Operating Systems – 02340123

**Homework Exercise 1 – Dry**

**Spring 2025**

Teaching assistant in charge:

**Omer Daube**

Assignment Subjects & Relevant Course material

**Processes and inter-process communications**

**Recitations 1-3 & Lectures 1-3**

# Submission Format

1. Only **typed** submissions in **PDF** format will be accepted. Scanned handwritten submissions will not be graded.
2. The dry part submission must contain a single PDF file named with your student IDs –

**DHW1\_123456789\_300200100.pdf**

1. The submission should contain 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.
2. Submission is done electronically via the course website, in the **HW1 – Dry** submission box.

# Grading

1. **All** question answers must be supplied with a **full explanation**. Most of the weight of your grade sits on your **explanation** and **evident effort**, and not on the absolute correctness of your answer.
2. Remember – your goal is to communicate. Full credit will be given only to correct solutions which are **clearly** described. Convoluted and obtuse descriptions will receive low marks.

# Questions & Answers

* The Q&A for the exercise will take place at a public forum Piazza **only**. Please **DO NOT** send questions to the private email addresses of the TAs.
* 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 .

# Late Days

* Please **DO NOT** send postponement requests to the TA responsible for this assignment. Only the **TA in charge** can authorize postponements. In case you need a postponement, please fill out the attached form:

<https://forms.office.com/r/54AAUcLgBR?origin=lprLink>

# Question 1 – Process management (40 points)

לפניכם קטע קוד:

1

2

3

4

5

6

7

8

9

10

11

12

#include <stdio.h>

#include <unistd.h>

int main() {

int pid = fork();

if (pid == 0) {

printf("2");

} else {

printf("1");

}

return 0;

}

בשאלה זו עליכם להניח כי לא נוצרים תהליכים נוספים במערכת, פרט לאלו שנוצרים ע"י התוכנית. בנוסף, הניחו שקריאות מערכת לא נכשלות.

סעיף 1

האם ניתן להוסיף שורת קוד אחת כך שתמיד יודפס 21 בהרצת התוכנית? אם כן – כתבו אותה (כולל מספרי השורות שביניהם להוסיפה) והסבירו, אם לא – הסבירו מדוע.

כן. נוסיף wait() בין שורה 8 ל9. נשים לב כי בקוד אנו משתמשים בfork(), משמע יוצרים תהליך בן לתהליך אב כשלהו. לאחר מכן אם זה תהליך אב הוא ידפיס 1 ואם זה תהליך בן (fork() יחזיר 0 לתהליך הבן) הוא ידפיס 2. כיוון שנרצה כי 2 יודפס קודם, אז נרצה להגיד לתהליך האב לחכות לפני ההדפסה שלו עד שתהליך הבן שלו יסיים, וזה בדיוק מה שפקודת wait() עושה (נשים לב כי הפעולה מחכה עד אשר אחד מתהליכי הבן יסיים אך לפי הקוד המוצג נראה כי ניצור רק תהליך בן אחד בmain שלנו).

סעיף 2

האם ניתן להוסיף שורת קוד אחת כך שתמיד יודפס 12 בהרצת התוכנית? אם כן – כתבו אותה (כולל מספרי השורות שביניהם להוסיפה) והסבירו, אם לא – הסבירו מדוע.

לא ניתן. בכדי להדפיס 12 בהרצת התוכנית נרצה להבטיח כי תהליך הבן יסיים (pid = 0) וידפיס 2 רק כאשר תהליך האב יסיים וידפיס 1. אין לנו פקודה קיימת שיכולה להחליט כי תהליך בן יחכה לסיום תהליך אביו. נשים לב כי פונקציית waitpid() מחכה רק עבור הבנים של תהליך מסוים ולא יכולה לחכות ספציפית לאב.

סעיף 3

האם ייתכן שתהליך האב ידפיס 2? הסבירו.

על מנת שתהליך האב ידפיס 2, הpid שלו צריך להיות שווה ל-0. דבר זה לא ייתכן כיוון שלאחר ביצוע fork, הפקודה חוזרת פעמיים, אחד לתהליך המקורי ומחזירה לאב ערך pid > 0, והשני לתהליך שהעתיקה, יקרא תהליך הבן אשר מקבל ערך pid=0 כך שנוכל להבדיל בין השניים.

סעיף 4

האם תיתכן ריצה בה יודפס מספר יחיד? הסבירו.

