

# מערכות הפעלה

5

תהליכים

# יצירת תהליך ביוניקס

תכנית 1

```
---  
pid = fork();  
if (pid > 0)  
    wait()  
---  
if (pid == 0)  
    exec()
```

תהליך הורה

```
---  
pid = fork();  
if (pid > 0)  
    wait()  
---  
if (pid == 0)  
    exec()
```

תהליך ילד

```
---  
pid = fork();  
if (pid > 0)  
    wait()  
---  
if (pid == 0)  
    exec()
```

תכנית 1

```
---  
pid = fork();  
if (pid > 0)  
    wait()  
---  
if (pid == 0)  
    exec()
```

תכנית 2

```
---  
---  
---  
---  
---  
---  
---  
---
```

# יצירת תהליך

- במערכת יוניקס, תהליך חדש נוצר על ידי קריאת המערכת `fork()`.
- התהליך החדש שנקרא **ילד** יורש את ההרשאות והמשאבים של התהליך היוצר שנקרא **הורה**.
- כל קטעי הזיכרון של ההורה (קוד, נתונים, מחסנית) **מועתקים** לילד, גם הקבצים שההורה פתח פתוחים עבור הילד.
- לאחר הקריאה ל- `fork()`, שני התהליכים ממשיכים לבצע החל מהפקודה שאחרי ה- `fork()`.
- הערך המוחזר מהקריאה ל- `fork()` הוא אפס עבור הילד ומספר התהליך של הילד עבור ההורה.
- בדרך כלל הילד יבצע `exec()` כדי להחליף את התכנית שהוא מריץ בתכנית אחרת, ואז:
  - מערכת ההפעלה תשחרר את קטעי הזיכרון של התכנית הקודמת.
  - תקצה קטעי זיכרון עבור התכנית החדשה.
  - תטען את התכנית מהדיסק לזיכרון.
  - הקריאה ל- `exec()` לא חוזרת (הקוד שקרא ל- `exec()` לא קיים).

## המתנה לסיום תהליך - wait()

- לאחר `fork()`, ההורה והילד מבצעים את התכניות במקביל.
- אם ההורה רוצה להמתין לסיום תהליך הילד, הוא יבצע `wait()`.
- `wait()` יעביר את ההורה למצב שינה, כשהילד יסיים ההורה יעבור למצב מוכן לרוץ.
- הערך המוחזר מ- `wait()` הוא מספר תהליך הילד שסיים.
- גם סטטוס הסיום של הילד מוחזר, `wait()` מעביר מצביע למשתנה שיכיל את הסטטוס.

```
pid = wait(&status);
```

- סטטוס הסיום נשמר ב- PCB.
- כאשר תהליך מסיים (בעקבות `exit()` או `exception`), מערכת ההפעלה משחררת את קטעי הזיכרון שתפס וסוגרת את הקבצים שפתח.
- אך ה- PCB עדיין נשמר בזיכרון כי הוא מכיל את סטטוס הסיום.
- לאחר שהורה קרא ל- `wait()`, ה- PCB של הילד ישוחרר.

# fork(), exec() and wait()

```
int main()
```

```
{
```

```
    int pid;
```

```
    int status;
```

```
    pid = fork();
```

```
    if (pid == 0) { // child
```

```
        execvp("ls", "ls", "-l", "a.out", NULL);
```

```
    }
```

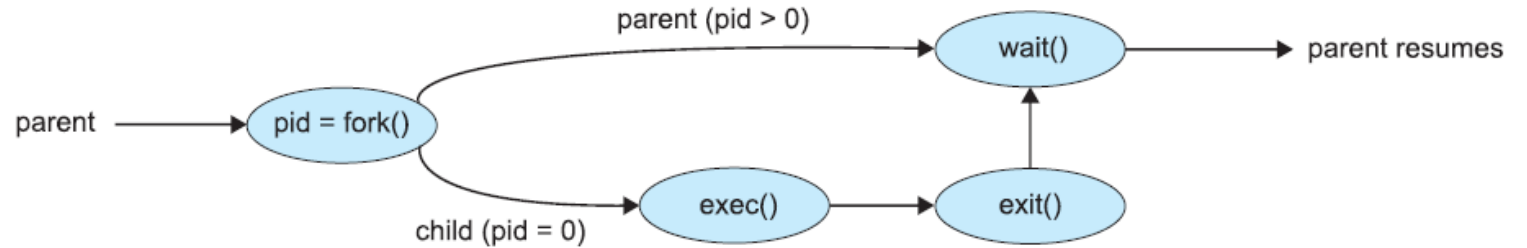
```
    else if (pid > 0) { // parent
```

```
        wait(&status);
```

```
        printf("exit status of child is: %d\n", status >> 8);
```

