מערכות הפעלה

2

קריאה וכתיבת קבצים

open(), read(), write(), lseek(), dup()

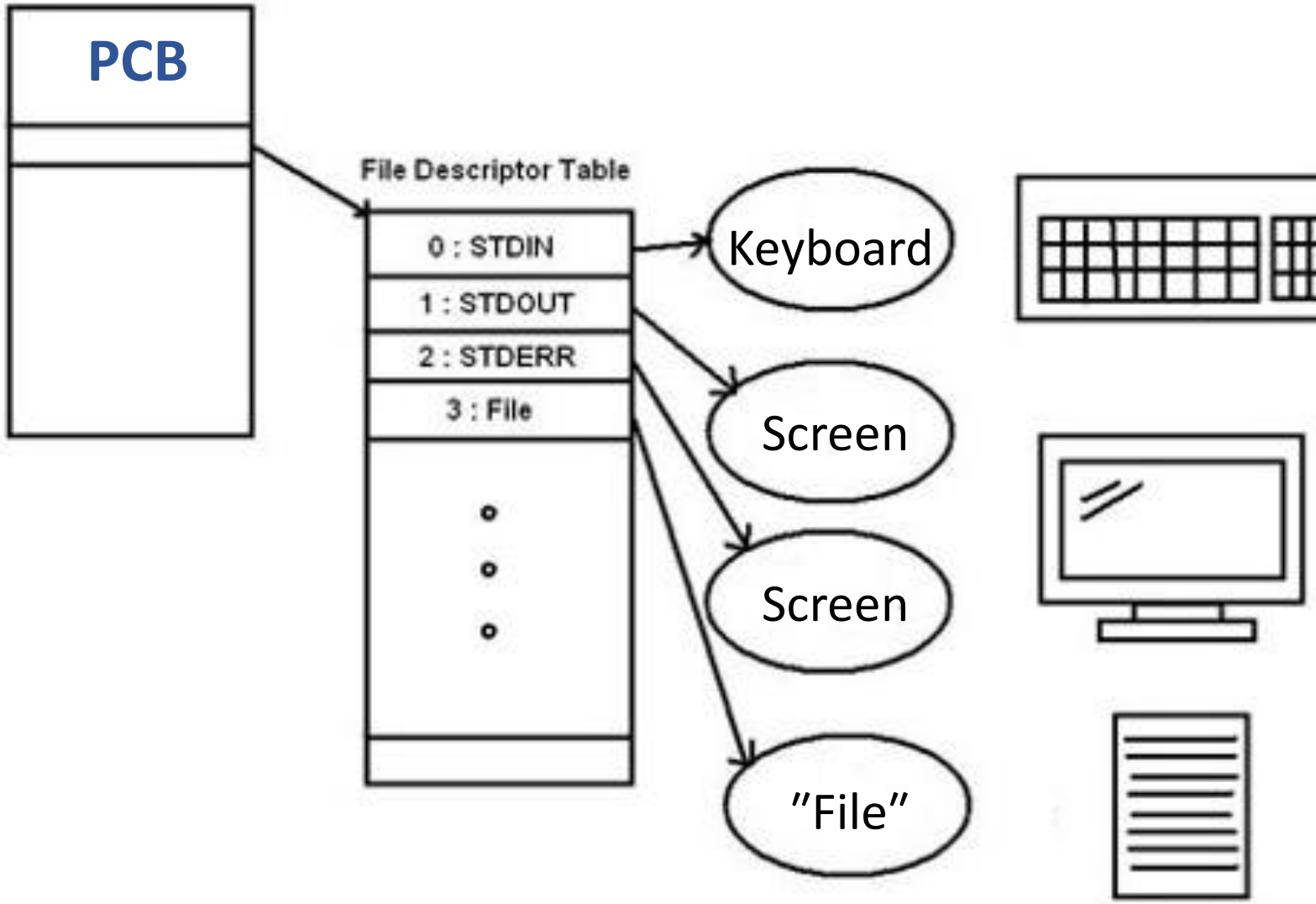
הפונקציה open()

