## שאלות ותשובות כלליות לתכנות מתקדם:

1. **מהו** buffering IO?

buffer הוא אזור מוגדר בזיכרון (בדרך כלל בגודל 4K) השומר נתונים הניתנים להעברה בין שני התקנים (devices) שונים או בין התקן ואפליקציה.

ה-buffer מתמודד במהירות בין העברת מידע (data stream) בין ה-consumer לבין ה-producer.

ה-buffer מיוצר בזיכרון הראשי כדי להגדיל את הבתים (Bytes) המתקבלים.

לאחר שנתונים מתקבלים ב-buffer הם מועברים ל-disk בפעולה אחת (לא באופן מידי), כדי שהמודם יקבל עוד נתונים הוא נדרש לייצר עוד buffer חדש, כאשר ה-buffer הראשון מתחיל להתמלא הוא מתחיל להעביר נתונים לדיסק, במקביל ה-modem מתחיל להתמלא, כאשר שני ה-buffer-ים מסיימים את המשימה שלהם, אז ה-modem עובר שוב ל-buffer הראשון והנתונים ב-buffer השני עוברים לדיסק.

ה-buffer מספק מספר ורציות של עצמו אשר מתאימות להתקנים שונים בעלי קיבולת זיכרון שונה, המטרה של ה-buffer היא להפחית את הקריאות למערכת.

בקריאה מהדיסק: בכל פעם שה-buffer ריק וגם בפעם הראשונה, מתבצעת קריאה בגודל 4K מהדיסק ל-buffer והאפליקציה קוראת בתים מה-buffer עד שמתרוקן.

בכתיבה לדיסק: האפליקציה כותבת בתים ל-buffer, כאשר מתמלא הנתונים ב-buffer מועברים לדיסק.

בעזרת הפקודה sync() מערכת ההפעלה כותבת את ה-buffers לדיסק כדי לא לאבד מידע אם המערכת קורסת, בעת פתיחת קובץ אם פתחנו אותו עם O_SYNC אז אחרי כל כתיבה ה-kernel יבצע sync לקובץ.

2. **קוד המעתיק קובץ עם** buffer **בגודל 1:**

```
#include <stdio>
int main(void) {
        char c;
        int in, out;
        in = open("in", O_RDONLY);
        out = open("out", O_WRONLY)
        while(read(in, &c, 1) == 1) {
                write(out, &c, 1);
        }
        exit(0);
}
```

kfir.goldfarb@msmail.ariel.ac.il

3. **קוד המעתיק קובץ עם** `buffer` **בגודל 1024:**

```c
#include <stdio.h>
#define BUFFER_SIZE 1024

int main(void) {
        char buffer[BUFFER_SIZE];
        int in, out, nread;
        in = open("in", O_RDONLY);
        out = open("out", O_WEONLY);
        while((nread = read(in, buffer, BUFFER_SIZE)) > 0) {
                write(out, buffer, nread);
        }
        exit(0);
}
```

4. **4 סיבות העיקריות להשתמש ב-**`fopen` **ולא ב-**`open`**:**
   a. `Fopen` כולל איתו את ה-`buffering IO` שהוא הרבה יותר מהיר מהשימוש ב-`open`.
   b. `Fopen` יודע לתרגם את הקובץ כאשר הוא לא נפתח כקובץ בינארי, דבר זה מאוד מועיל במערכות שלא מבוססות `UNIX`.
   c. `Fopen` אשר מחזיר את המצביע ל-`File` מאפשר להשתמש ב-`fscanf` ובפונקציות נוספות הקיימות בספריה `stdio`.
   d. ישנם פלטפורמות שתומכות רק ב-`ANSI C` שרק `fopen` תומך בזה ולא `open`.

5. **מה יכיל הקובץ "**`file.txt`**" לאחר כל אחת מארבע פקודות** `write()` **שבתוכנית:**

```c
int main(void) {
        char    buf1[] = "123456789012";
        char    buf2[] = "AA";
        int     fd, pos;
        fd = open("file.txt", O_WRONLY | O_TRUNC | O_CREAT
                , S_IRUSR | S_IWUSR);
        write(fd, buf1, 12);
        lseek(fd, -5, SEEK_CUR);
        write(fd, buf2, 2);
        lseek(fd, 3, SEEK_SET);
        write(fd, buf2, 2);
        lseek(fd, -2, SEEK_END);
        write(fd, buf2, 2);
}
```

פתרון:

קודם ניצור את הקובץ `file.txt`, אם קיים אז הוא מתאפס בגלל `O_TRUNC`,

ב-`write` הראשון אנחנו כותבים 12 תווים מ-`buf1`, ואז חוזרים 5 צעדים אחורה מאיפה שהסמן שלנו נמצא כלומר אנחנו נמצאים כרגע ב (סמן = `<>`):

`1234567<>89012`

נבצע כתיבה של 2 תווים מ-`buf2` מאיפה שהסמן **נמצא** ונקבל:

`1234567AA<>012`

נעביר את הסמן ל3 צעדים אחרי **ההתחלה** של הקובץ וניהיה ב:

`123<>4567AA012`

נרשום שוב 2 תווים מ`buf2` ונקבל:

`123AA<>67AA012`

נזיז את הסמן ל-2 מקומות לפני **סוף** הקובץ:

`123AA67AA0<>12`

נרשום שוב פעם 2 תווים מ-`buf2` וכעת נקבל את התוצאה הסופית:

`123AA67AA0AA`

kfir.goldfarb@msmail.ariel.ac.il

6. **בהינתן קובץ בגודל 2500 bytes ו-buffer בגודל 100 bytes, כמה קריאות מערכת יתבצעו אם נבצע את הפעולה copy?**
25 קריאות עבור read() ועוד 25 קריאות עבור write(), סה"כ 50 קריאות מערכת.
**מה אם ה-buffer יהיה בגודל 1000 bytes?**
יתבצעו 3 קריאות עבור read() ועוד 3 קריאות עבור write(), סה"כ 6 קריאות מערכת.

