תרגול 3

סיגנלים (Signals)

קלט/פלט של תהליכים

תקשורת בין תהליכים

TL;DR

- תהליכים בלינוקס יכולים לתקשר ביניהם במגוון אמצעים, למשל:

pipes	מערכת הקבצים
`>> ls | more`	`>> ls > temp` `>> more < temp`

- לינוקס מציגה ממשק **אחיד** לכל אמצעי התקשורת בעזרת קבצים.
  - מימוש של גישת "Everything is a file" שהומצאה ביוניקס.
  - אמצעי תקשורת – קבצים רגילים, התקני קלט/פלט (מסך, מקלדת, עכבר, ...) וערוצי תקשורת ייעודיים כמו pipes, sockets.

- בנוסף, לינוקס מאפשרת תקשורת מינימליסטית בין תהליכים באמצעות סיגנלים (**signals**) – אותות מספריים שלמים בין 1—31.

מנגנוני IPC בלינוקס

- בדומה למערכות הפעלה מודרניות אחרות, לינוקס מציעה מגוון מנגנונים לתקשורת בין תהליכים:
  - (באנגלית: Inter-Process Communication = IPC).

1. **signals:** הודעות אסינכרוניות הנשלחות בין תהליכים באותה מכונה (וגם ממערכת ההפעלה לתהליכים) על-מנת להודיע לתהליך המקבל על אירוע מסוים.
2. **pipes, FIFOs:** ערוצי תקשורת בין תהליכים באותה מכונה בסגנון יצרן-צרכן.
3. **sockets:** המנגנון הסטנדרטי ליצירת ערוץ תקשורת דו-כיווני בין תהליכים היכולים להימצא גם במכונות שונות. משמש לתקשורת ברשת האינטרנט.

סיגנלים (SIGNALS)

סיגנלים (signals)

- מנגנון לשליחת הודעות לתהליכים.
  - גם (1) בין תהליכים, וגם (2) בין מערכת ההפעלה לתהליכים.
  - המנגנון ממומש בתוכנה בלבד, ללא תמיכת חומרה.

- סיגנלים נשלחים באופן **אסינכרוני** ויכולים להגיע בכל נקודה בזמן.
  - אירוע אסינכרוני == אירוע חיצוני לקוד המשתמש אשר קוטע את ריצת התוכנית וגורם למעבד להתחיל לבצד את שגרת הטיפול באירוע.
  - תהליך לא צריך לקרוא או להמתין לסיגנלים, הסיגנלים פשוט "מגיעים" לתהליך.

- **שימו לב: סיגנלים ≠ פסיקות** (טעות נפוצה של סטודנטים).
  - המקור לטעות הוא (כנראה) שגם פסיקות הן אירוע אסינכרוני. אבל בניגוד לפסיקות, סיגנלים מטופלים במצב משתמש (user mode).

בלינוקס יש 31 סיגנלים, לכל אחד שם ומספר שלם בין 1—31

```
#define SIGHUP     1
#define SIGINT     2
#define SIGQUIT    3
#define SIGILL     4
#define SIGTRAP    5
#define SIGABRT    6
#define SIGBUS     7
#define SIGFPE     8
#define SIGKILL    9
#define SIGUSR1    10
#define SIGSEGV    11
#define SIGUSR2    12
#define SIGPIPE    13
#define SIGALRM    14
#define SIGTERM    15
...
```

המשתמש לחץ על CTRL+C ב-shell – תהליך ה-shell ישלח SIGINT לתהליך שרץ בחזית.

תהליך ביצע פקודה לא חוקית – מערכת ההפעלה תשלח לתהליך SIGKILL.

תהליך ניגש לכתובת לא חוקית בזיכרון – מערכת ההפעלה תשלח לתהליך SIGSEGV.

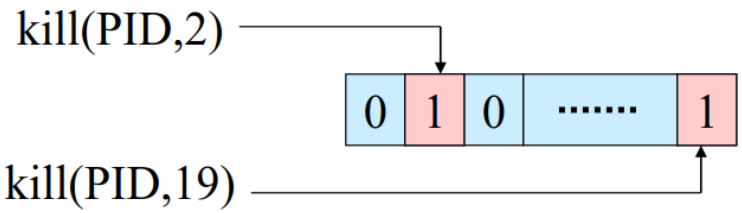
קריאת המערכת kill

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- פעולה: שולחת את הסיגנל שמספרו sig לתהליך המזוהה ע"י pid.
  - אם sig==0, אז הפעולה רק בודקת שהתהליך pid קיים מבלי לשלוח signal (שימושי לבדיקת תקפות pid).
- ערך מוחזר: 0 בהצלחה,
  -1 בכישלון (למשל אם אין תהליך בעל מזהה pid.)

העברת סיגנלים בשני שלבים

1. **<u>רישום</u>** – מערכת ההפעלה רושמת ב–PCB של תהליך היעד שיש לו סיגנל ממתין (**pending signal**).
   - הרישום מתבצע במערך **בינארי** בין 31 ביטים, ולכן לכל תהליך יכול להיות לכל היותר סיגנל ממתין **אחד** מכל מספר.



2. **<u>טיפול</u>** – בכל פעם שהתהליך חוזר **ממצב גרעין למצב משתמש**, מערכת ההפעלה בודקת אם יש סיגנלים ממתינים ומטפלת בהם.
   - בסיום הטיפול בסיגנל, מערכת ההפעלה תאפס את הביט המתאים במערך.
   - במידה ויש מספר סיגנלים ממתינים, סדר הטיפול מתחילת המערך לסופו.

טיפול בסיגנלים

- תהליך יכול לטפל בסיגנל במספר אופנים, לדוגמה:

1. **terminate** – סיום התהליך בתגובה לסיגנל.
2. **ignore** – התעלמות מהסיגנל והמשך הביצוע הרגיל.
3. **stop** – עצירת התהליך במצב TASK_STOPPED (בד"כ בשליטת debugger).
4. **continue** – המשך ביצוע תהליך שהיה במצב TASK_STOPPED (בד"כ בשליטת debugger).
5. **"תפיסת הסיגנל"** (catching signals) – הפעלת שגרת משתמש מיוחדת (**signal handler**) בתגובה לסיגנל.

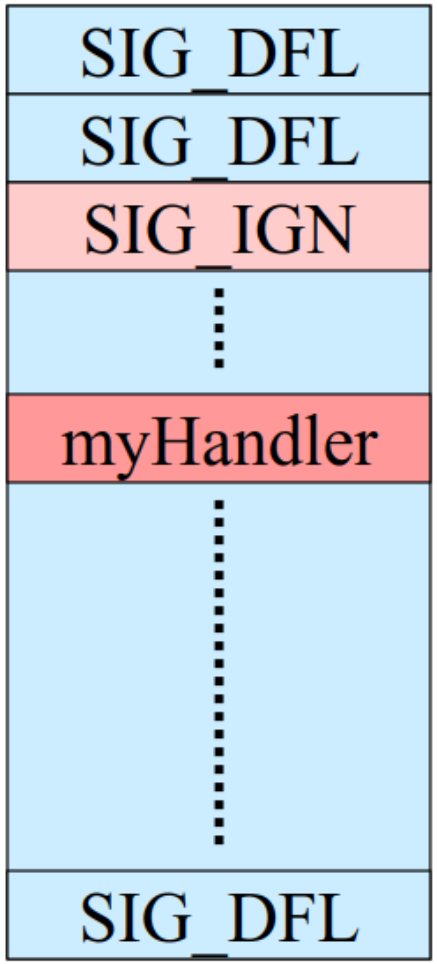
קריאת המערכת signal()

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum,
     sighandler_t handler);
```

- <u>פעולה:</u> משנה את אופן הטיפול בסיגנל שמספרו signum.
- <u>פרמטרים:</u>
  - signum – מספר בתחום 1—31 פרט ל-SIGKILL (9) ו-SIGSTOP (17).
  - handler – מצביע לפונקציית משתמש או SIG_DFL או SIG_IGN.
- <u>ערך מוחזר:</u>
  - בהצלחה, ערכו הקודם של ה-signal handler (פונקציה קודמת / SIG_DFL / SIG_IGN).
  - בכישלון, SIG_ERR.

המבנה signal_struct

- ב–PCB שמור מבנה בן 31 תאים ובו שמורות פעולות הטיפול בכל סיגנל.
- אופציות לטיפול:
  1. SIG_DFL – בצע את טיפול ברירת המחדל בסיגנל זה.
  2. SIG_IGN – התעלם מהסיגנל.
  3. קישור ל–signal handler שהוגדר ע"י המשתמש.



שגרות טיפול בסיגנלים

- תהליך יכול להתקין שגרת טיפול בסיגנל (signal handler) שתיקרא בעת קבלת סיגנל מסוים.
  - השגרה מבוצעת ב-**user mode**, בהקשר של התהליך שקיבל את הסיגנל.
  - ניתן להתקין שגרת טיפול חדשה לכל סיגנל פרט ל-SIGKILL (9) ו-SIGSTOP (17).

- איך מגנים על תהליך שהריץ קוד וקיבל סיגנל?
  - הקשר הביצוע של התהליך נשמר לפני התחלת ביצוע השגרה ומשוחזר אחרי סיומה, אך הפעלת השגרה לא גורמת להחלפת הקשר התהליך.
  - במהלך ביצוע השגרה נחסם זמנית (masking) טיפול בסיגנלים מהסוג שגרם לביצוע השגרה, על-מנת למנוע בעיות של reentrancy.

דוגמה

```
>> gcc signal.c
>> a.out &
[1] 3189
Waiting…
>> kill 3189
Hi
Bye
[1]+ Done a.out
>>
```

שליחת סיגנל מסוג **SIGTERM** באמצעות kill (bash utility)