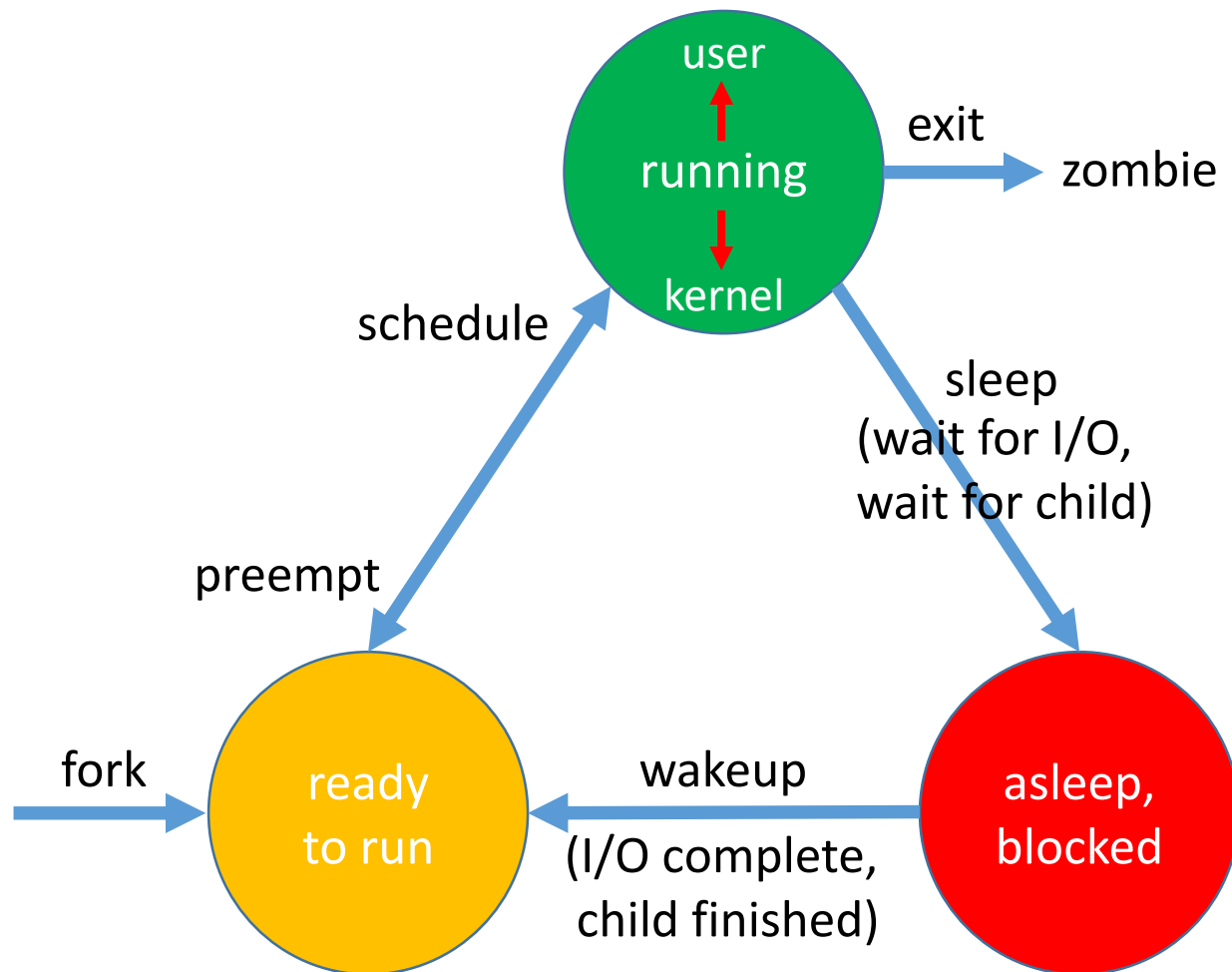
```
    }
```

```
}
```



wait.c

# מצבי תהליך



1. **ריצה** – תהליך שכעת מתבצע על ידי המעבד, אם יש מעבד אחד רק תהליך אחד יכול להיות במצב ריצה.

2. **מוכן לריצה** – תהליך שלא מחכה למאורע כלשהו ומוכן לרוץ כאשר מערכת ההפעלה תיתן לו את המעבד.

3. **חסום (ישן)** – תהליך שמחכה למאורע ולא יכול להמשיך עד שאותו מאורע יקרה, דוגמה: המתנה לקלט/פלט.

# הודעות לתהליך (signals)

**הודעה לתהליך (signal)** היא הודעה קצרה אודות מאורע, לכל הודעה (signal) יש מספר וסמל.

- ראינו שכאשר מתרחשת פסיקה, ריצת התכנית נפסקת והמעבד עובר לבצע קוד במערכת ההפעלה.

- אם הפסיקה נגרמה מבצוע התכנית (פסיקה פנימית), מערכת ההפעלה תשלח הודעה לתהליך.

- ישנם מקרים נוספים בהם המערכת תשלח הודעה לתהליך, כפי שיפורט להלן.

## סוגי הודעות (signals):

1. כאמור, פסיקה שנגרמה מבצוע התכנית:

- חלוקה ב-0 (SIGFPE), גישה לא חוקית לזיכרון (SIGSEGV).

2. מאורע שלא גרם לפסיקת חומרה:

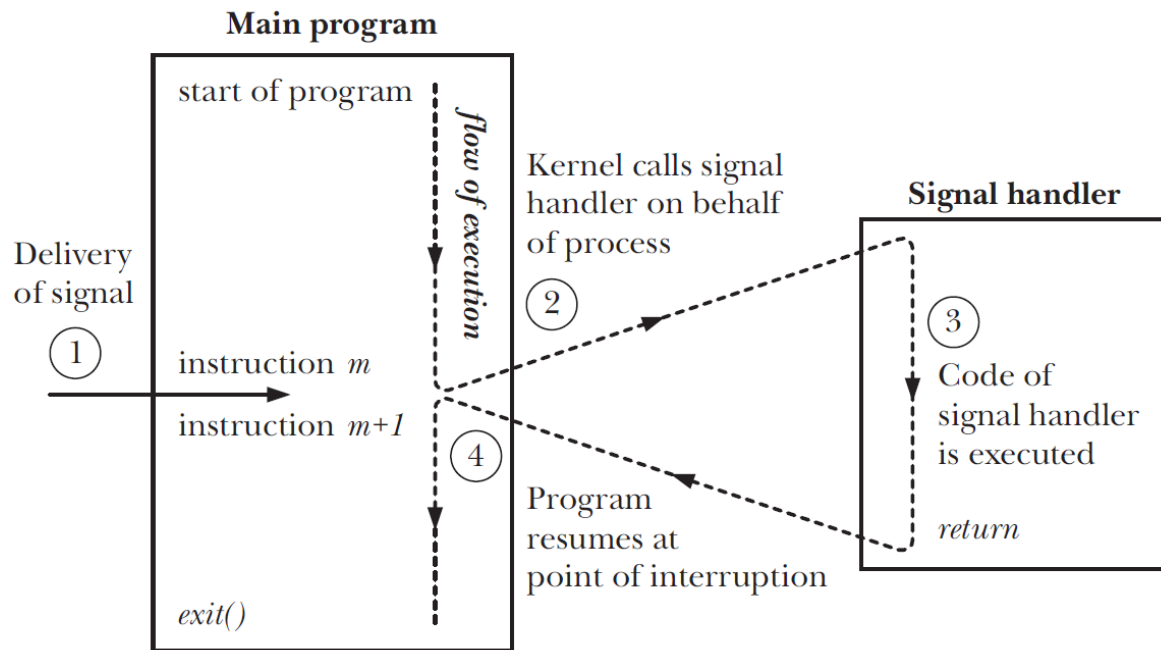
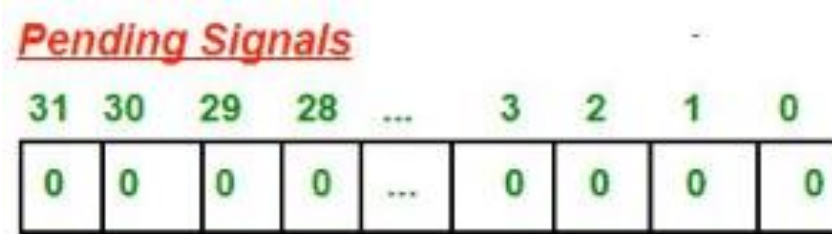
- ילד סיים כאשר ההורה לא חיכה לו (SIGCHLD), בקשה להודעה כעבור זמן (SIGALRM).