7. **נתון surface=4, sector_size=512 bytes, seek=5 ms, rational_rate=10,000 rpm, number of sectors=1,000, כמה זמן ייקח לקרוא קובץ בגודל 1mb (2000*512) כאשר הבלוקים מסודרים בצורה הטובה והגרועה ביותר?**

Full_rotation = 1 / 10,000 * 60 * 1000 = 6ms – כמה זמן ייקח לעשות סיבוב שלם, מחלקים 1/10,000 * 60 כדי לקבל כמה זמן ייקח לעשות מעבר על סקטור שלם במשל שנייה ואז מכפילים ב-1000 כי יש 1000 סקטורים.
Latency = full_rotation * 0.5 = 3ms
Transfer = full_rotation / 1000 = 0.006 ms
הקובץ מכיל 2000 בלוקים, בסידור גרוע:
(seek + latency + transfer) * 2000 = (5 + 3 + 0.006) * 2000 = 16,012 ms

בסידור טוב:
Seek + latency + 2 * full_rotation = 5 + 3 + 12 = 20ms

8. **לדיסק יש את הנתונים הבאים:**
**זמן חיפוש (seek) =** 10ms
**מהירות הסיבוב (סיבובים לדקה) =** 6000RPM
**מספר הסקטורים במסילה =** 100
**מהו זמן הקריאה של סקטור אחד (במקום אקראי)?**
פתרון:
נרצה קודם לחשב את מספר הסיבובים במילישניות ואז נוכל לחשב את הזמן שלוקח לקרוא סקטור אחד:
1/6000*60 ייתן לנו את הזמן בשניות, 1/6000*60*1000 ייתן לנו את הזמן במילישניות לקריאה של סיבוב שלכם, נקבל: 1/6000*60*1000=10ms, נתון 100 סקטורים ולכן נחשב: 10ms/100=0.1ms לקריאה של סקטור, נתון זה seek=10ms ולכן זמן ממוצע להגיע ספציפים לסקטור שאנחנו רוצים הוא 0.5ms וסה"כ נקבל: 10ms+5ms+0.1ms = 15.1ms

9. **הצג 3 שיטות עיקריות לתזמון דיסק:**
   א. FCFS/FIFO – הראשון שביקש הראשון לקבל, הוגן אבל לא יעיל.
   ב. SSTF – ייבצע את המשימה שנמצאת במרחק החיפוש הקצר ביותר מהבקשה הקודמת, הכי יעיל אבל יכול לגרום להרעבה.
   ג. Elevator Algorithms – הראש עובר מצד אחד לשני ומבצע משימות שיש בדיסק בדרך שבה הוא עובר, כשמגיע לקצה הוא עובר לכיוון השני וממשיך, אם עבר משימה ולא ביצע, המשימה תחכה שהראש יעבור בפעם הבאה, הוגן לכל המשימות ויותר יעיל מ-FCFS.

kfir.goldfarb@msmail.ariel.ac.il

**10. נתון בלוקים שנמצאים על מסילות של דיסק:** 98 ,183 ,37 ,122 ,14 ,124 ,65 ,67
**כאשר הראש נמצא על** 53.

**א. באיזה סדר יבצע הראש את הבלוקים שעל הדיסק?**

FCFS: 67 -> 65 -> 124 -> 14 -> 122 -> 37 -> 183 -> 98 -> 53

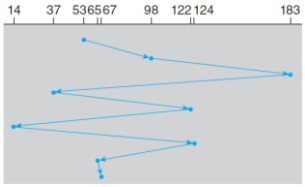
SSTD: 183 -> 124 -> 122 -> 98 -> 14 -> 37 -> 67 -> 65 -> 53

Elevator: 183 -> 124 -> 122 -> 98 -> 67 -> 65 -> 0 -> 14 -> 37 -> 53

**ב. כמה מעבר היה?**

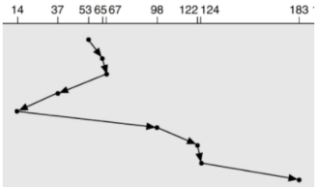
FCFS:

642 = 45+85+146+85+108+112+59+2



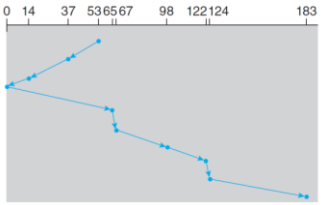

SSTD:

177 = 12+2+30+23+84+24+2




Elevator:

236 = 16+23+14+65+2+31+24+2+59




**מסקנה**:

FCFS – הוגן אבל לא יעיל.

SSTD - הכי יעיל אבל לא הוגן ויכול לגרום להרעבה (משימות רחוקות יחכו הרבה יותר).

Elevator – הכי הוגן ויותר יעיל מ-FCFS.

kfir.goldfarb@msmail.ariel.ac.il

**11. מהם הפעולות שניתן לבצע בדיסק SSD?**

א. קריאת דף: ניתן לקרוא כל דף בשלמותו – אם מציינים את מספר הדף, הפעולה קריאה היא מהירה.
ב. מחיקת בלוק: לפני כתיבה לדף צריך למחוק את כל הבלוק. הפעולה מחיקה היא איטית.
ג. כתיבת בלוק: לאחר מחיקת בלוק ניתן לכתוב בו דפים. הפעולה כתיבה יותר איטית מקריאה.

