Homework 1 Dry

**Due Date: 20/11/2018 23:30**

Teaching assistant in charge:

* Shalev Kuba

**Important:** the Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory, and it is your responsibility to be updated. A number of guidelines to use the forum:

* Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
* Be polite, remember that course staff does this as a service for the students
* You’re not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
* When posting questions regarding hw1, put them in the hw1 folder

Only the TA in charge can authorize postponements. In case you need a postponement, please fill out the following form: <https://docs.google.com/forms/d/e/1FAIpQLSftN-3vAFuM3pmzHjoWb-QokiROxWZG4q75osCe6ItVnwXLig/viewform>

Dry part submission instructions:

1. Please submit the dry part to the electronic submission of the dry part on the course website.
2. The dry part submission must contain a single dry.pdf file containing the following:
   1. The first page should contain the details about the submitters - Name, ID number and email address.
   2. Your answers to the dry part questions.
3. Only typed submissions will be accepted. Scanned handwritten submissions will not be accepted. Only PDF format will be accepted.
4. You do not need to submit anything in the course cell.
5. When you submit, **retain your confirmation code and a copy of the PDF**, in case of technical failure. It is **the only valid proof** of your submission.

## **יש לנמק כל תשובה, תשובות ללא נימוק לא יתקבלו.**

**שאלה 1 (50 נק')**

* **את שאלה זו יש לפתור לאחר שפתרתם את החלק הרטוב.**

1. (10 נק') מהו סדר הפונקציות/רוטינות בביצוע קריאת המערכת open?  
   1. sys\_open → system\_call → libc wrapper
   2. sys\_open →libc wrapper → system\_call
   3. system\_call → sys\_open → libc wrapper
   4. system\_call → libc wrapper→ sys\_open
   5. **libc wrapper→ system\_call → sys\_open**
   6. libc wrapper→ sys\_open → system\_call

פרטו את צורת התקשורת בכל קריאה לפונקציה/רוטינה (העברת פרמטרים והחזרת תוצאות חישוב):

* + - 1. **libc wrapper** – זו פונקצית המעטפת בUser Space. הפרמטרים מועברים אליה כמשתנים בC ע״י המתכנת. ערכי השגיאה מוחזרים במשתנה הגלובלי errno ומוחזר ערך מספרי המציין הצלחה (0) או שגיאה (-1). היא קוראת לinterrupt 128 (system\_call) ומעבירה את הפרמטרים לקריאת המערכת, לטיפול מערכת ההפעלה.
      2. **system\_call** – זו קריאת המערכת המטופלת ע״י interrupt 128 (0x80). היא מקבלת את הפרמטרים שלה מפונקציית המעטפת בתוך הרגיסטרים הרלוונטיים על המחסנית. את ערך החזרה היא שומרת ברגיסטר eax. ברגיסטר eax יתקבל מספר קריאת המערכת (5), ברגיסטר ebx נעביר את כתובת הקובץ לפתיחה, ברגיסטר ecx נעביר את הדגלים לטיפול בקובץ. קריאת המערכת תשמור את הרגיסטרים בסדר הפוך במחסנית
      3. **sys\_open** – נקראת ע״י מערכת ההפעלה (system\_call). מקבלת את הפרמרטים שלה בC (למעשה מקבלת אותם מהמחסנית ע״י מערכת system\_call), ומחזירה מספר שלם המציין את ערך החזרה עם סימן שלילי (0 עבור הצלחה, -errno עבור שגיאה). הערך שחוזר מפונקציית המעטפת יהיה 0 עבור הצלחה, ו--1 עבור כישלון, ובמקרה כזה ערך השגיאה בערך מוחלט יישמר בerrno הגלובלי.

תזכורת: בחלק הרטוב הגדרנו רמות הרשאה לתהליכים וכן ספי הרשאה למספר קריאות מערכת "מאובטחות": fork, wait, waitpid, sched\_yield. במערכת של תרגיל הבית, תהליך בעל רמת הרשאה x יוכל לבצע קריאת מערכת בעלת סף הרשאה y אם ורק אם .

