מערכות הפעלה

4

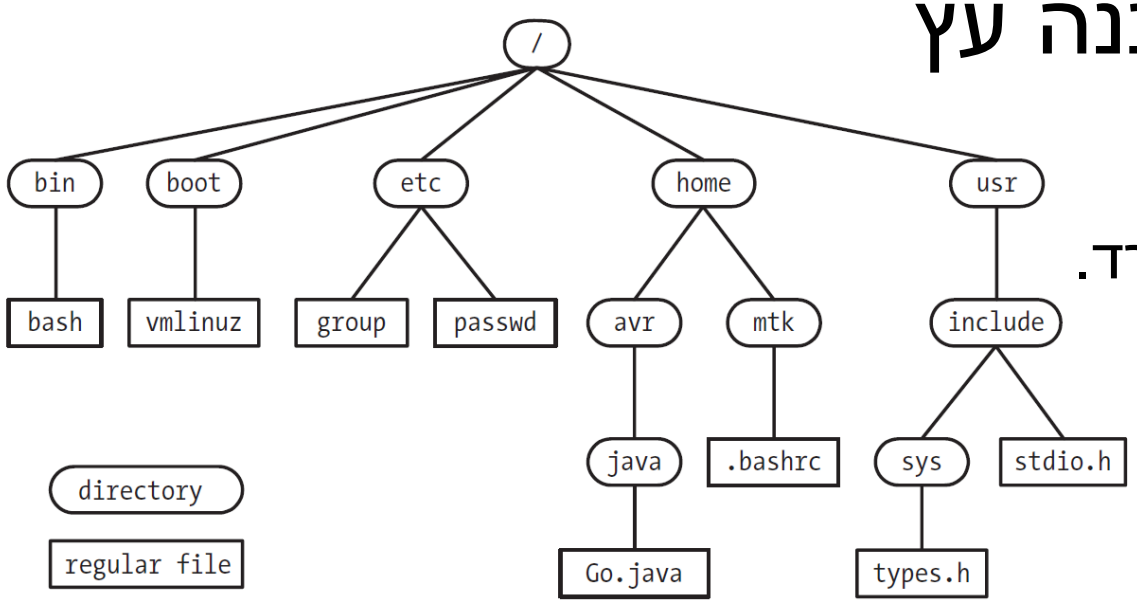
מערכת הקבצים

מערכת קבצים

- מערכת קבצים מכילה אוסף של קבצים ותיקיות.
- מערכת קבצים מאפשרת למשתמש לשמור אינפורמציה בצורה נוחה, בלי הצורך לדעת איך היא נשמרת.
- יוניקס מכיר עשרות סוגי מערכות קבצים, ובהם:

- The traditional **ext2** file system.
- **journaling** file systems, including **ext3** and **ext4**.
- Microsoft’s **FAT** and **NTFS** file systems
- **CD-ROM** file system **ISO 9660**.
- **network** file systems, including Sun’s widely used **NFS** and Microsoft’s **SMB**

תיקיות במבנה עץ



- סידור קבצים בתיקיות:
  - מאפשר לשמור קבצים של משתמשים שונים בנפרד.
  - מאפשר לשמור קבצים של נושאים שונים בנפרד.
- שם של קובץ במבנה עץ:
  - **נתיב מלא** (absolute path name)

    מכיל את כל התיקיות משורש העץ ועד לקובץ.

    ביוניקס, חלקי הנתיב מופרדים על ידי התוו "/", ובווינדוס על ידי התוו "\".

    דוגמה: cat /etc/passwd
  - **נתיב יחסי** (relative path name)

    אם הנתיב לא מתחיל בתוו "/" ("\"), אזי הוא יחסי **למקום הנוכחי** בתיקית הקבצים.
  - המקום הנוכחי (**current working directory**) הוא תכונה של כל תהליך, ניתן לשנות אותו עם הפקודה **cd**.

    דוגמה: cd /etc ; cat passwd

גישה לקבצים

- גישה **סדרתית** לקבצים - sequential access

cat, pico

  - קריאת קובץ לפי סדר הבתים החל מהבית הראשון.
  - אין דילוג או חזרה אחורה.
- גישה **אקראית** לקבצים - random access

lseek

  - קריאת קובץ לא לפי סדר הבתים.
  - קבצים שמכילים מסדי נתונים נקראים בגישה אקראית.

הקצאת בלוקים לקובץ

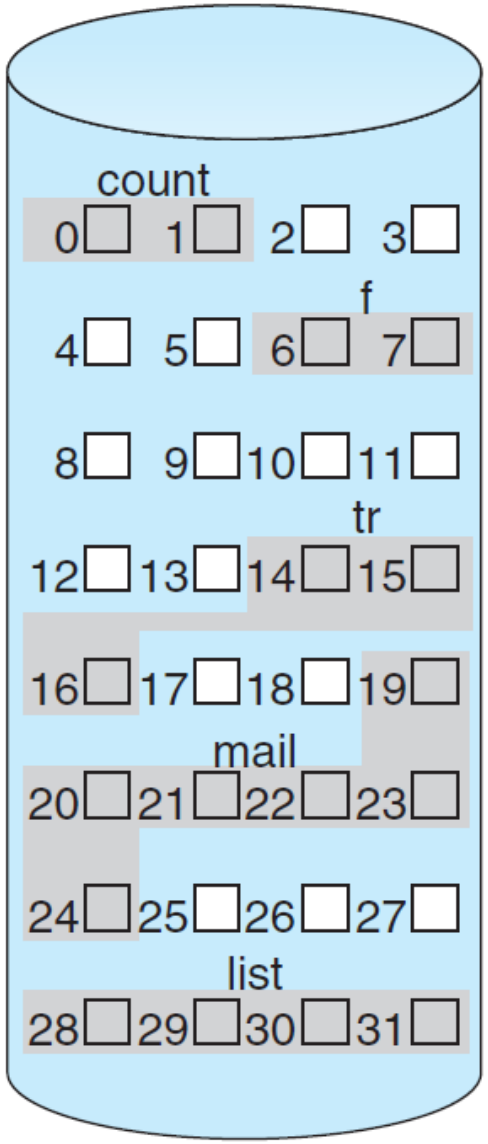
- קובץ מורכב מאוסף של בלוקים.
- מטרת מערכת ההפעלה בהקצאת הבלוקים לקובץ:
  - גישה מהירה לבלוקים של הקובץ.
  - ניצול טוב של הבלוקים בדיסק.
- ישנם כמה שיטות להקצאת בלוקים:
  1. הקצאה רציפה - Contiguous allocation
  2. הקצאה משורשרת - Chained allocation
  3. הקצאה עם מצביעים - Indexed allocation

1. הקצאה רציפה - Contiguous allocation

- כל קובץ נשמר כרצף של בלוקים.
- כדי לגשת לקבצים, מערכת הקבצים מכילה מדריך (directory) עם הפרטים של כל קובץ.
- יתרונות:
  - קריאה וכתיבה הן מהירות, בדרך כלל יש צורך בחיפוש (seek) אחד כדי להגיע לכל הבלוקים של הקובץ.
  - כדי להגיע לכל בלוק בקובץ, יש צורך לשמור במדריך רק את מספר הבלוק הראשון ואת מספר הבלוקים שבקובץ.
  - אם רוצים לדלג קדימה או לחזור אחורה, אפשר למצוא את הכתובת של כל בלוק על ידי הוספת מספר הבלוק לבלוק הראשון.

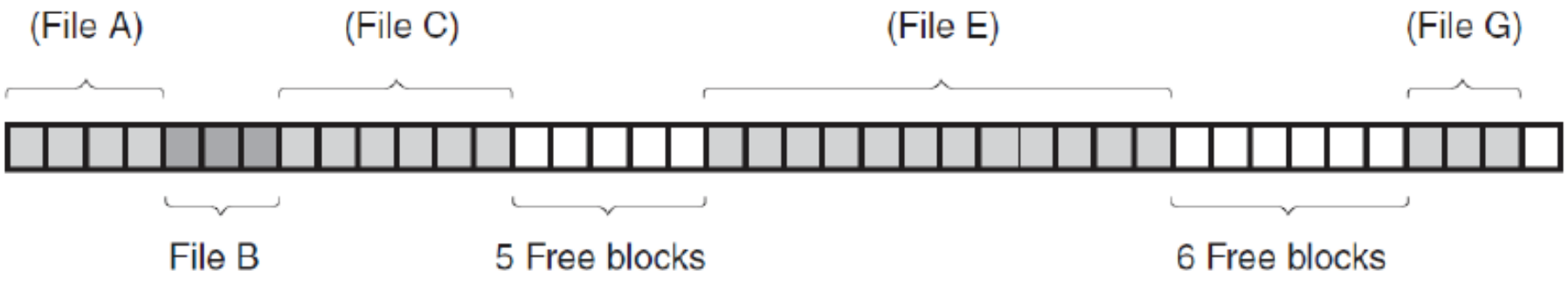
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2



שבירה פנימית ושבירה חיצונית

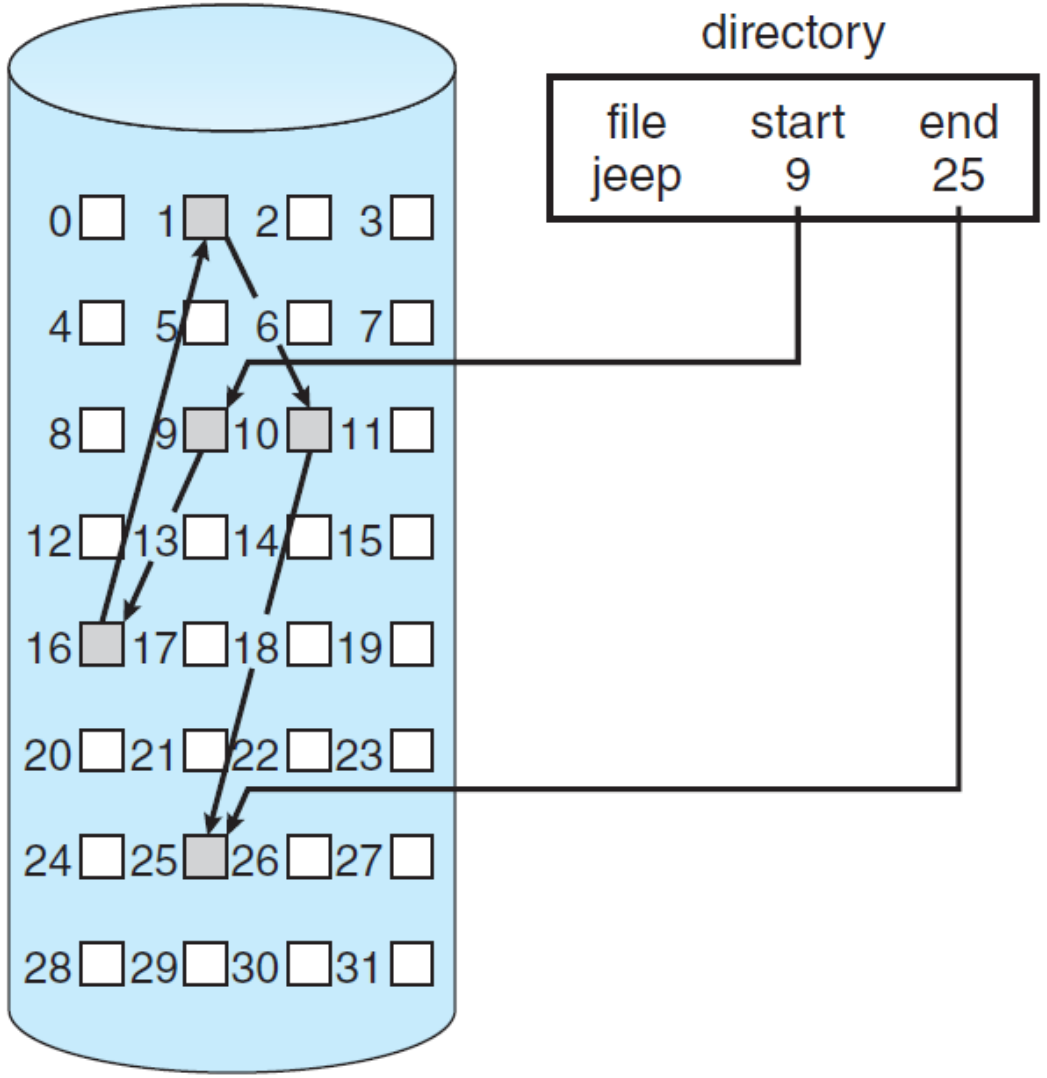
- שבירה פנימית
  - אם גודל הקובץ קטן מכפולה של בלוקים אזי יישארו בסוף הבלוק האחרון בתים ללא שימוש.
  - בממוצע יישאר חצי בלוק.
- שבירה חיצונית
  - לאחר הוספה ומחיקת קבצים, יישארו רווחים קטנים בין ההקצאות.
  - אם דרושה הקצאה רציפה, יתכן שלא תהיה אפשרות להקצות קובץ גדול למרות שצרוף הרווחים מספיק גדול.



