**פרויקט בסייבר - פיתוח מערכת הפעלה**

**הקדמה - מהי 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 היא רכיב חיוני בכל מחשב. - ויקיפדיה

**תוכן עניינים:**

**מערכת הפעלה 1 - מערכת הפעלה שלא עושה כלום ---------------------------------------4**

**מערכת הפעלה 2 - מערכת הפעלה שמדפיסה למסך ---------------------------------------4**

**מערכת הפעלה 3 - מערכת הפעלה שמדפיסה למסך בprotect mode**  **---------------5**

**מערכת הפעלה 4 - מערכת הפעלה עם GDT -----------------------------------------------8**

**מערכת הפעלה 5 - מערכת הפעלה עם IDT ------------------------------------------------9**

**מערכת הפעלה 6– מערכת הפעלה המנצלת את הפסיקות -------------------------------11**

**חלק א – מערכת הפעלה real mode**

**Real mode**

למעבדים קיימים מספר אופני פעולה שונים: במצב real mode המעבד עובד ללא שום מנגנוני הגנה - הגישה לחומרה נעשית באופן ישיר וללא שום בקרה וגודל האוגרים הוא 16 ביט.

**BIOS**

ה BIOSהוא הרכיב הראשון שמופעל כשאנחנו מדליקים את המחשב. ה- BIOS אחראי על ביצוע בדיקה ראשונית למחשב, בדיקה זו אחראית לוודא שכל רכיבי המחשב קיימים.

כמו כן, הBIOS מהווה ממשק אוניברסאלי לרכיבי החומרה, שמאפשר גישה אליה. ניצלתי יכולת זו בתחילת הדרך של כתיבת מערכת ההפעלה.

אחרי שהBIOS מוודא שהמחשב תקין הוא קורא את התוכנית הראשונה בהארד דיסק וטוען אותה לכתובת בזיכרון0x7c00 , לתוכנה הזו קוראים **mbr**.

**MBR**

הmbr מכיל רשימה של כל המחיצות בדיסק (מבחינת מערכת ההפעלה, כל מחיצה היא דיסק בפני עצמה). בנוסף ה MBRיכול להכיל קוד בסיסי שירוץ כאשר ה BIOSקורא אותו. ניצלתי תכונה זו בתחילת כתיבת המערכת.

**מערכת הפעלה 1 - מערכת הפעלה שלא עושה כלום**

כתבתי את קוד האסמבלי הזה:

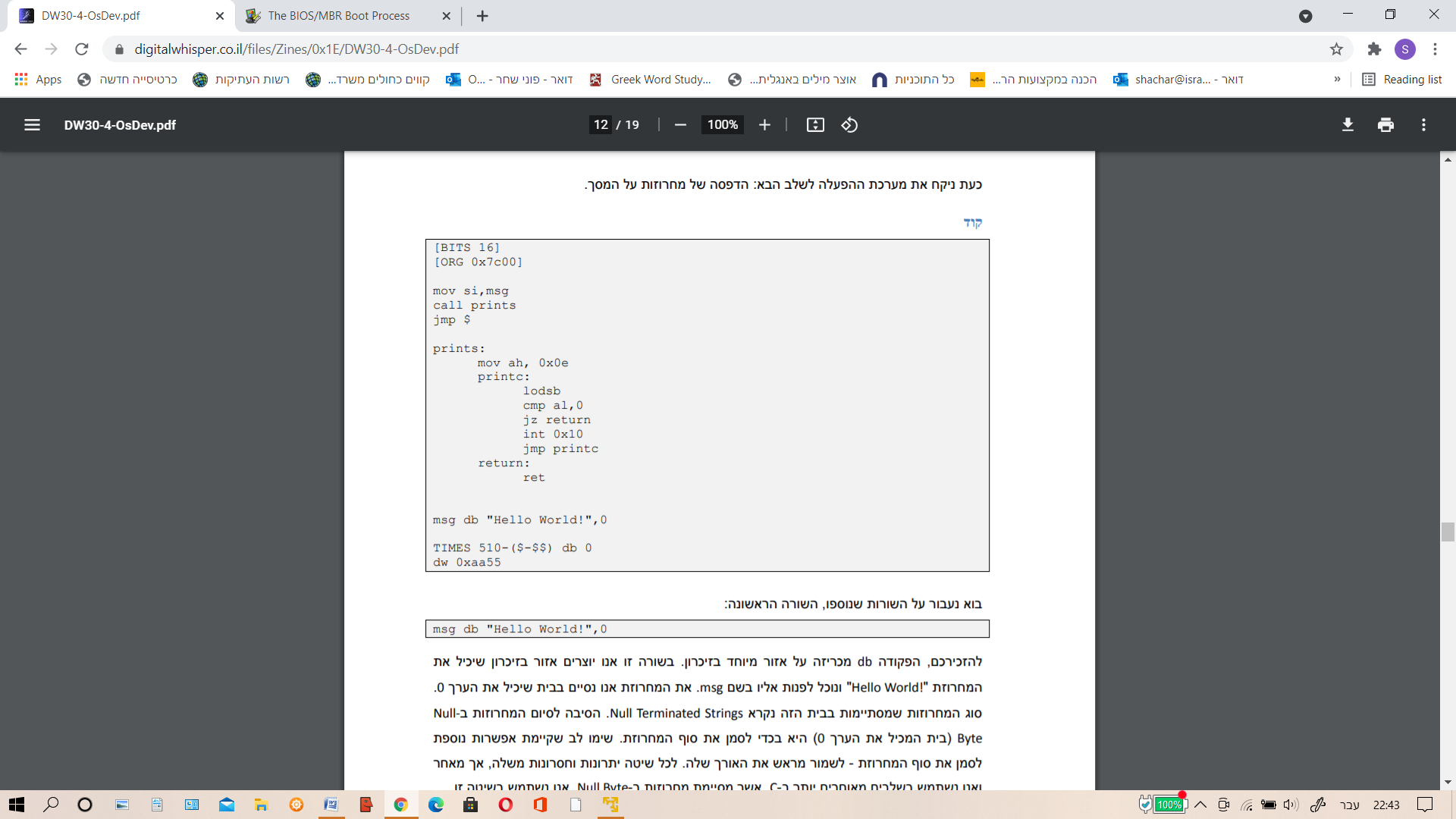


קוד זה אומר שבmbr תהיה לולאה אינסופית שלא עושה כלום. (הוא גם אומר להתייחס אליו כאל real mode ושהוא ייטען מ0x7c00. ) מכיוון שהmbr חייב להיות עם 512 בתים, הקוד ממלא את שאר הmbr באפסים ובסוף בחתימת הmbr.

בעזרת פקודות בLinux הפכתי את הקוד הזה לקוד מכונה, ואת קוד המכונה הפכתי לקובץ ISO(דיסק). את קובץ זה אפשר לצרוב לדיסק או להריץ מתוכו את מערכת ההפעלה דרך תוכנת וירטואליזציה שיודעת להתייחס אליו ככונן דיסקים לכל דבר.

