מערכות הפעלה- 02340123

תרגיל בית יבש 1

מגישים:

דריה בבין	TODO:	TODO:
אילון הלוי	328137831	eilon.halevy@campus.technion.ac.il

שאלה 1- Process management

חלק א- שאלות על קטע הקוד הבא:

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t pid1, pid2;

    printf("Parent Process PID: %d\n", getpid());

    pid1 = fork();
    if (pid1 == 0) {
        printf("First child (PID: %d, Parent PID: %d)\n", getpid(), getppid());

        pid2 = fork();
        if (pid2 == 0) {
            printf("Second child (PID: %d, Parent PID: %d)\n", getpid(), getppid());
        } else if (pid2 > 0) {
            wait(NULL);
            printf("First child (PID: %d) done waiting for second child\n", getpid());
        }
    } else if (pid1 > 0) {
        wait(NULL);
        printf("Parent (PID: %d) done waiting for first child\n", getpid());
    }

    printf("Process PID: %d exiting\n", getpid());
    return 256;
}
```

ההנחות על קטע הקוד הבא (לכל שאלה 1):

- קריאות המערכת fork(), wait(), getppid(), getpid() אינן נכשלות.
- כל שורה הנכתבת לפלט אינה נקטעת ע"י שורה אחרת.
- לא נוצרים תהליכים נוספים במערכת, פרט לאלו שנוצרים ע"י התוכנית. (כלומר ניתן להניח שמספרי PID מוקצים לתהליכים בצורה סדרתית עולה)

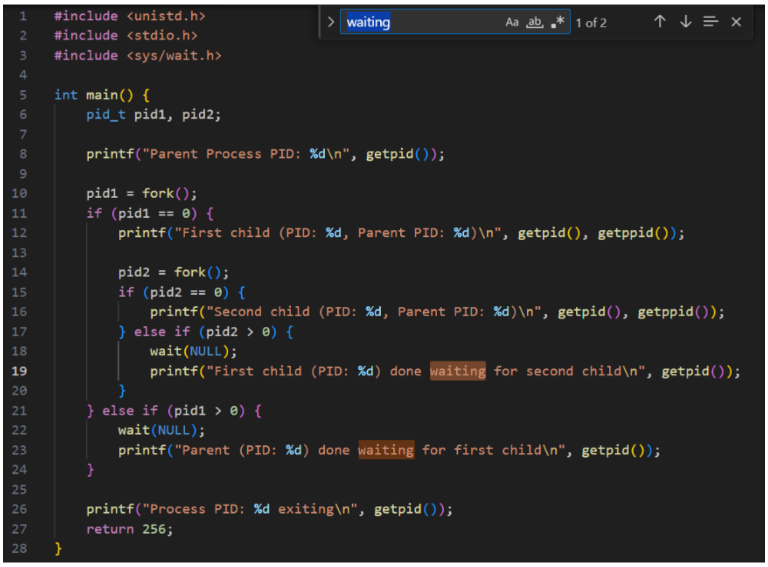
שאלה 1.א.1

בחרו באפשרות הנכונה ביותר בנוגע לריצת הקוד:

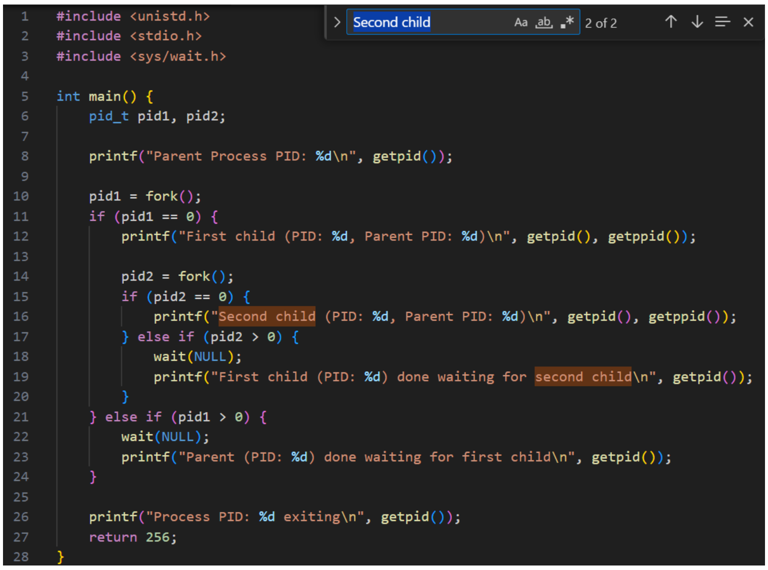
a. בפלט יופיע הביטוי “waiting” פעמיים בדיוק.
b. בפלט התכנית הביטוי “Second child” עשוי להופיע לפני או אחרי הביטוי "Parent ... done waiting".
c. כל התהליכים מסתיימים בסדר הפוך לסדר שבו נוצרו.
d. רק התהליך של הילד הראשון ממתין לתהליך אחר שיסתיים.
e. בפלט התוכנית מספר הודעות ה-"exiting" שווה למספר הקריאות ל-fork.
f. יש יותר מהיגד אחד נכון.

הסבר לפתרון (התשובה היא f, כיוון שהיגדים a ו- c נכונים)

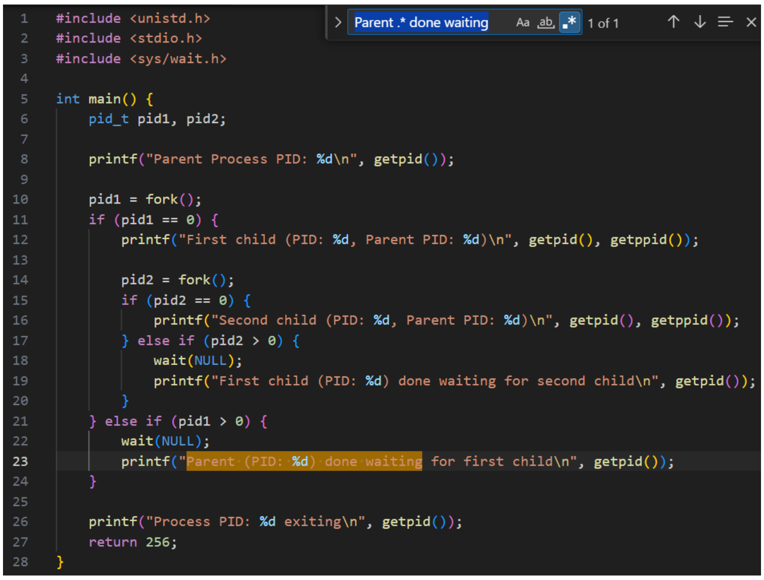
ראשית נמפה את השורות בהן מודפסות הודעות כמתואר:



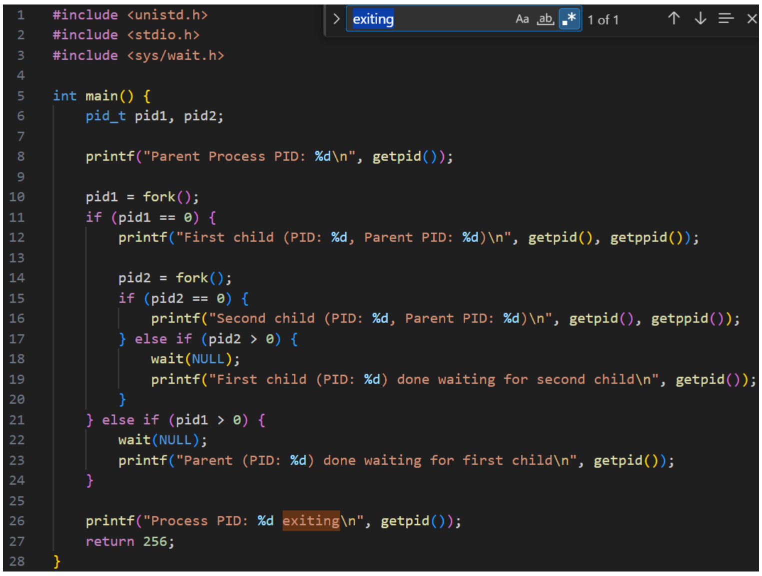
```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t pid1, pid2;