3. מאורע שנגרם מהקשה על מקש או צרוף מקשים:

- Control-C (SIGINT), Control-Z (SIGTSTP).

4. הפקודה kill שולחת signal (SIGKILL) לתהליך - בתנאי שהתהליך השולח מורשה.

# הודעות לתהליך (signals)



- כאשר נשלחת הודעה לתהליך (generation), היא רק נרשמת ב- PCB של התהליך בשדה .signals
- ההודעה נמסרת (delivery) לתהליך כאשר הוא עובר למצב ריצה.
- כאשר התהליך מקבל את ההודעה, הוא מבצע את אחת מהאפשרויות הבאות:
  1. **מתעלם** מההודעה - signal is ignored.
  2. **מפסיק לרוץ** - process is terminated.
  3. **מריץ פונקציה** שמטפלת בהודעה - signal handler.



# תקשורת בין תהליכים באמצעות זיכרון משותף או תיבת דואר

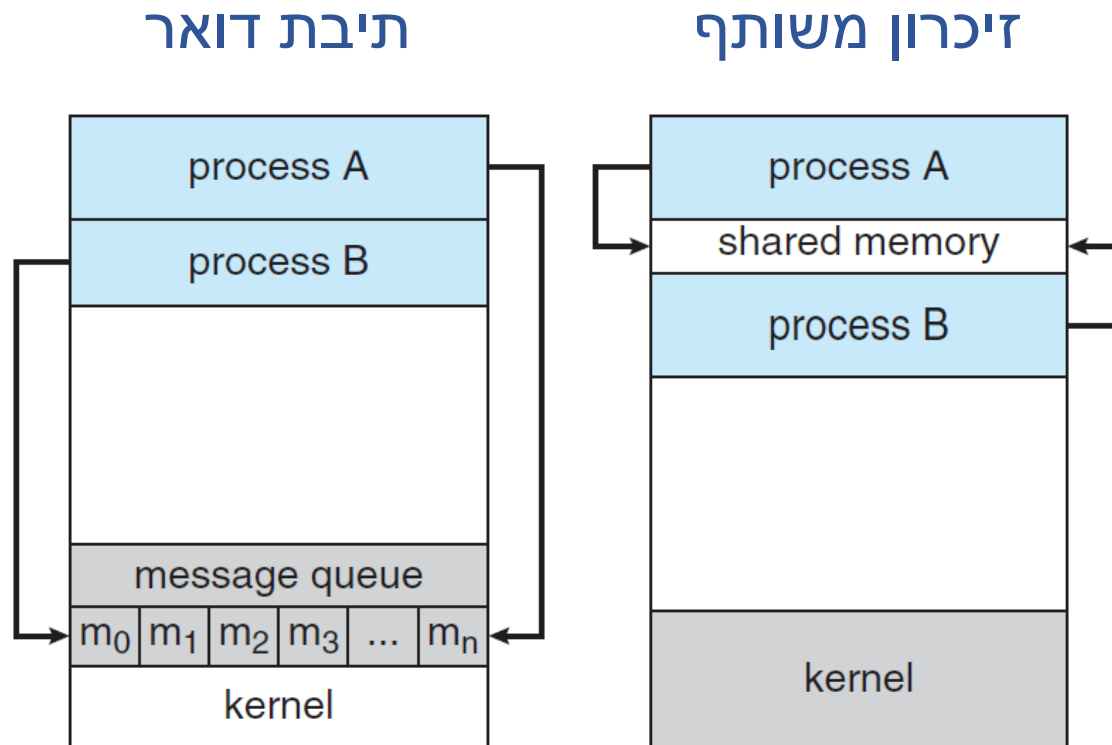
- כאשר תהליכים רוצים לשתף מידע, הם יכולים לעשות זאת בשתי דרכים:

## • זיכרון משותף - Shared Memory

- דומה ללוח הודעות.
- מהיר – אין צורך בקריאת מערכת עבור קריאה או כתיבה.
- יש צורך בסנכרון.
- מתאים להודעות ארוכות.

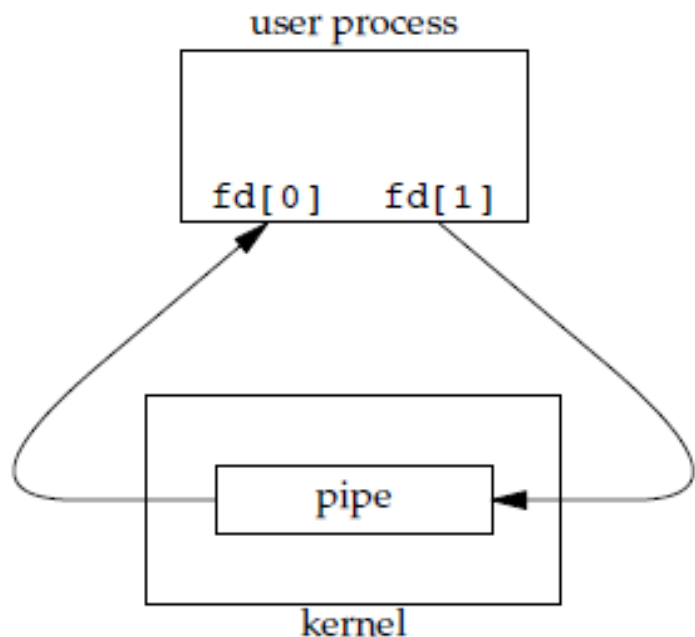
## • תיבת דואר - Message Passing

- דומה לאימייל.
- איטי - יש צורך בקריאת מערכת עבור כל קריאה או כתיבה.
- ויש צורך להעתיק פעמיים את ההודעה.
- מתאים להודעות קצרות.