נרצה להרחיב את מנגנון האבטחה כך שניתן יהיה לעדכן את רשימת קריאות המערכת המאובטחות ואת ספי ההרשאה שלהן ללא צורך בהידור מחדש של הגרעין. לשם כך, ניצור קובץ קונפיגורציה במיקום קבוע וידוע למערכת ההפעלה אשר מכיל רשימה של קריאות מערכת וסף ההרשאה המתאים לכל אחת מהן. המשתמש יכול לערוך את קובץ הקונפיגורציה באופן דינמי וכך לעדכן את מנגנון האבטחה ללא צורך בקידוד והידור מחדש של הגרעין.  
שימו לב: מנגנון האבטחה נדרש להיות מעודכן לכל הפחות לזמן הפעלת המערכת האחרונה. כלומר, עריכת הקובץ בזמן ריצת המערכת לא בהכרח משפיעה מיד על מנגנון האבטחה.

לשם מטרה נעלה זו התגייסו כוכבי "בית הנייר" והציעו מימושים שונים להרחבה.

ההצעה של טוקיו: רמת ההרשאה של תהליך תשמר במשתנה גלובלי בגרעין מטיפוס integer. כמו כן, בתחילת כל פונקצית שירות (sys\_X) תמקם קריאה לפונקציית גרעין בשם checkSec עם מספר קריאת המערכת. checkSec תקרא את קובץ הקונפיגורציה ותבדוק האם התהליך בעל הרשאה לבצע את קריאת המערכת.

1. (8 נק') האם המימוש של טוקיו תקין? אם לא, מה הבעיה במימוש זה?

**המימוש של טוקיו אינו תקין. במימוש זה, אם רמת ההרשמה של תהליך תישמר במשתנה גלובלי, כל תהליך אחר יוכל לגשת למשתנה זה ולשנות אותו ללא צורך בהרשאות מנהל. זה יוצר זליגת אבטחה במערכת ההרשאות**

ההצעה של ריו: רמת הרשאה של תהליך תשמר כשדה במתאר התהליך (PCB). כמו כן, ריו ימקם קריאה בתחילת system\_call לפונקציית גרעין בשם checkSec עם מספר קריאת המערכת. הפונקציה checkSec תקרא את קובץ הקונפיגורציה ותבדוק האם התהליך בעל הרשאה לבצע את קריאת המערכת.

1. (8 נק') האם המימוש של ריו תקין? אם לא, מה הבעיה במימוש זה?

**המימוש של ריו תקין, כל כך תהליך שומר את רמת ההרשאה שלו ותהליכים אחרים לא יכולים לגשת אליה בצורה ישירה.**

ההצעה של הפרופסור: רמת הרשאה של תהליך תשמר כשדה במתאר התהליך (PCB). כמו כן, הפרופסור יתחזק רשימה גלובלית בגרעין אשר מכילה מספרי קריאות מערכת וספי הרשאה המתאימים להן. את רשימה זו יאתחל רק בעלייה של המערכת על ידי קריאת קובץ הקונפיגורציה. בנוסף, ימקם קריאה בתחילת system\_call לפונקציית גרעין בשם checkSec עם מספר קריאת המערכת. הפונקציה checkSec תבדוק האם התהליך בעל הרשאה לבצע את קריאת המערכת על ידי מעבר על הרשימה הגלובלית.

1. (8 נק') האם המימוש של הפרופסור תקין? אם לא, מה הבעיה במימוש זה?

**המימוש של הפרופסור תקין, כל כך תהליך שומר את רמת ההרשאה שלו ותהליכים אחרים לא יכולים לגשת אליה בצורה ישירה.**

1. (8 נק') מה היתרון של ההצעה למימוש של הפרופסור על פני המימוש של ריו?

**במימוש של הפרופסור, הקריאה יותר מהירה, שכן קריאה ממשתנה בגרעיון מהירה הרבה יותר מקריאת קובץ. בהנחה יש המון תהליכים רצים במערכת, קריאות רבות לקובץ אחד יכולות לקחת הרבה זמן ולהעמיס על הגרעין. יתרה מכך, אם רוצים לשנות את הקובץ יש לכתוב אליו, וזה יכול לגרום לתחרות בין תהליכים בין כתיבה וקריאה לקובץ.**

1. (8 נק') מה היתרון של ההצעה למימוש של ריו על פני המימוש של הפרופסור?

