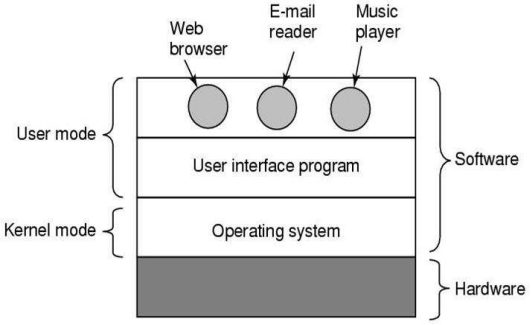
מערכות הפעלה לביולוגים (89-288) – סמסטר א' תש"ע (הרצאות) / אריאל פרנק

הרכב הציון: המבחן 60%, התרגילים 40% (יהיו 3-4 תרגילי תכנות בשפת C, בסביבת לינוקס)
הסילאבוס: www.cs.biu.ac.il/~ariel/os.html
אתר הקורס (בו נמצאות המצגות): www.cs.biu.ac.il/~ariel/download/os288
אתר הקורס בו נמצאות המצגות להרצאות 2-4 (מהקורס 'ארגון קבצים'): http://u.cs.biu.ac.il/~ariel/download/fo287/
מצגות ישנות (ועדכונים מהמרצה) נמצאות במערכת ה-high-learn בכתובת: http://hl2.biu.ac.il/

הרצאה מס' 1 - 19.10.09

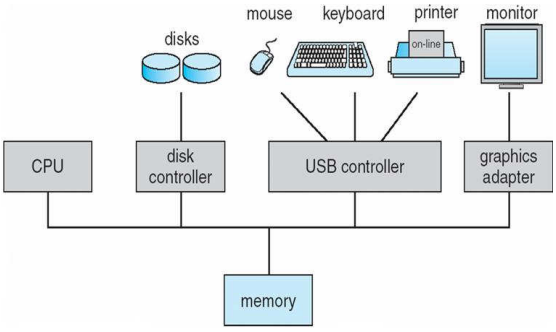
[מצגת os1-1 int]
הספר הראשון ברשימת הביבליוגרפיה מומלץ. [חלק מהספרים הצלחתי להוריד מטורנט בפורמט pdf, לטובת מי שמעוניין.]
באמצע שנות ה-40 התחילו להופיע המחשבים הגדולים, ה- main frame. המחשבים האישיים הופיעו לראשונה באמצע שנות ה-80.
מחשב מורכב ממעבד (או יותר) הנקרא יע"מ (יחידת עיבוד מרכזית) או cpu (central processing unit), התקנים חיצוניים וזיכרון.

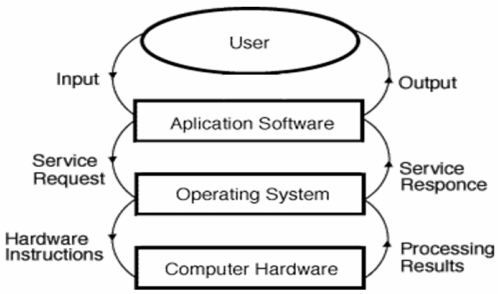


מערכת ההפעלה מנהלת את כל אלו, היא מהווה מנשק [זו הדרך הנכונה לומר, לא אומרים ממשק..] המקשר בין החומרה והמשתמש.
מערכת ההפעלה צריכה להיות ידידותית למשתמש כך שתנצל את משאבי המחשב באופן יעיל. החומרה זה הבסיס (הליבה = מרכז). יש הפרדה בין חומרה לתוכנה והפרדה נוספת בתוך התוכנה לחלק של מערכת ההפעלה (kernel) והחלק של המשתמש (user mode).

מה שמסופק ע"י מערכת ההפעלה:
- אמצעים ליצירת תוכנית: עורכי-תמלילים (בהם כותבים את הקוד), קומפיילרים, לינקרים, דיבאגרים..
- אמצעים לביצוע של תוכניות: גישה לזיכרון, פתיחת קבצים..

מה החשיבות של מערכות הפעלה?
מכילה מרכיבים הקשורים לארכיטקטורה, אלגוריתמיקה, מבני נתונים..





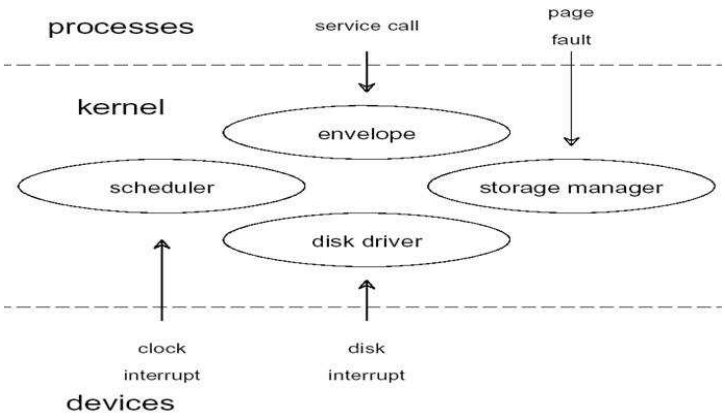
ארגון מערכת המחשב: הקו המחבר בין הוא בעצם ה-system bus ('פס המערכת') המקשר בין כל החלקים (זיכרון, מעבד, בקרים) ומשמש להעברת הנתונים בין החלקים.

מרכיבי המחשב:
- חומרה – זיכרון, מעבד, התקנים חיצוניים.
- מערכת הפעלה.
- תוכנות (למשל word..).
- המשתמש ☺

<u>יש 3 היבטים עיקריים למערכת הפעלה</u>:
- מנהלת המשאבים – מנהלת את משאבי המחשב (זיכרון, מעבד, התקנים..), מטפלת ומקצה משאבים לתוכניות הרצות באותו זמן על אותו זיכרון, פותרת התנגשויות בין הדרישות השונות לצורך ניצול יעיל של המשאבים.
- תוכנית שליטה / בקרה, האחראית על כל מה שמתבצע על המחשב – מנהלת את המרכיבים הפועלים וההתקנים החיצוניים, מבקרת ביצוע של תוכניות המשתמש.
- מבצעת הפקודות של המשתמש, מספקת סביבה להרצת היישומים – היבט אנושי, המערכת משרתת את המשתמש, מהווה מנשק בין המשתמש והמכונה. CLI = command language interface.
- היבט רביעי, מודרני יותר – מערכת הפעלה וירטואלית (virtual machine), אוסף של הרחבות תוכנה שהרמה הגבוהה ביותר היא זו שהמשתמש בא איתה במגע.

[בדיחה פלינדרומית: אדם אמר לחווה madam i'm adam – וחווה ענתה לו eve]
memory = זיכרון (למשל RAM). storage = החסן (התקנים חיצוניים כגון דיסקטים, דיסקים). זמזומילה = buzzword.

הגדרה למערכת הפעלה: אין הגדרה מקובלת אוניברסלית. החלק המינימלי של מערכת ההפעלה שרץ בכל רגע נתון – kernel.

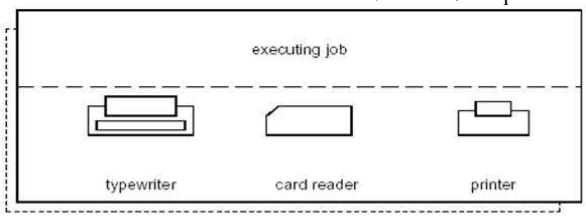


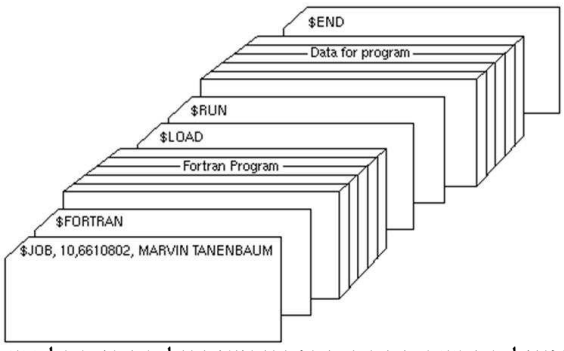
[מצגת os1-2 int] <u>אבולוציה של מערכות הפעלה</u>
מערכת הפעלה צריכה להיות מסוגלת להתמודד עם שדרוגי חומרה וסוגי חומרה חדשים, לספק תמיכה בשירותים חדשים (כגון חיבור לאינטרנט).

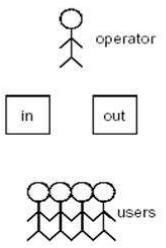
[בדיחה גסטרונומית: כאשר מזמינים חבר לארוחת ערב והוא מרוצה, למחרת הוא יזמין אותנו לארוחה שהוא בישל בעצמו. הדבר נקרא feed-back, הוא כביכול מאכיל אותנו בחזרה ☺]

המערכות המוקדמות ידעו להתמודד רק עם משתמש יחיד בכל רגע נתון (open shop – ברגע שנכנס לקוח החנות נסגרה וטיפלה רק בו). הקלטים למערכת היו כרטיסים מנוקבים. המערכות הראשונות היו מאד פרימיטיביות, קיבלו קלטים בינאריים. לאחר מכן התקדמו לבסיס אוקטלי (8) ובהמשך לבסיס הקסה-דצימלי (16) כדי לחסוך בסיביות. בתחילה כל התוכניות הכילו את **<u>כל</u>** המידע הנחוץ למערכת: קריאת כרטיס, מידע ראשוני וכן את הקוד הרצוי. זה כמובן יצר בזבוז, כי כל תוכנית דרשה זמן ריצה ראשוני של כל האתחולים הללו. בהתאם, ניצולת המעבד הייתה נמוכה, רוב הזמן נדרש עבור ההכנות, לפני שבכלל התחיל להתבצע הקוד הרצוי. היתרון היה באבטחה; לא היו וירוסים ושאר נוזקות (malware).

בהמשך היה מעבר למערכות אצווה (מלשון 'צוות') batch. שימוש בשפות ברמה גבוהה, קלטות מגנטיות במקום כרטיסים מנוקבים. רעיון האצווה – לא כל משתמש יריץ את התוכנית שלו, אלא יריצו ביחד את כל התוכניות שהיו צריכות לרוץ בשפה מסוימת, לאחר מכן את כל התוכניות שהיו צריכות לרוץ בשפה אחרת (קיבוץ תוכניות בהתאם לסוגן). על הקיבוץ הזה היה אחראי dispatcher ('משלח').







המתכנת מוסר את העבודה (התוכנית שהוא רוצה שתרוץ) שלו, מפעיל המחשב ממקם מספר עבודות על התקן הקלט של המחשב, תוכנית מחשב מיוחדת בשם 'מוניטור' מנהלת את הביצוע של כל עבודה ב-batch. היתרון הוא בחסכון זמן האתחול (setup) ע"י קיבוץ עבודות מסוגים דומים.

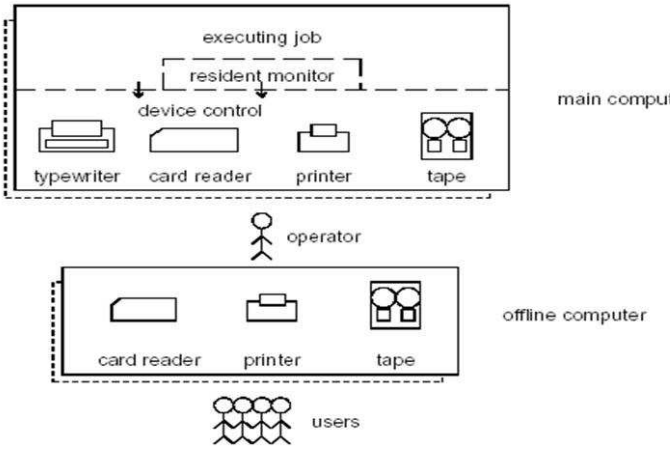
- JCL is the language that provides instructions to the monitor:
  - what compiler to use
  - what data to use
- Example of job format: ------->>
  - `$FTN` loads the compiler and transfers control to it.
  - `$LOAD` loads the object code (in place of compiler).
  - `$RUN` transfers control to user program.

```
$JOB
$FTN
...
FORTRAN
program
...
$LOAD
$RUN
...
Data
...
$END
```

שאלה: כיצד בעצם יודע המוניטור היכן בקלט מתחילה / מסתיימת עבודה, היכן מסתיים הקוד ומתחילים הנתונים, לאיזה סוג שייכת כל עבודה..? את המידע הזה העבירו ל'פקח תושב' (resident monitor) ע"י פקודות JCL.

resident monitor – הניצן של מערכת ההפעלה, כלל מרכיבים של הזנה ותרגום כרטיסי הקלט, הטענת התוכניות הרצויות והפעלת ההתקנים הדרושים לעבודה הרצה באותו זמן. אסור שעבודה מסוימת תשבש את החלק בזיכרון עליו יושב הקוד של ה-monitor.

בעיה אפשרית היא שה-monitor מאפשר לעבודה לרוץ, בעצם מעביר לידיה את השליטה על משאבי המחשב ואילו התוכנה נכנסה ללולאה אינסופית ולא החזירה את השליטה ל-monitor. בהמשך הוסיפו מחשבי לווין (offline operation) שטיפלו בקלט / פלט, כדי לאפשר למחשב המרכזי להיות זמין יותר לריצה של העבודות.



הפעולות היו בעצם במקביל: קריאת העבודה, הרצתה והדפסת הפלט שלה. כמובן שהדבר יקר יותר, כי צריך מספר מחשבים. Spool – הפעלה מקבילית של פעולות: תוך כדי ריצה של עבודה אחת, נעשית קריאת הקלט של העבודה הבאה והדפסת הפלט של העבודה הקודמת.
החסרון: כאשר רק תוכנית משתמש אחת רצה על המחשב (uni-programming), חלק מהזמן מתבזבז על המתנה בין ריצות של עבודות, משמע ניצול נמוך של המעבד (poor CPU usage).

תכנות מרובה (batch multi-programming) – מספר עבודות שמורות על הזיכרון המרכזי באותו זמן, כדי שבזמן ריצה של עבודה אחת, התוכנית הבאה תבצע את ההכנות הדרושות. זה מאפשר לניצולת המעבד להיות גבוהה יותר. כמובן שה'פקח תושב' צריך להיות חכם יותר כדי לבצע את המעברים האלו בין התוכניות באופן מהיר ויעיל. לצורך כך יש לו רשימת התוכניות שצריכות לרוץ.

[מצגת os1-3 int]
time-sharing – חלוקת זמן יעילה. ההיגיון: המחשב מהיר יותר מהמשתמש הבודד. המעבד מבצע מעברים בין העבודות באופן מהיר. דרוש זמן תגובה מהיר (של פחות משניה).

[בדיחה אינטרנטית – WWW פירושו world wide wait, כי מתבאסים כשמחכים לסרט שייטען.. ☺]

real-time systems – מערכות שצריכות לעבוד ב'ספי-זמן' (dead-lines). סביבת hard RT דורשת עמידה עיקשת בלוח הזמנים ואילו סביבת soft RT כגון מולטימדיה ברשת יכולה להתעכב (כמו שמחכים לטעינת סרט באתר אינטרנט).

[בדיחה מלחמתית – אם האחראי על מנגנון ירי טילי ההגנה בספינת קרב ישחק תוך כדי במשחק doom, שיגור הטיל הנגדי יתעכב, הספינה תופצץ ואכן יהיה doom.. ☺]

personal / desktop computers – סגירת מעגל, חזרנו למצב של משתמש מול מחשב. המחשב הוא בעל מעבד יחיד, בכל רגע נתון מתבצעת תוכנית יחידה. כיום כבר יש מעבדים שהם dual / quad, ריבוי של מעבדים, דבר שמאפשר להריץ מספר תוכניות במקביל.
<u>סוגים של מערכות רב-מעבדים</u>:
(1) א-סימטרי – מעבד אחד (master) השולט ומזמנן (מלשון schedule) את העבודות למעבדי ה-slave.
(2) סימטרי (SMP, symmetric multi-processing) – כל אחד מהמעבדים מריץ עותק זהה של אותה מערכת הפעלה, כל מעבד מזמנן לעצמו את המשימות שרצות עליו. מערכות הפעלה מודרניות תומכות ב-SMP (גם לכם יש מחשב core 2 duo ?).

Clustering (אישכול) – מספר מערכות מחשב המחוברות ביניהן, מה שמאפשר מהירות גבוהה של קלט / פלט. מכילות גם זיכרונות נפרדים וגם זיכרון משותף. מטרות האשכול – HPC, חישובים מקביליים מהירים. Networked systems – מחשבים המחוברים דרך רשת. לעומת מערכת-רשת, יש גם מערכת מבוזרת – ניסיון להפוך מערכת רשתית לשקיפה יותר, סוג של מחשב-על, היודע לנצל את כל ריבוי המחשבים במערכת.

הרצאה מס' 2 - 26.10.09

[מצגת fo1-2 int, מהקורס 'ארגון קבצים'. קישור להורדה נמצא בראש קובץ זה.]
היום נלמד קצת על מדרגי זיכרון: זיכרון פנימי וחיצוני (חומר מקורס של ארגון קבצים).
צריך לעבור לבד את המצגת הראשונה – ההבדלים בין נתונים (הקלט גולמי), מידע (הפלט) וידע (כללים, דבר מה המבוסס על דוגמאות, האלגוריתם לחישוב הפלט על סמך הקלט).

קיימים שני תתי-מדרגים של זיכרון: זיכרון פנימי וזיכרון חיצוני. כל מדרג זיכרון מורכב ממספר רמות וטכנולוגיות. היחידות הגבוהות ביותר שייכות לזיכרון הפנימי, הנמוכות לחיצוני (כל מיני התקנים חיצוניים).

בהיררכיה של הזיכרון הפנימי יש 4 יחידות בסיסיות: סיבית bit, נגיסה nibble, בית byte, מילה word. אלו יחידות בסיסיות כי יש משתנים המכילים חצאים / כפילויות שלהם (לדוגמה short).

סיבית – bit

לוגי	פיזי
Yes \ no	Bit = binary digit
False \ true	בינארית
On \ off	Bit is the atomic unit of computer science

נגיסה - nibble

לוגי	פיזי
תו הקסה-דצימלי (בסיס 16, מ-0 עד F)	רצף של 4 ביטים (4 bits), חצי מבית (half byte)
קבוע (constant) קטן	

בית – byte

לוגי	פיזי
תו char, סמל symbol, היסט offset, קבוע constant	רצף של 2 נגיסות

מילה – word

לוגי	פיזי
שלם / מספר integer	Sequence of 2 \ 4 \ 8 bytes
מען / מצביע address \ pointer	תלוי אם הארכיטקטורה היא 16, 32 או 64 ביטים
שדה / ערך field \ value	
משתנה variable	

קצת עברית: מעקובת – אוסף של דברים ש'עוקבים' זה אחרי זה ברצף.
מילה = 4 בתים = 8 נגיסות = 32 ביטים.

4 יחידות בסיסיות בזיכרון החיצוני: מגזר sector, גוש block, מסילה track, התקן device.
מגזר – sector, היחידה האטומית להעברה בין יחידות הזיכרון השונות (השם התקני למגזר – מקטע)

לוגי	פיזי
רשומה structure \ record	רצף של מילים
שדה קבוצתי group field (קבוצה של תתי-שדות: תאריך המכיל כמה מספרים.. יום חודש ושנה)	
שדה מחזורי periodic field (רשימת קורסים של סטודנט)	

גוש – block

לוגי	פיזי
רשומה לוגית logical record	רצף של מגזרים
מערך / מחרוזת array \ string	יחידת העברת-המידע המינימלית בין הדיסק והזיכרון הפנימי
מכלא buffer (אזור אחסון זמני)	
דף / מסגרת page \ frame	

מסילה – track

לוגי	פיזי
קובץ file	רצף של גושים

התקן – device

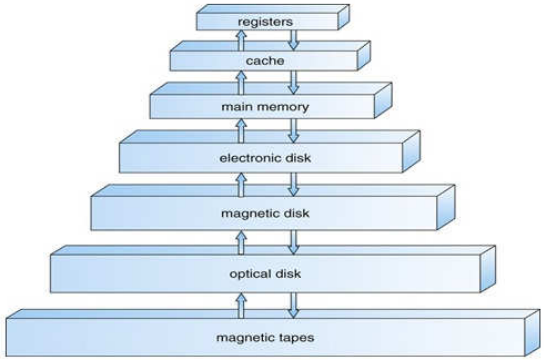
לוגי	פיזי
מסד נתונים database	רצף של מסילות
אוסף של קבצים בעלי הקשר לוגי	
סדרת קבצים בעלי הקשר לוגי	

מודל: התקן = מכיל מספר מסילות, דיסק = מכיל קובץ אחד או יותר. קיימת האנלוגיה disk = file, דוגמאות: ניתן ליצור image לדיסק, בפירמוט כביכול מקבלים קובץ ריק.

[מצגת fo3-1 mem]

<u>ארכיטקטורת מחשב</u>

מערכת המחשב נחלקת ל-3 חלקים: המעבד (data path) אלו החוטים בהם הנתונים עוברים ואילו control הוא מרכיב הבקרה שמעביר את הנתונים), התקנים חיצוניים, הזיכרון (memory).



מדרג הזיכרון נחלק לזיכרון פנימי וחיצוני. מהרמה הנמוכה לגבוהה:
דיסק מגנטי ← זיכרון ראשי ← cache ← רג'יסטר.

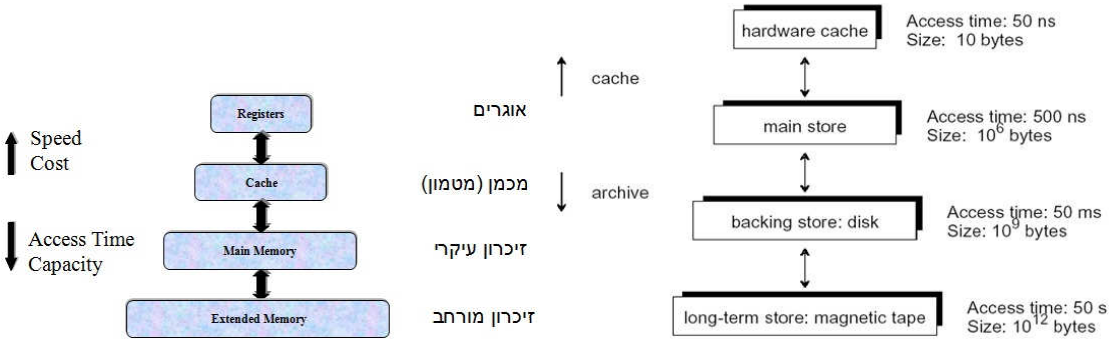
דיסק-און-קי / SSD (solid state disc) / התקן flash הם דוגמאות של דיסק אלקטרוני.

הזיכרון הראשי ("ג'וקים") הוא ה-RAM, random access memory.

<u>דרכים להערכת הטכנולוגיה של סוגי הזיכרון השונים</u>:
(1) זמן גישה (access time) – זמן גישה ליחידת מעינה addressable unit.
זמן גישה הוא ביחס הפוך למהירות ולביצועים. הנחה: ככל שרמת הזיכרון גבוהה יותר, כך זמן הגישה קצר יותר.
(2) קיבולת capacity – נפח גודל הזיכרון של היחידה (סיבית / בית / קילו- KB / מגה- MB / ג'יגה- GB/ טרה- TR).
[משהו שיעזור לזכות ב"מי רוצה להיות מיליונר?": גוגול – מספר עם 100 אפסים.. וגם המקור לשם google]
(3) עלות ליחידת זיכרון cost per unit – מכיל מרכיב של מטבע (דולר) ויחידת גודל (למשל ג'יגה), למשל ₪ ל- MB.
כלל: ככל שהזיכרון מהיר יותר, הוא יקר יותר ובעל קיבולת נמוכה יותר.
בשל המחיר, רק חלק מהזכרון ניקח מהיקר ואת השאר מהזול יותר, כמובן נשלם על כך בביצועים נמוכים יותר.

<u>דוגמאות של זיכרונות (חלק מה- main memory)</u>:
(א) RAM – random access memory, אפשר לגשת לכל מען באופן ישיר ובזמן אחיד. (בביכול היה צריך להיקרא RWM כי ניתן גם לכתוב עליו) זהו זיכרון 'נדיף' (=volatile), אם יופסק אליו זרם החשמל המידע שעליו ייפגע.
(ב) ROM – read only memory, זיכרון שאפשר רק לקרוא ממנו (מבחינת הגישה דומה ל-RAM). עליו יושבת מערכת ההפעלה הבסיסית (bios) ולאחר ה'צריבה' של המידע עליו, הוא נחסם לכתיבה. נקרא זיכרון 'לא נדיף' (= non-volatile), אם יופסק אליו זרם החשמל, הוא לא ייפגע.
(ג) PROM (programmable ROM) – ROM הניתן לתכנות מחדש.
(ד) ROM – E²PROM הניתן לתכנות בצורה אלקטרונית.

דוגמאות למדרגי זיכרון, עם פירוט של גדלים וזמני גישה:



זמנים: ms = מילי-שניה (10^{-3}), ns = ננו-שניה (10^{-9})
הבדל חשוב (הוזכר גם בהרצאה מס' 1): memory – תת-מדרג של הזיכרון הפנימי, storage – הזיכרונות לטווח ארוך יותר.

[בדיחה כספית: cache עולה הרבה cash וגם בעברית: cache=מטמון (שטומנים בו את הכסף..)]

מושגים חשובים: permanency – ההפך מנדיפות, reversibility – האם ניתן לשנות את המידע (ניתן לכתוב עליו).
תקליטורים נקראים לפעמים WORM – write once, read many.

[מצגת fo3-2 mem]
זיכרון פנימי – דרוש למחשב לפעילות שוטפת, פעיל כל עוד המחשב דולק.
זיכרון חיצוני – אמצעי אחסון בעל קיבולת גדולה יחסית.

<u>מדוע צריך את הזיכרון החיצוני ?</u> הרי מהירות הגישה אליו איטית ועלות הייצור (של זכרון פנימי) פוחתת עם הזמן. מספר סיבות:
(1) נדיפות volatility – הזיכרון נמחק כשמכבים את המחשב. אך אפשר להשתמש ב-ROM שהוא זיכרון פנימי לא נדיף, מתאים למערכת הפעלה אך לא ניתן לכתיבה.. הפתרון UPS, מערכת אל-פסק כדוגמת מצברים וגנרטורים. כאשר יש תנודה / הפסקה בזרם החשמל, הם נכנסים לפעולה ומונעים מהזכרון 'להתנדף'. אך גם מערכת כזו יכולה ליפול ולכן זה לא פתרון מוחלט לבעיית הנדיפות. אולי נוסיף UPS משני ל-UPS הראשוני..? לא יעיל. [בדיחה: מי שומר על השומר? ☺]
פתרון טוב יותר: הבזק flash, טכנולוגיית זיכרון פנימי הפיך (משמש גם לכתיבה) ולא נדיף. כל התצורה של המחשבים שלנו שנשמרת בין כיבוי להדלקה, נשמרת על flash. (לא להתבלבל עם ההתקן דיסק-און-קי USB שמחברים מבחוץ, כאן מדובר בזיכרון flash שיהיה חלק פיזי מהמחשב, אנו משתמשים בטכנולוגיה.) החסרון: העלות הכלכלית גבוהה ומהירות הכתיבה איטית.

(2) ניידות mobility – העברת נתונים ממחשב למחשב. ללא זיכרון חיצוני (כדוגמת USB), כיצד נפתור את בעיית הניידות? ניתן ע"י תקשורת נתונים, העברת מידע ברשת.

(3) גיבוי backup – בעיית גיבוי הנתונים שבזיכרון הפנימי. גם בעיה זו נפתור ע"י תקשורת בין מחשבים, נגבה את המחשב שלנו בזיכרון פנימי של מחשב נוסף שזו מטרתו.

(4) שדרוג upgrade – איך נתקין רכיבים חדשים או נחליף רכיב מקולקל ללא כיבוי המחשב? ע"י תוסף חם hot-plug, טכניקה המאפשרת להוסיף / להחליף רכיבים בזמן שהמחשב עובד. (פתרון יעיל יותר מגיבוי המידע על מחשב חלופי ואז הטענתו מחדש למחשב שלנו.)

(5) אבטחה security – מידע המאוחסן בזיכרון פנימי שמחובר לרשת אמנם נגיש יותר (ביחס למידע המאוחסן בזיכרון חיצוני), אך גם פחות ניתן לאבטחה מסיבה זו. פתרון: הצפנה, ע"י שימוש במידור, סיסמאות, אנטי-וירוס.

(6) מרחב מעינה זיכרון סופי finite internal addressing space – גודל הזיכרון הפנימי מוגבל ע"י ארכיטקטורת המחשב. מספר הסיביות (n) במען נתון, קובע את גודל מרחב הזיכרון (2^n). זה בעצם הדבר שמגביל אותנו (לפחות בינתיים) ומחייב אותנו להשתמש בזיכרון חיצוני. משמע, אם ננצל את מלוא המקום הפנוי בזיכרון הפנימי שלנו, הגענו ל'קיר חסום', זו מגבלה מובנית שאין לנו דרך להתגבר עליה. הזיכרון הוא סופי (בעל כמות סופית).

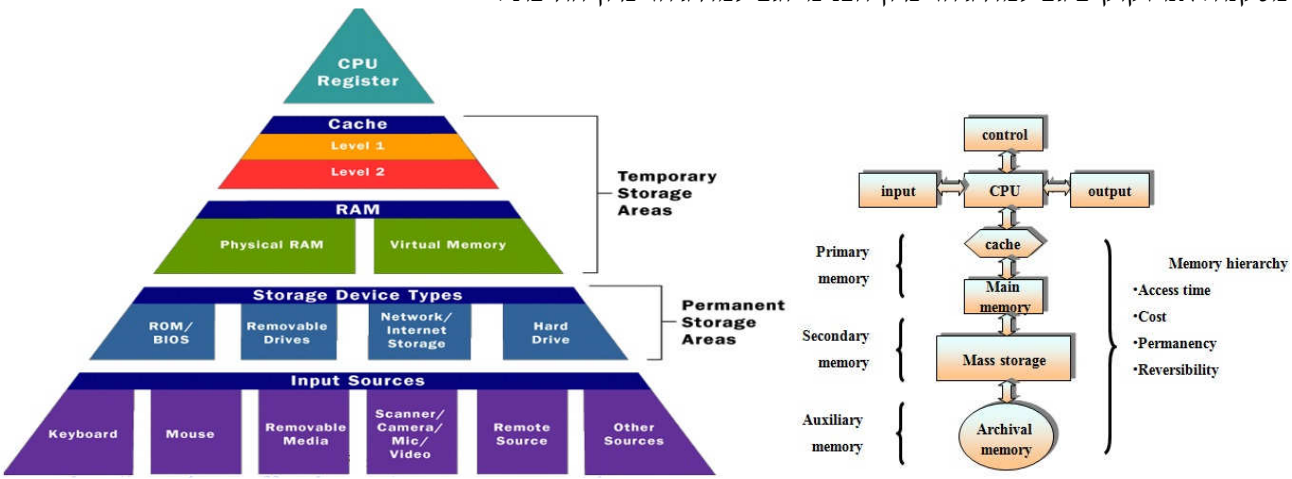
<u>אז אולי אפשר להסתפק רק בזיכרון חיצוני ?</u> לא, ממספר סיבות:

(1) גישה access – כיצד נריץ תוכנית ישירות מהזכרון החיצוני? פתרון: אולי נשתמש באוגרים (רגיסטר, תאים ברמה הגבוהה ביותר של מדרג הזיכרון, מבצעים פעולות חיבור והשוואה בין אוגרים. לאחר ההידור נעשה תרגום לשפת מכונה. בדר"כ יש כ-16, 32 או 64 אוגרים במחשב.) שהם חלק מהמעבד. החסרון: יש מעט מאד אוגרים, לא יוכלו לאחסן תוכנית שלמה.

(2) זמן / גודל גישה access time \ size – זמן הגישה איטי (פי מיליון) יחסית לזיכרון פנימי. הגישה היא לגוש של אלפי סיביות וקשה לגשת לסיבית בודדת. על קושי זה לא נוכל להתגבר.

(3) אנו מניחים כי גם כמות תת-המדרג של הזיכרון החיצוני היא סופית, גם אם היא גדולה משמעותית מהכמות של הזיכרון הפנימי. א-ב-ל(!) אמנם מספר ההתקנים שניתן לחבר למחשב ברגע נתון סופי, אבל ניתן להחליף מדיה כלשהי (CD) ולהחליפה בחדשה (החלפה ממצב מקוון בו ה-CD בתוך הכונן במחשב למצב לא-מקוון בו ה-CD נמצא מחוץ לכונן). היות ויש מדיה 'שליפה', נוכל בעצם ליהנות מאספקה אינסופית (ואולי יום אחד גם נוכל להשתלט על העולם, נכון פינקי? ☺).

מסקנה: אנו זקוקים גם למדרג הזיכרון הפנימי וגם למדרג הזיכרון החיצוני.



[מצגת fo3-3 mem]

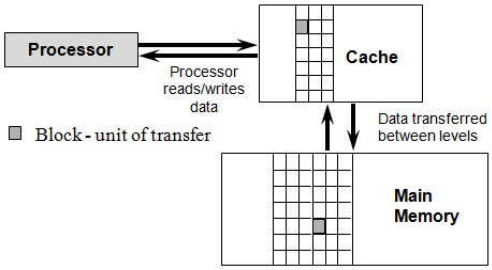
מטרת היררכית הזיכרון: נתינת אשליה של זיכרון בעל קיבולת גדולה ובלתי מוגבלת (מהרמות הנמוכות) אך מהירה (ברמות הגבוהות) וזולה (ברמות הנמוכות). מערכת הקבצים דואגת שהנתונים להם אנו זקוקים לתוכניות שלנו יהיה במיקום גבוה במדרג.

<u>עקרון המקומיות locality principle</u>

בכל פלח זמן, התוכנית ניגשת לחלק קטן יחסית ממרחב המיעון שלה. (לכל החננות, זה מזכיר את splay tree ממבני נתונים ☺)

(1) מקומיות במרחב spatial locality – סבירות גבוהה שפריטים שמעניהם קרובים, יהיה בשימוש בזמן הקרוב. למשל מערך או סידרה של פקודות.

(2) מקומיות בזמן temporal locality – אם השתמשנו בפריט מסוים, סביר להניח שנשתמש בו שוב בזמן הקרוב.



אם נדאג שחלק מהנתונים יהיו ב-cache, זמן הגישה אליהם יהיה מהיר. לכן, נדאג שהנתונים להם אנו זקוקים בתדירות גבוהה, יהיו זמינים לנו בקלות.

<u>אינטראקציות בין רמות סמוכות במדרג הזיכרון</u>:

פגיעה hit: הפריט שהמעבד ביקש נמצא ברמה הגבוהה.

החטאה miss: הפריט אינו נמצא ברמה הגבוהה.

יחס הפגיעה hit ratio: יחס בין כלל הפגיעות למספר הגישות.

יחס החטאה miss ratio : (1 – hit ratio) היחס בין כלל ההחטאות למספר הגישות.

<u>שתי גישות להיררכית זיכרון</u>:

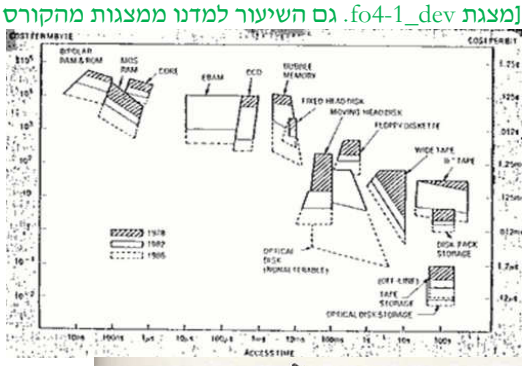
(א) מדרג זיכרון כולל של כל רמות האחסון overall memory hierarchy.

(ב) צירוף של מדרג זיכרון פנימי וחיצוני two sub-hierarchies – הפיצול לשני מדרגים נובע מהפער (gap) הגדול (סדר גודל של פי 10^6) במהירויות הגישה למידע בין המדרגים, עם השלכות רבות.

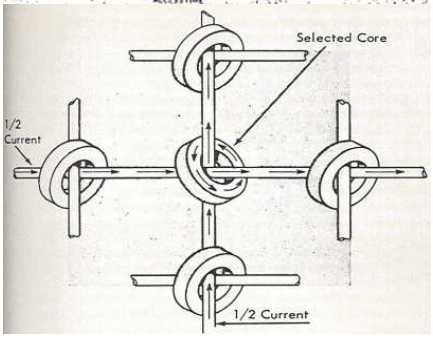
<u>חוק מור (moore's law)</u>: כל 18 חודשים, המהירות והקיבולת של הזיכרונות (הפנימיים) מכפילים את עצמם. יש גם גירסה אחרת של החוק, לפיה כל 4 שנים המהירות מוכפלת פי 3. בהתאם, יש אפקט הפוך של הוזלת רכיבי הזיכרון.

הרצאה מס' 3 - 2.11.09

[מצגת fo4-1_dev. גם השיעור למדנו ממצגות מהקורס 'ארגון קבצים'. בשיעור זה ניתן סקירה של פיתוחים לאורך הזמן.]



גרף של <u>היררכית טכנולוגיות זכרון</u>: ציר x מייצג זמן גישה (access time), ככל שנמוך כך מהירות הגישה גבוהה יותר. בצד שמאל מעלה, זו קבוצת הזכרון הפנימי המהיר. בצד ימין למטה, זו קבוצת הזכרונות החיצוניים. פחות או יותר באמצע, זו קבוצת הזכרונות של הדיסקים האלקטרונים. (טכנולוגיית SSD, מוליכים למחצה. בעבר נקראו SCD, שבאנגלית ראשי התיבות הם מוליכים למחצה.) אם נעביר קו אלכסוני משמאל-מעלה לימין-מטה, נראה שטכנולוגיה זו פחות אטרקטיבית, מבחינת היחס של עלות לעומת ביצועים. [אין לכם בעיית ראיה, גם במצגת הכתב לא ברור]



<u>זכרון פנימי</u>: סוג הזכרון הראשון שהיה, לפני המוליכים למחצה, נקרא core memory. היה מורכב מאוסף טבעות מגנטיות, שכל אחת מקוטבת עם / נגד כיוון השעון. בין הטבעות עברו חוטים אופקיים ואנכיים. מיקום טבעת יוצג במרחב כקואורדינאטות של (x,y). ע"י השראה חשמלית מתאימה, ניתן היה להפוך את הקיטוב של הטבעת. ע"י אוסף טבעות ממוגנטות, ניתן היה ליצור מערכים דו-מימדיים ואף תלת-מימדיים. היתרון של ה-core המגנטי, הוא בכך שגם כאשר הפסיקו את זרם החשמל, הזכרון לא היה נדיף(!). Core dump – כאשר התוכנית קורסת, יש 'הטלה' של תוכן זכרון הליבה.

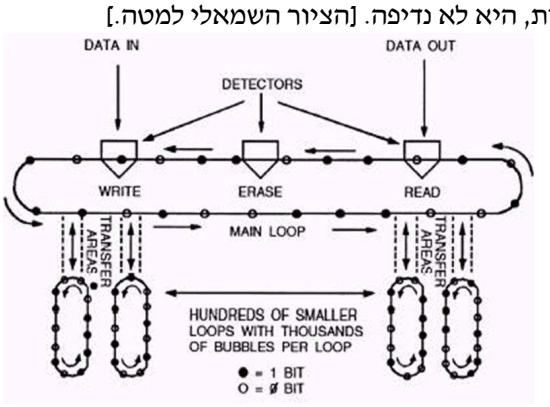
לאחר-מכן היו שבבי טרנס-פיוטר, שילוב של העברה ומחשב. [ואיך אפשר בלי הערה של אריאל פרנק לגבי עברית – 'מח-שבב', שילוב של מחשב ושבב . איפה מיטל שתגיד: "מי חננה של אמא?" ☺] בהמשך פיתחו שבבי זכרונות של ROM או RAM, שזה מה שאנחנו מכירים היום.

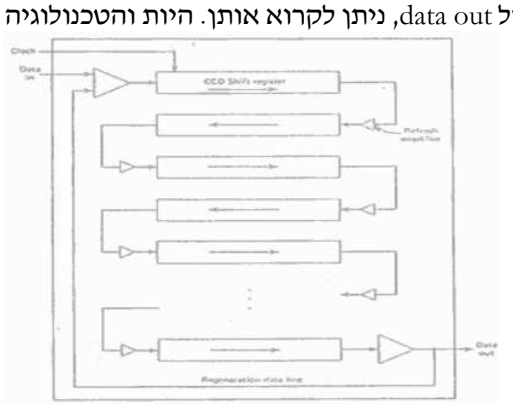
<u>דיסקים אלקטרונים</u>:

(1) EBAM – electronically beamed addressable memory. פועל ע"י קרני אור, קרן אלקטרונית מוכוונת מיעון.

(2) CCD – charge coupled device. הכיל סידרה של אוגרים (ניתן להתייחס אליהם בתור קבלים המכילים ערך של 0 או 1). ביניהם היו מגברים להעצמת האות החשמלי, שכולם היו מסונכרנים ע"י מקור משותף. את סיבית הפלט ניתן היה לקרוא וניתן גם היה להחזיר ולהעבירה שוב בתור מקור. על כל אוגר יש רעיון של shift, הסטה של מקומות ימינה (למשל שני מקומות ואז נכנסות שתי סיביות מצד שמאל ובעצם מחלקים את כל המספר פי 4). הסיביות יצאו לטיול ☺ על גבי המסלול שעובר בין האוגרים. זו טכנולוגיה איטית יותר, כי היא סדרתית, לא ניתן לגשת באופן ישיר לערך של הסיבית האמצעית לאורך מסלול-הטיול. לעומת זאת, RAM בעל זמן גישה אחיד: ניתן לגשת לכל סיבית / מילה באותו זמן. לכן בגרף למעלה, ה-CCD ממוקם באזור המילי-שניות זמן גישה (ולא באזור של מהירות גבוהה יותר). [הציור הימני למטה.]

(3) MBM – magnetic bubble memory. היחיד שלא נדיף. מורכב ממישורים שעליהם יש בליטות מגנטיות בצורת לולאות (בועות), המחוברות ע"י פסים צרים. כל מישור מייצג 0 וכל בועה מייצגת 1. כל לוח מורכב ממאות / אלפי בועות. יש גם לולאה מרכזית, main loop, גדולה יותר משאר הלולאות. יש תאים דרכם ניתן למחוק את המידע, להוציא מידע (data out), לכתוב מידע (data in). הסיביות יוצאות לטיול ☺ (שוב, גישה סדרתית) על גבי הבועות ורק כאשר הן מגיעות לאזור של data out, ניתן לקרוא אותן. היות והטכנולוגיה מגנטית, היא לא נדיפה. [הציור השמאלי למטה.]

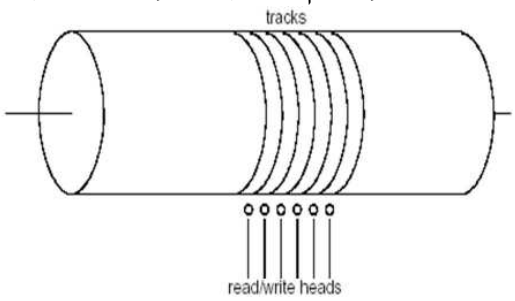




[בדיחה בינארית – איש אחד מספר לחברו שאישתו מדברת איתו בינארית: "אפס אחד" ☺]

<u>זכרון חיצוני</u>:
בתחילה היו סרטים מגנטים, בהמשך 'תופים' (drums) שהיו ממוגנטים מסביב לגליל ובסיום דיסקים.

(1) מבנה 'תוף': גליל שעליו מסילות (track) ממוגנטות. מול כל מסילה היה ראש קורא-כותב. בעצם הייתה גישה לכל מען, לכל סיבית. זו כבר לא גישה סדרתית(!). אך היחס בין פני השטח של הגליל לנפחו היה נמוך. טכנולוגיה זו הייתה יקרה וכמות המידע שניתן היה לרשום על פני השטח של הגליל הייתה נמוכה מדי. [הציור הימני למטה.]



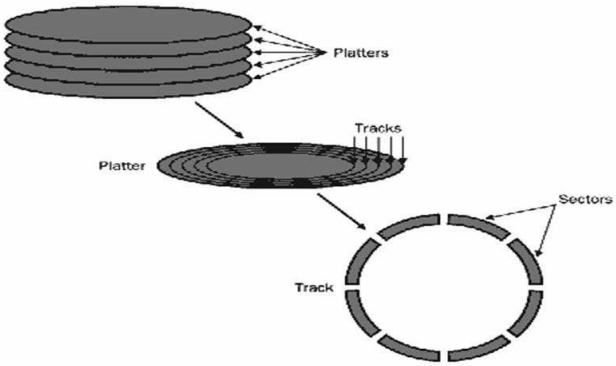
(2) סרטים מגנטיים – טכנולוגיה ליניארית, אין גישה ישירה. נחלקים למספר סוגים: (א) גלילי סרטים (כמו בקולנוע) tape reels בהם הקידוד היה על הסרט. בעיות: הרישום לא היה על תחילת הסרט כי שם נגעו הידיים של המפעיל שיכלו לפגוע במידע. יש גליל-ספק ויש גליל אחר שמקבל ממנו את המידע. שפורפרות ריק (בלי אוויר) שימשו לסיבוב הגלילים בצורה מהירה, ללא חיכוך. כך היה מעבר של הסרט בין שני הגלילים באופן מהיר. בחיבור על הסרט בין הגלילים היה ראש כותב וראש קורא. כל הסרטים שלא היו באותו זמן בתוך הכונן (היו בארון), נחשבו offline. היו מערכות רובוטיות ממוספרות שסייעו לארכב את כל הסרטים שלא בשימוש באותו רגע. הסרטים היו גדולים ומסורבלים. (ב) קסטות – מגמה של הקטנת המדיה. המנגנון פנימי, הסרט עצמו נמצא בתוך קופסת הפלסטיק. ע"י זיזים שנכנסים לתוך הגלגלים, ניתן לסובב את הגלגלים ולקרוא את המידע שעל הסרטים. כך גם הסרטים פחות פגיעים. (ג) קלטות מהירות – עד היום יש בהם שימוש (למשל בווקמנים, למי שעדיין יש ☺).

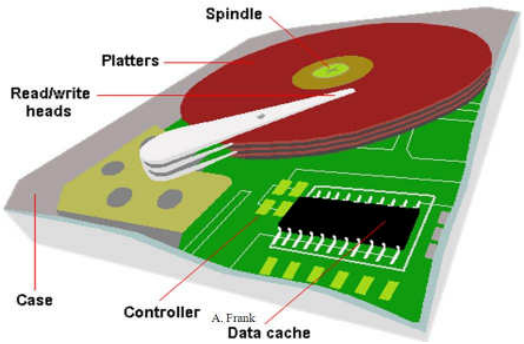
(3) דיסקים מגנטיים – זוכרים את הדיסקטים של פעם? (בעברית – תקליטון, floppy disk). היו בגדלים של 8, 5.25 ו-3.5 אינץ'. בהמשך התקדמנו לדיסקים. הם נחלקים לבעלי ראש קבוע ובעלי ראש נע. היה כמובן הבדל בקיבולת בין הדיסקטים (1.44 מגה-בייט) והדיסקים (700 מגה-בייט). הדיסקט גמיש ולכן שביר ואילו הדיסק חזק יותר ולא גמיש. הראש הקורא לא יכול לגעת בדיסק ממש, לכן יש שימוש בראשי קריאה-כתיבה 'מרחפים' העושים שימוש בהשראה של שדה מגנטי. אם יהיה מגע בין הראש הקורא-כותב והדיסק, הדיסק ייפגע.

בהתפתחות הטכנולוגיות של הדיסקים הקשיחים, הגידול בקיבולת נעשה כמובן בהתאם לחוק מור.

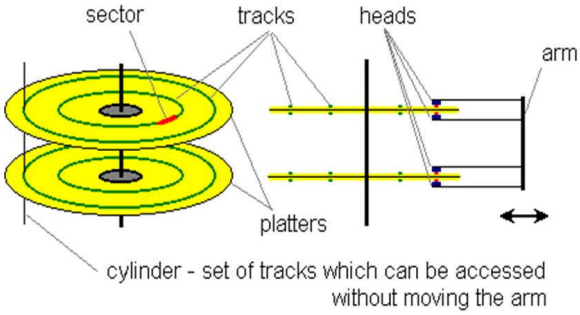
[מצגת fo4-2 dev]

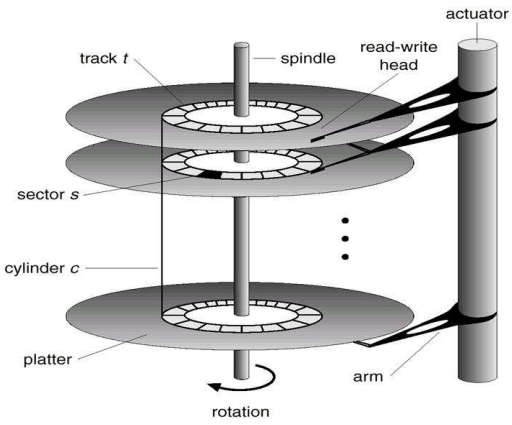
מארז דיסק (disk pack) – מספר תקליטים משותפי-מרכז המסתובבים כיחידה אחת. נעשה שימוש במספר תקליטים (דיסקים) הממוגנטים משני הצדדים, המחוברים ביניהם ע"י ציר מרכזי. הדיסק מסתובב והזרוע יודעת לנוע על גביהם פנימה והחוצה. הזרוע משמשת כראש קורא-כותב. מספר הזרועות הוא כמספר התקליטים מהם מורכב הדיסק. לכל תקליט יש משטח עליון ותחתון. כל תקליט נקרא platter (יש לו שני משטחים surfaces), הציר המרכזי שמסובב את התקליטים נקרא spindle. בקצה הזרוע נמצא הראש הקורא-כותב. בנוסף יש גם שבב שמהווה את הבקר של מארז הדיסק. כל דיסק מורכב ממספר מסילות (tracks), שכל אחת ממנה נחלקת למספר מגזרים (sector). מספר מסילות (מקבילות) ממספר דיסקים ביחד נקראים גליל cylinder.





