TECHNION
Israel Institute
of Technology

Computer
Science
Department

Operating Systems – 234123

Homework Exercise 1 – Dry

Spring 2023

{ Name: Roni Roitbord , ID: 313575599, email: roniro@campus.technion.ac.il

Name: Amit Gabay , ID: 206040768, email: amitg@campus.technion.ac.il }

Question 1 – Abstraction & OSs (15 points)

1. הסבירו: מהי אבסטרקציה במערכות מחשבים?
אבסטרקציה במערכות מחשבים היא פישוט של מערכות מורכבות על ידי הסתרת פרטים שאינם הכרחיים תוך הדגשת תכונות חיוניות. אבסטרקציה מהווה שכבת מפרידה בין חומרה לתוכנה.

2. מדוע אנו משתמשים באבסטרקציות במערכות הפעלה?
אנו משתמשים באבסטרקציות במערכות הפעלה מפני שתפקידיה כוללים חלוקה של משאבי החומרה בצורה יעילה והוגנת בין המשתמשים, הקלה על הפיתוח של אפליקציות וכן הגנה על המידע של המשתמשים מפני תוכנות זדוניות/נפילות חומרה.
אבסטרקציות במערכות הפעלה מספקת:
- **Cross-Platform:** מאפשר ליישומים להשתמש באותו API ללא קשר לחומרה הבסיסית או למערכת ההפעלה מה שמאפשר פיתוח אפליקציות בפלטפורמה אחת והרצה באחרת ללא צורך שינויים משמעותיים בקוד.
לדוגמה: מפתח משחקים יכול לפתח משחק שמשתמש בממשק לעיבוד גרפי מסוים במערכת הפעלה מסוג Windows, ולאחר מכן להעביר את המשחק בקלות להרצה במערכות הפעלה מסוג macOS או Linux על ידי קימפול הקוד מחדש עם הספריות המתאימות.
- **אבטחה:** שכבת הגנה נוספת בין אפליקציות למערכת ההפעלה, המונעת מיישומים לגשת או לשנות חלקים רגישים שלה.
לדוגמה: מערכת הקבצים מונעת מיישומים לגשת לקבצים שלא אמורה להיות להם גישה אליהם.
- **הסתרה של מורכבויות מיותרות:** מערכת ההפעלה היא כלי תכנתי שמתקשר עם רכיבי החומרה ומנהל אותם - אבסטרקציות עוזרות להסתיר את המורכבות הזו ומספקות ממשק פשוט למשתמשים ולמפתחי אפליקציות. זה מקל על השימוש והפיתוח של תוכנות במערכת ההפעלה.
לדוגמה: processes מסתירים את פרטי ניהול המשאבים של מערכת ההפעלה ומספקים תצוגה פשוטה של תוכנית שרצה.
- **וירטואליזציה של משאבים פיזיים:** אבסטרקציות המשמשות כשכבה וירטואלית נוספת בין התוכנה לבין החומרה ומאפשרות מיקבול, זמינות, גמישות.
לדוגמה: וירטואליזציה של הזיכרון - 'הקצאת' זיכרון וירטואלי ,'פרטי' לכל תהליך, כך שכל תהליך חושב שכל מרחב הזיכרון שייך לו (indirection level), מה שמאפשר ריבוי משימות (מספר תהליכים יכולים לפעול בו-זמנית) ו-time-sharing (לכל תהליך מוקצה נתח מזמן ה-CPU)

כלומר, אבסטרקציות הן חלק אינטגרלי ממערכות הפעלה, מכיוון שהן מספקות מענה יעיל ונרחב לרוב התפקידים של מערכת ההפעלה.

3. תנו דוגמה לאבסטרקציה מרכזית שמשתמשים בה במערכות הפעלה והסבירו מדוע משתמשים בה.
דוגמה מרכזית לאבסטרקציה היא וירטואליזציה של משאבים פיזיים, ובפרט – של מרחב הזיכרון. המשאבים הווירטואליים מספקים שכבת אבסטרקציה שמתעלמת מפרטי המימוש של החומרה הספציפית, ולכן הם פשוטים יותר לשימוש.
בפרט, מערכת ההפעלה מציגה למשתמש גרסאות וירטואליות של המעבד והזיכרון, כך שכל תכנית תקבל עותק וירטואלי של המשאב לשימושה ותחשוב שהיא רצה לבד.
הווירטואליזציה מאפשרת להריץ הרבה אפליקציות בו-זמנית, גם אם יש יותר אפליקציות להריץ מאשר ליבות במעבד, ולספק אשליה של מקבול המעבד.
משאבים וירטואליים אף מספקים הגנה ע"י הסתרת המשאב הפיזי, ובכך מונעים ממשתמשים או אפליקציות להשתלט עליו. (מצד שני, וירטואליזציה גם מוסיפה overhead שפוגע בביצועים).

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",…);
    char* argv[] = { "/bin/prog.out", NULL };
    execv(args[0], args);
    perror("execv");
    exit(-1);

} else {

   wait(NULL);

}
```

2. (7 נקודות)

לפניכם קטע קוד המשתמש ב-pipes.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  int main() {
6    int fd[2];
7    int count;
8    int id;
9    pipe(fd);
10   pid_t p = fork();
11   count = get_random_between(1, 1000000);//gets random integer in [1,1000000]
12   if (p == 0) {
13     id = 234123;
14     for(int i = 0; i < count; i++)
15     {
16    write(fd[1], (void*)(&id), sizeof(int));
17     }
18   } else {
19     close(fd[1]);
20     for(int i = 0; i < count; i++)
21    {
22    if(read(fd[0], (void*)(&id), sizeof(int)) > 0)
23       {
24       printf("My favorite course, %d\n", id);
25       }
26       else
27       {
28             break;
29       }
30    }
31  }
32   return 0;
33 }
```

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:

הבעיה בקוד היא שהבן שנוצר ע"י הקריאה ל-fork לא מבצע סגירה של ה-fd המשמש לקריאה ב-pipe .
התרחיש הבעייתי ביותר האפשרי הוא מצב שבו המשתנה count ששייך לזיכרון של האבא קטן יותר משמעותית מהמשתנה count ששייך לזיכרון של הבן, ואז הבן ימלא את ה-pipe בדאטה, ה-pipe יגיע ל-full capacity ויחסום את האפשרות לכתיבה נוספת של הבן.
בינתיים האבא סיים את ריצתו, מפני שלולאת הקריאה שלו רצה משמעותית פחות פעמים מלולאת הכתיבה של הבן, אך הבן לא ייעצר מקבלת הסיגנל SIGPIPE כי ה-pipe חושב שעדיין יש לו readers, מפני שהבן לא סגר את ה-fd של הקריאה.

חלק 2: (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).
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:

הקוד מגדיר handler שמטפל בסיגנל SIGFPE אשר פשוט מדפיס "Hello" ויוצאת מהתוכנית.
בפונקציה ()main נעשה ניסיון לחלק את המספר השלם 234123 ב-0, מה שיגרום ל-interrupt --> מעבר לטיפול בפסיקה ב-kernel mode --> מערכת ההפעלה שולחת את הסיגנל SIGFPE לתהליך --> חזרה לטיפול בסיגנל ב-user mode, כפי שראינו בתרגול 3.
מאחר והפונקציה ()fpe_cathcer הוגדרה כ-handler עבור הסיגנל הזה, היא נקראת, וההודעה "Hello" מודפסת לפני יציאת התוכנית.
הקריאה ל-()printf שאחרי החלוקה ב-0 (("Hi\n")printf) והקוד שאחריה לעולם לא יבוצעו, מכיוון שהתוכנית יוצאת לפני כן.

2. כעת נניח שבזמן כלשהו של ריצת הקוד הנ"ל, תהליך אחר שולח את הסיגנל SIGFPE לתהליך 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: לבאורה נחשוב ש-"Hello" יודפס פעמיים אם התהליך האחר ישלח את האות SIGFPE לתהליך שמריץ את הקוד ממש לפני ביצוע exit ובדיוק אחרי הדפסת ה-"Hello" הראשון.
אבל, מה שיקרה בפועל הוא לא שהפונקציה fpe_catcher תופעל מחדש באמצע טיפול בסיגנל הראשון, אלא תסיים טיפול בסיגנל הראשון, ולאחר מכן תעשה exit ותסיים לחלוטין את התהליך, מפני ששליחת סיגנל באמצע ריצת ה-handler של אותו סיגנל לא תעשה שום דבר, הסיגנל הנוסף פשוט ייחסם.

ייתכנו 2 מקרים:

I. האות נשלחה מהתהליך האחר לתהליך המריץ את התכנית לפני הרצת השורה:
```
int x = 234123 / (0)
```
והטריגה את fpe_catcher, שביצעה הדפסה יחידה של "Hello" ותסגור את **התהליך הרץ** (לא זה ששלח את הסיגנל!)

II. האות נשלחה מהתהליך האחר לאחר הרצת השורה הנ"ל, ובמקרה זה פונקציית ה-handler כבר באמצע טיפול בסיגנל, ולכן תבצע הדפסה יחידה ויציאה.

- לא יודפס "Hello" בכלל.

הקיפו: כן / לא

נימוק: אם תהליך אחר שולח את האות SIGFPE לתהליך המריץ את הקוד לפני שהפונקציה signal נקראת כדי להגדיר את ה-handler עבור האות SIGFPE (לפני שהתוכנית מספיקה להריץ את השורה הראשונה ב-main), אז יקרא handler-ברירת-המחדל של האות SIGFPE, שמסיים את התהליך, מבלי לתת לתוכנית הזדמנות לבצע קוד נוסף, ובפרט להדפיס הודעות נוספות.
במקרה זה, התוכנית בעצם תסתיים מיד, ללא ביצוע קוד נוסף או הדפסת הודעות כלשהן.
כתוצאה מכך, בתרחיש זה "Hello" לא יודפס למסך אפילו פעם אחת.

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

לאחר השורה המבצעת חלוקה ב-0 תופעל פסיקה שתטופל ב-kernel mode, ולאחר מכן מערכת ההפעלה תשלח סיגנל לתהליך, שיעורר את פונקציית ה-fpe_catcher handler ב-user mode, כפי למדנו בתרגול 3.
אם נסיר את הקריאה exit(0) מהפונקציה fpe_catcher(), אז התוכנית לא תצא לאחר הדפסת "Hello". במקום זאת, נחזור לשורה הבעייתית ב-main (כפי שנעשה בפסיקות מסוג זה), נבצע חלוקה ב-0, נחזור לפונקציית ה-handker וחוזר חלילה .
המשמעות היא שנקרא ל-handler אינסוף פעמים (או עד שייגמר זיכרון התכנית) ונדפיס "Hello" למסך אינסוף פעמים.

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:

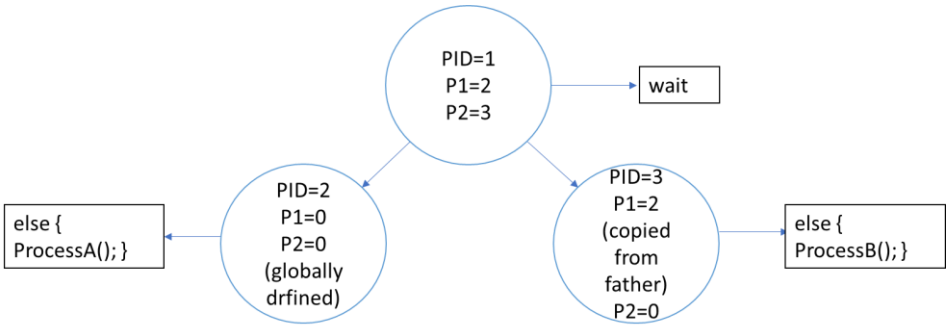
1. קריאות המערכת fork() וkill() אינן נכשלות.
2. כל שורה הנכתבת לפלט אינה נקטעת ע"י שורה אחרת.
3. כאשר תהליך מקבל סיגנל x הוא מסתיים וערך היציאה שלו הוא x + 128.

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:

בפונקציית ה-main, האב יוצר 2 בנים באמצעות fork().
הבן שנוצר בקריאה ל-fork() הראשון, אינו נכנס ל-if הראשון, מפני שערך החזרה שהוא מקבל מ-fork שווה ל-0, בהיותו הבן.
האב נכנס לביצוע ל-if, מפני שערך החזרה שהוא מקבל מ-fork הוא ה-PID של תהליך הבן שהוא יצר, שבהכרח שונה מ-0, כפי שראינו בתרגול 2.
באותו אופן, תהליך הבן שנוצר אינו נכנס ל-if השני, מפני שערך החזרה שקיבל הוא 0.
האב ממשיך ונכנס ל-if השני, וממתין.
הבן הראשון שנוצר בינתיים המשיך ל-else שמתחת ל-if הראשון (שכן הוא אינו נכנס ל-if) וקרא ל-ProcessA().
הבן השני שנוצר בינתיים המשיך ל-else שמתחת ל-if השני, וקרא ל-ProcessB().
זוהי תמונת המצב:



נשים לב שעבור הבן הראשון, הערך של המשתנה הגלובלי X לא משתנה ונשאר 1 (הוא משתנה עבור הבן השני, אבל יש להם מרחבי זיכרון נפרדים!), ולכן , לולאת ה-while בפונקציה ProcessA() היא לולאה 'אינסופית', שרצה עד שב-ProcessB מתבצעת הקריאה ל-killAll(), שבה P1=2 (שכן fork מעתיקה את מרחב הזיכרון מהאב) ו-P2=0 (שכן זהו ערך החזרה שהבן קיבל מ-fork).

לכן בפועל מתבצע:

```
if(0) kill(0, 15);
if(2) kill(2, 9);
```

לכן, נשלח SIG_KILL לתהליך הבן הראשון.
נשים לב ששני תהליכי הבנים רצים יחד – בו זמנית, לכן ייתכן שהבן השני שלח SIG_KILL לבן הראשון לפני שהבן הראשון הספיק לבצע `printf("process A\n")` ואז לא יודפס למסך הפלט process A, או שהבן השני שלח SIG_KILL לאחר ביצוע ההדפסה, ובמקרה זה יודפס למסך הפלט process A פעם אחת.
בשל האינסופיות של לולאת ה-while, לעולם לא תתבצע השורה שאחריה ולא יודפס למסך `"process A finished\n"`.

2. status: 1

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

נימוק:

בהמשך להסבר מסעיף 1, הבן הראשון יסתיים תמיד ע"י פקודת kill ששולח לו הבן השני עם SIG_KILL, ולכן אף פעם לא יספיק להגיע ליציאה עצמאית ע"י exit(1).
לכן, כפי שמצוין בהערה 3, כאשר תהליך מקבל סיגנל x הוא מסתיים וערך היציאה שלו הוא x + 128, הבן השני מקבל סיגנל שמספרו 9, ולכן ערך היציאה שלו תמיד יהיה 9+128=137, ואף פעם לא 1.
לכן תהליך האב ידפיס: `"status: 137"` ולעולם לא ידפיס: `"status: 1"`.
בינתיים, תהליך הבן השני ממשיך את ריצתו. נשים לב שבקריאה שלו ל-()killAll, הערך של p2 הוא תמיד 0, ולכן לא תהיה כניסה ל-if הראשון בפונקציה זו אף פעם.
לכן, ()ProcessB יחזיר 1 לפונקציית ה-main שקראה לו, וימשיך להריץ את שאר הפקודות בפונקציה ()main, עד שלבסוף יחזיר 3, מה שיטריג הרצה של exit(3) ע"י `main_start_libc()`, כפי שראינו בתרגול 2.
האב ממתין ששני תהליכי הצאצא יסיימו (ע"י שימוש בקריאת המערכת ()wait פעמיים) ואז ממשיך בביצוע, מדפיס פעם אחת `"status: 137"`,ופעם אחת `"status: 3"`, מפני שתהליך הבן השני הסתיים עם סטטוס יציאה 3.
לכן, גם במקרה של הבן השני, לא יודפס: `"status: 1"`.

3. status: 137

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

נימוק:

בהמשך להסבר מסעיף 2, הבן הראשון יסתיים תמיד ע"י פקודת kill ששולח לו הבן השני עם SIG_KILL, ולכן אף פעם לא יספיק להגיע ליציאה עצמאית ע"י exit(1).
לכן, כפי שמצוין בהערה 3, כאשר תהליך מקבל סיגנל x הוא מסתיים וערך היציאה שלו הוא x + 128, הבן השני מקבל סיגנל שמספרו 9, ולכן ערך היציאה שלו תמיד יהיה 9+128=137, ואף פעם לא 1.
הבן הראשון מסתיים ע"י סיגנל עם סטטוס יציאה 137, ולאחר ש-ProcessB() מחזירה 1, תהליך הבן השני ממשיך להריץ את שאר הפקודות בפונקציה main(), עד שלבסוף יחזיר 3, מה שיטריג הרצה של exit(3) ע"י `main_start_libc()`, כפי שראינו בתרגול 2.
לאחר ששני תהליכי הבנים הסתיימו, תהליך האב ממשיך בביצוע ומדפיס פעם אחת `"status: 137"`, כפי שהוסבר בסעיף 2.

4. status: 143

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

נימוק:

בהמשך להסבר מסעיף 1, תהליך הבן הראשון יסתיים עם סטטוס יציאה 137.
בינתיים, תהליך הבן השני ממשיך את ריצתו.
נשים לב שבקריאה שלו ל-killAll(), הערך של p2 הוא תמיד 0, ולכן לא תהיה כניסה ל-if הראשון בפונקציה זו אף פעם, לכן אף תהליך לא יקבל kill עם סיגנל מספר 15, ולכן אף תהליך לא יסיים עם סטטוס יציאה 143=15+128.
ProcessB() יחזיר 1 לפונקציית ה-main שקראה לו ויסתיים כרגיל עם סטטוס יציאה 3, ולעולם לא יודפס "status: 143".

5. The end

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

נימוק:

בהמשך להסבר מסעיף 2, תהליך הבן הראשון יסתיים בהכרח באמצע הרצת הפונקציה ProcessA ולעולם לא יחזור ל-main.
עבור תהליך הבן השני, לאחר ש-processB() מסיים את הביצוע ומחזיר 1, הוא ממשיך להריץ את שאר הפקודות בפונקציה main(), עד שלבסוף ידפיס "The end" ויחזיר 3.
תהליך הבן השני מסתיים עם סטטוס יציאה 3 באמצעות main_start_libc() ומחזיר את השליטה לתהליך האב.
בשלב זה, תהליך האב מסיים את הדפסת הסטטוסים וממשיך לבצע את הפונקציה main ומדפיס "The end". לבסוף התוכנית יוצאת עם סטטוס יציאה 3.
בסה"כ גם תהליך הבן השני וגם תהליך האב ידפיסו את הודעת סיום התוכנית ולכן היא תודפס פעמיים.