**12. מהו עקרון copy on write (העתקה בזמן כתיבה)?**

עקרון העתקה בזמן כתיבה אומר שכאשר יש מספר משאבים המשתמשים באותו מאותו מקום של זיכרון אז מיותר להעתיק את הזיכרון לכל משאב ולכן כל משאב יכיל מצביע לאותו מקום בזיכרון,

בלינוקס: כאשר מתבצעת הפעולה fork עותק של כל הדפים המותאמים לתהליך האב נוצרים ונטענים לאזור זיכרון נפרד על ידי מערכת ההפעלה לטובת תהליך הבן, פעולה זו אינה נדרשת במקרים מסוימים, למשל במקרה ונבצע דרך תהליך הבן את הקריאה execv אז אין צורך להעתיק את דפי האב, ו-execv מחליף את מרחב הכתובות של התהליך, במקרים כאלו עקרון COW בא לידי ביטוי, כאשר מתבצעת הפקודה fork דפי תהליך האב לא מועתקים לתהליך הבן, במקום זאת הדפים משותפים בין תהליך האב ותהליך הבן (לקוח מויקיפדיה).

כאשר רוצים לשתף קטע זיכרון בין תהליך אב לבן נדרש לסמן בטבלאות הדפים של האב והבן את הקטע בזיכרון כ-read only ולזכור שהם copy-on-write.

**13. מהי תכנות המקומיות?**

קיימות 2 תכונות:

א. **תכונת מקומיות בזמן** – אם תהליך ניגש לכתובת מסוימת כדי לקרוא או לכתוב או כדי לבצע פקודה מסוימת, סיכוי טוב שהוא יפנה שוב לאותה כתובת בזמן או הפקודה בזמן הקרוב.
ב. **תכונת מקומיות במקום** – אם תהליך פנה לכתובת מסוימת כדי לקרוא או לכתוב או כדי לבצע פקודה מסוימת, סיכוי טוב שהוא יפנה לכתובות סמוכות.

**14. הצג דוגמאות לתכונת מקומיות בזמן ודוגמאות לתכונת מקומיות במקום:**

**א. מקומיות בזמן:**
- חזרה על **אותו** משתנה בכל פעם שמבצעים סבב בלולאה כמו חישוב sum.
- חזרה על **אותם** פקודות (בלולאה או בסדר מסוים).

**ב. מקומיות במקום:**
- חזרה על פקודות בסדר מסוים.
- מעבר על איברי מערך לפי סדר.

**15. נתון הקוד הבא:**

```
int v[2];
int num = 10;
v[0] = num;
v[1] = v[0];
```

**א. איזה מקרה יש של מקומיות בזמן (בנתונים)?**

בשורה 3 כשאנחנו פונים למקום v[0] (כותבים) ומיד לאחר מכן בשורה 4 שאנחנו פונים שוב ל-v[0] (קוראים).
ובשורה 2 פונים למשתנה sum (כותבים) ואז פונים אליו שוב בשורה 3 (קוראים).

**ב. אליו שני מקרים יש של מקומיות במקום (אחד בקוד ואחד בנתונים)?**
בשורה 3 כשאנחנו קוראים ל-v[0] (כותבים) ומיד לאחר מכן בשורה 4 פונים ל-v[1] שזה מקום סמוך ל-v[0].

**16. מה זה systems calls (קריאות מערכת)?**
בקשה (פסיקה) שמבצעת תוכנת מחשב לליבת מערכת ההפעלה (kernel) כדי לבצע פעולה שהיא לא יכולה לבצע בעצמה, קריאות מערכת אחראיות על החיבור בין מרחב המשתמש למרחב הליבה, נותנת למשתמש מספר פונקציות שגורמות לפעולות בליבת מערכת ההפעלה (למשל יכולת קבלת גישה לרכיבי חומרה כמו קריאה מכונן קשיח, יצירת תהליך חדש, העברת מידע בין תהליכים, קריאה וכתיבה לקבצים ועוד).
קריאת מערכת execve(): במעבד מסוג x86 קריאת מערכת מתבצעת על ידי הפקודה int 0x80 שהוא מכיל את הפונקציה system_call שמטל בכל הקריאות מערכת, הפרמטרים לפונקציה מועברים בעזרת אוגרים, מספר הפסיקה מועבר באוגר שנקרא eax.

**17. מה זה context switch?**
בעברית – החלפת הקשר, בעצם זה הפעולה של המעבר בין 2 תהליכים שהמעבד מבצע, אפיה ניתן למצוא את החלפת הקשר בפעולה?
א. מספר תהליכים יכולים לחלוק את אותו מעבד וכל תהליך משתמש בכח עיבוד של המעבד בתורו, מי שמסדר את התהליכים (באיזה סדר ירוצו) הוא הסדר תהליכים (scheduler) שנותן לכל תהליך "זמן מעבד" כלומר הזמן שבו התהליך יעבוד עם המעבד.
ב. בארכיטקטורות מסוימות נדרש לבצע החלפת קשר כאשר קוראים וכותבים לדיסק.
ג. במעבר שבין מרחב המשתמש ומרחב הליבה (תלוי במערכת הפעלה).

**18. מה ההבדל בין מרחב משתמש למרחב ליבה?**
קיים ביט אחד במעבד המסמן את ה-mode שבו המעבד רץ, בין אם זה מרחב ליבה או משתמש, מרחב הליבה הוא האיזור שבו רץ כל הקוד של המערכת ההפעלה, לעומת זאת, כל הקוד שרץ על ידי המשתמש רץ במרחב הליבה, מרחב הליבה הוא היחידי שיכול לשלוט על (1) ארגון הזיכרון בדיסק (2) בקוד עצמו של המערכת ההפעלה (3) טיפול ב-O/I, כך שמרחב המשתמש לא יכול לגשת אליו והוא חייב לבקש ממרחב הליבה לבצע פעולות כאלו, זאת על מנת לשמור על בטיחות ויציבות המערכת (שהמשתמש לא ייגע בזיכרון של משתמשים אחרים למשל).

**19. מה גורם לעבור ממצב משתמש למצב מיוחס?**
כל פסיקה שהיא.

**20. מה גורם לחזור ממצב מיוחס למצב משתמש?**
הפקודה iret.

**21. מה מונע מתהליך במצב משתמש לגשת לזיכרון במערכת ההפעלה?**
כל עוד תהליך הוא במצב משתמש הוא לא יוכל לגשת לדפים שמסומנים כ-supervisor בטבלת הדפים.