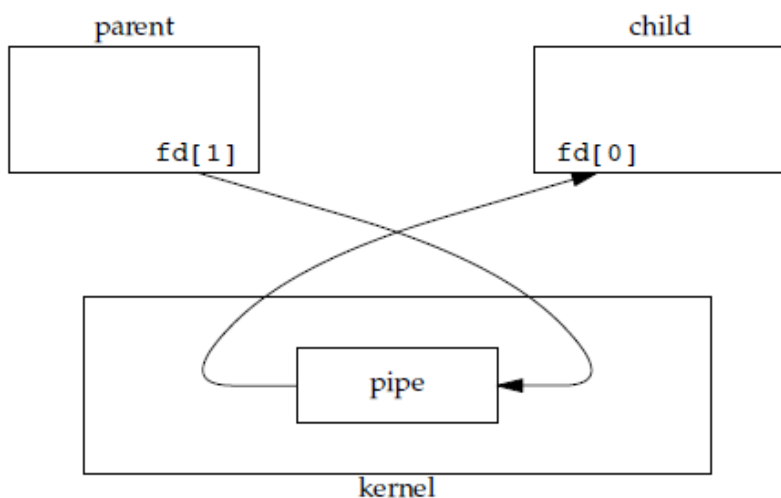
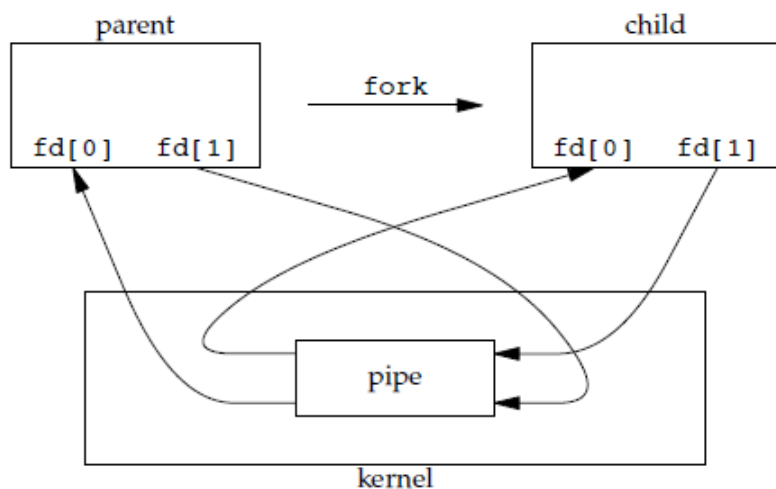


# תקשורת בין תהליכים באמצעות pipe

- `pipe` הוא מימוש של יוניקס לתקשורת בין יצרן וצרכן באמצעות זיכרון משותף.
  - כדי ליצור `pipe` משתמשים בקריאת המערכת `pipe(fd)`.
    - מערכת ההפעלה תקצה קטע זיכרון בקרנל (4K) לצורך העברת המידע.
    - מערכת ההפעלה תחזיר שני `file descriptors`, אחד ישמש לקריאה (צרכן) ואחד לכתיבה (יצרן).
    - בקריאה ל-`pipe` הפרמטר הוא מצביע למערך שיש בו שני מספרים: `int fd[2]`.
    - לאחר הקריאה, `fd[0]` יכיל את `file descriptor` של צד הקריאה, ו-`fd[1]` את `file descriptor` של צד הכתיבה.
- ```
int fd[2];  
pipe(fd);
```



# סגירת file descriptors בשימוש



- כדי ש-`pipe` ישמש לתקשורת בין תהליכים, לאחר שתהליך יצר `pipe` הוא יבצע `fork`.

- ואז ההורה והילד יהיו שותפים ב-`file descriptors` של הקריאה והכתיבה ויוכלו להעביר מידע.

- קריאה מ-`pipe` ריק כאשר יש כותב, תעביר את הקורא להמתנה במצב שינה.

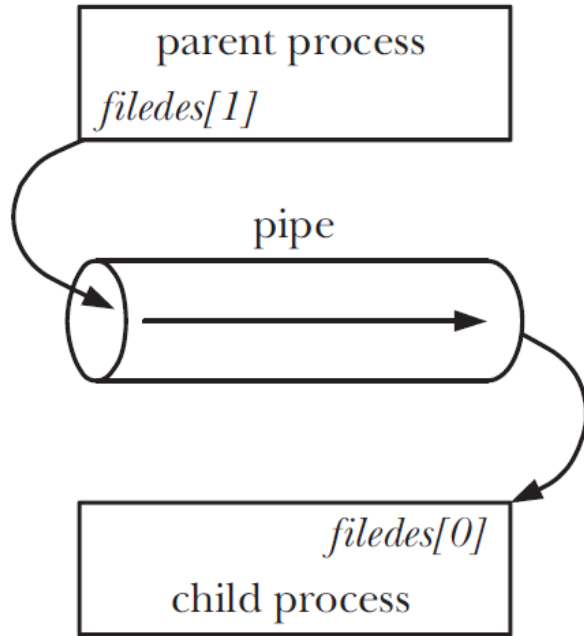
- אם הכותב סיים וסגר את צד הכתיבה ואין כותב אחר, קריאה מ-`pipe` ריק תחזיר סוף קובץ והקורא ימשיך לרוץ.
- לכן חשוב שהקורא יסגור את צד הכתיבה, כדי שיקבל סוף קובץ כאשר הכותב סיים ולא יעבור למצב שינה.

- כתיבה ל-`pipe` מלא כאשר יש קורא, תעביר את הכותב להמתנה במצב שינה.

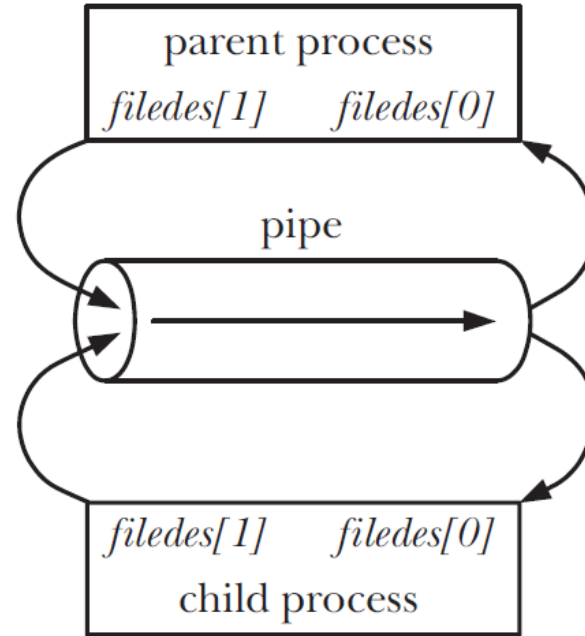
- כתיבה ל-`pipe` מלא כאשר הקורא סיים וסגר את צד הקריאה ואין קורא אחר, תשלח לכותב הודעה `SIGPIPE`.

- לכן חשוב שהכותב יסגור את צד הקריאה, כדי שיקבל הודעה כאשר הקורא סיים ולא יעבור למצב שינה.