שאלה: מה עדיף, שבכל אחד מהדיסקים נמלא מידע לפי גלילים (למשל את מסילה 7 בכל דיסק), או לפי מגזר, שתחילה נמלא משטח מסוים על כל הדיסקים. הזרוע נעה תחילה לפי גליל, מתייצבת על מסילה. לכן, עדיף לפי גליל. הזזת הזרוע היא פעולה אלקטרו-מכנית, פעולה יקרה יחסית. לאחר שהזרוע מתייצבת והמסילה מסתובבת מתחת לראש הקורא-כותב, לא צריך להזיז עוד את הזרוע וזו פעולה מהירה יותר.





בעיה טכנית: כדי שכל הזרועות יכתבו / יקראו במקביל, צריך להעביר מספר זרמים במקביל. יש טכנולוגיות מתקדמות בהן יש שתי זרועות (ראשיות), כך שסה"כ אמנם זורמים שני זרמים ולא רק אחד, אך כל זרוע (ראשית) שומרת למעשה את המידע באזור אחר, כך שזה לא לגמרי מקבילי. לכל זרוע יש שני ראשים קוראים-כותבים, אחד עבור המשטח התחתון של הדיסק העליון ואחד נוסף עבור המשטח העליון של הדיסק התחתון.

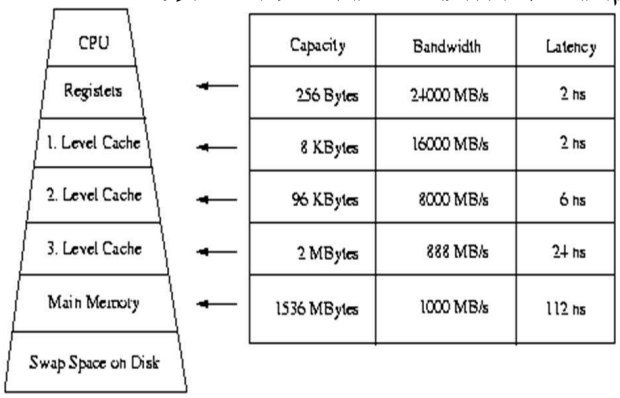
מארז הדיסקים מסתובב כל הזמן(!). יש טכנולוגיות בהן אותה כמות של מידע נכתבת על כל מסילה (מה שבעצם גורם לכך שצפיפות המידע ביחס לגודל המסילה משתנה, שהרי המסילות הפנימיות קטנות יותר מהחיצוניות). מארז הדיסקים סגור ואטום לחדירת אבק שיכולה לפגוע במידע ובביצועים. המרחק בין קצה הראש הקורא-כותב מהדיסק עצמו הוא בערך 3-4 מיקרון (לצורך השוואה, עובי שערה של אדם כ-10 מיקרון).

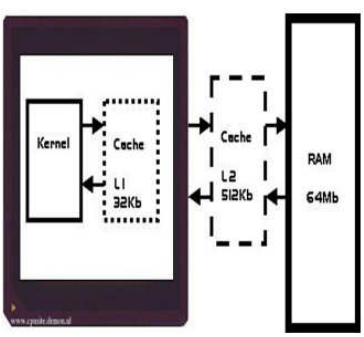
ישנה חלוקה בהתאם לסוג הראש: (א) דיסק עם ראש קבוע (fixed-head disk) – ראש קורא-כותב אחד לכל מסילה בכל גליל בדיסק. היתרון הוא בזמן הגישה המהיר. החסרון הוא בכך שאנו זקוקים למספר ראשים (אחד לכל מסילה), מה שגורם לעלייה במחיר ומקטין את מספר המסילות. (ב) דיסק עם ראש נע (moving-head disk) – זרוע ראשית שמזיזה את כל הזרועות, כל זרוע נוגעת במשטח של דיסק.
בהמשך פיתחו טכנולוגיה של דיסק שליף (removable disk) – כך שהיה ניתן לשלוף את מארז הדיסק מהכונן ולהחליפו באחר. היום השתכללו וכל הכונן שליף ☺ ואפילו יש לנו כוננים חיצוניים.
טכנולוגיית ראש קורא-כותב – יחידה אלקטרו-מגנטית. במהלך הכתיבה, כיוון הזרם גורם למגנוט המשטח. במהלך הקריאה, המגנוט גורם להיווצרות זרמים בקוטביות משתנה.

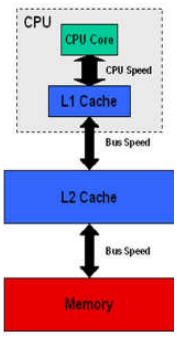
[מצגת fo5-1 his]
עד עכשיו דיברנו על ההיבט הסטטי של הזכרון: התקני קלט-פלט ודיסקים. כעת נסתכל על מדרג הזכרון באופן דינאמי / אקטיבי: התקנים על פס מערכת, בקרי התקנים, רוחב פס להעברת הנתונים. יש הבדל בין זכרון פנימי וזכרון עיקרי: הזכרון העיקרי הוא חלק מהזכרון הפנימי, בעל היכולת הגבוהה ביותר. ישנם מספר בקרים, כל אחד עבור התקן קלט / פלט: מתאם גרפי, בקרי USB, בקר דיסק. נתייחס גם למעבד כסוג של בקר, כי הוא מפעיל את האוגרים.

<u>פס המערכת (system bus)</u>
מחבר בין המעבד וההתקנים. מורכב מ-3 סוגי פסים: פס לנתונים, פס להעברת מען הקלט-פלט, פס שדרכו מועברות הוראות כגון כתיבה או קריאה. [שאלה אריאלית:] אז איפה נמצא ה-cache?? כיום נמצא על שבב המעבד. מדוע שה-cache לא יהיה התקן בעל בקר משלו? הסבר: ה-cache נקרא בעברית 'מכמן', כי הוא מוסתר. רמת המכמן לא הייתה קיימת במקור. עם הזמן נוצר צורך שתהיה טכנולוגיה בין המעבד והזכרון העיקרי. המודל הראשוני של מבנה מחשב עדיין תופס, כדי שתהיה 'תאימות אחורה'. מסיבה זו הפתרון הוא לשים את ה-cache בצורה נסתרת (בציור) כ- memory-I\O bus (השם מעיד על כך שהוא מקשר בין הזכרון הפנימי והזכרון החיצוני).
ה-cache הוא סף למדרג הזכרון. כשרוצים להעביר נתונים למעבד, בעצם מעבירים את הנתונים לאוגרים (הרגיסטרים), המשמשים לאחסון הנתונים של המעבד. מדוע לא מראים את האוגרים בצורה מפורשת בציורים? כי אנחנו אמורים לדעת את זה לבד ☺ שהם נמצאים שם, באופן טריוויאלי. מבחינת מדרג הזכרון, ה-cache יושב בין הזכרון העיקרי והמעבד: כאשר מועבר מידע דרך פס המערכת למעבד מהזכרון העיקרי, הוא נשאר גם ב-cache לצורך העברה חוזרת, מבלי להתייעץ עם בקר הזכרון. זאת כי ה-cache נמצא ברמה גבוהה יותר מבחינת מדרג הזכרון.

<u>ה-cache נחלק ל-3 רמות</u>: L1,L2,L3. הרמות האלו מייצגות את מיקום ומיצוב המטמון (cache) מבחינת קיבולת ומהירות הגישה, במדרג הזכרון. L1 יושבת על ליבת המעבד ולכן הכי מהירה (האוגרים שיושבים גם על המעבד מהירים יותר). הרמה הבאה L2 יושבת על אותו כרטיס שבו יושב המעבד, מרוחקת יותר. לכן מבחינת מדרג הזכרון היא נמוכה יותר, הגישה אליה איטית יותר, אך הקיבולת גבוהה יותר. העלות ליחידה שלה מעט יותר נמוכה משל L1. לפעמים גם יש רמת L3. אם היינו ביל גייטס, יכולנו להרכיב לעצמנו מחשב עם cache גדול במקום עם זכרון RAM גדול ☺ אך יש גם מגבלות ארכיטקטוניות שגורמות לייקור המחיר של הטכנולוגיה הזו. מספר האוגרים הוא כמה עשרות / מאות. לכל רמה של cache יש הגבלה מבחינת קיבולת / עלויות ולכן יש מספר רמות. [למטה, מצד שמאל] ייצוג האוגרים בתרשים בצורה מפורשת ופירוט סדרי גודל של האוגרים והרמות השונות של ה-cache. העמודה של רוחב הפס מייצגת את זמני / מהירויות הגישה. השכבה התחתונה בפירמידה מייצגת את הזכרון החיצוני. כביכול המעבד הוא הבקר ולא טכנולוגיה כלשהי במדרג הזכרון ולכן מיקומו בפירמידה לא כ"כ מתאים למצב בפועל.





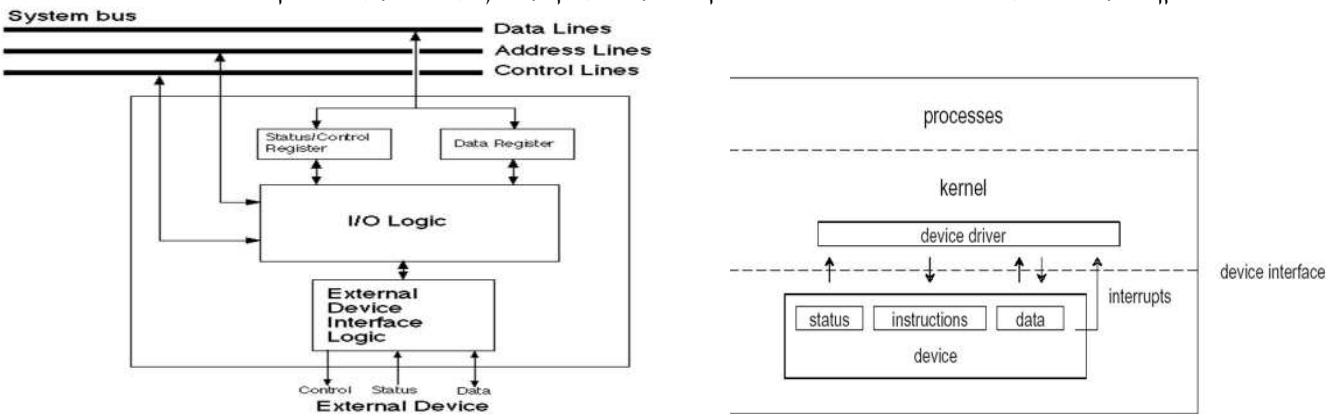


ALU (arithmetic logic unit) – יחידה ביצועית חשובה, הקיימת בכל יע"מ / CPU (יחידת עיבוד מרכזית). רוב פעולות המעבד מתבצעות ע"י ALU אחד או יותר. ALU טוען נתונים מאוגרי הכניסה ותוצאת החישוב נאגרת באוגר הפלט. ניתן להעביר מידע בין האוגרים לעצמם ובינם לזכרון.

<u>התפתחות פונקציונלית של מנשק מעבד-התקנים</u>:
שלושה היבטים: מבנה ההתקן, מנשק קלט/פלט, איתור סיום פעולת קלט/פלט. נסקור את ההיבטים במקביל.

(1) התפתחויות במבנה ההתקנים: התקן 'טיפש' ← התקן מבוקר ← בקר התקנים ← מחשב קלט/פלט (ערוץ) ← בקר חכם.
(א) ההתקן הטיפש dumb device, היה מורכב מחומרה בלבד, יחידה פסיבית נטולת מעבד / מרכיב חישובי. [באיור למטה בצד ימין] היות ולא היה ניתן לתכנת את היחידה, עבר זמן רב בין עדכוני חומרה. Device driver – נציג של ההתקן ואחראי על ניהול הפעלתו, מודול תוכנה, מייצג את תת-מדרג הזכרון. כל התקן בנוי אחרת ולכן דרוש לו driver שונה. [בדיחה משוגעת: לכל אדם יש את השגעון שלו ולכל שגעון יש את האדם שלו ☺] הדרייבר נמצא ב-kernel, כי מהווה חלק מהתוכנה. ההתקן הפשוט מכיל אוגרים מסוגים שונים: מצב (פעיל / סיימתי הוראת קלט-פלט / שגיאה), הוראות (דרכו מזרימים את ההוראות להתקן), נתונים (דרכו העבירו את הנתונים הלוך-חזור מהזכרון העיקרי להתקן ובחזרה). (ב) התקן מבוקר controlled device – ליחידת ההתקן (drive) יש גם בקר (משני, לעומת ה-CPU) שאחראי לו. ביחד, ההתקן והבקר מהווים את ה-device. סוג של מחשב זעיר שניתן לתכנת אותו. היתרון בעבודה מהירה יותר ויכולת לשנות את התנהגות ההתקן בהתאם לצורך, מבלי לתכנת מחדש את החומרה. (ג) בקר התקנים control unit for several devices, עבור מספר יחידות התקנים דומות. בעיקר כאשר מדובר בסביבה הומוגנית, למשל בקר של דיסקים המטפל במספר דיסקים. לבקר התקנים יש פקודות (orders) ברמה גבוהה לעומת ההוראות (instructions) ברמה הנמוכה שיש לבקר הטיפש, הוא מספיק מתוחכם כדי לדעת להתמודד איתן.

(2) התפתחות מנשקי קלט/פלט: תכנות קלט/פלט ← מיפוי אוגרים למרחב המעינה של הזכרון ← תוכנית ערוץ לגישה ישירה.
(א) תכנות קלט/פלט I\O programming – לכל התקן / בקר יש 3 סוגי אוגרים: מצב, הוראות, נתונים. עשו שימוש בפקודות מכונה מפורשות (load register \ store register). כיצד העבירו את ההוראות והמצב? ע"י פקודות מכונה מפורשות כגון in \ out. פקודת קלט / פלט צריכה להתייחס לשם ההתקן, לאוגר ההתקן והמען (שם המשתנה באוגר) שאליו מעבירים. [באיור למטה בצד שמאל] ניתן לראות את פס המערכות שנחלק ל-3 סוגים. במרכז נמצא הבקר של ההתקן ומעליו האוגרים שלו. למטה נמצא המנשק למצע האחסון עצמו, דרכו מועברות הפקודות.



הרצאה מס' 4 - 9.11.09

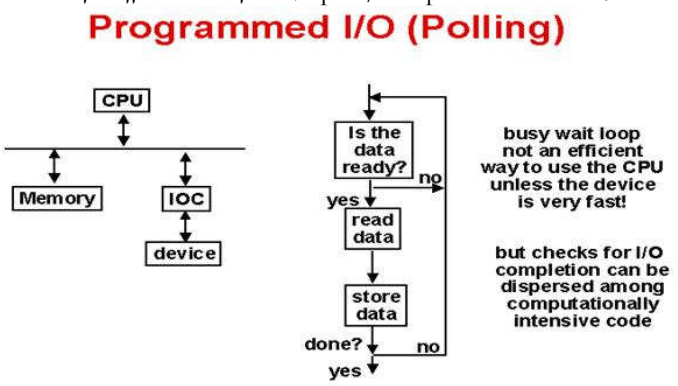
[מצגת fo5-1 his]
חזרה קצרה על שיעור קודם:
עסקנו במנשק מעבד-התקנים (CPU-devices) ובמקביל ערכנו סקירה היסטורית-תפקודית של הפיתוחים בתחום.

כל ההתקנים יושבים על פס המערכת, לכל התקן יש בקר שמבקר את פעולתו. גם המעבד הוא בקר, המבקר את האוגרים (הרמה העליונה של יחידת הזכרון). ההיבטים שהזכרנו: מבנה ההתקן, מנשק קלט-פלט, איתור סיום פעולת קלט-פלט. בתחילה היה בקר 'טיפש' שלא ניתן היה לשנות את התוכנה שלו, שהיה בעל 3 אוגרים שסימנו את מצבו, ההוראות והנתונים. בהמשך חיברו את הבקר והיחידה הפועלת ליחידה אחת: התקן. בהמשך היה בקר התקנים, בקר שהפעיל מספר יחידות התקנים דומות. ה-device driver הזה כבר היה חלק מה-kernel (מערכת הזכרון), תוכנה שיושבת על המעבד עצמו. בקר הוא סוג של מעבד, כי צריך להיות מסוגל להניע את ההתקנים אותם הוא מנהל.

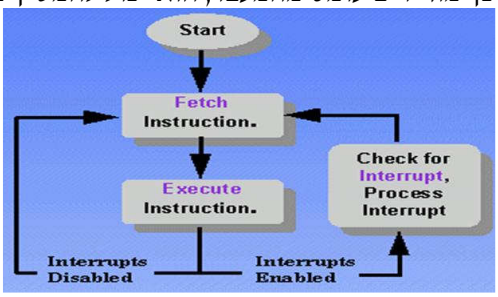
התפתחות מנשקי קלט-פלט:
בתחילה היה תכנות קלט-פלט. כל התקן / בקר הכיל 3 אוגרים: מצב, הוראות, נתונים. צריך לקיים הוראות מכונה שידעו להפעיל את הקלט-פלט. ראינו מספר דוגמאות, כגון in DEV1, status, destination – שם ההתקן שרוצים לפנות אליו, שם האוגר הספציפי שרוצים לפנות אליו בהתקן ובנוסף שם של מען / משתנה אליו / ממנו רוצים לשמור / לקרוא ערכים.

[מצגת fo5-2 his]
עד כאן חזרה על שיעור קודם, נמשיך לסקור התפתחות מנשקי קלט-פלט:
כיצד נאתר סיום פעולת קלט-פלט? ע"י תשאול, polling. המעבד נכנס ללולאה לקריאת אוגר המצב של ההתקן, עד שהסיבית המודיעה על גמר הפעולה נדלקת. א-ב-ל, כל עוד הסיבית לא נדלקה, זה בעצם בזבוז זמן של המעבד, לכן שיטה זו נקראת busy-wait, כביכול המעבד עסוק בהמתנה. החסרון הוא כמובן שבזמן שהמעבד רץ בלולאה ומחכה שהסיבית תידלק, הוא לא מבצע פעולות אחרות. צריך לזכור שמהירות ההתקן נמוכה בהרבה יחסית למהירות המעבד.

IOC (= input / output controller) הבקר של התקן הקלט-פלט.

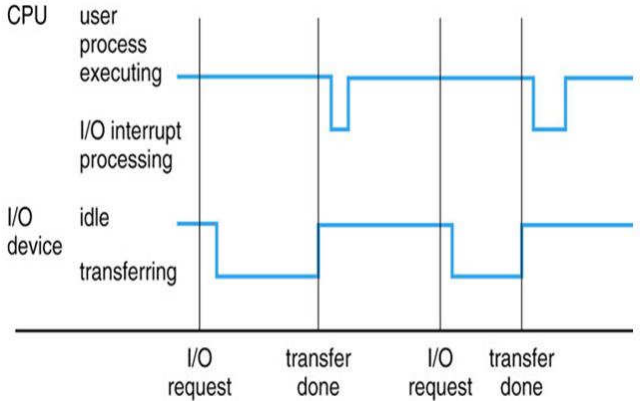


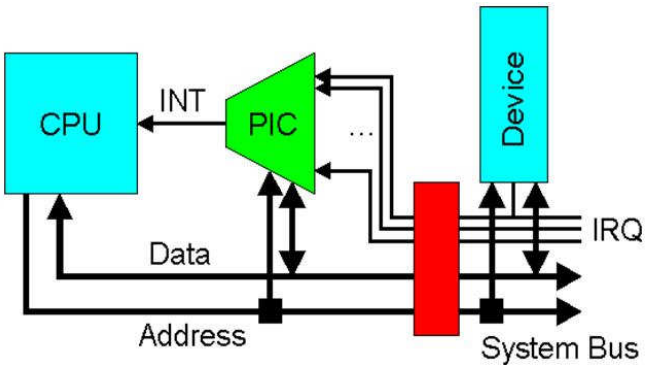
Interrupt – 'פסיקה', סוג של הודעה. במקום כל הזמן לשאול את ההתקן אם סיים לקלוט, נשתמש ב'פסיקה' – כאשר ההתקן יסיים הוא יודיע לנו שהוא סיים. 'פסיקה' מהווה הפרעה לפעילות התקינה, מושכת תשומת לב. ההתקן הוא יוזם הפסיקה.
השיטה הבאה: פסיקות קלט-פלט I/O interrupts. בסיום פעולת הקלט-פלט, ההתקן שולח פסיקה (אות אלקטרוני) למעבד. כך מורידים עומס מהמעבד, הוא יכול להמשיך בפעולות אחרות עד שתתקבל פסיקה מההתקן.



הפסיקות לא מתבצעות רק בין פקודות שמבצע המעבד, אלא גם תוך כדי. ניתן לחסום פסיקות מלהתפרץ במהלך ביצוע פקודה ואז רק כאשר המעבד סיים לבצע פקודה, הוא יבדוק אם יש פסיקה שמחכה להתייחסות שלו. [אנלוגיה: אדון מבקש כוס תה מהמשרת שלו ובינתיים ממשיך לקרוא עיתון ☺]

הרוטינה של מערכת ההפעלה שמטפלת בפסיקות נקראת interrupt handler. כל פסיקה צריכה להיות מזוהה ע"י מזהה, כדי שנוכל לדעת איזה התקן שלח אותה.

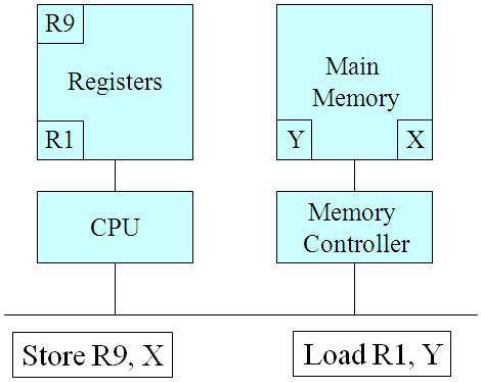




למעלה, ציר x מייצג את הזמן. ניתן לראות שבעקבות הפעלת התקן קלט-פלט יש הפסקת עבודה של המעבד לצורך טיפול בשליחת ההוראות להתקן. עד סיום פעולת ההתקן, המעבד מתפנה להמשיך לבצע את פקודות המשתמש. לאחר סיום פעולת הקלט-פלט, ההתקן שולח פסיקה וכאשר המעבד סיים לבצע את הפקודה הנוכחית של המשתמש, הוא מתפנה לטפל בפסיקה.

באיור למעלה: IRQ – interrupt request. כל ההתקנים שולחים את הפסיקות על פס המערכת, שמגיעות ל-PIC שאחראי על העברתן למעבד.

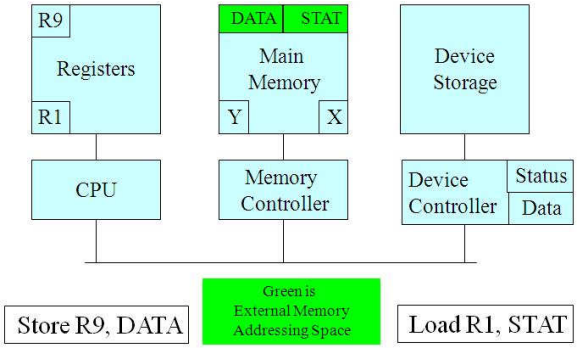
ריבוי התקנים ופעולות קלט-פלט, גורם לעומס של פסיקות, I/O interrupts load. זה נכון בעיקר כאשר ההתקן קולט תו אחד בכל פעם (character-oriented devices).



בחזרה למנשקי קלט-פלט, מנשק מעבד-זכרון: השיטה הבאה היא memory mapped I/O – מיפוי אוגרי קלט-פלט של בקרי ההתקנים לחלק קטן של מרחב המעינה של הזכרון העיקרי, שאינו בשימוש. כך נוכל לבטל את הצורך בפקודות in/out. פקודות המכונה המאפשרות העברת מידע בין האוגרים של המעבד ובין הזכרון העיקרי, ניתנות להתייחסות ע"י הפקודות store / load (אחסון / קריאה). אוסף האוגרים נקרא register file. דוגמה: store R9, X - לקחת את התוכן הכתוב באוגר R9 ולהעתיק אותו למשתנה X. בפס המערכת, ישנו קו מיוחד שתפקידו לאותת על כיוון הקלט-פלט: מההתקן אל הזכרון הפנימי או להפך.

פס המערכת מסונכרן ע"י שעון. בכל מחזור פס מועבר מען / נתון על הפס. פקודה של store / load, דורשת צמד מחזורי פס: (1) מחזור מען address cycle – שליחת המען ע"י המעבד, אליו נתייחס במחזור הפס הבא. בפקודה למעלה – X. (2) מחזור נתונים data cycle – שליחת הנתון עצמו ע"י המעבד / בקר. בפקודה למעלה – R9. בעמוד הקודם ניתן לראות הצגה גרפית של הפקודות. דגש קטן: המעבד אחראי על האוגרים, בעוד בקר הזכרון אחראי על הזכרון הראשי.

בנוסף, יש גם מנשק מעבד-זכרון חיצוני, CPU – external-memory interface: יש מיפוי חח"ע (חד-חד-ערכי) בין כל אוגר-בקר של התקן מסוים למען מסוים בזכרון הראשי. הפקודות store / load יודעות לעבוד עם המיעון בזכרון הפנימי.



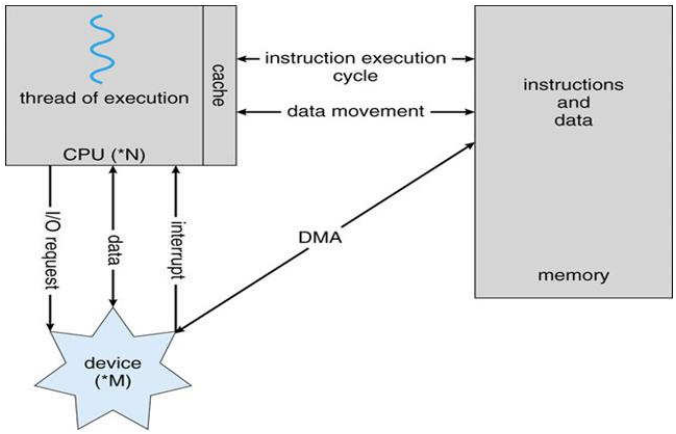
ע"י מיפוי האוגרים לזכרון, ניתן להשתמש בפקודות אלו גם ישירות מול המיעון שהוקדש עבור הזכרון החיצוני. הפקודות משתנות מעט, ניתן שמות של אוגרים של בקרי ההתקן, במקום שמות של מענים בזכרון הראשי, לדוגמה: load R1, STAT – טעינת התוכן של האוגר R1 לתוך אוגר-הבקר STAT. גם במקרה זה, הפקודה מתבצעת במשך צמד מחזורי פס. שמות המשתנים בפקודות אלו יהיו DATA או STAT, כשמות האוגרים בבקר ההתקן.

מרחב מיעון שאין עבורו שבבים פיזיים, לא ניתן לתקשר איתו. אנו מנצלים רק חלק ממרחב הזכרון כולל. לכן, ניתן 'לחתוך' חלק מהזכרון הראשי ולהתייחס אליו בתוך זכרון חיצוני. היות והגדרנו מרחב מיעון לבקר ההתקן, פקודות כמו אלו המופיעות מעלה (באיור), לא 'יעירו' את בקר הזכרון (למרות שהמידע עצמו נמצא בזכרון הראשי), אלא בקר ההתקן יפעל על-פיהן – כי המען הזה בזכרון מופה לבקר ההתקן. חשוב לזכור שהפקודות store / load יודעות לפעול רק מול הזכרון הראשי ולכן צריך לעשות את ה'טריק' הזה. ההתקנים אינם קבועים, ניתן להוסיף התקנים, למשל ע"י הפרוטוקול plug-and-play שמאפשר לבצע הרחבה של החלק המוקצה לזכרון חיצוני, מתוך הזכרון הראשי. חשוב: אנחנו אמנם כביכול מפחיתים ממרחב המיעון הפנימי של הזכרון הראשי, אך ההנחה היא שמרחב זה גם כך לא מנוצל ולכן אין באמת פגיעה בתפקוד הזכרון הראשי.

[מצגת fo5-3 his]

בשיטה הקודמת החסרון היה שהמעבד מבצע את הפעולות מול ההתקנים. בנוסף, בשיטה הקודמת הנחנו, מה שלא תמיד נכון, שכמות ההתקנים המחוברת למחשב היא קבועה (והרי כבר אמרנו שניתן להוסיף ע"י פרוטוקול plug-and-play, לדוגמה).

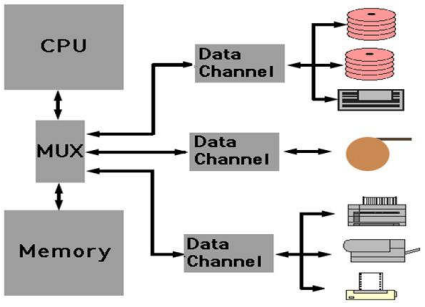
מנשק קלט-פלט הבא: גישה ישירה לזכרון, direct memory access – DMA. השיפור בגישה זו הוא שהעברת המידע בין ההתקן והזכרון לא תהיה תלויה במעבד, אלא רק בבקר הזכרון העיקרי ובבקרי ההתקנים. בשיטה זו נעשה שימוש בבקר חכם יותר, בקר DMA שבדר"כ גם יש לו cache משלו. כאשר המעבד מבצע פעולות בין האוגרים, פס המערכת פנוי. במחזורי הפס בהם המעבד לא מבצע פקודות store / load, בקרי ההתקנים יכולים להשתמש בפס המערכת, מתחזים ל'בוס'. הדבר נקרא 'פילוח' פס המערכת bus cycle stealing. ייתכן וכאשר פס המערכת פנוי, ירצו מספר בקרי התקנים להשתמש בו, לכן צריך שיהיה שבב שיחליט מי יזכה להשתמש בפס באותו זמן, זה נעשה ע"י 'בוררות' בפס bus arbitration.



הגישה הישירה מאפשרת לפקודות store / load לפעול ישירות בין הזכרון הראשי ובקר ה-DMA. כאשר בקר ה-DMA מסיים את פעולתו, הוא שולח פסיקת-DMA למעבד כדי להודיע על כך. הפסיקה נשלחת רק לאחר סיום פעולת DMA שלמה ולא אחרי כל תו. כך מורידים ב-2-1 סדרי גודל את מספר הפסיקות שמגיעות למעבד.

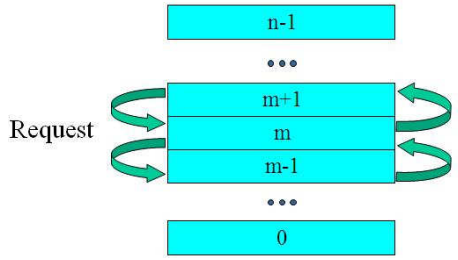
המנשקים עליהם דיברנו עד כה, ניסו להפחית את העומס על המעבד, ע"י הפחתת מספר הפסיקות המתקבלות. פיתוח נוסף של מנשק הקלט-פלט: תוכנית ערוץ, מחשב קלט-פלט, I/O processor (channel). זהו בעצם מחשב קלט-פלט ייעודי המבקר מספר בקרים, שכל בקר מבקר בעצמו מספר התקנים, ע"י יצירת היררכיה של בקרים. כביכול סוג של בקר-של-בקרים, בקר-על. [היות ואין בדיחות השיעור, עוד אנלוגיה: הדבר מזכיר את ההיררכיה של סייעות וגננת בגן-ילדים, קודם כל הילד מציק לסייעת ורק אם יש בעיה רצינית הסייעת מציקה לגננת ☺]

בקר הקלט-פלט שולט בכל פעולות הקלט-פלט. רק כאשר יסיים את כל התוכנית שהעביר לו המעבד (תוכנית ערוץ: אוסף פקודות של קלט-פלט), ישלח בחזרה פסיקת-ערוץ channel interrupt, מה שמוריד עומס ביחס לפסיקות DMA. כך מרחיקים התקנים דורשי תשומת-לב מהמעבד, כדי שהוא יהיה זמין כמה שיותר לפקודות של המשתמש ויבזבז פחות זמן על התקני קלט-פלט.



[מצגת fo5-4 his]

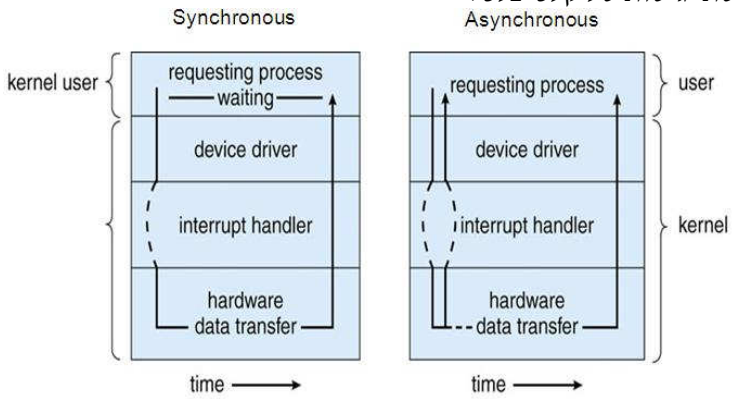
נתרכז בשני **סוגים** של מדרג:



(א) מדרג שכבתי layered. כל שכבה יכולה לבקש שרות request רק מהשכבה שמתחתיה ולתת שירות reply רק לשכבה שמעליה. אין עקיפות. מודל מובנה, מסודר, כל שכבה מכירה רק את זו שמעליה ואת זו שמתחתיה. כאשר מסר צריך לעבור, הוא עובר באופן סדרתי ולא ניתן לדלג על שכבות לצורך המעבר. מודולרי, קל לשינוי.

(ב) מדרג רמתי leveled. כל רמה יכולה לבקש שירות מכל רמה שמתחתיה ולספק שירות לכל רמה שמעליה. פחות מודולרי, פחות מסודר, יש עקיפות. מודל זה מהיר יותר, כי ניתן לדלג על רמות כאשר מעבירים מסר.

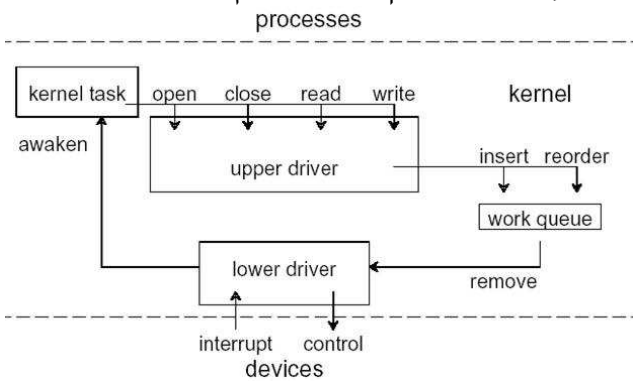
שתי גישות של קלט-פלט:



המודל השמאלי: אם A מעביר בקשה ל-B וממתין עד ש-B יבצע את פעולתו ויחזיר את התשובה ל-A, זהו מודל סינכרוני, יש רק אחד שפועל בכל זמן נתון. לעומת זאת, אם A ימשיך לבצע פעולות עד שיקבל את התשובה מ-B, זהו מודל א-סינכרוני. גישות להתקני קלט-פלט הן מתבצעות ע"פ מודל א-סינכרוני, בעוד גישה לזכרון ותשאול (pooling) הן סינכרוניות.

3 sub-layer/level

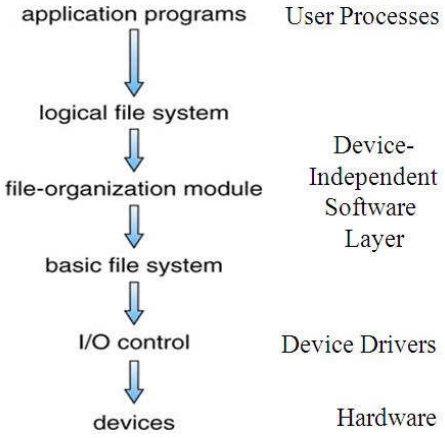
כל שכבה / רמה, יכולה בעצמה להתחלק ל-3 שכבות / רמות. נניח שאנו ברמה m: (א) תת השכבה התחתונה מבקשת שירות מהשכבה / רמה התחתונה m-1 (ב) תת השכבה / רמה העליונה נותנת שירות לשכבה / רמה m+1 שמעליה (ג) תת-השכבה / רמה האמצעית מהווה מתווך ומכילה לוגיקה ומבני נתונים משותפים.

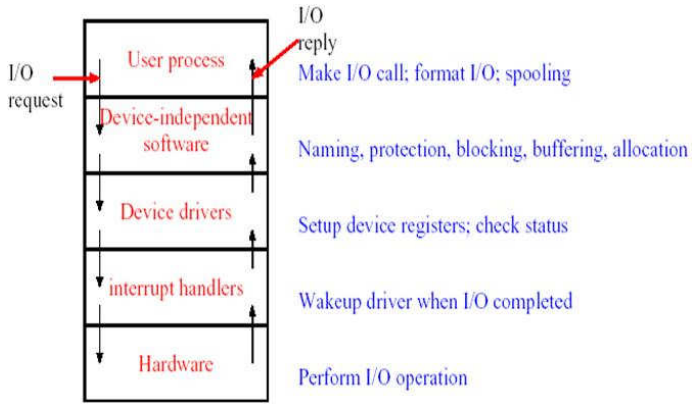


באיור: יש upper driver, יש lower driver ואילו תת-היחידה האמצעית שמבצעת את העבודה, נקראת work queue שיכולה גם לשנות את העדיפות של הפעולות אותן היא צריכה לבצע (פועלת כתור-עדיפויות). כאשר ההתקן סיים פעולתו, הוא שולח פסיקה כדי להודיע על כך (באיור מיוצג ע"י awaken).

הצורך בתור-העבודה, הוא כדי לאפשר שיטת עבודה א-סינכרונית. כך תת-הרמה העליונה ותת-הרמה התחתונה יכולות להמשיך לעבוד כל אחת בנפרד מבלי לחכות שהשכבה האחרת תסיים את

מבט אחר על שכבות הקלט-פלט: 3 הרמות האמצעיות באיור הימני, הן בעצם תת-יחידות של device independent software layer האחראית לבצע את מגוון הפעולות של שכבת ההתקן.





הרצאה מס' 5 - 16.11.09

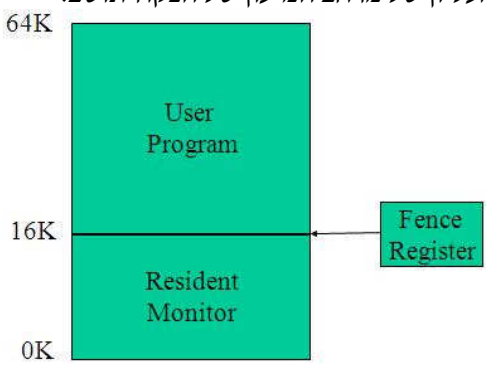
[מצגת os2-1 str : functional view of operating system]
בשיעור שעבר סיימנו את נושא מדרגי הזכרון, הבקרה על בקרי ההתקנים, מנשקים בין בקרי ההתקנים, הזכרון והמעבד.

היום נתחיל את פרק 2 : שירותים, תתי-מערכות, מבנה מערכת ההפעלה.
נתחיל בנושא 2.1 : סקירה תפקודית של מערכת הפעלה. המוטיב העיקרי – הגנה על הזכרון, המעבד והתקני הקלט-פלט.

תזכורת : בתחילה היה פקח-תושב שהיה כביכול סוג של מערכת הפעלה פרימיטיבית – קוד ראשוני שישב על הזכרון מאז ההקמה (אתחול, או בצורה תקנית 'תיחול') של המחשב, ידע לתמוך בקבצי batch (אצווה).
היום נרחיב את נושא הפסיקות (מלכודות, קריאות מערכת ופסיקות חיצוניות מהתקני הקלט-פלט).

ניהול זכרון : main memory management
(1) Minimal management – בתחילה היה זכרון מינימלי, תוכנית אחת שניהלה את הזכרון של עצמה.
(2) Memory split – בהמשך היה פיצול זכרון (ל-2 חלקים), ע"י פקח-תושב שהיה מחלק את הזכרון בין המשימות / עבודות.
(3) Memory division – חלוקה (ל-2 חלקים או יותר) של כלל המשאבים בין המשימות.
לכאורה אנחנו לא מפחדים שמערכת ההפעלה תהרוס ("trusted program"), לכן אנחנו דואגים להגן מפני תוכניות המשתמש.

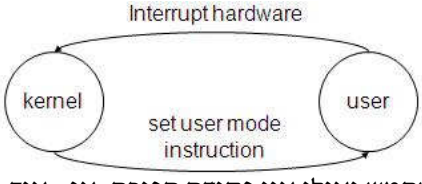
אם נשים את מערכת ההפעלה (הפקח-תושב) שלנו על זכרון מסוג ROM, הוא אכן לא יוכל להיהרס. א-ב-ל, צריך גם להתקין עדכונים למערכת ההפעלה, אנחנו כותבים אליה דברים, היא צריכה לשמור מגוון טבלאות-חישובים וערכים, שאינם יכולים להישמר על זכרון מסוג ROM – משום שזהו זכרון המשמש לקריאה בלבד (read only memory).
הפתרון נקרא fence register, אוגר-חסם : מהווה חלק מהאוגרים הייעודיים, dedicated registers. זו לוגיקה שמאפשרת להגן על הפקח-תושב. כחלק מהתיחול (אתחול) של הפקח-תושב, הוא משתמש בפקודת מכונה שטוענת את האוגר-חסם, המהווה את הגבול העליון של מרחב המיעון של הפקח-תושב.



לאוגר-חסם יש לוגיקה שאם נרצה להתייחס לכתובת החורגת ממיעון הזכרון המוקצה לפקח-תושב, יש בדיקה : אם זה לצורך קריאה, אין בעיה – אבל אם תוכנית המשתמש מנסה לכתוב למרחב המיעון של הפקח-תושב, הפעולה כמובן תיאסר, כי היא עלולה לגרום להשבתה של הפקח-תושב.
הלוגיקה של האוגר-חסם בודקת מי המשתמש בזכרון באותו זמן נתון : אם מדובר בפקח-תושב, היא מאפשרת לו לעשות כרצונו – אך אם מדובר בתוכנית-משתמש המנסה לבצע פעולת כתיבה, היא תבדוק היכן בדיוק מנסה התוכנית לכתוב. לדוגמה, במצב המופיע באיור מטה, תוכנית המשתמש תוכל לכתוב על גודל זכרון של 64-16=48K.
כל זאת בלוגיקה של פיצול הזכרון, memory split, לפיה הזכרון נחלק לחלק המיועד לפקח-תושב בלבד וחלק אחר המיועד גם לתוכניות המשתמש.

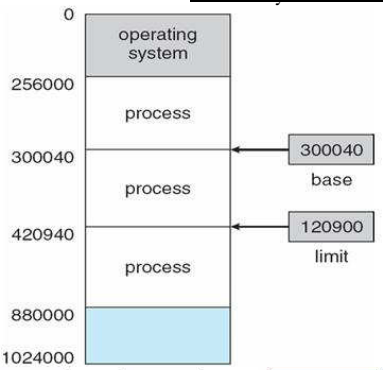
תעלומות ובעיות :
(1) איך נדע מי התוכנית שרצה כרגע (פקח-תושב / משתמש) ?
(2) אם נתפוס את תוכנית המשתמש מנסה לכתוב בחלק מהזכרון המיועד לפקח-תושב בלבד, מה נעשה ?
(3) מה מונע מתוכנית של המשתמש להפעיל פקודה הטוענת את האוגר-חסם, להתחזות לפקח-תושב ובעצם לגרום להשבתה של האוגר-חסם ?

פתרונות :
(3) הפקודה הטוענת את האוגר-חסם, ניתנת ליישום רק ע"י הפקח-תושב. בעיה חדשה : כיצד בעצם נכריז על 'פקודה מיוחסת'?
(1) dual-mode operation : יש סיבית-מצב, בעלת שני מצבים (לפחות) וכך ניתן לדעת מי הפעיל את התוכנית שרצה כרגע :
המשתמש או מערכת ההפעלה עצמה. ההנחה היא שבמצב 'משגוח' (מערכת ההפעלה) ניתן לבצע כ-ל פקודה, ואילו במצב משתמש לא ניתן לבצע פקודות מיוחסות ולגשת למקומות בזכרון השייכים למשגוח. סיבית המצב מהווה חלק מאוגר-מצב, בעצם דגל שאומר אם הוא במצב משתמש או במצב לא-משתמש. לפי ההנחה, ע"י בדיקת סיבית-המצב נוכל בקלות לדעת אם עלינו להגביל את הפקודה שרצה כרגע או שניתן לה חופש מוחלט. אך בעצם יצרנו בעיה נוספת, כי צריך להגן על סיבית-המצב, שחלילה המשתמש לא ישנה את הערך בה. אסור לאפשר למשתמש להפעיל פקודה שתוכל לגרום לו לבצע משימות בתור מערכת ההפעלה, כי הוא יוכל לגרום נזקים ☹.
(2) אין תשובה פשוטה, לא ניתן פשוט להגדיר את הפקודה השולטת על סיבית-המצב כ'פקודה מיוחסת' : אם אנחנו ה'משגוח', לא נרצה להעביר את השליטה למשתמש, בעוד אם אנחנו המשתמש לא נוכל להפעיל את הפקודה בכל מקרה – ולכן עדיף שלא תהיה פקודה כזו(!). אבל כן נדרשת פקודה להעברת המצב של סיבית-המצב ממצב 'משגוח' למצב משתמש, כדי לאפשר לתוכניות המשתמש לרוץ.



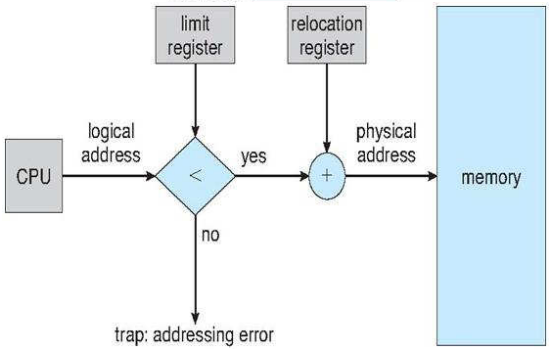
יש פקודה שמעבירה ממצב 'משגוח' למצב משתמש ואילו אין פקודה הפוכה. אז.. איך בעצם אנחנו חוזרים ממצב משתמש למצב 'משגוח'? הפתרון הוא מנגנון חומרה, בארכיטקטורה של המחשב עצמו, שיעביר אותנו למצב 'משגוח' – ע"י פסיקה. פסיקה מחוללת (hardware trap) שמשנה את המצב ל'משגוח' בלי קשר למצב שהיינו בו קודם לכן. אם ל'משגוח' אין מה לעשות (הוא רק שותה קפה ☺) הוא יכול כמובן להחזיר את השליטה (המצב) למשתמש.

Memory division:



עד כה ראינו איך ניתן להגן על הפקח-תושב מתוכנית-משתמש יחידה. אך אם יש לנו מספר תוכניות-משתמש? אנחנו צריכים להיות מסוגלים להגן על הפקח-תושב מפני תוכנית-המשתמש, וגם להגן על שאר תוכניות-המשתמש מפני תוכנית-המשתמש שרצה ברגע זה. השיטה של חלוקת הזכרון (memory split) אותה הזכרנו קודם לא תפתור את הבעיה. אז.. אולי נשתמש בשני אוגרי-חסם ? כך נוכל לקבוע קו גבול עליון וקו גבול תחתון עבור תוכנית המשתמש שרצה ברגע נתון.

באיור: base הוא אוגר-חסם תחתון ואילו limit הוא אוגר-חסם עליון, מגבילים את מרחב הכתיבה של התוכנית. אם נבצע בדיקה שהתוכנית הרצה ברגע מסוים לא חורגת מהגבולות שהוקצו לה, היא לא תפגע בשאר התוכניות.



כאשר תוכנית מבקשת לבצע פעולת כתיבה לזכרון, תחילה יש בדיקה האם היא חורגת מתחום הזכרון שהוקצה לה (אם היא חורגת – יש פסיקת מלכודת של החומרה, המחזירה את השליטה לידי מערכת ההפעלה), לאחר מכן יש המרה לכתובת פיזית בזכרון ופעולת הכתיבה מתאפשרת.

פסיקות מלכודת (traps) של החומרה: מופעלות במקרים של גישה למען לא חוקי, ניסיון לבצע פקודה מיוחסת ע"י המשתמש, ניסיון חלוקה באפס, חישובים עם גלישה / חמיקה בפעולות של מספרים עם נקודה צפה (floating point), ניסיון לבצע פקודה לא קיימת (שאין לה קוד כלל), גישה מחוץ למרחב המיעון שהוגדר לתוכנית המשתמש. כאמור, במקרים אלו, נחזור למצב 'משגוח'.

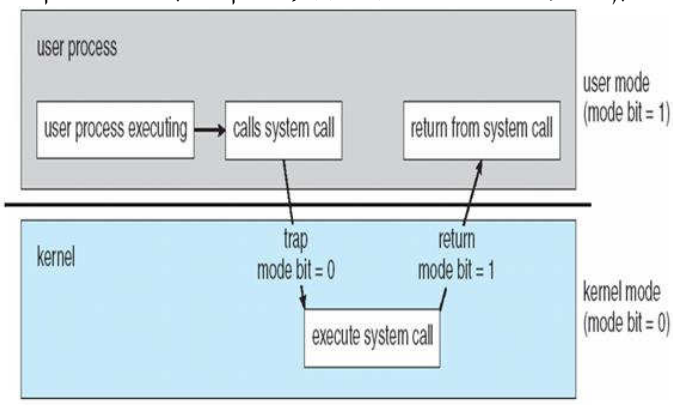
סיכום נושא הגנת הזכרון:

(1) הגנה על זכרון הפקח-תושב, ע"י שימוש באוגר-חסם ולוגיקה המונעת גישה של תוכניות המשתמש למרחב המיעון של הפקח-תושב.
(2) הגנה על אוגר-החסם, ע"י הפעלתו בעזרת פקודות 'מיוחסות'.
(3) הגנה על הפקודות המיוחסות (כך מוודאים שתוכנית המשתמש לא מנסה לבצען), ע"י סיבית-מצב הקובעת מי בשליטה כרגע (OS או המשתמש).
(4) הגנה על סיבית-המצב ע"י פסיקת-חומרה (hardware trap) שלא יכולה להתבצע ע"י המשתמש.