**מערכת הפעלה 2 - מערכת הפעלה שמדפיסה למסך**

הוספתי לקוד תוספת:



מערכת הפעלה זו משתמשת בפסיקות של הBIOS ומדפיסה Hello World!

**חלק ב – מערכת הפעלה protect mode**

**Protect mode**

בחלק זה עברתי לprotect mode - מצב בו הגישה לזיכרון ולחומרה מבוקרת ומוגנת, ומימשתי אותו. בנוסף, כתבתי למסך בעזרת כתיבה לזיכרון, בניגוד לשימוש בפסיקות הBIOS. ו גם כתבתי את תחילת הגרעין (kernel) בשפת C.

**Boot loader**

מנהל אתחול (Boot loader) היא תוכנה המופעלת כחלק מתהליך האתחול של מחשב וטוענת את מערכת ההפעלה. התוכנה שמורה במיקום מוגדר בזיכרון ההתקן והיא פשוטה וקטנה בהרבה ממערכת ההפעלה, ולכן ניתן לטעון אותה בקלות רבה יותר.

השתמשתי בGRUB – מנהל אתחול שיכול להפעיל כל מערכת הפעלה שתומכת בסטנדרט מסוים (Mutliboot Specifications).

**מערכת הפעלה 3 - מערכת הפעלה שמדפיסה למסך בprotect mode**

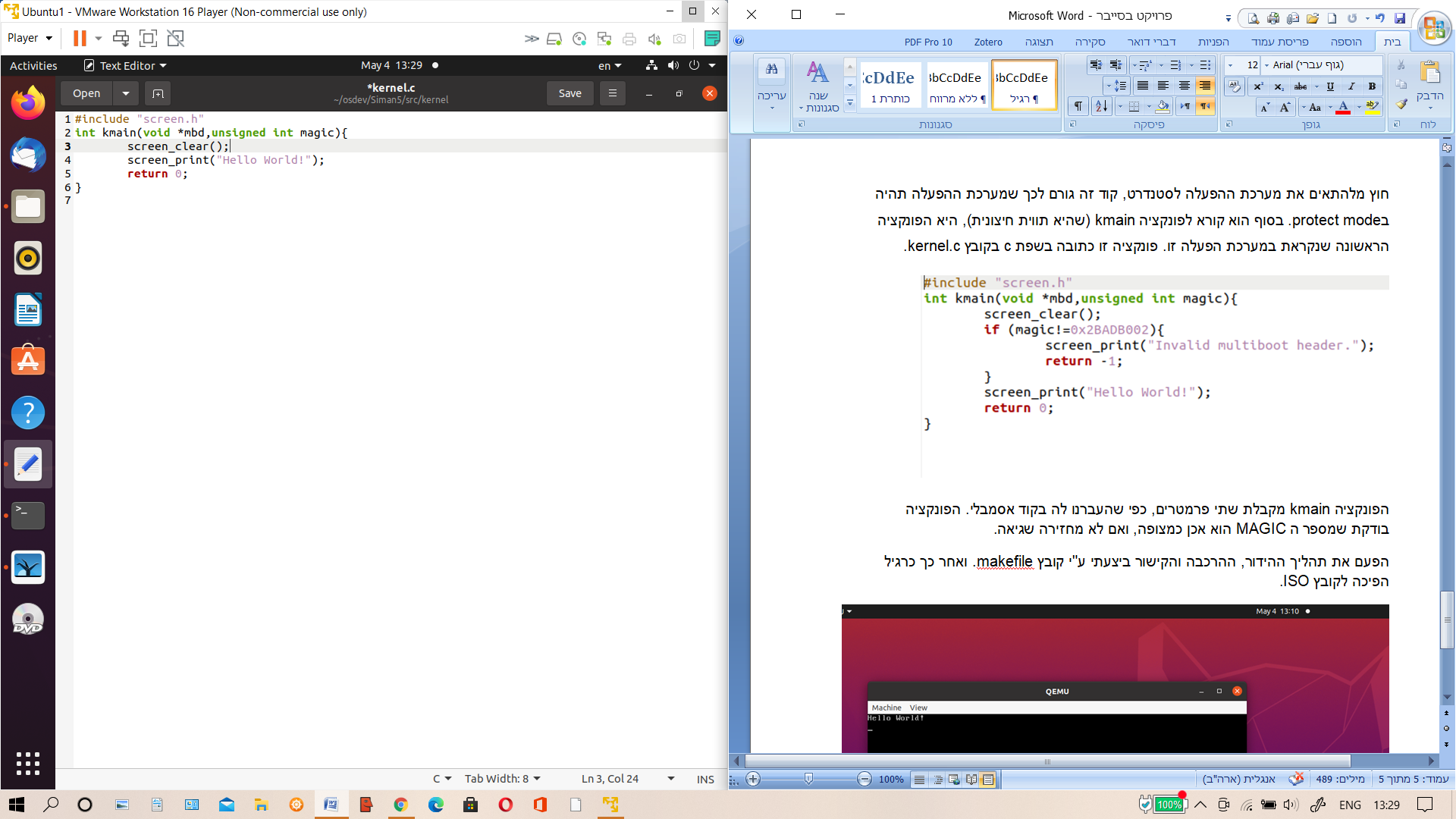
בניגוד לחלק הקודם, שהיה בreal mode ,ויכולתי להדפיס על ידי פסיקות BIOS ,בprotect mode אי אפשר לגשת ישירות לחומרה.

הפעם מה שחשוב הוא שמערכת ההפעלה תהיה בפורמט מסוים, כדי שה – bootloader יוכל לדעת כיצד לטעון אותה.

הפעם קוד האסמבלי נראה כך:

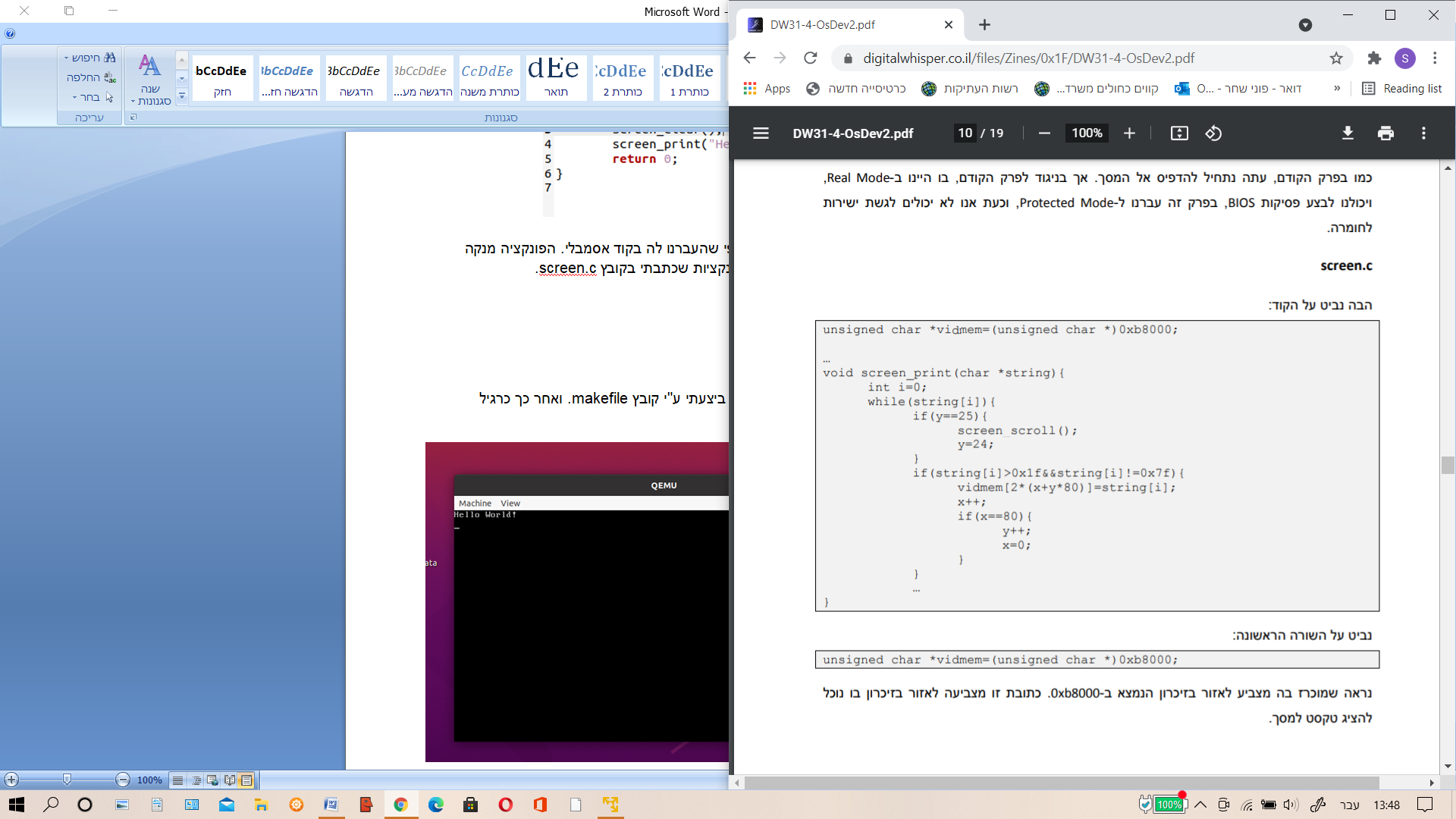


חוץ מלהתאים את מערכת ההפעלה לסטנדרט, קוד זה גורם לכך שמערכת ההפעלה תהיה בprotect mode. הקוד גם קורא בסוף לפונקציה kmain (שהיא תווית חיצונית), היא הפונקציה הראשונה שנקראת במערכת הפעלה זו. פונקציה זו כתובה בשפת c בקובץ kernel.c.



הפונקציה kmain מקבלת שני פרמטרים, כפי שהעברנו לה בקוד אסמבלי. הפונקציה מנקה את המסך ומדפיסה hello world! בעזרת פונקציות שכתבתי בקובץ screen.c.

נתמקד בפונקציה screen\_print:



**Text Mode Graphics**

הכתובת 0xb8000מצביעה לאזור זיכרון שממנו המעבד קורא ומדפיס תווים בצבעים שונים למסך. במצב text mode (המצב בו מערכת ההפעלה נמצאת לאחר האתחול) המסך מחולק ל- 80 על 25 תאים, ולכל אחד מהתאים מוקצה 2 בתים - הראשון מייצג את הערך הASCII של התו שהתא אמור להציג, והשני מייצג את התכונות של התו.

הפונקציה screen\_print פשוט מדפיסה את המחרוזת למסך. (הפונקציה מדפיסה את התו למסך, יורדת שורה במקרה הצורך, גוללת את המסך במקרה הצורך, ואם התו הינו תו שאינו ניתן להדפסה אך מסמל קפיצה מסוימת (שורה חדשה, TAB וכו') היא "מדפיסה" אותו באופן מלאכותי.

הפעם את תהליך ההידור, ההרכבה והקישור ביצעתי ע"י קובץ makefile. ואחר כך כרגיל הפיכה לקובץ ISO.



