מסוימים לצורך השלמת המשימה שלו. אם הבקשה אינה מאושרת במיידי, על התהליך לעצור ולהמתין עד לשחרור המשאבים הדרושים.

ב. שימוש. צריך לכתוב שטויות כדי שלא תהיה שורה ריקה מה?!.

ג. שחרור. על התהליך להודיע למערכת שהמשאבים אינם נחוצים לו עוד, חשוב שיתן להפנותם לתהליכים אחרים.

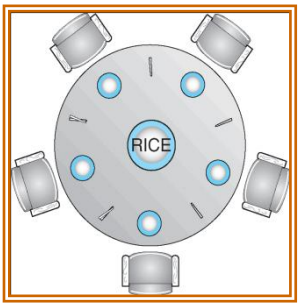
הבקשות והשחרורים הם קריאות מערכת (System Calls). על המערכת להחזיק ולנהל טבלה בה מפורטים כל המשאבים הקיימים במערכת, מה מצב כל משאב, ואם משאב מסוים תפוס לשמור נתוני התהליך המשתמש.

## תנאים הכרחיים לקיפאון

כדי שיתרחש מבוי סתום דרושים להתקיים ארבעת התנאים הבאים בו זמנית:

1. שיתוק הדדי – (Mutual Exclusion). כלומר, לפחות משאב אחד חייב להיות במצב בו הוא אינו זמין לשימושם של עוד תהליכים, כלומר אם תהליך מסוים מחזיק אותו, הרי שאף תהליך אחר לא יכול להשתמש בו באותו הזמן.
2. "החזק והמתן" – (Hold and Wait). על כל תהליך תקועה להחזיק לפחות משאב אחד, ולהזדקק לפחות לעוד משאב אחר המוחזק ע"י תהליך אחר.
3. אין חטיפה. כלומר, ברגע שתהליך קיבל משאב, אין אפשרות לקחת אותו ממנו אם הוא לא הודיע על שחרור.
4. המתנה מעגלית. כלומר כדי שקבוצת תהליכים תהיה במבוי סתום עליה לקיים את התנאי הבא: על התהליך הראשון לחכות למשאב שמחזיק תהליך X, שממתין למשאב שמחזיק תהליך Y, שממתין..... למשאב שמחזיק התהליך הראשון.

## בעיות נפוצות




א. "הפילוסופים הסועדים" (WTF!?!?)

בעיה זו מתארת חמישה סועדים סביב שולחן עגול. מול כל סועד ישנה צלחת. מימין לכל סועד ישנו מקל אכילה (הזה של הסינים). תתעלמו כרגע מהעובדה שהפילוסופים היו ביוון.) אבל כדי לאכול, כל סועד חייב להחזיק בשני מקלות בו זמנית, כלומר עליו להשיג גם את המקל משמאלו. לסיכום: ישנו משאב הנקרא "מקל". יש ממנו חמישה עותקים ישנם חמישה תהליכים אשר כל אחד דורש שני מקלות בכל פעם.

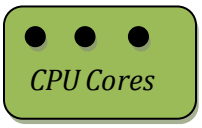
זוהי דוגמא קלאסית למבוי סתום מעגלי שכן כל סועד חייב לחכות למקל שמשמאלו, ובינתיים הוא מחזיק את המקל שלו שלימינו, כך שהמערכת תקועה למדי.

## גרף הקצאת משאבי מערכת

דרך ויזואלית נוחה לתיאור וניתוח מצבי מבוי סתום היא לצייר גרף המבטא את מצב משאבי המערכת והתהליכים הרצים ברגע מסוים. הגרף מורכב מ"קשתות" שהם בעצם חיצים, ומ"תאים" המבטאים או תהליכים בצורת שהם בעיגול, או משאבים שהם בצורת ריבוע עם נקודותכמספר המופעים של אותו סוג המשאב. חץ היוצא מתהליך למשאב מציין שהתהליך מבקש וממתין למשאב הזה חץ הפוך היוצא ממשאב אל תהליך מבטא שהמשאב הזה מוחזק ע"י התהליך שהחץ מורה עליו

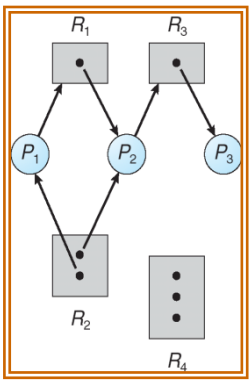
מקרא:

תהליך -

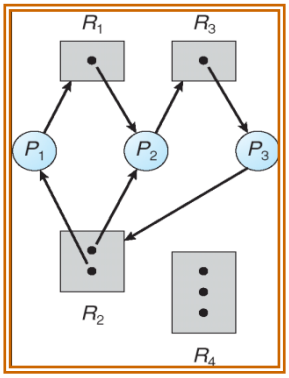
משאב- 


בקשה/החזקה - ←

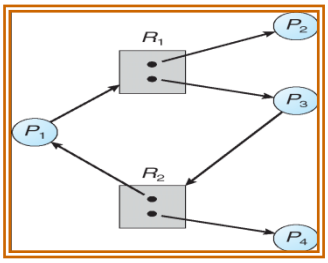
דוגמא לגרף:




דוגמא לגרף עם מבוי סתום:




הסבר: נסתכל בגרף עם המבוי הסתום ניתן לראות שיש מעגלסגור שלחיצים (כולם באותו הכיוון) כלומר, כל תהליך צריך משאב הדרוש לתהליך אחרוהוא עצמו מחזיק משאב הדרוש לתהליך אחר במעגל אך יתכנו מצבים בהם יש מעגל אבל עדיין לא יהיה מבוי סתום(ראה ציור), מכיוון שיש תהליכים המחזיקים משאבים הנמצאים במעגל, אך הם עצמם לא בתוכו (תהליכים $P2$ ו $P4$) ולכן הם עתידים לשחרר את המשאבים שלהם בקרוב




## לסיכום,

בהינתן גרף משאבי מערכת נתבונן בתנאים הבאים

האם יש מעגל סגור?

אם לא, בוודאי שאין מבוים סתומים

אם כן נשאל את עצמנו:

האם יש רק מופע אחד מכל משאב?

אם כן אז המבוי סתום.

אם לא נשאל:

האם כל המופעים מוחזקים ע"י תהליכים הנמצאים במעגל?

אם כן יש מבוי סתום.

אם לא, לא.

