

**פרויקט גמר הגנת סייבר**

**נושא העבודה:** מערכת הפעלה

**שם תלמיד:** יונתן פרידקין

**ת.ז תלמיד:** 209602564

**שם בית ספר ועיר :** קריית החינוך ע"ש עמוס דה-שליט, רחובות

**שם המנחה :** הילה גורן ברנע

**מועד הגשה :** 15/05/2024

תוכן עניינים

[1. מבוא 3](#_Toc121815660)

[2. תיאוריה 5](#_Toc121815661)

[3. תוצר סופי 10](#_Toc121815662)

[4. תהליך כתיבת הפרויקט 24](#_Toc121815663)

[5. מרכיבי פתרון 25](#_Toc121815664)

[6. תסריטי בדיקה 33](#_Toc121815665)

[7. רפלקציה 34](#_Toc121815666)

[8. הוראות התקנה ותפעול 35](#_Toc121815667)

[9. ביבליוגרפיה 35](#_Toc121815668)

[10. נספחים 36](#_Toc121815669)

1. **מבוא**
   1. נושא העבודה

הפרויקט עוסק בפיתוח מערכת הפעלה אישית באמצעות השפות c++, assembly. זוהי מערכת ההפעלה מסוג 32 ביט protected mode שמקיימת פיצ'רים כגון segmented memory, interrupt handling, תקשורת עם החומרה, ומערכת קבצים.

* 1. מטרות מרכזיות

המטרות היישומיות של הפרויקט הן:

* מימוש מנגנוני אבטחה מוכרים מבוססי תהליכי זיכרון
* מימוש מערכת קבצים שתומכת בקבצי טקסט ותומכת בתיקיות ותתי תיקיות
* מימוש חדר צ'אט בין מערכות הפעלה זהות ששוכנות במחשבים שונים
* ד

המטרות האישיות של העבודה הן:

* העמקת הידע שלי במערכות הפעלה וביסוסו על ניסיון מעשי
* חיזוק ההבנה שלי בשפות תכנות low – c++, assembly
* עניין אישי – נושא מערכות ההפעלה מעניין אותי מאוד והפרויקט הוא הזדמנות טובה לעסוק בכך.
  1. רציונל
  2. קישור לחומר הנלמד

במהלך השנה, התחלנו ללמוד את שפת התכנות c++ במהלך הלמידה על מערכות הפעלה ועל מערכות קבצים. מטרת הפרויקט היא להעמיק ולהוסיף על הידע הנלמד בכיתה כדי לשפר את הידע שלי בתחום מערכות ההפעלה וחיזוק ההיכרות שלי עם שפת התכנות c++ וassembly שמהוות שפות תכנות חיוניות לאלו שרוצים לעסוק בפיתוח קוד.

1. **תיאוריה**
   1. תיאוריה

לפני שניגש לקוד הפרויקט עצמו, יש צורך לדון בהרחבה על התיאוריה שעומדת מאחוריו. כדי לכסות את כלל הרקע התאורטי נדון במספר נושאים:

1. מושגים כלליים מעולם התכנות – assembly, c,c++, compiler, linker, gnu, gcc
2. תאוריה כללית של מערכות הפעלה – RAM, אזורי זיכרון, תקשורת עם המעבד, מבנה של קובץ שמורץ
3. יצירת סביבת עבודה וסביבת פיתוח נכונות ((cross compiler. לפני התאוריה של בניית מערכת ההפעלה נדון ביצירת סביבת העבודה הנכונה, שתפלוט קוד שנוכל להריץ כמערכת ההפעלה האישית – דגלי קומפילציה. נעסוק בקומפיילר, לינקר, gcc., qemu
4. סגנון הבנייה של מערכת ההפעלה – monolithic kernel
5. Booting & The booting process– התהליך שעובר המחשב מרגע ההדלקה שלו ועד לרגע שמערכת ההפעלה תקינה ומתפקדת (כלומר הקרנל וכל שאר הקבצים שהוא צריך טעונים לRAM).
6. תהליכי זיכרון – 16 bit real mode, 32 bit protected mode – gdt. זהו אחד מהתהליכים הראשונים שמערכת ההפעלה מקיימת במטרה לעבור ממצב 16 ביט למצב 32 ביט.
7. Interrupt descriptor table – IDT and hardware communication (PIC) – מימוש פסיקות – יצירת טבלת פסיקות כללית ולאחר מכן התקשרות עם רכיבי החומרה השונים (מקלדת, עכבר, דיסק, טיימר...)
8. FAT16 and disk communication – לאחר מימוש המצבים הקודמים ניתן לבנות מערכת קבצים ולתקשר עם הדיסק הקשיח. נדון בתאוריה של מערכת קבצים מסוג FAT16

**מושגים כלליים מעולם התכנות ומערכות ההפעלה**

מערכת הפעלה - מערכת הפעלה היא תוכנה מערכתית שמנהלת את החומרה, המשאבים והתוכנות של המחשב ומספקת שירותים משותפים לתוכנות המחשב. היא פועלת כמתווכת בין המשתמשים לחומרת המחשב, מאפשרת הפעלת יישומים ומנהלת את משאבי החומרה ביעילות.

Kernel - הליבה (kernel) היא החלק המרכזי של מערכת ההפעלה, אחראית על ניהול משאבי המערכת, הזיכרון, התהליכים ותקשורת החומרה. היא פועלת במצב פריבילגי עם גישה בלתי מוגבלת לכל משאבי המערכת, ומבטיחה שהיישומים והחומרה פועלים בצורה חלקהה ובטוחה.

מערכת קבצים - מערכת קבצים היא הממשק בין הקרנל לבין אמצעי אחסון פיזי כגון הדיסק. המערכת מארגנת ומנהלת נתונים על אמצעי אחסון, מספקת דרך לאחסן, לאחזר ולעדכן קבצים. היא מגדירה כיצד הנתונים מאוחסנים, נקראים ומנוהלים, ומאפשרת ניהול ושליפה יעילה של נתונים על גבי דיסקים ואמצעי אחסון אחרים.

C,C++ - C היא שפת תכנות שפותחה בשנות ה70 וידועה בשל יעילותה והשליטה שלה במשאבי מערכת, מה שהופך אותה למתאימה ופופולרית לכתיבת תוכניות רבות מסוגים שונים (ביניהן דפדפני רשת, קומפיילרים, ומערכות הפעלה) . C++ היא הרחבה של השפה C שפורסמה בשנות ה80. C++ היא מה שנקרא "סופרסט" לC - כלומר היא מכילה את כל המידע שקיים בC ומוסיפה עליו דברים כגון אפשור מחלקות ושדרוג הפונקציות הקיימות בstandard library- std

Assembly – אסמבלי, ובאופן ספציפי אסמבלי למעבדי x86 היא שפת תכנות low level. זוהי שפת התכנות הבסיסית ביותר והקרובה ביותר לשפת מכונה (שהיא רצף אפסים ואחדים שהמחשב יודע לעבד ולעבוד איתם)שבאמצעותה ניתן לגשת בקלות לרכיבים הפיזיים של המחשב (גישה פשוטה לports לדוגמה באמצעות הפקודות in, out)

Compiler – קומפיילר (או "מהדר" בעברית) הוא כלי תוכנה שמתרגם קוד שנכתב בשפה "גבוהה" (C או C++ לדוגמה) לקוד מכונה או לקוד ביניים קודם (שפת אסמבלי). תרגום הקוד מאפשר למחשב להריץ את התוכנית שנכתבה בשפה הגבוהה, שכן המעבד יודע לעבד רק קוד בשפת מכונה ולכן בהתחלה הוא אינו מסוגל להריץ קוד בשפה גבוהה.

Linker – הlinker הוא כלי תוכנה שמטרתו לבצע את השלב שבין קומפילציה של קוד (באמצעות הקומפיילר) לבין הרצת הקוד בפועל. תפקידו הבסיסי ביותר של הלינקר הוא לסמן בצורה נכונה את הסמלים של הקובץ ולהכין אותם לקראת טעינה והרצה. הקבצים, במצבם אחרי שלב הקומפילציה אינם מוכנים להרצה של המחשב ולכן יש צורך בשלב זה. בפועל, מעבר לתפקיד זה, הלינקר פעמים רבות מאחד בין קטעי קוד שונים שנמצאים במקומות אחרים. פעמים רבות בפרויקטים קבצים שונים מופרדים זה מזה אך קוראים זה לזה או משתמשים בספריות חיצונית. הלינקר מאחד את כל הקודים השונים ביחד לכדי קובץ סופי שמוכן לטעינה: הוא מאחד בין כל איזורי הקוד, כל איזורי הנתונים ומכין את כתובות הזיכרון באופן סופי להרצה.