```c
#include <stdio.h>
#include <signal.h>

void catcher1(int signum) {
   printf("Hi\n");
   kill(getpid(), 22);
}

void catch22(int signum) {
   printf("Bye\n");
   exit(0);
}

main() {
   signal(SIGTERM, catcher1);
   signal(22, catch22);
   printf("Waiting…\n");
   while(1);
}
```

שליחת סיגנלים בין תהליכים

- שימוש נפוץ בסיגנל הינו בלימה של ביצוע תהליך ע"י המשתמש.

- למשל בתוך ה-shell:
  - לחיצה על CTRL+C גורמת לשליחת SIGINT לתהליך.
    טיפול ברירת המחדל בסיגנל זה הינו סיום התהליך.
  - לחיצה על CTRL+Z גורמת לשליחת SIGTSTP לתהליך.
    טיפול ברירת המחדל בסיגנל הינו השהייה של ריצת התהליך.

מי התהליך שאחראי על שליחת הסיגנלים?

שליחת סיגנלים ע"י מערכת ההפעלה

תהליך מבצע פעולה לא חוקית (למשל, גישה לכתובת הזיכרון NULL)

המעבד יוצר חריגה ועובר לבצע את שיגרת הטיפול בחריגה במצב גרעין

מערכת ההפעלה מטפלת בחריגה ורושמת סיגנל לתהליך

בחזרה ממצב גרעין למצב משתמש (בסיום שיגרת הטיפול בחריגה) מערך הסיגנלים נבדק

שגרת הטיפול בסיגנל רצה במצב משתמש

סיגנלים – סיכום

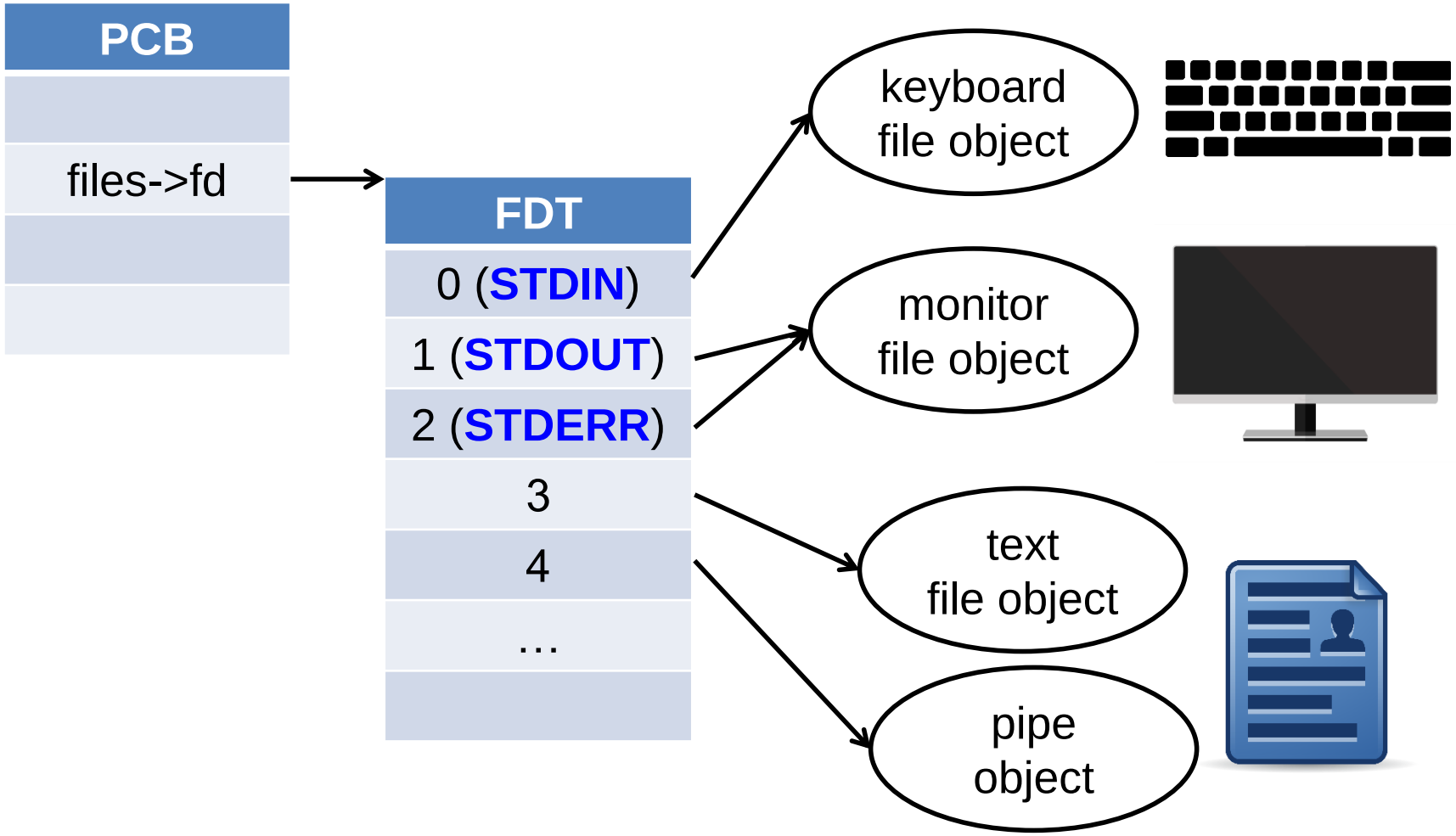
- סיגנלים נשלחים בשני שלבים: (1) רישום, (2) טיפול.

- מערכת הפעלה ← תהליך:
  - תהליך B יצר חריגה הדורשת את התערבות מערכת ההפעלה.
  - מערכת ההפעלה מודיעה לתהליך B על האירוע ע"י רישום סיגנל ב-PCB שלו.
  - במעבר ממצב גרעין לקוד משתמש של תהליך B, יטופל הסיגנל.

- תהליך ← תהליך:
  - תהליך A פונה למערכת ההפעלה שתרשום סיגנל לתהליך B.
  - מערכת ההפעלה רושמת סיגנל ב-PCB של תהליך B.
  - במעבר ממצב גרעין לקוד משתמש של תהליך B, יטופל הסיגנל.

קלט/פלט של תהליכים

Everything is a file!



FD (file descriptors)

- כל פעולות קלט/פלט של תהליך בלינוקס מבוצעות דרך "קבצים":
  - קבצים "**רגילים**" לאחסון מידע (/usr/file.txt) נמצאים **בדיסק**.
  - התקני **חומרה** גם כן מיוצגים כקבצים, אבל נמצאים **בזיכרון**.
    - למשל, העכברים המחוברים למחשב מיוצגים כ- /dev/input/mouseN .
  - גם ערוצי תקשורת כמו pipes מיוצגים ע"י קבצים שנמצאים בזיכרון.

- הקשר בין תהליך לבין קובץ שהוא ניגש אליו נשמר, ברמת המשתמש, ע"י מספר שלם שנקרא **file descriptor** **(FD)**.
  - לדוגמה: קריאת המערכת open() מחזירה FD.
  - המשתמש מעביר את ה-FD לקריאות מערכת כמו read(), write() כדי לקרוא ולכתוב לקובץ.

FDT (file descriptor table)

- ברמת הגרעין, FD הוא אינדקס לכניסה בטבלה הנקראת **file descriptor table** (**FDT**).
  - לכל תהליך יש FDT משלו, המוצבעת ע"י השדה fd→files ב-PCB.
- כל כניסה ב-FDT מצביעה על **אובייקט ניהול של קובץ פתוח** (**file object**).
  - אובייקט הניהול נגיש ומתוחזק ע"י גרעין מערכת ההפעלה בלבד.
  - file object מכיל מספר שדות.
    - למשל את "מחוון הקובץ" (**seek pointer**) המצביע למיקום הנוכחי בקובץ (כלומר, מהיכן לקרוא/לכתוב את הנתונים הבאים).
- כל הקבצים הפתוחים של כל התהליכים במערכת נשמרים גם הם בטבלה גלובאלית המנוהלת ע"י הגרעין – ה-Global FDT או **GFDT**.

ערוצי הקלט/פלט הסטנדרטיים

- ערכי ה–FD הבאים מקושרים להתקנים הבאים כברירת מחדל:
  - 0 – ערוץ הקלט הסטנדרטי (**STDIN**), בדרך–כלל מקושר למקלדת.
    - פעולות הקלט המוכרות, כדוגמת scanf() ודומותיה, קוראות למעשה מהתקן הקלט הסטנדרטי.
  - 1 – ערוץ הפלט הסטנדרטי (**STDOUT**), בדרך–כלל מקושר למסך.
    - פעולות הפלט המוכרות, כדוגמת printf() ודומותיה, כותבות למעשה להתקן הפלט הסטנדרטי.
  - 2 – ערוץ השגיאות הסטנדרטי (**STDERR**), בדרך–כלל גם הוא מקושר למסך.

דוגמת קוד

```
>> cat main.c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    FILE* f = fopen("file.txt", "w");
    fprintf(f, "Hello World!\n");
    return 0;
}

>> gcc main.c
>> strace ./a.out
...
write(1, "Hello World!\n", 13) = 13
open("file.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
write(3, "Hello World!\n", 13) = 13
...
```

מאחורי פונקציות libc אלו מסתתרות קריאות המערכת open(), read(), write()

strace הוא כלי המאפשר לעקוב אחרי קריאות מערכת במהלך התכנית

פתיחת קובץ לגישה

איך ייתכנו שתי חתימות שונות של פונקציות עם אותו שם ב-C?

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags);
int open(const char *path, int flags,
      mode_t mode);
```

- <u>פעולה:</u> פותחת את הקובץ המבוקש (לפי path) לגישה לפי התכונות המוגדרות ב-flags ולפי ההרשאות המוגדרות ב-mode.
- <u>ערך מוחזר:</u>
  - במקרה של הצלחה – ה-FD המקושר לקובץ שנפתח.
  - **האינדקס המוקצה בטבלה הוא האינדקס הפנוי הנמוך ביותר ב-FDT.**
  - במקרה של כישלון – (1-).

פתיחת קובץ לגישה (2)

- ההגדרה האמיתית של open() נראית כך:

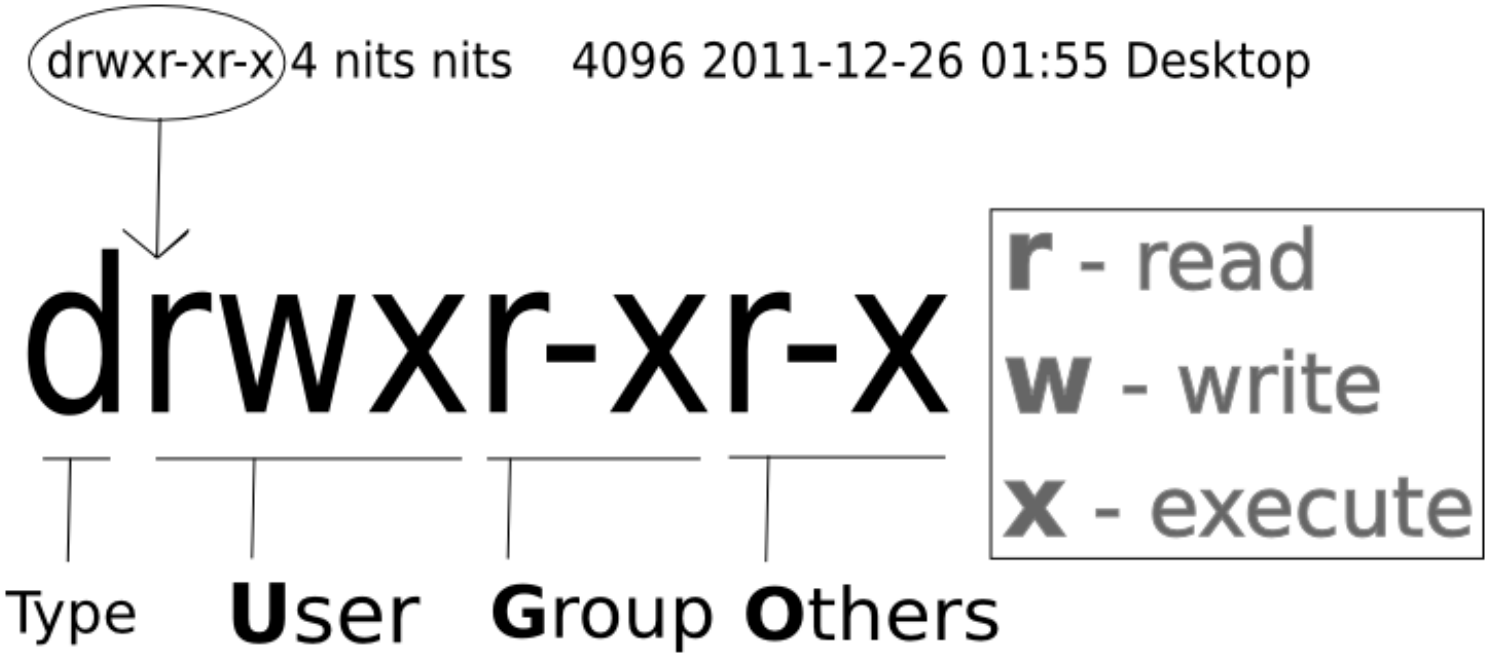
```
int open(const char *path, int flags, ...);
```

- open() מקבלת מספר משתנה של פרמטרים בדומה ל-printf().
  - פונקציות מסוג זה (variadic functions) מוגדרות עם הסימן "..." ברשימות הארגומנטים.
- הארגומנט השלישי (mode) יקרא ע"י open רק אם הארגומנט השני (flags) מאפשר **יצירת קובץ חדש**.
  - ולכן אם מגדירים יצירת קובץ חדש, **חייבים** להעביר את הארגומנט השלישי.