### דרכי התמודדות עם קיפאון(חוץ ממזגן ומעיל)

כאשר אנו מתכננים מערכת שתתמודד עם איום הקיפאון עלינו להתייחס לשלושה מרכיבים:

א. מניעה – למה להיחלץ מבעיה כשאפשר לא להיכנס אליה? ניתן למשל לדרוש שתהליך יקבל את כל המשאבים שהוא צריך בבת אחת כאשר הם יתפנו, ווכך למנוע מצב "החזק והמתן". הסיכון בשיטה כזאת הוא ניצולת משאבים נמוכה וסכנת הרעבה
כמו כן ניתן לנתח תהליכים לפני שהם רצים כדי לנסות לחזות את הדרישות שלהם מראש

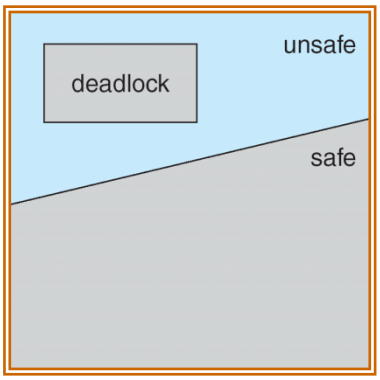
ב. אבחון מוקדם – עלינו לדאוג שאם וכאשר יקרה מצב של קיפאון במערכת, המערכת תדע על כך כדי שתוכל לטפל בו בהקדם

ג. החלמה / טיפול - (זה כבר מתחיל להישמע כמו איידס וזה מטריד, עמכם הסליחה) כאשר גילינו שיש קיפאון, צריך "להפשיר" אותו. לשם כך יש לנו כמה אפשרויות:

1. "היטלר" - אנו יכולים פשוט לחסל את כל התהליכים הנמצאים במעגל הקיפאון, כמו השיטה הנאצית, שיטה זו מאוד יעילה כדי לצאת מהקיפאון, אבל סבירות גבוהה לנזקי חיסלנו תהליכים בלי להשלים את המשימה שלהם ובלי לשמור את המידע שהם עיבדו עד כה
2. "דקסטר" – נתחיל לחסל תהליכים אחד– אחד, עד לשחרור המעגל. פחות נזק ואיבוד מידע על חשבון משך הזמן שייקח לצאת מהקיפאון, כמו כן צריך אלגוריתם שיקבע את מי מהתהליכים עדיף לחסל ואת מי עדיף להשאיר, ידוע גם בתור "הקוד האתי של דקסטר".
(למשל עדיף להימנע ככל שניתן לחסל את התהליך "svchost.exe". אני לא יודע בדיוק למה, אבל זה נורא ירגיז את ווינדוס)
3. "סטאלין" – הלאמת והחרמת משאבים. רגע, אמרנו שברגע שתהליך מחזיק משאב, אי אפשר לקחת לו אותו עד שהוא יודיע על שחרור!! Well... ברוכים הבאים לקומוניזם במערכות שמאפשרות הפקעה (Preemption) ניתן במקרה הצורך פשוט לקחת לתהליך את המשאב בו הוא מחזיק, ולתת אותו לתהליך תקוע, בדומה לשיטת "דקסטר", אנו מבטיחים נזק מינימאלי על חשבון הזמן שייקח למערכת להיחלץ. ניתן להקטין את הנזק עוד יותר על ידי כך שנקים מנגנון משטרה חשאית או בשמה המוכר "KGB", שתפקידה לעקוב אחרי תהליכים, וליצור עבורם "נקודות שחזור", אשר התהליך יחזור אליהן כאשר יופקע ממנו משאב שהוא החזיק, וכך יחזור על הצעדים שהוא ביצע לאחרונה שוב, כדי לנסות לשחזר את המצב בו הופקע ממנו המשאב.
החיסרון הגדול בשיטה זו הוא שאחרי כמה זמן אתם תתחילו לדבר רוסית (לא באמת).

ד. התעלמות – כן, גם בריחה זה סוג של התמדדות! האמת שכאשר המחיר של שלושת השיטות הדיקטטוריות לעיל, כל כך יקר במונחים של יעילות וסיבוכיות, אין פלא שהפתרון היעיל באמת הוא לעיתים להתעלם. וזה מה שעושות רוב מערכות ההפעלה המודרניות

### דרכי הימנעות




כדי שמערכת תוכל להימנע ממצבי קיפאון, עליה להחזיק מידע מסוים על התהליכים עוד לפני שהם מתחילים לרוץ, בדרך הנפוצה ביותר, נדרש מכל תהליך להצהיר מראש על מספר וסוג המשאבים המקסימאלי שהוא עשוי להידרש להם בהתקבל בקשה מתהליך על משאבים, על המערכת לבחון את המשאבים שהוא דורש לעומת המשאבים הזמינים ולהחליט האם ניתן להקצות לו אותם ללא סכנת קיפאון או שמא עליו להמתין.

עבור קבוצת תהליכים $\{P_1, P_2 \dots P_n\}$, תהליך $P_i$ נחשב "בטוח" אם כל המשאבים שהוא צריך זמינים במערכת, או מוחזקים ע"י התהליכים $\{P_1, P_2 \dots P_{i-1}\}$. כך בצורה רקורסיבית מובטח שכל תהליך יקבל בשלב זה או אחר את כל המשאבים הדרושים לו, והיה אם לא ניתן לספק לו את כל המשאבים, התהליך ימתין.

## אלגוריתם הבנקאי– The Banker's Algorithm

אלגוריתם הבנקאי נמצא בשימוש בהם יש למערכת יותר מעותק אחד למשאבים מסוימים
כל תהליך הנכנס למערכת מצהיר מראש מה מספר העותקים המקסימאלי הדרוש לו מכל משאב
על המערכת לבחון את המשאבים שהוא דורש ולבדוק האם הקצאה של משאבים אלו תשאיר את המערכת במצב בטוח. אם כן, המשאבים מוקצים. אם לא, התהליך ימתין.
למימוש אלגוריתם זה נדרש להחזיק כמה מבני נתונים

- מערך משאבים זמינים – *Available*. כל תא במערך מייצג משאב אחר. המספר בתוך התא מבטא את מספר העותקים הזמינים מאותו המשאב
- מטריצת דרישות מקסימאליות – *Max*. מטריצה בגודל $n X m$ (כאשר $n$ מספר התהליכים, $m$ מספר המשאבים). בטבלה זו מופיעים עבור כל תהליך הדרישות המקסימאליות מכל משאב
- מטריצת הקצאות נוכחיות – *Allocated*. מטריצה בגודל $n X m$ (כאשר $n$ מספר התהליכים, $m$ מספר המשאבים). במטריצה זו מופיע עבור כל תהליך כמה עותקים מכל משאב מוקצים לו כרגע