**22. מה הם חמשת השלבים בביצוע פקודת מכונה?**
א. Fetch Instruction – הבאת הפקודה ש-PC מצביע אליה, מהזיכרון למעבד.
ב. Decode Instruction – פיענוח של הפקודה והפעולה הנדרשת.

ג. Fetch Operand – הבאת הנפעלים למעבד.
ד. Execute Instruction – הרצה של הפקודה.
ה. Store Result – שמירת התוצאה של הפקודה.

**23. איזה פסיקות פנימיות (Exceptions) יכולים להתקיים בזמן ביצוע פקודת מכונה?**

א. Fetch Instruction – זיכרון לא קיים או לא נגיש.
ב. Decode Instruction – פקודה לא מוגדרת.
ג. Fetch Operand – זיכרון לא קיים או לא נגיש.
ד. Execute Instruction – חלוקה ב-0 או גלישת זיכרון.
ה. Store Result – זיכרון לא קיים או לא נגיש.

**24. מה הם שלבי הטיפול בפסיקות מערכת?**

א. שמירה של האוגרים של הפעולה שמתבצעת כרגע במעבד שמופסקת, בניהם: אוגר PC, אוגר state / status, אוגר מחסנית ואוגרים נוספים.
ב. ביצוע הקוד המטפל בפסיקה.
ג. החזרת הערכים של הפועלה שהופסקה, בניהם: ערך PC, אוגר state / status, אוגר מחסנית ואוגרים נוספים, ולהמשיך להריץ את הפעולה.

**25. מה הם שלושת הסוגים של פסיקות?**

א. פסיקות פנימיות (initial exceptions)
ב. פסיקות קריאות מערכת (system calls)
ג. פסיקות חיצוניות (interrupts).

**26. איזה סוגי פסיקות חיצוניות קיימות?**

א. קלט פלט (O/I) – למשל: דיסק מודיע למעבד על סיום פעולה, מקלדת מודעי למעבד שמקש נלחץ, כרטיס רשת מודיע שהגיע מידע.
ב. שעון (timer) – נגרם על יד שעון המחשב, מאפשר למערכת ההפעלה להפסיק ביצוע תכנית ולעבור לתוכנית אחרת.

**27. מה הוא מנהל הפסיקות (Interrupt Controller)?**

רכיב חומרה המאפשר חיבור של כמה התקנים לקו פסיקה אחד למעבד, לפי הקו ממנו הגיע הפסיקה מנהל הפסיקות שולח למעבד מספר על פס הנתונים ושולח פסיקה לקו הפסיקה של המעבד, המספר שהתקבל משמש את המעבד כאינדקס לטבלת קפיצות (Interrupt Vector Table) שמערכת ההפעלה הכינה בזמן האתחול, לפי הכתובת שבטלה, המעבד עובר לביצוע קוד שמטפל בפסיקה.

**28. איזה סוגי שגיאות קיימות בקריאות מערכת?**

א. EACCES – איסור בגלל הרשאות.
ב. EBADF – מספר קובץ לא טוב.

ג. EEXIT – קובץ כבר קיים.
ד. EINTR – system call הופסק בגלל הפרעה.
ה. EINVAL – ארגומנטים לא טובים
ו. ENOENT – קובץ לא קיים.

**29. איך נקבל את קוד השגיאה שקרתה בקריאות מערכת?**

בעזרת המשתנה הגלובלי errno שמכיל את קוד השגיאה, הפונקציה void perror(const char *str) מקבל מחרוזת של קוד השגיאה ומחזירה טקסט המתאר את השגיאה.

**30. נתון טבלת דפים כלשהי בעלת 2 עמודות (מזהה תהליך process ID ו-מספר דף לוגי), לשם מה מכיל כל איבר ברשימה את מספר הדף הלוגי ואת מספר התהליך?**

כדי לדעת לאיזה דף לוגי הדף הפיזי שייך, צריך לכתוב בטבלה את מספר הדף הלוגי.
לכמה תהליכים יכול להיות אותו מספר דף לוגי וכדי שלא ימופו לאותו דף פיזי, נשמור גם את המזהה של התהליכים שנדע להפריד בניהם.

**31. מה ההבדל בין שבירה פנימית לחיצונית?**

**א. שבירה פנימית:**
אם גודל קובץ קטן מכפולה של בלוקים אזי יישארו בסוף הבלוק האחרון בתים ללא שימוש (בממוצע ישאר חצי בלוק).

**ב. שבירה חיצונית:**
לאחר הוספה ומחיקת קבצים, יישאו רווחים קטנים בין ההקצאות ללא שימוש (אם דרושה הקצאה רציפה ייתכן שלא יהיה אפשר להקצות קובץ גדול למרות שצרוף הרווחים מספיק גדול).

**32. תכנית ניגשה לכתובת בזיכרון, כדי לתרגם את הכתובת הלוגית לפיזית ה-MMU מחפש ב-TLB. הסבירו מתי יקרה כל אחד מארבעת המקרים הבאים:**

**א. לא נמצא ב-TLB ולא נגרמה פסיקה.**
נמצא בטבלת הדפים, ביט valid בטבלת הדפים דלוק, החומרה מביאה מטבלת הדפים.

**ב. לא נמצא ב-TLB ונגרמה פסיקה.**
ביט valid בטבלת הדפים כבוי – לא טעון לזיכרון או לא באזורי הזיכרון של התהליך.

**ג. נמצא ב-TLB ולא נגמרה פסיקה.**
TLB hit (זיכרון נמצא).

**ד. נמצא ב-TLB ונגרמה פסיקה.**
גישה לזיכרון נדחתה בגלל הרשאות, ניסיון כתיבה ל-read only וגישה לדפים supervisor ממצב משתמש.

**33. נניח שגודף הדף הוא 1KB,**

**א. מה מספר הדף הלוגי שבו נמצאת הכתובת הדצימלית 215201?**
גודל דף הוא 1KB כלומר 2^1024 ולכן נחשב:
215201 / 1024=210

**ב. מה המרחק (offset) מתחילת הדף?**
215201 % 1024 = 161

**34. לכל אחד מיחסי פגיעה במטמון הבאים הסבירו את פירושו:**

א. **פגיעה –** hit:
הנתון נמצא במטמון
ב. **החטאה –** miss:
הנתון לא נמצא במטמון
ג. **יחס פגיעה במטמון –** hit rate:
מספר הפגיעות מחולק במספר הגישות לזיכרון
ד. **יחס החטאה –** miss rate:
אחד פחות ה- hit rate (1-hit_rate)

**35. תכנית ביצעה 2000 פקודות שניגשות לזיכרון לקריאה או כתיבה של נתונים, 1250 מהגישות נמצאו במטמון.**
**מהו יחס הפגיעה וההחטאה של הגישות לנתונים?**
יחס הפגיעה הוא: 1250/2000 = 0.625
יחס ההחטאה הוא: 1-0.625 = 0.375

**36. נתון:** t_memory = 100 cycles, t_cache = 2 cycles, miss_rate = 0.375,
**מהו הזמן הגישה הממוצע לנתון בזיכרון?**
ניזכר: t_cache + (miss_rate * t_memorey)
נחשב: 2 + 0.375 * 100 = 39.5 cycles

**37. מה זמן הגישה הממוצע לזיכרון כאשר יש שתי רמות של מטמון?**
ניזכר: access times: L1=1 cycle, L2=100 cycle, main memory=100 cycle
Miss rates: L1=5%, L2=20%
נחשב:
1 cycle + 0.05 * [10 cycle + 0.2 * (100 cycle)] = 2.5 cycles

**38. בשתי השורות המסומנות בקוד הבא:**

```
void mark(ptr p) {
   if ((b = isPtr(p)) == NULL)
     return;
   if (blockMarked(b))
     return;
   markBlock(b);
   len = length(b);
   for (i=0; i < len; i++)
     mark(b[i]);
   return;
}
```

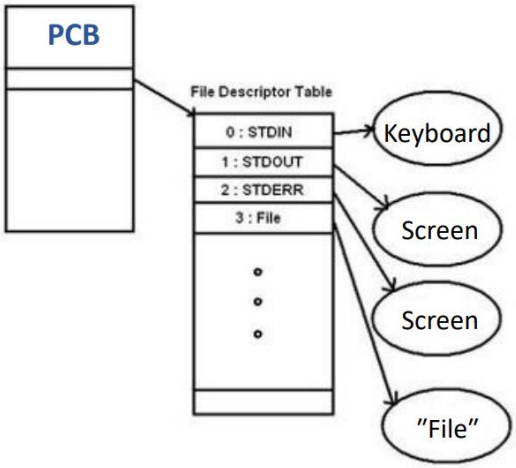
א. **מה בודקת הפונקציה** isPtr(p)**?**
האם המצביע לבלוק תפוס בערימה.
ב. **מה הצורך בלולאה** for (i=0; i<len; i++)**?**
בשפת C/C++ יתכן שכל מילה בבלוק היא מצביע.

## סיכום תכנות מתקדם:

redirect – אחראית להעביר קלט ופלט מהאמצעים הסטנדרטים לאן שנבחר:

0. stdin – הקלט שהוא מקבל באופן דפולטיבי הוא מהמקלדת.
1. stdout – באופן דפולטיבי הולך למסך.
2. stderr - באופן דפולטיבי הולך למסך.

מה הכוונה באופן דפולטיבי הולך למסך? לכל תוכנית בלינוקס קיימת טבלה המכילה מידע שרלוונטי לתוכנית (נקראת PCB), חלק מהמידע הרלוונטי הזה הוא באיזה קבצים יש שימוש על ידי התוכנית (נמצא בטבלה שנקראת file descriptor table), לכל רשומה בטבלה הזאת קיים File Descriptor המכיל integer (אינדקס) שערכיו הם 0-stdin או 1-stdout או 2-stderr וממשיך גם ל-3, 4 ... עבור קבצים נוספים שהתהליך משתמש, כל תהליך במערכת ההפעלה משתמש דפולטיבית בשלושת הקבצים הסטנדרטים הראשונים.




**דוגמה:** הפקודה ls מראה את כל הקבצים והתיקיות הנמצאות בתיקייה ספציפית ולכן התוכנית הזאת משתמשת בהוצאת למסך כלומר stdout,

נחזור לפקודה redirect, המטרה היא לשנות את הערך שמחזיק ה-(fb (file descriptor, כלומר במקום שפקודה מסוימת תדפיס למסך אולי נרצה נשמור את הפלט בקובץ או להעביר אותו כקלט לתוכנית אחרת,

נניח ונרצה לשמור את הפלט של הפקודה ls -l בקובץ שנקרא ls_out, נבצע:

```
~$ ls -l > ls_out
```

בעצם ביצענו redirect לפלט של התוכנית, אפשרות נוספת היא במקום לשמור את הפלט לקובץ נוכל להוסיף את הפלט לסוף הקובץ קיים ע"י:

```
~$ ls -l >> ls_out
```

בצורה דפולטיבית הפקודה redirect בעזרת הסימון משתמשת בערך דפולטיבי של 1 כלומר stdout, נוכל לבצע את הסימון 2> ואז במקום להשתמש ב-stdout נשמש ב-stderr:

```
:~$ ls -l 2> ls_out
```

**דוגמה:** נניח את התוכנית הפשוטה הבאה (כתובה ב-C):

```
1 #include <stdio.h>
2 int main(void) {
3         printf("STDOUT\n");
4         perror("STDERR\n");
5         return 0;
6 }
```

וכעת נבצע את הפקודה הבאה:

```
kfir@kfir-VirtualBox:~/Desktop$ 2>&1 ./main > ls_out
STDERR
: Success
```

נוכל לראות שרק ה-stderr מופיע על המסך אבל ה-stdout מופיע בתוך הקובץ ls_out:

```
1 STDOUT
```

ראינו שחץ ימינה > מוביל אל stdout, אז חץ שמאלה < מוביל אל stdin, למשל:

```
kfir@kfir-VirtualBox:~/Desktop$ cat < ls_out
STDOUT
```

הפקודה cat מדפיסה בצורה פשוטה ל-terminal שעליו אנחנו עובדים כעת.

