TECHNION
Israel Institute of Technology

Computer Science Department

Operating Systems – 234123

Homework Exercise 1 – Dry

Name: Nadav Orzech

ID: 311549455

Email: nadav.or@campus.technion.ac.il

Name: Roni Englender

ID: 312168354

Email: roni.en@campus.technion.ac.il

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?

   a. בספריה libc.
   b. במחסנית המשתמש.
   c. במחסנית הגרעין.
   d. בערימה.
   e. במתאר התהליך (ה-PCB).
   f. בתור הריצה (runqueue).

נימוק: מתאר התהליך הוא מבנה מסוג task_struct המכיל את מזהה התהליך (pid).

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;
}
```

2. מדוע הפתרון של שרה עדיף על המימוש המקורי?

בפתרון של שירה לא נעשה מעברים מרובים לkernel mode כמו במימוש המקורי, שכל קריאה לפונקציה בודקת מה הpid. כאן הבדיקה נעשית מחוץ לפונקציה, והיא נעשית רק פעם אחת.

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:

```
+   pid_t cached_pid = -1; // global variable

     pid_t getpid() {
         unsigned int res;
+        if (cached_pid != -1) {
+            return cached_pid;
+        }
         __asm__ volatile(
             "int 0x80;"
                 :"=a"(res) :"a"(__NR_getpid) :"memory"
            );
+        cached_pid = res;
            return res;
        }
```

- 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.

3. מהי התקלה שנוצרה בעקבות השינוי?

a. fork() עלולה לחזור עם אותו ערך בתהליך האב ובתהליך הבן.
b. fork() עלולה להיכשל במקרים בהם המימוש המקורי היה מצליח.
c. getpid() עלולה להחזיר pid של תהליך אחר.
d. getpid() עלולה להחזיר "-1".
e. execv() עלולה להיכשל במקרים בהם המימוש המקורי היה מצליח.

נימוק: התקלה הנ"ל עלולה להיווצר כאשר מתבצעת קריאה ל()fork. שיוצרים את תהליך הבן כל המשתנים מועתקים לתהליך הבן כולל המשתנה catched_pid ולכן שנבצע את הקטע הנ"ל catched_pid לא יהיה -1 אלא הערך האחרון שהיה לתכנית האב, ולכן יוחזר ערך הPid של האב ולא של הבן כפי שרצינו.

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.
4.  pid_t fork() {
5.  unsigned int res;
6.  __asm__ volatile(
7.  "int 0x80;"
8.  :"=a"(res) :"a"(__NR_fork) :"memory"
9.  );
10. + ???
11. return res;
12. }
```

4. 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:
אם res==0 משמע אנחנו בתהליך הבן, ונאפס מחדש ל -1 את cached_pid, כך שבפעם הבאה שנקרא לget_pid התנאים יהיו "מאופסים" ובדיקת הif תתבצע ככוונת המשורר, ונמשיך האלה להחזרת ערך הpid של הבן.

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.

```
uint64_t extend(uint32_t low, uint32_t high) {
uint64_t lsw = (uint64_t)low; // convert to 64 bit
uint64_t msw = (uint64_t)high << 32;
return msw | lsw;
}

int main(int argc, char* argv[]) {
// ...
uint64_t l = extend(0x1234, 0x51);
// ...
}
```

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

לפני הקריאה ב-main	...
_____ארגומנט ראשון_____	0x51
_____ארגומנט שני_____	0x1234
כתובת החזרה	return address
_רגיסטר ebp המצביע על בסיס המסגרת של ה_caller	*Old ebp*
___ה32 ביטים העליונים של המשתנה המקומי הראשון___	0x00
___ ה32 ביטים התחתונים של המשתנה המקומי הראשון__	0x1234
__ ה32 ביטים העליונים של המשתנה המקומי השני___	0x51
___ ה32 ביטים התחתונים של המשתנה המקומי השני__	0x00
_____	Old esi
_____	Old edi
_____	Old edx

ב. באיזה רגיסטר או רגיסטרים תוחזר תוצאת החישוב ומה יהיו הערכים ברגיסטרים אלו? הסבר.
התשובה תוחזר ברגיסטרים edx,eax. בוצע shift-left של 32 ביט לhigh כך שכעת הוא בmsb של msw ובוצע המרה ל64 ביט לlow ומכיוון שהוא אי שלילי הוא יופיע בlsb של lsw כאשר שאר הביטים הם 0. רגיסטר edx=0x51 ורגיסטר eax=0x1234.

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 קונבצניית הקוד רגיסטרים edx,eax,ecx צריכים להישמר ע"י הMain טרם הקריאה לextend, כיוון שהם באחריות הפונקציה הקוראת – Main.

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
Old ebp
0x00
0x1234
0x51
0x00
Old esi
Old edi
Old edx

ה.הצג מימוש קצר ככל האפשר ב-assembly לפונקציה extend מסעיף ד. הסבר את המימוש:
נבצע שימוש בret בלבד, כיוון שהערכים מועברים ברגיסטרים eax edx , כאשר הערך העליון מועבר בedx והתחתון בeax, והערכים הללו גם מוחזרים ברגיסטרים הנ"ל ולכן אין צורך לשנות כלום.