פתיחת קובץ לגישה (3)

- <u>פרמטרים:</u>
  - path – מסלול לקובץ (או התקן) לפתיחה. לדוגמה:
    - "file1" לציון הקובץ file1 בספריית העבודה הנוכחית.
    - "/usr/hw/file1" לציון קובץ בכתובת אבסולוטית.
  - flags – תכונות לאפיון פתיחת הקובץ. חייב להכיל אחת מהאפשרויות הבאות:
    - O_RDONLY – הקובץ נפתח לקריאה בלבד.
    - O_WRONLY – הקובץ נפתח לכתיבה בלבד.
    - O _RDWR – הקובץ נפתח לקריאה ולכתיבה.
  - ניתן להוסיף תכונות אופציונליות באמצעות OR (|) עם הדגל המתאים, למשל:
    - O_CREAT – צור את הקובץ אם אינו קיים.
    - O_APPEND – שרשר מידע בסוף קובץ קיים.
  - mode – פרמטר אופציונלי המגדיר את הרשאות הקובץ, במקרה שפתיחת הקובץ גורמת ליצירת קובץ חדש (למשל, כאשר flags מכילים O_CREAT).

הרשאות קבצים בלינוקס



File permissions in Linux

הרשאות קבצים בלינוקס – דוגמה

```
[idanyani@csm ~/Downloads]$ ls -l -h
total 210M
drwxr-xr-x 6 idanyani assist 8.0K Dec 13  2016 jre1.8.0_121
drwxr-xr-x 6 idanyani assist 8.0K Dec 20  2017 jre1.8.0_161
drwxr-xr-x 6 idanyani assist 8.0K Apr 10  2015 jre1.8.0_45
-rw-r--r-- 1 idanyani assist  71M Feb  8  2017 jre-8u121-linux-x64.tar.gz
-rw-r--r-- 1 idanyani assist  77M Feb 13  2018 jre-8u161-linux-x64.tar.gz
-rw-r--r-- 1 idanyani assist  61M Jul  4  2015 jre-8u45-linux-x64.tar.gz
-rw-r--r-- 1 idanyani assist 1.9K Aug 25  2019 launch.jnlp
-rw-r--r-- 1 idanyani assist 898K Jan  8  2019 p7-Swift.pdf
-rw-r--r-- 1 idanyani assist 274K Aug 11  2015 p84-henning.pdf
-rw-r--r-- 1 idanyani assist 655K Jul  8  2018 pmu-tools-master.zip
-rw-r--r-- 1 idanyani assist 4.1K Apr  2 19:56 viewer.jnlp
[idanyani@csm ~/Downloads]$
```

סגירת גישה לקובץ

```
#include <unistd.h>
int close(int fd);
```

- פעולה: סוגרת את הקובץ המוצבע ע"י fd. לאחר הסגירה לא ניתן לגשת לקובץ דרך fd.
- פרמטרים:
  - fd – ה–FD המיועד לסגירה.
- ערך מוחזר: במקרה של הצלחה – 0. במקרה של כישלון – 1–.

קריאת נתונים מקובץ

```
#include <unistd.h>
ssize_t read(int fd,
             void *buf,
             size_t count);
```

- פעולה: מנסה לקרוא עד count בתים מתוך הקובץ המקושר ל-fd לתוך החוצץ buf.
  - מחוון הקובץ (ה-seek pointer) מקודם בכמות הבתים שנקראו, כך שבפעולת הגישה הבאה לקובץ (קריאה, כתיבה וכד') ניגש לנתונים שאחרי הנתונים שנקראו בפעולה הנוכחית.
  - פעולת הקריאה עשויה לחסום את התהליך (כלומר, להוציא אותו להמתנה) עד שיהיו נתונים זמינים לקריאה, למשל עד שיגיעו נתונים מהדיסק.

קריאת נתונים מקובץ

- <u>פרמטרים:</u>
  - fd – ה–FD המקושר לקובץ ממנו מבקשים לקרוא.
  - buf – מצביע לחוצץ בו יאוחסנו הנתונים שייקראו.
  - count – מספר הבתים המבוקש.

- <u>ערך מוחזר:</u>
  - במקרה של הצלחה – מספר הבתים שנקרא בפועל מהקובץ לתוך buf.
    - ייתכן שייקראו פחות מ–count בתים, למשל אם נותרו פחות מ–count בתים בקובץ ממנו קוראים.
    - ייתכן גם שלא ייקראו בתים כלל, למשל אם מחוון הקובץ הגיע לסוף הקובץ (EOF).
    - אם read() נקראה עם count = 0, יוחזר 0 ללא קריאה.
  - במקרה של כישלון – (1–).

כתיבת נתונים לקובץ

```
#include <unistd.h>
ssize_t write(int fd,
              const void *buf,
              size_t count);
```

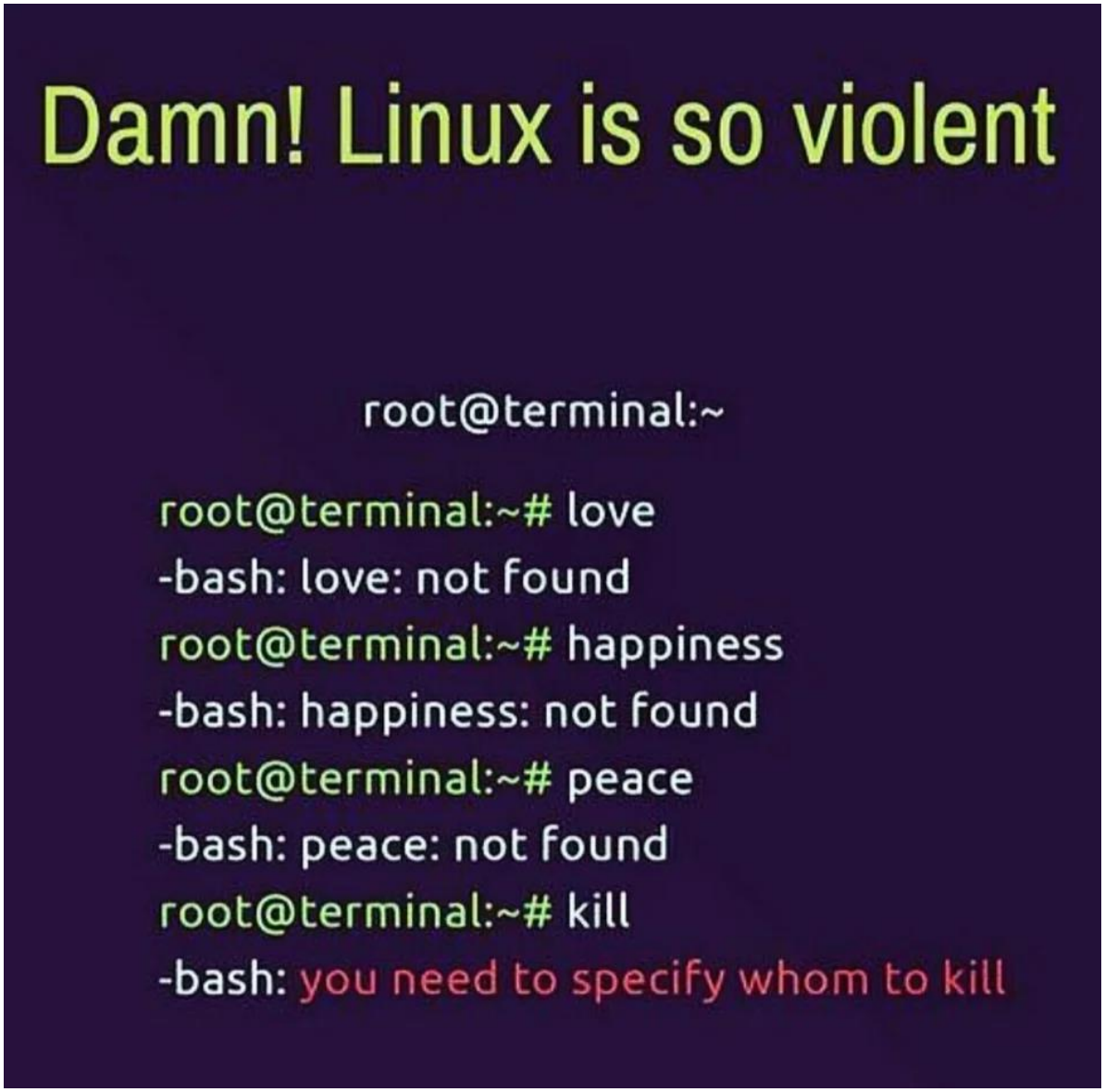
- <u>פעולה:</u> מנסה לכתוב עד count בתים מתוך החוצץ buf לקובץ המקושר ל-fd.
  - בדומה ל-()read, מחוון הקובץ מקודם בכמות הבתים שנכתבו בפועל, והגישה הבאה לקובץ תהיה לנתונים שאחרי אלו שנכתבו.
  - גם פעולת ()write יכולה לחסום את התהליך (כלומר, להוציא אותו להמתנה), למשל עד שתתאפשר גישה לדיסק.

