ארגון ותכנות המחשב

תרגיל 2 - חלק רטוב

המתרגל האחראי על התרגיל: בועז מואב.

שאלותיכם במייל בעניינים מנהלתיים בלבד, יופנו רק אליו.

כתבו בתיבת subject: רטוב 2 אתמ.

שאלות בעל-פה ייענו על ידי כל מתרגל.

הוראות הגשה (לקרוא!!!):

- ההגשה בזוגות.
- שאלות הנוגעות לתרגיל יש לשאול דרך הפיאצה בלבד.
- על כל יום איחור או חלק ממנו, שאינו בתיאום עם המתרגל האחראי על התרגיל, יורדו 5 נקודות.
  - ניתן להגיש לכל היותר באיחור של 3 ימים (כאשר שישי ושבת נחשבים יחד כיום אחד בספירה).
  - הגשות באיחור יש לשלוח למייל של אחראי התרגיל בצירוף פרטים מלאים של המגישים (שם+ת.ז).
- הוראות הגשה נוספות מופיעות בסוף בתרגיל.
- לתרגיל שני חלקים. אין קשר בין חלק א' לחלק ב'!

נושא התרגיל: תכנות אסמבלי, קונבנציית קריאות, פסיקות תוכנה ו-IDT.

חומר דרוש: לחלק א' נדרשים הרצאות ותרגולים 1-4. לחלק ב' נדרשים גם תרגול 6 וההרצאה על קידוד פקודות.

חלק א – שגרות, קונבנציות ומה שביניהן

מבוא והקדמה מתמטית

בחלק א' של תרגיל בית זה נממש כפל מטריצות מעל השדה $\mathbb{Z}_p$ (p ראשוני), ונתחיל מרקע מתמטי בסיסי, למי שלא זוכר אלגברה א', או למי שלקח אלגברה 1מ. אל חשש, התרגיל אינו דורש מתמטיקה ברמה גבוהה כלל. בחלק זה תממשו שתי פונקציות בקובץ `matrix.asm` שקיבלתם. תוכלו להוסיף בקובץ עוד פונקציות עזר כאוות נפשכם.

השדה $\mathbb{Z}_p$

כל מה שאתם צריכים לדעת בשביל תרגיל בית זה על השדה[1] $\mathbb{Z}_p$ הן התכונות הבאות:

1. איברי השדה הם כל השלמים $\{0,1,2,\dots,p-1\}$, כאשר p הוא ראשוני כלשהו.
2. חיבור וכפל של איברים בשדה מתבצעים מודולו p. מה זה אומר? חיבור וכפל מתרחשים כמו בשלמים, אך כל המספרים מוחלפים בשארית החלוקה ב-p. חשבון זה נקרא חשבון מודולרי[2]. נסמן תוצאה של חשבון מודולרי באמצעות $\equiv_p$ (כדי להבדיל מתוצאת שוויון "רגילה" בשלמים).

נדגים: נניח שאנו עובדים עם $\mathbb{Z}_5$. איברי השדה הם $\{0,1,2,3,4\}$. נביט בחישובים הבאים:

- $0\cdot 1 = 0\cdot 2 = 0\cdot 3 = 0\cdot 4 \equiv_5 0$
- $1\cdot 1 = 1,\quad 1\cdot 2 = 2,\quad 1\cdot 3 = 3,\quad 1\cdot 4 \equiv_5 4$
- $2\cdot 2 = 4 \equiv_5 4$. למה? כי 2 כפול 2 זה 4 ושארית החלוקה של 4 ב-5 היא 4.
- $2\cdot 3 = 6 \equiv_5 1$. למה? כי 2 כפול 3 בשלמים זה 6. שארית החלוקה של 6 ב-5 היא 1 ולכן נקבל 1.
- $3+4 = 7 \equiv_5 2$. למה? כי 3+4 בשלמים זה 7. שארית החלוקה של 7 ב-5 היא 2.
- $4+1 = 5 \equiv_5 0$. למה? כי 4+1 בשלמים זה 5 ושארית החלוקה של 5 ב-5 היא 0.
- $4\cdot 4 = 16 \equiv_5 1$. למה? כי 4 כפול 4 בשלמים זה 16 ושארית החלוקה של 16 ב-5 היא 1.

תזכורת: כפל מטריצות

בתרגיל הזה יהיו שתי מטריצות: המטריצה A מגודל $m\times n$ והמטריצה B מגודל $n\times r$. איברי המטריצות יהיו איברים של השדה $\mathbb{Z}_p$ ואנו נרצה לחשב את המכפלה $A\cdot B$, לפי ההגדרה לכפל מטריצות שלמדתם באלגברה.

נזכיר את הנוסחה לכפל מטריצות[3]:

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

כאשר $(AB)_{ij}$ הוא האיבר בשורה ה-i והעמודה ה-j במטריצה AB (תוצאת המכפלה), האיבר a_{ik} הוא האיבר בשורה ה-i והעמודה ה-k במטריצה A והאיבר b_{kj} הוא האיבר בשורה ה-k והעמודה ה-j במטריצה B.

שימו לב שההבדל היחיד (למי שלמד כפל מטריצות מעל הממשיים בלבד) הוא שהחיבור והכפל של איברים ב-$\mathbb{Z}_p$ (שנדרשים כאן כחלק מהחישובים של כפל המטריצות) נעשה מודולו p, כפי שהוסבר קודם.

נדגים שוב עם השדה $\mathbb{Z}_5$:

$$\begin{pmatrix}1&2&3&1\\4&0&1&1\\2&3&4&1\end{pmatrix}\cdot\begin{pmatrix}1&1\\0&1\\0&0\\1&1\end{pmatrix}=\begin{pmatrix}2&4\\5&5\\3&6\end{pmatrix}\equiv_p\begin{pmatrix}2&4\\0&0\\3&1\end{pmatrix}$$

והתוצאה $\begin{pmatrix}2&4\\0&0\\3&1\end{pmatrix}$ היא התוצאה שאנו רוצים לקבל בדוגמה זו (ולא "תוצאת הביניים" בשלמים, שהיא רק חישוב עזר).

---

[1] שדה בויקיפדיה
[2] חשבון מודולרי בויקיפדיה
[3] כפל מטריצות בויקיפדיה

שלב ראשון – גישה לאיבר במטריצה

תחילה, **תממשו באסמבלי** את הפונקציה `get_elemnt_from_matrix`. חתימת הפונקציה היא:

```
int get_elemnt_from_matrix(int* matrix, int n, int row, int col)
```

כאשר הפרמטר `matrix` הוא מצביע לתחילתו של מערך דו ממדי (המטריצה), מגודל $m \times n$ (שימו לב שהפרמטר `n` בפונקציה מייצג את מספר העמודות במטריצה) והפרמטרים `row` ו-`col` הם השורה והעמודה (ב-`zero base`!!![4]) אותם אנו רוצים לקרוא.

ניתן להניח כי $n > 0$, $col, row \geq 0$ ובנוסף כי $row < m$ וכי $col < n$, וגם המצביעי `matrix` אינו `NULL` (ובאופן כללי, חוקי ומכיל רצף זיכרון בגודל $m \cdot n$ תאים חוקיים) ואין צורך לבדוק זאת.

דוגמת הרצה:

```
int matrix[3][4] = {{1,2,3,1},{4,0,1,1},{2,3,4,1}};
int x = get_elemnt_from_matrix((int*)matrix,4,1,1);
printf("%d", x);
```

ידפיס "0" למסך, כי האיבר במקום `[1][1]` (שוב, שימו לב ל-`zero base`) הוא 0.

שלב שני – ביצוע כפל המטריצות

בשלב השני והאחרון בתרגיל, **תממשו באסמבלי** את הפונקציה `multiplyMatrices`. חתימת הפונקציה היא:

```
void multiplyMatrices(int* first, int* second, int* result, int m, int n, int r,
unsigned int p);
```

כאשר הפונקציה מכפילה את המטריצה `first`, מגודל $m \times n$, במטריצה `second`, מגודל $n \times r$, ואת התוצאה שמה במטריצה `result` מגודל $m \times r$, כאשר כל הפעולות מתבצעות מעל השדה $\mathbb{Z}_p$ (חשבון מודולרי שהוזכר במבוא המתמטי), כאשר p הוא הפרמטר האחרון של הפונקציה.

ניתן להניח כי $m, n, r > 0$, כי $p > 1$ וראשוני, וכי המצביעים `first`, `second`, `result` אינם `NULL` (ובאופן כללי, חוקיים ומכילים כמות זיכרון בגדלים המצוינים בפסקה הקודמת).

הערות:

1. תסופק לכם הפונקציה (כלומר, אינכם צריכים לממש אותה בעצמכם) עם החתימה הבאה:

```
void set_elemnt_in_matrix(int* matrix, int num_of_columns, int row, int
col, int value)
```

שמקבלת מצביע לתחילת מטריצה `matrix`, את מספר העמודות של המטריצה `num_of_columns`, תא במטריצה שנמצא בשורה `row` ועמודה `col` (ב-`zero base`), ואת המספר `value` ושמה את המספר בתא המבוקש.

פונקציה זו שקולה לפעולה בשפת `c` שתבצע:

```
matrix[row][col] = value;
```

דוגמת הרצה:

```
int matrix[3][4] = {{1,2,3,1},{4,0,1,1},{2,3,4,1}};
int x = get_elemnt_from_matrix((int*)matrix,4,2,3);
printf("%d, ", x);
set_elemnt_in_matrix((int*)matrix,4,2,3,3);
x = get_elemnt_from_matrix((int*)matrix,4,2,3);
printf("%d", x);
```

ידפיס "`1, 3`" למסך, כי בתא [3][2] במטריצה היה בהתחלה `1` ולאחר השימוש ב-`set_element_in_matrix` הוא השתנה ל-`3`.

הפונקציה תמומש בשפת `c` (נתונה דוגמה למימוש בקובץ ה-`c` שקיבלתם), לכן תשמור על קונבנציית System V שלמדנו בתרגול, ותצפה ממכם לקרוא לה בהתאם לקונבנציות אלה.

2. מומלץ לחלק את מימוש הפונקציה `multiplyMatrices` לפונקציות קטנות יותר, אך זו לא דרישה ובסופו של דבר תיבדק רק הפונקציה `multiplyMatrices`.

---

[4] Zero Base בויקיפדיה

שלב שלישי – טסטים

קיבלתם שני קבצים:

- `main.c`
- `matrix.asm`

בסופו של דבר תגישו אך ורק את הקובץ `matrix.asm`.

הקובץ `main.c` הוא קובץ הבדיקה, המכיל מימוש לדוגמה של הפונקציה שסיפקנו עבורכם, כמתואר בחלק השני (`set_element_in_matrix`), ופונקציית עזר נוספת בשם `print_result_matrix`, שתדפיס עבורכם את המטריצה כאשר תבדקו את הקוד שלכם (אין צורך לעשות בה שימוש בקובץ `matrix.asm`. היא משמשת לבדיקות בקובץ `main.c`).

את הקובץ `main.c` אתם לא מגישים, והוא ישמש אותנו (וגם אתכם) לטסטים.

שימו לב שסופק לכם בקובץ `main.c` טסט עם הרצה אחת לדוגמה של שתי הפונקציות שעליכם לממש בתרגיל זה.

על מנת לבדוק את התרגיל, צריך לבצע את 2 הפקודות הבאות:

```
as matrix.asm -o matrix.o
gcc main.c matrix.o -o main.out
```

אם השלמתם בהצלחה את המימוש של שתי הפונקציות (שלב ראשון + שלב שני), הפלט הצפוי עבור הטסט שסופק, יהיה:

```
student@ubuntu18:~/Spring21/HW2$ as matrix.asm -o matrix.o
student@ubuntu18:~/Spring21/HW2$ gcc main.c matrix.o -o main.out
student@ubuntu18:~/Spring21/HW2$ ./main.out
Element (1,1) in matrix1 is: 0
2, 4
0, 0
3, 1
```

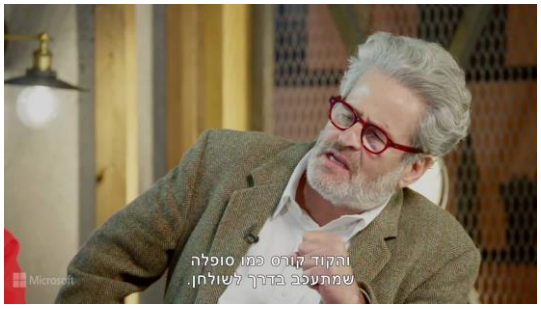
כמובן שלפני ההגשה, יש לבדוק את הקוד שלכם במקרים נוספים, מלבד המקרה שסופק לכם בקובץ `main.c`.

הערות

רגע לפני הסוף, אנא קראו בעיון:

1. אנא ודאו שהתוכנית שלכם יוצאת (מסתיימת) באופן תקין, דרך `main` של קובץ הבדיקה שקורא לפונקציה שלכם, ולא על ידי `syscall exit` שלכם, במידה והשתמשתם באחד (וכמובן שגם לא בעקבות קריסת הקוד[5]). הערה זו נכתבה בדם ביטים (של קוד של סטודנטים מסמסטרים קודמים).

   על מנת לוודא את ערך החזרה של התוכנית, תוכלו להשתמש בפקודת ה-`bash` הבאה: `echo $?` (תזכורת: ערך החזרה של התוכנית, במידה ויצאה בצורה תקינה, הוא הערך ש-`main` מחזירה ב-`return` האחרון שלה)
2. מומלץ להיעזר ב-GDB בעת דיבוג הקוד. מדריך לשימוש בדיבאגר GDB זמין באתר הקורס.
3. אם הכל עובד כשורה, אתם יכולים לעבור לחלק ב' של תרגיל הבית, ולאחריו לחלק ג', שהוא בסך הכל הוראות הגשה לתרגיל כולו (שימו לב שאתם מגישים את שני החלקים יחד!).



[5]

חלק ב – פסיקות (אין קשר לחלק א)

מבוא

קראו את כל השלבים בחלק זה, לפני שתתחילו לעבוד על הקוד.
בתרגיל זה נרצה לכתוב שגרת טיפול ב**פסיקת המעבד** המתבצעת כאשר מבצעים פקודה לא חוקית (כלומר, כאשר המעבד מקבל opcode שאינו מוגדר בו).

- כאשר המעבד מקבל קידוד פקודה שאינו חוקי, המעבד עוצר את ביצוע התוכנית וקורא לשגרת הטיפול בפסיקה ב-IDT.
- שגרת הטיפול נמצאת בקרנל, ובלינוקס שולחת סיגנל `SIGILL` לתוכנית שביצעה את הפקודה הלא חוקית. אפשר לראות זאת כאן למשל:

https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/arch/x86/kernel/traps.c#L321

נרצה לשנות את **קוד הקרנל** כך ששגרת הטיפול בפסיקה תשתנה. נעשה זאת באמצעות kernel module.

מה תבצע שגרת הטיפול החדשה?

שגרת הטיפול בפסיקה שלנו (שאותה **אתם הולכים לממש** באסמבלי בעצמכם, בקובץ ili_handler.asm), תיקרא **my_ili_handler** ותבצע את הדברים הבאים:

1. בדיקת הפקודה שהובילה לפסיקה זו. הנחות:
   - הניחו כי הפקודה השגויה היא פקודה של אופקוד בלבד. כלומר, לפני ואחרי ה-opcode השגוי אין עוד בייטים (אין legacy prefix, אין REX).
     - לכן, אורך הפקודה השגויה הוא באורך 1-3 bytes. בתרגיל זה הניחו כי אורך האופקוד השגוי הוא לכל היותר 2 בייטים.
2. קריאה לפונקציה what_to_do עם ה-byte האחרון של האופקוד הלא חוקי, כפרמטר.
   - היזכרו בחומר של קידוד פקודות:
     - אם האופקוד אינו מתחיל ב-0x0F, הוא באורך byte אחד.
     - אחרת (כן מתחיל ב-0x0F), אם הוא אינו מתחיל ב-0x0F3A או 0x0F38, אזי הוא באורך 2 בייטים. לכן, הניחו כי הבייט השני באופקוד אינו 0x3A או 0x38 (אין צורך לבדוק זאת).
   - דוגמאות:
     - עבור האופקוד 0x27, שהינה פקודה לא חוקית בארכיטקטורת x86-64, נבצע קריאה ל-what_to_do עם 0x27.
     - עבור האופקוד 0x0F04, גם לא חוקית, נבצע קריאה ל-what_to_do עם הפרמטר 0x04.
3. בדיקת ערך החזרה של what_to_do
   - אם הוא אינו 0 – חזרה מהפסיקה, כך שהתוכנית תוכל להמשיך לרוץ (תצביע לפקודה הבאה לביצוע מיד לאחר הפקודה הסוררת) וערכו של רגיסטר %rdi יהיה ערך ההחזרה של what_to_do.[6]
     - **שימו לב #1:** שימו לב ש-invalid opcode הינה פסיקה מסוג fault. חשבו מה זה אומר על ערכו של רגיסטר %rip בעת החזרה משגרת הטיפול ושנו אותו בהתאם.
     - **שימו לב #2:** היעזרו בספר אינטל, volume 3 [7], עמוד 222, המדבר על הפסיקה שלנו, בכדי לוודא את תשובתכם ל"שימו לב #1" וגם כדי להחליט האם יש error code או לא.
     - **שימו לב #3:** what_to_do הינה שגרה שתינתן על ידנו בזמן הבדיקה. אין להניח לגביה דבר, מלבד חתימתה (כלומר - שם השגרה, טיפוס פרמטר הקלט וטיפוס ערך החזרה).
   - אחרת (הוא 0) – העברת השליטה לשגרת הטיפול המקורית.

---

[6] בעולם האמיתי אסור לשנות ערכים של רגיסטרים וצריך להחזיר את מצב התוכנית כפי שקיבלתם אותו. כאן אתם נדרשים כן לשנות ערך של רגיסטר, כך שמצב התוכנית לא יהיה כפי שהיה כשהתרחשה הפסיקה. זה בסדר, זה לצורך התרגיל 😉

[7] https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325384-sdm-vol-3abcd.pdf

לפני תחילת העבודה – מה קיבלתם?

בתרגיל זה תעבדו על מכונה וירטואלית דרך qemu (בתוך המכונה הוירטואלית - Virtualiception). על המכונה הזו, אנחנו נריץ kernel module [8] שיבצע את החלפת שגרת הטיפול לזו שמימשתן בעצמכן.
היות והקוד רץ ב-0 ring (kernel mode), במקרה של תקלה מערכת ההפעלה תקרוס. אך זה לא נורא! עליכם פשוט להפעיל את qemu מחדש.

לרשותכם נמצאים הקבצים הבאים בתיקייה part 2:

- initial_setup.sh - הריצו סקריפט זה לפני כל דבר אחר. סקריפט זה מכין את המכונה הוירטואלית לריצת qemu. **עליכם להריץ אותו פעם אחת בלבד** (לא יקרה כלום אם תריצו יותר, אך זה לא נחוץ).
  - יכול להיות שתצטרכו להריץ את הפקודה הבאה, לפני ההרצה (בגלל בעיית הרשאות):

```
chmod +x initial_setup.sh
```

- compile.sh - הריצו סקריפט זה בכל פעם שתרצו לקמפל את הקוד ולטעון אותו (עם המודול המקומפל) למכונה הוירטואלית של qemu (**שימו לב:** עליכם לצאת מ-qemu קודם).
  - גם כאן ייתכן ותזדקקו להרצה של chmod באותו אופן כמו בסעיף הקודם.
- start.sh - הריצו סקריפט זה כדי להפעיל את המכונה הוירטואלית של qemu, לאחר שקימפלתם את תיקיית code וטענתם אותה אל המכונה הוירטואלית של qemu.
  - גם כאן ייתכן ותזדקקו להרצה של chmod באותו אופן כמו בסעיף הקודם.
- filesystem.img - המכונה הוירטואלית אותה תריצו ב-qemu.
- קבצי הקוד שנכתוב, כחלק מהמודול (והיא זו שתקומפל ותרוץ לבסוף ב-qemu) וה-makefile:
  - ili_handler.asm, ili_main.c, ili_utils.c, inst_test.c, Makefile

איך הכל מתחבר - כתיבת המודול

בתיקיה code סיפקנו לכן מספר קבצים:

- **inst_test.c** – simple code example that executes invalid opcode. Use it for basic testing.
- **ili_main.c** – initialize the kernel module – provided to you for testing.
- **ili_utils.c** – implementation of ili_main's functionality – **YOUR JOB TO FILL**
- **ili_handler.asm** – exception handling in assembly – **YOUR JOB TO FILL**
- **Makefile** – commands to build the kernel module and inst_test.

ממשו את הפונקציות ב-ili_utils.c, כך שהשגרה my_ili_handler תיקרא כאשר מנסים לבצע פקודה לא חוקית. איך? Well, זהו לב התרגיל, אז נסו להיזכר בחומר הקורס. כיצד נקבעת השגרה שנקראת בעת פסיקה? פעלו בהתאם.
לאחר מכן, ממשו את הפונקציה **my_ili_handler** ב-ili_handler.asm שתבצע את מה שהוגדר בשלב II.

[8] למי שלא מכיר את המונח kernel module, בלי פאניקה (כי panic זה רע, אבל זה עוד יותר רע בקרנל. פאניקה! בדיסקו זה דווקא בסדר) – מדובר בדרך להוסיף לקרנל קוד בזמן ריצה (ניתן להוסיף לקרנל קוד ולקמפל לאחר מכן את כל הקרנל מחדש, אך כאן לא הזמן ולא המקום לזה). למעשה, נכתוב קוד שירוץ ב-kernel mode ולכן יהיה בעל הרשאות מלאות. אנו נדרש לזה – הרי אנו רוצים לשנות את קוד הקרנל.

זמן בדיקות - הרצת המודול

לאחר שסיימתם לכתוב את המודול, בצעו את השלבים הבאים:

1. הריצו את `./compile.sh` כדי לקמפל את קוד הקרנל ולהכניסו למכונת ה-QEMU
2. הריצו את `./start.sh` כדי לפתוח מכונה פנימית באמצעות QEMU
   a. משתמש: root, סיסמא: root

   כעת אתם בתוך ה-QEMU וכל השלבים הבאים מתייחסים לריצת QEMU.
3. `./bad_inst` כדי להריץ את הקוד `inst_test.asm`, עם הפקודה הלא חוקית (ולקבל הודעת שגיאה בהתאם). ניתן גם להריץ את `bad_inst_2` כדי להריץ את הקוד ב-`inst_test_2.asm`.
4. `insmod ili.ko` כדי לטעון את המודול שלכם (ודאו שהוא נטען ע"י הרצת dmesg)
5. `./bad_inst` כדי להריץ שוב, אך לקבל התנהגות שונה מהקודמת, מכיוון שהפעם השגרה שלכם נקראה.

דוגמת הרצה תקינה ב-QEMU (עם הטסטים **inst_test** ו-**inst_test_2**, ומימוש **what_to_do** שסופק לכם כדוגמה):

```
root@ubuntu18:~# ./bad_inst
start
Illegal instruction
root@ubuntu18:~# insmod ili.ko
root@ubuntu18:~# ./bad_inst
start
root@ubuntu18:~# echo $?
35
root@ubuntu18:~# rmmod ili.ko
rmmod: ERROR: ../libkmod/libkmod.c:514 lookup_builtin_file() could not open buil
tin file '/lib/modules/4.15.0-60-generic/modules.builtin.bin'
root@ubuntu18:~# ./bad_inst_2
start
Illegal instruction
root@ubuntu18:~# insmod ili.ko
root@ubuntu18:~# ./bad_inst_2
start
Illegal instruction
```

(what_to_do מחזירה את הקלט שלה פחות 4. בטסט הראשון הפקודה הלא חוקית היא 0x27, לכן ערך החזרה הוא 0x23, שזה 35. ערך זה הוא גם ערך היציאה של התוכנית, כי כך נכתב הטסט[9], לכן echo $? מדפיס 35. בטסט השני, הפקודה הלא חוקית היא 0x0f04, לכן ערך החזרה של what_to_do הוא 0 והתוכנית חוזרת לשגרה המקורית לטיפול, ששולחת את הסיגנל Illegal Instruction)

פקודות שימושיות

insmod ili.ko

(טוען את המודול ili.ko לקרנל ומפעיל את הפונקציה init_ko שבמודול)

rmmod ili.ko

(מפעיל את הפונקציה exit_ko שבמודול ומוציא את המודול ili.ko מהקרנל)

SHIFT + page up

(גלילת המסך למעלה)

SHIFT + page down

(גלילת המסך למטה)

הערות כלליות

על מנת להבין מה קורה בקרנל – תוכלו להשתמש בפונקציה ()print המוגדרת בקובץ ili_main.c, ולראות את הודעות הקרנל ע"י dmesg.

תיעוד של qemu ניתן למצוא כאן: https://qemu.weilnetz.de/doc/qemu-doc.html

[9] הטסט נכתב כך שמיד לאחר הפקודה הלא חוקית יש ביצוע של קריאת המערכת exit. אתם משנים את %rdi בשגרת הטיפול, לכן ערך היציאה של הטסט ישתנה בהתאם.

חלק ג' - הוראות הגשה לתרגיל בית רטוב 2

אם הגעתם לכאן, זו בהחלט סיבה לחגיגה. אך בבקשה, לא לנוח על זרי הדפנה ולתת את הפוש האחרון אל עבר ההגשה – חבל מאוד שתצטרכו להתעסק בעוד מספר שבועות מעכשיו בערעורים, רק על הגשת הקבצים לא כפי שנתבקשתם. אז קראו בעיון ושימו לב שאתם מגישים את כל מה שצריך ורק את מה שצריך.

עליכם להגיש את הקבצים בתוך zip אחד:

hw2_wet.zip

בתוך קובץ zip זה יהיו 2 תיקיות:

- part1
- part2

ובתוך כל תיקייה יהיו הקבצים הבאים (מחולק לפי תיקיות):

- part1:
  - matrix.asm
- part2:
  - ili_handler.asm
  - ili_utils.c

בהצלחה!!!