כן, תתכן ריצה בה יודפס מספר יחיד, נתן דוגמא לריצה זו:

אם fork() נכשלת אז היא מחזירה את הערך -1 לpid ולכן יופעל else לאחר מכן וזה ידפיס 1.

סעיף 5

סטודנט בקורס ניסה להריץ תוכנית זו במקביל מספר רב של פעמים ונתקל בתקלה. ציינו 2 מקרים שונים אפשריים שיכלו להתרחש, והסבירו מה התרחש בהם.

**רמז**: חפשו באינטרנט את ה-posix manual על קריאת המערכת fork.

שני המקרים אשר יכלו להתרחש:

1. יכול להיות מקרה בו נפעיל פונקציה זו מספר רב של פעמים, כאשר כל הפעלה מעתיקה למקום חדש בזיכרון את התהליך ממנו קראנו לfork, ואז יווצר מצב בו ייגמר לנו המקום בזיכרון ונקבל שגיאה בשם ENOMEM על כל הפעלה של fork ולא ניצור תהליכים נוספים.

2. קיים מספר מקסימלי של תהליכים שיכולים לרוץ בו זמנית, לכן אם נקרא לקריאת fork מספר רב של פעמים, שם ניצור כל פעם תהליך חדש ותהליכים אלו יתחילו לרוץ בו זמנית, אז נוכל להגיע ל RLIMIT\_NPROC – יאמר לנו את ההגבלה של מספר התהליכים שיכולים לרוץ בו זמנית, ואם נגיע להגבלה זו פונקציית fork תוציא שגיאה בשם EAGAIN כל פכם שתיקרא מחדש והפונקציונליות שלה תיכשל.

סעיף 6

בתרגול ראינו כיצד ממומשת קריאת המערכת getpid(), סטודנט בקורס התבקש להוסיף לקוד את קריאת המערכת getppid() וביקש את עזרתכם במימושה בקוד הגרעין של לינוקס. השלימו את החסר.

long **sys\_getppid**(void) {

parent->pid;

}

סעיף 7

סטודנט בקורס למד על פקודת bash הנקראת strace, ובה אפשר להריץ תוכנית לבחירת המשתמש ולראות את קריאות המערכת שנקראו בה. הסטודנט טען שאם יריץ את הפקודה strace יחד עם התוכנית הנתונה, הוא לא יראה את קריאת המערכת exit מאחר והוא לא כתב אותה. האם הסטודנט צודק? הסבירו.

הסטודנט לא צודק. קריאת המערכת exit מתבצעת בסוף התוכנית גם אם לא נקראת בתוכנית שאנו רואים כאן – קריאת מערכת זו תמיד נקראת לסיום סטנדרטי של התוכנית, על מנת לסיים את ביצוע התהליך ולכחרר את כל המשאבים שברשותו, לכן כאשר התוכנית תסתיים תיקרא קריאת מערכת זו ונדפיס אותה בעזרת strace, גם אם לא קראנו לה מפורשות. (גם אם קריאה זו נכתבת בפונקציית המעטפת של התוכנית שלנו, הביצוע שלה יהיה בסיום התוכנית שאנו רואים כאן ולכן strace ידפיסה).

# Question 2 – Signals (40 points)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

#include <stdio.h>

#include <signal.h>

#include <stdlib.h>

void handler(int signum)

{

printf("Got Signal! Number = %d\n", signum);

exit(0);

}

int main()

{

signal(SIGFPE, handler);

int x = 120;

for(int i = 6; i >= 0; --i) {

printf("%d\n", x / i);

}

printf("Magic Number!\n");

return 0;

}

סעיף 1

האם יודפס בסוף התוכנית “Magic Number!”?

לא, בשורה 13 אנו מעדכנים את ה-handler של הסיגנל SIGFPE לפונקציה handler() שמבצעת יציאה מהתוכנית. כמו כן בשורה 16 באיטרציה האחרונה על הלולאה נבצע חילוק ב-0. חלוקה ב-0 תביא לשגיאת חלוקה ב-CPU. המעבד יבצע Interrupt וה-kernel ישלח את הסיגנל הסינכרוני SIGFPE ולכן handler() יקרא והתוכנית תיעצר טרם סיום הלולאה.