כתיבת נתונים לקובץ

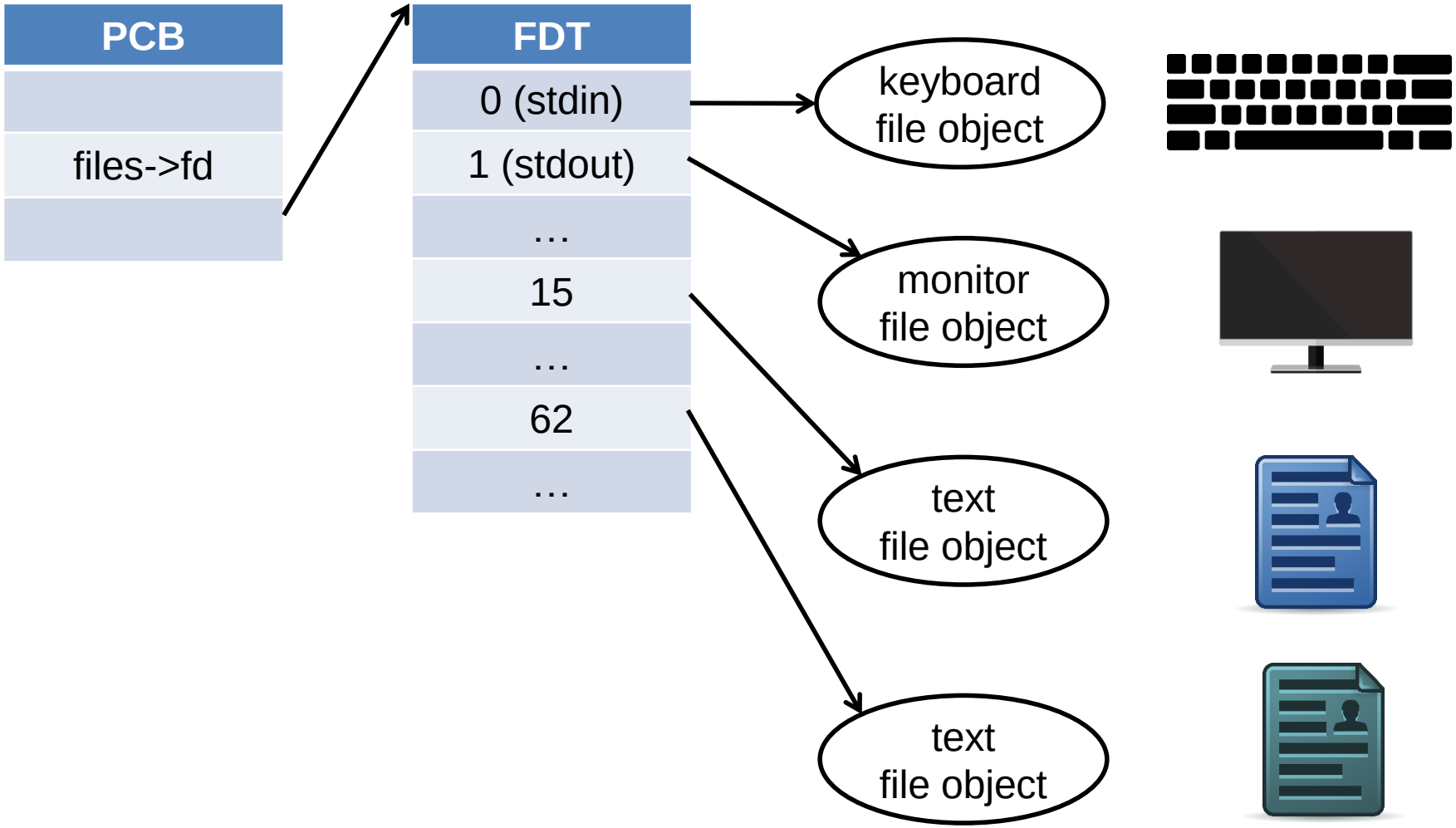
- <u>פרמטרים:</u>
  - fd – ה–FD המקושר לקובץ אליו מבקשים לכתוב.
  - buf – מצביע לחוצץ בו מאוחסנים הנתונים שייכתבו.
  - count – מספר הבתים המבוקש לכתיבה.

- <u>ערך מוחזר:</u>
  - במקרה של הצלחה – מספר הבתים שנכתב בפועל לקובץ מתוך buf.
    - ייתכן שייכתבו פחות מ–count בתים, למשל אם אין מספיק מקום פנוי בדיסק.
    - אם write() נקראה עם count = 0, יוחזר 0 ללא כתיבה.
  - במקרה של כישלון – (1–).

הפסקה


Damn! Linux is so violent
root@terminal:~
root@terminal:~# love
-bash: love: not found
root@terminal:~# happiness
-bash: happiness: not found
root@terminal:~# peace
-bash: peace: not found
root@terminal:~# kill
-bash: you need to specify whom to kill

file objects



file objects

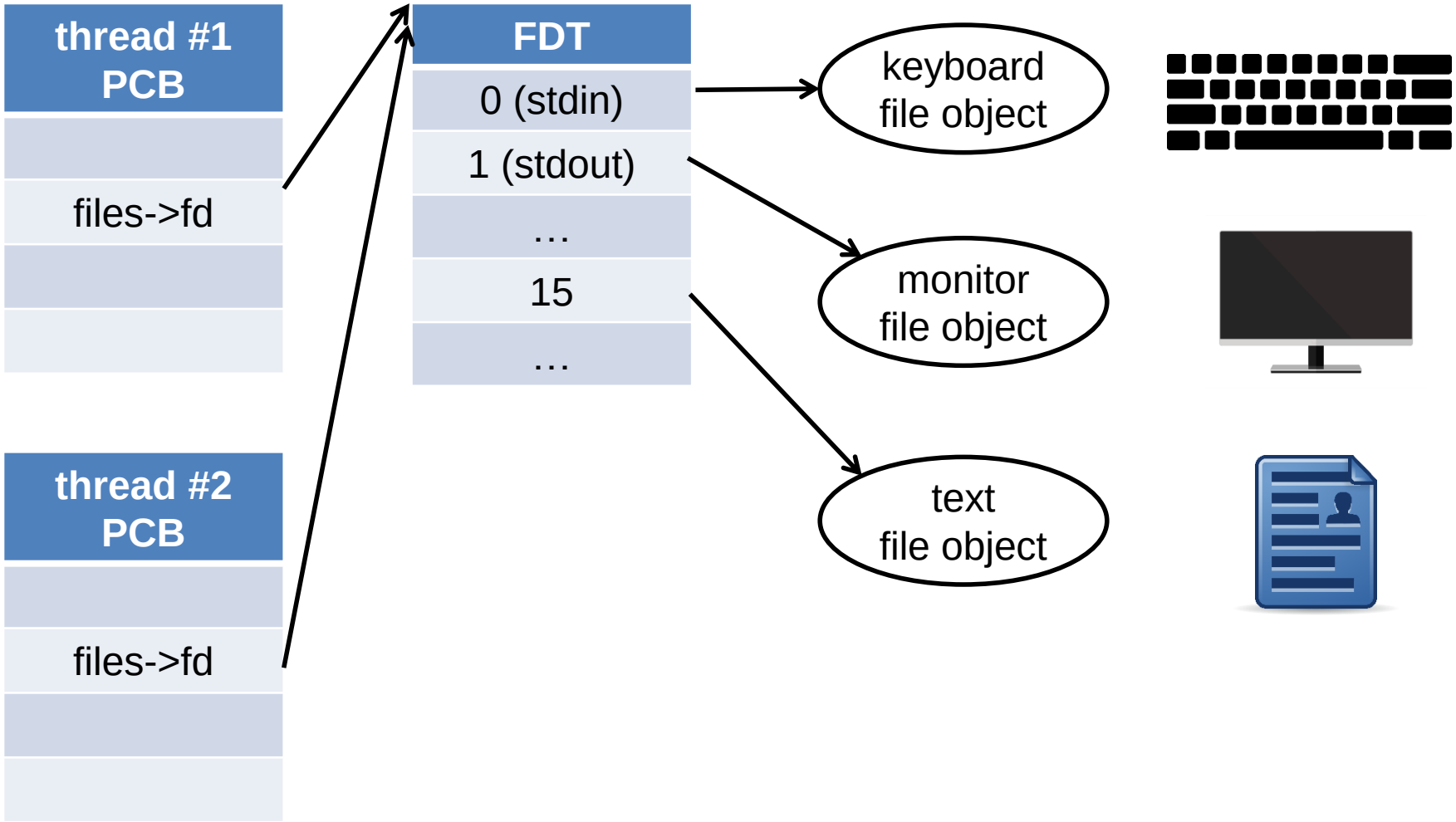
- קריאת המערכת open() מוסיפה כניסה חדשה **במקום הפנוי הראשון** בטבלת ה-FDT של התהליך.
- הכניסה החדשה מצביעה על אובייקט מטיפוס struct file:

```
struct file {
  atomic_t  f_count;
  ...
  loff_t  f_pos;
  mode_t  f_mode;
  ...
  struct file_operations* f_op;
};
```

file objects

- **f_count** – סופר את מספר ההצבעות לאותו אובייקט.
  - למשל: תהליכי אב ובן יצביעו לאותו אובייקט לאחר fork().
  - למשל: אותו תהליך יכול להצביע פעמיים לאותו אובייקט בעקבות dup().
  - המונה משמש לשחרור האובייקט ב-close() מהתהליך האחרון המצביע.
- **f_pos** – ה- seek-pointer – מצביע למיקום הקריאה או הכתיבה הנוכחי.
- **f_mode** – שומר את הרשאות הקובץ, למשל האם הקובץ ניתן לקריאה/כתיבה.
  - מערכת ההפעלה תבדוק את שדה זה לפני ביצוע הפונקציות read, write.
- **file_operations** – מבנה המכיל מצביעים למימוש של הפונקציות open,read,write,lseek ועוד רבות אחרות. נדבר על כך שוב בתרגול על מודולים ודרייברים.

