

Omer os

שם בית הספר: נעמי שמר גן יבנה

שם העבודה: פרויקט יב הנדסת תוכנה

שם התלמיד : עומר גולן

תז: 326247814

שם המורה: אנטולי פיימר

תאריך הגשה:

תוכן

3	מבוא
3	מערכת הפעלה -
3	יעדים בפרויקט -
3	בעיות תועלות וחסכוניות -
3	טכנולוגיית הפרויקט -
4	תיחום הפרויקט -
4	לוח תכנון הפרויקט -
4	סיכונים בפרויקט -
5	תיאור תחום הידע - פרק מילולי
5	תהליך הBOOT -
7	MBR -
7	ReadFromDisk -
7	getMemoryMap -
8	Protect mode -
11	KERNEL -
11	IDT
13	ISR - in
13	Interrupt.asm
14	PIC -IRQ
14	PIT -
15	Keyboard driver -
16	ניהול זיכרון -
17	ארכיטקטורת הפרויקט
18	תיאור הטכנולוגיות בפרויקט:
18	סביבת הפיתוח:
19	מסכי המערכת -
20	חולשות ואיומים -
20	חולשות בMBR -
20	חולשה בProtected mode -
20	חולשה בשימוש בפסקיות -
21	מימוש הפרויקט
27	התקנת המערכת -
30	רפלקציה
31	ביבליוגרפיה+מקורות מידע

מבוא

הפרויקט שבחרתי לעשות הוא בעצם ליצור מערכת הפעלה מאפס ללא כל תלות בספרייה חיצונית, בסופו של דבר מטרת מערכת ההפעלה היא לעלות מהביוס של המחשב ולתפקד כסוג של CMD בסיסי שאפשר יהיה להשתמש בו במקום מערכת ההפעלה של ווינדוס

הפרויקט אינו מיועד לקהל הרחב ויוצר למטרות למידה בלבד

לפני שאסביר לעומק את עקרונותיה של מערכת ההפעלה קודם כל אגדיר מה היא בכלל מערכת הפעלה

מערכת הפעלה -

"מערכת הפעלה היא תוכנה המגשרת בין המשתמש, החומרה ויישומי התוכנה. זו התוכנה הראשונה שעולה עם הדלקת המחשב והיא זו המאפשרת לו לפעול. מערכת ההפעלה מספקת שלושה ממשקים: ממשק משתמש (User Interface) ממשק עבור החומרה על ידי מנהלי התקנים וממשק תכנות היישומים (API) (מערכת ההפעלה היא רכיב חיוני בכל מחשב. תהליך טעינתה של מערכת ההפעלה, המתבצע עם הדלקת המחשב, קרוי אתחול -". ויקיפדיה.

יעדים בפרויקט -

המטרה המרכזית של המערכת אותה כתבתי היא ליצור ניהול זיכרון, ממשק משתמש בסיסי, ו"דיבור" עם החומרה

בעיות תועלות וחסכוניות -

מערכת ההפעלה מהווה במה משותפת שעליה יכולות לרוץ תוכנות שישרתו את משתמש הקצה. לדוגמה, כשאני כותב את ספר זה, המפתח של Office לא היה צריך לכתוב רכיב שמתממשק עם הלוח אם כדי לקבל את הקשות המקלדת שלי, ולא היה צריך לדבר ישירות עם כרטיס המסך כדי שהאותיות יופיעו לי עליו. במקום זאת, המפתח דאג להנחות את המחשב: "תגיב בצורה א' למקש כזה ובצורה ב' למקש כזה" ו"תצייר לי את האות ג". מערכת ההפעלה עשתה בשביל המפתח את העבודה והיוותה את הגשר בין החומרה לבין התוכנה.

כיום יש מגוון רחב של מערכות הפעלה ביניהם המוכרות ביותר הם ווינדוס של מייקרוסופט, הוצאות לינוקס ומערכת ההפעלה של אפל מאק OS. כל מערכות ההפעלה האלו נבנו על ידי חברות גדולות במשך שנים ומספקים מבחר עצום של ביצועים, בפרויקט שלי אראה מערכת הפעלה בסיסית שמהווה דרך למידה להבנה יותר טובה של איך כל מערכות ההפעלה המודרניות פועלות כיום

טכנולוגיית הפרויקט –

בפרויקט זה ישנם כמה טכנולוגיות בסיסיות שחובה להשתמש בהם על מנת לתפעל את הפרויקט

1. QEMU – זה תוכנת קוד פתוח המאפשרת לבצע ווירטואליזציה של מערכת ההפעלה באמצעות אימלוטור שאנו נבחר להשתמש בו (X86במקרה שלנו) מה שיאפשר לנו לא לעלות את מערכת ההפעלה על מחשב רגיל ולהדליק אותו כל פעם מחדש
2. לינקוס – את כל הפרויקט כתבתי על מערכת ההפעלה לינקוס מכיוון שיש שם מגוון רחב של כלים שאיתם נוח לפתח את הפרויקט
3. Gcc - זהו מהדר שיכול לקמפל קבצים לפורמטים וארכיטקטורות שונות מהמחשב שעליו הוא רץ, נשתמש בו על מנת לקמפל את קבצי הפרויקט
4. NASM-כמו GCC הנאסמ ישמש אותנו לקמפל את קבצי התוכנית

תיחום הפרויקט -

הפרויקט עוסק בתכונות Low level ועקרונות מערכת ההפעלה והחומרה הבסיסיים של המחשב

לוח תכנון הפרויקט –

בעת כתיבת הפרויקט חילקתי את העבודה ליעדים ומטרות

- כתיבת bootloader שיעלה בזמן עליית המחשב
- הגדרת GDT וכתיבת קרנל בסיסי
- כתיבת מערכת לניהול זיכרון באמצעות סקטורים
- הגדרת אינטראפטים - IDT
- כתיבת SHELL ו UI בסיסי
- אינטגרציה עם המקלדת

סיכונים בפרויקט –

מכיוון שהפרויקט מאוד מרוכב ודורש הרבה מחקר ולמידה עצמאית היה קיים חשש של אי עמידה בלוח הזמנים – בפועל לאחר עמידה בלוח זמנים מסודר הצלחתי לכתוב בהצלחה את הפרויקט

תהליך ה-BOOT -

על מנת להפעיל את מערכת ההפעלה הקוד הבא הוא הקוד הבסיסי אשר נדרש על מנת לעלות את מערכת ההפעלה מן הביוס:

```
[BITS 16]

[ORG 0x7c00]

jmp $ TIMES 510-($-$$)

db 0 dw 0xaa55
```

כשאנו מתחילים את הקוד בהוראה [BITS 16] אנו מנחים את המרכיב לתרגם את הקוד לארכיטקטורת 16 ביט Mode Real

CPU - רכיב ה, (Unit Processing Central) CPU-מעבד הוא המוח של המחשב - הוא האחראי העיקרי לביצוע של הוראות. מעבדים יודעים לדבר בשפה מאוד מוגדרת, ואוסף כל ה"מילים" שהם יודעים נקרא Instruction set (פקודות). את המילים המעבד מקבל בצורת קוד מכונה - אותות חשמליים שמייצגים את סט הפקודות. אנו יכולים לתקשר עם המעבד דרך שפה מתווכת בשם Assembly שפת סף. שפה זו היא בעלת התאמה חד-חד ערכית לקוד המכונה, ורכיב בשם Assembler מרכיב מתרגם את שפת הסף לקוד המכונה. כפי שציינתי, המרכיב שאני השתמשתי בו הוא NASM.