הקצאה רציפה

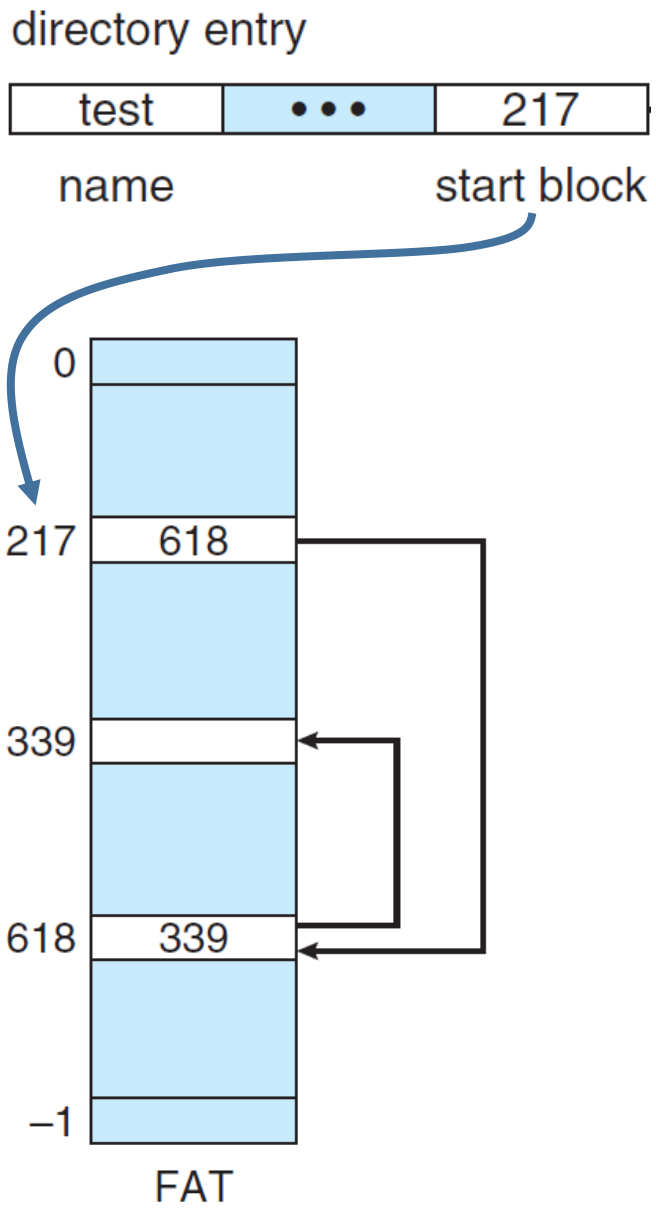
- כדי לבחור את הרווח הפנוי לקובץ נוכל להשתמש באחת מהשיטות הבאות:
  - **First fit**: Choose the first unused contiguous group of blocks of sufficient size from a free block list
  - **Next fit**: Choose the unused group of sufficient size that is closest to the previous allocation for the file to increase locality
  - **Best fit**: Choose the smallest unused group that is of sufficient size
- חסרונות של הקצאה רציפה :
  - מאחר שההקצאה רציפה תתכן **שבירה חיצונית**.
  - כשיוצרים קובץ יש צורך לדעת מראש את גודלו.
- במערכת קבצים של **CD ROM** גודל הקבצים ידוע מראש ואין מחיקה של קבצים - משתמשים בהקצאה רציפה.

2. הקצאה משורשרת - Chained allocation



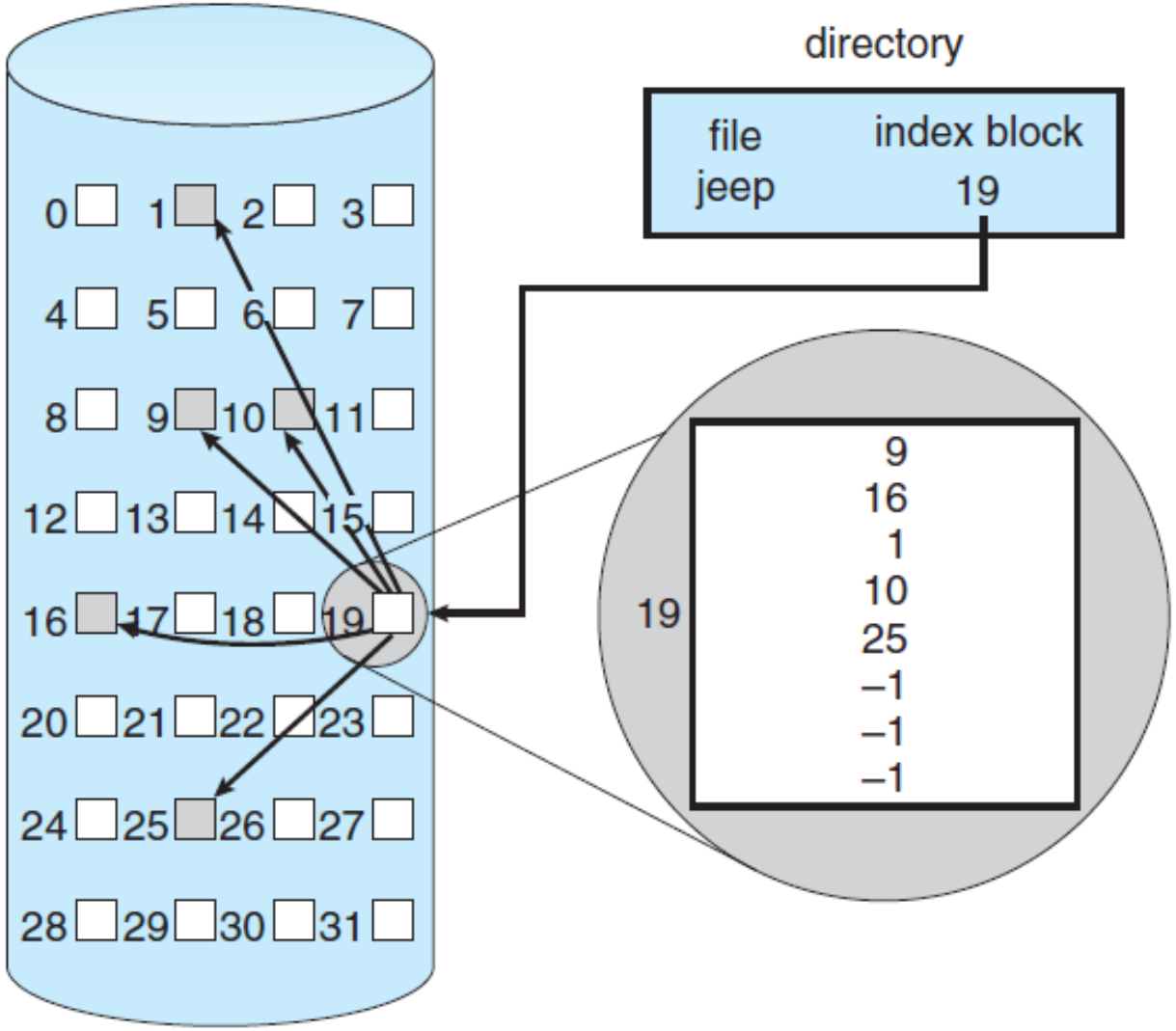
- קובץ הוא רשימה מקושרת של בלוקים.
  - בכל בלוק, הבתים הראשונים הם מצביע לבלוק הבא.
- יתרונות:
  - אין שבירה חיצונית – ניצול כל הבלוקים בדיסק.
  - יש צורך לשמור רק את מספר הבלוק הראשון ואת מספר הבלוק האחרון (או את מספר הבלוקים).
  - קובץ יכול לגדול כל עוד יש בלוקים פנויים בדיסק.
- חסרונות:
  - גישה אקראית היא איטית – יש צורך לקרוא את כל הבלוקים עד לבלוק המבוקש.
  - הנתונים בבלוק אינם בגודל חזקה של 2, כי המצביע תופס חלק מהבלוק.

הקצאה משורשרת עם File Allocation Table (FAT)