סיכום של המנשק בין הזכרון, המעבד וההתקנים: נתונים זורמים בין המעבד והתקני הקלט-פלט, ביניהם יכולה להתרחש פסיקה חיצונית. ההתקנים יכולים לבצע DMA (dynamic memory access, פסיקה עליה דיברנו בשיעור שעבר) 'פילוח' של פס המערכת, ע"י דילוג על המעבד ופעולה ישירות מול הזכרון.

פסיקה חיצונית external interrupt: מופעלת ע"י בקרי הקלט-פלט, קוצב הזמן שבעצם מהווה חלק מהמעבד, או תקלות חומרה.

הגנה על התקני הזכרון החיצוני והתקני הקלט-פלט:
כאשר קוראים קובץ אצווה לפקח-תושב, הוא אמור לדעת לזהות את ההתחלה והסוף של הנתונים ושל הקוד. אם נאפשר לכל כרטיס לקרוא משאר הכרטיסים (למשל נאפשר לעבודה מסוימת לקרוא גם מכרטיס של העבודה שאחריה, בתור חלק מהנתונים שלה), אנו עלולים ליפול לבור שחור (מסוכן.. ☺). מה שכמובן יוצר בעיה להגן על תוכניות הקלט-פלט.

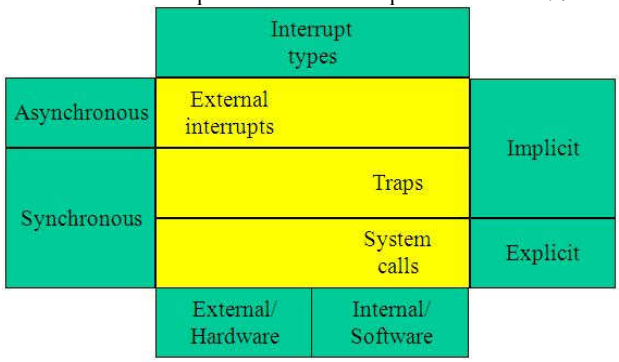


הפתרון: לצורך הגנה יש קריאות מערכת, system calls. העבודה הרצה מחוללת פנייה למערכת ודרכה מבקשת לבצע את הפעולות. זו פקודת מכונה שנקראת trap, מחוללת את מנגנון החומרה הגורם למלכודת ומעבירה למצב 'משגוח': המשתמש מאבד שליטה לטובת ה'משגוח', ה'משגוח' בודק אם הבקשה לגיטימית (אם כן מאפשרת למשתמש להמשיך, תחת שליטה), בסיום מחזירה את השליטה למשתמש (לאחר ששינינו בחזרה את סיבית המצב למצב משתמש).

הגנה על המעבד:
צריך להגן על זמן העבודה של המעבד, לא לאפשר לתוכנית משתמש לפעול במשך זמן רב מדי (לולאה אין-סופית) שתגזול את כל זמן העבודה של המעבד. [בדיחה שיווקית: סוכן מכירות אומר ללקוח שהמחשב המתקדם שהוא מנסה למכור יכול לבצע לולאה אינסופית ב-6 שניות ☺] איך נמנע מתוכנית המשתמש לרוץ במשך זמן רב מדי? נפעיל קוצב-זמן, שכאשר נגמר ה-counter שלו הוא מפעיל timer interrupt. במצב כזה ה'משגוח' יכול להחליט שהוא מחזיר את השליטה לתוכנית המשתמש, או שהוא מסיים את פעולתה (כי היא חוצפנית ולא הפסיקה ☺).

<u>סוגי פסיקות</u>:
(1) מלכודות hardware traps (ע"י החומרה).
(2) חיצוניות interrupts (ע"י בקרי ההתקנים).
(3) קריאות-מערכת systems calls (פניה למערכת שתבצע עבורנו דבר מה).

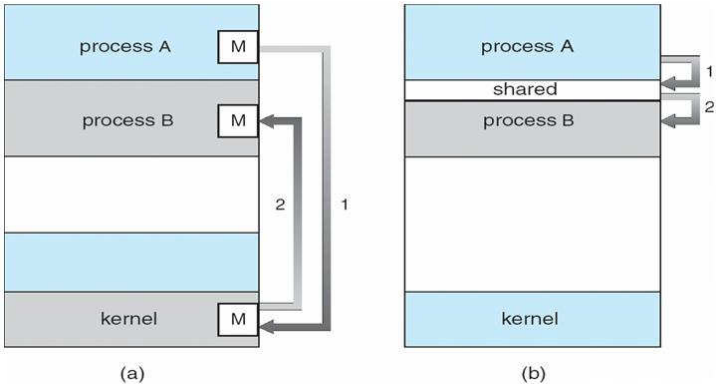
<u>מדוע בעצם לא ניתן להסתפק בסוג פסיקה יחיד? נבחן את הנושא במספר היבטים</u>:
(א) סינכרוני / א-סינכרוני.
(ב) חיצוני-חומרתי / פנימי-תוכניתי.
(ג) מפורש explicit / משתמע implicit.



פסיקה חיצונית-חומרתית היא באופן א-סינכרוני, ללא קשר למה שהתוכנית מבצעת באותו רגע. לעומת זאת מלכודות וקריאות מערכת הן סינכרוניות, שכן הן קשורות למה שהתוכנית שרצה באותו זמן מנסה לבצע. פסיקת-שעון-זמן למשל היא חיצונית, בעוד מלכודות וקריאות המערכת הן פנימיות, מתרחשות בגלל התוכנה שרצה כרגע. המלכודות הן באופן משתמע, לא משהו שהתוכנית של המשתמש ביקשה לבצע. לעומת זאת קריאות המערכת מתבצעות באופן מפורש, כי תוכנית המשתמש צריכה תחילה להיבדק אם בקשתה חוקית ולבקש מהמערכת שתבצע עבורה פעולות מסוימות.

[מצגת os2-2 str]

<u>מנשק המשתמש</u>: בתחילה היה command line, בהמשך קבצי batch ולאחר-מכן התפתח ממשק גרפי GUI. כיום כביכול יש MUI (multimedia user interface – מסכי מגע). שם חלופי ל-GUI זה WIMP (windows, icon, mouse, pointing-devices). מערכת ההפעלה מספקת מספר שירותים: הרצת תוכנית, גישה לקבצים, תקשורת, הקצאה ושחרור של משאבים, קלט-פלט, טיפול בשגיאות. ע"י קריאות המערכת, המשתמש יכול לנצל את שירותי המערכת.



<u>מודלים של תקשורת</u> (חלופיים זה לזה): message passing / sharing. אם שני תהליכים (לדוגמה A ו-B) רוצים לשתף ביניהם דבר-מה, יהיו חייבים להקצות מחיצת-זכרון משותפת ביניהם, שלשניהם תהיה גישה (איור b). א-ב-ל, מרחב המיעון המשותף צריך להיות מוגדר גם במרחב המיעון של תהליך A וגם במרחב המיעון של תהליך B. במודל של העברת מידע (איור a), המידע מועבר ע"י מערכת ההפעלה. מבחינת מהירות, יש תקורה בהקמת זכרון משותף (מחיצה), אך השיתוף קל יותר ואף הכרחי כאשר מדובר ברשת.

<u>ללמוד לבד: סוגי השירותים של מערכת ההפעלה (שקופיות 20-7).</u>
<u>הגנה לעומת אבטחה</u>: הגנה – אמצעים שנוקטים בתוך המחשב, אבטחה – הגנה כלפי חוץ מפני מזיקים שמגיעים דרך הרשת למשל.
[עד כה דיברנו בשיעור זה על הגנות, לעומת זאת אבטחת המחשב היא למשל כנגד וירוסים.]

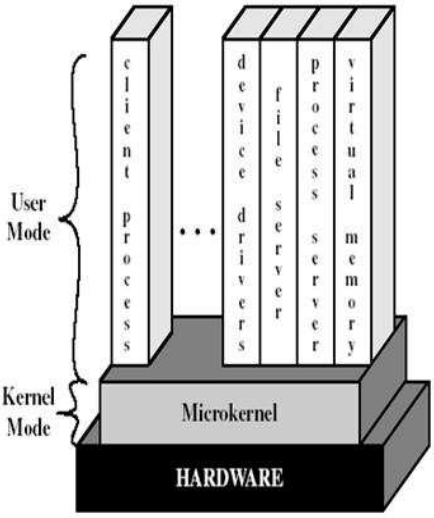
<u>קריאות מערכת system calls</u>:
מהוות מנשק בין תוכניות המשתמש ובין הגרעין (מערכת ההפעלה), מאפשרות גישה לקלט-פלט. קריאות המערכות מאפשרות לבצע זאת בצורה מוגנת. הקריאות יכולות להיות כתובות בשפת סף (אסמבלר, שפת מכונה) אך כתובות בעיקר בשפה עילית (C \ C++). קריאות המערכת מופעלות ע"י API (application programming interface) כדי לקבל את השירות המבוקש. ישנם מגוון מנשקי API. קריאת מערכת מהווה קריאה חיצונית, מבקשים ממערכת ההפעלה לבצע עבורנו דבר מה. prompt (כמו מנחה בתוכנית TV) נותן הוראה למשתמש, למשל בקשה להקלדת קלט. ישנו מציין, מספר-מזהה ייחודי, לזיהוי הסוג של הפסיקה / קריאת המערכת. קיימת טבלה שמבצעת קישור בין המספר-המזהה ובין הפקודה שצריך לבצע. בעת הפעלה של פסיקה, היא צריכה כמובן להזדהות.

כמובן, מי שמפעיל את קריאת המערכת, צריך רק לדעת מה השם של הפסיקה ואילו פרמטרים (ואת הסוגים והסדר שלהם) היא צריכה לקבל. איך היא מתבצעת בפועל – צריך להיות מוסתר ממי שמפעיל את קריאת המערכת (encapsulation, כימוס מידע).
מספר אפשרויות להעברת הפרמטרים לקריאות המערכת:
(1) הפשוטה ביותר, ע"י האוגרים. הצד הקורא (נקרא מזמן, caller) והצד הנקרא (מוזמן, callee) צריכים להסכים באילו אוגרים הם משתמשים להעברת הפרמטרים. חשוב לזכור שניתן להשתמש רק באוגרים הכללים general purposes (ולא באלו השמורים רק למערכת ההפעלה) שכמותם קטנה. ייתכן וצריך להעביר יותר פרמטרים משניתן להעביר ע"י האוגרים.
(2) שימוש ב'בלוק' / טבלה שתכיל את כל הפרמטרים. כך נוכל להעביר מצביע לטבלה (למשל ע"י אוגר, שישמש כמען גישה עקיף indirect address) מהצד הקורא לצד הנקרא. שיטה זו לא מוגבלת בגודל (מספר הפרמטרים שצריך להעביר).
(3) במקום להשתמש במבנה נתונים ספציפי – ניתן להשתמש במחסנית (stack) הספציפית של התוכנית: נדחוף את הפרמטרים בסדר הפוך למחסנית וכאשר הצד הנקרא מבצע פעולות pop הוא יקבל את הפרמטרים בסדר הנכון. שיטה זו היא הגמישה ביותר, מבחינת מספר הפרמטרים שניתן להעביר בין הצדדים.

תוכניות מערכת system programs:
לקטגוריה זו משתייכים עורכי תמלילים, מהדרים.. זו רמה מעל קריאות המערכת. מאפשרות לבצע פעולות של ניהול קבצים כגון יצירה, העתקה, מחיקה, שינוי שם, הדפסה.. [רגע של עברית: לבצע מניפולציה = טפלול (מזכיר טיפול).]

מטרות התכן של המערכת system design goals
מעל קריאות המערכת יש תוכניות מערכת ולמעלה יש את מערכת ההפעלה עצמה. אחד העקרונות בתכן המערכת, הוא להבדיל בין המדיניות והמנגנון. הרעיון: המדיניות לא צריכה להיות מושפעת מהמנגנון. צריך ליישם את המנגנון (למשל queue) ולהפעיל אותו ע"י הרמה הגבוהה יותר (למשל המשתמש) בהתאם לצורת היישום המבוקשת. הפרדה בין header ו-body.
ביצוע תיכון למערכת ההפעלה: בעבר זה נעשה בשפת אסמבלר, להיות 'קרובים' לחומרה. כיום זה כבר נעשה בשפות עיליות.
מערכות הפעלה פתוחות (כדוגמת לינוקס.. open source, OS) – סובלות מבעיות של זכויות יוצרים והרשאות, אך מאפשרות למתכנתים חובבים לכתוב כל מיני תוספות למערכת ההפעלה כמו למשל שיפורים.
יש קוניה שנקראת wintel – כל פעם מיקרוסופט מוציאה גירסה חדשה של חלונות, שמותאמת לגירסה חדשה של מעבד אינטל. המגמה שנלחמת בקוניה היא 'ענן' – הכל יושב בענן וניתן להשתמש על סוגי חומרה שונים.
[בדיחה דתית: ע"י שימוש ב'ענן' קל יותר לתקשר עם אלוהים, כי זו שיחה מקומית ☺]
SYSGEN (system generation) היא תוכנה המכילה מידע לגבי החומרה הספציפית עליה המערכת רצה, מאפשרת לחלל מערכת הפעלה. מרכיב ה-booting (אתחול) מפעיל מספר תהליכים שכל אחד גדול מקודמו, עד שכל מערכת ההפעלה אותחלה. boot strap (רצועה במגף המאפשרת לנעול אותו על הרגל) מהווה אמצעי לאתחול (נעילה של המגף) של מערכת ההפעלה. המרכיב הראשון הוא מה-ROM, בהמשך יש טעינה של חלק ממערכת ההפעלה מהדיסק, עד להפעלה המלאה.
מיפוי שגיאות debugging, פעולה חשובה שמתבצעת ע"י מתכני המערכת.
מיטוב של מערכת הפעלה (אופטימיזציה).
[בדיחה של דייגים: איך יוצרים תקשורת עם דג? מעלים אותו ברשת ☺]
[בדיחה קטלנית: למה דג גומר בצלחת? כי הוא פתח את הפה (אכל את הפיתיון) ☺]

[מצגת os2-3 str]
מבנה / ארגון מערכות ההפעלה:
בתחילה הייתה מערכת הפעלה מונוליטית, גוש יחיד שהכיל הכל, גם את שירותי המערכת. א-ב-ל, תוכנית יחידה היא גדולה מדי וכבדה מדי. תוכנה גדולה קשה לתחזוקה. ה-MS-DOS הראשוני היה מונוליטי, בהמשך הפכו אותו להיות שכבתי. גם הגרעין של UNIX היה מונוליטי. מערכת הפעלה היא לרוב בצורה של רמות ולא של שכבות. אם יש שתי שכבות, כאשר מדובר בהיבטים של ניהול זכרון וניהול תהליכים, כיצד נחליט מי מהן 'עליונה' ביחס לאחרת? ככל שלמערכות ההפעלה נוספו עוד אפשרויות ותוספות, היא כבר נהייתה 'נפוחה' מדי. [בדיחה מסריחה: מערכת הפעלה מנופחת – 'נופחה'] לכן פותחה גישה של גרעין זעיר micro-kernel, הכולל פונקציות חיוניות (זמנון תהליכים, ניהול זכרון פרימיטיבי, קלט-פלט, תקשורת בין תהליכים) ואילו כל שאר שירותי מערכת ההפעלה יהיו בהתקנים, במערכת וירטואלית ובמערכת הקבצים. כך הגרעין הופך להיות lean and mean (רזה וממזר ☺).



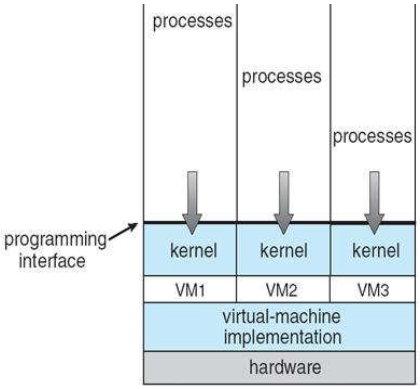
כל השירותים רצים בצורה אנכית. האיור בצד ימין מזכיר את מבנה החומרה של לוח-האם, mother-board: ניתן לחבר כרטיסים נוספים (או שירותים נוספים) ללוח-האם (למערכת ההפעלה). השירותים הללו מתקשרים ביניהם דרך הגרעין-הרזה. זוהי ארכיטקטורה גמישה, הגנתית, פחות קוד רץ במצב 'משגוח' לעומת המודל השכבתי הקלאסי. כמובן שיש מחיר, לא ניתן לבצע shared memory בין התהליכים הרצים עם הגרעין-הרזה, אלא צריך להשתמש בטכניקה של העברת מסרים. זו גישה של מכוון-עצמים. אפשר לחשוב על הגרעין-הרזה כעל סוג של 'פס' דרכו מועברות הפונקציות המקשרות בין מנשק-המשתמש ובין העצמים השונים.

אנקדוטה: כאשר IBM רצתה לצאת עם מערכת הפעלה מתחרה לחלונות של מיקרוסופט, הם גרמו ל-FUD (fear, uncertainty, doubt), טכניקה של זריעת בהלה, למנוע מאנשים לקנות תוכנה מסוימת כי כביכול אני עומד לצאת עם תוכנה מתחרה טובה יותר.

הרצאה מס' 6 - 23.11.09

[מצגת os2-3 str. התחלנו בשיעור שעבר, הפעם המשכנו החל משקופית 33]
ארכיטקטורות של מערכות הפעלה
תזכורת מהשיעור הקודם: בתחילה המערכות היו מונוליטיות ובלתי ניתנות לשינוי. בהמשך היה מדרג שכבתי/רמתי של היחידות השונות זו מעל זו (מודל אופקי) מה שגרם לבעיה כאשר צריך לעבור דרך מספר שכבות. בהמשך היה מודל של micro-kernel כאשר השכבות הן בצורה אנכית וכך ניתן להוסיף כרטיסים / לוחות / שירותים. כל השירותים רצים במצב של user mode ולכן לא יכולים להפיל איתם את מערכת ההפעלה.

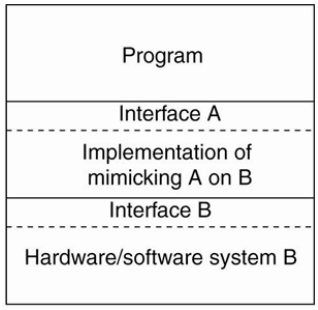
וירטואליזציה – virtual machines
ניתן להסתכל על מערכת הפעלה כעל סידרה של הרחבות תוכנה, עד שמגיעים ל-shell, סביבת האינטראקציה של המשתמש עם המחשב. בכל שלב יש מכונה וירטואלית שמולבשת על המכונה בשכבה הקודמת. בשנות ה-70 המוקדמות הייתה מערכת הפעלה של חברת IBM שנקראה VM, שהייתה מהפכנית: על החומרה רצה VM ועליה מערכות CMS (שיתוף זמנים, conversational monitor sharing). זו הייתה מערכת הוירטואליזציה הראשונה, עליה ניתן היה להריץ מערכות אחרות, סוג של hypervisor ('משגוח'-על).



בעצם כך ניתן לקחת מחשב פיזי יחיד ולהפוך אותו למספר מחשבים, כל משתמש מרגיש שהוא היחיד שמריץ תוכניות על המחשב הפיזי. זה דמוי-מקביליות. כמובן (אנחנו אמורים לזכור זאת מהשיעורים הקודמים), יש גם מקביליות אמיתית: המיקרו-בקרים של התקני החומרה פועלים במקביל למעבד של המחשב.

בצורה כזאת ניתן להריץ סוגים שונים של מערכות הפעלה, על אותו מחשב. כל אחת מה-VM לא מפריעה לשאר, כי היא רצה בנפרד, כמו process נפרד. בהמשך רצו לעשות וירטואליזציה גם לשרתים, לאמצעי אחסון, לאמצעי תקשורת ולא רק למערכות הפעלה.

ניתן להריץ מערכת הפעלה שרצה בצורת אצווה (batch, ברקע) ואחרת שרצה בצורת שיתוף (רצה בחזית), שתיהן על אותו מחשב.



כדי להריץ מערכת הפעלה A על מערכת ההפעלה B, צריך תחילה לממש סוג של 'התחזות'. המנשק של מערכת ההפעלה A צריך לדעת לתקשר עם מערכת ההפעלה B (מיפוי מ-A ל-B). באופן כזה תוכנית A תוכל לרוץ באופן חופשי. המכונה הוירטואלית A היא בעצם תוכנה בלבד. זו הדמיה, כי מדובר בוירטואליזציה של תוכנה בלבד.

הבדל בין החקיה להדמיה: שתיהן וירטואליזציה בהגדרה. הדמיה נעשית בתוכנה בלבד, החקיה (אמולציה) נעשית בשילוב של חומרה ותוכנה.

הטכניקה של וירטואליזציה ניתנת לביצוע ברמות שונות: במנשק בין החומרה למערכת ההפעלה (הרמה שבה VM של חברת IBM יושמה), במנשק בין מערכת ההפעלה ל(קריאות ה)ספריה, במנשק בין שירותי הספריה והיישומים עצמם. למשל, אפליקציות של java שאנחנו מפעילים כשגולשים באינטרנט, רצות בסביבת JVM, בדפדפן שלנו.

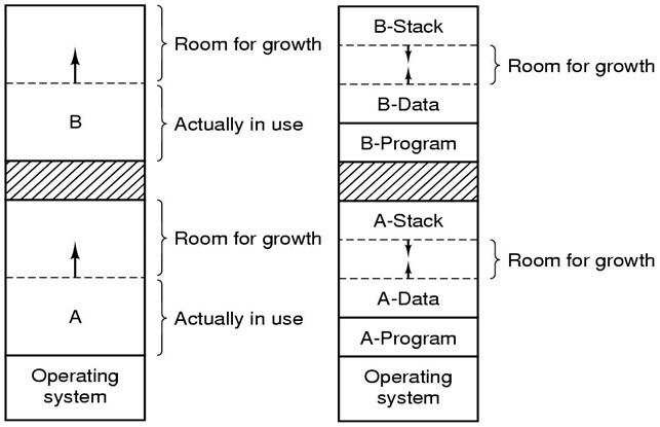
<u>מיחשוב 'ענני' – cloud computing</u>
מקבלים את שירותי המחשב משרתים חיצוניים. מערכת ההפעלה החדשה של גוגל (מיקרוסופט החדשה??) פועלת כך. כביכול סגירת מעגל: פעם היה מחשב יחיד וגדול שנקרא main-frame שנתן את השירותים לכולם. כיום מיחשוב ענן די דומה לזה, כי כולם צריכים לגשת דרך מחשבי הכף-יד שלהם לקבל שירות מה'ענן'. לא בדיוק סגירת מעגל, כי יש כאן קצת התקדמות (יופי, גוגל!): כבר לא מקבלים שירותים ממערכת ההפעלה המקומית, אלא מ'ענן' מרכזי – the network is the computer, מערכת ההפעלה נמצאת ברשת ולא על המחשב.

<u>סוגים של hyper-visiors</u> (השניים הראשונים הם העיקריים, על השלישי לא הרחבנו):
(1) רץ מעל החומרה.
(2) רץ על מערכת הפעלה מקומית. הוא 'רזה' יותר ומקבל את כל השירותים ממערכת ההפעלה המקומית. יש מערכת הפעלה יחידה שרצה פיזית ואלו שרצות עליה, רצות בצורה וירטואלית. בעצם מדובר בהכללה של רעיון ה-micro-kernel. כך חוסכים בעלויות, מערכת הפעלה שלא רצה ישירות על החומרה עולה פחות ☺. החברה VMware מתמחה בתחום.
(3) סוג נוסף "דמוי וירטואליזציה" para-virtualization. דורש התאמות לפני שניתן להריץ מערכת הפעלה נוספת.

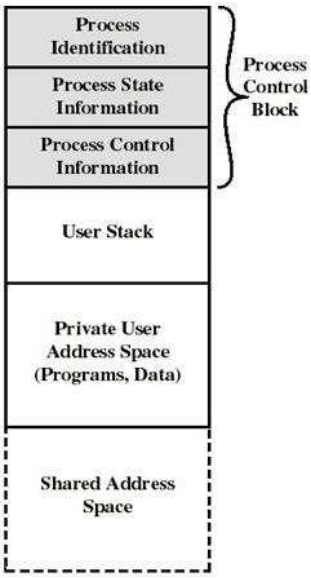
<u>וירטואליזציה של שרתים</u>
ניתן להריץ מספר שירותים על מחשב יחיד, על ידי יצירה של סביבה וירטואלית עבור כל סוג שירות, כך שהם לא יפריעו זה לזה. הם רצים בסביבת user mode ולכן הם לא יכולים להפיל את מערכת ההפעלה. אמנם המחשב הזה יצטרך להיות חזק יותר, כי נותן שירותים רבים יותר משרת יחיד, אך מאפשר חסכון של מספר מחשבים. וירטואליזציה של אחסון מאפשרת לשלוח למספר מקומות (משמש לגיבוי). כאשר מערכת האחסון מבינה שיש קבצים שלא ניגשים אליהם, היא 'דוחפת' אותם 'אחורה', לחלק פחות נגיש.

(הפסקה מתודית לצורך הכנת פופקורן, לפני הצפייה ב) [סרטים: server_virtualizaion , virtualization_tutorial ,storage_virtualization]. הסרטים נמצאים באתר הקורס, בתיקיה virtualization → resources. [כדי לצפות בסרטונים צריך להתקין נגן FLV שניתן להוריד למשל מכאן: http://download.cnet.com/FLV-Player/3000-13632_4-10467081.html.]

[מצגת os3-1 pro]
<u>מבוא למודל התהליכים</u>
כל ישות / התקן / משאב במערכת ההפעלה, אפילו פאסיבי כמו קובץ, מיוצג ע"י מרכיב אקטיבי שמפעיל אותו. תהליך הינו חלק מתוכנית בביצוע, קטע קוד שמתבצע ברגע זה. מושגים סטטיים: עבודה (job) המורכבת ממספר שלבים (step). כל עבודה מתבצעת במודולים (load module).
כל תהליך מכיל 3 מקטעים:
(א) הקוד.
(ב) הנתונים: משתנים גלובלים ואזורים להקצאות דינאמיות (heap, 'מצבור').
(ג) המחסנית: מכילה מידע הקשור לזימוני שירות.



שתי שיטות לניהול מרחב הכתובות של תהליכים: בשיטה הישנה (צד שמאל), חילקו מראש אזור בגודל מסוים לכל תהליך, בו הוא יכול לפעול כרצונו. אך ייתכן ותהליך מסוים ירצה לחרוג מהמקום שהוקצה לו בעוד תהליך אחר נטול צורך במקום נוסף. בשיטה החדשה (צד ימין), הקצאת הזכרון נכונה יותר, מאפשרת לתהליך 'לגדול' הן מהכיוון של הנתונים והן מהכיוון של המחסנית. כך, לא מגיעים למצב שבו הקצינו מקום עודף או שתהליך סיים את ההקצבה שלו ולא יכול להמשיך לגדול.



יש מגוון ישויות / משאבים במחשב (process entities), לכל אחת יש את המאפיינים הייחודיים שלה: ה-ID של התהליך, ה-ID של האבא (התהליך שיצר את התהליך הנוכחי), עד כמה התהליך 'מועדף' בביצוע ביחס לתהליכים אחרים, כמה שימוש התהליך עשה במשאבי המערכת.. ניתן לחלק את המאפיינים ע"פ סוגם: כאלו השייכים לניהול זכרון, כאלו השייכים לניהול תהליכים (איפה שמור התהליך, מי מורשה לשנות את התהליך). קיימת טבלת-תהליכים (process table) שכל כניסה בה מייצגת תהליך בודד, עם כל המידע שקשור אליו. גם אם המידע לא נמצא פיזית בטבלה זו, הכניסה בטבלה חייבת להכיל לפחות מצביע למיקום הפיזי בו נמצא המידע על התהליך.

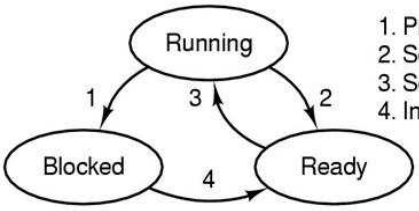
כל המידע של התהליך הבודד בטבלה נקרא גם process control block – PCB, מורכב ממידע מיקומי (היכן המידע של התהליך שמור), זיהויי (מספר הזהות של התהליך ושל תהליך-האבא שיצר אותו) ומצבי (המצב בו נמצא התהליך כרגע).

בצד ימין ניתן לראות process image (תמונה של התהליך), על כל החלוקה של האזור המכיל את ה-PCB והאזורים הקשורים להקצאות הזכרון, כדוגמת האזור המשותף לו ולתהליכים אחרים והאזור השייך רק לתהליך עצמו (יכול לשחק שם בבובות ברבי בלי שאף אחד ידע ☺).

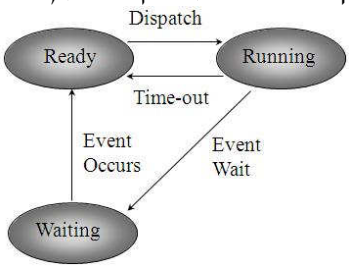
<u>הרחבה לגבי מצב התהליך</u>

כאשר תהליך רץ במערכת, הוא עובר בין מספר מצבים שונים. כל תהליך, בכל רגע נתון, יכול להיות רק במצב יחיד. המצבים העיקריים: (1) running state – התהליך שרץ כרגע במערכת. במחשב עם מעבד יחיד, יש תהליך יחיד כזה (זה יכול להיות תהליך 'משגוח' או תהליך של המשתמש). (2) ready state – מוכן לרוץ (נעל נעלי ספורט ☺). בכל רגע נתון יש אפס או יותר תהליכים כאלו, יושבים בתור של תהליכים המוכנים לריצה. יש תהליך סרק (ideal process), שמחכה לרוץ במערכת אחרון, הוא מייצג את העובדה שאין תהליך פרודוקטיבי אחר שמחכה לרוץ. (3) waiting\blocked state – תהליך שנמצא במצב המתנה / מצב חסום, מחכה למאורע (סיום בקשת קלט / פלט שהוא יזם, תקשורת בין-תהליכית) שיעורר אותו.

למטה: דוגמה לאוטומט (יילמד בקורס ביולוגיה חישובית) המעביר בין המצבים השונים של התהליך. dispatch: בתום ריצת התהליך / בסיום פלח הזמן הנוכחי, התהליך יכול לוותר על זמן המעבד ולאפשר לתהליך אחר לרוץ במקומו.



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available



ניתן גם לדמיין מכונת מצבים מורכבת יותר, המכילה 5 מצבים לעומת 3, הודות להוספה של שני מצבי קצה: (4) המצב התחילי של חילול (יצירה, generation) של תהליך (5) המצב הסופי-טרמינלי (מוות של תהליך). לאחר שתהליך מוכן לריצה, הוא עובר ממצב new למצב ready (מוכן לריצה). לאחר שתהליך סיים את ריצתו (רוצה להוריד נעלי ספורט ולהתקלח ☺) הוא עובר למצב terminated (מובא לקבורה). תהליך יכול להיות במצב המתנה בגלל מספר סיבות: השביתו אותו, מחכה לקלט מהמשתמש.. במצב new, התהליך רק נוצר (נמצא עדיין ברחם ☺), עדיין לא מוכן לריצה (קודם לומדים ללכת ורק אח"כ לרוץ ☺). יש גם תהליכים זומבים (מתו אך עדיין לא הובאו לקבורה ☹): כאשר תהליך סיים ריצתו אך המשאבים שלן עדיין לא הוחזרו למערכת, המידע בו התקלקל ולא ניתן לשחרר את הזכרון שלו, או שלא ניתן לסיים אותו בצורה מסודרת, הוא הופך להיות זומבי.
[בדיחה משפטית: תהליך זומבי סובל ממשפטפטנים (משפטן+פטפטן), 'עוכרי-דין' שמתעסקים בירושה שלו.]

<u>מודלים של יצירת תהליכים</u>
ישנו תהליך INIT ('תהליך האפס', מייצג את המערכת), שממנו 'מושרצים' מגוון תהליכי-ילד (ע"י הפונקציה ()fork). כך נוצר לנו 'עץ משפחתי' ☺. כמו שלמדנו בתרגול, יש מקרים חריגים, למשל מוות של הילד לפני ההורה ולהפך. למשל, לפני מוות של האבא, צריך קודם לדאוג שהילדים ירשו אותו. לכל תהליך יש מזהה ייחודי, PCB, קישורים.. ביוניקס כל אלו נוצרים אוטומטית ע"י הפונקציה fork. סיום של תהליך יכול להיות להתבצע ע"י : halt, kill (ע"י האבא), התרחשות של שגיאה כלשהי (ואז התהליך בעצם מגורש מעל פני האדמה ☺) כדוגמת כשל קלט-פלט, התאבדות (ביצוע ()exit ע"י התהליך עצמו).

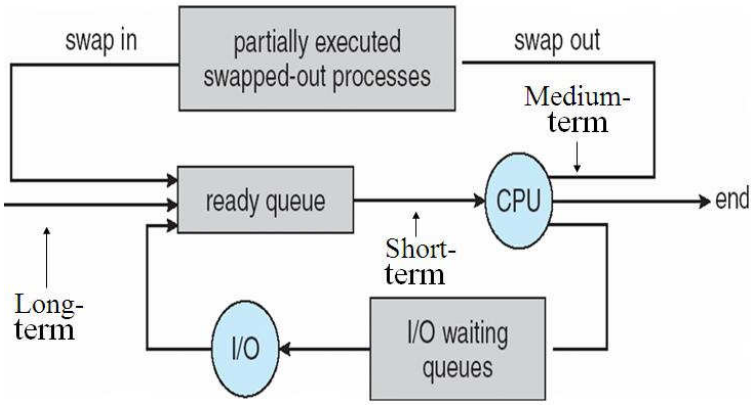
[מצגת os3-2 pro]
<u>זמנון (scheduling) תהליכים</u>
נושא זה עוסק בסדר בו התהליכים נטענים למערכת ממצב מוכן למצב ריצה.
הקדמה קצרה : כל עבודה יכולה להיות מיוצגת ע"י תהליך אחד או יותר. זה יכול להיות סדרתי, עבודה מסוימת שמריצה מספר תהליכים ברצף בזה אחר זה. יש גם עבודות שיכולות להריץ מספר תהליכים במקביל. כמו שיש מספר סוגים של עבודות, גם יש מספר סוגים של תהליכים, למשל : (א) CPU-bound, עתירי עיבוד שרצים זמן רב על המעבד. (ב) עבודות I\O bound, שבעיקר מחכות לקלט-פלט ופחות דורשות את המעבד לטובת חישובים מורכבים.

<u>למזמנן (scheduler) יש 3 מזמנים פנימיים</u>:

(1) מזמנן לטווח ארוך – אחראי על זמנון עבודות (jobs). המזמנן 'מתעדף' (נותן עדיפות) איזו תוכנית תיטען לזכרון. בעצם מעביר את התהליך ממצב new למצב ready. קובע את המידה של 'ריבוב' (מלשון ריבוי) התוכניות, multi-programming, לצורך הגדלת ניצולת המעבד. כמובן שלא ניתן להוסיף עד אינסוף עבודות למעבד, כי אז יהיה עומס. מוטלת עליו המשימה להשיג איזון, להעמיס מספיק עבודות כדי לנצל את המעבד בניצולת גבוהה, אך לא יותר מדי (כי אז המזמנן לטווח בינוני יתעורר). זהו המזמנן הפחות עסוק, מתעורר כל מספר שניות / דקות.

(2) מזמנן חירום (אחראי על כיבוי שריפות ☺) – פועל כאשר אין זכרון פנוי במערכת, עומס על המעבד, ריבוי מוגזם של תהליכים, ירידה בזמן התגובה למשתמש. מתעורר ומנסה להקל על המערכת. עובד ברמה של עבודות ותהליכים. יכול להחליט להוריד עבודה / תהליך של עבודה מהמעבד. הוא מבצע בדיקה אם התהליך עומד לסיים בקרוב ואז מחכה לסיומו התקין. זמנן החירום מבצע השקטה / שחלוף לתהליכים שמפריעים ו'דוחק' אותם החוצה לדיסק (שילכו לישון ☺) כדי לשחרר את המשאבים שלהם. אם הוא יוריד יותר מדי עבודות, רמת 'ריבוב' התוכניות (multi-programming) תיפגע. כאשר מפסיקים פעולת תהליך (swap out) הוא 'יורד' מהמעבד. בהמשך ניתן להמשיך את פעולתו (swap in) של תהליך שהופסק. כביכול נוספו לנו 2 מצבים נוספים לאוטומט המצבים של התהליך : מוכן ומושהה (ready, suspend) ולא-פעיל ומושהה (blocked, suspend). כך בתום ההשהייה נוכל להחזיר את התהליך למצב ממנו השהו אותו – מוכן או פעיל. בהתאם, גם יש חיצי-מעבר נוספים באוטומט התהליכים (יהיה תרגול שלם על זה בקורס ביולוגיה חישובית ☺).

(3) מזמנן לטווח קצר – עובד רק ברמה של תהליכים. אחראי על רשימת התהליכים המוכנים (ready list) ומחליט איזה מהם יעבור למצב ריצה. חלק ממנו הוא המשלח (dispatcher), שמעביר את התהליך שבראש התור למצב-ריצה. עובד ברמה של מיקרו-שניות / מילי-שניות.

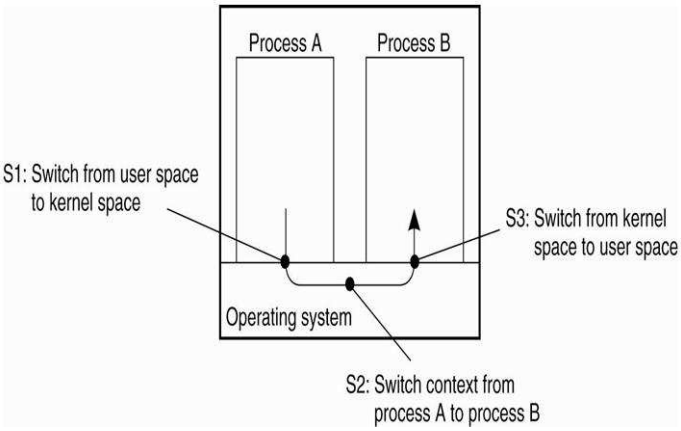


מימין, תרשים זרימה לסיכום נושא הזמנונים.
התהליכים עוברים בין התורים (queue) השונים (תור-ממתינים, תור-מוכנים) בהתאם לשינוי במצב שלהם (בירוקרטיה ☺). אלו תורי-עדיפויות ולא תורים רגילים. ע"פ ההנחה הפשטנית, כל תהליך מחכה רק למאורע יחיד ולכן ימתין רק בתור יחיד.

מזמנן החירום נקרא גם medium term.

<u>מיתוג (switching) תהליכים</u>
המנגנון להעברת השליטה (על המעבד) בין תהליכים השונים.
מיתוג תהליכים מתבצע בזמן פסיקה, כאשר מאבדים שליטה. בכל פעם שחוזרים למצב 'משגוח', קיימת האפשרות שמערכת ההפעלה תחליט לתת את השליטה במעבד לתוכנית אחרת. ה-switch הוא לא ישירות בין שני תהליכים, אלא הוא לוגי : תהליך א' הסתיים, העביר את השליטה לתהליך המערכת – שהוא זה שהעביר את השליטה לתהליך ב'. בין תהליך א' לתהליך ב' רצו מגוון תהליכי-מערכת, המעבר לא היה ישיר. חשוב לא לחשוב רק על ציר-המשתמשים אלא לקחת בחשבון גם את תהליכי המערכת המעורבים.

במקום לדבר על מיתוג תהליכים (process-switch), יותר נכון לדבר על מושג מכליל יותר : מיתוג-הקשר (context-switch), יחידות שניתן לזמנן ביניהן. כמו שניתן לזמנן בין תהליכים, ניתן לזמנן גם בין עבודות ותהליכונים (יפורט בהמשך). יש תקורה, המעבר בין יחידת זמנונית אחת לאחרת גוזל זמן ולכן רצוי להפחית שימוש במנגנון הזה. כמובן, לא ניתן לבטלו לחלוטין את השימוש במיתוג, כי רוצים לנצל את זמן המעבד בצורה המיטבית.



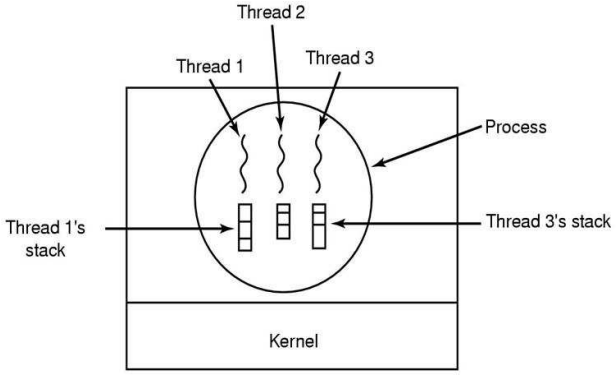
באיור מימין, מיתוג-הקשר בין שני תהליכים מורכב בעצם שני שינויים במצב המערכת: אחד בין תהליך א' לתהליך המערכת ושני בין תהליך המערכת לתהליך ב'. לכאורה רוב הפעולות של מערכת ההפעלה רצה על חשבון מיתוג-ההקשר, בין התהליכים א' ו-ב' שהיא דאגה למתג ביניהם. מערכת ההפעלה מתחזה להיות תהליך א' ומשתמשת במשאבים שלו, עד לרגע שהיא מעבירה את השליטה לתהליך ב'. הנקודות השחורות בתמונה מייצגות את שינוי מצב המערכת בין מצב 'משגוח' ומצב משתמש. במהלך מיתוג, יש שמירה של מצב התהליך לתוך ה-PCB שלו. חשוב לזכור: מיתוג ההקשר גורם ל-mode switch, שינוי במצב המערכת (בין 'משגוח' ומשתמש).

הרצאה מס' 7 - 30.11.09

[מצגת os3-3 pro]

בשיעור זה נמשיך לדבר על תהליכים.
תזכורת: המודל הקלאסי הוא המודל של התהליכים, לכל תהליך יש את המידע הרלוונטי. הפעולות העיקריות הן זמנון (באיזה סדר להריץ את התהליכים המוכנים-לריצה) ומיתוג (המנגנון הכרוך בהעברת השליטה בין התהליכים). החסרונות: תהליך והקשריו נקראים heavy weight process – HWP. תהליך הוא 'במשקל כבד' בהקשר שלו – כי כולל את כל המידע לגביו (מי האבא..) והצורך לתחל את כל הערכים מייקר את שלב הקמת התהליך. הצורך בשמירת ההקשר (כל המידע הנ"ל) דורש זמן רב וחבל. גם בכל פעם שמבצעים מיתוג, שוב נעשית שמירה של התהליך-היוצא והעלאת המידע של התהליך-הנכנס. ניקוי של מידע התהליך שהסתיים גם דורש זמן. היבט נוסף של כבדות, היא העובדה שלכל תהליך יש מרחב מיעון נפרד ואנו מעוניינים בשיתופיות בין תהליכים. היות ולאף תהליך אין גישה למרחב מיעון של תהליך-עמית, נצטרך להשתמש בחילופי-מסרים, או ששני התהליכים יהיו בסמיכות למרחב מיעון משותף. פעולות ההקמה של הזיכרון המשותף, החיבור והניתוק ממנו, בעלות עלות גבוהה.

לכאורה יש מודל נוסף, שאמור לבוא במקום המודל הקודם, כדי לפצות על החסרונות של המודל הקודם. היות ותהליך כבד, נפצל אותו: task – משימה, thread – תהליכון. החלק הבירוקרטי-סטטי יופרד מהחלק הדינאמי-ביצועי. החלק הסטטי ייקרא משימה, בעוד החלק הריצתי-זמנוני ייקרא תהליכון. מטרת התהליכון היא להתייחס למינימום הקשר: מרכיב המחסנית, מצב האוגרים. מרכיב התהליכון מכיל את המינימום הדרוש לגישה לתהליך. תהליכון מכונה low weight process – LWP, בעל 'משקל קל' (בניגוד לתהליך). סוג של רמאות, כי בעצם המשימה ירשה את רוב ה'משקל' של התהליך. אבל המשימה לא מייצגת את הצד הביצועי, כך שזה לא נורא. ניתן לכתוב את הנוסחה: process = task + thread. המודל הזה הוא מודולרי יותר, הפרדה לחלקים. היתרון: כל משימה יכולה להריץ מספר תהליכונים(!). נתייחס למשימה כעל 'אמא' שב'רחם' שלה מתרוצצים מספר תהליכונים. כך, כל עלות הקמה של תהליכון ומיתוגו, היא זולה, כי כל התהליכונים הם במרחב המיעון היחיד של המשימה (כולם באותו רחם ☺). ה'משקל הכבד' של המשימה לא בא לידי ביטוי בכל תהליכון, בניגוד למצב במודל של תהליכים. כאשר יש לנו עבודה סדרתית, פחות משנה אם יש לנו מספר תהליכים שרצים בזה-אחר-זה או שיש לנו משימה יחידה ומספר תהליכונים. אך אם מדובר בעבודה מקבילית, נעדיף כמובן במקום להקצות מספר תהליכים (כבדים), להקצות משימה יחידה שתריץ כחלק ממנה מספר תהליכונים. אם המכונה שלנו יכולה להריץ בכל רגע נתון n תהליכים ואילו אנחנו מריצים m תהליכים: אם m<n, כביכול אנחנו בסדר. אם m>n, כל הזמן נהיה עסוקים בטעינה של תהליכים ובהורדה שלהם, ע"י פעולת המיתוג, מה שיגזול זמן. לעומת זאת, במודל התהליכונים, נוכל להקים ולסיים תהליכונים בעלות זולה יחסית. בנוסף, העברת מסרים בין תהליכונים היא פשוטה יותר, שכן כל התהליכונים-בנים של אותה משימה-אמא היא טבעית, שהרי כולם באותו מרחב מיעון (רחם ☺) משותף. דגש: לא ניתן להקים משימה לבד, אנו מקימים משימה + תהליכון (כי שניהם ביחד שווי ערך ל-process מהמודל הקודם). כל משימה, ע"י התהליכון היחיד שאיתו היא נוצרת (תסביך אדיפוס? ☺) יכולה ליצור תהליכונים-בנים (אחים?? גילוי-עריות, תועבה ☺) נוספים. לא ניתן להסתפק במיתוג בין תהליכונים של אותה משימה, לפעמים נרצה לבצע מיתוג גם בין משימות – בין תהליכון של משימה א' לתהליכון של משימה ב'. במקרה כזה, עלות המיתוג שקולה כמובן לעלות המיתוג בין תהליכים: תחילה צריך למתג בין המשימות ורק אז בין התהליכונים. כמובן, נשתדל לבצע יותר מיתוגים בין תהליכונים של אותה משימה (כך באחוז גבוה נשלם את מחיר המיתוג הזול יותר) ופחות מיתוגים בין משימות (כך באחוז נמוך נשלם את מחיר המיתוג היקר יותר). קיים הבדל בין המידע השמור עבור כל תהליך (מידע רב) לעומת המידע השמור עבור כל תהליכון (מידע מועט יותר).

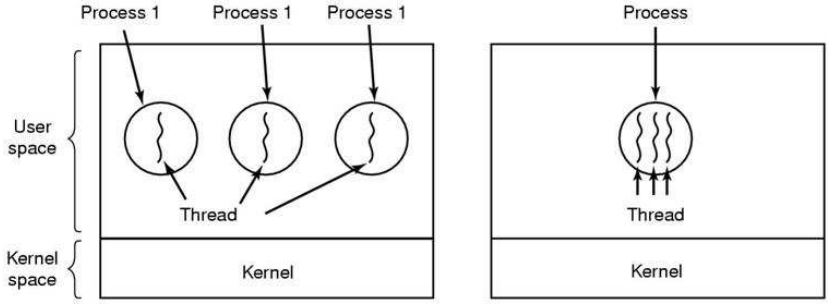


בתחילה יצרו ספרייה למשתמש, user level thread – ULT, שניהלה את כל המשחק של התהליכונים. הספרייה הזו רצה במסגרת התהליך הישן (סוג של מודל ביניים בין הקלאסי מהשיעור הקודם וזה הנוכחי). מאוחר יותר ביצעו את ההפרדה ע"י ה-kernel (נקרא KLT, יורחב בהמשך). האם התהליכונים הם אזרחים מדרגה ראשונה (☺)? אם כן, הם מוכרים בכל המערכת, וכל תהליכון מהווה יחידה זמנונית. במודל הביניים (ULT), בו התהליך היה זה ש'שיחק' עם התהליכונים, התהליכונים עצמם לא היו אזרחים מדרגה ראשונה, אלא היה משחק פנימי בין התהליכונים, במרחב המיעון של התהליך.

מבחינת התהליך, אם תהליכון ביצע קריאת מערכת, כביכול רק הוא ייחסם, האחרים ימשיכו לרוץ. אך מבחינת המערכת, היא חוסמת את כל התהליך, כי הוא זה שבפועל ביצע את קריאת המערכת – ובעצם שאר התהליכונים נחסמים גם הם. אך אם התהליכון הוא אזרח מדרגה ראשונה, מערכת ההפעלה יכולה להחליט לבצע יותר מיתוגי הקשר בין תהליכונים השייכים לאותה משימה, ופחות מיתוגי הקשר בין תהליכונים השייכים למשימות אחרות (פעולה יקרה יותר). בתוך משימה, לכל תהליכון (חוט / פתיל בעברית ☺, לציין שהוא קל-משקל) יש מחסנית (stack) אישית שלו. אפשר כמובן להשתמש במחסנית יחידה עבור כל התהליכונים במשימה, אך לדאוג לכך שכל תהליכון-ילד ישתמש בחלק אחר שלה.

