ארגון ותכנות המחשב

תרגיל בית 3 (רטוב)

המתרגל האחראי על התרגיל: בועז מואב.

הנחיות:

- שאלות על התרגיל ב- **Piazza** בלבד.
- 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.
- את התרגיל יש להגיש באתר הקורס בקובץ **zip**.
- תיקונים לתרגיל, אם יהיו, יופיעו ממורקרים.

חלק א – שגרות, קונבנציות ומה שביניהן (75 נק')

מבוא

בחלק א' של תרגיל הבית נממש כפל מטריצות. נעשה זאת בשלושה שלבים, כאשר כל אחד מהשלבים ייבדק בנפרד. התרגיל, בכל שלביו, מתייחס למטריצות המכילות מספרים שלמים בלבד שאינם חורגים מגבולות הייצוג של 32 ביט.

פונקציות למימוש בתרגיל

כעת נספק הסברים על שלוש הפונקציות שתממשו בתרגיל זה באסמבלי.
מותר ואף מומלץ להשתמש בפונקציות שכבר מימשתם בתור פונקציות עזר למימוש הפונקציות הבאות, אך עם זאת חשוב לשים לב שכל פונקציה תיבדק בפני עצמה.

get_element_from_matrix

חתימת הפונקציה הראשונה למימוש:

```
int get_element_from_matrix(int* matrix[], int n, int row, int col);
```

קלט: הפונקציה תקבל מצביע בשם `matrix` (פרמטר ראשון) למערך דו-ממדי (מטריצה) בעל `n` (פרמטר שני) עמודות. כמו כן, תקבל הפונקציה שני אינדקסים - `row` (פרמטר שלישי) ו-`col` (פרמטר רביעי).
פלט: היא תחזיר את האיבר שבשורה `row` ועמודה `col` במטריצה `matrix`.
שימו לב: העמודה והשורה נתונים בצורה zero based!!!
הנחות: המיקום `matrix[row][col]` קיים וחוקי במטריצה ובפרט $n > col$ (לא צריך לבדוק).

inner_prod

חתימת הפונקציה השנייה למימוש:

```
int inner_prod(int* mat_a[], int* mat_b[], unsigned int row_a, unsigned
            int col_b, unsigned int max_col_a, unsigned int max_col_b);
```

קלט: הפונקציה מקבלת שני מצביעים למערכים דו-ממדיים (מטריצות) `mat_a` (פרמטר ראשון) ו-`mat_b` (פרמטר שני). למטריצה `mat_a` יש `max_col_a` (פרמטר חמישי) עמודות ולמטריצה `mat_b` יש `max_col_b` (פרמטר שישי) עמודות. בנוסף, היא מקבלת מספר שורה במטריצה `mat_a`, שנתונה בפרמטר `row_a` (פרמטר שלישי) ומספר עמודה במטריצה `mat_b`, שנתונה בפרמטר `col_b` (פרמטר רביעי).
פלט: הפונקציה תחזיר את המכפלה הפנימית בין השורה `row_a` לבין העמודה `col_b`.
שימו לב: גם כאן האינדקסים של העמודה והשורה נתונים בצורה zero based.
הנחות: גם כאן אפשר להניח שהשורה `row_a` והעמודה `col_b` אינן חורגות מגבולות ממדי המטריצה. ניתן גם להניח שהממדים מתאימים (המכפלה הפנימית חוקית).

matrix_multiplication

חתימת הפונקציה השלישית למימוש:

```
int matrix_multiplication(int* res[], int* mat_a[], int* mat_b[], unsigned
            int m, unsigned int n, unsigned int p, unsigned int q);
```

קלט: הפונקציה מקבלת שני מצביעים למערכים דו-ממדיים (מטריצות) `mat_a` (פרמטר שני) ו-`mat_b` (פרמטר שלישי). המטריצה הראשונה מממד $m \times n$ (פרמטרים רביעי וחמישי) והמטריצה השנייה מממד $p \times q$ (פרמטרים שישי ושביעי). בנוסף, מקבלת הפונקציה מצביע למערך דו-ממדי של היעד, מטריצת היעד `res` (פרמטר ראשון).
פלט: אם המטריצות בממד תקין לצורך ביצוע כפל המטריצות $mat\_a \cdot mat\_b$ – המטריצה תחזיר במטריצת היעד `res` את תוצאת המכפלה וכערך חזרה תחזיר 1 (`true`). אחרת, לא תבצע דבר ותחזיר 0 (`false`).
שימו לב: כמו כן, ניתן ואף מומלץ להשתמש בפונקציה `set_element_in_matrix`, שנתונה לכם בקובץ `aux_hw3.o` המצורף לתרגיל. הסבר נוסף על הפונקציה יופיע מיד.
הנחות: המטריצה `res` מוגדרת באופן תקין, והוקצה לה מספיק זיכרון בהתאם לממדים הצפויים שלה.