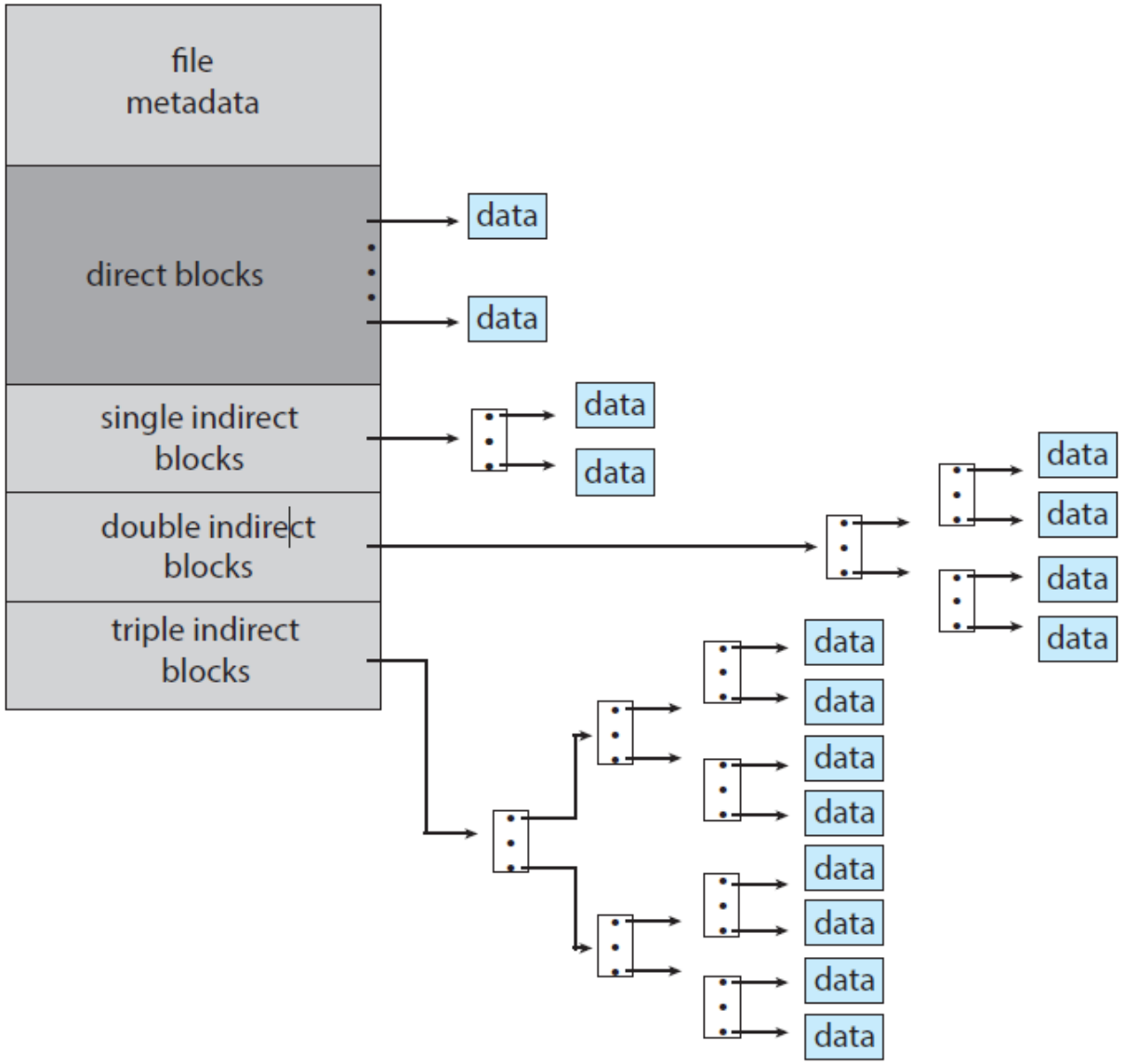
- במערכת הקבצים FAT (File Allocation Table) הקצאת הבלוקים היא בשיטה המשורשרת.
  - אבל המצביעים לא נמצאים בתוך הבלוקים אלא בטבלה נפרדת שנשמרת בתחילת הדיסק.
  - הטבלה מכילה שורה אחת עבור כל בלוק בדיסק.
  - כל שורה מכילה מצביע לבלוק הבא בשרשרת.
  - מספר הביטים של המצביעים בטבלה קובע את מספר הבלוקים המקסימלי (FAT32, FAT16).
- הטבלה מועתקת לזיכרון.
  - גישה אקראית תהיה יותר מהירה מהקצאה משורשרת שבה המצביעים הם בבלוק, והבלוקים יהיו בגודל 512.
- אם הדיסק גדול, הטבלה תתפוס הרבה מקום בזיכרון.
  - FAT32, בלוקים 512, ודיסק בגודל 1T - הטבלה תתפוס 8GB!
  - הפתרון הוא להשתמש בבלוקים לוגיים לדוגמה 4K.

3. הקצאה עם מצביעים - Indexed allocation



- כל המצביעים לבלוקים של הקובץ נמצאים בבלוק אחד בקובץ.
  - לכל קובץ יש את טבלת המצביעים שלו.
  - הטבלה מועתקת לזיכרון כאשר הקובץ בשימוש.
- יתרונות:
  - אין שבירה חיצונית – ניצול כל הבלוקים בדיסק.
  - קובץ יכול לגדול כל עוד יש בלוקים פנויים בדיסק.
  - גישה אקראית מהירה.
- חסרונות:
  - גישה איטית לקובץ יחסית להקצאה רציפה.
  - טבלת המצביעים תופסת מקום בזיכרון.
- מערכות הקבצים של unix ו- ntfs של Windows, משתמשות בשיטה זו.

הקצאה עם מצביעים - Indexed allocation



- אם המצביעים בטבלה לא מספיקים לכל הבלוקים, אפשר להקצות מצביעים שיצביעו לבלוק של מצביעים.
- שאלה:

נניח שגודל בלוק הוא 4K (4096 בתים), ומצביע לבלוק הוא בגודל 32 ביטים (4 בתים).

אם יש 12 מצביעים ישירים, ונניח שיש רק מצביע אחד (לא ישיר) לבלוק של מצביעים, מה גודל הקובץ המקסימלי?

- תשובה:

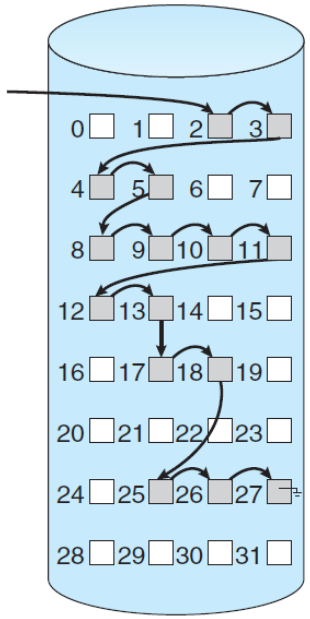
מספר המצביעים בבלוק הוא:

4096 / 4 = 1024

אם כן גודל הקובץ הוא:

(12 + 1024) x 4K = 4144K

ניהול הבלוקים הפנויים



1001101101101100
0110110111110111
1010110110110110
0110110110111011
1110111011101111
1101101010001111
0000111011010111
1011101101101111
1100100011101111
≈ ≈
0111011101110111
1101111101110111

A bitmap

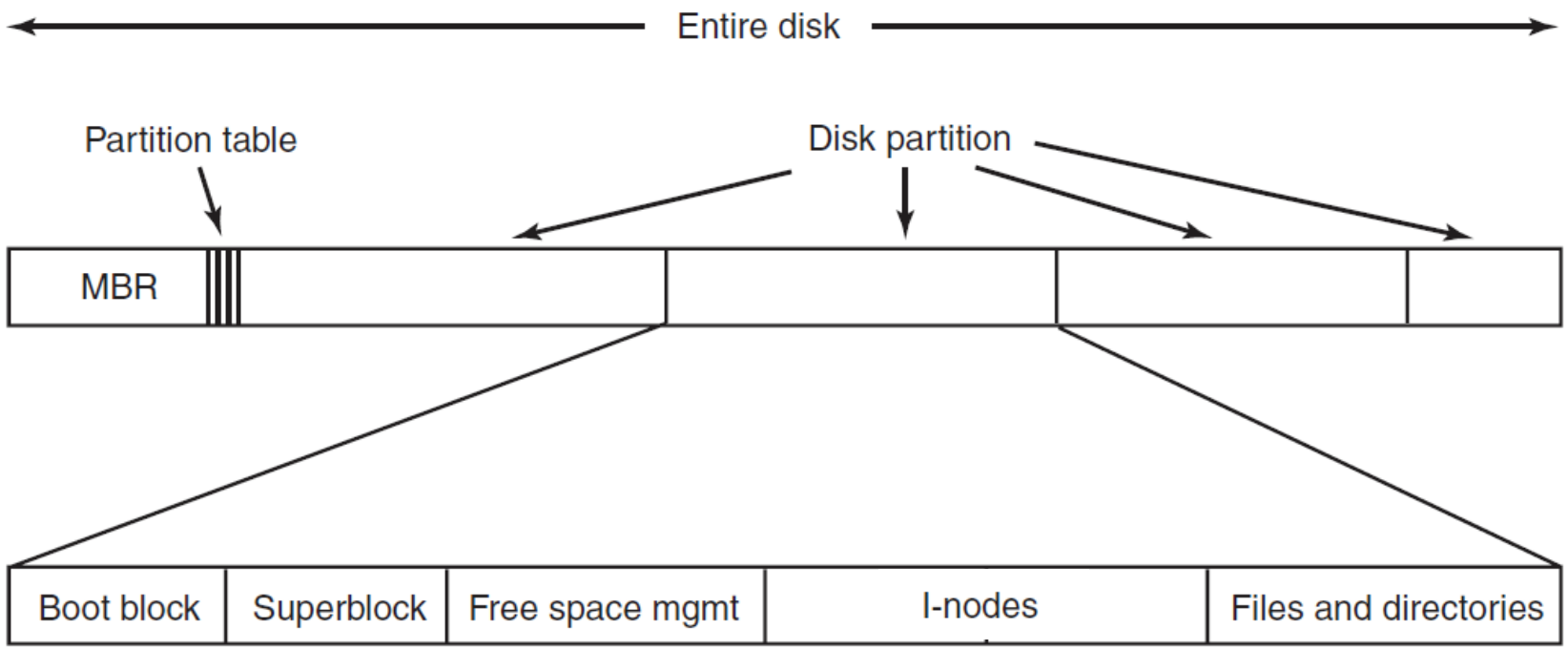
- שרשור של הבלוקים הפנויים ומצביע לבלוק הראשון.
  - כדי למצוא מספר בלוקים פנויים, צריך לקרוא את אותם בלוקים כדי לעבור על השרשור.
  - לא תופס מקום.
- מערך של ביטים, הבלוקים הפנויים מסומנים ב- 1.
  - אין צורך לקרוא את הבלוקים הפנויים.
  - קל למצוא **רצף** של בלוקים פנויים.
  - תופס מקום:

עבור דיסק של 1TB ובלוקים של 4KB, מערך הביטים יתפוס 32MB:

$$2^{40} \ / \ 2^{12} = 2^{28} \text{ bits} = 2^{25} \text{ bytes} = 2^{5} \text{ MB}$$

מערכת הקבצים של יוניקס

- במערכת הקבצים של יוניקס הבלוקים מחולקים לחלקים הבאים:
  - **Boot block** - הבלוק הראשון במחיצה מכיל קוד לאתחול המערכת.
  - **Superblock** - הבלוק הבא מכיל פרטים אודות כל מערכת הקבצים.
  - **bit map** - מערך של ביטים לניהול הבלוקים הפנויים.
  - **inode table** - לכל קובץ ישנה רשומה (inode) שמכילה את הפרטים של אותו קובץ.
  - **Data blocks** - רוב הבלוקים משמשים לאחסון הנתונים שהקבצים מכילים.