במודל הישן (בצד שמאל), כל תהליך היה בעצם משימה ותהליכון יחיד.

במודל החדש (בצד ימין), שנקרא גם 'רב-תהליכון' (multi-threading) יש משימה יחידה שבתוכה רצים מספר תהליכים.

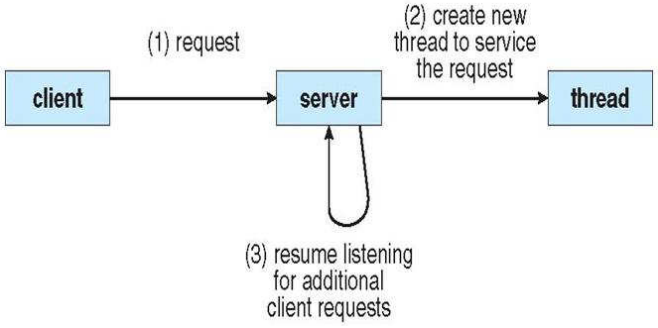


יישומים הבנויים מתהליכונים אמורים להיות מהירים יותר, בשל העובדה שמיתוג בין תהליכונים מתבצע בזמן קצר. כאשר משימה לא רוצה להתעכב בשל פעולה מסוימת (למשל בקשת קלט-פלט), היא מבקשת מתהליכון-בן שיבצע אותה. כך תהליכונים-בנים אחרים יכולים להמשיך לרוץ. רק התהליכון המסוים יתעכב. כאשר הפעולה (למשל קלט) תסתיים, אחד התהליכונים האחרים יקבל את המידע על כך. בעצם מי שמתפקד כ-interrupt handler הוא תהליכון. תזכורת: במודל הישן כל התהליך נחסם.

יתרון: היות וכל התהליכונים רצים באותו מרחב מיעון משותף, אין צורך בהקמה של זכרון משותף, הזכרון של המשימה הוא הזכרון המשותף. למעשה, אין לכל תהליכון מרחב מיעון אישי משלו.

דוגמאות ליתרונות מודל התהליכונים: מעבד תמלילים, שרת-אינטרנט. כל משימה מכילה באופן הגיוני 3 חלקים: קלט, עיבוד, פלט. לכן, מאד טבעי למדל משימה ע"י 3 תהליכונים הפועלים במקביל. dispatcher thread הוא התהליך האחראי בשם המשימה, התהליכון הראשוני, הוא זה שמייצר את ה'עובדים' (שאר התהליכונים). כמובן שאם יש 'אבטלה' ניתן לסגור תהליכון, ואם יש עודף כסף אפשר להוסיף תהליכון-עובד נוסף. לחילופין, אפשר להשתמש ברעיון של thread-pool, לפיו מחזיקים את מספר התהליכונים שניתן להחזיק, אך מספר התהליכונים שמפעילים בכל רגע נתון, הוא לפי הדרוש. דגש: התהליכון הראשוני חייב תמיד להיות קיים (ע"פ הנוסחה לפיה דרושים גם משימה וגם תהליכון כדי שנהיה שקולים לתהליך). ללא התהליך הראשוני, אין חיים למשימה.. קוד של תהליכון ראשוני יכול להיות: קבלת עבודה, שיגור עבודה לעובד. קוד של תהליכון-עובד יכול להיות: קבלת עבודה, ביצוע. כל יישום יכול להשתמש ב-cache (מטמון), שטח אחסון זמני, כדי להעביר מידע בין חלקים שונים, כדי לחסוך את הצורך בביצוע שמירות וטעינות יקרות. אפשר להתייחס לתהליכון הראשוני ככזה, כביכול המטמון של המשימה.

בכל פעם שיש בקשה נוספת לשירות, התהליכון הראשוני יוצר תהליכון-עובד נוסף. זהו מודל אפשרי, כי עלות הקמת תהליכון היא זולה יחסית (כמו שעלות ההעסקה של עבדי מטעים באפריקה היא זולה ☺). אפשר להתייחס לבקשה שמגיעה, כאל trigger שגורם להקמה (לידה ☺) של תהליכון חדש.

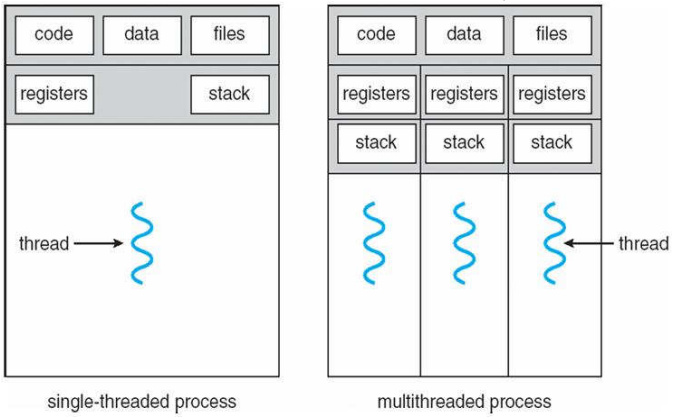


<u>סיכום יתרונות השימוש בתהליכונים</u>:

(א) עבודה מקבילית בתוך מרחב מיעון משותף.
(ב) עלות הקמה וסיום זולה ביחס לתהליך.
(ג) זמן מיתוג של תהליכונים לעומת תהליכים זול יותר.
(ד) תקשורת וסנכרון בין תהליכונים מהירה יותר מאשר בין תהליכים.
(ה) חסימה של תהליכון יחיד (למשל לצורך בקשת קלט-פלט) לא חוסמת את שאר התהליכונים, מה שמאפשר המשך עבודה של המשימה.

יתרון חשוב נוסף הוא scalability [בעברית: 'סילומיות', מלשון סולם]. מספר מימדים: (א) מימד גודל: אפשרות התרחבות – משימה יכולה להוסיף תהליכון-בן נוסף בכל שלב. ניתן להשוות זאת למערך שמוקצה דינאמית וניתן להגדילו כך שיכיל תאים נוספים. (ב) מימד גיאוגרפי: אם אלגוריתם עובד במקום מסוים, נרצה שיהיה מסוגל לפעול גם במקום אחר. למשל, אם פרוטוקול שיודע לעבוד ברשת תקשורת מקומית (LAN), ידע לעבוד גם ברשת תקשורת גלובלית כמו האינטרנט (WAN). (ג) מימד מנהלי: נניח שהרשת מחולקת לאיזורים שונים, שכל איזור נשלט ע"י פרוטוקול שונה, האם האלגוריתם יודע לעבור בין האזורים השונים. לכאורה יש סילומיות במודל התהליכונים. ניתן להתייחס לסילומיות כאל גמישות (במימד הגודל), שהרי משימה יכולה בקלות להקים מספר רב של תהליכונים ואילו קשה להקים מספר רב של תהליכים. משימה אמנם רצה על מעבד מסוים (יחיד), אך התהליכונים-בנים יכולים לרוץ על מעבדים שונים – ואז מנצלים את העובדה שהמחשב שלנו הוא רב-מעבד ☺.

כאשר מגיעה פסיקה, צריך לשמור את המידע של התהליכון שרץ באותו רגע, למקרה שנצטרך לעבור לתהליכון אחר באותה משימה, או לתהליכון אחר במשימה אחרת. לכל תהליכון ניתן להגדיר מחסנית-ביצוע, בנוסף לגישה למרחב המיעון של המשימה-אמא. ייתכן ויש מרחב מיעון (קטן) פרטי לכל תהליכון, אך בהכרח יש מרחב מיעון המשותף לכלל התהליכונים הרצים תחת אותה משימה. מוות של משימה, מחייב כמובן מוות של כל התהליכונים, שכן אין להם זכות קיום בפני עצמם. לתהליכון יש 3 מצבים: ריצה, מוכן לריצה, חסום. מצב מושהה לא קיים כ"כ, כי כל התהליכונים במשימה משתמשים באותו מרחב מיעון. כמובן, כאשר משהים משימה, בוודאי שכל אחד מהתהליכונים מושהה בפני עצמו..



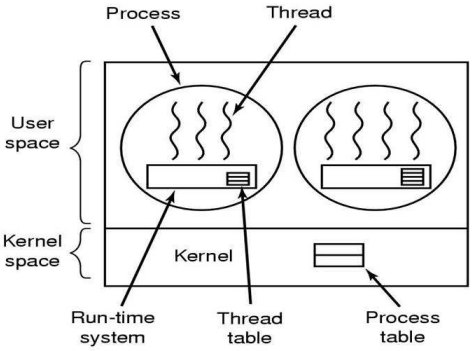
באיור משמאל המודל הקלאסי. מימין מודל התהליכונים, בו החלק של המשימה (code, data, files) משותף לכל התהליכונים, בעוד לכל אחד מהתהליכונים יש את החלק הפרטי שלו, הכולל את מצב האוגרים והמחסנית.

מערכות ההפעלה הראשונות היו במודל התהליכים, בהמשך פותחו מערכות הפעלה עם ריבוי-תהליכונים (התהליכונים קיבלו אזרחות ונהיו אזרחים מדרגה ראשונה ☺, בדומה לשחורים באמריקה שקיבלו זכות הצבעה).

[מצגת os3-4 pro]

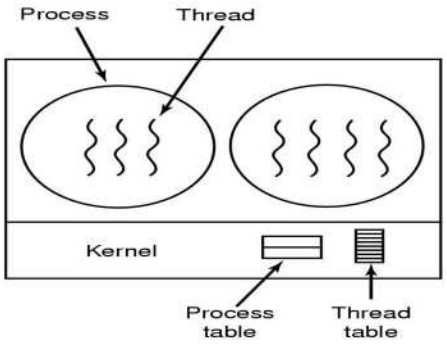
יישום תהליכונים במספר היבטים:

(1) בתחילה היה ULT (user-level thread), ספריה הקיימת רק ב-user space. בהמשך היה KLT (kernel-level thread), התהליכונים היו ברמת הגרעין של מערכת ההפעלה וקיבלו אזרחות מדרגה ראשונה. ספריה מכילה מידע אודות הקמת תהליכונים והעברת מסרים בין תהליכונים.



(א) הרחבה על ULT: תהליך רץ בגרעין, כל תהליכון רץ ברמת המשתמש. המיתוגים בין התהליכונים נעשו ע"י הלוגיקה של היישום עצמו שהייתה בנויה לתמיכה בכך. המשמעות היא שהמעבר בין התהליכונים נעשה ללא ידיעת / התערבות מערכת ההפעלה. באיור – מערכת ההפעלה מכירה רק בתהליכים ואינה יודעת ממה הם מורכבים בפועל (תהליכונים רבים).

ה-thread call הפרימיטיביים: יצירה, סיום והמתנה לתהליך. בעיות ברמת היישום: תהליכון אינו אזרח מדרגה ראשונה, לא נחשב כיחידה זמנונית. לכן כאשר מבצע קריאת מערכת, מערכת ההפעלה משביתה את כל המשימה על כל תהליכוניה.
היתרונות של ULT: ניתן להריץ בשיטה זו על כל מערכת הפעלה, רק זקוקים לספריה המכילה את הפקודות. חסרונות: תהליכון אינו יחידה זמנונית, לא ניתן להריץ תהליכונים שונים על מעבד שונים.



(ב) רעיון מתקדם יותר, KLT (kernel user thread): מערכת ההפעלה מכירה בתהליכונים, אחראית על המיתוג ביניהם. התהליכונים מהווים יחידה זמנונית. החסרון: כעת יש לנו שתי רמות של זמנונים – ברמת התהליכים (יקרה יותר) וברמת התהליכונים (זולה יותר). מערכות ההפעלה הנוכחיות תומכות ברעיון של KLT. בלינוקס ניתן ע"י הפקודה clone() לשתף מרחב מיעון של תהליכון עם משימת-האמא.

היתרונות של KLT: ניתן להריץ תהליכים של תהליכים שונים על מעבדים שונים. החסרון העיקרי הוא שכל מיתוג הקשר בין תהליכונים נעשה ע"י מערכת ההפעלה ולכן כרוך גם במיתוג מצב.

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

ישנם הבדלים בסדרי גודל של זמן (המדידות נעשו במיקרו-שניות) בין המודלים השונים, עבור הפעולות השונות. מפתיע לראות שדווקא המודל של ULT יעיל יותר מהמודל KLT, בו מערכת ההפעלה מודעת לתהליכונים ומבצעת מיתוג ביניהם.

(ג) מודל מורכב יותר: hybrid ULT\KLT. ניסיון למפות מספר תהליכונים ברמת המשתמש לתהליכון יחיד ברמת מערכת ההפעלה. כביכול לחסוך מיתוגי הקשר של מערכת ההפעלה כאשר מבצעים מיתוגים בין תהליכונים. ניתן לדמות זאת כאיחוד של מספר חוטים (תהליכונים ברמת המשתמש) לחבל עבה (תהליכון מערכת).

(2) מידול של ריבוי תהליכונים. (א) many-to-one – מערכת ההפעלה מריצה תהליך, שנחלק למספר תהליכונים ברמת המשתמש. בעצם מימוש של ULT. (ב) one-to-one – המודל הפשטני ביותר, כל תהליכון-מערכת שקול לתהליכון-משתמש, בעצם אין תהליכון אלא רק תהליכים. זהו בעצם המודל הקלאסי (משיעור שעבר), בו קיימים אך ורק תהליכים. (ג) many-to-many – תהליכים רבים ברמת המשתמש יכולים להיות ממופים למספר רב של תהליכים ברמת מערכת ההפעלה. זהו המודל הגמיש ביותר. (ד) two-level model – יש 3 רמות: user thread של המשתמש, רמת ביניים lightweight process ובנוסף גם את רמת המערכת, kernel thread. המודל הזה מורכב מדי, לא קיים כבר. (ה) תהליכונים בסביבת ג'אווה (כבוד ☺).

(3) תהיות לגבי תהליכונים: (א) האם סיגנל מגיע רק לתהליכון מסוים, או למשימה – ואז כיצד נקבע לאיזה תהליכון הוא מיועד? קיימות מגוון אפשרויות. הסיגנלים יכולים להיות ברמת החומרה או ברמת התוכנה. (ב) thread-specific data: האם לתהליכון יש שטח ונתונים פרטיים משלו או שהכל משותף, במרחב המיעון של המשימה? (ג) במקרה של מוות של תהליכונים – מי יורש תהליכונים במקרה של מוות בטרם סיימו את ריצתם? האם צריך ליידע את המשימה לפני מוות של תהליכון? (ד) צריך לתאם בין המזמנים השונים, לצורך מיתוג נכון בין התהליכונים. מורכבות היישום והשימוש גורמים לכך שהפסיקו להשתמש במודלים מורכבים מדי.

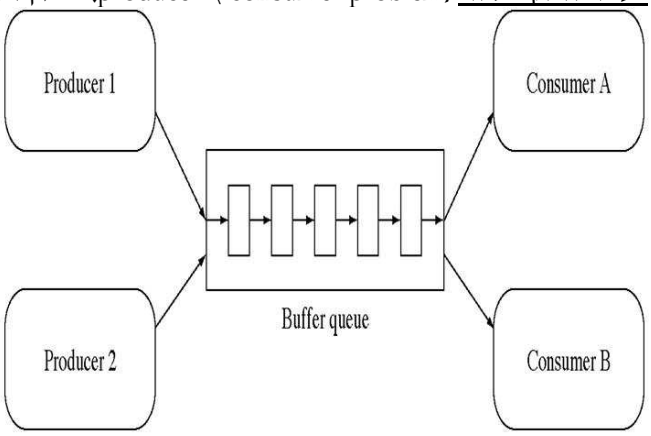
[מצגת os4-1 cop, הגענו עד שקופית 22 כולל]

<u>סנכרון בין תהליכים</u>

עד כה דיברנו על רמת התהליך הבודד ולא על ה-IPC (inter-process communication, תקשורת בין תהליכים). ישנם תהליכים עצמאיים, שלא מודעים לכך שרצים תהליכים אחרים במערכת ואין ביניהם קשר ישיר. לכן לא נעסוק בהם. אנחנו נעסוק בתהליכים השייכים לפעולה מקבילית, המחייבת תיאום ביניהם. תהליכים כאלו משפיעים ומושפעים מהתהליכים האחרים השייכים לאותה פעולה. תקשורת תקינה ביניהם יכולה לזרז את הפעולה. בעיה: תהליכים השייכים לאותה משימה [הערה של המרצה: משימה במובן דו-משמעי, ניתן להתייחס גם למשימה כ-task וגם לפעולה במובן רחב יותר] יכולים לסבול ממצב הנקרא race condition, מצב בו שניהם מנסים לשנות מידע השמור באותו מקום, ולכן צריך להגביל פעולות מסוימות כך שיתרחשו בכל רגע נתון רק ע"י תהליך יחיד (למשל כתיבה לקובץ משותף). אחרת תהליך המבצע קריאה (מהקובץ) עלול לקרוא חלק מהמידע שהיה קיים קודם לכן שאולי כבר לא מעודכן לאחר כתיבה של המידע ה'חדש'. מצב זה נקרא inconsistent data, מידע לא עקבי. לעומת זאת אם מדובר בפעולות קריאה, ניתן לבצע אותן במקביל. קטע קריטי (critical section) נועד לשמור על היושרה / אמינות של הנתונים. [למדנו על זה בהרצאה של תכנות מתקדם 1: מדובר בקטע קוד שתהליך יחיד יכול להריץ אותו ברגע נתון.] תקשורת בין תהליכים יכולה להיות מיושמת ע"י שליחת מסרים, אך יש לה חסרונות: (א) deadlock (קפאון) הוא מצב בו שני תהליכים ממתינים כל אחד למסר מהתהליך השני ובעצם שניהם עלולים להמתין לנצח. (ב) starvation (הרעבה) הוא מצב בו תהליך מחכה להודעה שלא מגיעה (לא מקבל אוכל ונהיה רעב ☺).

<u>בעיית המקביליות</u> (producer \ consumer problem): יצרן או יותר, אחראים לייצר פריטים, הנצרכים ע"י צרכן אחד או יותר.

למשל: היצרן מייצר תווים הנשלחים להדפסה ואילו הצרכן הוא המדפסת שמדפיסה את התווים. למה זו דוגמה לבעיה? כי אחד תלוי בשני, המדפסת לא יכולה להדפיס, עד שלא יתנו לה תווים להדפסה. תיאורטית, ה-buffer [בעברית: מכלא] המקשר בין היצרן והצרכן יכול להיות unbounded (לא מוגבל במקום), אך בפועל הוא bounded, חסום לגודל מסוים. היות והמכלא בעל גודל סופי, נרצה לבצע חסימה ליצרן, כך שהיצרן לא ינסה לכתוב מידע נוסף למכלא. לעומת זאת, אם המכלא ריק, צריך לחסום את הצרכן כי אין לו מידע לקרוא מהמכלא.



דוגמה לפתרון לבעיית היצרן-צרכן: מימוש מכלא בעל גודל סופי במערך מעגלי (לחובבי מבני-נתונים, מזכיר cyclic queue), ע"י 2 מצביעים: in יסמן את המקום הפנוי הראשון (שאליו היצרן יכול לכתוב), בעוד out יסמן את המקום התפוס האחרון (שממנו הצרכן יכול לקרוא). שאלה: אם in ו-out נמצאים על אותו אינדקס, איך נדע אם המכלא ריק / מלא לגמרי? תשובה: ניתן להוסיף ערך של מונה (נקרא לו count), שיגיד כמה מידע נמצא במכלא. בכל פעם שהיצרן יכתוב יחידת-מידע למכלא הוא יבצע count++, בעוד בכל פעם שהצרכן יקרא מידע מהמכלא הוא יבצע פעולת count--. בעיה: הפתרון הוא בעצם דוגמה למידע לא עקבי, מצב בו פעולות חישוב התלויות זו בזו מבוצעות במקביל ולא באופן סדרתי. כדי לפתור זאת, נבצע את הפעולות הנ"ל כפעולות אטומיות, אי-פריקות (שלא ניתנות לפירוק). שהרי אם לא כך, אם היצרן מקדם את count ואילו הצרכן מפחית את count, עלול להיווצר מצב של התנגשות, במקרה כזה המידע לא יהיה עקבי.

הרצאה מס' 8 - 7.12.09

[מצגת os4-1 cop, משקופית 18]

תזכורת משיעור שעבר: כאשר היצרן צריך לכתוב ל-buffer, עבור צרכן שקורא ממנו, יש חשש ששניהם ינסו לבצע פעולה לשינוי הערך של count בו-זמנית. דבר זה יגרום להתנגשות, כי אחד מהם ינסה להגדיל את הערך של count והשני ינסה להקטינו (מצב זה נקרא race condition). כתוצאה מכך עלול ערך ה-count להיות לא נכון.

Critical section problem – מדובר בקטע קוד קריטי, שמותר לתהליך יחיד להריץ אותו. בדוגמה משיעור שעבר, נגדיר שרק תהליך יחיד בכל רגע נתון יכול לשנות את הערך של המשתנה count, כדי למנוע התנגשות. נרצה להיות מסוגלים למנוע מהצרכן לקרוא תוך-כדי שהיצרן כותב וכן למנוע מהצרכן לקרוא יותר ממה שהיצרן כתב.

תזכורת משיעור שעבר: אם אנחנו רוצים לנצל כל תא ב'מכלא', נוסיף למשתנים in ו-out, מונה שייקרא count. לולא המונה, נעזרנו בבדיקה אם האינדקסים in ו-out סמוכים, כמהווה סימן שהמכלא ריק לחלוטין או מלא לחלוטין. בעזרת count נוכל להשתמש גם בתא הפנוי האחרון. באופן כזה, כאשר היצרן רוצה לכתוב, כל שעליו לבדוק הוא אם מספר הפריטים במכלא שווה ל-count (ולא את מיקום האינדקסים והאם הם צמודים זה לזה). חסרון לשיטה זו בה הצרכן והיצרן בודקים בכל ניסיון לעבור את הולואה (לצורך קריאה / כתיבה) את ערך ה-count, הוא שמצב זה מהווה busy-waiting, בדיקות חסרות-ערך שגוזלות זמן ריצה רב. [חשוב להבין: גם המשתנה count וגם המכלא, משותפים לכל תהליכי היצרנים ולכל תהליכי הצרכנים – הרי כל אחד מהם צריך גישה אליו כדי לכתוב / לקרוא.] פתרון חלקי לבעיה, הוא ע"י החומרה, בכך שהזכרון לא מאפשר ליותר מתהליך יחיד לגשת למען ספציפי בו. אבל כמובן, אסור לנו להניח שקיים הפתרון הזה. (מתכנת טוב צריך לחשוב על כל מקרי הקצה בעצמו ☺)

הפתרון יהיה להגדיר בשפת המכונה את הפעולות של קידום / הקטנת ערך המשתנה count כ'פעולות אי-פריקות' (אטומיות), כך שהן יהיו חייבות להתבצע עד תום, לפני שפקודה אחרת תוכל להתבצע. דגש חשוב: פסיקות (interrupts) יכולות להתבצע **בין** פקודות ולכן הגדרת פעולות אלו כפקודות אטומיות מונעות מאיתנו מצב בו יש פסיקה שתשבש לנו את הפעולה. ההנחה היא שלא ניתן לשנות ערך של משתנה בזכרון, אלא צריך תחילה לטעון אותו לאחד הרגיסטרים ורק אז ניתן לשנות את ערכו (כמובן שבסיום צריך לשמור אותו בחזרה למען בזכרון). אם נגדיר פעולות שינוי ערך ה-count כפעולות אטומיות, אין חשש שניסיון לקידום / הפחתת הערך של count יתבצעו בו-זמנית (למשל כאשר היצרן טוען אותו לרגיסטר ובטרם הספיק לשנותו, הצרכן מנסה לשנותו). זאת בזכות מיתוג-ההקשר (context switch) שמאפשר לתהליך יחיד לגשת לרגיסטרים בכל רגע נתון. לכן, מרגע שהצרכן טען את ערך המונה count לרגיסטר, עד שישנה את ערכו וישמור את הערך החדש למשתנה המתאים בזכרון (ובכך תסתיים הפעולה האטומית), היצרן לא יוכל להשתמש ברגיסטר, מפני שלא יתבצע מיתוג-הקשר שיעביר לידיו את השליטה על הרגיסטר.

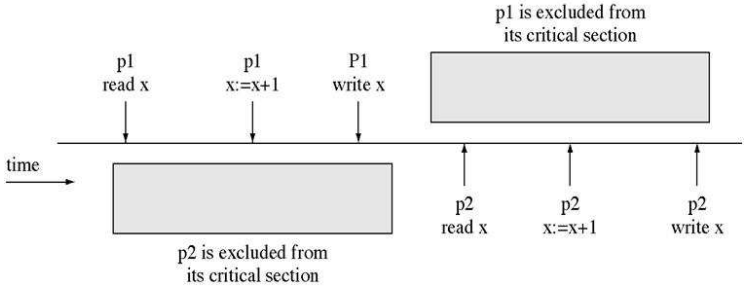
תרחיש להמחשת הבעיה (בטרם הגדרנו את פעולת שינוי ערך ה-count כפעולה אטומית): נניח שבשני יצרנים ישנה לולאה, כל אחד מהם מנסה להריץ לולאה של 10 קידומים לערך count. אם הערך התחילי של count הוא 0, טווח הערכים האפשרי ל-count בסיום הריצה של שני התהליכים הוא 2-20 (מפתיע, רוב הכיתה חשבה 0-20 או 10-20). **הסבר** [חשוב(!) להבין את התרחיש הזה]:

(א) כדי שבסיום הערך של count יהיה 10: אחד היצרנים יטען את הערך של count לרגיסטר (בעת הטעינה, ערכו 0 כאמור). היצרן השני יצליח לבצע 10 קידומים למשתנה count (כביכול מבחינתו הערך הוא 10). לאחר מכן היצרן השני יקדם את ערך ה-count ברגיסטר (מבחינתו הערך הוא 0, כפי שהיה שטען אותו) ואז יגדיל 10 פעמים.

(ב) כדי שבסיום הערך של count יהיה 2: אחד היצרנים יטען את הערך של count לרגיסטר. היצרן השני יקדם ברגיסטר 9 פעמים את ערך ה-count. כעת היצרן הראשון יקדם את הערך ובעצם ה-count יקבל ערך של 1 בלבד. כעת היצרן השני יטען את count לרגיסטר (כאשר ערכו 1). היצרן הראשון ימשיך את 9 הקידומים הבאים של הלולאה וכביכול יגדיל ל-10, אך היצרן השני ביצע טעינה לרגיסטר מיד כאשר ה-count היה רק 1 ולכן בסיבוב האחרון של הלולאה שלו הוא יגדיל את count ל-2.

(ג) כדי שבסיום הערך של count יהיה 20: האפשרות האינטואיטיבית, כל אחד מצליח לבצע את כל 10 הקידומים ללא הפרעה.

בציור מימין המחשת הפתרון: נגדיר את פעולת קידום המשתנה המשותף כפעולה אטומית וכ'קטע קוד קריטי', כך שמרגע שאחד התהליכים התחיל לבצע את הקוד הזה, לא תוכל להתרחש פסיקה עד תום ביצוע הקוד הזה. כך הצלחנו למנוע מצב מסוכן של race condition.



דגש חשוב: ע"י כך שאנחנו מאפשרים multi-threading, אנחנו בעצם מאפשרים למקביליות לוגית (גם אם במחשב שלנו יש מעבד יחיד), אך כתוצאה מכך מאבדים נכונות, כאשר מדובר במצב של race condition. כאשר אנחנו מגדירים קטע בקוד כקטע קריטי, אנחנו אומנם מבטלים את האפשרות של המקביליות הלוגית (בתוך הקטע עצמו), אך מרוויחים את נכונות הקוד.

[מצגת os4-2 cop]

<u>פורמליזציה לבעיית קטע הקוד הקריטי (Critical section, CS)</u>: במצב בו מספר תהליכים מנסים להשתמש במשתנים משותפים, נגדיר את קטע הקוד שניגש למשתנים הללו כקוד קריטי (ע"י הוספת קטע כניסה entry section לפניו וקטע יציאה leave section אחריו) ובכך נבטיח שרק תהליך יחיד יכול לבצע את קטע הקוד הזה בכל רגע נתון.

```
do {
    entry section
        critical section
    leave section
        remainder section
} while (TRUE);
```

ייתכן מצב בו כל תהליך מנסה לגשת למספר משתנים משותפים, כך שכל תהליך יכיל מספר קטעי קוד קריטיים. כמובן שבמצב כזה נדאג לבצע סוג של תיאום, כך שאם למשל תהליך אחד ניגש למשתנה המשותף k ותהליך שני ניגש למשתנה המשותף m, הדבר יהיה אפשרי, אך לא ייתכן מצב בו שני התהליכים מנסים לגשת למשתנה משותף זהה. אם תהליך מסוים מנסה לגשת למשתנה משותף יותר מפעם אחת במהלך הקוד שלו, עם זה אין לנו בעיה, שכן מדובר בתהליך יחיד.

פתרון נכון לבעיית קטע הקוד הקריטי צריך לקיים 3 תנאים (ב-100% מהמקרים):

(א) הדרה הדדית (מלשון 'להדיר רגליים') mutual exclusion: אם תהליך אחד נמצא בקטע הקריטי, תהליכים אחרים לא יוכלו להיות בו.

(ב) התקדמות progress: אם רק אחד התהליכים רוצה להיכנס לקטע הקוד הקריטי, יש לאפשר לו. אם יותר מתהליך אחד רוצה להיכנס, לאחד מהם חייבים לאפשר כניסה, כדי לאפשר 'התקדמות' של הקוד (אחרת יהיה מצב שנקרא dead lock, אף אחד לא נכנס לקטע הקריטי והתהליכים למעשה 'תקועים'). נקרא גם תנאי התקדמות גלובלית.

(ג) מניעת מצב של הרעבה bounded waiting: המתנה חסומה, מרגע שתהליך נכנס לקטע הכניסה לקוד הקריטי, זמן ההמתנה שלו יהיה חסום ומתישהו הוא בטוח ייכנס. נקרא גם תנאי התקדמות לוקלית, דאגה שכל תהליך שרוצה יצליח להתקדם בקטע הקוד הקריטי. לא ניתן לדעת מראש מה מהירות המחשב עליו מריצים את הקוד ולכן לא ניתן להחליט שנחסום תהליך הממתין להיכנס למספר שניות מסוים. לכן, נחסום במספר הכניסות לקטע הקוד הקריטי: לדוגמה אם

תהליך א' נכנס לקטע הקוד הקריטי ועכשיו תהליך ב' מעוניין להיכנס – נבצע חסימה כך שלאחר מספר מסוים של כניסות של תהליך א' – תהיה חייבת להיות כניסה של תהליך ב' (אחרת הוא יהיה 'מורעב').
דגש: קטע הקוד הקריטי צריך להיות קצר וממוקד, כדי להבטיח שהחסימה של שאר התהליכים תהיה מינימלית (כמובן לדאוג שהקטע הקריטי לא מכיל לולאה אינסופית..). מרגע שתהליך נכנס לתוך קטע קוד קריטי, אסור לו להתעכב, עליו לצאת משם מהר ככל האפשר – כל עיכוב מעכב את התהליכים האחרים שרוצים להיכנס.

נראה 3 דוגמאות שגויות כפתרונות (ברמת התוכנה, קוד) לבעיית קטע הקוד הקריטי. הדוגמה ה-4 שנראה תהווה פתרון נכון.
בדוגמאות שלנו נניח שיש רק 2 תהליכים (נקרא להם larry ו-jim) הפועלים במקביל. בנוסף, נגדיר משתנה משותף מסוג מחרוזת שייקרא turn, שיציין של מי התור להיכנס ולבצע את קטע הקוד הקריטי. בכל הדוגמאות נראה רק את הקוד של אחד התהליכים (larry, כי יש לו שם יפה ☺). כמובן, קטע הקוד של jim הוא סימטרי. נבדוק נכונות של כל פתרון ע"פ הקריטריונים שהגדרנו.

(1)

```
do {
  while (turn != "Larry");
    critical section
  turn = "Jim";
    remainder section
} while (TRUE);
```

כל עוד זה לא התור של larry, הוא יבצע busy-waiting (לכן בסוף שורת לולאת ה-while הפנימית יש נקודה-פסיק). לאחר ביצוע קטע הקוד הקריטי, larry (הנחמד ☺) מעביר את התור ל-jim. בהתאמה, לאחר ש-jim יבצע את הקוד הקריטי, הוא יעביר את התור של larry.

תנאי א' (הדרה הדדית): נטען בדרך השלילה ששני התהליכים נמצאים בתוך הקטע הקריטי, נגלה סתירה וכך נראה שהתנאי מתקיים. נניח שגם larry וגם jim נמצאים בקוד הקריטי. כדי ש-larry יהיה בתוך הקטע הקריטי, ערך המשתנה turn צריך להיות עם שמו, אך שינוי ערך המשתנה turn (והדגש כאן כאמור הוא שמדובר במשתנה המשותף לשני התהליכים!) משתנה רק לאחר שאחד התהליכים יוצא מהקטע הקריטי. לכן, כדי ש-jim יכנס לקטע הקריטי, תחילה צריך larry לצאת מהקטע הקריטי בעצמו ולהעביר 'בעלות' ל-jim. לכן תנאי א' מתקיים.
תנאי ב' (התקדמות): נבדוק מה קורה כאשר רק אחד מהתהליכים רוצה להיכנס לקוד הקריטי ומה קורה כאשר שניהם רוצים להיכנס בו-זמנית. אם שניהם ירצו להיכנס, אחד מהם בוודאות ייכנס, כי ערך המשתנה turn יכול להיות (וחייב להיות) larry או jim. (לא אכפת לנו שרק אחד מהם יוכל להיכנס כי תנאי א' מתקיים, אנחנו כרגע מתעסקים רק בתנאי ב', לצורך בדיקת תנאי זה לא מפריע לנו אם שני התהליכים יוכלו להיכנס). אם רק אחד מהם ירצה להתקדם, אין התקדמות. שהרי אם ערך המשתנה turn הוא larry (שהלך לישון ע"י פקודת sleep) ואילו jim דווקא רוצה להיכנס – הוא אינו יכול. לכן תנאי ב' בעצם לא מתקיים.
תנאי ג' (המתנה חסומה): יש המתנה חסומה והחסם הוא 1, כי ברגע שאחד התהליכים יצא מהקטע הקריטי, הוא מעביר את התור לתהליך השני. לכן, לא ייתכן מצב בו תהליך מסוים יצליח להיכנס יותר מפעם אחת ברציפות לקטע הקריטי (ולחילופין, לא ייתכן מצב בו אחד התהליכים ימתין להיכנס, בעוד תהליך אחר נכנס יותר מפעם אחת לקטע הקריטי).
לסיכום: תנאים א' ו-ג' מתקיימים אך תנאי ב' לא מתקיים ולכן זהו אינו פתרון לבעיה הקטע הקריטי.

(2)

```
do {
while (flag-jim);
  flag-larry = TRUE;
    critical section
  flag-larry = FALSE;
    remainder section
} while (TRUE);
```

שימוש בשני משתנים (דגלים) בוליאנים, אחד עבור כל תהליך, כך ששניהם מאותחלים לערך false. כל עוד jim הכריז שהוא רוצה להיכנס לקטע הקריטי, larry ימתין. לאחר שנכנס לקטע הקריטי, larry 'ירים' את הדגל שלו ולאחר שיצא 'יוריד' את הדגל שלו.

תנאי א' (הדרה הדדית): אין הדרה הדדית, נראה דוגמה נגדית. היות ושני הדגלים מאותחלים ל-false, שני התהליכים יוכלו בפעם הראשונה להיכנס (כי אף תהליך לא 'הכריז' שהוא רוצה להיכנס לקטע הקריטי). לכן התנאי לא מתקיים.
תנאי ב' (התקדמות גלובלית): מתקיים, כי שני הדגלים מאותחלים ל-false. הראשון שיגיע יוכל להיכנס לקטע הקריטי ו'להרים' את הדגל שלו. ייתכן ושניהם יוכלו להיכנס, אם שניהם ייכנסו מספיק מהר (אחד ייכנס ו'ירים' את דגלו, תהיה פסיקה שתעביר את השליטה לתהליך השני כך שגם הוא ייכנס.)
תנאי ג' (המתנה חסומה): לא מתקיים, אין חסם. נניח שבאחת הפעמים ש-larry נכנס ו'הרים' את הדגל, הייתה פסיקה שהעבירה את השליטה לתהליך jim. היות וכדי שהתהליך jim יכול להיכנס רק כאשר הדגל של larry 'למטה', הוא לעולם לא ייכנס.
לסיכום: תנאי ב' מתקיים אך תנאים א' ו-ג' לא מתקיימים.

(3)

```
do {
flag-larry = TRUE;
while (flag-jim);
  critical section
flag-larry = FALSE;
  remainder section
} while (TRUE);
```

שימוש בשני דגלים כמו קודם (מאותחלים ל-false). השינוי הוא שמיד בתחילת הלולאה (החיצונית, הגדולה) התהליך שרץ 'מרים' את הדגל שלו.

תנאי א' (הדרה הדדית): אם שני התהליכים נמצאים בתוך הקטע הקריטי, שניהם נכנסו כי כל אחד אומר שהדגל של השני היה false, אך בעצם כדי שתהליך ייכנס הדגל האישי שלו חייב להיות true. קיבלנו סתירה ולכן יש הדרה הדדית.
תנאי ב' (התקדמות): אם אחד רוצה להיכנס והשני לא, הוא 'מרים' את הדגל ונכנס. אם שניהם רוצים להיכנס, שניהם 'מרימים' את הדגל ושניהם לא יוכלו להיכנס. לכן אין התקדמות (כאשר אחד 'מרים' את הדגל, השני לא יכול להיכנס).

תנאי ג' (המתנה חסומה): יש חסם של 1. טיעון: נניח ש-jim רוצה להיכנס ומדליק את הדגל. לאחר ש-larry נכנס לקטע הקריטי ויצא, הוא 'מוריד' ו'מרים' את הדגל שלו. בזמן שבין הכיבוי וההדלקה של larry, התהליך jim יכול להיכנס.
לסיכום: תנאים א' ו- ג' מתקיימים ותנאי ב' לא.

(4)

```
do {
  flag-larry = TRUE;
  turn = "Jim";
  while (flag-jim and turn == "Jim");
    critical section
  flag-larry = FALSE;
    remainder section
} while (TRUE);
```

פתרון נכון: נשתמש גם בדגל שמציין של מי ה-turn וגם בדגל של כל אחד מהתהליכים (שילוב המשתנים המשותפים מהפתרונות הנ"ל). כאשר תהליך רוצה להיכנס, הוא 'מרים' את הדגל שלו. תהליך לא יוכל להיכנס, כל עוד turn קובע כי התור שייך לתהליך השני וגם התהליך השני רוצה להיכנס.

תנאי א' (הדרה הדדית): נטען בדרך השלילה ששני התהליכים נמצאים בקטע הקריטי. היות ובתנאי של לולאת ה-while הפנימית אנחנו בודקים שני תנאים בעצם, נצטרך לבצע בדיקה עבור כל אחד מהם. טיעון ראשון: larry בפנים כי המשתנה turn לא אמר שזה התור של jim. מה שאומר ש-jim עדיין לא נכנס לקטע הקריטי, כי הדגל שלו עדיין 'כבוי'. עכשיו נשאל מדוע jim בקטע הקריטי. הוא יטען אותו דבר – שהדגל של larry 'כבוי'. זו כמובן סתירה, כי הרי אמרנו ש-larry בפנים ובהכרח הדגל שלו 'מורם'. טיעון שני: larry נכנס כי הדגל של jim 'כבוי'. אז איך jim בפנים, למרות שהדגל של larry 'מורם'? כי התור היה jim. אבל אז בעצם jim לא יכול היה לעבור את לולאת ה-while, כי אחרת הדגל שלו היה 'מורם', אך במקרה כזה larry לא יכול היה להיכנס כי מתקיים שגם הדגל של jim למעלה וגם turn קובע שזה התור של jim. אבל אם larry נכנס, אז הדגל שלו למעלה. שוב, קיבלנו סתירה ולכן יש הדרה הדדית.
תנאי ב' (התקדמות): אם רק אחד התהליכים רוצה להתקדם, צריך שהדגל של האחר יהיה 'כבוי' וללא קשר למשתנה turn הוא יוכל להיכנס. אם שני התהליכים רוצים להתקדם, אחד בוודאות יוכל. גם אם הדגלים של שניהם 'דלוקים', המשתנה turn קובע שאחד מהם בוודאות יצליח. לכן תנאי זה מתקיים.
תנאי ג' (חסימה): בוודאות אחד לפחות יכול להיכנס לקטע הקריטי. כל אחד שרוצה להיכנס, בודק תחילה אם המשתנה turn העניק את התור לתהליך השני.
לסיכום: תנאים א', ב' ו-ג' מתקיימים ולכן הקוד הנ"ל פותר נכונה את בעיית הקטע הקריטי.

(5)

```
do {
  turn = "Jim";
  flag-larry = TRUE;
  while (flag-jim and turn == "Jim");
    critical section
  flag-larry = FALSE;
    remainder section
} while (TRUE);
```

נראה כיצד החלפת הסדר של שתי השורות שלפני לולאת ה-while הפנימית פוגמות בפתרון: קודם נעביר את turn לתהליך השני ורק אח"כ 'נרים' את הדגל שלנו.

במצב כזה אין קיום של תנאי א' (הדרה הדדית). נראה זאת ע"י דוגמה נגדית: larry יבצע את שינוי הערך של המשתנה turn כך שיציין שזהו התור של jim ובשלב זה תהיה פסיקה והשליטה תעבור ל-jim. jim ישנה את ערך המשתנה turn בעצמו, כך שכביכול הוא מעביר את התור ל-larry ובנוסף הוא 'מדליק' את הדגל שלו. כעת jim יכול להיכנס, שהרי הדגל של larry 'כבוי'. בשלב זה שוב תהיה פסיקה ונחזור ל-larry. כעת larry 'מדליק' את הדגל של עצמו והוא יכול להיכנס, כי המשתנה turn מאפשר לו. כתוצאה מכך, שני התהליכים הצליחו להיכנס לקטע הקריטי ☹. [ותודה לאדון הלל הזקן שהסביר לי את מה שלא הספקתי לכתוב בשיעור. שיזכה לאריכות ימים ולשידוד במהרה בימינו ☺]

<u>אלגוריתם המאפייה Bakery algorithm</u>: פתרון לבעיית הקטע הקריטי עבור n תהליכים.
נשתמש במכונה שמנפקת מספרים, כך שבכל שלב המספר שמנופק אינו קטן מאף אחד מהקודמים. אם שני תהליכים קיבלו את אותו המספר, נבדוק לפי המזהה הייחודי שלהם וכך נקבע שאחד מהם (בעל המזהה הנמוך יותר) יפעל לפני השני. שימוש בפונקציה max יבטיח שניקח את מספר הכרטיס המקסימלי, כדי שלא ננפק מספר קטן מהקודם. נשתמש במערך בוליאני של דגלים, באורך n שכל דגל בו יאותחל ל'שקר'. בנוסף יהיה לנו מערך של int עבור מספרי הכרטיסים שיאותחל ל-0, כי הגדרנו שכל הכרטיסים מקבלים מספרים חיוביים (וכך נוכל לדעת אם לתהליך מסוים אין כרגע מספר חוקי, מה שיעיד שהוא לא מעוניין להיכנס).

```
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 (TRUE);
```

קטע הכניסה: משנים את הדגל ל-true, מה שיעיד שהתהליך רוצה להיכנס. כעת בוחרים מספר עבור הכרטיס ו'מורידים' את הדגל. נרוץ על כל מערך הכרטיסים ונבצע בדיקה: אם אחד התהליכים בעל דגל 'מורם' – נאפשר לו לרוץ. לולאה נוספת – אם עבור תהליך מסוים הדגל שלו 'כבוי', נבצע בדיקה למי מאיתנו מגיע להיכנס קודם, ע"פ מספר הכרטיס והמזהה הייחודי שלנו. נוכל להיכנס לקטע הקריטי רק אם ניצחנו את כל התהליכים האחרים ע"פ מספר הכרטיס.

כעת נבדוק את קיום התנאים שהגדרנו:
תנאי א' (הדרה הדדית): כדי שתהליך ייכנס, תחילה הוא בדק שניצח לפי מספר הכרטיס את כל התהליכים האחרים. לא ייתכן שיותר מתהליך אחד ינצח את כל האחרים (בהנפקה מבצעים max+1). כאמור, מספר הכרטיס מתאפס ביציאה מהקטע הקריטי.
תנאי ב' (התקדמות): יש התקדמות, שהרי בוודאות יש 'מנצח' אחד שיוכל להיכנס.
תנאי ג' (המתנה חסומה): יש חסם והוא n-1. המקרה הגרוע הוא שכל n-1 התהליך האחרים יבחרו כרטיס לפניי. אבל בתחרות הבאה, אני בחרתי מספר לפניהם ולכן אני אכנס לפניהם.
לסיכום: כל התנאים מתקיימים ולכן מדובר בפתרון לבעיית הקטע הקריטי.

הבנה עצמית לשיעור הבא: מדוע באלגוריתם המאפייה, לפני הכניסה לקטע הקריטי אנחנו 'מרימים' את הדגל ואז 'מורידים' אותו.

הרצאה מס' 9 - 14.12.09

[מצגת os4-2 cop, משיעור קודם]
תזכורת משיעור קודם: קוד כניסה לקטע קריטי צריך לעמוד ב-3 תנאים כדי להיות תקין. התנאים: (א) הדרה הדדית – לא ייתכן ששני תהליכים יהיו בו-זמנית בתוך הקטע הקריטי. (ב) התקדמות גלובלית – לפחות אחד מהתהליכים יכול להיכנס בכל רגע נתון, אחרת נוצר מצב של קיפאון. (ג) המתנה חסומה (התקדמות לוקלית) – מניעת הרעבה, מרגע שתהליך מבקש להיכנס לקטע הקריטי, הזמן שהוא ממתין עד לכניסתו תלוי במספר כניסות מסוים של שאר התהליכים.
חזרה על אלגוריתם האפייה, המוצע כקוד כניסה לקטע קריטי, במקרה בו יש n תהליכים: מדובר במכונה פשוטה, מגרילה מספר לכל תהליך המעוניין להיכנס. תהליך מכריז על עצמו שרוצה להיכנס ('מרים' את הדגל האישי שלו) ובודק בלולאה אם אין מספר אחר (של תהליך אחר) שיותר נמוך משלו ואם אין תהליך אחר שכבר הכריז על עצמו שרוצה להיכנס לקטע הקריטי. [הוספנו בדיקה למקרה קצה בו לשני תהליכים יש מספר מוגרל זהה, כך שייכנס התהליך בעל המספר המזהה הייחודי הקטן יותר.] ביציאה מהקטע הקריטי, התהליך מאפס את מספרו, כדי לא 'להתחרות' מחדש על כניסה לקטע הקריטי עד ש'ידליק' מחדש את דגלו.
אלגוריתם האפייה מקיים את 3 התנאים אותם הגדרנו: בהכרח יש מספר מנצח והוא גם יחיד (כי הוספנו את עניין המזהה הייחודי במקרה של תיקו) ובמקרה הגרוע תהליך שמעוניין להיכנס יצטרך לחכות עד ש- n-1 התהליכים האחרים ייכנסו ויצאו.
בסוף שיעור שעבר שאלנו מדוע תהליך צריך 'להדליק' את הדגל ורק אח"כ לבחור מספר? האם כל שורות הקוד הבודקות את מצב הדגלים של התהליכים האחרים אכן הכרחיות כדי שהאלגוריתם הנ"ל יהיה נכון? האם אפשר לוותר על שורות 1,3,5 בקוד?
בלעדיהן התנאי של ההדרה הדדית לא היה מתקיים: נניח ששני תהליכים בוחרים מספר באותו זמן, שניהם קיבלו את המספר 1 (בהתחלה לכולם המספר מאופס). בעל המזהה הייחודי הנמוך ייכנס. אך נניח שלפני שבעל המזהה הנמוך הצליח להיכנס, הוא איבד שליטה ודווקא בעל המזהה הגבוה הצליח להיכנס. כעת בעל המזהה הייחודי הנמוך יקבל שוב שליטה ויצליח גם להיכנס. הדלקת וכיבוי הדגל האישי חיוניים, כדי למנוע מקרה כזה. לולאת ה-while הפנימית מוודאת שאנחנו משווים את המספר שלנו רק עם מספרים של תהליכים שאכן רוצים להיכנס (והדליקו את הדגל).
פתרונות ברמת התוכנה לבעיית הקטע הקריטי בעלי חסרונות: (1) לולאת כניסה לקוד קריטי מהווה busy wait וגוזלת זמן. (2) אם תהליך נתקע בקטע הקריטי, אין לנו דרך לדעת זאת. כעת נראה פתרונות ברמת חומרה / מערכת הפעלה.

[מצגת os4-3 cop]
<u>פתרונות ברמת החומרה לבעיית הקטע הקריטי</u>:
בתחילה נדון במחשב בעל מעבד יחיד. כאשר תהליך נמצא בקטע קריטי ומתרחשת פסיקה, אנחנו מאבדים שליטה ובעצם עד שאותו תהליך יקבל שליטה בחזרה, אף תהליך אחר לא יכול להתקדם. אולי נוכל לבטל את הפסיקות כאשר תהליך נמצא בקטע קריטי ואז נבטיח שהוא אכן ימשיך להתקדם בו ולא ייתקע בפנים עקב פסיקה? פתרון זה לא מומלץ, תמיד קיימת האפשרות שנבצע קריאת מערכת בתוך הקטע הקריטי או גישה למען (כתובת) אסור(ה) בזכרון ודווקא כן נרצה שיהיו פסיקות במקרים כאלו.
ביטול הפסיקות, נותן בעצם למתכנת הרשאה חזקה מדי, הוא עלול שלא להחזיר שליטה ל'משגוח' ולגרום להרס ☺. חסימת הפסיקות יכולה לפעול טוב במחשב חד-מעבד. אך חסימתן לא תפתור את הבעיה כאשר מדובר במספר מעבדים, כי הפסיקות בשאר המעבדים לא ייחסמו. אמנם יש ארכיטקטורות שמאפשרות לחסום את הפסיקות בכל המעבדים, אבל זה כבר נשק 'אטומי' מסוכן ו..עלול להיגרם נזק ☹. לכן הפתרון של חסימת פסיקות לא יעיל.