פונקציית עזר נתונות

לתרגיל זה נתונה פונקציית העזר `set_element_in_matrix` שחתימתה היא:

```
void set_element_in_matrix(int* matrix[], unsigned int num_of_columns,
                           unsigned int row, unsigned int col, int value);
```

הפונקציה מקבלת מצביע בשם `matrix` (פרמטר ראשון) למערך דו-ממדי (מטריצה) בעל `num_of_columns` (פרמטר שני) עמודות. בנוסף, היא מקבלת שני אינדקסים `row` ו-`col`, שורה (פרמטר שלישי) ועמודה (פרמטר רביעי), וגם את `value`, ערך (פרמטר חמישי) שאותו היא תעדכן במטריצה במקום המבוקש.

כלומר, היא תבצע (רעיונית): `matrix[row][col] = value`.

<u>שימו לב</u>: גם כאן האינדקסים של העמודה והשורה נתונים בצורה zero based.

<u>הערה</u>: הפונקציה שומרת על קונבנציית הקריאות `System V` לארכיטקטורת `x86-64` כפי שנלמדה בקורס.

מותר לכם להשתמש בפונקציה זו ומובטח שהיא תהיה קיימת גם בזמן הטסטים (בקובץ `aux_hw3.o`).

דרישות מימוש

את כל המימושים תשלימו בקובץ students_code.S. קובץ S הוא קובץ אסמבלי, המתאים ל-gcc. אין הבדל אמיתי בינו לבין קובץ asm [1] (ויש שיטענו, בצדק, שדווקא קבצי S מתאימים יותר מ-asm לקורס).

<u>הערות חשובות לגבי מימוש התרגיל:</u>

1. שמרו על הקונבנציות!!! בתרגיל זה נכתוב קוד שמשלב אסמבלי ו-C ולכן קוד שלא ישמור על קונבנציות, ייכשל בטסטים.
2. אסור לכם להוסיף קבצים מלבד students_code.S. בפרט לא את `aux_hw3.o`.
3. אסור לכם להוסיף משתנים ל-section data (אותו קיבלתם נתון).
   a. אם ברצונכם להשתמש במשתנים מקומיים, אתם כמובן יכולים (ואף מומלץ לעשות זאת) – אך תצטרכו לעשות זאת לפי הקונבנציות שנלמדו בקורס.
4. <u>מומלץ להיעזר ב-GDB בעת דיבוג הקוד</u>. מדריך לשימוש בדיבאגר GDB זמין באתר הקורס.
5. אנחנו מקמפלים את הקבצים שלכם בצורה שגוזרת עליכם את ההגבלה הבאה – אסור להשתמש בשיטת מיעון אבסולוטית, או בקבוע שהוא כתובת.

   בפרט, כשאתם יוצרים לעצמכם טסטים <u>אסור להוסיף את הדגלים fno-pie או no-pie</u> לשורת הקימפול (בהמשך הקורס נלמד על הדגלים האלה ונבין למה הוספת הדגלים קשורה לאיסור המדובר).
6. 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, במידה ויצאה בצורה תקינה, הוא הערך ש-`main` מחזירה ב-`return` האחרון שלה).
7. כתבו קוד סביר (מבחינת יעילות). קוד לא יעיל בצורה חריגה עלול להיכשל בטסטים.
8. 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!).

---

[1] https://stackoverflow.com/questions/34098596/assembly-files-difference-between-a-s-asm

[2]

חלק ב – פסיקות (25 נק')

מבוא

לפני שאתם מתחילים את חלק זה, אנחנו ממליצים לכם לחזור על תרגול וסדנה 6 ולראות שאתם יודעים לענות על השאלות הבאות – מה זה IDT? מהן הפקודות sidt ו-lidt? מה מכילות כניסות ב-IDT? מהי שגרת טיפול בפסיקה? באילו הרשאות היא רצה? מהי חלוקת האחריות בין שגרת הטיפול לבין המעבד? באיזו מחסנית נשתמש? איך היא נראית בכל שלב? במה זה תלוי? כעת, קראו את כל השלבים בחלק זה לפני שתתחילו לעבוד על הקוד.