    printf("Parent Process PID: %d\n", getpid());

    pid1 = fork();
    if (pid1 == 0) {
        printf("First child (PID: %d, Parent PID: %d)\n", getpid(), getppid());

        pid2 = fork();
        if (pid2 == 0) {
            printf("Second child (PID: %d, Parent PID: %d)\n", getpid(), getppid());
        } else if (pid2 > 0) {
            wait(NULL);
            printf("First child (PID: %d) done waiting for second child\n", getpid());
        }
    } else if (pid1 > 0) {
        wait(NULL);
        printf("Parent (PID: %d) done waiting for first child\n", getpid());
    }

    printf("Process PID: %d exiting\n", getpid());
    return 256;
}
```



```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t pid1, pid2;

    printf("Parent Process PID: %d\n", getpid());

    pid1 = fork();
    if (pid1 == 0) {
        printf("First child (PID: %d, Parent PID: %d)\n", getpid(), getppid());

        pid2 = fork();
        if (pid2 == 0) {
            printf("Second child (PID: %d, Parent PID: %d)\n", getpid(), getppid());
        } else if (pid2 > 0) {
            wait(NULL);
            printf("First child (PID: %d) done waiting for second child\n", getpid());
        }
    } else if (pid1 > 0) {
        wait(NULL);
        printf("Parent (PID: %d) done waiting for first child\n", getpid());
    }

    printf("Process PID: %d exiting\n", getpid());
    return 256;
}
```



```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t pid1, pid2;

    printf("Parent Process PID: %d\n", getpid());

    pid1 = fork();
    if (pid1 == 0) {
        printf("First child (PID: %d, Parent PID: %d)\n", getpid(), getppid());

        pid2 = fork();
        if (pid2 == 0) {
            printf("Second child (PID: %d, Parent PID: %d)\n", getpid(), getppid());
        } else if (pid2 > 0) {
            wait(NULL);
            printf("First child (PID: %d) done waiting for second child\n", getpid());
        }
    } else if (pid1 > 0) {
        wait(NULL);
        printf("Parent (PID: %d) done waiting for first child\n", getpid());
    }

    printf("Process PID: %d exiting\n", getpid());
    return 256;
}
```



```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t pid1, pid2;

    printf("Parent Process PID: %d\n", getpid());

    pid1 = fork();
    if (pid1 == 0) {
        printf("First child (PID: %d, Parent PID: %d)\n", getpid(), getppid());

        pid2 = fork();
        if (pid2 == 0) {
            printf("Second child (PID: %d, Parent PID: %d)\n", getpid(), getppid());
        } else if (pid2 > 0) {
            wait(NULL);
            printf("First child (PID: %d) done waiting for second child\n", getpid());
        }
    } else if (pid1 > 0) {
        wait(NULL);
        printf("Parent (PID: %d) done waiting for first child\n", getpid());
    }

    printf("Process PID: %d exiting\n", getpid());
    return 256;
}
```

תוצאת המיפוי:

- הביטוי "waiting" יודפס כפלט לתוכנית בשורות 19,23.
- הביטוי "Second child" יודפס כפלט לתוכנית בשורות 16,19.
- הביטוי "Parent ... done waiting" יודפס כפלט לתוכנית בשורה 23.
- הודעות ה-"exiting" תודפס כפלט לתוכנית בשורה 26.
- קריאות ה-fork מתבצעות בשורות 10, 14.

תיאור שלבי ריצת התוכנית:

1. בשורה 8, תהליך האב מדפיס את מזהה התהליך שלו (PID)

2. בשורה 10, תהליך האב ייצור תהליך בן באמצעות קריאת fork.
נקרא לתהליך שנוצר הבן הראשון.

3. כעת תהליך האב ותהליך הבן הראשון מתקדמים "במקביל" על ההרצה של אותו קוד החל משורה 10 (לא כולל)
   a. עבור תהליך האב מתקיים pid1 > 0 (מההנחה שקריאת ה-fork צלחה), ולכן האב יחכה לסיום תהליך הבן הראשון, בשורה 22.
   b. עבור תהליך הבן הראשון מתקיים pid1 == 0, ולכן תהליך הבן הראשון ימשיך את ריצת הקוד החל משורה 12 (כולל).

4. בשורה 12, תהליך הבן הראשון ידפיס First child
(תהליך האב עוד מחכה שתהליך הבן הראשון יסתיים)

5. בשורה 14, תהליך הבן הראשון ייצור תהליך בן באמצעות קריאת fork
נקרא לתהליך שנוצר הבן השני.

6. כעת תהליך הבן השני ותהליך הבן הראשון מתקדמים "במקביל" על ההרצה של אותו קוד החל משורה 14 (לא כולל)
   a. עבור תהליך הבן הראשון מתקיים pid2 > 0 (מההנחה שקריאת ה-fork צלחה), ולכן הבן הראשון יחכה לסיום תהליך הבן השני, בשורה 18.
   b. עבור תהליך הבן השני מתקיים pid2 == 0, ולכן תהליך הבן השני ימשיך את ריצת הקוד החל משורה 16 (כולל).
   c. תהליך האב עדין מחכה לסיום תהליך הבן הראשון.

7. תהליך הבן השני ידפיס Second child בשורה 16 (תהליך הבן הראשון מחכה לסיום תהליך הבן השני, ותהליך האב מחכה לסיום תהליך הבן הראשון)

8. תהליך הבן השני ידפיס exiting בשורה 26 (תהליך הבן הראשון מחכה לסיום תהליך הבן השני, ותהליך האב מחכה לסיום תהליך הבן הראשון)

9. תהליך הבן השני יסיים את ריצתו. (יבוצע exit בשורה 27)

10. תהליך הבן הראשון סיים לחכות (בשורה 18), תהליך האב עדין מחכה לסיום תהליך הבן הראשון.

11. תהליך הבן הראשון מדפיס First child (בשורה 19), תהליך האב עדין מחכה לסיום תהליך הבן הראשון.

12. תהליך הבן הראשון ידפיס exiting בשורה 26 (תהליך האב מחכה לסיום תהליך הבן הראשון)

13. תהליך הבן הראשון יסיים את ריצתו. (יבוצע exit בשורה 27).

14. תהליך האב סיים לחכות (בשורה 22)

15. תהליך האב ידפיס בשורה 23, Parent

16. תהליך האב ידפיס exiting בשורה 26

17. תהליך האב יסיים את ריצתו

<u>מסקנות:</u>

- הצבע הירוק (waiting) מופיע פעמיים בדיוק בתיאור ריצת הקוד, ולכן **טענה a נכונה**.
- הצבע הסגול (Second child) מופיע לפני הצבע האדום (Parent ... done waiting) בהכרח, ולא ייתכן שיופיע אחריו. לכן **טענה b אינה נכונה**.
- תהליך האב מסתיים אחרון, תהליך הבן הראשון מסתיים לפניו, ותהליך הבן השני מסתיים ראשון. תהליך הבן השני נוצר אחרון, ותהליך הבן הראשון נוצר לפניו, וכן תהליך האב נוצר ראשון. לכן התהליכים מסתיימים בסדר הפוך לסדר בו הם נוצרו. כלומר **טענה c נכונה**.
- במהלך תיאור ריצת התוכנית, גם תהליך האב וגם תהליך הבן הראשון מחכים לתהליך שיסתיים. לכן **טענה d אינה נכונה**.
- בפלט התוכנית, ישנן 3 קריאות תכלת (exiting), ושני מופעים של צבע צהוב (fork), ולכן **טענה e אינה נכונה**.
- סה"כ, טענות a, c נכונות ולכן **טענה f נכונה**.

<u>שאלה 2.א.1</u>

<u>נניח כי תהליך האב הוא בעל ה-PID 7000.מה ה-PID של התהליך האחרון שמדפיס exiting?</u>

<u>תשובה-</u> 7000

הסבר: בסעיף הקודם, ראינו כי תהליך האב מסתיים אחרון, ולכן מהנתון ה-PID שלו הוא 7000.

שאלה 3.א.1

נחליף את השורה wait(NULL); ב**תהליך האב** ל-wait(pid1, NULL, 0);, כלומר שורה 22.

<u>האם שינוי זה ישפיע על התוצאה או על אופן פעולת התוכנית?</u>

<u>תשובה-</u> לא

הסבר: לתהליך האב נוצר רק בן אחד במהלך התוכנית ולכן אנחנו באופן דיפולטיבי נחכה לסיום תהליך pid1 (שהוא תהליך הבן הראשון). כיוון שגם לפני החלפת השורה ביצענו את אותה פעולה של לחכות לסיום תהליך הבן הראשון, בעצם השינוי לא השפיע על אופן פעולת התוכנית, ובפרט על התוצאה שלה.

שאלה 4.א.1

משנים את את תנאי ה-else השני בקוד (אליו תהליך האב נכנס) באופן הבא:

```
21      } else if (pid1 > 0) {
22          int status;
23          wait(&status);
24          printf("status: %d\n", status);
25          printf("Parent (PID: %d) done waiting for first child\n", getpid());
26      }
```

מהו הערך שיודפס בסיום התוכנית למשתנה status?

<u>תשובה-</u> 0

הסבר:

- אופן ריצת התוכנית לא השתנה, כי תהליך האב עדין מחכה לסיום התהליך של הבן הראשון.
- מההנחה קריאת wait מתבצעת בהצלחה (של שורה 23), ולכן שומרת לערך של status את התוצאה שחזרה מה-exit (סיום התהליך) של הבן הראשון, שהיא 256 (את 8 הביטים הנמוכים שלה).
- מתקיים $256 = 2^8$, כלומר 8 הביטים הראשונים (LSB) במספר 256 הם 0, לכן הערך שיישמר ב-status הוא 0.

לכן הערך status שיודפס בסיום (התהליך היחיד שמבקר בשורות 22-26 הוא תהליך האב) הוא 0.

חלק ב

שאלה 1.ב.1

תארו שני מצבים בהם קריאת המערכת fork עלולה להיכשל.

תשובה-

- חוסר מקום בזיכרון להקצאת תהליך חדש (שמירת הנתונים בשבילו).
- הגענו למגבלת כמות התהליכים שיכולים להיות קיימים במערכת באותו הזמן.
  (קובעים מגבלה כזו כדי להגן על המחשב ממתקפת dos)

שאלה 1.ב.2

תארו בקצרה את ההבדל בין return ל-()exit.

תשובה-

השורה return נקראת בסיום פונקציה (מכריחה את סיומה) וממשיכה את התהליך החל מהשורה בו נקראה אותה פונקציה (לא כולל), וייתכן שמחזירה ערך.

במקרים מסויימים return יכולה להיכשל, לדוגמה, אם חזרנו כבר מפונקציה שקראה לה, בחזרה מ-main, התהליך קורא בקריאת המערכת exit.

בקריאת המערכת exit, התוכנית עוצרת, מחזירה ערך מתאים למערכת ההפעלה ומסיימת את התהליך במקום שבו היינו (לא בהכרח מ-main). לפי man, הפעולה אינה יכולה להיכשל.

נוסף על כך, כאשר נשלח את exit מ-main, לא נמחק את האובייקטים הלוקליים, התהליך הופך לזומבי ורק הפעולה wait תמחק את האובייקטים האלו מהמחסנית בעוד ש-return מוחקת מהמחסנית את האובייקטים הלוקאלים של הפונקציה.

שאלה 2- Signals

חלק א- שאלות על קטע הקוד הבא:

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

void seg_fault_catcher(int signum)
{
    printf("%d\n", signum);
    exit(0);
}

int main()
{
    signal(SIGSEGV, seg_fault_catcher);
    int* x = malloc(4 * sizeof(int));
    for (int i = 0; i < 4; i++) {
        x[i] = i;
    }

    int j = 4;
    while (1) {
        j--;
        printf("%d,", x[j]);
    }

    printf("Hi\n");
    return 0;
}
```

שאלה 1.א.2

בחרו באפשרות הנכונה ביותר בנוגע לריצת הקוד:

a. יודפס קודם "3,2,1,0" ואז "Hi".
b. ההדפסה האחרונה תהיה "Hi"
c. יודפס רק "3,2,1,0".
d. ההדפסה הנכונה תהיה "11".
e. לא ניתן לדעת מה תהיה ההדפסה האחרונה.

הסבר לפתרון: (התשובה היא d)

בשורה 13 של הקוד (תחילת התהליך), מקשרים את הטיפול בסיגנל SIGSEGV, כלומר טיפול במצב בו ניגשים לכתובת זיכרון "לא חוקית", לפונקציה seg_fault_catcher, ולא להריגה של התהליך (הטיפול הדיפולטי של מערכת ההפעלה).

לכן, בשורה 22, לאחר 4 ריצות לולאה, הערך j=-1 הוא כתובת לא חוקית במערך x (שהוקצו לו הכתובות 3,2,1,0). הגישה ל-$x[j]$ תפעיל את הסיגנל SIGSEGV.

מערכת ההפעלה, כתוצאה מהפעלת הסיגנל, תבצע interrupt לתהליך, ותטפל בסיגנל.

מאחר והטיפול בסיגנל SIGSEGV קושר לפונקציה seg_fault_catcher, זו הפונקציה שתקרא בעת הגישה הלא חוקית לזיכרון.

בפונקציה seg_fault_catcher, מודפס ערך הסיגנל שמערכת ההפעלה שלחה את הפונקציה לטפל בו. ספיציפית, ערך הסיגנל SIGSEGV הוא 11.

הפונקציה seg_fault_catcher לכן תדפיס "11", ולאחר מכן תבצע exit(0), כלומר תסיים את התהליך.

לכן ההדפסה האחרונה תהיה "11", כלומר הטענה d נכונה. (ושאר הטענות אינן נכונות)

שאלה 2.א.2

כעת נשנה את שגרת הטיפול בסיגנל כך:

```
void seg_fault_catcher(int signum)
{
    printf("%d\n", signum);
    signal(SIGFPE, seg_fault_catcher);
    exit(3/(11-signum));
}
```

<u>עבור כל אחת מהטענות הבאות, הכריעו האם היא נכונה או לא, ונמקו בקצרה:</u>

- הפונקציה seg_fault_catcher תקרא פעם אחת בלבד.
- התוכנית תסתיים בצורה תקינה (ע"י קריאה מוצלחת ל-exit)

<u>תשובה:</u> הפונקציה seg_fault_catcher תקרא <u>פעמיים</u>, ובפעם השנייה, תקרא ל-exit באופן מוצלח, ולכן <u>התוכנית תסתיים בצורה תקינה</u>.

<u>הסבר:</u>

הניתוח עד (לא כולל) הקריאה exit(0) נותר זהה.

נמשיך את ניתוח הריצה מאותה נקודת זמן (ההדפסה "11" של הפונקציה seg_fault_catcher).

לאחר ההדפסה "11", מקשרים את טיפול המערכת בסיגנל SIGFPE (חלוקה ב-0) לפונקציה seg_fault_catcher, ומבצעים את הפעולה האריתמטית 3/(11-signum).

הפעולה האריתמטית נכשלת (כיוון ש-signum הוא 11, ולכן חלוקה ב-0), ומפעילה את הסיגנל SIGFPE של מערכת ההפעלה. מערכת ההפעלה עושה interrupt לתהליך, ושולחת לטיפול בסיגנל.

כיוון שהגדרנו את הטיפול בסיגנל להיות seg_fault_catcher, מערכת ההפעלה קוראת לפונקציה עם הארגומנט SIGFPE (מספר שאינו 11, הרי לכל סיגנל יש ערך ייחודי).

הפונקציה seg_fault_catcher תדפיס את הערך SIGFPE, ותבצע exit(3/(11-signum)), כאשר הפעם החישוב האריתמטי לא ייכשל כיוון שההפרש בין 11 ל-signum לא יהיה 0.

לכן ייקרא exit בהצלחה, והתהליך ייסתיים באופן תקין.

שאלה 2- Signals

חלק ב

שאלה 2.ב.1

סמנו האם ההיגדים הבאים נכונים / לא נכונים:

a. סיגנלים משמשים לתקשורת בין תהליכים/חוטים בלבד (אין להם שימוש אחר). **לא נכון**

b. שגרת הטיפול של סיגנל מתבצעת במצב משתמש. **נכון**

c. תהליך יכול להוסיף סיגנל חדש. **לא נכון**

d. תהליך יכול לשנות כל שגרת טיפול (handler) בטבלת הסיגנלים. **לא נכון**

e. אם תהליך איננו מתעלם מסיגנל (SIG_IGN), מובטח שהוא יטפל בו. **לא נכון**

נגדיר את נקודת הזמן שבה תהליך "חטף" סיגנל להיות הרגע שבו התהליך התחיל לטפל בסיגנל.

f. תהליך יכול לחטוף סיגנל בכל נקודת זמן. **לא נכון**

g. תהליך יכול לחטוף סיגנל רק בעת מעבר מגרעין למשתמש. **נכון**

h. מנקודת המבט של תהליך, לחטוף סיגנל תמיד מהווה אירוע סינכרוני. **לא נכון**

שאלה 2.ב.2

איזה מבין ההיגדים הבאים נכון?

a. כשמגדירים signal handler בעבור הסיגנל SIGCONT, ה-handler נקרא מייד אחרי שממשיכים את התהליך.

b. כשמגדירים signal handler בעבור הסיגנל SIGKILL, ה-handler נקרא מייד אחרי שהורגים את התהליך.

c. כשמגדירים signal handler בעבור הסיגנל SIGSTOP, ה-handler נקרא מייד אחרי שעוצרים את התהליך.

d. יש יותר מהיגד אחד נכון.

e. כל ההיגדים שגויים.

תשובה- ההיגד הנכון היחיד הוא a.

הסבר:

כל תהליך לא יכול להתעלם מאף אחד מהסיגנלים SIGCONT, SIGKILL, SIGSTOP.

המשמעות היא שהתהליך:

- יהרג כאשר מבוצע SIGKILL
- יעצור כאשר מבוצע SIGSTOP
- ימשיך כאשר מבוצע SIGCONT

בשני הסיגנלים הראשונים, מערכת ההפעלה כופה על התהליך להיהרג/להסתיים ואין השפעה למימוש לקביעת שגרת טיפול בסיגנלים SIGKILL, SIGSTOP.

עבור הסיגנל SIGCONT, התהליך יפעיל את הטיפול שהוגדר ב-handler, ובכל זאת ימשיך את התוכנית לאחר סיום הטיפול. בפרט handler ייקרא מייד אחרי שממשיכים את התהליך.

לכן היגד a נכון, והיגדים b,c אינם נכונים.

מכך שהיגד a נכון, לא כל ההיגדים שגויים, ולכן היגד e שגוי.

בנוסף היגד d נכון אם"ם יש יותר מהיגד אחד נכון.

לכן היגד d יכול להיות הן נכון והן לא נכון 😊 (אם הוא נכון יש 2 היגדים נכונים ולא נקבל סתירה).

בכל אופן (ברור שהכוונה היא שרק היגד a נכון), היגד a הוא היחיד שנכון באופן חד-משמעי.

שאלה 3.ב.2

איזה מבין ההיגדים הבאים נכון?

a. תהליך עלול לקבל את הסיגנל SIGXCPU רק אם השתמשנו קודם לכן בקריאת המערכת setrlimit בכדי להגביל את צריכת המעבד של תהליך זה.

b. אפשר לשלוח את הסיגנל SIGCONT לתהליך רק אם קודם לכן שלחנו אליו את הסיגנל SIGSTOP.

c. תהליך עלול לקבל את הסיגנל SIGPIPE רק אם קודם לכן הוא השתמש בקריאת המערכת pipe אבל אז השתמש ב-pipe שנוצר לא נכון (כתב ל-pipe שנסגר לקריאה).

d. תהליך עשוי לקבל את הסיגנל SIGTRAP רק אם הוא רץ תחת דיבאגר.

e. כל ההיגדים שגויים.

f. כל ההיגדים נכונים.

תשובה: כל ההיגדים שגויים (מלבד e), ולכן ההיגד היחיד שנכון הוא e.

(אם ההיגד e אינו שגוי אז לא כל ההיגדים שגויים ולכן הוא כן שגוי 😊)

הסבר:

בעזרת קריאת המערכת kill, נוכל לשלוח לכל תהליך איזה סיגנל שנרצה, ובפרט את הסיגנלים:

- SIGXCPU
- SIGCONT
- SIGPIPE
- SIGTRAP

ולכן (בנוסף לדרכים שצויינו בשאלה לקבל כל סיגנל), לכל אחד מהסיגנלים הנ"ל יש יותר מדרך אחת לקבלו. לכן היגדים a,b,c,d שגויים. ובפרט לא כל ההיגדים נכונים. לכן היגד f שגוי גם הוא.

לכן כל ההיגדים שגויים מלבד e. (היגד e שגוי אם"ם הוא נכון ולכן נקבל מעין סתירה)

שאלה 3- Inter-Process Communication

נתון תהליך רץ בשם A.

סעיף א: האם ייתכן שהאירועים הבאים יגרמו לסיום הריצה המיידי של תהליך A? אם כן, תנו דוגמא. אחרת, נמקו.

אירוע	כן/לא	דוגמא/נימוק (בהתאם)
תהליך A כותב ל-pipe	כן	ייתכן ואין קורא ל-pipe אליו ניסינו לכתוב (מתוך A). לכן אנחנו נקבל SIGPIPE שבאופן דיפולטי מסיים את התהליך.
תהליך A קורא ל-kill()	כן	אם נשתמש בקריאת המערכת (מתוך A) kill(getpid(),SIGKILL), נהרוג את התהליך A.

סעיף ב: האם האירועים הבאים יגרמו בהכרח לסיום הריצה המיידי של תהליך A? נמקו.

אירוע	כן/לא	נימוק
תהליך A כותב ל-pipe	לא	ייתכן שהכתיבה ל-pipe מצליחה, ולא נקבל אף סיגנל.
תהליך A קורא ל-kill()	לא	התהליך A יכול לקרוא ל-kill(getpid(),SIGCONT) מתוך A, ושגרת הטיפול הדיפולטית SIGCONT על תהליך שרץ היא לא לעשות כלום (בפרט תהליך A לא יסתיים במיידי מהקריאה).

סעיף ג: האם האירועים הבאים יגרמו בהכרח למעבר מיידי ממצב משתמש למצב גרעין? נמקו.

(בתשובתכם התעלמו מהחלפות הקשר בין התהליכים / פסיקות)

אירוע	כן/לא	נימוק
תהליך A כותב ל-pipe (מחובר לתהליך B)	כן	כשתהליך כותב ל-pipe אנחנו משתמשים בפעולה write שמעבירה את הטיפול למערכת ההפעלה.
תהליך A קורא ל-dup()	כן	dup() הוא קריאת מערכת שמעתיקה את ה-FDT ולכן כמו כל קריאת מערכת היא מעבירה אותנו ל-kernel mode
תהליך B קורא מה-pipe (הוא רץ על ליבה שונה משל A)	לא	אם התהליכים של B,A זרים, כאשר B עובר ממצב משתמש למצב גרעין לא משפיע על שינוי ב-A ממצב משתמש למצב גרעין. בפרט עבור קריאה מ-pipe. (ייתכן למשל כי A הוא תהליך של פעולות אריתמטיות)

סעיף ד: האם execv() יוצרת מופע חדש בעבור האובייקטים הבאים?
הערה: הפעולה execv נוטשת את התהליך הקיים והופכת אותו לתהליך אחר.

	Stack	FDT	Heap	PCB
כן/לא	כן	לא	כן	לא

סעיף ה: האם fork() יוצרת מופע חדש בעבור האובייקטים הבאים?
הערה: הפעולה fork יוצרת עותק של כל תוכן התהליך.

	Stack	FDT	File Objects	PCB
כן/לא	כן	כן	לא	כן