

# מערכות הפעלה

"מחשב ללא מערכת הפעלה הוא כמו מכוניות ללא מנוע."

-ביל גייטס

## תוכן עניינים

4 .....	הרצאה 1 – מבוא
4 .....	מבנה המחשב
5 .....	מבנה מערכת מחשב
8 .....	הרצאה 2
8 .....	ממשק משתמש
8 .....	פונקציונליות שעוזרות למשתמש
9 .....	System Calls
9 .....	Virtual Machine
11 .....	הרצאה 4
11 .....	שיטות ל-scheduling
12 .....	חיזוי על פי תוצאות עבר
12 .....	Exponential smoothing
12 .....	Priority scheduling
12 .....	Priority Inversion
14 .....	הרצאה 5
16 .....	הרצאה 7 – Deadlock
16 .....	Deadlock
17 .....	התמודדות עם deadlock
17 .....	Safe state
17 .....	Algorithm Banker
18 .....	הרצאה 6
19 .....	Paging
20 .....	הרצאה 8 – זיכרון וירטואלי
20 .....	Demand Paging
21 .....	Page Fault
21 .....	Copy-On-Write
21 .....	Page Replacement
22 .....	Frame Allocation
24 .....	File System Interface – 10
24 .....	Open file locking
24 .....	Access Methods
25 .....	Partitions
25 .....	מבנה הקבצים במערכת
26 .....	Protection
27 .....	הרצאה 11 – ניהול זיכרון דיסק
27 .....	הקראת זיכרון לדיסק
27 .....	Disk scheduling
28 .....	Threads – 12
28 .....	User/Kernel Threads



# הרצאה 1 – מבוא

## מבנה המחשב

המחשב מבוסס על מודל Von Neumann. הארכיטקטורה מבוססת על 4 חלקים:

ALU – Arithmetic/Logical Unit	.1
CU – Control Unit	.2
Memory Unit	.3
I/O – Input/Output devices	.4

## Memory Unit

ניתן לבצע שתי פעולות מרכזיות:

- Fetch – מקבל ומעתיק מידע מהזיכרון. קריאה בלבד.
- Store – מאחסן מידע בזיכרון. יכול לדחוס מידע קיים.

## I/O

הגישה למכשירי קלט/פלט היא איטית בהרבה, אך חשובה כדי לקבל מידע מהסבירה ומהמשתנה.

## ALU

ה-ALU מיועד לבצע פעולות מתמטיות ולוגיות. ה-ALU מורכב מ:

- מעגלים לביצוע פעולות אריתמטיות/לוגיות.
- רגיסטרים לאחסון תוצאות חישוב ביניים.
- אפיק (Bus) שמחבר בין השנאים.

## Control Unit

התכנית שלנו מאוחסנת בזיכרון כפקודות בשפת מוכנה. תפקידיה של יחידת הבקרה הוא להריץ את התכנית על ידי חרזה על השלבים:

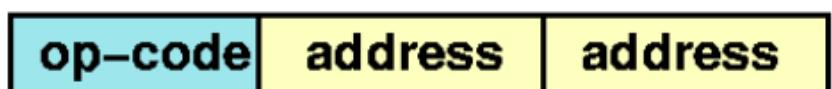
- Fetch – שילפת הפקודה הבאה לביצוע מידע.
- Decode – קביעת הפעולה שיש לבצע.
- Execute – ביצוע הפעולה על ידי שליחת אותן ל-ALU, לזכרון ולתתי הממערכות קלט/פלט.
- משיך עד לפועלות HALT (עצירה).

פקודות בשפת מוכנה:

## מבנה פקודה

פקודה מורכבת מ:

- קוד פעולה: מצין איזו פעולה יש לבצע.
- שדות כתובות: מצינים את כתובות הזיכרון של הרכיבים שעלייהם פועלת הפעולה.



## מבנה CU

ה-CU בנוי מ:

- שומר את הכתובת של הפקודה הבאה. Program Counter (PC)
- שומר את הפקודה שנשלפה מהזיכרון. Instruction Register (IR)

## **מבנה מערכת מחשב**

ניתן לחלק את מערכת המחשב לאربعة מרכיבים :

- .1. אפליקציות
- .2. מערכת הפעלה
- .3. חומרה
- .4. משתמש

### הגדרת מערכת הפעלה

מערכת הפעלה היא **תכנית** שחוצצת בין המשתמש של מערכת המחשב והחומרה.

מערכת הפעלה יכולה להימצא בכל מקום, החל מחשבים ופלאונים ועד מכוניות ובסביבות ענן. המטרות של מערכת הפעלה הן הרצת תכניות המשמש بصورة קלה, הפיכת השימוש במחשב לנוח יותר, ושימוש יעיל יותר בחומרת המחשב.

לכל מערכת הפעלה מטרות שונות, ובאופן כללי כל מערכת הפעלה בוחרת בין ייעילות לנוחות.

**מטרה מרכזית**: חלוקת משאבים, והחלה מה לעשויות במצב של קונפליקט, על מנת שיהיה שימוש הון ויעיל במשאבים. כמו כן מערכת הפעלה אחראית על ההרצה של התוכנות השונות.

**הגדרה**: החלק במערכת הפעלה שרצ כל הזמן, ללא הפסקה, נקרא **kernel**. כל שאר התהליכים מקוטלים ל- Application Program System Programs

### הפעלת המחשב

בהדלקת המחשב עולה bootstrap program, שמטרתו לטען את מערכת הפעלה. זהו הקוד הראשון שרצ עם הדלקת המחשב.

### Device Controller

כל Device Controller אחראי על סוג מסוימים של מכשיר.

המידע מ/אל המכשיר מנוהל באמצעות local buffer.

### מעבדים

ברוב המערכות יש מעבד אחד או יותר מסווג general purpose.

בנוסף יש מעבדים מסווג special purpose – המותאמים לפעולות ספציפיות. למשל DSP או GPU.

### Multiprocessors

מערכת עם שני מעבדים וייתר החולקים Bus ולפערמים גם clock ו זיכרון.

ישנם שני סוגי של מערכות : Multiprocessors

1. **א-סימטריות** : מיפויות מסוימות מוקצחות למעבדים מסוימים בלבד. ייתכן למשל שמעבד יחיד יהיה אחראי על כל פעולות הקלט/פלט וכו'.

2. **סימטרי** : מתייחס לכל רכיבי העבודה במערכת באופן זהה.

### Clustered Systems

כמה מערכות שונות שעבודות ביחד.

- ב-DYC חולקים זיכרות דרך אחסון בשם SAN.
- מספק שירותים עם פניות גבואה ששורדת כשלונות.
- Asymmetric Clustering – מכונה אחת במצב המתנה חם.
- Symmetric Clustering – כמה קודקודים שרצים על אפליקציות ומכוונים אחד את השני.
- חלק מה-clusters מכונים ליעילות גבוהה, כאשר אפליקציות בהם חייבות היכתב עם שימוש במקובל.

## פסיקה

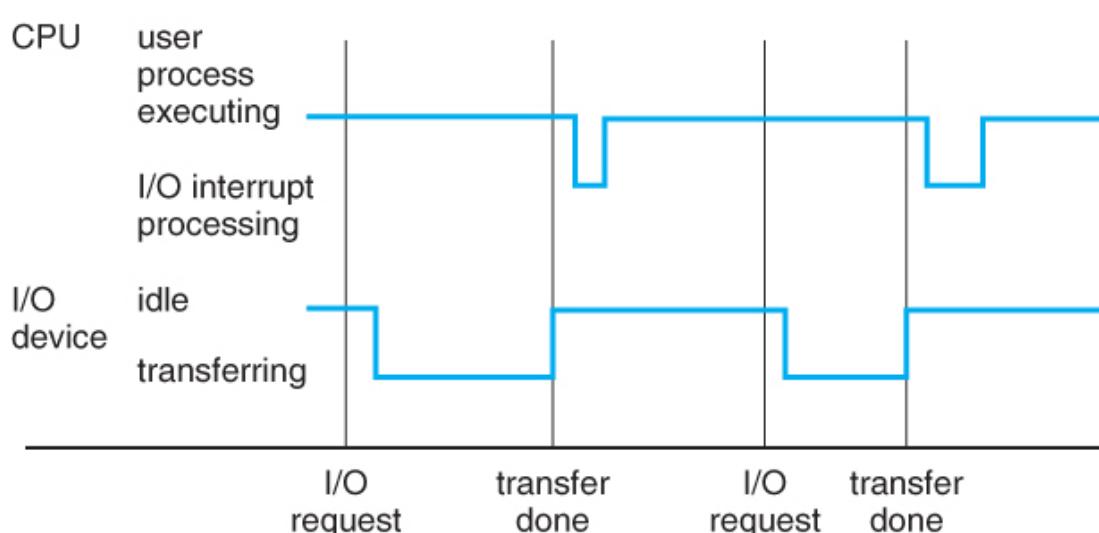
פסיקה (Interrupt) היא אוטומטת המתקבל במעבד מרכיב חומרה. הפעלה נוצרת על ידי מכשיר, והוא מעבירה את השיליטה במעבד ל-interrupt service routine. בעצם המעבד עוזב הכל ושומר את כתובות ה-Memory ושל כתובות הרגיסטרים. מכשיר -> בקר פסיקות -> מעבד -> ליבת -> זיכרון. משם נחזור לlibah ונוריץ את הקוד ששמרנו בזיכרון על מנת לטפל בפסיקה.

## Trap

Trap הוא כמו פסיקה, רק שייצרה אותו תוכנה ולא מכשיר. Trap נגרם על ידי שגיאה או בעקבות בקשת של משתמש. תחילה יש לקבועאיזה סוג של interrupt קרה (תזוזה של העכבר, מקלדת, וכו'). לאחר מכן ישן שתי דרכי לטפל בפסיקה:

1. Polling – נעבור על סוגי interrupts האפשריים ונבצע את הקוד שנרצה.
2. Vector – נשמר מה נctruck לעשות עבור כל סוג של interrupt. מהיר יותר.