פתרון חלופי אחר: אולי נגדיר פקודות מכונה, שמערכת ההפעלה תוכל להפעיל, להגדרת קטע קוד קריטי:
(א) test and set – בדיקת ערך של מילה בזכרון וקביעת ערכה בו-זמנית. כביכול שתי הפעולות מתרחשות בו-זמנית.
(ב) swap / שחלוף – מחליפים ערכים של שני משתנים. נוכל להשתמש במנעולים כקוד כניסה ויציאה לקטע הקריטי. כאשר נרצה להיכנס, נבדוק אם המנעול פתוח וננעל אותו בעצמנו, כאשר נרצה לצאת, נפתח את המנעול מחדש כך שתהליכים אחרים יוכלו להיכנס. יש כמובן מפתח יחיד למנעול (גם אתם נזכרתם בג'אווה וב-box? ☺).

נבדוק את הרעיונות שהצענו:

```
boolean TestAndSet(boolean *target) {
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
```

(א) הפקודה מקבלת מען של פרמטר בוליאני, מעתיקה את ערכו (test) למשתנה מקומי, משנה את הערך של המשתנה (set) שקיבלה ל-true ומחזירה את הערך המקורי (לאחר שנשמר במשתנה מקומי). הפקודה הזו מבצעת שתי משימות בו-זמנית: גם שומרת את הערך של המשתנה וגם משנה את ערכו ומחזירה את הערך החדש. נגדיר את הפקודה הזו כאי-פריקה, כך שלא ניתן לאבד שליטה במהלכה.

```
do {
  while (TestAndSet(&lock));
    critical section
  lock = FALSE;
    remainder section }
```

בקטע הכניסה לקטע הקריטי, בודקים אם הערך שמחזירה הפקודה הוא false (כך הוא אותחל) ואז נוכל להיכנס. לאחר שנכנסנו, ערכו השתנה ל-true ולכן תהליכים אחרים לא יוכלו להיכנס עד שנצא ונשנה את ערך המנעול בחזרה ל-false. הפתרון הזה לא מושלם: אמנם התנאי של הדרה הדדית מתקיים וגם התנאי של התקדמות, כי בהכרח אחד התהליכים יצליח להתקדם (זה שמגיע ראשון ומוצא את המנעול במצב false). אך התנאי של החסם לא מתקיים, כי ייתכן שהתהליך שלנו כל פעם יקבל שליטה לאחר שתהליך אחר כבר נכנס ואין לנו זמן חסום שנמתין עד שנוכל להיכנס ☹.

(ב) הפקודה מקבלת שני ערכים בוליאנים ומבצעת שחלוף ביניהם, ע"י שימוש במשתנה זמני. אמנם מדובר ב-3 פעולות בסיסיות, אך נניח כמובן שבשפת מכונה הדבר מתורגם לפעולה יחידה שהינה אי-פריקה. בכניסה לקטע הקריטי, אנחנו משנים את ערך המפתח האישי שלנו ל-true. היות וערך המנעול אותחל ל-false, לאחר ביצוע swap ערך המפתח שלנו הוא false ולכן נוכל להיכנס. לעומת זאת, לאחר שתהליך כבר נכנס, ערך המנעול true ולכן לאחר ביצוע פעולת swap, תנאי לולאת ה-while הפנימית תמיד יתקיים, מה שיגרום לכך ששאר התהליכים לא יוכלו להיכנס.

```
do {
 /* Each process has a local Boolean variable key */
key = TRUE;
 while (key == TRUE)
        Swap(&lock, &key);
   critical section
lock = FALSE;
   remainder section }
```

גם פתרון זה מקיים את תנאי ההדרה ההדדית ואת תנאי ההתקדמות, אך תנאי החסימה לא מתקיים. שוב, רק התהליך הראשון שיקבל שליטה כאשר ערך המנעול הוא false, יצליח להיכנס. לשאר התהליכים אין הבטחה למספר הניסיונות שיבצעו עד שיצליחו להיכנס.

דוגמה לשימוש נכון בפקודת TestAndSet כחלק מקוד כניסה לקטע קריטי, כך ש-3 התנאים יתקיימו: שימוש במערך בוליאני waiting (בעצם דגלים של התהליכים). לכל תהליך יש מפתח (key) שאותו הוא מדליק כאשר הוא רוצה להיכנס. לולאת הכניסה בודקת גם אם אני רוצה להיכנס וגם אם המנעול הוא false (ולכן התהליך הראשון יכול להיכנס). בקוד היציאה מהקטע הקריטי, נעשית בדיקה אם יש תהליך אחר שמעוניין להיכנס (ואז דגל ה-waiting שלו הוא true). בתוך הלולאה נבדוק את הדגל של n-1 התהליכים האחרים ואם אחד מהם 'הדליק' את הדגל שלו כדי לסמן שהוא רוצה להיכנס, 'נכבה' את הדגל שלנו. בסיום הלולאה, אם אף תהליך לא 'הדליק' את דגלו, נשנה את ערך המנעול בחזרה להיות false. קטע היציאה מבטיח את תנאי ההמתנה החסומה ולכן זה פתרון נכון לבעיית הקטע הקריטי. החסרון בפתרון זה שהוא מסובך: גם בקטע הכניסה וגם בקטע היציאה יש לולאות מורכבות שבודקות את הדגלים של כל התהליכים האחרים.

Process P_i

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
          critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i) lock = FALSE;
        else waiting[j] = FALSE;
          remainder section
} while (TRUE);
```

Semaphores: פתרון נוסף לבעיית הקטע הקריטי, ברמת קריאות מערכת (פרימיטיבים של ספריות). [בעברית: דגל, כמו זה שדגלן מוריד ומסמן לנהג הקטר שהוא יכול לצאת מהתחנה ☺]

הדגל יכול לקבל ערך של 0 או 1. ערכו מאותחל ל-1. ישנן שתי פעולות אטומיות על הדגל: wait (הורדת הדגל, מסומנת גם ע"י down \ P) ו-signal (הרמת הדגל, מסומנת גם ע"י up \ V). שימוש: בקטע הכניסה נבצע פעולת wait להורדת הדגל (כך שנהג הקטר יכול להיכנס ☺), וביציאה נבצע פעולת signal כך שתהליכים אחרים יוכלו להיכנס. חשוב להדגיש ששתי הפקודות wait ו-signal הן אטומיות.

```
do {
   wait(mutex);
      critical section
   signal(mutex);
      remainder section
} while (TRUE);
```

[בדיחה על נס (חנוכה?!): מדינת ישראל היא נס – או ששחור כאן או שיש או-נס (הנשיא קצב?) ☺]

wait(s): בעצם לולאה המאפשרת לראשון לעבור (כי הערך של הדגל מאותחל ל-1). התנאי של המתנה חסומה לא מתקיים..

```
wait (S):                    signal (S):
   while (S <= 0);              S++;
   S--;
```

נצטרך להשתמש ב-semaphore בצורה מתוחכמת יותר: נוסיף גם משתנה count (מאותחל לערך 1) ותור.

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

כעת שתי הפקודות השתנו. Wait: הראשון שמגיע משנה את ערך ה-count לאפס ולכן יכול להיכנס. כל השאר יגיעו וייכנסו לתור ההמתנה כדי להיכנס. Signal: מגדילים את ערך ה-count. תהליך שיצא מהתור עובר לרשימת ההמתנה (למקרה שירצה להיכנס שוב). גם כאן, החסם הוא n-1. (לחובבי מבני נתונים, מדובר בתור מסוג FIFO ☺). הפתרון הזה פחות מסובך מהקודם, כי עיקר הקוד 'מוסתר' בפעולות האטומיות.

ישנם שני סוגים של semaphores: בינארי ומונה (כללי). בסוג המונה (counting semaphore) אפשר להשתמש לא רק בקטע הקוד קריטי, אלא גם כסלקטור שמאפשר למספר תהליכים יכולים להיכנס לקוד מסוים – כאשר אין הכרח שמספר זה יהיה 1 (ואז מדובר בקטע קריטי). למשל ייתכן שנרצה לאפשר למספר תהליכים לבצע בו-זמנית קטע קוד מסוים. די בקלות ניתן לממש semaphore בינארי על סמך הכללי: הדבר נעשה ע"י שימוש בתור ורשימת המתנה-לריצה (מזכיר את הקוד המופיע למעלה, בו עשינו שימוש במשתנה שנקרא count).

ניתן גם לממש semaphore כללי על סמך בינארי, אך נזדקק לשניים כאלו. אחד מהם (S2) יאותחל לאפס והשני (S1) יאותחל ל-1. דוגמה למימוש: ערך המשתנה C יציין את מספר התהליכים שיכולים להיכנס לקטע הקריטי. נשתמש ב-S1 כסוג של ביקורת לקטע קריטי. את שתי השורות הראשונות ב-wait נצליח לעבור רק אם אין תהליך אחר שנמצא בפנים. כעת נפחית את ערך המשתנה C (לסמן שיש מקום אחד פחות בפנים). לעומת זאת, אם C שלילי, אנו צריכים להיחסם. נשחרר את הקטע הקריטי ע"י signal(S1) ונבצע wait(S2). היות ו-S2 מתוחל לאפס, הראשון שיגיע ייחסם. S1 מסייע להפוך את הפעולות wait ו-signal לאטומיות. S2 משמש כדי לספור את התהליכים שהצליחו להיכנס. כך, כל מי שמורשה להיכנס אכן מצליח ואילו מי שלא מורשה, ימתין בתור. כאמור ה-semaphore הבינארי S2 משמש לכך.

- *wait*

```
waitb(S1);
C--;
if (C < 0) {
        signalb(S1);
        waitb(S2);  }
signalb(S1);
```

- *signal*

```
waitb(S1);
C++;
if (C <= 0)
     signalb(S2);
else signalb(S1);
```

ניתן להשתמש ב-semaphore גם כאשר רוצים לבצע סנכרון, למשל כאשר יש קטע קוד מסוים שרוצים להריץ בתהליכון מסוים, רק לאחר שקטע קוד אחר הורץ בתהליכון אחר. במקרה כזה נשתמש ב-semaphore שמאותחל דווקא לאפס. ע"י שימוש בפקודות wait ו-signal, נוכל להבטיח זאת. בדוגמה: גם אם התהליך Pj יפעל מהר יותר, הוא לא יצליח לעבור, כי ערך הדגל אותחל לאפס. רק לאחר שתהליך

P_i	P_j
⋮	⋮
A	$wait(flag)$
$signal(flag)$	B

בעיות ידועות בעת שימוש ב-semaphores:

(א) nested deadlock – במקרים בהם צריך לעבור שני דגלים כדי להיכנס לקטע קריטי. נצטרך שכל התהליכים יבצעו את פעולות ה-wait ופעולות ה-signal באותו הסדר בדיוק. בדוגמה מימין, אם כל אחד יריץ את השורה הראשונה שלו, בעצם שני הקטעים הקריטיים חסומים ו..חבל. אף אחד לא יכול להיכנס לקטע הקריטי ☹.

P_0	P_1
$wait(S);$	$wait(Q);$
$wait(Q);$	$wait(S);$
⋮	⋮
$signal(S);$	$signal(Q);$
$signal(Q)$	$signal(S);$

(ב) הרעבה – אם בטעות הגדרנו את הדגל כך שיפעל כ-LIFO.

(ג) היפוך קדימויות, priority inversion – דווקא תהליך עם עדיפות נמוכה נכנס לקטע הקריטי ובעצם חוסם ביצוע של תהליך עם עדיפות גבוהה.

[מצגת os5-1 con]

בשיעור שעבר ראינו את בעיית היצרן-צרכן. הפתרון שהצענו אמנם היה נכון וקיים את 3 התנאים, אך התהליכים היו עסוקים בלולאת המתנה, busy waiting. ניתן לשפר את הפתרון ע"י שימוש בפקודות sleep ו-wakeup. כעת נראה פתרון לבעיה זו גם ע"י שימוש ב-semaphores.

נזדקק ל-3 דגלים, שכל אחד ידאג לפעולה מסוימת: הדרה הדדית, סימון מספר התאים המלאים במכלא (buffer), סימון מספר התאים הריקים שבמכלא. בעת אתחול, יש 0 תאים מלאים, n תאים ריקים, והערך של הדגל שאחראי להדרה הדדית מאותחל ל-1. הצרכן תחילה מחכה שיהיה תא מלא (עם מידע), לאחר מכן הוא לוקח את המידע ואז מפחית את הערך של מספר התאים שבהם יש מידע. אם נהפוך את הסדר של הפקודות waif(full) ו-wait(mutex) (גם ביצרן וגם בצרכן, שהרי אם נהפוך רק באחד מהם יהיה לנו בעצם מצב של nested deadlock) הפתרון ייכשל כי הכרזנו שנכנסנו לקטע קריטי ואז בעצם אנחנו נתקעים בתור ולא מתקדמים.

Producer Process

```
do {
    …
    produce an item in nextp
    …
    wait(empty);
    wait(mutex);
    …
    add nextp to buffer
    …
    signal(mutex);
    signal(full);
} while (TRUE);
```

Consumer Process

```
do {
    wait(full)
    wait(mutex);
    …
    remove an item from buffer to nextc
    …
    signal(mutex);
    signal(empty);
    …
    consume the item in nextc
    …
} while (TRUE);
```

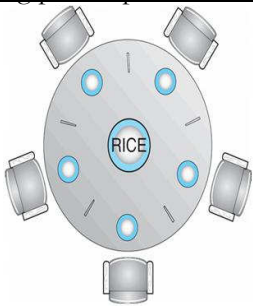
בעיות נוספות של מקביליות:

בעיית readers-writers (כותבים-קוראים)

יש מסד נתונים, חלק קוראים ממנו וחלק כותבים אליו. אין בעיה שברגע נתון יהיו יותר מקורא יחיד, כי קורא לא משנה את המידע. תמיד נשאל מי מקבל את העדיפות – קורא או כותב. למשל, אם יש קוראים בפנים ומגיע כותב, נוציא את הקוראים ונאפשר לכותב להיכנס ולבצע עדכון / שינוי למידע. הגדרת הבעיות: First – כל עוד אין כותב שמעוניין לכתוב, צריך לתת עדיפות לקוראים. Second – גם אם יש קוראים, צריך לאפשר לכותב לכתוב.

נראה פתרון לבעיית first: נשתמש בשני דגלים, אחד עבור הדרה הדדית ושני עבור חסימה של הכותבים. נוסיף גם משתנה שסופר כמה קוראים בו-זמנית יש כרגע. שני הדגלים יאותחלו ל-1 ואילו המונה של הקוראים ל-0. קוד של כותב: תחילה מבצע חסימה, אם מצליח לעבור הוא כותב ואז משנה בחזרה את הדגל. קוד של קורא: משנים את דגל ההדרה ההדדית ומגדילים את מספר הקוראים. כעת מבצעים חסימה ל-wrt (הדגל) כדי שכותב ייחסם. לאחר הקריאה, מפחיתים את מונה הקוראים ויש בדיקה אם מספר הקוראים התאפס – מאפשרים לכותב להיכנס.

בעיית dining philosophers (פילוסופים סועדים)



חמישה פילוסופים יושבים סביב שולחן עגול שעליו צלחת מרכזית של אוכל. בין כל שני סועדים יש צ'ופסטיק (או מזלג, למת(ע)קשים). צריך שני צ'ופסטיק כאלו כדי לאכול, מה שאומר שאם סועד מסוים אוכל, שני השכנים מימינו ומשמאלו לא יכולים לאכול. בשולחן של 5 סועדים, רק 2 פילוסופים יכולים לאכול בו-זמנית (כי ביחד הם הרימו 4 צ'ופסטיק מתוך ה-5 ולכן אף פילוסוף נוסף לא יכול לאכול). בעיה זו יכולה לגרום גם לקיפאון (אם כל סועד יחזיק את המזלג הימני / שמאלי שלו) כך שאף אחד לא יוכל לאכול, או הרעבה (אם לפחות אחד מהשכנים של פילוסוף מסוים לא מפסיק לאכול) ואז סועד מסוים לא יוכל לאכול לעולם.

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

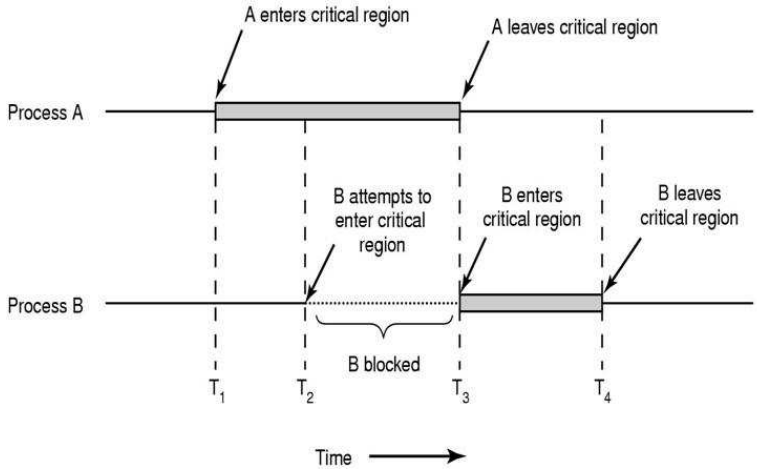
פתרון: נתייחס (באופן לא מפתיע) לכל סועד כתהליך נפרד. נשתמש במערך של semaphores בגודל מספר הסועדים. נאתחל את כל אחד מהדגלים ל-1, כדי שהסועד הראשון יוכל להיכנס. יש בעיה: הקוד הזה סימטרי מדי, ייתכן מצב של קיפאון, אם כל הסועדים ירימו תחילה את המזלג הימני / שמאלי בו-זמנית.

נציע דרכים לשבירת הסימטריות ע"י: (א) נגביל את גודל השולחן ל-4 – כך לפחות אחד יוכל לאכול, כי נשאיר את המזלג החמישי על השולחן שיהיה זמין. פתרון זה פוגע במקביליות. (ב) הגדרת פעולת האכילה כאטומית – נרים שני מזלגות בו-זמנית במקום להרים מזלג יחיד ולהמתין עד שנצליח להרים גם את המזלג השני. כך, או שנצליח לאכול, או שלא. (ג) ליד כל פילוסוף שמרים תחילה מזלג ימני, יתיישב פילוסוף שמרים תחילה מזלג שמאלי.
כאמור, פתרון א' לא פותר את בעיית ההרעבה, שהרי סועד שאוכל חוסם את שני שכניו שלא יאכלו. נוכל להוסיף תור (בצורת FIFO) לכל אחד מהמזלגות, כך יהיה לנו חסם מקסימלי של 2: כל סועד שמבקש מזלג תפוס, ייכנס להמתין בתור. לכל היותר יחכה שני תורות לשני המזלגות – וכך פתרנו את בעיית ההרעבה. נראה כיצד פתרון ב' יכול להיות תקין. נגדיר 3 מצבים לכל פילוסוף: חושב, רעב, אוכל. לכל סועד אנחנו שומרים את המצב בו הוא נמצא. נממש הפעולות כאטומיות. לקיחת מזלגות: הפעלת הדרה הדדית, שינוי מצב ל'רעב', בדיקה אם אפשר לקחת את שני המזלגות ואם כן שינוי מצב ל'אוכל'. הנחת מזלגות: חוזרים למצב 'חושב' ונותנים הזדמנות לשכנים לאכול. בפעולות אלו ניעזר בפונקציה נוספת בשם test(diner): אם אני רעב ושני השכנים שלי לא אוכלים, אני יכול לאכול ולכן אני מרים את שני המזלגות. מספיק שאחד השכנים שלי אוכל, כדי שאני לא אצליח לאכול.
יתבצע שינוי הסיגנל של המזלג ע"י up, כך בפעולה take forks נוכל לעבור את החסם down. לעומת זאת, אם ה-test לא הצליחה, לא נוכל לעבור את down ובעצם לא נוכל לקחת את שני המזלגות.

הרצאה מס' 10 - 21.12.09

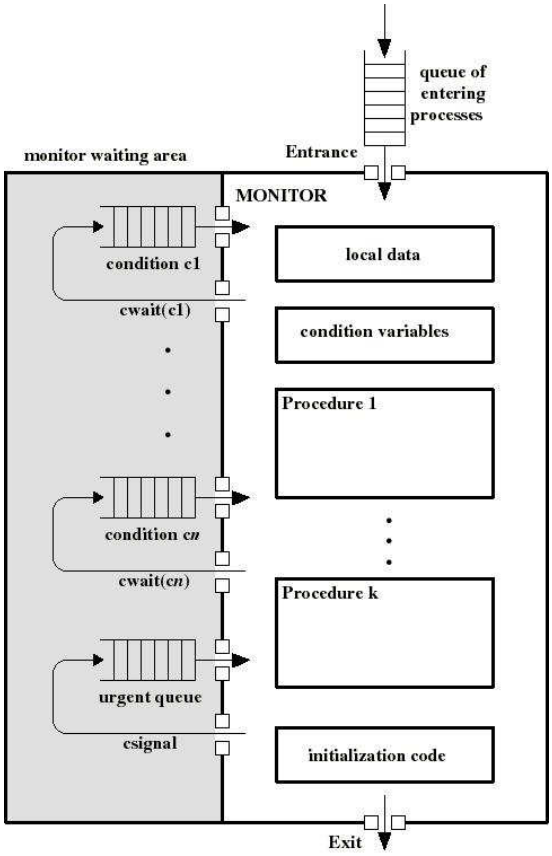
[מצגת os5-2 con]
בשיעורים הקודמים דיברנו על בעיות במקביליות כגון תחרות וקטע קריטי. ראינו פתרונות תוכנה לבעיית הקטע הקריטי וגם פתרונות חומרה ע"י semaphores. סקרנו מספר בעיות של מקביליות: יצרן-צרכן, כותבים-קוראים, פילוסופים סועדים.

היום נסקור פתרונות נוספות של בעיית הקטע הקריטי, ברמה של שפות תכנות. הפתרון הקלאסי נקרא critical region. בשפות עיליות יש לנו צרף-שפה (משתנה משותף) ומשתנה בוליאני שבודק אם אנחנו יכולים להיכנס לתחום הקריטי סביב המשתנה המשותף ואם כן – אילו פקודות אנחנו צריכים לבצע. את התחום הקריטי נוכל לבנות ע"י סמפורים. הפתרון הזה עוסק רק בתחביר, אך החשיבות היא למעשה בעצם קיום התחביר ('סוכר סינתטי'). כמובן, בזמן שאוסף הפקודות על המשתנה המשותף מתבצע, האוסף כביכול אטומי ואף תהליך אחר לא יכול לבצע אותן.



בקטע הכניסה לקטע הקריטי, תחילה מבצעים בדיקה של ערך המשתנה הבוליאני ש'שומר' על המשתנה המשותף. אם ה'שומר' מאפשר, ניתן להיכנס ולבצע את אוסף הפקודות. אם לא, התהליך שלנו נכנס לרשימת המתנה. נניח כמובן שמדובר בתור מסוג FIFO, כדי לקיים את התנאי של המתנה חסומה. ההנחה היא שהמהדר נבדק והוא מיישם את הקוד כמו שצריך. אך עדיין לא מדובר בפתרון קסם, כי נצטרך לדעת להגדיר את מבנה הנתונים כמו שצריך וגם את ה'שומר'. בנוסף, התחביר הזה מוגבל, ניתן אמנם להצהיר על קטע קריטי אך הוא לא מאפשר לנו 'לצאת' באמצע הקטע הקריטי (נניח במקרה שניסינו לבצע פעולה שאינה אפשרית ונרצה לצאת כדי לאפשר לתהליך אחר להיכנס לקטע הקריטי בזמן זה).

צרף-השפה הבא נקרא מוניטור. גם הוא בונה קטע קריטי, אך בעל יכולות גבוהות יותר (וגם מסובכות יותר) ביחס לתחום הקריטי. המוניטור מגדיר יחידה / מחלקה שמגנה על מבנה נתונים משותף ויוצרת קטע קריטי סביבו, עבור כל הפעולות המוגדרות במוניטור. יכולים להיכנס מספר תהליכים למוניטור, אך רק תהליך יחיד יכול להיות פעיל. המהדר עצמו משתמש בסמפורים. המוניטור כולל מספר פעולות / מתודות, קטע תיחול ומשתני-עזר מקומיים (וכמוסים) לשימוש המוניטור. תהליך יכול להיכנס לתחום של המוניטור ע"י הפעלה של פעולה של המוניטור. בנוסף, יש תור של תהליכים הממתינים להיכנס למוניטור. מוניטור מקיים את התנאי של הדרה הדדית. ע"י משתנה-תנאי ניתן למנוע תחרות. משתנה-תנאי הוא משתנה מקומי של המוניטור, אין גישה אליו מבחוץ. משתני-תנאי הם גם סמפורים, פועלים באותה לוגיקה. הפעולות המבוצעות עליהם נקראות cwait (שחוסמת מיד תהליך) ו-csignal (שמחזירה תהליך לביצוע ומכניסה לתחרות על הכניסה לקטע הקריטי אבל אז התנאי של המתנה חסומה לא מתקיים. בעצם תפקידה להעיר תהליך אחד או יותר הנמצאים בתור הכניסה). המוניטור לא נוצר רק לצורך פתרון בעיית הקטע הקריטי, אלא גם לפתרון בעיות נוספות. (גם סמפורים לא מיועדים רק לפתרון בעיית הקטע הקריטי)



היות והוספנו משתני-תנאי למוניטור, נוצרה לנו גם 'חצר-אחורית', תורים נוספים של התנאים השונים שהוספנו. כביכול אם תהליך לא מקיים תנאי מסוים, הוא נכנס לתור המתנה של משתנה-התנאי הזה.

בנוסף, יש גם תור-חירום: נניח שתהליך מבוצע באחת מפקודות המוניטור ונקראה הפעולה csignal. כעת יש שני תהליכים שיכולים לרוץ – התהליך הבא בתור של אותו משתנה-תנאי והתהליך הנוכחי. אך רק תהליך יחיד יכול לרוץ ולכן התהליך שהפעיל את csignal נכנס לתור-החירום ואילו התהליך שהתעורר כתוצאה מהפעולה הזו יהיה זה שיזכה לרוץ. פעולת ה-csignal צריכה להתבצע בסוף פקודת המוניטור, כי אחרת פעולת התהליך שביצע אותה תיגדע בטרם תסתיים פקודת המוניטור ואז בעצם יש לנו תהליך יחיד שרוצה לרוץ (התהליך שהתעורר) ואין צורך בקיום תור-חירום.

כמובן, תהליך הנמצא בראש תור-החירום הוא הראשון להתבצע, לפני התהליכים האחרים הנמצאים בראש תורים של תנאים אחרים, וגם לתהליכים אלו יש עדיפות על פני תהליכים הנמצאים בתור של התהליכים החדשים המחכים להיכנס לתוך המוניטור.

יישום של מוניטור לפתרון בעיית יצרן-צרכן:

```
ProducerI:          ConsumerI:
repeat              repeat
 produce v;          take(v);
 append(v);          consume v;
forever             forever
```

הפעולה append מוסיפה פריט למאגר ואילו הפעולה take צורכת את הפריט. המוניטור עצמו מכיל מכלא עבור הפריטים וכן שני משתני-תנאי notfull (ביצוע csignal עליו יסמן שהמכלא לא מלא, הצרכן יפעיל כדי לעדכן את היצרן לייצר פריט נוסף) ו-notempty (ביצוע csignal עליו יסמן שהמכלא לא ריק, כך היצרן מעדכן את הצרכן שהוא יכול לצרוך).

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

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

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

פעולת ה-csignal מבוצעת אחרונה ולכן יהיה תהליך יחיד שרוצה לפעול (ואין צורך בתור-חירום).

ניתן להשתמש במוניטור גם לפתרון בעיית פילוסופים סועדים:

```
Philosopher i:
dp.pickup(i);
  …
  eat
  …
dp.putdown(i);
```

נוסיף מערך של משתני-תנאי (נקרא לו self) בגודל מספר הסועדים. בעת האתחול, הפילוסופים במצב 'חושב'. בסוף הפעולה test נוסיף את הקריאה self[i].csignal שמודיעה שהתהליך הנוכחי מעוניין לאכול. בסוף הפעולה pickup (הרמת שני המזלגות בו-זמנית), לאחר ששינינו את המצב ל'רעב' ובדקנו שאנחנו אכן יכולים לאכול ע"י test, נבדוק אם אחד השכנים שלנו אוכל ואם כן נחסום את עצמנו ע"י self[i].cwait. בסוף הפעולה putdown (הנחת שני המזלגות בו-זמנית), נבדוק אם אחד השכנים שלנו מעוניין לאכול ואם כן נעיר אותו (אי אפשר לאכול כאשר ישנים ☺).

ביישום תהליך של פילוסוף, נשתמש בפעולות pickup ו-putdown של המוניטור (כזכור, רק המוניטור בעל גישה למשתני-התנאי). פתרון זה אכן מונע מצב של קפאון (כי מרימים את שני המזלגות בו-זמנית), אך עדיין תיתכן הרעבה – שהרי אין לנו הבטחה שהתור של csignal הוא מסוג FIFO (אם הוא כן, אז מנענו גם הרעבה). בניגוד לשימוש בסמפורים, בו היה לנו תור בגודל 2 על כל מזלג (כי כביכול שני הפילוסופים היושבים מצידי המזלג יכולים להמתין להשתמש בו), בפתרון הזה לאחר ביצוע של csignal יש לנו תהליך יחיד שמחכה וייתכן ויהיה מדובר באותו תהליך כל הזמן ותהליך אחר 'ירעב' (וגם הפילוסוף שהוא מייצג ☺).

```
Variables (for each procedure Pn )
semaphore mutex;  // (initially = 1)
semaphore next;   // (initially = 0)
int next-count = 0;
```

המהדר מיישם את המוניטור ע"י 2 סמפורים בינאריים (אחד מהם, mutex, משמש להדרה הדדית כך שרק תהליך יחיד מבצע את הפרוצדורה בכל רגע נתון) ומשתנה counter (שומר את מספר התהליכים המעוניינים לבצע את הפרוצדורה).

```
Each procedure Pn will be replaced by
    wait(mutex);
    ...
        body of Pn;
    ...
    if (next_count > 0)
        signal(next)
    else
        signal(mutex);
```

הפקודות של המוניטור יוחלפו ע"י פרוצדורה של המהדר, כך שכל פקודה 'תיעטף' ע"י קטע כניסה וקטע יציאה, כי הרי מדובר בקטע קריטי. לאחר שתהליך ביצע את הפרוצדורה, בקטע היציאה יש בדיקה האם יש תהליך אחר שמעוניין לסיים את ביצוע הפרוצדורה ואם כן נאפשר לו, ע"י פעולת signal. התהליך האחרון שיסיים את ביצוע הפרוצדורה ישחרר את הסמפור mutex. יש שתי אפשרויות שתהליך אחר יקדים אותנו בביצוע הפרוצדורה: או שתהליך אחר הקדים אותנו בשל mutex, או שהוא נמצא לפנינו בתור הביצוע של הפרוצדורה.

```
semaphore x_sem; // (initially = 0)
int x-count = 0;
```

עבור כל משתנה-תנאי של המוניטור, המהדר מגדיר סמפור ומשתנה counter (מספר התהליכים ממתינים על התנאי הספציפי הזה). הסמפורים מוכרים ע"י כל הפרוצדורות של המוניטור.

בתוך "body of Pn" (הגוף המקורי של הפרוצדורה) ייתכן ויבוצעו קריאות לפקודות cwait ו-csignal על כל אחד מהסמפורים של משתני-התנאי – ואז ה-next-count שמשתנה בפעולות אלו הוא של הפרוצדורה עצמה.

- **x.cwait**

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

הפקודה cwait מגדילה את ה-x-count (של משתנה-התנאי הספציפי x) כדי לציין שיש תהליך נוסף שממתין (אנחנו), אך לפני כן אנו בודקים אם יש תהליך אחר שמעוניין לרוץ (לבצע את אותה פרוצדורה) ע"י בדיקת המשתנה next-count של הפרוצדורה. אם כן אנו משחררים אותו ואם לא אנחנו נבצע signal(mutex) כדי שמישהו אחר יוכל להיכנס. רק לאחר מכן אנחנו חוסמים את עצמנו. כאשר יעירו אותנו, נפחית שוב את ערך המשתנה x-count כדי לציין שאנחנו כבר לא ממתינים.

- **x.csignal**

```
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;}
```

הפקודה csignal בודקת אם יש תהליכים שממתינים לריצה עבור הפרוצדורה המסוימת. אם כן נאפשר לראשון לרוץ (נעיר אותו) אחרי שהגדלנו את המשתנה next-counter של הפרוצדורה עצמה. אנו בודקים אם אנחנו יכולים לרוץ בעצמנו. בעצם לאחר ביצוע signal(x_sem) יש שני תהליכים שמעוניינים לרוץ – אנחנו והתהליך שהערנו. רק אחד מאיתנו אכן יצליח, הוא יצטרך לעבור את השורה wait(next). לאחר שהצלחנו לעבור את השורה הזו, יש תהליך אחד פחות שמחכה לבצע את הפרוצדורה ולכן נפחית את ערך המונה next-count של הפרוצדורה.

חשוב להדגיש: בכל אחת מהפעולות הנ"ל אנו דואגים גם להמשך הביצוע של הפרוצדורה ע"י תהליכים אחרים (במקרה של csignal). הקוד הזה מבטיח שהתהליכים יבצעו את הפרוצדורה ע"י תור, שמנוהל ע"י הסמפור next של הפרוצדורה.
ניתן לבצע broadcast כך שהפעולה signal מעירה את כל התהליכים שממתינים, אך רק הראשון יצליח לעבור את mutex ולבצע.

<u>סנכרון ברמה של מערכות הפעלה</u>:

(א) סולאריס – משתמשת במשתנים שנקראים adaptive mutexes, משתנים המוודאים הדרה הדדית. בעלת משתני-תנאי המאפשרים לבצע נעילה לפתרון בעיית כותבים-קוראים.

(ב) חלונות XP – משתמשת ב-spin-locks, כל המעבדים ניגשים לזכרון משותף ומי שלא מצליח להיכנס נמצא ב-bust-wait. יש גם dispatcher שפועל בדומה למשתנה-תנאי ואחראי לקבוע מי התהליך הבא שירוץ.

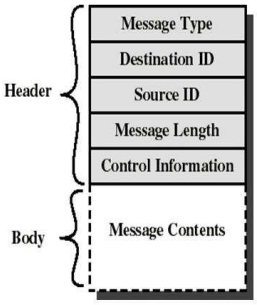
(ג) לינוקס – בגירסאות הראשונות, הגרעין (kernel) השתמש בפסיקות, כדי ליישם קטעים קריטיים קצרים. מגירסה 2.6 אין יותר שימוש בפסיקות, כעת הגרעין יכול לחסום את עצמו ולאפשר לתהליך אחר לרוץ במקומו. לינוקס מאפשרת שימוש בסמפורים וב-spin-locks.

(ד) סנכרון במודל של תהליכונים – מספק mutex locks ומשתני-תנאי הקיימים ב-API הכללי של Pthreads. בנוסף יש הרחבות התלויות במערכת ההפעלה עצמה. בשימוש ב-mutex lock (שהוא מסוג סמפור), התהליך הנחסם מפסיק לרוץ, בעוד מנעול מסוג spin-lock מניח שההמתנה קצרה ומאפשר לתהליך הנחסם להמשיך לרוץ.

[מצגת os5-3 con, הגענו עד שקופית 26 כולל]

<u>תקשורת בין-תהליכים (Inter-Process Communication, IPC)</u>:

עד כה עסקנו במודל של shared memory, כעת נדון ברעיון של העברת מסרים בין תהליכים, בו אנו מניחים שאין מען המשותף בין התהליכים (לכאורה המסר הוא המשותף כי גם השולח וגם המקבל בעלי גישה אליו ☺). אם הנמען ידוע, נפעיל את send(message) ואם לא, נפעיל send(destination, message). גם בפקודת קבלת המסר, ייתכן ונרצה לקבל מסר מכל שולח ע"י receive(message) או משולח מסוים ע"י receive(source, message). גודל המסר יכול להיות קבוע מראש או משתנה בהתאם לצרכים.



באיור מימין, דוגמה למבנה של message. אם מדובר במסרים בעלי גודל קבוע, ניתן לדעת בדיוק היכן מתחיל ונגמר כל מסר בתור-המסרים. לעומת זאת כאשר הגודל לא קבוע, ניתן להשוות זאת ל-pipe המשמש לצורך streaming (הזרמה) של מידע.

אם המסרים עוברים בתוך אותו מחשב, הדבר נעשה ע"י שיתוף זכרון. אם המסרים עוברים בין מחשבים שונים, נעשה שימוש בערוצי תקשורת פיזיים (למשל כבל תקשורת לצורך גלישה באינטרנט..). בזמן שליחה / קבלה של מסר, תהליך יכול להיחסם (ואז מדובר במודל סינכרוני) או להמשיך לרוץ (ואז מדובר במודל א-סינכרוני). אם תהליך אחד שולח מסר וממשיך לרוץ עוד בטרם ידוע אם תהליך היעד אכן קיבל את המסר, המודל נקרא semi-synchronized. במודל סינכרוני: אם נרצה להיות מתואמים מול הצד השני, נפעיל blocking send ובעצם ניחסם עד שהנמען קיבל את המסר. ניתן גם לחסום את הצד המקבל עד שיקבל מסר, ע"י בשימוש ב-blocking receive. במודל א-סינכרוני: נעשה שימוש ב-non-blocking send כדי לאפשר לשולח להמשיך לבצע פעולות לאחר שליחת מסר. נוכל לאפשר לצד המקבל לבצע פעולות, מבלי להיחסם עד שיקבל מסר, נשתמש ב-non-blocking receive. לצד השולח הגיוני שלא להיחסם לאחר ששלח מסר, כי אולי הוא צריך להמשיך לשלוח מסרים נוספים. לעומת זאת לצד המקבל, אם הוא כבר ניגש לתור ההמתנה למסר, הגיוני שהוא צריך לקבל מידע מסוים החיוני לצורך המשך ריצתו ולכן הגיוני שהוא כן ייחסם.
כאמור, יש סכנה ב-blocking, כי אולי לא יעירו אותנו או שלא יגיע מסר נוסף. קיימת גם האפשרות לחסימה מותנית בזמן: אנו מפעילים קוצב זמן ומודיעים שאנו מוכנים להיחסם לכל היותר לזמן מסוים (כמובן שאם יגיע מסר קודם לכן, יהיה אפשרי להעיר אותנו קודם). כדאי להבין: ישנם 3 שילובים הגיוניים לפעולות החסימה של השולח / מקבל (שליחה וקבלה חסומות, שליחה וקבלה שתיהן לא חסומות, שליחה לא חסומה בעוד קבלה כן) בעוד השילוב הרביעי לא הגיוני. כאמור, השילוב הפופולרי והיותר הגיוני הוא שפעולת השליחה תהיה לא חסומה בעוד פעולת הקבלה כן. למשל ב-web, מודל שרת-לקוח, הלקוח (הצד השולח) דווקא ישמח להמשיך לבצע פעולות עד שהשרת יחזיר לו תשובה. לעומת זאת השרת (הצד המקבל) מעוניין להיחסם, כי תפקידו רק להמתין לבקשות (אמנם, כשהוא מקבל בקשה הוא מעביר אותה לתהליכון-עבד שיבצע אותה).

אם שני תהליכים רוצים להתקשר ביניהם: צריך תחילה להיות מוקם ערוץ תקשורת ביניהם ובנוסף הם צריכים להחליף מסרים ביניהם ע"י הפקודות send \ receive. כל ערוץ תקשורת בעל שני מאפיינים: פיזי (כבל תקשורת, זכרון משותף, פס מערכת) ולוגי (חד-כיווני / דו-כיווני, מעביר מסרים בעלי גודל קבוע / משתנה). כאשר מעוניינים להגדיר ערוץ תקשורת, צריך תחילה לוודא שיש לנו תשובות לשאלות הבאות: האם התקשורת היא חד/דו-כיוונית, כמה מסרים יכולים להיות מועברים בין שני הצדדים בכל רגע נתון, מה הקיבולת של כל מסר (והאם היא קבועה / משתנה), האם כל מסר מצד שולח מיועד לצד מקבל יחיד..

<u>לוגיקות שונות בקביעת גודל המכלא המשמש להעברת המסרים</u>:

(א) zero capacity (נקרא גם rendezvous) – ניתן להעביר 0 הודעות. מדובר בהעברה מיידית, יש נקודת-מפגש בין התהליכים. בשום נקודת זמן אין מסר שנשלח וממתין להתקבל.
(ב) bounded capacity – ניתן לשמור עד n מסרים, מתבצעת חסימה לצד השולח אם ביצע שליחה של n מסרים ומתבצעת חסימה לצד המקבל אם יש כרגע 0 הודעות.
(ג) unbounded capacity – השולח לעולם לא נחסם, הוא יכול לשלוח אינסוף מסרים (כמובן במחשבים תמיד יש לנו הגבלה).

<u>מיעון המסרים</u>:

(א) מיעון ישיר direct addressing – כאשר הצד השולח והצד המקבל מכירים זה את זה, המסר נשלח באופן ספציפי מהשולח למקבל. בשיטה זו פועלים ע"י הפעולות: send(name, message) ו-receive(name, message) שהרי לכל תהליך ידוע השם של הצד השני (השולח / המקבל). במקרה כזה מדובר בערוץ בין שני תהליכים ספציפיים, כי השמות ידועים.
(ב) מיעון לא-ישיר indirect addressing – כאשר השולח משגר מסר ולא בהכרח מכיר את המקבל, כך גם המקבל פשוט ממתין למסר כלשהו ולא בהכרח יודע / מעוניין לדעת מי השולח. [הערה של המרצה: ככה פועלים מרגלים, לא בהכרח יודעים מי הצד המקבל ☺. הערה של עדי: בדיוק כמו משלוח מנות, אנחנו מכינים ולא יודעים בזמן ההכנה מי יקבל את כל הממתקים שלנו ☺.] בשיטה זו השולח מכניס את המסר שלו לתוך תא-דואר, מבלי לדעת איזה תהליך ספציפי יקבל את המסר. לתא הדואר צריכים להיות לפחות שני תהליכים בעלי גישה אליו: השולח והמקבל. תא דואר יכול לקבל מסרים ממספר שולחים ולהיות נגיש למספר מקבלים. ייתכן כמובן שתהיה מדיניות, לפיה המקבל מוחק את המכתב מתא-הדואר או שהמקבל קורא את המסר ומשאיר את המכתב, כדי שתהליכים נוספים יוכלו לקרוא את המסר. שיטה זו יעילה כאשר אנחנו לא רוצים להתחייב מראש (לפני זמן הריצה) להצהיר מי השולח ומי המקבל. כך נכריז על תא דואר כמשותף ומי שירצה ישלח ומי שירצה יקבל. במקרה כזה נשתמש בפעולות לשליחה ולקבלה, המכירות את תא הדואר: send(mailbox, message) ו-receive(mailbox, message).
כמובן שאפשר שבצד השולח המסר יהיה ישיר / לא-ישיר וכך גם בצד המקבל (ייתכנו מספר שילובים).

חידוד ההבדלים בין mailbox ו-port:
לשניהם ייתכן ויש מספר שולחים. לתא-דואר (mailbox) ייתכן ויש מספר מקבלים (לכל בני המשפחה יש מפתח לתא-הדואר המשפחתי ☺), בעוד ל-port יש בהכרח רק מקבל יחיד. כמובן, ייתכן והמקבל של ה-port הוא בעצם dispatcher שבעצמו דואג להעביר את המסר לתהליכונים-עבדים אחרים.. port נוצר ע"י הצד המקבל (עיר הנמל שאליה מגיעות ספינות עם סחורה מרחבי הגלובוס ☺) ומושמד כאשר התהליך המקבל מסתיים. לעומת זאת תא-דואר (mailbox) נוצר ע"י מערכת ההפעלה (חברת הדואר ☺) שמעבירה את הבעלות עליו לתהליך והוא מושמד כאשר התהליך מסתיים או ע"פ החלטתו.

ניתן ליישם הדרה הדדית גם במודל מיסור (מלשון מסרים..), בדומה למודל של זכרון משותף:

```
Process Pi:
var msg: message;
repeat
  receive(mutex,msg);
  CS
  send(mutex,msg);
  RS
forever
```

נגדיר mailbox mutex שתפקידו לשמור על ההדרה ההדדית. בקטע הכניסה לקטע הקריטי, רק מי שמגיע ראשון יקבל את המסר ויסיר אותו (כל השאר ייחסמו כי פעולת הקבלה חוסמת, ותא-הדואר בעצם התרוקן לאחר הסרת ההודעה). בקטע היציאה המקטע הקריטי, נבצע שליחה של מסר. כמובן, יש להניח שמדובר בתור מסוג FIFO ולכן הפתרון הזה מקיים גם את התנאי של המתנה חסומה ומהווה פתרון תקין לבעיית הקטע הקריטי.

דוגמה נוספת לכך שמודל המיסור יכול לשמש כמנגנון לתיאום:

```
Producer:
var pmsg: message;
repeat
  receive(mayproduce, pmsg);
  pmsg := produce();
  send(mayconsume, pmsg);
forever
```

```
Consumer:
var cmsg: message;
repeat
  receive(mayconsume, cmsg);
  consume(cmsg);
  send(mayproduce, null);
forever
```

נאתחל תא-דואר may-consume כך שהוא ריק. היצרן יצרף את ה'תוצרת' שלו בצורת מסרים לתא-דואר זה, וממנו הצרכן יכול לצרוך. בנוסף, נאתחל תא-דואר may-produce עם k מסרים ריקים, שיציין שהיצרן יכול ליצור כי יש מקום במכלא. לאחר כל הוספת 'תוצרת' ע"י היצרן מספר המסרים בתא-דואר זה יפחת, ואילו לאחר כל צריכה של הצרכן מספר המסרים יגדל בחזרה.

[על נושא זה עברנו ברפרוף]

API – POSIX בסגנון של UNIX, מנשק משותף המיושם במערכות הפעלה רבות, מאפשר לתוכנה לרוץ בסביבות שונות. נעשה שימוש במשתנה פרימיטיבי shmget (shared memory get) שמקצה לנו זכרון משותף עם הרשאות מסוימות. כדי להתקשר / להשתתף בזכרון המשותף, נשתמש ב-shmat (shared memory attach).

הרצאה מס' 11 - 28.12.09

[מצגת os7-1 rea]

Real memory management:

ניהול הזכרון הפנימי, העיקרי. נושא זה נחלק ל-2, מסיבות היסטוריות:

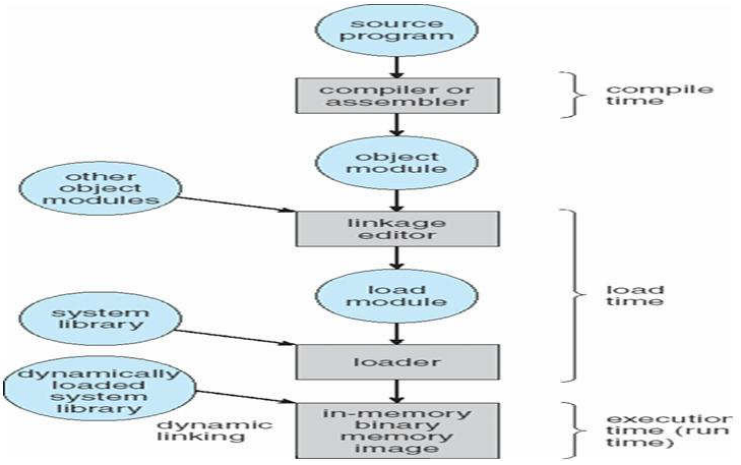
(א) בהתחלה היה זכרון פיזי (real) שהיה קטן יחסית לקיבולות של ימינו, דבר שגרם לבעיות כגון האטה בביצועים.

(ב) בהמשך עלו על רעיון הזכרון הוירטואלי, טכניקה המאפשרת את הביצועים של המחשבים של ימינו. ניהול הזכרון הוירטואלי מורכב מתתי-נושאים רבים.

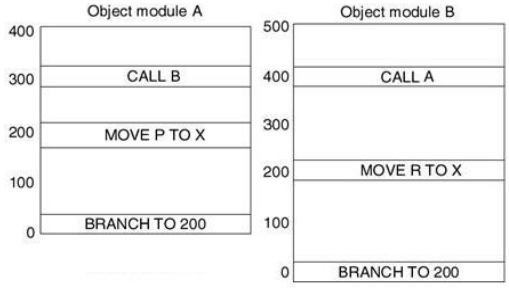
חזרה על מה שלמדנו לפני מספר שיעורים:

- כדי להריץ תוכנית, צריך תחילה לטעון אותה מהדיסק אל הזכרון, לתוך תהליך.
- המעבד יכול לגשת באופן ישיר רק לזכרון הראשי ולאוגרים.
- גישה של המעבד לאוגרים (registers) אורכת one CPU clock (מה שאומר שזו פעולה אטומית).
- לעומת זאת, גישה של המעבד לזכרון העיקרי לוקחת מספר cycles (מה שאומר שזו לא פעולה אטומית).
- לכן, יש להגן על הזכרון, כדי להבטיח תקינות הפעולות.

כאמור קיימת תת-מערכת לניהול הזכרון הפיזי, שדואגת שכל הנ"ל יתרחש באופן תקין.

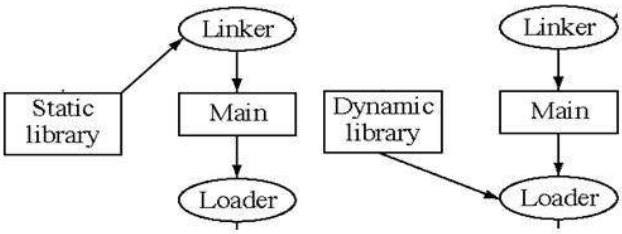


באיור מימין: מחזור חיים של תוכנית, מרגע שכותבים אותה וכל עוד היא רצה במערכת. בהתחלה יש לנו תוכנית-מקור הכתובה בשפת-סף (כדוגמת אסמבלר) או בשפה עילית (כדוגמת java). השפה ניתנת למהדר לצורך הידור (אם התוכנית בשפה עילית, צריך קודם לכן להמיר אותה לשפת-סף). ייתכן ומספר תוכניות יעברו הידור למספר מודולי יעד. כל מה שלא נפתר במהלך ההידור (קומפילציה) מבחינת זימוני שגרות (פונקציות) ומעני משתנים חיצוניים, נפתר במהלך שלב הקישור (linkage), ע"י עריכת מודולי היעד לכדי מודול יעד יחיד גדול. תפקיד ה-loader לקחת את מודול הטעינה ולטעון אותו כ-binary image (למשל קובץ a.out), כך התהליך מיוצג בזכרון. זמן הריצה של התוכנית נקרא execution time. הקישור יכול להיעשות גם בזמן הריצה של התוכנית (זוכרים שלמדנו על קבצי dll עם חלסצ"י בהקשר של dynamic linking בקורס תכנות מתקדם?).



Object module – בניגוד לקוד המכונה, מכיל טבלאות עם שמות משתנים חיצוניים וגלובליים שעדיין צריך לאתר במודולים אחרים, כדי למקם את כולם יחדיו, מה שדרוש לצורך פעולת הקישור. ייתכן ומכיל טבלת מענים שעדיין צריך לפתור. יש מודולים מוחלטים (המענים ידועים מראש), יש מודולים יחסיים, יש מיקומיים. באיור מימין: שני מודולי יעד, בכל אחד יש קריאה לאחר. לכן כאשר טוענים את הראשון הוא עדיין לא סופי. רק בשלב הקישור-שילוב, כאשר שני מודולי היעד יהיו ממוקמים, נדע בדיוק לאיזה מען נדע להפנות את הקריאות call a ו-call b.

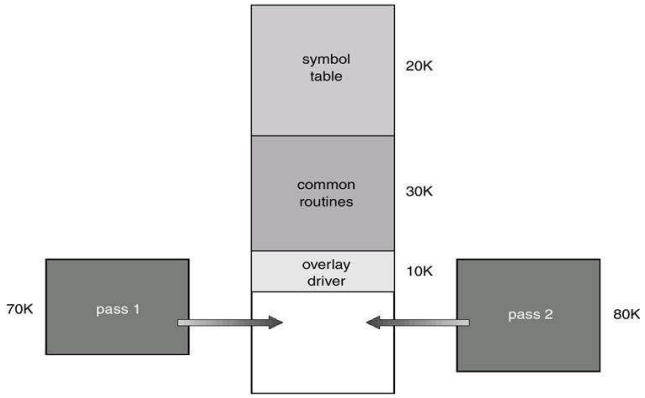
Dynamic linking – קישור דינאמי. מודולים חיצוניים עוברים קישור רק בזמן שלב הטעינה (load time) או אפילו רק בשלב הריצה (run time). כמובן שאם מלכתחילה יש לנו את כל המידע, הקישור יהיה פשוט יותר. ככל שנעשה את הקישור מאוחר יותר, יש לנו אמנם יותר גמישות, נוכל להתקשר ולהשתלב עם דברים מעודכנים יותר, אך הפעולה מורכבת יותר ועולה במחיר קישור שעלול לגזול זמן.



השוואה בין קישור סטטי וקישור בזמן הטעינה. בצד שמאל: קישור סטטי, שילוב הספריה הדרושה נעשה בשלב הקישור. בצד ימין: קישור דינאמי, שילוב הספריה הדרושה נעשה רק בשלב הטעינה. כאמור, יש גם אפשרות לעשות שילוב של הספריה, באופן חלקי, בזמן הריצה של התוכנית.

<u>השוואה בין גודל התכנית וגודל הזכרון</u>:
כאשר הזכרונות היו עדיין קטנים ואילו התוכניות הלכו וגדלו, כיצד ניתן היה להריץ תוכנית הגדולה מגודל הזכרון הפיזי?
יש 2 טקטיקות לפתרון: (א) רבדים, overlays (ב) קישור דינאמי (ע"י קבצי dll), dynamic linking.

(א) ריבוד (overlays):
נחלק את התוכנית לחלקים וכל חלק נכניס לזכרון הפיזי. זה כמובן לא מתאים לכל התוכניות, שהרי אם למשל יש מודול שמכיל גם את הלוגיקה של ההידור (קומפילציה) וגם את הלוגיקה של האיסוף, הוא יהיה גדול מאד. לכן נפריד את המודול למספר רבדים, אחד יתעסק בהידור, אחד באיסוף. הם ירוצו באופן סדרתי ולכן נוכל בכל רגע נתון לטעון לזכרון רק אחד מהם. חסרון: המתכנת צריך להכיר את כמות הזכרון הפיזי כדי לדעת לתכנת כל חלק בקוד שלו בפני עצמו.
בנוסף, נצטרך כמובן גם driver, מנגנון שיודע לטעון את התוכנית בחלקים, רובד-רובד. למעשה, כל שלב בפעולת ההידור כולל מספר תת-שלבים.



דוגמה: נניח שגודל הזכרון הוא 150k. נניח שנרצה לטעון תוכנית שגודלה 210k, נחלק אותה לשני מעברים של 80k ושל 70k. נניח שיש אזורים משותפים לשני החלקים (מופיעים למעלה, סה"כ 60k), שצריכים להיות בזכרון בכל זמן ריצת התוכנית. היות וחילקנו את התוכנית לרבדים, גם כאשר נטען את הרובד הגדול יותר לזכרון, עדיין נזדקק סה"כ רק ל-140k, כך שנוכל להריץ כל חלק של התוכנית בזכרון הפיזי. לשיטת הרבדים יש חסרונות: לא כל תוכנית מתאימה לחלוקה כזו, המתכנת צריך לתכנת את הקוד שלו בשיטה זו, יש תקורה (עלות זמן) של החלפת הרובד הנטען לזכרון, גם אם התוכנית נטענת ברבדים עדיין היא צריכה להיטען במלואה בסופו של דבר כך שלא ייתכן שכל אחד מהרבדים יהיה גדול מדי.

(ב) קישור דינאמי (dynamic linking):
יתרון: לא צריך לבצע קישור של כל התוכנית מראש, מה שדרוש לצורך החלוקה לרבדים. נספק למתכנת טכניקה, כך שיוכל ליצור תוכנית שלא מקושרת במלואה בשלב הסטטי. נאפשר למתכנת לשלב רוטינות בצורה דינאמית. אם למשל יש לנו רוטינות (פונקציות) שמבצעות קלט-פלט וחישובים של נקודה-צפה, לא תמיד מזמנים (call) את כל הרוטינות מלכתחילה, אלא רק בשלבים מאוחרים יותר. לכן, דרושה לנו טכניקה להצבת הרוטינות בתוכנית עצמה רק בשלב מאוחר יותר, בשלב הריצה למשל. כך נוכל לחסוך ולקצר את זמן הטעינה של התוכנית ואת זמן ההבאה של הרוטינה לזכרון, והרי ייתכן והלוגיקה של התוכנית לא תגיע כלל לביצוע הרוטינה הזו וכך נחסוך. אם תוכנית אחרת שרצה בו-זמנית זימנה את הרוטינה הזו, רק נצטרך לאתר אותה בזכרון ולא לטעון מחדש, מה שיכול לחסוך כפילויות. כאשר מנסים להפעיל רוטינה שלא מופיעה בזכרון, יש מצב trap, ויש לנו קישור דינאמי שטוען את הרוטינה לזכרון באופן מיידי בזמן הריצה. יתרון: המתכנת לא צריך לדעת על הגבלת גודל הזכרון מראש. מערכת ההפעלה תצטרך כמובן לדעת לבצע קישור דינאמי, עליו מכריז המתכנת. [קבצי dll בהקשר של חלונות, ביוניקס יש SO – shared objects]

<u>דרישות לצורך ניהול זכרון</u>:

(א) מיקום מחדש / הזחה (relocation) – מדיניות המיקום, היכן ממקמים את התוכנית / המודול שלנו מלכתחילה. לפעמים יש צורך להזיז את התוכנית שלנו מהמיקום המקורי למיקום חדש, מאילוצים כגון: (1) המזמנן לטווח בינוני מוציא תוכנית לדיסק (ביצוע swap out) ומחזיר אותה בחזרה (swap-in) ואז ייתכן שהמיקום בו ישבה התוכנית קודם-לכן, תפוס כרגע. (2) כאשר מריצים מספר תוכניות במקביל נוצרים חורים בניצול הזכרון ובמקום שיווצרו הרבה חורים קטנים נרצה לאחד אותם לחור גדול כדי להיות מסוגלים להריץ תוכנית נוספת (מבצעים כיווץ, compaction).

(ב) הגנה על חלקי תוכנית / תוכנית שלמה – לכל תוכנית יש הרשאות גישה (נקרא RWX, read \ write \ execute) שאומרות אם ניתן רק לקרוא את הקובץ, אם ניתן לשנותו ואם ניתן להריצו כקובץ-ריצה. דרושה גם הגנה על מרחב המיעון: בדיקה למי מותר לגשת למרחב המיעון שהוקצה והאם ניתן לחרוג ממנו.

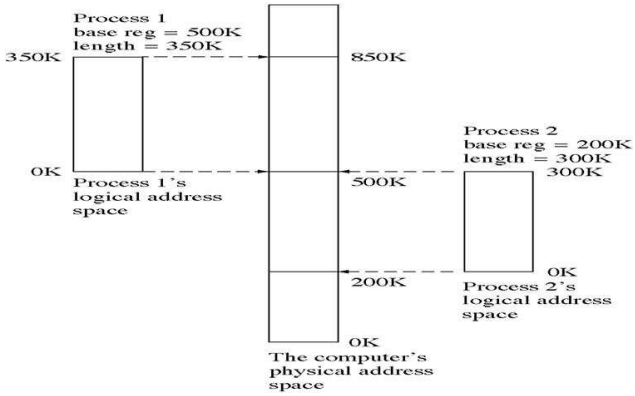
(ג) שיתוף של חלקי תוכנית / תוכנית שלמה – ייתכן ויש קטע קוד המשותף למספר תהליכונים במערכת. במקום שיהיו מספר העתקים פרטיים לכל תהליכון, נוכל פשוט לטעון את הקטע הזה פעם יחידה ולאפשר למספר תהליכונים להריץ אותו כדי לפתור בעיות של מחסור בזכרון. כמובן, הקוד צריך להיות read only כדי שנוכל לשתף אותו.

(ד) מיפוי המבנה הלוגי של התוכנית – ניתן להפריד בין יחידות הקוד (code segment) השונות וגם בין מקטעי הנתונים השונים (data segment), שחלקם יכולים להיות פרטיים וחלקם יכולים להיות ציבוריים. לכן, יש חשיבות היכן בדיוק ממקמים את כל המקטעים האלו. זה נקרא מידע-מיקומי, אנחנו צריכים לדעת היכן ממוקמים כל המקטעים השייכים לתוכנית, כדי לאפשר את הריצה התקינה שלה.

(ה) מיפוי המבנה הפיזי של התוכנית, בהתאם ליחידות שלה – היכן כל יחידה תושם (ברצף או במיקומים נפרדים).

Address type
כתובת פיזית היא מוחלטת, מתייחסת למקום בזכרון הפיזי, אינה ניתנת לשינוי. כתובת לוגית היא התייחסות של הזכרון הפיזי באופן עקיף. [בהקשר של זכרון פיזי נדבר על כתובת לוגית, בהקשר של זכרון וירטואלי נדבר על כתובת וירטואלית]

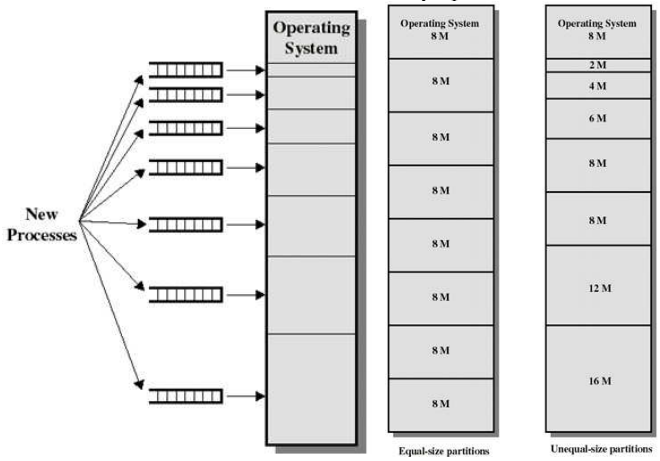
סוגים של מודולי הטעינה:
(א) מודול מוחלט – היו בהתחלה. מיד לאחר הפעולות הידור-איסוף-קישור-שילוב, כל המענים היו מוחלטים. גם המתאם היה מוחלט, כי היה משבץ כל מודול במיקום מוחלט ולא ניתן היה לשנות את מיקום המודול, אם היינו מוציאים אותו היינו צריכים אח"כ בהכנסה מחודשת להחזיר אותו למיקום המקורי. אם למשל הפקח-תושב היה 0-32k, אז כל המודולים היו מתחילים מ-32k והלאה. אבל בכל פעם שגודל הפקח-תושב ישתנה, נצטרך לשנות את המיקומים בהם יכולים להיות המודולים. זה לא גמיש!
(ב) מודול יחסי – כאשר מתאם לוגי-יחסי רוצה לטעון מודול, צריך לשנות את כל המודל היחסי למודול מוחלט, ע"י כך שיקבע שמעתה הוא יהיה בזכרון במיקום מסוים. כאשר המודול היחסי ניתן לזכרון, הוא הופך להיות מודול מוחלט. אמנם יש כאן מעט גמישות, אך עדיין לא מושלם. שיטה זו מאפשרת (בעקבות שחלוף / כיווץ) להזיח (למקם מחדש) מודולים בזכרון.
(ג) בעיה בשיטה הקודמת: אם המודול נמצא בזכרון באזור 48-64k אבל המשתנים שלו במיקום אחר, איך המודול ירוץ? ע"י שימוש באוגר בסיס (base register) שבזמן הריצה מוסיף היסט מתאים, כדי לשנות ממען יחסי-לוגי למען מוחלט וכך המודול יכול לרוץ. ההזזה של המודול נעשית ע"י MMU (memory management unit). אמנם התוכנית חושבת שהיא ניגשת למענים וירטואלים, אך בפועל היא ניגשת למענים פיזיים, בהם נמצאים ממש המשתנים / נתונים להם היא זקוקה.
מבחינה לוגית, ה-MMU הוא הבקר של הזכרון, אך ייתכן ויהיה על שבב-המעבד כדי לפעול באופן מהיר יותר, שהרי צריך שהוא יפעל בהתאם ללוגיקה של התוכנית. ה-MMU יודע לבצע הסטה למיעון המבוקש, ע"י שימוש ב-base register, כדי לגשת למען הפיזי בו נמצא המידע.
תזכורת מלפני כמה שיעורים: לא רק קו-הבסיס (חסם תחתון) חשוב, אלא גם החסם-העליון, כדי לבדוק שאנחנו לא ניגשים למיעון פיזי שאסור לנו לגשת אליו. לכן, נעשה שימוש גם באוגר limit register שבודק שהכתובת הפיזית שאנחנו מעוניינים לגשת אליה לא חורגת 'כלפי-מעלה' מטווח הכתובות הפיזיות שהוקצה לתוכנית שלנו. האוגר limit בעצם מהווה את האורך של טווח הזכרון שהוקצה לתוכנית. [בעמ' 15 מופיע איור שממחיש בצורה טובה]



מיפוי נקרא גם address binding ('קיבוע' של מיקום התוכנית).
(א) אם המיפוי נעשה בזמן ההידור, מקבלים בעצם קוד-מוחלט.
(ב) ניתן למפות בזמן הטעינה, נקבל קוד יחסי.
(ג) ניתן למפות בזמן הריצה, נקבל קוד שניתן למיקום מחדש (relocated code). כמובן שאם נשנה את מיקום הקוד, נצטרך לעדכן גם את ערכי ה-base register וה-limit register.
באיור מימין: דוגמה למיפוי מוחלט ויחסי.

[מצגת os7-2 rea]
Contiguous allocation
כרגע נניח שהתוכניות שלנו נטענות וממוקמות במיקום רציף בזכרון. דיברנו בעבר על חלוקה של הזכרון בין הפקח-תושב ותוכנית משתמש יחידה (memory split) או בין מספר תוכניות (memory division). [מופיע בעמ' 14-15]

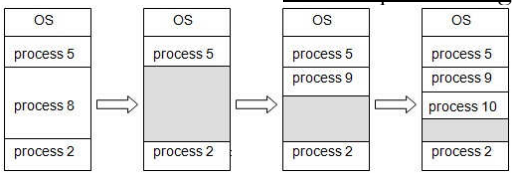
Fixed portioning:
חלוקה קבועה של הזכרון הפיזי, מתרחשת בזמן האתחול של המערכת. (ההשוואה בין הסוגים מופיעה באיור הימני)
(א) ניתן לבצע equal-size partitions כך שכל המחיצות באותו גודל. כמובן, יש הנחה סמויה שלא קיימת תוכנית הגדולה מגודל המחיצות שהגדרנו..
(ב) ניתן לבצע unequal-size partitions כך שכל מחיצה בגודל שונה. זו שיטה גמישה יותר. הרעיון הוא להתאים את גודל התוכנית לגודל המחיצה שמתאימה לה. אם מגיעה תוכנית בגודל מסוים, נשאף למקם אותה במחיצה שגודלה מאפשר מיקום של התוכנית תוך בזבוז כמה שפחות זכרון. ההנחה היא כמובן שכל מחיצה מאפשרת מיקום של תוכנית יחידה. פיצול פנימי (internal fragmentation) הוא מצב בו חלק מהמחיצה מבוזבז (אם למשל מיקמנו תוכנית בגודל 6M במחיצה שגודלה 12M).



החסרון: לכל גודל מחיצה אפשרי, נצטרך להשתמש בתור, שימקם את התוכניות במחיצות בעלות גודל זהה. כך, אם ניסינו למקם תוכנית בעלת גודל מסוים (למשל 10M) במחיצה מתאימה (למשל 10M) אך כל המחיצות הנ"ל תפוסות, נעביר את התוכנית לתור של מחיצות גדולות יותר (למשל 12M). (מופיע בחלק השמאלי באיור) שיפור: ניתן להשתמש בתור יחיד עבור כל המחיצות ולמקם את התוכנית במחיצה הראשונה הפנויה כך, שגודלה מאפשר למקם בה את התוכנית. כך נשיג יתרון: כל תוכנית נשאף למקם בגודל הקטן ביותר של מחיצה המאפשר למקם אותה בה, הגברת רמת המקביליות. החסרון: עלולים למקם תוכניות קטנות יחסית במחיצות גדולות יחסית, יהיה לנו בזבוז של מקום במחיצות.

דינאמיקה של fixed partitioning: אם כל המחיצות תפוסות, ניתן לבצע שחלוף-החוצה (swap out) כדי להעביר תוכנית לזכרון חיצוני ולפנות מחיצה לצורך מיקום של תוכנית חדשה.

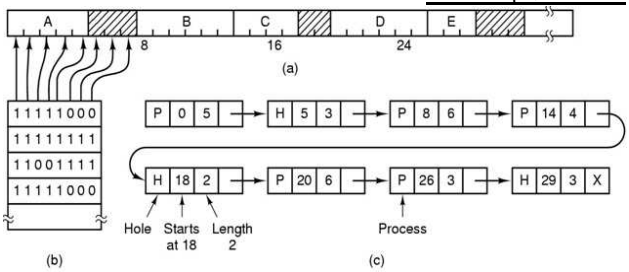
variable partitioning:



כאשר הבינו שחלוקת הזכרון הפיזי למחיצות מראש היא בעייתית, נוצרה שיטה זו: הזכרון הפיזי מלכתחילה ריק ובכל פעם שרוצים למקם תוכנית, נשנה את המחיצות בהתאם. (דוגמה באיור מימין) היתרון: במצב כזה כל מחיצה נוצרת באופן ייחודי לכל תוכנית, בגודל המתאים לה ואין לנו פרגמנטציה (פיצול, בזבוז זכרון) פנימית.

החסרון: יש פרגמנטציה חיצונית (פיצול חיצוני), כי עלולים ליצור שתי מחיצות שהמרחק ביניהן קטן יחסית, כך שלא יתאפשר למקם שם תוכנית נוספת. אמנם כל תוכנית מקבלת מחיצה המותאמת לה באופן מושלם, אך נוצרה לנו מחיצת-ביניים שלא ניתן למקם בה שום תוכנית והמקום הזה פשוט מבוזבז.

ניהול זכרון דינאמי:



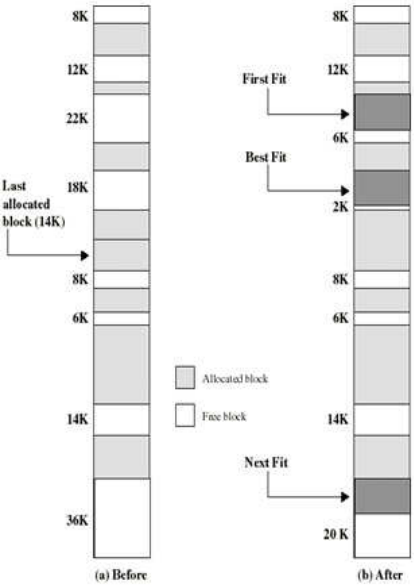
מערכת ההפעלה צריכה לבצע מעקב אודות המחיצות התפוסות והפנויות, כדי לדעת מתי נוצר פיצול פנימי / חיצוני ולתקן זאת. איור a מציין את המצב בזכרון הפיזי: שיטה b רק אומרת מה פנוי / תפוס. שיטה c טובה יותר, כי גם אומרת איזה מבין התהליכים תופס כל מחיצה תפוסה. ניתן כמובן להפריד לשתי 'שרשראות': של התפוסים ושל החורים. כדי למקם תוכנית חדשה, נצטרך לרוץ על שרשרת החורים ולחפש חור שגודלו מאפשר מיקום של התוכנית. שחלוף החוצה (swap out) עלול לגרום לפרגמנטציה חיצונית. שחלוף כמובן משבית את המערכת, כי בזמן ההזזה לא ניתן להריץ את הקוד שמוזז.

סיכום לגבי variable partitioning:
יש פיצול חיצוני, אין פיצול פנימי. כאשר נוצרים חורים, צריך לבצע כיווץ (compaction), שזו בעצם פעולת איחוד / מיזוג של כל החורים לכדי חור גדול, שאותו ניתן כמובן שוב לפצל למחיצות ולמקם בו תוכניות חדשות.

סיכום ביניים:
נמחיש מיפוי בין התפתחות סוגי המודולים וההתפתחויות בקביעת סוגי וגדלי המחיצות:

- מודול מוחלט ניתן להריץ רק במערכות הישנות.
- מודול יחסי-לוגי ניתן להריץ על מחיצות בעלות גודל קבוע.
- מודול מיקומי ניתן להריץ על variable partitioning, שמאפשר הזזה של המודול.

placement algorithm – הקצאה של מיקום בזכרון
כאשר מגיעה תוכנית חדשה ורוצים למקם אותה בזכרון הפיזי, יש מספר שיטות למיקום. מטרת כל השיטות כמובן 'לדחות את הקץ', לדחות את השלב בו גודל החורים הכולל גדל מדי ודרוש סידור מחדש של התוכניות בזכרון. השיטות:



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

(א) first-fit – נחפש את החור הראשון שמתאים למיקום התוכנית. נרוץ על רשימת החורים / מחיצות פנויות, נחפש את החור הראשון בעל גודל גדול/שווה לגודל התוכנית ונמקם בו את התוכנית. אם גודלו גדול מהדרוש, בעצם נוצר לנו חור קטן. מדיניות זו מנסה לחסוך זמן. החסרון: דווקא כן מבזבז זמן, כי בכל פעם אנו עוברים על רשימת החורים מתחילתה וייתכן שלאחר מספר הקצאות של תוכניות כל החורים הראשונים הפכו להיות קטנים מדי עבור כל תוכנית שנרצה לטעון לזכרון.

(ב) next-fit – בכל שלב נזכור מה החור האחרון שמיקמנו בו תוכנית, כדי שבפעם הבאה נעבור על רשימת החורים החל מהחור הבא ולא נתחיל את הבדיקה מתחילת הרשימה. בעצם, נתייחס לרשימת החורים כאל מעגלית וכך נחפש בעצם חור שהוא next-first-fit (= uniform distribution).

(ג) best-fit – נעבור על כל החורים ונחפש את החור שגודלו מאפשר למקם בו את התוכנית, כך שייווצר לנו חור פנימי קטן ככל האפשר, במקום למקם את התוכנית בחור הראשון שמוצאים שגודלו מאפשר את מיקום התוכנית בו.

(ד) worst-fit – נחפש את החור המתאים הגרוע ביותר. נמקם את התוכנית דווקא בתוך המחיצה הגדולה שפנויה, כדי שיישאר לנו חור מספיק גדול להקצות בו תוכניות נוספות. אמנם יכולנו למצוא חור קטן יותר שאולי יותר מתאים, אבל אם ייווצר לנו חור פנימי קטן מדי, לא נוכל למקם בו תוכנית נוספות. לעומת זאת אם ייווצר לנו חור מספיק גדול, נוכל למקם בו תוכנית נוספת. ע"פ חישובים סטטיסטיים, מדיניות זו היא אכן הגרועה ביותר, כי בה מבוזבזים החורים הגדולים ולכן תוכניות גדולות ידרשו כיווץ בשביל שיוכלו להיכנס (כיווץ דורש זמן).

שיטות ב+ג הן הטובות ביותר ולא ניתן להכריע ביניהן מי האופטימלית

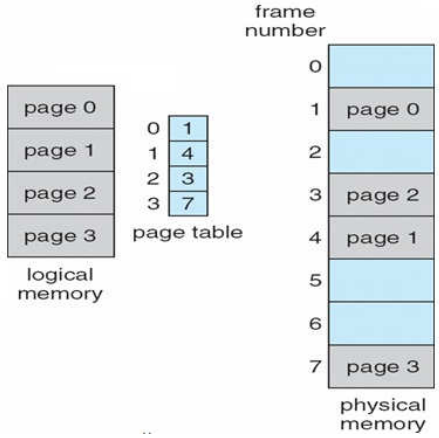
replacement algorithm – אלגוריתם למיקום מחדש של תוכניות
כאשר רוצים למקם תוכנית חדשה בזכרון הפיזי ואין לנו חור פנוי מספיק גדול, צריך לבצע החלטה חכמה איזה תוכנית נשחלף החוצה (swap out) לזכרון המשני.

שיטת Buddy system: כדי להחליט איזו תוכנית נשחלף, נבצע שילוב של השיטות best-fit ו- next-fit, כדי שלאחר השחלוף יהיה לנו מצב טוב יותר. נשאף להקצות מחיצות שגודלן 2^k (חזקה שלמה של 2), כאשר $L \le K \le U$ (כאשר L היא החזקה קטנה ביותר שנקצה, U החזקה הגדולה ביותר שנקצה שזה גודל הזכרון כולו, בעצם). כמובן, אם נצטרך להקצות מקום לתוכנית בגודל 100k, נקצה את החזקה השלמה הנמוכה ביותר של 2 שמאפשרת זאת, 128k. עבור כל גודל מחיצה אפשרי, נחזיק תור שבעזרתו נמקם את התוכניות בזכרון. בשיטה זו לכאורה אין פיצול פנימי, כי חיפשנו את המחיצה הקטנה ביותר האפשרית, אך עדיין חלק מהמחיצה יכול להיות לא בשימוש. אם נידרש למקם תוכנית בגודל מסוים (למשל 100k), אבל אין לנו מחיצה פנויה (למשל 128k) בגודל המתאים, נחפש מחיצה פנויה גדולה יותר (256k) ונחלק אותה. באותו אופן, ניתן גם לאחד מחיצות פנויות: אם נרצה למקם תוכנית בגודל פנויה 300k אך אין לנו מחיצה פנויה של 512k, נאחד שתי מחיצות פנויות של 256k. כמובן, נבצע איחוד / פיצול של מחיצות רק כאשר נידרש לכך. סטטיסטית: בממוצע יש פיצול פנימי של 25% (בזבוז של השטח!) וכל מחיצה תפוסה רק בכ-50% מגודלה. החסרון בשיטה זו: אמנם המחיצות לא קבועות, אך גדלי המחיצות הם קבועים (בהתאם לחזקה שלמה של 2) ולכן פיצול פנימי קיים. לכאורה אין כיווץ של זכרון באופן מפורש (בשום שלב לא מפסיקים למקם תוכניות ומאחדים את כל החורים לכדי חור גדול יחיד), אך פעולות איחוד / מיזוג של מחיצות שכנות המתבצעות באופן דינאמי עדיין גוזלות זמן. זו לכאורה שיטת ביניים בין מחיצות קבועות ובין מחיצות משתנות – כי אמנם יש לנו מיזוג / פיצול של מחיצות, אך גדלי המחיצות לא יכולים לקבל כל ערך.
[מומלץ לעבור על הדוגמה בשקופית מס' 24 במצגת]

[מצגת os7-3 rea, עד שקופית 5 כולל]

simple \ basic paging

פעולת paging – דפדוף בזכרון פיזי. כמו שבספר יש דפים וניתן לדפדף, ניתן להסתכל על הזכרון במחשב כעל ספר המכיל דפים, כך שניתן לדפדף בו. אנו לא זקוקים לספר שכל דפיו גלויים לנו בכל רגע נתון (כמו מגילה שניתן לפרוש), אלא רק לדף המסוים אותו נרצה לקרוא ברגע נתון. [אריאל מבקש להזכיר שבעבר לא היו לנו ספרים בצורתם הנוכחית, אלא היו מגילות ואז באמת כל הטקסט היה גלוי בו-זמנית.] [בדיחה יהודית: היהודים עובדים על דוס.] הרעיון: נחלק את הזכרון הפיזי לבלוקים בעלי גודל קבוע, להם נקרא frames, אך נפתח בפנינו את כל הספר (הזכרון). כל דף ייכנס לתוך frame. כל דף-לוגי ימופה לתוך frame-פיזי. כמובן, כל frame יהיה בגודל שהוא חזקה שלמה של 2. בהקשר של זכרון וירטואלי, גם את הזכרון הלוגי נחלק לבלוקים שווי גודל שלהם נקרא pages [למדנו על זה בתרגול עם מוטי..] ובכל שלב נתון נציג רק את הדפים להם אנחנו זקוקים.



בדומה למה שראינו בתרגול, יש טבלה שמסייעת לנו למפות בין הזכרון הלוגי והזכרון הפיזי. גודל הדף יכול להשתנות בין תוכניות / מערכות שונות. הערה: ניתן להתייחס לזה בדומה ל- fixed + equal-size portioning, כי כל הדפים הם גם שווי-גודל וגם מחולקים מראש. היתרון: פיצול פנימי לא יהיה. אמנם, בדף הלוגי האחרון יכול להיות חור קטן, כי ייתכן והגודל של התוכנית הוא לא חזקה שלמה של 2. פיצול חיצוני לא יהיה, כי כל דף-לוגי (page) מתאים בדיוק למסגרת-פיזית (frame), מבחינת הגודל.

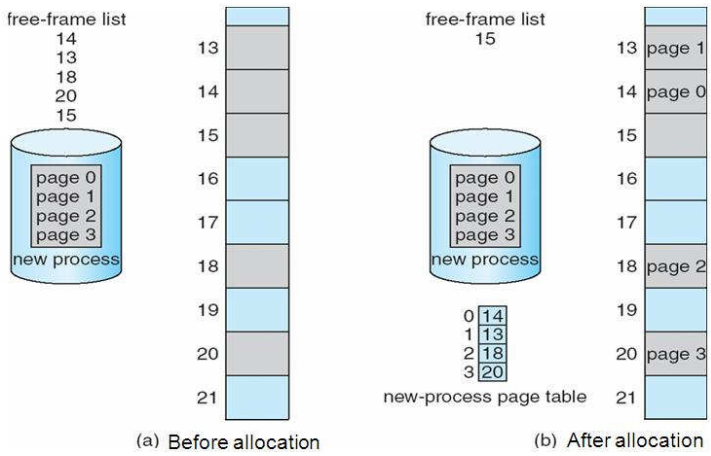
הרצאה מס' 12 - 4.1.10

[מצגת os7-3 rea, משיעור שעבר]

גם היום נדבר על ניהול זכרון. בשיעור הקודם דיברנו על מחיצות בזכרון, לצורך מיקום של תוכניות. בהתחלה הייתה חלוקה למחיצות בעלות גודל אחיד, כאשר החלוקה נעשתה מראש. בהמשך המחיצות חולקו לפי גדלים מוגדרים (חזקה שלמה של 2).
אח"כ דיברנו על מספר אלגוריתמים לשיבוץ תוכניות ב"חורים" פנויים בזכרון. בשיטת ה-buddy system שהזכרנו בשיעור שעבר, הנחנו שיש לטעון את התוכנית במלואה, באופן רציף, מלכתחילה. בסיום דיברנו על חלוקת הזכרון לדפים / מסגרות, שיטה בה אין לנו לא פיצול פנימי ולא פיצול חיצוני, עליה נדבר גם היום.

הנחנו שיש לנו טבלה שנקראת page table (מופיעה בדוגמה בסוף שיעור קודם), המאפשרת לנו להשתמש באינדקס הלוגי כדי למצוא את המספר של המסגרת הפיזי הרצוי לנו. בכך הורדנו את האילוץ שכל התוכנית צריכה לשבת בזכרון בצורה רציפה. אך האילוץ שכל הדפים נמצאים מלכתחילה בזכרון, עדיין נשאר. טבלת האינדקסים, צריכה גם היא לשבת בזכרון הפיזי, במסגרת כלשהי. אנו מניחים שיש לנו אוגר ייעודי (dedicated register) שיודע לגשת לטבלה הזו ולהחזיר לנו את הכתובת של המסגרת הפיזית הרצויה לנו.

במסגרת בניית ההקשר של התהליך, ה-loader צריך לקבל הרשאה מתת-מערכת ניהול הזכרון הפיזי, כדי לגשת למרחב הזכרון החופשי ולטעון את התוכנית הספציפית. בנוסף, הוא צריך לתחל את ה-page table, שתקשר בין הדפים הלוגיים ובין המסגרות הפיזיות, המשמשות את התוכנית.

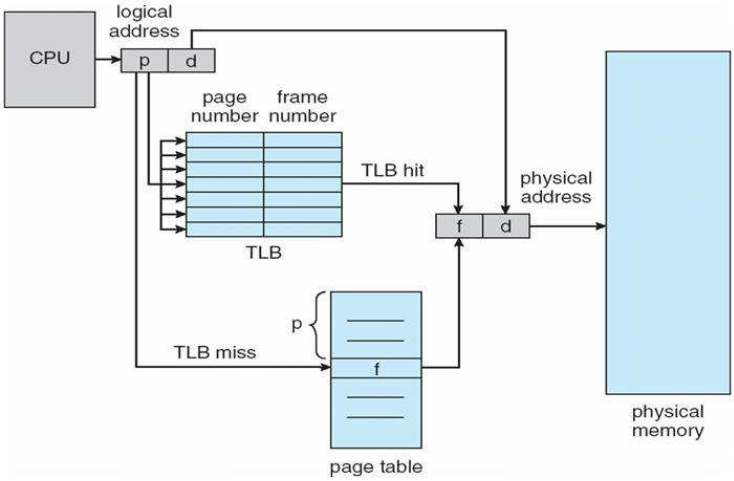


תזכורת משיעור קודם: גודל של דף לוגי שווה בדיוק לגודל של מסגרת פיזית. לכן, אם תוכנית דורשת מקום של n דפים לוגיים, היא כמובן תדרוש n מסגרות פיזיות. לצורך מיקום של תוכנית חדשה בזכרון, יש לעבור על רשימת המסגרות הפנויות.

בכל מסגרת פנויה, נוכל למקם דף-לוגי. אם יש מספר מסגרות פיזיות פנויות ברצף, ניתן כמובן לטעון מספר דפים ברצף לתוך מסגרות אלו. כך ניתן יהיה לקרוא את התוכנית ברצף (שיפור מסוים במהירות, אריאל לא התעכב על זה).

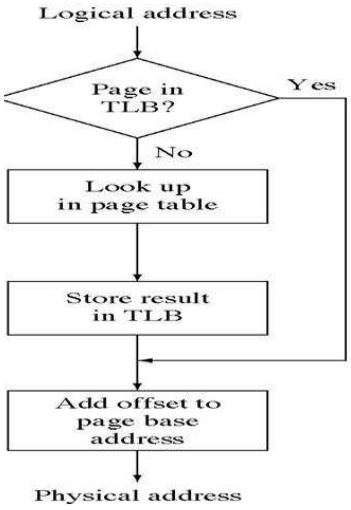
ב-page table, כל אינדקס המקשר בין דף למסגרת, צריך להיות מספר שנחלק לשני חלקים: (א) הכתובת הפיזית p, מספר המסגרת הפיזית הרצויה (ב) ההיסט (offset) היחסי d בתוך הדף, כדי שנוכל לגשת למיקום מסוים בתוך הדף הפיזי. באופן כזה אנחנו ממירים ממען לוגי-יחסי בדף מסוים של התוכנית, לשורה ספציפית במסגרת הפיזית. למעשה, ב-page table ניגש לאינדקס המתאים, נשלוף את כתובת המסגרת הפיזית הרצויה ונשרשר אליה את ההיסט. באופן כזה אנחנו ממפים כתובת לוגית (p,d) שמורכבת ממספר דף וההיסט, לכתובת פיזית (f,d) שמורכבת ממספר מסגרת פיזית והיסט. את המרכיב של מספר הדפים, ניתן לייצג ע"י מספר בגודל log2 ביטים, של מספר הדפים. בהתאם, את המרכיב של ההיסט, בתוך המסגרת הפיזית, ניתן לייצג ע"י מספר בגודל log2 של גודל המסגרת הפיזית. אם יש לנו 8 דפים וגודל דף הוא 4, נצטרך סה"כ מען לוגי של 5 (3 ביטים עבור מספר הדפים כי יש לנו 8 מסגרות פיזיות, ו-2 ביטים עבור ההיסט כי גודל כל מסגרת הוא 4). בדוגמה בשקופית מס' 17: כדי לייצג את j-9, המען הלוגי יהיה 010,01 (החלק השמאלי מציין את מספר הדף – 2, החלק הימני מציין את ההיסט – 1). לאחר שניעזר ב- page table, המען שלנו ישתנה ל-001,01 (כי הדף הלוגי 2 ממופה למסגרת מס' 1 ובנוסף יש לנו היסט של 1).

המערכת צריכה לדעת היכן ממוקמת ה-page table בזכרון. לצורך כך נניח שיש לנו שני אוגרים: (א) PTBR, אוגר יעודי שהוא מצביע ל-page table (כתובת-הבסיס של הטבלה, נקרא page-table base register) (ב) PTLR, אוגר שמציין את גודל הטבלה (נקרא page table limit register). (א) פתרון ראשון: נניח שהאוגר PTBR נמצא בזכרון הפיזי ואז נוכל למען כתובת לוגית לכתובת פיזית בקלות יחסית. אנו מניחים זאת, שהרי אם PTBR היה בזכרון הלוגי, היה צריך למפות גם את הטבלה.. מה שיגרום לשרשרת של מיפויים לוגיים, עד שנגיע למסגרת הפיזית הרצויה. לעומת זאת, אם ה-page table נמצאת בזכרון הפיזי, כל גישה לזכרון פיזי של תוכנית תדרוש 2 פעולות: אחת כדי להגיע למסגרת בה נמצאת הטבלה ונוספת כדי למצוא בטבלה את המסגרת הפיזית שאליה רצינו לגשת מלכתחילה. בעצם הדבר יגרום להכפלת הזמן הדרוש לנו עבור כלל המיפויים הנחוצים לריצה של תוכנית. (ב) פתרון שני: אולי בכלל כדאי לשים את ה-page table על החומרה, ב-MMU. באופן כזה, נבצע פעולה יחידה כדי לגשת למסגרת הפיזית הרצויה לנו, המיפוי ייעשה על השבב של המעבד עצמו (פעולה מהירה יותר, ביחס גישה לזכרון). החסרון: כמות הזכרון ב-MMU היא מוגבלת ולכן אם הטבלה גדולה מדי, הפעולה תהיה יקרה מדי. (ג) פתרון שלישי (ונכון יותר) הוא סוג של פשרה בין השיטות הקודמות: TLB, translation look-aside buffer (טבלת דפים צדדית). הטבלה תיבנה בטכנולוגיה של זכרון מסומך (associative memory) של 2 עמודות: השמאלית תהיה מספר העמוד הלוגי, הימנית תהיה מספר המסגרת הפיזית. הטבלה היא חלקית בלבד, מטרתה לאפשר מיפוי מהיר.



ההנחה היא שגודל טבלת ה-TLB קטן יותר ביחס לגודל ה-page table המלאה. הטבלה תשמור לנו את המיפויים שעשינו בהם שימוש לאחרונה, מתוך הנחה שיש דפים שאליהם אנו ניגשים בתדירות גבוהה יותר ביחס לדפים אחרים (חובבני 'מבני נתונים' בטח נזכרים עכשיו בעץ splay ומחייכים ☺). חיפוש של מען לוגי ב-TLB נעשה במקביל, שכן במקביל אנחנו יכולים לבדוק את כל הכניסות בטבלה. אחוז ההצלחה בגישה במציאת המיפוי המתאים ב-TLB, נקרא יחס-פגיעה (hit-ratio). המשלים ליחס-הפגיעה נקרא יחס-הפספוס (miss-ratio).

בטבלה יהיו לנו צמדים של מיפויים. כאשר יש לנו מען לוגי, נריץ בקשה במקביל גם ל-TLB (אופטימי, בתקווה שנצליח למצאו שם) וגם ל-page table (פסימי, במקרה שנפספס). אם מצאנו את המען הפיזי הרצוי לנו ב-page table (בדומה לעץ splay...) נכניס את הכניסה הזו ל-TLB, למקרה שנרצה לחפש אותה שוב בעתיד. (לא נתעמק בשיטות לדריסה של כניסות ב-TLB, במקרה שהטבלה הזו כבר מלאה).



לא נוכל להגיע ליחס פגיעות של 100%: (א) בהתחלה ה-TLB ריקה, כי לא נדע לאילו דפים לוגיים התוכנית תרצה לגשת (מלבד כמובן הדף הלוגי של ה-main כי משם התוכנית מתחילה לרוץ). (ב) גם כאשר ה-TLB מתמלאת, מדי פעם ניגשים לדף שלא נמצא ב-TLB. במערכות מתקדמות מגיעים ליחס פגיעה של 90%.

באיור מימין, מוצגת הלוגיקה של חיפוש מען פיזי. טבלת ה-TLB עובדת על סמך עקרון המיקומיות. ברוב המערכות יש טבלת TLB יחידה על ה-MMU. כאשר יש מיתוג תהליכים, נצטרך לשמור את תוכן ה-TLB בהקשר של התהליך שמופסק (מתוך הנחה שימשיך להשתמש במיפויים האלו גם כאשר הוא יחזור לרוץ). עוד סיבה מדוע מיתוג הקשר יקר כ"כ. בהתאם, צריך גם לטעון את ה-TLB של התהליך שקיבל עכשיו את השליטה, בעקבות הפסיקה.

זמן גישה אפקטיבי effective access time (EAT)

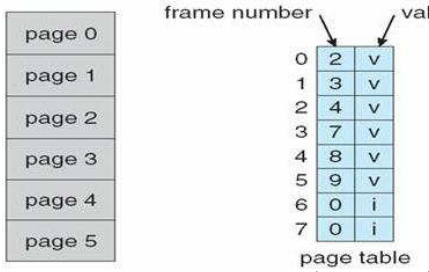
נניח שאנו פועלים בשיטה של basic paging (עליה דיברנו עד כה). זמן הגישה נע בין 2 (המען לא היה ב-TLB) ל-1 (כן היה). נרצה כמובן שהזמן הדרוש לאיתור המען המבוקש ישאף ל-1. זמן האיתור ב-TLB יסומן כ- ε ואילו יחס הפגיעה כ- α.

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

המחובר השמאלי במשוואה מציין את החישוב במקרה של פגיעה (גישה אחת כפול יחס הפגיעה). המחובר הימני במשוואה מציין את החישוב במקרה של פספוס (שתי גישות, כפול המשלים של יחס הפגיעה). אם יחס הפגיעה קרוב ל-1 (100%), ערך ה- EAT יהיה קרוב ל-1. אם יחס הפגיעה קרוב ל-0 (0%), ערך ה- EAT יהיה קרוב ל-2. כמובן, החישוב תקף כאשר הגישה ל-TLB היא **באופן סדרתי** ביחס לגישה ל-page table (כך שתחילה אנו מחפשים ב-TLB ורק אם לא מצאנו נמשיך לחפש ב-page table). אך אם הבקשות נעשות במקביל (כך שאם מצאנו ב-TLB נבטל את הבקשה ל-page table) כדי לחסוך זמן במקרה של פספוס, החישוב של ה-EAT יכלול רק את החלק הימני במשוואה והתוצאה תהיה $2 - \varepsilon$. [מומלץ לעבור על דוגמת החישוב בשקופית מס' 24 במצגת]

- Assume memory cycle time is 100 nanosecond.
- Associative memory lookup = 20 nanosecond.
- Hit ratio = 80% – EAT = 0.80 x 120 + 0.20 x 220 = 140 ns – So 40% slowdown in memory access time.

הגנה על הזכרון memory protection



עד כה הנחנו שגודל טבלת הדפים הוא כגודל המרחב הלוגי. ייתכן שחלק מהכניסות בעלות מיפוי תקף אך חלקן לא. אם מספר המסגרות שתוכנית זקוקה לה קטן מגודל טבלת הדפים המינימלית שביכולתנו להקצות, נצטרך לציין עבור כל כניסה בטבלה, אם הכתובת תקפה או לא (ע"י הוספת ביט שיהווה דגל בוליאני). לפני שנתרגם מיפוי לוגי למיפוי פיזי, תחילה נצטרך לבדוק את ערך הביט שיגיד לנו אם הכתובת הלוגית הזו אכן תקפה.

