שאלה 1

א. מה יודפס על המסך עבור הקוד הבא:

```
int main() {
    int res = fork();
    if (res != 0) {
            close(STDOUT);
    }
    int fd = open("myFile", O_RDWR);
    if (res != 0) {
            printf("Hello from father\n");
    }
    else {
            printf("Hello from son\n");
    }
}
```

- תהליך האב סוגר את $STDOUT$ שנמצא בכניסה ה-1 ב-PTD ואז פותח את $MyFile$ במקומו, לכן "$Hello\ from\ father$" ייכתב אל תוך הקובץ $MyFile$. תהליך הבן פותח את $MyFile$ בכניסה כלשהי ב-PTD (שאינה במקום ה-1), לכן "$Hello\ from\ son$" יודפס לטרמינל. לכן על המסך יודפס "$Hello\ from\ son$".

ב. כיצד תשתנה התשובה לסעיף הקודם אם נשנה את הקוד כך:

```
int main() {
    close (STDOUT);
    int res = fork();
    int fd = open("myFile", O_RDWR);
    if (res != 0) {
            printf("Hello from father\n");
    }
    else {
            printf("Hello from son\n");
    }
}
```

- תחילה תהליך האב סוגר את $STDOUT$ שנמצא בכניסה ה-1 ב-PDT. לאחר מכן פעולת ה-$fork$ יוצרת תהליך בן אשר גם עבורו הכניסה ה-1 ב-PDT פנויה. גם תהליך האב וגם תהליך הבן יפתחו את $MyFile$ בכניסה ה-1 בטבלת ה-PDT במקום $STDOUT$. לכן שני התהליכים יכתבו אל הקובץ $MyFile$ במקום אל המסך, ועל המסך לא יודפס דבר.

ג. נתון קטע הקוד הבא (הניחו ש-buf מאותחל במקום כלשהו)

```
#define BUF_SIZE 100000
int my_pipe[2];
pipe (my_pipe);
char buf[BUF_SIZE];
int status = fork();

// Filled buf with message…
if (status == 0) { /* son process */
    close(my_pipe[0]);
    write(my_pipe[1],buf,BUF_SIZE*sizeof(char));
    exit(0);
}
else { /* father process */
    close(my_pipe[1]);
    wait(&status);  /* wait until son process finishes */
    read(my_pipe[0], buf, BUF_SIZE * sizeof(char));
    printf("Got from pipe: %s\n" buf);
    exit(0);
}
```

1. מה הבעיה בקטע הקוד הנתון?

- $read$ ו-$write$ לא מתחייבות לקרוא/לכתוב את כל הנתונים שב-$pipe$ (בפועל מובטח כי לפחות תו אחד ייכתב או ייקרא) לכן לא מבוטח כי תהליך הבן יצליח לכתוב את כל מה שב-buf אל ה-$pipe$ וכן לא מובטח שתהליך האב יצליח לקרוא מה-$pipe$ את כל מה שתהליך הבן כתב אליו.

2. הראו כיצד ניתן לפתור את הבעיה ע"י תיקון הקוד שמבצע תהליך הבן.

- על מנת לפתור את הבעיה נציע לתקן את הקוד שמבצע תהליך הבן כך

```
close(my_pipe[0]);
int n = BUF_SIZE;
int res;
while (n > 0){
    res = write(my_pipe[1], buf, n*sizeof(char);
    n -= res;
    buf = ((char*)buf) + res;
}
exit(0);
```

כלומר, נוסיף לולאה בתהליך הבן שתדאג שכל התווים ב-buf יקראו אל תוך ה-$pipe$ לפני סיום קטע הקוד של תהליך הבן.

ד. נתון הקוד הבא:

```
int x = 0;
int i = 3;

void catcher3(int signum){
    i = 1;
}

void catcher2(int signum){
    if(i != 0){
            x = 5;
    }
}

void catcher1(int signum){
    printf("%d\n", i);
    i--;
    if (i == 0){
            signal(SIGFPE, catcher2);
            signal(SIGTERM, catcher3);
    }
}

int main(){
    signal(SIGFPE, catcher1);
    x = 10/x;
    printf("Goodbye");
}
```

1. מה יודפס למסך כאשר מריצים את הקוד כמו שהוא? נמקו.

- שגרת הטיפול ב-$SIGFPE\ signal$ מוחלפת בשגרה $catcher1$.

עם ביצוע השורה "$x = 10/x$" ב-$main$ נעשית חלוקה ב-0 ולכן נשלח ה-$SIGFPE\ signal$ ונקראת בשגרה $catcher1$. לכן על המסך מודפס i וערכו מוקטן ב-1, לאחר מכן התוכנית תחזור על השורה שגרמה לשליחת ה-$signal$, כלומר השורה "$x = 10/x$" תתבצע בשנית. על המסך יודפס כך:

```
3
2
1
```

לאחר שערכו של i מגיע להיות 0, מוחלפת שגרת הטיפול ב-$SIGFPE\ signal$ לשגרה $catcher2$, כאשר זו אינה תבצע דבר משום ש-i לעולם לא יהיה שונה מ-0 יותר. כלומר, התוכנית תכנס ללולאה אינסופית ו $Goodbye$ לא לעולם לא יודפס.

2. מה יודפס למסך אם נתון שלאחר 10 שניות מבצעים את הפקודה $kill < PID >$ בטרמינל (כאשר ה-PID הנתון הוא של התהליך שמריץ את הקוד הנ"ל).

- בהנחה כי 10 שניות הם זמן מספיק עבור ערכו של i להשתנות ל-0, ביצוע הפקודה $kill < PID >$ תשלח $signal$ $SIGTERM$ אשר יפעיל את שגרת הטיפול $catcher3$. ערכו של i ישתנה ל-1 וכאשר תתבצע השורה "$x = 10/x$" שוב, שגרת הטיפול $catcher2$ תשנה את ערכו של המשתנה x ל-5. לכן, בנוסף להדפסות שמבצעת השגרה $catcher1$, כעת תבצע התוכנית את השורה $x = 10/x$ מבלי לגרום ל-$SIGFPE\ signal$, ותודפס גם ההודעה $Goodbye$. על המסך יודפס כך:

```
3
2
1
Goodbye
```

3. מה יודפס למסך אם נוסיף פקודת $sleep(100);\ //100\ seconds$, אחרי השורה "$signal(SIGFPE, catcher2)$" שנמצאת בפונקציה $catcher1$? (כאשר עדיין שולחים סיגנל בעזרת $kill$ לאחר 10 שניות מתחילת ריצת התוכנית).

- הוספת הפקודה $sleep(100)$ תגרום לתוכנית להמתין במשך 100 שניות במצב בו לא יכולים להתקבל $signal$ מסוג $SIGFPE$ אך יכולים להתקבל $signal$ מסוגים אחרים. במהלך המתנה זו, ולפני החלפת השגרה לטיפול ב-$signal$ $SIGTERM$ לשגרה $catcher3$, מתקבל ה-$SIGTERM\ signal$. התוכנית, התהליך שמתעורר מהמתנה חוזר מרמת ה-$kernel$ לרמת ה-$user$, ומתפנה לטיפול ב-$signal$ זה. שגרת הטיפול הנקראת היא השגרה המקורית ולכן הטיפול נעשה לפי ברירת המחדל, כלומר בעזרת $terminate$, המסיים את התהליך. התהליך יסתיים כאשר על המסך מודפס כך:

```
3
2
1
```

שאלה 2

א. סערת פסיקות ($interrupt\ storm$) הוא כינוי לתופעה כשאר קצב הגעת הפסיקות גבוה מאוד. מה היא הבעיה במצב זה וכיצד להתמודד איתה?

- כאשר פסיקות מגיעות בקצב הגעה גבוה מאוד ייתכן מצב בו פסיקות מגיעות כאשר הפסיקות במעבד חסומות (דגל IF כבוי). כתוצאה מכך ביצוע פסיקות קריטיות עלול להתעכב ואף פסיקות עלולות להיאבד (וכך עלול להיאבד מידע). על מנת להתמודד עם בעיה זו, מחולק הטיפול בפסיקה לשלבי טיפול, כאשר הקטע הקריטי ($critical$) בכל טיפול מצומצם למינימום ומבוצע מיד עם קבלתה. לאחר השלב הקריטי, מודלק דגל IF חזרה והמעבד מאפשר קבלה של פסיקות חדשות. שאר חלקי הטיפול בפסיקה מטופלים מיד לאחר סיום השלב הקריטי של הפסיקה החדשה ($noncritical$), עבור חלקים הדורשים התייחסות דחופה (אך לא מידית), וחלקי טיפול אשר אינם דורשים התייחסות מידית מועברים למאגר משימות ממתינות אשר טיפולן נעשה בעדיפות נמוכה הרבה יותר.

ב. נתון כרטיס רשת שעובד בקצב של $20\ Gbit/sec$ ושולח ומקבל חבילות ($packets$) בגודל קבוע של $4096\ bit$. נניח כי מערכת ההפעלה רוצה לעבוד עם הכרטיס בשיטת $polling$.

1. מדוע לא כדאי לקבוע את תדר ביצוע ה-$polling$ ל- $0.5\ GHz$.

- מאחר וקצב עבודת כרטיס הרשת הוא $20\ Gbit/sec$, הוא מסוגל לשלוח (או לקבל) חבילה בגודל $4096\ bit$ אחת לכ-$0.2\ ms$. לא כדאי לקבוע את תדר ביצוע ה-$polling$ ל-$0.5\ GHz$ מאחר ומערכת ההפעלה תדגום את מוכנות כרטיס הרשת אחת ל-$0.002\ ms$, כך שרק אחת לכמאה דגימות יהיה המידע בכרטיס הרשת מוכן. מתוך כך נובע כי מערכת ההפעלה תבזבז הרבה זמן לחינם בעקבות $polling$ בתדירות כזו.

2. מדוע לא כדאי לקבוע את תדר ביצוע ה-$polling$ ל-$5\ KHz$.

- מאחר קצב עבודת כרטיס הרשת הוא $20\ Gbit/sec$, הוא מסוגל לשלוח (או לקבל) חבילה בגודל $4096\ bit$ אחת לכ-$0.2\ ms$. לא כדאי לקבוע את תדר ביצוע ה-$polling$ ל-$0.5\ GHz$ מאחר ומערכת ההפעלה תדגום את מוכנות כרטיס הרשת אחת ל-$200\ ms$, כך שקצב עבודת כרטיס הרשת יהיה פי אלף מהיר יותר מקצב דגימת מערכת ההפעלה. מתוך כך נובע כי מערכת ההפעלה לא תאפשר שימוש יעיל בכרטיס הרשת ואף עלולות לאבד חבילות, בעקבות $polling$ בתדירות כזו.

3. בתרגול ראינו כי בפונקציה $schedule$, לפני שניגשים לבצע פעולות על ה-$runqueue$ משתמשים בפקודה $spin\_lock\_irq$ אשר תופסת את ה-$spinlock$ של ה-$runqueue$ וחוסמת את הפסיקות במעבד הנוכחי.

א. מדוע יש צורך לחסום פסיקות ולא ניתן להסתפק בתפיסת מנעול?

- יש לחסום פסיקות בנוסף לתפיסת מנעול ה-$runqueue$ על מנת לדאוג לקונסיסטנטיות המערכת. מבנה הנתונים $runqueue$ נגיש גם כתוצאה מפסיקות ולכן הטיפול בפסיקות במהלך תזמון תהליכים והחלפת הקשר עלול לגרום לשינוי מצב המערכת תוך כדי ביצוע ההחלפה וכך להביא למצב של אי נכונות תוצאת פעולות אלו.

ב. באיזו בעיית סנכרון היינו נתקלים אם לא היינו חוסמים את הפסיקות? יש לתאר מצב בו מתרחשת בעיה זו.

- ללא חסימת הפסיקות ניתן להגיע למצב של $deadlock$ בעקבות קבלת פסיקה בזמן ריצת ה-$schedule$. אם במהלך ריצת ה-$schedule$, לאחר תפיסת המנעול תתקבל פסיקת שעון, במהלך הטיפול בה בפונקציה $scheduler\_tick$ יתבצע ניסיון לתפוס את המנעול גם כן.

4. נתונה מערכת בעלת שני מעבדים A ו-B אשר מריצים תהליכים PA ו-PB בהתאמה. ברגע תהליך PA שולח $USR1\ signal$ אל תהליך PB. הניחו כי מערכת ההפעלה אינה מבצעת החלפת הקשר ושהפסיקות והסיגנלים לא חסומים.

א. מה פרק הזמן המקסימלי בין הרגע בו נשלח הסיגנל לרגע בו PB יתחיל לטפל בו?

- פרק הזמן המקסימלי בין הרגע בו נשלח הסיגנל לרגע בו PB יתחיל לטפל בו הוא הזמן שיעבור עד אשר תתקבל פסיקת השעון הבאה במעבד B. במהלך החזרה לרמת ה-$user$ מרמת ה-$kernel$, העובדה שתהליך B קיבל סיגנל תתגלה והתהליך יתחיל בטיפול בסיגנל. לכן הזמן המקסימלי הוא הזמן שבין פסיקת שעון במעבד B ועד לפסיקת השעון הבאה.

ב. תן דוגמא למצב בו PB יתחיל לטפל בסיגנל לאחר פרק זמן קצר יותר.

- במידה ומיד לאחר שליחת הסיגנל על ידי תהליך PA, תהליך PB יצבע קריאת מערכת כלשהי, יתבצע מעבר לרמת ה-$kernel$ במעבד B. עם סיום הטיפול בפסיקה יתבצע מעבר חזרה לרמת ה-$user$ במהלכו יתגלה הצורך בטיפול בסיגנל. במידה וביצוע קריאת המערכת קצר יותר מהזמן בין שתי פסיקות שעון, נקבל מצב בו גילוי הסיגנל ותחילת הטיפול בו יתחילו בפרק זמן קצר יותר מזה שיעבור בין שתי פסיקות שעון. יש לציין שאכן קיימות קריאות מערכת אשר הטיפול בהן קצר מאוד (לדוגמא קריאת המערכת $getpid()$) המאפשרות מקרה כזה.

5. תארו את ההבדל בהתנהגות של הפקודה $iret$ בעת חזרה מפסיקה מקוננת לעומת חזרה מפסיקה לא מקוננת.

- הפקודה $iret$ משחזרת את הערכים שעל ראש המחסנית. השוני בהתנהגות הפקודה הוא שבעת חזרה מפסיקה לא מקוננת ישנו מעבר בין רמות (מרמת ה-$kernel$ לרמת ה-$user$) ולכן ישנו צורך לבצע החלפת מחסניות ולעבור למחסנית ה-$user$. לאומת זאת בחזרה מפסיקה מקוננת, אין צורך למעבר כזה ולכן אין צורך לבצע החלפת מחסנית.