Gnu – Gnu (gnu not unix) הוא פרויקט תוכנה חופשי שנוצר על ידי ריצ'רד סטולמן בשנת 1983 כדי ליצור מערכת הפעלה שלמה דמוית יוניקס שמורכבת מתוכנות חינמיות. פרויקט gnu מכיל כלי תוכנה רבים ביניהם הקומפיילר gcc (בו נשתמש בשביל לקמפל את הקוד שלנו) ועוד רבים אחרים.

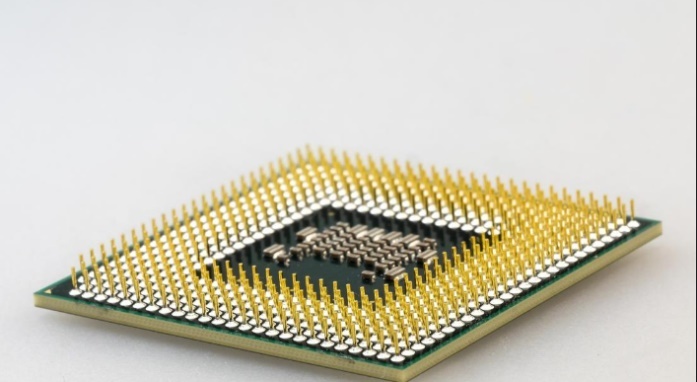
Makefile – makefile הוא קובץ שמכיל הוראות מסוגים שונים (קומפילציה, לינקינג, פתיחת תוכנות וביצוע פעולות בהן). הmakefile זהה בפונקציונליות שלו לשורת פקודה רגילה מטרמינל אך היתרון המשמעותי שלו על פניה הוא שבאמצעות makefile ניתן להפוך תהליכים לאוטומטיים. לדוגמה: במקום לכתוב את פקודות הקומפילציה שוב ושוב עבור כל אחד מקבצי הקוד שלי אני יכול להגדיר פעולה שמקמפלת אותם. כך במקום להריץ אותם בעצמי ולבזבז זמן אפשר לייעל את התהליך. בקצרה, makefile הוא כלי לייעול תהליכים, שכתיבת קוד בלעדיו תהיה קשה יותר באופן משמעותי.

מכונה וירטואלית -

**תאוריה כללית של מערכות הפעלה**

כדי להבין את מערכת ההפעלה שבניתי, ראשית יש צורך להבין מושגים בסיסיים שקשורים לעולם זה. נדון כעת במספר מושגים כאלו וביניהם אזורי זיכרון בקובץ, מעבד, ram, דיסק, תהליך הקומפילציה והלינקינג (linking), סוגי מערכות הפעלה ככלל ובפרט מרחבי זיכרון שונים (16 ביט, 32 ביט, 64 ביט), ואז נראה איך הכל מתחבר ביחד.

מעבד – המעבד (או cpu – central processing unit) הוא הרכיב שמריץ בפועל את כל הפעולות שמתקיימות במערכת ההפעלה שלנו. בין אם זה קריאה מקובץ, שליחת הודעה בווטסאפ או חיפוש בגוגל, כל הפעולות הללו מתבצעות בכך שהמעבד עובר על הקוד ומבצע אותו שורה אחר שורה. כאמור, המעבד יודע לעבוד רק עם קוד מכונה (אפסים ואחדים) ולכן יש צורך בתהליך קומפילציה ולינקינג לפני הרצת הקוד בעזרת המעבד.



Ram – הראם הוא גוף הזיכרון הראשי איתו המעבד עובד במהלך הריצה. כמעט בכל המקרים, המעבד צריך לשמור, לשנות ולקרוא מידע ולכן יש צורך בגוף אחסון אליו המעבד יוכל לגשת כדי לבצע פעולות אלו (דגש חשוב הוא שלמעבד ישנו זיכרון פנימי משלו בשם הרגיסטרים. מדובר ברכיב חשוב במיוחד אך הוא אינו כזה שיוכל לשרת את המעבד לבדו ולכן ישנו צורך מהותי בגוף אחסון כמו הראם). המעבד יודע לגשת לכתובות ספציפיות בזיכרון הראם באמצעות שימוש ברגיסטרים שלו. הראם הוא התקן זיכרון נדיף, כלומר אם נכבה ונדליק את המחשב המידע שהיה בו יימחק.

תמונה שמכילה חשמל, רכיב חשמלי, רכיב מעגל חשמלי, רכיב מחשב

התיאור נוצר באופן אוטומטי

דיסק קשיח – בגלל שהראם נדיף וקטן יחסית (לא יכול להריץ יותר מכמה תוכנות בו זמנית) יש לנו צורך בהתקן זיכרון נוסף ולא נדיף שיכיל את כלל התוכנות שלנו וממנו נטען אותן לראם. הדיסק הקשיח עונה לנו על צורך זה. הדיסק הקשיח הוא התקן זיכרון בגודל עצום (ביחס לראם: אם גודלו של הראם 16GB לדוגמה גודלו הממוצע של הדיסק הוא 1 TB – כלומר בערך פי מיליון יותר) שבו שמורות כל התוכנות שלנו. במידה ואנו רוצים להריץ תוכנה כלשהי, אנחנו טוענים אותה מהדיסק אל הראם ואז היא מורצת על ידי המעבד.

תמונה שמכילה חשמל, לנהוג, התקן לאחסון נתונים, דיסק קשיח

התיאור נוצר באופן אוטומטי

התקני I/O – מערכת הפעלה סטנדרטית חייבת להיות מסוגלת לקלוט קלט מהמשתמש. קלט זה מתבצע באמצעי התקני הI/O (input/output). דוגמה להתקנים כאלו הם המקלדת, העכבר והמיקרופון. שיטת ההתקשרות הסטנדרטית להתקנים אלו היא באמצעות מה שנקרא ports. אלו הם חיבורים פיזיים שמחוברים חשמלית למעבד. בכל פעם שהמשתמש מבצע קלט (באמצעות המקלדת לדוגמה) נשלח אות חשמלי למעבד דרך הport.

תמונה שמכילה שרטוט, תרשים

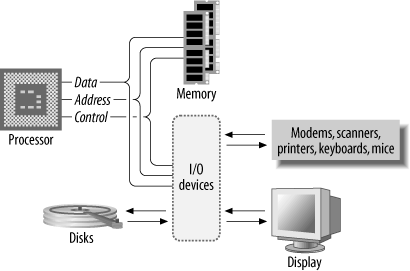
התיאור נוצר באופן אוטומטי

לוח אם – לוח האם הוא הרכיב שמקשר בין כל הגורמים שתוארו עד עכשיו. לוח האם משתמש בחיבורים מוליכים חשמלית (חיבורים אלו נקראים busses - בעברית פסים) כדי להעביר אותות חשמליים שמסמנים קבלת מידע. אם נסתכל על הדוגמה הקודמת עם התקני הI/O – הport שולח אות חשמלי למעבד דרך הport שמחובר חשמלית למעבד באמצעות bus.

תמונה שמכילה חשמל, רכיב חשמלי, מעגל חשמלי, רכיב מעגל חשמלי

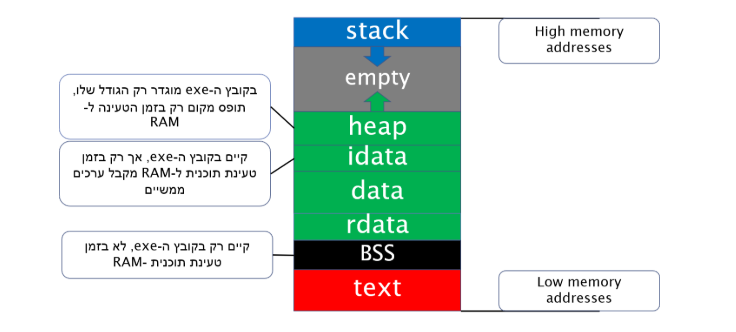
התיאור נוצר באופן אוטומטי

על אף השוני בין מערכות הפעלה שונות כמו לינוקס, ווינדוס, macOS כולן מורכבות בסופו של דבר מאותן הרכיבים ואם נסתכל עליהן ברמת הממשק עם רכיבי החומרה נמצא הרבה קווי דמיון. נתחיל בלהסתכל על סרטוט שמראה את ארכיטקטורת המחשב וכיצד רכיבי החומרה מחוברים זה לזה:



ניתן לראות את הרכיבים השונים בהם דנו. אפשר לראות שהמעבד (processor) מחובר באמצעות busses לזיכרון (ram) וגם להתקני הI/O שביניהם המקלדת, עכבר, מסך ודיסק. סגנון עיצוב זה נקרא ארכיטקטורת פון נוימן (שפותחה על ידי המתמטיקאי ג'ון פון נוימן בשנות ה40) והוא מתקיים בכל אחת ממערכות ההפעלה שקיימות כיום בשוק.