**sort:**

נניח וקיים לנו הקובץ fruit.txt המכיל שמות של פירות כך:

```
kfir@kfir-VirtualBox:~/Desktop$ cat fruit.txt
Peach
Apple
apple
Grapes
Apple
Banana
Mango
Avocado
Tometo
Watermelon
Orange
```

כמו שאנחנו מציגים בעזרת `cat` למסך שלנו (`terminal`) נרצה להציג בצורה ממיינת:

```
kfir@kfir-VirtualBox:~/Desktop$ sort fruit.txt
apple
Apple
Apple
Avocado
Banana
Grapes
Mango
Orange
Peach
Tometo
Watermelon
```

`pipeline`:

נרצה להפוך פלט של תוכנית אחת לקלט של תוכנית אחרת בעזרת הסימון |,

```
kfir@kfir-VirtualBox:~/Desktop$ sort fruit.txt | uniq
apple
Apple
Avocado
Banana
Grapes
Mango
Orange
Peach
Tometo
Watermelon
```

כמו שאנחנו רואים `uniq` היא תוכנית המקבלת טקסט ומוחקת מימנו מילים/שורות שהם אותו דבר.

`word count`:

סופרת שורות בטקסט, למשל כדי לדעת כמה קבצים יש בתיקייה הנוכחית נוכל לבצע:

```
kfir@kfir-VirtualBox:~/Desktop$ ls -l | wc -l
6
```

התוספת של `-l` בסוף ה-`wc` היא לספור את מספר השורות בטקסט הנכנס אליו כקלט, ואם נמחק את `-l` אז בצורה דפולטיבית `wc` ידפיס כמה שורות, מילים ותווים יש בטקסט המתקבל:

```
kfir@kfir-VirtualBox:~/Desktop$ ls -l | wc
      6      47     258
```

`wild cards`:

המטרה של `wild cards` היא לבצע פעולות על מספר קבצים בבת אחת, נראה מספר דוגמאות לפעולות בסיסיות בעזרת הפקודה `ls`, פעולות אלו עובדות גם עם הפעולות `cp (copy)`, `rm (remove)` ועוד, הפקודה `ls` כמו שאמרנו מראה את כל הקבצים בתיקייה:

```
kfir@kfir-VirtualBox:~/Documents$ ls
doc          doc2.txt  file        history        some_folder_14
doc1.txt     doc3.txt  folder_12   README.md
doc_22.txt   empty     folder_199  some_folder_12
```

מה אם נרצה לראות רק את הכל הקבצים אשר שמם מתחיל במילה `doc`?

```
kfir@kfir-VirtualBox:~/Documents$ ls doc*
doc  doc1.txt  doc_22.txt  doc2.txt  doc3.txt
```

נשים לב שיש לנו 3 קבצים טקסט בעלי פורמט דומה (`doc1.txt, doc2.txt, doc3.txt`) ולכן כדי להציג רק את הפורמט הזה נוכל להשתמש בסימן שאלה כך:

```
kfir@kfir-VirtualBox:~/Documents$ ls doc?.txt
doc1.txt  doc2.txt  doc3.txt
```

נניח ויש לנו את הקבצים בפורמט הבא (כמו בפורמט הקודם):

```
kfir@kfir-VirtualBox:~/Documents$ ls doc?.txt
doc1.txt  doc2.txt  doc3.txt  doc4.txt  doc5.txt  doc6.txt
```

נניח ונרצה לראות רק לפי תווך מסוים, נשתמש ב-`[]` כך:

```
kfir@kfir-VirtualBox:~/Documents$ ls doc[3-5].txt
doc3.txt  doc4.txt  doc5.txt
```

כלומר הגדרנו תווך של תצוגה רק בין 3 ל-5.

נוכל גם לקחת מהפורמט קבצים ספציפיים (אפשר להשתמש גם בלי פסיקים):

```
kfir@kfir-VirtualBox:~/Documents$ ls doc[1,2,5,6].txt
doc1.txt  doc2.txt  doc5.txt  doc6.txt
```

אפשר גם בעזרת סימן קריאה להביא תוצאות בלי קבצים ספציפיים לפי פורמט מסוים כלומר:

```
kfir@kfir-VirtualBox:~/Documents$ ls doc[!2].txt
doc1.txt  doc3.txt  doc4.txt  doc5.txt  doc6.txt
```

נניח וקיימים לנו 12 קבצים בשני פורמטים שונים (`txt, doc`):

```
kfir@kfir-VirtualBox:~/Documents$ ls doc?.txt
doc1.txt  doc2.txt  doc3.txt  doc4.txt  doc5.txt  doc6.txt
kfir@kfir-VirtualBox:~/Documents$ ls doc?.doc
doc1.doc  doc2.doc  doc3.doc  doc4.doc  doc5.doc  doc6.doc
```

מה אם נרצה להציג קבצים שהם משני הפורמטים הנ"ל? (נשתמש ב-{}):

```
kfir@kfir-VirtualBox:~/Documents$ ls {doc*.txt,doc*.doc}
doc1.doc  doc_22.txt  doc2.txt  doc3.txt  doc4.txt  doc5.txt  doc6.txt
doc1.txt  doc2.doc    doc3.doc  doc4.doc  doc5.doc  doc6.doc
```

`who & whomai`:

הפקודה `who` מדפיסה מי מחובר למחשב ומתי הוא התחבר:

```
kfir@kfir-VirtualBox:~/Documents$ who
kfir     :0           2022-04-11 10:21 (:0)
```

הדפסה של אותו דבר רק בצורה של טבלה:

```
kfir@kfir-VirtualBox:~/Documents$ who -H
NAME     LINE         TIME             COMMENT
kfir     :0           2022-04-11 10:21 (:0)
```

הפקודה `whoami` נותנת רק את השם של המשתמש שמחובר למחשב:

```
kfir@kfir-VirtualBox:~/Documents$ whoami
kfir
```

מתי לאחרונה המחשב ביצע boot:

```
kfir@kfir-VirtualBox:~/Documents$ who -b
         system boot  2022-04-11 10:20
```

באיזה מצב ריצה המחשב נמצא:

```
kfir@kfir-VirtualBox:~/Documents$ who -r
         run-level 5  2022-04-11 10:20
```

(run-level 5 הוא המצב הסטנדרטי שאומר שמספר משתמשים יכולים להתחבר למערכת הפעלה)

**הערה:** נשים לב שהזמן האחרון שבוא בוצע boot וגם המצב ריצה של המחשב נותנים אותו זמן כי לאחר שעשינו boot למחשב המחשב נכנס למצב ריצה הזה.

מספר המשתמשים המחוברים כרגע למחשב ושמותיהם:

```
kfir@kfir-VirtualBox:~/Documents$ who -q
kfir
# users=1
```

**:last**

פקודה המראה את כל החיבורים האחרונים למערכת:

```
kfir@kfir-VirtualBox:~/Documents$ last
kfir     :0           :0               Mon Apr 11 10:21    gone - no logo
reboot   system boot  5.13.0-39-generi Mon Apr 11 10:20   still running
kfir     :0           :0               Tue Apr  5 00:01 - down   (00:00)
reboot   system boot  5.13.0-39-generi Tue Apr  5 00:00 - 00:01  (00:00)
kfir     :0           :0               Mon Apr  4 17:15 - down   (06:45)
reboot   system boot  5.13.0-28-generi Mon Apr  4 16:37 - 00:00  (07:23)
kfir     :0           :0               Mon Feb 21 10:28 - down   (01:21)
reboot   system boot  5.13.0-28-generi Mon Feb 21 10:28 - 11:49  (01:21)
kfir     :0           :0               Mon Feb 21 10:18 - down   (00:01)
reboot   system boot  5.13.0-28-generi Mon Feb 21 10:18 - 10:20  (00:01)
kfir     :0           :0               Sun Feb 20 21:04 - down   (02:35)
reboot   system boot  5.13.0-28-generi Sun Feb 20 21:04 - 23:39  (02:35)
kfir     :0           :0               Sun Feb 20 16:52 - crash  (04:12)
reboot   system boot  5.13.0-28-generi Sun Feb 20 16:51 - 23:39  (06:48)
```

(הרשימה ארוכה מאוד)

בעצם כל המידע הנ"ל מגיע מקובץ temp (הנקרא wtmp) שמכיל את הפרטים על ההתחברות למחשב, התאריך שבו הקובץ התחיל לבצע מעקב נמצא בסוף כל הטקסט של הקובץ:

```
wtmp begins Sun Mar 28 23:13:54 2021
```

ניתן לקבל את כל המידע הנ"ל אבל רק על משתמש ספציפי אם נכתוב את שמו אחרי הפקודה last:

```
kfir@kfir-VirtualBox:~/Documents$ last kfir
kfir     :0           :0               Mon Apr 11 10:21    gone - no logout
kfir     :0           :0               Tue Apr  5 00:01 - down   (00:00)
kfir     :0           :0               Mon Apr  4 17:15 - down   (06:45)
kfir     :0           :0               Mon Feb 21 10:28 - down   (01:21)
kfir     :0           :0               Mon Feb 21 10:18 - down   (00:01)
kfir     :0           :0               Sun Feb 20 21:04 - down   (02:35)
```

**open:**

משתמשים ב-open כדי לפתוח או ליצור קובץ חדש.

ארגומנט ראשון: מחרוזת של נתיב הקובץ.

ארגומנט שני: אופציות לפתיחת הקובץ:

1. O_RDONLY – עבור קריאה בלבד.
2. O_WRONLY – עבור כתיבה בלבד.
3. O_RDWD – עבור קריאה וכתיבה.
4. O_APPEND – הוספה לקובץ (כתיבה).
5. O_TRUNC – אם הקובץ קיים, מאפס את התוכן שלו.
6. O_CREAT – מייצר את הקובץ אם לא קיים (נדרש ארגומנט נוסף של הרשאות).
7. O_EXCL – מחזיר שגיאה אם הקובץ לא קיים.

הערך המוחזר מהפונקציה הוא ה-file descriptor – אינדקס בטבלת הקבצים הפתוחים של התהליך.

**read ו-write:**

ארגומנט ראשון: הערך fd שקיבלנו מהפונקציה open.

ארגומנט שני: מצביע לשטח שממנו רוצים לכתוב או אליו רוצים לקרוא.

ארגומנט שלישי: מספר הבתים שמעוניינים לקרוא או לכתוב.

הערך המוחזר מ-read הוא מספר הבתים שנקראו, 0 אם סוף הקובץ, 1- אם שגיאה.

הערך המוחזר מ-write הוא מספר הבתים שנכתבו, אם המספר לא שווה למספר הבתים שרצינו לקרוא יש שגיאה, או שהדיסק מלא או שעברנו את ה-file_size.