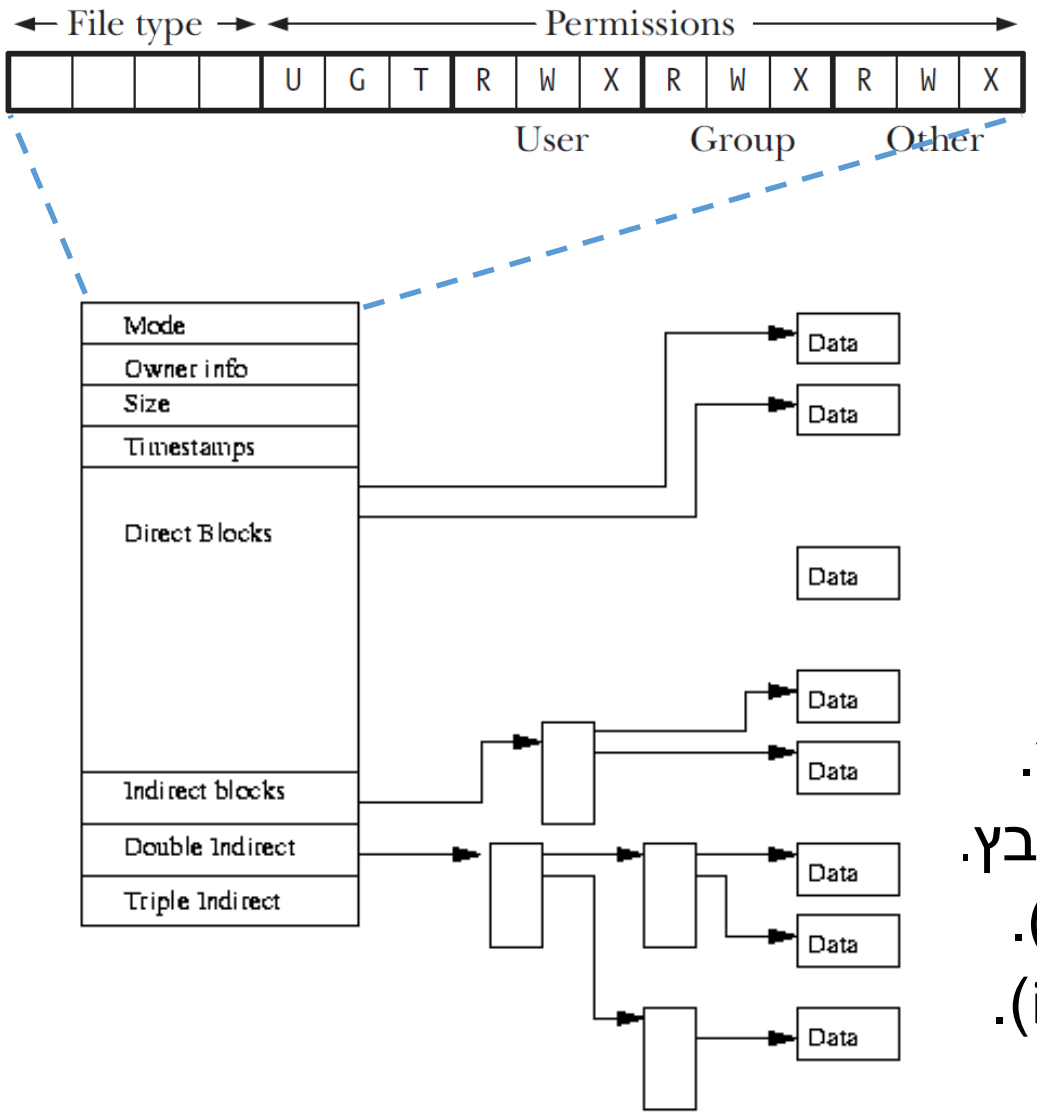
Superblock

- Superblock מכיל פרטים אודות כל מערכת הקבצים:
  - מה סוג מערכת הקבצים (ext2, ext3 או ext4).
  - מה גודל כל בלוק.
  - כמה בלוקים יש במערכת הקבצים.
  - כמה בלוקים פנויים.
  - כמה inodes יש במערכת הקבצים.
  - כמה inodes פנויים.
  - איפה נמצא מערך הביטים של הבלוקים.
  - איפה נמצא מערך הביטים של ה- inodes.

. . .

inode

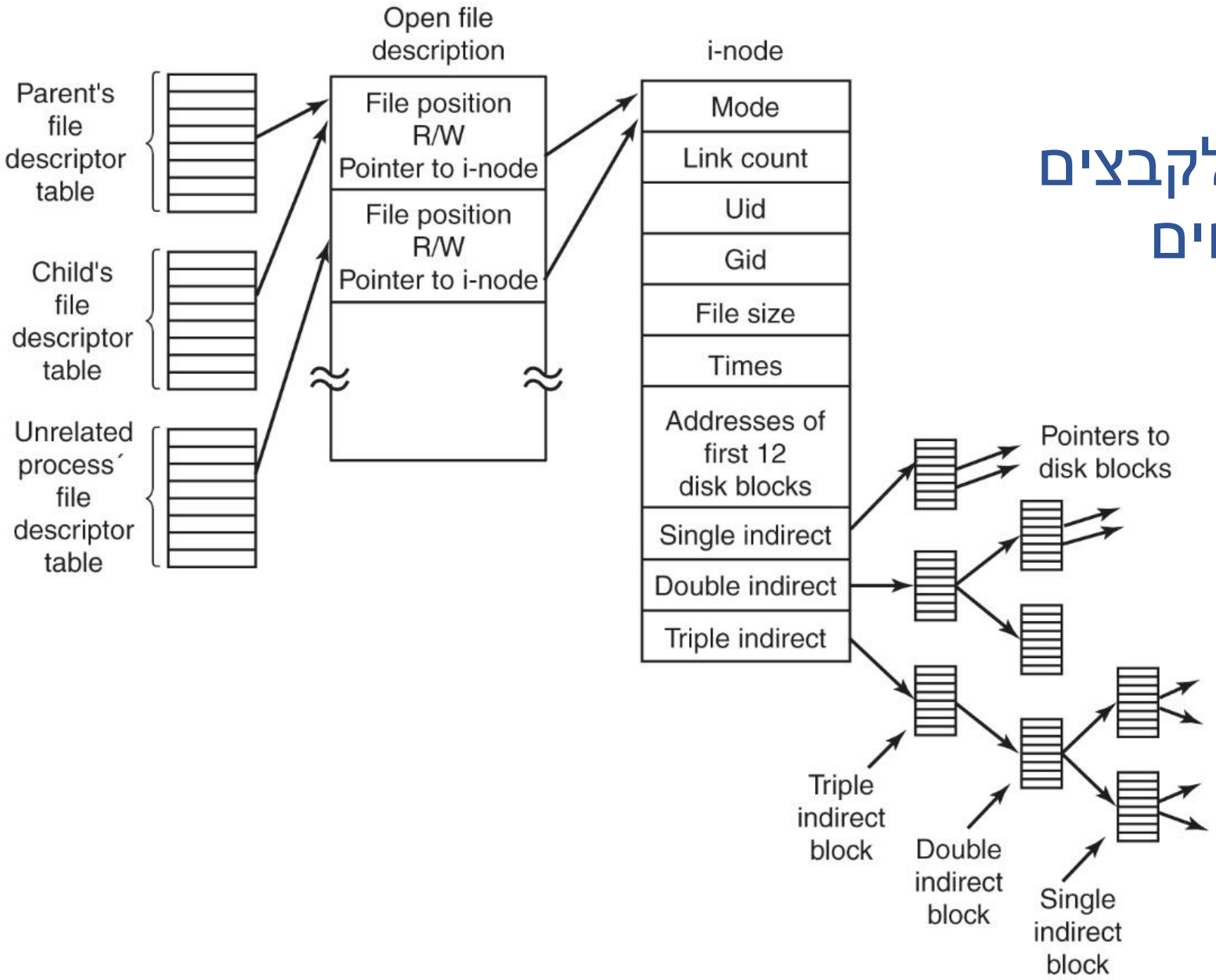
- לכל קובץ ישנה רשומה שמכילה את הפרטים (metadata) של אותו קובץ:
  - mode – סוג הקובץ והרשאות.
  - uid – למי שייך הקובץ.
  - gid – לאיזו קבוצה שייך הקובץ.
  - size – גודל הקובץ בבתים.
  - blocks – כמה בלוקים הקובץ תופס.
  - links_count – מספר הקישורים ל- inode.
  - atime – זמן הגישה האחרון לקובץ.
  - ctime – זמן העדכון האחרון ל- inode.
  - mtime - זמן העדכון האחרון לתוכן הקובץ.
  - block[15] - 15 מצביעים לבלוקים של הקובץ.
  - המצביעים מכילים את מספרי הבלוק של הקובץ.
  - 12 המצביעים הראשונים הם ישירים (direct).
  - 3 המצביעים האחרונים הם עקיפים (indirect).



מצביעים לבלוקים

- 12 המצביעים הראשונים הם ישירים (direct).
  - מכילים את מספרי הבלוק בדיסק של בלוקים 0 עד 11 בקובץ.
  - אם גודל הקובץ אינו גדול מ- 12 בלוקים, נוכל להגיע לכל הבלוקים ללא קריאת בלוק נוסף.
  אם גודל הבלוק הוא 4K נוכל להגיע לכל הבלוקים של קובץ בגודל 48K = 4K * 12 בתים.
- מצביע 13 מכיל כתובת בלוק של מצביעים (indirect).
  - אם גודל הבלוק הוא 4K (4096 בתים) וגודל מצביע הוא 32 ביטים (4 בתים) הבלוק יכיל:
    1024 = 4 / 4096 מצביעים.
    מצביעים אלו יכילו כתובות של בלוקים שמכילים עד 4M = 4K * 1024 בתים.
- מצביע 14 מכיל כתובת בלוק של מצביעים למצביעים (double indirect).
  - לדוגמה, אם גודל הבלוק הוא 4K וגודל מצביע הוא 32 ביטים הבלוקים יכילו:
    $1024^2 = (4096 / 4)^2$ מצביעים.
    מצביעים אלו יכילו כתובות של בלוקים שמכילים עד 4G = 4K * 1024^2 בתים.
- מצביע 15 (triple indirect) יכיל 1024^3מצביעים (עד 4TB בתים)