נעבור לדבר על מבנה הזיכרון של תוכנה ועל אזורי הזיכרון שזה (איזור הקוד, איזור הנתונים, איזור הסטאק):  
כאשר אנו כותבים קוד ושומרים אותו קובץ הקוד מקבל צורה ייחודית. הקובץ מכיל מספר איזורים שונים (sections) שלכל אחד מהם יש ייעוד אחר. הנה דיאגרמה שמראה כיצד נראה קובץ קוד לאחר שנשמר



(קרדיט ל<https://data.cyber.org.il/os/os_book.pdf> – עמוד 31)

אפשר להבחין פה בין מספר איזורים שונים: text, bss, rdata, heap, stack. לכל אחד מהאיזורים האלה תפקיד שונה וייחודי. לדוגמה, איזור הtext מכיל את ההוראות עצמן – את הקוד שהמעבד צריך לבצע. איזור הbss מכיל מקום פנוי שמוגדר עבור משתנים שאינם מאותחלים עם ערכים, איזור הrdata מציג את המידע שמוגדר כקריא בלבד וכן הלאה.

איזור חשוב שעלינו להתמקד בו הוא איזור הstack (כשנסיים עם החלק התאורטי ונגיע לקוד נראה שאנו מגדירים בו stack שתשמש את הקרנל שלנו). איזור הstack (מחסנית בעברית) משמש את הקובץ בשביל שמירת משתנים מקומיים של פונקציות והעברת פרמטרים לפונקציות. איזור המחסנית עובד כמו מחסנית בחיים האמיתיים ואנו עובדים איתה באמצעות הפקודות push,pop כלומר הכנסה והוצאה של ערך מהמחסנית.

המחסנית עובדת בשיטת LIFO (last in first out) כלומר הערך האחרון שנכניס אליה יהיה הראשון שיצא, בדיוק כמו במחסנית אמיתית.

**יצירת סביבת עבודה**

לפני שניגש לתאוריה של בניית מערכת הפעלה משלנו אנחנו צריכים להחליט באיזה פורמט נריץ את הקוד שלנו ובאיזו סביבות נפתח ונריץ אותו.

נתחיל בסביבת הפיתוח: כתבתי את הקוד לפרויקט בתוך מכונה וירטואלית linux mint שמורצת דרך התוכנה vmware. Mint היא הפצה מבוססת ubuntu, שהיא בין ההפצות המפורסמות ביותר למערכות לינוקס. החלטתי לעבוד בסביבת לינוקס ולא בסביבת ווינדוס (כמו המחשב הביתי שלי) ממספר סיבות:

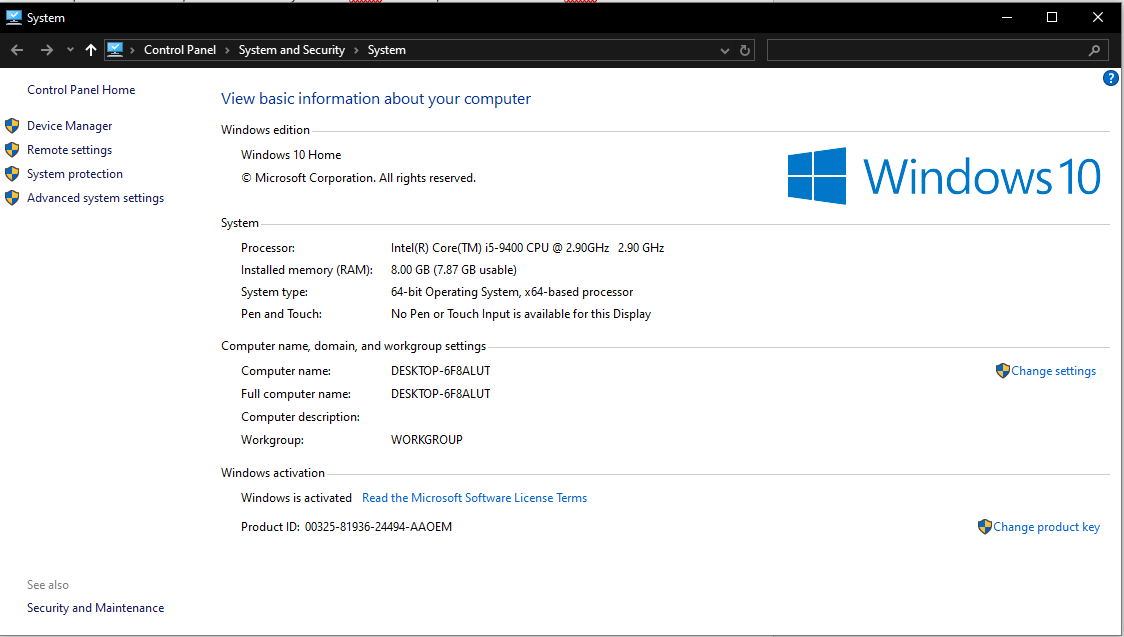
1. גישה נוחה לכלים רבים שיעזרו לי במהלך הפרויקט (אפשרויות ניתוח קבצים מגוונות באמצעות פקודות משורת הפקודה (fdisk, disk mounting נוח, ופקודות שימושיות אחרות כגון hexdump))
2. רצון להתנסות בפעם הראשונה עם מערכת ההפעלה לינוקס ולצבור ניסיון איתה.

לסביבת הפיתוח הזו (vmware) יש חיסרון אחד משמעותי והוא כמות הראם הגדולה שהיא צורכת. למחשב הביתי שלי יש 8 GB והמכונה הוירטואלית לבדה צורכת 4 GB. דבר זה הקשה עליי להשתמש ביותר מדי תוכנות במחשב במקביל, ושכן הצלחתי זה פגע משמעותית בביצועים של המחשב.

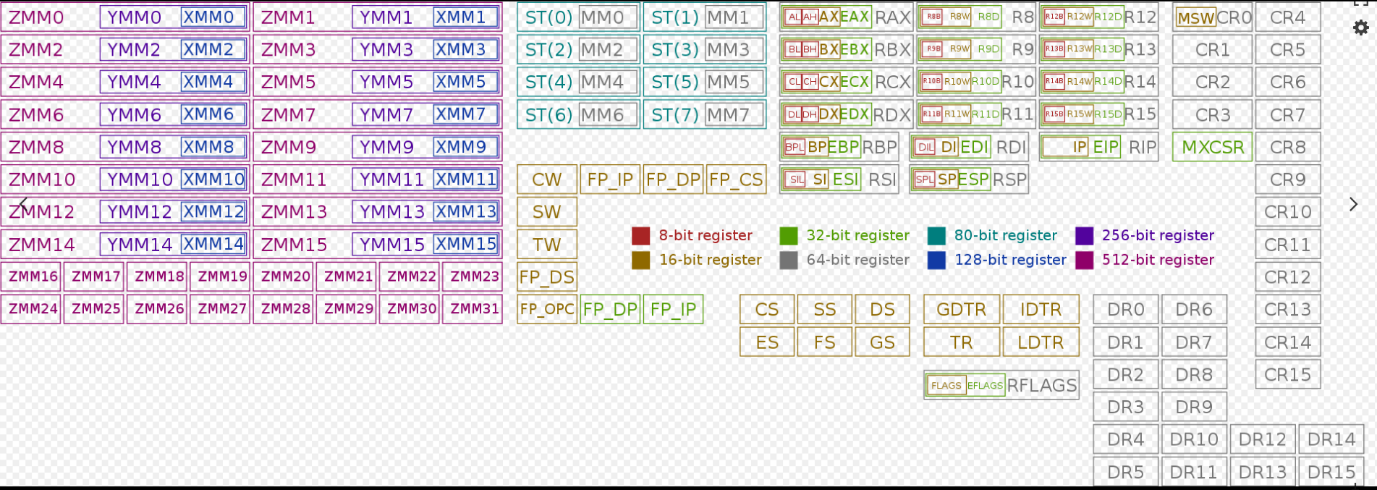
כדי להריץ את הקוד שלי אני משתמש בתוכנה qemu. Qemu היא אוסף של כלים שמאפשרת לנו להריץ מכונות וירטואליות באמצעות שורת הפקודה בקלות. בכך שנוסיף פרמטרים להוראות של שורת הפקודה נוכל להגדיר התנהגות שונה למכונה הוירטואלית בכל פעם ואף לטעון אליה רכיבי חומרה שונים כמו הדיסק הקשיח או לשנות את גודל הראם שהמכונה עובדת איתו (בפועל רכיבים אלו מסומלצים למערכת ההפעלה אך אין לנו מה לדאוג לגבי זה כיוון שqemu מטפל בזה). Qemu הוא כלי שמתועד היטב, ומעבר לכך הוא נפוץ מאוד לשימוש בקרב מפתחי מערכות הפעלה, כך שהייתי מסוגל לשאול שאלות לגביו כשהייתי זקוק לעזרה.

נדבר כעת על הפורמט בו נריץ את הקוד.

כיום, הרוב המוחלט של המחשבים המודרניים עובדים במצב זיכרון שנקרא 64 bit protected mode. מה זה אומר? ניכנס מאוחר יותר למשמעות של protected mode אבל נדון כעת במשמעות של 64 bit. אם תלכו למחשב שלכם, This pc -> properties סביר להניח שתראו שליד הקטגוריה system type יהיה כתוב 64 bit operating system, x64 based processor. כך זה נראה אצלי במחשב הביתי:



בחלק הקודם של התאוריה דיברנו על הראם וכיצד המעבד מתקשר איתו. כאמור, כדי לגשת לכתובות זיכרון בראם המעבד משתמש ברגיסטרים שלו כדי לבטא את הכתובת. ככל שעבר הזמן, המעבדים השתדרגו וגודל הרגיסטרים הפנימיים שלהם הלך וגדל. הרגיסטרים היו בגודל 16 ביט בתחילת דרכם ואז גדלו בהדרגה ל32 ביט, 64 ביט (מחשב ביתי מודרני) ואף המשיכו לגדול עד ל256 ביטים (רק רגיסטרים ספציפיים, אלו הבסיסיים מגיעים עד לגודל של 64 ביט). באופן טבעי, ככל שגודל הרגיסטרים יהיה גדול יותר כך מרחב הכתובות אליו יוכלו לגשת יהיה גם הוא גדול יותר. בדיאגרמה הבאה ניתן לראות את הרגיסטרים השונים (ואת הגרסאות שהם עברו) שקיימים במחשב מודרני ואת הגדלים של כל אחד מהם.



(קרדיט ל <https://en.wikipedia.org/wiki/X86>-)

את מערכת ההפעלה שלי החלטתי לבנות במצב 32 ביט מהסיבה שתהליך המעבר מ32 ביט ל64 ביט (נקרא גם מעבר לlong mode) הוא מורכב ולא משרת את הצרכים שלי. המטרה שלי היא לבנות מערכת הפעלה קטנה לשימוש אישי, שלא צורכת הרבה משאבים או מפעילה תוכנות כבדות. מהסיבה הזו אין צורך להגדיל את מרחב הכתובות, כיוון שבכל מקרה לא נשתמש בו (גם מרחב כתובות של 32 ביט הוא הרבה מהדרוש ולכן בהתחלה חשבתי על לעבוד במצב 16 ביט שנקרא גם real mode).

ההחלטה שלי להריץ מערכת הפעלה ב32 ביט יוצרת בעיה קטנה. כשאנחנו עובדים על המחשבים שלנו, שהם במצב 64 ביט, אנחנו גם משתמשים בקומפיילרים שמייצאים קוד בפורמט 64 ביט. כאמור, זה לא קוד מהסוג שאנחנו רוצים למערכת שלנו. למרבה המזל יש דרך פשוטה לפתור את הבעיה הזו והיא באמצעות מושג שנקרא דגלי קומפילציה. דגלי הקומפילציה הם הוראות מיוחדות שאפשר לתת לקומפיילר (gcc במקרה שלנו) כדי להכתיב פעולות מסויימות. לדוגמה, אם נקמפל באמצעות gcc ללא שום דגלי קומפילציה נקבל קוד בפורמט 64 ביט. לעומת זאת אם נשתמש בדגל הקומפילציה -m32 אנחנו למעשה אומרים לקומפיילר שיכין את קוד התוצר בפורמט 32 ביט, במקום ה64 שהוא היה אמור. בנוסף לדגל הקומפילציה -m32 אני משתמש במספר דגלים נוספים שמשמשים למטרות שונות. נעבור עליהם כעת:

בתוך קובץ הmakefile מפורטים דגלי הקומפילציה תחת המשתנה GPPPARAMS



נעבור כעת על הדגלים.   
-I./include – לפרט לקומפיילר שישנה תיקייה בשם include שבה נמצאים קבצי הheader שלי

-m32 כאמור, לפרט לקומפיילר להוציא פלט שמתאים להרצה בסביבת 32 ביט

-fno-use-cxa-atexit

-nostdlib מפרט לקומפיילר לא להשתמש בstandard library ולעשות לינקינג עם הקבצים שבתוכה. לפעמים, הקומפיילר יכול להחליט להוסיף פונקציה כזו או אחרת מהספריה הזאת. הבעיה עם שימוש בספריית standard library היא שפונקציות של ספרייה זו עושות שימוש בsyscalls. במילים אחרות, הן משתמשות במשאבים ובחומרים שקיימים רק במערכת הפעלה מבוססת כמו זה שאנו עובדים עליה ולכן אסור להשתמש בספריה זו.

-fno-builtin מפרט לקומפיילר שלא להוסיף פונקציות משלו (built in) לקוד שהוא מקמפל. כמו standard library, פונקציות אלו עושות שימוש בsyscalls ובמשאבי מערכת שמערכת ההפעלה שלנו לא יודעת לספק.

-fno-rtti מפרט לקומפיילר שלא יוסיף מידע על הקוד במהלך ההרצה (no run time type information). בדרך כלל משתמשים בדגל זה במטרה להפוך את הקוד שלנו לכמה שיותר יעיל ומהיר.

-fno-exceptions מפרט לקומפיילר שלא להוסיף תמיכה בבקרת שגיאות באמצעות פקודות try, catch. כמו הדגל הקודם, אין לנו צורך מהותי בתהליכים האלה של הקומפיילר ולכן אנחנו מבטלים אותם כדי שהקוד שלנו יהיה כמה שיותר מהיר.

-fno-leading-underscore דגל זה מפרט לקומפיילר שלא להוסיף מקף תחתון (\_) במהלך קמפול הקוד. דגל זה חשוב במיוחד כיוון שאנחנו מבצעים הרבה קריאות בין קבצים (קובץ אחד קורא לפונקציה שבקובץ אחר) – אם הקומפיילר ישנה את השמות של הסמלים שלנו, לא נוכל לקרוא להם בצורה נכונה.

-fpermissive דגל זה הופך חלק מהשגיאות של הקומפיילר לאזהרות ולכן מונע מהן להקריס לנו את הקוד – בעצם מבקשים מהקומפיילר שיהיה יותר "רחום".

-g מפרט לקומפיילר להוסיף מידע וסמלים לקובץ המקומפל במטרה שנוכל לדבג אותו באמצעות הדיבגר (gdb – נדון בו אחר כך).

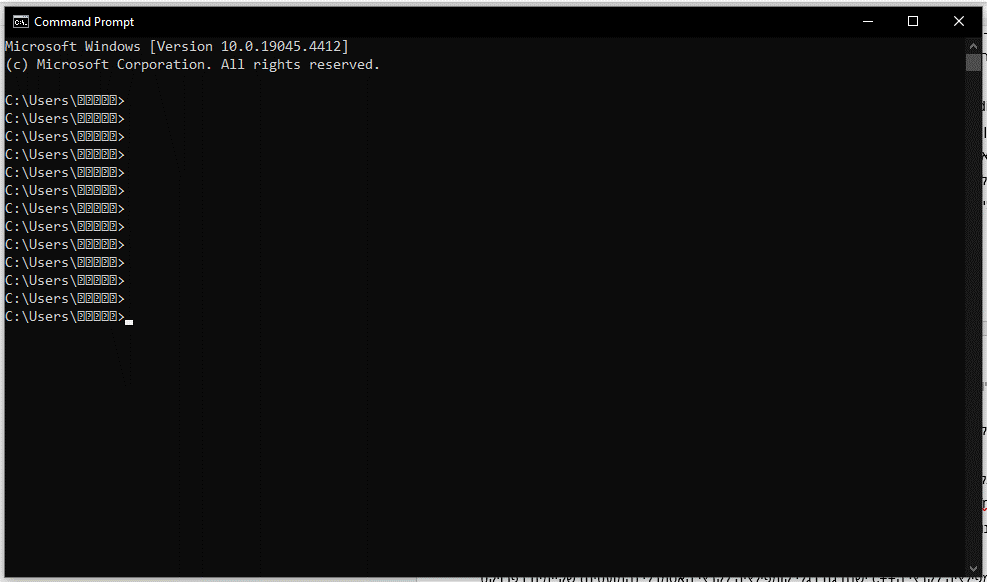
-nostdlibc++ דגל זה מפרט לקומפיילר שלא לחבר לקוד קבצי header של ספריות c++. כמו עם דגלי ה-fno-stdlib ו-fno-builtin יש קבצים שלא נוכל לצרף לקוד שלנו כיוון שהם משתמשים במשאבים ופונקציות שלא נוכל לספק (תאורטית נוכל לממש אותם בעצמנו אבל 1) זה ממש קשה 2) אנחנו לא צריכים את הפונקציות האלה בכל מקרה).

בנוסף לדגלי הקומפילציה לקבצי הc++ ישנם גם דגלי קומפילציה לקבצי האסמבלי (המועטים) שקיימים בפרויקט. הדגלים האלו זהים לאלו שראינו ולכן לא אפרט אותם בשנית.



**סגנון הבנייה של מערכת ההפעלה**

לאחר שהכנו את סביבת הפיתוח וההרצה אנו יכולים להגיש לעיצוב הרעיוני של המערכת. אני החלטתי לממש ממשק משתמש טקסטואלי פשוט בדומה לשורת הפקודה בווינדוס (cmd) או הטרמינל של מערכת לינוקס. במצב זה כל פקודה מובחנת אחת מהשנייה באמצעות < ובנוסף מודפס הdirectory הנוכחי בו אנו נמצאים (מתקשר למערכת קבצים בה נעסוק מאוחר יותר):



בנוסף, אופי הקוד מבוסס על עיצוב בשם monolithic kernel (ליבה שעשויה מיחידה אחת). עיצוב זה קובע שקוד מערכת ההפעלה מורכב מקובץ אחד גדול שהוא תוצאת הlinking של כל קובצי הקוד שלנו לקובץ אחד (אנחנו מממשים עיקרון זה בהוראת mykernel.bin שבקובץ הmakefile – יש לנו קובץ לינקר כללי שמקבל כקלט את כל קבצי הקוד. הסקריפט מאחד את כל אזורי הקוד לאזור אחד, את כל אזורי המידע לאזור אחד וגם את שאר האיזורים המיוחדים של הקבצים האינדיבדואליים – bss לדוגמה).

**Booting**

בחלק זה של הרקע התאורטי נתחיל לדבר על תאוריה ממשית של בניית מערכת הפעלה ונעסוק בחלקי התאוריה לפי הסדר שבו דברים מתרחשים בפועל כשמערכת ההפעלה נדלקת. לפיכך, הדבר הראשון בו נדון הוא מה התהליך שמתרחש כאשר מערכת ההפעלה נדלקת (booting process).

ראשית, כאשר אנו מדליקים את המחשב שלנו (או מכונה וירטואלית לצורך העניין) מוחל זרם חשמלי שמגיע ללוח האם. לוח האם מכיל רכיב מיוחד שהמידע שבו אינו נמחק גם בהיעדר זרם (רכיב מסוג EEPROM לרוב). ברכיב זה שמור קוד בשם BIOS (basic input output). זהו קוד מיוחד שנטען לראם עם עליית המערכת. לקוד זה יש שתי מטרות חשובות:

1. לפני ההתחלה של כל דבר האחר, הbios מבצע בדיקות לרכיבים הפיזיים איתם אנו מתקשרים במהלך פעילות המערכת: הbios בודק גישה לראם, לכרטיס הגרפי, ולהתקני החומרה כמו המקלדת והעכבר. בדיקה זו נקראת POST (power on self test) ובמקרים מסויימים אף ניתן לשמוע במהלכה צפצופים או לראות אורות כאשר נבדקים הרכיבים השונים.
2. לאחר שpost סיים לרוץ ווידא שהרכיבים הפיזיים של המחשב תקינים ומגיבים יש צורך לטעון את קוד מערכת ההפעלה. כדי לבצע זאת, הbios עובר על גופי הזיכרון של המחשב (דיסק קשיח, דיסק אופטי, ssd וכן הלאה) ומחפש בסקטור הראשון שלהם (סקטור הוא היחידה הכי קטנה של זיכרון אליה ניתן לגשת בדיסק הקשיח, הסקטור הראשון של הדיסק הוא 512 הבתים הראשונים שלו) קוד בעל חתימה תקינה (ליתר דיוק, הוא מחפש סקטורים שנגמרים בחתימה 0xaa55). ברגע שהוא מוצא אחד כזה הוא טוען אותו לראם ומסיים את תפקידו (טוען את 512 הבתים הראשונים של הדיסק). הקוד שנטען זה עתה לראם נקרא bootloader ותפקידו לטעון את מערכת ההפעלה מהדיסק ממנו בוצעה הקריאה. מתוך 512 הבתים שנטענו לראם רק 446 הם קוד בפועל. 64 בתים נוספים משמשים עבור טבלה שנקראת partition table והיא מגדירה את המחיצות השונות שמוגדרות על הדיסק (2 הבתים האחרונים הם החתימה 0xaa55). באמצעות הpartition table, קוד הbootloader מוצא את המיקום של הקרנל שלנו וטוען אותו לראם. כעת תהליך הטעינה של מערכת ההפעלה הסתיים והקוד שלנו נכנס לפעולה.

(במערכת ההפעלה שלי אני משתמש בbootloader בשם GRUB. זהו כלי מוכר מאוד ומתועד שחוסך לנו כתיבת bootloader משלנו.)

**מצבי זיכרון והכנת המערכת**

כאמור, בסוף תהליך הbooting מערכת ההפעלה שלנו קיבלה שליטה לאחר שנטענה מהדיסק קשיח באמצעות הbootloader. (או גוף זיכרון אחר - במקרה שלנו גוף הזיכרון cdrom הוא היכן שנשמר הקוד שלנו(

כפי שפרטתי קודם, בחרתי לפתח ולהריץ את המערכת שלי בסביבת עבודה 32 ביט protected mode. לכן, לפני שנוכל להריץ תהליכים רגילים (לקיים ממשק משתמש) עלינו להכין את מערכת ההפעלה שתעבוד במצב זה. כדי להשיג מטרה זו יש צורך לבצע מספר פעולות בסדר הבא:

1. מימוש gdt – global descriptor table
2. מימוש idt – interrupt descriptor table
3. הגדרת isr – interrupt service routine
4. הגדרת תקשורת עם הPIC – programmable interrupt chip
5. מימוש ממשק עם הדיסק הקשיח – ATA PIO interface

לאחר שנבצע פעולות אלו נוכל לקיים את פונקציות וממשק משתמש.

נתחיל בלדון בפעולה הראשונה ובתאוריה שמאחוריה בחלק זה ובחלקים הבאים נדון בשאר הפעולות.

* 1. מוצרים קיימים

לפרט על כמה מהמוצרים הקיימים בשוק, כמה שורות על כל מוצר. לסכם בטבלה:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | הפרויקט שלי | מערכת הפעלה windows 10 | מערכת הפעלה linux mint | מערכת הפעלה msdos 8086 |
| מרחב כתובות – 32 ביט לעומת 64 ביט | עבודה במרחב כתובות 32 ביט | עבודה במרחב כתובות 64 ביט | עבודה במרחב כתובות 64 ביט | עבודה במרחב כתובות 16 ביט |
| Userland & kernelspace | מימוש מרחב קרנל בלבד | מימוש מרחב קרנל ומרחב משתמש | מימוש מרחב קרנל ומשתמש | מימוש מרחב קרנל ומשתמש |
| שפת תכנות | C++, assembly | C, assembly | C, assembly | assembly |
| Booting process | שימוש במנגנון GRUB- כלי קיים לטעינת קוד של מערכת הפעלה | ווינדוס מממשת את הbootloader של עצמה | לינוקס משתמשת במנגנון GRUB | Ms-dos מממשת את הbootloader של עצמה |

1. **תוצר סופי**
   1. תיאור הפרויקט

התוצר הסופי הוא מערכת הפעלה מתפקדת שלאחר שעלתה וביצעה תהליכי booting ואתחולי זיכרון מסוגלת לספק ממשק טקסטואלי למשתמש. המשתמש יכול להשתמש במערכת הקבצים שמערכת ההפעלה מממשת – בסגנון FAT16. המשתמש יוכל לקרוא קבצים קיימים, ליצור קבצים חדשים, וליצור תיקיות ותתי תיקיו.