**חלק ג – מערכות הפעלה protect mode מתקדמות**

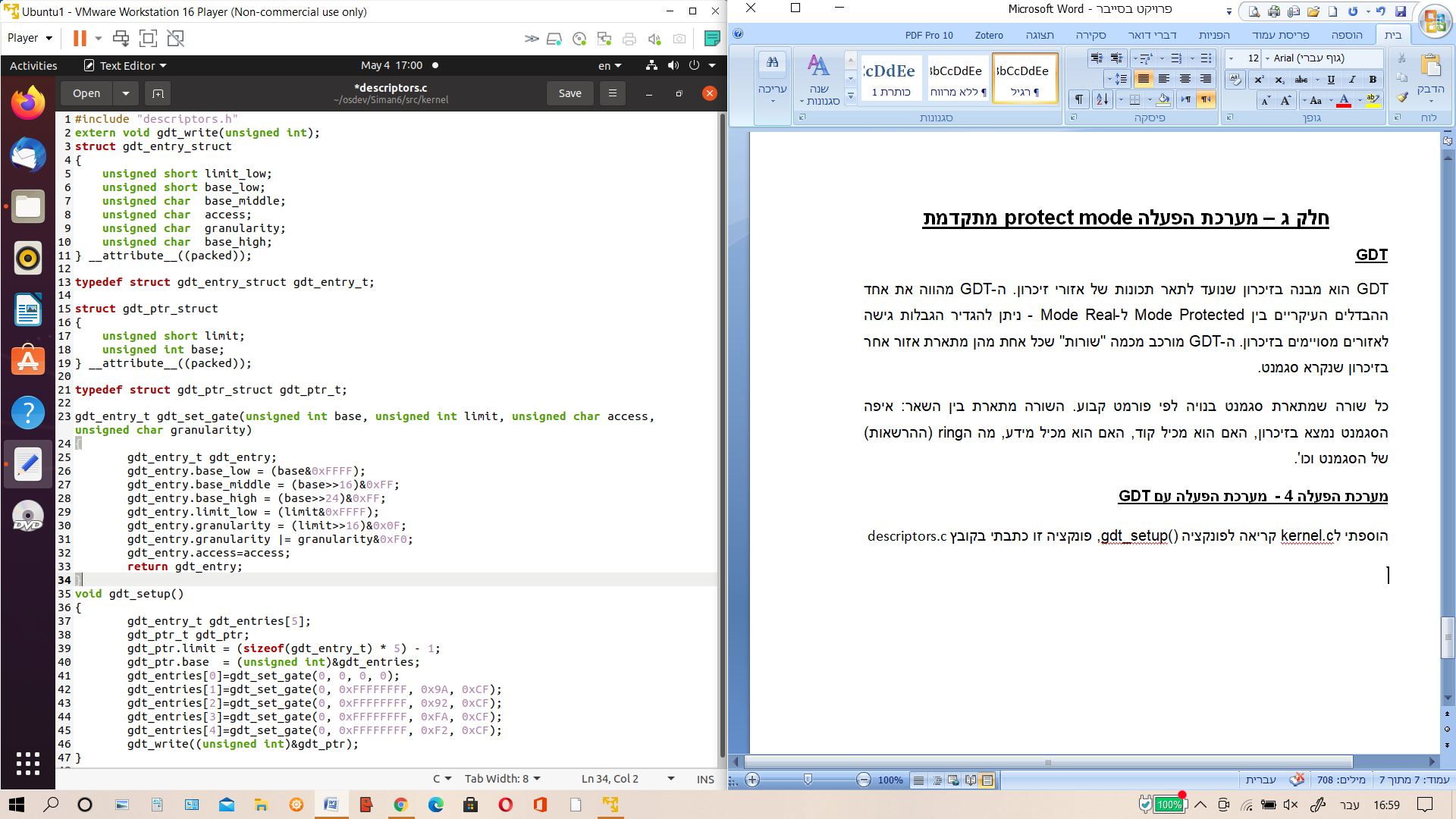
**GDT**

GDT הוא מבנה בזיכרון שנועד לתאר תכונות של אזורי זיכרון. ה-GDT מהווה את אחד ההבדלים העיקריים בין Mode Protected ל-Mode Real - ניתן להגדיר הגבלות גישה לאזורים מסויימים בזיכרון. ה-GDT מורכב מכמה "שורות" שכל אחת מהן מתארת אזור אחר בזיכרון שנקרא סגמנט.

כל שורה שמתארת סגמנט בנויה לפי פורמט קבוע. השורה מתארת בין השאר: איפה הסגמנט נמצא בזיכרון, האם הוא מכיל קוד, האם הוא מכיל מידע, מה הring (ההרשאות) של הסגמנט וכו'.

**מערכת הפעלה 4 - מערכת הפעלה עם GDT**

הוספתי לkernel.c קריאה לפונקציה gdt\_setup(), פונקציה זו כתבתי בקובץ **descriptors.c**



בשורה 2 יש פונקציה חיצונית - פונקציה שכתבתי באסמבלי:



הפונקציה gdt\_setup() יוצרת בהתחלה מערך של חמישה מבנים – כל מבנה הוא שורה בGDT שמתארת סגמנט בזיכרון. לאחר מכן היא קוראת לפונקציה gdt\_write הכתובה באסמבלי, פונקציה זו מבצעת את הפקודהlgdt - הפקודה מקבלת מתאר של ה-GDT, שמורכב משני בתים שמגדירים את גודל ה-GDT ,ו-4 בתים של מצביע בזיכרון ל-GDT. (את המתאר של ה-GDT הפונקציה קיבלה כפרמטר מgdt\_setup().) וטוענת את הGDT לזיכרון.

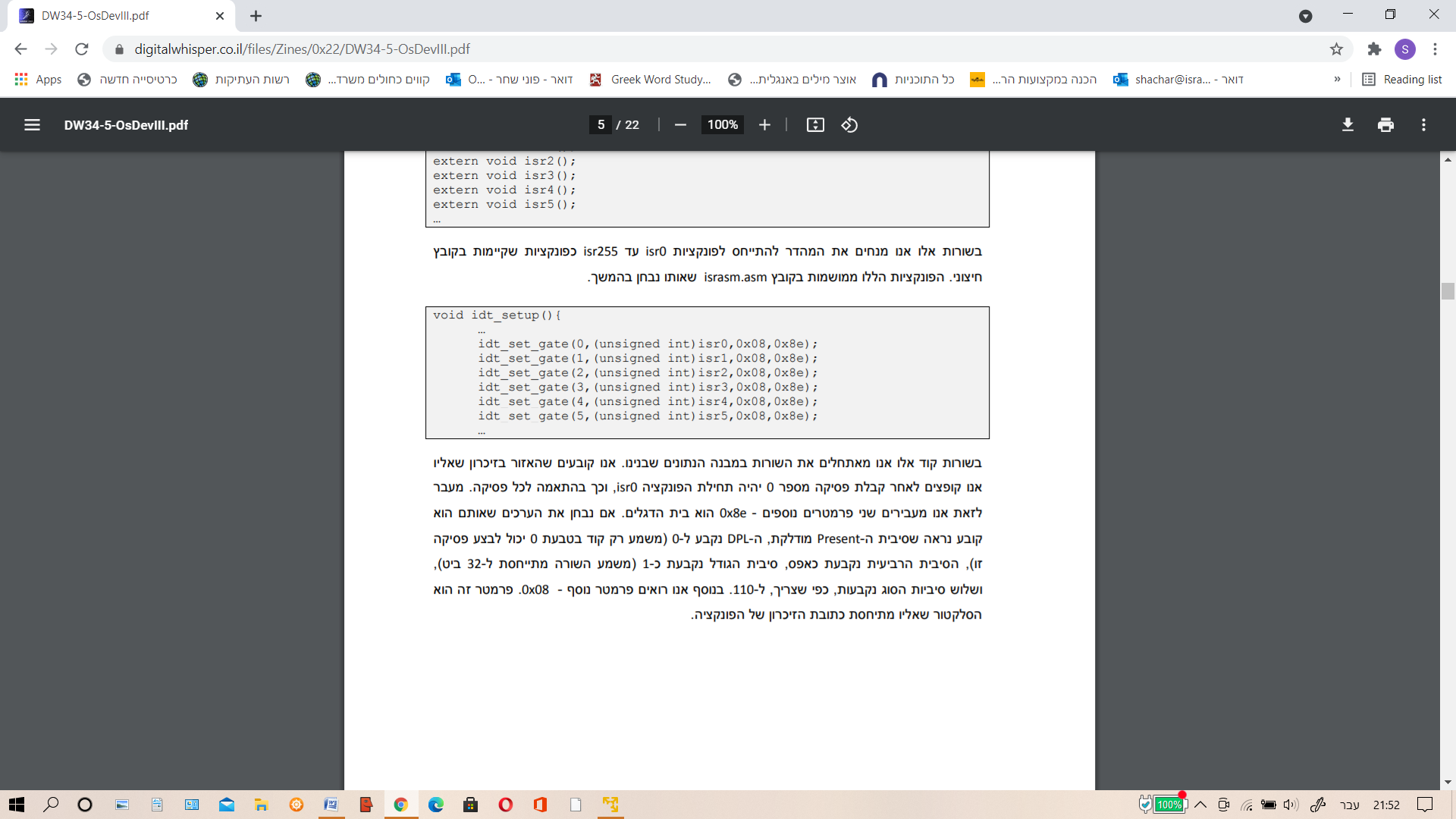
**מערכת הפעלה 5 - מערכת הפעלה עם IDT**

**IDT**

IDT הוא מבנה בזיכרון שנועד להנחות את המעבד כיצד לפעול בעת קבלת פסיקות. הוא המקביל לIVT שבו השתמשתי בחלק הראשון כדי להדפיס למסך בעזרת הפסיקות של הBIOS. הוא מורכב מ-256 שורות, שמקבילות ל-256 מספרי פסיקות שונים (0x00 – 0xff) שיכולים להיווצר. כל שורה מנחה את המעבד לאיזו פונקציה לקרוא כדי שתוכל לטפל בפסיקה שהתקבלה.