טבלאות לקבצים פתוחים

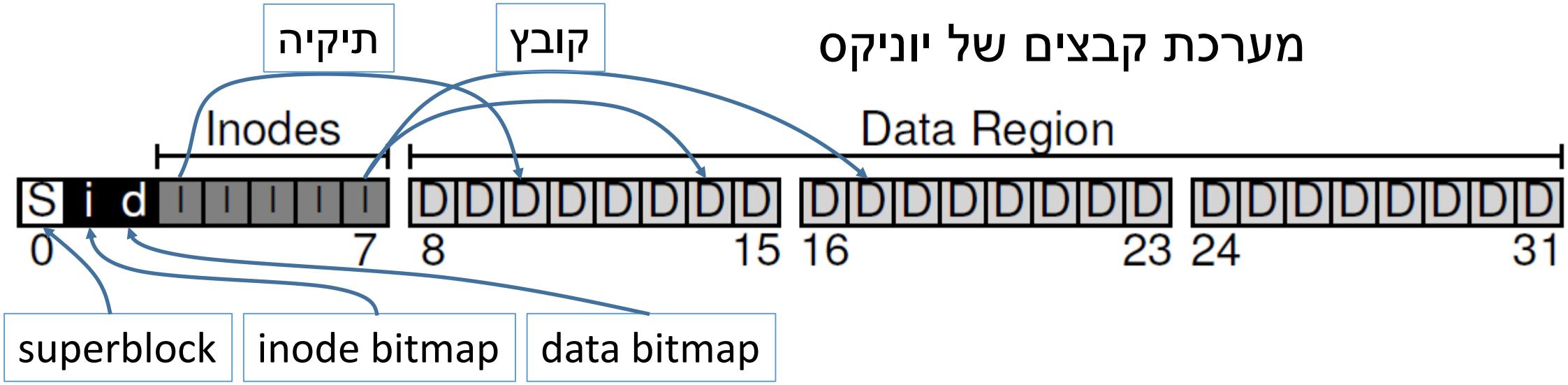


directory - תיקיה

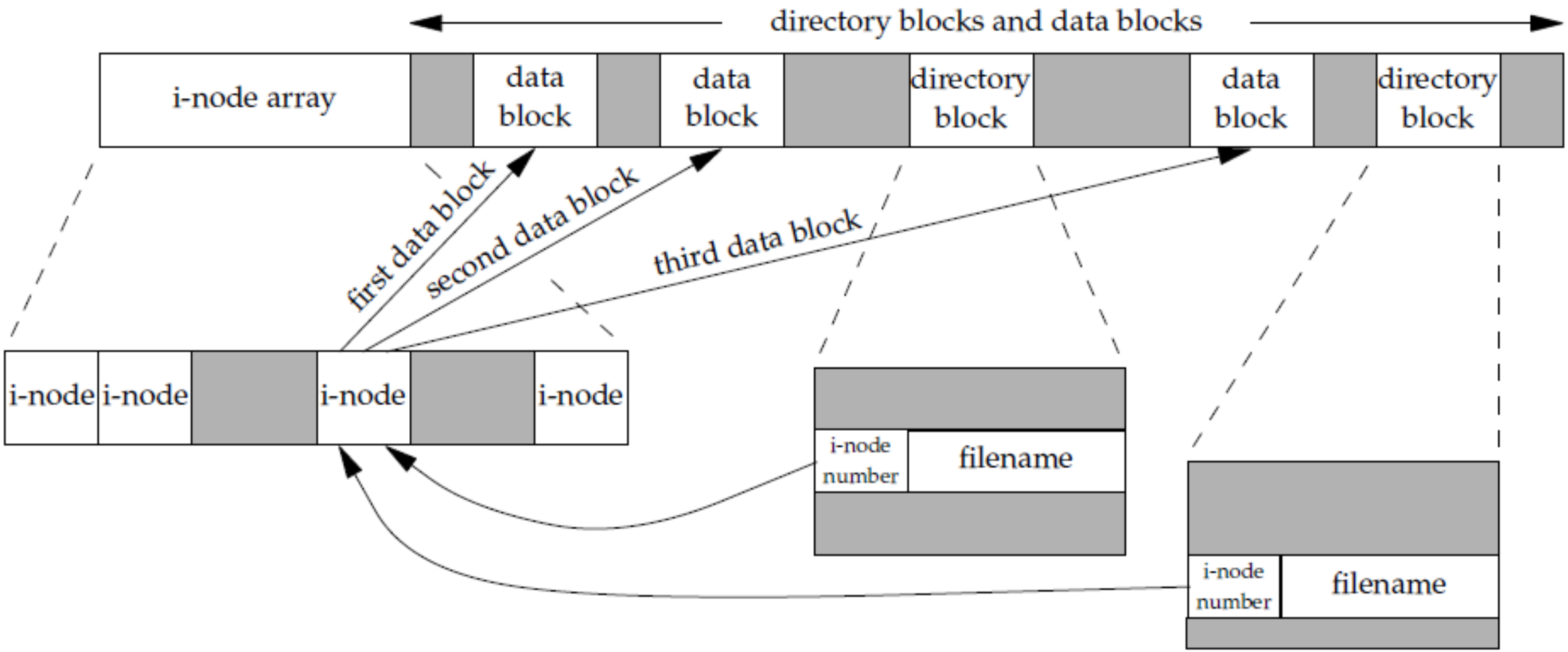
26	.
6	..
64	grants
92	books
60	mbox
81	minix
17	src

- במערכת הקבצים של יוניקס תיקיה היא קובץ שיש לו מבנה מוגדר.
- כמו קובץ רגיל, לתיקיה יש inode שמכיל את הפרטים שלה, תוכן התיקייה נמצא בבלוקים של הנתונים (data).
- מבנה התיקייה הוא רשימה של שמות קבצים וכתובת ה- inode שמכילים את הפרטים של אותם קבצים.

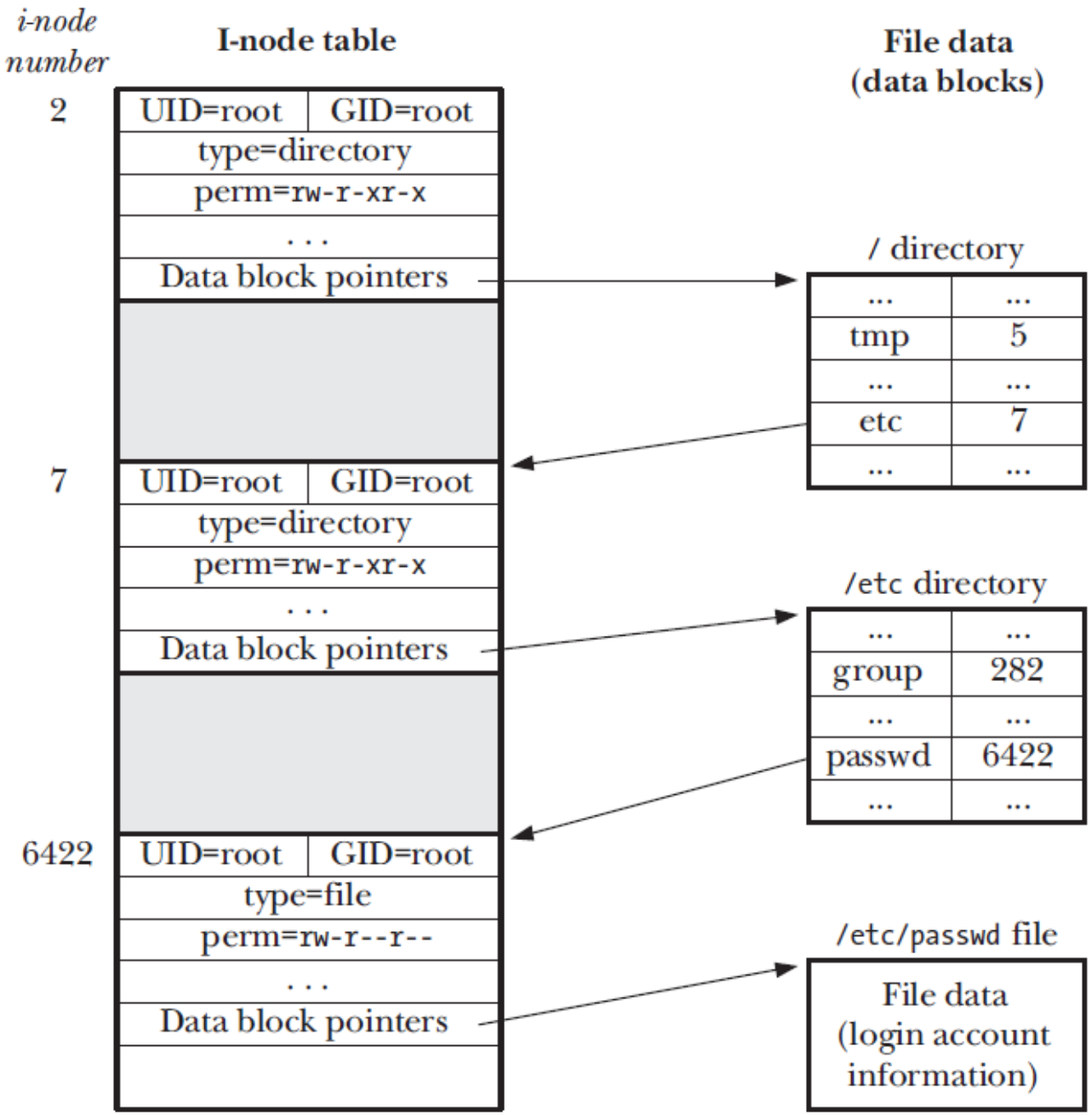
מערכת קבצים של יוניקס



directory - תיקיה

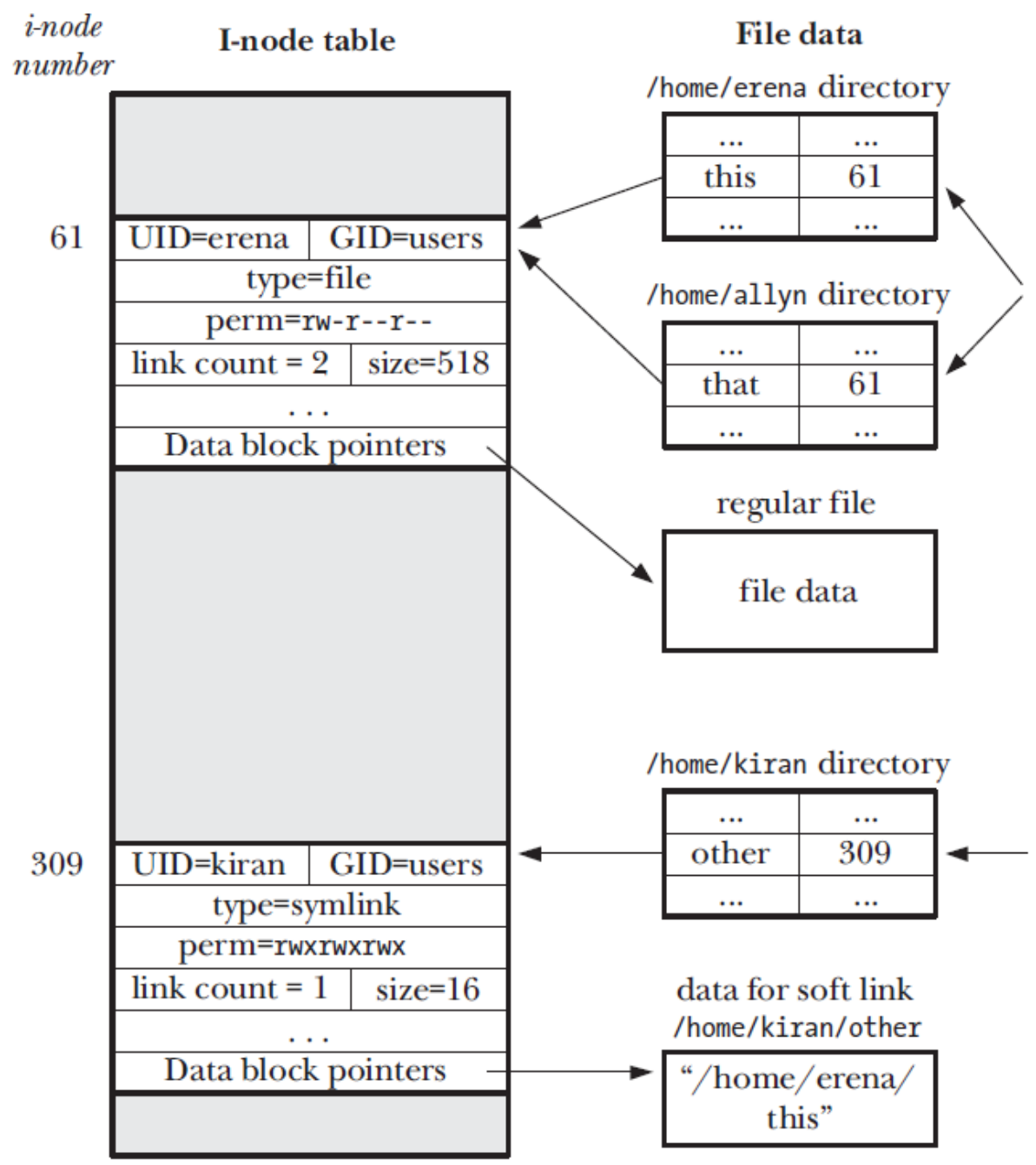


תרגום שם של קובץ למספר inode



- כאשר פותחים קובץ, מעבירים ל- open() את שם הקובץ.
- כדי לקרוא ולכתוב לקובץ יש צורך לדעת את ה- inode שלו.
- כדי לתרגם את השם ל- inode המערכת עוברת על חלקי הנתיב ומתחילה מה- inode של תיקית השורש.(שמקומו ידוע)
- משם עוברת לתיקית השורש עצמה ומחפשת את החלק הראשון בשם.
- וכן הלאה עבור יתר החלקים.
- בציור מתבצע מעבר על הנתיב: /etc/passwd

קישורים לקובץ



- ביוניקס יש שתי אפשרויות ליצור שם נוסף לקובץ:

1. **hard link**

- אם בתיקיות של מערכת קבצים יש שני מצביעים לאותו inode, אזי לקובץ של אותו inode יש שני שמות מסוג hard link.
- קריאת המערכת ליצירת hard link היא link(), והפקודה היא ln.

2. **symbolic link**

- symbolic link הוא קובץ נוסף שיש לו inode נפרד, התוכן של אותו קובץ מכיל נתיב לקובץ אחר.
- לפי סוג הקובץ, המערכת יודעת לפנות לקובץ האחר.
- קריאת המערכת ליצירת symbolic link היא symlink(),והפקודה היא ln -s.
- אפשר ליצור symbolic link בין שתי מערכות קבצים.

