zvimints@gmail.com
נכתב ע"י צבי מינץ ואילן צדוק

סיכום קורס מערכות הפעלה - סמסטר ב' 2019

עבדנו קשה, מי שנעזר בסיכום מוזמן לתת **endorse** ב-Linkedln ו/או **Star** בעמוד גיט
https://github.com/ZviMints/Summaries
https://www.linkedin.com/in/tzvi-mints-0ba18a180/
https://www.linkedin.com/in/eilon-tsadok-bb4691133/
בהצלחה.

הרצאה 1 + הרצאה 2

מערכות הפעלה: תוכנית שמתווכת בין **חומרת המחשב** לבין **המשתמש** ומייצרת סביבת עבודה בה יכול המשתמש להריץ תוכניות.
מערכת ההפעלה מייצרת נוחות **שימוש וניצול מקסמלי** של חומרת המחשב.

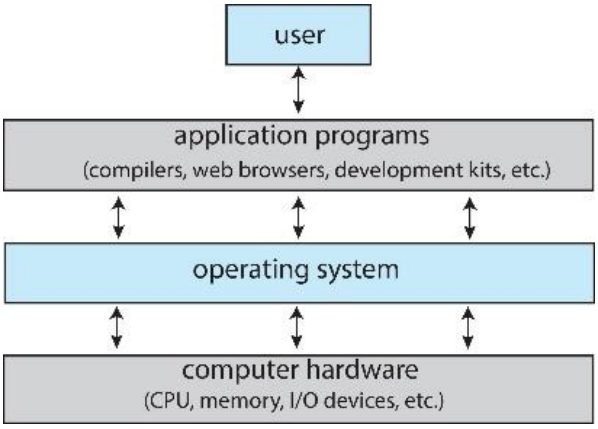
Kernel גרעין / ליבה – קטע קוד, אחד מקטעי הקוד הראשונים שעולים בהדלקת המחשב, כאשר הראשון שעולה הוא ה-bootloader שהוא טוען את ה-Kernel ולאחר מכן הוא היחידה **שרצה כל הזמן על המחשב** והיא **ליבת** מערכת ההפעלה.
מתעוררת בעקבות אירועים אשר נקראים **Interrupt**.

הקרנל שוכן **בזכרון הראשי** (ב-RAM) ואחראי על תפקוד מערכת ההפעלה, כלומר טיפול בפסיקות חומרה, ניהול פעולות קלט/פלט, תזמון שימוש תהליכים (Processes) במעבד וניהול מרחב הזכרון.

תפקידי מערכת ההפעלה:
1. להריץ את תוכניות המשתמש
2. להפוך את בעיות המשתמש לקלות יותר, להפוך את המחשב נוח לשימוש.
3. שימוש בחומרת המחשב בדרך יעילה.

מערכת המחשב מחולקת ל-4 מרכיבים:

1. **חומרה**
   תפקיד: נותנת את משאבי המחשוב הבסיסיים כגון CPU, זיכרון, מכשירי Input-Output (I/O)



2. **מערכת ההפעלה**
   תפקיד: שולטת ומתאמת את השימוש בחומרה בין אפלקציות שונות בשביל משתמשים שונים, תוך ניצול יעיל של מערכת המחשב
3. **אפלקציות**
   תפקיד: מגדירים את הדרכים שבהן יעשה שימוש במשאבי המערכת לפתירת בעיות מחשוב של משתמשים (קומפיילרים, מערכות בסיס נתונים, משחקים, תוכניות עסקיות וכו')
4. **משתמשים**
   תפקיד: אנשים, מכונות או מחשבים אחרים (משתמשי קצה)

קריאת מערכת System Call – קריאת מערכת היא בקשה **שמבצעת תוכנת מחשב** מליבת מערכת ההפעלה על מנת לבצע פעולה שהיא אינה יכולה לבצע בעצמה, קריאת מערכת אחראית על חיבור שבין המשתמש למערכת ובכך מאבזרת את המשתמש בגישה למרבית רכיבי החומר של המחשב.

Interrupt – הוא אות המגיע למעבד מרכיב **חומרה או תוכנה** ומאפשר לשנות את סדר ביצוע הפקודות בתוכנית מחשב שלא על ידי בקרה מותנית (PC).
בעת קבלת הפסיקה משהה המחשב את ביצועה הסדרתי של התוכנית, כדי להפעיל **שגרת טיפול בפסיקה**. לאחר הטיפול, ממשיך המחשב בביצוע הסדרתי של התוכנית. פסיקות משמשות כאמצעי תקשורת בין תהליכים במחשב.

פסיקה **מחומרה** יכולה לקרות בכל שלב ע"י שליחת Signal ל-CPU.
פסיקה **מתוכנה** מתבצעת ע"י System Call.

- המעבד בודק אם יש פסיקות ממתינות לטיפול בין ביצוע של כל שתי פקודות מכונה.
- פסיקה אינה קוטעת ביצוע של הוראת מכונה בסיסית.
- במידה ויש פסיקה ההמתינה, המעבד עובר ל-Kernel Mode וקורא **לשגרת הטיפול בפסיקה**.

שגרת טיפול בפסיקה – Interrupt Service Routine
בלוק מיוחד של קוד אשר מקושר עם פסיקה ספציפית.
למשל, קוראת את התו שנלחץ במקלדת ושומרת אותו במקום מתאים בזיכרון.

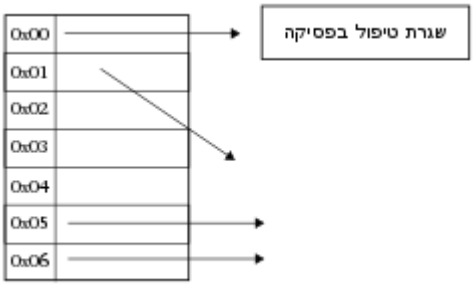
ניתן לחלק את הפסיקות למספר סוגים:

פסיקה **סינכרונית** – פסיקת תוכנה, נוצרת ע"י **המעבד** כתגובה על ביצוע פקודות מכונה אי חוקיות מסויימות.
למשל: חלוקה ב-0

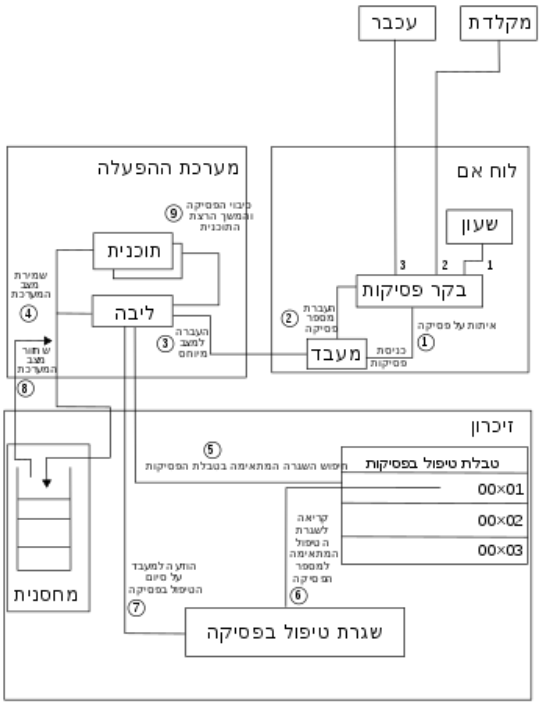
פסיקה **אסינכרוניות** – פסיקת חומרה – יכול להגיע בכל רגע ללא קשר למצב המעבד באותו הזמן
למשל: לחיצה על מקש במקלדת

תהליך טיפול בפסיקה:
1. החומרה מעבירה את הבקרה למערכת ההפעלה
2. שמירת מצב ה-CPU ע"י שמירת ערכי הרגיסטרים וה-PC
3. קביעת סוג הפסיקה בעזרת **Interrupt Vector Table**
4. הפעלת קטע המתאים עבור הפסיקה

טבלת פסיקות IVT: מבנה נתונים שמכיל רשימה של מצביעים ל- Interrupt Service Routine



כתובות 0-31 שמורות לפסיקות הנוצרות על ידי המעבד לדיווח על שגיאות (חריגות) והיתר פנויות עבור התקנים לשימושם.



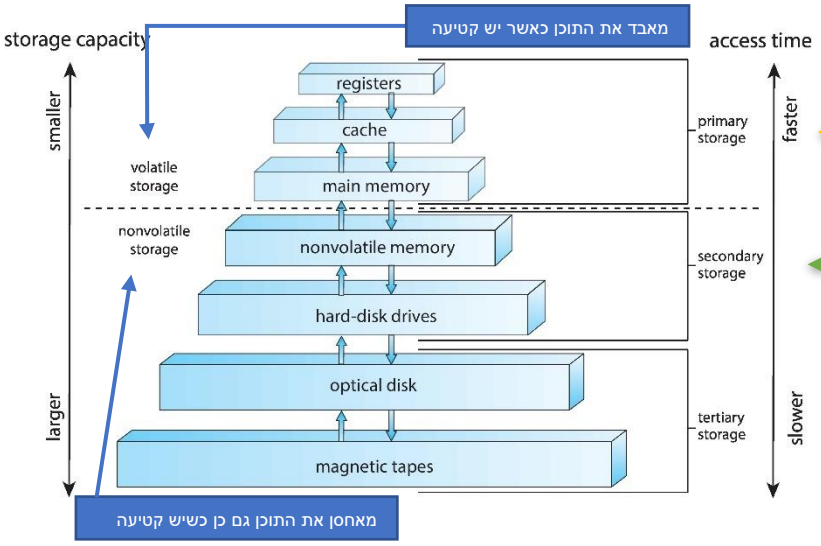
מעבר בין User Mode ל-Kernel Mode

תוכניות משתמש רצות ב-User Mode ומערכת ההפעלה מריצה את הפקודות שלה ב-Kernel Mode כאשר רק במצב זה המעבד מבצע פקודות מוגנות.
כאשר למשתמש יש צורך לגשת לרכיב קלט/פלט, הוא יכתוב פקודת System Call אשר תעביר את המעבד למצב Kernel Mode, המעבד יעשה את המוטל עליו, ויחזור ל- User Mode.

חשיבות שני מצבים: המשתמש יכול בקלות למחוק את כל מערכת ההפעלה, בנוסף תהליכים שונות יכולים לכתוב לרכבי פלט באותו זמן ללא בקרת תזמון.

Memory Main – זיכרון ראשי. המקום שבו תוכניות רצות. ה-CPU יכול לעבוד איתו בצורה ישירה.
זיכרון זה מתחלק ל-2 סוגים:
ROM - Memory Only Read – זמין לקריאה בלבד . מכיל מידע הדרוש למערכת ההפעלה על מנת להעלות את המחשב. לא נמחק כאשר המחשב נכבה. מידע צרוב.
RAM - Memory Access Read – זיכרון חשמלי, נמחק עם כיבוי המחשב.

Storage Secondary – זיכרון משני . הרחבות של הזיכרון הראשי המספקות יכולת אחסון מאוד גבוהה. זיכרון מכני. גישה אליו נעשית בעזרת interrupt.



BIOS – מנגון של חומרה קושחה שצרובה על החומרה שמשתמשת לאתחול המחשב, מכיל סדרה של פעולות שיש לבצע בשביל להדליק את החומרה הבסיסית של המחשב.
בנוסף ל-BIOS יש מבחן שנקרא POST (Power On Self Test) אשר מטרתו לוודא שהמחשב נפגש עם הדרישות להפעלה תקינה, אם המחשב לא עובר את POST אז נקבל סדרה של רעשים "ביפ" אשר לפיהם אפשר לדעת מה התקלה במחשב.

אתחול (Booting):

ספק כוח מפעיל סיגנל שהולך ללוח האם

↓

לוח האם טוען ומפעיל את ה-BIOS.

↓

ה-BIOS מבצע בדיקת תקינות חומרה (POST) – בודק שיש CPU, מנקה רגיסטים וכו'

↓

ה-BIOS טוען את ה- (Boot Loader) Boot Sequence (מערכת הפעלה פשוטה זו נקראת" **מנהל אתחול**" והיא אחראית על טעינת מערכת ההפעלה)

↓

ה-Boot Loader טוען את ההתקן הראשון שקיים (טוען את מ"ה)

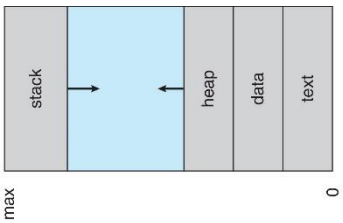
↓

שהוא טוען את ה-Kernel המתאים.

הרצאה 3

Process – תהליך: תוכנית בהרצה.

- מי דואג לניהול המחסנית? קומפיילר.
- כיצד תהליך נראה בזכרון?

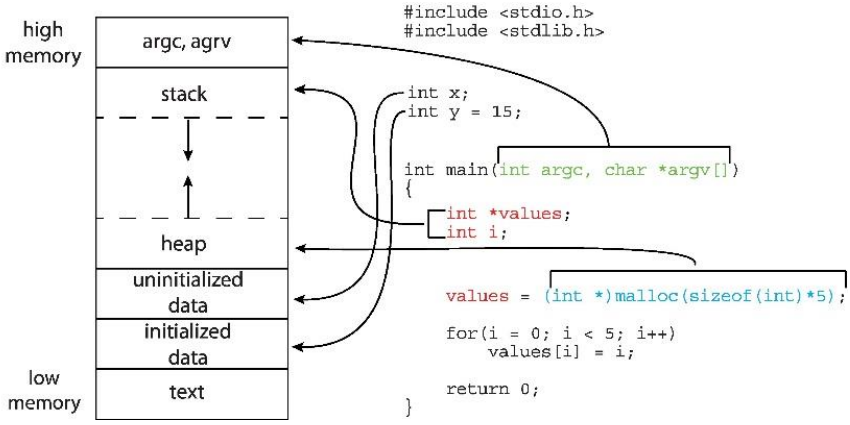


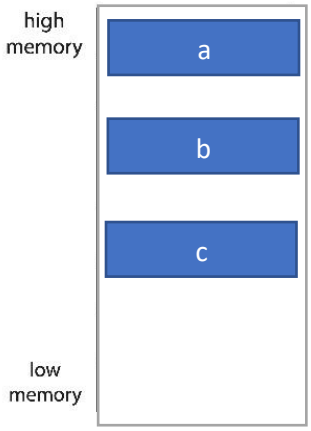
תזכורת: ניהול זכרון בשפת C:

סדרה כניסה למחסנית:

```c
int main()
{
    int a = 5;
    int b = 6;
    int c = 7;
    return 0;
}
```

```asm
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 5
        mov     DWORD PTR [rbp-8], 6
        mov     DWORD PTR [rbp-12], 7
        mov     eax, 0
        pop     rbp
        ret
```

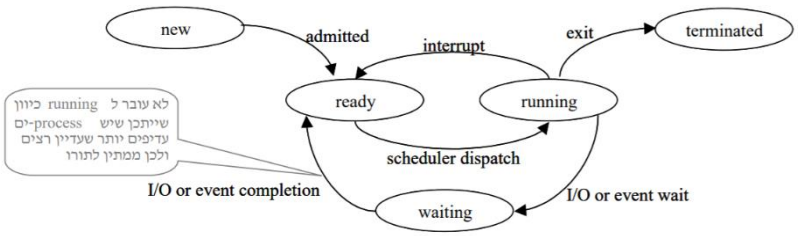




מספר התהליכים שרצים במקביל הוא כמקסימום מספר הליבות, כל **ליבה יכול להריץ תהליך אחד בלבד**.