יצירת הIDT דומה ליצירת הGDT, הוספתי לקובץ **descriptors.c** את הפונקציה idt\_setup שיוצרת 256 שורות של תיאור פסיקות( כל שורה חייבת להיות לפי הפורמט), ולאחר מכן קוראת לפונקציה חיצונית באסמבלי שטוענת את הIDT לזיכרון. (בדיוק כמו הGDT רק עם הפקודה lidt)

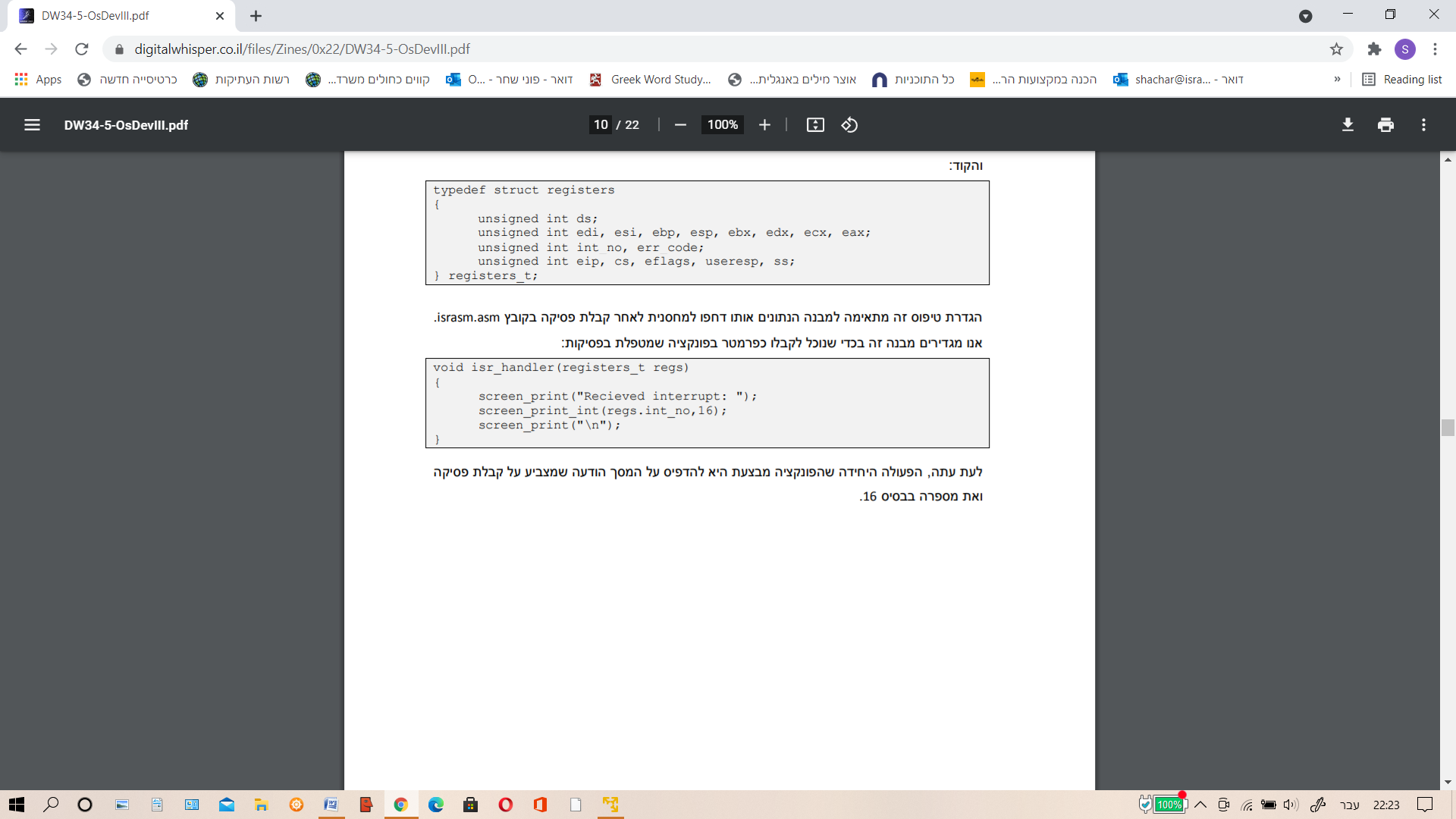
כל שורה שמתארת פסיקה היא בעלת פורמט קבוע. השורה מתארת בין השאר: מה ההרשאות של אותה פסיקה, מה סוג השורה, והכי חשוב: איפה נמצאת בזיכרון הפונקציה שמטפלת בפסיקה.



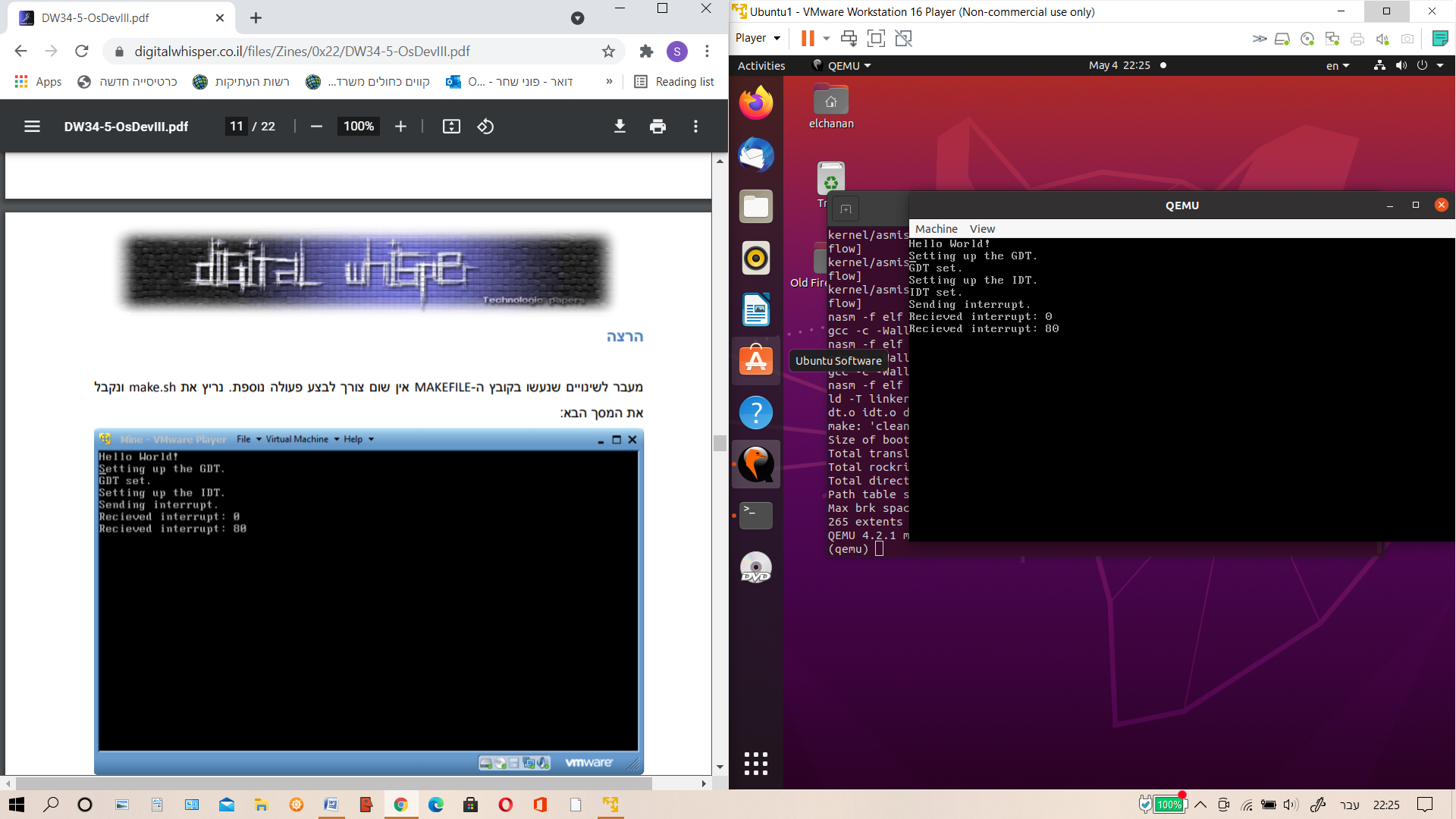
את מיקום הפונקציה בזיכרון אני מכניס בכל שורה על ידי כתיבת (unsigned int)isr% וזה מכניס את הכתובת בזיכרון של הפונקציה isr0 וכו'. פונקציות אלו נמצאות בקובץ asmisr.asm באסמבלי.

כל פונקציה ((isr0, isr1, isr2 קוראת לפונקציה isr\_handler ושולחת לו את המתאר שלה.

הפונקציה isr\_handler כתובה בשפת c ונמצאת בקובץ icr.c מדפיסה את מספר הפסיקה שהתקבלה:



התוצאה (בקרנל קראתי לפסיקה 0 ולפסיקה 80):

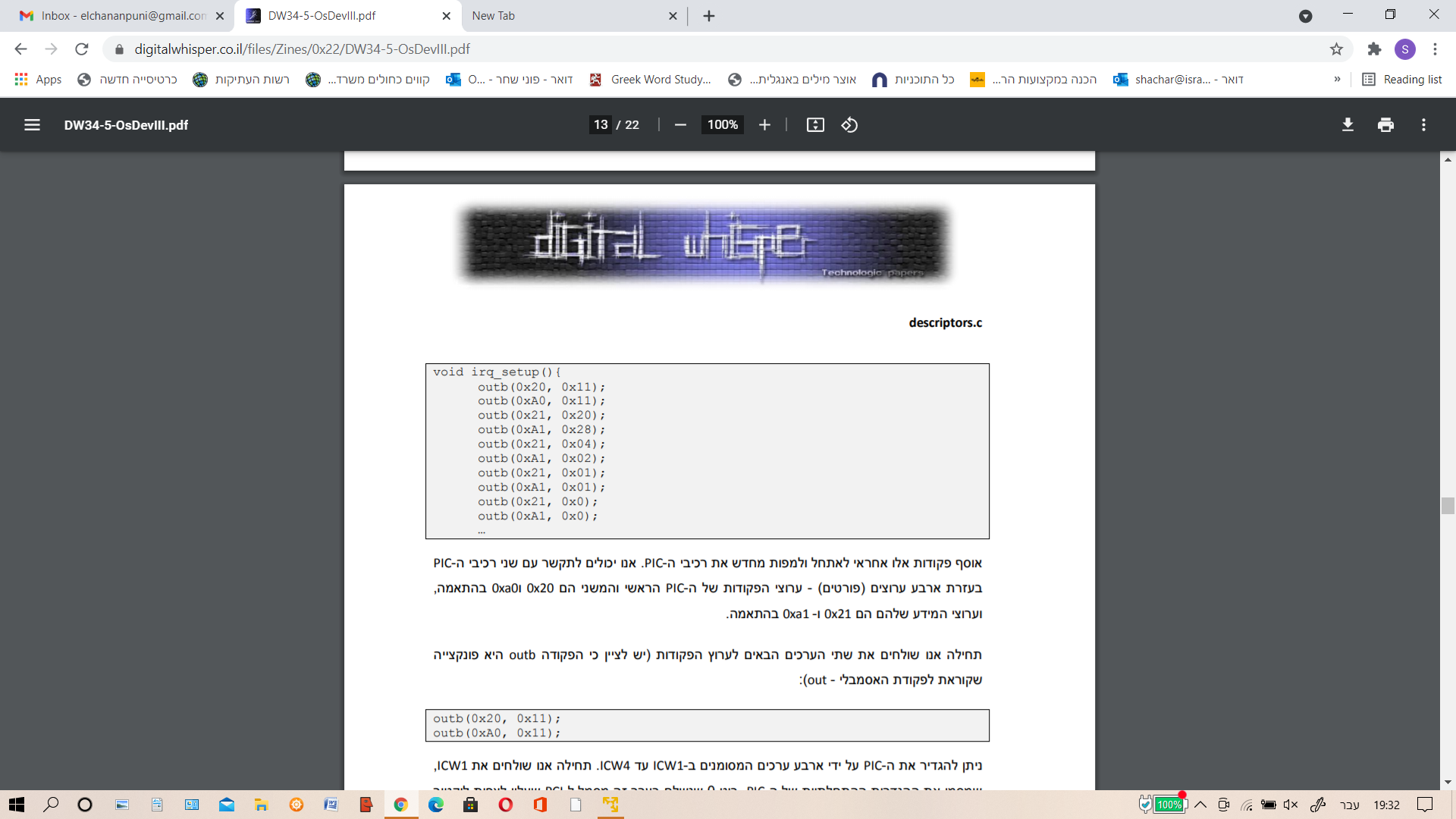


**חלק ד – מערכת הפעלה המנצלת את הפסיקות**

**Pic**

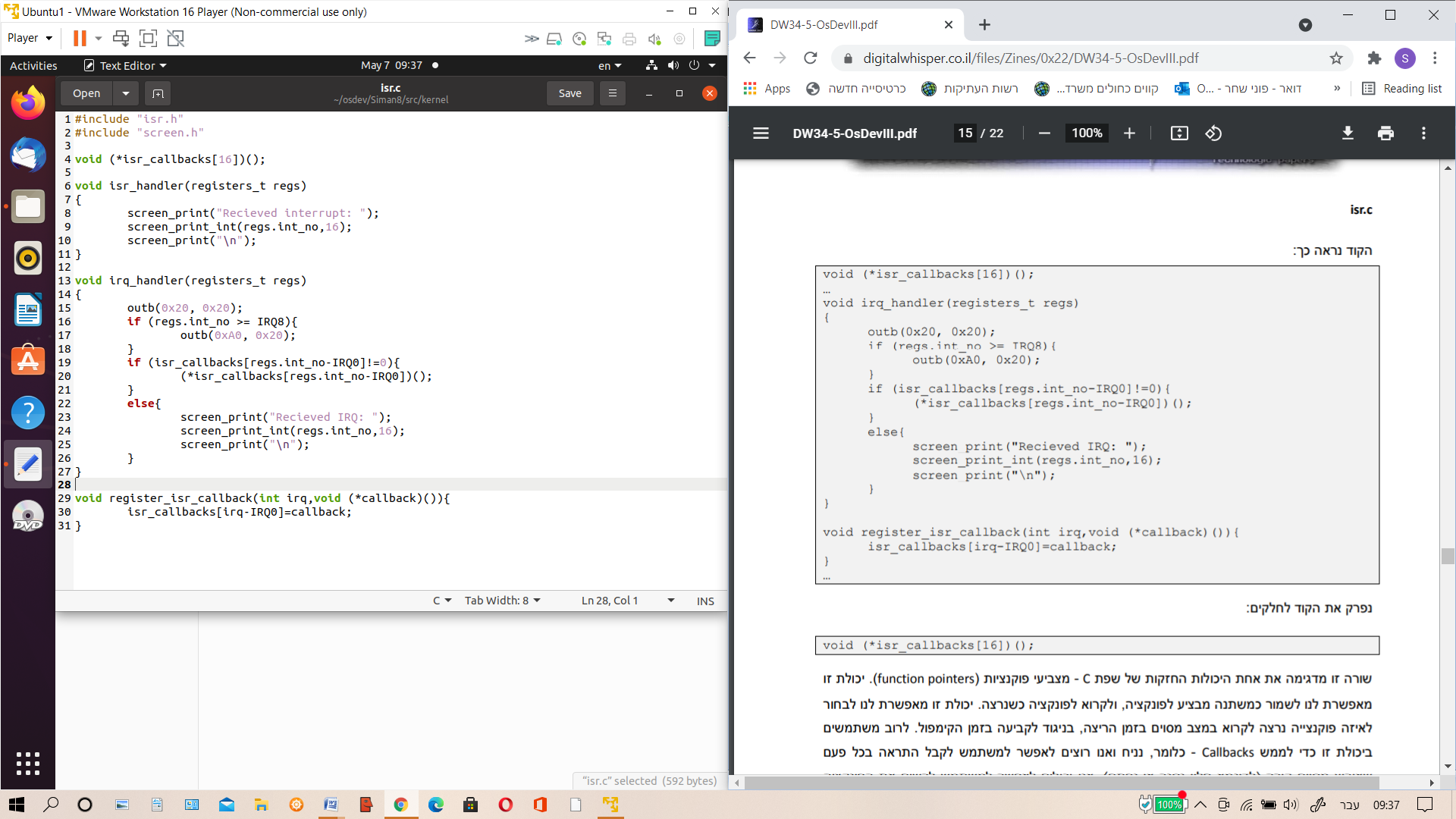
הpic הוא רכיב חומרתי המחובר לפסיקות של המעבד מצד אחד, ולרכיבים חומרתיים מצדו השני, יש במחשב שני picים שיכולים לחבר 15 רכיבי חומרה שונים ל15 פסיקות שונות. כל ערוץ המחבר בין רכיב חומרה לפסיקה נקרא IRQ, ונהוג ש15 הIRQ יהיו מחוברים לפסיקות 0x20 עד 0x2f (הפסיקות הראשונות הם פסיקות שהמעבד משתמש בהן). הpic מאפשר להשתמש בשיטה הנקראת polling שבה מערכת ההפעלה פועלת כרגיל, ואם מתקבלת פסיקה היא עוצרת ונותנת עדיפות לפסיקה (שיטה זו חוסכת הרבה מכוח העיבוד של המחשב, כי לדוגמא לא צריך לבדוק כל הזמן אם מקש נלחץ במקלדת).

כתבתי בdescriptors.c את הפונקציה irq\_setup() שמתאחל וממפה את שני רכיבי הpic.



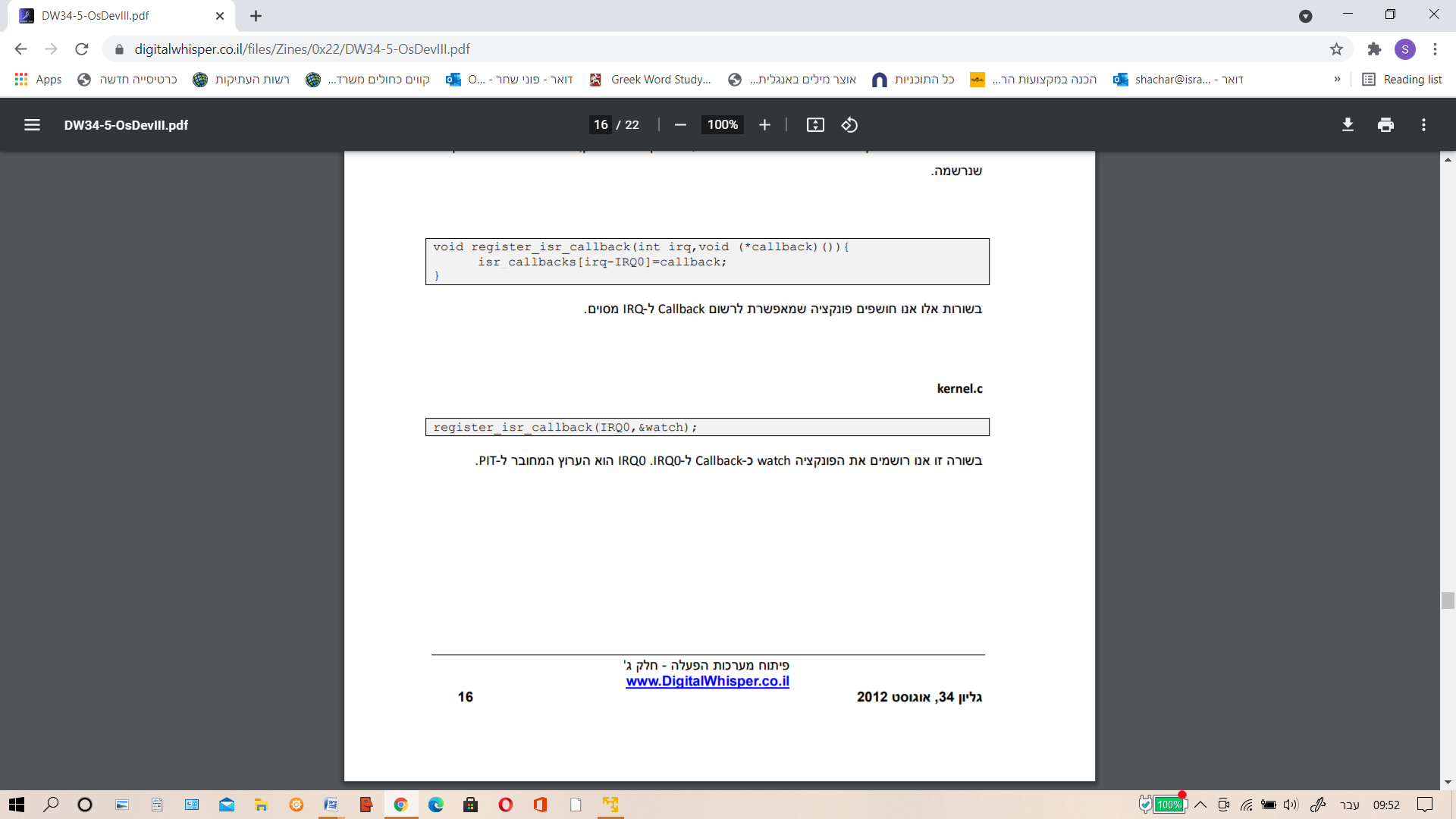
פקודות אלו קובעות בין השאר איזה פסיקות מחוברות ל15 הערוצים.

הוספתי גם לקובץ isr.c את הקוד הבא:



קוד זה מבצע ack (סימון ל PIC שקיבלנו את הפסיקה דרך ערוץ הפקודות שלו). ומאפשר שלאחר קבלת IRQ מסוים תתבצע פונקציה המיוחדת לאותו IRQ.

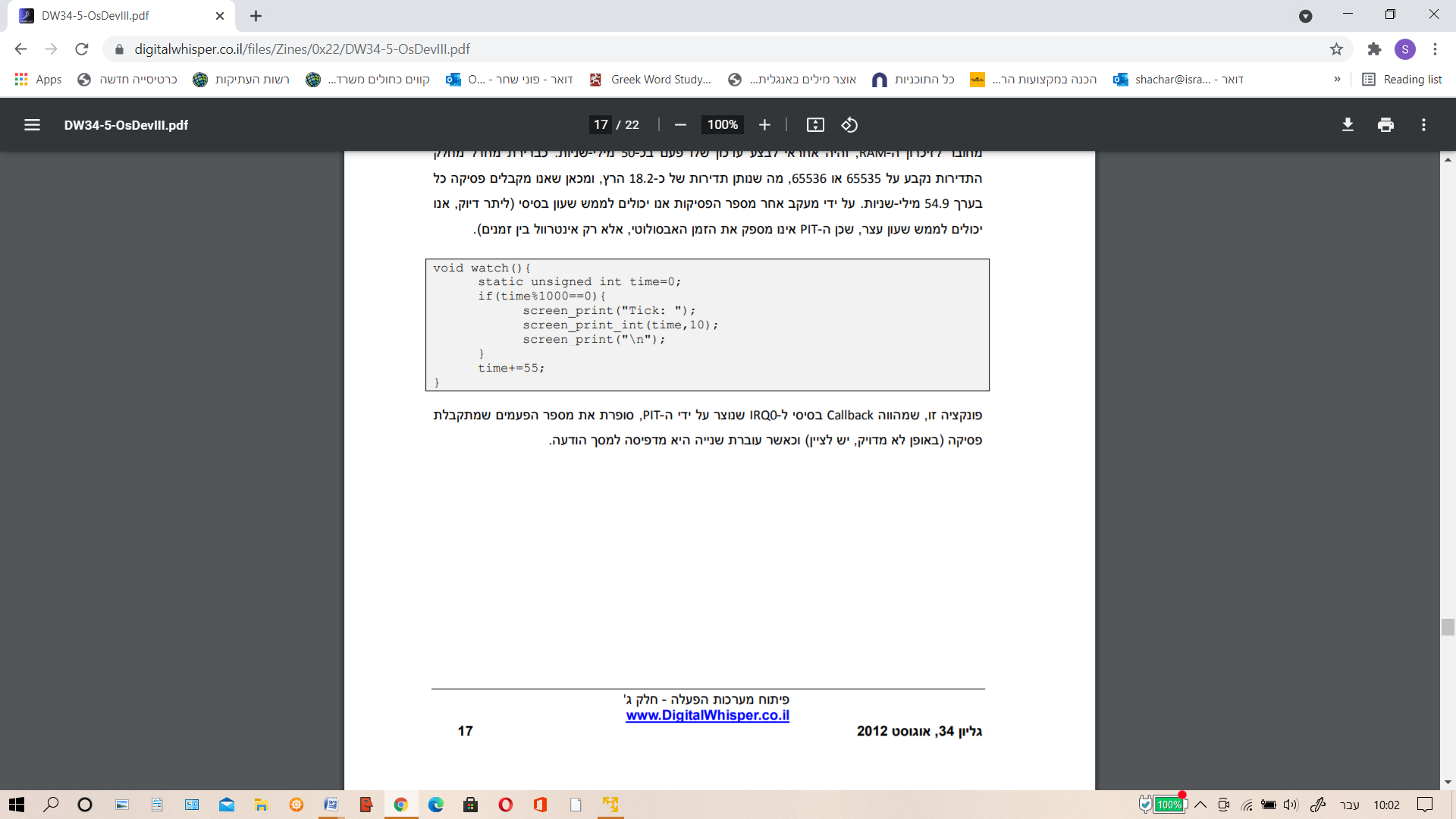
ובdescriptors.c נקרא לפונקציה watch כשמתקבל IRQ0.



**Pit**

Pit הוא רכיב ששולח לפי תדירות שנותנים לו קריאה לirq0.

כך כתבתי את פונקצית watch שמדפיסה את הזמן שעבר כל שנייה כשמתקבל IRQ0:



התוצאה:

