ארגון ותכנות המחשב

תרגיל 2 - חלק רטוב

המתרגל האחראי על התרגיל: בועז מואב

שאלות על התרגיל – ב- Piazza בלבד.

הוראות הגשה:

- ההגשה בזוגות.
- על כל יום איחור או חלק ממנו, שאינו באישור מראש, יורדו 5 נקודות.
    - ניתן לאחר ב-3 ימים לכל היותר.
- הגשות באיחור יתבצעו דרך אתר הקורס.
- יש להגיש את התרגיל לפי ההוראות בסוף המסמך. **אי עמידה בהוראות אלו תעלה לכם בנקודות יקרות**.

חלק א – שגרות, קונבנציות, syscalls ומה שביניהן

מבוא

בתרגיל זה אתם תממשו את האלגוריתם של Knuth ליצירת מילה בינארית במשקל מאוזן.
נתחיל בלהבין מה בכלל אומר משקל מאוזן, ולאחריו נציג את האלגוריתם של Knuth.
עבור חלק א' של התרגיל נדרש חומר הקורס עד הרצאה ותרגול 5 בלבד.

מה זה משקל האמינג?

מילה בינארית באורך n היא מחרוזת באורך n, כשכל אות במחרוזת היא 0 או 1. למשל 01110011 היא מילה בינארית באורך 8.
משקל האמינג של מילה בינארית הוא מספר הכניסות במילה שאינן 0 (כלומר, מספר הכניסות שהן 1). למשל, עבור המילה שפגשנו במשפט הקודם, 01110011, משקל האמינג שלה הוא 5.
נהוג לסמן:

$$w_H(01110011) = 5$$

ומה זה משקל מאוזן? אז נניח מעתה ועד סוף התרגיל כי n הוא כפולה של 8, ובפרט זוגי. כעת, תהא x, מילה באורך n, אזי היא מילה במשקל מאוזן אם היא מקיימת את התנאי:

$$w_H(x) = \frac{n}{2}$$

איך עובד האלגוריתם של Knuth?

האלגוריתם של Knuth מקבל מילה בינארית $x = (x_1, x_2, \dots, x_n)$ ומחזיר מילה $\tilde{x} = (\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_n)$ במשקל מאוזן.
אופן פעולת האלגוריתם:

1. $i \leftarrow 1$.
2. בצע פעולת NOT על i הביטים הראשונים, כך שנקבל $x' = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_i, x_{i+1}, \dots, x_n)$.
3. אם x' הוא במשקל מאוזן – סיים את האלגוריתם והחזר $\tilde{x} = x'$.
4. אחרת, $i \leftarrow i + 1$.
5. חזור לשלב 2.

נכונות האלגוריתם? טוב, כאן זה לא קורס באלגוריתמים 😊 אבל אתם יותר ממוזמנים לנסות לבד, לשאול, לקרוא את המאמר של Knuth[1], או לקחת את הקורס "קידוד ואלגוריתמים לזכרונות", בו למשל מוכיחים את המשפט. אבל האלגוריתם עובד. ואתם תכתבו מימוש שלו באסמבלי בתרגיל הקרוב.

[1] Efficient Balanced Codes, DONALD E. KNUTH

שלב ראשון – קריאת קלט מקובץ

תחילה, **תממשו באסמבלי** את הפונקציה שזו חתימתה:

```
unsigned long read_input(unsigned long* codeword);
```

כאשר את המצביע `codeword`, מצביע למערך, אתם צריכים למלא בביטים מתוך קובץ קלט.

- מאיפה מגיע קובץ הקלט? אתם תצטרכו לקרוא לפונקציה הבאה:

```
void get_path(char* _path);
```

שמקבלת כקלט מצביע לכתובת זיכרון וממלאת את הכתובת הזו במחרוזת, שהיא ניתוב (`path`) של קובץ הקלט שממנו אתם צריכים לקרוא, במימושכם ל-`read_input`.