טיפול בפסיקות שונות:



## בקשות O/I

ישן שתי דרכי לטפל בבקשות O/I: סינכרוני וא-סינכרוני.  
א-סינכרוני: רק כאשר תסתיימים הפעולה של ה-O/I, נשלח את הפסיקה.  
סינכרוני: נשלח בתחלת הפעולה, והשליטה חוזרת רק כאשר תסתיימים.  
אם יש כמה בקשות מאותו המכשיר, נשרשר את הבקשות לפי סדר בטוח.

## ניהול זיכרון

עובד בחילטה מה יהיה בזיכרון בכל רגע. מערכת הפעלה אחראית על מעקב ורישום החלקים בזיכרון שנמצאים כרגע בשימוש, החלטה על העברת תוכן זיכרון המשוויה לתהליכיים מאל הזיכרון.

זיכרון משני (אחסון)  
Disk Cache בנוסף לזיכרון ה-RAM. משמש לשימורה של זיכרון רב/זיכרון שרוצים לשמור לאורך זמן. המהירות בהקשר זה תלויות בדיסק.

כל שעוברים לזכרוון קרוב, כך הוא גם מהיר יותר (אך גם נדיף יותר).

### Caching

העברת מידע מאחסון מסוים לאחסון מהיר יותר. תמיד נבדוק קודם את האחסון המהיר ביותר, על מנת לבדוק האם המידע נמצא שם. אם לא, נמשיך לחפש בהיררכיות הזיכרוון עד שנמצא אותו. לבסוף שנמצא אותו המידע יועתק ל-cache וישמש מושם.

כאשר יש מספר מעבדים, ניאלץ לספק עקבות בחומרה על מנת שכל המעבדים יראו את הערך העדכני ביותר ב-cache שלהם.

### DMA – Direct Memory Access

מחבר בין הזיכרוון למכשיר.

משתמש בחטקי קלט/פלט מהירים, המסוגלים להעביר מידת ומהירות הקרובה למאהירות הזיכרוון. כך אין צורך ב-interrupt כל פעם שנרצה להעביר מידע מהזיכרוון לאחסון חיצוני או להיפק. רק נזדקק ל-interrupt בסיום.

לא טוב להשתמש בו במקומות בהם מעבירים דברים חשובים ומעטים. למשל לא נרצה להשתמש בו עבור הקלדות מקלדת, מכיוון שכן הוא ישמור את כל ההקלדות ובבת אחת יקליד את הכל בסיום.

### Multiprogramming

בעיה: כאשר יש לנו תוכנית בודדת, היא לא יכולה לשמר את המעבד וההתקנים יחד בניצול מלאה. התוכנית רצתה בקורס סדרתית ולפערמים מבצעת חישובים ולפערמים פונה ל-I/O.

אידיאלית היינו רצאים שבכל רגע נתון תהיה לנו תוכנית שתoruץ על המעבד.

בעזרת Multiprogramming ניתן הריצה של מספר תוכניות במקביל (יותר מכמה מעבדים).

### Timesharing

עדין עושים שימוש בתדריות גבוהה כל כך, שהמשתמש לא ישים לב שהתוכנית הפסיקה.

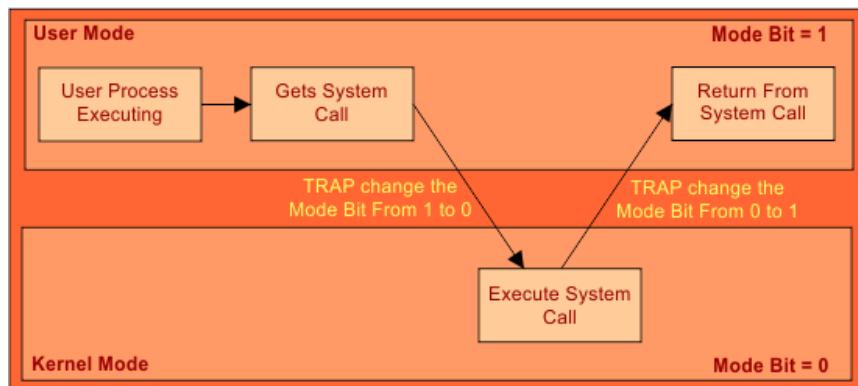
### טיפול בעביעות זרימה (Flow)

מערכת הפעלה מבצעת הגנה מפני לולאות אינסופיות, תהליכי שמנסים לשנות זיכרוון של תהליכי אחרים וכו'.

### מעבר בין Kernel Mode ו-User Mode

הבעיה – מערכת הפעלה נדרשת להגן על עצמה מפני שימוש לא נכון/zidoni.

הפתרון: נזכיר בית שיושב על המעבד. כשייה דלוק זה סימן שאנו מרים קוד של המשתמש. אחרת זה סימן שאנו פועלם בשם מערכת הפעלה. אם הוא על kernel מותר לו להריץ הכל. אם הוא על User לא ניתן לבצע פעולות מסוימות על מנת להגן עליו עצמו.



## **הרצאה 2**

הערה לפטיחה : ההבדל בין מערכת הפעלה למכשיר משתמש פשוט היא בעושר פעולות שניתן לבצע.

### **מכשיר משתמש**

כמעט לכל מערכות הפעלה יש UI, המוצע לקלוט את רצון המשתמש ולהפוך אותו לפעולות. בפרט, יש במערכות הרבות פעמים ממשק CLI – command line – מטרתו היא לקלוט פקודות שהמשתמש רושם.

המכשיר הנפוץ ביותר ביום הוא GUI – ממשק גרפי.

VUI – ממשק קולי (ALKSA, סירוי וכו').

ישנם עוד ממשקים רבים, שמטרתם זהה – לקלוט את רצון המשתמש ולהמיר אותו לפעולות.

נრחיב בעיקר על ממשק ה-CLI.

### **CLI**

המשתמש מזין פקודה, ה-CLI לוקח את הפקודה והופך אותה לרצף של system calls. ישן כמה דרכים למשוך CLI :

1. חלק מה-kernel. (תמיד נמצא בזיכרון, ה-command interpreter קופץ למקום המתאים בזיכרון).
2. באמצעות תכניות השמורות בדיסק, ככל פקודה היא למעשה שם של תוכנית.

זמן הריצה מהיר יותר ב-kernel. לעומת זאת בדיסק קל יותר לתחזק (הוספה feature בשיטה השנייה לא דורשת שינוי של ה-shell).

הרבבה פעמים יהיה לנו כמהinterpreters במערכת הפעלה – נקרא להם shells.

### **פונקציונליות שעוזרות למשתמש**

#### הרכבת תוכניות

1. טיענת התוכנית לזיכרון.
2. הרכבת התוכנית.
3. סיום הריצה – בין אם התוכנית מסיימת עצמה או מתווך שנייה/הפרעה.

### **טיפול ב-Ο/I**

תמיד נרצה שהגיישה ל-Ο/I תעבור דרך מערכת הפעלה משיקולי בטיחות.

### טיפול בקבצים

כתיבה וקריאה של קבצים, מחיקה ויצירה שלהם, חיפוש, הדפסת המידע שלהם, הרשאות.

### תקשורת בין תהליכיים

העברה המידע בין שני תהליכיים או יותר – מדובר בתקשורת גם בין תהליכיים הם באותו מכשיר וגם אם לאו.

### טיפול בשגיאות

שגיאות במעבד ובזיכרון, שגיאות בתוכני O/I, שגיאות בתכניות שרירות.

לכל סוג שגיאה על מערכת הפעלה לנוקוט בפעולה המתאימה כדי לטפל בה, על מנת להבטיח חישוב נכון ועקבי.

### הבטחת יעילות של המערכת

הकצתה משאבי חכמה, כאשר כמה משתמשים מחוברים במקביל.

### Accounting

רישום של מערכת הפעלה של כל השימושים במשאבי. פעולה כמו Log.

### אבטחה

להבטיח שכל גישה למשאבי המערכת תקין ותחת שליטה של המערכת.

## System Calls

הם מנגן המאפשר לתוכנות שרצות על המחשב לבקש שירותים מסוימים ממערכת הפעלה.

הן אלו שמהוות את משקל התוכנות שלנו לשירותים המשופקים על ידי מערכת הפעלה.  
השימוש שלהם בודך כלל באמצעות מספר שמיוחס לכל system call – API.

לרוב הקראיה ל-system calls לא יתבצעו על ידי קראיה ישירה אלא על ידי Application Programming Interface – API

ה-API מגדר את סט הפונקציות הזמינות.

יתרונות לקריאה דרך API:

1. פשוט יותר לשימוש: אין צורך לדעת את כל מה שקרה מאחורי הקלעים.
2. ניידות ווגमישות: שינוי במבנה או הוספה ל-system calls לא תשפיע על מי משתמש ב-API.

חסכנות לקריאה דרך API:

1. ביצועים נמוכים יותר, מכיוון שצריכים לעבור קודם ב-API שעובר לאחר מכן ב-syscalls.
2. שליטה פחות מדויקת. נובע מהעובדת שהחרנו לא לדעת את כל הפרטים הקטנים מאחורי הקלעים.

דוגמה: נניח שימוש קריאת syscall open(). בפועל הוא קורא ל-API open(), שימצא את ה-syscall interface. הוא ימצא את השימוש של open() וייחזר אותו.

דוגמה נוספת: printf היא פונקציית API מתקדמת, משתמשת ב-formatting, מכינה טקסט וכו'. היא משתמשת ב-write מאחורי הקלעים. write עצמה היא פונקציה פשוטה יותר שמדפיסה את הטקסט. היא משתמשת ב-system\_write מאחורי הקלעים.

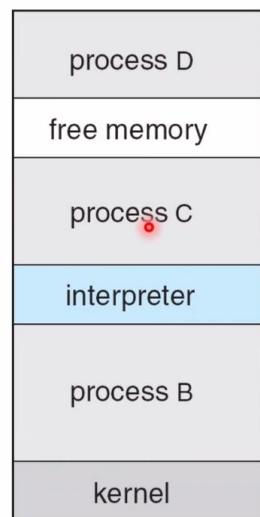
אם נדרשים להעביר פרמטרים ל-syscall, אנחנו יכולים להעביר דרך רשיטר, או לשים את המידע בזיכרון ולהגיד למערכת הפעלה איפה בזיכרון המידע נמצא. כמו כן אנחנו יכולים גם כMOVן להעביר דרך ה-stack.

