Operating Systems – 234123

**Homework Exercise 1 – Dry**

**Omer Stoler – 318471356 – stoler.omer@campus.technion.ac.il**

**Adi Arbel – 207919614 – adi.arbel@campus.technion.ac.il**

Teaching Assistant in charge:

**Yehonatan Buchnik**

Assignment Subjects & Relevant Course material

**Processes, System Calls, Calling Conventions**

**Recitations 1-2 & Lecture 1**

# 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://goo.gl/forms/R7n5YjsqO8XvR8m03>

# Part 1

חברת MaKore, הכורה (mining) מטבעות דיגיטליים, מריצה בכל רגע מספר גדול של תהליכים על-מנת להאיץ את פעולת הכרייה. התהליכים שומרים את תוצאות החישובים שלהם לקבצים בדיסק כדי למנוע אובדן מידע במקרה שהתהליך קורס לפתע. בכל שניה התהליך קורא לפונקציה הבאה כדי ליצור את שם הקובץ שבו ייכתב הפלט שלו:

|  |
| --- |
| #include <string>  using namespace std;  string create\_file\_name(time\_t timestamp) {  pid\_t pid = getpid();  string s = “results-” + to\_string(pid) + to\_string(timestamp);  return s;  } |

כפי שניתן לראות, כל תהליך מוסיף את ה-PID שלו לשם הקובץ כדי למנוע התנגשות בין קבצים של תהליכים שונים. לצורך הפשטות, לאורך כל השאלה הניחו כי החברה אינה משתמשת בחוטים כלל.

1. היכן הגרעין שומר את ה-PID של התהליך?  
   1. בספריה libc.
   2. במחסנית המשתמש.
   3. במחסנית הגרעין.
   4. בערימה.
   5. במתאר התהליך (ה-PCB).
   6. בתור הריצה (runqueue).

נימוק:

בהרצאה ובתרגול ראינו כי אחד השדות של הPCB הוא הPID . הPCB ומחסנית הגרעין נשמרים במקטע זיכרון ייעודי, כאשר הPCB מתחת למחסנית הגרעין ובו כל השדות הרלוונטיים לתהליך כולל הPID.

שרה, בוגרת הקורס ומהנדסת צעירה בחברה, הבחינה כי הפונקציה הנ"ל נקראת פעמים רבות במהלך הריצה של כל תהליך. לכן שרה הציעה את השיפור הבא לקוד המקורי:

|  |
| --- |
| pid\_t pid = getpid();  string create\_file\_name(time\_t timestamp) {  string s = “results-” + to\_string(pid) + to\_string(timestamp);  return s;  } |

1. מדוע הפתרון של שרה עדיף על המימוש המקורי?

בעבור תהליכים המריצים את הקוד שלעיל מראשיתו (כלומר לא תהליכים שבוצע fork לתהליך עם הקוד הקיים) האתחול של המשתנה הגלובלי pid יניב חסכון בקריאות מערכת, כיוון שמזהה התהליך נותר קבוע לאורך ריצתו ואין צורך בקריאות מיותרות. כל תהליך שמקורו בfork ומריץ את קוד זה אך לא מראשיתו, יבצע ביצירתו העתקה של מרחב הזכרון הכולל של תהליך האב, ובתוכו העתק של המשתנה הגלובלי של האב – ללא קריאה נוספת לgetpid() שתעדכן את ערכו לערך הpid האמיתי עבור הבן. עבור תהליכים כאלו לא נקבל שיפור כי נכונות הקוד תיפגע.

דנה, מהנדסת בכירה בחברה, התלהבה מהרעיון של שרה והחליטה לקחת אותו צעד אחד קדימה. דנה עדכנה את פונקצית המעטפת (wrapper function) של קריאת המערכת ()getpid כפי שמופיעה בספרית libc באופן הבא:

|  |
| --- |
| 1. + pid\_t cached\_pid = -1; // global variable 2. pid\_t getpid() { 3. unsigned int res; 4. + if (cached\_pid != -1) { 5. + return cached\_pid; 6. + } 7. \_\_asm\_\_ volatile( 8. "int 0x80;" 9. :"=a"(res) :"a"(\_\_NR\_getpid) :"memory" 10. ); 11. + cached\_pid = res; 12. return res; 13. } |

* שורות מסומנות ב-"+" הן שורות שדנה הוסיפה לקוד המקורי. אלו השורות היחידות שהשתנו בספריה.
* תזכורת: שורת האסמבלי שומרת את הערך "NR\_getpid\_\_" ברגיסטר eax לפני ביצוע הפקודה, ומציבה את ערך eax לאחר ביצוע הפקודה במשתנה res.שרה השתמשה בספריה החדשה (של דנה), אך תוכניות מסוימות שעבדו לפני השינוי הפסיקו לעבוד כנדרש עם הספריה החדשה.