מצבים של תהליך:

1. New – ברגע שה-process נוצר.
2. Running – ה-process מריץ פקודות. **בזמן נתון רק process אחד יכול להימצא במצב זה.**
3. Waiting – ה-process ממתין ל-event (לרוב I/O). ממצב זה לא ניתן לעבור לריצה.
4. Ready – ה-process ממתין לקבל את ה-CPU. מוכן לעבודה.
5. Terminated – ה-process הסתיים.



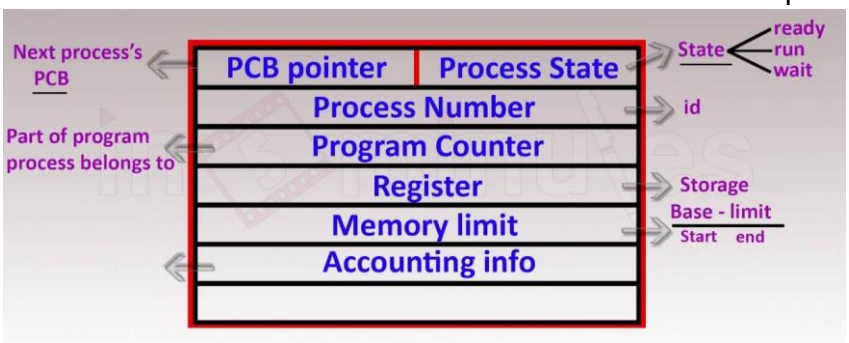
בלוק שליטה - PCB - Block Control Process:

process state
process number
program counter
registers
memory limits
list of open files
• • •

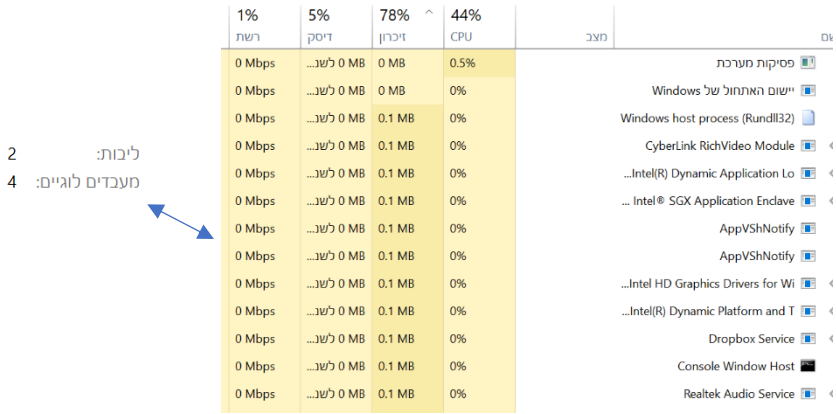
כל תהליך מיוצג במערכת ההפעלה ע"י PCB, המכיל מידע המשויך לתהליך מסוים. בין היתר כולל ה-PCB:

1. מצב התהליך.(... ready, new)
2. Program Counter – כתובת הפעולה הבאה שצריכה להתבצע ע"י התהליך.
3. CPU registers – כאשר מתרחש interrupt יש צורך לשמור את מצב הרגיסטרים במערכת, במטרה לאפשר לתהליך לחזור למצבו הקודם, לפני התרחשות ה-Interrupt.
4. CPU scheduling information – עדיפות התהליך, מצביע לתור התזמונים ופרמטרים נו ספים של תזמון. Unix מבטא עדיפות ע"י הפרמטר nice, שמציין כמה אתה מוכן לוותר על ה- CPU לתהליכים אחרים.
5. מידע על ניהול זיכרון – כולל בין היתר ערך ה-base register ו-limit register, טבלת הדפים ועוד.
6. Accounting information – בכמה CPU הוא משתמש, גבולות זמנים, מספרי תהליכים ועוד.
7. מצב I/O – רשימת התקנים בהם משתמש התהליך, רשימת קבצים פתוחים.

- PCB זה מבנה נתונים בתוך **הקרנל** שנותן הקשר בנוגע למצב של התהליך.
- **תוכנית שרצה ללא PCB זה מערכת ההפעלה**. ולכן בפרט מערכת ההפעלה היא לא תהליך, היות ולכל תהליך יש PCB.



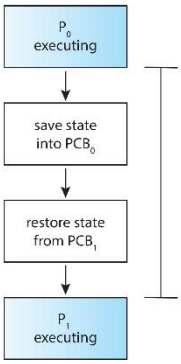
בגלל שיש הרבה תהליכים שרצים במקביל ובגלל שיש יותר מעבדים מאשר ליבות אז נצטרך **מתזמן תהליכים**.



מתזמן תהליכים:

- ממקסם את צריכת המעבד
- בוחר מכלל התהליכים אשר נמצאים במצב Ready את התהליך הבא שירוץ על מעבד

החלפת קשר – Context Switch



מעבר בין הרצת שני תהליכים באמצעות המעבד. באמצעות החלפת הקשר, מספר תהליכים יכולים לחלוק את אותו מעבד.
החלפת ה-CPU לתהליך אחר דורשת שמירת המצב הקיים(PCB) של התהליך הישן , וטעינת המצב של התהליך החדש.

תהליך החלפת ההקשר נחשב לתהליך בזבזני מבחינת משאבי מערכת ועל כן מערכות הפעלה מנסות לבצע אופטימיזציה בשימוש בהן.

מתי יש צורך בשיתוף תהליכים:

א. שיתוף מידע.
ב. זירוז חישוביות. במקום שתהליך אחד יבצע הכול, כל תהליך מבצע פעולה קטנה.
ג. מודולאריות.
ד. נוחות.

הרצאה 4 + הרצאה 5

קריאת מערכת	חתימה	תפקיד	ערך חזרה
Fork()	`pid_t fork()`	יוצר תהליך בן עם העתק של ה-PCB.	כישלון: -1 אחרת, 0 לבן ו-pid לאב.
Exit()	`void exit(int status);`	מסיימת את ביצוע התהליך הקורא ומשחררת את כל המשאבים שברשותו. התהליך עובר למצב zombie עד שתהליך האב יבקש לבדוק את סיומו ואז יפונה לחלוטין	הקריאה אינה חוזרת, **תמיד** מצליח!
Wait()	`pid_t wait(int *status);`	גורמת לתהליך הקורא להמתין עד אשר אחד מתהליכי הבן שלו יסיים **status** – מצביע למשתנה בו יאוחסן סטטוס הבן שסיים	**אם** אין בנים, או שכל הבנים כבר סיימו וכבר בוצע להם wait() - חזרה מיד עם ערך -1 **אם** יש בן שסיים, כלומר זומבי, ועדיין לא בוצע לו wait() – חזרה מייד עם ה-pid של הבן הנ"ל ועם סטטוס הסיום שלו. **אחרת**, המתנה עד שבן כלשהו יסיים
Pipe()	`int pipe(int filedes[2]);`	יוצרת pipe חדש של 2 descriptors, אחד לקריאה ואחד לכתיבה	0 בהצלחה ו-1 בכשלון
Getpid(	`pid_t getpid(void);`	להחזיר את ה-pid של התהליך הנוכחי	↔
Getppid()	`pid_t getppid(void);`	להחזיר את ה-pid של **האבא** של התהליך הנוכחי	↔

תהליך זומבי:
תהליך **שסיים** אך מחכה **לעדכן** את תהליך האב. אף אחד לא קרא את ערך החזרה עדיין.

כאשר תהליך מסתיים, כל הזיכרון והמשאבים ששויכו אליו משוחררים חזרה למערכת. יחד עם זאת, הרשומה של התהליך בטבלת התהליכים אינה מוסרת. התהליך האב שיצר את התהליך הבן שהסתיים יכול לקרוא את ערך היציאה של הבן באמצעות קריאת המערכת wait, ורק מרגע שזו נקראה התהליך הזומבי מוסר מהרשימה. לאחר שהזומבי מוסר מרשימת התהליכים, מזהה התהליך שלו מתפנה לשימוש חוזר בידי תהליך חדש אחר שנוצר. יחד עם זאת, אם ההורה לא קרא ל-wait, הזומבי יישאר ברשימת התהליכים עד סיום ריצתו של התהליך האב. מרגע שהתהליך האב הסתיים, עובר תהליך הבן לתהליך init, שעובר באופן מחזורי (טיימר) על רשימת התהליכים הזומבים ומשחרר אותם על ידי קריאה ל-wait.

הרצאה 6

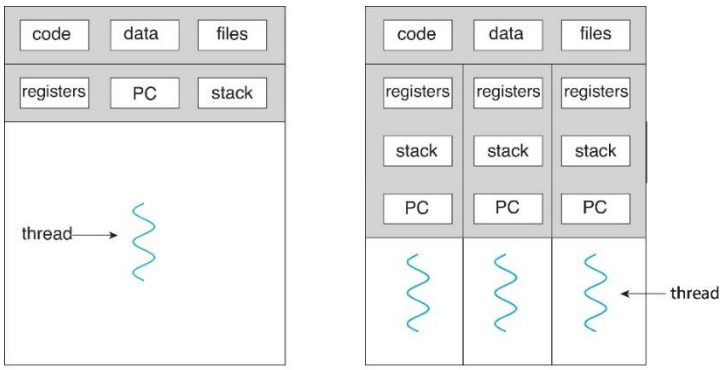
תהליכון Thread : חלק מתהליך, משתף משאבים עם תהליכונים אחרים אשר נמצאים באותו תהליך, מכיל:

1. **PC**
2. **Registers**
3. **Stack**

מופרדים, וכל השאר מה-PCB אותו דבר.

בכל תהליך יש מספר תהליכונים והם חולקים **יחד**:

1. Code Section
2. Data Section - נתונים שלא במחסנית כגון: משתנים גלובליים, סטטיים, קבצים שנפתחו וכו'.



single-threaded process multithreaded process

הערות:

- ברגע ש process רץ לבד, והוא לא מייצר process-ים אחרים, אז הוא thread אחד.
- fork לא מייצרת thread-ים , היא מייצרת process חדש, כלומר מייצרת code section ו data – section חדשים.
- תהליך יצירה של thread הוא תהליך הרבה יותר מהיר היות ולא משכפל את הPCB במלואו ביחס לתהליך יצירה של process
- אם נרצה יותר משאבים ממה שמערכת מביאה אז נרצה ליצור process חדש ולא thread
- נרצה להשתמש בthread כאשר נרצה לעשות פעולות מהירות על אותו מרחב זכרון.

תקשורת:

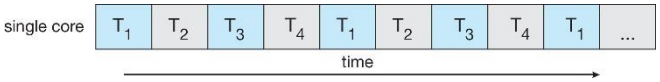
תקשורת בין תהליכים מתבצעת בעזרת Pipes (|) או בעזרת Signals או בעזרת Shared Memory, ואילו תקשורת בין תהליכונים היא לכתיבה לאזור משותף בערמה.

תהליך	תהליכון
יש לו Heap -ו Data Segment	אין לו Heap -ו Data Segment
לתהליך יש לפחות תהליכון אחד	לא יכול לחיות בעצמו, חייב להיות חלק מתהליך
	יכול להיות יותר מתהליכון אחד בתהליך
יצירה יקרה	יצירה לא יקרה
החלפת הקשר יקרה	החלפת הקשר לא יקרה
	תקשורת יעילה
אם תהליך מת, אז כל הזכרון משתחרר וכל התהליכונים מתים 💀	זכרון משתחרר במוות 💀

Concurrent

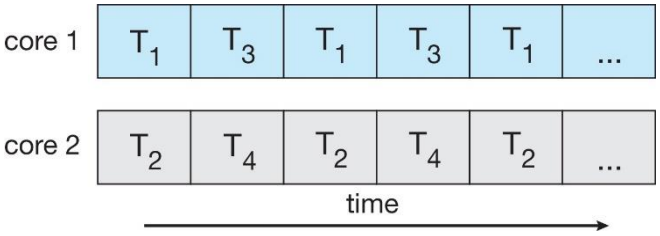
עיבוד בו זמנית של כמה תהליכים או תהליכונים.

איך מבצעים זאת? ע"י החלפת הקשר מהיר, **מקבליות** ע"י מעבד **אחד**.



Parallelism

יש צורך **יותר** מליבה אחת.



- נרצה לשים תהליכונים של אותו תהליך **באותה ליבה** כי אז **המיתוג** יהיה הרבה יותר מהיר כי הם נמצאים באותו מרחב זכרון.

Amdahl's Law

משמש למציאת חסם עליון לשיפור הצפוי במערכת מחשב, כאשר חלקה ממוקבל וחלקה נותר סדרתי. החוק משמש לרוב במחשוב מקבילי לחיזוי ההאצה (Speedup) התאורטית המקסימלית המתקבלת מהרצת המערכת על מספר מעבדים במקביל.

המטרה: לעבוד כמה שיותר מהר

לא ניתן לקבל יותר מ-speedup

S חלק סדרתי | N מספר ליבות

$$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$

הרצאה 7

ישנם 2 סוגים של Threads :

User Thread: מיושם ע"י המשתמש, הגרעין לא מודע כלפיו.

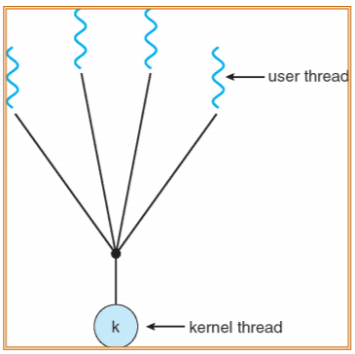
Kernel Thread: מיושם ע"י מערכת ההפעלה, הגרעין מנהל אותו.

ההבדל בין User Thread או Kernel Thread הוא בגדול מי מנהל את התהליך, האם מערכת ההפעלה (קרנל) או המשתמש.

Kernel Thread	User Thread
איטי יותר ליצרה וניהול	מהיר ליצירה וניהול
מערכת ההפעלה תומכת ביצירה של Kernel Thread	מנוהל ע"י המשתמש
Kernel Thread ספציפים למערכת ההפעלה	גנרי ויכול להיות מיושם על כל מערכת הפעלה
יתרונות:	**יתרונות**
הקרנל מודע לתהליכון, אם תהליכון אחד נתקע אז הוא מתזמן אחר	מהיר יותר, אין System Call ליצרה או להחלפת הקשר
חסרונות:	**חסרונות:**
איטי, הקרנל עושה את היצירה ואת התזמון	הקרנל לא מודע להם אז הם הוא לא יכול לבצע תזמון בשבילם, כלומר, אם תהליכון אחד נתקע אז כולם נתקעים. (לדוגמא Seg Fault)

מודלים של מולטי-טרדינג:

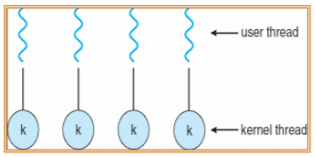
Many-to-One



כמה User-Level-Threads ממופים לKernel Thread יחיד.