**במימוש של הפרופסור, בכל פעם שרוצים להחיל שינוי על מערכת ההרשאות, יש להפעיל מחדש את מערכת ההפעלה, כי קובץ הקונפיגורציה נקרא פעם אחת ומתעדכן בעליית הגרעין. לעומת זאת במימוש של ריו, מתבצעת קריאה בכל פעם מקובץ הקונפיגורציה ולכן מערכת ההרשאות תמיד מעודכנת ואין צורך בהפעלה מחדש.**

**שאלה 2 (50 נק')**

1. (18 נק')
2. (10 נק') ניר, ששונא לחכות, וגם מאמין במשפט "מה ששנוא עליך אל תעשה לתהליכך", החליט שבכל תוכניות המחשב שהוא כותב, הוא לעולם לא ישתמש בקריאת המערכת  
   ()wait. ליאור העיר לניר שאם לא ישתמש בקריאת המערכת הנ"ל ייאגר לו מידע בזיכרון על תהליכיו אשר סיימו להתבצע אך לא בוצע להם wait ("זומבים"), האם ליאור צודק? הסבר את טענתך.  
   **הערה**: ניתן להניח כי ניר לא כותב תכניות בהן קיים תהליך שרץ זמן רב.

**ליאור אינו צודק. אנחנו יודעים שתהליך הוא zombie אם הוא קיים כרשומת נתונים בלבד לאחר שסיים את ביצועו. במקרה כזה, הרשימה תימחק כאשר האב קרא ל-wait() וקיבל חזרה ערך המצביע על כך שהתהליך סיים. במקרה של ניר, הפונקציה wait() לא נקראת לעולם. אבל אנחנו יודעים שתהליך שאביו מסיים לפעול ולא קרא ל-wait() עבורו, הופך ל״יתום״. כל התהליכים היתומים הופכים להיות בנים של התהליך init, וזה אחראי לקרוא ל-wait() עבור בניו, ובכך לשחרר את רשומות הנתונים שלהם. לכן, גם אם ניר לא יקרא ל-wait(), התהליכים הבנים בתכנית ישוחררו בסופו של דבר על ידי init.**

1. (8 נק') שקד, שלמד על קריאת המערכת ()fork, רצה להתנסות בבית בשימוש בה, ולכן כתב את קטע הקוד הבא:

int main(){

int forkId=fork();

if(forkId==0){//son

printf("hey father, I am your son\n");

}else{//father

printf("hey son, I am your father\n");

}

return 0;

}

למרבה הצער, על המסך הודפס הפלט הבא (בהרצה מסוימת):

hey son, I am your father

hey father, I am your son

שקד התבאס מאוד שכן רצה שקודם הבן ידפיס למסך את ההודעה ורק לאחר מכן האב ידפיס את ההודעה שלו. עזרו לשקד, ע"י הוספת שורת קוד אחת בלבד, לגרום לתוכנית להדפיס **בכל הרצה** את הפלט:

hey father, I am your son

hey son, I am your father

**תשובה:**

int main(){

int forkId=fork();

if(forkId==0){//son

printf("hey father, I am your son\n");

}else{//father

**wait(NULL); // השורה שהוספנו – האב מחכה לבן ורק אז מדפיס את ההודעה שלו**

printf("hey son, I am your father\n");

}

return 0;

}

1. (15 נק') כזכור, מתאר התהליך מאוחסן ביחד עם מחסנית הגרעין שלו בקטע זיכרון בגודל 8KB המתחיל בכתובת מיושרת.

חברת נינוקס, החליטה לפתח מערכת הפעלה מודרנית יותר מהמערכת הנלמדת בתרגולים. בפרט, החברה טענה שלא יתכן שגודל כל מתאר תהליך יהיה מוגבל בגודלו, ולכן הפרידה את מתאר התהליך ממחסנית הגרעין (מתאר התהליך ומחסנית הגרעין כבר אינם צמודים כפי שנלמד בתרגולים) כך שגודל מתאר התהליך אינו מוגבל במערכת החדשה. נינוקס, שהעתיקה מלינוקס את קוד הקרנל הנלמד בתרגולים בלי לשנות דבר מלבד הפרדת מתאר התהליך ממחסנית הגרעין, הופתעה לגלות, יום לפני ההפצה של המערכת, שכאשר מבצעים קריאת מערכת שדורשת גישה למתאר התהליך, המערכת קורסת. עזרו לנינוקס להבין היכן הטעות שלה. על תשובתכם להיות מפורטת.

**כאשר אנחנו פונים למתאר התהליך בקרנל, אנחנו עושים זאת באמצעות המאקרו current המוגדר בקובץ current.h. מאקרו זה מאפס את 13 הביטים הנמוכים של כתובת הesp, וכך ניתן לחשב את תחילת בלוק ה-8KB המיושר בזיכרון, שמתחיל כאמור במתאר התהליך. חברת נינוקס החליטה להפריד את מתאר התהליך ומחסנית הגרעין, כלומר כעת הם אינם יושבים על אותו בלוק 8KB ביחד. במצב כזה, כאשר יתבצע ניסיון לקרוא ממתאר התהליך (שסביר שייעשה באמצעות המאקרו current) תחושב הכתובת הלא נכונה ויוחזר ערך זבל, שיגרום לקריסת המערכת (למשל, בגלל segmentation fault הנובע מקריאת ערכים מזיכרון לא חוקי)**

1. (17 נק')

בשאלה זו נדון בקשרי המשפחה כפי שבאים לידי ביטוי בשדות התהליך (p\_(o)pptr, p\_ysptr,…) ונלמדו בתרגולים:

1. (10 נק') עבור קטע הקוד הבא, ציירו את הגרף המתאר את קשרי המשפחה, כנלמד בתרגולים, כפי שנראה במערכת רגע לפני שתהליך כלשהו מסתיים (ניתן להניח כי כל התהליכים בתכנית נוצרים לפני שתהליך כלשהו נגמר). הקפידו לרשום על כל חץ את שם השדה ובתוך הצומת רשמו את המחרוזת שאותה התהליך מדפיס:

**///(continue)**

**.**

**.**

**for(int i=4;i<6;i++){**

**forkId=fork();**

**if(forkId==0){**

**printf("P%d",i);**

**return 0;**

**}**

**}**

**return 0;**

**}**

**int main(){//father**

**printf("P0");**

**int forkId=fork();**

**if(forkId==0){**

**printf("P1");**

**forkId=fork();**

**if(forkId==0){**

**printf("P2");**

**return 0;**

**}**

**forkId=fork();**

**if(forkId==0){**

**printf("P3");**

**}**

**return 0;**

**}//more code on the right**

**ציירו כאן את תשובתכם**

p\_(o)pptr

p\_(o)pptr

p\_(o)pptr

p\_osptr

p\_osptr

p\_ysptr

p\_ysptr

p\_ysptr

p\_osptr

p\_(o)pptr

p\_(o)pptr

p\_cptr

p\_cptr

1. (7 נק') תנו דוגמה לקריאת מערכת שנלמדה בתרגול, שבה משתמשים בקשרי המשפחה כדי לבצעה? הסבירו איך בא לידי ביטוי השימוש בקשרי המשפחה בה:  
     
   **תשובה**:  
     
   **בפונקציית המערכת exit() נעשה שימוש בקשרי המשפחה, על מנת לאתר בנים יתומים שעבורם תהליך האב לא קרא ל-wait(), ולהפוך אותם לבנים של התהליך init כדי שיוכל לשחררם. נתבונן בפיסת הקוד מתוך הפונקציה exit\_notify() שנקראת ע״י exit():**

**static void** exit\_notify(**void**)

{

...

**while** (current->p\_cptr != NULL) {  
 p = current->p\_cptr;  
 current->p\_cptr = p->p\_osptr;  
 p->p\_ysptr = NULL;  
 p->ptrace = 0;  
  
 p->p\_pptr = p->p\_opptr;  
 p->p\_osptr = p->p\_pptr->p\_cptr;  
 **if** (p->p\_osptr)  
 p->p\_osptr->p\_ysptr = p;  
 p->p\_pptr->p\_cptr = p;  
 **if** (p->state == TASK\_ZOMBIE)  
 do\_notify\_parent(p, p->exit\_signal);

...

}

**...**

**}**

**אנחנו רואים שהתהליך עובר על כל הבנים של התהליך שהם במצב zombie, ומחליט האם הם יתומים ויש להפוך אותם לבנים של init.**