אופני פעולה של המעבד-

למעבדים מודרניים מאז שחרור סט הפקודות הידוע בשם 80286 ב-1982 קיימים מספר אופני פעולה שונים.

הראשון נקרא - Mode Real מצב זה מקביל לאופן הפעולה של כל מעבד שנוצר לפני סדרת ה. 80286 -

במצב זה המעבד עובד ללא שום מנגנוני הגנה - הגישה לחומרה נעשית באופן ישיר וללא שום בקרה. מנגד, Mode Protected מאפשר להשתמש בתכונות כמו זיכרון וירטואלי, ריבוי משימות והגדרת רמות שונות של הרשאות בעת הרצת קוד.

מערכות הפעלה מודרניות משתמשות ב, Mode Protected-אך בכדי לאפשר תאימות לתוכניות שנכתבו ל- Mode Real קיימים אופן פעולה נוסף בשם, Mode Virtual המאפשר שימוש בתכונות שקיימות ב Real Mode מתוך, Mode Protected זאת משום שלא ניתן לעבור מ Mode Protected ל- Mode Real ללא ביצוע אתחול מחדש למעבד

דבר שמן הסתם ממש לא רצוי בעת שימוש רגיל במחשב. (הבדל עיקרי נוסף הוא ש Mode Protected-מאפשר לעבור משימוש ב-16 ביט לשימוש ב-32 ביט. ההבדל בין 16 ביט ל-32 ביט נעוץ בשני שינויים עיקריים. ראשית, ה Registers-אוגרים במעבד גדלים ל 32-ביט, אך ההבדל השני, והעיקרי יותר, הוא שגודל הזיכרון שהמעבד יכול לגשת אליו גדל. מאז הגדילה בנפחי הזיכרון בשוק קיים אופן פעולה נוסף בשם Mode Long

שמאפשר להשתמש ב-64 ביט. כיום, כדי לאפשר תאימות עם מערכות הפעלה ישנות, מחשבים נדלקים כברירת מחדל למצב, Mode Real ולכן אנו חייבים לכתוב את הקוד הראשוני במצב זה

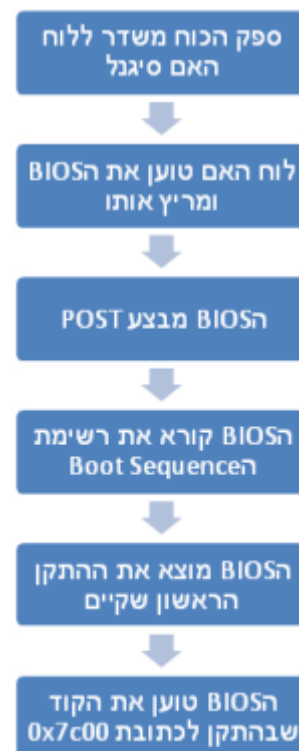
אופן	Real Mode	Protected Mode	Virtual Mode	Long Mode
Address Space	20 ביט	32 ביט	20 ביט	64 ביט
גודל אוגרים	16 ביט	32 ביט	16 ביט	64 ביט
גישה לחומרה	ישירה	מוגנת	מוגנת	מוגנת
קיים מאז	תמיד	80286	80386	Opteron
גישה לזיכרון	סגמנטים	דפדוף	סגמנטים	דפדוף

[org 0x7c00] - גם פקודה זו מנחה את המרכיב, ולא את המעבד. הפקודה אומרת ל NASM-שהקוד שלנו נטען מהכתובת, 0x7c00בניגוד ל-. 0x0000 סיבה זו נעוצה בכך שה BIOS-טוען את התכנית לכתובת זו.

הביוס –

ה (BIOS (System Output Input Basic הוא הרכיב הראשון שמופעל כשאנחנו מדליקים את המחשב. ה BIOS - אחראי על ביצוע בדיקה ראשונית למחשב המכונה ("POST בדיקה עצמית לאחר הפעלה"). (בדיקה זו אחראית לוודא שכל רכיבי המחשב קיימים.

כמו כן הביוס אחראי למצוא ולהפעיל את מערכת ההפעלה המתוקנת על המחשב, העלאת מערכת ההפעלה קוראת על פי הסדר הבא:



TIMES 510-(\$-\$) db 0

- MBR

ה (Record Boot Master)-MBR הוא חלק בהארד דיסק שמשמש שתי מטרות:

1. הוא מכיל רשימה של כל המחיצות בדיסק. מחיצות הן חלוקות לוגיות של הדיסק שמקלות על גישה וחלוקה של הדיסק. מבחינת מערכת ההפעלה, כל מחיצה היא דיסק בפני עצמה.
2. MBR-יכול להכיל קוד בסיסי שירץ כאשר ה BIOS-קורא אותו.

מידע	טווח בתים
איזור קוד. איזור זה יכול להכיל קוד שירץ לאחר שה-BIOS קורא את הMBR.	1-440
חתימת הדיסק.	441-444
ממולא ב-NULL.	445-446
טבלת המחיצות. מחולקת ל-4 רשומות של 16 בתים, שמייצגות את מספר המחיצות הפיזיות האפשריות על הארד דיסק יחיד. היום ניתן גם ליצור מחיצות לוגיות שיכילו מספר תתי מחיצות.	447-510
0x55 - בית ראשון של חתימת הMBR	511
0xAA - בית שני של חתימת הMBR	512

- READFROMDISK

נוסף על כך בתהליך ה boot קיים קובץ הנקרא ReadFromDisk – אותו לקחתי מהאינטרנט ומטרתו היא להראות למערכת ההפעלה מאיפה היא צריכה לקרוא מן הדיסק הקשיח, המיקום נקבע במיקום ES:BX

באוגר DH – נמצא מספר הסקטורים הנמצאים אותם נרצה לקרוא

```
times 25600 db 0 ; 50 blank sectors
```

הקוד הבא מאתחל 50 סקטורים ריקים בדיסק אותם כל אחד בגודל של 512 בייט

- GETMEMORYMAP

עוד דבר שנצטרך לעשות על מנת להפעיל מערכת ההפעלה בצורה טובה עם שימוש בפסיקות אנו נצטרך לדעת לטפל בIVT (Interrupt vector table,) בעת עליית מערכת ההפעלה ולשם כך נרצה להגדיר את ה Memory map - קישור למידע נוסף - [כאן](#)

- PROTECT MODE

השלב הבא בהתליך הBOOT הוא להעביר מערכת ההפעלה ממצב של REAL MODE למצב של Protected mode עליו הסברתי בחלק הקודם. חלק קריטי במעבר מReal moden למצב של protected mode הוא הגדרת GDT בצורה נכונה