- כדי לפתוח קובץ או ליצור קובץ חדש קוראים לפונקציה open():

```
fd = open("myfile", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- הארגומנט הראשון הוא מחרוזת שמכילה את הנתיב לקובץ שאותו רוצים לפתוח.
- הארגומנט השני מפרט את האופציות לפתיחת הקובץ, ומתבצע `or` בין האופציות:
  - `O_RDONLY` Open for reading only
  - `O_WRONLY` Open for writing only (#define O_WRONLY 00000001)
  - `O_RDWR` Open for reading and writing
  - `O_APPEND` Append to the end of file (#define O_APPEND 00002000)
  - `O_TRUNC` If the file exists, truncate its length to 0
  - `O_CREAT` Create the file if it doesn't exist - requires **mode**
  - `O_EXCL` Error if O_CREAT and file exists
- אם יוצרים קובץ (`O_CREAT`), נחוץ פרמטר שלישי כדי לפרט את ההרשאות.
- הערך המוחזר מהפונקציה `open` הוא `file descriptor`, זהו מספר (קטן) שהוא אינדקס בטבלת הקבצים הפתוחים של התהליך.

Unix File Descriptors



- לכל תהליך ישנה רשומה במערכת ההפעלה שמכילה את פרטי התהליך, הרשומה נקראת **PCB**.
- בתוך ה- **PCB** ישנה טבלה הנקראת:

File Descriptor Table

קבצים שתהליך פותח נרשמים בטבלה זו, האינדקס בטבלה משמש כדי לקרוא או לכתוב לקובץ.

- המערכת פותחת עבור כל תהליך את הקבצים 0, 1, 2:

0 - standard input

1 - standard output

2 - standard error

הפונקציות read() ו- write()

- כשרוצים לקרוא או לכתוב לקובץ משתמשים בפונקציות `read()` ו- `write()`.

1. הארגומט הראשון לפונקציות אלו הוא ה- `file descriptor` שהוחזר מ- `open`.
   - ארגומט זה מציין את הקובץ שממנו רוצים לקרוא או אליו רוצים לכתוב.
2. הארגומנט השני בפונקציות אלו הוא מצביע לשטח בזיכרון התכנית לקריאה או כתיבה.
3. הארגומנט השלישי הוא מספר הבתים שמעוניינים לקרוא או לכתוב.

- הערך המוחזר מ- read() הוא מספר הבתים שנקראו, 0 אם סוף הקובץ, 1- אם ארעה שגיאה.
  - מספר הבתים שנקראו יכול להיות קטן מהמספר המבוקש - אם הגענו לסוף הקובץ.

```
num_read = read(fd, buffer, 1024);
```

- הערך המוחזר מ- write() הוא מספר הבתים שנכתבו, אם המספר לא שווה למספר הבתים שרצינו לכתוב, ארעה שגיאה.
  - שגיאה יכולה לקרות אם הדיסק מלא או עברנו את `file size`.

```
num_written = write(fd, buffer, 1024);
```

kernel buffering of file I/O

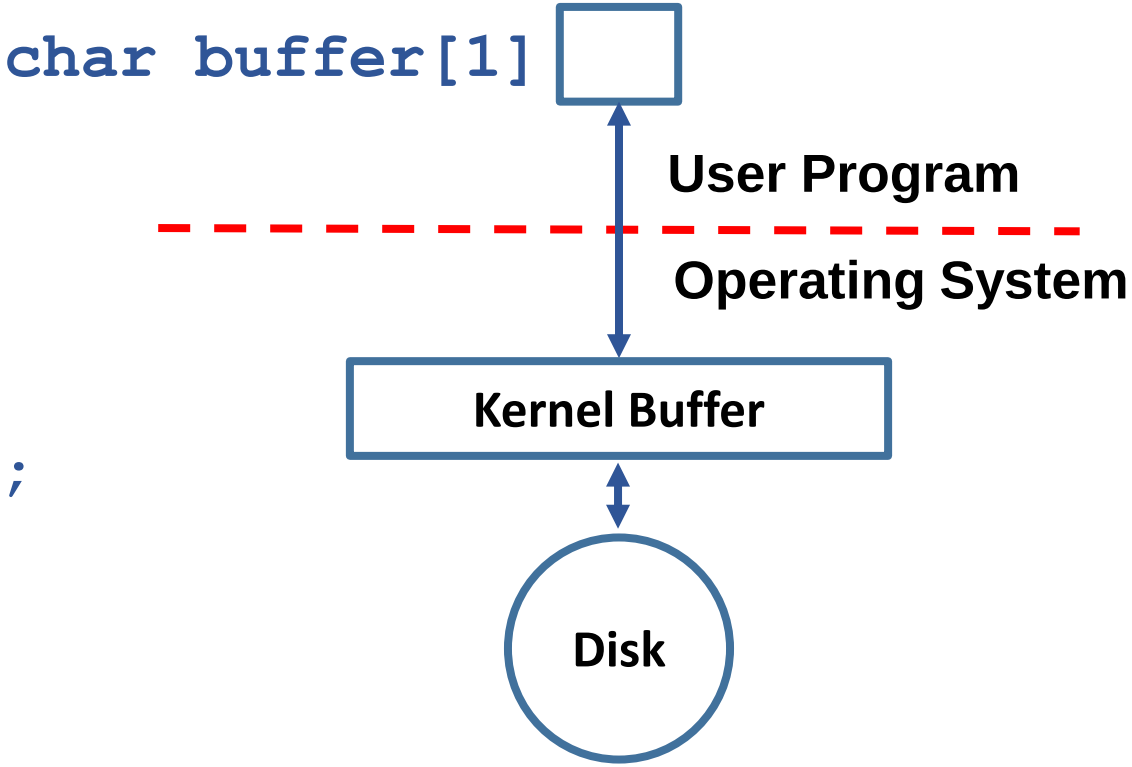
- בקריאה וכתיבה לדיסק, הפונקציות **read()** ו- **write()** לא קוראות או כותבות ישירות לדיסק.
- הפונקציות קוראות מחוצץ (buffer) שנמצא באזור הזיכרון של הקרנל וכותבות לחוצץ כדי לחסוך זמן קריאה או כתיבה לדיסק.
- כל חוצץ (buffer) בקרנל הוא בגודל בלוק של דיסק.
