234123 – מערכות הפעלה

תרגיל בית 1 - יבש

אביב 2023

מגישים:

נועה פריאנטה 206200305

pariente@campus.technion.ac.il

ששון שמואל למעי 325172351

sason-shmuel@campus.technion.ac.il

Question 1 – Abstraction & OSs (15 points)

1. הסבירו: מהי אבסטרקציה במערכות מחשבים?

אבסטרקציה במערכות מחשבים זו הפעולה של הסתרת פרטי מימוש של פונקציות או מודלים מסוימים, כך שבסופו של דבר יש לנו רק את המידע על מה הפונקציה אמורה לעשות אך לא איך היא עושה זאת.

2. מדוע אנו משתמשים באבסטרקציות במערכות הפעלה?

אנו משתמשים באבסטרקציות במערכות הפעלה ממספר סיבות, ראשית, בכך אנו יוצרים שכבת הגנה על הפונקציה, כי לא יודעים את המימוש שלה ובפרט לא את המשתנים, משאבים וכו' שהיא משתמשת בהם. שנית, זה מקל על פיתוח של אפליקציות בכך שהמשתמש לא צריך להיות מוסח ממימוש שלא אמור להשפיע עליו.

3. תנו דוגמה לאבסטרקציה מרכזית שמשתמשים בה במערכות הפעלה והסבירו מדוע משתמשים בה.

דוגמה אחת מרכזית לשימוש באבסטרקציה: וירטואליזציה של משאבים פיזיים (מעבד וזיכרון) עבור המשתמש. מערכת ההפעלה מציגה למשתמש גרסאות וירטואליות של המעבד והזיכרון שמספקים אבסטרקציה לחומרה הפיזית ובכך הופכים את השימוש לפשוט. מלבד זאת ככה אנו מגנים על המשאב הפיזי כי אף משתמש לא יידע היכן המידע באמת קיים.

Question 2 – Inter-Process Communication (45 points)

השלימו את קוד ה-C הנתון הממשמ shell כך שיבצע את פקודת ה-bash:

/bin/prog.out < in.txt 2> err.txt > out.txt

עליכם להשתמש אך ורק בקריאות המערכת הבאות:

int **open**(const char *path, …);

int **execv**(const char *filename,

char *const argv[]);

int **close**(int fd);

- אין עליכם חובה לבדוק שקריאות המערכת צלחו.
- פתחו קבצים באמצעות הקריאה open(path,…), כלומר בפרמטר השני יש לרשום "..." (גם לקבצים קיימים וגם לחדשים).
- אין לבצע קריאות מערכת מיותרות.
- תזכורת:

"err.txt <2" מציין redirection של 2=fd (STDERR) של התהליך לכתיבה לקובץ err.txt.

```
pid_t pid = fork();
if (pid == 0) {
    close (0);
    close (1);
    close (2);
    open("in.txt", "...") ;
    open("out.txt", "...") ;
    open("err.txt", "...");
    chr* args[] = {"/bin/prog.out", NULL};
    execv(args[0], args);
} else {
    wait(NULL);
}
```

2. (7 נקודות)

לפניכם קטע קוד המשתמש ב-pipes.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main() {
  int fd[2];
  int count;
  int id;
  pipe(fd);
  pid_t p = fork();
  count = get_random_between(1, 1000000);//gets random integer in [1,1000000]
  if (p == 0) {
    id = 234123;
    for(int i = 0; i < count; i++)
    {
   write(fd[1], (void*)(&id), sizeof(int));
    }
  } else {
    close(fd[1]);
    for(int i = 0; i < count; i++)
   {
   if(read(fd[0], (void*)(&id), sizeof(int)) > 0)
      {
      printf("My favorite course, %d\n", id);
      }
      else
      {
            break;
      }
   }
 }
  return 0;
}
```

בשני הסעיפים הבאים, הקף את התשובה הנכונה: (x21 = 2 נקודות)

אם היינו מחליפים את שורות 10 ו-11, ערכי המשתנה count בשני התהליכים -
הקף: בהכרח זהים \ בהכרח שונים \ ייתכן ששונים וייתכן שזהים

אם **לא** נחליף את שורות 10 ו-11, ערכי המשתנה count בשני התהליכים –
הקף: בהכרח זהים \ בהכרח שונים \ ייתכן ששונים וייתכן שזהים

האם הקוד תקין? (5 נקודות)
הקיפו: כן \ לא

אם בחרתם לא, תנו דוגמא לתרחיש הבעייתי ביותר האפשרי:
תהליך הבן לא סגר את FD של הקריאה בpipe לכן, גם אם האבא שלו יסיים לקרוא הבן ימשיך לכתוב עד מגבלת count שהוא הגריל. מקרה הכי גרוע, נגריל את המספר מליון עבור הבן והמספר 1 עבור האב, הבן יכתוב ברצף מליון int'ים (או כל מספר שיגרום לכתיב של יותר מ-64kb כיוון שגודל של int זה 4 בתים נקבל 160,000) , דבר שגם משפיע על הביצועים שמרחשים לחינם וגם עלול לגרום לשגיאות אם אי אפשר כבר לכתוב כי אין מקום (במימוש של הpipe הוא נחסם)

<u>חלק 2:</u> (10 נקודות)

לפניכם קטע קוד:

```
#include <stdio.h>

#include <signal.h>

#include <stdlib.h>


void fpe_catcher (int signum) {

  printf("Hello\n");

  exit(0);

}

int main() {

  signal(SIGFPE, fpe_catcher);

  int x = 234123 / (0);

  printf("Hi\n");

  while(1);

  return 0;

}
```

תזכורת: SIGFPE הוא הסיגנל המתאים לשגיאות אריתמטיות, כמו חלוקה ב-0 (גם במקרה של מספרים שלמים).

1. בחרו באפשרות הנכונה בנוגע לריצת הקוד: (4 נקודות)

   1. יודפס קודם "Hi" ואז "Hello".
   2. יודפס רק "Hello".
   3. יודפס רק "Hi".
   4. לא יודפס כלום.
   5. תשובות i,ii אפשריות.

נמקו:

תיווצר החריגה בגלל החלוקה ב0, שגרת הטיפול תיקרא, היא תדפיס Hello ואז מבצעת קריאת המערכת exit() שעל פי הגדרתה מסיימת את ביצוע התהליך הקורא ומשחררת את כל המשאבים שברשותו.

2. כעת נניח שבזמן כלשהו של ריצת הקוד הנ"ל, תהליך אחר שולח את הסיגנל SIGFPE ל תהליך המריץ את הקוד.

עבור כל אחד מהתרחישים הבאים, הכריעו האם הוא **אפשרי** או לא, ונמקו בקצרה: (4=x22 נקודות)

- יודפס "Hello" פעמיים.

הקיפו: כן \ לא

נימוק:

כפי שלמדנו שגרת הטיפול מתבצעת במלואה, גם אם בזמן השגרה מתקבל signal נוסף מאותו סוג. ה-hello מודפס כחלק משגרת הטיפול, ומכיוון שהשגרה מסתיים ב-exit() לא יתקיים מצב שיודפס פעמיים hello.

לא יודפס "Hello" בכלל.

הקיפו: כן \ לא

נימוק:

אם ישלח SIGFPE בתחילת ריצת התוכנית (בפקט לפני שליחת הסיגנל) ההנדלר לשגרת הטיפול יהפוך להיות ההנדלר הדיפולטי בחילוק ב-0 אשר איננה כוללת הדפסת hello אלה רק מסיימת את התוכנית.

3. תארו את ריצת התכנית במידה והיינו מסירים את פקודת ה-exit(0): (2 נקודות)

כפי שלמדנו בהרצאה נקודה החזרה של ההנדלר היא לאותה פקודה שהורצה ברגע תחילת הטיפול בסיגנל. לכן אם נסיר את exit() בסוף ההנדלר, התהליך יבצע את החלוקה באפס והסיגנל יקרא שוב ושוב ולכן יודפס אינסוף פעמים (על שהתהליך מת) "hello\n" כלומר hello בכל שורה.

Question 3 – Process management (40 points)

```
int X = 1, p1 = 0, p2 = 0;
     int ProcessA() {
          printf("process A\n");
          while(X);
          printf("process A finished\n");
          exit (1);
     }
     void killAll(){
          if(p2) kill(p2, 15);
          if(p1) kill(p1, 9);
     }
     int ProcessB() {
          X = 0;
          printf("process B\n");
                 killAll();
          printf("process B finished\n");
          return 1;
     }
     int main(){
          int status;
          if((p1 = fork()) != 0)
               if((p2 = fork()) != 0){
                    wait(&status);
                    printf("status: %d\n", status);
                    wait(&status);
                    printf("status: %d\n", status);
               } else {
                    ProcessB();
          } else {
               ProcessA();
          }
          printf("The end\n");
          return 3;
     }
```

בשאלה זו עליכן להניח כי:

1. קריאות המערכת fork() וkill() אינן נכשלות.
2. כל שורה הנכתבת לפלט אינה נקטעת ע"י שורה אחרת.
3. כאשר תהליך מקבל סיגנל x הוא מסתיים וערך היציאה שלו הוא x + 128.

עבור כל אחת משורות הפלט הבאות, סמנו כמה פעמים הן מופיעות בפלט כלשהו, נמקו את תשובתכן.

1. process A
   a. 0
   b. 0 or 1
   c. 1
   d. 1 or 2
   e. 2

נימוק: יתכנו שני מקרים: פונקציה ProcessA() תקרא לפני לפני שבפונקציה ProcessB() קורא ל-killAll() ובכך יודפס על המסך "process A" או שיבוצע killAll() לפני שהקריאה לפונקציה ProcessA() מתרחשת ובקרה זה לא יודפס.

2. status: 1
   a. 0
   b. 0 or 1
   c. 1
   d. 1 or 2
   e. 2

נימוק: מכיוון ש-x=1 הבן הראשון מריץ לולאה אינסופית ב-while ולכן exit ב-processA לעולם לא יקרא. לכן הבן הראשון יקבל פקודת kill מהבן השני תמיד ובכך יסיים את ריצתו ויודפס status: 137 (סיגנל kill הינו בעל מספר 9 ו הרי 9+128=137) והבן השני יסתיים בסיום ריצת התוכנית (סיום main) ויודפס status:3

3. status: 137
   a. 0
   b. 0 or 1
   c. 1
   d. 1 or 2
   e. 2

נימוק: כמו שצויין סעיף קודם, הבן הראשון יקבל פקודת kill מהבן השני תמיד ובכך יסיים את ריצתו ויודפס status: 137.

4. status: 143

   a. 0
   b. 0 or 1
   c. 1
   d. 1 or 2
   e. 2

נימוק: killAll אינה שולחת סינגל לתהליך B מכיוון שהוא זה שקורא לפונקציה ולכן p2=0 ו-p1 שווה לpid של תהליך A ולכן לא יתכן שיודפס status: 143

5. The end

   a. 0
   b. 0 or 1
   c. 1
   d. 1 or 2
   e. 2

6. נימוק: גם האבא הראשי וגם הבן של תהליך B יגיעו לשורת קוד המדפיסה The end ולכן יודפס פעמיים