  - היעזרו במשתנה `path` שנתון לכם ב-`bss` בתור `buffer` שתשלחו ל-`get_path`.
- איך נראה קובץ הקלט? הקובץ מתחיל במספר. המספר יהיה כתוב ב-`ASCII` (ספרות ב-`ASCII` למען הסר ספק, אין צורך לבדוק זאת) ואורכו יהיה לא ידוע, אך לא יותר מ-7 ספרות.

איך תדעו שנגמר המספר? יופיע התו `'\n'`, תו ירידת השורה, שערך ה-`ASCII` שלו הוא `0xa`. המספר שקראתם הוא אורך המשך הקלט (לאחר `'\n'` ולא כולל אותו) ב-`bytes`.

- כעת צריך לתרגם. איך תתרגמו את המספר מ-`ASCII` לערך מספרי? אתם תצטרכו לקרוא לפונקציה הבאה:

```
long atam_atol(char* num);
```

הפונקציה תקבל `null-terminated string` של ספרות בלבד (באורך 8 לכל היותר) ותחזיר את הערך המספרי. שימו לב שהמחרוזת חייבת להיות `null-terminated` (עם `'\0'` בסוף).

- אחרי שתבצעו את קריאת המספר ותרגומו, תדעו את n, מספר ה-`bytes` שנותרו לקריאה בקובץ לאחר `'\n'`. כעת נשארו עוד שני שלבים לפונקציה `read_input`.
  - בשלב הראשון, עליכם להעתיק לתוך `codeword` (הקלט שלנו) את n הבייטים שבקובץ. ההעתקה תתבצע כך שה-`byte` הראשון בקובץ, יהיה ה-`byte` הראשון של `codeword`.
  - לבסוף, בשלב השני עליכם להחזיר את $\frac{n}{8}$, כאשר n הוא ה-n שקיבלתם מהקובץ (צריכים להחזיר את מספר המילים המרובעות, ולא את מספר הבייטים).

דוגמה

מסופק לכם הקובץ `test` שתוכנו הוא:

```
$ hexdump -C test
00000000  31 36 0a ff f0 00 00 ff  00 ff 00 fe 00 00 00 ff  |16..............|
00000010  00 00 00                                          |...|
00000013
```

כלומר, שני התווים הראשונים בקובץ הם `'1'` ו-`'6'`, שלאחריהם התו `0xa`, שהוא `'\n'`. ולכן יש 16 תווים לקרוא לאחר מכן אל תוך `codeword`. כלומר, תוכן הזיכרון שמתחיל ב-`codeword` לאחר ביצוע הפונקציה:

codeword	codeword+1	codeword+2	codeword+3	codeword+4	codeword+5	codeword+6	codeword+7
0xff	0xf0	0x00	0x00	0xff	0x00	0xff	0x00
codeword+8	**codeword+9**	**codeword+10**	**codeword+11**	**codeword+12**	**codeword+13**	**codeword+14**	**codeword+15**
0xfe	0x00	0x00	0x00	0xff	0x00	0x00	0x00

הנחות

- המימוש של הפונקציות `get_path` ו-`atam_atol` יסופק לכם בשלב הטסטים ואין להניח לגביו דבר. מימוש לדוגמה של שתיהן נתון בקובץ `get_path.o` המצורף.
  - המימוש שקיבלתם של `get_path` משתמש במחרוזת `"./test"` בתור ה-`path` לקובץ.
  - אתם יכולים ליצור קובץ `test` כזה בעצמכם (לטסטים שלכם) באמצעות `create_input.sh` שסופק לכם.
- **שימו לב:** אל תניחו דבר לגבי המימוש הפנימי של פונקציות אלו. הן יקיימו את קונבציות `System V` ויעשו את המתואר לעיל לגבי כל אחת, אך המימוש הפנימי בשלב הטסטים שלנו לא נתון לכם ולא ניתן להניח אותו.
- ניתן להניח כי הקלט תקין, יש מספיק מקום ב-`codeword` וכי `get_path` ו-`atam_atol` עובדות תקין ואין צורך לבדוק זאת.