בתרגיל זה נרצה לכתוב שגרת טיפול בפסיקת המעבד המתבצעת כאשר מבצעים פקודה לא חוקית (כלומר, כאשר המעבד מקבל opcode שאינו מוגדר בו).

- כאשר המעבד מקבל קידוד פקודה שאינו חוקי, המעבד קורא לשגרת הטיפול בפסיקה ב-IDT.
- שגרת הטיפול בלינוקס שולחת סיגנל SIGILL לתוכנית שביצעה את הפקודה הלא חוקית. למשל:

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

נרצה לשנות את קוד הקרנל כך ששגרת הטיפול בפסיקה תשתנה (שאלה למחשבה - למה חייבים לשנות את קוד הקרנל?) ונעשה 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 223, 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] ב"עולם 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 שמימשתם בעצמכם.
היות והקוד רץ ב-ring (0 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. ביצד נקבעת השגרה שנקראת בעת פסיקה? פעלו בהתאם.
שימו לב כי ב-ili_utils.c אנו רוצים לגשת לחומרה. מהי הדרך לשנות קוד כאשר אנו רוצים לבצע פקודות ב-low level?
לאחר מכן, ממשו את הפונקציה my_ili_handler ב-ili_handler.asm שתבצע את מה שהוגדר בשלב II.

[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.

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. התעלמו מכל המלל שיופיע, כולל הערות בצבעים אדומים וירוקים. זה תקין.
   b. לאחר העליה – 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, מכיוון שהפעם ה-handler **שלכם** נקרא.

שימו לב שלאחר ביצוע שלב 1 תקבלו את האזהרה הבאה:

```
WARNING: could not find /home/student/Documents/atam2/part2/.ili_handler.o.cmd f
or /home/student/Documents/atam2/part2/ili_handler.o
```

יש להתעלם מאזהרה זו. שגיאות אחרות עלולות להצביע על בעיה מהותית ואף לאי יצירת הקבצים הרלוונטיים (בעיות בנייה, בעיות קישור ועוד), לכן שימו לב מהן ההערות שמופיעות בעת הרצת compile.

גם בעת עליית ה-QEMU ירוץ הרבה טקסט במסך, בצבעים משתנים של אדום, ירוק וצהוב. לא להיבהל, זה תקין.

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).

[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.

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)

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.
באופן כללי, במהלך העבודה ייתכן שתצטרכו למחוק את filesystem.img אצלכם ולהחליף אותו בעותק שבאתר, לאחר שניסיתם לבצע שינויים ומשהו השתבש בעותק של הקובץ שעליו אתם עובדים. מומלץ לשמור עותק ללא שינויים בצד, כדי לבצע את ההחלפה הזו במהירות (ולא להוריד כל פעם מהאתר 😊).

הערות כלליות

נשים לב שבתרגיל זה שינינו את הקוד של הגרעין! ומכיוון שזה קוד של גרעין אז אין לנו דרך לדבג אותו ולהבין אם הוא אכן עובד כפי שציפינו שיעבוד. **דיבוג קוד קרנל הוא קשה**. כאן לא תוכלו להיעזר ב-gdb. תצטרכו, ברוב הפעמים לנסות "לבדוק בצד" את מה שכתבתם, על דוגמאות צעצוע, ולראות שהקוד עושה מה שהוא אמור לעשות. רוב המחשבה והדיבוג נעשים "בראש" ולכן הקוד שאתם אמורים לכתוב הוא יחסית פשוט וקצר.

ובכל זאת, על מנת לעזור מעט להבין מה קורה בקרנל – תוכלו להשתמש בפונקציה print() המוגדרת בקובץ ili_main.c, ולראות את הודעות הקרנל ע"י כתיבה של הפקודה dmesg בטרמינל של ה-qemu.
למה print() ולא printf()?
הפונקציה printf() היא פונקציה של הסיפרייה libc, וכאשר אנו כותבים קוד גרעין או עושים לו מודיפיקציה אין לנו את האפשרות לגשת לסיפרייה זו (ואנחנו גם לא צריכים כי יש לנו רמת הרשאה 0, ו-libc היא ספרייה עבור משתמשים), לכן אנו צריכים לגשת לפונקציות שנמצאות בגרעין – אז עזרנו לכם וכתבנו עבורכם מעטפת נחמדה 😊
דבר נוסף, אנא קראו את ההערות בקבצי הקוד שעליכם למלא. זה יכול רק לעזור.

תיעוד של qemu ניתן למצוא כאן: https://qemu.weilnetz.de/doc/2.11/qemu-doc.html

חלק ג' - הוראות הגשה לתרגיל הבית

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 (שם הקובץ לא משנה).
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!!!