סעיף 2

בסעיף זה אינכם יכולים להניח דבר על המערכת שבה רצה התוכנית, בפרט, עשוי להיות תהליך אחר שרץ במערכת. באיזה תרחיש ייתכן שהתוכנית תדפיס “Got Signal…” ללא הדפסות נוספות לפני כן?

ייתכן תהליך שרץ במקביל בbackground של המערכת, ואילו תהליך זה עשוי לעשות פעולה שתפעיל שליחת signal ממנה לתהליך שבו רצה התוכנית שלנו בזמן מאוד ספציפי - לפני שהתוכנית שלנו תספיק להגיע ללולאת הfor ואחרי קריאת המערכת signal(). הדבר עשוי להיעשות על ידי פעולת kill(pid\_our\_process, SIGFPE), דבר שיגרום לתוכנית שלנו להפעיל את הsignal של SIGFPE שיפעיל את הhandler שהגדרנו.

**בסעיפים הבאים נתון כי הוחלט למחוק את שורה 8 (exit(0)) מהקוד.**

סעיף 3

תארו מה יתרחש בריצת התוכנית.

הלולאה תתבצע באופן תקין עד האיטרציה האחרונה בה מתבצעת חלוקה ב-0. כעת יקרא ה-handler() ושיבצע את ההדפסה. לאחר ביצוע ה-Handler() התוכנית תחזור בחזרה לשורה 16 ותבצע אותה שוב. אך מכיוון שחזרה לריצה רגילה לאחר ביצוע SIGFPE היא חסרת משמעות ונחלק שוב ב-0 ה-handler() יקרא בשנית. התנהגות זו בלתי מוגדרת.

סעיף 4

נתון כי שינו את הגדרת לולאת ה-for (שורה 15) כך:

for(int i = 6; i > 0; --i) {

בסעיף זה לא ניתן להניח דבר על המערכת בה רצה התוכנית, בפרט, עשויים להיות תהליכים נוספים השולחים סיגנלים. בנוסף נתון שהתוכנית רצה פעם אחת בלבד, באופן רציף וללא החלפות הקשר, וכי בריצה זו היא הייתה במצב גרעין רק פעם אחת (לצורך שאלה זו התעלמו מהמעבר למצב גרעין לצורך קריאת המערכת signal). מהו המספר המירבי של ההדפסה “Got Signal…” שתודפס?

תחילה נראה כי שינוי הגדרת הfor יגרום לכך שנמנע מחילוק באפס ולכן נמנע מקריאת signal של SIGFPE שישלח אותנו לפונקציה שתדפיס את הנדרש – לא נבצע קריאת סיגנל ל"עצמנו". נשים לב גם כי העברת signals מתבצע בשני דרכים – רישום הסיגנל, וטיפולו. כיוון שעשויים להיות תהליכים נוספים השולחים סיגנלים, רישום הסיגנל יכול להתבצע לתהליך שלנו מספר אינסופי של פעמים (רישום על ידי תהליך אחר, לא עצמנו). טיפול הסיגנל יתבצע רק כאשר הגרעין חוזר ממצב הגרעין למצב המשתמש – שם מערכת ההפעלה בודקת האם יש סיגנלים מסומנים ומפעילה בהתאם. לכן, כיוון שריצה זו הייתה במצב הגרעין רק פעם אחת (ריצה רציפה ללא החלפות הקשר), לא משנה כמה פעמים נשלח את הסיגנל המתאים, המספר המירבי של ההדפסות הרצויות יהיה **אחד** -> רק פעם אחת נעבור ממצב גרעין למשתמש לכן רק פעם אחת נוכל לבדוק האם קריאת הסיגנל SIGFPE נקראה.

סעיף 5

נתון כי השורה הראשונה בפונקציה handler (שורה 7) הוחלפה ב-

printf("Got Signal! Number = %d\n", 100 / (8 – signum));

בהנחה שלא נשלחים סיגנלים חיצוניים, כמה פעמים תיקרא הפונקציה handler? בחר את התשובה הנכונה ונמק:

1. 0 פעמים.
2. פעם אחת.
3. פעמיים.
4. אינסוף פעמים.

פעם אחת. עקב כך שמספר הסיגנל SIGFPE הוא 8 נקבל שתבוצע ב-handler() חלוקה ב-0. באופן כללי במהלך הרצת handler של סיגנל כלשהו, הסיגנל נחסם עד סיום הריצה של ה-handler שלו. אך במקרה זה הסיגנל הוא סינכרוני ולא ניתן לדחות את הטיפול בו לחזרה מהפונקציה. לכן החלוקה ב-0 תגרום להפעלת ה-handler() הדיפולטי של SIGFPE שיעצור את התוכנית.