שיתוף קלט/פלט בין חוטים

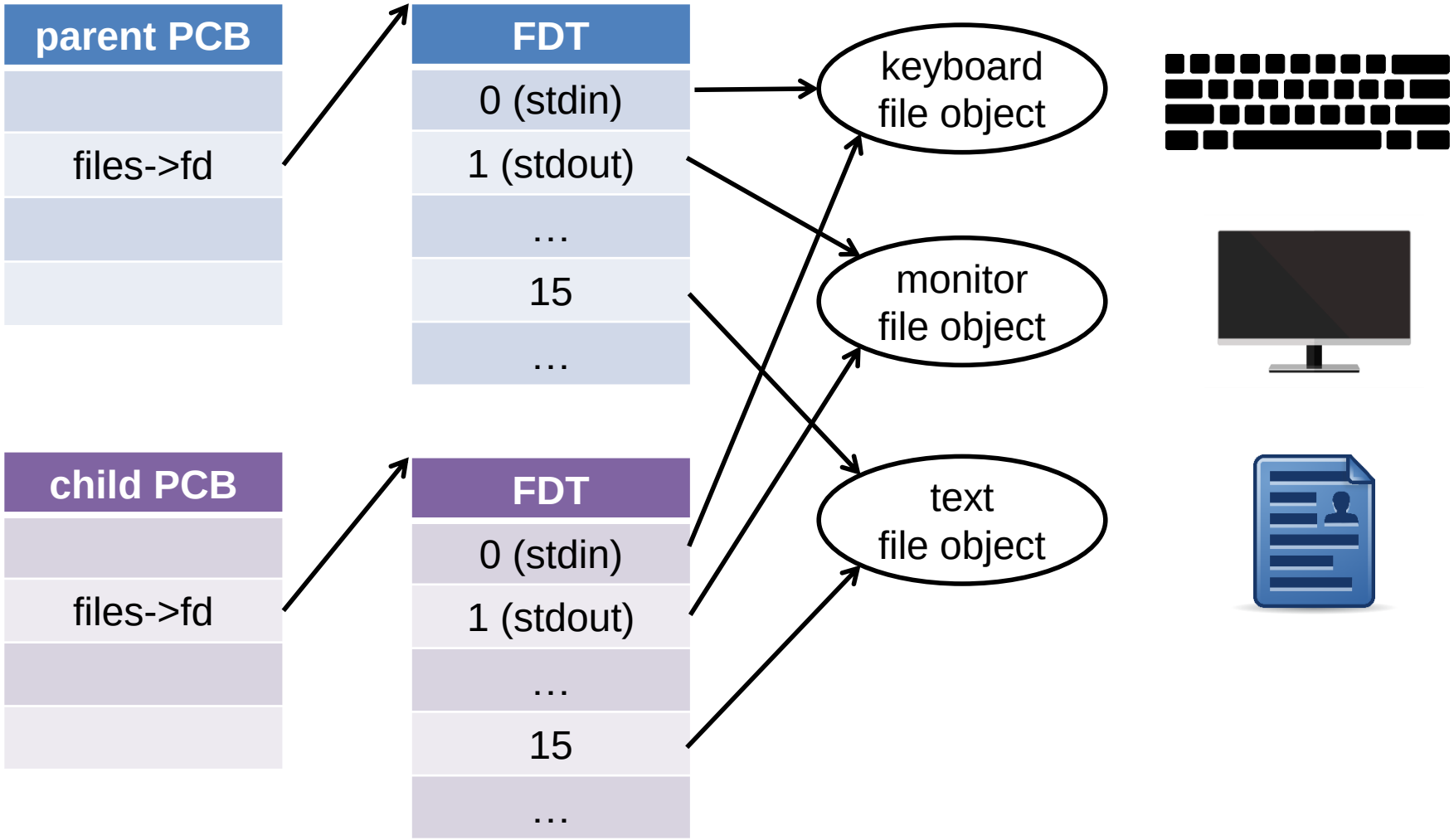


שיתוף קלט/פלט בין חוטים

- חוטים של אותו תהליך משתפים ביניהם את ה–FDT.
  - מתארי התהליכים של כל החוטים מצביעים על אותו FDT.
  - אם חוט אחד פותח קובץ גם החוט השני יכול לגשת לקובץ הזה.
  - אם חוט אחד סוגר קובץ אז גם החוט השני לא יוכל לגשת אליו יותר.

- חוטים (ותהליכים) המשתמשים ב–FD משותפים צריכים לתאם את פעולות הגישה לקבצים על–מנת שלא לשבש זה את פעולת זה.
  - לינוקס מציעה מגוון אפשרויות לנעילה של קבצים – מעבר לחומר הקורס.

שיתוף קלט/פלט בין תהליכים



שיתוף קלט/פלט בין תהליכים

- קריאת המערכת fork() מעתיקה את ה-FDT לתהליך הבן,
  - כל שינוי ב-FDT אצל האב/הבן לאחר fork() לא נראה אצל השני.
  - <u>לדוגמה</u>: אם תהליך הבן פותח קובץ חדש הוא לא נפתח אצל האבא.
- אבל ה-file objects (הקבצים הפתוחים) זהים אצל האב ואצל הבן.
  - שדות ב file object כמו **מחוון הקובץ**, יהיו זהים.
  - <u>לדוגמה</u>: אם האב קורא 10 בתים מהקובץ ואחריו הבן קורא 3 בתים, אז הבן יקרא את 3 הבתים שאחרי ה-10 של האב.
- שימו לב: כל פתיחה של קובץ מייצרת file object חדש.
  - לדוגמה: אם אותו תהליך פותח פעמיים את אותו קובץ, אז הגרעין ישמור שני file objects שונים המצביעים לאזורים שונים באותו הקובץ.

שחרור file object

- **שאלה**: מי מבצע את שחרור הזיכרון של file object?
  - מתי ניתן לשחררו?

- ייתכנו מצבים בהם תהליכים שונים מצביעים לאותו file object, לכן שחרור ה-file object יכול להתבצע רק לאחר ביצוע close() **מכל התהליכים** החולקים את אותו ה- file object.
  - נזכר – ל-file object יש מונה (f_count) הסופר את כמות התהליכים המצביעים עליו בכל רגע נתון. המונה קטן באחת עם כל פעולת close() על האובייקט.
  - כאשר המונה מתאפס, ה-file object ישוחרר.

שיתוף קלט/פלט לאחר execv()

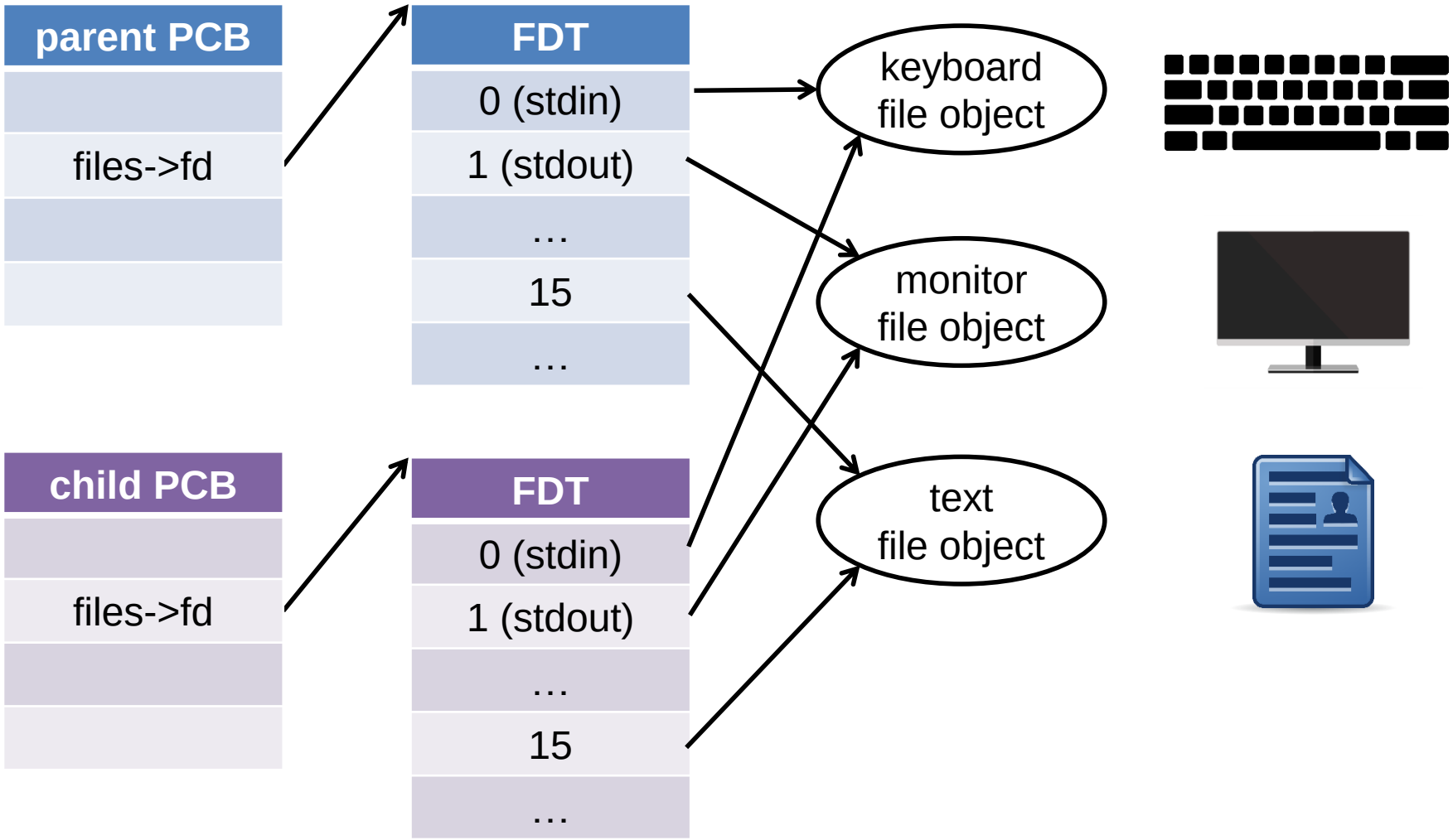
- פעולת execv() ודומותיה אינן משנות את ה-FDT של התהליך, למרות שהתהליך מאותחל מחדש.
  - כלומר, קבצים פתוחים אינם נסגרים.
- התכונה הזו שימושית להכוונת קלט/פלט של תכניות (input/output redirection).
  - למשל, ניתן לכתוב את התאריך והשעה הנוכחיים לקובץ במקום לפלט הסטדנרטי באמצעות:

```
>> date > file.txt
```

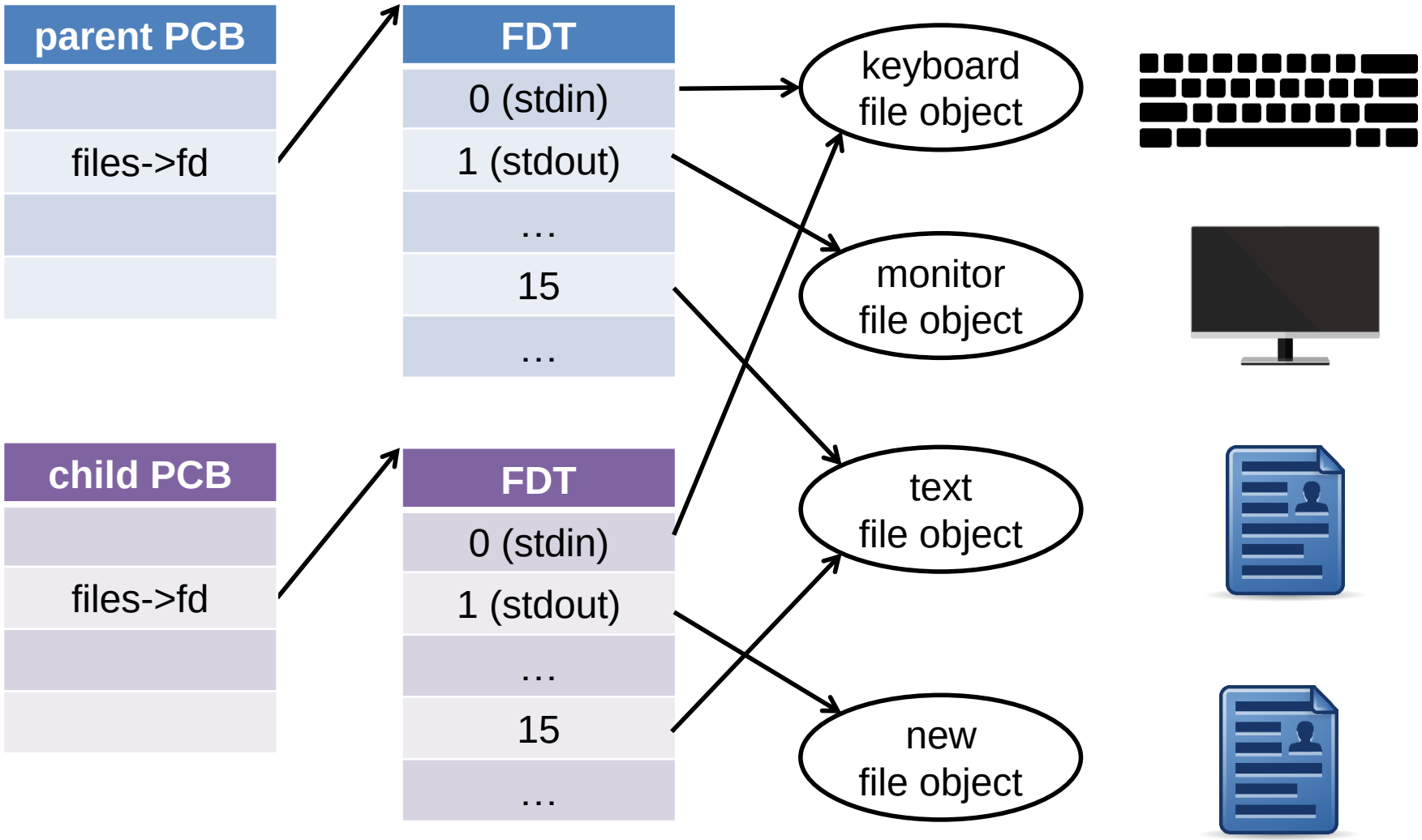
shell code:

```
pid_t pid = fork();
if (pid == 0) {
    close(1);
    open("file.txt",
        O_CREAT …, …);
    char* args[] =
        {"date", NULL};
    execv(args[0],
        args);
} else {
    wait(NULL);
}
```