באיור: הטבלה (משמאל) מציינת שהתוכנית המסוימת זקוקה ל-6 דפים לוגיים, בעוד (מימין) הטבלה שביכולתנו להקצות מכילה 8 כניסות. לכן, עבור חלק מהכניסות ביט הדגל יציין שהן לא תקפות (invalid).

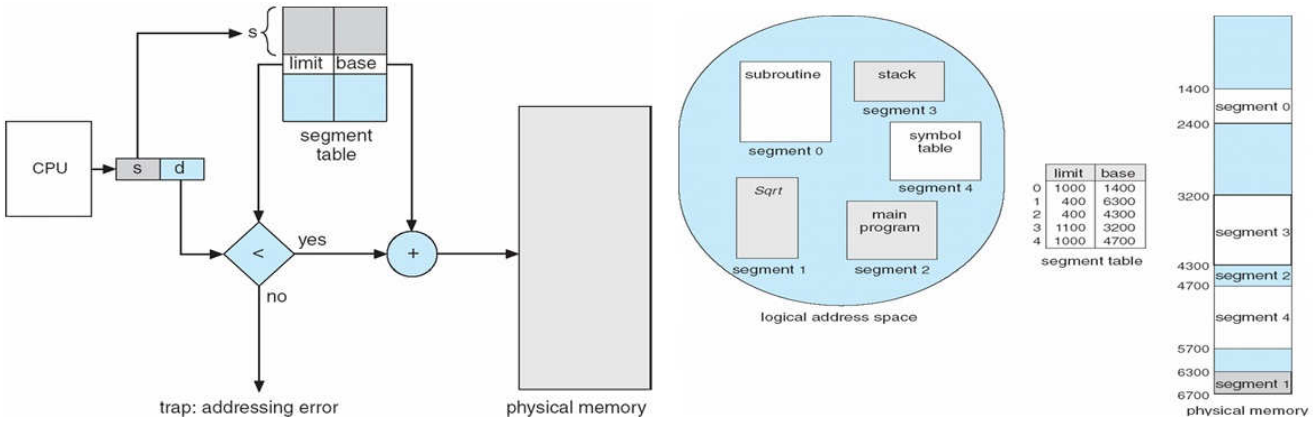
חשוב לזכור: כאשר משחלפים תוכנית מהזכרון הפיזי לזכרון המשני (למשל לצורך טעינה של תוכנית חדשה), צריך למצוא תחילה מסגרות פנויות בזכרון המשני ולתחל את טבלת הדפים לערכים החדשים.

שיתוף דפים (למשל בין תהליכים): נניח שנרצה לפתוח שני עותקים של עורך-תמלילים. אם כל תהליך יזדקק גם לעותק משלו של העורך וגם כמובן לנתונים שלו, נצטרך מקום רב, כי את הקוד של העורך נצטרך לטעון למספר מסגרות הזהה למספר התהליכים. לעומת זאת, אם הקוד של העורך עצמו הוא משותף, כך שמספר תהליכים יכולים להריץ אותו, נבצע שיתוף. כך נוכל לחסוך שימוש במסגרות פיזיות ובעצם לחסוך זכרון. בהתאם, בטבלאות ה-page table של שני התהליכים יהיה מיפוי לאותה מסגרת פיזית.

[מצגת os7-4 rea, עד שקופית 22 כולל]

קיטוע בסיסי simple \ basic segmentation

ישנם חסרונות בטכניקה הקודמת של דפדוף, בהיבטים של שיתוף והגנה. היות והחלוקה הלוגית לדפים היא שרירותית (נניח דפים בגודל k), הבעיה היא שניתן לשתף דף מלא ולא חלק מדף. אך ייתכן הרי שקטע קוד שנרצה לשתף (לאפשר למספר תהליכים להריץ), תופס רק חלק מדף ואולי אפילו נחלק, כך שהוא מופיע בסוף של דף אחד ובהתחלה של דף אחר. במקום להסתכל על התוכנית של המשתמש כעל אוסף של יחידות תחביריות בעלות גדלים קבועים, נסתכל עליהן כבעלות גדלים משתנים: במקום לחלק את הזכרון הפיזי למחיצות בעלות גודל קבוע, ננהל ניהול מחיצות משתנה, נקצה מחיצות בגדלים שונים, בהתאם לצורך. אמנם תשתית התמיכה יקרה יותר, ביחס לזו הדרושה לתמיכה במסגרות פיזיות שוות-גודל, אך כעת הדפדוף (לרבות ההיבטים של שיתוף והגנה) טבעי יותר: אם נרצה לשתף קטע קוד בגודל ספציפי, נוכל לשתף מסגרת פיזית ספציפית ולא מסגרת שאולי גדולה יותר, כמו בשיטה הקודמת.



בכל כניסה בטבלת ה-segment table נצטרך לשמור את הבסיס של הכניסה ואת האורך שלה (כי ייתכן והאורכים שונים, בהתאם לגודל המבוקש לנו). האורך דרוש לנו גם כדי לא לזלוג החוצה מהמקטע. בהתאם לאינדקס בטבלת המקטעים, אנחנו מקבלים את כתובת הבסיס של המחיצה הרצויה ומוסיפים את ההיסט כדי להגיע למיקום הפיזי הרצוי. כמובן, מתבצעת בדיקה אם בתוספת של ההיסט, אנחנו חורגים מהאורך (limit) של המחיצה הספציפית. הכתובת הלוגית מורכבת מ-s,d כאשר s מציין segment.

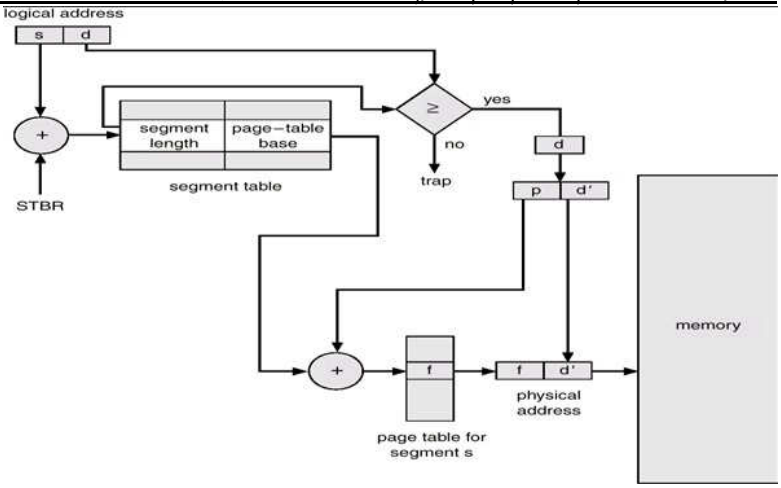
השוואה בין טכניקות הדפדוף והקיטוע:

(1) דפדוף מתבצע באופן שקוף, החלוקה היא באופן שרירותי. הקיטוע נראה, כי המתכנת קובע את גודל היחידות בהן הוא עושה שימוש, לכן המתכנת גם מודע לכך שהיחידות יכולות להיות משותפות ומוגנות באופן טבעי.

(2) מקטעים בעלי גדלים משתנים, בעוד דפים בעלי גודל קבוע. בדפדוף מקבלים תמיכה של החומרה, בזכות העובדה שהגודל קבוע. לעומת זאת, התמיכה למקטעים נעשית ברמת התוכנה, בשל ההקצאה הדינאמית של הגדלים המשתנים.

(3) בדפדוף אין בעיה של פיצול פנימי / חיצוני. במקטעים אין פיצול פנימי (הקצאת המקום בהתאם לצורך) אך יש פיצול חיצוני, כי ייתכן ויווצרו לנו חורים בין מחיצות, שהיה ניתן לנצלם לצורך מיקום של תוכנית נוספת.

מדוע שלא נשלב בין דפדוף וקיטוע, כדי ליהנות מהיתרונות של שתי השיטות?



מערכת ההפעלה MULTICS (מופיעה במצגת של שיעור ההיסטוריה מתחילת הסמסטר..) ניסתה לאחד את השיטות. זו המערכת הראשונה שנכתבה בשפה עילית. נניח שגודל המקטע גדול מגודל הדף. לכן, הגיוני לחלק את המקטע לדפים ולא את הדף למקטעים. נקבל מדרג ניהול זכרון בשתי רמות: מלכתחילה המרחב הלוגי נחלק למקטעים ובחלוקה נוספת כל מקטע נחלק לדפים.

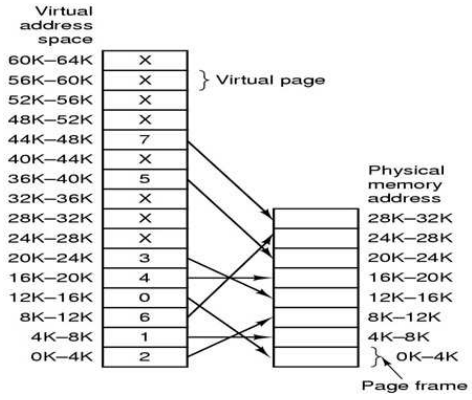
אם נצטרך לגשת למקטע מסוים, אנחנו צריכים לגשת רק לדף מסוים בתוך המקטע. בהתאם, המען הלוגי נחלק ל-3: (א) מרכיב המקטע (ב) מרכיב הדף בתוך המקטע (ג) מרכיב ההיסט בתוך הדף.

בשלב הראשון יש לנו מקטע והיסט בתוך המקטע. בטבלת המקטעים (גם כאן יש אוגר STBR שיודע איפה תחילת הטבלה), בכניסה המתאימה, יהיה לנו מצביע לטבלת הדפים השייכים למקטע. בשלב השני נבצע המרה של ההיסט בתוך המקטע, להיסט בתוך הדף ונחבר אותו למספר המסגרת הפיזית הרצויה לנו. יש לנו בדיקה בשלב הראשון, שאכן ההיסט לדף הרצוי במקטע, לא חורג מגודל המקטע אליו אנו ניגשים. נניח יש לנו גם טבלת מקטעים צדדית וגם טבלת דפים צדדית. במקרה כזה, החסרון הוא שחיפוש ידרוש לנו 3 פעולות: ראשונה בטבלת המקטעים שב-MMU (בדומה ל-TLB), שנייה בטבלת המקטעים שבזכרון ושלישית בטבלת הדפים של המקטע הרצוי. היתרון: כל הפעולות של שיתוף והגנה נעשות ברמה העליונה של מקטעים ואילו כל פעולות הדפדוף נעשות ברמה התחתונה של טבלת הדפים, ברמה יעילה.

[מצגת vir os8-1, עד שקופית 20 כולל]

ניהול זכרון וירטואלי virtual memory management

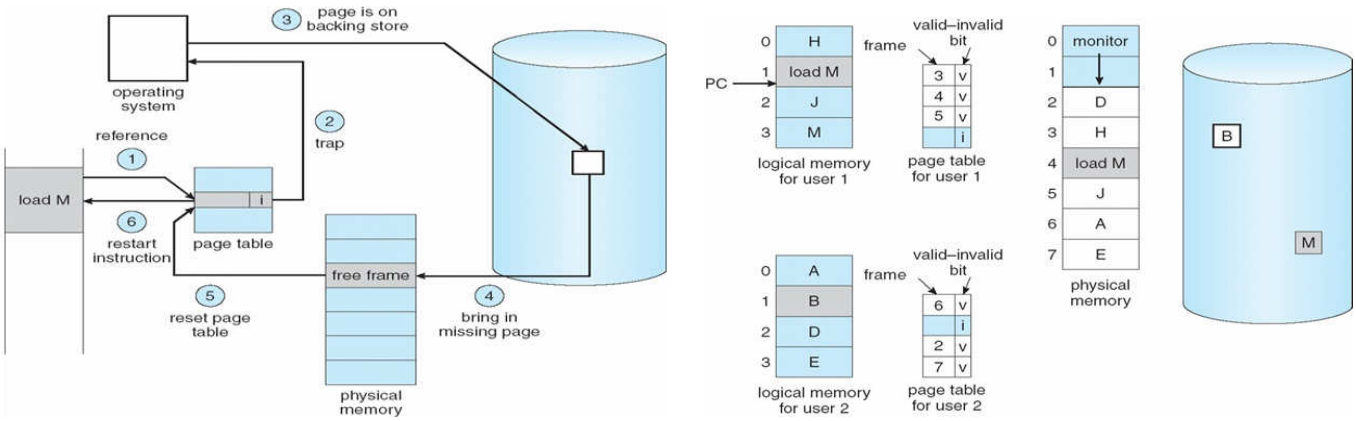
עוד קודם לכן, ביטלנו את האילוץ שהתוכנית צריכה להיטען לזכרון ברצף, כך שניתן להכניסה בחלקים, לתוך אזורים פיזיים (בין אם מדובר בדפים ובין אם מדובר במקטעים). כעת נבטל גם את הדרישה שכל הדפים הדרושים לתוכנית צריכים להיות טעונים בזכרון בכל זמן ריצת התוכנית. במקום זאת, נדרוש שרק הדפים הדרושים לנו באותו שלב בריצת התוכנית, יהיו זמינים לנו בזכרון (בדומה לקריאה בספר, רק הדף אותו אנחנו קוראים צריך להיות נגיש לנו). ההחלטה לדפדף ברמה של דפים היא פשרה: קיצוניות אחת היא שכל התוכנית תהיה בזכרון בכל רגע נתון, הקיצוניות השנייה היא לדפדף ברמת מילים(!) או שורות בקוד שלנו. הפשרה מאפשרת לנו שכל תוכנית, עד כמה שתהיה גדולה, תדרוש בכל זמן נתון רק מספר קטן של מסגרות פיזיות / מחיצות (בשיטה של קיטוע). (באופן לא מפתיע..) שיטה זו נקראת on-demand, כי למעשה הדפים נטענים לזכרון רק ברגע בו הם נדרשים לצורך ריצת התוכנית. במהלך ריצת התוכנית, ייתכן וחלק מהדפים של התוכנית לא ייטענו לזכרון כלל.



אם בוצעה טעינה של דף לזכרון, שבהמשך השתנה בעקבות כתיבה עליו, כשנשחלף אותו החוצה (swap out), נשמור אותו באזור זמני של דפים 'מלוכלכים'. כאשר נצטרך לטעון מחדש לזכרון את הדף הזה, נטען אותו מהאזור ה'מלוכלך' ולא מהאזור בו נמצאים כל הדפים של התוכנית.

לא משנה מה גודל התוכנית / מספר הדפים שהיא דורשת, כי בכל זמן נתון הזכרון הפיזי קטן יותר ביחס לגודל התוכנית, עדיין נוכל לטעון אותה ולהריצה. החסרון: מנגנון הדפדוף דורש תקורה של טעינה / הורדה של דפים. היתרון: רק חלק מהתוכנית נטען לזכרון, מה שמאפשר להריץ תוכניות גדולות בו-זמנית.

נניח שתוכנית רצה והיא נדרשת לדף שלא נמצא כרגע בזכרון. נצטרך לבצע 'עצירה' לתוכנית, יתבצע מיתוג-הקשר שיעביר את השליטה לידי ה'משגוח', מערכת ההפעלה תטען את הדף הדרוש on-demand לזכרון (בין אם הוא שמור על קובץ כלשהו בכונן הקשיח, או באזור ה'מלוכלך'), יתבצע עדכון לטבלת הדפים וכמובן בסיום נחזיר את השליטה לתוכנית. [בדומה לדוגמה המופיעה בעמוד 41] נוסיף את הדגל הבוליאני שאומר האם מסגרת רצויה היא תקפה או לא (עבור דף שלא נמצא בזכרון באותו זמן, הביט יציין שהוא invalid).



בדוגמה מימין, יש לנו שתי תוכניות שרצות. בדיסק, נדגיש את הדפים B ו-M, כדי לציין שהם לא טעונים לזכרון כרגע. לכל אחת מהתוכניות (user 1, user 2) יש דף אחד שלא ממופה. האוגר PC (program counter) שמציין את המען של הפקודה הבאה שמתבצעת, מצביע על הדף הלוגי מס' 1 של התוכנית העליונה. בדף זה כתוב שצריך לטעון את הדף הפיזי M (ייתכן והתוכנית מנסה להפעיל פונקציה או לגשת לערך של משתנה, הכתובים בדף זה). היות והדף הפיזי M לא נמצא בזכרון, כביכול ניסיון לגשת למען לוגי-חוקי (מבחינת המרחב הלוגי), שאינו מען פיזי-חוקי (כי הדף לא טעון כרגע לזכרון). תקלה זו נקראת תקלת-דף, page fault. המערכת תצטרך לתפוס שליטה, להשהות את התוכנית שביקשה את הדף החסר, לטעון את הדף הרצוי, לעדכן את טבלת הדפים ובסיום להחזיר שליטה לתוכנית. התרשים משמאל, מכיל פירוט של אופן הטיפול בתקלת דף (נקראת גם page exception). בביצוע פקודה יחידה בתוכנית, ייתכנו מספר תקלות-דף (אחת לפקודה עצמה ואחת עבור כל משתנה בו נעשה שימוש בפקודה). בנוסף, אם כל אחד מהמשתנים הוא לא גישה ישירה, אלא מצביע למשתנה אחר, ייתכנו לנו עוד יותר תקלות-דפים (תחילה נטען את הדף של המצביע ורק בהמשך את הדף שעליו מצביע המצביע ☺). כמובן, ייתכן ומישהו שחלף לנו החוצה דף ביניים, בזמן שניסינו להביא דף המשך. נצטרך תחילה לטעון מחדש את דף הביניים (למשל במקרה של מצביע) ורק אז לטעון את הדף שהיה רצוי מלכתחילה.

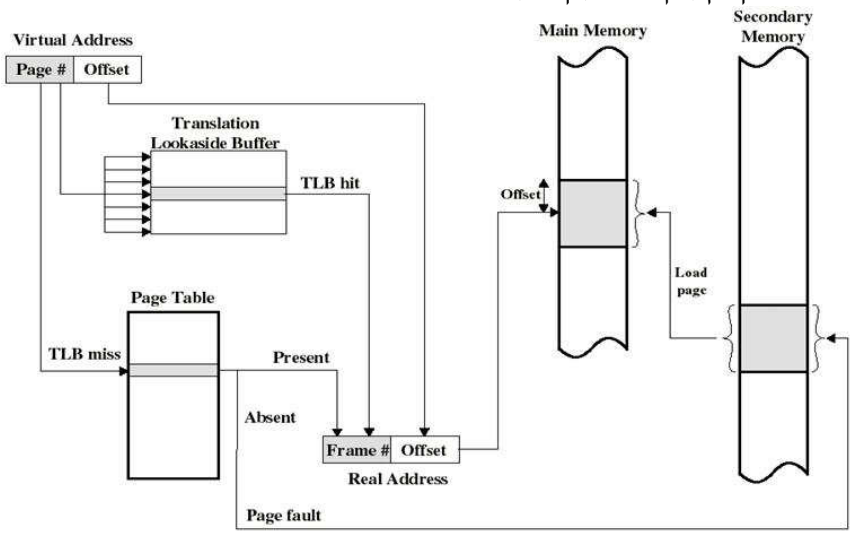
הרצאה מס' 13 - 11.1.10

[מצגת os8-1 vir, משקופית 19]

Address translation in paging system

חזרה על הסוף של שיעור שעבר: לכל אחת מהתוכניות הטעונה בזכרון, יש 3 דפים-לוגיים טעונים בזכרון. ה-program counter מצביע על דף לוגי מס' 1 של התוכנית העליונה, מה שאומר שאנחנו צריכים לטעון את הדף הלוגי מס' 3. בטבלת ה-page table של התוכנית, אנחנו מגלים שהדף הלוגי לא טעון למסגרת פיזית, הכניסה לא תקפה. במצב כזה יש לנו מלכודת. המזהה הייחודי של התקלה מגלה שמדובר בתקלת-דף (page-fault). לאחר מכן הדף מאותר בטבלה של המערכת (הוא יכול להגיע או מהקובץ המקורי בדיסק או מהאזור של הדפים ה'מלוכלכים'). נניח כמובן שיש לנו סיבית dirty-bit שאומרת אם הדף נקי / השתנה מרגע טעינתו לזכרון. לאחר טעינת המסגרת הפיזית לזכרון, נצטרך לעדכן את ה-page table ולסמן שהכניסה כן תקפה. לאחר כל זאת, נכניס את התוכנית שנזקקה לדף החסר לתור התוכניות שהן ready.

הבעיה מתחילה כאשר אין לנו מסגרת פנויה בזכרון. במקרה כזה, נצטרך לשחלף-החוצה מסגרת כלשהי, כדי לפנות מקום בזכרון הפיזי למסגרת הדרושה לנו כרגע. נצטרך למצוא דף-קורבן שאותו נשחלף החוצה מהזכרון, כדי לפנות מקום לדף החדש. לפני שנוציא את הדף הנבחר החוצה, נעדכן את טבלת הדפים של התוכנית ונסמן שהכניסה לא תקפה. נבצע swap out victim page. לאחר מכן נכניס את הדף הרצוי ונעדכן בכניסה המתאימה של הדף הלוגי ב-page table שהכניסה תקפה. בהמשך נרחיב על השיטה למציאת דף-קורבן שישוחלף החוצה.



האפשרות הראשונה להמרת כתובת לוגית לכתובת פיזית, היא לחפש ב-TLB (הטבלה הצדדית). אם לא מצאנו שם, נחפש ב- page table. השוני בין זכרון ממשי לזכרון וירטואלי, הוא שאם גם שם לא מצאנו, יש לנו שגיאת-דף (page-fault) ונצטרך להביא את הדף מהדיסק.

תזכורת לגבי TLB: כל הכניסות בטבלה הזו הן תקפות, אחרת ניהול הזכרון שלנו לא יעיל ☹. כאשר אנחנו משחלפים החוצה דף מהזכרון לדיסק, צריך כמובן לבדוק אם הוא נמצא ב-TLB ו'לנקות' אותו משם.

דוגמה לטבלת TLB: יש מגוון נתונים, לא רק כתובת וירטואלית וכתובת פיזית, אלא גם מידע אם הדף 'מלוכלך' והרשאות הגישה.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31

חישוב הערך EAT (effective access time) במקרה של demand paging: כאשר יש לנו תקלת-דף וצריך להביא את הדף הדרוש, יש לנו מחיר I\O (פעולת קלט-פלט מהדיסק) לשלם. כמובן, אם הדף 'מלוכלך' ולא נביאו מהדיסק, נצטרך לשלם מחיר כפול: תחילה כתיבה של הדף החוצה לאזור ה'מלוכלכים' ואז קריאה של הדף החדש פנימה. נסמן באות p את ההסתברות שיש תקלת דף (0 = אין, 1 = יש). במקרה ולא התרחשה תקלת-דף, נשלם רק מחיר של גישה לזכרון (hit). אם יש תקלת-דף, נצטרך לשלם מחיר של שחלוף החוצה ושחלוף פנימה (בנוסף גם מחיר זניח של החזרת התוכנית לתור התוכניות שרוצות לרוץ).

$$EAT = (1-p) \times \text{memory access} + p \times (\text{page fault overhead} + [\text{swap page out}] + \text{swap page in}$$

(הסוגריים המרובעים, מציינים שאת עלות הפעולה של שחלוף-החוצה, נשלם רק במידה ואין לנו מסגרת פנויה)

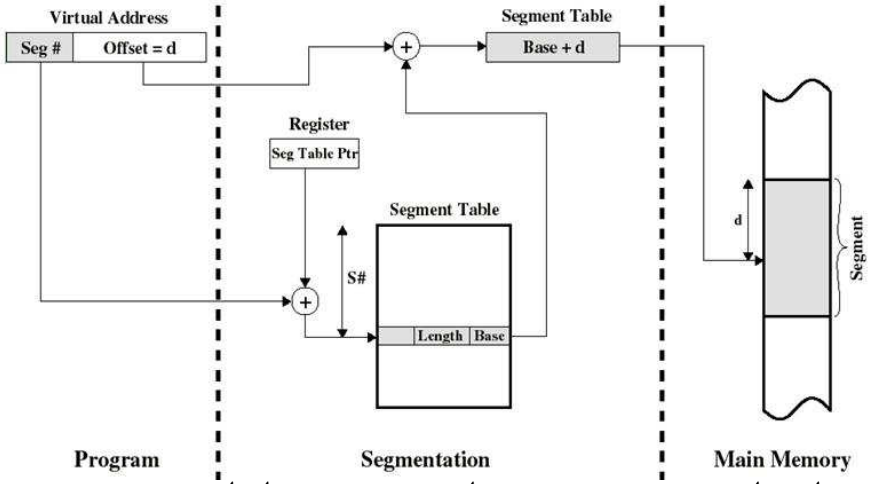
דוגמת חישוב: נניח שזמן גישה לזכרון אורך 200 ננו-שניות וזמן טיפול ממוצע בתקלת דף אורך 8 מילי-שניות. במקרה כזה:

$$EAT = (1-p) \times 200 + p\ (8\ \text{milliseconds}) = (1-p \times 200 + p \times 8{,}000{,}000 = 200 + p \times 7{,}999{,}800$$

אם נניח שרק ב-1 מתוך 1,000 ניסיונות גישה לדף, התרחשה תקלת-דף, ערך ה-EAT יהיה 8.2 מיקרו-שניות, פי 40 איטי לעומת זמן הגישה לזכרון. לכן, קל להבין שהיות ועלות הטיפול בשגיאת-דף היא יקרה מאד (פעולות קלט-פלט גוזלות זמן רב), ביחס לזמן גישה לזכרון, יש האטה של ביצועי המחשב באופן משמעותי.

[מצגת os8-2 vir]

Address translation in a segmentation system



נניח שגם בטבלת המקטעים, יש לנו סיבית שאומרת לנו אם המקטע נמצא בזכרון או לא (אם לא, נקרא לזה segment fault). אם צריך להביא מקטע, היות ומדובר בניהול זכרון דינאמי, נצטרך למצוא מקטע מספיק גדול בזכרון, כדי שנוכל למקם את המקטע הדרוש מהדיסק. לאחר מכן כמובן נעדכן את הכניסה המתאימה בטבלת המקטעים (segment-table).

בשילוב של דפים ומקטעים, תרגום של כתובת יצריך תחילה למצוא את המקטע המתאים, לפני שנמצא את הדף המתאים בתוכו. כמובן, בכל אחד משני שלבים אלו, ייתכן ויש לנו שגיאה, שהמקטע / הדף לא נמצא בזכרון ואז צריך כמובן להביאו מהדיסק..

Paging considerations

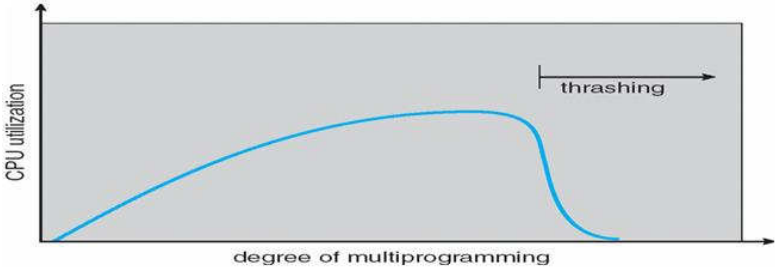
בתחום ניהול הזכרון (בעיקר הוירטואלי), יש מגוון נושאים. על חלק מהנושאים נרחיב היום.

עקרון המקומיות (Locality) – מאפשר לשיטה של הוירטואליזציה לעבוד. [מופיע בעמ' 6] יש מקומיות במרחב ויש גם מקומיות בזמן. ברגע שמביאים דף מסוים, הדף בעצם מגדיר מקומיות במרחב. הבאה של מקטע היא מקומיות טבעית יותר. יש גם טכניקות של 'הקדמת תרופה למכה', לנסות להביא דפים לפי תחזית מסוימת של שימוש בהם, pre-paging. למשל כאשר צריך להביא דף מסוים, נביא גם את הדף הקודם לו ואת הדף העוקב אחריו.

ניתן לשרטט גרף שבו ציר X ייצג את הזמן ואילו ציר Y את מספרי הדפים. כך, בכל שלב בריצת התוכנית ניתן לראות באילו דפים התוכנית התרכזה. גרף כזה מאפשר לעקוב אחרי המענים שהתוכנית ניגשת אליהם. ניתן לבדוק מהי סידרת הדפים שהתוכנית ניגשת אליהם במהלך ריצתה. מה שמעניין אותנו הוא לא כמה פעמים ניגשים לדף מסוים, אלא מהי מחרוזת-ההתייחסות, איך אנחנו עוברים בין הדפים, ללא חשיבות לכמה זמן נשארנו בכל דף ולאילו מענים בכל דף פנינו. ע"פ עקרון המקומיות, כאשר אנחנו מתרכזים בדף מסוים, אנחנו לא מתרכזים בדפים אחרים. בכל רגע נתון מספר הדפים שהתוכנית מתרכזת בהם הוא קטן יחסית. לכן, צריך שמספר הדפים שיהיו בזכרון בכל רגע נתון (working-set) יהיה קטן, בהתאם לצורך של התוכנית, גם אם התוכנית כולה תופסת דפים רבים. לפעמים מבלבלים את עקרון המקומיות עם חוק ה-80/20 (לפיו ב-80% מהזמן אנחנו נמצאים ב-20% מהדפים של התוכנית, מה שמגביל את כמות הדפים לה התוכנית נדרשת). לפי החוק הזה, רוב הדפים של התוכנית לא מטפלים ב- steady state, אלא הם כוללים רק תיחול / סיום של התוכנית או טיפול במקרי קצה – ואילו מיעוט הדפים כוללים את ה-main code. ייתכן שמיעוט הדפים העיקריים מכילים נתונים וייתכן שהם כוללים קוד.

דשדוש thrashing

הטכניקה של הזכרון הוירטואלי עובדת בזכות עקרון המקומיות. אם נסתכל על כלל התוכניות שרצות על מערכת ההפעלה, נתקלים בבעיה של דשדוש. גרף המקשר בין רמת המקביליות של מערכת הפעלה (ציר X) ובין הניצולת של המעבד (ציר Y):



כזכור, המזמנן לטווח ארוך אחראי על טעינת התוכניות למערכת, בעצם קובע את מספר התוכניות שרצות במקביל. הנחה סבירה – ככל שנטען יותר תוכניות למערכת, יהיו יותר תוכניות במצב ready שירצו לרוץ – וניצולת המעבד תגדל בהתאם.

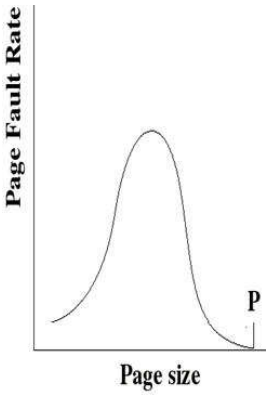
אך ההנחה נכונה כל עוד אנחנו טוענים מספר תוכניות שלא גורם לעומס על המערכת, מבחינת חוסר מקום בזכרון וחוסר זמן (כל תוכנית צריכה זמן מעבד כדי לרוץ..).

בנוסף, יש גם את המזמנן לטווח בינוני שאחראי לכבות שריפות, תפקידו לאפשר למצב 'משגוח' להשתלט. אם נוסיף גם את המנגנון של זכרון וירטואלי, נוצרת לנו בעיה: כאשר יש עומס על המערכת, יש גם עומס על המסגרות (frames) בזכרון. אם התוכניות הטעונות לזכרון דורשות יותר מסגרות ממה שיש לנו בפועל, לכל תהליך נוכל להקצות פחות מסגרות ממה שהוא זקוק לו, נצטרך לדפדף בתכיפות גבוהה, כדי שלכל תוכנית נספיק לטעון את ה-working set הדרושים לה כדי לרוץ. במצב כזה, המערכת רק תכניס דפים לזכרון ולא נשאר זמן למערכת להריץ את התוכניות עצמן, כדי שהן יסיימו את ריצתן ונוכל לטעון תוכניות אחרות. לסיכום:
מצב של דשדוש (מדשדשים בבוץ ☺) הוא מצב בו אנחנו מבזבזים יותר מדי זמן על שחלופים של דפים פנימה / החוצה.
מצב של דשדוש הוא 'כדור שלג', משתלטים על מסגרות של תהליכים אחרים, מה שרק מחריף את הבעיה כי אז גם את הדפים שלהם נצטרך לטעון..

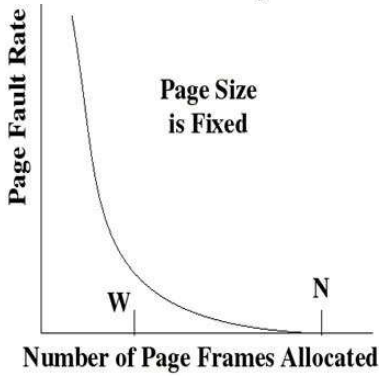
<u>כיצד נחליט מה יהיה גודל דף – page size issue</u>
יש מגוון שיקולים להגדלת הדף ככל האפשר ומנגד שיקולים לפיהם הדף צריך להיות קטן:

- ככל שהדף יהיה קטן, פוחתים הסיכויים שיהיה פיצול פנימי (לכן גודל הדף כגודל המסגרת).
- ככל שהדף יהיה גדול, כך מספר הכניסות בטבלת הדפים יהיה קטן יותר, מה שיגדיל את ה-hit ratio בטבלת ה-TLB. בהתאמה, כך המרחב הלוגי של התוכנית דורש פחות דפים ונדרשים פחות דפים לוגיים כדי למקם את התוכנית.
- שיקולים של קלט/פלט – אם כבר ניגשים לדיסק, עדיף להביא חלק גדול ככל האפשר. זאת בשל הבקשה, ההזזה של הזרוע של הדיסק, עצם הגישה (כל אלו פעולות יקרות) – לכן כדאי לקרוא כמה שיותר ולכן עדיף שהדף יהיה גדול.
- לעומת זאת, היינו רוצים גודל מכלא קטן, כדי לבזבז פחות מקום – ואז נרצה דף קטן יותר..
- אם הדף גדול מדי, כאשר אנחנו רוצים לגשת למען קטן מתוכו, נרצה בעצם דף קטן יותר..

המגמה היא להגדלת גודל הדף, בשל המגמה של הגדלת גודל הגוש בדיסק, זאת כי פעולות הקלט-פלט עולות זמן רב. לכן עדיף גושים גדולים, כדי שכל קובץ יתפוס פחות גושים – וכל גוש יכיל יותר נתונים.



סיכום לגבי גודל דף רצוי:
גרף של גודל הדף (ציר X) לעומת קצב תקלות-דף (ציר Y). כאשר גודל דף קטן, המקומיות יעילה, הדף מכיל בדיוק את הקוד שאנחנו צריכים, יש פחות שגיאות-דף. כאשר גודל דף גדול, הוא מכיל מלבד האזור שאנחנו צריכים, הרבה קוד שלא שייך לעקרון המקומיות הדרוש לנו כרגע, מבזבז לנו מקום בזכרון על חשבון דפים אחרים שאנחנו אולי נידרש להם ויש יותר שגיאות-דף. בגרף ניתן לראות שאם גודל הדף גדול-ממש, הדף יכיל את כ-ל התוכנית, מה שיגרום לכך שלא יהיו לנו כלל שגיאות-דף ☺. לעומת זאת, אם גודל הדף הוא בינוני, יהיו לנו הרבה תקלות דף (כי נסבול משני הכיוונים: גם יהיה לנו בזבוז שהדף מכיל קוד שלא דרוש לנו כרגע וגם הדף גדול מדי כך שגוזל לנו מקום על חשבון דפים אחרים הנחוצים לנו).



גרף של יחס בין מספר המסגרות המוקצות (ציר X) ובין מספר תקלות-הדף (ציר Y).
ככל שנקצה יותר מסגרות, כך מספר תקלות הדף יקטן. אם תוכנית דורשת N מסגרות ואכן הקצנו לה N מסגרות, לא יהיו לנו תקלות-דף כלל, כי כל התוכנית טעונה לזכרון. המספר האידיאלי הוא W, כך עברנו את הירידה החדה ויש לנו מספר קטן יחסית של שגיאות-דף. הקצאה של מספר מסגרות גדול מ-W עבור תוכנית יחידה, יגרום לכך שלתוכניות אחרות לא יהיו מספיק דפים טעונים לזכרון – וחבל. בנוסף, צריך לזכור שיש תוכניות קטנות (tinyUML ☺) ויש תוכניות כבדות (MS office). לכן, צריך לאפשר גושים בגדלים שונים – ודפים בגדלים שונים, כדי שנוכל לטעון לזכרון תוכניות בגדלים שונים. אם נאפשר הקצאה של דפים במספר גדלים שונים, יהיה לנו אמנם מסובך לטפל בעניין, אבל נייעל את הריצה של התוכניות.

<u>כמות הזכרון אליה ניתן לגשת מהטבלה הצדדית – TLB reach</u>
בטבלת ה-TLB יש מספר כניסות קטן. שאלה: עד כמה ממרחב הלוגי של התוכנית, הטבלה מכסה? תשובה: גודל הטבלה (מספר הכניסות) כפול גודל דף. כמובן, אם גודל הדף הוא קטן, הכיסוי הלוגי של הטבלה, ביחס לגודל הזכרון, הוא קטן.

$$\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$$

נשאף לכיסוי גבוה ככל האפשר, כדי להגדיל את יחס ה-hit ratio. איך נעשה זאת? ניתן להגדיל את מספר הכניסות ב-TLB (יקר, היא נמצאת ב-MMU). פתרון טוב יותר יהיה לאפשר גדלים שונים של דפים, כך גם עבור תוכניות גדולות וגם עבור תוכניות קטנות, יידרשו לנו מספר דומה של כניסות כדי לכסות אותן.

שיקול תכנותי (למדנו גם בהרצאה של תכנות מתקדם 1): נניח שיש לנו מטריצה בגודל k^2 ואנו רוצים לעבור על כל התאים.

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

עם הגישה לזכרון פיזי, אין לנו בעיה. אבל כאשר מדובר בזכרון וירטואלי, זה משנה אם דף לוגי מכיל שורה או עמודה – כי זה יקבע כמה תקלות-דף יהיו לנו כאשר נעבור על כל התאים במטריצה.

אם למשל כל דף ישמור שורה ואילו אנחנו עוברים לפי עמודות, יהיה לנו מספר גדול של שגיאות-דף (על כל תא במטריצה, כי כל פעם נביא דף של שורה אבל נסתכל רק על תא יחיד בו). לרוב, המהדר יודע לעשות אופטימיזציה ויכול לשנות את הקוד שלנו כך שנרוץ תחילה לפי שורות / עמודות, בהתאם למה שיעיל יותר.

- קישור שהמרצה פירסם: How Virtual Memory Works

הקשר בין קלט-פלט לניהול זכרון וירטואלי – I\O interlock

בקר ה-DMA [עמ' 12] לא מטריד את המעבד, הוא אחראי על פעולות קלט-פלט. נניח שאנחנו מעוניינים להשתלט על מסגרת-קורבן, בזמן שהמסגרת עסוקה כרגע בקלט-פלט, פעולה הדורשת זמן רב. זה לא יהיה טוב! נצטרך לבצע נעילה, כדי לסמן שהמסגרת הזו לא יכולה להיות משוחלפת החוצה. כמובן, אם פעולת הקלט-פלט מתפרשת על יותר ממסגרת יחידה, נצטרך לבצע נעילה לכל המסגרות העסוקות הממתינות לפעולת בקר ה-DMA.

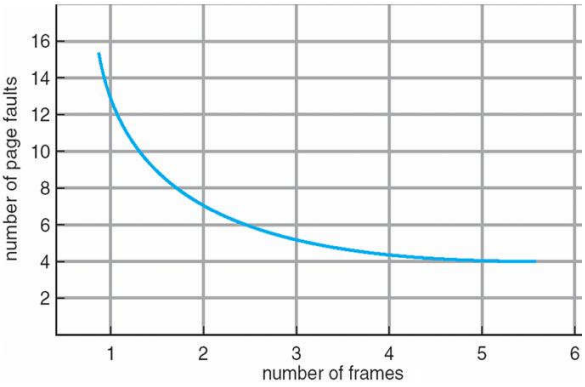
Copy-on-write

כאשר יוצרים תהליכי-בן, הקוד משותף כדי למנוע העתקות כפולות של משתנים. אם ההורה / בן משנים משהו בדף, צריך לשכפל אותו. באופן כזה אנחנו מבצעים את השכפול רק בזמן הדרוש לצורך השינוי.

[מצגת vir 3-os8]

שיטות לבחירת דף-קורבן לשחלוף החוצה – page replacement

נושא מאד מורכב: האם לבחור קורבן מבין המסגרות של התוכנית הנוכחית, או לבחור קורבן של תהליך אחר – ואז התנהגות גרועה של תוכנית אחת עלולה לפגוע בתפקוד של תוכניות אחרות. איך נעקוב אחרי ה-working set וההתאמה בין מספר הדפים שתוכנית צריכה, לבין מספר המסגרות שמוקצה לה. המטרה שלנו היא כמובן קצב תקלות-דפים נמוך. ככל שיש יותר תקלות-דף, הדפים יותר 'מלוכלכים', המערכת מתקשה למצוא מסגרות פנויות, מתקשים למצוא קורבנות – הקץ מתקרב, מגיעים למצב של דשדוש בבוץ ☺. המזמנן לטווח בינוני לא אמור לפעול רק לכיבוי שריפות, ממש לפני הדשדוש, אלא צריך לפעול עוד קודם לכן, במטרה למנוע זאת ולפנות את השטח לצורך הקלה על העומס.

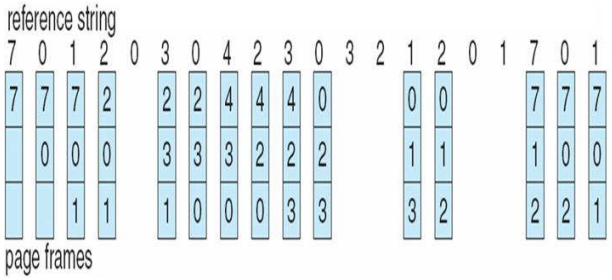


מזעור מספר תקלות הדף נעשה ע"י ניהול חכם של המסגרות: אם אין מסגרת פנויה, נצטרך לבחור בצורה מושכלת דף-קורבן שישוחלף החוצה. למשל, נעדיף לשחלף דף נקי במקום דף מלוכלך. לעומת זאת, אפשר לטעון שדף נקי אולי נצטרך בזמן הקרוב ושוב נצטרך לטעון אותו ואילו דף מלוכלך גם ככה יש לכתוב לדיסק עם השינויים, כך שכביכול אולי עדיף לשחלף-החוצה דווקא אותו.

בגרף מימין: הגדלת מספר המסגרות המוקצות לתוכנית, יפחית את מספר תקלות-הדף של התוכנית הספציפית. מצד שני, מנקודת המבט של המערכת, ככל שתוכנית מסוימת מקבלת הקצאה של יותר מסגרות, כך תוכניות אחרות מקבלות פחות. זהו פתרון לא גלובלי.

אלגוריתמים לבחירת דף-קורבן – page replacement algorithms:

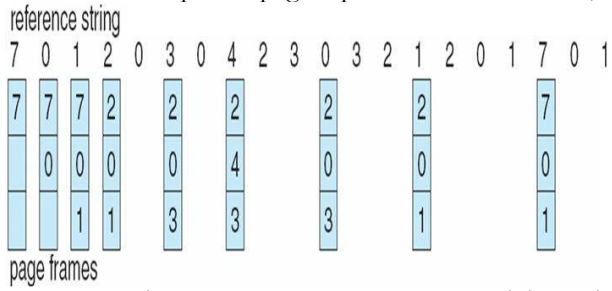
(1) FIFO – ראשון נכנס, ראשון יוצא. נתייחס לדפים הנטענים לזכרון באופן-מעגלי: דף שנטען ראשון (לפני הרבה זמן) כנראה שכבר לא זקוקים לו עכשיו, לעומת אלו שנכנסו לאחרונה ועדיין דרושים. לכן נשחלף-החוצה את הדף הישן ביותר שנמצא בזכרון. האלגוריתם קל למימוש, צריך רק מצביע שיוצר לנו את המעגליות. דוגמת ריצה בשימוש ב-FIFO:



מחרוזת-התייחסות (reference string) אומרת לנו את סדרת הדפים אליהם ניגשנו (כמובן שבכל דף אנו ניגשים לפחות למען אחד, אחרת לא היינו טוענים אותו..). נניח שמוקצות לנו 3 מסגרות לשימוש התוכנית (= מספר מסגרות קבוע) ורק אל דפים אלו התוכנית מתייחסת בכל רגע נתון. עבור מחרוזת-ההתייחסות שבדוגמה, נספור כמה תקלות-דף יתרחשו. ע"פ FIFO, נשחלף-החוצה את הדף הכי ותיק (אין כבוד לפז"מניקים ☺).

FIFO לא מתעניין אם הדף נקי / 'מלוכלך'. בדוגמה, יש 20 דפים במחרוזת-ההתייחסות ו-15 תקלות דף (בהן הדף המבוקש לא היה קיים בזכרון והיינו צריכים למצוא קורבן ולשחלף אותו). היות ומספר המסגרות המוקצות הוא 3, יש לנו סה"כ (20-15=) 12 החלפות דף (3 תקלות הדף הראשונות בכל מקרה יתבצעו, כי בתחילה 3 המסגרות ריקות). היות והמרחב הלוגי של התוכנית (8 דפים) גדול מהמרחב הפיזי של התוכנית (3 מסגרות מוקצות לה), ברור לנו שיהיו החלפות-דף. אם המספרים היו זהים, היינו לנו רק תקלות-דף בהתחלה, היינו טוענים את כל הדפים ו.. זהו, לא היינו מגיעים למצב שצריך להחליף-דפים. באלגוריתם FIFO יש אנומליה [שקופית מס' 7 במצגת]: למרות שיש לנו יותר מסגרות, יש יותר שגיאות-דפים. [בדיחה מעושה: זו דוגמה 'מעושה', עשו אותה בכוונה, היא לא מתרחשת במציאות ☺] לעומת זאת, לאלגוריתמים שפועלים בשיטה של מחסנית (stack) אין את האנומליה הזו. לסיכום: קל למימוש, אך לא יעיל.

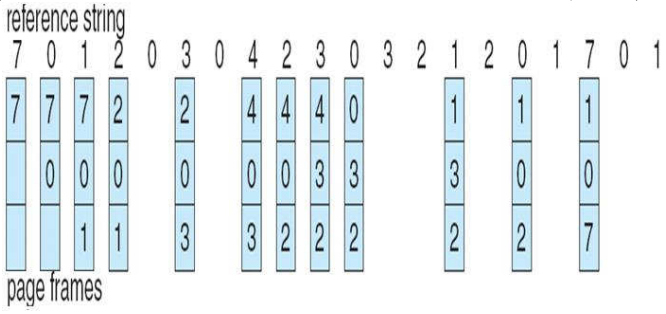
(2) דף אופטימלי (optimal page replacement) – כביכול מוציא את הדף שהסיכוי שישתמשו בו הבא בתור הוא הנמוך ביותר.



דוגמת ריצה: בעת שחלוף דפים, נבדוק מי הדפים שבזכרון ומי הדפים הבאים שנצטרך (לפי מחרוזת ההתייחסות). בדוגמה זו יש לנו רק 6 החלפות דף, מחצית ביחס ל-FIFO. לכאורה דף שנצטרך בעתיד הקרוב, לא נשחלף אותו. אך מחרוזת ההתייחסות לא נתונה במהלך ריצת התוכנית. לכן לא נוכל לדעת מה הדפים אליהם התוכנית תתייחס בעתיד. האלגוריתם הזה הוא תיאורטי לחלוטין, ישמש לנו רק כקנה-מידה, כדי לדעת עד כמה האלגוריתם אותו נציע רחוק מהאופטימום. ככה נדע למה לשאוף ☺.

לא נוכל להריץ את התוכנית מספר פעמים ולנחש מה תהיה מחרוזת ההתייחסות שלה באופן ממוצע, כי ייתכנו מקרי קצה וריצת התוכנית תלויה בקלט שלה.

(3) LRU (least recently used) – נשחלף החוצה את הדף שעשינו בו פחות שימוש לאחרונה (כביכול קירוב לבדיקה המשלימה לקודמת, נלמד על העתיד ע"י הסתכלות על העבר). מספר הבדיקות-אחורה, כמספר המסגרות הפנויות (פחות 1).

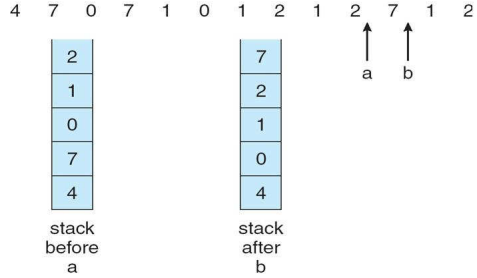


בדוגמת הריצה בצד ימין, מספר החלפות-הדף 9, טוב יותר מ-FIFO. למשל כשנכניס את 4, נראה שב-0 ו-3 נעשה שימוש לאחרונה, לכן נסיק שאת 2 עלינו לשחלף-החוצה. ניתן להראות ש-LRU פועל בשיטה של מחסנית ולכן אין לו את האנומליה של FIFO.

מימוש המדיניות של LRU (הוא פשטני מדי, צריך גם לממש..):
נשאף להוזיל את העלויות של האלגוריתם. נשאף לאלגוריתם שיהיה פשוט וזול כמו FIFO, אך יעיל כמו LRU. רעיונות:

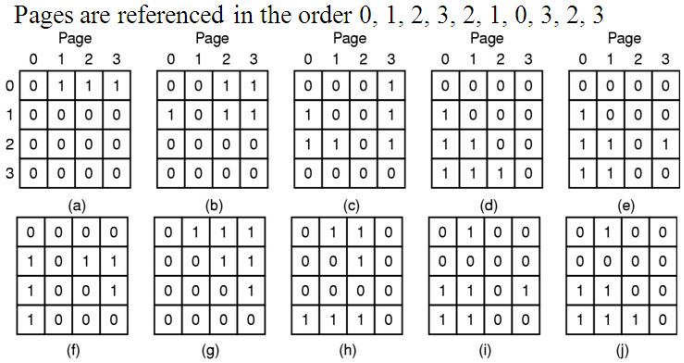
(א) שמירת זמן גישה ע"י counter – בכל פעם שניגשים לדף, נשמור את זמן הגישה בו ניגשנו לדף, כדי לדעת מי הדף שהשתמשנו בו הכי פחות לאחרונה. הכניסה בטבלה עם תג הזמן הנמוך ביותר, היא הישנה ביותר, לא ניגשו אליה לאחרונה – נשחלף אותה החוצה. הטריק הזה הוא עבור דפים שלא נמצאים ב-TLB, כי דף שנמצא ב-TLB – ברור שהשתמשנו בו לאחרונה. העלות של הפתרון הזה יקרה: יש להוסיף שדה נוסף בכל כניסה בטבלה, מנגנון חומרה מורכב, דרושה לוגיקה מורכבת, מעט מאד מחשבים יכולים לשלם את המחיר הזה של true LRU.

(ב) מחסנית – בכל פעם שמשתמשים בדף, מחלצים אותו ממנה ושמים אותו בראש.



דף שנמצא בתחתית המחסנית הוא זה שלא היה בשימוש לאחרונה, לכן אותו נשחלף החוצה. מימוש זה יקר מאד, כי כדי לאתר דף במחסנית ולהעלותו לראש המחסנית, צריך לעבור על כל המחסנית.

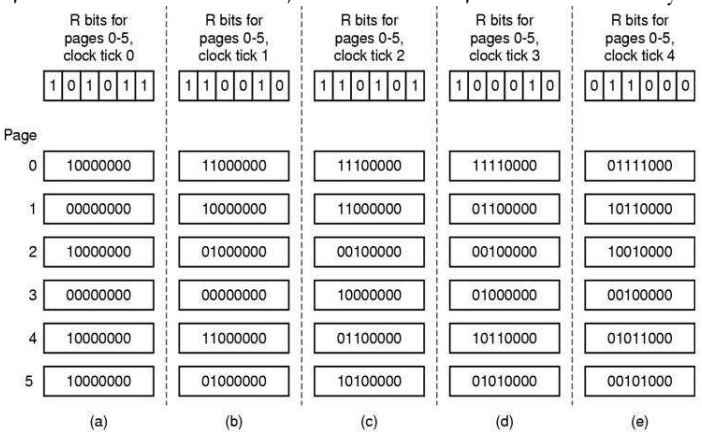
(ג) מטריצת חומרה (hardware matrix) – גודל המטריצה הוא מספר המסגרות המוקצות (בריבוע). המטריצה מאופסת בעת האתחול. כאשר אנו ניגשים לדף, הופכים את השורה המתאימה לאחדות ולאחר מכן את העמודה המתאימה לאפסים.



כאשר נחפש דף-קורבן, נעבור על שורות המטריצה: שורה שבה יש הכי הרבה אפסים, מייצגת דף-לוגי אליו פנינו הכי פחות לאחרונה ולכן אותו נשחלף-החוצה. כמובן, ייתכן מצב שבו ליותר מדף-לוגי יחיד יש אותו מספר של אפסים – נבחר אקראית את הדף שנשחלף החוצה. לצערנו, גם הטיפול במטריצה הוא יקר.

(ד) קירוב ל-LRU, במחיר זול יותר – נשאף להגיע ליחס של 20/80 (ב-80% מהזמן האלגוריתם טוב כמעט כמו LRU וב-20% מהזמן הוא 'לא משהו'). reference bit – לכל כניסה בטבלה נוסיף סיבית שמאותחלת לאפס. בכל התייחסות לדף הסיבית תיהפך ל-1. נבחר בקורבנות שיש להם 0 בסיבית הזו. החסרונות: בין מספר כניסות להן הסיבית היא אפס, לא נדע במי לבחור. בנוסף, לאחר שניגשנו ל-3 דפים, לא נדע למי התייחסנו מוקדם יותר..

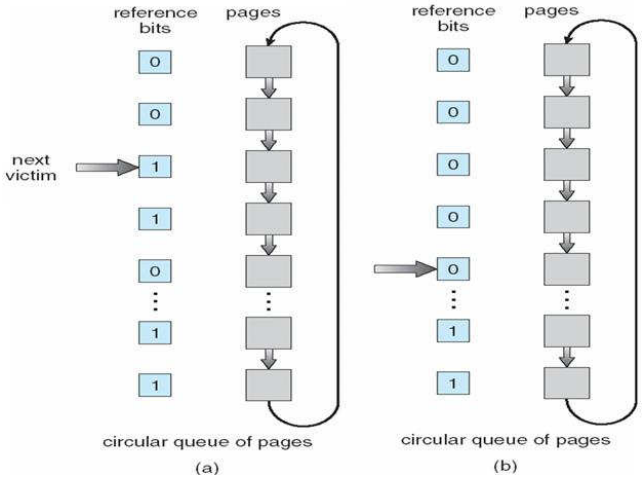
(ה) reference byte – במקום סיבית יחידה, נשמור 8 סיביות כדי לציין טוב יותר מתי התייחסנו לדף.



בעצם אנחנו יוצרים לנו logical clock tick, שאומר כל כמה זמן לבדוק את סיבית ההתייחסות. באתחול נאתחל ל-0. בכל פעם שנעשה בדיקה (ע"פ המערך למעלה, שמציין אם הדף היה בשימוש), נעשה left-shift (נוסיף סיבית מצד שמאל, נדחוף ימינה, הסיבית הימנית תידחק החוצה). כאשר סיבית ההתייחסות של דף היא 0, נדחוף לו 0 משמאל, הסיבית הימנית תימחק. ב-a, לדף 0 (למשל) היה 1, לכן דחפנו לו 1 משמאל (הסיבית הימנית נפלה). הקורבן יהיה בעל הערך המספרי הנמוך ביותר. בעמודה e זה יהיה דף מס' 3, כי יש לו הכי פחות אחדות במקומות ה'גבוהים'.

ככל שיהיו לנו יותר סיביות, נשמור היסטוריה של זמן ארוך יותר. הלוגיקה של ה-shift מיושמת ע"י פקודות מכונה בחומרה, הדבר נעשה באופן מהיר. אך הבדיקה למי יש את המספר הכי קטן גוזלת זמן.

(ו) clock (second chance) policy – בכל דגימה, נאפס קורבנות שלא בחרנו בהם, כך שיהיו קורבנות פוטנציאלים בעתיד ☺.



מתייחסים ל-reference bit בצורה מעגלית, בודקים מי הסיבית-קורבן האחרונה אליה התייחסנו. לחלק מהדפים יש 0, לחלק יש 1, לפי המחיקה האחרונה. כאשר מחפשים קורבן, נחפש החל מהמצביע next victim. כל עוד יש אחדות, נאפס אותם ולא נבחר אותם כקורבן, כי התייחסו אליהם (עובדה שיש להם 1). כאשר נגיע ל-0 הראשון, נבחר אותו כקורבן, כי לא התייחסו אליו לאחרונה (גם אם בהמשך יש אפסים נוספים). כך נחפש קורבנות בצורה מעגלית, כולם מתאפסים בדרך לקורבן הבא. רק אלו שנשארים 0 סיבוב שלם, ייבחרו כקורבן. המימוש הוא קל, מתייחסים ל-reference bit בצורה מעגלית. לאחר שנכנס דף חדש, זה היה כמובן לצורך התייחסות, נעדכן את הביט שלו ל-1.

(ז) ניעזר במונה (counter), כל פעם שניגש לדף נגדיל את המונה. ניתן לפעול בשתי לוגיקות: MFU (most frequently used) – הדף שהיה הכי בשימוש לאחרונה, אולי כבר לא נצטרך אותו, לכן נוציא. לעומת זאת LFU (least frequently used) – הדף שהיה הכי פחות בשימוש לאחרונה, כנראה שגם בהמשך לא נצטרך אותו, נוציא. מצד שני, דף חדש שרק הובא, המונה שלו קטן וכבר אנחנו עלולים להוציא דווקא דף חדש..

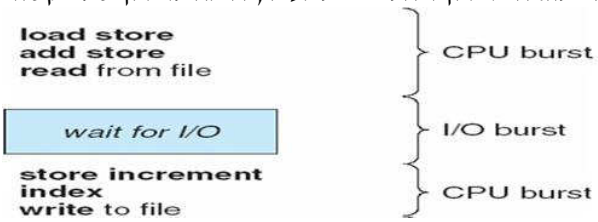
(ח) page buffering – לא נבחר בקורבן ברגע האחרון, אלא נקדים תרופה למכה. בזמנים 'מתים' של המעבד, נבחר קורבנות, נעביר אותם למאגר קורבנות פוטנציאלי (מחנה ריכוז ☺). נשמור את הקורבנות בסדר של FIFO, תמיד נבחר את הראשון (הכי ותיק שהכנסנו למאגר הזה). אבל, אם הכנסנו דף למאגר של הקורבנות ופתאום אנחנו כן משתמשים בו, נוציא אותו מיד מהמאגר. לכן, מי שנכנס למאגר הקורבנות הפוטנציאלים ולא עשינו בו שימוש עד הרגע שאנחנו אכן צריכים לבחור קורבן – סימן שהוא אכן 'ראוי' להיות קורבן (☺) ונשחלף-אותו החוצה. כמובן, נעדיף לצרף למאגר הזה רק דפים נקיים, כדי שלא נתעכב על כתיבת דף 'מלוכלך' למאגר.

הרצאה מס' 14 - 18.1.10

[מצגת os9-1 cpu]

<u>זמנון מעבד – CPU scheduling</u>

אנלוגיה: זמנון תהליכים למעבד, דומה לזמנון של בקשות קלט-פלט לדיסק.



ניתן להניח שכאשר יש תהליך יחיד במערכת, בחלק מהזמן הוא 'שולט' במעבד (CPU-burst) ומנסה לקדם את עצמו ואילו בחלק מהזמן הוא נמצא בהמתנה לקלט-פלט (I\O burst). ייתכן כמובן שתהליך-האבא שלו השהה אותו לפרק זמן מסוים, או שהוא ממתין לסמפור שיאפשר לו גישה לזכרון משותף..

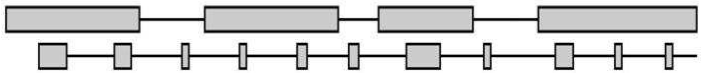
אנו נניח שהתהליך נמצא כעת במצב 'ריצה'. יש לו CPU burst-cycle (מחזור ההדגמים, יוסבר בהמשך), המחזור שהמעבד מאפשר לו לעבוד ולאחר מכן להמתין (כל אחד מפרקי זמן אלו יכול להיות באורך שונה).

עד כה הכרנו את המושגים: עבודה (job) שנחלקת לתהליכים (processes). אך למעשה כל תהליך נחלק גם בעצמו למספר הדגמים (instances). במחזור החיים של התהליך, לאחר מצב new, הוא מוכנס לתור המוכנים-לריצה. לאחר מכן יש לו זמן ריצה (r, מלשון run), זמן המתנה (w, מלשון wait) וחוזר חלילה עד לסיום פעולתו (מה שסביר להניח יקרה בזמן ריצה, אך יתכן גם בזמן המתנה אם ההורה שלו מחליט להרוג אותו ☺). כל צמד של מצב המתנה ומצב ריצה רצופים (w,r), נקראים הדגם.

המושג הדגם מאפשר לנו להבין טוב יותר את תפקידי הזמננים (לזמן ארוך, בינוני וקצר) עליהם למדנו. ניתן לחשוב על הזמננים השונים ע"פ המדרג שהזכרנו עכשיו: הזמנן לטווח ארוך אחראי על זמנון עבודות (מתוך ה-job pool). הזמנן לטווח קצר עובד ברמת ההדגמים, כי למעשה מי שממתין בתור של המוכנים-לריצה, הוא למשל הדגם מס' 42 של תוכנית מס' 5, משמע תהליך שהתחיל את ריצתו והוא מעוניין להמשיכה. כך שמי שנמצא בתור המוכנים-לריצה הם לא בדיוק התהליכים, אלא ההדגמים, *המייצגים* את התהליכים. הזמנן לטווח בינוני הוא זה שפועל ברמה של תהליכים, פועל במצב חירום. הוא לא מתעסק עם הדגמים, לא מעניין אותו באיזה הדגם נמצא כל תהליך, הוא רק צריך לדעת איזה מהתהליכים יש לשחלף החוצה לדיסק, איזה מהם יש להשהות.
לסיכום: יש לנו 3 סוגי ישויות (עבודה, תהליך, הדגם) ויש לנו 3 זמננים שיודעים להעביר את השליטה ביניהם. לכן, נוכל לשפר [ולחובבי 'עברית שפה יפה': לעשות עידון, refine] את ההגדרה: לא רק שעל כל מעבד יכול לרוץ בכל רגע נתון תהליך יחיד, ליתר דיוק רק הדגם יחיד של תהליך מסוים נמצא בזמן ריצה בכל רגע נתון.

ישנן עבודות שדורשות הרבה זמן ריצה (עיבוד), 'טוחנות מספרים' (number crunching), נקראות long-jobs. הן כמובן נקראות ארוכות ביחס לעבודות הידודיות (interactive) קצרות (מפתיע?) שמבצעות פעולות מול המשתמש, למשל קליטת תווים מהמקלדת, שנקראות short-jobs. לעבודות קצרות יש צורך במעט זמן מעבד, כי רוב הזמן הן ממתינות לקלט-פלט מהמקלדת / למסך. לכן, לעבודות קצרות יש פרצי עיבוד יחסית קצרים. לעומת זאת, לעבודות ארוכות יש פרצי עיבוד ארוכים יותר, כי הן צריכות לבצע חישובים מורכבים. ניתן בהתאם לדבר על תהליכים קצרים / ארוכים, הפועלים מטעם עבודות אלו.

דוגמה של עבודה בעלת פרץ עיבוד ארוך (למעלה) לעומת עבודה בעלת פרץ עיבוד קצר (למטה):
ריבוע אפור מציין פרץ עיבוד (r, CPU-burst) ואילו קו שחור מסמן זמן המתנה (w, IO burst).



<u>שיקולים בקביעת מדיניות הזמנון</u> [מופיעים בשקופית מס' 10]:

- ברמת המערכת, אפשר להתייחס ל'הוגנות' (fairness), נרצה לתת לכל התהליכים משך זמן מעבד זהה. צריך גם לדאוג לאיזון של פעולת המעבד ופעולות הקלט-פלט, מה שאומר שיש לאזן את כלל הפעולות במחשב.
  נפריד בין מערכות-אצווה ומערכות אינטראקטיביות:
- במערכות-אצווה נרצה תפוקה (throughput) גבוהה, כמה שיותר עבודות יהיו מוכנות לריצה בכל רגע נתון. 'זמן-סבב' (נקרא turn-around time) הוא הזמן הלוקח מרגע הגשת העבודה ועד הזמן שהיא סיימה את ריצתה. רמת המקביליות במערכת, מספר העבודות הטעונות, קובעת את ניצולת המעבד. נרצה כמובן שניצולת המעבד תהיה יחסית גבוהה. כמובן, אם המעבד עסוק בדשדוש, הוא לכאורה אמנם נראה עסוק, אך הוא לא מקדם התקדמות תהליכים.
- במערכות אינטראקטיביות (foreground) לא מתעניינים בזמן-סבב, אלא מדברים על זמן תגובה (response time, תוך כמה זמן הפלט יופיע בעקבות הקלט). נשאף כמובן לזמן-תגובה קצר ככל האפשר, העיקר שהמשתמש ירגיש שהוא מקבל שירות טוב ☺.
- במערכות real-time, המטרה העיקרית (היחידה) היא לעמוד בספי-זמן (dead-lines). יש כמובן הבדל בין hard real-time כך שחייבים לעמוד בזמנים שהוגדרו, לעומת soft real-time, למשל כאשר טעינת סרט מתעכבת. בכל מקרה, אין פשרות לגבי יכולת ההזרמה.

אנו נתרכז בעיקר במערכות אצווה (batch systems) ובזמנון במעבד יחיד. כאמור, נרצה תפוקה גבוהה, ומנגד זמן-סבב וזמן-תגובה קצרים ככל האפשר. הזכרנו את זמן ההמתנה (w): ככל שזמן ההמתנה יתקצר, זמן הסבב (turn-around time ,TT) יתקצר גם הוא, שכן הוא סכום של כלל ההדגמים של כל התהליכים (של כל העבודות..) הטעונות על המעבד. בפועל, מעניין אותנו זמן הסבב הממוצע (average turn-around time ,ATT), שהוא הסכום של זמני הסבב של כלל התהליכים לחלק למספר התהליכים.
ניתן לנתח את כל השיקולים גם ע"פ חלוקה מנקודת המשתמש / מנקודת המערכת. מבחינת המיטוב, נרצה כמובן תפוקה מקסימלית מהמעבד ואילו נרצה להקטין ככל האפשר את זמן הסבב וזמן ההמתנה.
דגש חשוב: זמן המתנה (w) נחלק לשניים: המתנה לקלט-פלט וזמן המתנה בתור המוכנים-לריצה. את זמן הקלט-פלט לא ניתן כמובן לצמצם, אך כן ניתן לצמצם את זמן ההמתנה של התהליך בתור המוכנים-לריצה.

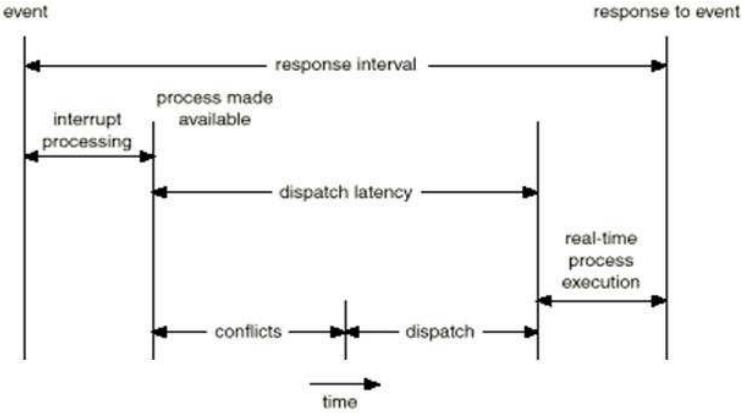
לרוב, כאשר אומרים מזמנן-מעבד, מתכוונים לזמנן לטווח-קצר, האחראי על המעברים בין המצבים השונים של התהליכים:

1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.
4. Terminates.

ישנם מספר סוגי אלגוריתמים לזמנון. בתחילה, הגישה הייתה non-preemptive (לא לוקחים בכוח), כל תהליך שקיבל את השליטה על המעבד, נתנו לו לרוץ כמה שהוא רצה. אך היה חשש שהתהליך ייכנס ללולאה אין סופית / לא יחזיר את השליטה לידי המערכת.
הבינו שצריך לפתח אלגוריתמים שידעו לקחת בכוח את השליטה על המעבד מידי התהליך, לטובת המערכת. לקצוב את זמן הריצה של כל תהליך, כך שלאחר זמן מסוים מחזירים את השליטה למערכת (לזמנן לטווח-קצר) שתחליט מחדש לידי מי להעביר את השליטה על המעבד. לכן, פיתחו אלגוריתמי preemptive שידעו לפעול בעזרת קוצב-זמן (timer). לאחר שהסתיים הזמן שניתן לתהליך מסוים שרץ כרגע על המעבד, הייתה פסיקת-זמן (timer interrupt) שהייתה מחזירה את השליטה לידי הזמנן לטווח-קצר.

- selection function, פונקציית הבחירה – מחליטה מי מבין התהליכים בתור המוכנים-לריצה יהיה הבא שירוץ. כרגע נניח שיש מנגנון בזמנן לטווח-קצר שמסדר את תור העדיפויות הזה, לכן נניח שהראשון בתור הוא בעל עדיפות הריצה הגבוהה.
- decision mode – הזמן שבו מבצעים את הבחירה הזו, איזה מבין התהליכים יהיה הבא שירוץ. כאמור, כאשר עובדים במצב של non-preemptive, תהליך רץ עד שהוא מחזיר שליטה למערכת (למשל לצורך קלט-פלט). לעומת זאת, במצב של preemptive, המערכת מקציבה זמן לכל תהליך.

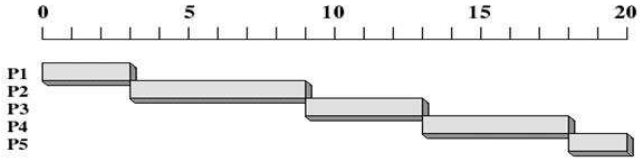
dispatcher, המשלח – [הרצאה מס' 6, עמוד 20] לוקח את התהליך / ההדגם הבא שצריך לרוץ, טוען אותו למערכת ו'משלח' אותו (עם נעלי ספורט ☺) לריצה.
dispatch latency הוא הזמן שלוקח למשלח לבצע את פעולתו, נשאף לזמן קצר ככל האפשר. באיור מימין סיכום הזמנים שהזכרנו עד כה. למעלה: מאורע מסוים (למשל פסיקה), לאחר מכן יש זמן שדרוש לצורך הטיפול במאורע, צורך בזמן להחליט איזה מההקשר הבא ייטען וכמובן שמירה של ההקשר הנוכחי.



ההבדל בין מודל התהליכים / תהליכונים:
משימה (task) מורכבת מתהליכונים, עבודה (job) מורכבת מתהליכים, אך הם שווי ערך מבחינתנו.

<u>האלגוריתמים לזמנון התהליכים</u>:
ניתן לחשב את סך זמני ההמתנה וסך זמני הריצה כדי לחשב את ממוצע זמן הסבב. חשוב להבין: זמן הריצה הוא קבוע, ללא קשר לאלגוריתם (ויעילות ה)זמנון (שהרי לכל תהליך דרוש זמן מסוים לצורך ריצתו, ללא קשר אם הוא מבצע זמן זה ברצף או בחלקים). ההבדל יהיה בזמני ההמתנה – אלגוריתם שיצליח לצמצם זמנים אלו, יצליח לשפר את זמן הסבב של כלל התהליכים. לכן, בהקשר זה נדבר על WT (waiting time) או על AWT (average waiting time).

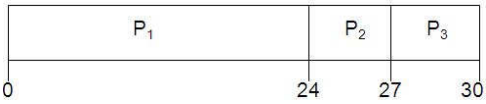
- FCFS (first come first served) – התהליך שממתין הכי הרבה זמן בתור המוכנים-לריצה הוא שירוץ (FIFO). נניח שאין פסיקות, שמדובר במצב של non-preemptive (אין לקיחה בכוח של השליטה) ונניח שהתוכנית רצה עד לסיום ברצף.



Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

בדוגמה: נניח שנטענו למערכת 5 תהליכים. מצוין לנו זמן ההגעה (זמן הטעינה). בהתאם ללוגיקה, התהליכים ירוצו ע"פ סדר ההגעה שלהם. התהליך היחיד שיכול לרוץ בזמן 0, הוא היחיד שנמצא בטעינה, תהליך מס' 1. מדובר באלגוריתם פשוט, לכאורה הוגן, כל אחד רץ בתורו ללא הפרעות, ומקבל את כל זמן המעבד הדרוש לו. דוגמה נוספת:

תרשים גאנט gantt, פעילות לפי זמנים



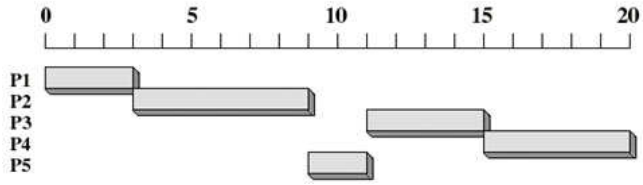
Waiting time for $P_1 = 0; P_2 = 24; P_3 = 27$

Average waiting time: $(0 + 24 + 27)/3 = 17$

Process	Burst Time
P_1	24
P_2	3
P_3	3

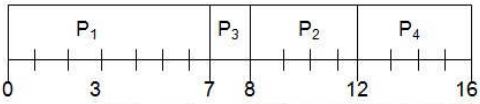
חסרונות: ממוצע זמן הריצה (24+3+3 / 3) הוא 10 ואילו זמן ההמתנה הממוצע הוא 17. לכן, זמן הסבב הממוצע הוא 27. אם התהליכים ירוצו בסדר שונה (הפוך), לא ע"פ סדר ההגעה, זמני ההמתנה יתקצרו משמעותית: זמן ההמתנה הממוצע יהיה רק 3 ולכן ממוצע זמן הסבב יהיה 13. משמע, זמן הסבב הממוצע מושפע מסדר טעינת התהליכים. בעיה נוספת: אפקט השיירה convoy effect – עבודות קצרות צריכות לחכות לעבודות ארוכות. [כמו שאנחנו נוסעים ברכב ספורט אחרי משאית איטית שלא מאפשרת לנו לעקוף אותה ☹] תהליכים ארוכים גוזלים זמן מעבד רב ומונעים מהתהליכים הקצרים לסיים את ריצתם ולחסוך בזמן ההמתנה הכולל.

- SJF (shortest job first) – סדר הריצה יהיה ע"פ זמן הריצה הדרוש. לאלגוריתם זה ישנם מספר שמות נוספים, למשל SPN (shortest process next) ו-STF (shortest time first). השם המודרני יותר הוא זמנון הדגמים (SIF), כי אנחנו מעוניינים להריץ את ההדגם הקצר יותר בכל פעם. מניחים שמדובר במצב של non-preemptive, כך שאין לקיחה בכוח של המערכת. יש טענה שאלגוריתם זה הוא האופטימלי, אך היא שגויה (עובדה שיש לנו אלגוריתמים מתקדמים יותר ☺).



Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

דוגמה בה הרצנו את התהליכים לפי SJF, מה שחוסך זמני המתנה. כמובן, אם שני תהליכים דורשים את אותו זמן הריצה, נריץ אותם בהתאם לסדר ההגעה (הגיע מוקדם – ירוץ מוקדם). דוגמה טובה לחישוב זמני המתנה, בהתאם לזמן ההגעה:



Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

לגבי הטענה לאופטימליות, נניח שיש לנו את האלגוריתם kuku שמריץ את התהליך long (לציין שהזמן הדרוש לו הוא ארוך) לפני התהליך short (בהתאמה, הזמן הדרוש לו הוא קצר). לעומת זאת, SJF היה מריץ בסדר הפוך, תחילה מריץ את short ורק לאחר מכן את long. מלבד השינוי הזה, סדר ההרצות של שאר התהליכים זהה. ל-SJF זמן המתנה קטן יותר ביחס ל-kuku. לכאורה SJF אופטימלי. [לצורך הדוגמה, נניח כמובן שאנחנו יודעים מה זמני הריצה הדרושים לכל תהליך, מראש..] SJF אכן אופטימלי ביחס לקבוצה סטטית, כאשר זמני ההגעה ידועים מראש. לעומת זאת, כאשר זמני ההגעה לא ידועים מראש, הוא נכשל. מדוע? נניח שהגיעה עבודה ארוכה והיא היחידה, לכן היא תרוץ. מעט לאחר מכן הגיעו מספר עבודות קצרות, הן יאלצו לחכות זמן רב עד לסיום ריצת העבודה הארוכה. אך אם היינו קוטעים את זמן הריצה של העבודה הארוכה, מריצים את הקצרות ואז ממשיכים להריץ את העבודה הארוכה, היינו משפרים את זמני ההמתנה.

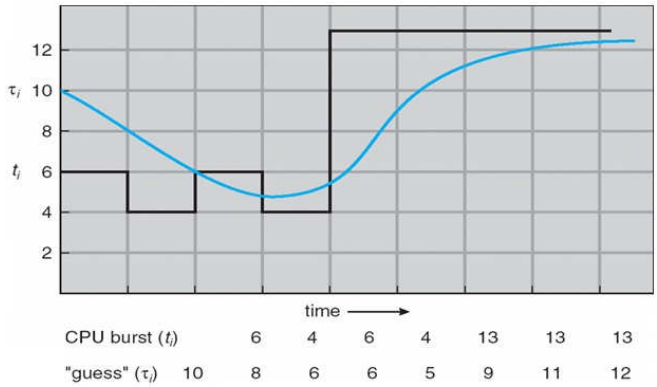
בעיה: כיצד בכלל נוכל להעריך מה פרץ העיבוד הבא הדרוש לכל הדגם? מדובר ב'הערכה טובה' (guesstimate), שילוב של ניחוש והערכה-צפויה. לאחר שהדגם מסוים סיים לרוץ, אנו יודעים כמה זמן הוא רץ בפועל. נסמן ב-t_n את זמן הריצה של ההדגם ה-n-י. את ההערכה של זמן הריצה של ההדגם הבא נסמן ע"י האות 'טאו' τ_{n+1}. החישוב של האלגוריתם המחולל (generative function) [בעצם מזכיר רקורסיה..] נראה כך:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n \quad \text{כאשר} \quad 0 \le \alpha \le 1$$

המרכיב השמאלי נקרא המרכיב האקטואלי (משך זמן פרץ העיבוד האחרון). המרכיב הימני נקרא היסטורי (מתייחס לערכים בעבר, הזמן שלקח לכל אחד מההדגמים הקודמים לרוץ).

אם נבחר את הערך של אלפא להיות 1, אזי נקבל ש'טאו' של ההדגם הבא יהיה בעצם t_n, זמן הריצה של ההדגם הבא יהיה בעצם כמו של ההדגם הקודם. אך אם הערך של אלפא הוא 0, המרכיב הזה מתבטל, ההערכה זמן הריצה הדרוש להדגם הבא, תלויה בעצם בהערכה של ההדגם הקודם, שתלויה בקודם.. עד הזמן שהערכנו שירוץ ההדגם הראשון..

בתור פשרה ניקח ערך אלפא 0.5, ניתן משקל (זהה) גם למרכיב האקטואלי וגם למרכיב ההיסטורי. בעצם נקבל פונקציית מדרגה. בעקומה בכחול ניתן לראות את ההנחה ואילו בעקומה השחורה את זמן הריצה בפועל. ייתכן ובשלב מסוים נגיע ל-steady state: בכל פעם אנחנו עושים ממוצע (כי אלפא = 0.5) בין זמן ההערכה וזמן הריצה בפועל, לכן הגיוני שלאחר כמה פעמים באמת נתכנס לזמן הריצה הדרוש לתהליך (להדגם הבא). זאת בזכות העובדה שאנחנו לוקחים בחשבון גם את המרכיב ההיסטורי וגם את המרכיב האקטואלי. כמובן, ניתן לשנות את ערך אלפא כך שיהיה למשל 0.6, כדי להתקרב טוב יותר, כדי שההבדל בין זמן הריצה בפועל של ההדגם לבין זמן ההערכה שלנו יהיה קטן ככל האפשר.



כמובן, כאשר התהליך 'משתולל' (☺), פתאום דורש יותר זמן, הניחוש יהיה גרוע. ככל שהתהליך ידרוש יחסית את אותו הזמן כל פעם, אנחנו אכן ניתן לו את הזמן המתאים. בעצם SJF מזכיר את LRU [בהקשר של טעינת דפים לזכרון, ראש עמ' 47]. כפי שניתן לראות מהפיתוח של הנוסחה: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\alpha\, t_{n-1} + \ldots + (1-\alpha)^j \alpha\, t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0$ היא בעצם מחשבת ממוצע אספוננציאלי-חזקתי, מה שהופך אותה ליותר מדויקת מאשר ממוצע חשבוני רגיל.

חסרון נוסף של SJF: יש את בעיית אפקט השיירה, אך כאן הוא משמעותי פחות, כי ברגע שנגמרים תהליכים קצרים ונכנס תהליך ארוך, הקצרים שיגיעו מיד לאחר מכן יאלצו לחכות זמן רב. כאן הבעיה האמיתית היא הרעבה של עבודה ארוכה, כי כל עוד יש תהליכים קצרים, תהליך ארוך ימתין עד שלא יהיו כאלו.. בעצם אפליה לרעה ☹ של תהליכים ארוכים.

- priority scheduling – זמנון ע"פ עדיפות. נסדר את ההדגמים בתור, ע"פ עדיפות. נשתמש בפתרון של 'הזקנה' aging: ככל שתהליך ממתין יותר זמן בתור המוכנים-לריצה, נגדיל את העדיפות שלו באופן מלאכותי. זאת, עד למצב שהעדיפות תהיה כ"כ גבוהה שהוא 'ינצח' את כל האחרים, יפסיק להיות מורעב ויזכה לרוץ ☺.

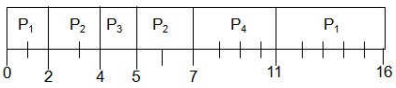
[מצגת os9-2 cpu]

נזכיר גם אלגוריתמים שהם כן preemptive, מאפשרים למערכת לקיחת-שליטה בכוח. אלגוריתמים אלו 'משלחים' את התהליכים לביצוע, תוך אמירה לתהליכים שיש להם פלח-זמן מסוים לרוץ. אם תהליך לא יחזיר את השליטה במסגרת זמן זה, תופעל פסיקת-זמן שתיקח מידיו בכוח את השליטה. פלח הזמן נקרא time quantum (בקיצור q).

- SRJF – shortest remainder job first. כאשר עבודה רצה, חלק מהזמן הדרוש לה היא כבר ניצלה. נבדוק כמה זמן דרוש לה, בהתאם לזמן שהיה דרוש לה מלכתחילה, פחות הזמן שהיא כבר רצה. כאמור, אם יש לתהליך פלח-זמן ארוך יותר מהדרוש לו, הוא יכול להחזיר שליטה כשהוא מסיים את ריצתו.

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SRJF (preemptive) with q = 2

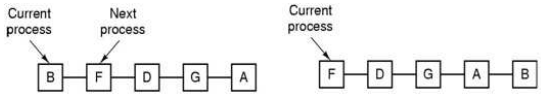


Average waiting time = (9 + 1 + 0 +2)/4 = 3

בדוגמה: לאחר שתהליך 3 סיים לרוץ, לתהליך 2 נותר הכי פחות זמן ולכן הוא זה שרץ. אלגוריתם זה מזכיר בעצם את SJF, רק עם התוספת של פלח-זמן.

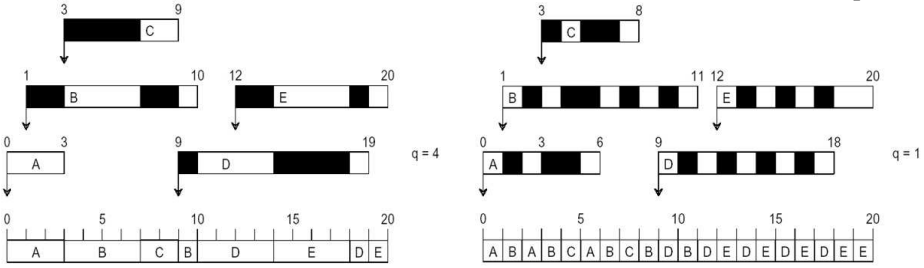
- round-robin – RR (דומה ל-FCFS). כולם נכנסים לתור, לפי סדר ההגעה. בסיום פלח-הזמן, מתווספים לסוף התור.

כביכול מדמה תור מעגלי. **דגש**: אם נטען תהליך חדש באותו זמן שתהליך סיים פלח זמן, התהליך החדש יהיה האחרון בתור.



כאשר ערך ה-q (גודל פלח-הזמן) נמוך מדי, יהיו לנו יותר מדי מיתוגי הקשר ונבזבז זמן על שמירה / טעינה של הדגמים. נרצה q גדול מספיק (אך כמובן לא גדול מדי, כי אז בעצם נאפשר לתהליך לרוץ עד שיסתיים ואין מקביליות..).

לכל תהליך מצוין זמן ההגעה (שמאל) וזמן הסיום (ימין). צבע לבן מייצג זמן ריצה, שחור מייצג זמן המתנה. כאשר q=4 במקום q=1, יש פחות מיתוגי הקשר (פחות מעברים בין צבע לבן לשחור ולהפך).



ככל שיש יותר תהליכים, נרצה שפלח-הזמן יהיה גדול יותר, אחרת יהיו לנו יותר מדי מיתוגי הקשר. חשוב להבין: מיתוג הקשר בעצמו גוזל זמן וזמן זה אינו זניח (למרות שבדוגמאות הנחנו שזמן זה באורך 0..). נסמן: q פלח זמן, s זמן מיתוג הקשר, t ממוצע זמן עד לבקשת הקלט-פלט הבאה. אם $0 \le q \le s$, זו תהיה בחירה לא טובה, שכן רוב הזמן המעבד יהיה עסוק במיתוגי הקשר במקום בהרצה של הדגמים. אם $s < q \le t$, פלח הזמן קצר מדי, לפני ה'ישיא' שהתהליך מבקש קלט-פלט, לכן הוא ישלים את הפעולה רק בריצה הבאה. אם q שואף לאינסוף (ממש גדול), כביכול האלגוריתם דומה ל-FIFO, התהליך ירוץ עד סיומו, זה לא הרעיון. הבחירה הנכונה: $t < q < t + \Delta$, נאפשר לתהליך לבקש קלט-פלט אך לא הרבה יותר מזה.

לא נשאל כיצד לקבוע את גודל הדלתא. לפי הכלל 20-80, נרצה לבחור פלח-זמן כך ש-80% מהתהליכים יצליחו להספיק את הבקשות שלהן במסגרתו. כך נבטיח שזמן התגובה למשתמש הוא מספיק טוב, בממוצע. כאשר מדובר בתהליכי-אצווה ארוכים, פלח-הזמן פחות חשוב, שהרי גם כך יהיו להם הדגמים רבים. כאשר מדובר בתהליכים קצרים, גודל פלח-הזמן משפיע יותר. בזכות העובדה שיש פלח-זמן, יש פחות אפליה לתהליכים הארוכים (הבעיה שהייתה לנו ב-SJF). אמנם, גם כאן לתהליכים הקצרים יישאר פחות זמן לרוץ אחרי שהם ירוצו חלק מהזמן.. אך קיצרנו אותם, כך שהתהליכים הארוכים יחכו פחות.. אין ב-RR אפקט שיירה, כי אחרי שהרצנו תהליך ארוך למשך חלק מהזמן הדרוש לו, נבדוק אם יש תהליכים קצרים שניתן להריץ.
הבעיה של האלגוריתמים שהם preemptive, שהם 'שמים את כל הביצים באותו סל'. יש תהליכים עם מאפיינים שונים: דורשים פלחי-זמן קצרים / ארוכים. כולם יושבים באותו תור ולכאורה יש את אותה מדיניות של זמנון. כאשר מוסיפים את עניין פלח-הזמן, ניתן להציע אלגוריתמיקה מתקדמת יותר. ננהל תורים שונים, לכל תור נשתמש במדיניות שונה. לכן, הרעיון הבא אותו נציע ייקרא: multiple priorities.

- multi-level scheduling – ננהל מספר תורים שונים עם עדיפויות שונות. ככל שהתהליך מופיע בתור בעל מספר גבוה יותר, העדיפות שלו גבוהה יותר. בעיה: כשתהליך מגיע, באיזה תור משבצים אותו? דוגמה: תהליכי אצווה הם ארוכים בעוד תהליכים אינטראקטיביים הם קצרים, לכן נריץ כל אחד בתור אחר. את הקצרים ע"י RR, את הארוכים ע"פ FCFS. בעיה נוספת: צריך גם להחליט איך לחלק את זמן המעבד בין התורים השונים(!). מספר גישות:

(א) ניתן לקבוע שיש עדיפות לתהליכים הקצרים ורק כשאין תהליך קצר, נאפשר לתהליך ארוך לנסות להתקדם קצת.

(ב) לפי הכלל 20-80, ניתן 80% מהזמן לתהליכים קצרים, 20% מהזמן לתהליכים ארוכים.
נניח לדוגמה שהעבודות שייכות למחלקות, וכל מחלקה מגדירה תור ובהתאם גם מגדירה עדיפויות בנפרד. הבעיה: מרגע שהחלטנו שתהליך שייך לתור מסוים, שם נולד ושם ימות, אין מנגנון להעברה בין התורים.

- multi-level feedback queue – אלגוריתם רב-תורי משובי (מלשון משוב). בהתחלה תהליך נמצא בתור עם עדיפות גבוהה, אם יתנהג 'לא יפה' (לא יחזיר שליטה במסגרת הזמן שהוקצב לו) נוריד אותו בדרגה לתור בעל עדיפות נמוכה יותר. בהתאם, אם בהמשך יתנהג 'יפה' יוכל חזרה לעלות לתור בעל עדיפות גבוהה.

בדוגמה, יש לנו 3 תורים עבור סוגים שונים של תהליכים: קצרים, בינוניים, ארוכים. כל עוד נחזיר שליטה במסגרת ה-8 מילי' שניות, נישאר ב-q_0. אחרת, נרד ל-q_1 וכן הלאה.

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR with time quantum 16 milliseconds
- Q_2 – FCFS

כמובן, צריך להחליט מראש על מספר התורים, מדיניות ההשמה ההתחלתית, מדיניות הזמנון בכל תור, מדיניות העלאה בדרגה / הורדה (demote) בדרגה. פרמטר נוסף: מה הסדר בין התורים – העדיפות שניתנת במעבד לכל תור. זה אלגוריתם שמכליל בעצם את כל הבאים אחריו. אם נשתמש רק בתור יחיד, הידרדרנו לאלגוריתמים שהזכרנו קודם לכן.

כיצד נקבע איזה אלגוריתם טוב יותר? עשו מגוון מחקרים והדמיות. קשה לקבוע חד-משמעית. ככלל, עדיף להשתמש באלגוריתם מתקדם יותר. הרעיון של RR טוב לתהליכים אינטראקטיביים, עם פלח-זמן לא קטן מדי (יחסית למשך הזמן t הדרוש לבקשת קלט-פלט). לעומת זאת, גם תהליכי אצווה דורשים זמן רב.

[מצגת os10-3 dsk]

<u>זמנון דיסק – disk scheduling</u>

לבקשת קלט-פלט יש 3 מרכיבים: (א) זמן חיפוש seek time (הזזת הזרוע לגליל היעד המתאים מהגליל הנוכחי), (ב) השהייה סיבובית (עד שהבית הראשון של הגוש שרוצים לקרוא, עובר מתחת לראש הקורא-כותב), (ג) זמן הקריאה של הגוש עצמו. יש חשיבות לסדר של בקשת הפעולות (רצף של מספרי הגלילים המבוקשים), תלוי בזמן החיפוש (תלוי בגורמים האלקטרו-מכניים). נרצה שהזמן שהראש הקורא-כותב ינוע יהיה קצר. הזמן של הפעולה עצמה (חלק ג' לעיל) לא תלוי באלגוריתמיקה עצמה.

נניח שיש לנו 200 גלילים (ממוספים מ-0 עד 199). שני דברים מעניינים אותנו: היכן הזרוע כרגע והאם מדובר בקלט סטטי כך שכל הבקשות ידועות מראש או דינאמי שכל פעם יודעים רק מה הבקשה הבאה. נתון לנו תור הבקשות, אנחנו כרגע בגליל 53.

queue = 98, 183, 37, 122, 14, 124, 65, 67 head starts at 53

נניח שזמן החיפוש מונוטוני (גם זה לא נכון, לא ניכנס לזה). נבחן מספר אלגוריתמים:

(א) FCFS (first come first served, דמוי FIFO) – תור העדיפות הוא לפי סדר ההגעה, אין שום מיון של התור.

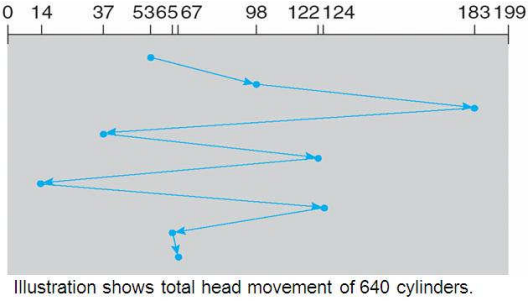


Illustration shows total head movement of 640 cylinders.

חסרונות: תופעה של זיג-זג גלובלי, סך מרחקי החיפוש יחסית גבוה (מרחק החיפוש הכולל עבור הדוגמה הוא 640 גלילים), ניתן 'לשגע' אותו עם סדר בקשות לא טוב.

(ב) SSTF (shortest seek time first) – (כמו SJF, בהקשר של זמנון מעבד) נלך לגליל עם זמן החיפוש הקצר ביותר, בכל שלב.

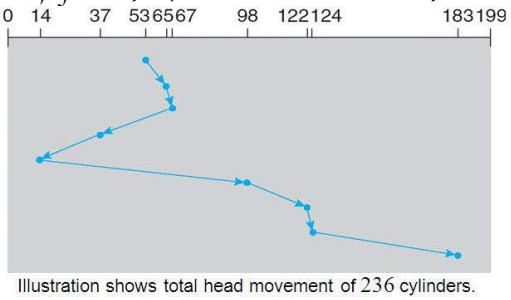


Illustration shows total head movement of 236 cylinders.

יתרון: אין לו זיג-זג גלובלי, אך יש לו מקומי. יכולנו למשל קודם לשרת את (להגיע לגלילים שמספרם) 37-14 ורק אח"כ ללכת למספרים הגדולים ובכך לחסוך זמן חיפוש נוסף. אך כאן טמון החסרון של הרעבה: הראש הקורא-כותב קודם יישאר בתחום מסוים לטובת בקשות שלוקח מעט זמן להגיע אליהן ורק אח"כ ילך רחוק יותר לבקשות שדרוש יותר זמן לחפש אותן.

בהקשר זה, צריך לדבר על גלילים אמצעים ועל קיצוניים. כאן יש אפליה-לטובה לטובת הגלילים האמצעים (יחסית לקיצוניים). נוכל להתחכם ולפעול הפוך-על-הפוך, אם נדע מראש שזהו האלגוריתם, ע"י כך שנשים את הקבצים החשובים בגלילים האמצעיים. אבל זה כבר כדור שלג, כי זה יגרום לראש הקורא-כותב עוד יותר להישאר באמצע ☺. הרעבה: אם מגיע רצף אינסופי של בקשות לגליל מסוים, הזרוע לא תזוז ממנו. לכן, לחלק מהאלגוריתמים יש היגיון של בקשה-יחידה-לגליל, ולאחר מכן הם עוברים לגליל אחר, בהכרח. בדוגמאות עד כה עסקנו בקלט סטטי, בו ידועות לנו מראש כל הבקשות, כל מספרי הגלילים אליהם נצטרך לגשת.

(ג) pick-up (טרמפ) – בעיה: אם אני בדרך מ-14 ל-98 ומגיעה בקשה לגליל 60 – האם נספיק לעצור ולשרת אותו? אלגוריתם זה מתנהג כמו FIFO, הולך לבקשה הבאה לפי הסדר, אבל אם בדרך יש בקשה שהגיעה באופן דינאמי והוא יכול לשרת אותה (עדיין לא הגיע בתנועתו ליעד הסופי, אלא מספיק קרוב ליעד הביניים), הוא ישרת. זה שילוב של FCFS ואלגוריתם נוסף שנראה בהמשך (Look).

האלגוריתמים הבאים שנסקור נקראים 'אלגוריתמי מעלית' (משפחה של אלגוריתמים):

Direction \ Go until	Go until the last cylinder	Go until the last request
Service both directions	**Scan**	**Look**
Service in only one direction	**C-Scan**	**C-Look**

לגבי השמות בטבלה: C מלשון מעגלי (cyclic), שירותיות רק בכיוון יחיד.
Scan – לסרוק עד הגליל הקיצוני, Look – לא להגיע עד הקצה אלא רק עד הבקשה הקיצונית.

נניח שאנו נמצאים במעלית בקומה כלשהי ומבקשים לעלות קומה וגם לרדת קומה. מה המעלית תעשה? היא תמשיך בכיוון התנועה שלה: אם הייתה במהלך עלייה, תמשיך לנוע קודם למעלה ורק אז תרד. יש כאן שני מימדים לדיון: (א) האם אנחנו משרתים בשני הכיוונים, גם מהגלילים החיצוניים לפנימיים וגם בכיוון המנוגד, או רק בכיוון יחיד ואז בבת-אחת חוזרים לקצה ומתחילים שוב לשרת באותו כיוון. (ב) האם זזים עד לגליל הקיצוני (פעולה של סריקה מוגדרת כהגעה עד לקצה), או עד לבקשה האחרונה שיש לנו ואז משנים כיוונים. סה"כ 4 אפשרויות.

(א) Scan – כמו האופטימיזציה ל-SSTF: אם אנו בתנועה שמאלה, נמשיך עד הסוף (Scan מגיע עד הסוף, תמיד) ורק אח"כ נלך ימינה עד הסוף.

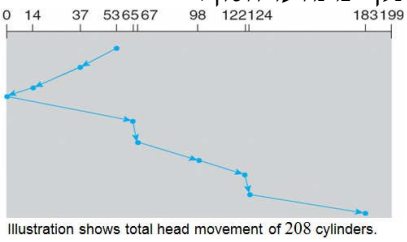


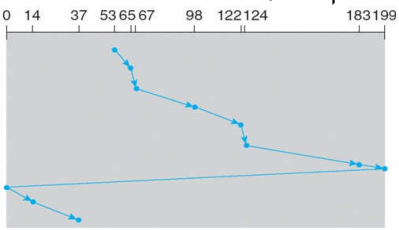
Illustration shows total head movement of 208 cylinders.

יתרונות: חסכנו זיגזוג פנימי, ביחס ל-Look פחות מפלה לרעה גלילים קיצוניים, כי מגיע עד הקצה ונשאר שם קצת זמן לצורך הסריקה. כך אם מגיעה בקשה דינאמית לגליל קיצוני (בדוגמה, בעל מספר נמוך), יוכל לשרת אותה למשל כאשר הוא נע מ-14 ל-0 או בחזרה.

(ב) Look – לא מגיע עד הקצוות, אלא רק עד הגליל הקיצוני אליו הייתה בקשה. יסתובב ב-14 וב-183, יחסוך עוד קצת. מרחק החיפוש הטוב ביותר: משרת בקשות במהלך תנועתו בשני הכיוונים, לא רץ עד הקצה סתם בניגוד לאלגוריתמי הסריקה. החסרון: היות ובניגוד ל-Scan לא מגיע עד הקצוות, נשאר פחות זמן בצדדים, מפלה לרעה גלילים קיצוניים. מבין כולם, אלגוריתם זה מפלה (לטובה) את הגלילים האמצעיים במידה רבה ביותר.

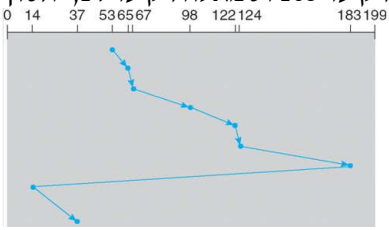
שני האלגוריתמים האלו (Scan ו-Look) אמנם טובים ביחס לקודמים שהזכרנו, אך גם להם יש חסרונות: מפלים לטובה את האמצעיים, על הקצוות עוברים רק פעם יחידה בכל סיבוב. הגלילים האמצעיים מופלים לטובה, עוברים עליהם פעמיים בכל סיבוב, גם במהלך התנועה ימינה וגם במהלך התנועה שמאלה.

(ג) C-Scan – יסרוק רק ימינה (בה"כ) תוך שירות בקשות והגעה עד לקצה, ירוץ עד הסוף שמאלה ללא שירות תהליכים וחוזר חלילה. אמנם התנועה שמאלה היא מהירה, כי לא עוצר לטובת בקשות שירות במהלכה, אבל הוא לא משרת אף תהליך במהלך תנועה זו.



מבין כולם, אלגוריתם זה מפלה (לטובה) הכי פחות את הגלילים האמצעיים.

(ד) C-Look – ינוע בכל כיוון רק עד מיקום הגליל האחרון אליו הייתה בקשה. בדוגמה שלנו, הראש הקורא-כותב ינוע ימינה רק עד 183 ושמאלה רק עד 14, יחסוך קצת.



מפלה את הגלילים החיצוניים יותר ביחס ל-C-Scan, כי לא נע עד הסוף.

היות ושני האלגוריתמים האלו (C-Scan ו-C-Look) משרתים בקשות רק בכיוון אחד, הם מגיעים גם לאמצע וגם לקצה רק פעם אחת בכל סיבוב. כתוצאה מכך לא מפלים לטובה את הגלילים האמצעיים.

סיכום:

- כאשר יש עומס נמוך על הדיסק (low load), האלגוריתם FCFS (משרת את הבקשות לפי סדר הגעתן) מספיק טוב, כי הבקשות מגיעות בטפטוף.
- כאשר יש עומס בינוני, למשל כאשר יש מספר בקשות לאזור ספציפי ומספר בקשות לאזור אחר, האלגוריתם SSTF (משרת את הבקשה עם זמן החיפוש המינימלי תחילה) הכי טוב כי עובד אזור-אזור.
- כאשר יש עומס כבד על הדיסק, עם התפלגות יחסית אחידה של בקשות, המפוזרות על כל מרחבי הדיסקים (בכל הגלילים), אלגוריתמי המעלית הם המתאימים ביותר. [גם במעלית אמיתית] שינוי כיוון של מעלית דורש התערבות אלקטרו-מכנית, לכן תנועה רציפה מעלה-מטה (תנועה מהירה בכיוון מסוים ללא שירות בקשות תוך כדי) יעילה יותר. [במעלית אמיתית] יש פחות חיכוך ובלאי וכך גם תנועת הזרוע מהירה יותר. נרצה במקרים אלו את Scan או C-Scan. בעומס כבד, נרצה אלגוריתם מעלית בשילוב של אלגוריתם טרמפ (pick-up).
- כאשר תוכנית מוציאה קלט-פלט לאותו גליל כל הזמן, נרצה אופטימיזציות כך שלא נישאר כל הזמן באותו גליל. כמובן, אם יש כמה בקשות לאותו גליל, נצטרך לטפל בהן בסדר מסוים כדי לשפר את זמן הטיפול בהן.

בהצלחה במבחן ☺