- GDT – global describer table

GDT הוא מבנה בזיכרון שנועד לתאר תכונות של אזורי זיכרון. ה-GDT-מהווה את אחד ההבדלים העיקריים בין Protected Mode ל-Real Mode-ניתן להגדיר הגבלות גישה לאזורים מסויימים בזיכרון. ה-GDT-מורכב מעד 8192 "שורות" שכל אחת מהן מתארת אזור יחיד שנקרא סגמנט, והיא נראית כך:

```

1 GDT_start:                ; to define the Global Descriptor Table we define bytes, words and double words
2                            ; to write them "raw" in the bin file (like we did in the disk reading example, or to define the
3                            ; end of the bootsector)
4
5 GDT_null:                  ; mandatory NULL descriptor
6     dd 0x0
7     dd 0x0
8
9 GDT_code:                  ; code segment descriptor
10    dw 0xffff               ; limit, bits 0-15 (completed at line 14)
11    dw 0x0                  ; Base, bits 0-15
12    db 0x0                  ; Base, bits 16-23 (completed at line 15)
13    db 0b10011010           ; Flags (look at paper for meaning)
14    db 0b11001111           ; Flags, Limit bits 16-19
15    db 0x0                  ; Base, bits 24-31
16
17 GDT_data:                  ; data segment descriptor
18    dw 0xffff               ; limit, bits 0-15
19    dw 0x0                  ; Base, bits 0-15
20    db 0x0                  ; Base, bits 16-23
21    db 0b10010010           ; Flags (look at paper for meaning)          DATA FLAGS ARE DIFFERENT IN DATA SEGMENT
22    db 0b11001111           ; Flags, Limit bits 16-19
23    db 0x0                  ; Base, bits 24-31
24
25 GDT_end:
26
27 GDT_descriptor:
28     dw GDT_end - GDT_start - 1    ; Size of GDT - 1
29     dd GDT_start                  ; start of GDT

```


סיביות	שדה
0-15	16 הסיביות הראשונות של גבול הסגמנט שאותו השורה מתארת.
16-39	24 הסיביות הראשונות של בסיס הסגמנט שאותו השורה מתארת.
40-47	בית הגישה
48-51	4 הסיביות האחרונות של גבול הסגמנט.
52-55	דגלים
56-63	8 הסיביות האחרונות של בסיס הסגמנט.

סיבית	שדה
7	Present. האם הסגמנט קיים. אמור להיות דלוק תמיד.
5-6	DPL. הטבעת (Ring) שאותה צריך בכדי לגשת לזיכרון.
4	תמיד 1.
3	Executable. אם הסגמנט דלוק הוא קוד, אם הוא מכובה הוא מידע.
2	Direction/Conforming <ul style="list-style-type: none"> אם הסגמנט הוא מידע, הסיבית קובעת אם הסגמנט "גדל" למטה או למעלה. אם הסגמנט הוא קוד: ערך של 0 קובע שניתן להריץ את הקוד רק מהטבעת שקבועה ב-DPL. ערך של 1 קובע שניתן להריץ את הקוד מטבעת שווה או נמוכה יותר משקבועה ב-DPL.
1	Readable/Writeable. אם סגמנט הוא קוד, הסיבית קובעת האם האזור ניתן לקריאה. אם הסגמנט הוא מידע הסיבית קובעת האם
0	Accessed. אנו לא אמורים לשנות סיבית זו. המעבד משנה אותה כאשר הוא ניגש לסגמנט שאותו השורה מתארת.

סיבית	שדה
3	Granularity. אם הוא מכיל 0 הסגמנט מיושר לגודל של בית, אם הוא מכיל 1 הוא מיושר לגודל של עמוד (4KB).
2	Size. אם הוא מכיל 0, הסגמנט הינו סגמנט בעל ערך 16 ביט, אם הוא מכיל 1 אז הסגמנט הוא סגמנט בעל ערך 32 ביט.
1	תמיד 0.
0	תמיד 0.

השלב האחרון בתהליך הBOOT הוא הקפיצה לKERNEL שם בעצם יהיה כתוב כל הקוד שישמש למערכת

ניתן לעשות זאת בקלות באמצעות הפקודה הבאה:

KERNEL_LOCATION equ 0x1000 – קביעת המיקום הקבוע של הקרנל

בעת קימופל התוכנית נגיד ללינקר לטעון את הקרנל לכתובת 0x1000

```
i386-elf-ld -o "Binaries/full_kernel.bin" -Ttext 0x1000 "Binaries/kernel_entry.o"
```

לאחר מכן נוכל פשוט לקפוץ לכתובת הזאת מתוך קובץ הBOOT

```
BEGIN_PM:
    mov bx, [Extended_Memory_Size] ; record memory s
    jmp KERNEL_LOCATION ; jumps to entry_kernel
```

ומשם לקפוץ לקוד האמיתי של הקרנל

```
Kernel > asm kernel_entry.asm
1 section .text
2 [bits 32]
3
4 global MemSize
5     MemSize: db 0, 0
6     mov [MemSize], bx ; Get memory size from bx
7
8 [extern main]
9 call main ; calls kernel function main()
10
11 jmp $
12
13 %include "../intDef/interrupt.asm"
14
15
16
```

אחרי שהצלחנו לבצע את תהליך הBOOT בהצלחה קפצנו לקוד של הקרנל שלנו. עכשיו מה עושים?
דבר ראשון אם נרצה להדפיס תו כלשהו למסך לא נוכל להשתמש בשום פסיקת ביוס (כרגע לפחות)
לכן מה שנצטרך לעשות זה לכתוב בצורה ישירה לווידאו ממורי
שמתחיל בכתובת 0xb8000 למצב טקסט (ניתן גם לכתוב במצב גרפי אך מטעמי נוחות בחרתי
להשתמש מצב טקסט)

```
unsigned char *vidmem=(unsigned char *)0xb8000;
```

ביכולת זאת נשתמש בהמשך שנרצה לכתוב את הSHELL שלנו ופונקציות הדפסה למסך
אך מה יקרה אם נרצה לקבל קלט מן המקלדת? או להשתמש בפסיקות אחרות של הביוס על מנת לקבוע את מיקום
הסמן

בשביל זה נצטרך להגדיר משהו שנקרא IDT

– Interrupt Descriptor Table - IDT

IDT הוא מבנה בזיכרון שנועד להנחות את המעבד כיצד לפעול בעת קבלת פסיקות. הוא המקביל ב Mode Protected-
ל, IVT.