- מערכת ההפעלה כותבת מידי פעם את החוצצים (buffers) לדיסק באמצעות הפקודה **sync()**, כדי לא לאבד מידע במקרה של קריסת המערכת.
- אם פתחו קובץ עם **O_SYNC**, הקרנל יבצע **sync** אחרי כל כתיבה.

```
fd = open(pathname, O_WRONLY | O_SYNC);
```

- במערכת בסיס נתונים משתמשים באפשרות זו כדי לוודא שהנתונים נשמרו בדיסק, למרות שזה מאט את הכתיבה.

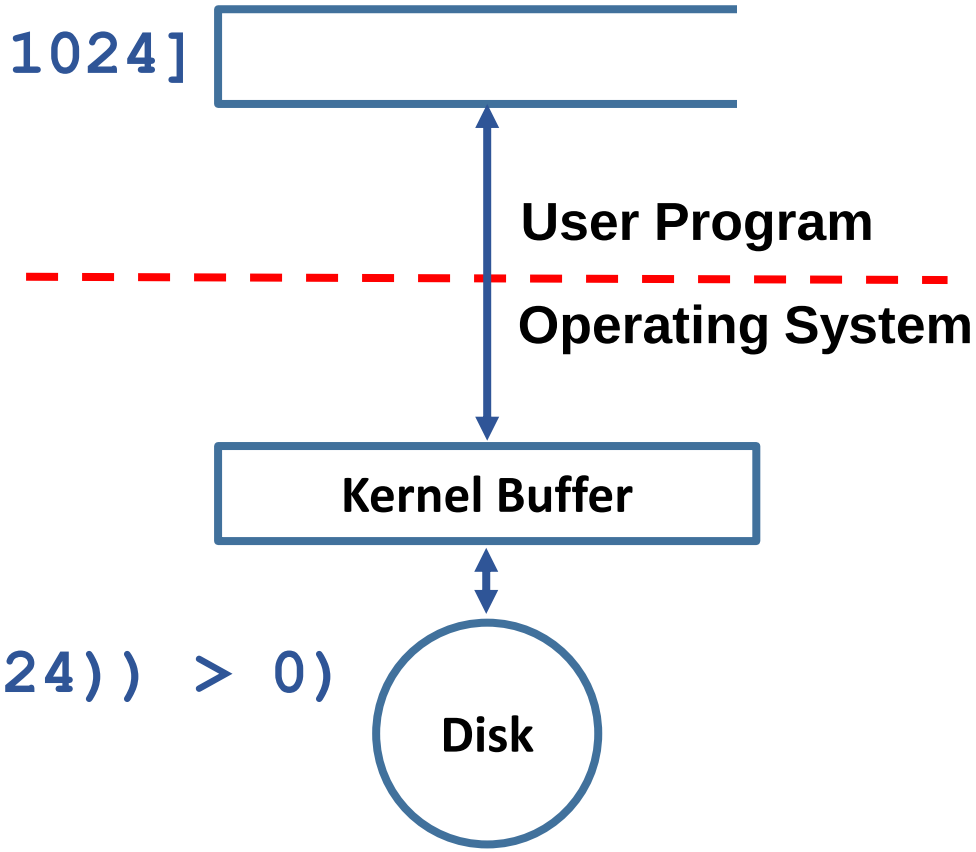
העתקת קובץ עם buffer בגודל 1