מטריצת צרכים – *Need*. מטריצה בגודל $n X m$ (כאשר $n$ מספר התהליכים, $m$ מספר המשאבים).
במטריצה זו מופיע עבור כל תהליך אילו וכמה עותקים הוא צריך כדי להשלים את המשימה שלו(מלבד אלו שכבר מוקצים לו– ע"פ מטריצת ההקצאות הנוכחיות כמובן שעל סכום המטריצות (צרכים, הקצאות) להיות נמוך ממטריצת הדרישות המקסימאליות בכל רגע נתון). מטריצה זו נוצרת מחיסור המטריצות– $Need = Max - Allocated$.
כעת נעבור למימוש האלגוריתם
נגדיר יחס $\leq$ בין שני וקטורים באותו אורך יהיו $X,Y$ וקטורים באורך $n$. אומרים ש $X \leq Y$ אמ"מ מתקיים שעבור כל $0 < i \leq n$: $X[i] \leq Y[i]$. כלומר תא ב $X$ קטן מהתא המקביל לו ב $Y$.
הקטע הבא מגדיר האם מערכת קיימת נמצאת במצב בטוח נגדיר מערך עזר בוליאני עם אורך כמספר התהליכים הקיימים, מאותחל כולו ל *False*. למערך נקרא *didFinish* וערך כל תא $i$ מציין האם תהליך מספר $i$ סיים לרוץ או שלא (לכן האתחול ל *false*).

חפש את ה $i$ שעבורו:
{
אם ($didFinish[i] == false$ וגם $Need[i] \leq Available$) // שימו לב ש $Need[i]$ הוא וקטור.
{
ניתן לספק את הצרכים של תהליך זה כרגע התהליך ירוץ עד שיסיים ואז ישחרר את המשאבים שלו:
$Available += Allocated[i]$. המשאב שחרר את כל המשאבים שהיו לו
$didFinish[i] = true$.
}
} כל עוד שקיים $i$ כזה.

המערכת במצב בטוח אם בסוף התהליך כך מערך *didFinish* יהיה אמת, כלומר יש דרך בה כל התהליכים מקבלים את המשאבים הדרושים להם
כעת נתאר את האלגוריתם המקבל בקשה חדשה למשאבים מתהליך, ובודק האם אפשר להיענות לה ולהישאר במצב בטוח או לא
יהי *Request* וקטור המכיל את הבקשה של תהליך כל תא $i$ מכיל מספר שמייצג את מספר העותקים ממשאב $i$.

אם $(Request > Need)$ זרוק חריגת דרישות // התהליך חרג מהדרישות מקסימום עליהן הצהיר
אחרת:
}
אם $(Request > Available)$ אין מספיק משאבים, התהליך ימתין.
אחרת:
}
מבצע סימולציית הקצאה:
$Available -= Request$.
$Process \rightarrow Allocated += Request$.
$Process \rightarrow Need -= Request$.
אם (המצב החדש המתקבל הוא מצב בטוח) מקצה את המשאבים בפועל.
אחרת: התהליך ימתין והמערכת תחזור למצב הקודם
{
{

דוגמא:

נתונה המערכת הבאה

| | Allocated | | | Need | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|
| | $A$ | $B$ | $C$ | $A$ | $B$ | $C$ | $A$ | $B$ | $C$ |
| $P_0$ | 0 | 1 | 0 | 7 | 4 | 3 | 3 | 3 | 2 |
| $P_1$ | 2 | 0 | 0 | 1 | 2 | 2 | | | |
| $P_2$ | 3 | 0 | 2 | 6 | 0 | 0 | | | |
| $P_3$ | 2 | 1 | 1 | 0 | 1 | 1 | | | |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 1 | | | |

המערכת כרגע במצב בטוח. כעת תהליך $P_1$ מבקש: *Request=(1,0,2)*. התנאים הראשון והשני מתקיימים כי $(1,0,2) \leq (3,3,2)$ && $(1,0,2) \leq (1,2,2)$. לכן ניתן לבצע סימולציית הקצאה. נעדכן את הטבלה בהתאם:

| | Allocated | | | Need | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|
| | $A$ | $B$ | $C$ | $A$ | $B$ | $C$ | $A$ | $B$ | $C$ |
| $P_0$ | 0 | 1 | 0 | 7 | 4 | 3 | 2 | 3 | 0 |
| $P_1$ | 3 | 0 | 2 | 0 | 2 | 0 | | | |
| $P_2$ | 3 | 0 | 2 | 6 | 0 | 0 | | | |
| $P_3$ | 2 | 1 | 1 | 0 | 1 | 1 | | | |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 1 | | | |

האם כעת המצב בטוח? כן, כי המערכת יכולה להיענות לצורכי התהליכים בסדר הבא לדוגמא $P_1, P_3, P_0, P_4, P_2$. לכן ההקצאה חוקית והיא תתבצע פועל

### מנגנון מבוסס חומרה–Watch Dog

דרך נפוצה נוספת למניעת מבוים סתומים היא"כלב השמירה". מנגנון זה הוא מנגנון חומרתי; כלומר נדרשים רכיבים פיזיים כדי לממש אותולכל התקן במערכת מוקצה מעין טיימר, אשר מאותחל מדי פעם לערך גבוה ע"י התהליכים המשתמשים באותו ההתקן.אם הטיימר מגיע לאפס, כלומר אף תהליך לא אתחל את הטיימר של ההתקן זמן רב מדי, ככל הנראה כתוצאה ממבוי סתום ההתקן הנ"ל מכבה ומדליק את עצמו מחדש כך שמעגל המבוי הסתום יישבר ואף אחד מהתהליכים לא יחזיק בו יותרמנגנון זה יכול לעבוד במקביל לאלגוריתמים אחרים מבוססי תוכנה.אבל הוא מצריך שיתוף פעולה של התהליכים- באתחול הטיימר תוך כדי פעולתם אם רוצים מנגנון יותר מדוייק אשר ידע בדיוק מי התהליך התקוע כדי להרוג אותו, ניתן להקצות טיימר לכל תהליך(עלות גבוהה מול יעילות).

### אבחון מבוי סתום

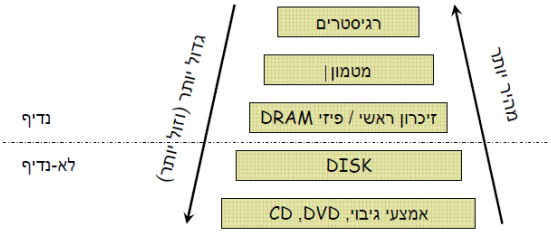
במערכות מהן לא מונהגת שיטה למניעה של מבוי סתום, פעולה הצורך להתמודד איתו אם וכאשר הוא יתרחש. לשם כך נדרש מנגנון המאפשר למערכת להבין שישנה קבוצת תהליכים הנמצאת במבוי סתום, על המנגנון הנ"ל לבצע שני משימות עיקריות:

1. לבחון את המערכת באופן קבוע ולקבוע האם קיים מבוי סתום
2. לספק אלגוריתם ליציאה מהמבוי הסתום (כאמור, גם התעלמות היא דרך לגיטימית).

## פרק 8 – ניהול זיכרון

### מבוא

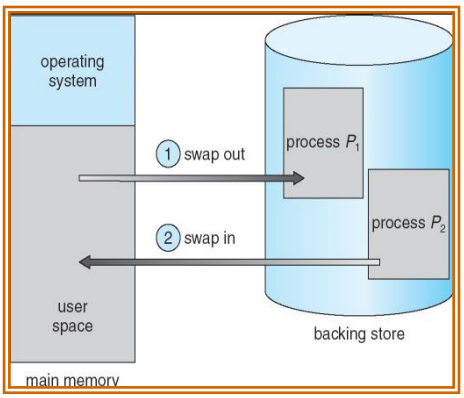
תפקידה של כל מערכת מחשב היא להריץ תוכניות, כדי שתוכנית תוכל לרוץ היא חייבת להיות בזיכרון הראשי (RAM). כדי לנצל בצורה מרבית את המעבד ולקצר את זמן התגובה, בד"כ רצים מספר תהליכים במערכת, ולכן הם גם נמצאים מזיכרון הראשי.
קיימות שיטות רבות המשקפות גישות שונות לניהול זיכרון, כאשר היעילות של כל אחת משתנה בהתאם לנסיבות ולמטרות המערכת.




### תפקידי מערכת ההפעלה

תפקידה של מערכת ההפעלה בניהול הזיכרון מתבטא באופנים הבאים:

- להקצות באופן קבוע חלק מהזיכרון למערכת ההפעלה עצמה
- לחלק את הזיכרון הנותר בין שאר התהליכים הרצים כך שתהליכים לא יגעו בזיכרון של תהליך אחר.
- להקצות זיכרון לתהליך חדש
- לשחרר זיכרון של תהליך שסיים לרוץ




### *Swapping*

כאשר תהליך מסיים לרוץ מערכת ההפעלה מעבירה אותו מהזיכרון הראשי אל הדיסק הקשיח כאשר תהליך מופעל על מערכת ההפעלה להביא אותו מהדיסק הקשיח אל הזיכרון הראשי פעולה זו נקראת *Swapping*.
כזכור, לכל תהליך יש את מרחב הכתובות שלו הכולל את הקוד, נתונים, מחסנית ועוד. האם כל תהליך באמת צריך את כל הזיכרון הזה? בעיקרון התשובה היא כן, אבל לא בזיכרון הראשי!

### עיקרון הלוקאליות

בכל רגע נתון, תהליך משתמש רק בחלק מאוד זעיר מהזיכרון העומד לרשותו ואם תהליך השתמש בנתון מסוים סביר להניח שהוא ישתמש גם בנתונים הסמוכים לו בזיכרון.
בגלל עיקרון זה מערכת הזיכרון בנויה בצורה היררכית ובגללו אנחנו לא נחזיק את כל בלוק הכתובות של תהליך בזיכרון הראשי, אלא נביא ממנו חלקים מהדיסק הקשיח אל הזיכרון.
לשם כך עלינו ליצור מנגנון שינהל ויבקר פעולה זו ועליו להתייחס לכמה פרמטרים:

- מה נמצא בזיכרון הראשי ואיפה הוא נמצא
- מה צריך להעביר אל/מ הדיסק הקשיח.
- לזכור היכן הנתונים שלנו מאוחסנים בדיסק הקשיח לקריאה חוזרת

### מרחב הכתובות

התוכניות שהמחשב מריץ עושות שימוש הן בנתונים מהזיכרון, והן בפקודות עצמן שהן מבצעות, שגם הן נמצאות בזיכרון. כעת נחשוב: אם הזיכרון שלנו משותף לכל התהליכים, מה מבטיח לנו שהתוכנית שלנו תטען כל פעם על אותן הכתובות בזיכרון? ואם נאמר שאכן שום דבר לא מבטיח זאת, אז איך יתכן שהתוכנית תדע כל פעם מחדש היכן נמצאים הנתונים שלה? שהרי אם בקוד כתוב "גש לכתובת $X$", ושום דבר לא מבטיח לי שהנתון אכן בכתובת $X$, הרי שאני עשוי לגשת לנתונים לא רלוונטיים ואף אסורים

### זיכרון וירטואלי (כאילו זיכרון לא אמיתי כזה כאילו)

זיכרון וירטואלי היא **שיטה** לניהול זיכרון במערכת מרובת תוכניות, שבה ניתן להקצות לכל תהליך את מרחב הכתובות שלו ללא תלות בגודל הזיכרון הראשי

בעת כתיבת תוכנה, לא ניתן לדעת היכן זו תימצא במחשבו של המשתמש, ולכן לכאורה אין דרך שבה התוכנה תדע לגשת לזיכרון, שכן היא לא יודעת את כתובות הנתונים שברצונה לקבל בעזרת הזיכרון הווירטואלי, כל תוכנה "רואה" מרחב זיכרון משלה, והיא ניגשת לנתונים ולפקודות בו באמצעות כתובות וירטואליות מרחב זה מכיל בין השאר את שירותי מערכת ההפעלה הממופים תמיד לאותן כתובות

### כתובת לוגית (וירטואלית) וכתובת פיזית

כאשר מערכת עובדת עם זיכרון וירטואלי, כל תוכנה מקבלת מרחב כתובות וירטואלי משלה. כאשר היא תרצה נתון, היא תפנה לנתון הנמצא במרחב הווירטואלי אותן היא "רואה". הכתובת אותה התוכנה תבקש היא **כתובת וירטואלית** המעבד יקבל את הכתובת הוירטואלית ויבצע המרה **לכתובת פיזית**, שהיא הכתובת האמיתית של הנתון בזיכרון. כדי שהמעבד ידע להמיר את הכתובות, על כל תוכנה לספק "טבלת כתובות" בה מפורטות כל הכתובות הפיזיות המתאימות לכתובות הוירטואליות, אם מתברר שהנתון המבוקש אינו קיים בזיכרון הראשי וצריך להביא אותו מהדיסק, נדרשת פסיקה שתודיע על כך

לכן, מימוש יעיל של זיכרון וירטואלי מצריך תמיכה מהחומרה:

א. כדי שהמעבד ידע לבצע המרה של כתובות

ב. כדי לתמוך בפסיקות הנדרשות

שיטה זו עונה על כל הצרכים שהגדרנו בראשית הפרק, בתוספת היתרון של חלוקת משאבים מצומצמים למספר גדול של תהליכים בו זמנית

## חלוקת זיכרון קבועה

כדי לממש את שיטת הזיכרון הווירטואלי, השתמשו בעבר בשיטה הבאה:

מערכת ההפעלה מחלקת את הזיכרון הפיזי לחתיכות בעלות גודל קבוע (נקרא לו $SIZE$). כל חתיכה יכולה להכיל חלק ממרחב כתובות של תהליך, לכל תהליך מוקצית **חתיכה אחת** מהזיכרון הפיזי, והיא תכיל את המידע הדרוש לתהליך בטווח המייד; שאר מרחב הכתובות שלו יישאר בדיסק לתהליך שרץ כרגע ישנם

שני רגיסטרים: $Base_{Phisical}$, $Base_{Virtual}$ שתפקידם לשמור את כתובת ההתחלה של מרחב הכתובות הפיזי (בדיסק) של התהליך, ולשמור את הכתובת של ההתחלה של החתיכה שנמצאת כרגע בזיכרון הראשי תרגום כתובת וירטואלית:

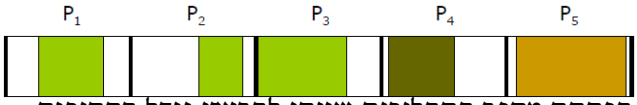
תהי כתובת וירטואלית $VirtualAdress$. הכתובת נמצאת כרגע בזיכרון הראשי אם היא בתחום $[Base_V\ ,\ Base_V + SIZE]$. אם היא לא בתחום, צריך להביא אותה מהדיסק

כאשר רוצים להביא נתונים חדשים צריך להחליף את המידע הנמצא בחתיכת הזיכרון הראשי כרגע. לשם כך מציבים ברגיסטר $Base_{Virtual}$ את כתובת התחלת החלק הווירטואלי החדש שנטען לזיכרון.

כאשר מחליפים תהליך במעבד (Context Switch), מציבים ברגיסטר $Base_P$ את כתובת החתיכה בזיכרון הפיזי הראשי השייכת לתהליך החדש, כמו כן מעדכנים את $Base_V$ כך שיתאים לתהליך החדש

### חסרונות השיטה הישנה

1. ההחלפה מתבצעת בחתיכות שלמות, למרות שלעיתים נדרש בסך הכל נתון קטן אחד.
2. שברור – *Fragmentation*. כל תהליך מקבל חתיכה בגודל קבוע, ללא תלות במספר הנתונים שהוא אכן ישתמש בסופו של דבר. כתוצאה מכך, יתכן מצב של בזבוז זיכרון שכן אם תהליך משתמש במעט נתונים, הוא עדיין תופס חתיכה שלמה. לכן תמונת הזיכרון בשימוש תראה לא רציפה.



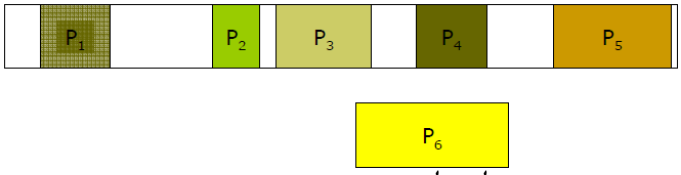

התוצאה הישירה של תופעה זו היא הפחתת מספר התהליכים שניתן להריץ, כי גודל החתיכות גדול ממה שנדרש ומספר התהליכים אינו עולה על מספר חתיכות. שברור מסוג זה נקרא **שברור פנימי**, כי איבוד הזיכרון קורה בתוך כל חתיכה של כל תהליך.

### פיתרון פשוט

כתוצאה מבעית השברור פותחה שיטה הבנויה על השיטה הישנה, עם הבדל מרכזי אחד: גודל החתיכה היה משתנה בצורה דינאמית לכל תהליך. רגיסטר חדש במערכת החזיק את גודל החתיכה של תהליך המתבצע כרגע, כך שניתן היה להגדיל או להקטין אותה במידת הצורך. השיטה זו אמנם פותר את בעיית השברור הפנימי אבל מעוררת בעיה חדשה.

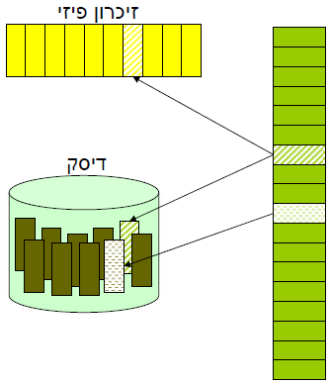
### שברור חיצוני

נניח שתהליך קיבל חתיכה מסוימת המסתיימת בכתובת $X$. תהליך אחר מאוחר יותר קיבל חתיכה המתחילה בכתובת $X+1$. כעת נניח שהתהליך הראשון דיווח שהוא איננו צריך יותר את כל החתיכה שלו והוא יכול להסתפק בחתיכה יותר קטנה. המערכת מקצרת לו את החתיכה כך שהיא כעת עד כתובת $X-5$. מה שקרה עכשיו הוא שנוצר מרווח קטן של 6 כתובות בין החתיכות של שני התהליכים. אף תהליך לא יכול להשתמש במרווח כלכך קטן, ולכן המקום הזה פשוט מבוזבז. תופעה זו נקראת **שברור חיצוני**, כי איבוד הזיכרון קורה בין החתיכות של תהליכים שונים.



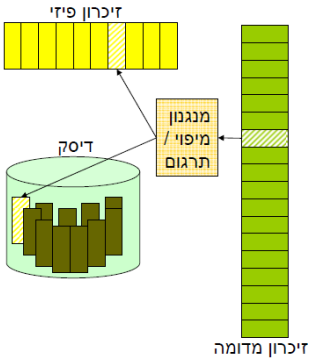

ניתן לראות בתמונה שלא נותר שום מקום **לציף** המספיק לתהליך מספר 6.

## השיטה המודרנית – עימוד (חלוקה לדפים) - *Paging*




כיום ממשים זיכרון וירטואלי בזורה מתוחכמת יותר. מחלקים את הזיכרון הווירטואלי לחלקים בגודל שווה הנקראים **דפים – *Pages***. על הדפים להיות בגודל כזה שמצד אחד יאפשר קריאה וכתיבה יעילים לדיסק, ומצד שני יהיו מספיק קטנים לאפר גמישות בחלוקה. הגודל המקובל כיום הוא *4Kb* לעמוד.

את הזיכרון הראשי, אנו מחלקים לחתיכות הנקראות **מסגרות – *Frames***. בגודל עמוד כל אחת. כל מסגרת יכולה להכיל עמוד אחד מהזיכרון הווירטואלי. (לא לשכוח כמובן שזיכרון וירטואלי הוא לא יותר מאשר אזור מסוים בדיסק הקשיח המוקצה למטרה זו ולא זיכרון מיוחד כלשהו). כל עמוד בזיכרון הווירטואלי חייב להיות כתוב בדיסק.



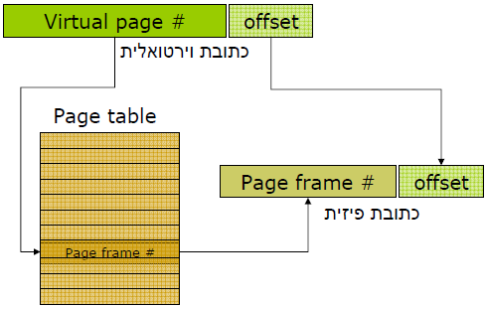

### מנגנון תרגום כתובת וירטואלית

כמו בשיטה הישנה, גם כאן נדרש מנגנון המקבל כתובת וירטואלית וממיר אותה לכתובת פיזית בזיכרון הראשי, או בדיסק אם היא לא נמצאת. כמובן שגם מנגנון זה מצריך תמיכה של החומרה.

כאמור, כל תהליך מחזיק טבלה, אשר בה מפורטות הכתובות הלוגיות והערך הפיזי שלהן. כל שורה בטבלה מתייחסת לדף וירטואלי מסוים בעל מספר אינדקס מסוים. בכל שורה גם נמצאת כתובת פזית של המסגרת המכילה את הדף הזה (אם ב*RAM* ואם ב*H.D*).

כל כתובת וירטואלית מורכבת משני שדות:

1. מספר הדף הווירטואלי; נקרא גם *VPN – Virtual Page Number*.
2. שדה היסט *(Offset)* המציין את המיקום בתוך אותו הדף




שדה ה*VPN* מהווה מצביע לאיזו שורה להסתכל בטבלת הדפים של התהליך; כדי למצוא את כתובת המסגרת הפיזית שמחזיקה אותו, הוספת שדה ההיסט נותנת את הכתובת הפיזית המדויקת המקבילה לכתובת הוירטואלית.

בכל שורה בטבלת הדפים נמצא גם מידע נוסף הנתון לנו אינדיקציה לפרמטרים מסוימים כדי שנוכל לנהל זיכרון ביעילות רבה יותר. פרמטרים אלה הם:

1. סיבית המציינת: האם הכתובת רלוונטית? כלומר האם קיים בה מידע ממשי ולא אותות אקראיים
2. סיבית המציינת: האם ניגשו לדף הזה?
3. סיבית *Dirty*- האם נעשה שינוי בדף זה? רלוונטי כשנרצה להחזיר דף ל*HD*.
4. שדה "סיווג ביטחוני" – מציין מה מותר ומה אסור לבצע בדף זה

בצורה סכמטית כל שורה בטבלה נראית כך:

| V | R | M | protect | Page frame # |
|---|---|---|---|---|

## שגיאת עמוד – *Page Fault*

שגיאת עמוד היא שגיאה המתרחשת כאשר תהליך מבקש נתון אשר הכתובת שלו אינה בזיכרון הראשי אלא ב*HD*. אם התרחשה שגיאה מסוג זה נדרשות מספר פעולות:

- הבאת העמוד בו נמצא הנתון אל הזיכרון הפיזי
- במידה ואין מקום צריך להחליט איזה מסגרת לפנות
- במידה והחלטנו לפנות מסגרת ייתכן ונדרש לעדכן את הדף המתאים ל*HD* (תלוי במדיניות הכתיבה)
- כאשר מסתיים תהליך ההחלפה יש לבצע את הפקודה מחדש

שגיאה זו יכולה להיגרם גם כתוצאה מגישה לדפים מחוץ לתחום, דפים לא קיימים וכ"ו.

## יתרונות שיטת העימוד

1. פותר לחלוטין בעיית שברור חיצוני: גודל המסגרת אינו גמיש
2. מצמצם מאוד שברור פנימי: גודל עמוד מספיק קטן, כך שנדיר מאוד שתהליך לא ישתמש בכל חייו אפילו בדף שלם אחד
3. גודל הדף מאפשר (כמובן בעזרת תמיכת חומרה) העברה בבת אחת של דף שלם
4. לא חייבים למחוק מסגרות, ניתן פשוט לסמנן כלא-רלוונטיות (סיבית רלוונטיות).
5. ניתן להקצות שטח זיכרון לא רציף לכל תהליך; כל מסגרת מתאימה לדף שלם, לא משנה מה הכתובת שלו או שלה

## חסרונות

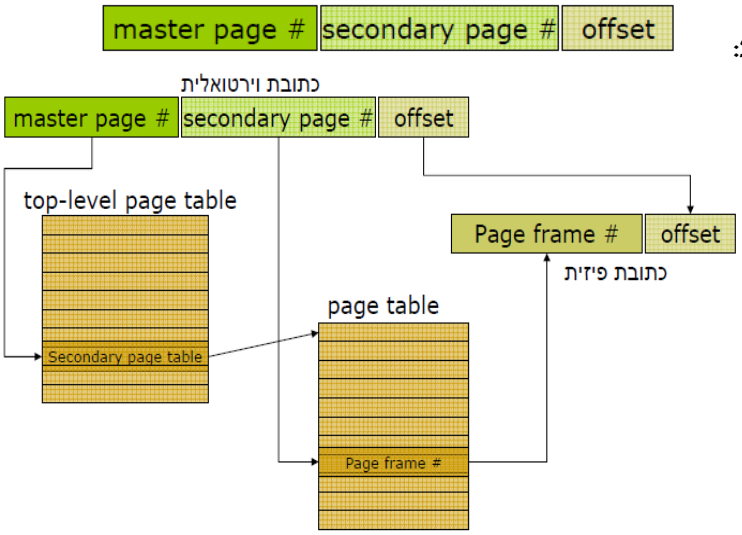
1. להחזיק טבלת דפים לכל תהליך עשוי להיות גורם מאוד זולל זיכרון
2. עדיין יש שברור פנימי מועט
3. כל תרגום כתובת מצריך גישה לזיכרון לטבלת הכתובות, מה שאומר פי שנים יותר ממה שהיה צריך להיות

### פתרונות - טבלה רבת רמות

גודל של טבלת עמודים עשוי להיות כה גדול (4Mb!!) שלא יעיל להחזיק אותן, לכן פותחה שיטה המשפרת את שיטת העימוד כך שלא נחזיק טבלאות עמודים גדולות מדי, לשם כך חילקו את טבלת העמודים לשתי רמות: הרמה הראשונה היא בגודל דף בלבד והיא נמצאת כל הזמן בזיכרון הראשאבל היא איננה מצביעה לכתובת פיזית של נתון אלא לטבלה ברמה השנייה שנמצאת בדיסק הרמה השנייה אינה אלא אוסף של טבלאות נוספות שנמצאות דרך קבע בדיסק זאת אומרת שעלינו לעדכן גם את מבנה הכתובת הווירטואלית כדי שיתאים למבנה החדשלכתובת החדשה שלוש שדות:

1. שדה הרמה העליונה– מצביע לשורה המתאימה ברמה העליונה, שמצביעה לטבלה מסויימת ברמה השנייה.
2. שדה הרמה השנייה– מכיל את מספר השורה הנדרש ברמה השנייה, שמצביעה לכתובת פיזית של מסגרת.
3. שדה ההיסט– מכיל את הכתובת הפנימית בתוך המסגרת

מבנה הכתובת החדשה יראה בערך כך:




ובצורה סכמטית ניתן לתאר כך את המערכת:
החיסרון בשיטה הזאת הוא שכדי לתרגם כתובת אנו צריכים לגשת פעמיים לזיכרון ואחת מהן היא לדיסק הקשית הפתרון לבעיה זו הוא פתרון חומרת:

### *TLB – Translation Look aside Buffer*

ה *TLB* הוא זיכרון מטמון קטן מאוד, המכיל מספר מסוים של כתובות וירטואליות ממופות (שעברו תרגום לכתובת פיזית). כתובות אלו הן הכתובות האחרונות בהן השתמש התהליך הרץ כרגע. בגלל עיקרון המקומיות בזמן ובמקום, 99% מהכתובות שניגש אליהן הן אותן הכתובות שנמצאות ב *TLB*.

### קבוצת עבודה של תהליך

קבוצת עבודה (*Working Set*) $WS_P(w)$ מוגדרת להיות אוסף הדפים אליהם ניגש תהליך $P$ ב $w$ הגישות האחרונות שלו. ככל שקבוצת העבודה יותר קטנה, כלומר התהליך ניגש פחות או יותר לאותם הדפים בזמן האחרון, כך מידת המקומיות של התהליך גדולה יותר. קבוצת עבודה גדולה מדי עשויה למוטט מערכת הפעלה כי היא תהיה עסוקה רק בהביא ולפנות דפים מהזיכרון.
כזכור, כאשר תהליך מבקש כתובת בדף שלא נמצא בזיכרון, צריך להביא אותו מהדיסק, ואם צריך אז לפנות דף אחר כדי לעשות לו מקום

## סגמנטציה – חלוקה לקטעים בעלי משמעות

בשיטה זו מחלקים את מרחב הכתובות של כל תהליך לקטעים (סגמנטים) בעלי משמעות מסויימת: למשל פונקציה מסוימת, משתנים גלובאליים, מחסנית, ערימה וכ"ו. כמובן שכל סגמנט יכול להיות באורך שונה ולכן יתכן מצב של שברור חיצוני:

### מבנה הכתובת ותרגומה

כתובת וירטואלית בשיטת הסגמנטציה מחולקת לשני שדות, שדה הסגמנט ושדה ההיסט שדה הסגמנט מציין את מספר הסגמנט המבוקש. שדה ההיסט מציין את הכתובת הפנימית בתוך הסגמנט. בדומה לשיטת העימוד, גם כאן נדרשות "טבלאות סגמנטים" המחזיקות את מספר הסגמנט, הכתובת הפיזית שלו ונתוני עזר נוספים. בנוסף, בטבלה יש גם מידע על אורך כל סגמנט מהסיבה שהוא איננו קבוע

### יתרונות

העיקרון העומד מאחורי שיטת הסגמנטציה מאוד דומה לשיטת העימוד אבל הוא מנצל את עיקרון המקומיות בצורה הטובה ביותר שניתן, כי אנו מתייחסים לחלקים בעלי משמעות כגוש אחד. כלומר אם אני נמצא בפונקציה מסוימת, אני אביא לזיכרון את הסגמנט המכיל את כל הפונקציה, ואת כל המשתנים שלה, כך שכמעט אין סיכוי שאני אבצע גישות לזיכרון שמחוץ לסגמנטים אלו.

### חסרון
חסרונה היחיד של שיטה זו לעומת שיטת העימוד היא האפשרות לשברור חיצוני

## שיטה משולבת – מערכת ההפעלה *MULTICS*

ניתן לשלב יחד את שיטות הסגמנטציה והעימוד, כך שכל סגמנט יהיה מורכב ממספר מסוים של דפים, כך ששברור חיצוני לא יהיה כלל, והשברור הפנימי יהיה מינימאלי ביותר. שיטה זו מממשת את יתרונות שתי השיטות.
במקרה זה על הכתובת הוירטואלית לספק גם את מספר הסגמנט, גם את מספר העמוד, וגם היסט בתוך העמוד.

## מדיניות הבאה ופינוי דפים

### מדד השוואה – *EAT – Effective Access Time*
אנו רוצים לדעת כמה זמן לוקח לגשת לנתון מסוים מרגע הבקשה בתהליך. אך נתון זה תלוי בגורמים כגון, האם הנתון בדיסק, האם צריך להביא אותו וכו'. לכן פיתחו מדד שיהווה אינדיקציה למידת יעילות של שיטת ניהול זיכרון והוא מבטא את זמן הגישה האפקטיבי. כלומר, גם נתונים כגון מספר שגיאות העמוד נלקחים בחשבון. נגדיר את יחס הפגיעות (פגיעה – בקשת דף הנמצא בזיכרון) להיות: $\alpha = \frac{Hits}{All}$.
נוסחת זמן הגישה האפקטיבי היא: $\alpha * (MemAccessTime) + (1-\alpha) * (DiskAccessTime)$.
הסבר: החלק היחסי של הפגיעות כפול משך זמן גישה לזיכרון הראשי, בתוספת החלק היחסי של ההחטאות כפול משך זמן הגישה לדיסק

### מדיניות הבאה
בהבאת דפים לזיכרון קיימות שתי גישות:

1. הבאה לפי דרישה – מביאים אך ורק דפים שתהליך מנסה לגשת אליהם
2. חיזוי מראש – המערכת תנסה לנבא מה העמודים שהתהליך יצטרך בקרוב, ותביא אותם כשהיא תוכל, עוד לפני שהתהליך ביקש. דורש מנגנון חיזוי; אם עובד כמו שצריך חוסך הרבה גישות לזיכרון.

בכל שיטה ניתן להחליט אם להביא עמוד אחד או מספר עמודים הסמוכים לו

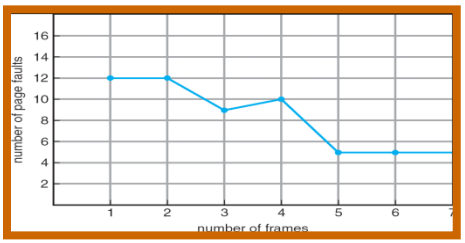
### מדיניות פינוי
השאלה הגדולה ביותר שנדרשים לענות עליה כשרוצים לפנות דף היא: איזה דף לפנות? מי אמר שלא נפנה דף שתהליך אחר זקוק לו ביותר? ישנם מספר גורמים שצריך להתייחס אליהם בהחלטה זאת:

1. האם נעשו שינויים בדף? יתכן שנצטרך לכתוב את הדף לדיסק אם נעשו בו שינויים
2. האם הדף נמצא בשימוש תדיר?
3. כמה פז"ם יש לדף בזיכרון הראשי?

לרוב, פינוי הדפים מתבצע ברקע, כלומר גם ללא דרישה ספציפית. ברגע שהמערכת פנויה היא מפנה דפים כמו כן גם כתיבה של דפים "מלוכלכים" מתבצעת ברקע, כדי שכאשר נפנה אותם לא נצטרך לכתוב אותם לדיסק.
האלגוריתמים המטפלים בפינוי משתמשים מנתונים מטבלת העמודים שהזכרנו לעיל, כדי להשיג את מספר שגיאות העמוד הנמוך ביותר שניתן וזה יהיה מאוד.. נחמד.

### אלגוריתם *FIFO*.
בשיטה זו העמוד שהגיע ראשון ונמצא הכי הרבה זמן בזיכרון יהיה הראשון להתפנות (כמו גוש קטיף רק בלי ההפגנות). אלגוריתם זה הוא נורא פשוט אך לא מגביר יעילות בשום דרך ויכול רק להזיק

### אנומליית Belady. (מקור השם אינו ידוע, אם כי אין סיכוי שהוא תימנע)




כאשר משתמשים באלגוריתם FIFO עשויה להתרחש תופעה מוזרה: כדי להעריך יעילות של אלגוריתמים פיתחו מדד שמשווה את מספר ה-Page Faults המתרחשות בכל שיטה עבור סדרת גישות מסויימת. ההיגיון אומר שככל שיהיו לנו יותר מסגרות זמינות, יהיו פחות שגיאות עמוד כי יש יותר מסגרות, ולכן יש יותר סיכוי שעמוד מסוים יהיה בתוך מסגרת קיימת. אך למעשה מתברר שלעיתים הגדלת מספר המסגרות רק מגדילה את מספר השגיאות. כלומר, הגדלת הזיכרון לא תמיד מועילה כשעובדים עם FIFO.

דוגמא:

| | A | B | C | D | A | B | E | A | B | C | D | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | A | A | A | D | D | D | E | E | E | E | E | E |
| 1 | | B | B | B | A | A | A | A | A | C | C | C |
| 2 | | | C | C | C | B | B | B | B | B | D | D |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | A | A | A | A | A | A | E | E | E | E | D | D |
| 1 | | B | B | B | B | B | B | A | A | A | A | E |
| 2 | | | C | C | C | C | C | C | B | B | B | B |
| 3 | | | | D | D | D | D | D | D | C | C | C |

בטבלה העליונה רואים את מצב הזיכרון כאשר ישנן שלוש מסגרות. ניתן לראות שמספר הפעולות שביצענו על הזיכרון הוא 9. בטבלה התחתונה רואים זיכרון בעל ארבע מסגרות, ואילו שם מספר הפעולות הוא 10. זוהי תופעה נדירה אך אפשרית.

### אלגוריתם- LRU – Least Recently Used

אלגוריתם זה לא לוקח בחשבון את משך הזמן שעמוד נמצא בזיכרון, אלא את משך הזמן בו הוא נמצא ללא שימוש. העמוד שהכי פחות משתמשים בו יהיה הראשון ללכת. לשם כך נדרש מנגנון שיעקוב אחרי הדפים ויעקוב אחרי מידת השימוש בהם. ניתן לבצע זאת ע"י החזקת מחסנית של מספרי הדפים. אשר כל גישה לדף מקפיצה אותו לראש המחסנית. כאשר נרצה להוציא דף, נוציא את הדף שמספרו נמצא בתחתית המחסנית. דרך נוספת היא הדבקת "חותמת" עם השעה הנוכחית לכל עמוד שניגשנו אליו. והעמוד עם השעה המוקדמת ביותר יהיה הראשון ללכת.

### אלגוריתם "החמדן"

אלגוריתם זה מפנה את העמוד שמשך הזמן עד הגישה הבאה אליו הוא הכי גדול. כמובן שהוא מצריך מנגנון מאוד מסובך שאמור לנסות לנבא עבור כל עמוד מתי הוא ייקרא.

### אלגוריתם "הזדמנות שנייה"

אלגוריתם זה בשימוש רק במערכות בהן הפינוי מתבצע ברקע. כאשר נתבצע פינוי המערכת בודקת את "סיבית הגישה" בטבלת העמודים. אם לא ניגשו – פינוי מיידי. אם ניגשו (הסיבית = 1) – מאפסים את הסיבית. כלומר, כל עוד שעמוד נמצא בשימוש בין פינוי למשנהו, הוא לא יפונה. עמוד שלא ניגשו אליו במשך שני פינויים רצופים יפונה בוודאות.

### אלגוריתם מנייה

בשיטה זו מחזיקים "מונה גישות" לכל עמוד אשר מעלים אותו ב 1 בכל גישה. העמוד בעל מספר הגישות הנמוך/גבוהה ביותר יפונה.

תודה רבה לכל מי שעזר לי עם הסיכום הזה וענה לי על שאלות,
וגם לחגית עטייה על המצגות שלה, ואפילו לד"ר פרדג' שהיה מאוד.... נחמד.

הסיכום נכתב בעיקר בשעות לילה מאוחרות ובהאזנה שוטפת ל-Dream Theater.
אני אשמח לקבל כל הערה והערה ולתת הסברים יותר מפורטים במקרה
הצורך, וכן המלצות מוזיקליות במקרים דחופים

אברהם שוקרון AvrahamShuk@gmail.com.