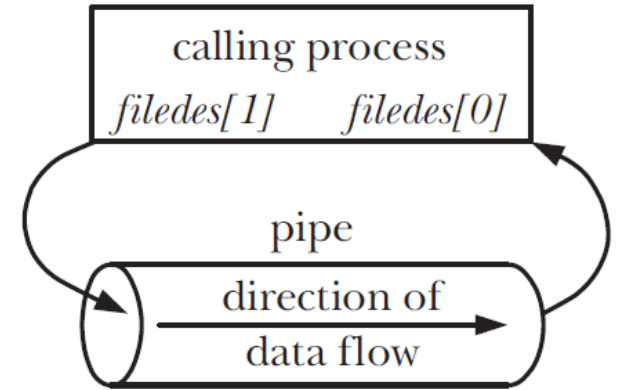
# סיכום השלבים ביצירת pipe



3. אחרי סגירת הצדדים  
שאינם בשימוש



2. אחרי ביצוע `fork()`



1. אחרי ביצוע `pipe()`

# תכנית שמעבירה הודעה באמצעות pipe

```
#define BUFFER_SIZE 25
int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];                /* two file descriptors */
    int pid;
    if (pipe(fd) < 0) { /* create the pipe */
        fprintf(stderr, "Pipe failed");
        return 1;
    }
```

# תכנית שמעבירה הודעה באמצעות pipe

```
pid = fork();          /* fork a child process */
if (pid > 0) {          /* parent process */
    close(fd[0]);       /* close the unused read end */
    write(fd[1], write_msg, strlen(write_msg)+1);
    close(fd[1]);       /* close the write end */
}
else {                  /* child process */
    close(fd[1]);       /* close the unused write end */
    read(fd[0], read_msg, BUFFER_SIZE);
    printf("read %s", read_msg);
    close(fd[0]);       /* close the read end */
}
return 0;
}
```

# שימוש ב-pipe לחיבור מסננים (filters)

- פקודות יוניקס כתובות כך שניתן לחבר אותם בשרשרת כדי לפתור בעיות יותר מורכבות:

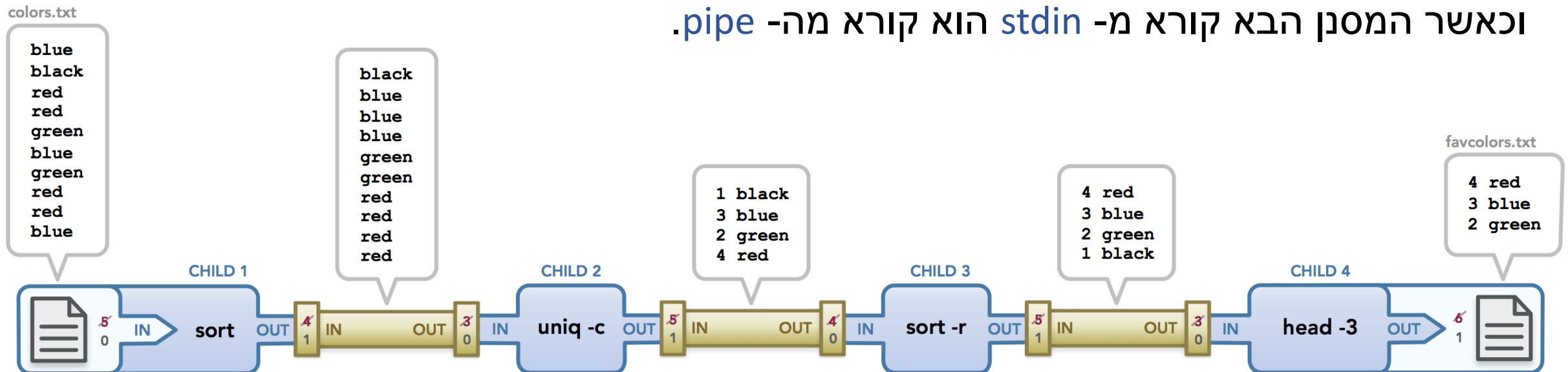
- `cat, cut, grep, head, sort, uniq, tail, wc`

- בדוגמה למטה, מחברים פקודות כדי למצוא מתוך רשימת צבעים את שלושת הצבעים השכיחים ביותר.

- כדי שהפתרון יהיה מהיר, שיתוף המידע בין התכניות הוא באמצעות זיכרון משותף (ולא קובץ).

- הפקודות המחוברות נקראות מסננים (`filters`), וכדי שנוכל לחבר ביניהם כל מסנן צריך לקרוא מ-`stdin` (מקלדת) ולכתוב ל-`stdout` (מסך).

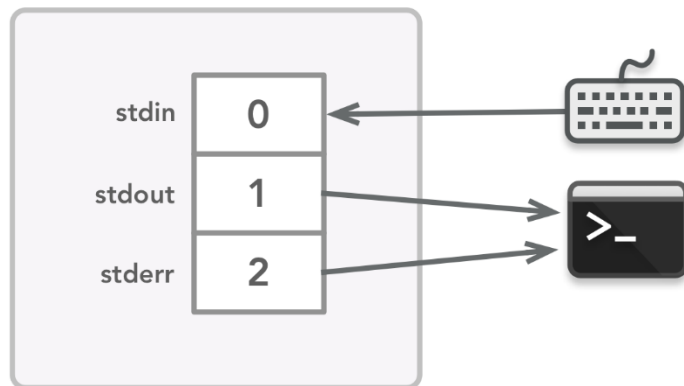
- החיבור מתבצע באמצעות `pipe`, וצריך לסדר שכאשר מסנן כותב ל-`stdout` זה נכתב ל-`pipe`, וכאשר המסנן הבא קורא מ-`stdin` הוא קורא מה-`pipe`.