הכוונת קלט/פלט



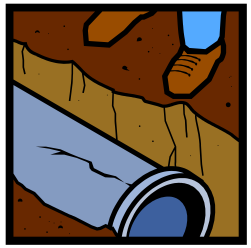
הכוונת קלט/פלט



Check Point – מה ראינו עד עכשיו

- דיברנו על דרך לספר לתהליך על אירוע.
  - בעזרת סיגנלים (signals).
- דיברנו כיצד מתבצע קלט/פלט בלינוקס.
  - בעזרת קבצים (files).
    - "Everything is a file"
- ועכשיו – נדבר על "קבצים" מיוחדים לתקשורת בין תהליכים:
  - Pipe – תקשורת בין תהליכים עם קשר משפחתי.
  - FIFO – תקשורת בין תהליכים כלשהם.

pipes בלינוקס



pipes בלינוקס

- ערוץ תקשורת **חד-כיווני** בסדר FIFO.
  - pipes מאפשרים העברת מידע בין תהליכים.
  - ניתן להשתמש ב-pipes גם לסנכרון בין תהליכים.

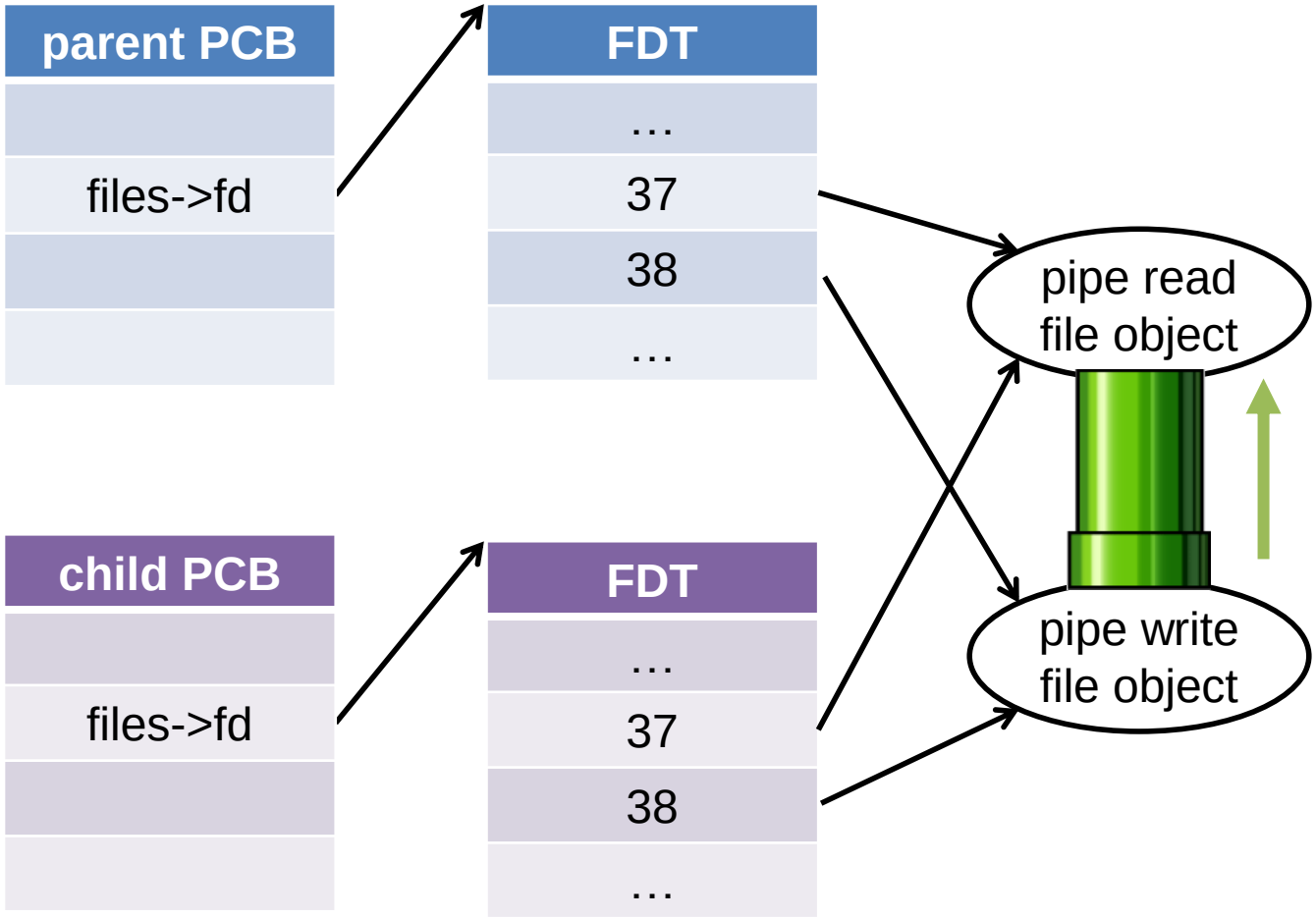
- מערכת ההפעלה לינוקס מממשת pipes באמצעות "קבצים".
  - קיים FD לכתיבה, ו-FD לקריאה.
  - קריאה וכתיבה כמו לקבצים רגילים (ע"י קריאות המערכת read/write).
  - המימוש אינו צורך כל שטח דיסק, ואינו מופיע בהיררכיה של מערכת הקבצים.
  - המידע שנכתב ל-pipe נשמר בחוצצים (buffers) של מערכת ההפעלה.

יצירת pipe חדש

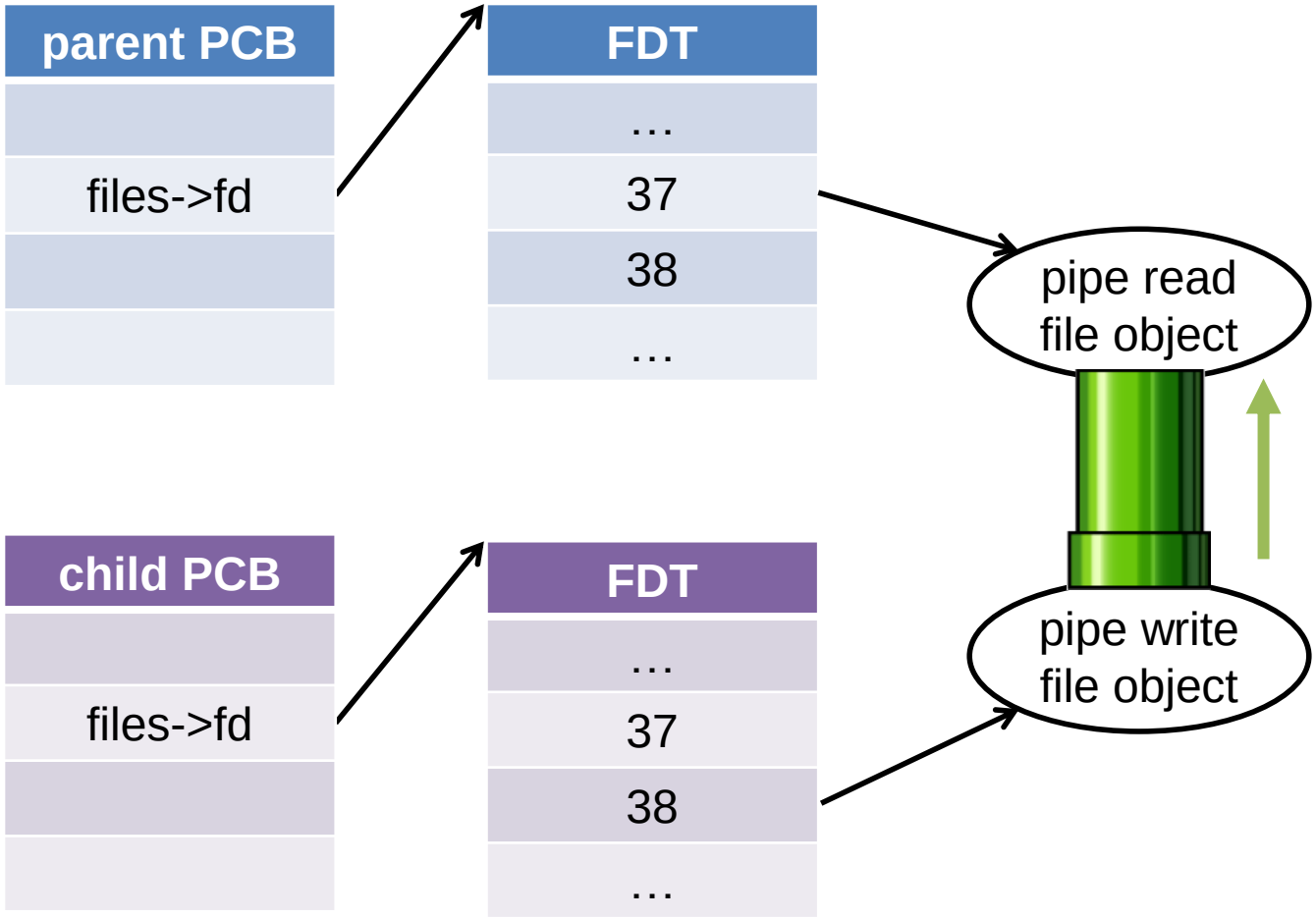
```
#include <unistd.h>
int pipe(int filedes[2]);
```

- פעולה: יוצרת pipe חדש עם שני FD (שני קצוות הצינור): אחד לקריאה מה-pipe ואחד לכתיבה אליו.
- פרמטרים:
  - filedes – מערך בן שני תאים.
  - ב-filedes[0] יאוחסן ה-FD **לקריאה** מה-pipe.
  - ב-filedes[1] יאוחסן ה-FD **לכתיבה** מה-pipe.
  - הכניסות המוקצות ל-pipe הן הראשונות הפנויות ב-FDT.
- ערך מוחזר: 0 בהצלחה ו- (-1) בכישלון.

pipes בלינוקס



pipes בלינוקס



שיתוף pipes

- ה–pipe הנוצר הינו פרטי לתהליך ואינו נגיש לתהליכים אחרים.
- שיתוף pipe יתבצע בדרך אחת בלבד – **בעזרת קשרי משפחה**:
  - תהליך אב יוצר pipe – שתי כניסות חדשות נוספות ל–FDT שלו.
  - תהליך האב יוצר תהליך בן באמצעות fork() – ה–FDT משוכפל לתהליך הבן.
  - כעת לשני התהליכים, האב והבן, יש גישה ל–pipe באמצעות ה–FD שלו.

- pipe הוא חד–כיווני ולכן האב והבן משחקים תפקידים שונים: האב קורא והבן כותב, או להפך.
  - האב והבן צריכים לסגור את ה–FD השני שאינו בשימוש.
- לאחר סיום השימוש ב–pipe מצד כל התהליכים (סגירת כל ה–FD) מפונים משאבי ה–pipe באופן אוטומטי.

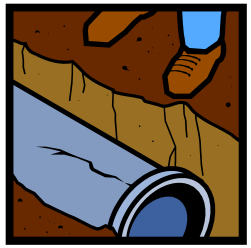
pipe – תכנית דוגמה

```
int main() {
  int my_pipe[2];
  char buff[6];
  pipe(my_pipe);

  if (fork() == 0) { // son
    close(my_pipe[0]);
    write(my_pipe[1], "Hello", 6);
  } else { // father
    close(my_pipe[1]);
    read(my_pipe[0], buff, 6);
    printf("Got from pipe: %s\n", buff);
  }
  return 0;
}
```

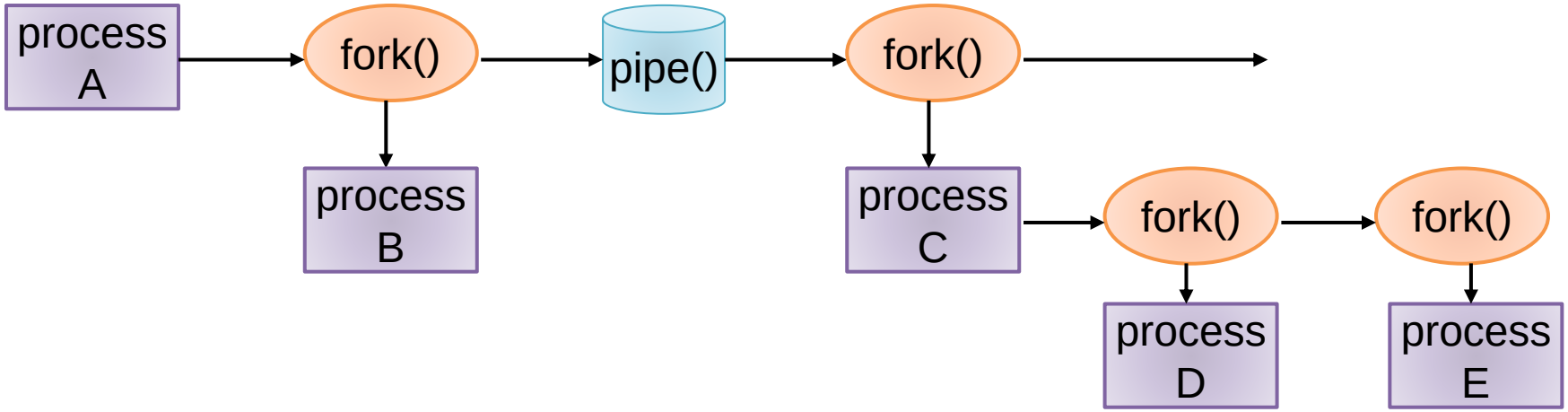
מה קורה אם האבא מתחיל לרוץ לפני הבן?

מהו הפלט של התכנית הנ"ל (בהנחה שכל קריאות המערכת מצליחות)?



pipes בלינוקס

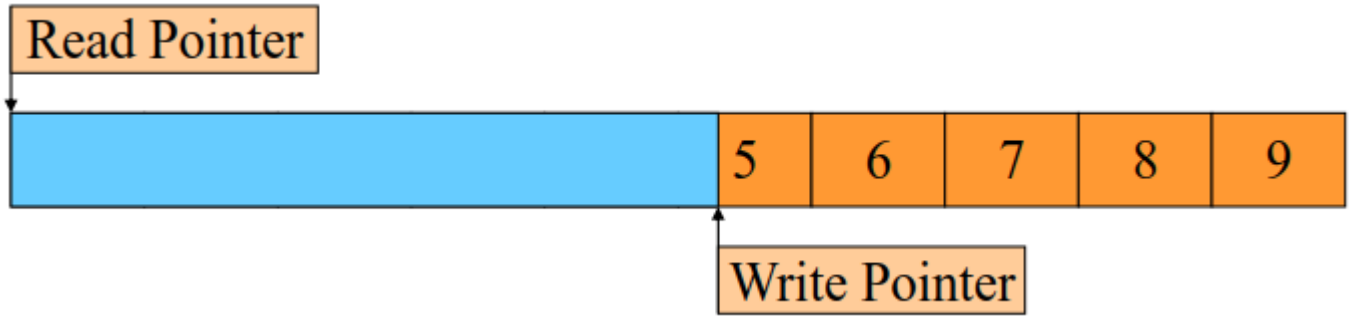
- למי מהתהליכים הבאים יש גישה ל-pipe שיוצר תהליך A?

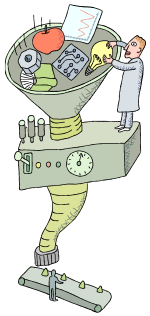


- תשובה: לכל התהליכים פרט ל-B.

קריאה וכתיבה ל-pipe (1)

- פעולות קריאה וכתיבה מתבצעות באמצעות read() ו-write() על ה-FDs של ה-pipe.
- ניתן להסתכל על pipe כמו על תור FIFO עם מצביע קריאה יחיד (להוצאת נתונים) ומצביע כתיבה יחיד (להכנסת נתונים).
  - כל קריאה (מכל תהליך שהוא) מה-pipe מקדמת את מצביע הקריאה. באופן דומה, כל כתיבה מקדמת את מצביע הכתיבה.





קריאה וכתיבה ל-pipe (2)

- **<u>קריאה מ-pipe תחזיר:</u>**
  - את כמות הנתונים המבוקשת אם היא נמצאת ב-pipe.
  - פחות מהכמות המבוקשת אם זו הכמות הזמינה ב-pipe בזמן הקריאה.
  - 0 (EOF) כאשר כל ה-write descriptors נסגרו וה-pipe ריק.
  - **תחסום** את התהליך אם יש כותבים (write descriptors) ל-pipe **וה-pipe ריק**. כאשר תתבצע כתיבה, יוחזרו הנתונים שנכתבו עד לכמות המבוקשת.
- **<u>כתיבה ל-pipe תבצע:</u>**
  - אם אין קוראים (read descriptors) – הכתיבה תיכשל והתהליך יקבל סיגנל בשם SIGPIPE (broken pipe error).
    - טיפול ברירת מחדל בסיגנל זה: הריגת התהליך.
  - ה-pipe מוגבל בגודלו (כ-64KB). אם יש מספיק מקום פנוי ב-pipe, תתבצע כתיבה של כל הכמות המבוקשת מיד.
  - אם אין מספיק מקום פנוי ב-pipe, הכתיבה **תחסום** את התהליך עד שהקוראים האחרים יפנו מקום וניתן יהיה לכתוב את כל הכמות הדרושה.

למה צריך לסגור קצוות מיותרים?

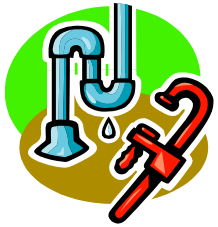
```
int main() {
  int fd[2];
  int grade;

  pipe(fd);
  if (fork() == 0) { // teacher
    close(fd[0]);
    do {
      grade = get_random_between(0, 100);
      write(fd[1], (void*)(&grade), sizeof(int));
    } while (grade != 0);

  } else { // student
    close(fd[1]);
    while (read(fd[0], (void*)(&grade), sizeof(int)) > 0) {
      if (grade == 100) break;
    }
    printf("Grade = %d\n", grade);
  }
  return 0;
}
```

מה יקרה אם נסיר את השורה הזו?

מה יקרה אם נסיר את השורה הזו?



קריאת המערכת dup()

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- <u>פעולה:</u> משכפלת את ה-FD שמספרו oldfd ל-FD **אחר** בטבלה.
  - עבור dup: ה-FD החדש הינו ה-FD הפנוי **בעל הערך הנמוך ביותר בטבלה.**
  - עבור dup2: ה-FD החדש הינו newfd, לאחר סגירה, אם היה פתוח.
  - לאחר פעולה מוצלחת, oldfd וה-FD החדש מצביעים לאותו file object.
- <u>פרמטרים:</u>
  - oldfd – ה-FD המיועד להעתקה – חייב להיות פתוח לפני ההעתקה.
- <u>ערך מוחזר:</u>
  - בהצלחה, מוחזר ה-FD החדש.
  - בכישלון מוחזר (1-).