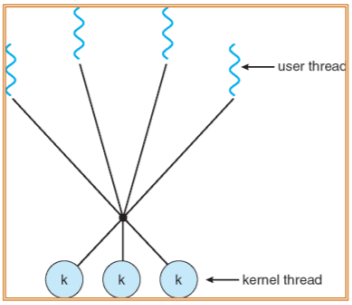
יתרונות:
מהיר מכיוון שלא צריך קריאת מערכת ליצירה שלהם
חסרונות:
אין מקבליות (כמה ליבות), אם תהליכון אחד נתקע אז כולם נתקעים. (לדוגמא המתנה לI/O)

One-to-One



כל User-Level-Thread ממופה ל-Kernel Thread יחיד.
יתרונות:
יותר **Concurrent**
כאשר תהליכון אחד מתקע, אחרים יכולים לרוץ.
ביצועים טובים יותר בכמה ליבות.
חסרונות:
יקר, יצירה וניהול התהילכון מערב את הקרנל בפעולות שלו וצורך את משאבי הקרנל.

Many-to-Many



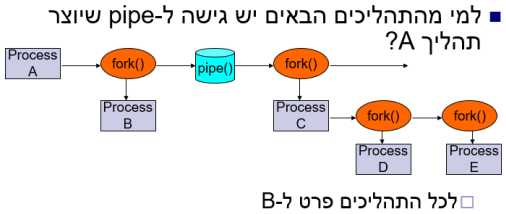
הרבה User-Level-Thread
להרבה Kernel-Level-
Thread, כאשר $U \geq K$

יתרונות:
גמיש.

חסרונות:
בד"כ ישתמש במיפוי 1:1

מהמטלה:

- **Pipes** הם ערוצי תקשורת **חד-כיוונים** המאפשריים העברת נתונים לפי סדר **FIFO**
- Pipes משמשים גם לסנכרון תהליכים.
- הגישה ל-pipe היא באמצעות שני descriptors: אחד לקריאה ואחד לכתיבה, כאשר מקום 0 זה לקריאה ומקום 1 לכתיבה.
- היצירה של pipe היא באמצעות קריאת המערכת ()pipe.
- ה-pipe הנוצר הינו פרטי לתהליך, כלומר אינו נגיש לתהליכים אחרים במערכת.
- הדרך היחידה לשתף pipe בין תהליכים שונים היא באמצעות קשרי משפחה
- תהליך אב יוצר pipe ואחריו יוצר תהליך בן באמצעות ()fork – לאב ולבן יש גישה ל-pipe באמצעות ה-descriptors שלו, המצויים בשניהם
- לאחר סיום השימוש ב-pipe מצד כל התהליכים (סגירת כל ה-descriptors) מפונים משאבי ה-pipe באופן אוטומטי.

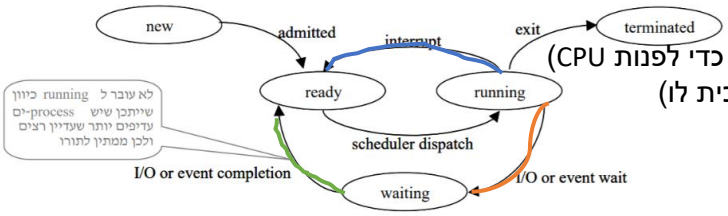


הרצאה 8

CPU Scheduler

ברגע שה-CPU נכנס למצב של סרק, מערכת ההפעלה בוחרת תהליך מתוך ה-Ready Queue ונותן לו לרוץ על ה-CPU, החלטות המתזמן עשויות לקרות באחד מארבעת הנסיבות הבאות:

1. ממצב running למצב wait (מחכה לקלט לדוג')
2. ממצב running למצב ready (בתזמון **פרמטיבי**, כדי לפנות CPU)
3. ממצב waiting למצב ready (קבלת את מה שחכית לו)
4. התהליך מסתיים – זה קורה ביוזמת התהליך.



במקרים 4,1 אין אפשרות בחירה.
התהליך מסתיים מיוזמתו, ותהליך חדש חייב להיבחר מתוך התור.
מקרים מסוג זה נקראים **Non-Preemptive**

Preemptive (נותן זכות קדימה)
מערכת מסוג זה ערה לשינויים, והיא בודקת כל הזמן את מצבי התהליכים ועשויה להפסיק תהליך אחד עבור תהליך בעל עדיפות גבוהה יותר. תזמון מסוג זה גורם לעלויות – זמן.
באחד מהמצבים (2) או (3)

Non-Preemptive

התהליך שומר על ה-CPU עד שהוא משחרר אותו, ולא ייתכן מצב שמערכת ההפעלה תפריע לו באמצע, או תעצור אותו ותיתן עדיפות לתהליך אחר. זוהי מערכת שאדישה לשינויים. תהליך בעל עדיפות גבוהה יותר לא יכול להפריע לתהליך שכבר רץ.

Dispatcher (תזכורת)

מודל זה מעביר את הבקרה של ה-CPU לתהליך שנבחר ע"י ה Scheduler.
פעולה זו כוללת:
1. Context Switch
2. מעבר ל-User Mode
3. קפיצה למקום הנכון בתוכנית המשתמש במטרה להתחיל להריץ אותה (PC).

מודל זה צריך להיות מהיר ככל שאפשר, שכן הוא נקרא בכל פעם שמחליפים תהליך. **הזמן** שלוקח לDispatcher להפסיק את התהליך הרץ ולהתחיל בהרצת תהליך חדשה נקרא **Dispatch latency**

קרטריונים לתזמון:

כאשר אנו מחליטים באיזה אלגוריתם להשתמש במצב מסויים, יש לקחת בחשבון את המאייפנים של כל האלגוריתם. הקרטריונים בהם משתמשים להשוואות אלגוריתם של מתזמני CPU הם :

1. ניצול CPU – נרצה לוודא שה-CPU עסוק כל הזמן. ניצול ה-CPU יכול לנוע מ-0 אחוז ל-100 אחוז. במערכת אמיתית הטווח צריך להיות בין 40 אחוז ל-90 אחוז.
2. תפוקה (Throughput) – יעילות הרצת תהליכים חופפים. אחת הדרכים לבדוק את עבודת האלגוריתם היא מספר התהליכים שסיימו ביחידת זמן אחת.
3. זמן סבב (Turnaround) – מנקודת מבט של תהליך ספציפי , הקריטריון החשוב הוא כמה זמן לוקח להריץ את התהליך . המרווח בין הזמן שהתהליך ביקש לרוץ לבין הזמן שהתהליך יסתיים נקרא זמן סבב. זמן זה הוא סכום משך הזמנים שהתהליך בזבז בהמתנה לזיכרון , המתנה בתור ready, ריצה ב-CPU והפעלת I/O.(מדידה מהרגע שנכנס ל memory queue)
4. זמן המתנה – האלגוריתם אינו משפיע על הזמן בו תהליך רץ או משתמש ב- I/O, אלא רק על הזמן שתהליך מעביר בהמתנה בתור ה- ready. זמן ההמתנה הוא סכום משך הזמנים הכולל שהתהליך העביר בתור ה-ready.(כשזמן ההמתנה אינסופי אז יש הרעבה)
5. זמן תגובה – במערכת אינטראקטיבית, זמן הסבב אינו יכול להיות קריטריון מתאים, משום שלעיתים קרובות, תהליך יכול לספק חלק מהפלט די מהר , ולהמשיך בחישובים אחרים בזמן שחלק מהפלט כבר מוצג למשתמש . לכן אמת מידה נוספת הי א הזמן שלוקח מרגע הבקשה ועד לתגובה הראשונה . זמן התגובה הנו משך הזמן שלוקח להתחיל להגיב, אבל לא כולל את זמן התגובה עצמו.
קובע את מידת ההגינות של המערכת – זמן תגובה דומה לכל התהליכים זה ממתי שנכנס ל ready queue ועד שקיבל cpu ועבר ל running.

המטרה היא להביא את ניצול ה- CPU ואת התפוקה למצב מקסימ אלי, ולמזער כמה שיותר את זמן הסבב , זמן ההמתנה וזמן התגובה. ברוב המקרים מנסים לייעל את הממוצע של כל הקריטריונים , אך קיימים מצבים בהם רצוי לייעל את ערכי המקסימום או המינימום ולא את הממוצע . לדוגמא, כאשר רוצים להבטיח שכל המשתמשים יקבלו שירות טוב, נרצה להקטין את זמן התגובה המקסימאלי.

אלגורתמים שונים לתזמון:

FCFS – ראשון בא, ראשון ישורת. First Come, First Served :

המעבד מוקצה ראשון לתהליך הראשון שביקש אותו.
הדרך הפשוטה ביותר לממש אלגורתים זה הוא בעזרת תור FIFO.
FIFO מתזמן את התהליכים באופן הגעתם ל-Ready Queue, כלומר, התהליך שמגיע ראשון יתוזמן ראשון והתהליך שבא אחריו יתוזמן אך ורק לאחר שהוא יסיים.

אלגוריתם Non-Preemptive.

זמן ההמתנה הממוצע במקרה זה הוא לרוב ארוך, והוא תלוי בזמן ההגעה של התהליכים.

דוגמא:

Process	Burst time
P1	24
P2	3
P3	3

הזמן שלוקח לסיים

אם התהליכים הגיעו על פי הסדר הבא: P1, P2, P3, נקבל את המצב הבא:

זמן ההמתנה של P1 יהיה 0 (משום שהוא התחיל מיד), זמן ההמתנה של P2 יהיה 24, וזמן ההמתנה של P3 יהיה 27. זמן ההמתנה הממוצע שמתקבל הינו (0+24+27)/3=17.

לעומת זאת, אם התהליכים היו מגיעים בסדר P2,P3,P1 אז זמן הממוצע שהיה מתקבל הוא $\frac{0+3+6}{3} = 3$.

- באלגוריתם זה תיתכן הרעבה, ברגע שתהליך שקיבל את זמן המעבד נכנס ללולאה אינסופית.

הקצר ביותר ראשון – SJF Shortest Job First

אלגוריתם Non-Preemptive.

Algorithm:

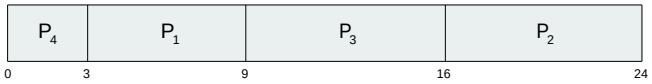
```
1- Sort all the processes in increasing order
   according to burst time.
2- Then simply, apply FCFS.
```

- זהו אלגוריתם אופטימאלי הנותן זמן המתנה **מינמאלי**
- אם יש הרבה תהליכים קצרים ותהליך אחד ארוך, אז יתכן הרעבה של תהליך הארוך.
- הקבלה לאלגוריתם זה הוא "אפשר רק לשאול שאלה קצרה?"
- ה-Turnaround Time לא אופטמלי כלל (הזמן שתהליך מחכה מהרגע שנוצר עד רגע הסיום)

דוגמא:

Process	Burst time
P1	6
P2	8
P3	7
P4	3

שימוש באלגוריתם זה יתזמן את התהליכים בסדר הבא: P4,P1,P3,P2, נקבל כי זמן ההמתנה הממוצע יהיה $\frac{3+16+9+0}{4} = 7$



הבעיה: איך יודעים מה זמן הריצה של כל אחד?
פתרון: לאסוף סטטיקה לאורך זמן ע"י

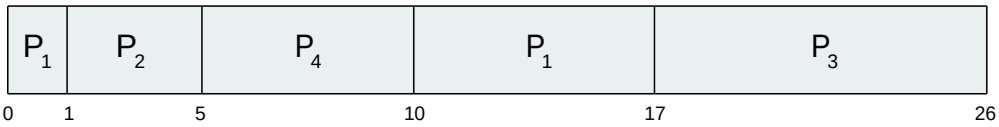
1. t_n = actual length of nth CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

על מנת לחשב צריך הערכה אחרונה וריצה אחרונה.

Shortest Remaining Time First

גרסא **Preemptive** של SJF
באלגוריתם זה, הריצה של תהליך כלשהו יכול להעצר לאחר זמן מסויים, בעת ההגעה של כל תהליך, המתזמן את התהליך בעל זמן הריצה הקצר ביותר שנשאר כל פעימה של השעון.
דוגמא:

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



זמן המתנה ממוצע: $\frac{[(10-1)+(1-1)+(17-2)+(5-3)]}{4} = \frac{26}{4} = 6.5\ msec$

Round-Robin Scheduling – תזמון סרט נע

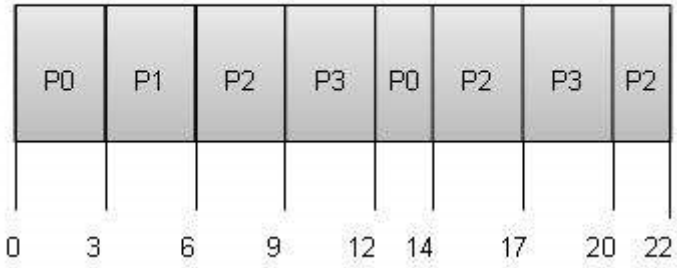
אלגוריתם Preemptive.

כל תהליך מקבל זמן קבוע לביצוע, זמן זה נקרא quantum (בד"כ 10-100 מילישניות)
מתייחס אל תור ה-Ready Queue כאל תור מעגלי.
מתזמן המעבד עובר על התור ומקצה זמן עיבוד בכל פעם לתהליך אחר לזמן של quantum, כאשר הזמן מסתיים, התהליך נוסף לסוף ה-Queue.
אם יש n תהליכים ב-Ready Queue אזי כל תהליך מקבל $\frac{1}{n}$ מהזמן הCPU בצ'אנקים של לכל היותר quantum יחידות זמן, אין תהליך שיחכה יותר מ- $(n-1)\cdot quantum$
אם $quantum$ יהיה מאוד גדול, נקבל FCFS, לעומת זאת, אם הוא יהיה קטן מאוד, נגיע לבעיה, היותת ולוקח זמן למתג בין תהליכים ונתחיל להתנגש עם dispatch latency.