הוא מורכב מ-256 שורות, שמקבילות ל-256 מספרי פסיקות שונות (0x00 int - 0xff int) שיכולים להיווצר. כל שורה
מנחה את המעבד לאיזו פונקצייה לקרוא כדי שתוכל לטפל בפסיקה שהתקבלה

כשהמעבד מקבל פסיקה, הוא מחפש את ההיסט שלה ב IDT- (sizeof(idtEntry)*interrupt_number) על מנת שיוכל
לדעת כיצד לטפל בה:

בדומה לGDT הIDT הוא data struct והוא נראה כך:

```

struct idt_entry          // IDT structure
{
    unsigned short base_lo;
    unsigned short sel;
    unsigned char always0;
    unsigned char flags;
    unsigned short base_hi;
} __attribute__((packed));

struct idt_ptr            // IDT pointer
{
    unsigned short limit;
    unsigned int base;
} __attribute__((packed));

struct idt_entry idt[256];
struct idt_ptr _idt;

extern "C" void _idt_load();          // ---> interrupt.asm

void idt_set_gate(unsigned char num, unsigned long base, unsigned short sel, unsigned char flags)
{
    idt[num].base_lo = (base & 0xFFFF);
    idt[num].base_hi = (base >> 16) & 0xFFFF;

    idt[num].sel = sel;
    idt[num].always0 = 0;
    idt[num].flags = flags;
}

```

סיביות	שדה
0-15	16 הסיביות הראשונות של ההיסט (Offset) בזיכרון של הפונקציה שמטפלת בפסיקה.
16-31	הסלקטור שאליו מתייחס ההיסט. מן הסתם נרצה שסלקטור זה יהיה סלקטור הקוד של הקרנל.
32-36	שמור לשימוש עתידי. מומלץ לאתחל כאפסים.
37-39	חסר שימוש. יש לקבוע כאפסים.
40-47	דגלים
48-63	16 הסיביות האחרונות של ההיסט בזיכרון של הפונקציה שמטפלת בפסיקה.

בית הדגלים נראה כך:

סיביות	שדה
7	Present. האם הפסיקה בשימוש.
5-6	ה-DPL (Descriptor Privelege level) של הפסיקה. בדומה לשדה ב-GDT, דבר זה מאפשר להגדיר הרשאות מינימליות שמהם יכול קוד לקרוא לפסיקה זו.
4	לא נמצאה דוקומנטציה בעלת סימוכין לשימוש בסיבית זו. מצאתי מספר מקורות שכנראה כולם נובעים ממקור משותף אחד שטוענים שזה "Storage Segment" (עד לכתיבת שורות אלו טרם הצלחתי להבין את כוונת המשורר). בכל מקרה, בספציפיקציות של אינטל נראה שיש לקבוע את סיבית זאת כ-0.
3	גודל הזיכרון שאליו מכוונת שורה זו. 0 מייצג 16 ביט, 1 מייצג 32 ביט.
2-0	סוג השורה. מכיוון שכרגע אנו מתעסקים אך ורק בפסיקות יש לקבוע אותן כ-110 שמסמן שורה מסוג interrupt gate.

ISR - Interrupt service routine

בקובץ זה אנחנו ממשיכים להגדיר את הIDT

```
extern void isr0();

extern void isr1();

extern void isr2();

extern void isr3();

extern void isr4();

extern void isr5();
```

בשורות אלו אנו מנחים את המהדר להתייחס לפונקציות isr0 עד isr255 כפונקציות שקיימות בקובץ חיצוני. הפונקציות הללו ממומשות בקובץ interrupt.asm

```
void isrs_install()
{
    idt_set_gate(0, (unsigned)_isr0, 0x08, 0x8E);
    idt_set_gate(1, (unsigned)_isr1, 0x08, 0x8E);
    idt_set_gate(2, (unsigned)_isr2, 0x08, 0x8E);
    idt_set_gate(3, (unsigned)_isr3, 0x08, 0x8E);
    idt_set_gate(4, (unsigned)_isr4, 0x08, 0x8E);
    idt_set_gate(5, (unsigned)_isr5, 0x08, 0x8E);
    idt_set_gate(6, (unsigned)_isr6, 0x08, 0x8E);
}
```

בשורות קוד אלו אנו מאתחלים את השורות במבנה הנתונים שבנינו. אנו קובעים שהאזור בזיכרון שאליו אנו קופצים לאחר קבלת פסיקה מספר 0 יהיה תחילת הפונקציה, isr0 וכך בהתאמה לכל פסיקה.

– INTERRUPT.ASM

בקובץ זה אנו יוצרים את כל הISR (Routines Service Interrupt)-פונקציות אלו הם הפעולות האחריות לבצע את הנדרש לאחר קבלת פסיקה

- PIC1 IRQ

הוא רכיב חומרתי המאפשר לנו "לתרגם" פסיקות חומרתיות לפסיקות תוכניות. רכיב זה חוסך לנו זמן וכוח עיבוד יקר - במקום לעבור על כל רכיבי התוכנה באופן רציף ולתשאל אותם בכדי לבדוק האם מצבם השתנה או לא, אלא נרשם צריכים להפסיק את פעולתה הרגילה של מערכת הפעלה לאחר קבלת פסיקה שמקבלת אוטומטית עדיפות על הקוד הרגילה

לדוגמה, נניח ואנו רוצים לממש התממשקות עם מקלדת. בהנחה שהיינו ממשים יכולת זו בצורה הרגילה, אנו היינו צריכים כל כמה מילי שניות לקרוא למקלדת ולבדוק האם מקש נלחץ. שיטה זו הייתה מצריכה אחוז גבוה של כוח העיבוד שלנו ויש להכפיל מספר זה במספר רכיבי החומרה שאנו רוצים להתממשק איתם.

במקום זאת, אנו מגדירים את המקלדת לשלוח לנו פסיקה כאשר מקש נלחץ. כך, אנו יכולים להתעלם לחלוטין מקיומה של המקלדת, עד אשר מתקבלת פסיקה זו. בעת קבלת הפסיקה אנו נטפל בה, ולאחר מכן נחזור לרצף ההוראות הרגילה.

- PIT

ה (PIT) (Timer Interval Programmable) הוא רכיב חומרתי המאפשר לנו לעקוב אחר זמנים. רכיב זה עובד בתדירות של 1.193182 מגה הרץ מספר הנראה אקראי, אך למעשה נובע משיקולים של נוחות

קיימים במחשב שני מתנדנים שהמכפלה המשותפת של התדירויות שלהם היא התדירות של ה (PIT) (PIT-עובד בצורה הבאה: אנו קובעים מחלק תדירות (Divisor Frequency), וכאשר ה PIT-מבצע את מספר סיבובי השעון שצויין במחלק התדירות, הוא משחרר פולס דרך אחד הערוצים שמחוברים אליו. ה PIT-מחובר לשלושה ערוצים - ערוץ 0 מחובר ל- PIC הראשי דרך IRQ0

```

6  int timer_ticks = 0;
7  int seconds = 0;
8
9  void timer_phase(int hz)
10 {
11     int divisor = 1193180 / hz; /* Calculate our divisor */
12     outb(0x43, 0x36); /* Set our command byte 0x36 */
13     outb(0x40, divisor & 0xFF); /* Set low byte of divisor */
14     outb(0x40, divisor >> 8); /* Set high byte of divisor */
15 }
16
17 void timer_handler(struct regs *r)
18 {
19     /* Increment our 'tick count' */
20     timer_ticks++;
21
22     if (timer_ticks % 18 == 0)
23     {
24         seconds++;
25         //kprint(toString(seconds, 10));
26     }
27 }
28
29
30
31 void sleep (int ticks){
32     int startTicks = timer_ticks;
33     while(timer_ticks < startTicks + ticks){}
34     return;
35 }
36
37
38 void timer_install()
39 {
40     irq_install_handler(0, timer_handler);
41 }
42