1

שאלה

הכוונת קלט/פלט באמצעות pipes

- ממשו בעזרת dup או dup2 ו-pipe קטע קוד קצר המדמה את הפקודה הבאה ב-shell:

```
>> ls | more
```

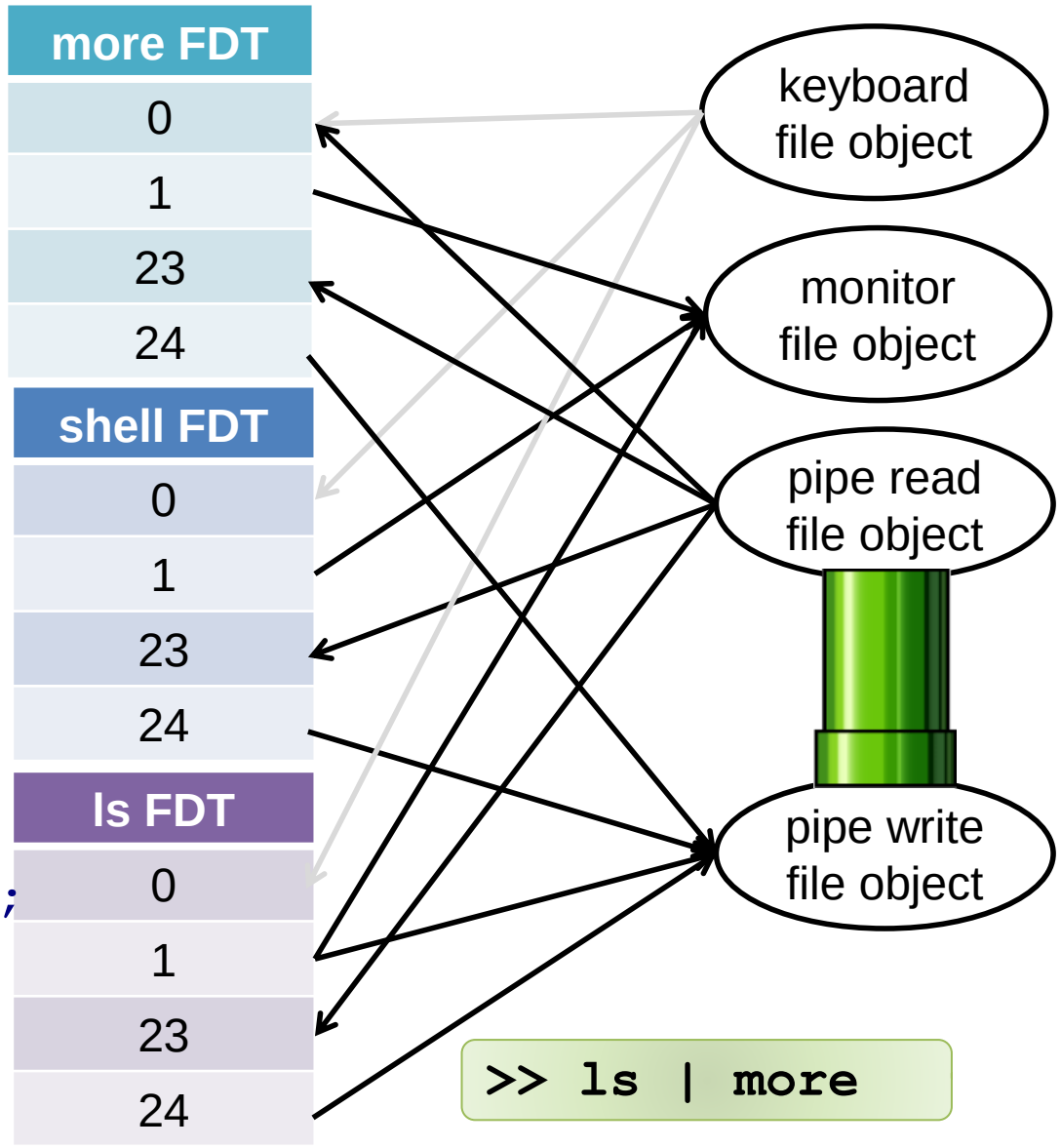
- תרגום:
  - על ה-shell להתחיל שני תהליכים המריצים את ls ו-more.
  - יש לבצע redirection בין ה-STDOUT של תהליך ה-ls ל-STDIN של תהליך ה-more באמצעות מנגנון ה-pipe.

הכוונת קלט/פלט באמצעות pipes – פתרון

```
// shell Code
int fd[2];
pipe(fd);

if (fork() == 0) {
   // second child
   dup2(fd[0],0);
   close(fd[0]);
   close(fd[1]);
   execv("/bin/more",…);
}
close(fd[0]);
close(fd[1]);
```

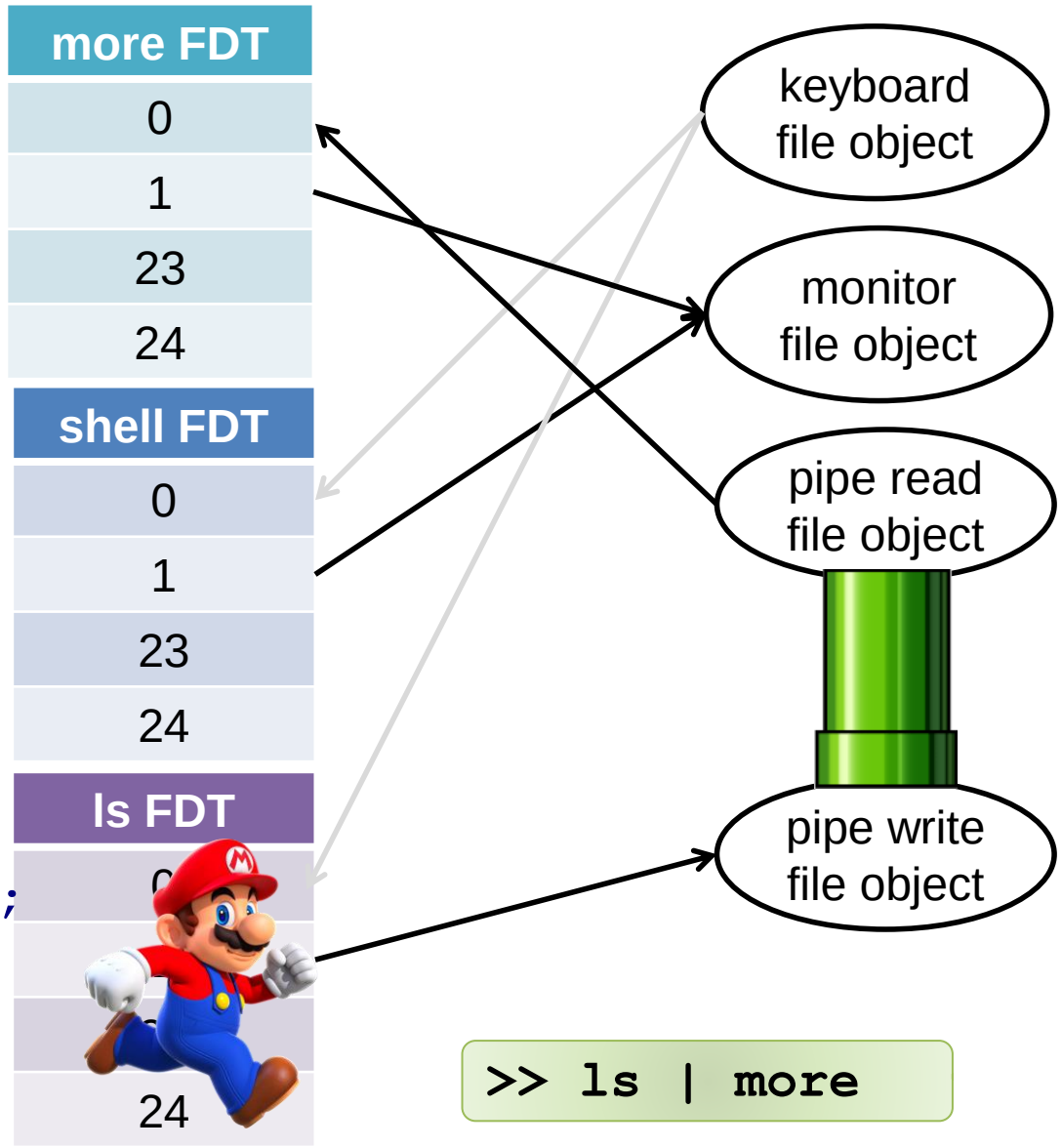
שימו לב: תזמון הפעולות יכול להיות **שונה** מהסדר שבו ההנפשות בשקף זה מוצגות (לדוגמה אם הבן השני התחיל לרוץ לפני הראשון)



הכוונת קלט/פלט באמצעות pipes – פתרון

```
// shell Code
int fd[2];
pipe(fd);

if (fork() == 0) {
   // first child
   dup2(fd[1],1);
   close(fd[0]);
   close(fd[1]);
   execv("/bin/ls",…);
}
if (fork() == 0) {
   // second child
   dup2(fd[0],0);
   close(fd[0]);
   close(fd[1]);
   execv("/bin/more",…);
}
close(fd[0]);
close(fd[1]);
```




```
>> ls | more
```

FIFOs (או named pipes)

- FIFO הוא למעשה pipe בעל "שם" במערכת הקבצים שדרכו יכולים כל התהליכים במכונה לגשת אליו – **pipe "ציבורי"**.
  - שם ה–FIFO הוא כשם קובץ במערכת הקבצים, למשל: /home/yossi/myfifo .
  - שימו לב: FIFO הוא קובץ **למרות שאיננו נשמר על הדיסק**.
  - pipes הם חסרי שם ולכן נקראים לפעמים anonymous FIFO.
- השימוש העיקרי של FIFO (או של כל אובייקט תקשורת בעל "שם") הוא כאשר תהליכים רוצים לתקשר דרך ערוץ קבוע מראש **מבלי שיהיו ביניהם קשרי משפחה**.
  - למשל, כאשר תהליכי לקוח צריכים לתקשר עם תהליך שרת.
- FIFO נוצר ע"י קריאת המערכת mkfifo(), אשר מעניקה לו את שמו.

קריאת המערכת mkfifo()

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname,
      mode_t mode);
```

- פעולה: יוצרת FIFO המופיע במערכת הקבצים במסלול pathname והרשאות הגישה שלו הן mode.
- פרמטרים:
  - pathname – שם ה-FIFO וגם המסלול לקובץ במערכת הקבצים.
  - mode – הרשאות הגישה ל-FIFO שנוצר. ניתן להכניס ערך 0777 (777 אוקטאלי) כדי לאפשר הרשאות מלאות.
- ערך מוחזר: 0 בהצלחה, (-1) בכישלון.

תקשורת באמצעות FIFO

- בניגוד ל-pipe, FIFO הינו ערוץ תקשורת **דו-כיווני** ובעל FD יחיד (ניתן לבצע הן קריאה והן כתיבה דרך אותו FD)
  - ניגשים ל-FIFO באמצעות פקודת open(). כותבים וקוראים באמצעות read/write.

- תהליך שפותח את ה-FIFO לקריאה בלבד נחסם עד שתהליך נוסף יפתח את ה-FIFO לכתיבה, וההפך.
  - פתיחת ה-FIFO לכתיבה וקריאה (O_RDWR) איננה חוסמת.
  - חוקי הקריאה והכתיבה עובדים באופן דומה ל-pipe.

ניקוי שאריות

- כאובייקט בעל שם במערכת הקבצים, FIFO אינו מפונה אוטומטית לאחר שהמשתמש האחרון בו סוגר את הקובץ.
- כן יש לפנותו בצורה מפורשת באמצעות פקודות או קריאות מערכת למחיקת קבצים (למשל, פקודת rm או קריאת המערכת ()unlink).

אז מה היה לנו היום:

- **המטרה**: לתקשר בין תהליכים.
  - signals (דיווח לתהליך על אירוע).
  - files (בין היתר בשביל תקשורת בין תהליכים).
- Linux – Everything is a File
  - FDT.
  - file objects.
  - שיתוף אחרי fork.
- קבצים מיוחדים לתקשורת בין תהליכים:
  - pipe
  - FIFO
  - sockets (לא דיברנו על זה היום).
- קריאות מערכת:
  - כאלו שכבר הכרתם: open, close, write, read
  - כאלו שלא הכרתם: signal, kill, pipe, mkfifo.