דוגמא:

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16

Quantum = 3

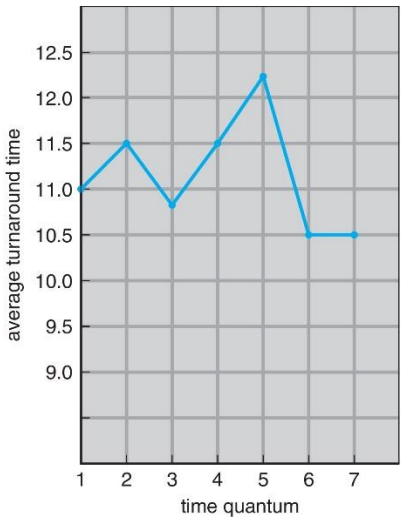


תהליך	Burst time	זמן המתנה: (זמן ביצוע – זמן הגעה)
P_0	5	(0-0) + (12-3) = 9
P_1	6	(3-1) = 2
P_2	8	(6-2)+(14-9)+(20-17) = 12
P_3	3	(9-3)+(17-12) = 11

זמן המתנה ממוצע: $\frac{9+2+12+11}{4} = 8.5$

בד"כ זמן הממוצע שכל תהליך מחכה הוא יותר גדול מ-SJF אבל מגיב לשינויים טוב יותר.
העיקרון: Context Switch צריך להיות זניח ביחס ל-Quantum time

איך ה- לQuantum time משפיע ע"י ה-Turnaroundtime?
נרצה לעשות כמה שפחות מעברים בין תהליכים עד רמה מסויימת, אולם לא לתת לתהליכים הארוכים לעכב את כולם, ולכן נחפש את עמק השווה.
ניתן לראות בדוגמא כי עמק השווה הוא בערך כש- ל Quantum = 6



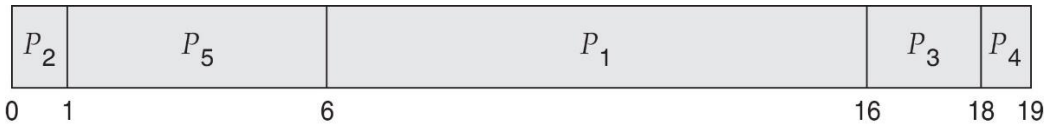
process	time
P_1	6
P_2	3
P_3	1
P_4	7

Priority Scheduling

אלגוריתם Non- Preemptive.
אלגוריתם אשר מסתכל על תעדוף בין תהליכים. המעבד יקצה זמן לתהליכים עם עדיפות הגבוהה ביותר. ככל שהמספר שה-Priority קטן יותר אז העדיפות גבוהה יותר.

בעית הרעבה: תהליכים בעלי עדיפות נמוכה יכולים לא לקבל זמן מעבד אף פעם.
פתרון: Aging (תהליך הזדקנות) ככל שעובר זמן ותהליך מסויים לא קיבל עדיפות, נעלה לו את ה-Priority בצורה יזומה.
דוגמא:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

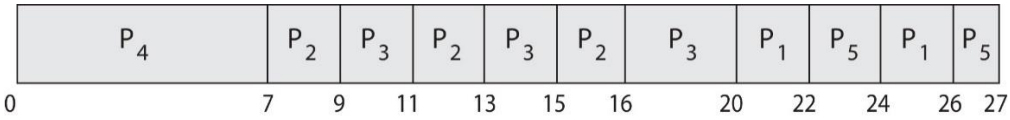


זמן המתנה ממוצע: $8.2\ msec$

Priority Scheduling w/ Round-Robin

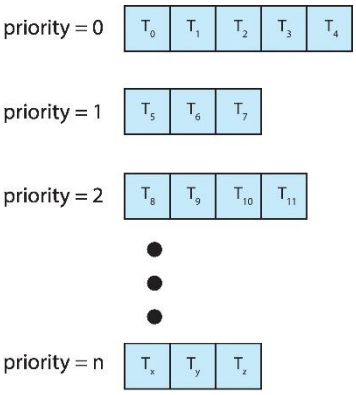
שילוב של 2 האלגורתמים, כאשר קיימים 2 תהליכים עם אותה עדיפות, נתזמן בינהם עם Round-Robin
דוגמא:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3



Multilevel Queue

לכל עדיפות ניצור תור של תהליכים.
בתוך כל תור יש למעשה אלגורתמים תזמון בין התהליכים השונים שנצטרך להחליט עליו, לכל תור נוכל לתת אסטרטגיה שונה.



איך נחליט איזה תהליך שייך לאיזו עדיפות?
לפי סוג התהליך. תהליכים שהם Real-Time (תהליכים שצריכים לסיים את פעולתם עד זמן נתון) יהיו בעדיפות ראשונה, לאחר מכן יבואו תהליכי מערכת ולאחר מכן תהליכי משתמש ולאחר מכן Batch Process (תהליכי אופטמזציה).

Multilevel Feedback Queue

שילוב עוד יותר מתקדם של כמה תורים, נייצר אינטרציה בין התורים.
נקח תהליך מתור אחד ונעביר לתור אחר, נוכל לממש ככה Aging.
Aging – אם תהליך היה בעדיפות נמוכה ולא קיבל זמן עיבוד, נוכל להגדיל את העדיפות שלו ע"י העברה לתור אחר.

הרצאה 11-10-9

הקדמה:
זמן הגישה ל- Register הינו 1 Nano Second, כאשר כל אחד במדרגה (של הזכרון) מגדיל פי ~10, כלומר ל-RAM נגיע ב 100ns.

10^{10} – קילו
10^{20} - מגה

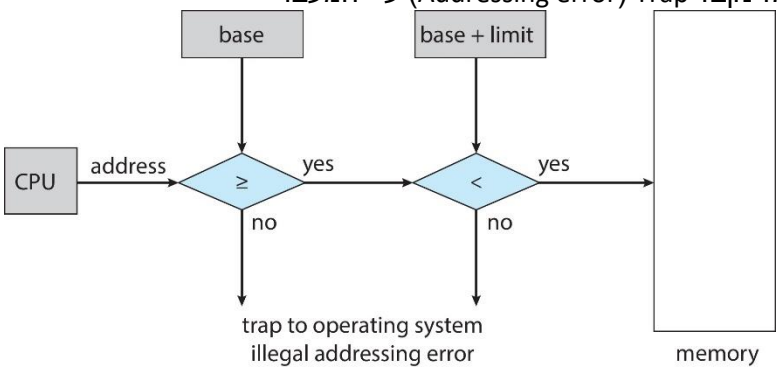
מטרות ניהול זכרון:
1. יכולת להריץ מספר תוכניות בו-זמנית תוך כדי הגנה על הזיכרון של כל תוכנית מפני האחרות.
2. יכולת להריץ בו-זמנית תוכניות שסך כל הזיכרון שהוקצה להן גדול מהזיכרון המותקן במחשב.
3. יכולת להשתמש בדיסקים כהרחבה זולה אך איטית של הזיכרון ;המעבד אינו יכול להתייחס ישירות למידע בדיסק
4. יכולת להזיז את מבני הנתונים של תוכנית במהלך ריצתה בלי שהתוכנית מודעת להזזות הללו; הזזות כאלו מאפשרות לנצל "חורים" קטנים בזיכרון קטנים בזיכרון, או לאחד חורים לזיכרון פנוי רצוף או לאחד חורים לזיכרון פנוי רצוף, או להעביר מבנה נתונים מהזיכרון לדיסק ומשם למקום אחר בזיכרון.

הגנה על מידע:
צריך להגן על הזיכרון של התהליך כך שכל תהליך יכול לגשת אך ורק למרחב כתובות שלו והוא לא יכול לגשת למרחב הכתובות של התהליכים האחרים, כאשר אנחנו קוראים/כותבים אנחנו בונים על זה שאף אחד לא שינה את התוכן, בנוסף, אם יש תהליך זדוני שמנסה לגנוב מידע אז נרצה שיהיו הפרדות ברמת הזכרון בין תהליכים.

אנחנו נדבר על כמה מודלים לפתור בעיה זו.

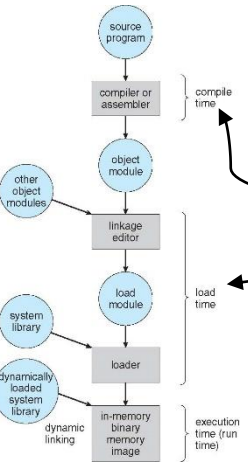
המודל הראשון (מנגנון חומרתי) מדבר על הקצאת Base ו-Limit, כך שלכל תהליך יהיה מרחב זכרון בתחום כתובות [Base, Base + Limit), דבר זה ימנע את האפשרות לגשת מתהליך כלשהו למרחב זכרון שלא שלו.

אם הקומפיילר היה קבוע את ה-Base וה-Limit, אז אפשרות זו **לא טובה**, כי המכונה לא תוכל להריץ את אותו תוכנית פעמיים כי הם היו מעל אותם מרחב כתובות.
אם יש חריגה אז נקבל Trap (Addressing error) ע"י המעבד



Address Binding

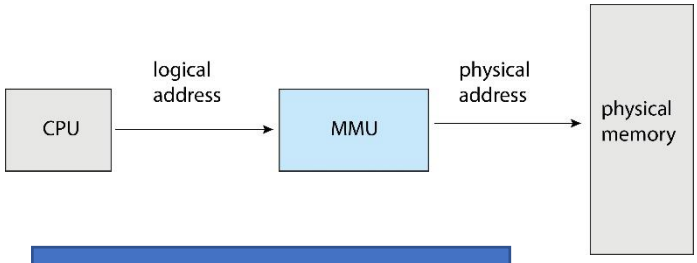
בתור תהליך נקבע רק כתובות יחסיות, היות וזה לא משנה לתהליך איזה Base או Base + Limit נקבל, אלא רק איפה אנחנו ביחס ל-Base – Offset, כי שם אנחנו נשמור/נכתוב מהזכרון.
ולכן כאשר אנחנו **נטען** את התהליך (ע"י מערכת ההפעלה), הוא יקבל Base (ע"י מערכת ההפעלה) ואז בכל פעם שהוא רוצה לגשת לזכרון הוא יבצע פעולת חיבור (את כתובת הBase לOffSet שלו).



מתי אפשר לעשות Address Binding? באחד מהמצבים הבאים:

מה קבלנו? שהתהליך שלנו מאוד מכיר את הכתובות הפיזיות שלנו, בגלל שעשינו בינדינג ועכשיו התהליך מאוד תלוי לכתובות ספציפיות, כך שהוא לא יכול לצאת מהגבול שקיבל ולכן אם יצטרך **לגדול** הוא **בבעיה**, כי לא ניתן לבצע הזזה לתהליכים אחרים/לתהליך שלנו היות והם גם כן עשו Binding ולכן גם זו **לא** השיטה.

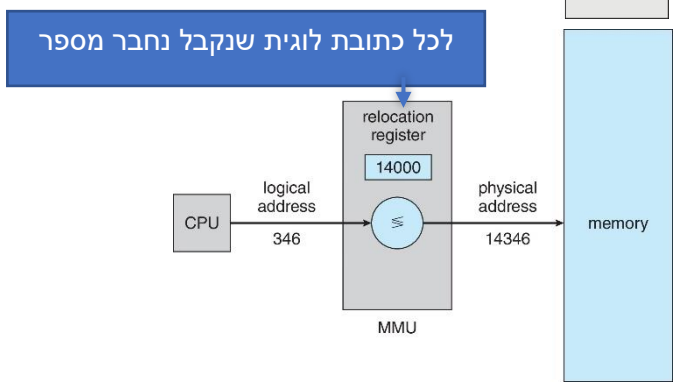
נרצה למצוא מודל אחר שמאפשר לתהליך לגדול מעבר לגבול שקיבל ולכן נעבור למודל הבא.
נרצה להפריד בין התהליך לכתובות הפיזיות, כי אם נפריד בין כתובות פיזיות לכתובות לוגיות אז זה **יתן לנו את היכולות לעשות משחק דינמאים בזמן ריצה בהתאם לדרישה.**
כתוביות פיזיות – הכתובות שיושבות באמת בזכרון.
כתובית לוגית – לא מה שקורה בערוצים הפייזים, אלא מה שהתוכנית רואה. נוכל לתת כתוביות לוגיות ולמפות אותם לכתובות פיזיות.
ואז נוכל לפתור את הבעיה שצוינה קודם.

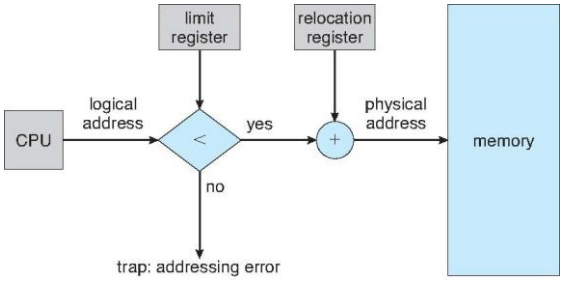


ולכן נסתכל על **המודל** הבא:
המעבד מריץ תהליך שמכיר כתוביות לוגיות שנעזר בMMU שממפה אותם לכתוביות פזיות ולכן ה-MMU ידאג לנהל את המיפוי של כתובות לוגיות מול כתובות פיזיות (בעזרת מערכת ההפעלה)

- הקומפיילר קובע את הכתובת הלוגית.

ולכן מנגנון הבקרה יראה כך:

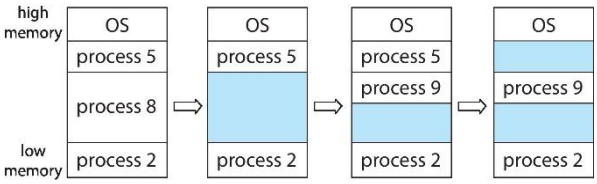




- כשהתהליך פונה לכתובות הוא נגיד פונה לכתובת 92 אז הוא ימופה לכתובת הפיזית Base + 92 כאשר ה-(+) הוא לפי המנגנון שצויין למעלה. (הכתובת הכי נמוכה ממופה לבייס והכי גבוהה לבייס + לימיט)
  התהליך אפילו לא יודע שיש לו בייס – זה השוני מהמודל הקודם.

נשים לי כי המודל החדשה שלנו טוב יותר, היות וכבר אין לנו Base אלא יש רק Offset ולכן אם יש תהליך שנגמר לו הזכרון והוא רוצה לגדול כלפי מעלה, נוכל להעתיק אותו לקטע אחר בזכרון, שהוא **גדול** יותר.

<u>מחיצות משתנות</u>



תהליכים תופסים מקומות בגדלים שונים בזכרון.
חור – בלוק של זיכרון זמין, חורים בגדולים שונים מפוזרים בזיכרון.
כאשר תהליך מגיע מוקצה לו זיכרון מהחור שמספיק גדול כדי לתמוך בו. תהליך שיוצא משחרר את החלק שלו.
לפי המודל הזה הזיכרון **חייב להיות רציף**.

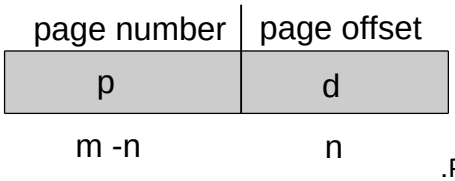
כשנכנס תהליך, נרצה לגבש אחת משלושת האסטרטגיות הבאות:
First Fit – להקצות של החור הראשון שמספיק גדול O(1)
Best-Fit – להקצות את החור הקטן ביותר שהוא מספיק גדול בשביל אותו תהליך. O(n)
Worst – Fit – למצוא את האזור הכי גדול ונשים אותו שם. O(1)

<u>Fragmentation</u>:
External Fragmentation – יש זכרון להקצות לתהליך, הבעיה שהוא לא רציף.
Internal Fragmentation – הזיכרון שמוקצה לתהליך הוא גדול יותר ממה שהוא צריך.

