הרצאה 1:

מהי מערכת הפעלה ?

- תכנה המתנהגת כמתווך בין משתמש במחשב לבין חומרת המחשב.

מטרות מערכת ההפעלה:

- בקרה / הרצה של אפליקציות.
- גורמת למחשב להיות ידידותי לשימוש.
- מקלה בפתרון בעיות.
- שימוש יעיל יותר בחומרת המחשב.

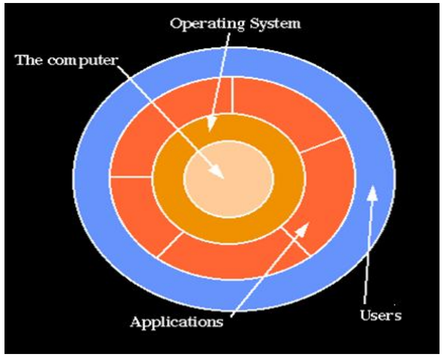
שירותים הניתנים ע"י מערכת ההפעלה:

- שירותים ליצירת תוכנות:
  - עורכים, קומפיילרים, לינקרים, דיבאגרים וכו'...
- הרצת תוכנות
  - הטענה בזיכרון, קלט/פלט ואתחול קבצים
- גישה לקלט/פלט וקבצים
  - מתעסק עם התקני קלט/פלט וסוגי קבצים.
- גישת מערכת
  - פותר התנגשויות של מחלוקות משאב.
  - הגנה בגישה למשאבים ומידע.

למה מערכות הפעלה חשובות ?

- חשוב להבין ולדעת כיצד להשתמש נכון כאשר כותבים אפליקציות משתמש.
- מערכות גדולות ומסובכות שיש להן השפעה כלכלית גדולה
- רק למעטים מאוד תהיה מעורבות בתכנון מערכת הפעלה (OS) ויישומה.
  אף על פי כן, ניתן ללמוד טכניקות כלליות רבות וליישמן במקומות אחרים.
- משלב הרבה קונספטים מהרבה תחומים ממדעי המחשב:
  ארכיטקטורה, שפות, מבני נתונים, אלגוריתמים.

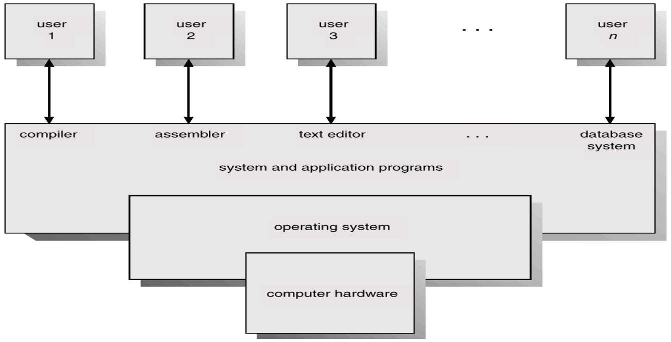
היררכיה של מערכת מחשב:



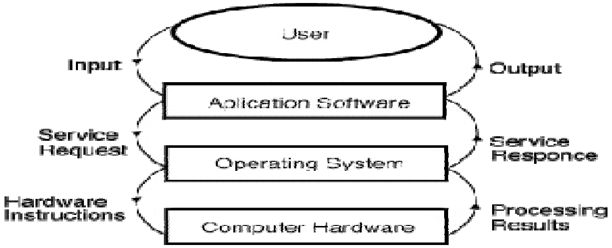
מרכיבים של מערכת מחשב:

- חומרה – מעניקה משאבי מחשוב בסיסיים (CPU, זיכרון, התקני קלט/פלט).
- מערכת הפעלה – מבקרת ומתאמת את השימוש בחומרה בקרב האפליקציות השונות עבור משתמשים שונים.
- Utilities (עזרים) – תוכניות המסייעות בניהול מערכת ופיתוח תוכנה.
- תוכניות אפליקציה – מגדירות את הדרכים שבהם משאבי המערכת משמשים לפתירת בעיות מחשוב עבור המשתמשים (קומפיילרים, מערכות db, וידאו וכו'...).

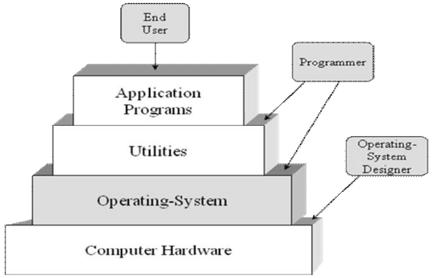
מראה סטטי של מרכיבי מערכת:



מראה דינאמי של מרכיבי מערכת:



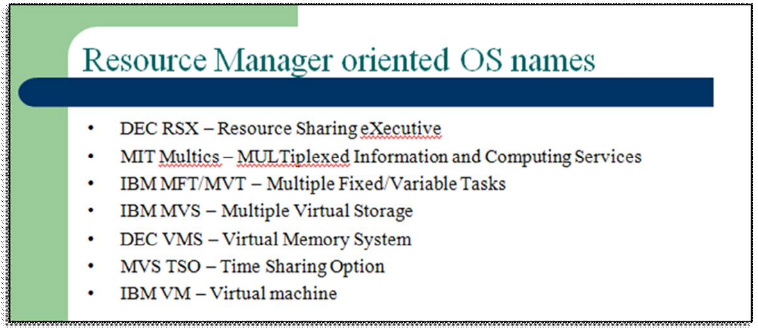
שכבות של מערכת מחשב:



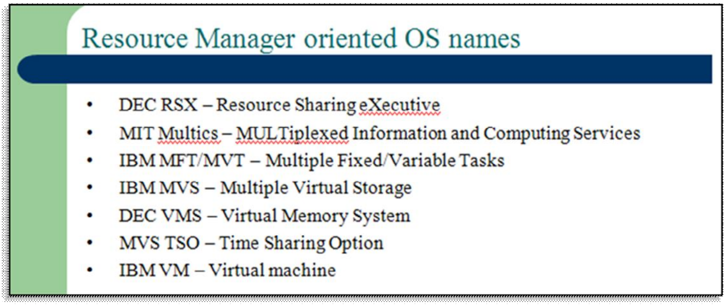
<u>מראות במערכת הפעלה:</u>

- ישנן 3 מראות קלאסיות (בספרות):
  1. Resource Manager (מנהל משאב) – מנהל ומקצה משאבים.
  2. Control program (בקר תכנה) – מבקר את הרצת תוכניות המשתמש ופעולת התקני I/O.
  3. Command Executer (מבצע פקודה) – מעניק סביבה להרצת פקודות משתמש.
- מראה מודרני נוסף: מערכת ההפעלה כמערכת וירטואלית (Virtual Machine)

1. <u>Resource Manager</u>
   - מטפל במשאבי מחשב מרובים: CPU, זיכרון חיצוני/פנימי, תהליכים, משימות, אפליקציות ועוד'...
   - מנהל ומקצה משאבים למשתמשים מרובים או עבודות מרובות הרצות בו זמנית (לדג', זמן מעבד, מרווח זיכרון, התקני קלט/פלט).
   - מארגן את חומרת המחשב לשימוש יעיל (מיקסום תפוקה (throughput), מינימיזציה של זמן תגובה) ובאופן הגיוני.
   - Sort of a bottom-up view.

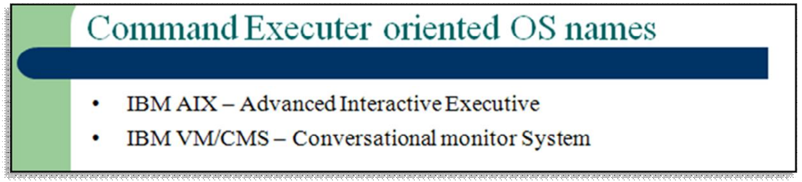


2. <u>Control Program</u>
   - מנהל את כל המרכיבים במערכת מחשב מסובכת באופן משולב.
   - מבקר את הרצת תוכניות משתמש והתקני I/O כדי למנוע תקלות ושימוש לא תקין של משאבי המחשב.
   - מתבונן ומגן על המחשב: מנטר, משגיח, ביצועי, בקר, מאסטר, מתאם ...
   מעין ראיה כקופסה שחורה



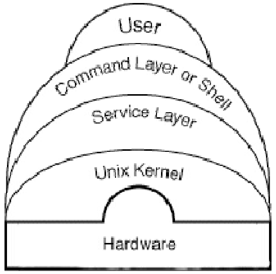
3. <u>Command Executer</u>
   - מממשק בין משתמשים והמכונה.
   - מעניק שירותים / Utilities למשתמשים.
   - מעניק למשתמשים (Command Language Interface) CLI נוח, נקרא גם Shell (ב-Unix), להכנסת פקודות משתמש.

   - Sort of a top-down view



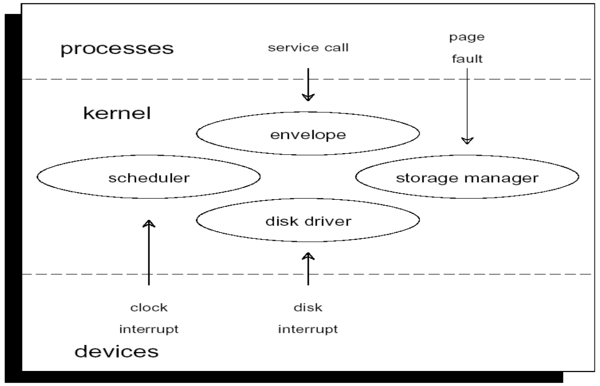
<u>ראיה מודרנית: מערכת וירטואלית:</u>
- ממשק בין המשתמש וחומרה המחביא את פרטי החומרה (למשל I/O).
- מרכיב משאבים ברמה גבוהה יותר (וירטואלים) מתוך משאבים ברמה נמוכה (פיזיים) (לדג' קבצים).
- **הגדרה:** OS היא אוסף של תוכנות מוגברות, הרצות על החומרה "הערומה", culminating in a high-level virtual machine that serves as an advanced programming environment
  - מערכת וירטואלית = תוכנה מוגברת = מערכת מורחבת = מערכת מופשטת = שכבה = רמה = טבעת.

<u>מערך מערכת Unix</u>

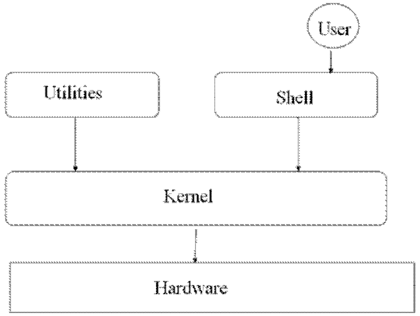


<u>מהו ה- Kernel (גרעין) ?</u>
- מערכת ההפעלה היא ה- Kernel – ה- Kernel הוא התוכנה היחידה שרצה בכל הפעמים.
- ה- Shell נהוג היה להיות בתוך ה- Kernel אך כעת הוא Utility (שווה בין שווים) מחוצה לו:
  - קל לשנות / לדאבג.
  - הרבה סוגים שלו (sh, bsh, csh, ksh, tcsh, wsh, bash)
  - ניתן להחליף ביניהם (chsh).

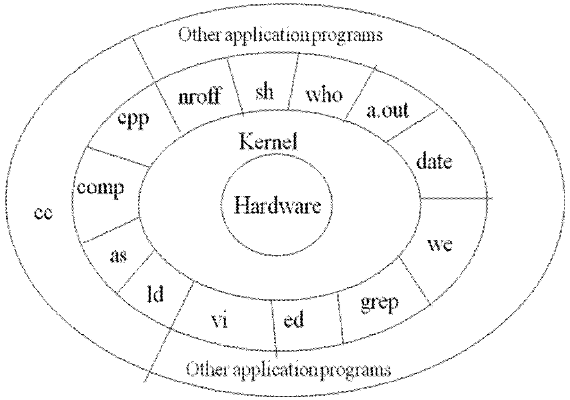
<u>מבט על ה- Kernel:</u>



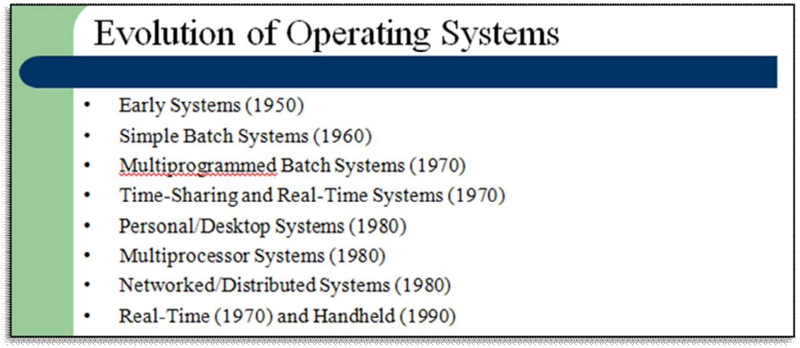
<u>Shell של Unix ו- Utilities</u>



<u>ארכיטקטורה של Unix</u>



<u>התפתחות של מערכת ההפעלה</u>



<u>מאפיינים של מערכות מוקדמות:</u>

- תוכנה מוקדמת: אסמבלרים, תיקיות של סאב-רוטינות שכיחות, דרייברים, קומפיילרים, לינקרים.
- צורך בזמן הגדרה משמעותי.
- התקני I/O איטיים מאוד.
- ניצול נמוך של ה- CPU.
- אבל מחשבים היו מאוד מאובטחים.

<u>מערכות אצוה (batch) פשוטות:</u>

- שימוש בשפה גבוהה, טייפים מגנטיים.
- עבודות נאספות ביחד ע"י סוג השפה.
- שוכרים מפעיל לבצע משימות נשנות של טעינת עבודות, אתחול המחשב, ואיסוף של הפלט.
- לא היה בר ביצוע עבור משתמשים לבחון זיכרון או להטלות (patch) תוכניות בישירות.

<u>פעולות של מערכות אצוה פשוטות</u>

- המשתמש שולח עבודה (כתובה על כרטיסים או טייפ) למפעיל מחשב.
- מפעיל המחשב ממקם אצוה של מספר עבודות בקלט של ההתקן.
- תוכנית מיוחדת, המוניטור, מנהל את ביצוע של כל תוכנית באצוה.
- מנטר Utilities נטענים כשצריך.
- מוניטור מקומי תמיד בזיכרון הראשי וזמין לביצוע.

<u>רעיון מערכות אצוה פשוטות</u>

- הפחתת זמן הגדרה ע"י קיבוץ עבודות דומות.
- ביצוע חלופי בין תכנת משתמש ותכנת ניטור.
- מסתמך על חומרה זמינה. to effectively alternate execution from various parts of memory.
- שימוש ברצף עבודות אוטומטי – מעביר בקרה אוטומטית מעבודה אחת כאשר הוא מסיים לאחת אחרת.

<u>כרטיסי בקרה</u>

- בעיות:
  1. כיצד המוניטור יודע על טבעה של העבודה (לדג' Fortran מול Assembly) או איזו תוכנה להריץ ?
  2. כיצד המוניטור מבחין:
     - עבודה מעבודה ?
     - מידע מתוכנית ?
- פתרון: הכרות Job Control Language (JCL) וכרטיסי בקרה.

- כרטיסים מיוחדים האומרים למוניטור אילו תוכניות לרוץ:

- תווים מיוחדים מבחינים בין כרטיסי בקרה ממידע או כרטיסיות תוכנית:

$ in column 1
// in column 1 and 2
709 in column1

<u>השפעות JCL</u>

- כל קריאת הוראה (בתכנית משתמש) גורמת לקריאת קלט של שורה.
- Causes (OS) input routine to be invoked:
  - checks for not reading a JCL line.
  - skip to the next JCL line at completion of user program.

<u>Resident Monitor</u>

- Resident Monitor הוא OS ראשון בסיסי.
- Resident Monitor:
  - בקרת אתחול בתוך המוניטור.
  - טוען את התכנית הבאה ומעביר אליו את הבקרה.
  - כאשר עבודה מסתיימת, מעבר הבקרה חוזר למוניטור.
  - מעביר בקרה אוטומטית מעבודה אחת לאחרת, אין זמן Idle (בטלה) בין תכניות.

חלקי ה- Resident Monitor

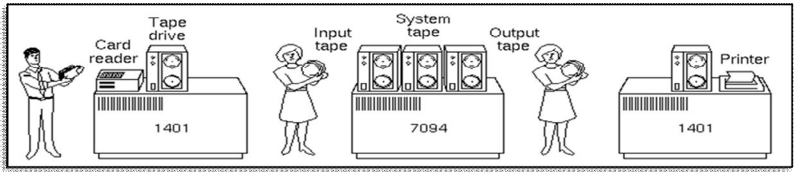
- שליטה במתרגם הכרטיס – אחראי לקריאה ונשיאתם של ההוראות על הכרטיסים.
- טוען (Loader) – טוען תכניות מערכת ואפליקציות לתוך הזיכרון.
- התקני דרייברים – יודע תווים מיוחדים ומאפיינים עבור כל אחד מהתקני ה- I/O במערכת.

<u>תכונות רצויות בחומרה:</u>

- הגנת זיכרון
  - לא מאפשר לשנות את אזור הזיכרון המכיל את המוניטור ע"י תכנית משתמש.
- הוראות בעלות זכויות יתר (Privileged Instructions)
  - רק המוניטור יכול להריצן.
  - מתרחשת מלכודת אם תוכנית ניסתה הוראות כאלו.
- פסיקות (Interrupts)
  - מעניקות גמישות לבקרת ויתור ורוכש בקרה מתכניות משתמש.
  - פסיקות טיימר מונעות מעבודה לקחת מונופול על המערכת.

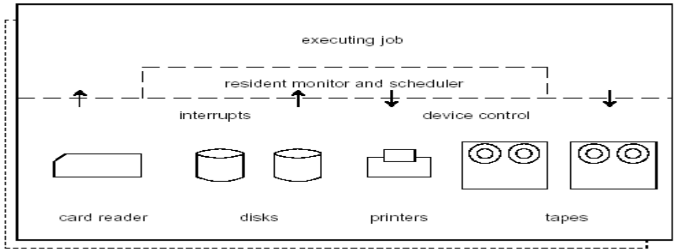
<u>פעולות לא מקוונות (Offline)</u>

- בעיה:
  - קורא כרטיס איטי, מדפסת איטית (בהשוואה לטייפ).
  - אין חפיפה בין I/O ו- CPU.
- פתרון: פעולות לא מקוונות (מחשבי לווין) – מאיצים את החישוב ע"י הטענת עבודות לזיכרון מטייפ בעוד שקריאת כרטיס והדפסת שורה מבוצעים בצורה בלתי מקוונת ע"י מכונות קטנות יותר.

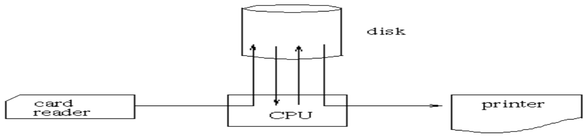


<u>Spooling</u>

- בעיה:
  - קורא כרטיס איטי, מדפסת איטית (בהשוואה לטייפ).
  - אין חפיפה בין I/O ו- CPU.
- פתרון: Spooling
  - חפיפת I/O של עבודה אחת עם החישוב של עבודה אחרת (שימוש בחוצץ כפול, DMA וכו'...)
  - טכניקת Spooling: Simultaneous Peripheral Operation On Line



- כאשר מבצעים עבודה אחת, מערכת ההפעלה:
  - קוראת את העבודה הבאה מקורא הכרטיס לתוך אזור אחסון על הדיסק (Job Spool).
  - מניבה תדפיס של העבודה הקודמת מהדיסק למדפסת.



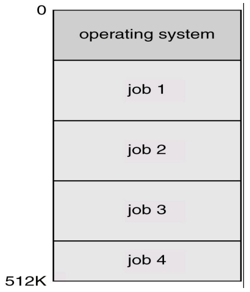
  - Job Spool – מבנה נתונים המאפשר ל- OS לבחור איזו עבודה להריץ פעם הבאה ע"מ להגביר את נצילות ה- CPU.

<u>Uniprogramming עד עכשיו</u>

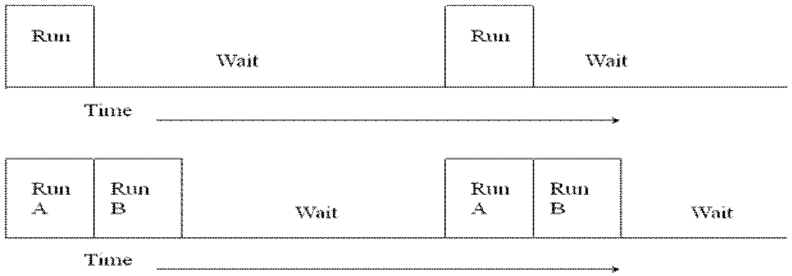
- פעולות I/O הן איטיות מאוד (בהשוואה לביצוע הוראות).
- תוכנית המכילה אפילו מספר פעולות קטן של I/O, תבזבז את רוב זמנה לחכות להם.
- לכן: שימוש דל ב CPU רק כאשר תוכנית אחד נוכחת בזיכרון.

<u>Multiprogrammed Batch System</u>

מספר עבודות נשמרות בזיכרון הראשי באותו זמן, וה- CPU מרבב ביניהן.



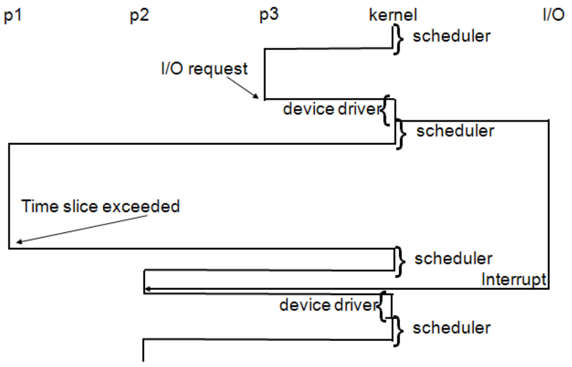
- אם זיכרון יכול להחזיק מספר תוכניות, אז ה- CPU יכול למתג לאחת אחרת בכל עת כאשר תוכנית ממתינה ש- I/O יסתיים.
- זהו multiprogramming.



<u>Batch Multiprogramming</u>

מאפשר למעבד להריץ תוכנית אחרת בעוד שתוכנית אחת מחכה להתקן I/O.

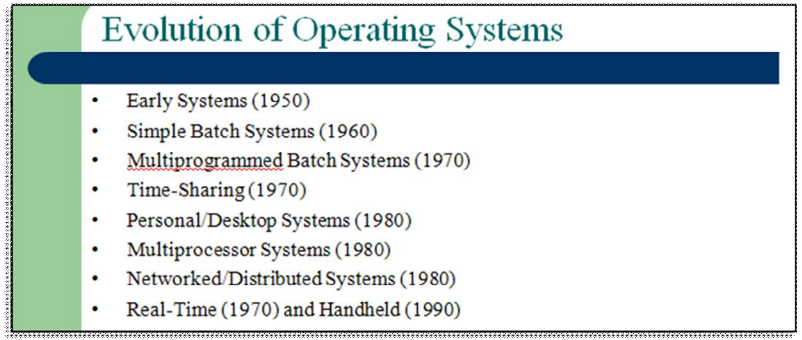
דג' ל- Multiprogramming:



<u>דרישות עבור Multiprogramming:</u>

- תמיכת חומרה
  - פסיקות I/O ובקרי DMA:
    - ע"מ לבצע הוראות בעוד שבתקן I/O עסוק.
  - פסיקות טיימר ל- CPU כדי לרכוש בקרה.
  - ניהול זיכרון:
    - מס' עבודות מוכנות-להרצה חייבות להישמר בזיכרון.
  - הגנת זיכרון (נתונים ותוכניות).
- תמיכת תכנה מה- OS:
  - לתזמון (איזו תכנית תורץ הלאה).
  - לנהל מחלוקות משאבים.

<u>אבולוציה של מערכות הפעלה</u>



<u>מערכות שיתוף זמנים – Time Sharing Systems (TSS)</u>

- Batch multiprogramming לא תומך באינטראקציה עם משתמשים
- TSS מרחיב את ה- Batch multiprogramming לטיפול אינטראקטיבי בעבודות – זהו Multiprogramming אינטראקטיבי.
- מספר משתמשים יכולים לגשת למערכת באמצעות טרמינל.
- זמן המעבד משותף בין מספר משתמשים.

<u>למה TSS עובד ?</u>

- בגלל זמן תגובה אנושי איטי, משתמש אופייני צריך 2 שניות של זמן עיבוד כל דקה.
- אז יתאפשר להרבה משתמשים לחלוק אותה מערכת מבלי לשים לב לעיכוב בזמן תגובת המחשב.
- המשתמש צריך לקבל זמן תגובה טוב.

<u>פעולות של TSS</u>

- ישנה תקשורת מקוונת בין המשתמש למערכת.
- כאשר מערכת ההפעלה מסיימת ביצוע פקודה אחת, היא מבקשת את הבאה ממקלדת המשתמש.
- ה- CPU מרובב בקרב מספר עבודות הנשמרות בזיכרון ועל הדיסק (ה- CPU מוקצה לעבודה רק אם העבודה בזיכרון).
- יכול להיות שעבודה תוחלף פנימה והחוצה מהזיכרון לדיסק.

<u>מערכות אישיות/נייחות – Personal\Desktop Systems</u>

- Personal Computers – מערכת מחשב המוקדשת למשתמש יחיד.
- I/O Devices – מקלדת, עכברים, תצוגה, ...
- נוחיות משתמש ותגובתיות.
- יכולה לאמץ טכנולוגיה למערכות הפעלה גדולות יותר; בד"כ לאינדיבידואלים ישנו שימוש יחיד למחשב ואין צורך בניצול CPU מתקדם.
- יכול להריץ מספר סוגים שונים של מערכות הפעלה (Windows, MACOs, UNIX)...

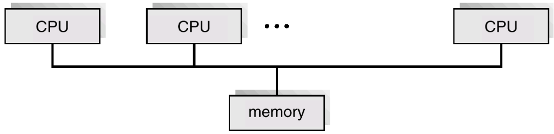
<u>2 קטגוריות של מערכות מחשב</u>

- Single Instruction Single Data (SISD)
  - מעבד יחיד מבצע רצף הוראה בודד להפעלה על המידע המאוחסן בזיכרון בודד.
  - זהו Uniprocessor.
- Multiple Instruction Multiple Data (MIMD)
  - אוסף של מעבדים המבצעים בו זמנית רצפים של הוראות שונות על מאגרי מידע שונים.
  - זהו Multiprocessor.

<u>מערכות מרובות מעבדים – Multiprocessor Systems</u>

- מערכות עם מספר מעבדים בתקשורת סגורה.
- מערכת צמד הדוק (Tightly coupled system)
  - מעבדים חולקים זיכרון ושעון.
  - תקשורת באה לידי ביטוי באמצעות זיכרון משותף.
- יתרונות:
  - תפוקה מוגדלת
  - חסכני
  - אמינות מוגדלת

<u>ארכיטקטורת Multiprocessor</u>

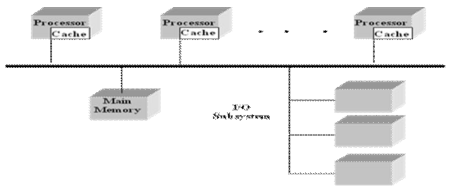


<u>מערכות Multiprocessor</u>

- עיבוד אסימטרי מרובה (*Asymmetric multiprocessing*)
  - מעבד ראשי (master) מתזמן ומקצה עבודה למעבדי המשנה (slave)
- עיבוד סימטרי מרובה (*Symmetric multiprocessing (SMP)*)
  - כל מעבד מריץ עותק זהה של מערכת הפעלה.
  - כל מעבד עושה תזמון-עצמאי מהמאגר של התהליך הזמין
  - רוב מערכות ההפעלה המודרניות תומכות SMP.

<u>עיבוד סימטרי מרובה (Symmetric Multiprocessing (SMP))</u>

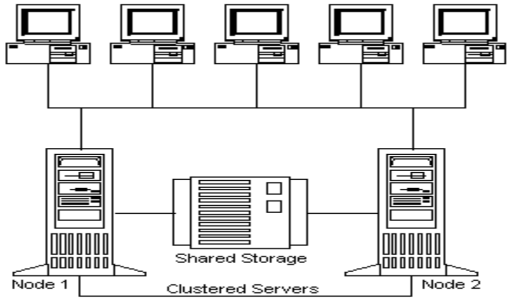
- כל מעבד יכול לבצע פעולות זהות ולשתף את אותו זיכרון ראשי ושירותי I/O (סימטריים).
- מערכת ההפעלה מתזמנת תהליכים / threads לרוחב כל המעבדים (הקבלה אמיתית).
- קיום מעבדים מרובים הינו שקוף למשתמש.
- עליה בצמיחה: פשוט הוסף CPU נוסף !



<u>מערכות מקובצות – Clustered Systems</u>

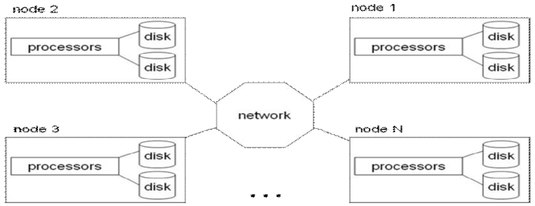
- קלאסטרים מאפשרים לשתי מערכות או יותר לחלוק אחסון חיצוני ולאזן את עומס ה- CPU.
- *Asymmetric clustering:* שרת אחד מריץ אפליקציה בעוד ששרתים אחרים ממתינים.
- *Symmetric clustering:* כל ה- N מארחים מריצים את האפליקציה.

- *Closely coupled system*:
  - למעבדים יש גם זיכרון חיצוני משלהם.
  - התקשורת באה לידי ביטוי בערוצים בעלי מהירות גבוהה.
  - מעניק אמינות גבוהה.



<u>מערכות רשתיות/חלוקה - Networked/Distributed Systems</u>

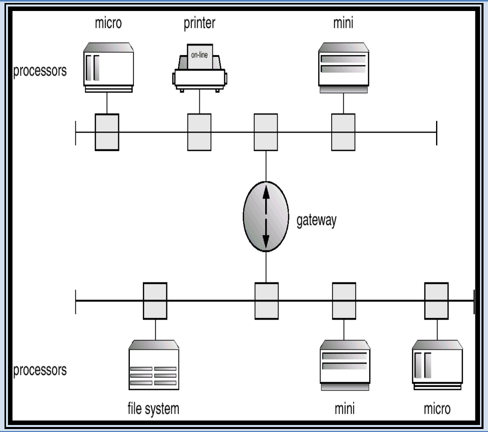
- חלוקת משאבים וחישוביות בקרב מספר מעבדים פיזיים.
- *Loosely coupled system*:
  - לכל מעבד זיכרון מקומי משלו.
  - מעבדים מתקשרים אחד עם השני באמצעות קווי תקשורת שונים.
- יתרונות:
  - שיתוף משאבים
  - האצת חישוביות – שיתוף בעומס
  - אמינות



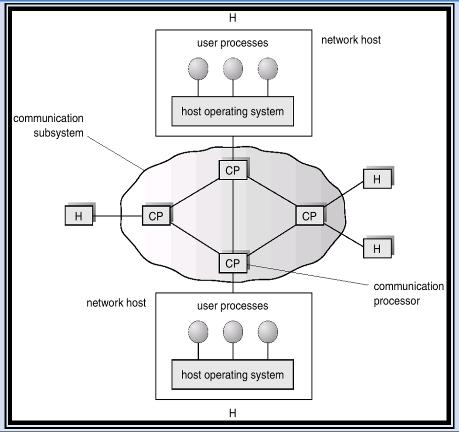
<u>מערכות רשתיות - Networked Systems</u>

- דורשות תשתית רישות.
- Wide Area Network (WAN) או Local Area Network (LAN)
- יכולות להיות שרת ממורכז או לקוח-שרת או מערכות Peer-to-Peer (P2P)

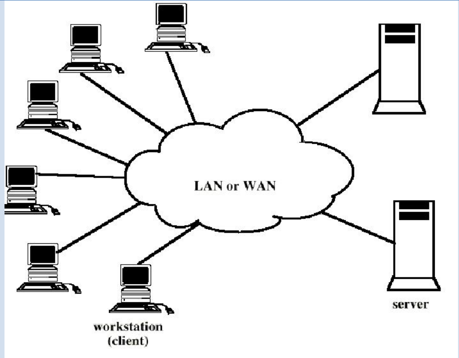
מבנה LAN



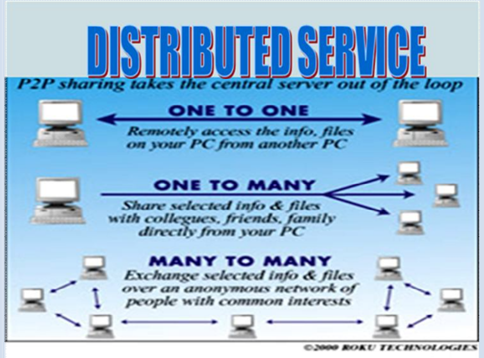
מבנה WAN



סביבת Client-Server



מערכות P2P



<u>מערכות הפעלה של מערכות רשתות/חלוקה</u>

- Network OS (NOS)
  - מעניקה בעיקר שיתוף קבצים
  - כל מחשב רץ עצמאית ברשת
- Distributed OS (DOS)
  - מעניק רושם שינה מערכת הפעלה בודדת השולטת ברשת.
  - רשת היא בעיקרה שקופה – זוהי מערכת וירטואלית עוצמתית.

<u>מערכות זמן אמת - Real-Time Systems (RTS)</u>

- נשים לב שלא כל מערכות ההפעלה הינן מערכות general-purpose (כלל תכליתיות).
- מערכות זמן אמת (Real-Time (RT הינן מערכות ייעודיות הצריכות להיצמד לתאריכי-יעד (לדג' כפיה בזמן).
- דיוק בחישוביות תלוי לא רק בתוצאה הלוגית אלא גם בזמן שבו מופקות התוצאות.

<u>מערכות זמן אמת קשות - Hard Real-Time Systems</u>

- חיובות לעמוד בזמני היעד.
- מתנגשות עם מערכות שיתוף זמן, לא נתמכות ע"י מערכות הפעלה רב תכליתיות.
- בד"כ משמשות כהתקן בקרה באפליקציה ייעודית:
  - בקרה תעשייתית.
  - רובוטיקה
- אחסון משני מוגבל או לא קיים בכלל, המידע מאוחסן בזיכרון זמני, או זיכרון קריאה בלבד (ROM).

<u>מערכות זמן אמת עדינות - Soft Real-Time Systems</u>

- יש צורך בזמני יעד אך אינם מחייבים
- שימושיות מוגבלת בתעשיית הבקרה או הרובוטיקה
- שימושיות באפליקציות מודרניות (מולטימדיה, מציאות מדומה), דורשות תכונות OS מתקדמות.

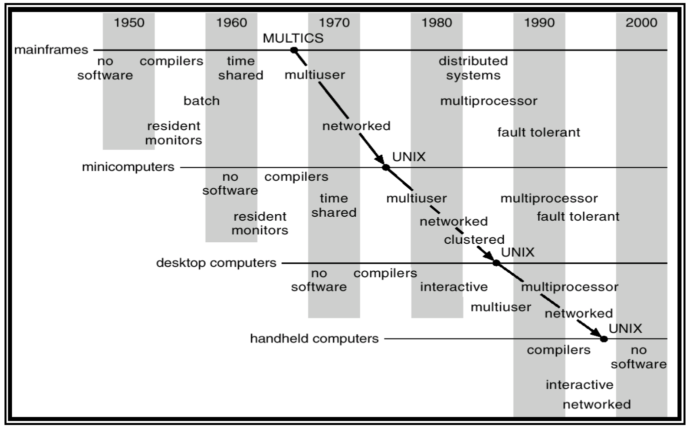
<u>Handheld Systems</u>

מכשירי PDAs, טלפונים סלולאריים.

- סוגיות:
  - זיכרון מוגבל
  - מעבדים איטיים
  - צגים קטנים
  - תמיכה במולטימדיה (תמונות, וידאו)

.

<u>מיגרציה של מונחי OS ותכונות</u>



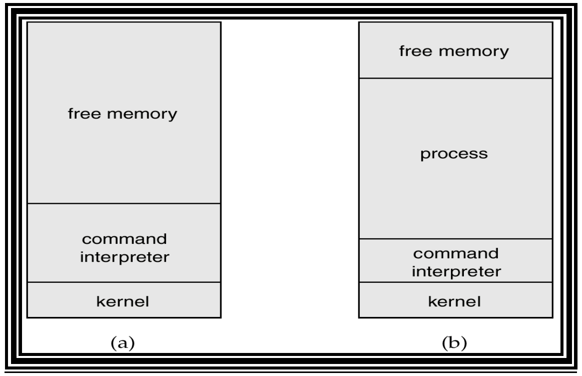
הרצאה 2:

<u>טכניקות ניהול זיכרון ראשוניות</u>

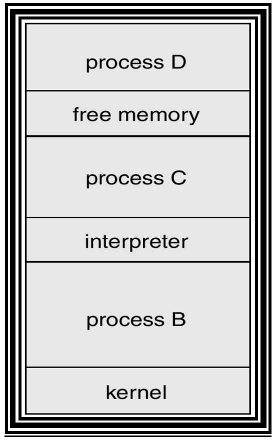
1. מינימום ניהול – תוכנית אחת שמנהלת את הזיכרון לעצמה, אין בעיות הגנת זיכרון.

2. חציית זיכרון – יחידת ניהול הזיכרון (Resident Monitor) ועבודת/תוכנת המשתמש חולקות את אותו הזיכרון.
3. חלוקת זיכרון – מערכת ההפעלה ומספר תוכניות משתמש חולקות את הזיכרון הזמן ביניהן.

<u>MS-DOS memory split</u>



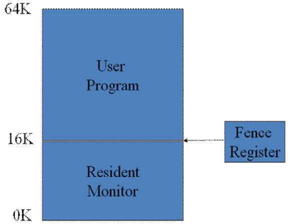
<u>UNIX memory division</u>



<u>ניהול זיכרון</u>

- שיתוף משאבי מערכת דורשים ממערכת ההפעלה לוודא שתוכנית שגויה לא יכולה לגרום לתוכניות אחרות לפעול בצורה שגויה.
- Resident monitor היא תוכנית מהימנה, אך איך אנו מונעים ממנה להינזק ע"י תוכנית משתמש?
- פתרון: רגיסטר ייעודי מגן ושימוש בלוגיקת גישה לכתובות (Addressing access logic).

<u>Memory split</u>



<u>רגיסטר מגן - Fence Register</u>

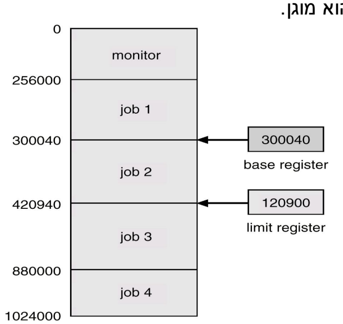
- טעון בבסיס תוכנית המשתמש (שגם מהווה הגבול של Resident monitor).
- תוכנית המשתמש יכולה לקרוא כל כתובת, אך יכולה לכתוב אך ורק לכתובות שהן גדולות יותר מהערך שנמצא ב - Fence Register . זהו מנגנון הלוגיקה של גישה לכתובות.
- פקודה שטוענת את הערך ב- Fence Register חייבת להיות privileged (כלומר בעלת הרשאה, רק המוניטור יכול להריץ פקודה מסוג זה) איך אנו מוודאים זאת?

<u>מצב פעולה דואלי - Dual mode operation</u>

- לספק תמיכת חומרה שמבדילה בין 2 מצבים של פעולות:
  1. User Mode- הרצה נעשית ע"י משתמש.
  2. Monitor Mode – הרצה נעשית ע"י מערכת ההפעלה.
- וידוא שתוכנית משתמש לעולם לא תוכל לקבל שליטה על המחשב במצב Monitor Mode.
- פקודות מסוג privileged יכולות לרוץ רק ב- Monitor Mode.
- פתרון: Mode bit ב- Status register הקובע באיזה מצב המערכת נמצאת (0 עבור מצב מערכת ו-1 עבור מצב משתמש.
- כאשר מתרחשת פסיקה, גורם לכך שהמצב בחומרה יתחלף למצב מערכת (trap) ברוטינת שיחת הנכונה במרחב הכתובות של המוניטור - כך נבטיח בטיחות.
- הערה – האם לאפשר פקודה מסוג Monitor mode? לא!

<u>חלוקת זיכרון - Memory Division:</u>

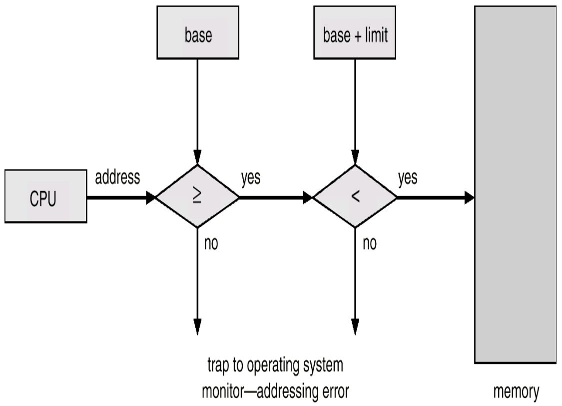
- כדי להבטיח הגנה נוסיף 2 רגיסטרים שקובעים את הטווח הכתובות החוקיות שתוכנה יכולה לגשת אליהן.
  - Base register – שומר את הכתובת הפיסית החוקית המינימאלית של התוכנית.
  - Limit register – שומר את גודל הטווח.
- זיכרון מחוץ לטווח זה הוא מוגן.



<u>הגנת חומרה - Protection Hardware:</u>

- כאשר מריצים במצב מוניטור למערכת ההפעלה יש גישה לא מוגבלת לזיכרון המערכת והמשתמש.
- פקודות הטעינה עבור רגיסטרים base- ו limit הן מסוג privileged (פקודות הקריאה אינן מסוג privileged).
- פקודות מסוג privileged ניתנות להרצה רק במצב מוניטור.

<u>Logic of Protection Hardware</u>



<u>סיכום הגנת זיכרון – Memory Protection</u>

- הגנה על מרחב הכתובות באמצעות Fence registers, Addressing logic access.
- הגנה על Fence registers באמצעות privileged load instruction.
- הבטחת הרצה של פקודה מסוג privileged ב - Monitor Mode ע"י Mode bit.
- הגנה על Mode bit ע"י שינוי המצב ל – Monitor Mode רק ע"י מלכוד חומרה.

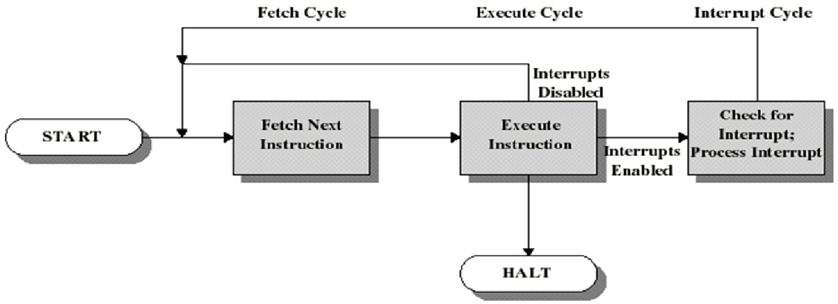
<u>Traps</u>

- פסיקה המיוצרת ע"י תוכנה הנגרמת עקב שגיאה, לדוגמא:
  - גלישה אריתמטית (מלמעלה/למטה)
  - חלוקה באפס
  - הרצת פקודה לא חוקית
  - הפניה מחוץ למרחב הזיכרון של המשתמש.
- כמו-כן מיוצרת בסוף אתחול ע"י פסיקה חיצונית או קריאת מערכת – System call (כל סוגי הפסיקות).

<u>פסיקות חיצוניות</u>

- פסיקה חיצונית היא השהייה זמנית של תהליך ונגרמת עקב אירוע חיצוני לתהליך זה, ומבוצעת בצורה כזאת שהתהליך יכול להמשיך מאותו מקום, לדוגמא:
  - קלט/פלט
  - טיימר
  - כשל בחומרה

<u>Instruction cycle with interrupts</u>



- המעבד בודק אחרי כל פקודה האם יש בקשה לפסיקה.

- אם אין בקשה לפסיקה, הבאת הפקודה הבאה מהתוכנית הנוכחית.
- אם מחכה פסיקה, השהיית ההרצה של התוכנית הנוכחית והפעלת Interrupt Handler.

Interrupt Handler

- תוכנית אשר קובעת את טבע הפסיקה ומבצעת את הפקודות הנדרשות.
- השליטה מועברת לתוכנית זו.
- השליטה חייבת לחזור חזרה לתוכנית שהופסקה על מנת שתהיה אפשרות להמשיך מאותה נקודה בה הופסקה התוכנית.
- נקודת הפסיקה יכולה לקרות בכל מקום בתוכנית, לכן צריך לשמור את מצב התוכנית, כלומר תוכן PC, אוגרים וכו'.

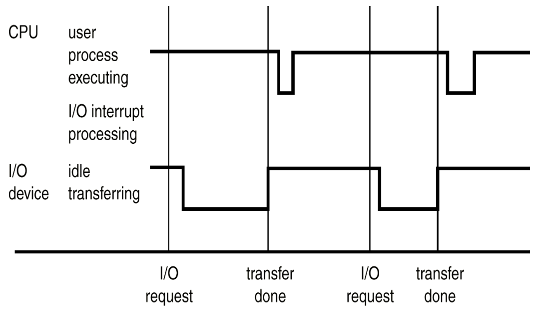
פונקציות נפוצות של פסיקות חיצוניות

- הפסיקה מעבירה את השליטה ל – Interrupt service routine (Interrupt Handler) בדרך כלל דרך וקטור הפסיקה (interrupt vector) אשר מכיל את כל כתובות רוטינות השירות - service routines.
- ארכיטקטורת הפסיקה חייבת לשמור את הכתובת של פקודת הפסיקה.
- כאשר מבוצעת פסיקה, בד"כ אין קבלה של פסיקות נוספות כדי לא לאבד את הפסיקה (lost interrupt).

קלט/פלט מונע פסיקה (Interrupt driven I/O)

- התקני קלט/פלט והמעבד יכולים לרוץ בו-זמנית.
- כל מנהל התקן אחראי על בקרת סוג מסוים של התקן.
- לכל מנהל התקן יש חוצץ לוקאלי (local buffer).
- המעבד מעביר/מקבל מידע מהזיכרון באמצעות חוצים לוקאליים.
- קלט/פלט מועבר מההתקן אל החוצץ הלוקאלי של אותו מנהל התקן שאחראי עליו.
- מנהל התקן מודיע למעבד שהוא סיים את פעולתו ע"י פסיקה חיצונית.

Interrupt Time Line for a Process doing I/O

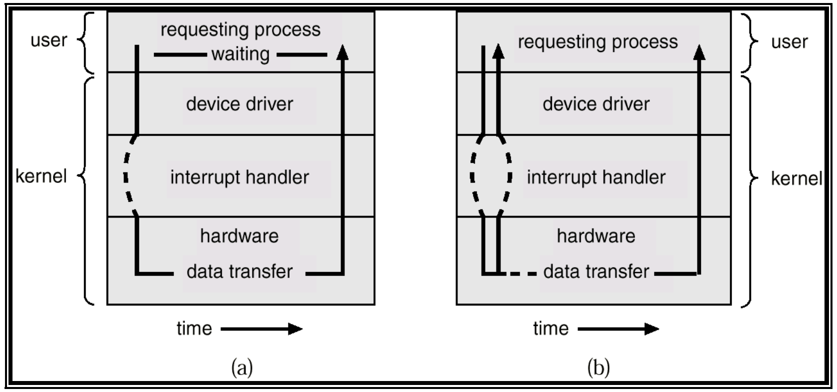


מבנה/שיטות קלט/פלט - I/O Structure/Methods

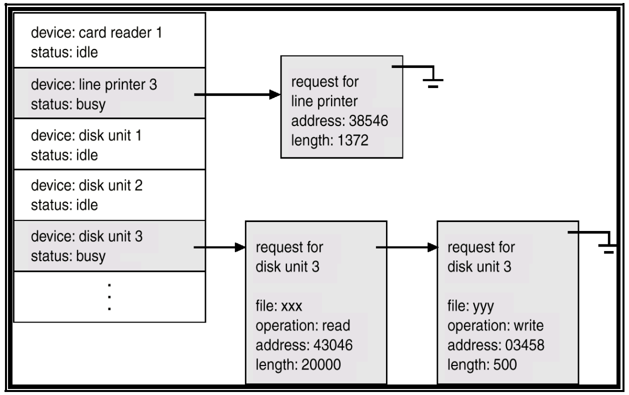
- קלט/פלט סינכרוני: לאחר שקלט/פלט מתחיל, שליטה חוזרת לתכנית המשתמש רק לאחר סיום פעולת קלט/פלט.
  - פקודה משהה את המעבד עד הפסיקה הבאה.
  - לולאה מחכה (מאבק על גישה לזיכרון).

  - ללא עיבוד קלט/פלט סימולטני - רק בקשה אחת גג לא מטופלת מסוג קלט/פלט.
- קלט פלט א-סינכרוני: לאחר שקלט/פלט מתחיל, שליטה חוזרת לתכנית המשתמש ללא סיום פעולת קלט/פלט.
  - בקשה ממערכת ההפעלה להרשות לתכנית המשתמש לחכות לסיום פעולת קלט/פלט - system call.
  - Device status table – מכילה ערך עבור כל התקן קלט/פלט ומכילה את סוגו, כתובת, ומצבו.
  - מערכת ההפעלה פונה לטבלה זו כדי לקבוע את מצבו של ההתקן ולשנות את הערך בטבלה לכלול פסיקה.

סינכרוני א-סינכרוני



Device Status table



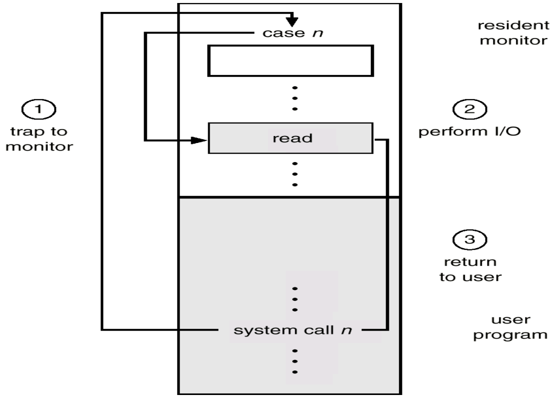
הגנת קלט/פלט – I/O Protection

- כל התקני הקלט/פלט צריכים להיות מוגנים משימוש לרעה ע"י המשתמשים.
- כל פקודות הקלט/פלט הן מסוג privileged.
- בעיה: אם פקודות קלט/פלט הן מסוג privileged, איך תוכנית המשתמש מבצעת פעולת קלט/פלט?
- פתרון: קריאות מערכת (system calls) מתוכניות המשתמש.

קריאות מערכת - System calls

- תהליך מבקש פעולה מסוימת ממערכת ההפעלה:
  1. בד"כ משתמש בצורה של trap למיקום ספציפי ב-וקטור הפסיקה. השליטה עוברת דרך וקטור הפסיקה אל רוטינת השירות במערכת ההפעלה ו ה- Mode bit מאותחל למצב מערכת.
  2. המערכת מוודא שהפרמטרים הם נכונים וחוקיים ומבצעת את הבקשה.
  3. חזרה של השליטה אל הפקודות העוקבות שלאחר קריאת המערכת (system call).

<u>שימוש בקריאת מערכת לביצוע פעולת קלט/פלט - Use of A System Call to Perform I/O</u>

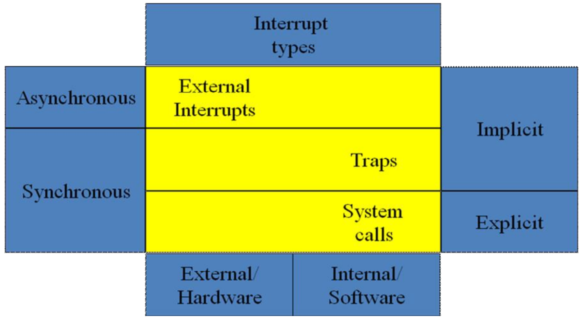


<u>הגנת מעבד – CPU Protection</u>

- טיימר (Timer) – גורם לפסיקה לאחר זמן ספציפי כדי להבטיח שמערכת ההפעלה תשמור על השליטה.
  - הטיימר יורד כל מחזור שעון.
  - כאשר הטיימר מגיע ל-0 מתבצעת פסיקה.
- פקודת טעינה של טיימר היא מוגן privileged.
- שימוש נפוץ בטיימר למימוש Time Sharing.

<u>סוגי פסיקות ותכונות - Interrupt Types and Attributes</u>

- מערכת הפעלה היא מונעת פסיקות:
  - פסיקות חיצוניות – External interrupts.
  - מלכודות – Traps.
  - קריאות מערכת – System calls.
- מגוון תכונות:
  - פסיקות חיצוניות/חומרה הן א-סינכרוניות
  - מלכודות וקריאות מערכת הן פסיקות פנימיות/תוכנה
  - פסיקות פנימיות הן סינכרוניות לתוכנית הרצה.
  - קריאות מערכת הן מפורשות, השאר לא מפורשות.



<u>שירותי מערכת הפעלה - Operating System Services</u>

- הרצת תוכנית – המערכת מסוגלת לטעון תוכנית לזיכרון ולהריץ אותה.
- ביצוע פעולות קלט/פלט – כיוון שתוכניות משתמש אינן יכולות להריץ פעולות אלו ישירות, מערכת ההפעלה מאפשרת למשתמש דרך לבצע פעולות אלו.
- תפעול מערכת הקבצים – יכולת לקרוא, לכתוב, ליצור ולמחוק קבצים.
- תקשורת – החלפת מידע בין תהליכים אשר רצים על אותו המחשב או על מחשבים שונים שמחוברים יחד ע"י חיבור רשת (מימוש ע"י זיכרון שיתופי - shared memory או message passing)
- זיהוי שגיאות – וידוא חישוב נכון ע"י זיהוי שגיאות בחומרת המעבד והזיכרון, התקני קלט/פלט ובתוכניות המשתמש.

<u>מרכיבי מערכת נפוצים – Common System Components</u>

- ניהול תהליך - Process Management
  - תהליך הוא תוכנה בזמן ריצה.
  - לתהליך צריך להקצאת משאבי מערכת (זמן מעבד, זיכרון, קלט/פלט...)
  - מערכת ההפעלה אחראית ליצירה, מחיקה, סנכרון ותקשורת של תהליכים.

- ניהול זיכרון ראשי - Main Memory Management
  - מחליט אילו תהליכים יטענו לזיכרון
  - עוקב אחר החלקים בזיכרון שנמצאים בשימוש וע"י מי.
  - זיכרון וירטואלי – חלקים מהתוכנה והנתונים נשמרים בבלוקים על הדיסק.

- ניהול קבצים – File Management
  - קובץ הוא אוסף של נתונים קשורים, מוגדרים ע"י יוצר הקובץ. בד"כ קבצים מייצגים תוכניות (both source and object forms) ונתונים.
  - מערכת ההפעלה אחראית על הפעילויות הבאות:
    - יצירה ומחיקה של קבצים.
    - יצירה ומחיקה של תיקיות.
    - תמיכה בפרמיטיביים למניפולציית קבצים/תיקיות.
    - מיפוי קבצים לאחסון משני
    - גיבוי קבצים על מדיה יציבה (שאינה פגיעה)

- ניהול התקני קלט/פלט - I/O System Management
  - התקני קלט/פלט הם קשים מאוד לתכנות יעיל ונכון.
  - מערכת ההפעלה מספקת דרייברים (device drivers) שמבצעים פעולות אלו עבור תוכנות.
  - מערכת זו כוללת:
    - מערכת חוצץ-מטמון (buffer-caching system)
    - ממשק דרייבר כללי (general device driver interface)

- דרייברים לרכיבי חומרה ספציפיים

- ניהול אחסון חיצוני - External Storage Management
  - רוב מערכות המחשב המודרניות משתמשות בדיסק כאמצעי אחסון אונליין עבור תוכניות ומידע.
  - מערכת ההפעלה אחראית על הפעילויות הבאות:
    - ניהול מקום חופשי בדיסק
    - הקצאה של מקום אחסון
    - תזמון דיסק

- רשות – Networking
  - התהליכים במערכת קשורים דרך רשת תקשורת (Communication network)
  - תקשורת מתבצעת באמצעות פרוטוקול (protocol)
  - מערכת מקשורת מאפשרת למשתמש גישה למשאבי מערכת מגוונים
  - גישה למשאב משותף מאפשר (Shared resource)
    - האצה של חישובים
    - מעלה את כמות המידע הזמין
    - מגביר אמינות

- מערכת פענוח פקודות - Command-Interpreter System
  - המערכת שקוראת ומבצעת את הפקודות הניתנות ע"י מערכת ההפעלה.
  - דוגמאות: command.com (MS-DOS), shell (UNIX)
  - במערכות windows הממשק הוא על בסיס עכבר ותפריט – WIMP (Windows, Icons, Menu and Pointing device)

- מערכת הגנה - Protection System
  - מכניזם ששולט בגישה של תוכניות, תהליכים, או משתמשים למשאבי מערכת ומשתמש.
  - מכניזם זה מוכרח:
    - להבדיל בין שימוש מוסמך ללא מוסמך
    - לפרט את השליטה שיש לאכוף
    - לספק אמצעי אכיפה

- זיהוי שגיאות - Error Detection/Response
  - Error Detection
    - שגיאות חומרה פנימיות וחיצוניות
      - שגיאות זיכרון
      - כשלון התקן
    - שגיאות תוכנה
      - גלישה אריתמטית
      - גישה אסורה למיקומי זיכרון
  - Response Detection
    - פשוט שלח הודעת שגיאה לאפליקציה
    - תנסה לבצע שוב את הפעולה
    - בטל את הפעולה

<u>Accounting</u>

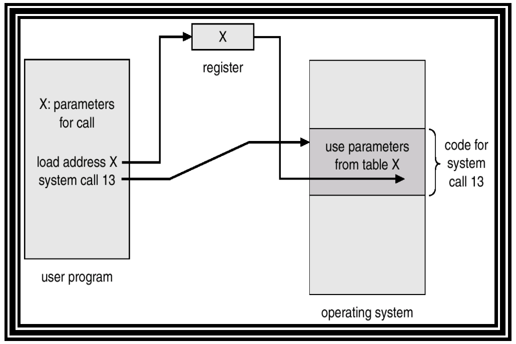
- עוקבת ורושמת עבור כל משתמש כמה ואיזה משאבי מערכת הוא משתמש.
  - אוספת סטטיסטיקות אחרי שימוש במשאב מערכת

  - ביצועי המוניטור - זמן תגובה
  - משמש עבור פרמטרי מערכת לכיוון ושיפור ביצועים
  - שימושי עבור ציפיות לשיפורים עתידיים
  - משמש כאמצעי חיוב משתמשים (על מערכות מולטי-משתמשים)

<u>קריאות מערכת - System calls</u>

- מספקות ממשק בין תוכנית רצה למערכת ההפעלה.
  - בד"כ זמינות כפקודות אסמבלי
  - שפות עיליות כמו C,C++ מאפשרות להחליף פקודות אסמבלי ולבצע קריאות מערכת ישירות דרכן
- שלוש שיטות עיקריות מאפשרות העברת פרמטרים בין תוכנית רצה למערכת ההפעלה:
  1. העברת פרמטרים דרך רגיסטרים.
  2. לשמור את הנתונים בטבלה בזיכרון והכתובת לטבלה מועברת כפרמטר ברגיסטר.
  3. דחיפה של הפרמטרים לתוך המחסנית ע"י התוכנה ושליפת הנתונים ע"י מערכת ההפעלה.

<u>Passing parameters as a table</u>



<u>תוכניות מערכת - System Programs</u>

- תוכניות מערכת מספקות סביבה נוחה לפיתוח תוכנות והרצתן, אפשר לחלקן:
  - מניפולציית/שינוי קבצים
  - מידע מצב - Status information
  - תמיכה בשפות תכנות
  - טעינת תוכניות והרצתן
  - תקשורת
  - תוכנות אפליקציה
- רוב החזות של מערכת ההפעלה בעיניי המשתמשים מאופיינת ע"י תוכניות המערכת, לא קריאות המערכת עצמן.

<u>יעדים בתכנון מערכת - System Design Goals</u>

- יעדי משתמשים – מערכת הפעלה צריכה להיות נוחה לשימוש, קלה ללימוד, אמינה, בטוחה ומהירה.
- יעדי מערכת – מערכת הפעלה צריכה להיות קלה לתכנון, מימוש, ותחזוקה, וגם כן גמישה, אמינה, ללא שגיאות, ויעילה.

<u>מכניזם ומדיניות - Mechanisms and Policies</u>

- מכניזם קובע איך משהו יעשה, מדיניות קובעת מה צריך להיעשות.
- הפרדה ביניהם זהו עקרון חשוב מאוד שמאפשר מקסימום גמישות אם החלטות המדיניות משתנות בעתיד.

<u>מימוש מערכת - System Implementation</u>

- באופן מסורתי כתובה בשפת אסמלי, כיום מערכות הפעלה יכולות להיכתב בשפה עילית.
- קוד שרשום בשפה עילית:
  - יכול להיכתב מהר יותר
  - יותר קומפקטי
  - יותר קל להבנה ולדיבוג.
- הרבה יותר קל להעביר מערכת הפעלה מחומרה אחת לאחרת (port) כאשר היא נכתבת בשפה עילית.

<u>System Generation</u>

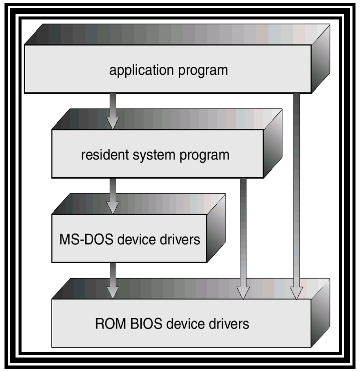
- מערכות הפעלה מתוכננות לעבוד על כל קלאס של מכונות, המערכת חייבת להיות מעוצבת עבור על מחשב ספציפי.
- תוכנית SYSGEN אוספת מידע עבור קונפיגורציית החומרה של המערכת.
- אתחול (Booting) – התחלת המחשב ע"י טעינת ה – kernel.
- Bootstrap program – קוד השמור על זיכרון ROM אשר יכול למצוא את ה - kernel ולהריץ אותו.

<u>מבנה של מערכת הפעלה - Structure of Operating-System</u>

- מבנה/ארגון/מערך של מערכת הפעלה:
  - מונולית' – Monolithic
  - מרבד – Layered
  - Microkernel
- מערכות וירטואליות - Virtual Machines

<u>מבנה של דוס - MS-DOS System Structure</u>

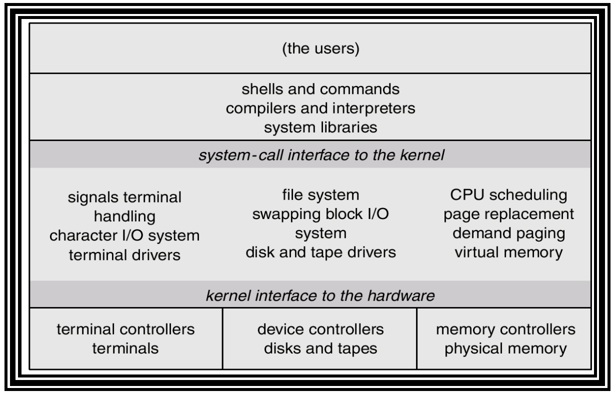
- דוס נכתבה לתת את המקסימום פונקציונאליות במינימום מקום:
  - לא מחולקת למודולים – Monolithic
  - למרות שיש לה מבנה הממשקים והרמות הפונקציונאליות אינן מופרדות



<u>מבנה של יוניקס - UNIX System Structure</u>

- UNIX מוגבלת ע"י פונקציונאליות חומרה, למערכת הפעלה המקורית הייתה מינימום מבנה.
- ל-UNIX 2 חלקים מופרדים:
  - System programs

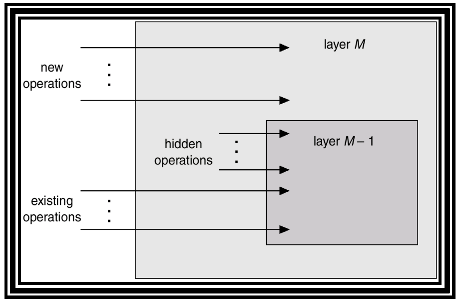
- The kernel
  - מכיל כל מה שנמצא מתחת לממשק קריאות מערכת (system calls) ומעל החומרה.
  - מספק את מערכת ניהול הקבצים, תזמוני המעבד, ניהול הזיכרון, ופונקציות מערכות הפעלה אחרות. מספר רב מאוד של פונקציות לרמה אחת.



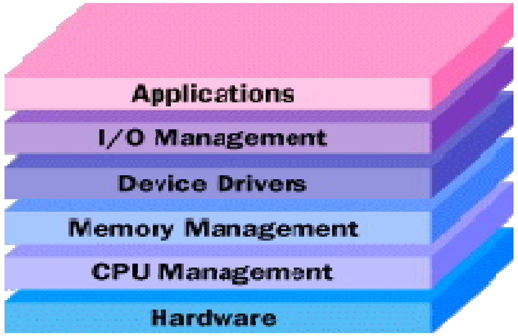
<u>גישת רבוד - Layered Approach</u>

- מערכת ההפעלה מחולקת למספר רבדים (רמות), כל אחד בנוי על רבדים נמוכים יותר. הרמה הנמוכה ביותר היא החומרה (layer 0), הרמה הגבוהה ביותר היא רמת המשתמש (layer N).
- כל רובד יכול להשתמש בפונקציות ושירותים של רבדים נמוכים ממנו.

<u>An operating system layer</u>

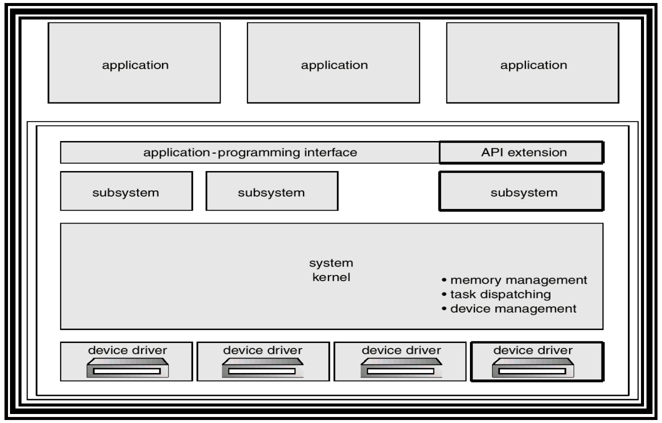


<u>General OS layers</u>



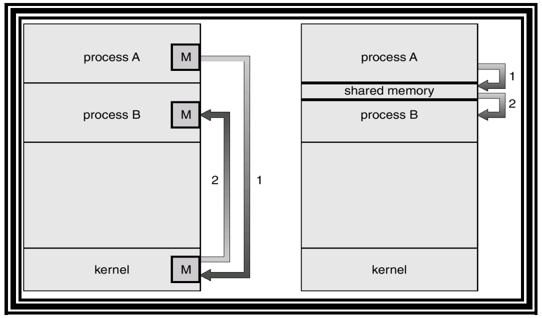
Older Windows System Layers

OS/2 Layer Structure



Communication Models - מודולי תקשורת

- שימוש בזיכרון שיתופי (shared memory) או באמצעות שליחת הודעות (message passing)



Microkernel System Structure

- העברת מקסימום פונקציונאליות מהגרעין (kernel) אל מרחב המשתמש.
- רק מספר פונקציות חיוניות בגרעין:
  - ניהול זיכרון פרימיטיבי
  - ניהול קלט/פלט ופסיקות

  - Inter-Process Communication (IPC)
  - ניהול תזמונים בסיסי

- שירותים אחרים של מערכת ההפעלה מסופקים ע"י תהליכים שרצים במצב משתמש.
  - Device drivers
  - מערכת קבצים
  - זיכרון וירטואלי

- תקשורת מתרחשת בין מודולי משתמש דרך Message passing.
- החלפת קריאות מערכת (system calls) ל- Message passing בין תהליכים גורמת לירידה בביצועים
- יתרונות:
  - גמישות - flexibility
  - קלה להרחבה, אמינות – Extensibility/Reliability
    - תכנון מודולרי
    - קל להוסיף שירותים נוספים
    - גרעין קטן – יותר קל לבדוק בקפידה
  - ניידות – Portability
    - שינויים שיש לעשות כדי להעביר את המערכת למעבד חדש נעשים רק בגרעין ולא בשאר השירותים.
  - תמיכה ב – Distributed system
    - הודעות נשלחות בלי לדעת מכונת היעד
  - מערכת הפעלה מונחית עצמים
    - מרכיבים הם אובייקטים עם ממשקים מוגדרים וברורים שיכולים להתחבר ולהרכיב תוכנה

Mach 3 Microkernel Structure

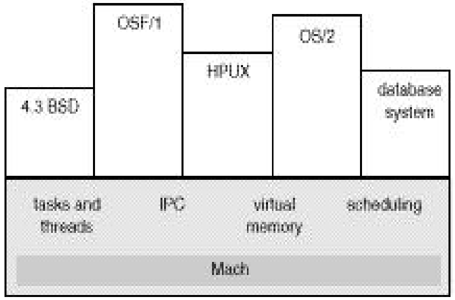
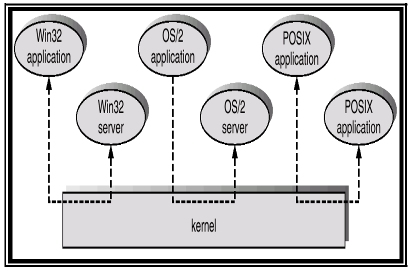


Figure A.1 Mach 3 structure.

Mach 3 Microkernel Structure



Windows NT 4.0 Architecture

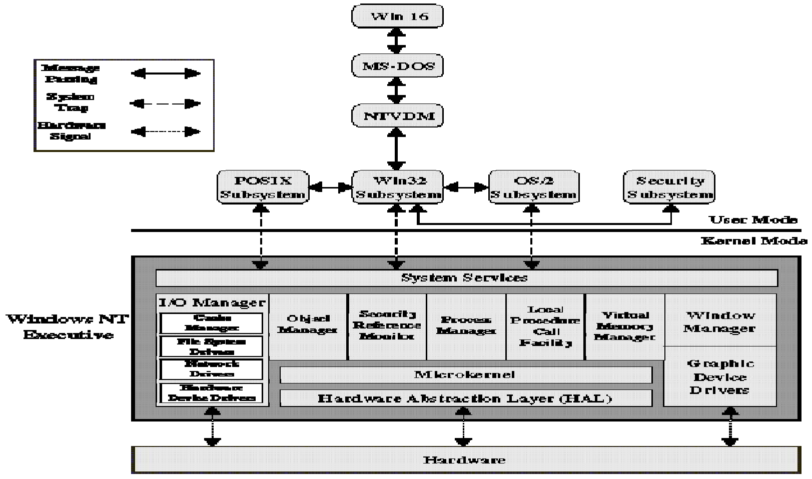
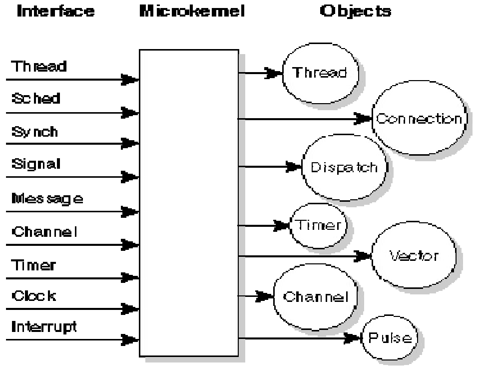


Figure 2.13 Windows NT 4.0 Architecture

The Neutrino Microkernel

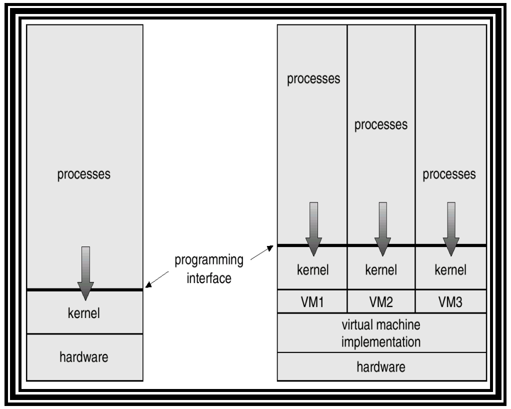


מכונות וירטואליות - Virtual machines

- מכונה וירטואלית לוקחת את גישת הרבדים למסקנה הלוגית שלה. היא מתייחסת לחומרה ולגרעין מערכת ההפעלה כאילו היו הכל חומרה.
- מכונה וירטואלית מספקת ממשק זהה לחומרה המוסתרת.
- מערכת ההפעלה יוצרת את האשליה של ריבוי תהליכים, כל אחד רץ על מעבד משלו עם זיכרון (וירטואלי) משלו.

- המשאבים הפיזיים של המחשב משותפים על מנת ליצור את המכונות הוירטואליות.
    - תזמון המעבד יכול ליצור מראה כאילו לכל משתמש מעבד משלו.
    - File system,Spooling מספקים כרטיסי קריאה וירטואליים וקווי מדפסת וירטואליים.
- טרמינל משתמש זמן-שיתופי נורמאלי משמש כקונסול של המכונה הוירטואלית.

System models



Non Virtual machines Virtual machines

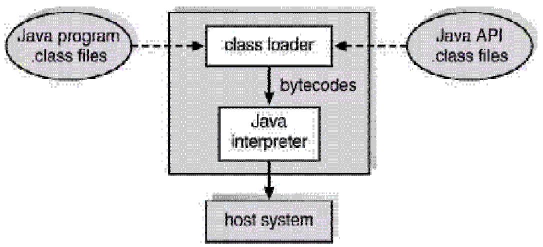
יתרונות וחסרונות של מכונות וירטואליות

- מספקת הגנה מלאה על משאבי מערכת כיוון שכל מערכת וירטואלית מופרדת מכל מערכת וירטואלית אחרת. פירוד זה מאפשר למנוע שיתוף משאבים ישיר.
- פיתוח ומחקר על מכונה וירטואלית הרבה יותר נוח ובטיחותי מאשר על מכונה פיזית כיוון שלא צריך להפריע לפעילות הנורמאלית של המערכת הפיזית.
- קשה למימוש כיוון שצריך לספק העתק של החומרה המוסתרת.

Java Virtual Machine

- תוכנות מקומפלות ב-Java הן platform-neutral byte codes מורצות ע"י Java Virtual Machine.
- JVM מכילה:
    - טוען מחלקה
    - מוודא מחלקה
    - מתרגם זמן ריצה
- JIT Compilers – מגדילים את הביצועים.

The Java Virtual Machine

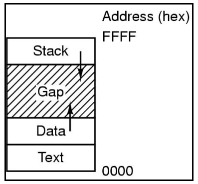


הרצאה 3

הקדמה לתהליכים / משימות (Processes)

- כל מערכות ההפעלה רב-משימתיות בנויות סביב המונח של תהליכים
- תהליך – תכנית בהרצה
  - מונחים דומים: Job, Step, Load Module, Task, Thread
- תהליך כולל 3 סגמנטים:
  - תכנית: קוד / טקסט
  - נתונים: משתני תכנית
  - מחסנית: עבור קריאות לפרוצדורות והעברת פרמטרים

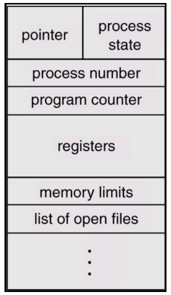
<u>אזורים/סגמנטים בזיכרון עבור תהליך</u>



<u>דוגמה לתכונות של תהליך</u>

- מזהה – Process ID
- מזהה תהליך אב – Parent process ID
- מזהה משתמש – User ID
- סטאטוס תהליך – Process state
- מונה תכנית – Program counter
- אוגרי מעבד – CPU Registers
- אזורים בזיכרון המוקצים לתהליך
- קבצים פתוחים
- עדיפות
- מידע תמחור – Accounting information

<u>בלוק בקרה של תהליך – Process Block Control (PCB)</u>



<u>מרכיבים של ה- PCB</u>

- PCB – נתונים הקשורים עם כל תהליך
- PCB (הקשר ביצוע) הם הנתונים הדרושים (תכונות התהליך) ע"י ה- OS כדי לשלוט בתהליך:
  - נתונים עד מיקום התהליך
  - נתונים על זיהוי התהליך

- נתונים על סטאטוס התהליך
- נתונים על בקרת התהליך

נתוני מיקום התהליך - Process Location Information

- כל תמונת תהליך בזיכרון:
  - יכול שלא "לשבת" בטווח כתובות סמוך (תלוי בסכמת ניהול הזיכרון שבשימוש)
  - יכול להשתמש בן במרווח כתובות פרטי (Private) ומשותף (Shared)
- המיקום נמצא ב- Process Table.
- כדי שה- OS תנהל את התהליך, לפחות חלק מתמונת התהליך חייב לשבת בזיכרון הראשי.

נתוני זיהוי תהליך - Process Identification Information

- ייתכן ויהיה שימוש במספר מזהים נומריים:
  - מזהה תהליך ייחודי – Unique process ID (PID)
    - מאנדקסים (באופן ישירי או עקיף) לתוך טבלת התהליך
  - מזהה משתמש – User ID (UID)
    - המשתמש האחראי לעבודה
  - מזהה של התהליך שיצר את התהליך הזה (PPID)
- אולי שמות סימבולים הקשורים למזהים הנומריים.

נתוני סטאטוס תהליך - Processor State Information

- תכנים של אוגרי התהליך
  - אוגרים הנראים למשתמש (User-visible registers)
  - אוגרי סטאטוס ובקרה
  - מצביעי מחסנית
- מילת בקרה תכנית (Program Status Word – PSW)
  - מכילים נתוני סטאטוס
  - לדג': אוגר ה- EFLAGS במכונות הפנטיום.

נתוני בקרה של תהליך – Process Control Information

- נתוני תזמון ומצב
  - מצב תהליך (לדג', רץ, מוכן, חסום...)
  - עדיפות של תהליך
  - אירוע אליו התהליך ממתין (אם חסום)
- נתוני מבנה נתונים
  - ייתכן ויחזיקו מצביעים ל- PCBs אחרים לתורים של תהליך, קשרים של תהליך בן ומבנים אחרים.

נתוני בקרה של תהליך בתוך PCB - Process Control Information (in PCB)

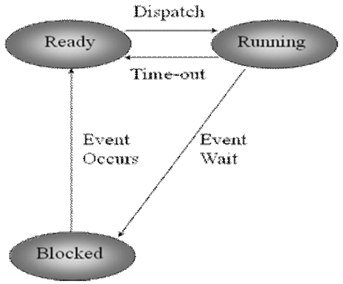
- תקשורת בין תהליכים –
  - ייתכן ותחזיק דגלים וסמלים עבור IPC
- בעלות משאב ותועלת –
  - משאב בשימוש: פתיחת קבצים, התקני I/O
  - היסטוריה של שימוש (של זמן מעבד, I/O ...)
- פריבילגיות תהליך (Access Control) –
  - גישה למיקומים מסוימים בזיכרון, למשאבים וכו'...
- ניהול זיכרון –
  - מצביעים לטבלאות סגמנט/עמוד המוקצים לתהליך זה

מצבי תהליך – Process States

נתחיל עם 3 מצבים:

1. מצב ריצה – Running State –
   a. התהליך שבריצה (CPU יחיד)
2. מצב מוכן – Ready State
   a. כל תהליך המוכן לביצוע
3. מצב חסום/מוכן – Blocked/Waiting state
   a. כאשר תהליך לא יכול להתבצע עד שאירועים מסוימים מתרחשים (לדג', סיום של I/O)

מודל 3 המצבים



מעברי תהליכים

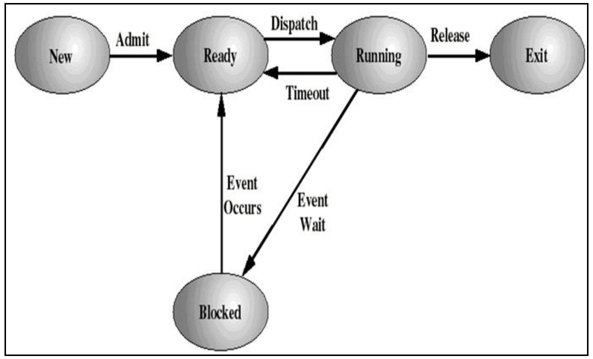
- Ready → Running
  - כאשר הגיע הזמן, המשגר בוחר להריץ תהליך חדש
- Running → Ready
  - התהליך שרץ סיים את חריץ הזמן שלו
  - התהליך שרץ מופסק (Interrupt) מכיוון שישנו תהליך בעל עדיפות גבוהה יותר במצב Ready.
- Running → Blocked
  - כאשר תהליך מבקש משהו שבשבילו צריך להמתין:
    - שירות (Service) שה- OS אינה מוכנה לבצע
    - גישה למשאב אינה זמינה עדיין
    - מאתחל I/O וחייב להמתין לתוצאה
    - מחכה שהתהליך יעניק קלט
- Blocked → Ready
  - כאשר האירוע שאליו הוא מחכה מתרחש

מצבים שימושיים אחרים - Other Useful States

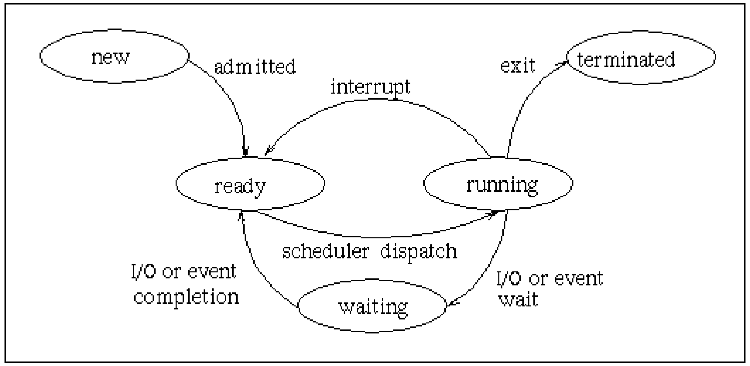
- מצב חדש – New State –
  - ה- OS ביצעה את הפעולות הנחוצות כדי ליצור את התהליך:

- יצרה מזהה תהליך
- יצרה טבלה הדרושה לניהול התהליך
  - אך לא התחייבה לבצע את התהליך (עוד לא הוכנסה)
    - מכיוון שהמשאבים מוגבלים
- מצב יציאה/סיום – Exit/Terminated State
  - סיום תכנית מעביר את התהליך לטבלה זו
  - לא ראויה יותר לביצוע
  - טבלאות ונתונים אחרים שמורים זמנית עבור תכנית סיוע –
    - לדג': תכנית ניהול חשבונות הצוברת שימוש משאב לצורך חיוב משתמשים
- התהליך (והטבלאות שלו) נמחקים כשאין יותר צורך במידע.

מודל 5 המצבים



ובמבט נוסף:



מתי תהליך נוצר ?

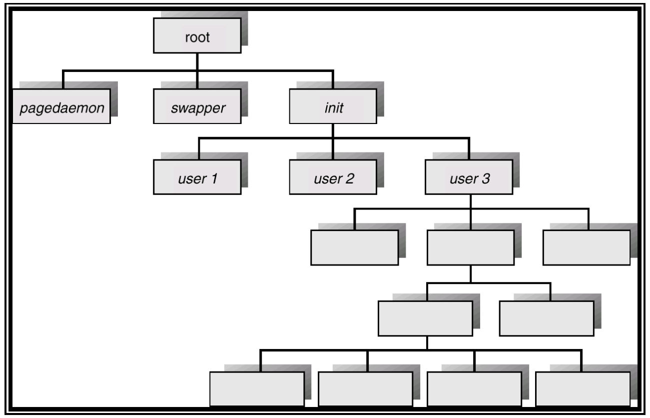
- מסירה של עבודת אצווה (batch job)
- משתמש מתחבר
- נוצר ע"י ה- OS כדי להעניק שירות למשתמש (לדג' הדפסת קובץ)
- ע"י תהליך קיים -
  - תכנית משתמש יכולה להכתיב יצירה של מספר תהליכים

יצירת תהליך

- תהליך אב יוצר תהליכי בנים, שבתורם יוצרים תהליכים אחרים ויוצרים עץ תהליכים.
- שיתוף משאב:
  - אב ובנים חולקים את כל המשאבים
  - בנים חולקים חלק ממשאבי האבות
  - אב ובן לא חולקים משאבים
- ביצוע:
  - אב ובנים מתבצעים במקביל
  - אבות מחכים לסיום הבנים

- מרחב כתובות (Address Space)
  - בן שכפול של האב
  - לבן יש תכנית הטעונה לתוכו
- דג' ב- UNIX:
  - קריאת מערכת **fork** יוצרת תהליך חדש
  - קריאת מערכת **exec** בשימוש לאחר **fork** כדי להחליף את מרווח הזיכרון של התהליך בתכנית חדשה.

עץ תהליכים במערכת UNIX



- הקצאת מזהה תהליך ייחודי
- מקצה מרווח לתמונת תהליך
- מאתחל בלוק בקרה של תהליך
  - הרבה ערכי ברירת מחדל (לדג', מצב הוא New, אין התקני I/O או קבצים...)
- הגדרת קישורים מתאימים
  - לדג': הוספת תהליך לרשימה מקושרת משמשת לתזמון התור

<u>מתי תהליך מסתיים ?</u>

- עבודת אצווה המוציאה הוראת Halt
- משתמש מתנתק
- תהליך מבצע בקשת שירות לסיום
- אב הורג (kill) תהליך בן
- תנאי Error ו- fault

<u>סיבות לסיום תהליך</u>

- סיום נורמאלי
- חריגה מהגבלת הזמן
- זיכרון לא זמין
- הפרה של הגבלות זיכרון
- שגיאת הגנה – לדג': כתיבה לקובץ שהוא לקריאה בלבד
- שגיאה אריתמטית
- Time overrun
  - תהליך המתין יותר מהמאקסימום לאירוע
- כישלון I/O
- הוראה שגויה – מתרחש כאשר מנסים לבצע נתונים
- הוראת פריבילגיה
- התערבות ה- OS – כמו למשל כשמתרחש קיפאון
- אב מבקש לסיים את הבן
- אב מסתיים ככה שתהליך הבן מסתיים

<u>סיום תהליך ב- UNIX</u>

- תהליך מבצע את ההוראה האחרונה ומבקש מה- OS לסיים (**exit**)
  - פלט נתונים מהבן לאב (באמצעות **wait**)
  - Process' resources are deallocated by operating system.
- אב מסיים ביצוע של תהליך בן (**abort**)
  - בן חרג מהקצאת המשאבים
  - משימה שהוקצה לבן אינה דרושה יותר
  - אב יוצא:
    - ה- OS לא מרשה לבן להמשיך אם האב סיים
    - סיום שרשרת (cascade)

מעבר בין משימות וזימון

<u>תזמונים – Schedulers</u>

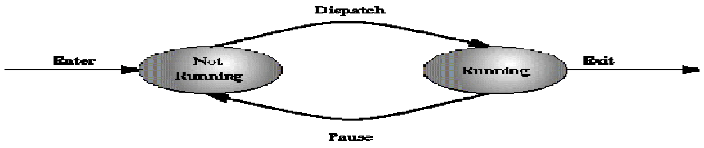
1. Long-term scheduler (or jobs scheduler) - בוחר אילו תכניות / תהליכים יובאו לתוך התור המוכן.
2. Medium-term scheduler (or emergency scheduler) – בוחר איזו תכניות / תהליך יוחלף כאשר המערכת טעונה.
3. Short-term scheduler (or CPU scheduler) – בוחר איזה תהליך יבוצע להבא ויקציב CPU.

<u>Long-term scheduler</u>

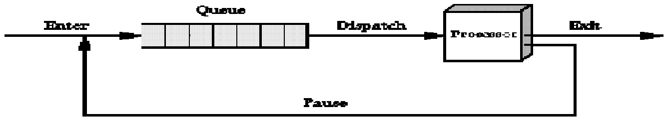
- קובע אילו תכניות יוכנסו למערכת לעיבוד.
- שולט על דרגת ה multiprogramming.
- אם מוכנסים עוד תהליכים:
  - סביר להניח שכל התהליכים יחסמו – שימוש טוב יותר ב CPU.
  - לכל תהליך יש חלק קטן יותר ב CPU.
- יכול לשמור תערובת של עיבוד – תהליכי bound ו- I/O bound.

<u>Short-Term Scheduling</u>

- קובע איזה תהליך יבוצע להבא (נקרא גם CPU scheduling)
- ידוע גם כמשגר (שהוא חלק ממנו)
- נקרא במקרה שיכול להוביל לבחירת תהליך אחר לביצוע:
  - פסיקות שעון
  - פסיקות I/O
  - קריאות OS ומלכודות.
  - סימנים (sginals)



(a) State transition diagram



(b) Queuing diagram

<u>תזמונים – המשך...</u>

- ה- Long-term scheduler נקרא לעיתים לא קרובות (seconds, minutes) ⇒ (may be slow).
- ה- Short-term scheduler נקרא לעיתים קרובות (must be fast) ⇒ (milliseconds).

<u>Medium-Term Scheduling</u>

- עד כה, כל התהליכים היו חייבים להיות (או לפחות חלקם) בזיכרון הראשי.
- אפילו עם זיכרון וירטואלי, שמירת תהליכים רבים יגרע מטיב המערכת.
- ה- OS צריכה להחליף תהליכים לדיסק, ואז להחליפם חזרה פנימה.
- החלטות ההחלפה מבוססות על הצורך לנהל multiprogramming.

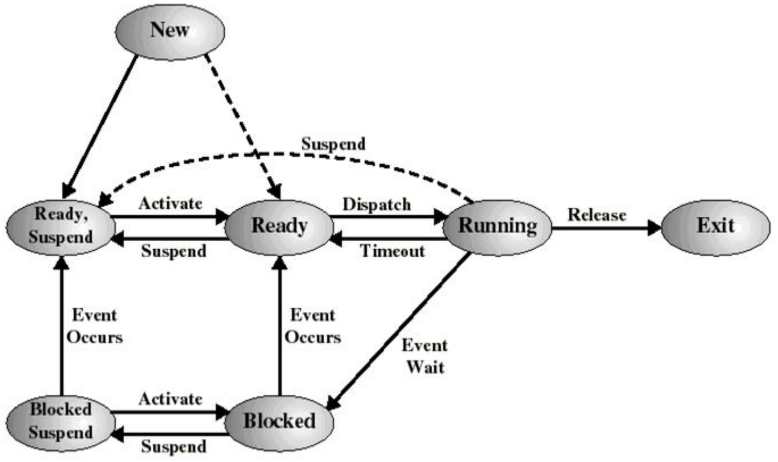
<u>החלפות דינאמיות - Dynamics of Swapping</u>

- תהליך יכול להיות מוחלף זמנים מחוץ לזיכרון ל- *backing store*, ואז להיות מובא חזרה לזיכרון להמשך ביצוע.
- Backing store – דיסק מהיר וגדול דיו לאכסן עותקים של בבואות הזיכרון עבור כל המשתמשים; חייב להעניק גישה ישירה לבבואות זיכרון אלו.
- חלקים נרחבים מזמן ההחלפה הוא זמן ההעברה (transfer time); זמן ההעברה הכללי הינו פרופורציונאלי לכמות הזיכרון שהוחלף.

<u>תמיכה להחלפה - Support for swapping</u>

- ה- OS צריכה להשעות כמה תהליכים, לדג', להחליפם החוצה לדיסק ואז להחליפם חזרה פנימה.
- אנו מוסיפים 2 מצבים:
  - Blocked Suspend: תהליכים חסומים שהוחלפו החוצה לדיסק.
  - Ready Suspend: תהליכים מוכנים שהוחלפו החוצה לדיסק.

<u>מודל 7 השלבים של תהליך</u>



<u>מעברי מצב חדשים - New state transitions</u>

- Blocked --> Blocked Suspend
  - כאשר כל התהליכים חסומים, ה- OS יעשה מקום בזיכרון כדי להביא תהליך מוכן.
- Blocked Suspend --> Ready Suspend

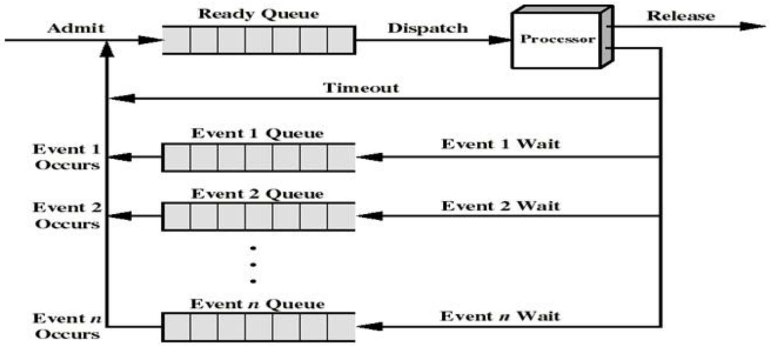
  - כאשר האירוע לו מחכים מתרחש (מידע על המצב זמין ל OS).
- Ready Suspend --> Ready
  - כאשר אין עוד תהליכים מוכנים בזיכרון
- Ready--> Ready Suspend (לא סביר)
  - כאשר אין תהליכים חסומים וחייבים לשחרר זיכרון לאיכות הולמת.

<u>משגר - Dispatcher (short-term scheduler)</u>

- תכנית של ה- OS המעבירה את המעבד מתהליך אחד לאחר.
- מונע מונופול של עיבוד יחיד
- מחליט מי ממשיך הלאה לפי אלגוריתם התזמון.
- ה- CPU תמיד יריץ הוראות מהמשגר בזמן ההחלפה מתהליך A לתהליך B.

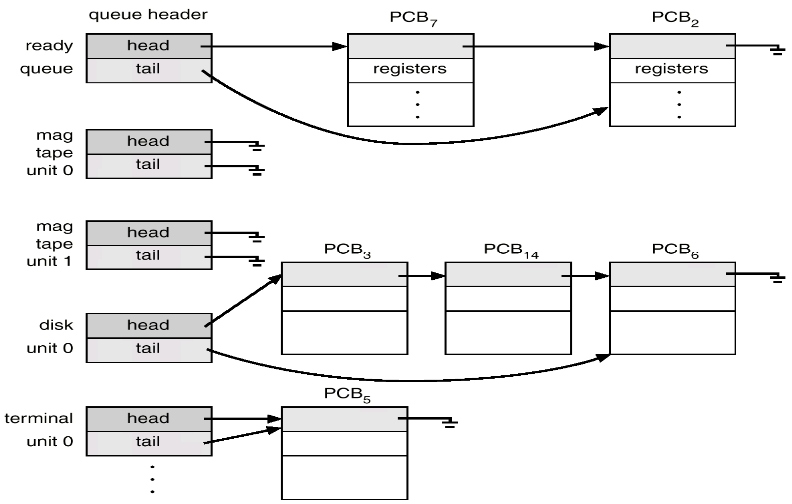
<u>תורים שונים של תהליך - Various Process Queues</u>

- Process queue – אוסף של כל התהליכים במערכת.
- Ready queue – אוסף של כל התהליכים השוכנים בזיכרון הראשי, מוכנים וממתינים להרצה.
- Device queues – אוסף של כל התהליכים הממתינים להתקן O/I.



- When event n occurs, the corresponding process is moved into the ready queue

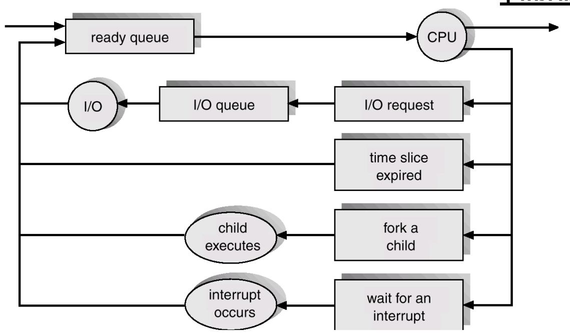
תור מוכן ותורים שונים של התקני I/O - Ready Queue and various I/O Device Queues



מתי למתג (switch) תהליך ?

- מיתוג תהליך יכול להתרחש בכל פעם שה- OS השיגה שליטה על המעבד, למשל:
  - Supervisor Call
    - בקשה מפורשת ע"י התכנית (לדג,' פתיחת קובץ).
      סביר להניח שהתהליך יהיה חסום.
  - Trap
    - שגיאה שתוצאתה מההוראה האחרונה.
      יכול לגרום לתהליך לעבור למצב Exit.
  - Interrupt
    - הגורם הוא חיצוני לביצוע של ההוראה הנוכחית.
      השליטה מועברת ל- Interrupt Handler.

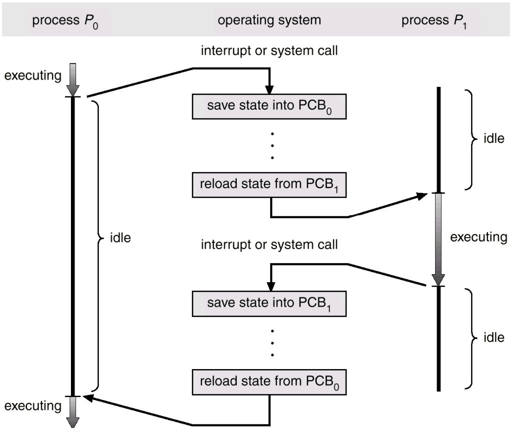
סיבות למיתוג תהליך



מיתוג הקשר - Context Switch

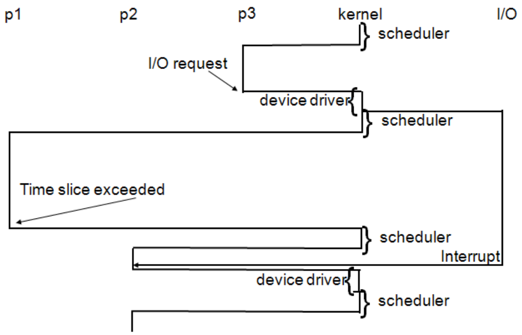
- כאשר ה- CPU ממתג לתהליך אחר, המערכת חייבת לשמור את המצב של התהליך הישן ולטעון את המצב השמור עבור התהליך החדש.
- זה נקרא מיתוג הקשר context switch.
- הזמן שזה לוקח תלוי בתמיכה החומרה.
- זמן Context-switch הוא תקורה;
  המערכת לא עושה עבודה שימושית בזמן המיתוג.

מעבד ממתג בין תהליכים



שלבים במיתוג תוכן - Steps in Context Switch

- שמירת הקשר של תהליך כוללת מונה תכנית ואוגרים אחרים.
- עדכון ה- PCB של התהליך הרץ עם המצב החדש שלו ומידע משותף נוסף.
- העברת PCB לתור המתאים – ready, blocked.
- בחירת תהליך נוסף לביצוע.
- עדכון ה- PCB של התהליך הנבחר.
- שחזור הקשר CPU מזה של התהליך הנבחר.



החלפת אופן - Mode Switching

- יתכן ויקרה שפסיקה לא תייצר מיתוג הקשר.
- השליטה יכולה לחזור לתכנית הפסיקה.

- אז רק נתוני מצב המעבד צריכים להישמר במחסנית.
- זה נקרא mode switching (משתמש למצב קרנל כאשר נכנסים לתוך Interrupt Handler).
- פחות תקורה: אין צורך לעדכן את ה PCB כמו עבור מיתוג הקשר.

3-3 מבוא למשימות ו- threads

מאפייני תהליך - Process Characteristics

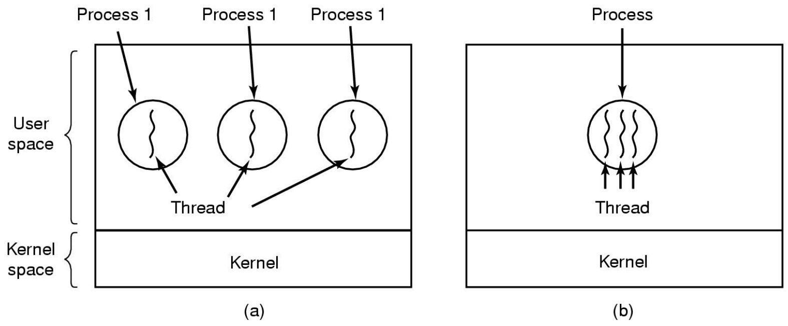
- יחידה של בעלות משאב – לתהליך מוקצה:
  - מרווח כתובת כדי להחזיק את בבואת התהליך
  - שליטה על כמה משאבים (קבצים, התקני I/O...)
- יחידה של שיגור – תהליך הוא נתיב הרצה דרך תכנית אחת או יותר:
  - ביצוע יכול להיות מופרד מתהליכים אחרים
  - לתהליך יש מצב ביצוע ועדיפות שיגור.

- שני מאפיינים אלו מטופלים באופן עצמאי ע"י כמה מערכות הפעלה:
  1. היחידה של בעלות המשאב בד"כ מיוחסת כמשימה או (מסיבות היסטוריות) גם כתהליך.
  2. היחידה של שיגור בד"כ מיוחסת כסיב (Thread) או תהליך קל משקל Light-Weight Process (LWP).
- יכולים להיות מספר Threads באותה משימה.
- ה- Weight Process (HWP)- זהה למשימה עם Thread אחד.
- אך המושגים משימה ותהליך משמשים כמושגים בלתי ניתנים לחלופה (מעורר רחמים)

משימות ו- Threads - Tasks and Threads

- פריטי משימות (משותפים ע"י כל ה- threads במשימה):
  - מרווח כתובת המחזיק את בבואת התהליך
  - משתנים גלובליים.
  - גישה מאובטחת לקבצים, I/O ומשאבים אחרים.
- פריטי Thread:
  - מצב הרצה (running, ready...)
  - מונה תכנית, אוסף אוגרים
  - מחסנית הרצה

תהליך כמול מודל Thread ?



a - שלושה תהליכים כ"א עם Thread אחד
b – תהליך אחד עם 3 Threads

תועלת אפליקציה מ- Threads - Application benefits of threads

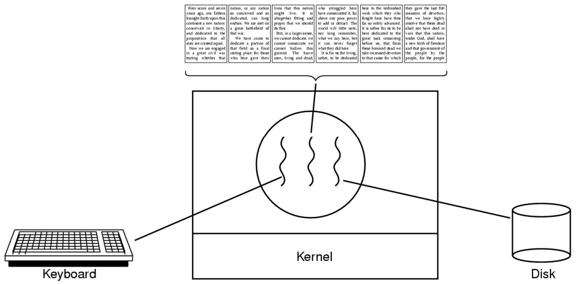
- נניח ויש אפליקציה המורכבת ממספר חלקים עצמאי שלא צריכים לרוץ ברצף.
- כל חלק יכול להיות מיושם כ- Thread.
- כאשר Thread אחד נחסם ממתין ל- I/O, יתכן וההרצה תמתג ל- Thread אחר מאותה אפליקציה (במקום לחסום אותה ולהחליף לאפליקציה אחרת).

- מכיוון ש- Threads מאותו תהליך חולקים זיכרון וקבצים, הם יכולים לתקשר אחד עם השני מבלי לקרוא ל- Kernel.
- לכן חיוני לסנכרן את הפעילות של Threads שונים כך שלא יקבלו ראיה לא עקבית של המידע הנוכחי.

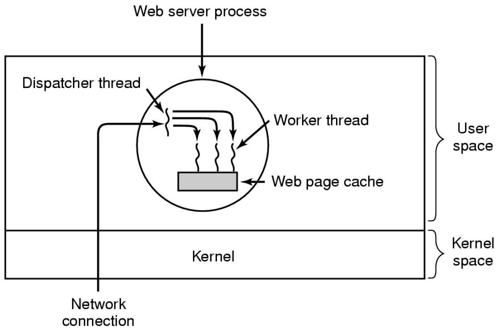
דג' לתועלת של Threads - Examples of benefits of threads

- דג' 1: Thread אחד מציג תפריד וקורא את קלט המשתמש בזמן שה- Thread האחר מריץ פקודות משתמש – האפליקציה יותר responsive
- דג' 2: שרת קבצים/WEB בתוך ה- LAN.
  - צריך לטפל במס' בקשות קבצים/עמודים בזמן קצר.
  - לכן יעיל יותר ליצור (ולהרוס) Thread יחיד עבור כל בקשה.
  - במערכת SMP: מספר threads יכולים להתבצע סימולטנית במעבדים שונים.

דג' לשימוש ב- Thread – Word Processor



דג' לשימוש ב- Thread – WEB Server



Web Server Threads

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

(a) Dispatcher thread
(b) Worker thread

התועלת של Threads - Benefits of Threads

Threads מאפשרים פעילות מקבילית בתוך מרווח כתובת בודד:

- כאשר Tread שרת אחד נחסם וממתין, Thread שני באותה משימה יכול לרוץ.
- פחות זמן ליצור ולסיים Thread מאשר תהליך (מכיוון שלא צריכים עוד מרווח כתובת).
- פחות זמן למתג בין 2 Threads מאשר בין תהליכים.
- תקשורת בין Threads וסנכרון היא מאוד מהירה.

תועלות מובנות - Perceived Benefits

- שיתוף משאבים
- תגובתיות
- חסכוני
- ניצול של ארכיטקטורות מרובות תהליכים
- ניצול של ארכיטקטורות מרושתות

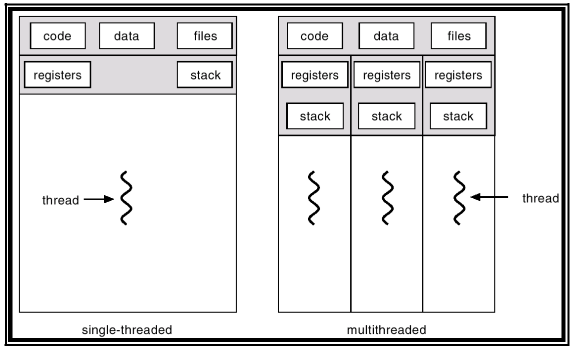
מאפיינים של Threads - Thread Characteristics

- יש לו מצב/הקשר ביצוע
- הקשר ה- Thread נשמר כאשר הוא לא מתבצע
- בעל מחסנית ביצוע וגם אחסון עבור כל Thread עבור משתנים מקומיים.
- בעל גישה למרחב הזיכרון ולמשאבים הקשורים למשימה שלו:
  - כל Threads של משימה חולקים את זה.
  - כאשר Thread מסוים משנה רכיב זיכרון (לא פרטי), כל שאר ה- Threads רואים זאת.
  - קובץ פתוח עבור Thread מסוים, זמין לכל האחרים.

מצבים של Threads - Threads States

- 3 מצבים משמעותיים: running, ready, blocked.
- אין להם מצב השהיה מכיוון שכל ה- Threads שבאותה משימה חולקים את אותו מרווח כתובות.
  - אכן: השהיה (לדג' החלפה) של Thread בודד כורך השהיה של כל ה- Threads באותה משימה.
- סיום של משימה, מסיים את כל ה- Threads באותה משימה.

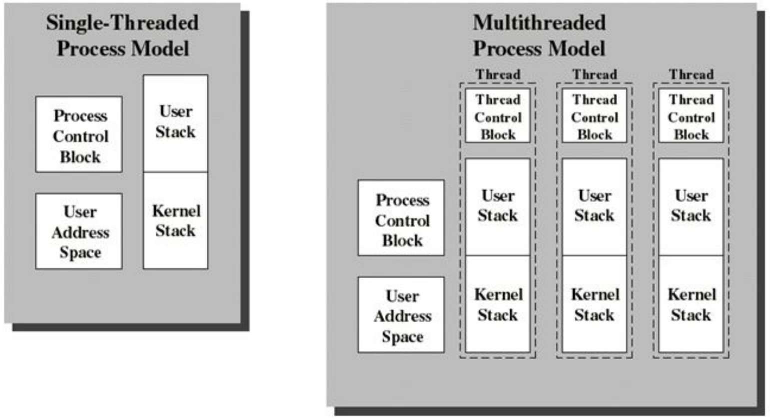
תהליכים יחידים ותהליכים מרובי Threads - Single and Multithreaded Processes



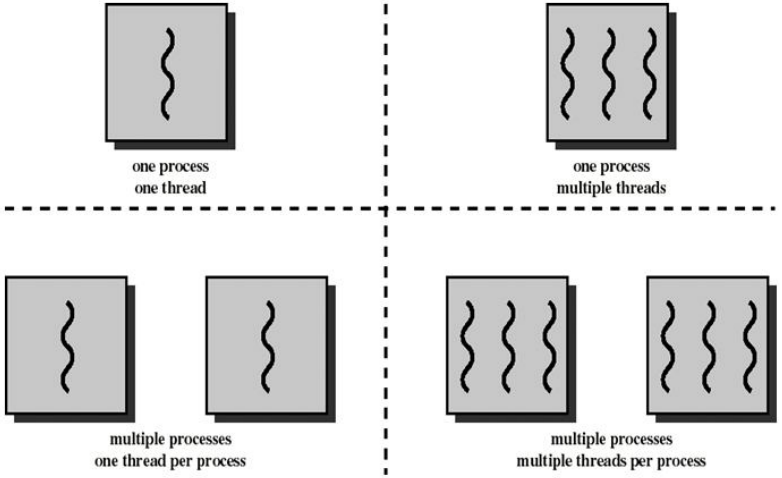
ריבוי Threads כנגד Thread בודד - Multithreading vs. Single threading

- Single threading: כאשר ה- OS לא מזהה את קונספט ה- Thread.
- Multithreading – כאשר ה- OS תומכת בביצוע מס' מרובה של Threads בתוך משימה בודדת.
- ה- MS-DOS תומך במשימת משתמש בודדת ו- single thread.
- UNIX מבוגרות יותר תומכות במס' מרובה של תהליכי משתמש אך תומכת רק ב- Thread אחד לכל משימה.
- מערכות Solaris ו- Windows NT תומכות ב- Threads מרובים.

Single Threaded and Multithreaded Models



Combinations of Threads and Processes



4-3

מודלים/רמות של Multithreading - Multithreading Levels/Models

- רמות
  - ברמת המשתמש ((User-Level Threads (ULT)
  - ברמת הליבה ((Kernel-Level Threads (KLT)
  - גישה מעורבת (Hybrid ULT/KLT Approach)
- מודלים
  - Many-to-One
  - One-to-One
  - Many-to-Many

User-Level Threads (ULT)

- ה- Kernel אינו ער לקיומם של Threads.
- כל ניהול ה- Thread נעשה ע"י אפליקציה תוך שימוש בספריית ה- Thread.
- מיתוג Thread לא דורש פריבילגיות מה- Kernel.
- Scheduling היא אפליקציה ספציפית.

המשך...

- ספריית ה- Thread מכילה קוד עבור:
  - יצירה והריסה של Threads.
  - העברת הודעות ונתונים בין Threads.
  - תזמון ביצוע של Thread.
  - שמירה ושחזור של Threads.

פעילות ה- Kernel עבור ULTs - Kernel activity for ULTs

- ה- Kernel לא מודע לפעילות ה- Threads אך עדיין הוא מנהל את פעילות התהליך.
- כאשר Thread מבצע System Call, כל המשימה תחסם.
- אך עבור ספריית ה- Thread, ה- Thread הזה עדיין במצב running.
- אז מצבי ה- Thread לא תלויים במצב התהליך.

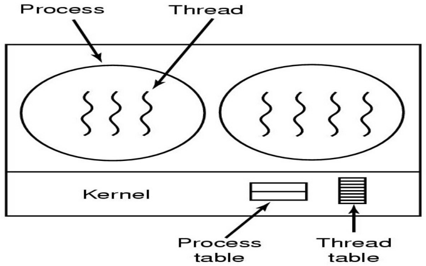
יתרונות וחוסר נוחות של ULT - Advantages and inconveniences of ULT

יתרונות	אי נוחות
• מיתוג Thread אינו כרוך ב- Kernel: אין מיתוג מצב. • תזמון יכול להיות אפליקציה ספציפית: בחר את האלגוריתם הטוב ביותר. • ULTs יכולים לרוץ על כל OS. רק צריך את ספריית ה- Thread.	• רוב ה- System Calls חוסמות וה- Kernel חוסם תהליכים. כך שכל ה- Threads בתוך התהליך יחסמו. • ה- Kernel יכול רק להקצות תהליכים למעבדים. 2 Threads בתוך אותו תהליך לא יכולים לרוץ בו זמנית על 2 מעבדים.

Kernel-Level Threads (KLT)

- כל ניהול ה- Thread נעשה ברמת ה- Kernel.
- אין ספריית Thread אך ישנו API ל- Kernel.
- ה- Kernel מחזיק מידע הקשור לתהליך ול- Threads.
- מיתוג בין ה- Threads דורש את ה- Kernel.
- תזמון על בסיס Thread.

יישום Threads ב- Kernel - Implementing Threads in the Kernel



- Threads supported by the Kernel.
- Examples
  - Windows NT/2000/XP
  - OS/2
  - Solaris 2
  - Tru64 UNIX
  - BeOS
  - Linux

Linux Threads

- מתייחס אליהם כמשימות (tasks) יותר מאשר כ- Threads (או processes).
- יצירת Thread נעשית באמצעות System Call - clone().
- ה- clone() מאפשר למשימת בן לחלוק את מרווח הכתובות של משימת האבא (תהליך).
- שיתוף זה של מרווח הכתובות מאפשר למשימת הבן המשוכפלת להתנהג כמו Thread נפרד.

Advantages and inconveniences of KLT

יתרונות	אי נוחות
• ה- Kernel יכול לתזמן סימולטנית הרבה Threads מאותו תהליך על הרבה מעבדים. • חסימה נעשית ברמת ה- Thread. • שגרת ה- Kernel יכולה להיות multithreaded	• מיתוג Threads בתוך אותו תהליך כרוך ב- Kernel. ישנם 2 מצבי מיתוג לכל מיתוג thread. • יוצר איטיות משמעותית.

גישה משולבת - Hybrid ULT/KLT Approaches

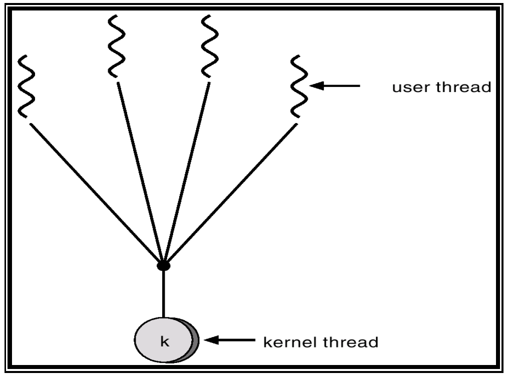
- יצירת ה- thread נעשית במרווח המשתמש.
- רוב התזמון והסנכרון של ה- threads נעשה במרווח המשתמש.
- המתכנת יתאים את מס' ה- KLTs.
- יתכן וישלב את הטוב ביותר של 2 השיטות.
- לדג' Solaris.

מודלים

- Many-to-One
- One-to-One
- Many-to-Many

<u>Many-to-One Model</u>

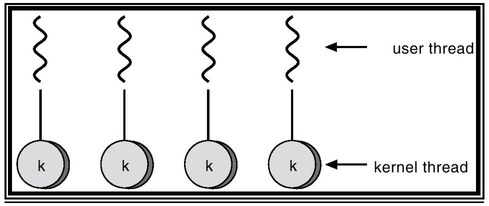
- הרבה Threads ברמת המשתמש ממופים ל- single kernel-level thread.



- ניהול ה- Thread נעשה ע"י מרווח המשתמש, כך שזה יעיל.
- מכיוון שרק Thread אחד יכול לגשת ל- Kernel בזמן נתון, מס' רב של threads לא יכולים לרוץ במקביליות על ריבוי מעבדים.
- בשימוש ע"י ספריות ULT על מערכות שאינן תומכות ב- Kernel threads (KLT).
- לדג':
  - Green Threads – ספריית thread זמינה ל- Solaris 2.

<u>One-to-One Model</u>

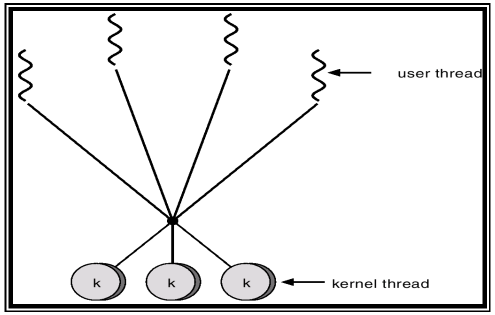
- כל thread ברמת משתמש ממפה ל- thread ברמת ה- Kernel.



- מעניק יותר בו-זמניות מאשר מודל many-to-one:
  - מאפשר ל- thread אחר לרוץ כאשר thread נחסם.
  - מאפשר למס' מרובה של threads לרוץ במקביל על מס' מעבדים.
  - יצירת thread משתמש דורש יצירה מתאימה של Kernel thread.
  - דג':
  - Windows NT/2000/XP
  - OS/2

<u>Many-to-Many Model</u>

- מאפשר ריבוי של threads ברמת משתמש להיות ממופים למס' קטן יותר (או שווה) של threads ברמת ה- Kernel .



- מאפשר ל- OS ליצור מס' מספיק של kernel threads (ספציפי לכל אפליקציה או מכונה) – הכי גמיש.
- דג':
  - Solaris 2
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Windows NT/2000 with the *ThreadFiber* package.

הרצאה 4:

תהליכים משתפי פעולה - Cooperating Processes

הקדמה

- תהליך עצמאי אינו מושפע או משפיע על ריצת תהליך אחר.
- תהליך משתף פעולה יכול להיות מושפע או להשפיע על ריצת תהליך אחר.
- יתרונות תהליכים משתפי פעולה:
  - שיתוף אינפורמציה
  - האצת חישובים
  - מודולריות
  - נוחות

שיתוף פעולה בין תהליכים באמצעות שיתוף - Cooperation Among Processes by Sharing

- תהליכים משתמשים ומעדכנים מידע שיתופי כמו משתנים, קבצים, ומסד נתונים.
- כתיבה חייבת להיות הדדית-ייחודית כדי למנוע מצבי מרוץ שיובילו לאי התאמה כאשר צופים בנתונים.
- מקטעים קריטיים מספקים את אמינות הנתונים.
- כאשר תהליך מבקש מקטע קריטי אין לעכב אותו, לא להגיע למצב קיפאון (deadlock) או רעב (starvation).

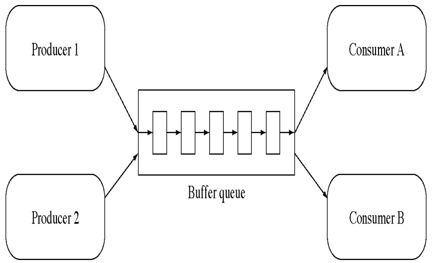
שיתוף פעולה בין תהליכים באמצעות תקשורת

- תקשורת מספקת דרך לסנכרן ולתאם בין פעילויות שונות.
- אפשרות להגיע למצב קיפאון:
  - כל תהליך מחכה להודעה מתהליך אחר.
- אפשרות להגיע למצב רעב:
  - שני תהליכים שולחים הודעות אחד לשני בעוד תהליך אחר מחכה להודעה.

בעיית יצרן/צרכן - Producer/Consumer (P/C) Problem

- פרדיגמת שיתוף פעולה בין תהליכים – תהליך יצרן מייצר מידע אשר נצרך ע"י תהליך צרכן.
  - דוגמא 1 – תוכנית הדפסה מספק תווים אשר נצרכים ע"י מדפסת.
  - דוגמא 2 – אסמבלר מייצר אובייקט מודל אשר נצרך ע"י הטוען (loader).
- אנו צריכים חוצץ אשר ישמור פריטים שנוצרים ולאחר מכן יצרכו:
  - חוצץ לא חסום – אין חסם פרקטי על גודלו של החוצץ.
  - חוצץ חסום – מניחים שיש גודל קבוע לחוצץ.

Multiple Producers and Consumers



דינמיקת יצרן/צרכן – Producer/Consumer (P/C) Dynamics

- תהליך יצרן מייצר מידע אשר נצרך ע"י תהליך צרכן.
- בכל זמן נתון, פעילות תהליך יצרן יכולה לייצר מידע.
- בכל זמן נתון, פעילות תהליך צרכן יכולה לרצות לקבל מידע.
- המידע צריך להישמר בחוצץ עד אשר הוא נזקק.

- במידה והחוצץ הוא מסוג חסום, אנו רוצים לחסום תהליך אשר המידע החדש שלו יגרום להצפה (overflow) של החוצץ.
- כמו-כן אנו רוצים לחסום תהליך צרכן אם אין מידע זמין אשר הוא מבקש.

רעיון לפתרון בעיית צרכן/יצרן - Idea for Producer/Consumer Solution

- נממש את החוצץ החסום ע"י מערך מעגלי ו-2 פוינטרים- in, out.
- פוינטר in יצביע על המקום הפנוי הבא בחוצץ.
- פוינטר out יצביע על המקום המלא הראשון בחוצץ.

פתרון באמצעות זיכרון שיתופי לחוצץ החסום- Bounded-Buffer – Shared-memory Solution

- זיכרון שיתופי:

Bounded-Buffer – Shared Data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

תהליך יצרן – חוצץ חסום - Bounded-Buffer – Producer Process

Bounded-Buffer – Producer Process

```
item nextProduced;
while (1) {
   while (((in + 1) % BUFFER_SIZE) == out); //do nothing
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

תהליך צרכן – חוצץ חסום - Bounded-Buffer – Consumer Process

Bounded-Buffer – Consumer Process

```
item nextConsumed;
while (1) {
    while (in == out);   /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

בעיות בהרצה שיתופית- Problems with concurrent execution

- תהליכים או תהליכונים משתפי פעולה לעיתים קרובות צריכים לחלוק מידע ומשאבים (מתוחזק באמצעות זיכרון שיתופי או קבצים).
- אם אין שליטה על הגישה למידע השיתופי יכול לקרות מצב שתהליך יקבל מידע לא עקבי.
- זה יגרום לכך שהפעולה שרוצים לבצע באמצעות התהליכים משתפי הפעולה תהיה תלויה בסדר ההרצה של התהליכים.

דוגמא לאי עקביות- Inconsistent View Example

```
Static char a;
Void echo()
```

```
{
    Cin >> a;
    Cout<<a;
}
```

- תהליכים P1 ו-P2 מריצים את אותה הפרוצדורה ולשניהם גישה לאותו המשתנה a.
- תהליכים יכולים להיות מופרעים בכל זמן.
- אם תהליך P1 ראשון להיות מופרע לאחר קליטת נתונים מהמשתמש ותהליך P2 רץ כולו, אז התו שנכתב ע"י תהליך P1 יהיה זה שיקרא ע"י תהליך P2.

<u>עקביות נתונים – Data Consistency</u>

- תחזוקת עקביות נתונים דורשת מכניזם שמוודא את סדר ההרצה של תהליכים משתפי פעולה.
- הפתרון לחוצץ החסום בעזרת זיכרון שיתופי מאפשר n-1 פריטים שמורים באותו זמן. הפתרון לשמירת n באותו הזמן הוא לא פשוט.
  - נניח ואנו מוסיפים משתנה חדש בשם counter אשר מאותחל ל-0 ובכל פעם שאנו מוסיפים פריט נעלה את ערכו ב-1.

<u>Bounded-Buffer – Shared Counter</u>

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

<u>Bounded-Buffer – Producer Process</u>

```
item nextProduced;
while (1) {
    while (counter == BUFFER_SIZE);  /*   do nothing    */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

<u>Bounded-Buffer – Consumer Process</u>

```
item nextConsumed;
while (1) {
    while (counter == 0);   /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
     counter--;
}
```

<u>Bounded-Buffer – Shared Counter – מונה שיתופי</u>

- הפקודות – counter++, counter— צריכות להתבצע בצורה אטומית (*atomically*).
- פקודה אטומית/בלתי ניתנת לחלוקה (Atomic/Indivisible) משמעותה פקודה שמסתיימת בשלמותה ללא הפרעה.
- הפקודה counter++ יכולה להיות ממומשת בשפת מכונה כך:

```
Register1 = counter
Register1 = register1 + 1
```



```
Counter = register1
```

- הפקודה --counter יכולה להיות ממומשת בשפת מכונה כך:

```
Register2 = counter
Register2 = register2 - 1
Counter = register2
```

- אם תהליכים יצרן וצרכן מנסים לעדכן את החוצץ באותו הזמן, הפקודות בשפת המכונה אולי ישולבו.
- השילוב של הפקודות תלוי בתזמון תהליך היצרן ותהליך הצרכן.
- דוגמא: הנח שהמונה מאותחל תחילה ל-5. שילוב של הפקודות:

```
Producer: register1 = counter (register1=5)
Producer: register1 = register1 + 1 (register1=6)
Consumer: register2 = counter (register2=5)
Consumer: register2 = register2 – 1(register2=4)
Producer: counter = register1 (counter=6)
Consumer: counter = register2 (counter=4)
```

- ערך המונה יהיה 4 או 6, בעוד התוצאה הנכונה היא 5.
- מצב כזה נקרא מרוץ (Race Condition).

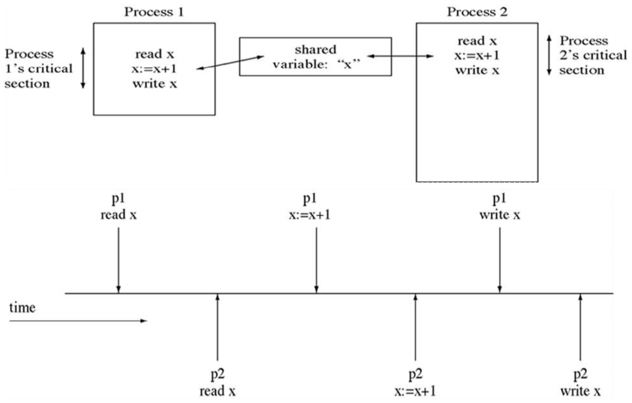
<u>מרוץ – Race Condition</u>

- מצב בו מספר תהליכים ניגשים/משנים מידע שיתופי בו זמנית. הערך הסופי תלוי בתהליך שמסתיים אחרון.
- כדי למנוע מרוצים יש לסנכרן ולתאם בין תהליכים משתפי פעולה.

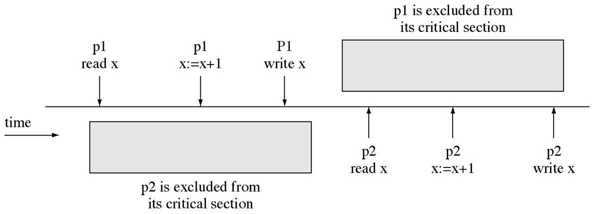
<u>Race condition updating a variable - מרוץ בעדכון משתנה</u>

```
Shared double balance;
Code for p1:                Code for p2:
…                           …
Balance += amount;          Balance += amount;
…                           …
Code for p1:                Code for p2:
…                           …
Load R1, balance            Load R1, balance
Load R2, amount             Load R2, amount
Add R1, R2                  Add R1, R2
Store R1, balance           Store R1, balance
```

<u>Race condition updating a variable</u>



Critical section to prevent a race condition



- Multiprogramming מאפשר הקבלה לוגית, משתמש בהתקנים בצורה יעילה, אך מאבדים נכונות במצבי מרוץ. לכן אנו לא מאפשרים הקבלה לוגית באזורים קריטיים. אנו מאבדים מעט מקביליות אך זוכים בנכונות.

__בעיית האזור הקריטי - The Critical-Section Problem__

- N תהליכים, כולם מתחרים להשתמש באותו מידע שיתופי.
- לכל תהליך יש מקטע קוד שנקרא CS (Critical Section) אשר אליו ניגשים למידע שיתופי.
- בעיה – יש להבטיח שכאשר תהליך רץ ב-CS שלו, לאף תהליך אחר אסור לגשת ל-CS שלו.

__דינמיקת בעיית האזור הקריטי - CS Problem Dynamics__

- כאשר תהליך מריץ קוד אשר מבצע מניפולציות על מידע שיתופי או משאב, אנו אומרים שהתהליך נמצא באזור הקריטי (לאותו מידע שיתופי).
  ההרצה באזורים קריטיים צריכה להיות ייחודית-הדדית – בכל זמן נתון רק לתהליך אחד מרשים לרוץ באזור הקריטי (אפילו כאשר יש כמה מעבדים).
  לכן תהליך צריך קודם לבקש רשות להיכנס לאזור הקריטי.
- מקטע הקוד המממש בקשה זו מקרא ES (Entry Section).
- LS (Leaving/Exit Section) יכול לעקוב סיום CS.
- שאר הקוד נקרא RS (Reminder Section).
- בעיית האזור הקריטי היא לתכנן פרוטוקול שהתהליכים יוכלו להשתמש בו כך שהפעולה לא תהיה מושפעת מסדר הרצתם.

__פתרון בעיית האזור הקריטי - Solution to Critical-Section Problem__

- 3 דרישות לפתרון נכון:
  - ריחוק הדדי – Mutual Exclusion

    - אם תהליך Pi רץ באזור הקריטי שלו, אז אף תהליך אחר אינו יכול לרוץ באזור הקריטי שלו.
      - אזורים קריטיים צריכים להיות ממוקדים וקצרים
      - עדיף שלא להיכנס ללולאה אינסופית שם
      - אם תהליך איכשהו נתקע/מפסיק באזור הקריטי שלו, אסור לו להפריע לתהליכים אחרים
  - התקדמות – Progress
    - אם אף תהליך אינו באזור הקריטי, ויש תהליכים שרוצים להיכנס לאזור הקריטי שלהם, ההחלטה על מי יכנס קודם אינה יכולה להידחות ללא גבול:
      - אם רק תהליך אחד רוצה להיכנס צריך לאפשר לו
      - אם שניים או יותר רוצים להיכנס, אחד מהם צריך להצליח
  - השהייה זמנית (חסומה) – Bounded Waiting
    - צריך להיות חסם על מספר הפעמים שתהליכים אחרים מאופשרים להיכנס לאזור הקריטי שלהם לאחר שתהליך ביקש להיכנס לאזור הקריטי שלו ולפני בקשה זו הבקשה הקודמת לא נענתה.
      - הנח כי כל תהליך מתבצע בזמן שאינו אפס.
      - אין הנחה לגבי מהירות יחסית של N התהליכים.
- אנו יכולים לבדוק עבור כל פתרון מוצע, האם שלושת הדרישות מתקיימות, למרות שמספיק שאחת לא מתקיימת כדי להגיד שהפתרון אינו טוב.

__סוגי פתרון לבעיית האזור הקריטי - Types of solutions to CS problem__

- פתרונות תוכנה
  - אלגוריתם אשר נכונותו אינו מסתמך על השערות אחרות.
- פתרונות חומרה
  - מסתמך על פקודות מכונה מיוחדות.
- פתרונות מערכת הפעלה
  - לספק מספר פונקציות ומבני נתונים למתכנת דרך קריאות מערכת/ספריה (System/Library calls)
- פתרונות שפת תוכנה
  - Linguistic constructs provided as part of a language

__פתרונות תוכנה - Software Solutions__

- אנו לוקחים בחשבון תחילה מקרה של 2 תהליכים.
- ניסיון התחלתי לפתור את הבעיה:
  - מבנה כללי של תהליך Pi (השני הוא Pj):

```
Do{
    Entry section
    Critical section
    Leave section
    Remainder section
}while(1);
```

  - תהליכים יכולים לחלוק כמה משתנים אשר יסנכרנו את פעולתם
- אלגוריתם מס' 1 – גרסת Larry/Jim:
  - משתנים משותפים:

```
String Turn; // initially turn="Larry" or "Jim" (no matter)
Turn = "Larry"; // Larry can enter his CS
Process Larry:
        Do{
```

```
While(turn != "Larry");
Critical section
Turn = "Jim";
Remainder section
}while(1);
```

גרסת Jim היא פשוט הפוכה (Jim/Larry)

- אלגוריתם מס' 1 – גרסת Pi/Pj:
  - משתנים משותפים:

```
int Turn; // initially turn=0
Turn = Pi; // Pi can enter his CS
Process Pi:
        Do{
            While(turn != i);
            Critical section
            Turn = j;
            Remainder section
        }while(1);
```

  - **דרישות ריחוק הדדי והשהייה זמנית מתקיימים (Mutual Exclusion, Bounded waiting) אך Progress לא מתקיים**

- אלגוריתם מס' 2 – גרסת Larry/Jim:
  - משתנים משותפים:

```
Bolean flag-larry, flag-jim;
//initially flag-larry=flag-jim=flase
Flag-larry = true;  // larry ready to enter his CS
Process Larry:
        Do{
            While(flag-jim);
            Flag-larry=true;
            Critical section
            Flag-larry=false;
            Remainder section
        }while(1);
```

- אלגוריתם מס' 2 – גרסת Pi/Pj:
  - משתנים משותפים:

```
Bolean flag[2];
//initially flag[0]=flag[1]=flase
Flag[i] = true;  // Pi ready to enter his CS
Process Pi:
        Do{
            While(flag[j]);
            Flag[i]=true;
            Critical section
            Flag[i]=false;
            Remainder section
        }while(1);
```

  - **דרישת Progress מתקיימת אך לא מתקיימים: Mutual Exclusion, Bounded waiting**

- אלגוריתם מס' 3 – גרסת Larry/Jim:
  - משתנים משותפים:


```
Bolean flag-larry, flag-jim;
//initially flag-larry=flag-jim=flase
Flag-larry = true;  // larry ready to enter his CS
Process Larry:
        Do{
            Flag-larry=true;
            While(flag-jim);
            Critical section
            Flag-larry=false;
            Remainder section
        }while(1);
```

- אלגוריתם מס' 3 – גרסת Pi/Pj:
  - משתנים משותפים:

```
Bolean flag[2];
//initially flag[0]=flag[1]=flase
Flag[i] = true;  // Pi ready to enter his CS
Process Pi:
        Do{
            Flag[i]=true;
            While(flag[j]);
            Critical section
            Flag[i]=false;
            Remainder section
        }while(1);
```

  - **דרישת Mutual Exclusion מתקיימת אך לא מתקיימים: Progress, Bounded waiting**

- אלגוריתם מס' 4 – גרסת Larry/Jim:
  - משתנים משותפים: שילוב של אלגוריתמים 1, 3/2
  - Process Larry

```
Do{
    Flag-larry=true;
    Turn = "Jim";
    While(flag-jim and turn = "Jim");
    Critical section
    Flag-larry=false;
    Remainder section
}while(1);
```

- אלגוריתם מס' 4 – גרסת Pi/Pj:
  - משתנים משותפים: שילוב של אלגוריתמים 1, 3/2
  - Process Pi

```
Do{
    Flag[i]=true;
    Turn = j;
    While(flag[j] and turn = j);
    Critical section
    Flag[i]=false;
```

```
Remainder section
}while(1);
```

- **כל שלושת הדרישות מתקיימות, פותר את בעיית CS עבור 2 תהליכים**

- אלגוריתם מס' 5 – גרסת Larry/Jim:
  - כמו אלגוריתם 4 אך עם 2 הפקודות של entry section מוחלפות – האם עדיין פתרון נכון?
  - Process Larry

```
Do{
    Turn = "Jim";
    Flag-larry = true
    While(flag-jim and turn = "Jim");
    Critical section
    Flag-larry=false;
    Remainder section
}while(1);
```

- אלגוריתם Bakery:
  - אזור קריטי עבור N תהליכים:
    - לפני הכניסה לאזור הקריטי תהליך מקבל מספר (כמו במאפיה). המחזיק במספר הנמוך ביותר נכנס לאזור הקריטי
    - סכמת יצירת המספרים היא בסדר עולה תמיד – 1,2,3,3,3,4,5,6......
    - אם תהליך Pi ו- Pj מקבלים את אותו המספר, אז לפי האינדקס קובעים מי יכנס ראשון, אם i<j אז תהליך עם אינדקס i ישורת ראשון, אחרת Pj (מניחים כי PID הוא ייחודי).
  - בחירת מספר:
    - הגדרה: Max(a0,a1….an-1) = k, כך ש- k>ai לכל i=0…n-1
  - סימון עבור סדר לקסיקוגרפי (Ticket #, PID #):
    - (a,b)<(c,d) מתקיים אם a<=c ו – b<d
  - מידע שיתופי:

```
Bolean choosing[n];
Int number[n];
```

מבני נתונים מאותחלים ל- false ו-0 בהתאמה

  - האלגוריתם:

```
do {
    choosing[i] = true;
    number[i] = max(number[0], …, number[n – 1]) +1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) ;
        while ((number[j] != 0) &&  ((number[j],j) < (number[i],i))) ;
    }
    critical section
    number[i] = 0;
    remainder section
```



```
} while (1);
```

<u>מה לגבי כישלונות תהליך? - What about process failures?</u>

- אם כל שלושת הדרישות מתקיימות, פתרון טוב ייתן חסן נגד כשלון תהליך ב-RS שלו.
  - כיוון שכשלון RS הוא כמו לקבל RS אינסופי
- לעומת זאת, פתרון טוב לא ייתן חסן נגד כשלון תהליך ב-CS שלו.
  - תהליך Pi אשר נכשל ב- CS שלו לא מודיע על כך לשאר התהליכים, עבורם Pi עדיין נמצא בשלב ה-CS שלו.

<u>חסרונות של פתרונות תוכנה - Drawbacks of software solutions</u>

- פתרונות תכונה הם נורא עדינים.
- תהליכים אשר מבקשים להיכנס ל-CS שלהם עסוקים בלחכות (צורכים זמן מעבד ללא צורך)
  - אם CS הם ארוכים, יהיה יותר אפקטיבי לחסום תהליכים שמחכים להיכנס ל-CS

<u>פתרונות חומרה - Hardware solutions: interrupt disabling</u>

```
Process Pi:
  Repeat
  Disable interrupts
  Critical section
  Enable interrupts
  Remainder section
  Forever
```

- במעבד ליבה אחת (Uniprocessor) – Mutual Exclusion נשמרת אך יעילות הריצה יורדת – כאשר ב-CS, אנו לא יכולים לשלב ריצה עם תהליכים אחרים שנמצאים ב- RS.
- במעבד עם כמה ליבות (MultiProcessor) – Mutual Exclusion לא נשמרת:
  - CS עכשיו אטומי אך ריחוק הדדי לא
  - באופן כללי לא פתרון מתקבל

<u>פתרונות חומרה - Hardware solutions: special machine instructions</u>

- באופן נורמאלי, גישה לזיכרון מונעת גישה אחרת לאותו מקום בזיכרון.
- הרחבה: מעצבים הציעו פקודות מכונה שמבצעות 2 פעולות אטומיות על אותו מקום בזיכרון (קריאה וכתיבה).
- הרחבה כזו היא Mutual Exclusion גם למעבד MultiProcessor.
- אפשר להשתמש בפקודה זו כדי לספק ריחוק הדדי, אך יש להשלים במכניזם נוסף שיקיים את 2 הדרישות הנוספות לבעיית ה-CS.

<u>The test-and-set instruction</u>

- אלגוריתם שמשתמש ב-test-set עבור ריחוק הדדי (Mutual Exclusion):
  - משתנה משותף b מאותחל ל-0.
  - התהליך Pi הראשון שמאתחל את b נכנס ל-CS

```
Process Pi:
  Repeat
    Repeat{}
    Until testset(b);
    CS
    B=0;
  Forever
```

  - קוד C++ שמתאר test-set:

```
Bool testset(int& i)
```

```
{
    If(i==0){
        I=1;
        Return true;
    }
    Else{
        Return false;
    }
}
```

<u>TestAndSet Synchronization Hardware</u>

- בדוק ושנה תוכן מילה באופן אטומי (גרסא בוליאנית):

```
Bolean testandset(Bolean& target){
        Bolean RV = target;
        Target = true;
        Return RV;
}
```

<u>Mutual Exclusion with TestAndSet</u>

- מידע שיתופי:

```
Bolean lock = false;
```

- Process Pi:

```
Do{
    While(testandset(lock));
    CS
    Lock=false;
    RS
}
```

<u>Swap Synchronization Hardware</u>

```
Void swap(bolean& a,Bolean& b){
  Bolean temp = a;
  A=b;
  B=temp;
}
```

<u>Mutual Exclusion with TestAndSet</u>

- מידע שיתופי:

```
Bolean lock = false;
```

- Process Pi:

```
Do{
    Key=true;
    While(key==true)
```



```
        Swap(lock,key);
    CS
    Lock=false;
    RS
}
```

<u>סמפורים – Semaphores</u>

- כלי סנכרון (מסופק ע"י מערכת ההפעלה) שלא דורש המתנה ארוכה.
- מבחינה לוגית, סמפור S הוא משתנה מסוג Integer שחוץ מאתחול אפשר לגשת אליו דרך 2 פעולות אטומיות ורחוקות הדדית (Mutual Exclusion):
  - Wait(s)
  - Signal(s)

<u>אזור קריטי עבור n תהליכים – Critical Section of *n* Processes</u>

- מידע שיתופי:

Semaphore mutex;<u> //initially mutex=1</u>

- Process Pi:

```
Do{
  Wait(mutex);
  CS
  Signal(mutex);
  RS
} while(1);
```

<u>סמפורים - המשך – Semaphores</u>

- גישה דרך 2 פעולות אטומיות:

```
Wait(s):
While s<=0 do no-op;
s--;
signal(s):
s++;
```

- אך כדי להימנע מהמתנה ארוכה: כאשר תהליך צריך לחכות נשים אותו בתור של תהליכים חסומים שמחכים עבור אותו אירוע.

- לפיכך, סמפור הוא רשומה (מבנה):

```
Type semaphore = record
Count : integer;
Queue : list of process
End;
Var  s: semaphore ;
```

- כאשר תהליך חייב לחכות לסמפור S, הוא נחסם ומוכנס לתור הסמפור.
- פעולת signal (הנח מדיניות הוגנת כגון FIFO) מורידה תהליך אחד מהתור ושמה אותו בתור התהליכים המוכנים.

- <u>פעולות הסמפור - Semaphore's operations</u>

```
Wait(s):
```

```
    s.count--;
    if(s.count<0){
      block this process
        place this process in s.queue
    }
signal(s):
    s.count++;
    if(s.count <=0){
        remove a process p from s.queue
        place this process p on ready list
    }
```

- <u>מימוש הסמפור - Semaphore Implementation</u>
    - הגדר את הסמפור כמבנה:

```
Typedef struct{
        Int value;
        Struct process *L;
}semaphore;
```

    - הנח 2 פעולות פשוטות:
        - Block – השהה את התהליך שקרא לו
        - Wakeup(P) – המשך את פעולתו של התהליך החסום P
    - כעת פעולות הסמפור מוגדרות כך:

```
Wait(s):
    s.value--;
    if(s.value <0){
        add this process to s.L;
        block;
    }

signal(s):
    s.value++;
    if(s.value <=0){
        removes a process P from s.L ;
        wakeup(P)
    }
```

<u>סמפורים ככלי סנכרון כללי– Semaphore as a General Synchronization Tool</u>

- הרץ את פעולה B בתהליך Pj רק לאחר שפעולה A התבצעה בתהליך Pi.
- השתמש בדגל סמפור שמאותחל ל-0.
- קוד:

```
Pi              Pj
.               .
.               .
.               .
A               wait(flag)
```



```
Signal(flag)    B
```

<u>קיפאון והרעבה - Deadlock and Starvation</u>

- קיפאון (Deadlock) – 2 תהליכים או יותר מחכים ללא גבול לאירוע שיכול להיגרם רק ע"י אחד מהתהליכים הממתינים.
- נסמן ב-S ו-Q שני סמפורים שמאותחלים ל-1

```
   P0          P1
Wait(S);    Wait(Q);
Wait(Q);    Wait(S);
   .           .
   .           .
   .           .
signal(S);  signal (Q);
signal(Q);  signal (S);
```

- רעב (Starvation) – חסימה ללא גבול. תהליך אולי לעולם לא יוסר מתור הסמפור אשר משהה אותו.

<u>סמפור בינארי - Binary Semaphores</u>

- הסמפורים שדנו בהם עד כה נקראים counting (integer) semaphores.
- קיימים גם סמפורים בינאריים:
    - דומים ל counting semaphores – רק שהמשתנה count הוא בוליאני
    - Counting semaphores יכולים להיות ממומשים בעזרת סמפורים בינאריים.
    - באופן כללי יותר קשים לשימוש מאשר counting semaphores (למשל – הם לא יכולים להיות מאותחלים למספר k>1)

```
waitB(s):
    if(s.value = 1){
        s.value = 0;
    }else{
        Block this process;
        Place this process in s.queue;
    }

signalB(S):
    if (S.queue is empty) {
        S.value = 1;
    } else {
        remove a process P from S.queue
        place this process P on ready list
    }
```

<u>שני סוגי סמפור - Two Types of Semaphores</u>

- Counting semaphores – משתנה מסוג integer יכול לנוע בתחום לא מוגבל
- Binary semaphores – משתנה מסוג integer יכול לנוע בתחום 0-1 בלבד – יכול להיות פשוט יותר למימוש.
- אנו יכולים לממש counting semaphore כ- binary semaphore (ובכך להגן על ה- counter שלו).

מימוש סמפור S כסמפור בינארי - Implementing *S* as a Binary Semaphore

- מבנה נתונים:

```
Binary-semaphore S1,S2;
Int C;
```

- אתחול:

```
S1=1;
S2=0;
C=initial value of semaphore S;
```

```
Wait operation:
      Waitb(S1);
      C--;
      If(C<0){
         Signalb(S1);
         Waitb(S2);
      }
      Signal(S1);
```

```
signal operation:
      Waitb(S1);
      C++;
      If(C<=0){
         Signalb(S2);
      }
      Else{
         signalb(S1);
      }
```

בעיות עם סמפורים

- סמפורים מספקים כלי חזק כדי לאכוף Mutual Exclusion ולתאם בין תהליכים.
- אך פעולות Wait(S), Signal(S) מפוזרות בקרב מספר תהליכים, לכן קשה להבין את דרך השפעתם.
- שימוש בהם צריך להיות נכון בכל התהליכים.
- תהליך רע (או זדוני) אחד יכול לגרום לכישלון של הרבה תהליכים אחרים.

הרצאה 5:

הקדמה למקביליות - Introduction to Concurrency

בעיות קלאסיות של מקביליות - Classical Problems of Concurrency :

- יש הרבה כאלה, נראה 3 מהם:
  - **בעיית חוצץ חסום - Bounded buffer problem**
    - אנו צריכים 3 סמפורים:
      - סמפור mutex עבור mutual exclusion עבור גישה לחוצץ
      - סמפור full כדי לסנכרן את תהליכים יצרן וצרכן עם מספר הפריטים שאפשר לצרוך
      - סמפור empty עבור סנכרון תהליכים יצרן וצרכן עם מספר המקומות הריקים
    - מידע שיתופי:

```
Semaphore mutex, full, empty;
// initially:  full=0, empty=n, mutex=1
```

- Bounded-Buffer - Producer Process:

```
Do{
      …
      produce an item in nextp
      …
      Wait(empty);
      …
      add nextp to buffer
      …
      signal(mutex);
      signal(full);
} while (1);
```

- Bounded-Buffer - Consumer Process:

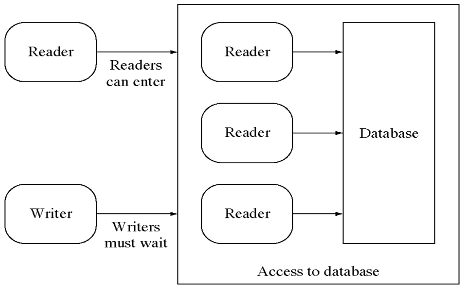
```
Do{
      Wait(full);
      Wait(mutex);
      …
      Remove an item from buffer to nextc
      …
      signal(mutex);
      signal(full);
      …
      Consum the item in nextc
      …
} while (1);
```

- הערות על פתרון חוצץ חסום - Notes on Bounded-Buffer Solution:
  - לשים signal(full) בתוך ה-CS של היצרן (במקום מחוץ ל-CS) לא משפיע מאחר והצרכן צריך לחכות לשני הסמפורים לפני שהוא ממשיך.
  - הצרכן חייב לבצע wait(full) לפני wait(mutex), אחרת יקרא קיפאון אם הצרכן יכנס ל CS כאשר החוצץ ריק.
  - שימוש בסמפורים זו אומנות קשה....

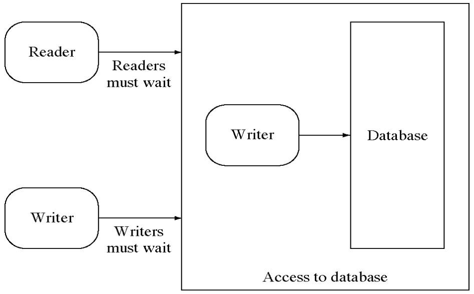
- **Readers-Writers Problem - בעיית קוראים-כותבים**

- כל מספר של פעולות קריאה וכתיבה יכולות רצות.
- בכל זמן נתון, פעולת קריאה תרצה לקרוא נתונים.
- בכל זמן נתון, פעולת כתיבה תרצה לשנות את הנתונים.
- בזמן שתהליך כתיבה כותב/משנה את המידע השיתופי לאף תהליך קריאה/כתיבה אסור לגשת אליו.
- כל מספר של תהליכים קוראים יכולים לגשת למידע המשותף בו זמנית.

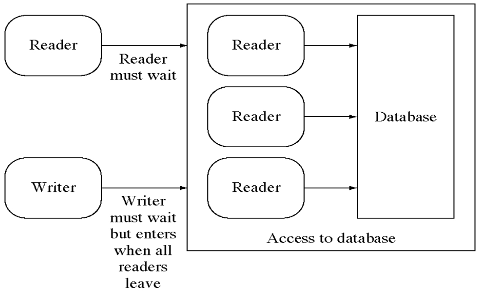
Readers-Writers with active readers



Readers-writers with an active writer



Should readers wait for waiting writer?



- כל מספר של פעולות קריאה וכתיבה יכולות רצות.
- בכל זמן נתון, פעולת קריאה תרצה לקרוא נתונים.
- בכל זמן נתון, פעולת כתיבה תרצה לשנות את הנתונים.
- בזמן שתהליך כתיבה כותב/משנה את המידע השיתופי לאף תהליך קריאה/כתיבה אסור לגשת אליו.
- כל מספר של תהליכים קוראים יכולים לגשת למידע המשותף בו זמנית.

- יש מספר גרסאות עם תכונות קוראים-כותבים שונות:
  - הבעיה הראשונה דורשת שלא יהיה תהליך קורא שמחכה, אלא אם כן תהליך כותב משנה באותו הזמן מידע שיתופי.
  - בבעיה השנייה הדרישה היא שברגע שכותב הוא פנוי, אין לתת לקורא חדש להתחיל את פעולתו (קריאה).
  - בפתרון למקרה הראשון יכול להיות מצב של רעב עבור תהליכים כותבים.
  - בפתרון למקרה השני יכול להיות מצב של רעב עבור תהליכים קוראים.

- פתרון עבור הבעיה הראשונה:
  - מידע שיתופי:

```
Semaphore mutex,wrt;
Int readcount;
// initially
 mutex = 1, wrt = 1,  readcount = 0
```

  - Readcount - סופר כמה תהליכים קוראים כעת.
  - סמפור mutex מספק mutual exclusion עבור עדכון readcount
  - סמפור wrt מספק mutual exclusion עבור הכותבים, כמו כן בשימוש עבור הקורא הראשון/אחרון שיוצא/נכנס אל CS.
  - תהליך כותב - First Readers-Writers - Writer Process

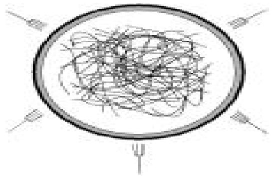
```
Wait(wrt);
...
Writing is performed
...
Signal(wrt);
```

  - תהליך קורא - First Readers-Writers - Reader Process

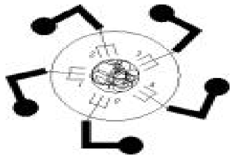
```
Wait(mutex);
Readcount++;
 If(readcount==1){
```

```
        Wait(wrt);
  }
 Signal(mutex);
 …
 Reading is performed
 …
 Wait(mutex);
 Readcount--;
 If(readcount==0){
      Signal(wrt);
  }
 Signal(mutex);
```

- **בעיית פילוסופים אוכלים - Dining Philosophers Problem**
  - מושיבים 5 פילוסופים סביב שולחן עגול.
  - מול כל אחד מהם יש קערה עם אורז.
  - בין כל זוג יש צ'ופסטיק, כך שיש 5 צ'ופסטיקים.
  - צריכים 2 צ'ופסטיקים כדי לאכול אורז, לכן כאשר פילוסוף מספר n אוכל, פילוסופים n+1, n-1 אינם יכולים לאכול.
  - זה מתאר את הקושי בהקצאה של משאבים בין תהליכים בלי קיפאון ורעב.
  - כל אחד חושב מעט, לוקח את הצ'ופסטיקים שהוא צריך ומניח אותם במעגל אינסופי.



  - האתגר הוא להעניק אישור לצ'ופסטיקים מבלי להגיע למצב קיפאון/רעב.
  - קיפאון יכול להתרחש כאשר כולם מנסים בו זמנית לקחת צ'ופסטיק. כל אחד מרים צ'ופסטיק שמאלי, ונתקע כיוון שהצ'ופסטיק הימני הוא השמאלי של מישהו אחר.



- **פתרון מס' 1בעיית הפילוסופים האוכלים** - Dining Philosophers Solution:

```
Process Pi:
        Repeat
            think;
            wait(fork[i]);
            wait(fork[i+1 mod 5]);
            eat;
            signal(fork[i+1 mod 5]);
            signal(fork[i]);
```



```
        forever
```

- כל פילוסוף הוא תהליך.
- סמפור אחד עבור כל fork:
  - Fork: array[0..4] of semaphores
  - אתחול -
    - Fork[i].count=1
    - For i=0..4
- בניסיון הראשון – קיפאון אם כל פילוסוף מתחיל בהרמה של הצ'ופסטיק השמאלי.

- **פתרון מס' 2 בעיית הפילוסופים האוכלים:**
  - פתרונות אפשריים למניעת רעב:
    - לאפשר ל-4 פילוסופים ולא יותר לשבת ליד השולחן בלבד.
    - הפילוסופים במושבים האי זוגיים ירימו את הצ'ופסטיק השמאלי ראשונים, והפילוסופים במושבים הזוגיים ירימו את הימניים.

- **פתרון מס' 3 בעיית הפילוסופים האוכלים:**

```
Process Pi:
        Repeat
            think;
            wait(T);
            wait(fork[i]);
            wait(fork[i+1 mod 5]);
            eat;
            signal(fork[i+1 mod 5]);
            signal(fork[i]);
            signal(T);
        forever
```

- הכנס רק 4 פילוסופים שמנסים לאכול בו זמנית.
- לכן אחד יכול לאכול, בעוד ששלושת האחרים מחזיקים צ'ופסטיק אחד.
- אנו יכולים להשתמש בסמפור נוסף T אשר יגביל ל-4 את מספר הפילוסופים שיושבים סביב השולחן.
- אתחול: T.count=4

הערה למהנדסים 2009: עפ"י האמור בהיילרן, מכאן ועד הסוף לא כלול בחומר למבחן. רזי.

אזורים קריטיים - Critical Regions :

- אזור קריטי – הוא אזור בו התהליך עשוי לעדכן או לשנות משתנים המשותפים לשני התהליכים העובדים במקביליות ולכן הוא בעייתי
- המאפיין החשוב של המערכת הוא לדאוג, שכאשר תהליך נמצא ב-critical section שלו, אף תהליך אחר לא יוכל להיכנס לקטע הקריטי שלו. בצורה כזאת, ריצת התהליכים בקטעים הקריטיים שלהם היא mutually exclusive (בלעדית) בזמן.

אזורים קריטיים – איך זה מתבצע? :

- High-level linguistic synchronization construct.
- משתנה משותף v מסוג t מוכרז כך: v:shared t
- ניגשים למשתנה v רק בביטוי באזור v כאשר B עושה את S, כאשר B ביטוי בוליאני

- כאשר S בביצוע, אף תהליך אחר אינו יכול לגשת למשתנה v.
- אפשרי למימוש ע"י סמפורים.
- אזורים המתייחסים לאותו משתנה משותף שוללים אחד את השני במשך הזמן.
- כאשר הליך מנסה לגשת לאזור קריטי מתבצע התהליך הבא:
  - נבדק ערכו של B
  - אם ערכו True התהליך מתבצע
  - אם ערכו False התהליך ימתין עד שערכו של B ישתנה

Bounded buffer – Critical Regions :

- הרעיון הוא להשתמש בחוצץ בכדי למנוע התנגשות של שני תהליכים הניגשים לאותו אזור קריטי
- דרך פעולה :

```
struct buffer {
    int pool[n];
    int counter, in, out;
```

- תהליך יצרן מכניס את היצרן הבא לתוך החוצץ המשותף

```
region buffer when(counter < n) {
        pool[in] = nextp;
        in = (in+1) % n;
        counter++;
}
```

- תהליך צרכן מוציא אלמנט מן החוצץ ומכניס את הצרכן הבא

```
region buffer when (counter > 0) {
        nextc = pool[out];
        out = (out+1) % n;
        counter--;
    }
```

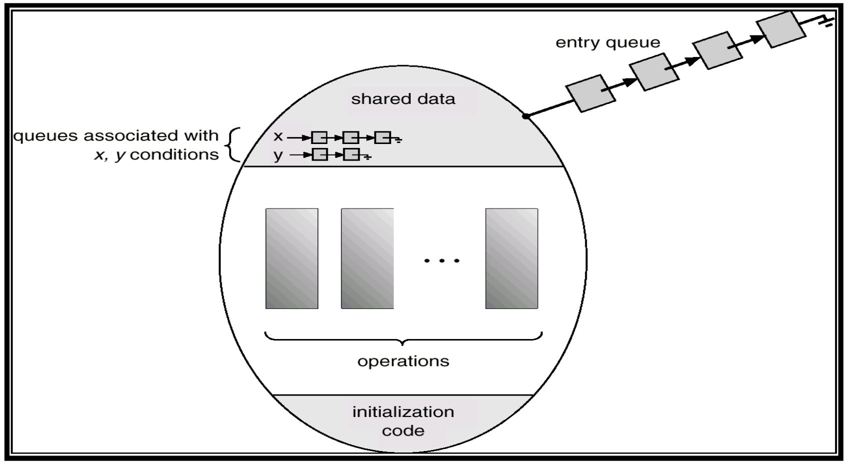
Monitors - משגוחים :

- משגוחים הם מבנים הזהים בשימוש לסמאפורים אך קלים יותר לשליטה
- קיימים הרבה סוגים של משגוחים
- קיים בשפות רבות כמו פסקל וג'אווה
- ניתן ליישום ע"י סמאפורים
- Monitor הוא מודל תוכנה המכיל
  - תהליך אחד או יותר (אופרציות)
  - רצף אתחול
  - משתני Data מקומיים
- מאפייני המשגוח :
  - גישה למשתנים המקומיים ע"י שגרות המשגוח בלבד
  - תהליך נכנס למשגוח ע"י קריאה לאחד השגרות שלו
  - רק תהליך אחד יכול לשהות במשגוח בכל זמן נתון
  - משגוח מבטיח Mutual exclusion – לפחות משאב אחד מוחזק באופן שלא ניתן לחלוק בו. כלומר, בכל פעם רק תהליך אחד יכול להשתמש במשאב. כאשר תהליך אחד יבקש את המשאב הנ"ל, הוא יאלץ להמתין עד שהמשאב יתפנה.
  - ולכן בעצם המשגוח מגן על Data הנמצא בתוכו – הוא נועל את המידע המשותף

תנאיי המשגוח (Monitor Conditions) :

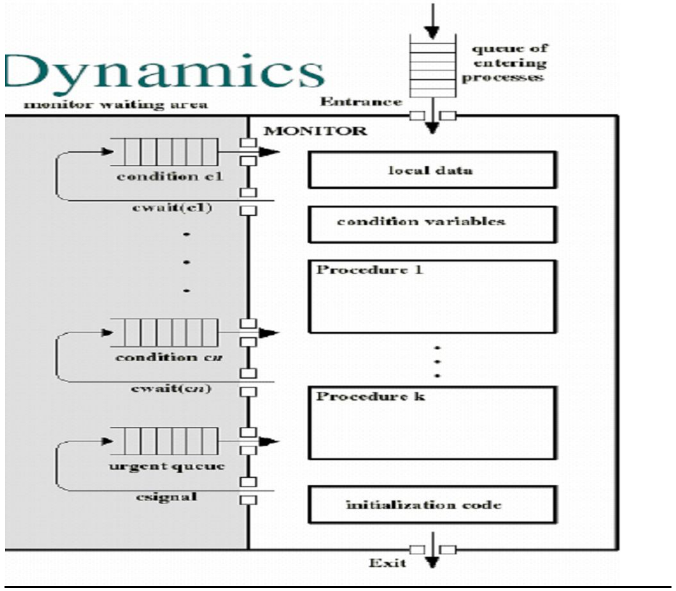
- בכדי לאפשר לתהליך לשהות בתוך המשגוח חייב להתקיים משתנה תנאי המוצהר מראש (לדוגמא Condition X)
- משתני התנאי הינם לוקליים למשגוח ולכן ניתן לגשת אליהן רק מתוכו

- האופרציות המפעילות את התנאי לגבי התור המדובר הן:
  - Cwait – השהיית תהליך בודד עד עירור של תהליך נוסף
  - Csignal – מחדשת ביצוע של תהליך מושהה אחד , אם אין תהליך מושהה אין לה השפעה
- באיור ניתן לראות שני תורות של תנאים X ו Y , הפקודה Cwait(x) תגרור השהיית תהליך בתור X



דוגמא לדינאמיקה אפשרית של משגוח :

- התהליכים ממתינים בתור הכניסה למשגוח או באחד מתורי התנאי
- תהליך מכניס עצמו לתור תנאי Cn ע"י פקודת wait(Cn)
- פקודת Csignal(Cn) מכניסה למשגוח תהליך אחד מתוך התור של התנאי Cn
- לכן פקודת Csignal(Cn) חוסמת את התהליך הנקרא ומכניסה אותו לurgent queue



<u>בעיית היצרן/צרכן :</u>

- תזכורת : שני סוגי תהליכים : <u>היצרן</u> יוצר אינפורמציה שתהליך <u>הצרכן</u> צורך הבעיה מתעוררת כאשר מתקיים מצב של Bounded buffer – מאגר מוגבל. כאשר הוא מתמלא היצרן צריך להמתין שהצרכן ירוקן אותו.
- פתרון ע"י <u>משגוח</u> :
  - המשגוח יאחסן את הBuffer שהוא בעצם מערך של עצמים
  - שני תנאי משגוח נוספים לפתירת בעיה זו :
    - Csignal(notfull) – מציין שהחוצץ אינו מלא
    - Csignal(notempty) – מציין שהחוצץ אינו ריק
  - משתנים נוספים לפתירת הבעיה :
    - Nextin – מצביע לעצם הבא שעומד להתווסף לחוצץ
    - Nextout – מצביע לעצם הבא שנקח
    - Count - מספר העצמים שנמצאים כרגע בחוצץ
- <u>מימוש :</u>
  - היצרן והצרכן יחזרו כל הזמן על ההליכים הבאים בלולאה אינסופית :
  - <u>יצרן :</u>
    - Procedure v
    - Append(v)
  - <u>צרכן :</u>
    - Take(v)
    - Consume(v)

```
Monitor boundedbuffer:
  buffer: array[0..k-1] of items;
  nextin:=0, nextout:=0, count:=0: integer;
  notfull, notempty: condition;

  append(v):
    if (count=k) cwait(notfull);
    buffer[nextin]:= v;
    nextin:= nextin+1 mod k;
    count++;
    csignal(notempty);

  take(v):
    if (count=0) cwait(notempty);
    v:= buffer[nextout];
    nextout:= nextout+1 mod k;
    count--;
    csignal(notfull);
```

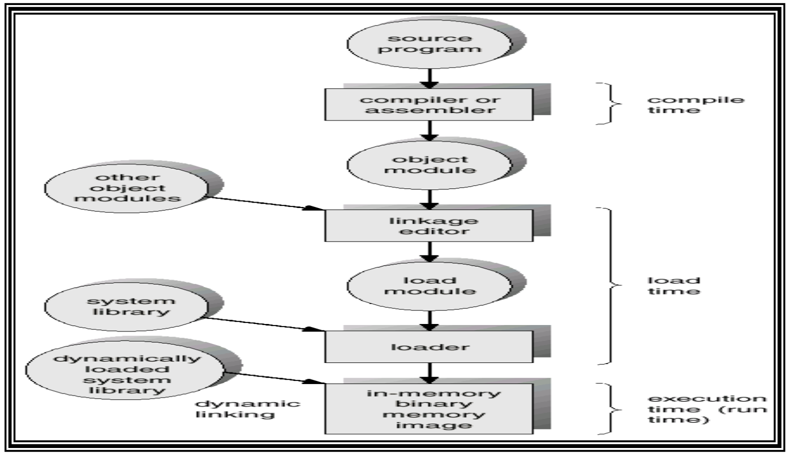
A. Frank - P. Weisberg

<u>בעיית הפילוסופים הסועדים :</u>

- תזכורת : 5 פילוסופים יושבים סביב שולחן עגול. בין כל 2 צלחות יש מקל אכילה אחד. על מנת לאכול, כל פילוסוף זקוק לשני מקלות אכילה. כל פילוסוף רוצה לאכול ולדבר.אם כל פילוסוף ייקח תחילה את המקל שמימינו, ורק לאחר מכן את המקל שמשמאלו, המקל כבר יילקח ע״י הפילוסוף האחר.
- במצגת מצוין פתרון לבעיה זו ע"י שימוש במשגוח :
  - הפתרון מבטיח ששני שכנים לא יאכלו באותו הזמן
  - הפתרון חסין ממצב של קיפאון אך תתכן הרעבה
  - לפתרון המלא ראה מצגת 5.2 עמ' 20-23

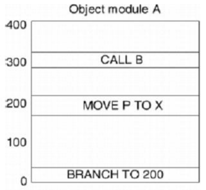
הרצאה 7: ניהול זיכרון ממשי

- מוטיבציה : בכדי להריץ תוכנה יש תחילה "להביא" אותה לזיכרון ולסווגה כתהליך
- ניהול הזיכרון נעשה ע"י מערכת ההפעלה והחומרה בכדי לתמוך בריבוי תהליכים בזיכרון הראשי
- בכדי שתוכנה תרוץ נדרש מעבר בין כמה שלבים שמודגם ע"י הסכימה הבאה המתארת טעינת תהליך לזיכרון :



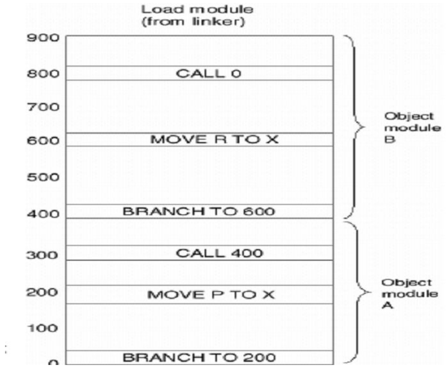
מודל האובייקט (Object module):

- מודל האובייקט מכיל את השדות הבאים :
  - Relocation dictionary – מכיל רשימה של הוראות שהאופרנדים שלהן הן כתובות
  - קוד המכונה
  - Data – התוכנית עצמה
  - External names table – מוגדר ע"י מודלי אובייקט אחרים ומכיל רשימה של הוראות שהאופרנדים שלהן הן שמות האובייקטים האחרים
  - Public names table – לשימוש מודלי אובייקט אחרים
  - Module identification – קוד זיהוי המודל
- רק ה Data וקוד המכונה יטענו לזיכרון הממשי , שאר השדות הינם לשימור ה Linker ויוסרו מאוחר יותר.
- דוגמא למודל אובייקט :



ה- Linker :

- בתחילה לכל מודל אובייקט יש מרחב כתובת משלו משמע שכל הכתובות הינם בהיסט לכתובת ההתחלתית של אותו מודל האובייקט , תפקידו של הלינקרים הוא לקשר ביניהם.
- ישנם 2 סוגים של קישור בין האובייקטים :
  - **קישור סטטי :**
  - הלינקר מקשר בין האובייקטים ויוצר מרחב כתובות ליניארי
  - הכתובת היחסית החדשה הינה היסט מן הכתובת ההתחלתית של מודל הטעינה (של הלינקר)



  - **קישור דינאמי :**
  - קישור חלק מן האובייקטים נעשה לאחר יצירת מודל הטעינה (קובץ ההפעלה של התוכנית)
  - נבדיל בין שני סוגים של קישור דינאמי :
    - קישור בזמן טעינה (Load Time) – בקובץ ההפעלה ישנן הפניות למודלים חיצוניים והדבר נפתר בזמן טעינת התוכנית.
    - קישור בזמן ריצה (Run Time) - הפניות למודלים חיצוניים רק כאשר מתבצעת קריאה ממשית לתהליך שמוגדר במודל החיצוני המדובר , משמעות הדבר היא שתהליך שלא השתמשנו בו במהלך התוכנית כלל לא יטען לזיכרון הממשי.

גודל הזיכרון מול נפח התוכנה :

- הבעיה היא שנפח התוכנית גדול מנפח הזיכרון שאנו יכולים להקצות לה , נדון בשני פתרונות אפשריים :
  - **1.Overlays :**
    - שומר בזיכרון רק את ההוראות והמידע שהתוכנית צריכה בכל זמן או שלב נתון.
    - עובד רק במקרה שהתוכנה מתאימה למודל העבודה הנתון.
    - ממומש ע"י המתכנת ולא ע"י מערכת ההפעלה ולכן לא מצריך תמיכה מיוחדת שלה.
    - עיצוב התוכנה במבנה של overlays הוא מסובך.
  - **2.Dynamic Linking (Libraries – Dll's) :**
    - שימוש בקישור דינאמי הוא שימושי כאשר מקטע גדול מן הקוד נמצא בשימוש רק בחלק מועט מן הזמן.
    - תהליך נטען לזיכרון רק ברגע שהוא נקרא.
    - ניצול טוב יותר של הזיכרון - תהליכים שלא נקראים במהלך התוכנית לא יטענו לזיכרון.
    - לא דרושה הרבה תמיכה ממערכת ההפעלה – ממומש ע"י כותב התוכנה

איך מתבצע קישור דינאמי :

- הקישור נדחה לזמן ביצוע התוכנה (בניגוד לסטטי שמתבצע לפני).
- מקטע קטן של הקוד הנקרא Stub משמש בכדי לאתר השגרה הרצויה .
- הStub מחליף את עצמו בכתובת השגרה ומבצע אותה.
- מערכת ההפעלה צריכה לבדוק האם השגרה נמצאת בזיכרון כחלק מתהליך.
- קישור דינאמי יעיל במיוחד בספריות המשותפות לכולן או ששכיחות פעילותן גבוהה.

יתרונות הקישור הדינאמי :

- קבצי ההפעלה יכולים להשתמש בגרסא שונה של מודל חיצוני מבלי הצורך לעדכן אותם
- כל תהליך מקושר לאותו מודל חיצוני – חוסך מקום אחסון בדיסק
- אותו מודל חיצוני יטען לזיכרון רק פעם אחת – תהליכים יכולים לשתף קוד ובכך לחסוך בזיכרון
- דוגמא : בחלונות המודלים החיצוניים הם קבצי DLL

דרישות מניהול זיכרון :

- ניתן לטעון לזיכרון הראשי רק חלק קטן מן התהליכים בו זמנית ולכן רוב התהליכים יהיו במצב של המתנה לקלט/פלט והמעבד יהיה פנוי , לכן חשוב שהזיכרון יהיה מנוהל בצורה טובה על מנת שנוכל לכלול בו כמה תהליכים שרק ניתן.
- בנוסף יש לתת תמיכה גם לנושאים הבאים :
  1. Relocation
  2. Protection
  3. Sharing
  4. Logical Organization
  5. Physical Organization

Relocation.1 (מיקום מחדש):

- המתכנת אינו יכול לדעת היכן התוכנית תאוחסן בזיכרון לאחר ההפעלה.
- תהליך עלול לשנות מקום כתוצאה מפעולות של מערכת ההפעלה :
  - Swapping – החלפה מאפשרת למערכת ההפעלה מאגר גדול יותר של תהליכים מוכנים לביצוע.
  - Compaction – דחיסה מאפשרת למערכת ההפעלה יותר זיכרון עוקב בכדי לאחסן בו תוכניות.

Protection.2 (הגנה):

- אין על תהליכים לגשת לזיכרון המוקצה לתהליך אחר ללא הרשאה מתאימה
- לא ניתן לבדוק כתובות בתוכנית בזמן הקומפילציה או הריצה מפני שהמערכת יכולה למקם את התוכנית מחדש.
- הפנייה לכתובות צריכה להיבדק בזמן הביצוע ע"י החומרה.

.

Sharing.3 (שיתוף):

- שיתוף מאפשר לכמה תהליכים לגשת לאותו מקטע בזיכרון הראשי מבלי להתפשר על אבטחה.

  - עדיף לאפשר לכל תהליך לגשת לאותו העותק של תוכנה מאשר ליצור לכל תהליך עותק נפרד.
  - על תהליכים הפועלים בשיתוף פעולה לגשת לעיתים לאותו מבנה נתונים.

Logical Organization.4 :

- המשתמשים כותבים תוכניות בעלי אפיון שונה :
  - תוכניות הניתנות לביצוע בלבד.
  - מודלי מידע המאפשרים קריאה/כתיבה בלבד.
  - חלק מן המודלים פרטיים וחלקם פומביים.
- בכדי להתמודד עם תוכניות המשתמש באופן יעיל על מערכת ההפעלה והחומרה לתמוך באופן בסיסי של מודל בכדי לספק את השיתוף והאבטחה הנדרשת.

Physical Organization.5 :

- הזיכרון החיצוני מאחסן את התוכניות והמידע לטווח ארוך ואילו הזיכרון הראשי מאחסן את התוכניות והמידע שכרגע בשימוש.
- הזזת המידע בין שני היררכיות זיכרון אלה חשובה מאוד לניהול הזיכרון ומאוד לא יעיל לתת אחריות זאת בידו של מפתח התוכנה.

הצורך במיקום מחדש :

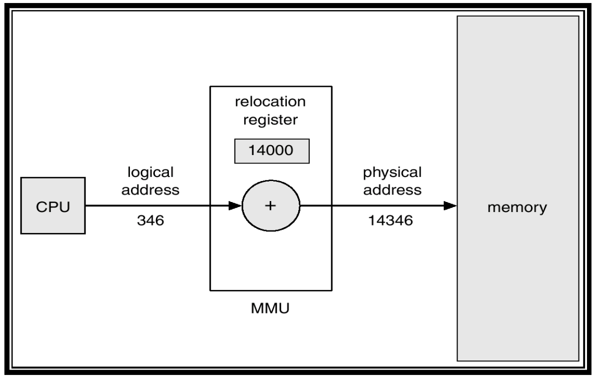
- בגלל הצורך של מערכת ההפעלה לבצע הליכים של החלפת מקום (Swapping) או דחיסה (compaction) תהליך יכול להימצא בכתובות שונות בזיכרון במהלך "חייו"
- הדבר יוצר בעיה: היסט של כתובת ע"י תהליך לא יכולה להיות מקובעת
- פתרון הבעיה נעשה ע"י הבחנה בין כתובת לוגית לכתובת פיזית:
  - כתובת פיזית – (נקראת גם אבסולוטית) היא הכתובת בזיכרון הראשי
  - כתובת לוגית – (נקראת גם וירטואלית) היא הפנייה לזיכרון בצורה בלתי תלויה לארגון הפיזי של הזיכרון.
- הקומפיילר מפיק קוד בו כל ההפניות לזיכרון הן לוגיות
- דוגמא לכתובת לוגית היא כתובת יחסית – כתובת המהווה היסט מנקודה ידוע כגון התחלת התוכנית (זהו גם הסוג השימושי ביותר של כתובות לוגיות)

סכמת המיקום מחדש :

- מודלים המאפשרים מיקום מחדש נטענים לזיכרון הראשי כאשר כל הפניות הזיכרון נשארות בצורה היחסית.
- כתובת פיזית מחושבת כאשר ההוראות מבוצעות.
- המרת הכתובת הלוגית לפיזית מתבצעת ע"י החומרה.

Memory-Management Unit (MMU) :

- יחידת ניהול הזיכרון היא רכיב חומרה שממפה כתובות לוגיות לכתובות פיזיות.
- היחידה בעצם מתרגמת כתובות מן המעבד לזיכרון
- מבצע זאת הוספת הערך שנמצא ברגיסטר הריאלוקציה לכתובת הלוגית (ראה סכימה).
- קיים גם רגיסטר לימיט המציין את הכתובת הפיזית הגבוהה ביותר של תהליך זה
- התהליך (process) לעולם לא רואה את הכתובת הפיזית אלא פועל מול הכתובת הוירטואלית בלבד.



מתי לכרוך הוראות/מידע לזיכרון ?

- כריכת המידע לזיכרון יכולה להתבצע באחד משלושת השלבים הבאים:
  - זמן קומפילציה –אם מיקום הזיכרון ידוע לנו נבצע כריכה בזמן זה
  - זמן טעינה – אם מיקום הזיכרון לא ידוע לנו בזמן קומפילציה יש ליצור קוד יחסי
  - זמן ביצוע – הכריכה מתעקבת עד זמן הריצה אם ניתן לבצע ריאלוקציה של התהליך (בזמן הביצוע הוא יכול להיות ממוקם מחדש ע"י מערכת ההפעלה)

כתובת פיזית מול כתובת לוגית

- הרעיון של מרחב כתובות לוגיות הכרוך במרחב כתובות פיזיות הוא מרכזי לניהול נכון של הזיכרון
- הכתובות הפיזיות והלוגיות זהות בסכמות כריכת הכתובת בזמן הקומפילציה והטעינה
- הכתובות הפיזיות והלוגיות שונות בסכמות כריכת הכתובת בזמן הריצה

דינמיקת תרגום הכתובת ע"י החומרה

- כאשר תהליך (process) מתחיל לרוץ רגיסטר הריאלוקציה (נקרא גם בסיס) נטען בערך של הכתובת הפיזית ההתחלתית של התהליך
- רגיסטר הלימיט נטען בכתובתו הפיזית הסופית של התהליך
- כאשר נתקל בכתובת יחסית נוסיף אותה לתוכן של רגיסטר הבסיס ואז נשווה אם רגיסטר הלימיט
- תצורה זאת מוסיפה לנו הגנת חומרה – כל תהליך רשאי לגשת רק לזיכרון שמוקצה לו

הקצאת רצף זיכרון - Contiguous Allocation

- תהליך בהרצה חייב להטען במלואו לתוך הזיכרון (במידה ולא נשתמש בoverlays)
- הזיכרון הראשי בדר"כ מפוצל לשני חלקים או יותר:
  - מערכת ההפעלה הנוכחת ממוקמת בחלקו הנמוך
  - תהליכי המשתמש ממוקמים בחלקו הגבוהה
  - סכימת רגיסטר הריאלוקציה מגינה מפגיעה של תהליכי המשתמש אחד בשני או במערכת ההפעלה

טכניקות ניהול זיכרון

- מספר טכניקות שהיו קיימות בעבר , כיום נפוץ הזיכרון הוירטואלי:
  1. Fixed/Static Partitioning
  2. Variable/Dynamic Partitioning
  3. Simple/Basic Paging
  4. Simple/Basic Segmentation

1. Fixed/Static Partitioning :

- חלוקת הזיכרון הראשי לא חופפים , כל חלק נקרא מחיצה (Partition)
- גדלי המחיצות יכולים להיות שווים או שונים בגודלם
- שטח האחסון הנותר במחיצה לאחר השמתה לתוכנית נקרא internal fragmentation

אלגוריתם השמה בחלוקה סטטית

- **גודל מחיצות זהה** – אם ישנה מחיצה פנויה ניתן לטעון לתוכה תהליך
  - מפני שגודל כל המחיצות זהה לא משנה באיזו מהן נבחר
  - אם כל המחיצות מלאות בחר תהליך אחד ובצע Swap לדיסק

- **גודל מחיצות שונה** – ישנן 2 דרכים ליישום השיטה:
- שימוש בכמה מבני תור :
  - הקצה כל תהליך לתוך המחיצה הקטנה ביותר אליה הוא יתאים
  - קיים תור לכל גודל מחיצה
  - שיטה זאת מנסה למזער את אפקט ה internal fragmentation אך יוצרת בעיה : חלק מן התורים יכולים להיות ריקים בזמן שהשאר מלאים

- שימוש במבנה תור בודד :
  - הקצה כל תהליך לתוך המחיצה הקטנה ביותר אליה הוא יתאים
  - נותן יותר אפקט ל Multiprogramming על חשבון internal fragmentation

דינאמיקה בחלוקה סטטית

- כל תהליך שגודלו פחות מגודל המחיצה רשאי להיטען לתוכה
- אם כל המחיצות מלאות מערכת ההפעלה תבצע Swap ותפנה מחיצה
- אם תוכנית גדולה מדי בכדי להתאים למחיצה בודדת על המתכנת לתכננה בעזרת overlays

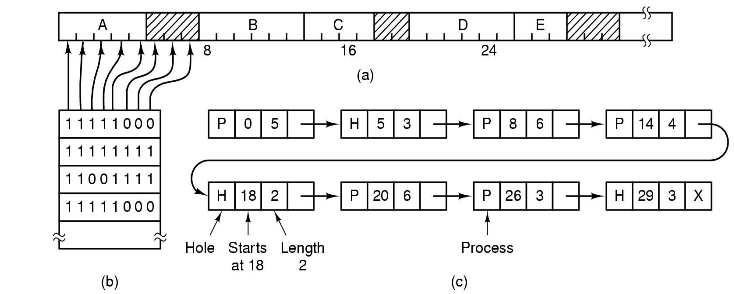
הערות

- השימוש בחלוקה סטטית אינו יעיל מפני שכל תוכנית לא משנה עד כמה קטנה היא תתפוס מחיצה שלמה – הדבר יוצר הרבה internal fragmentation
- למרות הפיתרון המוצע של מחיצות לא שוות בגודלן הבעיה עדיין לא נפתרת לחלוטין

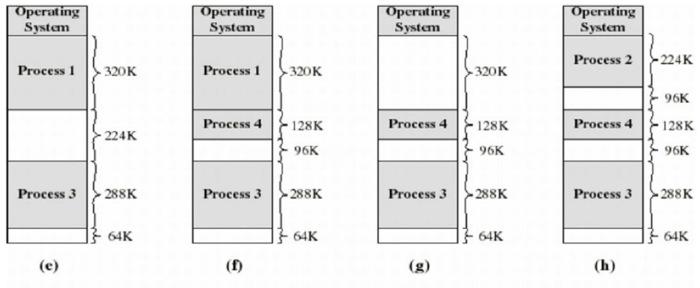
2. Variable Partitioning :

- כאשר מגיע תהליך מוקצה לו זיכרון מ Hole הגדול מספיק בכדי להכיל את התהליך
- Hole הוא בלוק של זיכרון זמין , בזיכרון מפוזרים חורים בגדלים שונים
- מערכת ההפעלה מתחזקת מידע לגבי :
  - המחיצות שהוקצו לתהליכים
  - המחיצות החופשיות (חורים)

- אם אין מקום לתהליך שמגיע מערכת ההפעלה תבצע Swap בכדי שיתפנה לו חור גדול מספיק להכילו
- בסכימה הבאה המקום הראשון בזיכרון תפוס ע"י תהליך שאורכו 5 ולאחר מכן יש חור שאורכו 3



- בשלב F המערכת עושה swap לתהליך מס' 1 ובכך מפנה חור של 320k בכדי להכיל את תהליך מס' 2 – לאחר הכנסתו (שלב H) נוצר חור חדש של 96k



Fragmentation Internal/External

- ישנם 2 סוגים של Fragmentation במקרה זה
  1. Fragmentation פנימי – הזיכרון שהוקצה יכול להיות גדול במעט מהזיכרון המבוקש , פנימי לגבי כל מחיצה (בדומה למקרה של מחיצות סטטיות)
  2. Fragmentation חיצוני – קיים זיכרון בכדי להכיל את התהליך המבוקש אך הוא לא רציף אלא מורכב מכמה חורים במקומות שונים

טיפול בבעיית ה External Fragmentationl :

- טיפול בבעיה זו דורש דחיסה :
  - "ערבוב" תוכן הזיכרון בכדי למקם את כל הזיכרון החופשי בבלוק אחד גדול
  - מתאפשר רק כאשר מדובר בריאלוקציה דינאמית ומתבצע בזמן הריצה
  - בעיות ביצוע לגבי I/O :
    - נועל את העבודה בזיכרון במהלך ביצוע I/O ולכן לא ניתן לבצע ערבוב
    - פתרון : ביצוע I/O לתוך I/O buffer

הערות

- מחיצות בעלי אורך ומספר שונה
- לכל תהליך מוקצה בדיוק כמות הזיכרון הדרוש לו
- לאורך הדרך נוצרים חורים בזיכרון שגורמים ל External Fragmentation
- משתמש בדחיסה בכדי להתגבר על בעיה זו

בעיית ההשמה

- נרצה למקם תהליך בגודל מסוים כאשר נתונים לנו מס' חורים חופשיים , ישנן 4 גישות:
  1. First Fit – הקצאת ה-hole הראשון ברשימה אשר גדול מספיק בשביל התהליך.
  2. Next Fit – כמו First Fit רק שהחיפוש יתחיל מהנקודה האחרונה שבה עצרנו.
  3. Best Fit – הקצאת ה-hole הקטן ביותר אשר גדול מספיק בשביל התהליך. יש לעבור על כל הרשימה.
  4. Worst Fit – הקצאת ה-hole הגדול ביותר ברשימה. שוב יש לעבור על כל הרשימה.

אלגוריתם ההשמה

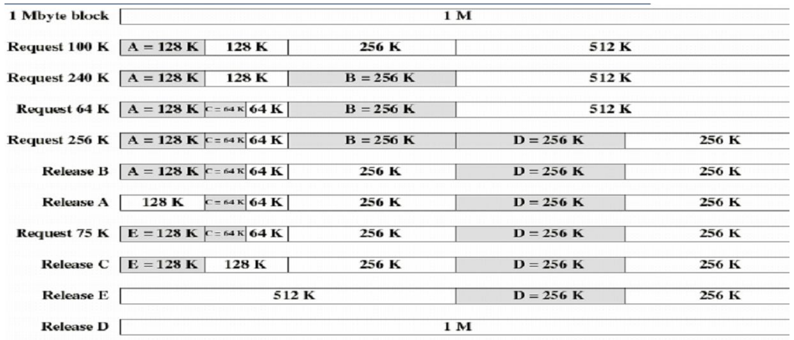
- משמש בכדי להחליט באיזה בלוק חופשי למקם את התהליך
- המטרה : להפחית שימוש בדחיסה – לוקח הרבה זמן
- הערות:
  - First Fit מקצה בדר"כ בתחילת הבלוק ולרוב יוצר פחות בעיות fragmentation מאשר Next Fit
  - Next Fit בדר"כ מבצע השמה של הבלוק הגדול ביותר בסוף הזיכרון
  - Best Fit משאיר אחריו שברים (fragments) קטנים מפני שהוא מחפש את הבלוק האופטימאלי לאחסון ולכן המערכת תצטרך לבצע דחיסה בתכיפות גדולה יותר
  - Worst Fit כשמה היא הרגועה ביותר במהירות וגם ביעילות האיחסון

אלגוריתם ההחלפה

- כאשר כל התהליכים בזיכרון הראשי חסומים על מערכת ההפעלה להחליט איזה תהליך יש להחליף (swap)
- במקרה זה תהליך חייב להיות מוחלף ע"י תהליך שמוכן לפעולה או תהליך חדש

Knuth's Buddy System :

- ניסיון להתגבר על החסרונות של המחיצה הקבועה והמשתנה
- הביצוע הוא לחלק את כלל שטח הזיכרון ל-2 ולהמשיך את החלוקה ב-2 של כל קטע עד מציאת גודל מתאים



- נעצור את תהליך הפיצול כאשר נמצא חלק בגודל שחיפשנו
- לכל 2 חלקים שנוצרו מחלק בודד נקרא "חברים" וכאשר שניהם כבר לא יהיו בשימוש נאחדם לגודלם המקורי חזרה

הערות

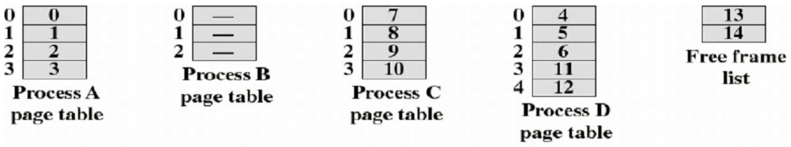
- יעיל ביותר כאשר גודל הזיכרון מתחלק ב-2
- בממוצע internal fragmentation הוא 25% וכל בלוק מנוצל לפחות ב 50%
- תוכניות אינם מוזזות בזיכרון – מפשט את ניהול הזיכרון

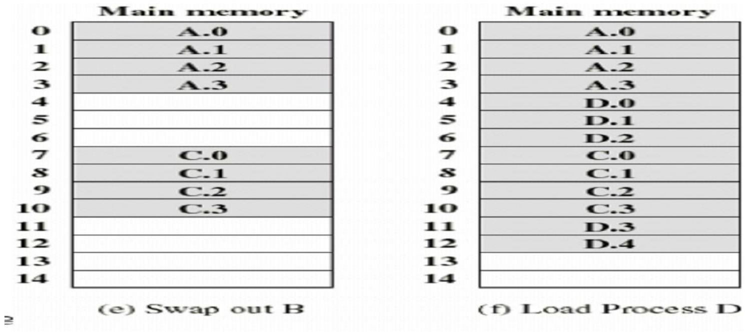
3. Simple/Basic Paging :

- הרעיון : לאפשר לתהליך לעבוד עם מרחב כתובות לוגי לא רציף , וכך התהליך מוקצה לתהליך זיכרון פיזי היכן שהוא פנוי.
- הזיכרון הפיזי (RAM) מחולק לבלוקים בגודל קבוע הנקראים frames. (בדר"כ חזקה של 2)
- הזיכרון הלוגי מחולק גם הוא לבלוקים באותו גודל הנקראים pages.
- לכן דפי התהליך יכולים להיות ממופים לכל כתובת זיכרון במרחב , לאו דווקא רציף
- למימוש השיטה יש ליצור page table בכדי לתרגם את הכתובת הלוגית לכתובת הפיזית
- Internal fragmentation אפשרי – עמוד אשר מכיל סוף של תוכנית
- יש גם לשמור טבלת free-frame list בכדי לעקוב אחרי ה frames שנמצאים בשימוש

הערות

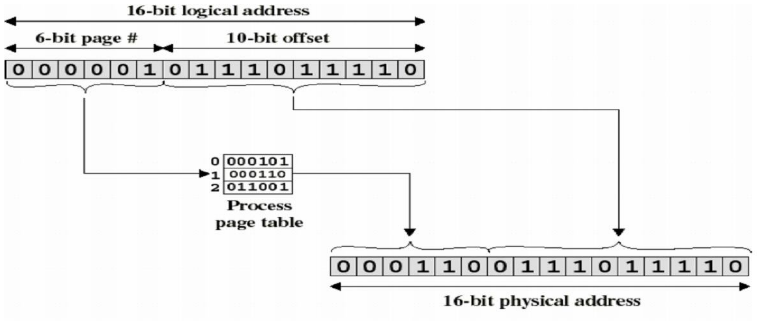
- כעת על מערכת ההפעלה להחזיק בזיכרון הראשי page table לכל תהליך
- בכל מופע של הטבלה מצוין היכן נמצא הפריים הנ"ל בזיכרון הפיזי
- קיימת גם רשימה של free frame table/list (ראה דוגמא)





הכתובת הוירטואלית בPaging :

- הכתובת הלוגית הינה כתובת יחסית כאשר מדובר על גודל דף שהוא חזקה של 2
- לדוגמא : גודל הכתובת הינו 16ביט וגודל הדף הינו 1K – ה-Offset בגודל של 10 (1K) ואילו שאר הביטים (6) משמשים אותנו כאינדקס למספר העמוד
- הכתובת הנוצרת הינה יחסית לתחילת התהליך

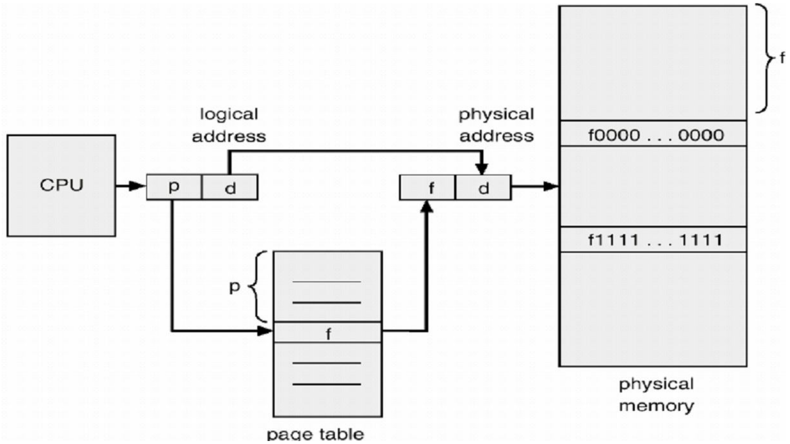


הכתובת הוירטואלית בPaging (המשך) :

- כל כתובת מכילה מספר עמוד והיסט בתוך עמוד זה
- קיים רגיסטר המחזיק בתוכו את הכתובת ההתחלתית של התהליך שרץ כרגע
- בהינתן הכתובת הלוגית המעבד ניגש לpage table בכדי לחשב את הכתובת הפיזית

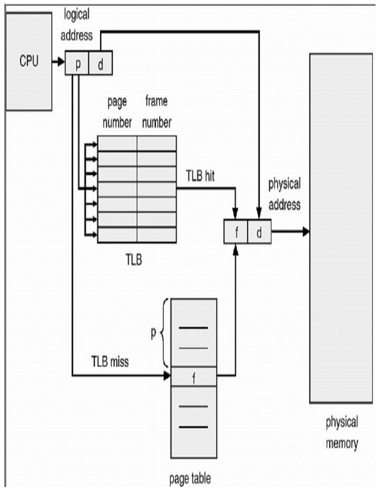
סכימת תרגום הכתובת :

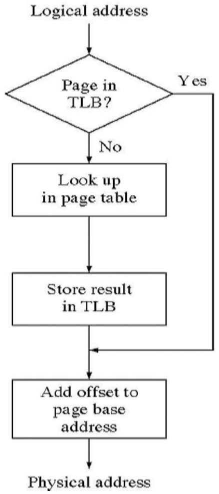
- הכתובות הלוגית הנוצרות ע"י המעבד מכילות 2 חלקים :
  - *Page number*
  - *Page offset*
- ע"י שימוש בגודל דף בחזקה של 2 המשתמש או התוכנה בעצם אינם מודעים לקיום הדפים
- תרגום הכתובות מתבצע ע"י החומרה (ראה שירטוט)



איך ליישם Page Table :

- שמור את ה page table בזיכרון הראשי
  - שמור רגיסטרי בסיס וגודל לטבלה
  - נוצרת בעיה שכל הוראה בעצם תיצור שני גישות אחת לpage table ואחת להוראה עצמה
- שמור את ה page table בחומרה
  - הטבלה יכולה להיות די גדולה – יקר מדי
- פתרון לבעיות הנ"ל הוא שילוב בין השתיים :
  - שימוש בחומרה מיוחדת הנקראת TLB – החיפוש הראשון הוא בTLB אם הדף לא נמצא אז המעבד יביא אותה מן הpage table הנמצאת בזיכרון (ראה איור)





אז למה בעצם העיקרון של TBL עובד? :

- עיקרון הלוקאליות

- הTLB משתמש במיפוי אסוציאטיבי שבודק בו זמנית את כל המופעים בטבלת הTLB (מהיר)
- Hit rate = 90%
- לכל תהליך שרץ מתאימה טבלה שונה בTLB ולכן בכל פעם שתהליך חדש מתחיל לרוץ הטבלה תתרוקן ותתמלא במופעים המתאימים לתהליך זה

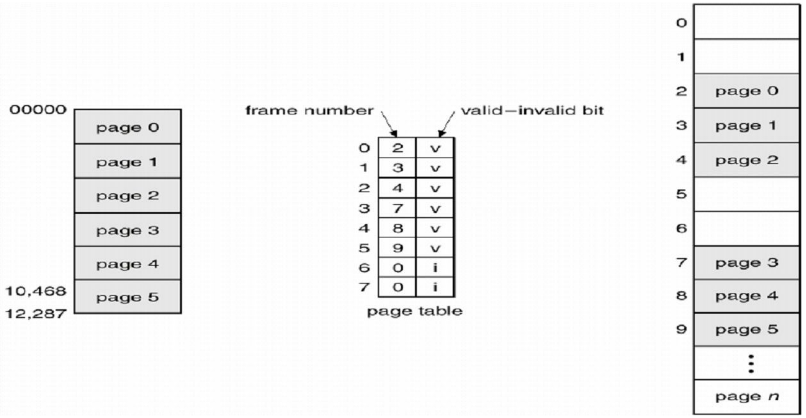
Effective Access Time (EAT) :

- זמן הגישה האפקטיבי הוא בין 1 ל 2 זמני גישה וקרוב יותר ל-1
- בהנחה שmemory cycle לוקח מיקרו שנייה
- α = Hit ratio הוא מספר המציין (באחוזים) את המצאות הדף המתאים ב TLB
- Associative (Memory) Lookup = ε time unit

$EAT = \alpha(\varepsilon + 1) + (1 - \alpha)(\varepsilon + 2) = 2 + \varepsilon - \alpha$

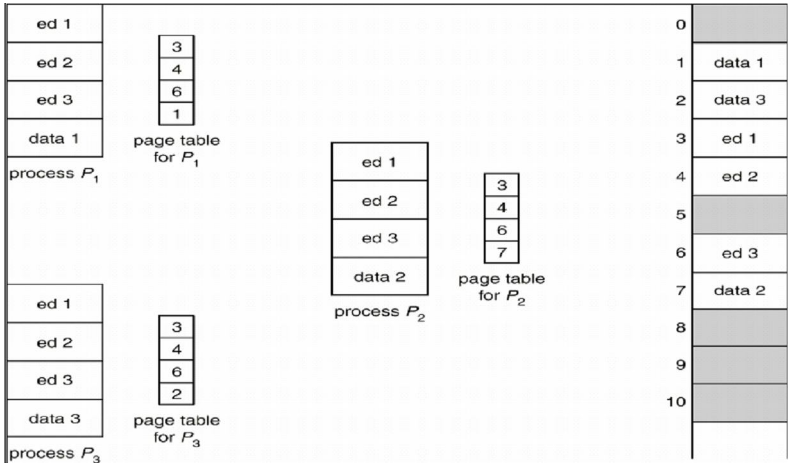
Memory Protection:

- ההגנה על הזיכרון מתבצעת ע"י הוספת ביט הגנה לכל Frame שהוא הבלוק בזיכרון הפיזי
- לכל מופע בטבלת הTLB נוסף ביט *Valid-invalid*
  - Valid – הדף הינו חוקי , הוא נמצא במרחב הכתובות הלוגי של התהליך
  - Invalid – הדף אינו נמצא במרחב הכתובות של התהליך



Shared Pages :

- אם אותו הקוד משותף לכמה משתמשים שונים זה יעיל לשמור אותו רק פעם אחת בזיכרון (לדוגמא מעבד תמלילים)
- הקוד המשותף בלתי ניתן לשינוי עצמי בכדי ששני ההליכים יוכלו לבצע את אותו הקוד
- קיימות שתי Page Tables – אחת לכל תהליך המובילות שתיהן לאותן פריימים בזיכרון – ולכן עותק אחד בלבד בזיכרון הראשי
- עמודי הData של כל משתמש הינם פרטיים לו עצמו (ראה איור)

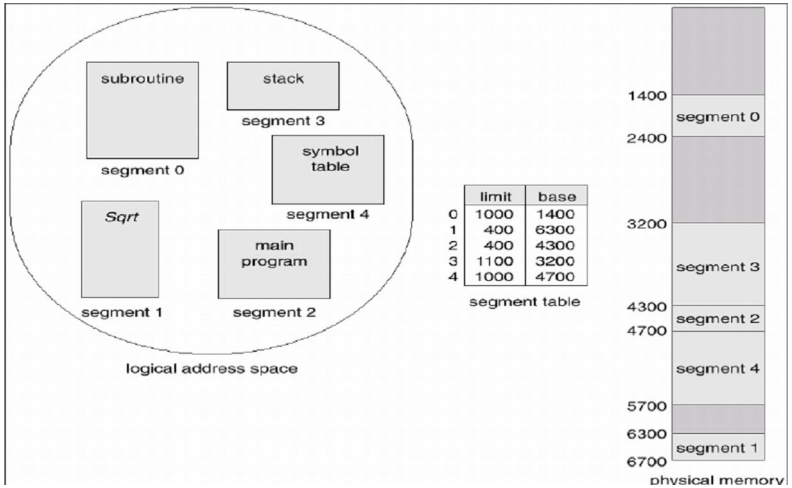


Simple/Basic Segmentation.4 :

- אחד האספקטים החשובים בניהול זיכרון אשר נעשה בלתי-נמנע בעבודה עם paging הוא ההפרדה בין איך שהמשתמש רואה את הזיכרון לבין איך שהזיכרון באמת בנוי פיזית.
- הרעיון : מבט לגבי איך שהמשתמש רואה את הזיכרון
- תוכנית שכתב המשתמש בנויה מסגמנטים שונים כגון :
  - התוכנית הראשית
  - אובייקט
  - מחסנית
  - וכו'

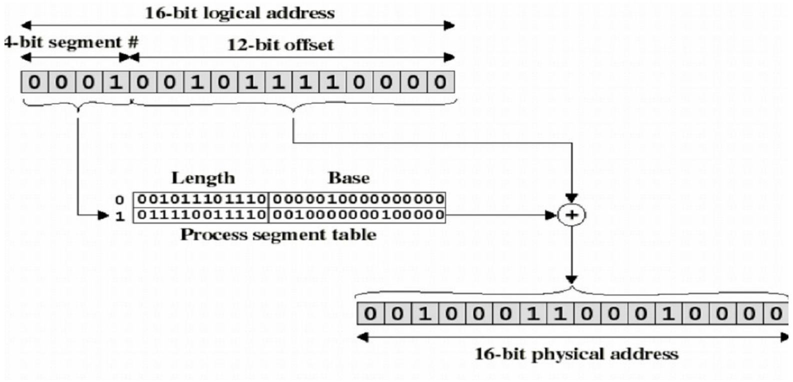
דינאמיקה של פילוח(Segmentation) פשוט :

- כל תוכנית מחולקת לבלוקים בגדלים לא שווים הנקראים סגמנטים
- כאשר התהליך נטען לזיכרון הסגמנטים שלו מאותרים
- כל סגמנט הוא מלא לחלוטין במידע – אין internal fragmentation
- ה External fragmentation פוחתת כאשר משתמשים בסגמנטים קטנים
- בניגוד לPaging בפילוח המשתמש מבין את החלוקה בזיכרון
  - עוזר לארגון לוגי של התוכנה
  - המשתמש חייב לדעת שגודל הסגמנט הוא מוגבל
- מערכת הפעלה מכילה טבלת סגמנטים לכל תהליך המכילה :
  - הכתובת הפיזית ההתחלתית של הסגמנט
  - אורך/גודל הסגמנט (להגנה)



כתובות לוגיות בפילוח :

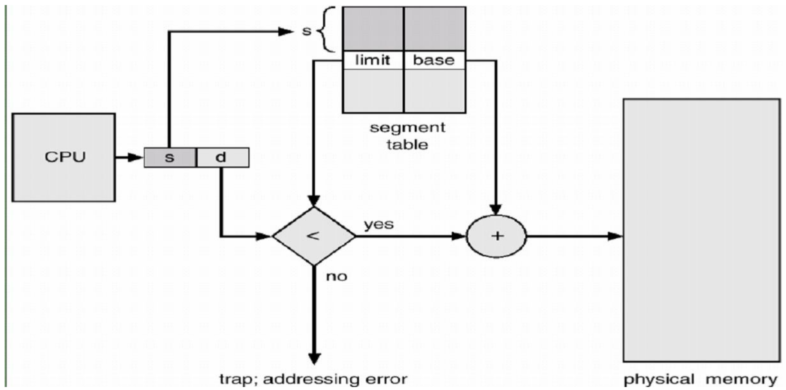
- כאשר תהליך נטען לזיכרון, רגיסטר מיועד מקבל את הכתובת ההתחלתית של טבלת הסגמנטים המתאימה לו
- בהינתן כתובת לוגית (מספר הסגמנט,היסט) המעבד מתאים את מספר הסגמנט לטבלה ומשם לוקח את תחילת כתובתו הפיזית של הסגמנט ואורכו
- חישוב הכתובת הפיזית נעשה ע"י הוספת ההיסט לתחילת כתובתו הפיזית של הסגמנט (ראה איור)
  - החומרה גם משווה בין גודל הסגמנט להיסט בכדי לקבוע האם הכתובת חוקית



ארכיטקטורת הפילוח :

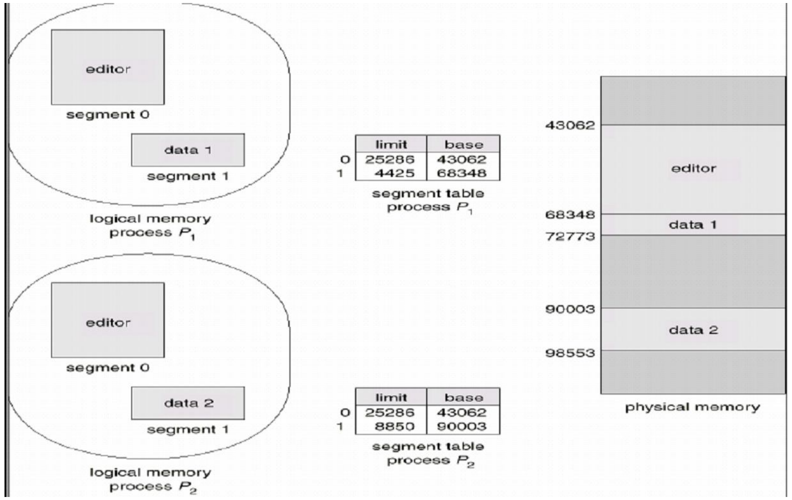
- **Logical address** - מכילה את השדות <segment-number, offset>

- **Segment table** - ממפה זיכרון פיזי , מכילה את השדות הבאים :
  - Base - תחילת כתובתו הפיזית של הסגמנט בזיכרון
  - Limit – אורך הסגמנט
- *Segment-table base register (STBR)* - מיקום טבלת הסגמנטים בזיכרון
- *Segment-table length register (STLR)* - כמה סגמנטים בשימוש ע"י התוכנה , הסגמנט מוגדר חוקי אם מתקיים *STLR* >S



סוגיות לגבי פילוח :

- הקצאת זיכרון (Allocation) :
  - הקצאה דינאמית
  - קיים רק external fragmentation
- מיקום מחדש (Relocation) :
  - דינאמי
  - נעשה ע"י טבלת הסגמנטים
- שיתוף :
  - שיתוף סגמנטים ע"י הליכים שונים
  - חלק מן הסגמנט משותף לשניהם , סגמנט Data ייחודי לכל הליך (ראה איור)



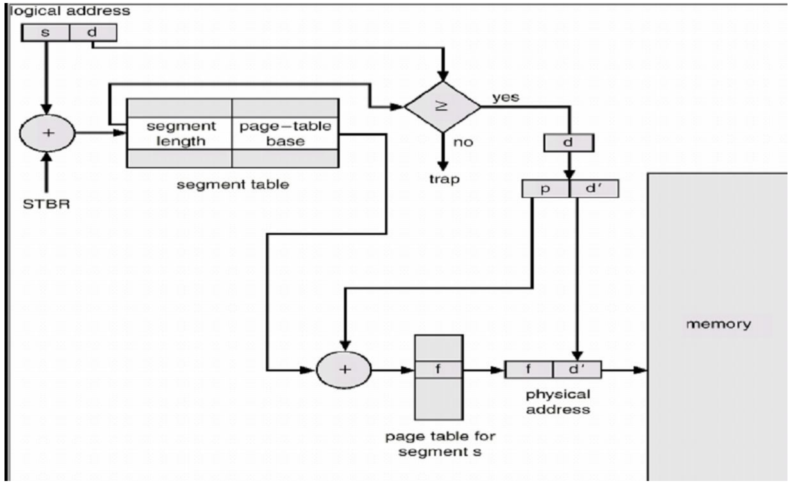
- הגנה :
  - ביט Validation – אם ערכו 0 הסגמנט לא חוקי
  - הרשאות קריאה/כתיבה/ביצוע
- מכיוון שאורך כל סגמנט שונה הקצאת הזיכרון לסגמנטים יוצרת בעיה

השוואה בין Paging ל Segmentation :

- פילוח נראה לעיניי המשתמש בניגוד לPaging
- הפילוח תומך בהגנה ובשיתוף
- פילוח נותן יתרון למשתמש – ניתן לתת לחלקים שונים בקוד הגנה שונה
- גודל הסגמנט הוא לא קבוע בניגוד לגודל דף
- פילוח דורש לתרגום הכתובת הלוגית
- פילוח סובל מ External fragmentation בעוד Paging סובל רק מחלק קטן של Internal fragmentation

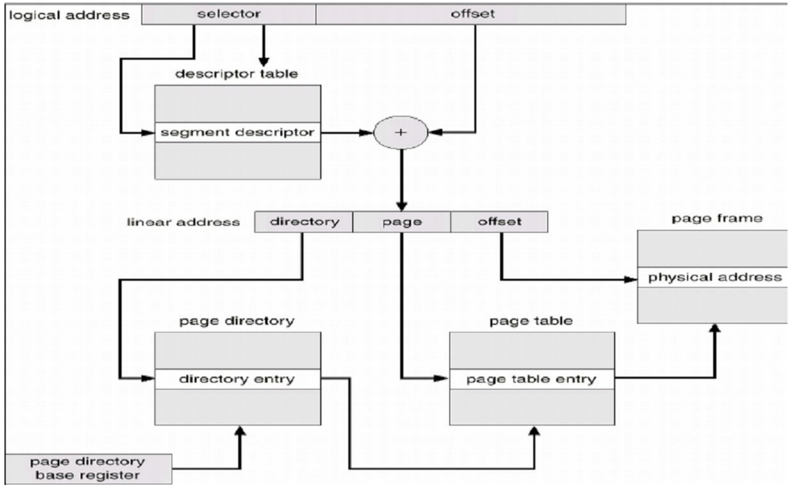
שילוב בין Paging ל Segmentation - Multics :

- מערכת הMultics פתרה בעיות של External fragmentation וזמני חיפוש גבוהים ע"י paging של הסגמנטים
- כעת מופע בטבלת הסגמנטים מכיל גם את הכתובת ההתחלתית של הpage table המתאימה לסגמנט זה



דוגמא – מעבד 386 של אינטל :

- מעבד זה משתמש במולטיקס עם 2-way level paging

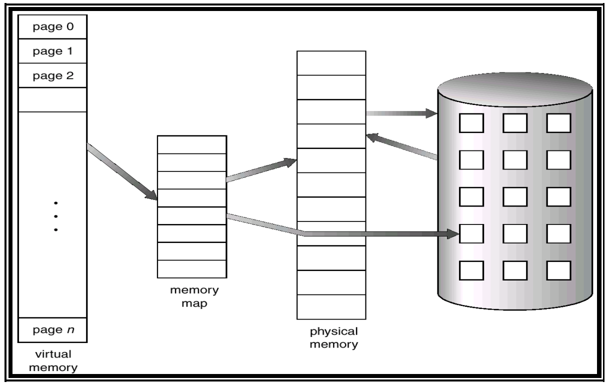


הרצאה 8: ניהול זיכרון וירטואלי

רקע

- מבוסס על דפדוף,סגמנטציה, תהליך יכול להתפרק לחתיכות (דפים או סגמנטים) אשר לא צריכים לשבת באופן רציף בזיכרון.
- על בסיס עיקרון הלוקאליות, כל החתיכות של התהליך אינן צריכות להיטען לזיכרון הראשי במהלך הריצה. כל הכתובות הן וירטואליות.
- זיכרון אשר מיוחס לכתובות וירטואלי נקרא זיכרון וירטואלי:
  - בעיקר מתוחזק בזיכרון המשני (דיסק)
  - חתיכות מהתהליך מגיעות לזיכרון הראשי רק כאשר צריך אותן

Virtual Memory that is larger than Physical Memory



Advantages of Partial Loading - יתרונות בהטענה חלקית

- יותר תהליכים יכולים להיות מתוחזקים בזיכרון הראשי:
  - טוען רק חלקים מכל תהליך
  - כאשר יש יותר תהליכים בזיכרון הראשי, יש יותר סיכוי לכך שתהליך יהיה במצב Ready state בזמן נתון.
- כעת גם תהליך גדול יותר מגודל הזיכרון הראשי יכול לרוץ:
  - אפשרי אפילו להשתמש ביותר ביטים עבור כתובת לוגית מאשר הדרושים לכתובת פיזית.
- דוגמא:
  - רק 16 ביטים דרושים למען נפח זיכרון של 64KB.
  - נשתמש בגודל דף של 1KB כך ש-10 ביטים דרושים עבור הסטים בעמוד.
  - עבור החלק של מספר העמוד בכתובת הלוגית אנו יכולים להשתמש במספר ביטים גדול מ-6, נגיד 22 (ערך צנוע!), בהנחה וכתובת היא בגודל 32 ביטים.

Support Needed for Virtual Memory - תמיכה הדרושה לזיכרון וירטואלי

- חומרת ניהול הזיכרון צריכה לתמוך ב-עמודים,סגמנטים.
- מערכת הפעלה צריכה להיות מסוגלת לנהל מעברים של סגמנטים ודפים בין זיכרון חיצוני לזיכרון הראשי, כולל הצבה והחלפה של סגמנטים/דפים.

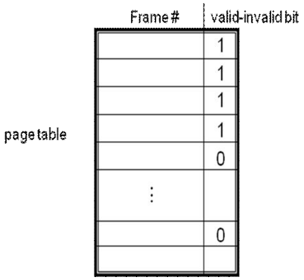
<u>הרצת תהליך - Process Execution</u>

- מערכת ההפעלה מביאה לזיכרון הראשי רק מספר חתיכות של התכנית (כולל את נקודת ההתחלה).
- כל ערך בטבלת הסגמנט/עמוד מכילה ביט תוקף (valid) שערכו 1 רק כאשר החתיכה המתאימה שלו נמצאת בזיכרון הראשי.
- Resident set - חלק מהתכנית שנמצא בזיכרון הראשי בשלב מסוים.
- Memory fault נוצר כאשר שיוך זיכרון לחתיכה לא נמצא בזיכרון הראשי.
- מערכת ההפעלה שמה את התהליך במצב חסום (blocking state).
- מערכת ההפעלה דואגת לקריאת I/O מהדיסק להבאת הנתון לזיכרון הראשי.
- עוד תהליך משוגר לעבוד בזמן הקריאה מהדיסק.
- פסיקה מופעלת בסיום הקריאה מהדיסק, זה גורם למערכת ההפעלה לשים את התהליך הרלוונטי במצב Ready state חזרה.

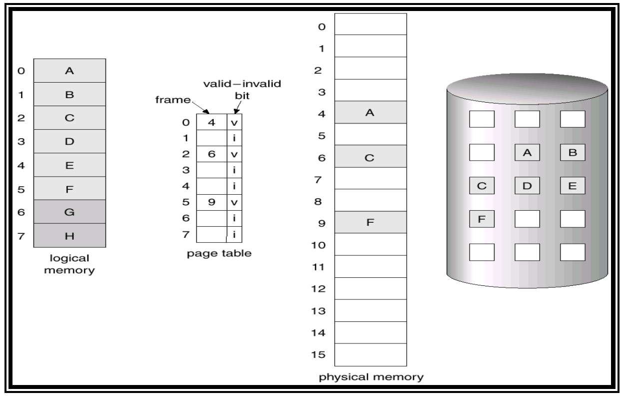
<u>הרעיון של זיכרון וירטואלי - Idea of Virtual Memory</u>

- הפרדה אחרונה בין זיכרון לוגי של משתמש לבין זיכרון פיזי:
  - רק חלק מן התוכנה צריך להיות בזיכרון במהלך ריצה.
  - מרחב הזיכרון הלוגי יכול להיות הרבה יותר גדול ממרחב הזיכרון הפיזי.
  - מאפשר יצירה יעילה יותר של תהליכים.
  - מאפשר שיתוף מרחב כתובות ע"י כמה תהליכים.

- זיכרון וירטואלי יכול להיות ממומש ע"י:
  - Demand Segmentation
  - Demand Paging
    - הבאת עמוד לזיכרון רק שצריך אותו:
      - פחות פעולות I/O נדרשות
      - פחות זיכרון נדרש
      - תגובה מהירה יותר
      - יותר משתמשים
    - זקוקים לעמוד – הפניה אליו:
      - הפניה פגומה – ביטול
      - לא נמצא בזיכרון – הבאה לזיכרון
    - ביט תוקף (valid bit) :
      - עבור כל ערך בטבלת העמודים, קיים ביט שקובע האם העמוד בזיכרון (1=) או לא קיים בזיכרון (0=).
      - תחילה על הערכים בטבלה לביט זה מאותחלים ל-0.
      - כאשר מתרגמים את הכתובת , אם הביט שווה ל-0 זה נקרא page fault.

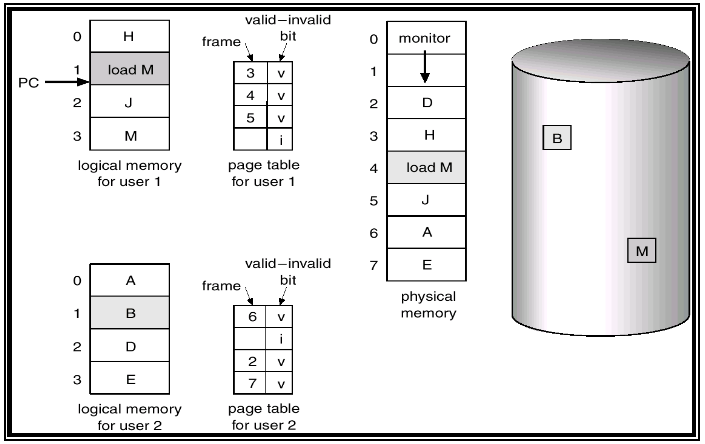


<u>Page Table When Some Pages Are Not in Main Memory</u>

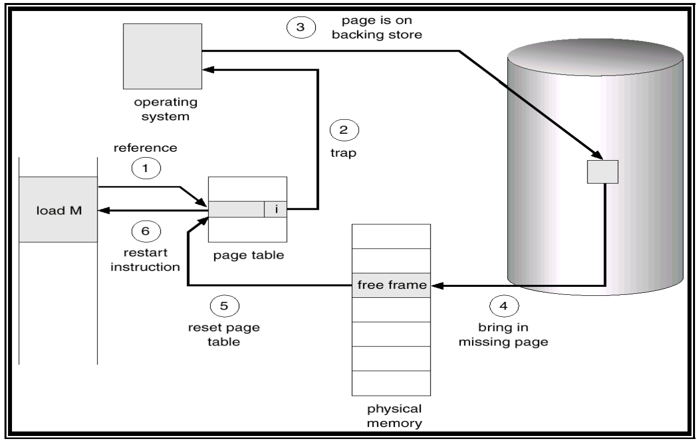


- דינמיקת Demand Page:
  - באופן טיפוסי, לכל תהליך טבלת דפים משלו.
  - כל ערך בטבלה מכיל ביט תוקף (valid bit) שקובע האם הדף בזיכרון הראשי או לא.
    - אם כן, הערך בטבלה מכיל את ה- frame number של העמוד המתאים בזיכרון הראשי.
    - אם לא, הערך בטבלה אולי מכיל את הכתובת של העמוד בדיסק, או שמספר הדף מהווה אינדקס לטבלה נוספת (בד"כ ב- PCB) כדי להביא את הדף מהדיסק.
  - ביט נוסף (modified bit) קובע האם העמוד השתנה מאז הפעם האחרונה שהוא נטען לזיכרון הראשי:
    - אם לא נעשה שינוי, הדף לא צריך להיכתב לדיסק לפני שהוא מוחלף.
  - ביטים נוספים יכולים להיות נוכחים אם ההגנה מתוחזקת ברמת העמוד:
    - ביט - read/write only.
    - Protection level bit- עמוד משתמש/kernel (אם יש יותר מ-2 רמות הגנה משתמשים ביותר מביט אחד).

<u>Need for page fault/replacement</u>

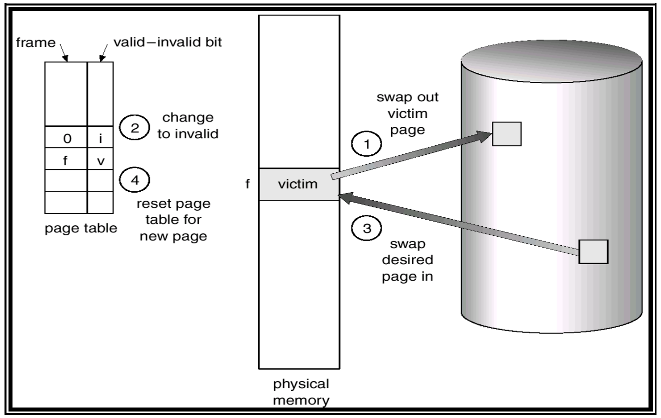


<u>Steps in handling a Page Fault</u>



- צעדים בטיפול ב- Page fault:
  - אם יש הפניה לעמוד שלא בזיכרון, פניה ראשונה אליו תגרום ל- page fault.
  - Page fault מטופל ע"י מערכת ההפעלה בעזרת רוטינות השירות המתאימות.
  - מצא את העמוד הרלוונטי בדיסק.
  - החלף עמוד במסגרת חופשייה (בהנחה וזמין)
  - אתחל טבלאות עמודים – ביט תוקף = 1.
  - אתחל את הפקודה

<u>Steps in handling a Page replacement</u>



- צעדים בטיפול ב- Page Replacement:
  - מצא את מיקום העמוד הדרוש בדיסק.
  - מצא מסגרת חופשייה:
    - אם יש, תשתמש בה.
    - אם אין, השתמש באלגוריתם החלפת עמוד ובחר עמוד קורבן.
    - קרא את העמוד הדרוש למסגרת החופשייה, עדכן את טבלאות העמוד.
    - אתחל את התהליך.

- הערות על- Page Replacement:
  - מנע הקצאה מיותרת של זיכרון ע"י שינוי רוטינת שירות עבור page fault לכלול page replacement.
  - שימוש ב- dirty bit כדי להפחית את מספר העברות הדפים – רק דפים שהשתנו יכתבו לדיסק.
  - Page replacement משלימה את הפרדת הכתובת הלוגית מהכתובת הפיזית, כתובת לוגית גדולה יכולה להתאפשר עבור זיכרון פיזי קטן.

- הצורך ב-TLB:
  - כיוון שטבלת הדפים היא בזיכרון הראשי, כל פניה לזיכרון הוירטואלי צורך 2 גישות לזיכרון הפיזי:
    - אחת כדי להביא את הערך המתאים מטבלת הדפים
    - אחת כדי להביא את המידע
  - כדי להתגבר על בעיה זו יש צורך בזיכרון מטמון מיוחד אשר יכיל ערכים של טבלת הדפים, שמו TLB (Translation Look-aside buffer):
    - מכיל את הערכים מטבלת הדפים שהיו הכי הרבה בשימוש לאחרונה.
    - עובד בדומה לזיכרון מטמון של הזיכרון הראשי.
  - דינמיקת TLB:
    - מתקבלת כתובת לוגית, המעבד מחפש אותה ב-TLB.

- אם היא נמצאה (hit), מוחזר ה- frame number והכתובת הפיזית מחושבת.
- אם היא לא נמצאה (miss) מספר העמוד (page number) משמש כדי לחפש בטבלת העמודים:
  - אם ה- valid bit מאותחל (=1), ניגשים ל-frame המתאים ובונים את הכתובת הפיזית.
  - אם ה - valid bit אינו מאותחל (=0), זהו page fault ויש להביא את הדף מהזיכרון הראשי.
- טבלת ה-TLB מעודכנת לכלול את הערך החדש.

- TLB – הערות נוספות:
  - TLB משתמשת בחומרת מיפוי אסוציאטיבית, כדי לחקור בו זמנית את כל הערכים בטבלה על מנת למצוא התאמה למספר העמוד.
  - חייבים לעשות flush ל-TLB בכל פעם שתהליך חדש נכנס ל-Running state.
  - המעבד משתמש ב-2 רמות cache על כל הפניה לזיכרון וירטואלי:
    - קודם ה-TLB: כדי להמיר את הכתובת הלוגית לכתובת פיזית.
    - ברגע שיש את הכתובת הפיזית המעבד מחפש בcache הרגיל את המילה.

- ביצועי Demand Paging:
  - קצב page fault ($0<=p<=1$):
    - אם $p=0$, אין page faults
    - אם $p=1$, עבור כל פניה יש page fault
  - EAT – Effective Access Time
    - EAT = (1-p) * memory access + p * (page fault overhead + [swap page out] + swap page in + restart overhead)
  - דוגמא:
    - Memory access time = 1 micro sec
    - 50% מהזמן העמוד המוחלף השתנה, ולכן צריך להתחלף החוצה.
    - Swap page time = 10ms = 10000micro sec
    - $EAT = (1-p)*1+p*(5000+10000)=1+15000p$ (in microseconds)

- דינמיקת סגמנטציה (Dynamics of Segmentation)
  - בד"כ לכל תהליך יש טבלת סגמנטציה.
  - בדומה – paging, כל ערך בטבלה מכיל את הכתובת ההתחלתית ואורך הסגמנט.
  - ביטים של שליטה אחרים יכולים להיות נוכחים במידה והגנה/שיתוף נתמכים ברמת הסגמנט.

  - תרגום כתובת לוגית לפיזית דומה ל- paging, רק שהחישוב הוא חיבור ה-offset לכתובת ההתחלתית (במקום הוספה).

- הערות על סגמנטציה
  - בכל ערך בטבלה יש לנו גם את הכתובת ההתחלתית וגם את האורך של הסגמנט, לפיכך הסגמנט יכול בצורה דינמית לגדול/לקטון לפי הדרישה.
  - But variable length segments introduce external fragmentation and are more difficult to swap in and out.
  - זה טבעי לספק הגנה ושיתוף ברמת הסגמנט כיוון שסגמנטים הם גלויים למתכנת בעוד דפים אינם.
  - ביטים להגנה מועילים בטבלת הסגמנטציה:
    - Read-only/Read-Write bit
    - Kernel/User bit

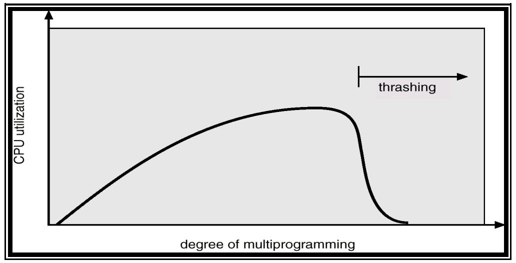
- Combined Segmentation and paging
  - על מנת לשלב את יתרונותיהם, מספר מערכות הפעלה מעמדים (page) את הסגמנטים.
  - כמה שילובים קיימים – בהנחה שלכל תהליך יש:
    - טבלת סגמנטים אחת
    - כמה טבלאות דפים – טבלת דפים עבור כל סגמנט
  - הכתובת הוירטואלית מורכב מ:
    - מספר סגמנט – לשימוש אינדוקס טבלת הסגמנטים אשר בערך בה נותן את הכתובת ההתחלתית של טבלת העמודים עבור אותו סגמנט.
    - מספר עמוד – לשימוש אינדוקס טבלת העמודים כדי לקבל את ה- frame number.
    - Offset – לשימוש איתור המילה ב- frame.

- Simple combined Segmentation and paging
  - בסיס הסגמנט הוא הכתובת הפיזית של טבלת העמודים עבור אותו סגמנט.
  - Present/modified bits – נוכחים רק בערך טבלת העמודים.
  - באופן הכי טבעי - מידע ההגנה והשיתוף יושבים בערך של טבלת הסגמנטים

- שיקולי paging:
  - לוקאליות, זיכרון וירטואלי, Trashing
  - נושא גודל עמוד
  - TLB reach
  - מבנה תוכנית
  - קישור I/O
  - יצירת תהליך

<u>לוקאליות וזיכרון וירטואלי</u>

- עקרון הלוקאליות: גישות לזיכרון נוטות להתקבץ.
- לפיכך – רק כמה חתיכות של התהליך יידרשו בהמשך זמן קצר.
- אפשרות לניחושים מושכלים עבור אילו חתיכות נצטרך בעתיד.
- זה מציע שזיכרון וירטואלי יכול לעבוד בצורה יעילה (trashing לא צריך לקרות לעיתים קרובות)



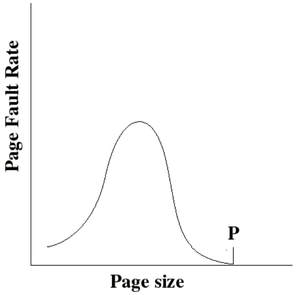
- למה paging עובד?
  - מודל הלוקאליות:
    - תהליך מהגר מלוקאליות אחת לאחרת
    - לוקאליות יכולה לחפוף
- למה trashing מתרחש?
  - סכום הגדלים של הלוקאליות גדול מגודל הזיכרון.

<u>הסיכוי ל- trashing</u>

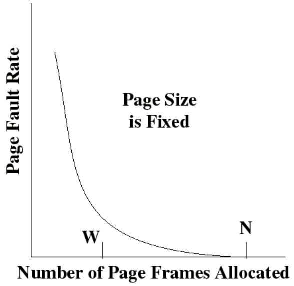
- Trashing = תהליך עסוק בהחלפת דפים פנימה והחוצה.
- אם לתהליך אין "מספיק" דפים, ה- page fault rate מאוד גבוה, זה גורם ל:
  - ניצולת נמוכה של המעבד
  - מערכת ההפעלה חושבת שהיא צריכה להגדיל את רמת ה- Multiprogramming
  - עוד תהליך נוסף למערכת
  - זה רק מגדיל את העומס על הזיכרון הפיזי
- כדי לספק את המקסימום תהליכים האפשריים, רק כמה חתיכות של כל תהליך מתוחזקות בזיכרון הראשי.
- אך הזיכרון הראשי יכול להיות מלא – כאשר מערכת ההפעלה מביאה חתיכה אחת פנימה, היא חייבת להחליף בחתיכה אחרת.
- אסור למערכת ההפעלה להחליף החוצה חתיכה שיש בה שימוש בקרוב
- אם היא עושה זאת לעיתים קרובות מידי זה מוביל ל- trashing:
  - המעבד מבזבז את רוב הזמן בהחלפת חתיכות במקום להריץ פקודות משתמש.

<u>נושא גודל הדף - The Page Size Issue</u>

- גודל דף מוגדר ע"י חומרה, תמיד בחזקת 2 עבור תרגום יעיל יותר מכתובת לוגית לפיזית. אך לקבוע בדיוק איזה גודל זו שאלה קשה:
  - דפים גדולים זה טוב כיוון שעבור דפים קטנים נצטרך הרבה דפים עבור כל תהליך וזה אומר טבלאות דפים גדולות יותר, וזה אומר הרבה טבלאות דפים בזיכרון הוירטואלי.
  - כמו-כן דפים גדולים זה טוב כיוון שדיסקים מתוכננים (באופן יעיל) להעביר בלוקים גדולים של מידע.
  - דפים גדולים זה אומר שיש פחות דפים בזיכרון הראשי, זה מגדיל את ה- hit rate עבור ה-TLB.
  - דפים קטנים טובים למזער את הפרגמנטציה הפנימית.
  - עם דפים ממש קטנים, כל עמוד מתאים לקוד שבשימוש – faults are low.
  - עבור דפים גדולים כל דף מכיל יותר קוד שלא בשימוש – page faults rise.
  - Page faults יורדים אם אנו מגיעים לנקודה P בה גודל העמוד שווה לגודל כל התהליך.



- קצב ה-Page fault נקבע גם ע"פ הפריימים המוקצים עבור כל תהליך.
- Page faults יורדים לערך סביר כאשר W פריימים מוקצים (ראה שרטוט).
- Page faults יורדים ל-0 כאשר מספר הפריימים הוא כך שכל התהליך בזיכרון.



- דפים בגודל 1KB-4KB הם הכי נפוצים בשימוש. הגדלת גודל העמוד מקושר לטרנד הגדלת גודל הבלוקים.
- אך הנושא אינו טריוויאלי. כמה מעבדים תומכים כעת בריבוי גדלי עמודים, לדוגמא:
  - פנטיום תומך ב-2 גדלים – 4KB, 4MB
  - R4000 תומך ב-7 גדלים – 4KB to 16MB

<u>TLB Reach</u>

- הגדרה – כמות הזיכרון הנגישה מטבלת ה- TLB.
- TLB Reach = TLB size * Page Size
- באופן אידיאלי, הסט המקומי של כל תהליך שמור ב-TLB, אחרת יש דרגה גבוהה של page faults.

<u>הגדלת גודל ה- TLB Reach</u>

- הגדלת ה-TLB – יכול להיות יקר.
- הגדלת גודל העמוד – זה יכול לגרום להגדלת הפרגמנטציה הפנימית כאשר לא כל אפליקציה דורשת דפים גדולים.
- לספק ריבוי גדלים של דפים – זה מאפשר לאפליקציות שמשתמשות בדפים גדולים ההזדמנות לשימוש בהם ללא הגדלת הפרגמנטציה.

<u>מבנה תוכנית</u>

Program structure

int A[][] = new int[1024][1024];

Each row is stored in one page.

```
Program 1:   for (j = 0; j < A.length; j++)
               for (i = 0; i < A.length; i++)
                 A[i,j] = 0;
```

we have 1024 x 1024 page faults

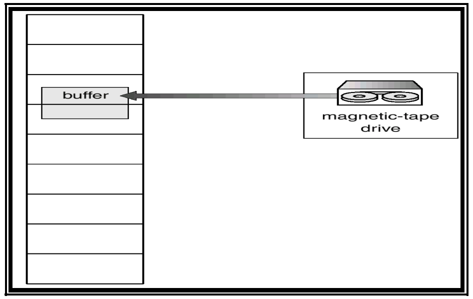
```
Program 2:   for (i = 0; i < A.length; i++)
               for (j = 0; j < A.length; j++)
                 A[i,j] = 0;
```

we have 1024 page faults

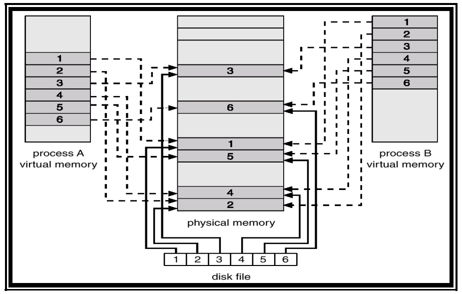
<u>I/O Interlock</u>

- I/O Interlock – לפעמים צריך לנעול עמודים בזיכרון.
- קח בחשבון I/O. עמודים שבשימוש להעתקת קובץ מהתקן, חייבים להיות נעולים מבחירת אלגוריתם page replacement.

<u>Why frames used for I/O must be in memory</u>



<u>יצירת תהליך - Process Creation</u>

- זיכרון וירטואלי מאפשר הטבות אחרות במהלך יצירת תהליך:
  - Copy-on-Write (COW)
    - מאפשר לתהליך אבא ובן בתחילה לחלוק (share) את אותם הדפים בזיכרון. אם אחד מהתהליכים משנה דף משותף, רק אז הדף מועתק.
    - מאפשר יצירת תהליך יותר יעילה כי רק דפים שהשתנו מועתקים.
  - Memory-mapped files
    - Memory-mapped file I/O – מאפשר לקובץ קלט/פלט להיות מיוחס כרוטינת גישה לזיכרון ע"י מיפוי בלוק דיסק לדף בזיכרון.

- תחילה, קובץ נקרא בעזרת demand paging. חלק מהקובץ בגודל עמוד נקרא ע"י מערכת הקבצים לתוך דף פיזי. קריאות/כתיבות עוקבות מהקובץ/לקובץ מיוחסות כגישות רגילות לזיכרון.
- מפשט גישה לקובץ ע"י התייחסות ל-file I/O דרך הזיכרון במקום קריאות מערכת (system calls) – ()read, ()write
- כמו-כן מאפשר לכמה תהליכים למפות את אותו הקובץ, שמאפשר לדפים הזיכרון להיות משותפים.