רכיבי המערכת העיקריים:

* פירוט
* על
* כל
* רכיב
  1. אלגוריתמים עיקריים
  2. דרישות ומגבלות מערכת

למערכת מספר דרישות ומגבלות.

**בהיבט הדרישות:**

* לפרט על דרישות כמו שלמות, מהירות, תמיכה בסוגי פורמטים שונים...

בהיבט המגבלות:

* היבטים כמו תמיכה רק במערכות הפעלה מסוימות, רק בכמות משתמשים מסוימת וכו'. לפרט על כל מגבלה.
  1. התייחסות לנושא אבטחה
  2. ממשק משתמש

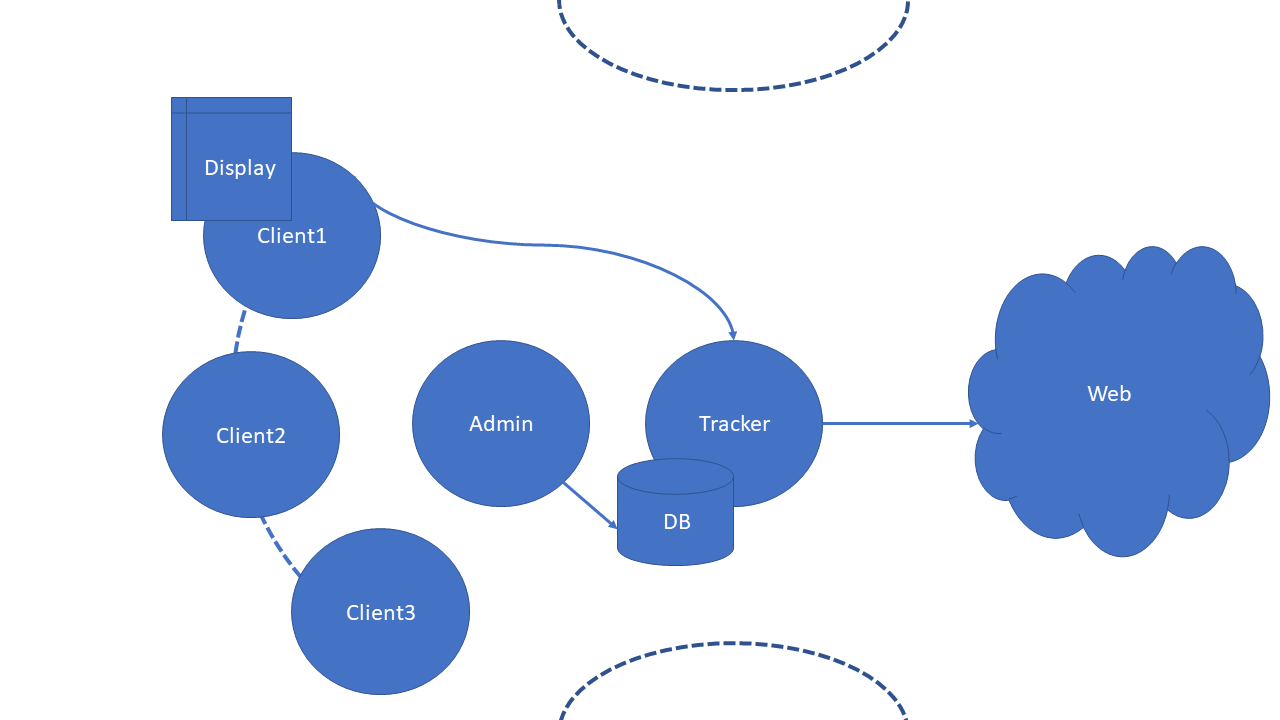
צילום תרשימי מסך של כל אחד מהחלונות לפי הלוגיקה והסדר של התהליך בפרויקט. הסבר קצר על כל חלון.

* 1. תרחישים עיקריים

תרשים זרימה לכל אחד מהתהליכים

1. **תהליך כתיבת הפרויקט**
   1. תהליך הפרויקט
   2. אתגרים ואופציות שונות למימוש
2. **מרכיבי פתרון**
   1. תיחום הפרויקט

* תקשורת –
* מערכות הפעלה –
* תצוגה –
* אבטחה –
* מבנה נתונים –
* שימוש במסד נתונים
* ארכיטקטורת קוד –
* תיעוד –
  1. סביבת העבודה (טכנולוגיה)
* שפות התכנות: C++, assembly, shell scripts (bash), makefile
* סביבת פיתוח: linux mint 64 bit virtual machine (vmware), qemu virtual machine
  1. מבט טופולוגי

המערכת מורכבת משרת ראשי, מ-Clients ומ-Admin.

Client1, Client2, Client3 מייצגים עמיתים ברשת הפנימית, שמשתפים pieces זה עם זה.

Client1 ראה שחסר לו piece ושהוא איננו נמצא ברשת הפנימית, לכן הוא פונה לרשת החיצונית לצורך השגת ה-piece החסר.

כל העת ה-Admin ניגש למסד הנתונים המשותף לו ולשרת הניהול ומעדכן את הסטטיסטיקות.

* 1. מבנה נתונים

:Tracker

* admin\_ips – רשימה של כל ה-admins המחוברים
* not\_listening – רשימה של העמיתים אליהם לא יאזין שרת הניהול, עמיתים יוספו למבנה נתונים זה כשהם מעלים קובץ לשרת הניהול ויוסרו בסיום ההעלאה
* requests – רשימה של כמות הפניות לשרת הניהול ומילון של המשתמשים שפנו וכמה פניות ביצעו

User:

* Pieces – מילון של ה-pieces ואילו עמיתים מחזיקים ב-pieces אלו
* peer\_list – רשימה של האובייקטים של העמיתים השונים (לצורך הורדה מהם)
* peer\_thread – מילון של ה-threads הפתוחים ואובייקט העמית אליו ממנו מורד piece ב-thread זה
* currently\_connected – עמיתים אליהם מחובר משתמש המערכת בעת ההורדה
* peers – רשימה של כל העמיתים שהושגו מהשרתי הניהול השונים ברשת
* announce\_list – שרתי הניהול הזמינים (במקרה של קובץ גלובלי) או העמיתים הזמינים (במקרה של קובץ מקומי)

Admin:

* ip\_files – רשימה שמכילה בתוכה את כל העמיתים שלא שיתפו בטווח זמן נתון מרגע שנוספו, שולחת לשרת הניהול הודעה להסרתם ומאפסת את הרשימה
  1. מסד נתונים
  2. מבט מודולרי
  3. פירוט מודלים עיקריים

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Input\Output | Function | Class |
|  |  |  |  |
|  |  |  |  |

1. **תסריטי בדיקה**
   1. דגשים בבדיקה
   2. תסריטי בדיקה עיקריים
2. **רפלקציה**

פרויקט הסייבר השנה היה חוויה מעניינת ומחנכת בשבילי. למדתי המון דברים חדשים, תאורטיים (חומר של מערכות הפעלה), מעשיים (כתיבת קוד בc++, אסמבלי) וגם אישיותיים. אני מרגיש שהשנתי וקידמתי את ידע הסייבר שלי במידה רבה, על אף ששאפתי לקדם אותו אף יותר.