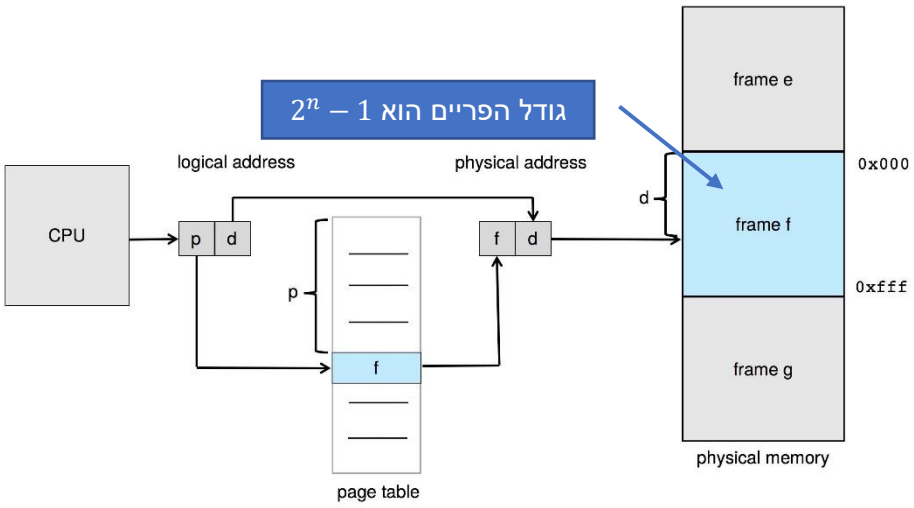
ניתן לפתור את ה-External Fragmentation ע"י **Compaction** (דחיסה), כלומר ברגע שיש תהליך שיש מספיק מקום אך לא רציף אז נדחוס את הזכרון ע"י העברה של מרחבי זכרון של תהליכים.

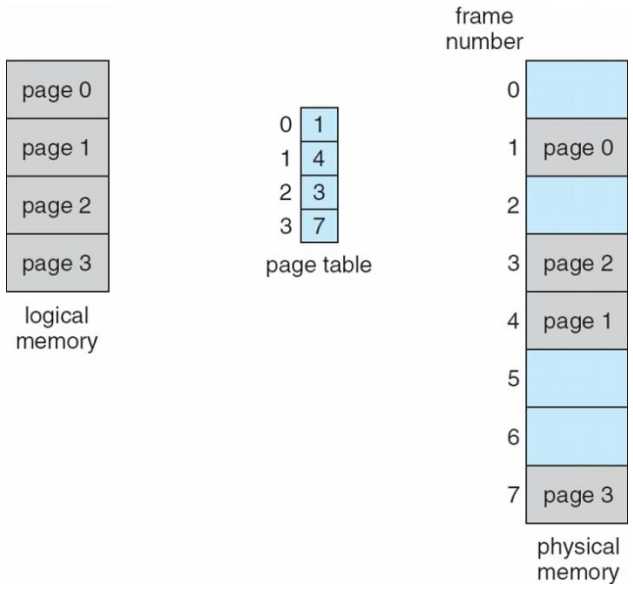
נעבור מהמודל שהזכרון הפיזי רציף למגנון יותר דינאמי שהוא **אינו רציף.**

<u>Paging</u>

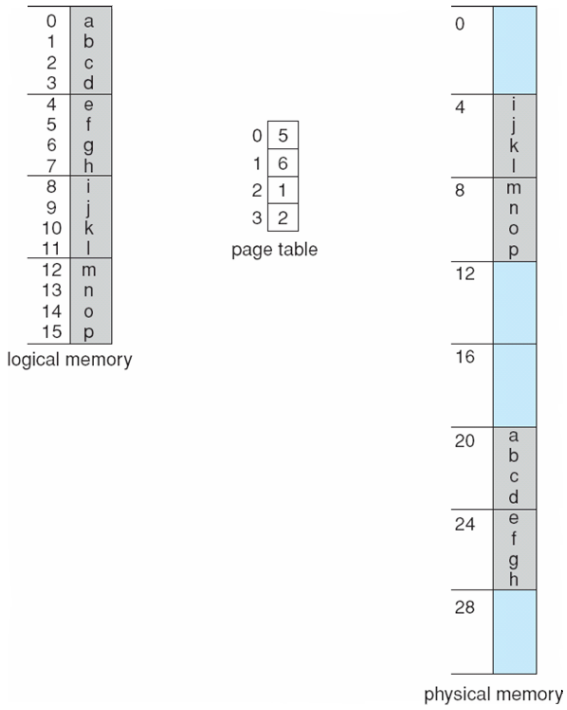


במקום זכרון אחד רציף, נייצר מיפוי שמבוסס על מקטעים בגודל קבוע.
נסתכל על מקטעים בגודל קבוע.
נקח את כל ה- RAM שלנו ונחלק את כל הזכרון שלנו למקטעים בגודל 4K (לדוגמא 2^{12})
ולכל תהליך נשמור טבלת מיפוי (Page ל-Frame) שנקראת Page Table.
Page Number – משומש כאינדקס ב-Page Table.
Page Offset – משולש עם כתובת הבסיס כדי להגדיר את כתובת הזיכרון הפיזית שנשלחת ליחדת הזיכרון.
החומרה תומכת (וקובעת) בגדלים קבועים של פריימים שהם בגודל 2^n, לכן עבור $n = 12$ נקבל תמיכה ב-*4K*
תהליך המיפוי נראה כך:





ולכן נקבל זכרון לוגי שהוא **רציף**.
וזכרון פיזי שהוא בכלל **לא רציף** – כי נגיד מגיע תהליך חדש אשר מבקש הקצאה של 3 דפים (12K)
אז נוכל לבנות טבלת דפים חדשה (דף 0, דף 1, דף 2)
ונמפה אותם לפריים 6,5,2 (לכל תהליך Page Table
משל עצמו – כלומר הLogical Memory)



דוגמא:
נניח שקבלנו כתובת בגודל 4 ביט
כש-2 ביט מחולק לOffset
נסתכל על האות k אשר נמצאת בכתובת 10

$$10 = 1010_2$$

ולכן נחלק ל-2 ביט שהם offset ו-2 ביטים לpage number

$$\underbrace{10}_{2_{10}}\ \overbrace{10}^{2_{10}}$$

ולכן ניגש לתא 2 שממופה לפריים 1, ולכן נלך לפריים 1 עם Offset 2 שזה בדיוק *k*.

מה גודל הframe שטוב לנו?
פריימים גדולים מדי יצרו יותר מדי קיטועים, עבור פריימים קטנים נצטרך להגדיל את גודל טבלת הדפים כי מספר הכניסות הפוטנציאלי הולך וגדל.
במערכת 32 ביטים החלוקה זה [12 | 20] עבור [$Page\ Number \mid Page\ Offset$]
מה גודל טבלת הדפים – Page Table ? $2^{m-n} \cdot 4$ כש-4 זה גודל הכניסה.

נפתרנו מבעיית ה-External Fragmentation כי הורדנו את דרישת הרציפות, כי בכל פעם שיש לנו מספיק Frame פנויים אז נוכל לתת לתהליך את הזכרון, הבעיה שנוצרת זה המקום בתוך הדפים עצמם – Internal Fragmentation.
לדוגמא:

Page size = 2,048 bytes
Process size = 72,766 bytes
35 pages + 1,086 bytes

Internal fragmentation of 2,048 - 1,086 = 962 bytes

ה-**Internal Fragmentation** יהיה בממוצע $\frac{(1+Page\ Size)}{2}$

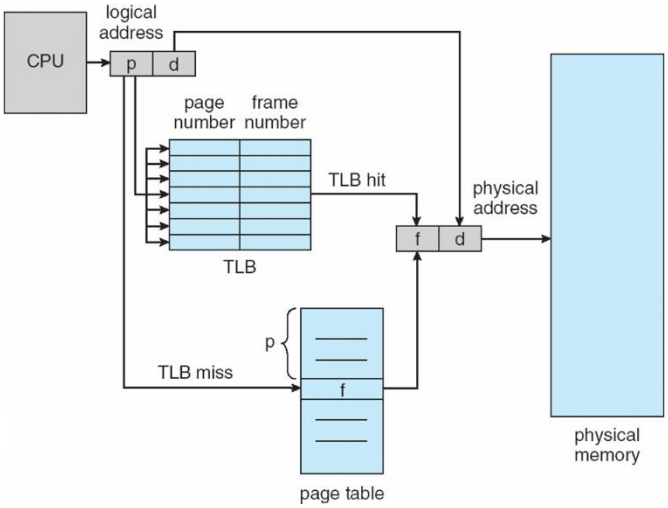
- כל פעם שנבצע Malloc(1) נקבל Page בגודל 2^n

ה-Page Table הוא משהו מאוד גדול ולכן הוא ישמר בזכרון RAM ולא במעבד, הגישה לזכרון היא פי 100 מהגישה למעבד ולכן אם נרצה לגשת לזכרון נצטרך לבצע 2 גישות, אחת ל-Page Table ואחת ל-Physical Memory. ולכן במקום לגשת ישר לזכרון ב100ns קבלנו גישה ב200ns.

ולכן על מנת להוריד גישות נשתמש במגנון Caching שנקרא **TLB** – **מנגנון** שעושה Caching ל- Page Table.

Trade-off: חוסך הזזות – אבל גישה לזכרון פי 2 זמן.

המנגנון נראה כך: ←



ה-TLB יושב בתוך המעבד, אבל סילקון בתוך המעבד זה יקר ולכן נכניס משהו יותר מצומצם ולכן אנחנו בונים על עקרון הלוקליות, שאומר שאם פנינו לדף ספציפי אז כנראה שנבקש אותו שוב, ככל שה-TLB יותר **קטן** נקבל יותר **התנגשיות**.

ניתן להוסיף סיכוים על מנת להגדיל את הסיכוים לHit Ratio- כמו הגדלת הדפים שנשמרים בTLB (N-Way) או לחלופין להגדיל את מספר הכתובות הפונטציאות שאפשר לחפש עליהן.

Effective Access Time ("שאלה שתמיד מופיעה במבחן" – מוטי גבע, הרצאה 11, 1:16:13).

Hit Ratio - ההסתברות שדף מסויים נמצא ב-TLB

דוגמא:

נסתכל על מכונה עם Hit Ratio 80% ו-10ns זמן גישה לזכרון. נקבל כי

EAT = $0.8 \cdot 10 + 0.2 \cdot 20 = 12\ ns$ (תוחלת)

נרצה לקבל במערכות אמיתיות 10.1 (99%)

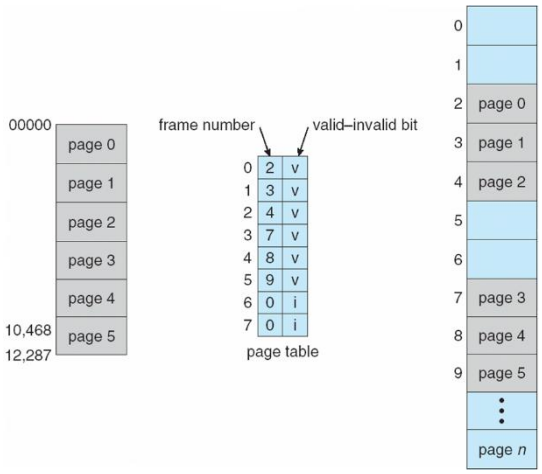
הרצאה 12

אם עד עכשיו מיפינו דף לפריים, השלב הבא יהיה להוסיף לכל דף גם ביט. ביט שאחראי להגיד אם מותר לגשת לדף או אסור (Valid / Invalid)

אם נבוא למפות את דף 3, נלך לטלה שלו ונראה שהוא ממופה לפריים מספר 7 שהוא Valid ולכן הכל בסדר. אם נמפה את דף 6 נקבל Invalid ולכן לא ניתן לגשת אליו כי הוא לא נמצא בזיכרון ואז הוא יזרוק Exception (Page Fault – Interrupt) ותעיר את מערכת ההפעלה.

החומרה היא זו שעושה את הניפוי, מערכת ההפעלה ממלאת את הטבלה.

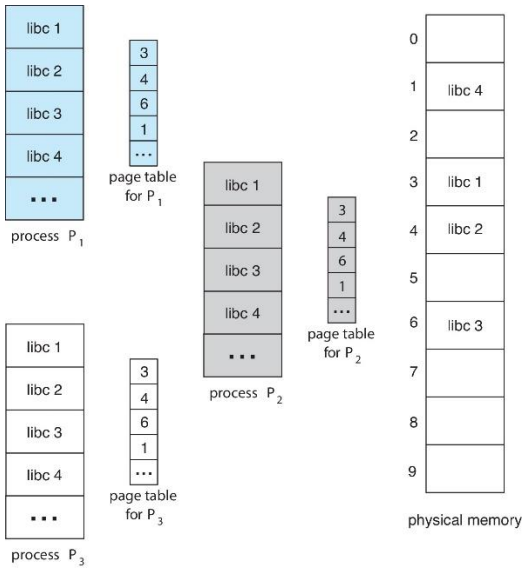
כך זה נראה:



טבלת המיפוי תכיל 2 דגלים, אם ניתן לגשת או לא ניתן לגשת.

Shared Pages – דפים משותפים

קוד משותף – העתק אחד שלא קוד שהוא לקריאה בלבד שמשותף בין תהליכים. דומה לכמה טרדים שחולקים את אותו מרחב תהליך. אפשר לעשות את זה גם על דפים שמכילים מידע, בין תהליכים נצטרך לעשות זאת בצורה **מפורשת** (תהליך לא ישתף עם אחר אלא אם כן הוא ביקש במפורש)
Libc – ספרייה שכמעט כל תוכנית של C משתמשת בה. בגלל שיש הרבה תהליכים שמשתמשים בה, אז במקום לשכפל אותה כמה פעמיים, אז נייצר דף משותף לכמה תהליכים (קריאה בלבד).

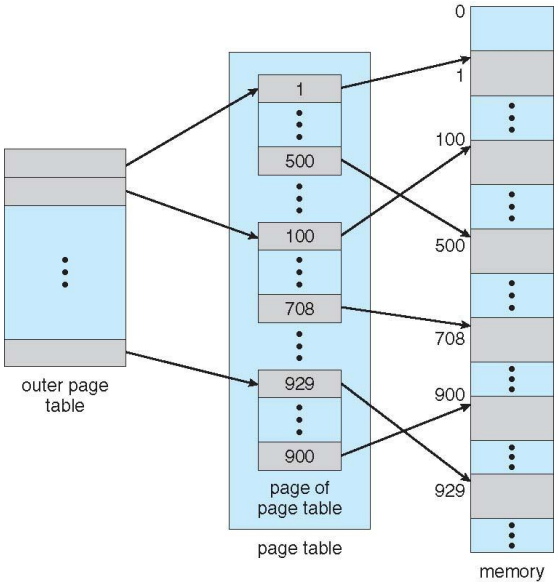


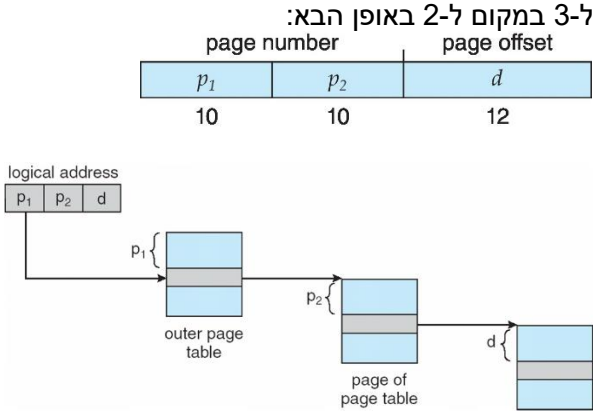
אם נרצה לשתף ביט אחד ל2 תהליכים אז בפועל נקבל כמה שהחומרה הגדירה את ה-Page.
מי קובע מה הרשאות? (נמצא ב-Page Table כמו שיש דגל Invalid ו-Valid אז גם דגל של Read,Write וכו')

נניח כי בארכיטקטורה מסויימת Page size מקבל 12 ביטים ו- # Of Pages מקבל 20 ביטים, ולכן נקבל כי Page size = $2^{12} = 4k$
$\# \ of \ pages = 2^{20} = 1M$
ולכן נגיד ונרצה ליצור מערך של $int\ pt[1M]$ אזי $sizeof(pt) = 1M \cdot 4B = 4MB$ שזה כמות הזיכרון שיש לתת לכל תהליך, שזה זיכרון מאוד מאוד גדול ולכן נסתכל על **המודל** הבא.

היררכית טבלת הדפים

נוכל לחלק את טבלת הדפים למספר טבלאות דפים, וכך נייצר היררכיה.
נשבור את מרחב הכתובות הלוגי אל הרבה טבלאות דפים.
הטבלה החיצונית תצביע להרבה טבלאות דפים שהן יכילו את המיפוי לפריימים.
כך נוכל להחזיק ברגע נתון רק חלק מהטבלה בזיכרון ולא כולה.
טכניקה פשוטה היא Two-level page table (נכון לגביי N-Level Page Table).
אם בפעם הקודמת התייחסו ל-Page Number כ20 בתים, אז עכשיו נוכל להתייחס לחלק את הכתובת ל-3 במקום ל-2 באופן הבא:





מה קבלנו?

- גודל ה-Outer Page Table הינו $2^{10} = 1k \Rightarrow 1k \cdot 4b = 4kb$
- גודל כל Inner-Table הינו $2^{10} = 1k \Rightarrow 1k \cdot 4b = 4kb$

ולכן כדי להשלים מיפוי שלם נצטרך להחזיק לפחות $8kb$ (אחד Outer ואחד Inner)
קבלנו במקום שכל תהליך יחזיק $4mb$ אז נצטרך להחזיק $8kb$ **(הרבה פחות זיכרון, אבל הגישה לזיכרון עולה)**
אבל כעת נקבל שיש 3 גישות לזיכרון במקום 2.

outer page	inner page	offset
p_1	p_2	d
42	10	12

נקבל כי $0.8 \cdot 10 + 0.2 \cdot 3 \cdot 10$ = **EAT**
נוכל לחלק אפילו יותר באופן הבא:

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

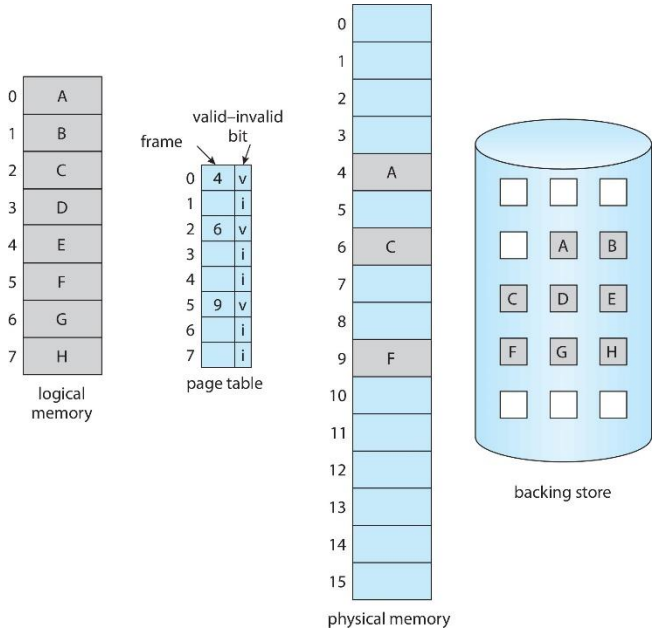
נקבל כי $0.8 \cdot 10 + 0.2 \cdot 4 \cdot 10$ = **EAT** במודל הבא:
(מ Outer2 ל-Outer לInner ל-Offset ל-Physical)

<u>Virtual Memory</u>

זיכרון וירטואלי זו טכניקה המאפשרת הרצת תהליכים שאינם נמצאים בשלמותם בזיכרון, ומה שנמצא בו לא נשמר באופן רציף.
היתרון המרכזי הוא שהזיכרון הלוגי יכול להיות גדול יותר מהזיכרון הפיזי.

יתרונות:

- תוכנית לא מוגבלת בכמות הזיכרון הפיזי הפנוי, משתמשים יוכלו לכתוב תוכניות למרחב כתובות לוגי גדול יותר שיקל על העבודה
- יותר תוכניות יוכלו לרוץ בו זמנית ויהיה דרוש פחות I/O לטעינה והחלפה של תוכניות משתמש.
- כל התוכניות ירוצו מהר יותר.

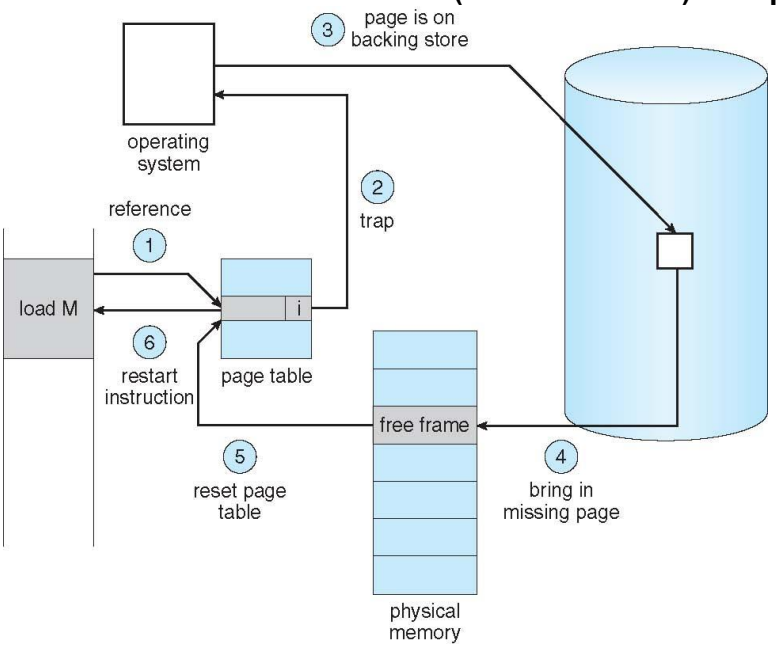


לדוגמא מגיע תהליך שפונה לכתובת ב- $Page = 1$, אבל מגיע המעבד ורואה שיש דף שהוא $Invalid$ ולכן הוא יעשה Interrupt ויעיר את מערכת ההפעלה, מערכת ההפעלה תלך ותבדוק אם הפנייה היא לגטימית או לא, אם לא אז נקבל $Seg\ Fault$
אחרת, נלך להביא תדף הזה. מאיפה? מה-Backing Store

Swapping

כשיש חומרה סופית ויש הרבה תהליכים שרצים, בנקודה מסויימת כל הזכרון מתמלא בתהליכים, ולכן כאשר מגיע תהליך חדש ומבקש הקצאה חדשה אנחנו בבעיה.
ישנן 2 אופציות, או להכשל ולא להקצות זכרון (פתרון פחות טוב), או לחלופין לפנות מקום בזיכרון ולהכניס פנימה דף חדש.

זה עובד באופן הבא: (לפי סדר המספרים) – במידה ויש Free Frame

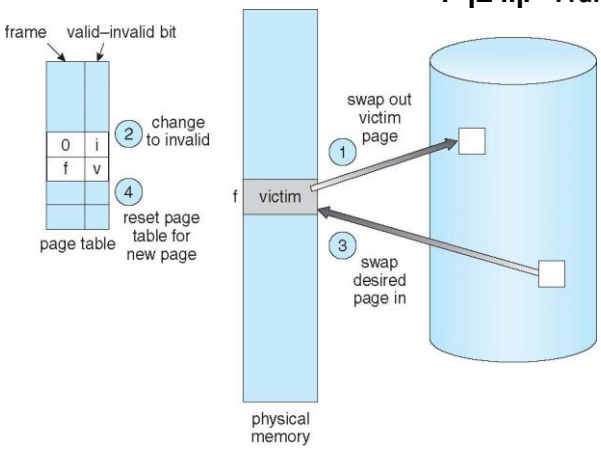


Backing Store – דיסק מהיר הגדול מספיק כדי להכיל עותקים של כל התמונות מהזיכרון לכל המשתמשים. חייב לספק גישה ישירה אל תמונות הזיכרון הללו.

במידה ואין Free Frame אז נעשה את הדבר הבא:

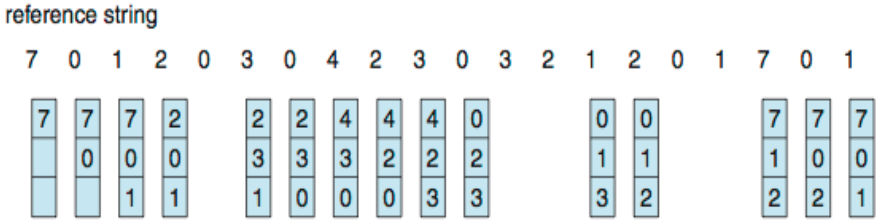
1. נמצא Frame "קורבן" – נקרא לו A
2. כותב את A לדיסק (Backing Storage)
3. הופך את A ל-Invalid
4. על אותו Frame נבצע את המגנון שלמעלה (נכתוב עליו את הFrame הנכון)

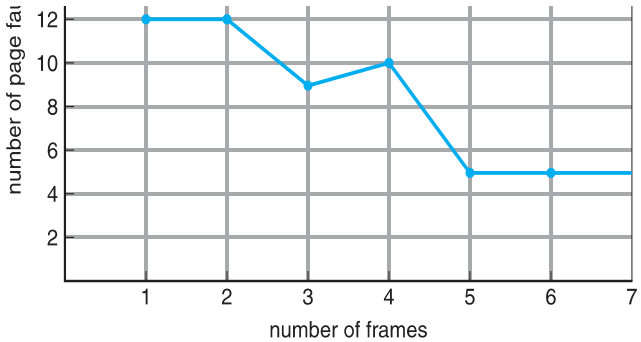
איך נבחר את אותו Frame "קורבן"?



אלגוריתם ראשון: FIFO

נסתכל על הרצף הבא: (15 Page Fault)



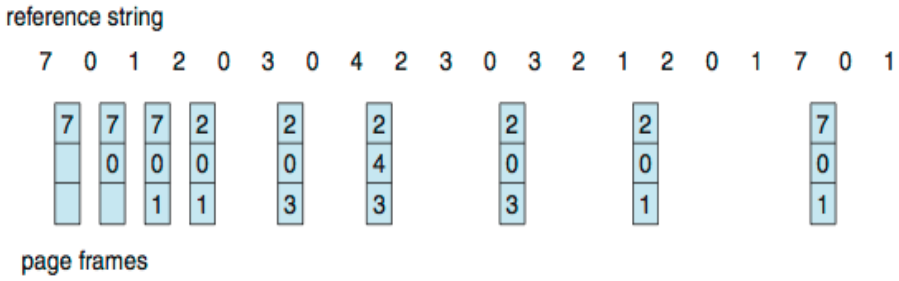


הנקודת עלייה ב-4 נקראת **Belady's Anomaly**, בעצם הוספנו זיכרון אבל קבלנו יותר Page Fault וזה קורה בגלל תזמון ובחירה לא טובה של האלגוריתם.

אלגוריתם שני: Optimal Algorithm (אלגוריתם תאורטי – אם היה אפשר לחזות קדימה)

במקום להסתכל אחורה הוא מסתכל קדימה (מי הכי רחוק)

נסתכל על הרצף הבא: (9 Page Fault)

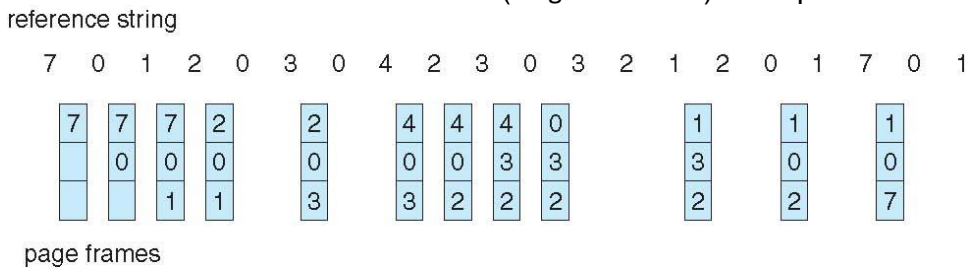


אלגוריתם שלישי: LRU (Least Recently Used (LRU) Algorithm)

מתבסס על עיקרון הלוקאליות.

תהליך בעדיפות נמוכה יוחלף כדי שיכנס תהליך בעדיפות גבוהה. (מחליפים את האחרון שהשתמשנו בו)

נסתכל על הרצף הבא: (12 Page Faults)



איך ממשים את LRU ?

בתוך ה-Page Table נשתמש ב-2 ביטים, אחד נקרא Access Bit והשני נקרא Modify Bit

כאשר Access Bit – אם משהו קרא מהדף הזה.

ו- Modify Bit – אם משהו כתב לדף הזה.

באופן כללי אפשר לקרוא ל-2 הביטים האלה Reference Bit.

Reference Bit מעודכן ע"י המעבד כאשר הוא קורא / כותב מאותו הדף.
Dirty Bit – שם אחר ל-Modify Bit. (משהו כתב לדף)

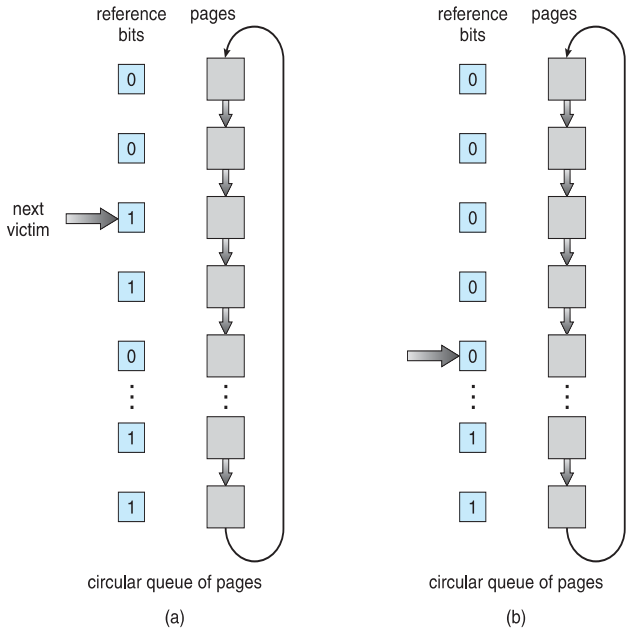
למה זה חשוב?