מבנה UNIX:

## Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code

Interpreter is  
always running  
because of the  
multiprogramming



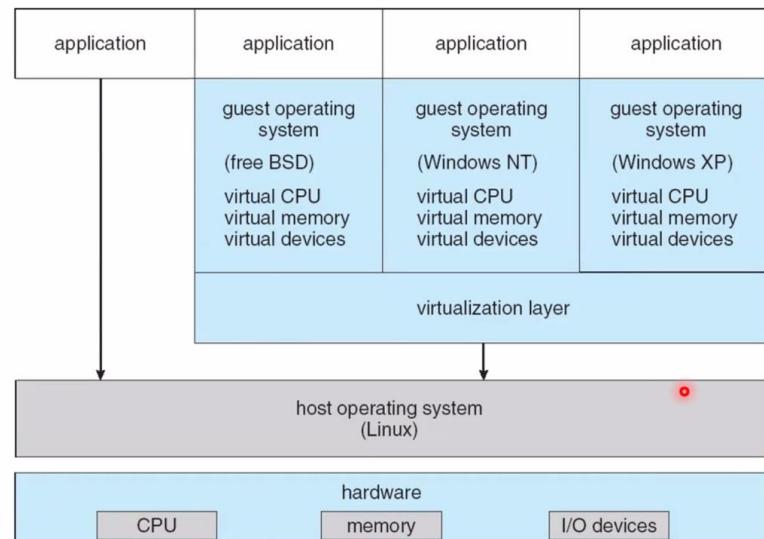
## Virtual Machine

מתיחס לחומרה ולמערכת הפעלה כאילו שניהם יחד הם רק חומרה. מערכת הפעלה הקיימת היא המארחת. ה-VM משתמש ב"שכבת וירטואלייזציה" על מנת לדמות מערכת הפעלה אחרת ממנה שקיימת בפועל.

מה שכבת הווירטואליזציה עשויה בפועל זה לתרגם את פקודות ה-API של ווינדוס לפקודות של לינוקס, ואז להשתמש בלינוקס שモתקן כבר כדי להריץ אותו.

הארQUITקטורה של ה-VM נראה כך :

## VMware Architecture



### Debugging

המחשב שומר Log Files לגבי מידע השגיאה. למשתמשים אין הרבה מה לעשות עם זה, אבל הרבה פעמים ניתן לשולח את המידע ל-Microsoft והם ידעו לפתור את הבעיה.

## הרצאה 4

מחזור החיים של תהליך בינוי מרצף של מעברים בין CPU execution ו-I/O.

מחזור החיים תמיד מתחילה ומסתיים ב-CPU burst.

### CPU scheduler

ה-CPU scheduler אחראי על לבחור איזה תהליך מבין התהליכים שנמצאים ב-ready queue לקבל את המעבד.

מצבים בהם כדי להפעיל את ה-scheduler :

- .1. כאשר תהליך עובר ממצב waiting ל-running.
- .2. כאשר תהליך עובר ממצב ready-running ל-ready.
- .3. כאשר תהליך עובר ממצב ready-ready ל-waiting.
- .4. כאשר תהליך מסיים את פעולתו.

אם ה-scheduler פועל רק באירועים מהסוג הראשון והרביעי, אנחנו אומרים שהוא nonpreemptive. במקרה כזה, המעבד מחזק בתהליכי קלשנו עד שהוא מסיים או עובר למצב waiting.

### Dispatcher

ה-Dispatcher בפועל מבצע את החלטות השה-scheduler קיבל. מעביר את השליטה על המעבד לתהליך שנבחר.

תפקידו :

- .1. ביצוע context switch
- .2. מעבר ל-user mode
- .3. עוד

### Scheduling

לפי מה נקבע כמה טוב ה-scheduling ?

- .1. ניצול המעבד – נשאף לניצולם ובואה ככל האפשר.
- .2. Throughput – מספר התהליכיים המסיימים ריצה בפרק זמן.
- .3. Turnaround time – משך הזמן הכלול להרצה התהליך. כולל:
  - הזמן שהמתין עד שהוכנס לארון.
  - המתנה ב-ready queue.
  - זמן הריצה על המעבד.
  - הזמן שבילה בביבט (או המתנה ל-IO).
- .4. Waiting time – הזמן היחיד שבאותו מושפע מה-scheduling.
- .5. Response time – הזמן מהרגע שהתהליך ביקש את המעבד ועד הרגע שקיבל אותו לראשונה.

### קריטריוני minmax

נדיף לא רק למזער את הזמן, אלא שכל התהליכים יקבלו זמנים יחסית שוויים. Ndif זמן של 20 שניות בממוצע, שככל תהליך מקבל שנייה, מאשר 15 שניות בממוצע שיש לתהליך אחד שלוקח 10 שניות.

## scheduling ל-

first come-first serve. מי שmagiu קודם מקבל שיבוץ למאבד קודם.

Shortest job-first – בוחר את התהליך שאורך ה-CPU burst שלו הוא מינימלי. ניתן למשוך זאת בשתי דרכים :

- .1. Nonpreemptive – ברגע שתהליך שובץ, אפילו אם הגיע אחד עם זמן קצר יותר מהזמן שנותר לתהליך הנוכחי, נכח שיטתיים התהליך הנוכחי.
- .2. Preemptive – אם מגע תהליך חדש עם CPU burst קצר יותר מהזמן שנותר לתהליך הנוכחי, ניתן לו לירוץ ואז נחזיר.

ה-jf scheduling מבטיח את הזמן המתנה הממוצע המינימלי עבור כל סט של תהליכים – כאשר הוא preemptive וזמן ה-context switch זניא. זה מתבסס על ההנחה שהחלפה של תהליך ארוך בקצר תביא בהכרח להקטנת זמן המתנה הממוצע. הבעיה העיקרית בשיטה היא שאנו לא באמת יודעים את משך ה-burst הבא.

## היזוי על פי תצפיות עבר

דרך שכיחה לשער כמה זמן ייקחurst-הבא על פי תצפיות עבר.

## Exponential smoothing

השיטה מספקת ממוצע משוקל נע של כל תצפיות העבר. מטאים בעיקר למידע שאין לו מגמה (עליה או ירידה). המטריה היא להערך את הממוצע הנוכחי ולהשתמש בה כאומדן לעתיד. באופן פורמלי:

$$F_{t+1} = \alpha y_t + (1 - \alpha)F_t$$

כאשר  $F_{t+1}$  הוא החיזוי ל-*burst* הבא.  $\alpha$  הוא קבוע *smoothing* הנכפה בסדרה במחזור  $t$ .  $F_t$  היא התחזית הקודמת.

בעצם כל פעם "מתקנים" את התחזית הקודמת בהתבסס על הערך החדש שנכפה בפועל.  $\alpha$  קבוע כמה נרצה להסתמך על התצפית האחרונה/על התחזיות הקודמות.

הנוסחה שකולה לנוסחה:

$$F_{t+1} = \alpha y_t + \alpha(1 - \alpha)y_{t-1} + \dots + \alpha(1 - \alpha)^t y_0$$

ערך  $\alpha$  גבוע – אנחנו משתמשים יותר על התצפיות האחרונות.

ערך נמוך – יציבות בחיזוי. נסתמך יותר על התחזיות הקודמות.

## Priority scheduling

לכל תהליך מצמידים מספר שמייצג את החשיבות שלו. המעבד מוקצה לתהליכי עם החשיבות הגבוהה ביותר. ניתן לבצע את השיטה זו בעזרת צורת preemptive או nonpreemptive.

ה-*SJF* הוא סוג של priority scheduling, שבו ה-*burst* הוא משך *burst* הבא.

הבעיה העיקרית בשיטה זו היא Starvation/Aging.

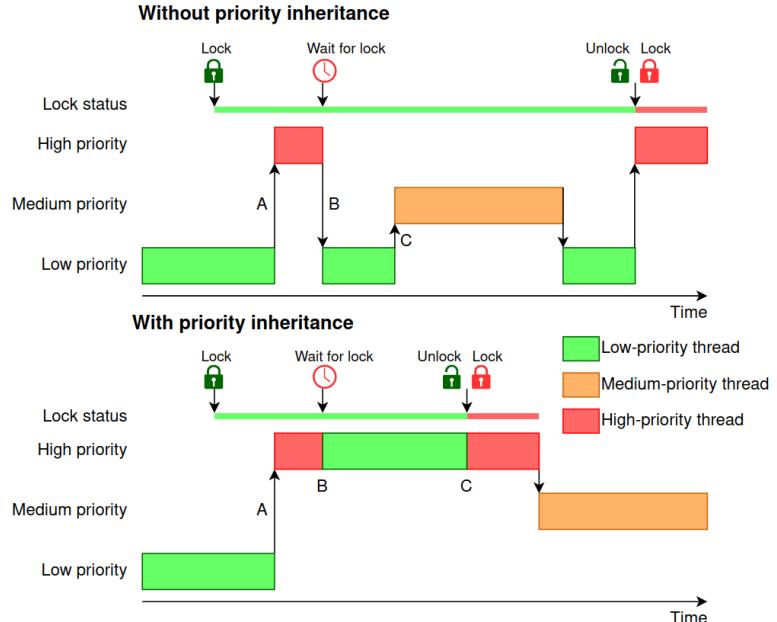
## Priority Inversion

נניח שתהליכי עם חשיבות נמוכה פותח קובץ, ואז אנחנו עובדים לתהליכי עם חשיבות גבוהה. בעת התחלתם עם החשיבות הגבוהה ניסה לפתוח את אותו הקובץ. נדרש לחזור לתהליכי הקודם, להפסיק להסתמים בקובץ, ורק אז לחזור לתהליכי עם החשיבות הגבוהה.

בעיה: אם כשנוחזר לתהליכי הנמוכה ייכנס עוד תהליכי עם חשיבות בינונית, נדרש לחכות עוד זמן

מוסף עד שנוכל לחזור לתהליכי החשוב.

פתרון: נשנה את החשיבות של התהליכי הנמוך לחשיבות של התהליכי הגבוהה שתלו依 בו.



### Round Robin

שיטה המיעדת במיוחד למערכות time-sharing. כל תהליך מקבל זמן יחסית קצר על גבי ה-CPU. אם תהליך לא הסתיים, הוא מועבר חזרה לסוף התוור.

### Multilevel Queue

נחלק את ה-ready queue לכמה תורים נפרדים. לכל תור נקצת אלגוריתם scheduling מסויל (לא בהכרח זהים). scheduling מבוצע בין התורים. למשל ככל להחלטת שאנו לנו מיד עדיף לשרת תהליך מתוך ה-foreground. כמו כן, כל תור יוכל לקבל כמות מסוימת של זמן מעבד שאותה יוכל לחלק בין התהליכים. למשל: FCFS background עם RR 20% מהזמןforeground. ב-MLFQ, הרמה הנמוכה ביותר בדרך כלל מתנהל בעזרת FCFS.

### Scheduling עם יותר מעבד אחד

נפריד בין שני מצבים:

1. Asymmetric multiprocessing – רק מעבד אחד פונה ל-data structures הקשורות בתזמון ומנהל את התזמון.
  2. Symmetric processing – כל מעבד מזמין לעצמו.
- Processor affinity – לתהליכי יכול להיות affinity (צמידות) לمعالג עלייו הוא כרגע רץ. זה על מנת להשתמש באופן היעיל ביותר. מערכת הפעלה תומכת ב-affinity בשתי צורות:
1. Soft affinity – אין הבטחה שאותו המעבד יהיה בשימוש.
  2. Hard affinity – אם התהליך מבקש, הוא לא יועבר למעבד אחר.

מערכות real-time – מבקשת לסיים משימה קריטית בתוך פרק זמן מסוים. המתזמן יבטיח את קיום הבקשה או שידחה אותה מראש.

מערכות Soft real time – תהליכיים קריטיים מקבלו עדיפות, אך לא מובטח שהבקשה תתקיים בזמן המבוקש.

### שערוך אלגוריתמים

ניתן להעיר את המערכת היעילה ביותר לאלגוריתם נתון בכמה דרכי:

1. לוחץ סביבת עבודה נתונה מראש ומגדיר את הייעולות של כל אלגוריתם על סביבת העבודה זו.
2. תורת התורים – תחום שעוסק בחישוב ביצועים של מערכות ניהול תורים.
3. סימולציה – נריץ סימולציה של המתזמון וכך ניתן לחשב את זמן המתנה.
4. אימפלמנטציה – נבדוק את כל המתזומנים ונבדוק עם מה היה הכי טוב.

## הרצאה 5

### מנגנון סינכרונייזציה

אם למשל נתונים לשני תהליכי לגשת בו זמנית למשתנה מסוים, יכולות להיות בעיות (מידע לא מסונכרן וכו'). במצב זה, שבו כמה תהליכים משנים את אותו ה-*data* נקרא *race condition*. – קטע קוד שבו התהליך משנה משתנים משותפים עם תהליכים אחרים.

בהתנן מערכת עם *n* תהליכים  $\{k_0, k_1, \dots, k_{n-1}\}$ , שלכל אחד יש critical section, נרצה שם תהליך נמצא ב-*critical section* שלו, אף תהליך אחר לא יירץ את ה-*critical section* שלו. critical section נבנה פרוטוקול, לפיו כל פעם שאנו נכנסים וכל פעם שנצא מ-*critical section*, נריץ קוד כלשהו, שינהל את הכניסה/יציאה.

### פתרונות לבעיה

#### פיטרסון:

פתרון מבוסס קוד, מבוסס על שני משתנים משותפים :

Int turn;

Boolean flag[2];

המשתנה *turn* קובע תור איזה תהליך.

המשתנה *flag* מסמל מי מבין שני התהליכים רוצים להיכנס ל-*critical section*.

```
Flag[i] = true;  
Turn = j;  
While(flag[j] && turn==j);  
    // Critical section  
Flag[i] = false;
```

כל הפתרונות מבוססים על lock כלשהו. כמו רמזו שלא ניתן למכוון מכמה כיוונים להיכנס ביחד.

### Lock

נעלו לפני ונפתח אחרי ה-*critical condition*. בעיה : יכול להיות שני אלץ לחכות זמן לא מוגבל.

הՃדרה : פקודה אוטומית היא פקודה שלא ניתן לעשות לה interrupt.

דוגמאות לפקודות אוטומיות :

- .1 – *Test\_and\_set* – מחזיר את הערך של המשתנה המקורי, מעדכן את הערך שלו ל-true.
- .2 – *Swap* – מחליף בין ערכם של שני משתנים בוליאניים.

### Bounded waiting

התהליך מכחח שיתפונה הקטע ומתייל להריץ אותו. לאחר מכן התהליך בודק בצורה ציקלית אם יש תהליך אחר שמחכח לרוץ. אם כן הוא נותן לו את הגישה והקטע עובר לתהליך הזה.

הפרוטוקול הזה מקיים את כל 3 התנאים :

- .1 – Mutual exclusion – רק אחד בכל פעם עם גישה.
- .2 – בודאות אחד מהתהליכים הממתינים יוכל להתקדם.
- .3 – במקורה הכירע מחייבים ( $0 - n$ ) – Bounded waiting

## Semaphore

משתנה שנייתן לבצע עליו פעולות indivisible. משמש לנסכוון.

## בעיית הפילוסופים האוכלים

ישנם 5 פילוסופים מסביב לשולחן, שבכל רגע הם הוגים או אוכלים. על השולחן 5 מקומות אכילה ומרכזו קערת אורו. כאשר פילוסוף נהייה רעב הוא מנשה להרים את שני מקומות האכילה הקרובים אליו.

אלגוריתם :  
נכחה לשני מקומות האכילה, ואז נאכל. נחזיר אותם ואז נחזר לחשוב.  
בעיה : עלול לגרום ל-starvation.

## פיטרסון עם יותר משני תהליכיים

כל תהליך מתקדם בשלבים לעבר הכניסה ל-critical section.

כל פעם התהליך יסמן באיזה שלב הוא ב-flag (1,2 עד 1 – a). בנוסף ב-turn נסמן את התהליך באינדקס של השלב. (אם אני תהליך i בשלב k, אז  $i = [turn[k]]$ ).

ברגע שכל התהליכיים בשלבים שהם מוכנים مثل התהליך, או שאחד התהליכיים האחרים "חשב" שהוא לפני התהליך, זה סימן שנייתן לקפוץ בראש התור ולהריץ את ה-critical section. לאחר מכן נסמן את השלב שלנו ב-1 וначזור על התהליך.

## הרצאה 7 – Deadlock

— מצב בו כל תהליך מוחכה לתהיליך אחר שיסיים (כדי שימושו כלשהו ישחרר). יוצא מצב שכל התהליכים מוחכים בלי להתקדם.

### מבנה המערכת

מערכת מורכבת מהתהליכים ומשאבים. משאבי יכולם להיות consumable או reusable. את המשאים אנחנו מחלקים לקבוצות שנקראות resource types ( $R_1, R_2, \dots, R_m$ ). לכל משאב  $R_i$  יש  $W_i$  מופעים. אנחנו משתמשים במשאבים באופן של request-use-release.

### תנאים לדד-лок

נגיד שאנו במצב של דד-лок, אם מתקיים :

1. Mutual exclusion – בכל רגע נתון רק תהליך אחד יכול להשתמש במשאב מסוים.
2. Hold and wait – תהליך שמחזיק לפחות אחד מחייב לקבל משאים נוספים שמהזקם על ידי תהליכי אחרים.
3. No preemption – משאב יכול להיות משוחרר רק כשהתהליך שמחזיק אותו שחרר אותו מרצו.
4. Circular wait – קיים סט תהליכי מותניים  $\{P_0, \dots, P_n\}$  כך ש-  $P_0$  מוחכה למשאב שמחזיק על ידי  $P_1, P_2, \dots, P_{n-1}$  מוחכה למשאב שמחזיק על ידי  $P_n, P_0$  מוחכה למשאב שמחזיק על ידי  $P_0$ .

### גרף הקצאת משאים

נייצג תהליכי באמצעות קודקודים, ואת המשאים באמצעות ריבועים. כמות הנקודות בתוך הריבועים מייצגים את כמות המופעים של אותו משאב. בעת חץ מהתהליך למשאב מסמל שהתהליך מבקש לשימוש במשאב, וחץ מהמשaab לתהליך מסמל שהמשאב מוקצה לתהליך.

נשים לב : אם הגרף לא מכיל מעגל  $\Leftrightarrow$  אין דד-лок.  
אם הגרף מכיל מעגל  $\Leftrightarrow$  אם לכל משאב יש מופע אחד, זה בהכרח דד-лок. אחרת, יש רק אפשרות לדד-лок.

## התמודדות עם Deadlock

יש לנו שלוש שיטות להתמודד עם דד-лок :

1. נודא שאנו לא נכנסים אף פעם לדד-лок, באמצעות תקיפת ארבע התנאים לדד-лок.
2. נמצא דרך להתח渺ש מdad-lock במצב שכן נכנסו אל אחד.
3. הפעולות – נועלם מהעובדה שהמערכת יכולה להיכנס לדד-лок.

איך נודא שלא ניכנס לדד-лок? איך נתמודד עם כל אחד מהתנאים?

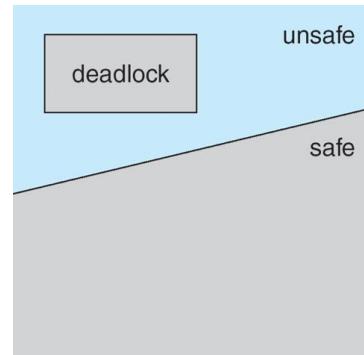
1. Mutual exclusion – לא נדרש עבור משאים שיתופיים (למשל קבוע read-only). עבור משאים אחרים נדרש לוודא שתפקיד.
2. Hold and wait – ננסה לוודא שכאר תהליך מבקש משאב, הוא לא מחזיק אף משאב אחר. למשל נוכל להחיליט שהתהליך יבקש את כל המשאים שהוא צרי מהתחליה, אז נקצת לו הכל בתבאת אחת. אופציית נוספת היא לאפשר לו לבקש משאים רק כאשר המשאב לא משומש, על ידי אף תהליך אחר. בעיה : הਪתרונות האלה עלולים לגרום לstarvation.
3. No preemption – אם תהליך שמחזיק משאים מבקש משאב נוסף שלא ניתן להקצות לו מייד, השחרר את כל האחרים שלו. משאים שנקבעו מראש נוספים לרשימה המשאים שהתהליך ממתין להם. תהליך יאותחל רק כשהוא יכול לקבל מחדש את המשאים שהוקצו לו, כולל החדים שהוא בקש.
4. Circular waiting – קודם נסדר את כל המשאים. לאחר מכן נדרוש שככל תהליך יבקש משאים בסדר עולה של ספירה.

גישה שנייה – התחממות מdad-lock בזמן הריצה

הגישה הפюטוטה ביותר דורשת שככל תהליך יזכה על כמות מקסימלית של משאים שהוא עלול להזדקק להם. לאחר מכן נשתמש בdeadlock-avoidance algorithm כדי לוודא שלא מתקיים deadlock-condition. מצב הקצאות משאים מוגדר על ידי : 1. כמות המשאים הזמינים, 2. כמות המשאים המוקצים, 3. הדרישה המקסימלית של התהליכים.

## Safe state

נגיד שהמערכת נמצאת ב-safe state, אם קיימים רצף של כל התהליכים במערכת  $(P_1, P_2, \dots, P_n)$  כך שלכל  $P_i$ , כל המשאבים שהוא עדין יכול לבקש הם פנוים או שМОוחזקים על ידי  $P_j$  כאשר  $i < j$ . כך, כל פעם שתתהליך  $P_i$  מסיים, התהליך  $P_{i+1}$  יכול לרוֹץ. נשים לב: safe state גורר שהכרח אין דלוק, אבל אם אנחנו לא ב-safe state זה לא בהכרח אומר שההדרה דלוק.



כיצד נשמר על המערכת ב-safe state?

- אם לכל משאב יש מופע אחד, נשתמש בגרף הקצתה משאים.
- אחרת, נשתמש באלגוריתם Banker.

בגרף נסמן  $R_i \rightarrow R_j$  עם חז מקוקו, אם יתכן ש- $R_i$  יבקש את המשאב  $R_j$  כשתהליך מבקש את המשאב בפועל, החז יהפוך לחז רגיל (לא מקוקו). כאשר הוא מקבל את המשאב החז יהפוך לחז בקשה לחז הקצתה (בפועל הוא ישנה כיון בגרף).

אלגוריתם לגרף הקצתה משאים: בקשה  $R_i \rightarrow R_j$  תתקבל רק אם אחרי שהתקבלה לא נוצר מעגל.

## Algorithm Banker

ניתור את המשתנים הבאים:

$.Available$  – רשימה של  $m$  מספרים. אם  $Available[i] = k$  קיימים  $k$  מופעים פנוים של  $R_i$ .  
 $.Max$  – מטריצה  $m \times m$ . אם  $Max[i, j] = k$  תהליך  $P_i$  יכול לבקש בכל היותר  $k$  מופעים של  $R_j$ .  
 $.Allocation$  – מטריצה  $m \times m$ . אם  $Allocation[i, j] = k$  קרגע מ Każה  $k$  מופעים של  $R_j$ .  
 $.Need$  – מטריצה  $m \times m$ . אם  $Need[i, j] = k$  צריך עוד  $k$  מופעים של  $R_j$  על מנת לסייע את המשימה.

האלגוריתם:

1. נגיד וקטוריים  $.finish[i] = false$  ( $i = 1, \dots, n$ ),  $.work = Available$
2. נמצא  $i$  עבورو:  
 $.finish[i] = false$   
 $Need_i \leq work$
3. אם לא קיימ  $i$  כזה, דלג לשלב 4.  
 $work += Allocation$   
 $.finish[i] = true$
4. אם  $.finish[i] == true$  לכל  $i$ , המערכת נמצאת ב-safe state

## הרצאה 6

כאשר שני CPU מחלקים ביניהם זיכרון, כל אחד שומר את הזיכרון שלו בשני אופנים – כתובות פיזיות וכתובות לוגיות.

כתובות פיזיות : הכתובת של משאנו במקום המקורי שלו על ה-RAM.

כתובות לוגיות : הכתובת של משאנו ביחס לזכרון המקורי לאוטו CPU או תחילה.

על מנת להשתמש בכתובות לוגיות, שכן נוחות בהרבה לשימוש, אנחנו צריכים להציג לכל כתובות פיזיות כתובות לוגיות, או לפחות דרך "לתרגם" ביניהם. נוכל לעשות זאת או בזמן הקימפול (זה יחייב קומpileציה מחדש במקורה ורוצים להריץ במקום אחר בזיכרון), או בזמן ה-loading (זה יחייב יצירה של execution, או בזמן המה-מפלץ, relocatable code (relocatable code), או בזמן ה-translation).

ביצוע התהיליך הזמן הריצה מאפשר הזזה של תחילה בזיכרון תוך כדי ביצוע. בעיה: זה דורש כל פעם לתרגם מחדש הכתובת פיזית.

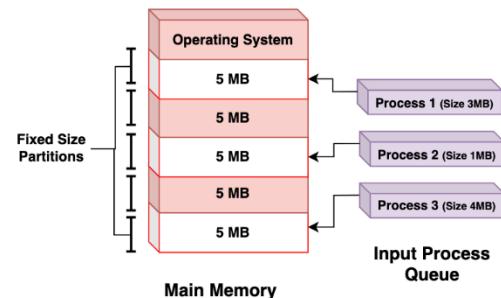
### MMU

מטרת ה-MMU שדיברנו עליו בהרצאה הראשונה היא לתרגם בין כתובות לוגיות לפיזיות. זה נעשה על ידי ה-relocation register (מוסיפים אותו לכתובת הלוגית).

הערה : כל זה נכון לגבי זיכרון לוגי שמור בצורה רציפה, לגבי זיכרון שאינו שמור בצורה רציפה נדבר בהמשך.

### Contiguous Allocation

נחלק את הזיכרון לשני חלקים רציפים. באחד נשים את מערכת הפעלה ובשני את התהליכים שMRI צים המשמשים. נשתמש ב-base register על מנת להגן על התהליכים של המשמשים זה מזיה. משתמשים בשני גxisטים: אחד base שאומר מה הכתובת היכי מוכחה, ואחד limit שאומר את הכתובת היכי גבוהה שהתחילה אמרור לרוֹץ בו. אם חורגים מטווח הזיכרון שהוקצה, נפעיל trap.



בכל פעם שmagiu process חדש נחפש לו חור בזיכרון שמספק גודל עבורו.

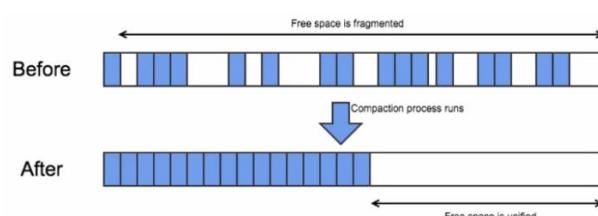
ונכל להקצות או לפי החור הראשון שאנו מוצאים שמתאים, או לפי החור היכי קטן שמתאים, או לפי החור היכי גדול שמתאים.

### Fragmentation

פרגמנטציה חיצונית – יש זיכרון פנוי ב-RAM שמספיק ע"מ להכנס תחילה חדש, אבל הוא מלא בחורים, ככלומר הוא אינו רציף.

פרגמנטציה פנימית – ניתן לתהיליך יותר זיכרון ממה שהם צריכים, לדוגמה שהתחילה יגדל. כתע אם מגיע לתהיליך חדש זה נראה כאילו אין לנו מספיק זיכרון עבורו אבל בפועל לא כל הזיכרון משומש.

נפתר את הבעיה האלה באמצעות compaction.



## Paging

הweeney: אחסון תהליכיים בזיכרון לצורה לא רציפה. ניקח את הזיכרון הפיזי ונחלק אותו לבלוקים בגודל קבוע, להם נקרא frames. גודל הframes הוא חזקה כלשהי של 2. גם את מרחב הכתובות הלוגי נחלק לבלוקים, שלחם נקרא page table. לכל page הוזה נקרא frame. למיפוי זהו נקרא pages.

### TLB

טבלת הדפים נשמרת בזיכרון מהיר TLB, בדומה ל-cache. נחפש בו את התרגומים לדף כלשהו. אם מצאנו את התרגומים מצב זהה ייקרא TLB hit, אחרת TLB miss.

מה תוכלת הזמן לגישה לזמן ב-TLB? נקרא לו בקיצור EAT. נסמן את זמן החיפוש הוא  $\epsilon$  ייחידות זמן. בנוסף נניח שזמן הגישה לזמן ב-TLB הוא 1. נסמן את יחס הפגיעה ב- $\alpha$ . זהו אחוז הפעמים שקיבלו hit בחיפוש.

אזי:

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

מכיוון  $\alpha$  הוא אחוז הפגיעה, ו- $(1 - \alpha)$  הוא אחוז הפספוסים.

### Memory Protection

כדי למנוע מתחילה לגשת לדפים שהוא לא אמור לגשת אליהם, נוכל להשתמש ב-bit valid שישמן האם דף מסוים הוא valid.

באמצעות טבלת הדפים נוכל גם לחלק דף בין שני תהליכיים. נרצה לעשות זאת ב-shared memory או כאשר תהליכיים נוספים צריכים לגשת למידע מתחילה כלשהו.

### Two level paging

במקום לשומר טבלה אחת עם מלא דפים, נשמר סוג של "טבלה חיצונית" שתעזר לנו לנוט אל העמוד. אנחנו בצע paging לטבלה עצמה, ונשתמש בטבלה החיצונית כדי לנוט לחלק בטבלה שאנו צריכים, ומשם אל העמוד שאנו צריכים. ניתן באופן דומה לבצע Three level paging וכן הלאה.

נוסחה כללית ל-EAT של n-level paging :

$$EAT = \alpha(\epsilon + m) + (\alpha - 1)(\epsilon + (n + 1)m)$$

כאשר :

$\epsilon$  – אחוז הפגיעה ב-TLB – hit rate –  $\alpha$ .

$m$  – זמן הגישה ל-TLB.

$n$  – זמן הגישה לזמן הפיזי.

הרבה פעמים ניתן להניח  $m = 1$ .

## הרצאה 8 – זיכרון וירטואלי

תזכורת : דפים מאפשרים לנו לשמור את הזיכרון כאשר הוא לא שמור כלו בראציפות, אך בכל דף בנפרד מתקיים רציפות.

זיכרון וירטואלי מהווה הפרדה חזקה יותר בין logical memory ו-physical memory. הוא בעצם מאפשר רק חלק מהתוכנית לזכור לשם הרצה – כלומר ניתן להריץ תוכנית גדולה יותר מאשר מהזיכרון עצמו.

### Demand Paging

הweeney – נביא לזכור את תוכן ה-page רק כשהאחת נדרשת – כלומר רק כאשר הוא מכיל instruction שדרישים להריץ או שהוא מכיל data שאנו צריכים.

Lazy swapper – נביא לזכור דף רק אם הוא באמת נדרש. pager השוכן בדפים נקרא swapper.

אופן פעולה : אם דף נדרש, נבצע reference אליו.

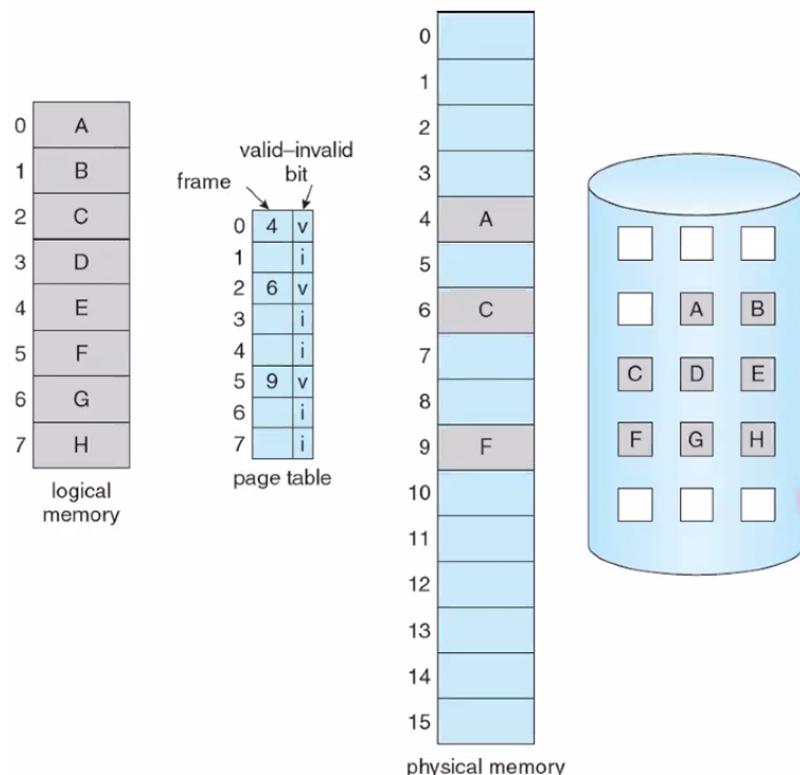
אם ה-reference לא קיים -> abort. אחרת, אם הדף חוקי אך לא נמצא בזיכרון -> נביא אותו לזכורו.

יתרונות של ה-swapper : lazy swapper

- פחות O/I ביצירת התהילך.
- נדרש פחות זיכרון להרצאת התהילך.
- מהירות גבוהה יותר ביצירת התהילך (אך לא בהרצה עצמה).
- ניתן לשרת יותר משתמשים על בסיס אותה כמות זיכרון.

אפשר להשתמש ב-table page על מנת לקבוע האם כלשהו נמצא בזיכרון בעורת ה-bit valid.

כך זה נראה כאשר חלק מהדפים לא מצויים בזיכרון עצמו, אלא בדיסק :



Logical memory – מתרגם בין מספר פריים לעמודות.

Page table – מראה איפה כל פריים ממוקם בזיכרון, והאם הוא valid/invalid.

Physical memory – הזיכרון – RAM.

## Page Fault

## איך נטפל ב-page fault

נניח שרצינו לגשת לעמוד כלשהו וקיים בו שהוא invalid. אנחנו נוציא trap למערכת ההפעה, ומערכת ההפעלה תקבע האם מדובר ב-**invalid reference** (או בצע abort) או שפשות הדף לא נמצא בזיכרון. אם מדובר פשוט בדף שאינו בזיכרון, מערכת הפעלה תחליף מקום המתאים בדיסק, תכניס אותו לזכרון, ותעדכן את ה-**page table**. לבסוף נבצע **instruction restart**, וכעת יוכל לבצע אותה.

מהו ה-EAT? נסמן ב-p את אחוז ה-page fault שלנו. איזי:

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

## Copy-On-Write

אפשר לתחליך אב ובן לחולק מלכתחילה את כל הדפים השמורים בזיכרון. רעיון: בעקבות שינוי באחד הדפים שנוצר כתוצאה מפקודה באחד התהליכים, נוצר עותק חדש.

בנוסף הוא מאפשר יצירה מהירה יותר של תהליכי הבני מכיוון שרק דפים שבהם מתבצע שינוי מועתקים.

## Page Replacement

מה געשה במאכט שבו אאנחנו רוצחים להבייא דף ליזיכרון ואין מקום עבורו?  
נמצא דף שכרגע בזיכרון איך איינו בשימוש, ונחליף אותו בדף שאאנחנו רוצחים להכenis. (לדף שנוציא קוראים ה-  
.victim frame).

לאחר החקלה נעדכן את ה-**page table** (גם של הדף שהוחזקנו וגם של הדף שהכנסנו), וביצע **restart** ל-

### אלגוריתמים ל-page replacement

## FIFO

נוציא את הדף שהגיע לזכרו הכי מוקדם מבין כל הדפים.

בעה : האנומליה של בלדי (Bélády's anomaly)  $\Leftarrow$  יותר page faults.

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 3 frames (3 pages can be in memory at a time per process)

□ 4 frames

1	1	4	5
2	2	1	3
3	3	2	4
			9 page faults

1	1	5	4
2	2	1	5
3	3	2	
4	4	3	
			10 page faults

## Optimal Algorithm

דומה לאלגוריתם האסטרטגיות שראינו באלו. כאשר נרצה להוציא דף כלשהו, נוציא את הדף שהשימוש הבא בו יהיה המאוחר ביותר. זהו אלגוריתם חமדי שתמיד יביא לכמהות ה-*page fault* הקטנה ביותר אפשרית.

LRU – least recently used

נוציא את הפריים שהשימוש האחרון בו היה לפני הכיל הרבה זמן.

בעה : כל פעם נctrיך למוצר עבור איזה פריים השימוש האחורי בו יהיה לפני הכיה הרבה זמן. נctrיך לשמר בכל דג counter של מתי הייתה הפעם האחרונה שנגע בו, וכל פעם שנגע בו נעדכן את ה-counter

לזמן הנוכחי.

ה策עה נוספת למשוך: נשמר stack, וכל פעם שנגענו בדף נעביר אותו מהמיוקם הנוכחי שלו ב-stack להכי למעלה. רעיון: במקומות שבהם נמצא ה-LRU, נמצא אפרוקסימציה. אולי לא תמיד נמצא את ה-LRU עצמו, אבל נמצא משחו קרוב אליו. נתחל מערך שכלי אפסים, כאשר כל בית במערך מייצג דף. כל פעם שנגע בדף, נשנה את ערך הביט המוחס אליו ל-1. כאשר כל המערך הוא 1, אפס את כלו.

נרצה בעצם להוציא דף כלשהו שערכו במערך הוא 0.

יש צורך במערך בגודל כמות הדפים שנמצאים בזיכרון.

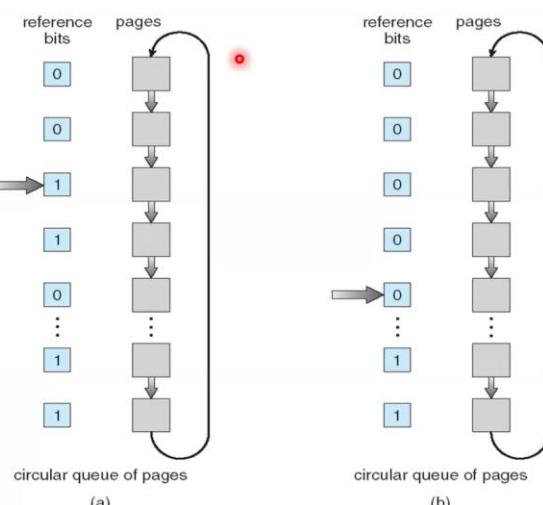
### Second-Chance

גם כאן לכל דף נשמר reference bit כמו באלגוריתם הקודם, יחד עם pointer על בית כלשהו במערך. כל פעם שנגעים בדף נשים ב-bit reference bit שלו ערך של 1.

אם קורה page fault, נתחיל לעבור על הביטים בסדר ציקלי. כל עוד הביט עליו מצביע ה-pointer הוא 1, נשנה אותו לאפס ונשאיר אותו בזכרון (כאן בא לידי ביטוי ה"הזרמת השניה"). ברגע שאנחנו מגיעים לביט שערכו 0 נבחר אותו להיות הקורבן, ונחליף אותו.

#### □ Second chance

- Need reference bit
- Clock replacement
- If page to be replaced (in clock order) has reference bit = 1 then:
  - set reference bit  $next\ victim$  0
  - leave page in memory
  - replace next page (in clock order), subject to same rules



### Counting Algorithms

אלו אלגוריתמים שמקצים לכל דף counter משלו, ואז לפי זה בוחרים את מי להוציא.

— מוציאה את הדף עם ה-counter ה-LFU.

— מוציאה את הדף עם ה-counter הכי גבוהה. (הגיון – הדף עם ה-counter הכי נמוך בטח הוכנס לא מזמן).

### Frame Allocation

כמה דפים צריכים על מנת להריץ תוכנית?

תשובה: למשל ב-IBM 370 דרישים 6 דפים כדי לטפל בפקודה MOVE: גודל הפקודה יכול להיות עד 2 עמודים, יחד עם 2 עמודים לטפל ב-from ושני עמודים לטפל ב-to.

### שיטות ל-Allocation

#### Fixed Allocation

ניתן כמות קבועה של פרטימרים לתוכנית. ניתן למשל לחלק באופן שווה את כל המשאים בין כל התוכניות, או קבוע מראש שמחולקים את המשאים ביחס לגודל התוכנית.

#### Priority Allocation

נשתמש בחלוקת פרופורציונאלית, לפי עדיפות במקום לפי גודל. אם בתהיליך  $P_i$  מתרחש page fault, יש לנו שתי אפשרויות:

1. נחליף את הדף בדף שכבר נמצא בזיכרון של  $P_i$ .
2. נחליף את הדף עם דף שנמצא בזיכרון של process עם עדיפות נמוכה יותר.

יכולים להיות לנו שני סוגים של מדיניות החלפה:

1. החלפה גlobלית: נחליף פריים מתוך כל הפריים בזיכרון.
2. החלפה מקומית: נחליף פריים מתוך כל הפריים בזיכרון של ה-process הנוכחי.

Thrashing – מצב בו לתהיליך כלשהו לא מוקצים מספיק פריים. זה גורם ל-page-fault גובה מאוד, וגם גורם לניצול CPU נמוכה ולכך גורם למערכת הפעלה לחשב שהיא צריכה להכניס עוד תהיליך. זה יגרום למצב שבו ניצול ה-CPU תהיה אפלו יותר נמוכה, כי התהיליך מבלה את רוב הזמן שלו בהוצאה והכנסה של דפים, ובפועל בקושי מרץ פקודות.

### עקרון הלוקליות

הסיבה ש-page demand עובד היא בגלל עקרון הלוקליות. כלומר אם היתי צריך עמוד כלשהו לא זמן, יש סיכוי גבוהה שאני אזדקק אליו עמו בקרוב.

גודל הלוקליות: כמות הדפים שהטהיליך צריך בקטע הזמן עד לשינוי הלוקליות הבא.  
אם לתהיליך מסוימים מוקצים פחות פריים מגודל הלוקליות, הוא נמצא ב-thrashing.

איך נמדד את גודל הלוקליות? נסמן  $\Delta$  בתור חלון זמן כלשהו אחריה (למשל, 10,000 פעולות אחרת). בעת נשמר ערך  $WSS_i$  עבור התהיליך  $P_i$  שיבדק כמה דפים השתמשו בחלון הזמן  $\Delta$ .

אם  $NSS_i = \sum D$ , נוכל למצוא את כמות הדפים הכוללת שנמצאו בשימוש בפרק הזמן האחרון. זהה בכךם הדרישה הנוכחית לפריים.

אם  $m > D$  (כאשר  $m$  היא כמות הפריים שיש לנו ב-RAM), זה סימן שייש לנו Thrashing.

לבסוף כל הזמן מהו working set דרוש המון עבודה, וכן נרצה להשתמש בזיכרון. למשל נשמר מערך של ביטים, שככל פעם שנגע בעמוד הנחוץ את הבית המתאים ל-1. כאשר מגיע 10,000 נגיעות (כלומר שנמצאה את  $\Delta$ ), נAPS את כל המערכת. אם נרצה ברגע נתוך לדעת את  $WSS$ , פשוט נספר את כמות הביטים הדולקים.

SHIPOR: נשתמש בשני מערכים, כל אחד מהם לחצוי אחר של  $\Delta$ . נעדכן את כל הערכים במערך העליון, וכשעברנו לחצוי בין המערך העליון והתחתון, ונAPS את המערך שעה למעלה. ואז נעדכן את הערכים של העליון וכשעבר ערד שוב נחליף ונAPS. בעת ה-WSS שלו נכוון בין 5,000 ל-10,000 נגיאות האחרונות.

### Page-Fault frequency Scheme

שיטת הבחירה פריים לתוכנית. בכל שלב נחשב את ה-page-fault של תוכנית מסוימת, אם הוא נמוך מדי נוריד פריים מהתוכנית, ואם הוא גבוהה מדי נוסיף פריים לתוכנית.

### Memory mapped files

ניתן להרחיב את הרעיון של page faults ושמירה בדיסק לקבצים שאנו רוצים לפתוח. נרצה לפתח קובץ, אז נחפש אותו בזכרון. אם הוא לא בזכרון נביא אותו מודיסק ואז הקריאה וכתיבות אליו יהיו מהירות בהרבה.

: העורט

1. לא חייבים להשתמש ב-swap lazy. ניתן למשתמש בעזרו בזכרון הלוקליות ולהביא יחד עם הדף שהיינו צריכים עוד כמה דפים שנמצאים לידיו בזכרון. יכול להיות שנרווח מזה וכי יכול להיות גם שנפסיד מזה. תלוי סיטואציה.
2. צריכים לחשב מראש באיזה גודל עמוד נרצה להשתמש. יתכו יתרונות/חרונות לגדים שונים.
3. TLB REACH מסמל את גודל ה-TLB כפול גודל הדפים. זהה כמוות המידע שנitin להציג מה-TLB.

# הרצאה 10 – File System Interface

קובץ – רצף של זיכרונות שמוחסן בזיכרון המשני.

מאפיינים של קובץ :

- שם – החלק היחיד שמוחסן בשפה קריאה לבני אדם.
- סוג – על מנת שנדע איך לקרוא אותו.
- מספר ייחודי (ID).
- פויינטר למידע עצמו בזיכרון.
- גודל הקובץ.
- סיווג הקובץ – מי יכול לגשת אליו ואיזה פעולות הוא יכול לבצע.
- זמן, תאריך וזיהוי משתמש – נשמר למטרות אבטחה. (מי יצר? מתי שינה? וכו').

פעולות system calls על קבצים :

Create – מחפש מקום, יוצר כניסה במקום הקובץ.

Write – מחפש את מקום הקובץ, מחזיק פוינטר לאיפה צריך לכתוב (כתיבת עצמה מבוצעת על ידי device controller).

Read – מחפש את מקום הקובץ, מחזיק פוינטר לאיפה צריך לקרוא (קריאה עצמה מבוצעת על ידי device controller).

File Seek – מחפש את מקום הקובץ, משנה את מקום הקובץ לערך כלשהו (לא דרוש באמת O/I).

Delete – מצא את הקובץ, שחרר את כל הזיכרונו שלו, מחק את הכניסה לקובץ.

Turncate – כמו delete אבל שומר על תוכנות הקובץ קיימות.

על מנת להלך קובץ פתוח, דרושים כמה דברים כמו פוינטר למקום read/write, סופר לכמות הפעמים שפתחנו את הקובץ, מקום הקובץ בדיסק והרשאות גישה.

כאשר נפתח קובץ, חלק ממנו נשמר ב-cache לקבצים, על מנת לזרז תהליכיים בעתיד אם נרצה לפתוח אותו שוב.

## Open file locking

כאשר מספר תהליכיים רוצים לגשת לקובץ בביטחון אחד, השתמש בנעילה על מנת שלא יהיה בויתות כמו שני תהליכים שכותבים יחד לקובץ, או תהליך שקורא בזמן אחר כותב. יש כמה סוגים נעילה :

Shared lock – כאשר תהליך מושב על קובץ, הוא מותר רק לקרוא על ידי התהליכים הניגשים.

Exclusive lock – כאשר תהליך צריך לכתוב לקובץ, הוא יכול להציג עליו exclusive lock. בכך, אף תהליך אחר לא יוכל להציג עליו נעילה בזמן שהוא כותב. תהליכיים אחרים לא יכולים לקרוא או לכתוב לקובץ בזמן זהה.

## Access Methods

הדרך שבה הקומפיילר ניגש לקבצים שונה משל בני אדם שרצו לקרוא את הקובץ. בדרך כלל הקומפיילר משתמש ב read/write בקרה כזו:

*read next → write next → read next → ... → reset*

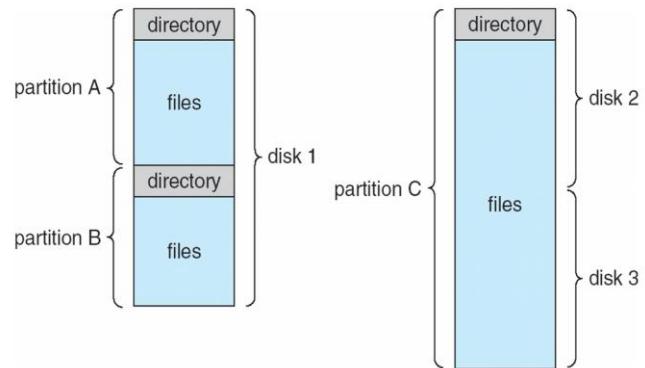
לעומת זאת, דרך שבה תהליכיים או בני אדם ניגשים לקובץ היא בדרך כלל :

```
read n  
write n  
position to n  
    read next  
    write next
```

שתי השורות האחרונות מקדימות את המצביע, בשביל הקריאה/כתיבה הבאים.

## Partitions

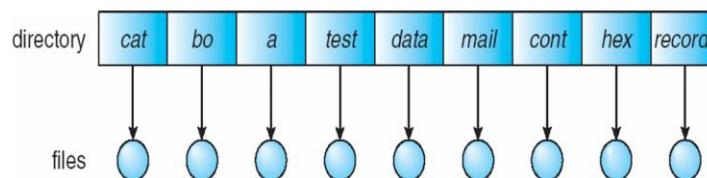
מחלקים את הדיסק לחלקים. ניתן ליצור לכל קובץ partition אחד לפחות לכמה partitions.



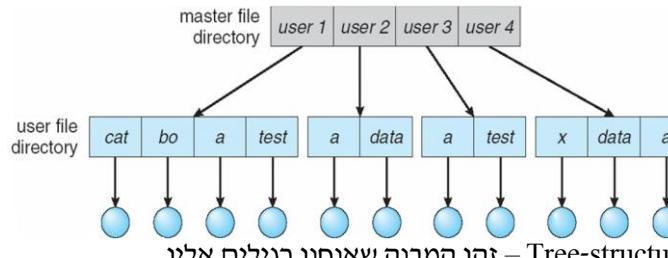
## מבנה הקבצים במערכת

ניתן לסדר את מבנה הקבצים בכמה צורות:

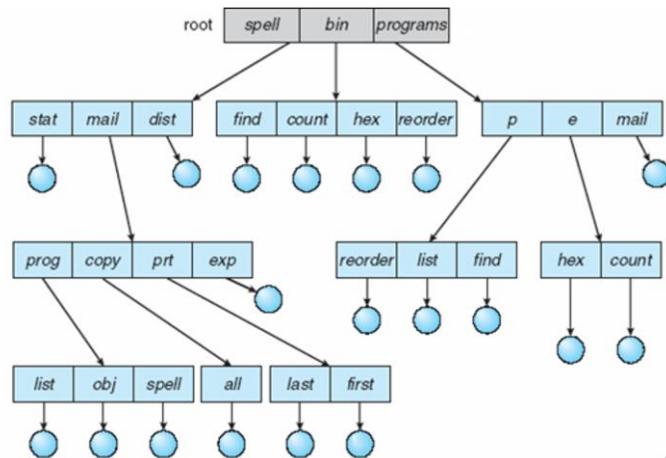
Single-level .1



Two-level .2

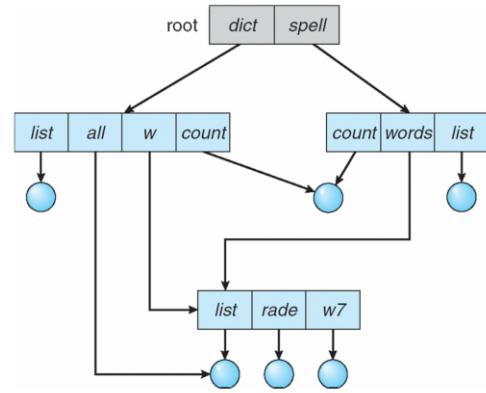


זהו המבנה שאנונו רגילים אליו. Tree-structured .3



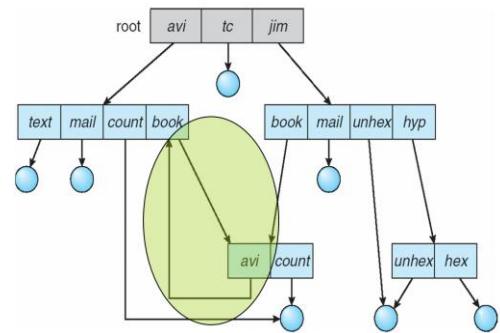
ישנם דרכים נוספות לאorgan את מבנה הקבצים, אבל יש איתן גם בעיות.

מבנה גראף א-צייקלי:



בעיה : אם מוחק את list, תהייה לנו בעיה של dangling pointer .  
פתרון : נחזיק גם פוינטרים אחורנית.

מבנה גראף כללי :



## Mounting

התהיליך של לעלות מערכת קבצים הנמצאת בדיסק כלשהו, לעץ הקבצים נקרא mounting .

## Protection

כאשר נרצה לאפשר לכמה משתמשים לגשת לקובץ במשותף, נctrיך להגן עליו AiCSHe. ישן כמה דרכי לעשנות זאת :

לפי ID user – נסתכל על כל משתמש ונחליט על הרשות והגנתם לפי כל משתמש.

לפי ID group – ניתן לקבוצות מסוימות לגשת, ונחליט על הרשות והגנה לפי כל קבוצה.

בעל הקובץ מחליט מי יכול לקרוא, לכתוב ולהריץ את הקובץ.

הרשאות הגישה מאוחסנת ב-3 ביטים, לפי הסדר *RWX*. למשל 100 – יכול לקרוא אך לא לכתוב או להריץ. 101 – יכול לקרוא ולהריץ, אך לא לכתוב.

## הרצאה 11 – ניהול זיכרון דיסק

הגדרות :

1. סקטור – היחידה הקטנה ביותר של זיכרון שאנו עובדים איתה בדיסק. יכול להיות 512 בתים או 4kb במערכות חדשות יותר.
  2. בלוק – רצף של כמה סקטורים. בפועל כאשר נרצה לכתוב, לקרוא או להציג זיכרון אנחנו ניגש לבlok שלם ולא לסקטור.
  3. קלאסטר – רצף של כמה בלוקים. כאשר אנחנו מזמנים זיכרון אנחנו יכולים להציג קלאסטר שלם במקומות בלוק כדי להשתמש בעקרון הולוקיות.
- דיסקים מיוצגים בתורת מערך חד ממדי של בלוקים לוגיים.

### הציגת זיכרון בדיסק

#### Contiguous Allocation

כל קובץ מאכלס כמה בלוקים ברצף. גישה פשוטה, שמאפשרת random access. בעיה : קבצים לא יוכלים לגודל. פתרון : Extent based system - כל הציגת תהיה במקבץ של כמה בלוקים. כאשר יוצרים קובץ מסוים לו extent אחד או יותר במקרה שהוא גדול.

#### Linked allocation

כל בלוק מיוצג על ידי רשימה מקוושת של בלוקים. בלוקים יכולים להיות מפוזרים בכל מקום בדיסק.  
יתרונות : פשוטות – נדרש רק כתובת התחלת וסוף, אין בזבוז זיכרון.  
חסרונו : random access אין בזבוז.

FAT (file-allocation table) – דרך לגשת רק פעם אחת לדיסק על מנת להגיע לבlok  $m$ , במקום לגשת  $m$  פעמים. עולה את כל המצביעים לזכרון, ואז שנרצה לגשת לבlok כלשהו נטייל על המצביעים כמו שהיינו עושים בדיסק, אבל אנחנו עושים את זה בזיכרון אז זה לא נחשב בזמן ריצה.

#### Indexed allocation

נשתמש בבלוק אחד שיצבע לכל שאר הבלוקים של הקובץ.  
יתרונו : יש random access (אבל צרכיים קודם לזכור בלוק אחד בשבייל זה, לא גישה ממש ישירה). בלי external fragmentation. הערה : ביצירת הקובץ, כל המצביעים יהיו null.

דוגמה מאוד ל-paging. (מצביעים לבлокים שמופיעים לזכרון ומצביעים לנוטונים).  
ניתן גם לשלב בין השיטות. עבור קבצים קטנים ניתן אל המידע מיידית. ככל שהקובץ גדול יותר נשתמש ביותר "שכבות" ונוצר מבנה דומה לעץ עבור מציאת הבלוקים.

#### Free space management

ኒצוץ מערך בגודל כמה הבלוקים שיש לנו. בכל מקום במערך נשים 1 אם הבלוק המתאים תפוס, 0 אם הוא פנוי. על מנת למצוא בלוק פנוי נוצר פשוט לחפש 0.

ונוכל גם להשתמש ברשימה מקוושת של כל הבלוקים הפנויים.

Counting – בבלוק ההתחaltı נשמר כמה רצף של בלוקים פנויים יש אליו, ומיל הבלוק הבא שאינו ברצף.  
.indexed allocation – ניצוץ קובץ שכלו בלוקים ריקים בשיטת ה-

### Disk scheduling

נושא זה מדובר רק על דיסק מגנטיקי.

Access time – מרכיב מהזמן שלוקח להזיז את הראש על פני הטעויות השונות (Seek time), ועוד הזמן שלוקח לסובב את הדיסק.

יש לנו רשימה של מקומות לגשת אליהם. איך נבחר את הסדר?

## הרצאה 12 Threads

עד כה הנחנו כי כל תחילה מריצת תכנית באמצעות thread אחד בכל פעם (בכל רגע מבוצעת פקודה אחת). ניתן להשתמש ב-multithreading כדי למקבל תהליכיים. למשל שרת יכול להשתמש בו על מנת לטפל בכמה בקשנות במקביל.

ה-thread מורכב מ:

Thread id	.1
Program counter	.2
Register set	.3
Stack	.4

הוא חולק עם שאר threads של התהילה את:  
Code section, Data section, Other OS resources such as open files  
לעומת זאת כל אחד בנפרד שומר רגיסטרים ו-stack משלו.

יתרונות ל-threads:

- Responsiveness – מאפשר לתכנית לרוץ גם כאשר חלק ממנה ב-blockes או שמבצע פעולה ארוכה.
- Resource Sharing – threads חולקים את הזיכרון והמשאבים של התהילה.
- Economy – יותר כלכלי לייצור ולבצע context switch ל-threads (מכיוון שהם חולקים את משאבי התהילה).
- Scalability – אותו תחילה יכול לרוץ על מספר מעבדים במקביל באמצעות threads.

### User/Kernel Threads

- Kernel threads – נוצרים ומשובצים על ידי ה-kernel.
- User Threads – נוצרים על ידי ה-user threading library ומשובצים ומונוהלים על ידי הספרייה עצמה:

- כל ה-user threads שייכים לתהילה שיצר אותם.
- ה-kernel לא מודע אליהם.

ה-user threads "יקרים" משמעותית מה-kernel threads.

### Many-to-one

- מספר user threads שטמופים ל-kernel thread אחד.
- כל ניהול threads מבוצע בספריה היושבת ב-user space – יעיל ביותר משיקולי overhead.

חרונות:

- התהילה יכול ייכנס למאובט blocked אם אחד מה-threads קורא ל-system call שהוא blocking.
- אין כאן באנט הרצה במקביל כאשר למעבד יש מספר ליבות (או שיש מספר מעבדים).

-חסר- One-to-one

-חסר- Many-to-many

### Two-level Mode

דומה ל-M:M, אבלאפשר ל-user threads להיות מחוברים ל-kernel threads.