פקודות שמציגות מספרי inode

- show file inode number:
```
ls -i /etc/passwd
```
- directory and dot have the same inode:
```
cd /etc
ls -id /etc .
```
- In the root directory, dot and dot-dot have the same inode:
```
cd /
ls -id . ..
```
- Create a hard link:
```
touch myfile
ln myfile myfile-hard
```
- They have the same inode, permissions and size:
```
ls -il myfile myfile-hard
```
- create a symlink:
```
ln -s myfile myfile-sym
readlink myfile-sym
```
- They have different inodes.
```
ls -il myfile myfile-sym
```

הרשאות לקבצים

- בשדה mode שב- inode של כל קובץ, ישנם 9 ביטים עבור הרשאות ועוד 3 ביטים עבור הרשאות מיוחדות.
- 9 הביטים מחולקים ל- 3 סוגי הרשאות:

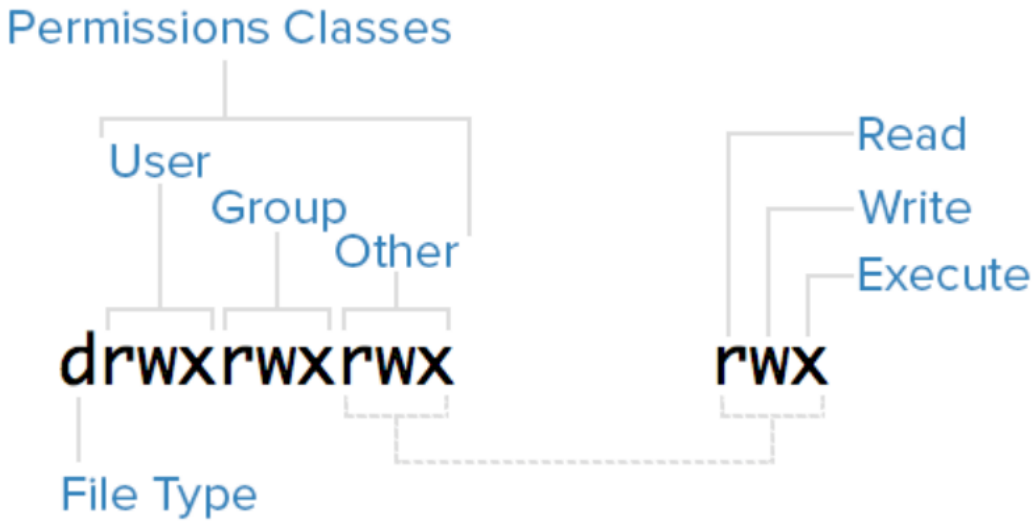
user - read, write, execute

group - read, write, execute

other - read, write, execute

- הפקודה ls -l מציגה את ההרשאות בצורה הבאה (האות הראשונה היא סוג הקובץ).

לדוגמה: -rw-r--r--



סוגי קבצים שכיחים

- **-** Regular file
- **d** Directory
- **l** Symbolic link

הרשאות לקבצים

הרשאה	משמעות עבור קובץ	משמעות עבור תיקיה
r (read)	**read** file contents	**read** directory contents
w (write)	**change** file contents	**create/delete** files in directory
x (execute)	**execute** the file	**search** the directory

משמעות ההרשאות עבור תיקיה:

- הרשאת **קריאה** לתיקייה מאפשרת לקבל את **רשימת הקבצים** שבתיקייה.
- הרשאת **כתיבה** לתיקייה מאפשרת לשנות את תוכן התיקייה – **ליצור קובץ** חדש או **למחוק קובץ** קיים.
- הרשאת **ביצוע** לתיקייה פירושה הרשאת **מעבר בתיקיה**.
  - כדי לפתוח קובץ, ליצור קובץ או למחוק קובץ יש צורך בהרשאת ביצוע (מעבר) בכל התיקיות מהשורש ועד לקובץ.

שינוי הרשאות עם קריאת המערכת chmod()

```
int chmod(const char *pathname, mode_t mode);
```

- הפונקציה chmod() מקבלת שני פרמטרים:
  - נתיב לקובץ.
  - צרוף ביטים של ההרשאות החדשות שאפשר לבטא בצורה אוקטלית, כגון 0644.
  - לחילופין, ניתן לבטא את ההרשאות באמצעות פעולת OR על הסמלים הבאים:

```
S_IRUSR user-read      S_IRGRP group-read    S_IROTH other-read
S_IWUSR user-write     S_IWGRP group-write   S_IWOTH other-write
S_IXUSR user-execute   S_IXGRP group-execute S_IXOTH other-execute
```

דוגמה:

```
chmod ("hello", S_IRUSR | S_IXUSR);
```

- תשנה את ההרשאות של הקובץ `hello` לקריאה וביצוע למשתמש, **ללא** הרשאות נוספות.

שינוי הרשאות עם הפקודה chmod

- בפקודה chmod:
  - כדי לציין "user", "group", "other" , כותבים **u, g, o** .
  - כדי לציין "read", "write", "execute" , כותבים **r, w, x** .
  - כדי להוסיף להרשאה הקודמת כותבים **+** .
  - כדי להפחית מההרשאה הקודמת כותבים **-** .
  - כדי שההרשאה הקודמת תוחלף, כותבים **=** .
- דוגמאות:

```
chmod u+x    myfile
chmod g=rw   myfile
```

הרשאות לפני: `-rwxr-xr-x  file.txt`

```
chmod o=r    file.txt
```

הרשאות אחרי: `-rwxr-xr--  file.txt`

שינוי הרשאות עם הפקודה chmod ומספרים אוקטליים

כל ספרה אוקטלית מתפרשת כסכום הספרות 0, 1, 2, 4:

0 (בינרי 000) – ללא הרשאות.

1 (בינרי 001) – הרשאת ביצוע.

2 (בינרי 010) – הרשאת כתיבה.

4 (בינרי 100) – הרשאת קריאה.

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

במקום לכתוב:

```
chmod u=rwx,g=rx,o=r myfile
```

אפשר לכתוב:

```
chmod 754 myfile
```

7	5	4
111	101	100
drwx	rwx	rwx

הספרה 7 היא צרוף של 4+2+1 (read, write, execute) (בינרי 111)

הספרה 5 היא צרוף של 4+0+1 (read, execute) (בינרי 101)

הספרה 4 היא צרוף של 4+0+0 (read) (בינרי 100)

מסכה להרשאות - umask

- לכל תהליך יש תכונה umask (שנמצאת ב- PCB).
  - umask היא מסכה או מסננת שמונעת חלק מההרשאות.
- כאשר תהליך יוצר קובץ (open(...O_CREAT...)) הוא נותן הרשאות לקובץ החדש.
  - ההרשאות שהקובץ יקבל בפועל הן לאחר הסינון של umask.
- כל ביט שדלוק ב- umask יכבה את ההרשאה המתאימה, כלומר:

```
permissions = mode & ~umask
```

- תהליך יכול לשנות את ה- umask שלו על ידי קריאת המערכת umask (הערך החדש מוחזר):

```
mode_t umask(mode_t mask)
```

- אפשר לשנות את ה- umask של ה- shell על ידי הפקודה **המובנית** umask, ה- shell יוריש את התכונה לתהליכים שהוא מריץ:

`umask 022` מונע מהקבוצה ומאחרים לכתוב.

`umask 002` מונע מאחרים לכתוב.

`umask 077` מונע מהקבוצה ומאחרים לקרוא לכתוב ולבצע.

קובץ המשתמשים /etc/passwd

- קובץ המשתמשים /etc/passwd מכיל שורה עבור כל משתמש.
- כל שורה מכילה את השדות הבאים מופרדים על ידי נקודתיים:

`name`	user name
`passwd`	encrypted password
`uid`	numerical user ID
`gid`	numerical group ID
`gecos`	comment field
`dir`	working directory (home)
`shell`	shell (user program)

- דוגמה:

```
root:x:0:0:root:/root:/bin/bash
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

- השורה הראשונה היא עבור המשתמש `root` שמספרו 0 (לפי מספר זה מערכת ההפעלה נותנת לו הרשאות מיוחדות).
- במערכות מודרניות העבירו את הסיסמאות לקובץ אחר והסיסמה מסומנת ב- x.

קובץ הסיסמאות /etc/shadow

- הסיסמאות לא נשמרות כפי שהוקלדו (clear text) אלא לאחר הצפנה (encrypted).
  - לכן בעבר לא חששו שהקובץ שהכיל את הסיסמאות (/etc/passwd) יהיה קריא לכולם.
- אבל משתמשים נוטים לבחור סיסמאות חלשות וזה מאפשר לפורצים לנחש את הסיסמה על ידי הרבה ניסיונות.
- כדי להקשות על הפורצים, העבירו את הסיסמאות לקובץ נפרד (/etc/shadow), קובץ זה קריא רק ל- root.
- הקובץ המקורי (/etc/passwd) נשאר קריא לכולם, כי כל משתמש צריך לקרוא אותו.
  - לדוגמה, כדי לתרגם מספרי משתמש לשמות משתמש.

הרשאת set-user-id

- משתמש יכול להריץ קובץ תכנית ששייך למשתמש אחר, אם המשתמש האחר נתן לו הרשאה.
  - ההרשאות של התהליך הרץ (גישה לקבצים) נקבעות לפי המשתמש שמריץ את התכנית.
- לפעמים יש צורך שההרשאות של התהליך הרץ יהיו לפי מי שקובץ התכנית שייך לו.
  - לכן, לכל תהליך יש (ב- PCB) שתי זהויות של המשתמש שמריץ את התהליך:

    real user ID - המשתמש שמריץ את התכנית.

    effective user ID - המשתמש שלפיו **נקבעות** ההרשאות של התהליך.
- בדרך כלל effective user ID הוא העתק של real user ID ושניהם נקבעים לפי מי שמריץ.
- אבל אם לקובץ הריצה יש הרשאה מיוחדת של set-user-id אזי effective user ID נקבע לפי מי שקובץ הריצה שייך לו.
- דוגמה לצורך בהרשאת set-user-id:
- התכנית passwd משמשת לשינוי הסיסמה בקובץ הסיסמאות shadow.
- למשתמש רגיל אין גישה לקובץ זה - כדי לערוך את קובץ הסיסמאות צריך להיות root.
- הרשאת set-user-id לקובץ התכנית הופכת את המשתמש הרגיל ל- root למשך ריצת התכנית.

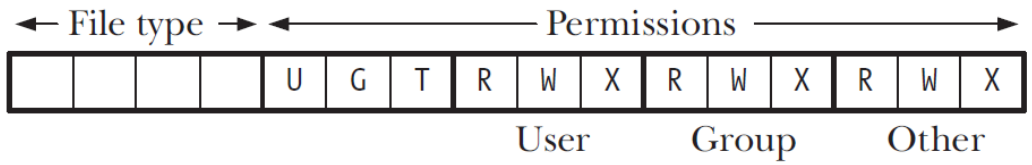
```
-rwsr-xr-x   root root   /usr/bin/passwd
```

הרשאת sticky bit

- הרשאה מיוחדת נוספת נקראת sticky bit ומתייחסת רק לתיקיות.
- ההרשאה נועדה לפתור את הבעיה הבאה:
- כפי שראינו, כדי למחוק קובץ צריך הרשאת כתיבה וביצוע לתיקיה שמכילה את הקובץ.
  - אין צורך בהרשאות כלשהן לקובץ הנמחק.
- ישנם תיקיות שמשמשות את כל המשתמשים.
  - לדוגמה, התיקיה /tmp שמשמשת לשמירת קבצים זמניים.
  - כדי שכל משתמש יוכל לשמור קבצים בתיקיה, יש לה הרשאת כתיבה וביצוע לכולם.
  - אבל זה מאפשר למשתמש למחוק קבצים של משתמש אחר.
- אם מוסיפים לתיקיה הרשאת sticky bit אזי משתמש יוכל למחוק רק קבצים השייכים לו.