סעיף 6

בתרגול הוסבר שלא ניתן להתקין שגרת טיפול ל-SIGSTOP. סטודנט בקורס טען כי לדעתו יוצרי לינוקס בחרו זאת כי ב-shell חשוב מאוד ש-CTRL+Z (שזו הדרך של משתמש לעצור תוכנית שרצה בחזית ב-bash) תמיד יעבוד, אף אם מפתח התוכנית לא רוצה בכך. האם הסטודנט צודק? נמקו.

הסטודנט טועה כיוון שCTRL+Z שולח signal של SIGTSTP ולא SIGSTOP. ניתן לשנות ולהתקין שגרת טיפול ל SIGTSTP כך ש CTRL+Zלא תמיד יעבוד, כמו שמפתח התוכנית עלול לרצות, ולכן CTRL+Z לא קשורה לעובדה כי לא ניתן להתקין שגרת טיפול לSIGSTOP.

סעיף 7

סטודנט בקורס מערכות הפעלה למד על הגרעין של לינוקס בתרגולים, התלהב, והחליט לכתוב תוכנית ולבדוק אותה. נתון שבזמן שהתוכנית הייתה במצב גרעין, הסטודנט רצה לסיים את ריצתה, ואז לחץ קודם על CTRL+Z ולאחר מכן על CTRL+C. מה יתרחש כאשר התוכנית תחזור למצב משתמש?

מכיוון שהתוכנית במצב גרעין ידלקו שני הביטים המתאימים במערך הביטים המייצג את ה-pending signals. מכיוון ש-CTRL+C מדליק את סיגנל SIGINT=2 ו-CTRL+Z מדליק את סיגנל SIGTSTP=20 (על x86) נקבל כי ביציאה ממצב גרעין יבוצע קודם ה-handler של SIGINT. בהנחה שלא בוצע שינוי בשגרת הטיפול נקבל כי התוכנית תעצור את ריצתה ולא תבוצע השגרה של SIGTSTP.

# Question 3 – Inter-Process Communication (20 points)

סעיף 1

האם האירועים הבאים יגרמו *בהכרח*לתוצאה הנתונה? עבור כל אירוע, אם הוא בהכרח גורם לתוצאה הסבירו מדוע, ואם הוא אינו בהכרח גורם לתוצאה הביאו דוגמה נגדית.

|  |  |  |  |
| --- | --- | --- | --- |
| אירוע | תוצאה | כן / לא | נימוק |
| תהליך A כותב לpipe קובץ בגודל 100KB | הקובץ לא ייכתב ל-pipe | נכון | בPipe יש מקום מוגבל בגודל כ-KB64, לכן כשננסה לכתוב קובץ בגודל KB100, יכתבו רק הKB64 הראשונים, ואילו שאר הקובץ ימתין להיכתב, ולא נקבל כי כל הקובץ ייכתב. |
| תהליך A קורא לexit() | סיום ריצת תהליך A | נכון | קריאת מערכת זו תפקידה לסיים את התהליך הקורא. |
| תהליך בעל מספר PID מקסימלי קורא ל-fork() | כישלון fork() | לא נכון | דוגמא נגדית - Pid מספר 23 פנוי ולא מקושר לתהליך מסוים, לכן ברגע שתהליך בעל מספר pid\_max עושה fork(), יש לפחות pid אחד אפשרי שיחזור – 23 לדוגמא, וזה יהיה הpid של הבן שהfork() יצר. |

סעיף 2

האם האירועים הבאים יגרמו *בהכרח*למעבר מיידי ממצב משתמש בתהליך A שרץ למצב גרעין? נמקו.

(בתשובתכם התעלמו מהחלפות הקשר בין תהליכים / פסיקות)

|  |  |  |
| --- | --- | --- |
| אירוע | כן / לא | נימוק |
| תהליך A כותב לpipe (מחובר לתהליך B) | כן | התהליך יעבור מיידית למצב גרעין שכן הפקודה write אשר כותבת ל-pipe היא syscall. |
| תהליך A קורא לdup2() | כן |  |
| תהליך B קורא מהpipe (הוא רץ על ליבה שונה מזו של A) | כן |  |

סעיף 3

סטודנט חרוץ בקורס קרא את ה-man של לינוקס על קריאות המערכת vmsplice() ו-splice():

**splice**() moves data between two file descriptors without copying between kernel address space and user address space.