פונקציות הקריאה והכתיבה של הספרייה הסטנדרטית (stdio) הן: scanf() ו-printf(), פונקציות אלה משתמשות ב-buffer בגודל 4k כדי לחסוך מספר קריאות ל-kernel:

```c
#include <stdio.h>

int main(void) {
        int c;
        FILE * in, * out;
        in = fopen("file.in", "r");
        out = fopen("file.out", "w");
        while((c = fgetc(in)) != EOF) {
                fputc(c, out);
        }
}
```

בעזרת הפונקציה `fflush()` נוכל לכתוב ישירות ל-`kernel` מבלי להשתמש ב-`buffer`:

```c
int fflush(FILE * fp);
```

**טבלאות לניהול קבצים פתוחים:**

1. `Descriptor Table` – לכל תהליך יש טבלה כזאת, המכילה את הקבצים שהתליך משתמש בהם, כל קובץ כזה מצביע לאובייקט בטבלה הבאה:
2. `Open File Table` – מידע אודות קבצים פתוחים במערכת ההפעלה, מכיל גם את המיקום של המצביע בתוך הקובץ (מרחק נוכחי מתחילת הקובץ), מכיל את ה-`mode` של הקובץ (קריאה כתיבה), כל אובייקט כזה מצביע לאובייקט בטבלה הבאה:
3. `I-NODE TABLE` – עותק בזיכרון של תכונות הקובץ (נמצא בדיסק), התכונות: גודל קובץ, הרשאות, הבלוקים שבהם נמצא הקובץ ועוד.

`lseek`:

לכל קובץ קיים רכיב הנקרא `offset` שיודע היכן נמצע הסמן בקובץ כאשר הוא פתוח, תמיד כשפותחים קובץ הסמן נמצא בהתחלה, כשנשתמש בפונקציה `read()` ב-`C` נכניס אליה כפרמטר כמה נרצה לקרוא מהקובץ, על מנת כדי לקרוא בקובץ הפתוח הסמן יעבור את מספר הביטים שהכנסנו, זאת על מנת שפעם הבאה שנקרא מהקובץ בעזרת `read()` נוכל להמשיך את הקריאה מאיפה שהפסקנו בפעם הקודמת ולא מההתחלה.

הפקודה `lseek` מטרתה היא לשנות את מיקום הסמן,

`CUR_SEEK` – המרחק של הסמן ביחס למיקום הנוכחי.

`SET_SEEK` – המרחק של הסמן ביחס לתחילת הקובץ.

`END_SEEK` – המרחק של הסמן ביחס לסוף הקובץ.

דוגמאות:

```c
lseek(fd, -10, SEEK_CUR); // Ten bytes prior to current
lseek(fd, 0, SEEK_SET); // Start of file
lseek(fd, 5, SEEK_SET); // byte 5 (first byte is 0)
lseek(fd, 0, SEEK_END); // Next byte after the end
lseek(fd, -1, SEEK_END); // Last byte of file
lseek(fd, 100, SEEK_END); // 101 bytes past last byte
```

**:dup**

מאפשר ליצור בטבלה file descriptors מצביע נוסף לאותו מיקום בטבלה open file table:

```
new_fd = dup(fd);
```

כעת הקובץ החדש החליף את הקובץ הישן ב-file descriptors וקובץ החדש ממצביע לאותו אובייקט בטבלה open file table שהקובץ הישן הצביע עליו (בדומה לחוקיות של fork()), הפונקציה מחזירה את האינדקס החדש או 1- אם קרתה שגיאה, הפונקציה:

```
fd = dup2(fd1, fd2)
```

מייצר את האינדקס החדש וסוגרת את הישן.

**דיסקים:**

ערכים שצריך להכיר:

ms – מילי שנייה 10^-3 s

us – מיקרו שניה 10^-6 s

ns – ננו שניה 10^-9 s

דיסק מחולק למסילות (טבעות), כל מסילה מחולקת לסקטורים, סקטור הוא יחידת הזיכרון הקטנה ביותר שאפשר לכתוב או לקרוא אליה, גודלו המינימלי הוא 512 בתים או יותר.

בתחילת כל סקטור ישנה כותרת שמכילה את מספר הסקטור, בסוף כל סקטור ישנה סיומת שמכילה קוד לבדיקת / תיקון שגיאה, פעולת פרמוט הדיסק ברמה נמוכה מחלקת את כל המסילות ומוסיף כותרת וסיומת.

**זמני דיסק:**

T_Access – זמן ממוצע לקריאה או כתיבת סקטור בודד, מחשבים לפי:

T_Access = T_Seek + T_Latency + T_Transfer

T_Seek – הזמן שלוקח למקם את הראש מעל המסילה (9-4ms).

T_Latency – זמן ההמתנה לתחילת הסקטור שיגיע לראש (2-4ms).

T_Transfer – זמן המעבר של הסקטור מתחת לראש (בערך 0.002ms).

אם הבלוקים של המידע מפוזרים על הדיסק במסילות שונות , יידרשו שני הזמנים הראשונים לכל מעבר לבלוק, אם הבלוקים של המידע רציפים בדיסק, יידרשו שני הזמנים הראשונים רק לבלוק הראשון, ולשאר המעבר לבלוקים נחשב לפי זמן העבר על סקטורים.

לכן במערכת הפעלה מודרנית יש קריאה וכתיבה בדיסק ביחידות של כמה בלוקים רצופים על גודל של 4K = 8 סקטורים.

**דיסק** SSD**:**

יותר מהיר מ-HDD ויותר יקר.

דיסק SSD מחולק לבלוקים וכל בלוק מחולק לדפים, כל בלוק הוא בגודל של 128KB או 256KB והדפים בגודל 4K.

**מיפוי דפים לוגיים לפיסיים** (FTL – Flash Translation Layer)**:**

ל-ssd קיים גם זיכרון RAM ובקר (Controller) שמבצע תוכנה שנקראת ftl, תוכנה זאת מקבלת פקודות ממערכת ההפעלה לקריאה וכתיבה של דפים לוגים ומתרגם את הפקודה לקריאה, מחיקה וכתובה לדיסק של דפים פיסיים.

kfir.goldfarb@msmail.ariel.ac.il