```

– KEYBOARD DRIVER

השימוש במקלדת מתבצע באמצעות פקודת Port של אסמבלי המאפשרת גישה ישירה חומרה

```
unsigned char scancode;
scancode = inb(0x60);
```

שימוש בפקודה INB 0x60 מחזירה את הסקאן קוד של הלחץ הנלחץ מן המקלדת ושומרת אותו בתוך המשתנה

Key	Down	Up	Key	Down	Up	Key	Down	Up	Key	Down	Up
Esc	1	81	[1A	9A	, <	33	B3	center	4C	CC
1 !	2	82]	1B	9B	. >	34	B4	right	4D	CD
2 @	3	83	Enter	1C	9C	/ ?	35	B5	+	4E	CE
3 #	4	84	Ctrl	1D	9D	R shift	36	B6	end	4F	CF
4 \$	5	85	A	1E	9E	* PrtSc	37	B7	down	50	D0
5 %	6	86	S	1F	9F	alt	38	B8	pgdn	51	D1
6 ^	7	87	D	20	A0	space	39	B9	ins	52	D2
7 &	8	88	F	21	A1	CAPS	3A	BA	del	53	D3
8 *	9	89	G	22	A2	F1	3B	BB	/	E0 35	B5
9 (0A	8A	H	23	A3	F2	3C	BC	enter	E0 1C	9C
0)	0B	8B	J	24	A4	F3	3D	BD	F11	57	D7
- _	0C	8C	K	25	A5	F4	3E	BE	F12	58	D8
= +	0D	8D	L	26	A6	F5	3F	BF	ins	E0 52	D2
Bksp	0E	8E	;	27	A7	F6	40	C0	del	E0 53	D3
Tab	0F	8F	' "	28	A8	F7	41	C1	home	E0 47	C7
Q	10	90	~	29	A9	F8	42	C2	end	E0 4F	CF
W	11	91	L shift	2A	AA	F9	43	C3	pgup	E0 49	C9
E	12	92	\	2B	AB	F10	44	C4	pgdn	E0 51	D1
R	13	93	Z	2C	AC	NUM	45	C5	left	E0 4B	CB
T	14	94	X	2D	AD	SCRL	46	C6	right	E0 4D	CD
Y	15	95	C	2E	AE	home	47	C7	up	E0 48	C8
U	16	96	V	2F	AF	up	48	C8	down	E0 50	D0
I	17	97	B	30	B0	pgup	49	C9	R alt	E0 38	B8
O	18	98	N	31	B1	.	4A	CA	R ctrl	E0 1D	9D
P	19	99	M	32	B2	left	4B	CB	Pause	E1 1D 45 E1 9D C5	-

כמו עם הטיימר ממקודם אנחנו נשתמש בirq על מנת "להתקין" את את handlern של המקלדת על מנת שלא נצטרך לבדוק כל הזמן עם לחצן נלחץ ורק כאשר הוא נלחץ הקוד של Interruptn יופעל

ניהול זיכרון –

בתחילת המאמר הזכרתי שאיפסנו את 50 הסקטורים לאפסים דבר זה נותן לנו זיכרון איתו נוכל לעבוד

ניהול הזיכרון במערכת ההפעלה שהכנתי פועל בצורה מאוד פשוטה המדמה את פונקציית Malloc ב C הפונקציה מחזירה פוינטר למקום פנוי בזיכרון שמאופס לאפסים

```
#define FREE_MEM 0x10000;
```

הכתובת שבה החלטתי להתחיל את הזיכרון הפנוי הוא הכתובת 0x10000

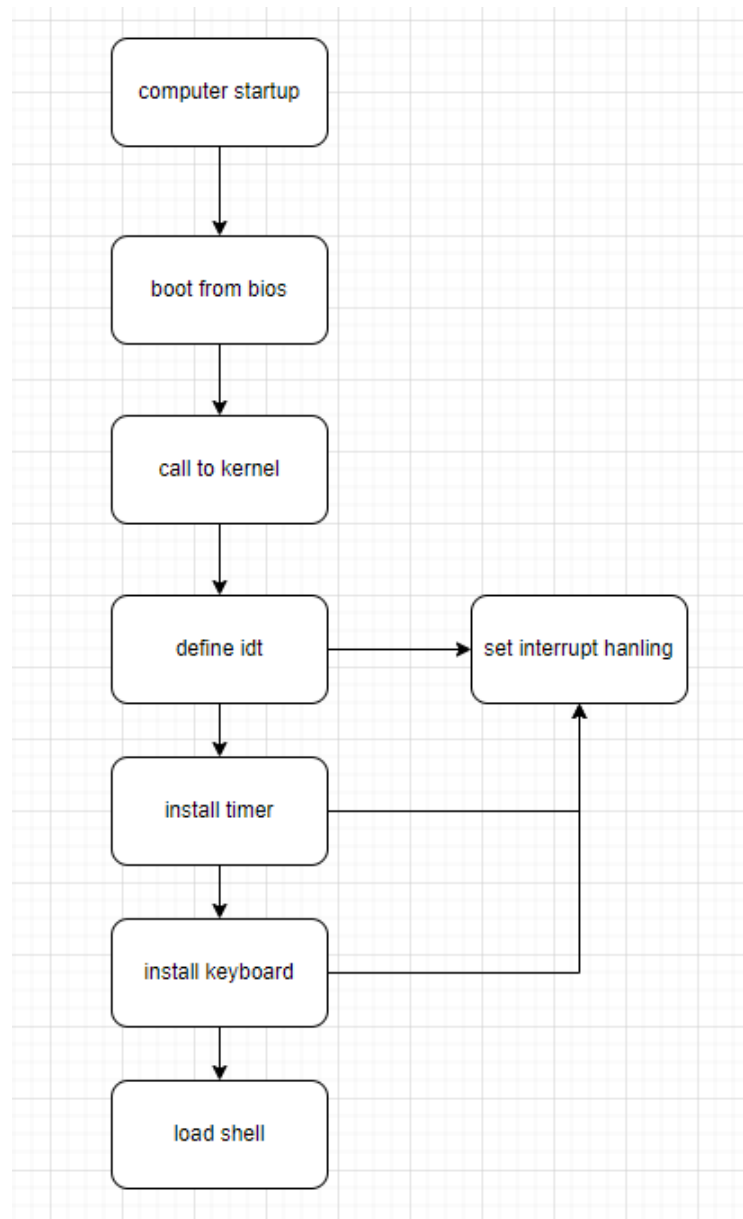
– Shell + VGA DRIVERS

במערכת ההפעלה השתמשתי VGA driver אשר נותן את היכולת לכתוב למסך ועד מגוון פונקציות המאפשרות לבצע אינטרקציה עם הווידאו ממורי

בחלק מן המקרים כתבתי באופן ישיר לווידאו ממורי ובחלק אחר מן מקרים השתמשתי בפקודה PORT על מנת למקם את הסמן על המסך

באמצעות שימוש בVGA drivers ועוד כמה פונקציות ניתן להציג UI לפרוייקט שמתפקד כסוג של CMD

תרשים זרימה מרכזי למערכת -



תיאור הטכנולוגיות בפרויקט:

כל המערכת נכתבה באסמבלי X86 ו C וקומפלה באמצעות GCC ו NASM

סביבת הפיתוח:

על מנת לפתח את מערכת ההפעלה אפשר להשתמש בכל מערכת הפעלה קיימת אך אני בחרתי באופן ספציפי להשתמש ב Kali linux מטעמי נוחות (התקנת GCC ו NASM אפרט כיצד להתקין אותם באופן יותר מפורט בהמשך)

- GCC

אוסף המהדרים של גנו (GCC) היא מערכת מהדר המיוצרת על ידי פרויקט גנו התומכת בשפות תכנות שונות. GCC הוא מרכיב מפתח בשרשרת הכלים של GNU והמהדר הסטנדרטי עבור רוב מערכות ההפעלה דמויות יוניקס.

פקודת gcc בלינוקס. GCC הוא ראשי תיבות של GNU Compiler Collections המשמש להידור בעיקר שפת C ו C++.

NASM – אסמבלר קוד פתוח שיתופי עבור מעבדי INTEL

אסמבלר: היא תוכנית מחשב המתרגמת או: מהדרת (תוכנית שנכתבה בשפת סף (Assembly) לשפת מכונה הניתנת לביצוע על ידי המחשב.

האסמבלר מותאם בדרך כלל לשפת מכונה ספציפית, אם כי אסמבלרים מסוימים יכולים לטפל בתוכניות הכתובות בשפות סף של מעבדים שונים.

– IDE

סביבת פיתוח משולבת (באנגלית Integrated Development Environment: או בקיצור IDE) היא תוכנת מחשב המסייעת למתכנתים לפתח תוכנה.

ניתן להשתמש בכל IDE אך באופן אישי העדפתי להשתמש ב VSCODE

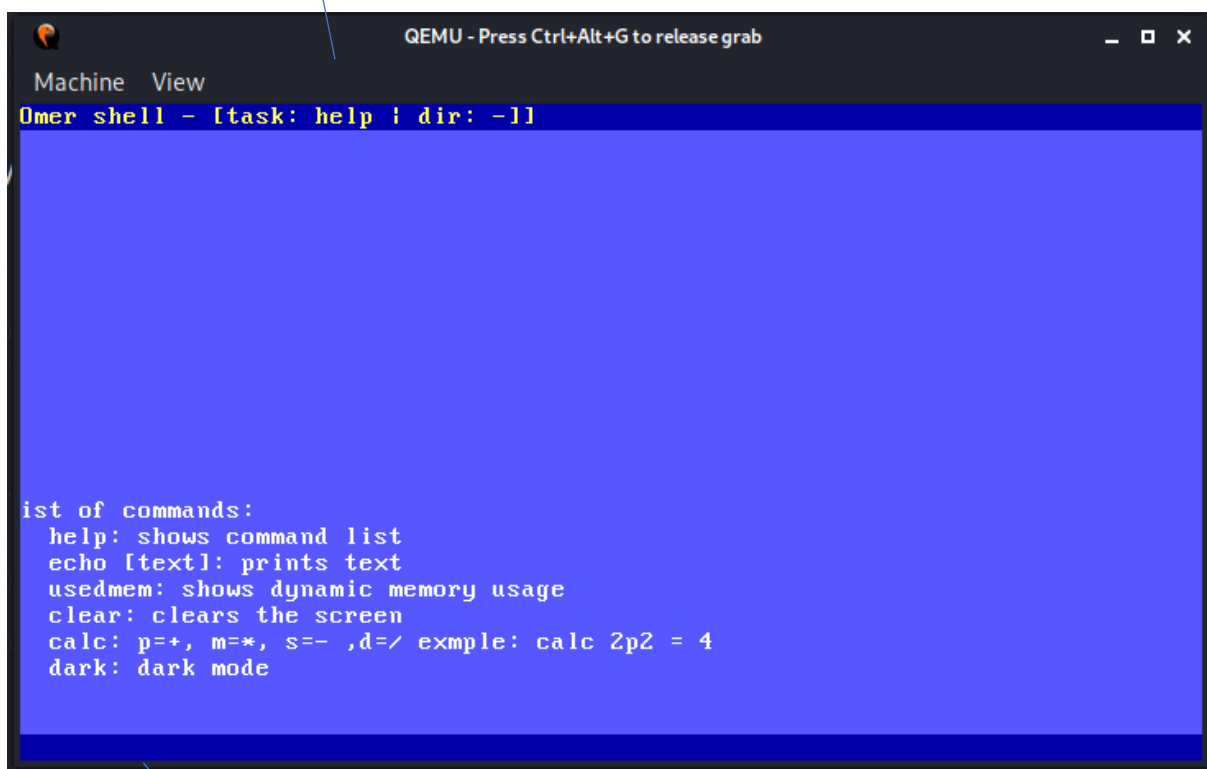
– QEMU

Qemu הוא אמולטור מעבד המבוסס על המרת קוד בינארי לקוד ברמה גבוהה יותר כך שניתן יהיה להבין אותו על ידי המכונה המארחת. זו הייתה המשימה הראשונה שלה, אם כי **ניחן** גם **ביכולות וירטואליזציה** בזכות הפונקציה הראשונית שלה, וזה בדיוק מה שהיא משמשת לה במקרה שלנו.

למערכת יש מסך אחד ראשי אשר מתפקד בתור CMD

ניתן לכתוב Help וללחוץ אנטר על מנת לקבל את רשימת הפקודות

הפקודה האחרונה שהתבצע



```
QEMU - Press Ctrl+Alt+G to release grab
Machine View
Omer shell - [task: help | dir: -]

list of commands:
help: shows command list
echo [text]: prints text
usedmem: shows dynamic memory usage
clear: clears the screen
calc: p=+, m=*, s=-, d=/ example: calc 2p2 = 4
dark: dark mode
```

מקום בוא ניתן לכתוב את הפקודות

חולשות ואיומים –

חולשות ב-MBR –

ה-MBR הוא רכיב פגיע ביותר ובעייתי מבחינת אבטחת מידע. ראשית, הוא אינו נמצא תחת מערכת הפעלה מסוימת, כך שגם מערכת ההפעלה המוקשחת ביותר לא יכולה להגן או לשחזר שינויים כאשר היא אינה הרכיב הראשון שמופעל. שנית, שינויים בו קשים לזיהוי על ידי משתמש הקצה - אין קבצים מוזרים שנשארים ולא קיימים תהליכים שהמשתמש לא מודע להם.

ניתן לבצע שני שינויים עיקריים ל-MBR-שהמטרה משתנה בין הרס לבין

- וירוס בעל הראשות כתיבה ל-MBR יכול פשוט למחוק את כל הרשומות ולמנוע ממערכת ההפעלה להיטען. בסבירות גבוהה ביותר המשתמש (וגם הרבה טכנאי מחשב) יניח שכל הדברים שלו נמחקו, ויבצע פרמוט ללא בדיקה לגבי קיום תבניות שיזהו מחיצות של מערכות הקבצים על הדיסק. מתקפה זו לא דורשת ידע טכני רב או תחכום בקוד - פשוט צריך למלא את הסקטור הראשון בדיסק בזבל

- מתקפה מתוחכמת יותר יכולה לבצע שתי מניפולציות עיקריות על ה-MBR-שינוי של הקוד שמוכל ב-MBR או הוספת מערכת הפעלה ראשונית שתופעל ורק אז תפעיל את מערכת ההפעלה העיקרית. ניתן לנצל שני שינויים אפשריים אלו בכדי לבצע מודיפיקציה מקודמת של הזיכרון או ה-Registry אך אפשרות מעניינת במיוחד היא הפעלת מערכת הפעלה ראשונית שתתפקד כ-Hypervisor באופן פשוט, רכיב שיוצר מכונה וירטואלית, ותפעיל את מערכת ההפעלה העיקרית בתוך הסביבה הוירטואלית הזו. ממצב זה ניתן לזייף ולשנות כל רכיב אפשרי, מהבקשות והתשובות שחוזרות מרכיבי החומרה עד זמן המחשב. באופן תיאורטי לא ניתן לזהות מתקפה זו מתוך מכונה נגועה כל עוד המתקפה ממומשת כראוי כל עוד היא לא מכילה חתימות ידועות שניתן לזהות או שגיאות שניתן ליצור. סוג מתקפה זה הודגם על ידי רוטקיט בשם blue pill

חולשה ב-PROTECTED MODE –

מעבר ל-Mode Protected בכלל, וה-GDT-בפרט, מהווים נכס גדול לאבטחת המידע של מחשבים אישיים. לולא היינו יכולים לנהל את הגישה לזיכרון ברמה חומרתית, לא ניתן היה לאבטח בצורה מהותית את המחשב. כמו כל כלי שנועד להגן, קיימות מספר חולשות שניתן לנצל אותן בכדי לעקוף מנגנון זה. בין היתר, נמצאו כמה חולשות שנוצלו על ידי Rootkits שונים שאפשרו להוסיף שורה נוספת ב-GDT-בשורה זו הוגדר סגמנט שחולש על כל הזיכרון ובעל הרשאות מלאות. כך הקוד הזדוני יכול לגשת לאיזה מקום בזיכרון שירצה.

חולשה בשימוש בפסיקות –

בלינוקס, למשל, הפסיקה 0x80 שמורה לשימוש של System Call. תוקף שמוסגל בדרך מסוימת לשנות את ה-IDT-מוסגל להפנות ל-ISR-שבנה בעצמו (טכניקה הנקראת Hooking) בדרך זו התוקף יכול להקשיב לכל קריאה שמתבצעת למערכת ההפעלה, ולשלט כמעט על כל פעולה שמתבצעת משינוי זמנים על ידי ה-PIT-לשינוי המידע שנשמר על הדיסק הקשיח. מכאן שה-IDT-משך את תשומת הלב של חוקרי האבטחה וכותבי הוירוסים כאחד, ושימש פעמים רבות כקטור שוירוסים ורוטקיטים השתמשו בו, אם כי היום כמעט ולא, לאור הקלות שבגילוי המתקפה. ניתן לחשוב על שלוש דרכים עיקריות לבצע מתקפה כזו:

1. אם לקוד יש הרשאות מלאות, הוא פוטנציאלית יכול להריץ את הפקודה lidt וכך לטעון IDT חדש
2. אם התוקף יודע איפה יושב ה-IDT-בזיכרון (בין אם זה מקום או קבוע, כפי שקורה ב-Windows-או בין אם הוא קיבל את המידע דרך הפקודה sidt), הוא יכול לשנות אותו ישירות בזיכרון על ידי mov פשוט
3. לפעמים מערכות הפעלה חושפות לדרייברים אפשרות להרשם כקולבק על IRQ מסוים. אם התוקף יכול להשתיל דרייבר כזה בתחילה רשימת הקולבקים, הוא יכול לשנות את המידע ששאר הדרייברים בתחתית הרשימה מקבלים.

מימוש הפרויקט

קובץ/מחלקה (מכיוון שהפרויקט כתוב ב C אין באמת מחלקות אך אתייחס לקבצים בצורה כזאת)	יעוד המחלקה

BOOT	

Boot.asm	אחראי על העלאת המערכת מהביוס
GDT -> boot.asm	מבנה GDT
readFromdisk.asm -> boot.asm	קריאה סקטורים מהדיסק
EnterPmode.asm -> boot.asm	כניסה למצב מוגן
printString.asm/ptstringStringPM.asm	מיועד להדפסה בעת עליית הבוט... בעיקר בשביל דיבאגינג

DRIVERS	

Colors.h	מכיל רשימה של קבועים בהם הקוד ב HEX של הצבעים
CONVERSIONS.cpp	מיועד להמרת מספרים ל STRING או להפך בבסיסים שונים
Keyboard.cpp	דרייבר שמאפשר שימוש במקלדת
Port.cpp	דיבור עם החומרה
String.cpp	מספר פונקציות הנועדו לשימוש בסטרינגים
VGA.cpp	דרייבר שעוזר לתקשר עם המסך

kernel	

Empty_end.asm	מיועד לסוף הקובץ שם יהיו סקטורים ריקים
Kernel_entry.asm	קובץ שאליו נקפוץ מתהליך הבוט
Kernel.cpp	הקובץ המרכזי שממנו נקראות כל הפונקציות השונות במערכת

IntDef - מטרת הקבצים בתקיה זו היא ההגדרה של השימוש בפיסקות במערכת, אני לא אפרט על כל קובץ בנפרד מכיוון שעשית זאת מקודם בצורה מפורטת	

ניהול הזיכרון במערכת	Memory.cpp
אחראי על תפקוד SHELL	Shell.cpp
קובץ המכיל את הפונקציות של SHELL	shellFunctions.cpp

בדיקות ודיבוג :

מכיוון שאנחנו עובדים עם הקוד הראשון שרץ בעת עליית המחשב קשה מאוד לדבג אותו בצורה נוחה, אך אם זאת אפשר לעשות שימוש ב-GDB על מנת לדבג את הפרויקט כמו כן יש להשתמש ב-3 INT על מנת להראות לגDB היכן נמצא breakpoint

מדריך למשתמש

ניתן להשיג את כל קבצי הפרויקט מן הקישור הבא - <https://github.com/omergolan1/OmerOS>

מערכת קבצי הפרויקט –

OmerOS

- | A_Setup
- | | setup-gcc.sh
- | Binaries
- | | boot.bin
- | | CmdMode.o
- | | CodeMode.o
- | | Conversions.o
- | | empty_end.bin
- | | Floppy.o
- | | full_kernel.bin
- | | idt.o
- | | irq.o
- | | isr.o
- | | kernel_entry.o
- | | kernel.o
- | | Keyboard.o
- | | mem.o
- | | OS.bin
- | | port_io.o
- | | shellFunctions.o
- | | shell.o

- | |string.o
- | |timer.o
- | |VGA_Text.o
- └ |zeroes.bin
- |Binariestimer.o
- |Bootloader
- | |AvailableMemory.asm
- | |boot.asm
- | |EnterPM.asm
- | |GDT.asm
- | |PrintString.asm
- | |PrintStringPM.asm
- | |ReadFromDisk.asm
- └ |zeroes.asm
- |Drivers
- | |CodeMode.cpp
- | |CodeMode.h
- | |colors.h
- | |Conversions.cpp
- | |Conversions.h
- | |Keyboard.cpp
- | |Keyboard.h
- | |port_io.cpp
- | |port_io.h
- | |string.cpp
- | |string.h

- | Typedefs.h
- | VGA_Text.cpp
- └─| VGA_Text.h
- | intDef
- | idt.cpp
- | idt.h
- | interrupt.asm
- | irq.cpp
- | irq.h
- | isr.cpp
- | isr.h
- | timer.cpp
- └─| timer.h
- | Kernel
- | empty_end.asm
- | kernel.cpp
- └─| kernel_entry.asm
- | Memory
- | mem.cpp
- └─| mem.h
- | README.md
- | run.sh
- | Shell
- | shell.cpp
- | shellFunctions.cpp
- | shellFunctions.h



— L | shell.h

— L todo.txt

התקנת המערכת –

על מנת להוריד ולהפעיל את הפרויקט יש להוריד סביבה וירטואלית של לינוקס (מומלץ להשתמש בהרחבה של לינוקס לווירנדוס מבוסס וובנטו)

לאחר מכן יש להריץ את כל הפקודות מן הקובץ של הSETUP הנמצא בתוך הפרויקט :

```
#nasm and qemu
```

```
sudo apt install nasm
```

```
sudo apt install qemu
```

```
sudo apt-get install qemu-kvm
```

```
#GCC cross compiler for i386 systems (might take quite some time, prepare food)
```

```
sudo apt update
```

```
sudo apt install build-essential
```

```
sudo apt install bison
```

```
sudo apt install flex
```

```
sudo apt install libgmp3-dev
```

```
sudo apt install libmpc-dev
```

```
sudo apt install libmpfr-dev
```

```
sudo apt install texinfo
```

```
export PREFIX="/usr/local/i386elfgcc"
```

```
export TARGET=i386-elf
```

```
export PATH="$PREFIX/bin:$PATH"
```

```
mkdir /tmp/src
```

```
cd /tmp/src
```

```
curl -O http://ftp.gnu.org/gnu/binutils/binutils-2.35.1.tar.gz
```

```
tar xf binutils-2.35.1.tar.gz
```

```
mkdir binutils-build
```

```
cd binutils-build
```

```
../binutils-2.35.1/configure --target=$TARGET --enable-interwork --enable-multilib --disable-  
nls --disable-werror --prefix=$PREFIX 2>&1 | tee configure.log
```

```
sudo make all install 2>&1 | tee make.log
```

```
cd /tmp/src
```

```
curl -O https://ftp.gnu.org/gnu/gcc/gcc-10.2.0/gcc-10.2.0.tar.gz
```

```
tar xf gcc-10.2.0.tar.gz
```

```
mkdir gcc-build
```

```
cd gcc-build
```

```
echo Configure. . . . . :
```

```
./gcc-10.2.0/configure --target=$TARGET --prefix="$PREFIX" --disable-nls --disable-libssp --  
enable-language=c++ --without-headers
```

```
echo MAKE ALL-GCC:
```

```
sudo make all-gcc
```

```
echo MAKE ALL-TARGET-LIBGCC:
```

```
sudo make all-target-libgcc
```

```
echo MAKE INSTALL-GCC:
```

```
sudo make install-gcc
```

```
echo MAKE INSTALL-TARGET-LIBGCC:
```

```
sudo make install-target-libgcc
```

```
echo HERE U GO MAYBE:
```

```
ls /usr/local/i386elfgcc/bin
```

```
export PATH="$PATH:/usr/local/i386elfgcc/bin"
```

לאחר סיום ההורדה יש להיכנס לתקיית הפרוקיט ולבצע את הפקודות הבאות:

1. Chmod +x run.sh – לתת הרשאות הרצה לקובץ

2. ./run.sh



אחרי זה הפרויקט אמור לרוץ ללא בעיות

הבחירה לעשות מערכת הפעלה מאפס ללא שום ידע קודם באה מן הרצון שלי לאתגר את עצמי וללמוד על ההיבטים הכי נמוכים של מערכת ההפעלה וכיצד היא פועלת. בהתחלה לא היה לי מושג לאן אני נכנס וכמה זמן/השקעה התהליך יקח. הייתי צריך לחקור ולקרוא הרבה מאוד מאמרים שונים כמו כן על מנת להצליח לבנות מערכת הפעלה חייב להיות ידע בסיסי באסמבלי C ו במהלך כתיבת הפרויקט נתקלתי בהרבה מאוד אתגרים שחלקים קשורים למבנה של השפה עצמה במהלך העבודה על הפרויקט יכולת המחקר שלי וההבנה של איך עובד המחשב השתפרה מאוד. בראייה לאחור הייתי יכול לשפר את הפרויקט אפילו יותר ולהוסיף לו עוד מגוון דברים מעניינים כמו מערכת קבצים או כתיבת דרייבר רשת כמו כן מבחינת יעול הקוד היה אפשרי להשתמש ב-MACRO על מנת לשפר את נראות הקוד ולהימנע מכפל קוד לסיכום אני שמח מאוד שבחרתי לעשות את הפרויקט הזה למדתי ממנו המון והוא נתן לי הרבה מאוד כלים להמשך אני רוצה להגיד תודה לאנטולי המורה שלי שעזר לי בכל הדרך ובלעדיו לא הייתי חושב בכלל על לעשות פרויקט בסדר גודל כזה אני גם רוצה להגיד תודה לכל האנשים שעזרו לי בתהליך כתיבת הפרויקט

מדריכים של אינטל.

: <http://www.intel.com/content/www/us/en/processors/architectures-software-developermanuals.html>

דוקימנטציה על מספר רב של פקודות x86

<http://faydoc.tripod.com/cpu/index.htm>

רשימת האינטרפטים של ראלף בראון

<http://ctyme.com/rbrown.htm>

osdevwiki - המקור בו השתמשתי הכי הרבה מכיל כל מה שצריך לדעת

<http://wiki.osdev.org>