**vmsplice**() splice user pages to/from a pipe.

הסבירו את היתרון של שימוש ב-vmsplice יחד עם pipe על פני אופן השימוש הרגיל ב-pipe שנלמד בתרגול.

היתרון בשימוש פקודה זו לעומת השימוש הרגיל ב-pipe הוא החיסכון בהעתקה של המידע. בשימוש הרגיל, כאשר תהליך כותב ל-pipe, המידע מועבר לזיכרון הנמצא ב-kernel space, ולאחר מכן תהליך אחר שקורא מה-pipe שואב את המידע מאותו אזור – מה שעשוי לכלול העתקות מיותרות בין זיכרון המשתמש לזיכרון הליבה. לעומת זאת, vmsplice, המבוססת על splice, מאפשרת לקשר ישירות בין דפי זיכרון של תהליך המשתמש לבין קצוות ה-pipe, ובכך לחסוך את הצורך בהעתקה, לשפר את הביצועים ולצמצם שימוש בזיכרון – במיוחד כאשר נעבוד עם כמויות גדולות של נתונים.

סעיף 4

סטודנטית בקורס מתבקשת לכתוב תוכנית אשר קוראת תחילה לפונקציה חיצונית (בלי שהיא יכולה לדעת מה נכתב בה), ולאחר סיום ריצת הפונקציה החיצונית, התבקשה הסטודנטית לכתוב קוד נוסף שמקבל קלט מהמשתמש ומבצע חישובים נוספים.

כלומר,

int main(){

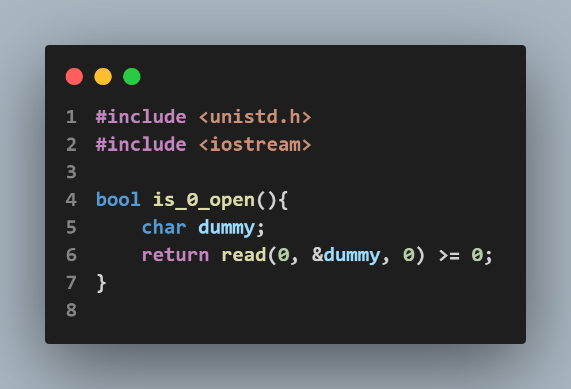
    external\_function(); // Can't know what this function does

    /\* Get input and perform calculations \*/

    return 0;

}

הסטודנטית חושדת כי במהלך ריצת הפונקציה החיצונית, בוצע close() לערוץ הקלט הסטנדרטיstdin) , כלומר file descriptor מספר 0), מבלי לקשר אותו מחדש ל-file object אחר (כלומר,file descriptor מספר 0 לא מקושר לאף אובייקט עד תחילת ריצת קוד הסטודנטית).

כתבו קוד עבור הסטודנטית על מנת לבדוק אם stdin אכן נסגר במהלך ריצתה של הפונקציה החיצונית.

סעיף 5

סטודנט בקורס כתב תוכנית שמבצעת fork ומשתמשת ב-fifo על מנת לתקשר בין תהליך האב לתהליך הבן במקום ב-pipe. כתבו חיסרון אחד ויתרון אחד לגישה זו.

חסרון בגישה זו – fifo הינו pipe "ציבורי", כלומר כל התהליכים במכונה יכולים לגשת אליו, ולכן אם נרצה לתקשר בין תהליך האב לתהליך הבן בצורה מסודרת, לא נוכל להבטיח כי תהליך צד שלישי לא ייקח חלק בצינור זה ויקרא\ יכניס ערכים שיראה לנכון. לכן התקשורת בין התהליכים יכולה להיפגע.

יתרון בגישה זו – fifo הינו ערוץ תקשורת דו כיווני בעל FD יחיד (אשר ניתן לבצע קריאה וכתיבה דרך אותו FD) לכן קריאה וכתיבה לקובץ תוכל להתבצע בצורה קלה יותר לשני הכיוונים עבור תהליך האב והבן, בלי צורך לסגור קצוות מסוימים של הpipe ולפתוח אחרים אם נרצה לשנות כל פעם את כיוון התקשורת בין האב לבן.