- אחד נגיד ה-Modify Bit=0 אז לא צריך לעשות Swap out כי הגירסא האחרונה שיש בדיסק היא מעודכן.

אז המימוש הוא כזה: להחזיק תור של דפים שנגענו בהם, כל פעם דוגמים את ה-Refernce Bit ואז מעדכנים ביחס לאותה רשימה מי הכי רחוק נגעו בו.

אלגוריתם רבעי: Second-Change (clock) Page-Replacement Algorithm – אלגוריתם קירוב

שומר רשימה מעגלית של דפים.
כאשר כל פעם שנכנס דף מכניסים אותו לסוף התור.
נחזיק פוינטר לקורבן הבא.
מגיע האלגוריתם – הופך Reference Bit 1 ל-0 ואז עובר להבא ברשימה, הראשון שהוא 0 הוא מוציא אותו.



חישוב EAT (חזרה ל-Swapping)

Page Fault Rate

נגדיר $0 \le p \le 1$ כאשר:
$p = 0$ אז **אין** $Page\ Fault$
$p = 1$ אז **כל** גישה היא $Page\ Fault$
ולכן נקבל כי:

$$\textbf{EAT} = (1-p)\cdot memory\ access + p\cdot(\textbf{page fault overhead}\ (nano\ ms) + \textbf{swap page out}\ (ms) + \textbf{swap page in}\ (ms))$$

$$milli = 10^{-3} > micro = 10^{-6} > nano = 10^{-9}$$

ולכן כאשר מתכננים מערכת אז ניתן להחליף מעבד, או לפעמיים זה לא הבעיה כי במערכת כזאת אם נשדרג רק את המעבד אז נקבל זמן עיבוד בין פקודות מאוד מהיר אבל כל פעם שיש $Page\ Fault$ ל-$Swap\ in, Swap\ out$ אז נאבד הרבה מאוד זמן, אז מה נעשה ? נוריד את ההסתברות של $Page\ Fault$ ע"י **הגדלת הזיכרון**.

דוגמא מספרית:

נניח כי $Memory\ acess\ time = 200\ nano\ seconds$
ו- $Average\ page - fault\ service\ time = 8\ milliseconds$
אז:

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

ולכן אם כל דף אחד מ1000 הוא $Page\ Fault$ אז נקבל כי :

$$\textbf{EAT} = 8.2\ micro\ seconds$$

ולכן האטנו את המערכת פי 40! (אם רוצים להקטין אפשר להגדיל שיהיה שגיאה בדף אחד מ-400,000)

לסיכום:

Base Limit Model → MMU → Paging, TLB → Hierarchical Page, Swapping Page

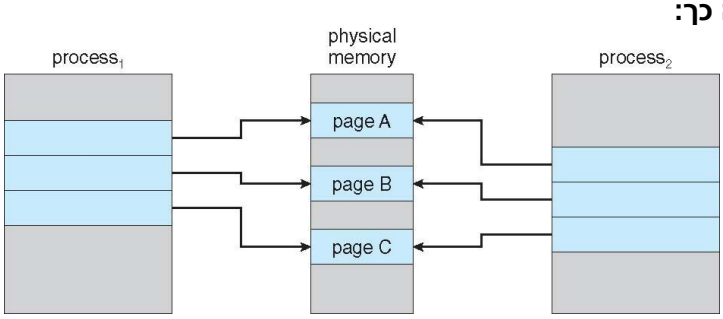
Copy-on-Write

כשעושים Fork אז בעצם מעתיקים את כל מרחב הזיכרון של תהליך מסיים לתהליך הבן, שזוהי בעצם פעולה יקרה ביותר.

נוכל לייעל את התהליך באופן הבא:

1. להפוך את כל הדפים ל-Read Only ולכן כל הדפים יכולים להיות משותפים ולכן תהליך האב ותהליך הבן חיים על אותו מרחב זיכרון

זה נראה כך:

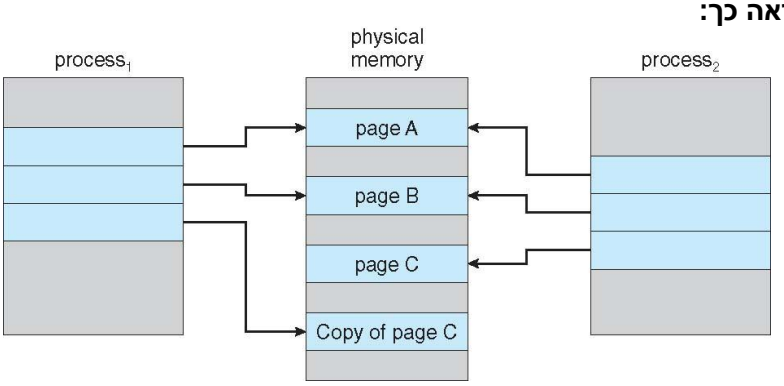


2. כאשר תהליך האב או תהליך הבן רוצים לכתוב לזיכרון אז הם צריכים לקבל עותק, כי אף אחד מהתהליכים לא יכול להשפיע על הזיכרון של התהליך השני, ולכן כאשר אחד מהתהליכים יבצע פעולת כתיבה אז ורק אז נבצע העתקה של הדף.

ולכן כאשר נרצה לכתוב לזכרון אז נקבל Page Fault כי ההרשאות הם רק קריאה, מערכת ההפעלה תתעורר ואז נבצע העתקה (Demand Paging)

Demand Paging – **(כשצריך)** מערכת ההפעלה מעתיקה דף מהדיסק לתוך ה-Physical Memory רק אם היה ניסיון גישה לדף הזה והדף לא נמצא בזיכרון (Page Fault קורה)

זה נראה כך:



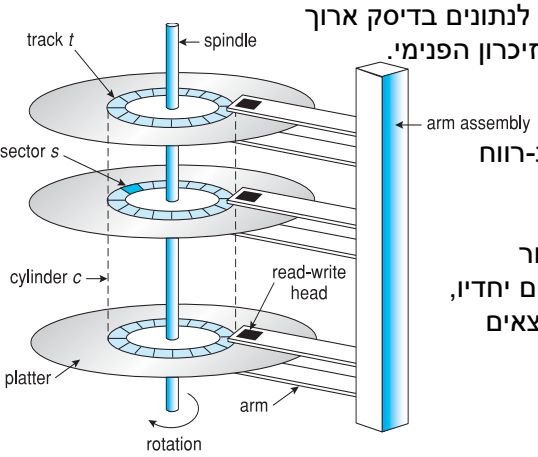
מה חסכנו? את כל ההעתקה. ולכן Fork יכולה להיות פעולה מאוד מהירה.

Vfork()– פעולה שעושה fork ואז נותנת לתהליך הבן לרוץ לפני תהליך האב מתוך שאיפה שהוא יבצע execv מאוד מהיר ואז במקום להעתיק את כל מרחב הזיכרון אז טענו מרחב זיכרון חדש במקום לבצע העתקה.

הרצאה 13 - מערכות קבצים

דיסק מגנטי:

דיסק מגנטי הוא רכיב במחשב המשמש לשמירת נתונים. דיסק קשיח יכול להכיל בדרך כלל כמות גדולה של נתונים לעומת זיכרונות אחרים, אך פעולתו איטית לעומת הזיכרון הפנימי של המחשב (RAM). הדיסק הקשיח הוא התקן זיכרון בלתי נדיף (צרוב) המאפשר אחסנה אמינה של נתונים דיגיטליים בנפח גדול ובזמן גישה קצר יחסית להתקני זיכרון מכניים אחרים. בהשוואה להתקני זיכרון אחרים באותה הקיבולת, הדיסק הקשיח זול משמעותית, אולם זמן הגישה לנתונים בדיסק ארוך בהשוואה לזיכרון הפנימי (RAM) - דיסק קשיח מכני איטי פי 100,000 מהזיכרון הפנימי.



הדיסק הקשיח מורכב **מדיסקות שטוחות** העשויות מאלומיניום או זכוכית, מצופות חומר מגנטי ומחוברות יחדיו לציר אחד. בין הדיסקות מצויה טבעת-רווח צרה החובקת את הציר ומטרתה לשמור על מרחק קבוע ביניהן.
לכל צד של דיסקה צמוד **ראש קריאה וכתיבה** המותקן על **זרוע**.
במהלך פעולת הדיסק, הדיסקות מסתובבות יחד ואילו הראש נע הלוך וחזור כדי לאפשר לו להגיע לכל נקודה ונקודה על פני הדיסקה. כלל הראשים נעים יחדיו, כך שכשראש אחד נמצא על גליל מסוים על גבי הדיסקה, כלל הראשים נמצאים על גליל זה בשאר הדיסקות.

הדיסק מסתובב במהירות קבועה ואנחנו נותנים זווית לראש קורא כותב, מה שבוחר מאיזה צילנדיר (גליל) הוא קורא.
הקריאה והכתיבה מתבצעת לזמן מסויים, כאשר הזמן זמן הקריאה/כתיבה הוא קבוע והוא מספיק בשביל לבצע את הפעולות.
יש 2 פעולות מכניות – הסיבוב והזזה של הראש קורא כותב, ולכן עדיף לקרוא תמיד ברצף מאותה צילנדר ו-track מאשר לגרום לו לקרוא מ-track-ים שונים (זו פעולה מכנית לעומת פעולה חשמלית)

- ברדיוס קטן הצפיפות של הביטים גדולה יותר וככל שההיקף גדול יותר אז הביטים מרווחים יותר.
- ניתן לקרוא או לכתוב לפחות בלוק (לא ניתן ביט בודד) כאשר גודל הבלוק הוא בערך $512B$.

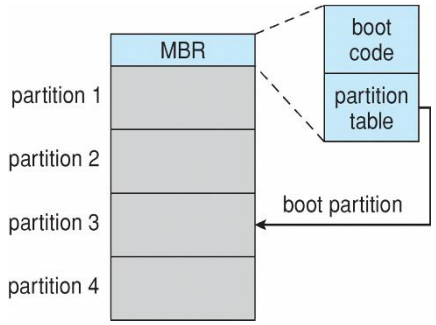
SSD – Nonvolatile Memory Devices (חשמלי – צרוב (לא נדיף))

מטרתם של זיכרונות אלו לשמש תחליף לדיסק קשיח (HDD). מהירותם נובעת מכך שהם, כאמור, אינם מכילים חלקים נעים ולכן זמן הגישה לנתונים קצר משמעותית. כמו כן, צריכת החשמל שלהם לרוב נמוכה יותר (למעט כוננ SSD בעלי ביצועים גבוהים במיוחד), משקלם קטן יותר, ופעולתם שקטה יותר - כל זאת בהיעדר חלקים נעים. בנוסף, תכונה זו מגדילה גם את אמינות הכונן. לעומת זאת, מחיר אחסון נתונים פר-מגה ביט גדול יותר, הקיבולת המקסימלית קטנה יותר, ובמקרים מסוימים אורך חיי ההתקן עשוי להיות קצר יותר משל דיסק קשיח.
אין שום דבר מכני, ולכן הוא הרבה יותר מהיר, אבל מוגבל מקריאה וכתיבה.

Tape - מאוד טוב אם רוצים לרוץ על הרבה מידע ברצף אבל אם נרצה לזוז בין לבין אז נצטרך פיזית להזיז.

Device Storage Management

במקום לקבל דיסק אחד ארוך אז אפשר לקחת דיסק ולחלק אותו למחיצות (Partition) כאשר כל אחד מהמחיצות יכול להכיל File-system משל עצמו.



Master Boot Record (**MBR**) – אזור על הדיסק שכאשר מדליקים את המחשב אז ה-BIOS יודע לקרוא אותו, לטעון אותו ולהריץ אותו.

מערכות קבצים

מערכת קבצים היא שיטה לאחסון וארגון קבצים במחשב על מנת להקל על הגישה אליהם. מערכות קבצים עשויות להשתמש בהתקן אחסון נתונים (כגון דיסק קשיח) ולנהל את המיקום הפיזי של הקבצים.

מערכת הקבצים תלויה במערכת ההפעלה, לעיתים קרובות ישנן מספר מערכות הפעלה התומכות באותה מערכת קבצים.

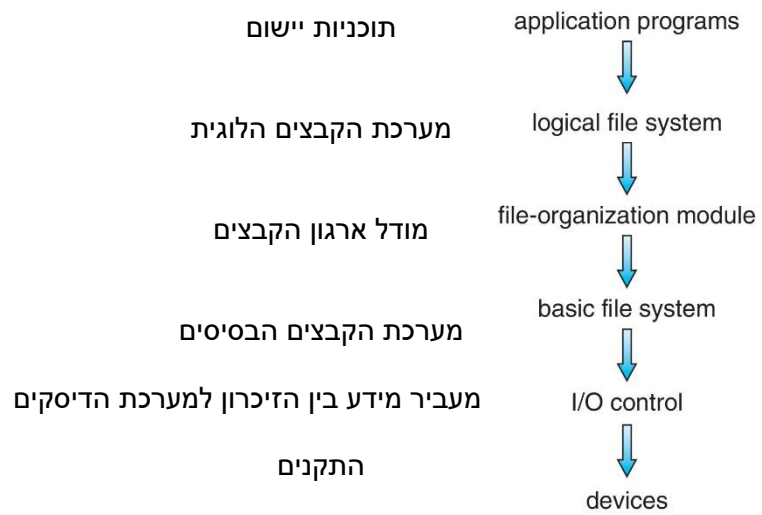
תפקיד של מערכת ניהול קבצים (File-system):
נותנת את האפשרות למשתמשים לעבוד עם קבצים (אנחנו לא יודע כלום איפה הקובץ נמצא וכו', אנחנו סה"כ נותנים מחרוזת ו- הרשאות קריאה/כתיבה וכו' ומערכת ניהול הקבצים מספקת זאת)
לספק למשתמש את האפשרות לקרוא ולכתוב קבצים.
לדעת לתרגם את הקלט לאיזה רכיב פיזי זה נמצא, כלומר על איזה בלוקים רלוונטים ולהנגיש את זה למשתמש.

<u>הערה:</u> על הדיסק נשמר אך ורק מידע בינארי.
שאומרים למערכת ניהול הקבצים לפתוח קובץ Text אז אני אומר לתוכנה שאני משתמש בה לפרש את השורות החדשות בצורה מסויימת (Ascii וכו')

מה מאפיין קבצי שהם **executable**? (יכול להיות שפת מכונה או Script)
1. שצריך לתת להם הרשאות של Execute
2. בד"כ יש להם Header למעלה שמתאר את הקובץ עצמו. כמו שיש בתמונה, לדוגמא JPG יש כותרת שמכילה את המידע על הקובץ, כמו ציר ה-X, ציר ה-Y, איזה דחיסה וכו', כאשר יש תוכנה שרוצה לפתוח את הקובץ היא קוראת את הכותרת וכך היא יודעת איך לפענח את הקובץ.

<u>Directory</u> (תקייה / Folder) – ספרייה שמנוהלת ישירות ע"י מערכת ניהול הקבצים ורק הן קוראות וכותבות אליה.
<u>File Attributes</u> – תכונות שיש לקבצים, מכיל את הפרטים הבאים:
(**שם**, **מזהה** (ID), **סוג** (Text,Binary), **מיקום** (איפה יושב על הדיסק), **גודל**, **הגנה** (מי יכול לקרוא/לכתוב), **חותמת זמן** תאריך ומזהה משתמש (מתי יצרו את הקובץ, מתי שינו וכו')) מי ששומר את המידע הזה זה בד"כ מערכת קבצים.
פעולות שאפשר לעשות על קבצים:
1. **לייצר**
2. **לכתוב**
3. **לקרוא** (ישות סדרתית, ממשיך לקרוא מאותה נקודה שהפסקנו, אז אם צריך לקרוא או לכתוב אז צריך לעשות פעולות חיפוש ואז לעשות Reposition לראש קורא כותב לתוך הקובץ)
4. **לחפש**
5. **למחוק** – מוחק את הקובץ עצמו.
6. **Truncate** – לקחת קובץ שהכיל מידע ולהפוך אותו לקובץ בגודל 0. (למחוק את התוכן אבל להשאיר את הקובץ)

הרצף של מערכת הקבצים: (כתוכנית עושים fopen() או fread() וכו', כך זה נראה מהצד של מערכות ההפעלה)



לכל קובץ נגדיר את בעל הקובץ (Owner) והקבוצה (Group) אליה הקובץ שייך.
כאשר ניגש לקובץ, מערכת ניהול הקבצים תבדוק האם הקובץ שייך ל-Owner, לאחר מכן ל-Group ואם לא אז ל-ACL. (משמאל לימין הבדיקה)

ACL – רשימת בקרת גישה היא רשימה של הרשאות המוצמדות לקובץ, תהליך תוכנית מחשב וכדומה. ה-ACL מגדיר איזה משתמש או מערכת מותרים בגישה אל הקובץ וכן לביצוע פעולות על הקובץ.
כל רשומת ACL קלאסית מגדירה את המשתמש ואת הפעולה הרלוונטית. לדוגמה ACL שמכילה בתוכה את הרשומה (Alice, delete) פירוש הדבר הוא שלמשתמש אליס (Alice), ישנה הרשאת מחיקה (delete) לקובץ.

File Control Block – FCB

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

File Control Block – FCB
מכיל Meta-Data (פרטים) על הקובץ.

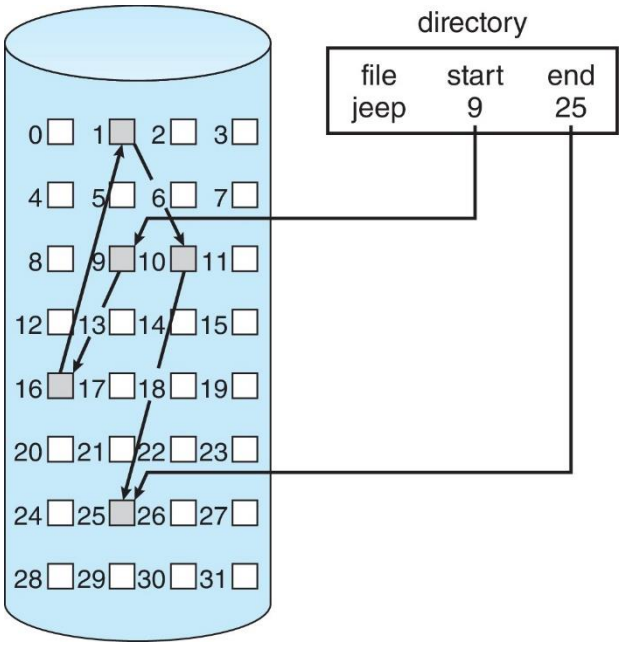
File Control Block (FCB)
לכל קובץ יש File Control Block אשר משמש את מערכת ניהול הקבצים (בשביל מ"ה).
הכותרת (Header) של הקובץ מופיע בשורות הראשונות של הלינק אשר שולח ה-

הערה: אין בעיה שיהיה כפילות בין FCB
הערה: חסר ב-FCB את השם של הקובץ. (בכוונה)

Linked Allocation - שיטת ההקצאה המגדירה כיצד הקבצים מאוחסנים בבלוקים של הדיסק

הרעיון המרכזי הינו:
1. ניצול יעיל של שטח הדיסק
2. גישה מהירה לבלוקים

יש בעצם directory שלכל קובץ ניתן לו שם ואז יהיה start ו-end, כאשר start מצביע על הבלוק הראשון, נוכל להתקדם בין בלוקים ע"י כך שיש **פוינטר** בכל תחילת הבלוק או סוף הבלוק.
תזכורת: תמיד קוראים או כתובים בכפולות של בלוקים



זה נראה כך:

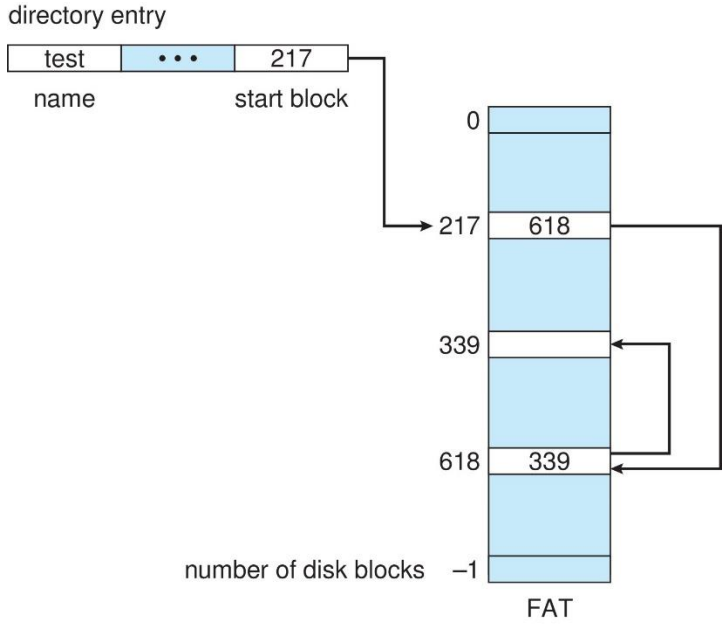
בשביל להגיע לבלוק החמישי צריך לעשות 5 קריאות, מהראשון לשני וכך הלאה.
תהליך חיפוש מכריח לקרוא את כל הבלוקים בדרך.
יתרון:
מאוד גמיש מבחינת גודל הקובץ, גודל הקובץ יכול לגדול בקלות בגלל המערכת אינה צריכה לחפש נתח רציף של זיכרון, בנוסף היא לא סובלת מפרגמנטציה חיצונית ולכן נקבל שהגישה טובה במונחים של ניצול הזיכרון
חסרון:
בגלל שהבלוקים מופצים באופן אקראי על הדיסק, מספר גדול של חיפושים צריכים להתבצע על מנת לקבל גישה לכל בלוק בנפרד. פעולה זו הופכת את ההקצאה המקושרת לאיטית יותר, ולא תומך בRandom Access.
בנוסף, בגלל שיש פוינטרים אז זה מצריך טיפה יותר overhead.

כמה עולה להגיע לאמצע הקובץ אם כל קובץ בגודל מסויים?
תשובה: כמספר הקריאות שיש לעשות בדרך.
כלומר, גישה לבלוק ה-k מצריך מעבר ב-k בלוקים הקודמים דרך הפונטרים.

אופטמצזיה: לשמור את הפוינטרים ב-Cache. (אלה שכבר עברנו)

File Allocation Table (FAT)

כל בלוק מיוצג כרשימה מקושרת של בלוקים.
Fat במערכת עם פוינטרים של 16 ביט ניתן להחזיק $2^{16} = 64K$ פוינטרים, ואם כל בלוק הוא $512B$
אז גודל המקסמלי של File System הוא $512B \cdot 64K = 256\ MB$
כל פעם שנגדיל בלוק נקבל לכל הפחות $4K$ (אי אפשר לתת קובץ קטן מבלוק)
נקבל כי $Size \leq Size\ on\ Disk$ (כי $Size\ On\ Disk$ הוא כפולה של $4K$)



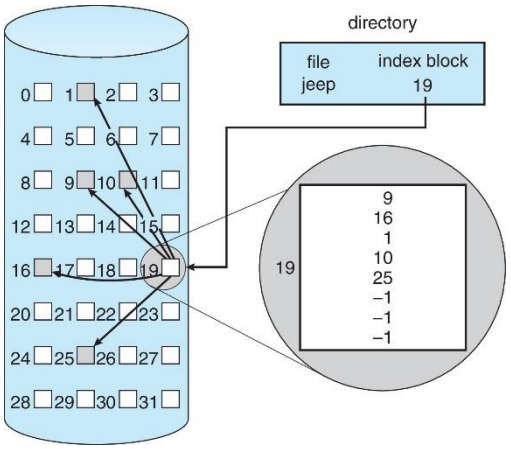
FAT32

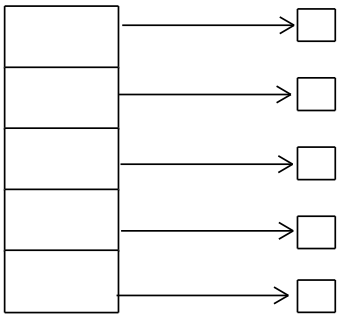
ההבדל בינה לבין ה-FAT הוא גודל המצביע.
במערכת FAT 32 אז גודל כל פוינטר הוא 32 ביט ולכן נקבל כי ניתן לקבל עד $2^{32} = 4G$ פוינטרים.
ולכן נקבל כי גודל מערכת הקבצים היא $4G \cdot (\sim 0.5K) = 2TB$
ובמערכת קבצים שכל בלוק הוא $4K$ אז נקבל $16TB$

אם נרצה להגיע לאמצע הקובץ אז צריך לעבור על הבלוקים שלפני ולקרוא את הכל, זה לא כל כך יעיל, ולכן נוכל לפתור זאת בעזרת המודל הבא:

Indexed Allocation

חלק ממה שאנחנו שומרים ב-FCB יהיה Index Table שזה בעצם טבלה שמצביעה על הבלוק של הקובץ, כך זה נראה:
הבעיה שזה שומר הרבה מידע על הקובץ.





יתרונות:

- תומך בגישה ישירה לבלוקים ולכן מספק גישה מהירה לבלוקים (רוצים את האמצעי? הולכים ב$O(1)$ לקובץ האמצעי)
- מתגבר על בעיית ה-External Fragmentation

חסרונות:

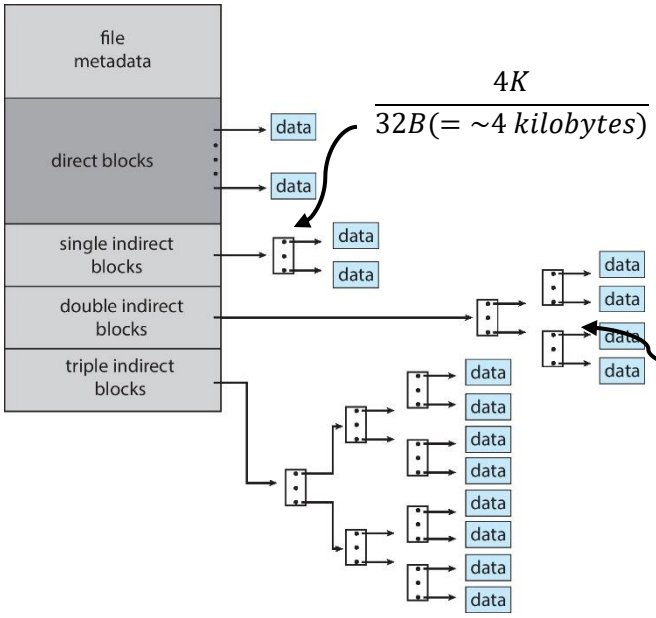
- המצביע גורם ל-Overhead הוא גדול יותר מה-Linked Allocation
- עבור קבצים קטנים אשר מתפרסים על 2-3 בלוקים, הIndexted Allocation ישמור בלוק אחד שיהיה Index Block על מנת להצביע, שזה לא יעיל מבחינת ניצול זיכרון
בעוד שב-Linked Allocation אנחנו מפספסים זיכרון של פוינטר אחד כל בלוק.

בנוסף, **אם** שומרים את ה-Index Table בFCB אז זה אומר שמספר הכניסות שלנו מוגבל כי עבור כל קובץ שומרים מספר קבוע עבור FCB (זה לא משתנה) ולכן המקום שיש לנו לשמור את האינדקס בתוך הFCB הוא מצומצם.
בממוצע יושבים בתוך FCB אינדקס של 10 כניסות, ואם כל אחד מצביע על $4K$ אז נקבל כי קובץ שהגודל המקסמלי שלו הוא $10 \cdot 4K = 40K$ (לא מספיק, אבל מצד שני נגמר לנו המקום ב-FCB)
אז הפתרון לבעיה הוא :

Combined Scheme: UNIX UFS

סדר הגודל של ה-10 הראשונים ישבו בFCB
אם יש יותר מ10 בלוקים אז מצביעים על בלוק שהוא בלוק של פוינטרים.
ולכן אם יש $4K$ וכל פוינטר הוא מסדר גודל של $32B$ אז
נקבל כי כמות הפוינטרים שיש בבלוק הוא

$$\frac{4K}{32B(=\sim 4\ kilobytes)} = \frac{4K}{4\ kilobytes} = 1\ kilobytes = 1000$$



ולכן גודל הקובץ הוא $1K \cdot 4K = 4M$
שזה לא מספיק בשביל קובץ PDF

ולכן נעבור ל-Double Indirect Blocks
כי נקבל שכמות הפוינטרים ב-Double Indirect Block
בכל אחד מהריבועים הלבנים עם הנקודות זה 1000
(חשבנו קודם)
ולכן נקבל $1000 \cdot 4M = 4GB$

ולכן אם צריך עוד אפשר לעבור ל-Triple Indirect Block.

החיפוש יחסית מהיר. כי אם נרצה בקובץ של $4GB$
אפשר לדעת באיזה בלוק בדיוק אנחנו צריכים.

סוף ההרצאה:

i-node : מבנה נתונים המתאר קובץ במערכת הקבצים המקומית
קבצים בגודל 0 – אזור ב-I-Node שנשאר חופשי ואז אם קובץ הוא קטן מאותם מספרים חופשיים אז ניתן להכניס את תחילת המידע שלו בתוך ה-FCB עצמו.