```
int main()
{
    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);
}
```



העתקת קובץ עם buffer בגודל 1024

```
int main()
{
    char buffer[1024];
    int in, out;
    int nread;
    in = open("in", O_RDONLY);
    out = open("out", O_WRONLY);
    while((nread = read(in, buffer, 1024)) > 0)
        write(out, buffer, nread);
    exit(0);
}
```



תרגיל: העתקת קובץ

1. צרו קובץ בשם file.in בגודל 10M.
(אפשר להשתמש בפקודה: `fallocate -l 10M file.in`)
הורידו ממודל את הקובץ `copy1.c`, קמפלו את הקובץ באמצעות הפקודה:

```
gcc copy1.c
```

הריצו את קובץ הריצה עם מדידת זמן:

```
time ./a.out
```

כעת הורידו את הקובץ `copy2.c` קמפלו:

```
gcc copy2.c
```

הריצו שוב עם מדידת זמן והשוו את הזמנים.
מה הסיבה להבדל בזמני הריצה?

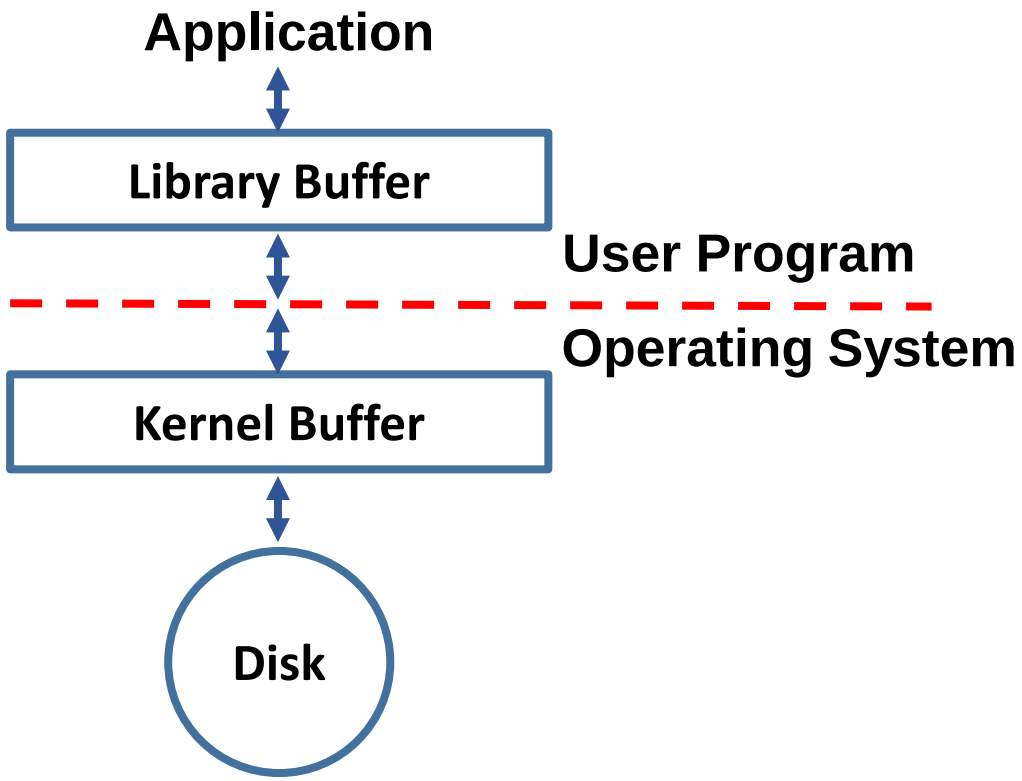
2. צרו קובץ בגודל 2K והריצו את `a.out` של `copy2.c` עם הפקודה:

```
strace -e trace=open,close,read,write ./a.out
```

העתקת קובץ עם stdio

- פונקציות הקריאה והכתיבה של הספרייה הסטנדרטית (printf(), scanf() ...) משתמשות ב- buffer (4K) כדי להפחית את מספר הקריאות למערכת, ולכן המשתמש לא צריך להגדיר buffer גדול.
- ה- buffer של הספרייה stdio הוא כדי לחסוך קריאות לקרנל, ה- buffer של הקרנל הוא כדי לחסוך גישה לדיסק.

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



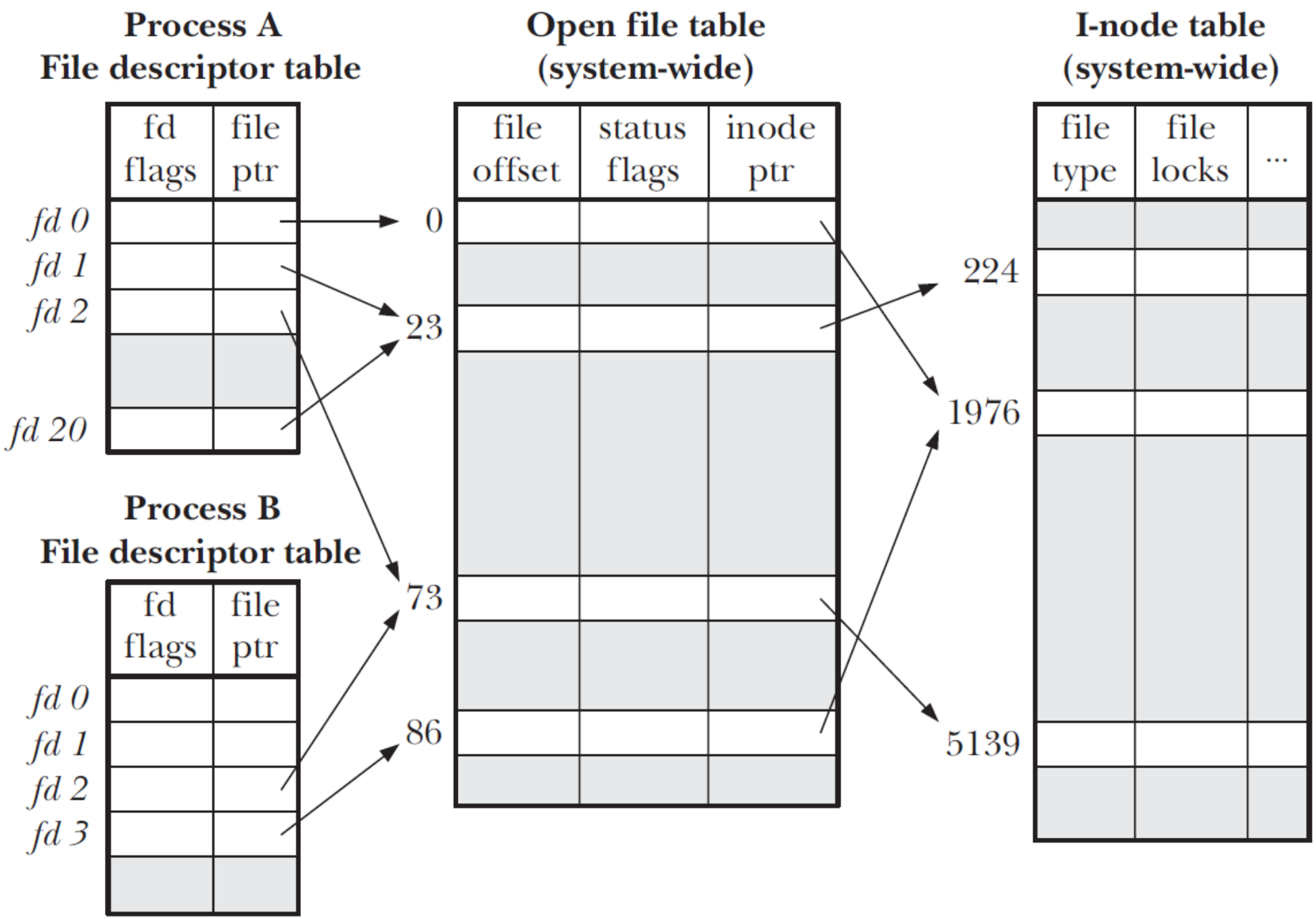
buffering of stdio stream

- דיסק – fully buffered
  - בקריאה מדיסק, בפעם הראשונה ובכל פעם שה- buffer ריק, מתבצעת קריאה בגודל ה- buffer (4K), האפליקציה קוראת מה- buffer תווים עד שמתרוקן.
  - בכתיבה לדיסק, האפליקציה כותבת ל- buffer (4K) עד שמתמלא, ואז הספרייה כותבת לדיסק.
- מקלדת ומסך - line buffered
  - בקריאה ממקלדת, מתבצעת קריאה בגודל שורה (עד לתוו new line).
  - בכתיבה למסך, מתבצעת כתיבה כאשר האפליקציה כותבת שורה (עד לתוו new line).
  - כתיבה למסך באמצעות stderr היא ללא buffer כדי שהשגיאות יופיעו מיד.
- ריקון ה- buffer
  - אפשר להגיד לספריה להעביר את הכתיבה לקרנל מיד ולא להמתין למילוי ה- buffer , עם הפונקציה `fflush()`:

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

טבלאות לניהול קבצים פתוחים

- קבצים פתוחים מנוהלים על ידי שלוש טבלאות:
- **Descriptor table**
  - לכל תהליך יש טבלה כזו, הטבלה מכילה את הקבצים שהתהליך פתח.
  - כל קובץ פתוח מצביע לאובייקט של הטבלה הבאה:
- **Open file table**
  - אובייקט בטבלה זו מכיל פרטים אודות הקובץ הפתוח:
    - מה המרחק הנוכחי מתחילת הקובץ, האם הקובץ נפתח לקריאה או כתיבה ועוד.
  - כל אובייקט בטבלה זו מצביע לאובייקט של הטבלה הבאה:
- **I-node table**
  - עותק בזיכרון של תכונות הקובץ הנמצאות בדיסק ומצורפות לקובץ, תכונות הקובץ - גודל הקובץ, הרשאות, הבלוקים בהם נמצא הקובץ ועוד.



טבלאות לניהול קבצים פתוחים

מדוע לא לאחד את הטבלאות?

הפונקציה lseek()

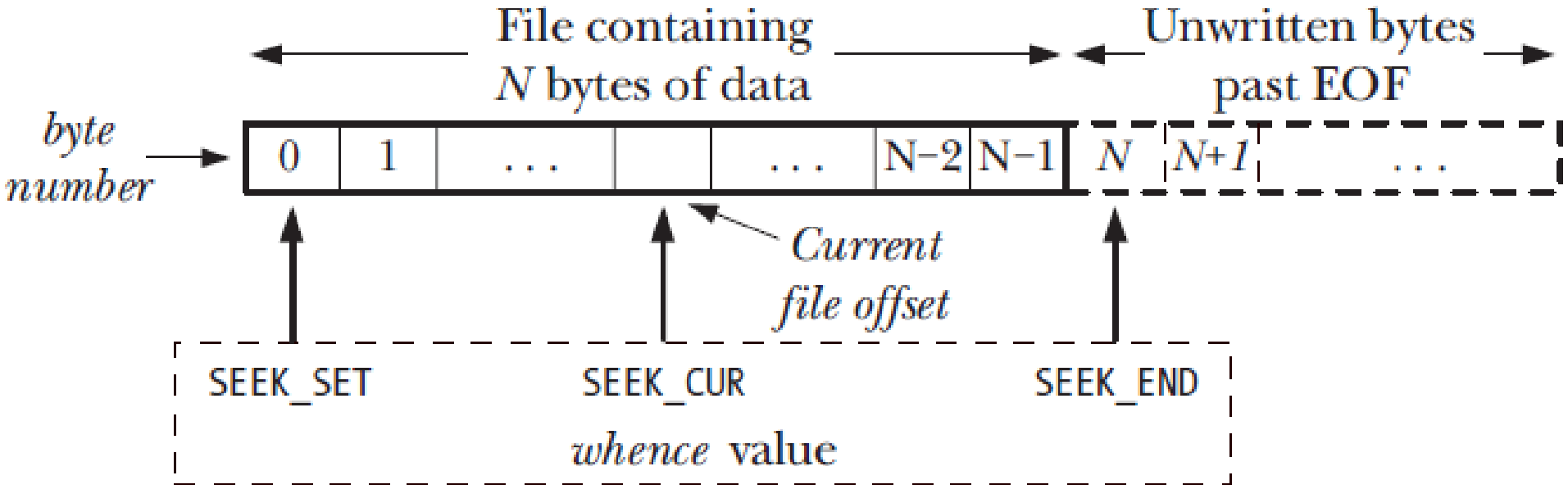
- עבור כל קובץ פתוח ישנו משתנה שמכיל את המרחק הנוכחי בבתים מתחילת הקובץ - `file offset`.
  - בדרך כלל, לאחר פתיחת קובץ המשתנה מכיל `0` – הבית הראשון בקובץ.
  - אם הקובץ נפתח עם האופציה `O_APPEND`, המשתנה מכיל את מספר הבית שאחרי סוף הקובץ.
  - פעולות קריאה או כתיבה מקדמות את המשתנה לפי מספר הבתים שנקראו או נכתבו.
- אפשר לשנות את המשתנה באמצעות קריאה לפונקציית המערכת `lseek()`:

```
new_offset = lseek(fd, offset, whence);
```

  - הפונקציה מחזירה את המקום החדש בקובץ או `-1` אם ארעה שגיאה.
  - הפונקציה רק משנה את המקום בקובץ ואינה קוראת או כותבת.

הפונקציה lseek()

- המשמעות של **offset** בפונקציה lseek() תלויה בערך של **whence**:
- **SEEK_CUR** – המרחק נקבע יחסית למקום הנוכחי.
- **SEEK_SET** – המרחק נקבע יחסית לתחילת הקובץ.
- **SEEK_END** – המרחק נקבע יחסית לסוף הקובץ (יחסית לבית שאחרי האחרון).
- המרחק יכול להיות חיובי או שלילי.



הפונקציה lseek()

- דוגמאות:

```
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
```

- כדי למצוא את המקום הנוכחי בקובץ, אפשר לבצע את הפקודה:

```
currpos = lseek(fd, 0, SEEK_CUR);
```

  - מאחר שזזנו 0 בתים יחסית למקום הנוכחי, הפונקציה תחזיר את המקום הנוכחי.

lseek()

```
int main(void) {
    char    buf1[] = "12345678\n";
    char    buf2[] = "AA";
    int     fd, pos;
    fd = open("file.txt", O_WRONLY|O_TRUNC|O_CREAT,_);
    write(fd, buf1, 9);
    lseek(fd, -6, SEEK_CUR);
    write(fd, buf2, 2);
    lseek(fd, -2, SEEK_END);
    write(fd, buf2, 1);
    pos = lseek(fd, 0, SEEK_CUR);
    printf("%d\n", pos);
}
```

מה יכיל הקובץ, מה תדפיס התכנית? תשובה: 123AA67A\n , 8

הפונקציה dup()

- הפונקציה `dup()` מאפשרת ליצור בטבלה `file descriptors`, מצביע נוסף לאותו מקום בטבלה `open file table`.

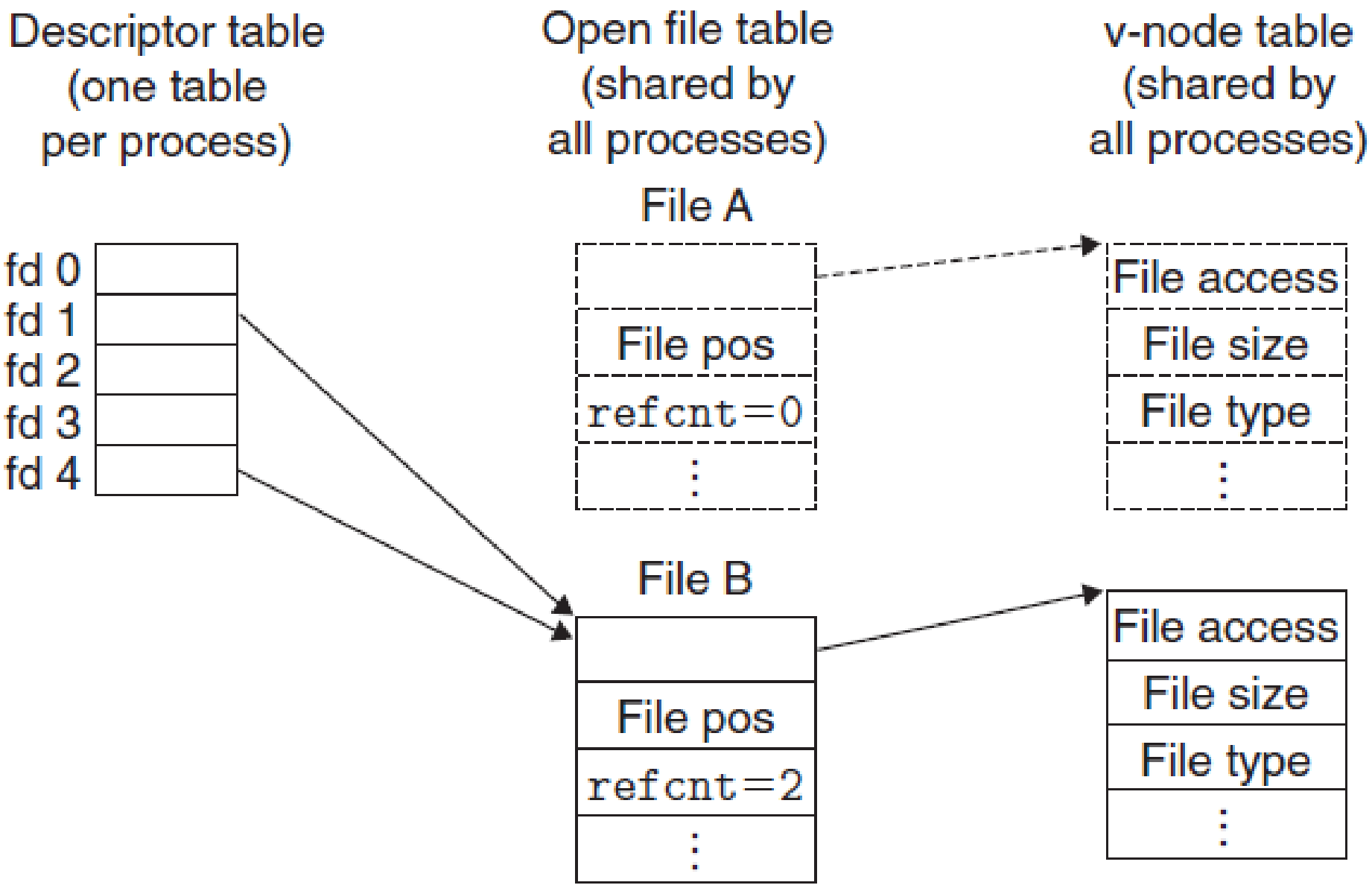
```
newfd = dup(fd);
```

  - הפונקציה מחזירה את האינדקס החדש או `-1` אם ארעה שגיאה.
  - הפונקציה תמיד תחזיר את האינדקס הפנוי הקטן ביותר.
  - הפונקציה `dup2()` מאפשרת לציין את האינדקס החדש (האינדקס הקודם ייסגר) :

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

- אחרי `fork()`, בדומה ל- `dup()`, ההורה והילד יצביעו לאותם קבצים פתוחים ולכן לאותם מקומות בטבלה `file table`.

טבלאות הקבצים לאחר קריאה ל- dup2(4,1)



טבלאות הקבצים לאחר קריאה ל- fork()