שלב שני – חישוב משקל האמינג של הקלט

תממשו באסמבלי את הפונקציה שזו חתימתה:

```
unsigned long hamming_weight(unsigned long* codeword, unsigned long len);
```

כאשר codeword היא מילה בינארית, המורכבת מ-len חלקים בגודל 8 bytes.

עליכם לחשב את משקל האמינג של codeword ולהחזירו. משקל האמינג צריך להיות מחושב על פני כל המילה

דוגמה

עבור הקלט ב-*test* שראינו קודם, נוצרת המילה *codeword* המורכבת משני חלקים (של *8 bytes*)

codeword	codeword+1	codeword+2	codeword+3	codeword+4	codeword+5	codeword+6	codeword+7
0xff	0xf0	0x00	0x00	0xff	0x00	0xff	0x00
codeword+8	**codeword+9**	**codeword+10**	**codeword+11**	**codeword+12**	**codeword+13**	**codeword+14**	**codeword+15**
0xfe	0x00	0x00	0x00	0xff	0x00	0x00	0x00

לכן קריאה לפונקציה באופן הבא: `hamming_weight(codeword,2);`

צריכה להחזיר 43, כמספר הביטים הדולקים במילה זו.

שלב שלישי – ביצוע NOT על k ביטים ראשונים (LSb)

תממשו באסמבלי את הפונקציה שזו חתימתה:

```
unsigned long negate_first_k(unsigned long codeword, unsigned char k);
```

כאשר codeword היא מילה באורך 8 בייטים ו-k הוא מספר הביטים שצריך לבצע עליהם NOT. ערך החזרה הוא ביצוע ה-NOT על k הביטים הראשונים של codeword.

דוגמה

עבור 8 הבתים הבאים, להם נקרא *codeword*:

codeword	codeword+1	codeword+2	codeword+3	codeword+4	codeword+5	codeword+6	codeword+7
0xff	0xf0	0x00	0x00	0xff	0x00	0xff	0x00

שהיא גם מיוצגת באופן הבא (שימו לב ל-little endian): `0xff00ff0000f0ff`

ביצוע של `negate_first_k(codeword, 5);` יחזיר את הפלט: `0xff00ff0000f0e0`

שלב רביעי – הפעלת האלגוריתם של Knuth

תממשו באסמבלי את הפונקציה שזו חתימתה:

```
unsigned long bring_balance_to_the_word(unsigned long* codeword, unsigned long len);
```

כאשר codeword היא מילה בינארית, המורכבת מ-len חלקים בגודל 8 bytes.

עליכם להפעיל את האלגוריתם של *Knuth* שמתואר בתחילת התרגיל, כך שהמילה הבינארית *codeword* תהיה מאוזנת בסוף ריצת הפונקציה, ועליכם להחזיר את האינדקס בו האלגוריתם הפסיק את פעולתו. מותר ואף מומלץ להשתמש בפונקציות שכתבתם בשלבים הקודמים.

דוגמה

תוכן הזיכרון שמתחיל ב-codeword לפני ביצוע האלגוריתם:

codeword	codeword+1	codeword+2	codeword+3	codeword+4	codeword+5	codeword+6	codeword+7
0xff	0xf0	0x00	0x00	0xff	0x00	0xff	0x00
codeword+8	**codeword+9**	**codeword+10**	**codeword+11**	**codeword+12**	**codeword+13**	**codeword+14**	**codeword+15**
0xfe	0x00	0x00	0x00	0xff	0x00	0x00	0x00

ולאחר ריצת האלגוריתם, תוכן הזיכרון שמתחיל ב-codeword יהיה:

codeword	codeword+1	codeword+2	codeword+3	codeword+4	codeword+5	codeword+6	codeword+7
0x00	0x0f	0xff	0xff	0x00	0xff	0x00	0xff
codeword+8	**codeword+9**	**codeword+10**	**codeword+11**	**codeword+12**	**codeword+13**	**codeword+14**	**codeword+15**
0x01	0xff	0xff	0x07	0xff	0x00	0x00	0x00

וניתן לשים לב כי הפכנו את הביטים של 11 ה-bytes הראשונים, ועוד 3 ביטים מה-byte ה-12. סה"כ האינדקס בו הפסקנו הוא **11*8+3=91**. ולכן האלגוריתם יחזיר 91, בנוסף לשינוי של הזיכרון בו codeword נמצאת.

שלב חמישי – טסטים

כל אחד מארבעת השלבים בתרגיל ייבדקו בפני עצמם, על ידי טסטים נפרדים. קובץ ה-main המצורף הוא דוגמה לארבעה טסטים שונים, כל אחד על שלב שונה. הפלט הצפוי:

```
$ ./main.out
Input: 0xff00ff0000f0ff, 0xff000000fe
Hamming weight: 43
After negating the first 5 bits of the first 8-byte: 0xff00ff0000f0e0
The index at which Knuth's algorithm stopped: 91
The balanced word: 0xff00ff00ffff0f00, 0xff07ffff01
```

כאשר את `main.out` יוצרים כך:

```
gcc get_path.o students_code.S main.c -o main.out
```

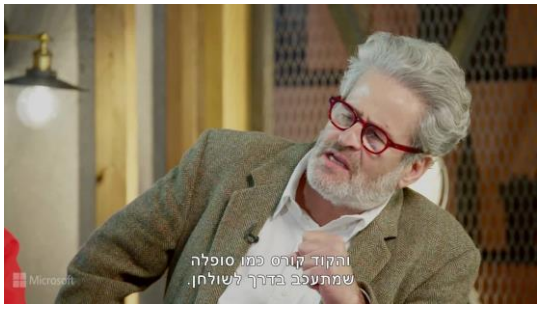
הערה חשובה:

אסור לשנות את את שורת הקימפול הנ"ל, ובפרט אסור להוסיף את הדגל `-fno-pie` או `-no-pie`, מכיוון שאנחנו לא נשתמש בהם והקוד שלכם ייכשל בשלב היצירה. העובדה הזו גוזרת עליכם את ההגבלה הבאה – **אסור להשתמש בשיטת מיעון אבסולוטית.** בהמשך הקורס נלמד את הסיבה לכך שחוסר השימוש ב-`no-pie` גורר זאת.

הערות נוספות

רגע לפני הסוף, אנא קראו בעיון את ההערות, שנוגעות לדרך בה נבדק התרגיל.

1. את הקוד שלכם אתם צריכים לכתוב **בעצמכם** ו**באסמבלי**.
2. אנא ודאו שהתוכנית שלכם יוצאת (מסתיימת) באופן תקין, דרך `main` של קובץ הבדיקה שקורא לפונקציה שלכם, ולא על ידי `syscall exit` שלכם, במקרה שבו השתמשתם באחד (וכמובן שגם לא בעקבות קריסת הקוד[2]). 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, אם יצאה בצורה תקינה, הוא הערך ש-`main` מחזירה ב-`return` האחרון שלה).
3. שימו לב שיהיה `timeout` (20-120 שניות, בהתאם לטסט) איתו הטסטים ייבדקו. כתבו קוד יעיל ככל האפשר.
4. 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!).



[2]

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.[3]
     - **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 [4], 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.

---

[3] 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 😉

[4] 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 [5] 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.

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`.

[5] 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.

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[6], 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)

תקלות נפוצות (מתעדכן)

במקרה של תקלת "אין מקום בדיסק" שמתקבלת בזמן הרצת compile./ – עליכם להוריד מחדש את הקובץ filesystem.img ולהחליף את העותק הישן באחד החדש ואז להריץ את compile./ שוב.

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

[6] 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:
  - students_code.S
- part2:
  - ili_handler.asm
  - ili_utils.c

בהצלחה!!!