1. מהי התקלה שנוצרה בעקבות השינוי?  
   1. ()fork עלולה לחזור עם אותו ערך בתהליך האב ובתהליך הבן.
   2. ()fork עלולה להיכשל במקרים בהם המימוש המקורי היה מצליח.
   3. ()getpid עלולה להחזיר pid של תהליך אחר.
   4. ()getpid עלולה להחזיר "1-".
   5. ()execv עלולה להיכשל במקרים בהם המימוש המקורי היה מצליח.

נימוק:

באופן דומה כל תהליך שמקורו בfork ומריץ את קוד זה אך לא מראשיתו, כאשר בוצעה בתהליך האב מתישהו קריאה לgetpid , העתקה של מרחב הזכרון הכולל של תהליך האב תביא העתק של המשתנה הגלובלי של האב וכך הערך בcached יהיה לא תקין, אך יהיה שונה מהערך 1- ולפי הקוד שלעיל הוא שיוחזר.

כדי לתקן את התקלה שנוצרה, דנה מציעה **בנוסף** את התיקון הבא של פונקציית המעטפת של fork:

|  |
| --- |
| 1. + // the same global variable from above 2. + extern pid\_t cached\_pid; 3. pid\_t fork() { 4. unsigned int res; 5. \_\_asm\_\_ volatile( 6. "int 0x80;" 7. :"=a"(res) :"a"(\_\_NR\_fork) :"memory" 8. ); 9. **+ ???** 10. return res; 11. } |

1. השלימו את התיקון הנדרש בשורה 10:

|  |  |
| --- | --- |
| .a | if (res == 0) cached\_pid = -1; |
| .b | if (res == 0) cached\_pid = getpid(); |
| .c | if (res == 0) return cached\_pid; |
| .d | if (res > 0) cached\_pid = -1; |
| .e | if (res > 0) cached\_pid = getpid(); |
| .f | if (res > 0) return cached\_pid; |

נימוק:

הבעיה בקריאת הfork המקורית היא שבפיצול התהליכים אין עדכון של ערך המטמון אצל הבן. עלינו לאתחל אותו או לערך 1- או לערך PID של הבן. נניח כי נרצה לעדכן את ערך המטמון לPID של הבן מיידית, לאור העדכון של הפונקציה getpid לא נוכל לעשות זאת כיוון שקריאה אליה בתהליך הבן, כשערך המטמון עוד מאותחל לערך האב, תחזיר מטעמי המימוש שלה את ערך המטמון ישירות – כלומר חזרנו למשבצת הראשונה. לכן, עלינו לאתחל את המשתנה הגלובלי כפי שהיינו עושים אילו התהליך היה מורץ מראש התוכנית, כלומר ל1- כיוון שהמימוש מסתמך על עובדה זו בעדכון ערך המטמון.

# Part 2

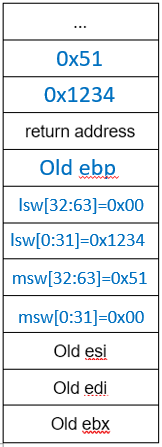
נתון הקוד של התוכנית C הבאה.

הבהרה: uint32\_t מייצג טיפוס שלם אי-שלילי של 32-ביט. באופן דומה uint64\_t מייצג טיפוס שלם אי-שלילי של 64-ביט.

הנחה: המערכת היא **לינוקס 32-ביט** כפי שנלמדה בתרגולים.

|  |  |
| --- | --- |
| 1 | uint64\_t extend(uint32\_t low, uint32\_t high) { |
| 2 | uint64\_t lsw = (uint64\_t)low; // convert to 64 bit |
| 3 | uint64\_t msw = (uint64\_t)high << 32; |
| 4 | return msw | lsw; |
| 5 | } |
| 6 |  |
| 7 | int main(int argc, char\* argv[]) { |
| 8 | // ... |
| 9 | uint64\_t l = extend(0x1234, 0x51); |
| 10 | // ... |
| 11 | } |

א.השלימו את תמונת המחסנית בהרצת התוכנית כאשר היא נמצאת בשורה 4 לעיל. הסבירו את תשובתכם



|  |  |
| --- | --- |
| לפני הקריאה ב-main | ... |
| ארגומנטים בסדר הפוך במחסנית ע"מ לתמוך בפעולת פונקציות  בעלות מספר ארגומנטים לא ידוע מראש (כמו printf). |  |
|  |  |
| כתובת החזרה  %ebp – פוינטר לתחילת ה-frame, כדי לדעת לאן לחזור  בסגירת ה-frame. כמו כן, ממנו ב-offset ידוע נמצאים הארגומנטים של ה-frame. | return address |
|  |  |
| ה-local vars של התוכנית מוקצים במחסנית לפי סדר הגדרתם.  ארכיטקטורת IA32 הינה מסוג little endian ועל כן, בשמירת מידע ביותר מ-32 בתים (אורך מילה) |  |
|  |  |
|  |  |
|  |  |
|  | Old esi |
|  | Old edi |
|  | Old ebx |

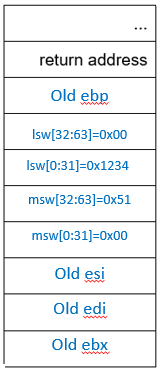
ב. באיזה רגיסטר או רגיסטרים תוחזר תוצאת החישוב ומה יהיו הערכים ברגיסטרים אלו? הסבר.

לפי הקונבנציה שלמדנו בתרגול , ערך בגודל של עד 64 ביט מוחזר כצמד edx:eax. לכן בשני רגיסטרים אלו (eax ו-edx) תוחזר תוצאת החישוב, וזאת כשלפי הסדר – eax מקבל את 4 הבתים התחתונים של ערך ההחזרה 0x1234, ו-edx מקבל את 4 הבתים העליונים של ערך ההחזרה 0x51. (זאת על פי המוצג בשקף 25 בתרגול 1)

ג. בהנחה ש-main עושה שימוש בכל הרגיסטרים לפני הקריאה ל-extend ולאחר הקריאה ל-extend אילו רגיסטרים חייבים להישמר על-ידי main טרם הקריאה ל-extend? הסבר.

לפי המוצג בשקף 17, באחריות הפונקציה הקוראת לשמור לעצמה את eax ,edx במידה וברצונה לשמר את ערכיהם לאחר הקריאה לפונקציה הנקראת, כיוון שערך ההחזרה יעבור בהם, לכן הקוראת שתבצע בהם שינויים לא דואגת לשימורם. על כן אלו 2 הרגיסטרים שעליה לשמור.

ד. קונבנציית קריאה בסגנון FASTCALL אומרת כי עד 3 פרמטרים מועברים לפונקציה **ברגיסטרים** לפי הסדר הבא: eax, edx, ecx (משמאל לימין).  
 באופן דומה לסעיף א, השלימו את מצב המחסנית בשורה 4 בתוכנית לעיל, כאשר קונבנצית הקריאה לפונקציה מסעיף א מוגדרת להיות בסגנון FASTCALL.



|  |
| --- |
| ... |
| return address |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

ה.הצג מימוש קצר ככל האפשר ב-assembly לפונקציה extend מסעיף ד. הסבר את המימוש:

ret

הרגיסטרים בהם מועברים הפרמטרים הם בדיוק הפרמטרים בהם הם המוחזרים. הפרמטר הרשון הוא החלק התחתון של 32 הביט התחתונים, ועל כן יש פשוט לקחת אותו ולהחזירו ברגיסטר המכיל את ערך ההחזרה הנמוך – בשני המקרים זהו eax. באופן דומה הפרמטר השני המועבר בedx מוחזר ברגיסטר המכיל את הערך המכיל את 32 הביט העליונים של ערך ההחזרה שזהו גם edx ועל כן על הפונקציה לא לבצע דבר במקרה הנתון.