# שימוש ב-pipe לחיבור מסננים

## 1. לפני pipe()

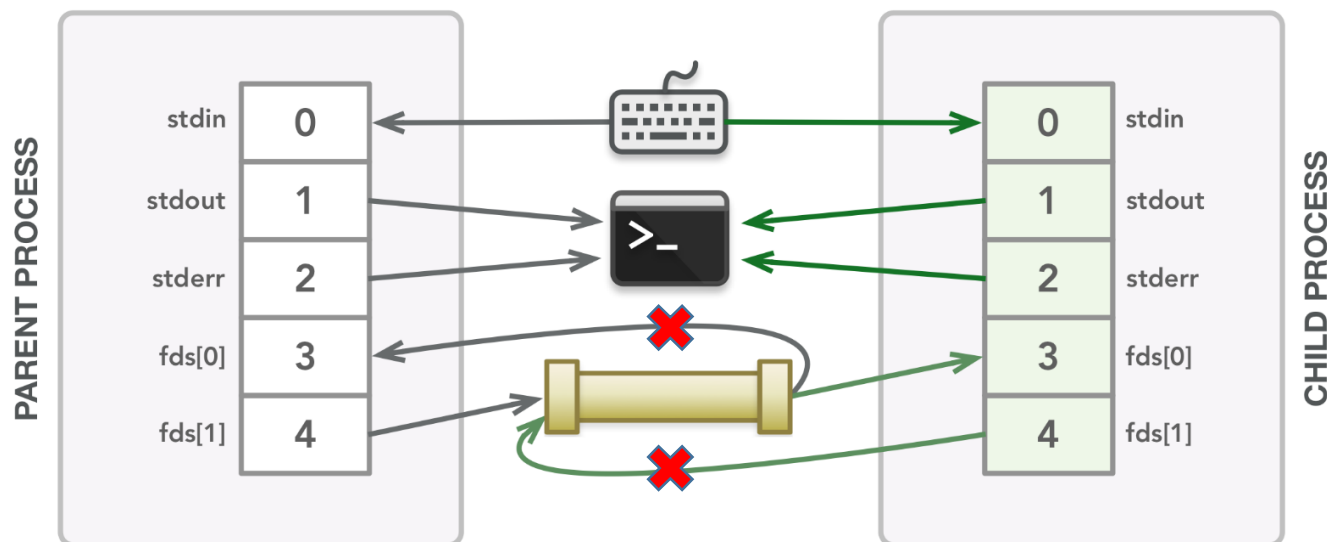
default state



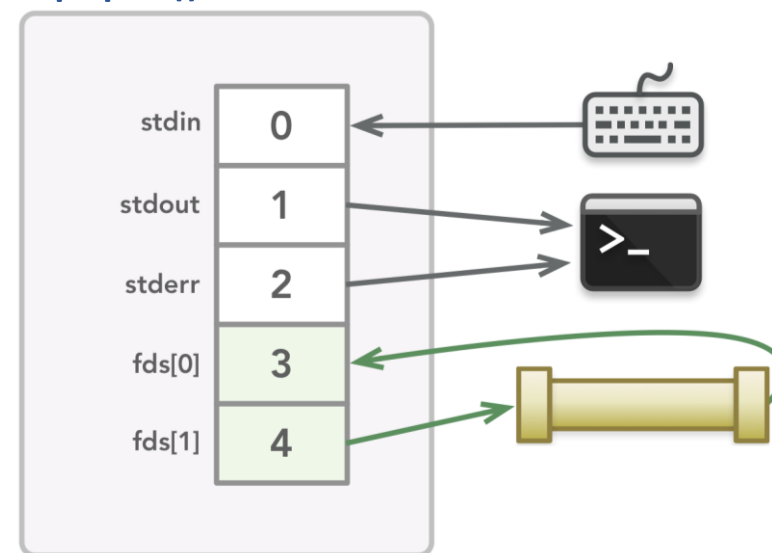
בעיה:

- המקלדת והמסך פתוחים באמצעות file descriptors 0 ו-1.
- מערכת ההפעלה תקצה file descriptors גבוהים יותר לפתיחת ה-pipe (3 ו-4).
- אם כן איך הקריאה מ-0 (stdin) והכתיבה ל-1 (stdout) תקרא ותכתוב ל-pipe (3 ו-4)?

## 3. אחרי fork() וסגירת fds שאינם בשימוש



## 2. אחרי pipe()





# שימוש ב-pipe לחיבור מסננים

פתרון:

- נעתיק באמצעות `dup` את ה-`file descriptors` (מצביעים) של ה-pipe ל-`file descriptors` של `stdin` ו-`stdout`.

כותב יבצע:

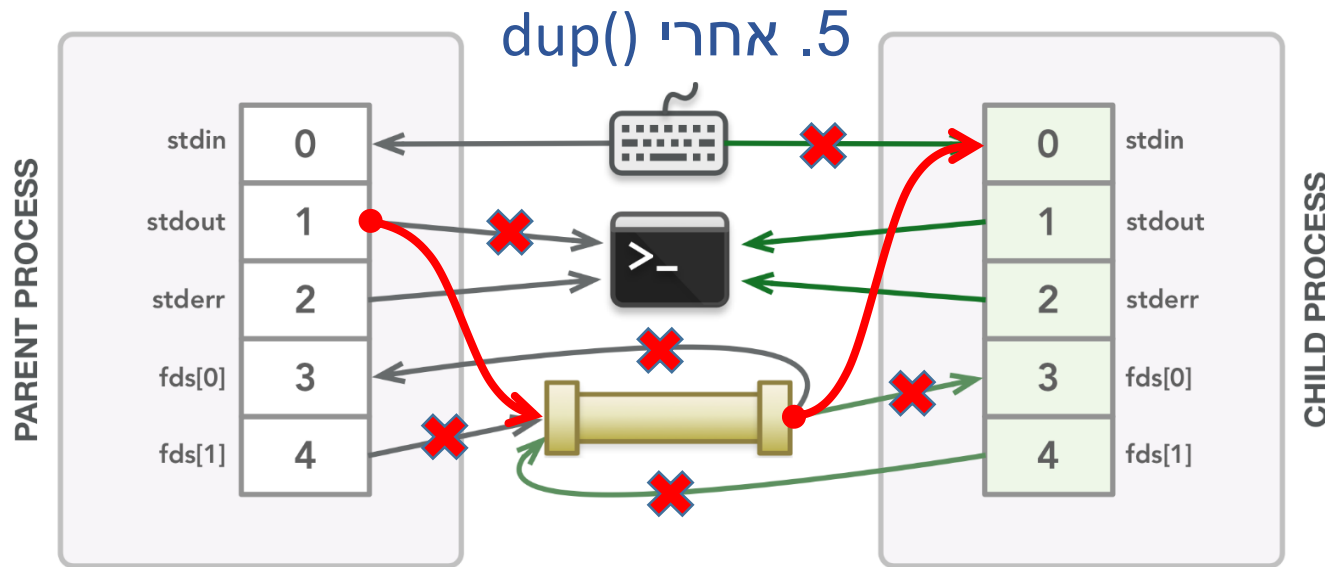
```
dup2 (4 , 1) ;
```

```
close (4) ;
```

קורא יבצע:

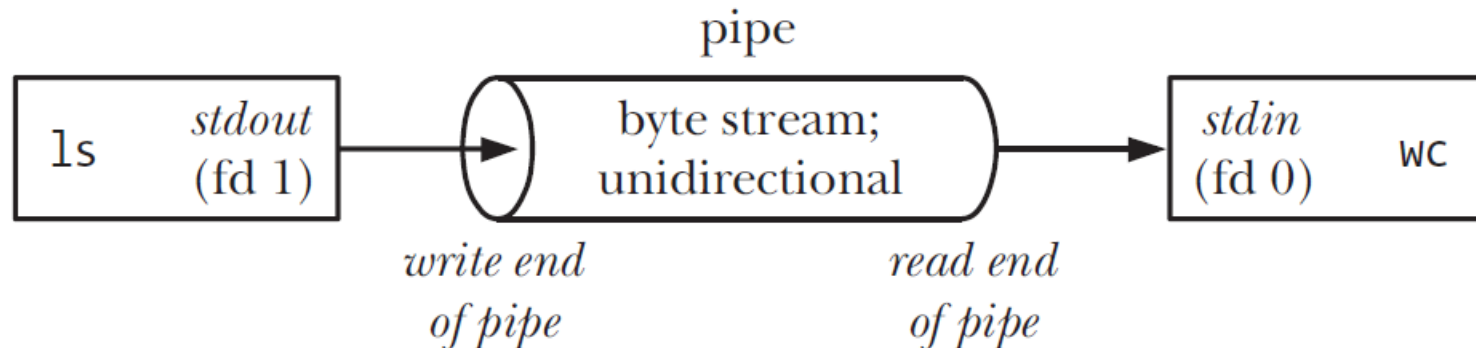
```
dup2 (3 , 0) ;
```

```
close (3) ;
```



# ניתוב הפלט של ls לקלט של wc

```
int main(int argc, char *argv[]) {
    int fd[2]; // Pipe file descriptors
    if (pipe(fd) == -1) // ... error
    switch (fork()) {
        case -1: // ... error
        case 0: // First child: exec 'ls'
            if (close(fd[0]) == -1) // ... error
            if (dup2(fd[1], STDOUT_FILENO) == -1) // ... error
            if (close(fd[1]) == -1) // ... error
            execlp("ls", "ls", NULL); // Writes to pipe
        default: // Parent falls through to create next child
            break;
    }
}
```

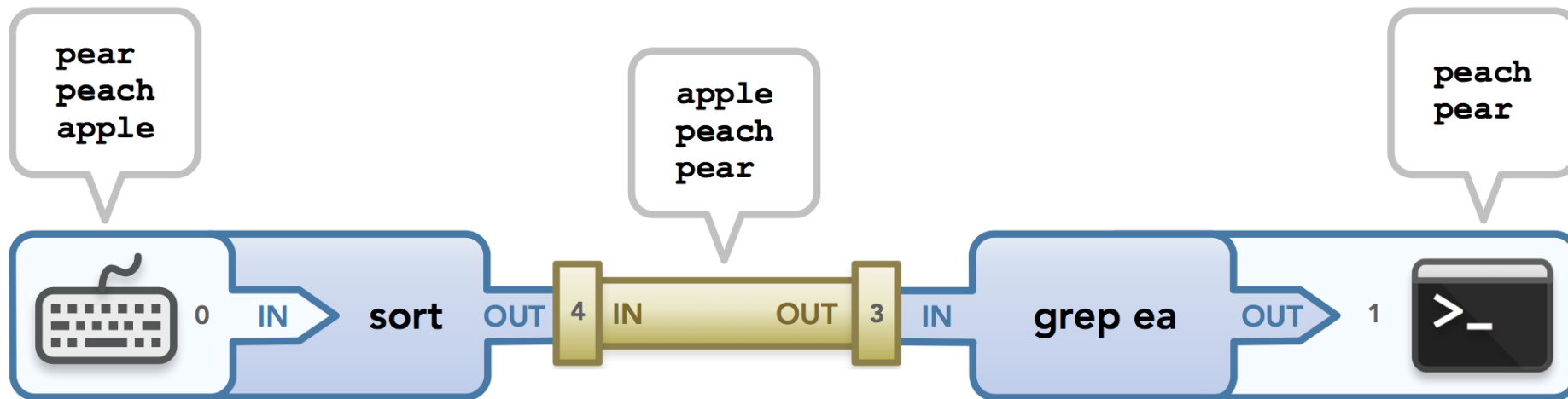


## ניתוב הפלט של ls לקלט של wc

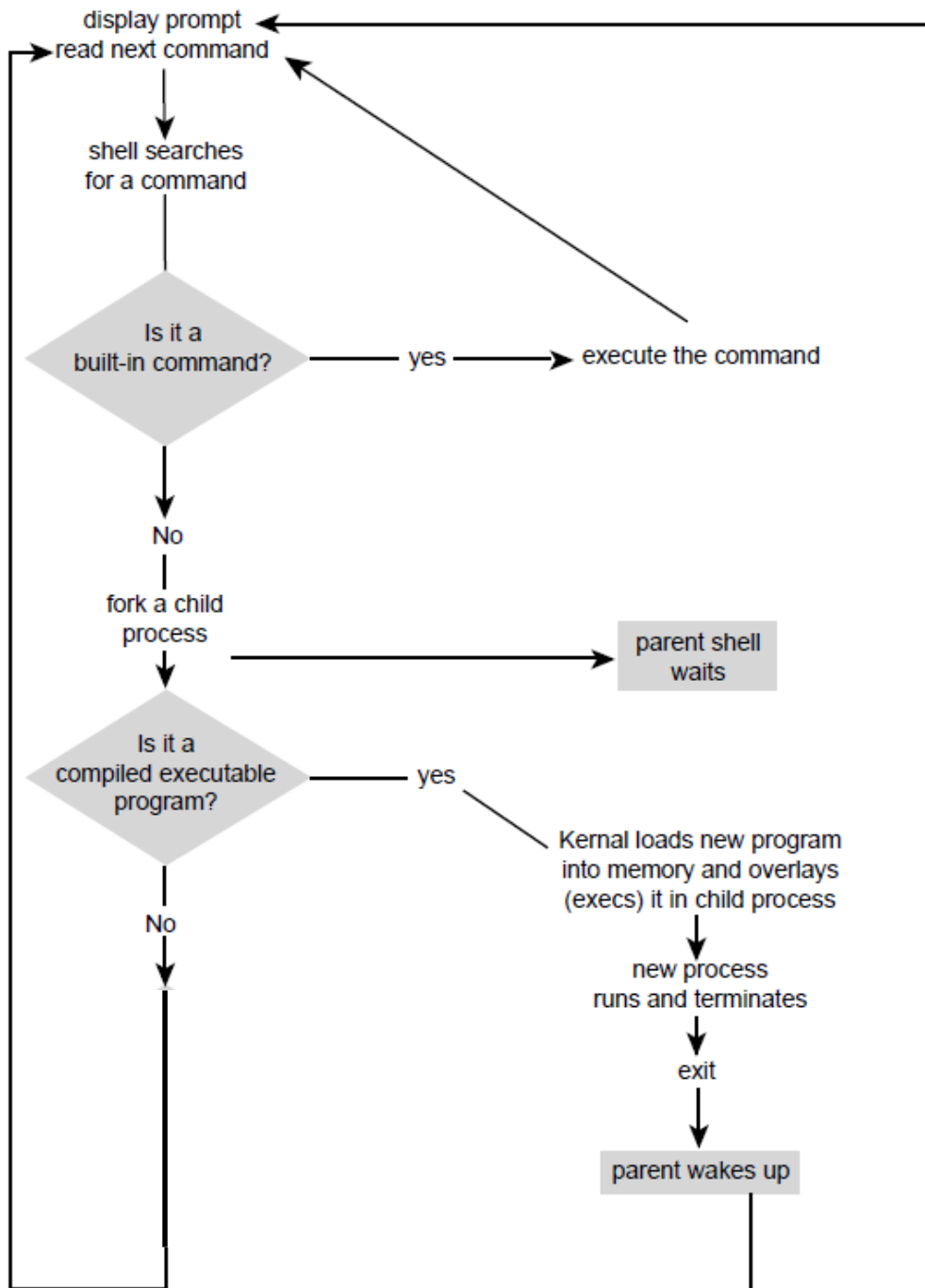
```
switch (fork()) {
    case -1: // ... error
    case 0:  // Second child: exec 'wc'
        if (close(fd[1]) == -1) // ... error
        if (dup2(fd[0], STDIN_FILENO) == -1) // ... error
        if (close(fd[0]) == -1) // ... error
        execlp("wc", "wc", "-l", NULL); // Reads from pipe
    default: // Parent falls through
        break;
}
if (wait(NULL) == -1) // Parent waits for child
if (wait(NULL) == -1) // Parent waits for child
}
```

# ניתוב קלט ופלט ב-shell

- ">" מנתב את הפלט של הפקודה לקובץ במקום למסך: `ls -l > temp`
- ">>" מוסיף את הפלט של הפקודה לקובץ: `ls -l >> temp`
- "<" הפקודה קוראת מקובץ במקום לקרוא מהמקלדת: `wc -l < myfile`
- "|" מנתב את הפלט של פקודה לקלט של פקודה אחרת: `sort | grep ea`
- הסימן "|" נקרא `pipe`, ה-shell מממש את "|" באמצעות `pipe` של יוניקס.



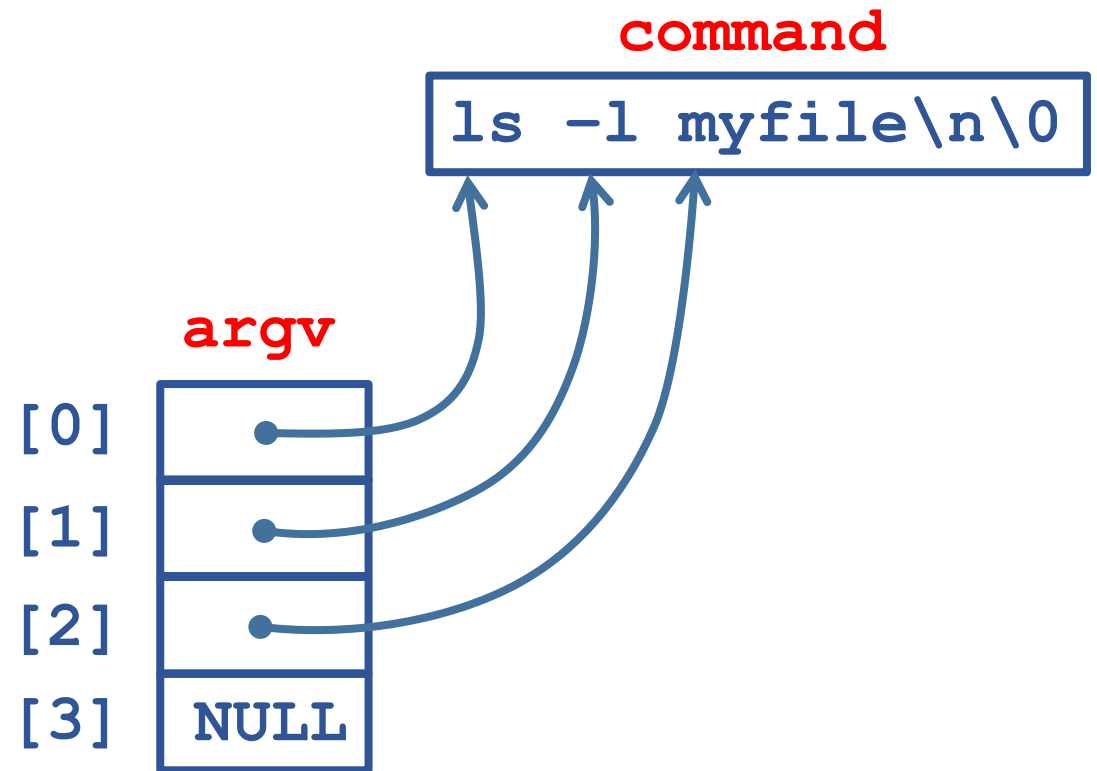
# מבנה Shell



```
while (TRUE) {  
    display_prompt();  
    read_command();  
    pid = fork();  
    /* Parent */  
    if (pid > 0) /* Parent */  
        /* wait for child */  
        wait(&status);  
    else /* Child */  
        /* Execute Command */  
        exec(command);  
}
```

# shell עם ניתוב פלט

```
int main() {  
    char *argv[10]; char command[1024]; char *token, *outfile; int i, fd, redirect;  
    while (1) {  
        printf("hello: ");  
        fgets(command, 1024, stdin);  
        command[strlen(command) - 1] = '\\0';  
        i = 0;  
        // parse command line  
        token = strtok (command, " ");  
        while (token != NULL) {  
            argv[i] = token;  
            token = strtok (NULL, " ");  
            i++;  
        }  
        argv[i] = NULL;  
        if (argv[0] == NULL) // Is command empty  
            continue;  
    }  
}
```



# shell עם ניתוב פלט

```
if (i > 1 && ! strcmp(argv[i - 2], ">")) { // redirection of output ?  
    redirect = 1;  
    argv[i - 2] = NULL;  
    outfile = argv[i - 1];  
}  
else  
    redirect = 0;  
// put here internal commands; continue;  
if (fork() == 0) { // child  
    if (redirect) { // redirection of output ?  
        fd = creat(outfile, 0660); // if exists truncates  
        dup2(fd, STDOUT_FILENO);  
        close(fd);  
    }  
    execvp(argv[0], argv);  
}  
wait(); // parent waits for child  
} // end while(1)
```