האתגרים המרכזיים בהם נתקלתי במהלך הפרויקט היו:

1. למידת החומר ויישומו בקוד: לקח לי המון זמן להבין קודם כל את החומר הכללי של מערכות הפעלה (לקרוא את כל הספר של מערכות הפעלה של גבהים – ללמוד על נושאים כגון מבני זיכרון, הראם, מעבדים, סטאק, זיכרון דינמי וכו) ואז גם להבין את החומר הספציפי שדרוש לפיתוח מערכת הפעלה משלי (מימוש כל תהליכי הזיכרון כמו gdt, idt, isr, irq, ata). רוב החומר היה בצורה כתובה שלעיתים מנוסחת רע ומבלבל (בעיקר בosdev) מה שגרם ללמידה להיות מאוד מתאגרת. פעמים רבות הדוגמאות שנתנו הן חלקיות במקרה הטוב ומקשות מאוד על הבנה מלאה של הנושא. דבר זה גרם לי פעמים רבות לאבד מוטיבציה והקשה עליי מאוד לקדם את עשיית הפרוייקט – שמורכבת בחלקה הגדול מלמידה וקריאה.
2. ניהול זמנים נכון שמאפשר עבודה טובה: מעבר לפרוייקט הסייבר היו במהלך השנה המון עומסיםו וקשיים אחרים (בית ספר, מלחמה) שמנעו ממני למצוא זמן לשבת ולהתרכז בפרויקט. בפועל, אני יודע שיכולתי למצוא הרבה זמן אם הייתי מתאמץ ולומד כמו שצריך אבל דברים אלו כן הקשו עליי ובלעדיהם אני מאמין שהייתי עובד יותר.
3. **הוראות התקנה ותפעול**

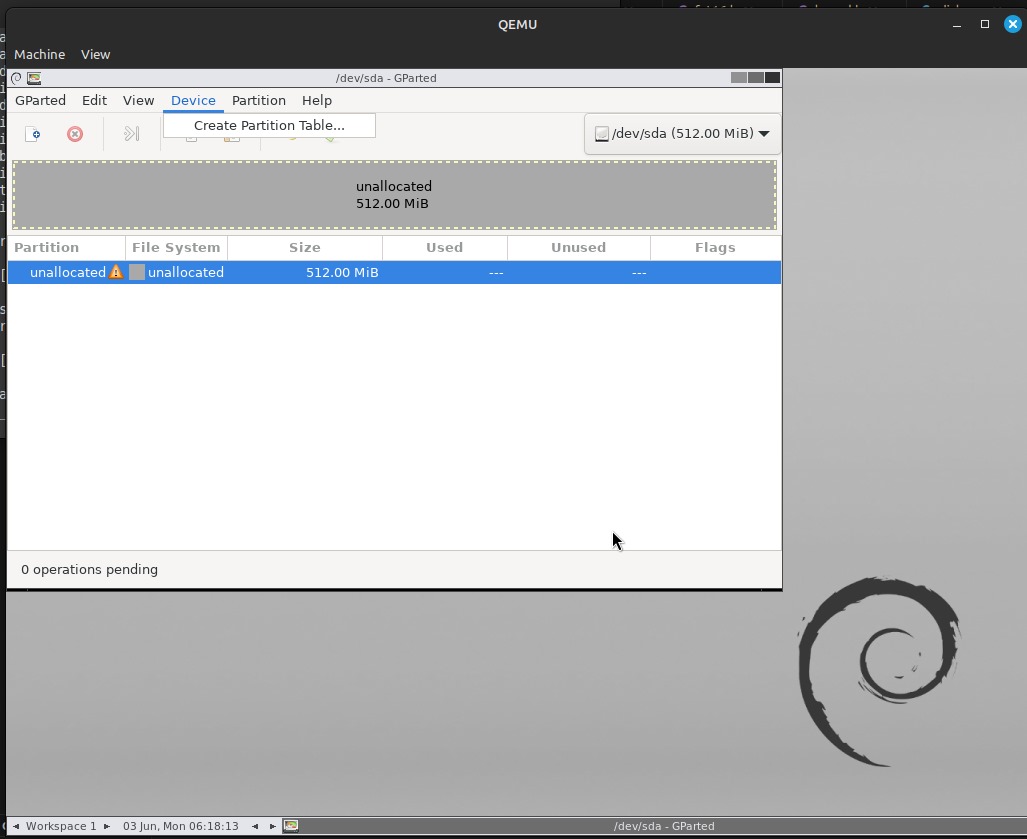
מומלץ לעבוד על מכונת לינוקס מינט (mint) במידה ואתם רוצים לקמפל את הקוד בעצמכם.   
**על לינוקס מינט:**

כדי להריץ את הקוד מנקודת אפס (קומפילציה בשלמותה, יצירת קובץ iso, יצירת דיסק וירטואלי חדש ללא מידע עליו כלל):

1. במכונת לינוקס מינט: התקינו gcc, התקינו make, התקינו qemu, הוסיפו את התוכן שהורד לenviroment variables - path   
   sudo apt-get install gcc

Sudo apt install make  
sudo apt install qemu qemu-utils qemu-system-x86

1. הריצו את פקודה make all שנמצאת בקובץ הmakefile  
   במכונה הוירטואלית GParted שנפתחת פרמטו את הכונן הקשיח os-disk.qcow2 כך שיהיה בעל partition אחד שמוגדר לעבור כfat16 (כשתריצו את המכונה הוירטואלית Gparted ייפתח לכם אוטומטית menu של התכונה Gparted – לחצו שם על יצירת partition ואז על הפרמוט שלו כfat16.

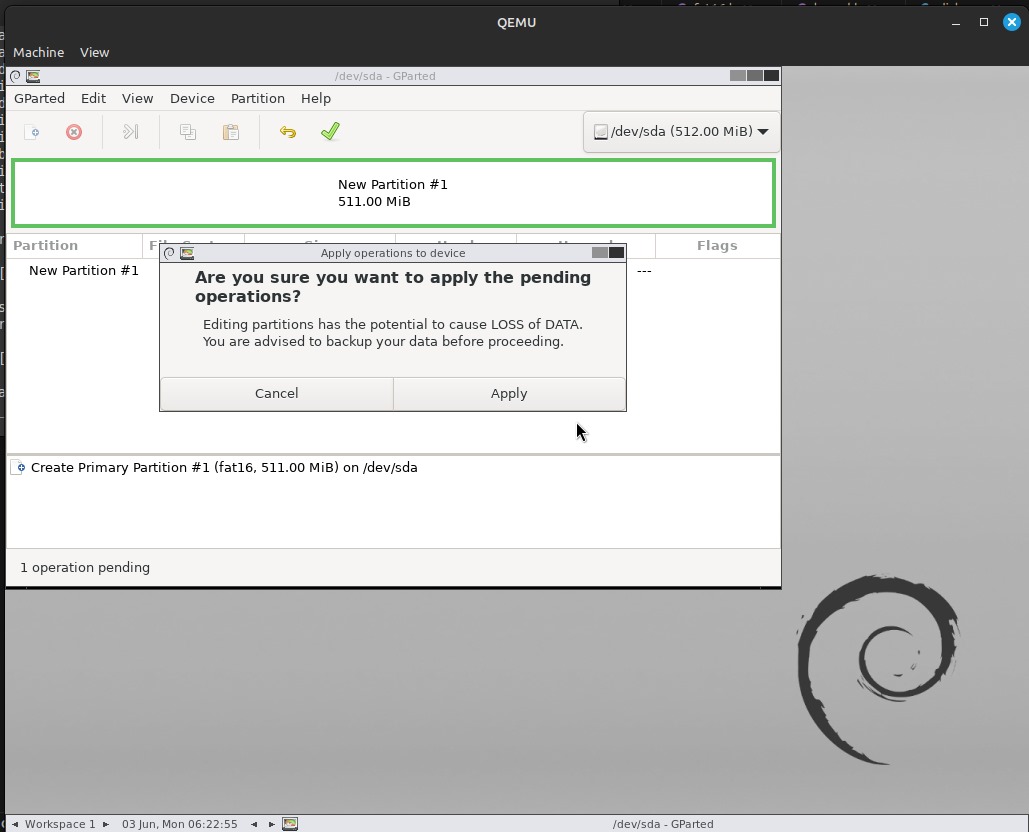


תמונה שמכילה טקסט, צילום מסך, תוכנה, סמל מחשב

התיאור נוצר באופן אוטומטי

תמונה שמכילה טקסט, צילום מסך, תוכנה, סמל מחשב

התיאור נוצר באופן אוטומטי



1. סגרו את GParted באמצעות כפתור הexit
2. המכונה הוירטואלית עם קובץ הiso תיפתח – אתם חופשיים להשתמש במערכת ההפעלה!