```
drwxrwxrwt   root root   /tmp
```

סוגי קבצים



File_type	Description
0	
1	Regular file – מכיל רצף של בתים שיכולים להיות תווי טקסט או בינריים
2	Directory – מכיל שמות קבצים ואת מספר ה- inode שלהם
	סוגי הקבצים הבאים מכילים inode ואינם מכילים בלוקים של נתונים:
3	Character device – קובץ שמיצג התקן שהקריאה והכתיבה אליו היא בבתים
4	Block device - קובץ שמיצג התקן שהקריאה והכתיבה אליו היא בבלוקים
5	Named pipe - pipe קובץ שמיצג תקשורת באמצעות
6	Socket - socket קובץ שמיצג תקשורת מקומית באמצעות
7	Symbolic link - מכיל קישור לקובץ אחר (אם הקישור קצר הוא כלול ב- inode)

קובץ הֶתקן - Device File

- כדי לקרוא ולכתוב להתקן, משתמשים באותם קריאות מערכת כמו בקריאה וכתיבה של קבצים רגילים, ה- device driver של כל התקן מכיל פונקציות שמבצעות open, read, write ועוד.
- קבצי ההתקן נמצאים בתיקיה `/dev` .
  - הקבצים המתחילים ב- b הם קבצי התקן מסוג בלוק, הקריאה והכתיבה היא ביחידות של בלוקים ויש גישה אקראית לבלוקים.
  - הקבצים המתחילים ב- c הם קבצי התקן מסוג תווים, הקריאה והכתיבה היא ביחידות של תווים.
  - לכל קובץ התקן יש שני מספרים, השמאלי מזהה את ההתקן (driver), הימני מזהה התקן מסוים, לדוגמה אם כמה דיסקים מחוברים לאותו כרטיס.

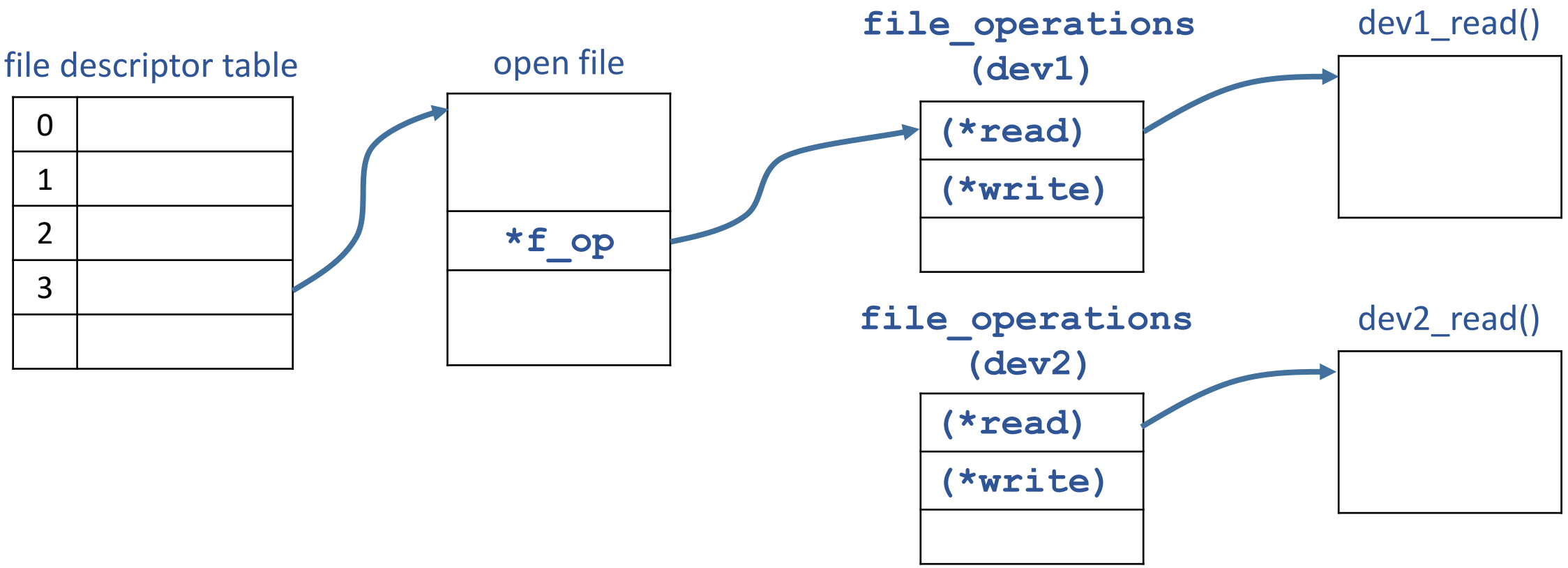
```
ls -l /dev
brw-rw----   ...   8,1   sda1
crw--w----   ...   4,10  tty10
...
```

קובץ הֶתקן - Device File

- ראינו שקריאה וכתיבה של קבצים ושל התקנים שונים מתבצעת על ידי פונקציות שיש להם אותו שם (open, read, write), אבל לכל סוג קובץ ולכל התקן יש קוד שונה לביצוע הפונקציות.
- הפונקציות אם כן צריכות להיות פולימורפיות, אבל הקרנל כתוב בשפת C.
  - אפשר ליצור פולימורפיזם ב- C עם מצביעים לפונקציות.
  - אם משנים את ההצבעה, שם הפונקציה נשאר אבל הפונקציה תבצע קוד שונה.
- לאובייקט "קובץ פתוח" (open file) יש משתנה שמצביע לטבלת המצביעים של הפונקציות.
- בזמן פתיחת הקובץ, הקרנל מכוון את המצביע לטבלה של ה- driver של אותו התקן.

```
struct file {
    struct file_operations *f_op;
    ...
struct file_operations {
    off_t (*lseek) (struct file *, off_t, int);
    ssize_t (*read) (struct file *, char *, size_t, off_t *);
    ssize_t (*write) (struct file *, char *, size_t, off_t *);
    ...
// to invoke read: file->f_op->read(...);
```

Device File - קובץ הֶתקן



מערכת הקבצים הווירטואלית של לינוקס

- ראינו שהפעולות (קריאה, כתיבה ועוד) בקובץ פתוח הן פולימורפיות.

  לכן הן פועלות **בהתקנים** שונים.

- מסיבה זו הן גם פועלות **במערכות קבצים** שונות (ולכן מערכת הקבצים היא וירטואלית).

  לדוגמה:

  הפקודה cp מעתיקה מכל מערכת קבצים לכל מערכת קבצים.

  אפשר להעתיק קובץ מהתקן USB שמכיל מערכת FAT לדיסק שמכיל מערכת ext3.

```
cp /usb/test /dir/test
```

התוכנית cp מבצעת פונקציות read ו- write, מערכת ההפעלה דואגת שתתבצע הפונקציה המתאימה **למערכת הקבצים**.

התקנים לוגיים - logical devices

- בתיקיית ההתקנים ישנם התקנים שאינם פיזיים:

- מה שנכתב להתקן נזרק, קריאה מההתקן מחזירה קובץ ריק – **/dev/null**

```
verbose_command > /dev/null
cp /dev/null myfile // empty myfile
```

- קריאה מההתקן מחזירה תווי null (`0x00`) - **/dev/zero**

```
hexdump -cv /dev/zero
```

משמש לאיפוס זיכרון.

- קריאה מההתקן מחזירה תווים אקראיים - **/dev/random**

```
cat /dev/urandom
```

מערכת הקבצים proc/

- ישנם בלינוקס מערכות קבצים וירטואליות, הן לא נשמרות בדיסק אבל נראות למשתמש במבנה של קבצים ותיקיות.
- כדי לקרוא ולכתוב למערכת קבצים וירטואלית, משתמשים בפקודות קריאה וכתיבה של קובץ רגיל (הקבצים נוצרים תוך כדי קריאה).
- בתיקיה proc/ תלויה מערכת קבצים ווירטואלית שמאפשרת גישה נוחה למידע על מערכת ההפעלה.
- כל תהליך שרץ, מיוצג ב- proc/ באמצעות תיקיה ששמה כמספר התהליך.
  - בתוך תיקיה זו ישנם קבצים עם מידע אודות התהליך.
- דוגמה, הצג ושנה את הערך pid_max :

```
# cat /proc/sys/kernel/pid_max
# echo 100000 > /proc/sys/kernel/pid_max
```

קריאת המערכת stat()

```
int stat(const char * pathname, struct stat * buf );
```

- הפונקציה stat מחזירה מבנה שנקרא stat ומכיל מידע אודות הקובץ.
- סוג הקובץ נמצא בשדה `st_mode` של המבנה.
- לנוחיות התכניתן ישנם פונקציות עזר (macro) לסוג הקובץ:

```
S_ISREG()    regular file
S_ISDIR()    directory file
S_ISCHR()    character special file
S_ISBLK()    block special file
S_ISFIFO()   pipe or FIFO
S_ISLNK()    symbolic link
S_ISSOCK()   socket
```

struct stat

```
struct stat {
   mode_t st_mode; /* file type & permissions */
   ino_t st_ino; /* i-node number */
   dev_t st_dev; /* device number */
   dev_t st_rdev; /* device this file represents, dev file */
   nlink_t st_nlink; /* number of hard links */
   uid_t st_uid; /* user ID of owner */
   gid_t st_gid; /* group ID of owner */
   off_t st_size; /* size in bytes, for regular files */
   struct timespec st_atim; /* time of last access */
   struct timespec st_mtim; /* time of last modification */
   struct timespec st_ctim; /* time of last status change */
   blksize_t st_blksize; /* best I/O block size */
   blkcnt_t st_blocks; /* number of 512B blocks allocated */
};
```

הדפסת סוג הקובץ לפי stat()

```
int main(int argc, char *argv[]) {
   int i;  struct stat buf; char *ptr;
   for (i = 1; i < argc; i++) {
      printf("%s: ", argv[i]);
// if a soft link, lstat returns stat about the link itself
      if (lstat(argv[i], &buf) < 0) {
         printf("lstat error"); continue; }
      if (S_ISREG(buf.st_mode)) ptr = "regular";
      else if (S_ISDIR(buf.st_mode)) ptr = "directory";
      else if (S_ISCHR(buf.st_mode)) ptr = "character special";
      else if (S_ISBLK(buf.st_mode)) ptr = "block special";
      else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
      else if (S_ISLNK(buf.st_mode)) ptr = "symbolic link";
      else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
      else ptr = "* unknown *";
      printf("%s\n", ptr);
   }
} // ./mystat file.txt a.out mydir mylink ...
```

stat.c

זמנים של קובץ

- לכל קובץ נשמרים שלושה זמנים:

Field	Description	Example	ls option
atim	access time of file	read	-u
mtim	modification time of file	write	default
ctim	change time of attributes	chmod, chown	-c

- access – זמן הגישה האחרון לקובץ.
- modification – זמן העדכון האחרון לתוכן הקובץ.
- change – זמן העדכון האחרון ל- inode (תכונות הקובץ).

יצירת ומחיקת תיקיה

- הפונקציה mkdir יוצרת תיקיה ריקה:

```
int mkdir(const char *pathname, mode_t mode);
```

- דוגמה:

```
mkdir("mydir", S_IRUSR | S_IWUSR | S_IXUSR)
```

- כשהקרנל יוצר תיקיה הוא יוצר בתוכה שתי תיקיות שנקראות . (נקודה) ו- .. (נקודה נקודה)
  - התיקיה . היא שם נוסף לתיקיה הנוכחית.
  - התיקיה .. היא שם נוסף לתיקיה שמעל לתיקיה הנוכחית.
- הפונקציה rmdir מוחקת תיקיה ריקה:

```
int rmdir(const char *pathname);
```

כתיבה וקריאה של תיקיה

- כדי שמערכת הקבצים לא תשתבש, כתיבה וקריאה של תיקיה מתבצעות באמצעות הקרנל.
- כתיבה לתיקיה מתבצעת כאשר יוצרים או מוחקים קובץ בתיקיה.
- כדי לקרוא תוכן תיקיה (רשימת הקבצים) משתמשים בקריאות המערכת:

```
DIR * opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
```

- לתוך המבנה **DIR** קוראים את תוכן התיקיה (בדומה למבנה **FILE**).
- לאחר מכן קוראים לתוך מבנה **dirent** את כל אחד מקבצי התיקיה.
  - המבנה **dirent** מכיל:

```
ino_t d_ino;        /* inode number */
char d_name[];      /* filename */
```

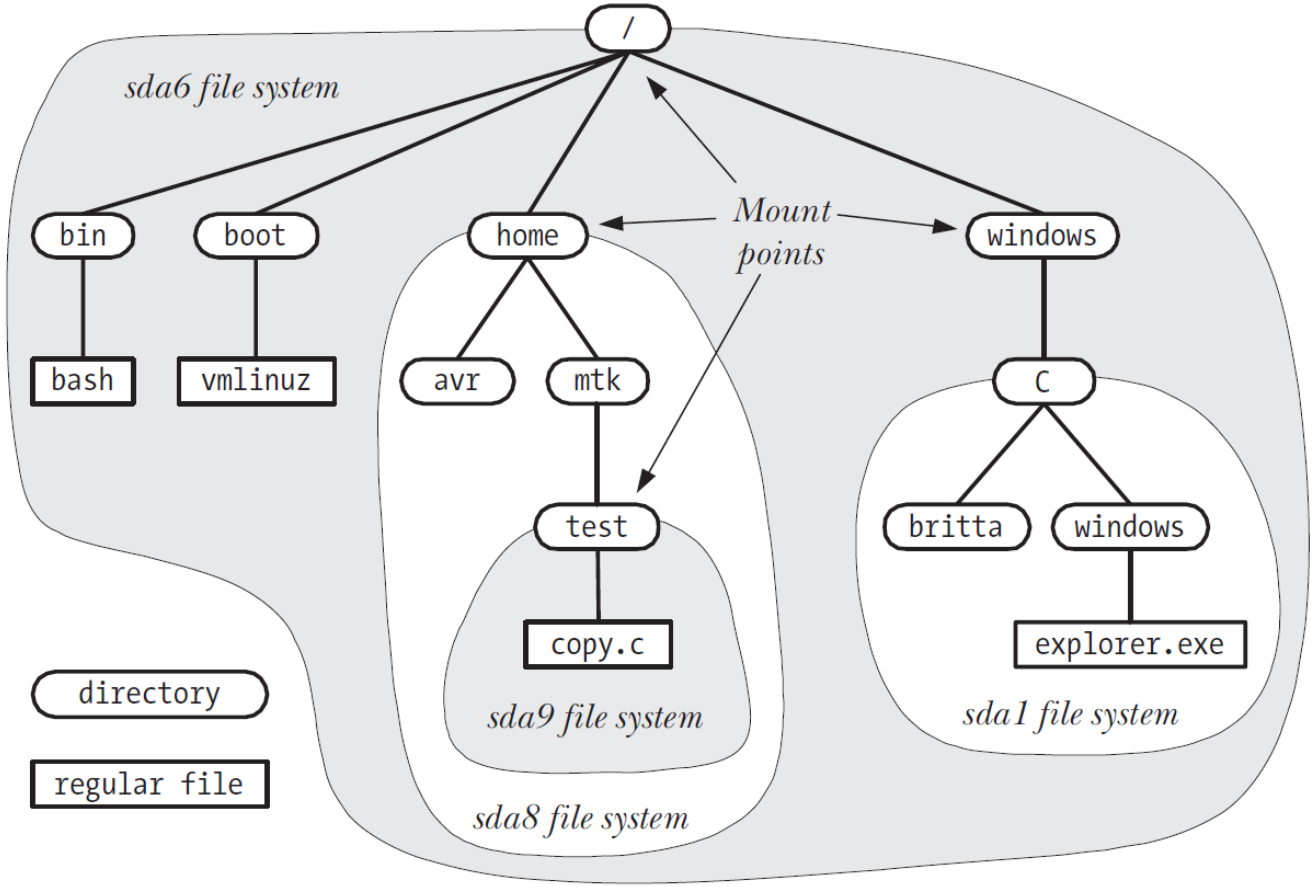
רשימת הקבצים בתיקיה

```
int main(int argc, char *argv[])  {
   DIR  *dp;
   struct dirent *dirp;
   if ((dp = opendir(argv[1])) == NULL) {
      printf("can't open %s", argv[1]);
      exit(1);
   }
   while ((dirp = readdir(dp)) != NULL) {
      if (strcmp(dirp->d_name, ".") == 0
          || strcmp(dirp->d_name, "..") == 0)
          continue;
      printf("%lu\t", dirp->d_ino);
      printf("%s\n", dirp->d_name);
   }
   closedir(dp);
}
```

הוספת מערכת קבצים לעץ הקבצים - mounting

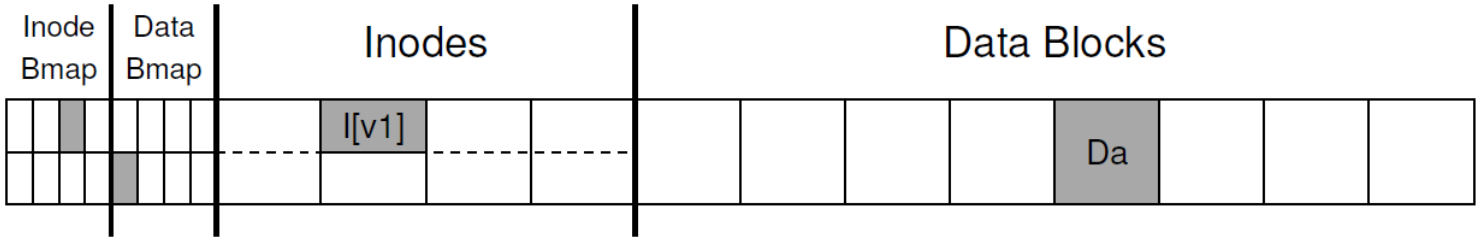
- ביוניקס, כל מערכות הקבצים מאוחדים לעץ קבצים אחד.
- הפקודה mount משמשת כדי לשלב מערכת קבצים נוספת בעץ הקבצים.
- דוגמה:

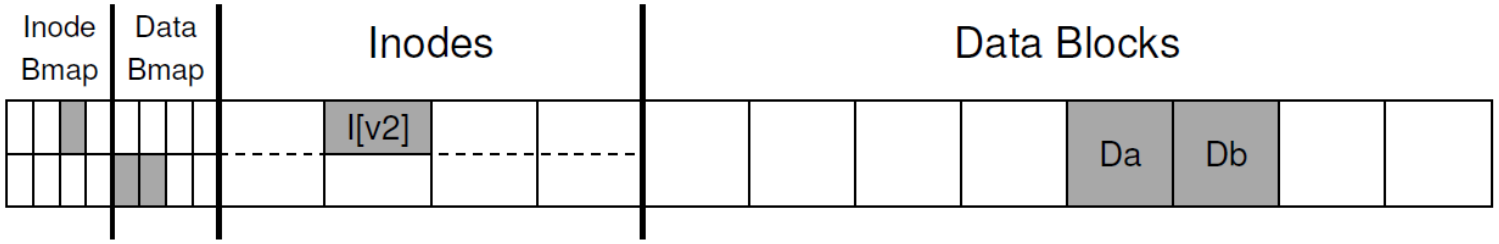
```
mount /dev/sda8 /home
```



Journaling File Systems

- פעולות במערכת הקבצים (יצירת קובץ, כתיבה לקובץ) מעדכנות כמה מבני נתונים.
  - אם תוך כדי עדכון, המערכת קרסה או שנפסק החשמל רק חלק מהפעולות יבוצע.
- דוגמה:
- נניח שיש קובץ בגודל בלוק אחד ומתבצעת כתיבה שמוסיפה בתים לקובץ.
- יש צורך להוסיף לקובץ בלוק, לעדכן את ה- Data bitmap, ולעדכן את ה- inode.
- נניח שמערכת קבצים רק הוסיפה בלוק ולא סימנה את הבלוק כתפוס ולא עדכנה את המצביעים ב- inode, המידע ילך לאיבוד.
- יתכן שמערכת הקבצים ביצעה אחת מהפעולות האחרות או שתיים מהפעולות, התוצאות יהיו שונות.



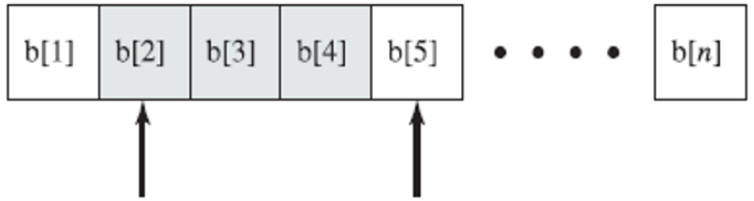


Journaling File Systems

- כדי למנוע מצב שבו מערכת הקבצים מעודכנת באופן חלקי, כותבים את כל העדכונים ל- log (Journal), ואחר כך מבצעים בדיסק.
  - ה- log הוא מקום בדיסק שמיועד לכך.
  - אם המערכת קרסה לפני השלמת הכתיבה ל- log, העדכונים לא יבוצעו.
  - אם המערכת קרסה לאחר השלמת הכתיבה ותוך כדי הביצוע בדיסק, מבצעים שוב את הפעולות הרשומות ב- log.
  - הכתיבה ל- log מאיטה את ביצועי הדיסק.
- בהמשך לדוגמה:
  - צריך לעדכן שלושה בלוקים: את ה- inode, את ה- bitmap, ובלוק נוסף.
  - את שלושת **הבלוקים** המעודכנים כותבים ל- log (physical logging).
  - לפני הבלוקים של העדכון ישנו בלוק שמסמן את תחילת העדכון ומכיל את כתובות היעד של הבלוקים, ובלוק שמסמן את סוף העדכון.

Journaling File Systems

- לאחר כתיבת הבלוקים ללוג (commit), יש צורך להעתיק אותם (checkpoint) ליעדם בדיסק.
- הלוג הוא מעגלי, לאחר ההעתקה העדכון נמחק והבלוקים פנויים לכתיבה.



- אם המערכת קרסה, אזי בזמן עליית המערכת בודקים את העדכונים שב- log:
  - אם אין לעדכון בלוק סיום – לא מבצעים אותו.
  - אם יש לו בלוק סיום – מעתיקים את כל הבלוקים ליעדם, יתכן שנעתיק שוב בלוק שכבר העתקנו.
- metadata Journaling – כדי לחסוך בהעתקה כפולה לדיסק, אפשר להעתיק את הבלוקים של ה- data ישירות לדיסק **ולאחר** מכן לכתוב ל- log רק את ה- metadata.