**על ווינדוס:**

1. התקינו wsl, אישית אני ממליץ על הפצת ubuntu (הסינטקס של הפקודות הוא בubuntu – אם תשמשו בהפצה אחרת כנראה תצטרכו פקודה אחרת בשביל להתקין את הכלים)
2. בתוך משתמש הwsl התקינו את הכלים gcc, qemu, make באמצעות הפקודה   
   sudo apt-get install gcc make qemu qemu-utils qemu-system-x86
3. הריצו את פקודת הmakefile באותו האופן: make all
4. פרמטו את הכונן הקשיח באותו האופן
   1. התקנה
5. **ביבליוגרפיה**

חומרי הסבר על c++ ועל תאורית מערכות הפעלה כללית (stack, heap...)

<https://www.youtube.com/watch?v=2ybLD6_2gKM>

<https://www.youtube.com/watch?v=TGwl3tJYFRg> - סרטון הסבר קצר על inline in c++ ועל צורת העבודה שלה.

<https://www.youtube.com/watch?v=_8-ht2AKyH4>

השגתי את הידע הראשוני והכללי שלי בנושא מערכות הפעלה באמצעות הספר של גבהים בנושא מערכות הפעלה: <https://data.cyber.org.il/os/os_book.pdf>

<https://data.cyber.org.il/assembly/assembly_book.pdf>

בנוסף נעזרתי רבות באתר osdev בשביל כל אחד משלבי הפיתוח שלי:

<https://stackoverflow.com/questions/172587/what-is-the-difference-between-g-and-gcc> - הבדל בין הקומפיילר g++ לקומפיילר gcc. Gcc = gnu c compiler, g++ - gnu c++ compile. כיוון שאנחנו ממשים בקוד שלנו מחלקות אין לנו ברירה אלא להשתמש בקומפיילר g++ (זכרו כי שפת C לא מכילה תמיכה במחלקות, מה שקיים רק בc++).

<https://wiki.osdev.org/GCC_Cross-Compiler> - עוסק ביצירת cross compiler וביצירת סביבת עבודה נכונה שתאפשר קמפול קוד (אנו עובדים בסביבת עבודה 64 ביט אך צריכים ליצור קוד שירוץ בסביבת 32 ביט, ובנוסף אנחנו לא יכולים להשתמש בספריות מסויימות של c++ כייוון שהן משתמשות באופן מובנה בsyscalls – שלא נוכל לממש)

<https://wiki.osdev.org/What_Order_Should_I_Make_Things_In%3F> – מספק רעיון כללי לסוג העיצוב של מערכת ההפעלה שאני החלטתי לעקוב אחריו - lino commando ספציפית

<https://wiki.osdev.org/BIOS> - תהליך הbios שמתרחש בזמן עליית המחשב

<https://www.youtube.com/watch?v=1T26DpuKnVs> – סרטון יוטיוב שמסביר על תהליך הbooting, MBR, partitions.

<https://wiki.osdev.org/Inline_Assembly> - כפי שראינו, במקרים ספציפיים בקוד, במקום להשתמש באמסבלי באופן ישיר אנחנו יכולים להיעזר בפיצ'ר מיוחד של c,c++ שנקרא inline assembly – שמקצר לנו משמעותית את הקוד ומאפשר לנו לאחד בין קודם שהיו אחרת נפרדים.

<https://wiki.osdev.org/Bare_Bones> - מושג ראשוני על איך ליצור קוד התחלתי בפועל ולגרום לו לרוץ

<https://wiki.osdev.org/Meaty_Skeleton> - מאמר המשך למאמר הקודם

<https://wiki.osdev.org/Boot_Sequence> - עוסק בתהליך הbooting של המחשב ומסביר לעומק את הדרך שבה אפשר לקשר בין הbootloader לבין הkernel. בפרויקט זה, בחרנו במה שנקרא במאמר ב"easy way out" – כלומר השתמשנו בכלי קיים (GRUB) כדי שיבצע בשבילנו את תהליך הbooting

<https://wiki.osdev.org/Global_Descriptor_Table>

<https://wiki.osdev.org/GDT_Tutorial> - שני מאמרים שמסבירים 1) על טבלת הgdt עצמה שמחזיקה את רשומות הסגמנטים השונים 2) תהליך טעינת הטבלה לרגיסטר gdtr שמהווה מצביע לטבלה זו

<https://wiki.osdev.org/Interrupts> - חיבור הקרנל לרכיב הPIC שמהווה גורם מתווך בין המעבד לבין רכיבי החומרה (מקלדת, טיימר, עכבר)

<https://wiki.osdev.org/Interrupt_Descriptor_Table> - מימוש idt - interrupt descriptor table. זוהי כאמור הטבלה שמחזיקה את ההגדורת עבור כל אחת מהinterrupts השונים

<https://wiki.osdev.org/PS2_Keyboard> - חיבור המקלדת לקרנל, מה שמתבצע לאחר הגדרת הinterrupt למקלדת וחיבור הPIC.

<https://wiki.osdev.org/Interrupt_Service_Routines> - הגדרה כללית של רוטינות שמטפלות בinterrupts שונים – גם כאלה שנוצרות על ידי המעבד – exceptions

<https://www.youtube.com/watch?v=pfj95yQbF4M> – סרטון על פסיקות חומרה

<https://www.youtube.com/watch?v=lWhyXLXflt4&list=PLFjM7v6KGMpiH2G-kT781ByCNC_0pKpPN&index=13> – סרטון על פסיקות חומרה

<https://wiki.osdev.org/How_Do_I_Use_A_Debugger_With_My_OS#Use_GDB_with_QEMU> – עוסק בצורת הדיבוג של המערכת – באמצעות הכלי gdb – gnu debugger

<https://wiki.osdev.org/IA32_Architecture_Family> - ארכיטקטורת המעבדים 8086 ובפרט IA32 שהתחילה מהמעבד 80386 שהיה הראשון למממש זיכרון וירטואלי 32 ביט.

<https://wiki.osdev.org/Segmentation>

<https://wiki.osdev.org/File_Management>

<https://wiki.osdev.org/File_Systems>

<https://wiki.osdev.org/FAT>

<https://wiki.osdev.org/GAS> - GAS - gnu assembly הוא סינטקס האסמבלי של הקומפיילר gcc, בו אני משתמש בשביל הקומפילציה של הקוד שלי

<https://wiki.osdev.org/LD> - ld הוא תוכנת הלינקר של gcc, בה אני משתמש כדי לחבר ביחד את קבצי הקוד שלי ולקבל את הקובץ mykernel.bin שהוא קוד הקרנל בפועל שנטען למכונה הוירטואלית

<https://wiki.osdev.org/QEMU> - qemu היא המכונה הוירטואלית בה בחרתי כדי להריץ את הקוד שלי. כאמור, בחרתי בסביבה זו כי מדובר בכלי נוח שניתן להריץ במהירות משורת הפקודה

<https://www.youtube.com/watch?v=myU2x27FIIc> – סרטון יוטיוב שמסביר על הבדלים בין PATA ובין SATA שהן גרסאות מתקדמות יותר של ata – ממשק הדיסק בו בחרתי לעבוד.

<https://www.reddit.com/r/linux4noobs/comments/12j9chi/demystifying_the_cpu_what_x86_x86_64_i386_i686/> - פוסט רדיט שמספר בצורה בהירה על דורות המעבדים של אינטל וההתפתחות שלהם ממעבד ה8086 עד למעבדים כיום שעובדים עם 64 ביט.

<https://stackoverflow.com/questions/78525061/error-variable-or-field-keyboard-declared-void-implemeting-and-compiling-fu> - שאלות ששאלתי באתר stack overlflow ונענו על ידי אנשים מהקהילה.

<https://stackoverflow.com/questions/78509221/gas-with-att-syntax-creating-a-label-by-concatenating-a-variable-of-a-macro>

<https://stackoverflow.com/questions/78498314/error-expected-constructor-destructor-or-type-conversion-before-extern-ha>

<https://stackoverflow.com/questions/78466322/a-question-regarding-an-iso-building-a-simple-operating-system>

<https://stackoverflow.com/questions/78564332/building-a-file-system-creating-a-ata-pio-interface>

<https://stackoverflow.com/questions/78568547/virtual-hard-disk-ata-pio-saving-state-of-hard-drive-write-operations-in-th>

בנוסף, ברצוני להודות לקהילת האונליין של לינוקס וספציפית חדרי הIRC ושרת הדיסקורד של osdev.

1. **נספחים**

לינק לפרופיל הגיטהאב שלי: <https://github.com/yfrandom2020>

שם הפרויקט: yb-os

קוד הפרויקט: