מערכות הפעלה

ברק גונן

המרכז לחינוך סייבר
CYBER EDUCATION CENTER

מערכות הפעלה

Operating Systems / Barak Gonen

גרסה 1.0

<u>כתיבה</u>:

ברק גונן

<u>עריכה</u>:

שלומי בוטנרו

אין לשכפל, להעתיק, לצלם, להקליט, לתרגם, לאחסן במאגר מידע, לשדר או לקלוט בכל דרך או אמצעי אלקטרוני, אופטי או מכני או אחר – כל חלק שהוא מהחומר שבספר זה. שימוש מסחרי מכל סוג שהוא בחומר הכלול בספר זה אסור בהחלט, אלא ברשות מפורשת בכתב מהמרכז לחינוך סייבר, קרן רש"י.

© כל הזכויות שמורות למרכז לחינוך סייבר, קרן רש"י. מהדורה ראשונה תש"ף 2020

http://www.cyber.org.il

תודות

זכיתי לשתף פעולה עם כמה מהאנשים המוכשרים ביותר שיש בקהילת הסייבר בארץ, שנוסף על ידע אדיר יש להם גם תשוקה אמיתית לחינוך ואני מלא הערכה אליהם.

ל**שלומי בוטנרו**, שהקדיש זמן רב בהתנדבות לעריכת הספר ומאות הערותיו המועילות שזורות בכל הפרקים.

ל**יותם גיגי** ו**חגי לוי** שהידע הנרחב והנסיון הפדגוגי שלהם תרמו רבות והשפיעו על התהוות הספר עוד טרם שהתחלתי בתהליך הכתיבה.

ל**אפיק קסטיאל** עורך העיתון הדיגיטלי האיכותי Digital Whisper, שלא אחת אני משוטט בגליונותיו, על שפרסם טיוטות מהספר ועזר לשייף אותן.

ל**מפתחים של Sysinternals**, שככל הנראה לא יקראו את ההקדשה הזו בעברית, אבל בלעדיהם כמחצית מהספר לא היה קיים :)

תודה, אלופים!

תוכן עניינים

מבוא 1

ידע מקדמים ... 1
מיקוד הספר ושיטת הלימוד ... 1
פרקי הלימוד ... 3
תיקונים והצעות לשיפור ... 4

פרק 1– ידע מקדים: כלי דיאגנוסטיקה Sysinternals ... 5

1.1 ה-Registry ותוכנת RegEdit ... 5
1.2 תוכנת Autoruns ... 8
1.3 תוכנת Process Explorer ... 10
1.3.1 תצוגת Processים ... 10
1.3.2 מידע על ניצול המשאבים ... 11
1.3.3 מידע מפורט על Processים ... 12
1.3.4 מציאת Process ספציפי ... 12
1.3.5 "חיסול" Processים ... 13
1.3.6 מציאת Processים שמחזיקים קובץ פתוח ... 13
1.3.7 בדיקה אימות ואבטחה ... 13
1.4 תוכנת Procmon ... 14
1.5 סיכום ... 19

פרק 2– מחקר אזורי הזיכרון של תוכנה ... 20

2.1 אופן פעולת המעבד ... 20
2.1.1 שפת מכונה ... 20
2.1.2 תזכורת- מה משמעות מעבד 32 למול 64 ביט? ... 20
2.1.3 כיצד עובד המעבד ... 21
2.1.4 משפה עילית לשפת מכונה ... 23
2.2 מחקר אזורי הזיכרון השונים של תוכנה ... 23
2.2.1 שלב א'- פקודות ... 24
2.2.2 שלב ב' - קבועים ... 24
2.2.3 שלב ג' – משתנים גלובליים ... 25
2.2.4 שלב ד' – משתנים מקומיים ... 26
2.2.5 שלב ה' – סוגי משתנים נוספים ... 27
2.2.6 משתנים לא מאותחלים ... 29
2.2.7 אזורי זיכרון נוספים שלא חקרנו עד כה ... 30
2.2.8 סיכום אזורי הזיכרון ... 30
2.3 נספח – מדריך בסיסי לעבודה עם IDA ... 33
2.3.1 ה-Disassembly הראשון שלנו ... 33

2.3.2 מבצעים Disassembly לקוד של עצמנו 36
2.3.3 מציאת ה-main........ 37
2.3.4 הדרך הקלה 43
פרק 3– תפקידי מערכת ההפעלה........ 45
3.1 מהם Processים........ 45
3.2 מהם Driverים........ 46
3.3 תפקידי מערכת ההפעלה 47
3.4 סוגים של מערכות הפעלה 49
3.5 התפתחות המעבדים ומערכות ההפעלה........ 50
3.5.1 מעבד ה-8086, Real Mode........ 50
3.5.2 מעבד ה-386........ 52
3.5.3 מעגלי הרשאה ו-Protected Mode 52
3.6 הרחבה – מעגל ההרשאה של Driverים........ 54
3.6.1 פרצות אבטחה ב-Driverים 54
3.7 ביצוע Syscall ופסיקת Sysenter 55
3.8 שימוש ב-Objects וב-Handles........ 57
3.8.1 מבנה נתונים של אובייקט........ 59
3.9 נספח - מבוא ל-WinAPI........ 65
3.9.1 רקע........ 65
3.9.2 טיפוסי משתנים בסיסיים 65
3.9.3 מצביעים........ 66
3.9.4 מחרוזות........ 67
3.9.5 פונקציות עם סיומת A לעומת סיומת W 68
3.9.6 קריאה לפונקציות של WinAPI........ 69
3.9.7 שימוש יעיל בקודי שגיאה 71
3.10 סיכום........ 71
פרק 4– תהליכים ותהליכונים - Processes and Threads 72
4.1 תהליכונים - Threads........ 72
4.2 ריבוי תהליכונים - Multi Threads 72
4.3 תהליכים - Processes 77
4.4 ריבוי תהליכים - Multi-Process 79
4.5 ביצוע Context Switch........ 80
4.6 שימוש ב-Multi-Process לעומת Multi-Thread........ 81
4.7 מהם Services / Daemons (הרחבה) 84
4.7.2 שלב ג' – WaitForSingleObject 90
4.8 סיכום........ 96
פרק 5– סנכרון Processים ו-Threadים 97

5.1 מהו Race Condition 97
5.2 מהו Lock 100
5.3 מנעול מסוג CriticalSection 102
5.4 מצב Deadlock 104
5.5 מצב Starvation 104
5.6 מנעול מסוג Mutex 106
5.6.1 שימוש ב-Mutex 108
5.6.2הרחבה: שימוש ב-Mutexים בנוזקות 113
5.7 מנגנון איתות Semaphore 115
5.8 סיכום 117
פרק 6– זיכרון 118
6.1 זיכרון פיזי 118
6.1.1 גישה לכתובות בזיכרון הפיזי 122
6.1.2 חלוקת הזיכרון ל-Segmentים, כתובות לוגיות 122
6.1.3 יחידת ה-Segmentation והמרה ל-Linear Address 124
6.2 זיכרון וירטואלי 125
6.2.1 יחידת ה-Paging והמרה ל-Physical Address 128
6.2.2 חלוקה ל-Page Directories 130
6.2.3 תהליך Context Switch בין Processים 133
6.2.4 מנגנון ה-TLB 134
6.2.5 תפקידים נוספים של Page Table ו-Page Directory 135
6.2.6 שגיאת Page Fault 136
6.2.7 הקובץ Pagefile.sys 137
6.2.8 תוכנת VMMap 138
6.3 סיכום 141
פרק 7– שיתוף זיכרון 142
7.1 מיפוי זיכרון קלט/פלט - Memory Mapped I/O 142
7.2 מיפוי קובץ- Memory Mapped File 143
7.2.2 פונקציית CreateFileMapping 146
7.2.3 פונקציית MapViewOfFile 146
7.2.4 פונקציית UnmapViewOfFile 147
7.2.5 פונקציית CloseHandle 148
7.2.6 סיכום תרגיל מודרך 154
7.3 זיכרון משותף - Shared Memory 154
7.3.1 דוגמאות ל-Shared Memory 155
7.4 תוכנת RAMMap 157
7.5 סיכום 159

פרק 8– תהליך ה-Linking ויצירת DLLים 163
8.1 תהליך המרת קוד משפת C לקובץ הרצה exe 164
8.1.1 תהליך ה-Compiling 165
8.1.2 תהליך ה-Assembling 166
8.1.3 תהליך ה-Linking 168
8.2 יצירת Library וביצוע Static Linking 171
8.3 קובץ DLL 172
8.3.1 יצירת DLL 175
8.3.2 יבוא DLL ב-Load Time 179
8.3.3 יבוא DLL ב- Run Time 181
8.4 שימוש ברגיסטרי ל-DLL Injection 182
8.5 סיכום 189
פרק 9– פורמט PE ותהליך ה-Loading 190
9.1 ה-Loader 190
9.1.1 תפקידי ה-Loader 190
9.1.2 המחשה ל-Address Resolution על ידי ה-Loader 192
9.1.3 ביצוע Relocation 194
9.1.4 הרחבה- שימוש בסיסי ב-WinDbg 195
9.2 פורמט PE 196
9.2.1 אזור ה-DOS Header 196
9.2.2 אזור ה-NT Header 198
9.2.3 אזור ה-File Header 198
9.2.4 אזור ה-Optional Header 200
9.2.5 אזור ה-Data Directories 207
9.2.6 אזור ה-Section Headers 208
9.2.7 אזור ה-Import Directory 209
9.3 מנגנון ה-ASLR 214
9.4 סיכום 228

רשימת תרגילים

תרגיל 1.1 תרגול Process Explorer 12

תרגיל 1.2 תרגיל מודרך Procmon 14

תרגיל 1.3 תרגילי Procmon 16

תרגיל 1.4 תרגיל מסכם – Naughty Window 16

תרגיל 3.1 תרגיל מסכם מודרך - Syscall 61

תרגיל 4.1 מחקר משאבים של Process 78

תרגיל 4.2 מחקר Services 86

תרגיל 4.3 תרגיל מודרך – יצירת Thread 86

תרגיל 4.4 תרגיל מודרך - יצירת Process 91

תרגיל 4.5 תרגיל מסכם 96

תרגיל 5.1 תרגיל מסכם CriticalSection 105

תרגיל 5.2 תרגיל ביניים Mutexים 110

תרגיל 5.3 תרגיל מסכם **Mutex** 114

תרגיל 6.1 פירוק כתובת וירטואלית למרכיביה 131

תרגיל 7.1 תרגיל מודרך מיפוי קובץ לזיכרון, שלב א 143

תרגיל 7.2 תרגיל מודרך מיפוי קובץ לזיכרון, שלב ב 148

תרגיל 7.3 תרגיל מסכם 159

תרגיל 7.4 תרגיל מסכם לפרקי הלימוד עד כה – "זמן של מספרים" 160

תרגיל 8.1 תרגיל מסכם - ביצוע DLL Injection 186

תרגיל 9.1 שדה ה-Magic 196

תרגיל 9.2 שדה ה-lfanew 197

תרגיל 9.3 תרגיל אתגר לחובבי אסמבלי - DOS Stub 198

תרגיל 9.4 שדה ה-Machine 199

תרגיל 9.5 שדה ה-NumberOfSections 199

תרגיל 9.6 מכונת הזמן 199

תרגיל 9.7 תרגיל Optional Header Magic 201

תרגיל 9.8 תרגיל Entry Point 203

תרגיל 9.9 תרגיל Padding 205

תרגיל 9.10 תרגיל Section Headers 208
תרגיל 9.11 מנגנון ה-ASLR 215
תרגיל 9.12 תרגיל מסכם- IAT Hooking 216

מבוא

ברוכים הבאים לעולם מערכות ההפעלה! ספר זה ילווה אתכם בצעדים הראשונים בעולם המרתק הזה. מדוע שמישהו ילמד מערכות הפעלה? יש לכך כמה סיבות עיקריות:

- מערכות הפעלה הן נדבך ידע בסיסי ומרתק במדעי המחשב. הרעיונות התיאורטיים שפותחו במשך עשרות שנים קרמו עור וגידים והפכו לקוד מורכב ומתוחכם שמנהל את המחשב האישי שלכם ושרתי מחשב בעולם. ישנן בעיות קלאסיות במדעי המחשב הקשורות לעולם מערכות ההפעלה, ונראה כיצד מערכת ההפעלה שלנו מתמודדת איתם. לדוגמה, כיצד ניתן להריץ מספר תוכניות בו זמנית על אותו מחשב? כיצד מחלקים בין התוכניות השונות את משאבי המעבד ואת הזיכרון?
- עולם מערכות ההפעלה קשור באופן הדוק לעולם אבטחת המחשבים. כדי לפתח מערכות הגנה מפני נוזקות יש צורך להבין איך עובדת נוזקה. נוזקות משתמשות במשאבי המחשב, כגון מעבד וזיכרון. לכן כדי לפעול הן צריכות לחיות לצד מערכת ההפעלה. דבר זה מעלה מספר שאלות מעניינות, לדוגמה: כיצד נוזקות נטענות לזיכרון המחשב? מי מריץ אותן? כיצד נוזקות לא נמחקות לאחר כיבוי והדלקה של המחשב? אילו כלים יש לנסות ולאתר תוכנות חשודות שרצות במחשב שלנו?
- ידע במערכות הפעלה הוא בסיס שמאפשר מעבר לנושאים מתקדמים יותר כגון Reverse Engineering וניתוח נוזקות, נושאים שהלימוד שלהם הוא מרתק וגם שכרם בצידם $$$

ידע מקדים

ספר זה הוא הספר הרביעי בסדרת ספרי הלימוד של המרכז לחינוך סייבר. עד כה פורסמו:

א. תכנות בשפת פייתון – ברק גונן

ב. רשתות מחשבים – עומר רוזנבוים ושלומי הוד

ג. מבנה המחשב ושפת סף – ברק גונן

ידע בשפת פייתון וברשתות מחשבים אינו נדרש כלל לטובת לימוד ספר זה, ואם אתם בעד קיצורי דרך תוכלו לדלג עליהם.

לעומת זאת אם אינכם שולטים עדיין בבסיסי ספירה, בכתובות בזיכרון, בגדלים של משתנים, במחסנית (Stack) ובאופן כללי במבנה המחשב - עיצרו בשלב זה. הספר אינו מיועד למתחילים. לימדו קודם כל את הספר "מבנה המחשב ושפת סף" ורק לאחר מכן המשיכו בלימוד מערכות הפעלה.

מיקוד הספר ושיטת הלימוד

ספר זה יתמקד במערכת ההפעלה Windows של מיקרוסופט, אך התיאוריה שמאחורי הנושאים שמפורטים בו משותפת גם למערכות הפעלה אחרות, כגון Linux. לדוגמה, הרעיון הבסיסי של שימוש ב-Processים ו-Threadים אינו שייך רק ל-Windows, וכך גם נושאים אחרים. הבחירה ב-Windows נעשתה מהסיבות הבאות:

א. Windows היא מערכת ההפעלה הנפוצה ביותר במחשבים אישיים ולכן יש תועלת בהיכרות טובה שלה.

ב. שיפור אוריינות המחשב של הלומדים - עבודה בסביבת Windows, התקנת תוכנות בסביבת Windows, היכרות עם כלי דיאגנוסטיקה של Windows, כל אלו ישפרו את מיומנות המחשב ובאופן עקיף יסייעו למרבית הלומדים בחיי היום יום בעבודה על המחשב.

ג. נגישות - קרוב לוודאי שיש לכם מחשב הפעלה עם מערכת Windows. לימוד בסביבה זו חוסך התקנה של מכונה וירטואלית עם מערכת הפעלה Linux. זו אינה משימה מורכבת במיוחד למי שמנוסה בעבודה על מחשב, אך עלולה להרחיק מי שאין לו אוריינות מחשב טובה ועושה את צעדיו הראשונים.

ד. כאמור, הנושאים התיאורטיים חופפים. מי שמבין איך הדברים עובדים ב-Windows יכול ללמוד ביתר קלות איך הדברים עובדים ב-Linux או במערכות הפעלה אחרות. לדוגמה, פורמט קובץ הרצה. ב-Windows נלמד אודות פורמט PE, ואילו ב-Linux הפורמט של קבצי הרצה נקרא ELF. למרות השוני בפורמטים, הרעיון הבסיסי הוא דומה והבנה של נושא מסויים כפי שהוא ב-Windows מסייעת להבנת הנושא גם במערכות הפעלה אחרות.

שיטת הלימוד של הספר היא Hands On, מתוך אמונה שידע תיאורטי בלבד אינו שוקע לעומק כמו תרגול מעשי. לכן כל נושא ייסקר באופן תיאורטי ולאחר מכן נבצע תרגילים שונים. הנושאים התיאורטיים משותפים ברובם לכל מערכות ההפעלה, ואילו התרגילים יבוצעו על מערכת הפעלה שנבחרה לתרגיל.

אייקונים

בספר אנו משתמשים באייקונים הבאים בכדי להדגיש נושאים ובכדי להקל על הקריאה:

"תרגיל מודרך". עליכם לפתור תרגילים אלו תוך כדי קריאת ההסבר

תרגיל לביצוע. תרגילים אלו עליכם לפתור בעצמכם, והפתרון בדרך כלל לא יוצג בספר

פתרון מודרך לתרגיל אותו היה עליכם לפתור בעצמכם

רקע היסטורי, או מידע העשרתי אחר

הפניה לסרטון

פרקי הלימוד

פרק 1 – כלי דיאגנוסטיקה של Windows. פרק זה יעניק לנו מספר כלים שימושיים שנשתמש בהם בפרקי הלימוד הבאים, תוך כדי היכרות ראשונית עם מושגים חשובים כגון Processים ו-Threadים, זיכרון, קבצים, רגיסטרי. בעקיפין, מטרת הפרק היא לשפר את מיומנות המחשב של הלומדים בסביבת Windows. בסוף הפרק נלמד להסיר תוכנה לימודית שובבה.

פרק 2 - "יישור קו" של נושאים בסיסיים בפעולת המחשב. המטרה היא ללמוד כיצד מאורגנים אזורי זיכרון של תוכנית ולהבין את הקשר בין הקוד שאנחנו כותבים לדרך שבה הדברים מאורגנים לאחר מכן בזיכרון המחשב. מטרה עקיפה של הפרק היא לשפר את מיומנות התכנות בשפת C של הלומדים, מיומנות שהיא הכרחית לטובת ביצוע התרגילים בהמשך הספר. לאחר שניזכר כיצד עובד המעבד, נענה על השאלה כיצד תוכניות נשמרות בזיכרון המחשב? כדי לענות על שאלה זו, ננקוט בגישה מחקרית. נקמפל קטעי קוד שונים ונראה היכן מתבצעת השמירה בזיכרון. הקוד ייכתב בשפת C, קימפול התוכניות יתבצע באמצעות Visual Studio ואילו המחקר יתבצע בעזרת כלי בשם IDA. ההיכרות עם IDA תפתח לנו צוהר לעולם ה- Reverse Engineering המרתק, ומי שיאהב את הטעימה מיכולת המחקר שהתוכנה מציעה יוכל להעמיק באופן עצמאי.

פרק 3 - עולם מערכות ההפעלה. בפרק זה נלמד מה התפקידים של מערכת ההפעלה ונעשה זאת במבט מגבוה, בלי לצלול פנימה אל עומק הנושאים. המושג המרכזי שנלמד הוא Kernel לעומת User. נבצע תרגיל מחקרי נוסף, שיאפשר לנו להבין כיצד מתבצעות הקריאות ל-Windows Kernel.

פרק 4 – Processes and Threads. נבין כיצד מערכת ההפעלה מריצה מספר תוכנות במקביל, מהם היתרונות של שימוש במספר Threadים. נכתוב תוכניות C שיוצרות Processים חדשים ו-Threadים.

פרק 5 – סנכרון. מה קורה כאשר מספר Threadים השייכים לאותו Process צריכים לשתף מידע ביניהם? ואיך מתבצע התיאום בין Processים שונים? ואיך כל זה קשור לעולם המירוצים? נלמד מהו Mutex ומהו Critical Section וכמובן לא נסיים את הפרק בלי לתכנת קוד C שעושה שימוש בדברים החדשים שלמדנו. נפתור בעיה קלאסית מתחום מדעי המחשב- הפילוסופים הסועדים. נמדוד זמני ריצה וכך נלמד איזו שיטת סנכרון יותר יעילה.

פרק 6 - זיכרון. אילו סוגי זיכרונות קיימים? מהו זיכרון וירטואלי? כיצד מתורגמות כתובות וירטואליות לכתובות פיזיות? בפרק זה נלמד מה ההבדל בין זיכרון RAM ו- Cache, מה היתרונות שלהם על פני הדיסק הקשיח. נבין איך עובד מנגנון ה-Paging של מערכת ההפעלה ומהו Page Fault. נלמד מהו Context Switch ומהו המחיר שמערכת ההפעלה משלמת על ביצוע שלו.

פרק 7 - שיתוף זיכרון. נלמד מדוע יש צורך לשתף זיכרון בין Processים. כדי להבין את שיתוף הזיכרון נלמד קודם כל איך מתבצע mapped memory ל-RAM. כפי שאפשר לצפות, לאחר מכן נכתוב ב-C תוכנית שמבצעת מיפוי זיכרון ושיתוף שלו.

פרק 8 – תהליך ה-Compiling, תהליך ה-Linking, יצירת DLL. קיצור של Dynamic Link Library. מדוע בכלל יש צורך ב-DLLים? איך נראה קובץ DLL? נלמד לכתוב DLL ולטעון אותו לתוכנה אחרת, הן באמצעות

טעינה לפני ריצה והן באמצעות טעינה בזמן ריצה. לסיכום נלמד טכניקה שנקראת DLL Injection, המאפשרת למי שמשתמש בה לגרום לתוכנה כלשהי להריץ קוד שלא תוכננה להריץ.

פרק 9 – פורמט PE ותהליך ה-Loading. קבצי הרצה בסיומת EXE בנויים לפי פורמט שמיקרוסופט הגדירה, פורמט זה נקרא Portable Executable. הבנת הפורמט שבו בנויים קבצים אלו תאפשר לנו להבין את המנגנון במערכת ההפעלה שמאפשר את טעינת קבצי ההרצה לזיכרון והרצה שלהם. נענה על שאלות כגון: איך מערכת ההפעלה יודעת מאיזו פקודה להתחיל להריץ את התוכנה? איך טוענים קובץ EXE לזיכרון של תוכנית ואיך מונעים מצב שבו כתובות בזיכרון מתנגשות בין תוכנית שאנחנו מריצים לבין DLL שהתוכנית טוענת. כתרגיל מסכם, נגרום לתכנית להתנהג באופן שונה מכפי שהיא תוכננה להתנהג, באמצעות תהליך שנקרא IAT Hooking.

פרק 10 – מערכות קבצים ותהליך ה-Boot – צפוי בינואר 2021

פרק 11- powershell – צפוי בינואר 2021

תיקונים והצעות לשיפור

יש לכם הערות לגבי חומר הלימוד? מצאתם טעות כלשהי או שאתם רוצים להציע משהו עבור גרסאות עתידיות של ספר הלימוד? מוזמנים לשלוח מייל ל- barakg@cyber.org.il
לימוד פורה ומהנה!

ברק גונן
המרכז לחינוך סייבר, 2020

פרק 1 – ידע מקדים: כלי דיאגנוסטיקה Sysinternals

את לימוד מערכות ההפעלה אנחנו נתחיל דווקא לא ממערכת ההפעלה עצמה. לא נשאל מהם תפקידי מערכת ההפעלה, לא נלמד מושגים ועקרונות, לא נתכנת. לכך יהיה לנו המון מקום בפרקים הבאים. במקום זאת, נתחיל דווקא מלימוד של כלים שימושיים, כלים שנותנים מבט יותר עמוק על מה שמתרחש בקרביים של המחשב שלנו. הסיבה המרכזית לכך היא שהכלים שנלמד כאן ישמשו אותנו כמעט בכל הפרקים הבאים וילוו אותנו בתהליך הלימוד המעשי. אבל, תוך כדי נרוויח כמה דברים חשובים:

1. כשנסיים את הפרק הזה, יהיה לנו כבר ידע שימושי שכבר אפשר להביא לידי ביטוי, לדוגמה בטיפול בתקלות במחשב.
2. שימוש בכלים האלה יחזק את הביטחון שלנו בשימוש במערכת ההפעלה Windows, דבר חשוב במיוחד למי מאיתנו שאין לו ניסיון מעשי רב.
3. אלו כלים סופר מעניינים שכיף ללמוד!

אז במה כן נעסוק בפרק זה? נלמד ארבעה כלים שימושיים:

- RegEdit
- Autoruns
- Process Explorer
- Process Monitor

חבילת הכלים שנלמד מאפשרת הבנה טובה למדי של המתרחש במחשב, טיפול בתקלות ואפילו מציאת נוזקות שנמצאות במחשב שלנו. תוך כדי הלימוד נכיר את אתר VirusTotal המעולה ולתרגול נסיר נוזקה לימודית שנחדיר לתוך המחשב שלנו.

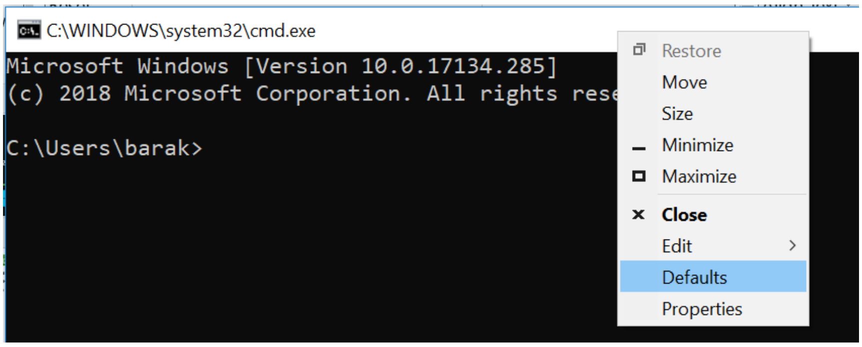
1.1 ה-Registry ותוכנת RegEdit

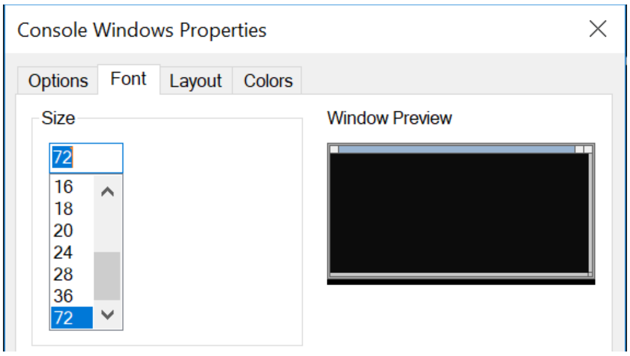
האם אי פעם שמתם לב לכך שכאשר אתם פותחים תוכנה, נראה כאילו התוכנה ממשיכה מאותו מקום בו היא היתה בפעם האחרונה כשסגרתם אותה? פיתחו תוכנה כלשהי, כגון דפדפן או אקסל. כעת שנו את הגודל של חלון התוכנה, ושימו אותו במקום כלשהו על המסך, לדוגמה בפינה השמאלית העליונה. סיגרו את התוכנה. פיתחו שוב את התוכנה. היא מופיעה בדיוק במקום בו השארתם אותה על המסך כשסגרתם אותה. כיצד זה קורה?

בואו נעשה ניסוי נוסף. עקבו אחרי הצעדים הבאים:

1. הקלידו על המקלדת בו זמנית על מקש החלונות (נמצא לצד שמאל של מקש הרווח) ועל התו R. ייפתח לפניכם מסך עם הכותרת Run, בו תוכלו לבחור איזו תכנית ברצונכם להריץ.
2. כיתבו "cmd" ולחצו Enter. ייפתח לפניכם מסך של שורת הפקודה, ה- command line.
3. עמדו עם העכבר על השורה הלבנה למעלה והקליקו קליק ימני. מבין האפשרויות שיפתחו לפניכם בחרו את האפשרות Defaults.
4. שנו את גודל הפונט ל-72.

סיגרו את ה-cmd ופיתחו מחדש (שלבים 1 ו-2).

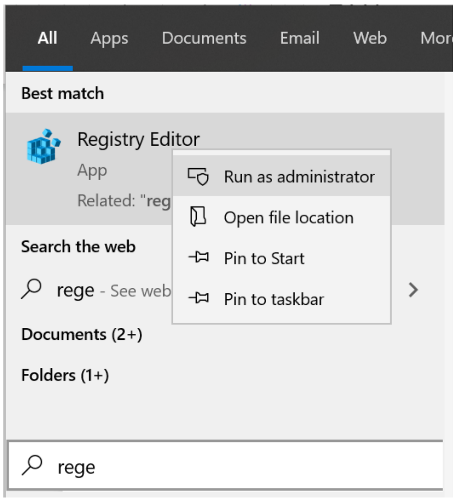




כפי שאתם רואים, גודל הפונט נותר 72. אבל היכן נשמר המידע הזה? איך תוכנה סגורה שומרת מידע עד הפעם הבאה שנפתח אותה?

התשובה היא ה-Registry. זהו מאגר נתונים של מערכת ההפעלה, שמכיל אוסף של שדות וערכים. לכל שדה יש את הערך המתאים לו. ה-Registry מכיל הגדרות של תוכנות ושל מערכת ההפעלה, כגון הגדרות גרפיקה, הגדרות עבור התקני חומרה, אבטחה, ועוד.

לטובת צפיה ועריכה של ה-Registry, מיקרוסופט פיתחה תוכנה בשם Regedit. זהו כלי נסתר מעט, הוא לא מופיע ברשימת התוכניות, ככל הנראה כדי למנוע ממשתמשים לא מנוסים לגרום נזק למחשב שלהם. פיתחו אותה - בריבוע החיפוש של windows הקלידו regedit, ואז הקליקו קליק ימני ובחרו Run as administrator.

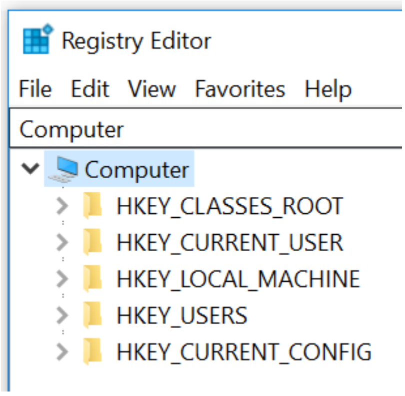


אנחנו לא עומדים לשנות דברים מהותיים, אבל כדי לוודא ששום דבר שתעשו אולי בטעות לא ישפיע על המחשב שלכם, מומלץ לבצע Export (תחת טאב File). כל הערכים ב-Registry שלכם יישמרו לקובץ, אותו תוכלו לטעון בהמשך באמצעות Import וכך לבטל כל שינוי שאולי תעשו.

ברוכים הבאים לרגיסטרי! לאן הגענו?

הרגיסטרי הוא מאגר מידע שמכיל הגדרות הן אודות תוכנות שמותקנות על המחשב שלנו והן עבור מערכת ההפעלה. מאגר המידע הזה נטען לזיכרון ה-RAM (מושג שנלמד עליו בהמשך) בזמן שהמחשב נדלק ומערכת ההפעלה עולה.

הרגיסטרי מחולק למספר ענפים מרכזיים, כפי שתוכלו לראות בעצמכם.



כדי לא להכביר במידע עודף בשלב מוקדם זה, נציין תפקידים של שני ענפים בלבד:

- HKEY_CURRENT_USER – או בקיצור HKCU. במערכת ההפעלה יש למספר משתמשים אפשרות לבצע Log In, יכול להיות מחשב שמשמש לדוגמה אמא ובן, כאשר לכל אחד יש שם משתמש וסיסמה משלו. ה-HKCU אלו הגדרות ספציפיות למשתמש שמחובר כרגע. לדוגמה, אילו אייקונים יופיעו על הדסקטופ? מה תהיה תמונת הרקע על הדקסטופ? אילו תוכנות יופיעו בתפריט ההפעלה? היכן ימוקמו על המסך חלונות שיפתחו? ומה יהיה גודל הפונט של תוכנות שנשתמש בהן? אלו רק חלק זעיר ממגוון ההגדרות האפשריות.
- HKEY_LOCAL_MACHINE – או בקיצור HKLM. אלו הן הגדרות שמשותפות לכל המשתמשים של המחשב. לדוגמה, לאילו התקני חומרה המחשב מחובר? היכן יש דיסקים שאפשר לקרוא מהם קבצים? מהם הסיסמאות של המשתמשים השונים במחשב ואילו הרשאות יש למשתמשים הללו? לדוגמה, במעבדות מחשבים של בתי ספר, נהוג שתלמידים לא יכולים להתקין תוכנות ולמחוק קבצים. ההגדרות הללו ועוד נשמרות ב-HKLM.

עד כאן הדרכה זריזה לגבי הרגיסטרי. כעת פעלו לפי השלבים הבאים:

1. בצד שמאל, בחרו את הענף HKEY_CURRENT_USER ובתוכו את הענף Console.
2. בצד ימין, מיצאו את המפתח שנקרא FontSize. הקליקו עליו כדי לערוך את הערך שלו.
3. שנו את הערך מ-48x0 (כלומר 72) להיות 12x0 (כלומר 18).
4. פיתחו cmd ובידקו שאכן גודל הפונט השתנה!

אם נתקלתם בבעיה שאין לכם הרשאה לשנות ערכים ברגיסטרי, הקליקו קליק ימני על הענף Console ובחרו Permissions. בצעו עריכה להרשאות של המשתמש שלכם כך שיהיה לו Full Control. להלן לינק להסבר כיצד יש לבצע זאת:

https://www.learningpenguin.net/2017/02/02/regedit-in-windows-10-error-writing-the-values-new-contents/

לסיכום מה שראינו עד כה, הרגיסטרי שומר את ההגדרות של התוכנות השונות שאנחנו משתמשים בהם, וגם הגדרות שונות של מערכת ההפעלה. כדי להעמיק את הידע שלנו על הרגיסטרי, נכיר כלי נוסף- Autoruns.

1.2 תוכנת Autoruns

כלי זה, כמו שני הכלים הנוספים שנכיר מיד (Process Explorer, Process Monitor) פותח על ידי חברת Sysinternals. הכלים של Sysinternals מאפשרים הבנה עמוקה של מה שמתרחש אצלנו במחשב תחת מערכת ההפעלה Windows ולכן הם שימושיים מאד למשתמשים ביתיים שמעוניינים לשדרג את ההבנה ויכולת הטיפול בבעיות שונות במחשב. חברת מיקרוסופט יצרנית Windows שיתפה פעולה עם Sysinternals ולבסוף ממש רכשה אותה. מיקרוסופט ממשיכה לפתח את הכלים ומאפשרת הורדה שלהם חינם מהרשת. כמו כן מיקרוסופט משחררת סרטוני הדרכה לגבי שימוש בתוכנות הללו, מעניינים מאד בהקשר של הסרת נוזקות ממחשב אישי. לדוגמה:

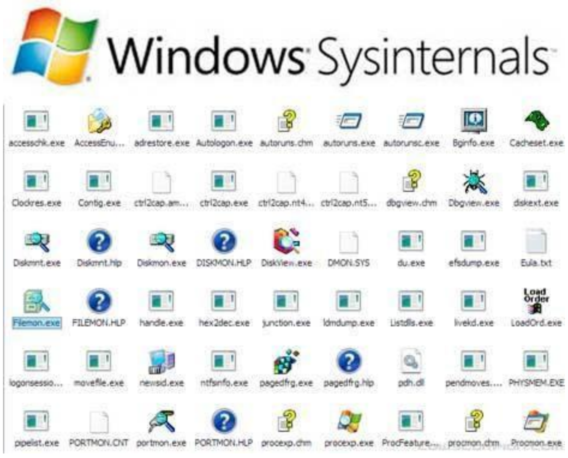
https://www.youtube.com/watch?v=7heEYEbFim4

חפשו בגוגל "sysinternals tutorial video" או "Mark Russinovich sysinternals".

את חבילת התוכנות של Sysinternals ניתן להוריד מהקישור:

https://download.sysinternals.com/files/SysinternalsSuite.zip

לאחר שתעתיקו את התוכנות לתיקיה לבחירתכם, יהיה ברשותכם אוסף כלים שימושיים. הכלי הראשון שנעסוק בו הוא כאמור Autoruns.



כדי להבין את תפקידו של Autoruns, חשוב לדעת כי חלק מהערכים שנמצאים ברגיסטרי קשורים להרצה אוטומטית של תוכנות. נסקור חלק מהמקרים שבהם מחשב צריך להריץ תוכנות באופן אוטומטי:

- ביצענו התקנה של תוכנה. ההתקנה דורשת הדלקה מחדש של המחשב כדי להסתיים. התוכנה תרשום לרגיסטרי, כדי שבזמן עליית המחשב יסתיים תהליך ההתקנה אוטומטית, בלי שהמשתמש יצטרך שוב פעם להריץ את תוכנת ההתקנה.
- אנחנו רוצים שבכל פעם שהמחשב שלנו עולה, יופעל אוסף תוכנות באופן אוטומטי, בלי שנצטרך להפעיל אותן אחת אחת. לדוגמה, יש היגיון שנרצה שהאנטי-וירוס ירוץ אוטומטית. ייתכן שתרצו שתוכנות שיתוף קבצים כמו Dropbox יעלו וידאגו שהקבצים שלכם מסונכרנים. ייתכן שתרצו שגם תוכנות כמו Skype או Spotify יעלו אוטומטית.
- ישנן משימות שצריכות להתבצע באופן שגרתי. לדוגמה, בדיקה אם יש עדכון לאנטי-וירוס, הגיוני לבצע אחת ליום. נרצה שהדברים יתוזמנו ויתרחשו אוטומטית, בלי שהמשתמש יצטרך להכניס לעצמו ליומן להריץ בדיקת עדכונים.

הכלי Autoruns מבצע מיפוי של כ-200 מקומות ברגיסטרי שקשורים להרצה אוטומטית של תוכנות. יש לכלי חשיבות משני היבטים. הראשון, אם אנחנו רוצים לדעת מה אולי מאט את עבודת המחשב שלנו וגורם לכך שלוקח זמן רב מהדלקת המחשב עד שניתן לעבוד, Autoruns יכול לספק לנו מידע שימושי. ההיבט השני, הוא שנוזקות עלולות להשתמש ברגיסטרי כדי לגרום למחשב להריץ אותן בכל פעם שהוא מודלק. בעזרת כלי שממפה לנו כל מה שרץ אוטומטית, הרבה יותר קל למצוא דברים חריגים ולנטרל אותם.

הריצו את Autoruns בתור מנהל מערכת. אין זה משנה אם אתם מפעילים את Autoruns או Autoruns64 (גרסה למערכות הפעלה של 64 ביט, נלמד בהמשך). השאלות הבאות יסייעו לכם ללמוד בעצמכם על הכלי:

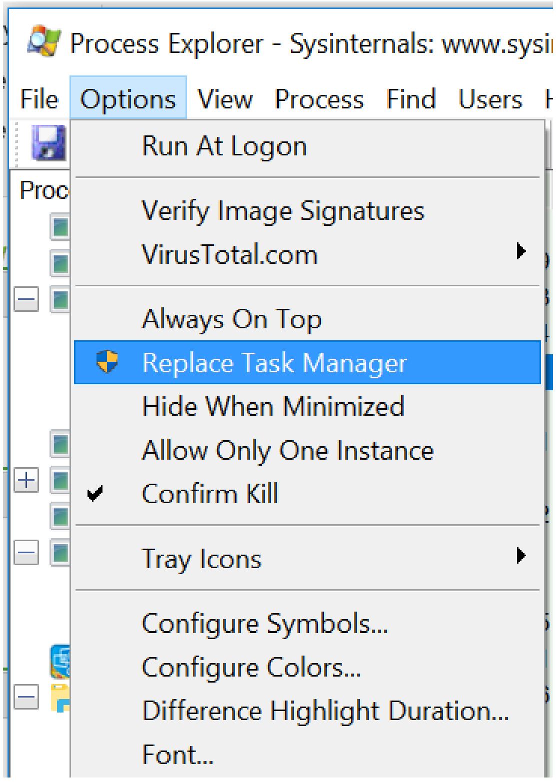
1. תחת Everything, מיצאו את כל התוכנות שמופעלות אוטומטית.
2. מיצאו את ה-Scheduled Tasks שלכם.
3. אתר Virus Total הוא אתר שמרכז מידע על נוזקות, מידע שנאסף מאנטי-וירוסים שונים. ניתן להגיש לאתר קובץ, ולקבל בחזרה מידע האם הקובץ מזוהה כנוזקה על ידי האנטי-וירוסים השונים. אתר זה הוא שימושי ביותר אם אתם חושדים שקובץ כלשהו שיש לכם על המחשב עלול להיות נוזקה. בטאב Options, תחת Scan options, הוסיפו בדיקה של התוכנות באתר Virus Total. כעת המתינו עד שכל המידע בטאב Everything נשלח ומתקבל. האם זוהו אצלכם תוכנות זדוניות? שימו לב, שעלולים להיות פספוסים. אם רק אנטי-וירוס אחד או שניים מזהים קובץ כנוזקה, ייתכן שזו טעות. אם הרבה אנטי-וירוסים מזהים קובץ כנוזקה, סביר שנדבקתם.
4. מצאתם תוכנה שחשודה כתוכנה זדונית? הקליקו קליק ימני על השורה בה היא כתובה ובחרו באפשרות Search Online. תקבלו מידע על התוכנה ומה רמת הסיכון שלה. אין אצלכם תוכנה חשודה? בחרו את החיפוש על תוכנה אחרת כלשהי שאתם רוצים לדעת מה תפקידה.
5. תוכנות שנכתבו על ידי יצרן תוכנות מוכר אמורות להיות חתומות בחתימת אבטחה. החתימה נוצרת באמצעות פונקציה מתמטית שעוברת על כל המידע בקובץ (מוזמנים לקרוא על digital signature). החתימה הדיגיטלית מבטיחה שהקוד של התוכנה לא שונה מרגע החתימה, וכך מקשה על גורמים

זדוניים לקחת תוכנה תמימה, לשתול בתוכה קוד זדוני ולהפיץ אותה בתור התוכנה המקורית כביכול. אם יש לכם תוכנות שאין להן חתימה דיגיטלית ורצות אוטומטית, ייתכן שמדובר בנוזקות. כדי לראות למי יש חתימה דיגיטלית, בטאב Options, תחת Scan Options, הוסיפו בדיקה חתימות.

1.3 תוכנת Process Explorer

המושג Process חדש לנו. ישנו פרק שלם שמוקדש לProcessים, אך בשלב זה לא נרצה להיכנס לכל הפרטים. לצורך מה שנלמד כעת, Process הוא תוכנה שרצה על המחשב. התוכנה Process Explorer נותנת לנו מבט מעמיק על כל מה שמחשב שלנו מריץ כרגע. לבטח אתם מכירים את ה-Task Manager הותיק, שניתן להגיע אליו באמצעות צירוף המקשים Alt+Ctrl+Del. התוכנה Process Explorer היא, כפי שאומרים, "Task Manager על סטרואידים".

עקב היכולות הגבוהות של Process Explorer, מומלץ מאד לשנות את הגדרות ברירת המחדל, כך שלחיצה על Alt+Ctrl+Del תפתח אותו.



אנחנו נלמד חלק מהיכולות המרכזיות של הכלי באמצעות מספר ניסויים קטנים. חלק אחר של היכולות נלמד מאוחר יותר, תוך כדי פרקי הלימוד עצמם.

1.3.1 תצוגת Processים

הכלי מאפשר תצוגה של כל הProcessים שרצים כעת על המחשב שלנו. הריצו תוכנה כלשהי, לדוגמה דפדפן Chrome. התוכנה תופיע לכם במסך הראשי. מיצאו את המידע על התוכנה שהרצתם. מה שם החברה שכתבה את התוכנה?

כפי שאתם רואים, יש צבעים שונים לשורות. כל צבע מסמן משהו:

- ירוק – Process חדש. פיתחו Chrome, תמצאו שה-Process שלו צבוע ירוק לזמן קצר.
- אדום- Process שעומד להיסגר. ליחצו על X בחלון התוכנה של Chrome, הצבע ישתנה לאדום ואז ייעלם.
- סגול בהיר- Processים שרצים באותו חשבון משתמש שבו רץ הכלי Process Explorer כרגע.
- ורוד – Processים של מערכת ההפעלה.
- אפור- Processים שנמצאים במצב מושהה. פיתחו את Chrome. כעת דרך Process Explorer העבירו אותו למצב מושהה (קליק ימני ואז Suspend). נסו לעבור אל החלון של Chrome, תראו שאין תגובה. החזירו את הProcess לפעילות (כפתור ימני ואז Resume).
- סגול – קבצי הרצה דחוסים. אלו קבצים שהמהות האמיתית שלהם מתבררת רק בזמן ריצה, ולכן הם מאד אופייניים לנוזקות. אם נתקלתם בקובץ כזה שרץ אצלכם, מהרו לבדוק אותו באתר Virus Total וחפשו עליו פרטים נוספים (קליק ימני ו-Search Online).

1.3.2 מידע על ניצול המשאבים

הכלי מציג בין היתר את אחוז ה-CPU (קיצור של Central Processing Unit) וכמות ה-Physical Memory. מהם, ולמה הם חשובים?
אחוז ה-CPU מראה עד כמה המעבד שלכם עמוס. אולי צפיתם בסרט "זמנים מודרניים" של צ'רלי צ'פלין, בסצנה שבה צ'פלין הוא פועל ייצור שלא עומד בעומס?

https://www.youtube.com/watch?v=6n9ESFJTnHs

המעבד הוא כמו צ'פלין שעובד על פס ייצור, ומגיעים אליו חלקים שהוא צריך להרכיב. אם צ'פלין עובד ב-100% עומס, זה אומר שאין לו זמן לטפל באף חלק נוסף. אם נכניס חלקים נוספים לפס הייצור, הוא עדיין יעבוד ב-100% עומס, אבל כעת יהיו לו פספוסים וחלק מהחלקים לא יורכבו. באותו אופן, מעבד שעובד בעומס שקרוב ל-100% לא מצליח למלא את כל המשימות שלו. בכלל, מעבד שעובד באחוזי CPU נמוכים יגיב לפקודות משתמש מהר יותר, כיוון שאין משימות שתופסות אותו ו"מתחרות" על המשאבים שלו.
בידקו בשורה העליונה: מה אחוז ה-CPU שהמחשב מנצל כרגע? אם האחוז גבוה, יש שתי אפשרויות: או שהמעבד מריץ תוכנות שאין לכם צורך בהן, או שיש לכם מעבד מיושן וכדאי לשדרג את המחשב.
ה-Physical Memory מודד בכמה זיכרון מסוג RAM המחשב שלכם משתמש כרגע. בהמשך נלמד על זיכרון RAM, אך כעת פשוט דמיינו שיש לוח מחיק שעומד לרשותכם ואתם משתמשים בו לצורך כלשהו, נניח כדי לפתור תרגילים. אם תפתרו תרגילים רבים, מתישהו הלוח המחיק יתמלא. בשלב זה יש לכם מספר אפשרויות:

- למחוק את הלוח ולאבד את התרגילים שפתרתם כבר
- להעתיק את הפתרונות למחברת
- להשיג לוח מחיק נוסף

ה-RAM הוא כמו לוח מחיק, שכל התוכנות רוצות לעשות בו שימוש. אם ה-RAM קטן מדי, נצטרך לפתור את הבעיה באחת מהדרכים בה פתרנו את בעיית הלוח המחיק: למחוק מידע, להעתיק אותו למקום אחר שבו יש מקום רב יותר (פעולה שלוקחת זמן רב יחסית), או להגדיל את כמות ה-RAM.

תרגיל 1.1 תרגול Process Explorer

ליחצו על Ctrl+I או הגיעו דרך התפריט אל System Information. בידקו את הדברים הבאים:

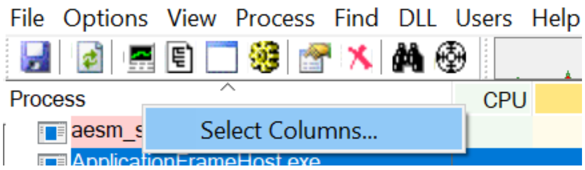
- מה אחוז ה-CPU שהמחשב שלכם מנצל?
- הקליקו על טאב ה-CPU. כמה מעבדים יש לכם במחשב? המספר רשום תחת הערך Topology.
- מהי כמות ה-Physical Memory שאתם מנצלים כרגע?
- כמה זיכרון RAM יש לכם בסך הכל במחשב? כדי למצוא, הקליקו על טאב הזיכרון. הערך רשום תחת Physical Memory, לצד הכיתוב Total. אם הזיכרון הפיזי שבשימוש מתקרב לסך כל הזיכרון RAM שיש לכם במחשב, אתם צפויים לחוש האטה בעבודת המחשב. אם אתם מגיעים למצב זה לעיתים קרובות, כדאי שתשקלו לרכוש RAM נוסף.

1.3.3 מידע מפורט על Processים

התצוגה של ה-Processים נפתחת עם מספר שדות ברירת מחדל:

- Process- שם קובץ ההרצה, יחד עם האייקון שלו אם ישנו כזה
- CPU – אחוז הזמן שהמעבד הקדיש ל-Process הזה מאז העדכון האחרון של נתון ה-CPU
- Private Bytes- כמות הזיכרון שמערכת ההפעלה הקדישה ל-Process הזה, לא כולל זיכרון שמערכת ההפעלה שיתפה גם עם Processים אחרים (בהמשך, כשנלמד על DLLים, נבין יותר טוב)
- Working Set- סך כל כמות הזיכרון ב-RAM של-Process יש גישה אליו
- PID- קיצור של Process I.D, כלומר מספר "תעודת הזהות" של ה-Process. לכל Process יש מספר בין 0-65535, שניתן לו על ידי מערכת ההפעלה ברגע שהוא נוצר
- Description – תיאור ה-Process
- Company Name- שם יצרן התוכנה. אנחנו מצפים שרוב הProcessים שלנו יהיו של מיקרוסופט

אפשר לשנות את התצוגה כך שהיא תכלול פרטים נוספים. הקליקו קליק ימני על העמודה Process ובחרו באפשרות Select Columns.



בתפריט שנפתח לכם, הכנסו ל- Process Image ובחרו באפשרות Window Title. נוספה לכם עמודה עם כותרת החלון של כל תוכנה שפתוחה אצלכם! בהמשך נלמד עמודות שימושיות נוספות.

1.3.4 מציאת Process ספציפי

פיתחו דפדפן כרום. בתוך Process Explorer גשו אל סמל המטרה (נמצא למעלה, לצד סמל המשקפת). לחצו על המטרה ואל תשחררו את העכבר. סמל העכבר שלכם הפך למטרה. גררו אותו אל כרום ושחררו את הלחיצה. ה-Process Explorer יתמקד מיד בProcess של כרום!

בדרך זו קל למצוא מי הProcess שמקפיץ חלון מסויים, כגון הודעת שגיאה.

1.3.5 "חיסול" Processים

לאחר שבסעיף הקודם מצאתם בקלות את הProcess chrome, הקליקו עליו קליק ימני ובחרו באפשרות Kill Process. אפשרות זו היא שימושית אם Process מסויים "תקוע" ולא ניתן לסגור אותו באמצעות הקלקה על חלון המשתמש, או אם זהו Process שאין לו חלון משתמש ולמרות זאת היינו רוצים לסגור אותו.

1.3.6 מציאת Processים שמחזיקים קובץ פתוח

הכלי Process Explorer מאפשר לבדוק אילו תוכנות עושות שימוש במשאב מסויים. המשאב יכול להיות DLL (על כך נלמד בהמשך) או קובץ.

פיתחו באמצעות word קובץ כלשהו. בתוך Process Explorer, לחצו Ctrl+F או על טאב Find בתפריט. בחלון החיפוש, חפשו חלק משם הקובץ שפתחתם באמצעות word. איך קוראים לProcess שמצאתם? אם מצאתם את winword, הצלחתם.

1.3.7 בדיקה אימות ואבטחה

בדומה לכלי Autoruns אותו סקרנו, גם ל-Process Explorer יש אפשרות לבצע בדיקות חתימה ובדיקות לגילוי נוזקות.

הקליקו קליק ימני על העמודה Process ובחרו באפשרות Select Columns. הוסיפו לתצוגה את השדות "Virus Total" ו- "Verified Signer". אם יש Process לא חתום, שיש עליו אתרעות של Virus Total, ייתכן מאד שנדבקתם בתוכנה זדונית.

1.4 תוכנת Procmon

השם Procmon הוא קיצור של Process Monitor. הכלי הזה הוא "יומן אירועים" של כל מה שמתבצע במחשב שלנו, יחד עם יכולת חיפוש וסינון.

פתיחת Procmon סתם כך תציף אותנו במידע אינסופי כמעט, מכיוון שכל ארוע, Event שמתבצע במחשב שלנו, מתועד. ויש כמות לא קטנה של Eventים.

מה הנתונים שאנחנו מקבלים על כל Event? ברירת המחדל כוללת:

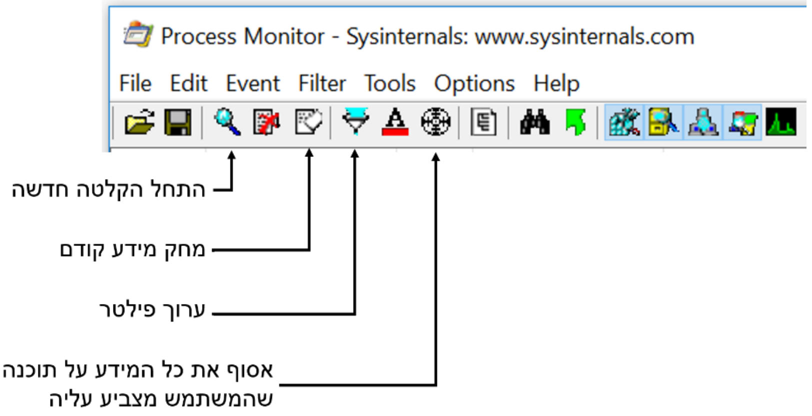
- שעה
- שם הProcess שביצע את ה-Event
- ה-PID של הProcess שביצע את ה-Event
- Operation: מה סוג ה-Event
- Path: מה הנתיב של המשאב שה-Operation פועל עליו. לדוגמה, פתיחת קובץ- מה הנתיב של הקובץ? לדוגמה, גישה לרגיסטרי- לאיזה ענף ברגיסטרי בוצעה הגישה?
- תוצאת Result. האם הפעולה התבצעה בהצלחה או שחלה בעיה כלשהי, כגון קובץ לא נמצא?
- Detail- פרטים נוספים

תרגיל 1.2 תרגיל מודרך Procmon

התרגיל מבוסס על המדריך הבא מתוך אתר MSDN:

https://docs.microsoft.com/en-us/archive/blogs/appv/process-monitor-hands-on-labs-and-examples

כהכנה לתרגיל, פיתחו את Procmon והתחילו הקלטה חדשה. להלן מדריך ללחצני השליטה הראשיים:



פיתחו את תוכנת notepad, ושנו את גודל הפונט וסוג הפונט (כדי לבצע את השינוי הכנסו לטאב Format ובתוכו בחרו Font). סיגרו את notepad ועיצרו את ההקלטה.
כעת ברצוננו למצוא היכן ברגיסטרי נשמרות ההגדרות של גודל הפונט וסוג הפונט?
הטכניקה היא לצמצם את החיפוש ככל האפשר. ראשית, נסיר את כל הProcessים שאינם notepad. ישנן שתי דרכים לעשות זאת:

1. למצוא שורה כלשהי ששייכת לProcess notepad, קליק ימני ולבחור Include ואז מבין האפשרויות לבחור Process Name
2. להכנס לתוך עורך הפילטרים Ctrl+L ולהוסיף פילטר "Process name is notepad.exe"

בשתי הדרכים התוצאה תהיה זהה-

Column	Relation	Value	Action
Process Name	is	notepad.exe	Include

וכמובן שרק ארועים שקשורים ל- notepad.exe יופיעו.

דרך טובה לראות שאתם אכן מקליטים ארועים ומפעילים עליהם פילטר באופן מוצלח היא לבדוק בפינה השמאלית התחתונה כמה ארועים נקלטים וכמה מתוכם מפולטרים, לדוגמה:

Showing 265 of 404,955 events (0.065%)

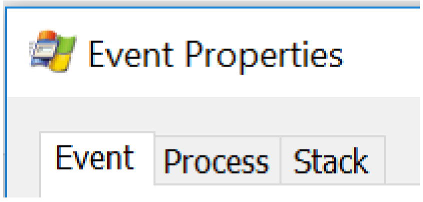
אם אף ארוע לא מוקלט, סימן שלא התחלתם הקלטה. אם אף ארוע לא מוצג, סימן שהפילטר שלכם לא תפס שום ארוע, וייתכן ששגיתם במשהו. די בכך שכתבתם את שם הProcess עם שגיאת כתיב והפילטר לא יעבוד.

כעת נוסיף פילטר נוסף- אנחנו רוצים לצפות רק בארועים שקשורים לכתיבה לרגיסטרי.

בשורת האייקונים העליונה ישנם מספר לחצנים שכל אחד מהם מבטל או מאפשר סוג מסויים של ארועים. השאירו לחוץ רק את לחצן ארועי הרגיסטרי:

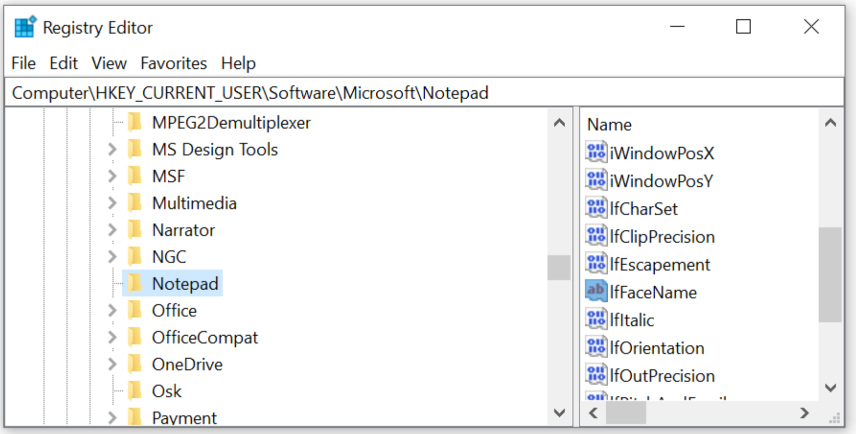
כעת נתחיל לבטל ארועי רגיסטרי שאינם קשורים. עמדו על ארועים כגון RegEnumKey, RegCloseKey, RegOpenKey ובאמצעות קליק ימני ואז Exclude Operation צרפו אותם לפילטר. נותרנו עם ארועים מסוג RegSetValue.כל הארועים הללו נמצאים בנתיב HKCU\Software\Microsoft\Notepad, שהוא נתיב מאד הגיוני עבור רישום לרגיסטרי לתוכנת Notepad. מבין מעט הארועים שפילטרנו, תמצאו בקלות- על פי התיאור – את הארוע בו שיניתם את הפונט: שם הפונט החדש יופיע בתוך התיאור.

הקלקה על ארוע כלשהו תאפשר לקבל מידע מפורט יותר על הארוע.



– תחת Process יופיעו Modules- מאפשרים לבדוק האם הProcess קורא לקוד חשוד, שם מודול שאינו מוכר או שהיצרן שלו אינו מוכר
– תחת Stack תופיע שרשרת הקריאות לפונקציות. נשתמש ביכולת זו בהמשך.

לסיום, הקליקו קליק ימני על הארוע הנ"ל, ולחצו Jump To. תגיעו באופן מיידי אל ערך הרגיסטרי בתוך RegEdit. זהו, סיימנו!



תרגיל 1.3 **תרגילי** Procmon

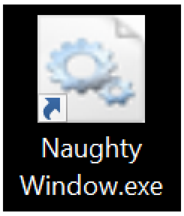
כעת תורכם לבצע כמה משימות:

1. הציגו את כל מה שמבצע chrome.exe
2. הציגו את כל מי שמבצע WriteFile
3. הציגו את כל מי שמבצע RegQueryValue
4. איך נקראת הפונקציה של ntdll ש-RegQueryValue קוראת לה? (טיפ: בידקו בתוך ה-Stack)

הציגו את כל מי שקורא את הרגיסטרי במיקום HKCU\Console\FontSize ופיתחו cmd – מה שם ה-Process?

(טיפ: בצעו Include ל-Path)

תרגיל 1.4 תרגיל מסכם – Naughty Window



רקע

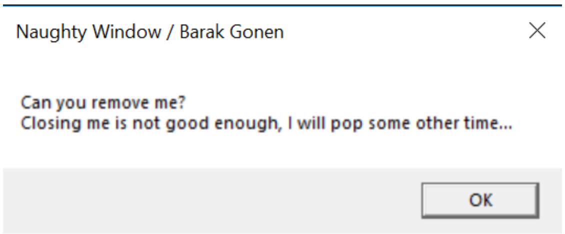
בתרגיל זה תתקינו תוכנה לימודית שובבה, שפיתחנו במיוחד לצורך התרגיל. התוכנה טורדנית מעט, אך היא אינה עושה נזק למחשב. מטרתכם היא להסיר אותה. מיד תראו שמי שמתקין את התוכנה הם אתם, ויכול להיות שתשאלו את עצמכם האם זה משקף את המציאות? כלומר, מדוע שמישהו יתקין תוכנה בעייתית במחשב שלו בעצמו? למעשה זה נפוץ מאוד. נוזקות רבות מציגות את עצמן כתוכנות שמבצעות דברים טובים ומשכנעות אותנו להוריד ולהתקין אותן. לאחר שהתקנו אותן צריך להתמודד איתן ולמצוא דרך להסיר אותן.

התקנת התוכנה הלימודית

הורידו את התוכנה הלימודית מהלינק הבא:

https://data.cyber.org.il/os/NaughtyWindow.msi

הקליקו על חבילת ההתקנה Naughty Window. יווצר קיצור דרך על ה-desktop שלכם. הפעילו אותו באמצעות לחיצה ימנית ובחרו Run as administrator (חשוב מאד! בלי הרשאה מתאימה ההתקנה לא תעבוד כמו שצריך). זהו סיימתם, כעת נסו להסיר אותה.



דגשים:

- השתמשו בכלים שנלמדו בשיעור
- חושבים שהסרתם את התוכנה? הדליקו את המחשב מחדש ובידקו התוכנה כבר לא מקפיצה הודעות למסך

מה להגיש?

עליכם לכתוב מדריך להסרת התוכנה לאחר שהותקנה. כלומר, אין צורך להסביר את כל שלבי המחקר שלכם אלא מהי הדרך הקצרה והיעילה ביותר להסיר את התוכנה. היכן ללחוץ, מה לשנות.

תרגיל 1.4.1 תרגיל למתקדמים

כיתבו את התוכנה בעצמכם!

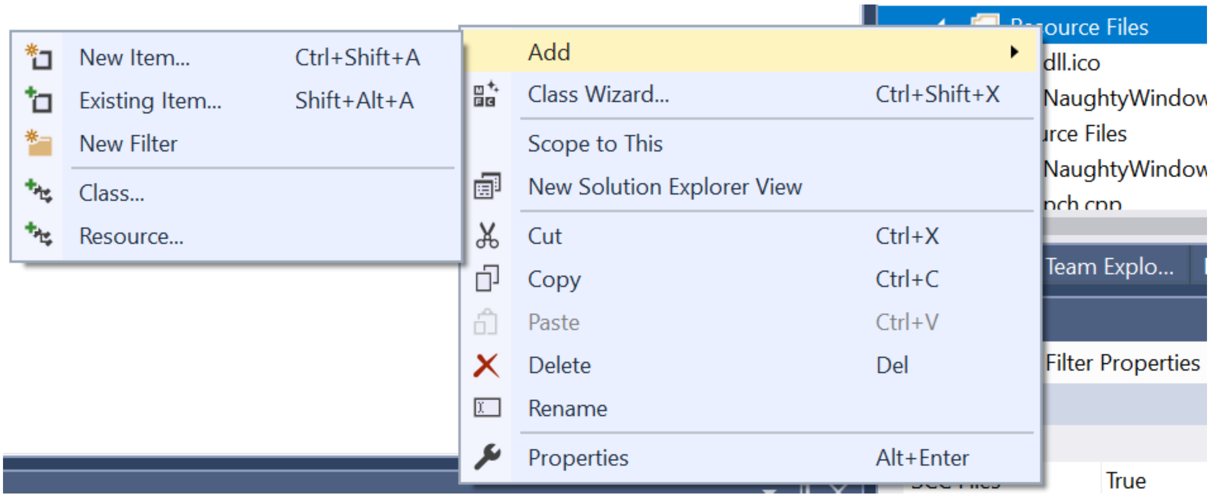
שלבים:

- לימדו אילו פקודות WinAPI כותבות לרגיסטרי ובחרו להיכן ברגיסטרי לכתוב את הערכים הנחוצים לכם
- להלן הסבר על איך מוסיפים אייקון לתוכנה
- להלן הסבר על איך יוצרים חבילת התקנה

הוספת Resourceים

בלי resourceים התוכנה שלנו חסרת icon ופרטים.

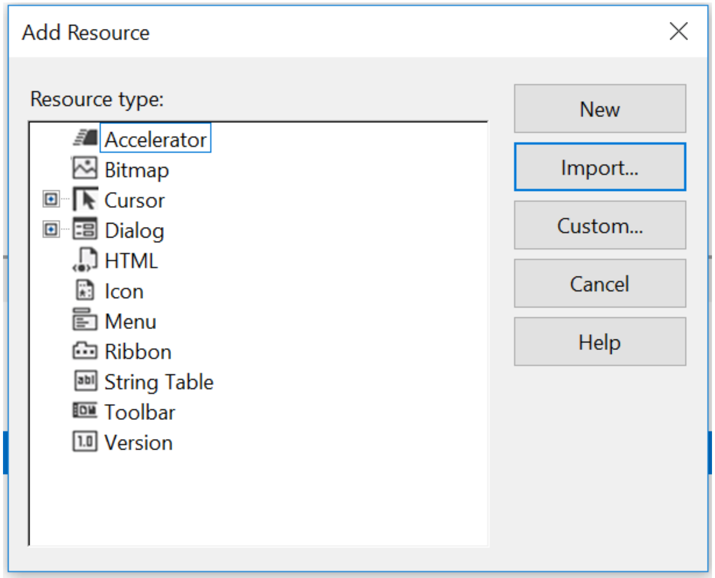
בתוך Solution Explorer נעמוד על Resource Files ונקליק קליק ימני. נבחר Add ואז Resource



ראשית נבחר קובץ Version, נזין לתוכו את כל השדות כגון יצרן, גרסה.

הכנת קובץ icon: נוריד קובץ icon בפורמט jpg. האתר ico converter ממיר אותו לקובץ ico. ניתן לקבוע גודל כרצוננו.

כעת נוכל לעשות add לקובץ ה-ico. לאחר ביצוע add ואז resource, בוחרים import ומשנים את סוג הקובץ ל-ico.



יצירת Installer

לטובת התקנה אוטומטית, נוריד תוכנת Advanced Installer (גרסה 15.3 ומעלה), אשר משתמשת ב- Visual Studio. התוכנה מקבלת את קובץ ה-sln של visual studio ויוצרת בעצמה את ה-build בתצורה הנדרשת. לתוכנה יש wizard נוח. יש לבחור באופציה של יצירת msi ומשם להמשיך לפי ההוראות.

https://www.advancedinstaller.com/download.html

1.5 סיכום

בפרק זה למדנו אודות הרגיסטרי, הבנו את המבנה ואת התפקיד שלה. ראינו באמצעות מספר דוגמאות, שחלקן חקרנו בעצמנו, כיצד הרגיסטרי מאפשר לשמור קונפיגורציה של תוכניות.
באמצעות Autoruns ביצענו מיפוי למקומות ברגיסטרי שנוזקות עלולות "להתלבש" עליהן וכך לרוץ שוב ושוב בכל פעם שהמחשב נדלק. למדנו להפעיל שתי תוכנות חשובות נוספות מבית SysInternals, תוכנות שמאפשרות לנו ניתוח טוב למדי של מה שמתרחש במחשב שלנו. כלים אלו, Process Explorer ו- Process Monitor, ישמשו אותנו בפרקים הבאים. המשך לימוד מוצלח!

פרק 2 – מחקר אזורי הזיכרון של תוכנה

בפרק זה נבצע "יישור קו" של נושאים בסיסיים בפעולת המחשב. לאחר שניזכר כיצד עובד המעבד, נענה על השאלה ממה מורכבות תוכנות מחשב? כדי לענות על שאלה זו, ננקוט בגישה מחקרית. נקמפל קטעי קוד שונים ונראה היכן מתבצעת השמירה בזיכרון. הקו
ד ייכתב בשפת C, קימפול התוכניות יתבצע באמצעות Visual Studio ואילו המחקר יתבצע בעזרת כלי בשם IDA. מטרה עקיפה של הפרק היא לימוד שפת C תוך כדי התנסות. הפרק אינו מדריך לשפת C ואם אינכם מכירים את השפה סביר שתצטרכו ללמוד דברים מסויימים בכוחות עצמכם, אך הפרק יעניק לכם כמה קטעי קוד בסיסיים ופשוטים שתוכלו לחקור ולהתקדם באמצעותם. ההיכרות עם IDA תפתח לנו צוהר לעולם ה-Reverse Engineering המרתק, ומי שיאהב את הטעימה מיכולת המחקר שהתוכנה מציעה יוכל להעמיק באופן עצמאי.

2.1 אופן פעולת המעבד

2.1.1 שפת מכונה

המעבד מריץ פקודות בשפת מכונה. שפת מכונה היא שפה של אחדות ואפסים והיא קשורה באופן הדוק למעבד הספציפי שאנחנו עובדים איתו. לדוגמה, הפקודה "העתק את X לתוך Y" תיכתב באופן אחד במעבדים מתוצרת שונה. לשם המחשה, פקודת העתקה במעבד מתוצרת חברה א' עשויה להיות "10100000" ואילו פקודת העתקה במעבד מתוצרת חברה ב' עשויה להיות "10001100". נדגיש כי יש מגוון רב של פקודות העתקה, והביטים משתנים לפי סוג המקור והיעד, אך העקרון הוא שלכל מעבד יש פקודות משלו ולא תמיד יש תאימות בין המעבדים השונים. למעשה, גם בין משפחות מעבדים שונים של אותו יצרן יש הבדלים. לדוגמה במעבדי אינטל יש הבדלים בין פקודות שמיועדות למשפחת מעבדי ה-32 ביט לעומת מעבדי ה-64 ביט. בדרך כלל ישנה תאימות לאחור, כלומר מעבד של 64 ביט יודע להריץ גם פקודות של 32 ביט, אך ההיפך אינו מתקיים מעבד 32 ביט לא יודע להריץ פקודות של 64 ביט, גם אם הן של אותו יצרן מעבד.

עם זאת יכול להיות גם שחברות שונות יאמצו את אותן הפקודות, כך לדוגמה חברת AMD מייצרת מעבדים עם סט פקודות שתואם את מעבדי אינטל, כלומר גם משמעות הפקודות וגם שפת המכונה הם תואמים. לכן קוד שמיועד לרוץ על מעבד אינטל יכול לרוץ גם על AMD.

2.1.2 תזכורת- מה משמעות מעבד 32 למול 64 ביט?

לכל מעבד יש חלקי חומרה הנקראים רגיסטרים ומשמשים לביצוע כל פעולות החישוב, הגישה לזיכרון והרצת התוכנית. מספר הרגיסטרים שונה בין דורות שונים של מעבדים, וגם גודל הרגיסטרים שונה בין דורות המעבדים. כאשר אומרים "מעבד 32 ביט" מתכוונים לכך שגודל הרגיסטרים שלו הוא 32 ביט. למעבד 64 ביט יש, כצפוי, 64 ביט בכל רגיסטר. מה משמעות הדבר?

רגיסטר גדול יותר יכול להחליף יותר מידע עם הזיכרון בכל גישה לזיכרון. נניח שיש לנו זיכרון בגודל 16 בתים (בית= 8 ביט) ואנחנו מעוניינים להעתיק אותו למקום אחר בזיכרון. רגיסטר של 32 ביט מסוגל לטפל ב-4

בתים של זיכרון בכל פעם, לכן נצטרך לגשת אל הזיכרון 8 פעמים. מתוכן 4 פעמים יהיו קריאות מהיזכרון ממנו מעתיקים, ו- 4 יהיו גישות לזיכרון אליו מעתיקים. לעומת זאת, רגיסטר של 64 ביט יוכל לבצע את אותה עבודה בחצי מכמות הגישות לזיכרון וכך לחסוך לנו זמן רב.

הבדל משמעותי נוסף הוא שינוי בגודל פס הכתובות (Address Bus). למעבד 32 ביט יש פס כתובות בגודל 32 ביט. כלומר ניתן לייצג 2 בחזקת 32 כתובות שונות, או במילים פשוטות, בערך 4GB (ג'יגה בתים). המשמעות היא שגם אם יש למחשב שלנו יותר מ-4GB זיכרון, יש קושי לנצל אותו. המעבד פשוט לא יודע איך לגשת לזיכרון שאי אפשר לייצג את הכתובת שלו בפס הכתובות. בעבר 4GB נחשב דמיוני, אולם כיום קשה לרכוש מחשב שיש בו רק 4 ג'יגה זיכרון RAM, שלא לדבר על דיסק קשיח.

יצרנית המעבדים אינטל פיתחה שיטה לפרוץ את מגבלת ה-4 ג'יגה ולתמוך בעד 64 ג'יגה זיכרון גם במעבדי 32 ביט. השיטה נקראת Physical Address Extension, או בקיצור PAE. כדי להבין את השיטה יש צורך להבין כיצד בנויה כתובת וירטואלית ולכן לא נכנס לפרטים. מה שחשוב להבין בשלב זה הוא שיש קשר בין כמות הביטים בפס הכתובות לבין כמות הזיכרון שאפשר לגשת אליו. אם אתם סקרנים כיצד עובד PAE, מומלץ לקרוא על הנושא לאחר שתסיימו את הפרק אודות זיכרון.

כדי לעבוד עם כתובות של 64 ביט, לא די בכך שהמעבד יתמוך בכך. יש צורך שגם מערכת ההפעלה תתמוך בכך. לכן למרבית מערכות ההפעלה יש גרסאות 32 ו-64 ביט. מה בדיוק צריכה לעשות מערכת ההפעלה כדי לתמוך ב-64 ביט? על כך נלמד בפרק שעוסק בזיכרון.

2.1.3 כיצד עובד המעבד

המעבד מריץ רצפים של פקודות בשפת מכונה, המונח הלועזי הוא Instruction. יכול להיות Instruction להעתקה של ערך לתוך רגיסטר, Instruction לחיבור של שני רגיסטרים וכו'. שפת אסמבלי היא תרגום פשוט של רצפים בשפת מכונה לפקודות שאדם יכול לקרוא, הנה כך נראות מספר Instructions בשפת אסמבלי שנכתבו עבור מעבד אינטל 32 ביט:

```
mov     eax, 2
mov     ebx, 3
add     eax, ebx
```

בזיכרון של המחשב הפקודות הללו לא שמורות כפי שהן כאן, אלא שמור התרגום שלהן לשפת מכונה.

בשפת מכונה, הפקודות הללו ייראו כך:

```
B8 02 00 00 00
BB 03 00 00 00
01 D8
```

בהמשך, כאשר נתבונן בזיכרון המחשב באמצעות תוכנות שונות, לא נראה רצף של אחדות ואפסים אלא ספרות הקסדצימליות.

לנו בני האדם קשה לעבוד עם מספרים בינאריים, קל מאד להתבלבל. נסו להקריא ולזכור את רצף המספרים הבא: "1001111111100011". לא נעים! מצד שני, אם נתרגם את אותו רצף לספרות הקסדצימליות הוא ייכתב כך: "9FE3". נכון עכשיו זה ברור? טוב, לא בדיוק, עדיין מדובר בשפת מכונה, אבל כעת יותר קצר

לכתוב את הפקודה ולזכור אותה. לכן כל כלי דיבוג שמציג לנו את הזיכרון של המחשב יציג זאת לא בבינארי אלא בהקסצימלי.

הבנו עד כה שכל הפקודות שהמעבד מריץ רשומות בשפת מכונה. נניח שיש לנו בדיסק הקשיח קובץ הרצה, שהסיומת שלו היא EXE. סיומת זו מעידה על כך שמדובר בקובץ עם פקודות בשפת מכונה, קובץ שמערכת ההפעלה Windows יודעת לגרום למעבד להריץ. מה קורה כאשר אנחנו המשתמשים מבקשים להריץ את הקובץ (לדוגמה באמצעות הקלקה כפולה עם העכבר)?

בשלב הראשון מבוצעת טעינה Loading של הקובץ מהדיסק הקשיח אל זכרון מהיר שנקרא RAM. בהמשך נלמד על ה-RAM ונבין בדיוק איך מתבצע תהליך הטעינה, כרגע חשוב להבין רק שהקוד מועתק אל זיכרון שהמעבד יכול לגשת אליו. מיד לאחר הטעינה, הקוד מתחיל לרוץ לפי שלושת השלבים הבאים:

א. Fetch - הבאת הפקודה מזיכרון ה-RAM אל המעבד

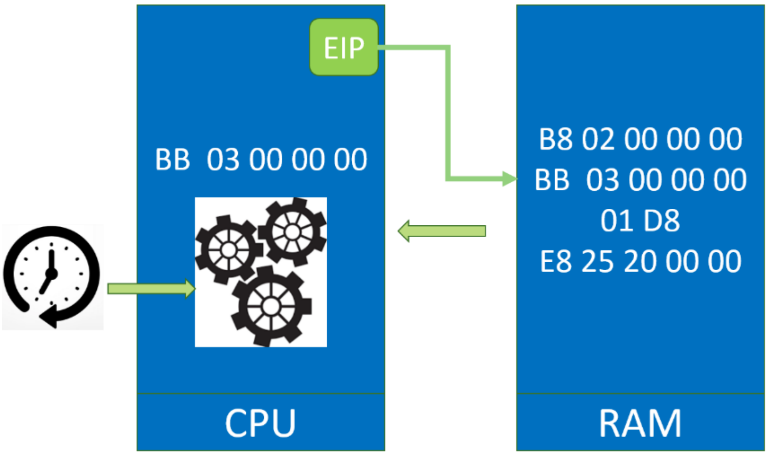
ב. Decode- פענוח הפקודה. לדוגמה, מה סוג הפקודה? אם הפקודה כוללת גישה לכתובת בזיכרון, מהי הכתובת המבוקשת?

ג. Execute- הרצה. ביצוע הפקודה, לדוגמה חישוב הסכום של שני מספרים.

לאחר הרצת פקודה בשפת מכונה, מתבצע מעבר לפקודה הבאה ושלושת השלבים חוזרים על עצמם, כך שוב ושוב. הדבר היחיד שיכול להפסיק את מעגל ה-Fetch Decode Execute הוא ניתוק של המעבד ממקור מתח חשמלי.

המעבד מכיל רגיסטר שנקרא IP, קיצור של Instruction Pointer, והוא זה שמצביע על הפקודה הבאה שיש להריץ. בכל פעם שהשעון של המעבד מתקדם, ה-IP עובר להצביע על הפקודה הבאה. עד אז המעבד חייב להשלים את ביצוע הפקודה הנוכחית.

השם של הרגיסטר משתנה עם דורות של מעבדים. השם IP נקבע במקור עבור מעבדי 16 ביט. מעבדי 32 ביט צריכים רגיסטר גדול יותר, שמסוגל להכיל כתובות של 32 ביט, ולכן הרגיסטר שודרג ל- Extended Instruction Pointer, או בקיצור EIP. כיוון שלרוב בספר זה נדבר על ארכיטקטורת 32 ביט, נשתמש ברגיסטר EIP. במעבד 64 אותו הרגיסטר נקרא RIP.



בכל "תקתוק" של השעון המעבד מבצע את שלבי ה-*Fetch-Decode-Execute* על הפקודה שהרגיסטר *EIP* מצביע עליה ו-*EIP* מתקדם להצביע על הפקודה הבאה

2.1.4 משפה עילית לשפת מכונה

נניח שאנחנו כותבים קוד בשפה עילית, כגון C, כיצד בסופו של דבר המעבד יוכל להריץ את הקוד שלנו בשפת מכונה?

כדי להמיר קוד משפת C לשפת מכונה מתבצעים שני שלבים. השלב הראשון מבוצע על ידי קומפיילר, תוכנה שממירה משפת C לשפת אסמבלי. השלב השני הוא המרת הקוד משפת אסמבלי לשפת מכונה, שרק אותה המעבד יכול להריץ. ההמרה מבוצעת על ידי תוכנה שנקראת אסמבלר.

לעיתים קרובות שני השלבים הללו יבוצעו בו זמנית על ידי סביבת הפיתוח שלנו. לדוגמה Visual Studio מבצעת את שני השלבים בזה אחר זה באופן אוטומטי, כך שלמשתמש נראה שהקוד בשפת C הופך מיידית לשפת מכונה. עם זאת, אם תריצו את הפרוייקט שלכם ב-Visual Studio במצב Debug, תוכלו לפתוח חלון של Disassembler ולראות את הפקודות בשפת אסמבלי בעודכם מדבגים את התוכנה.

המסקנה מהתיאור שלנו עד כה, היא שכל עוד אנחנו מתכנתים בשפה עילית כגון C איננו צריכים להיות מוטרדים לגבי סוג המעבד עליו תרוץ התוכנית שלנו. הקוד בשפה עילית נראה אותו דבר על סוגי מעבדים שונים. זה אחד היתרונות המרכזיים של שפה עילית- אנחנו לא מוטרדים מהחומרה הספציפית עליה הקוד ירוץ.

אולם, ברגע שנרצה להמיר את הקוד שלנו לשפת אסמבלי וומנה לשפת מכונה, נצטרך לבחור לאיזה סוג מעבד הקומפיילר והאסמבלר יבצעו עיבוד.

כדי להבין יותר לעומק איך מורכבות תוכנות מחשב, נחקור כיצד קוד שאנחנו כותבים שמור כשפת מכונה. כדי לעשות זאת נצטרך לרכוש ידע בכלי מתאים למשימה. לפני הכל, קיראו את הנספח "מדריך בסיסי לעבודה עם IDA". לאחר שתשלטו בבסיס של התוכנה, המשיכו לביצוע המשימות הבאות.

2.2 מחקר אזורי הזיכרון השונים של תוכנה

את החלק הזה נלמד באמצעות משימות מחקר קטנות. בכל משימה כזו ניקח קטע קוד קטן ונבדוק כיצד נשמר בזיכרון דבר מה שמעניין אותנו. מה מעניין אותנו? אנחנו רוצים לדעת היכן בזיכרון נשמר קוד של תוכנה שאנחנו מריצים. לתוכנה יש כל מיני חלקים:

- פקודות
- קבועים
- משתנים גלובליים
- משתנים מקומיים
- אזורי זיכרון שמוקצים תוך כדי ריצה (פקודת malloc)
- פונקציות שאנחנו מייבאים בזמן ריצה (מתוך קבצים מסוג DLL, נלמד עליהם בהמשך)

לכן בכל פעם ניקח תוכנה שכתבנו וכוללת את אחד הדברים האלה ונבדוק היכן מוקצה הזיכרון.
דגש חשוב להמשך: אנחנו חוקרים כעת את הקוד כפי שהוא שמור בקובץ ההרצה שיצרנו, קובץ מסוג EXE. קובץ ה-EXE שמור על הדיסק הקשיח ולא טעון אל ה-RAM. בתהליך הטעינה ל-RAM יש דברים רבים

שמשתנים, ונעמוד עליהם בהמשך. כרגע מה שחשוב לזכור הוא שאנחנו צופים בתמונת מצב של אזורי הזיכרון כפי שהם מוגדרים על הדיסק הקשיח, לא בתמונת הזיכרון בתוך ה-RAM.

2.2.1 שלב א'- פקודות

השתמשו בתוכנית hello.exe שיצרתם עבור נספח לימוד ה-IDA. היכן נשמרות הפקודות עצמן? כפי שאתם רואים, באזור זיכרון שנקרא "text". מה אפשר לדעת על אזור זיכרון זה?

כשאתם בתצוגת קוד (לא תצוגה גרפית) גללו למעלה עד שתגיעו לתחילת אזור ה-text. יופיע לפניכם טקסט דומה לזה:

```
; Section 2. (virtual address 00002000)
; Virtual size                  : 00000019 (     25.)
; Section size in file          : 00000200 (    512.)
; Offset to raw data for section: 00000600
; Flags 60000020: Text Executable Readable
; Alignment     : default

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
assume cs:_text
;org 402000h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
```

על מה אנחנו מסתכלים? IDA לוקחת את ההגדרות של אזורי הזיכרון מתוך ההגדרות השמורות בקובץ ה-exe. התוכנה מציגה לנו בצורה נוחה לקריאה את משמעות המידע שנמצא ב-exe. השורה שמעניינת אותנו היא היכן שכתוב "Flags 60000020: Text Executable Readable". הפירוש של השורה ניתן מיד אחר כך בהערות האפורות- זהו אזור זיכרון שהוא "Pure Code" ויש לו הרשאות של קריאה והרצה.

2.2.2 שלב ב' - קבועים

חיקרו בעצמכם היכן שמורה המחרוזת "Hello!" שהגדרתם.

רק לאחר מכן עיברו על ההסבר הבא ובידקו שהגעתם לאותה תוצאה.

בתוך ה-main שלנו, נתמקד בשתי שורות. הראשונה, push offset aHello. שורה זו דוחפת לתוך המחסנית את הכתובת בזכרון של מחרוזת Hello. אם נקליק על aHello נופנה לכתובת של המחרוזת בזכרון:

```
.rdata:00416B92                    align 4
.rdata:00416B94 aHello             db 'Hello!',0Ah,0        ; DATA XREF: sub_412320+1E↑o
.rdata:00416B9C aStackMemoryCor db 'Stack memory corruption',0
```

ניתן לראות שהמחרוזת מוגדרת בכתובת 416B94, ולאחריה מוגדר התו 0Ah (כזכור מלימודי האסמבלי, ירידת שורה) ולאחריה התו 0 (כזכור, מציין סיום המחרוזת לטובת פונקציות הדפסה).

אך מהו rdata? אם נגלול למעלה את הזיכרון, נגיע אל תחילת החלק בזיכרון בו מוגדר rdata:

```
.rdata:00416000 ; Section 3. (virtual address 00016000)
.rdata:00416000 ; Virtual size                  : 00001FE9 (   8169.)
.rdata:00416000 ; Section size in file          : 00002000 (   8192.)
.rdata:00416000 ; Offset to raw data for section: 00005200
.rdata:00416000 ; Flags 40000040: Data Readable
.rdata:00416000 ; Alignment     : default
.rdata:00416000 ; =========================================================
.rdata:00416000
.rdata:00416000 ; Segment type: Pure data
.rdata:00416000 ; Segment permissions: Read
.rdata:00416000 _rdata          segment para public 'DATA' use32
.rdata:00416000                 assume cs:_rdata
```

רואים ש- rdata הוא אזור בזיכרון בעל האפיון "Data Readable". ההרשאות של אזור זה בזכרון הן "Read". אי אפשר לכתוב לאזור זה בזיכרון. המשמעות היא שהמחרוזת Hello שהגדרנו בקוד היא קבועה, נשמרת במקום שמיועד לקריאה בלבד. ואכן בהמשך נראה שאילו היינו רוצים שהמחרוזת שלנו תוכל להשתנות היה עלינו להגדיר אותה בצורה שונה.

2.2.3 שלב ג' – משתנים גלובליים

הגדירו אוסף של משתנים גלובליים מסוגים שונים וצפו במקום שבו הם נשמרים בזיכרון.

```
// global variables
char    my_char = 'b'
short   my_short = 4000;
int     my_int = 10;
long    my_long = 65535;
double  my_double = 4.32;
```

שימו לב, שכדי שהמשתנים הללו יופיעו בקוד ה-exe, קימפול התוכנית חייב להיות עבור גרסת Debug. כאשר מקמפלים עבור גרסת Release, הקומפיילר נוטה להעיף חלקי קוד שאין בהם שימוש.
אם ביצעתם את הדברים נכון, מצאתם את המשתנים הללו שמורים באזור זיכרון שנקרא data. מה ההבדל בינו לבין אזור הזכרון rdata, ששימש לשמירת הקבוע "Hello!" ?

```
.data:00403000 ; Flags C0000040: Data Readable Writable
.data:00403000 ; Alignment     : default
.data:00403000 ; =====================================================
.data:00403000
.data:00403000 ; Segment type: Pure data
.data:00403000 ; Segment permissions: Read/Write
```

כפי שרואים, data הוא בעל הרשאות כתיבה וקריאה, לא רק קריאה. זה הגיוני- אחרי הכל הגדרנו משתנים, לא קבועים.

2.2.4 שלב ד' – משתנים מקומיים

כעת נגדיר פונקציה ובתוכה נשים מספר משתנים מקומיים. חשוב להגדיר פונקציה שמקבלת פרמטרים, כדי שנוכל לראות גם היכן נשמרים הפרמטרים שמועברים לפונקציה. לשלב הזה, וכן לשלב הבא במחקר, מומלץ להשתמש בקוד הבא:

```
#include "pch.h"

int mult(int);
int max_num = 10;
int *buf = (int*)malloc(max_num * sizeof(int));

int main()
{
    for (int i = 0; i < max_num; i++) {
        buf[i] = mult(i);;
        printf("%d\n", buf[i]);
    }
    return 0;
}

int mult(int num)
{
    static int my_num = 5;
    int result = num*2 + my_num;
    return result;
}
```

לפני שתמשיכו לקרוא, חפשו בעצמכם וענו על השאלות הבאות:

- באיזה אזור זיכרון שמורים המשתנים המקומיים שהגדרנו (i, result)?
- באיזה אזור זיכרון נמצא הפרמטר num שמועבר לפונקציה mult?
- כיצד IDA קורא לפרמטר num? וכיצד IDA מציין שמות של משתנים מקומיים?
- מהו המיקום של הכתובות בזיכרון של המשתנים המקומיים לעומת הפרמטרים? האם זה תמיד חייב להיות כך?

תשובה:
בוודאי מצאתם את כל המשתנים הללו שמורים באזור זיכרון שנקרא Stack, המחסנית.
המשתנים שנשלחים לפונקציה נקראים args, והמשתנים בתוך הפונקציה נקראים vars. שימו לב לכתובות בזיכרון של args לעומת ה-vars. כזכור לנו מלימודי האסמבלי, ה-args נדחפים למחסנית בכתובות גבוהות בזיכרון, לפני הכניסה לפונקציה. כאשר הפונקציה מגדירה משתנים מקומיים הם תמיד יהיו בכתובות נמוכות יותר מאשר הפרמטרים שהפונקציה מקבלת.

2.2.5 שלב ה' – סוגי משתנים נוספים

ישנו בקוד הדוגמה משתנה מקומי שהוא static. כזכור, משתנים אלו מיוחדים בכך שערכם נשמר בין קריאות לפונקציה. היכן הייתם מצפים שמשתנה static יישמר?

לבסוף, נסו להבין באיזו כתובת נמצא המערך בגודל עשרה int‏ים שמוגדר על ידי פקודת ה-malloc. דעו מראש, שתצליחו למצוא רק את המיקום בזיכרון של המצביע buf, לא של המערך עצמו. בהמשך נבין מדוע.

תשובה:
משתנה מסוג static אינו יכול להשמר על ה-Stack כמו משתנים מקומיים אחרים, מכיוון שהמידע על ה-Stack לא נשמר בין קריאות לפונקציה. לכן הדרך היחידה היא לשמור אותו במקום שבו נשמרים משתנים גלובליים, ה-data.

פקודת ה-malloc מקצה זיכרון מתוך ה-Heap. זהו אזור זיכרון שמיועד להקצאות זיכרון בזמן ריצת התוכנית. כדי לדבר על מיקום ה-Heap בזיכרון נפריד בין המצביע על אזור הזיכרון (המשתנה buf) לבין אזור הזיכרון שהוקצה (כלומר הכתובת בזיכרון שהמצביע מצביע עליה).

המצביע לאזור הזיכרון שהגדרנו נמצא באזור ה-data, כיוון שהגדרנו אותו גלובלי, אך הוא גם יכל להיות על המחסנית אם היינו מבצעים את ה-malloc מתוך קוד של פונקציה כלשהי. אך <u>הערך</u> ש-buf מצביע עליו הוא כתובת שנמצאת בתוך אזור זיכרון אחר. זיכרון זה מוקצה בזמן ריצה, לכן לא נוכל לראות אותו ולבדוק את הכתובות שלו באמצעות IDA. זאת כיוון ש-IDA הוא כלי לניתוח סטטי של קוד (ניתוח קוד לא בזמן ריצה).
ישנם כלים אחרים, שמאפשרים ניתוח דינמי (בזמן ריצה), כגון x96dbg או WinDbg, אך הם מחוץ להיקף של פרק זה. על WinDbg נפרט באופן בסיסי בהמשך. מה שכן נוכל לעשות, הוא להדפיס את הכתובות של המשתנים בזמן הריצה. בצעו עריכה לקוד של main:

```
int main()
{
    int i;
    for (i = 0; i < max_num; i++) {
        buf[i] = mult(i);;
        printf("%d\n", buf[i]);
    }
    printf("Address of local variable i: %x\n", &i);
```

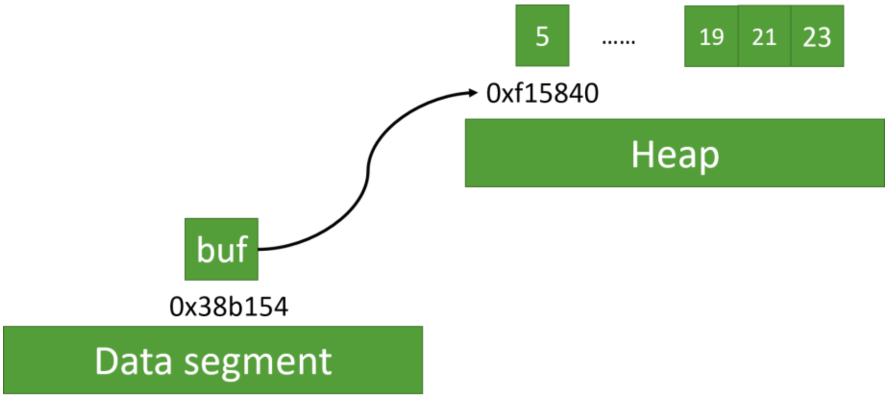
```
    printf("Address of global variable max_num: %x\n", &max_num);
    printf("Address of pointer to array: %x\n", &buf);
    printf("Address of array on heap: %x\n", buf);
    printf("value of 1st element in array: %x\n", *buf);
    return 0;
}
```

בשלושת ההדפסות הראשונות אנחנו מדפיסים את הכתובות שניתנו למשתנים i, max_num, buf. הכתובת של המשתנה buf, שרשומה בתור &buf, היא כתובת שמאחסנת מצביע לזיכרון, ולא את המערך עצמו. הכתובת של המערך נמצאת במצביע לזיכרון, כלומר בתוך buf. לטובת שלמות הדוגמה, אפשר לראות גם כיצד ניתן להוציא את הערך הראשון מתוך המערך, וזאת באמצעות *buf. פעולת ה- *, שנקראת dereferencing, ניגשת אל המקום בזיכרון שהמשתנה מצביע עליו.

אם תריצו את התוכנית, תוכלו לראות את הכתובות שהוקצו בזיכרון. כמובן שהכתובות עשויות להשתנות בין הרצה להרצה, כיוון שה-Stack וה-Heap מוקצים ב-RAM בכתובות חדשות כל פעם שהתוכנית נטענת ל-RAM. הרצה לדוגמה:

```
Address of local variable i: cffa44
Address of global variable max_num: 38b000
Address of pointer to array: 38b154
Address of array on heap: f15840
value of 1st element in array: 5
```

נמחיש כיצד המשתנה buf מצביע על הזיכרון שמוקצה ב-Heap:



מה ניתן להסיק מהדמיון והשוני בין הכתובות שמודפסות בזמן ריצה?

המשתנים max_num ו-buf הם משתנים גלובליים. אפשר לראות שאכן הכתובות שלהן בזיכרון דומות למדי-שתיהן מהצורה 0x38Bxxx (שימו לב לצורת הרישום: הכתובת מתחילה ב-0x כדי לציין שמדובר בספרות הקסדצימליות). המשתנה i נמצא על המחסנית ולכן המיקום שלו בזיכרון 0xCFFA44, ניכר שנמצא באזור זיכרון אחר. המשתנה buf מצביע על הזיכרון שהקצינו עם malloc ואשר נמצא בכתובת 0xF15840, השוני

בינה לבין הכתובות שנמצאות על ה-data segment ועל ה-Stack הוא בולט ומעיד על כך שמדובר באזור זיכרון אחר, שהוא כמובן ה-Heap.

2.2.6 משתנים לא מאותחלים

אזור ה-BSS משמש להקצאה של זיכרון עבור משתנים לא מאותחלים. הקוד הבא מגדיר שני מערכים- המערך הראשון אינו מאותחל והמערך השני מאותחל:

```
#include <stdio.h>
#pragma bss_seg("bss")

char uninit_str[10000000];
char init_str[2] = {'A',0};

int main()
{
    uninit_str[0] = 'H';
    uninit_str[1] = 'i';
    uninit_str[2] = 0;
    printf("%s\n%s\n", uninit_str, init_str);
    return 0;
}
```

בתחילת התוכנית אנו מקצים משתנה גלובלי בגודל עצום, 10 מליון בתים, אך לא מכניסים לתוכו ערכים. אין סיבה שקובץ ה-exe שיווצר יכלול 10 מליון בתים שאינם מכילים מידע. הרבה יותר חסכוני לשמור באזור זיכרון מיוחד את המידע על כך שישנו משתנה בגודל 10 מליון בתים, ולהקצות את הזיכרון עצמו רק בזמן הריצה, כאשר צריכים אותו. ראשי התיבות של BSS מייצגים Block Started by Symbol, אולם בגלל החסכוניות של ה-BSS לעיתים מפרשים את ראשי התיבות כ- Better Save Space ☺ אזור הזיכרון הזה קיים רק בקובץ ב-exe, לעומת זאת בזמן טעינת התוכנית שלנו לזיכרון המשתנים יוגדרו ב-data.

הפקודה pragma bss_seg מנחה את הקומפיילר לאסוף את כל המשתנים הלא מאותחלים לתוך אזור זיכרון מיוחד. אם נחקור את קובץ ההרצה באמצעות IDA נראה את ההבדלים בין הכתובות בזיכרון של uninit_str לעומת init_str:

```
push    offset unk_F8C000
push    offset byte_417000
push    offset aSS      ; "%s\n%s\n"
```

המשתנה uninit_str נמצא בתוך ה-bss, בכתובת 0x417000. המשתנה init_str נמצא באזור ה-data וכתובתו 0xF8C000. בשורה התחתונה, מה שראינו הוא שלמרות שגם המערך המאותחל וגם המערך שאינו מאותחל הם משתנים גלובליים, הם אינם מוגדרים באותו אזור זיכרון. איך דבר זה מסייע למטרה שלנו, חיסכון בגודלו של קובץ ה-exe? אזור הזיכרון של ה-BSS לא באמת נזקק למקום בקובץ ה-EXE, מכיוון שאין בו מידע. הדבר היחיד שנשמר הוא גודל האזור הזה. כך, קובץ ה-exe נשאר קומפקטי. רק עם הרצת התוכנה מתבצעת הקצאה של מקום בזיכרון עבור המשתנים הלא מאותחלים, בתוך אזור ה-data.

2.2.7 אזורי זיכרון נוספים שלא חקרנו עד כה

קובץ exe עשוי להכיל הגדרות של אזורי זיכרון נוספים. חלק מאזורים מוגדרים על ידי מיקרוסופט לצורך דיבוג או שימושים אחרים. אזור ה-rsrc כולל משאבים שהתוכנה שלנו משתמשת בהם (כפי שאתם זוכרים, בפרק הקודם ראינו איך ליצור משאב אייקון לתוכנה).

מהו idata? על סמך הידע הנוכחי שלנו קשה להסביר את הדברים לעומק. אך היו בטוחים שהדברים יתבררו בהמשך. מבלי להתעכב על יותר מדי פרטים, כאשר אנחנו מתכנתים, כמעט תמיד נרצה להשתמש בפונקציות מתוך ספריות קוד שמיובאות לקוד שלנו בזמן ריצה. לספריות קוד אלו קוראים DLL. לדוגמה רבות מהפונקציות של מערכת ההפעלה מיובאות לקוד שלנו בזמן ריצה באמצעות DLL שנקרא Kernel32.DLL. בהמשך ישנו פרק מיוחד שמדבר על DLLים. אזור ה-idata משמש לצורך שמירת הכתובות של הפונקציות שאנחנו מייבאים, אך אזור זה אינו קיים בקובץ ה-exe מכיוון שהכתובות של הפונקציות החיצוניות עדיין לא ידועות והן נקבעות רק בזמן הטעינה של התוכנית ל-RAM. בהמשך ישנו פרק מיוחד שמוקדש לטעינה של תוכניות ל-RAM. נבין טוב יותר בעתיד.

מהו reloc? גם נושא זה מעט מורכב להסבר כרגע. בתהליך הטעינה של תוכנה ל-RAM נדרש לעיתים לשנות כתובות של פונקציות ומשתנים בזיכרון (דמיינו מצב שבו תוכנית נטענת ל-RAM לכתובות שונות ממה שהיא ציפתה. אם ההמחשה לא ברורה- נבין בדיוק בהמשך).

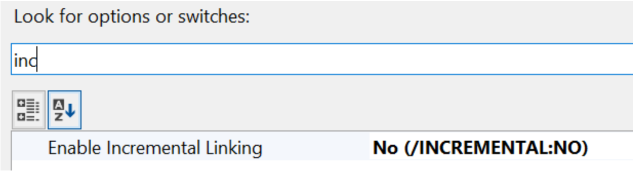
2.2.8 סיכום אזורי הזיכרון

חיזרו אל תוכנת ה-IDA והקישו על Shift+F7. פעולה זו תעביר אתכם אל מסך שבו מרוכזים כל אזורי הזיכרון שבקוד. הגודל של אזורי הזיכרון צפוי להשתנות בין תוכנית לתוכנית, אולם המבנה של הטבלה נשאר דומה וצפוי להראות כך:

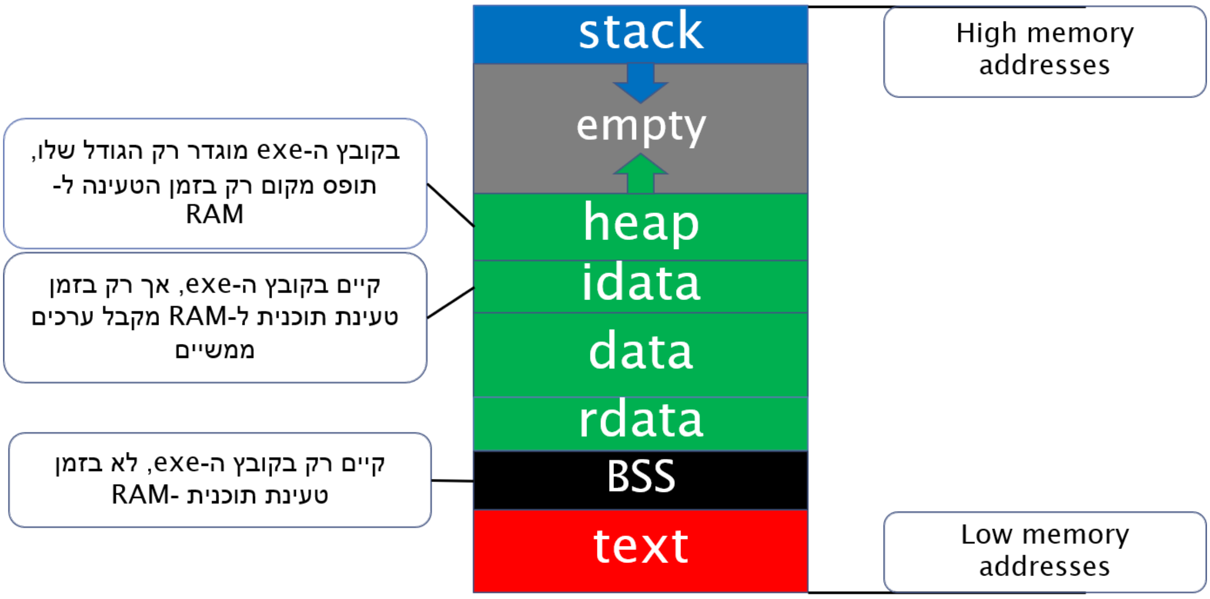
Name	Start	End	R	W	X
.textbss	0000000000401000	0000000000411000	R	W	X
.text	0000000000411000	0000000000417000	R	.	X
bss	0000000000417000	0000000000F89000	R	W	.
.rdata	0000000000F89000	0000000000F8C000	R	.	.
.data	0000000000F8C000	0000000000F8D000	R	W	.
.idata	0000000000F8D000	0000000000F8D1BC	R	.	.

אפשר לראות את שמות אזורי הזיכרון, כתובת ההתחלה והסיום שלהם ומהן ההרשאות שלהם. כפי שראינו, ישנן 3 הרשאות – קריאה, כתיבה והרצה. אזור זיכרון יכול לבחור כל צירוף מבין שלושת ההרשאות הללו. כל אזור זיכרון מתחיל בכתובת שהיא מיד לאחר הכתובת האחרונה של אזור הזיכרון שלפניו.

אזור הזיכרון היחיד שלא דנו בו הוא textbss. זהו אזור זיכרון שמשמש את Visual Studio בתהליך ה-Linking של התוכנה (חלק מתהליך הפיכת התוכנה לקובץ בשפת מכונה, נלמד עליו בהמשך). אפשר לבטל אותו אם מבטלים את אופציית ה-Incremental Linking בהגדרות הפרוייקט-



נסדר את מבנה אזורי הזיכרון בצורה גרפית:



זהו המבנה הקלאסי מספרי הלימוד של אזורי זיכרון של תוכנה במערכת ההפעלה Windows, אך מצאנו אותו לבד באמצעות הכלים שלמדנו ☺. המבנה הנ"ל מציג את מה שחקרנו לבד, מה שחסר בו הוא אזור זיכרון נוסף שהחקירה שלנו "פספסה", האזור בו נמצא קוד מערכת ההפעלה. אל דאגה נגיע אליו בפרק הבא.

חשוב להבין, ונדון בכך בפירוט בהמשך, שבזמן הטעינה של קובץ exe ל-RAM משתנים דברים רבים. יש אזור זיכרון ש"נעלם", ה-BSS. יש אזור זיכרון ש"נוצר", ה-Heap. ויש אזור זיכרון שרק בזמן הטעינה ל-RAM מקבל ערכים בעלי משמעות, ה-idata. השינויים בין המצב שקיים על קובץ ה-exe לבין המצב ב-RAM מפורטים בהערות לצד אזורי הזיכרון שמצאנו.

לסיכום, אנחנו רואים שכל תוכנית בנויה מאזורי זיכרון שונים, אשר לכל אזור יש הרשאות שונות (כתיבה, קריאה, הרצה). אזורי הזיכרון צמודים זה לזה וניתן לבנות "מפה" של הזיכרון. אולם שאלות רבות נותרו עדיין

ללא מענה- איך נראה הזיכרון כאשר יותר מתוכנה אחת רצה? האם שתי תוכנות יכולות לגשת לאותו זיכרון? והיכן הזיכרון ששייך למערכת ההפעלה? על שאלות אלו ורבות אחרות נענה בהמשך הלימוד.

מצויידים בידע הזה אנחנו יכולים להתחיל להעמיק את הידע שלנו אל תוך מערכת ההפעלה.

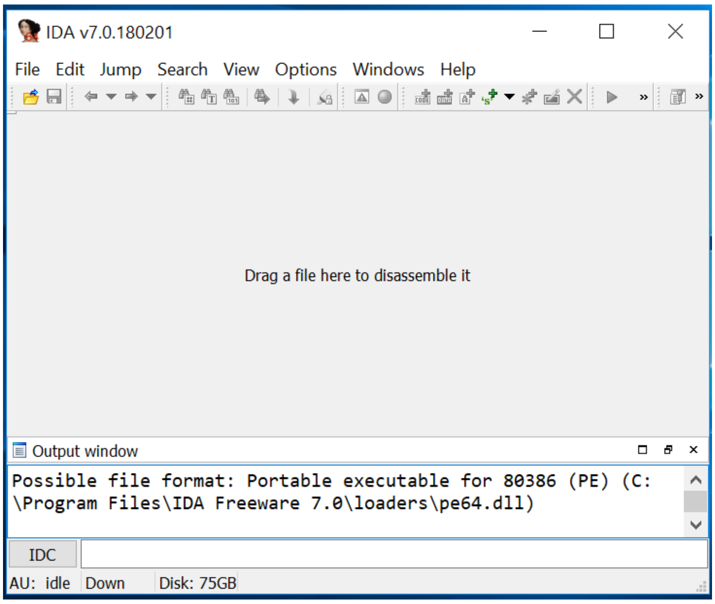
2.3 נספח – מדריך בסיסי לעבודה עם IDA

השם IDA הוא שם של כלי מסחרי, קיצור של Interactive DisAssembly. דיסאסמבלי הוא כינוי כללי לתוכנה שיודעת לקחת קוד בשפת מכונה ולהציג אותו בשפת אסמבלי. תוכנת IDA היא מהכלים הנפוצים ביותר בתעשיה ולמרות שהיא עולה כסף רב, לעיתים עשרות אלפי דולרים לרשיון, אפשר להוריד חינם גרסה שלה שמאפשרת לנו את כל מה שאנחנו צריכים בשלב זה.

בחרו את הגרסה העדכנית ביותר של IDA אשר כתוב לידה freeware (המנעו מלהתקין demo בעל תכונות מלאות, אך עם רשיון לזמן מוגבל).

https://www.hex-rays.com/products/ida/support/download.shtml

לאחר שסיימנו את ההתקנה נעלה את IDA:



2.3.1 ה-Disassembly הראשון שלנו

נוריד את הקובץ add.exe מהלינק הבא: https://data.cyber.org.il/os/add.exe

נגרור את קובץ ה-exe אל החלון (או נשתמש בטאב file->open).

במסך הבא נאמר ל-IDA לטעון את הקובץ שלנו כקובץ PE, קיצור של Portable Executable, הפורמט הנפוץ לקבצי הרצה.

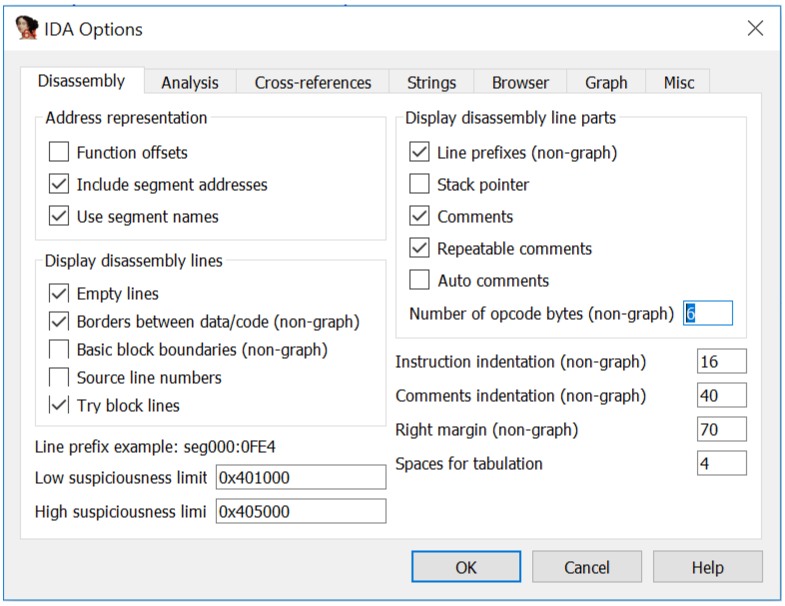
```
public start
start proc near
mov     eax, 2
mov     ebx, 3
add     eax, ebx
call    sub_404036
push    0               ; uExitCode
call    ds:ExitProcess
start endp
```

והנה, IDA הצליחה לקחת את הקוד שלנו בשפת מכונה ולתרגם אותו בחזרה לקוד בשפת אסמבלי. אפשר לזהות את אותן הפקודות שכתבנו בקוד המקור:

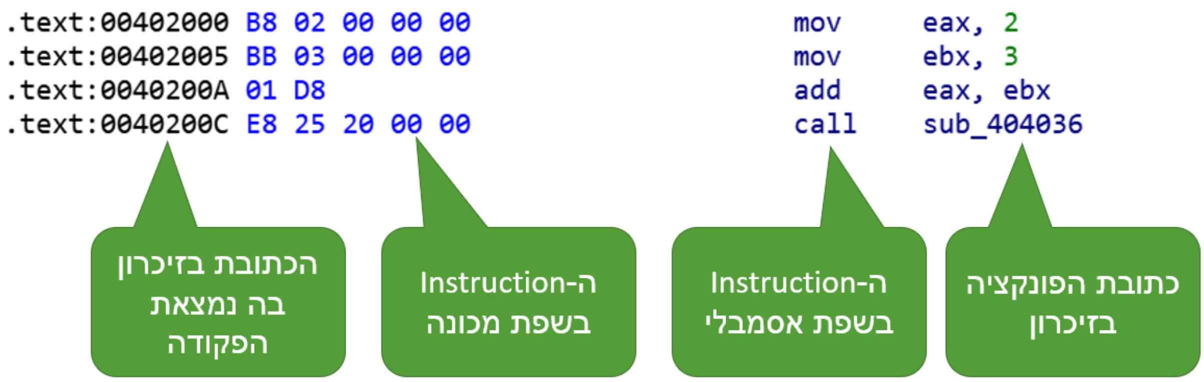
כזכור, המטרה שלנו היתה לצפות בקוד בשפת המכונה. לכן נלחץ על מקש הרווח. מקש הרווח מעביר את התצוגה בין מצב גרפי למצב קוד:

```
.text:00402000                  public start
.text:00402000 start            proc near
.text:00402000                  mov     eax, 2
.text:00402005                  mov     ebx, 3
.text:0040200A                  add     eax, ebx
.text:0040200C                  call    sub_404036
.text:00402011                  push    0               ; uExitCode
.text:00402013                  call    ds:ExitProcess
.text:00402013 start            endp
```

מיד נבין מה אנחנו רואים כאן. ישנו רק דבר אחד נוסף שנרצה לעשות. הכנסו לתפריט Options ובחרו בתוכו את האפשרות General. יופיע לפניכם המסך הבא, בתוכו שנו את הערך של Number of opcode bytes שבמקום שיהיה 0 יהיה 6, כמו בדוגמה הבאה:



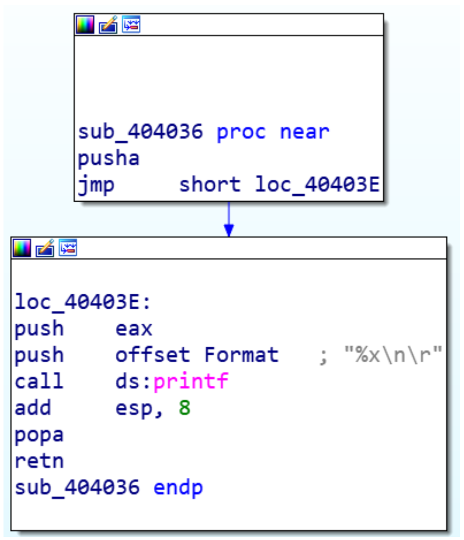
כעת התצוגה היא כפי שרצינו. מה אנחנו רואים?



בצד שמאל, היכן שכתוב text ולידו מספר, זוהי הכתובת בזיכרון בה נמצאת הפקודה. לאחר מכן, אפשר לראות את הפקודות בשפת מכונה ממש (הידד! ☺). שימו לב לדבר הבא: הפקודה הראשונה, המתחילה ברצף B802, גודלה בדיוק 5 בתים. זאת מכיוון שכל ספרה הקסדצימלית מייצגת 4 ביטים, ולכן שתי ספרות הקס שוות לבית אחד. מהו ההפרש בין הכתובת של הפקודה השניה לפקודה הראשונה?

402005 פחות 402000 שווה אכן 5 בתים. כך אנחנו רואים שכל פקודה נמצאת בזיכרון בדיוק לאחר הפקודה הקודמת.

שתי העמודות הבאות נראות בדיוק כמו הקוד בשפת אסמבלי. בקצה השורה האחרונה יש משהו חדש- פקודה שנקראת "call" ולאחריה המילים "sub" ואז רצף ספרות. פקודה זו היא, כזכור מלימודי האסמבלי, קריאה לפרוצדורה. רצף המספרים הוא בדיוק הכתובת שבה הפרוצדורה שמורה בזיכרון. כדי לראות זאת, לחצו שוב על מקש הרווח כדי לעבור למוד גרפי, הקליקו על התווית "sub_404036" (ייתכן שאצלכם שם התווית יהיה שונה מכיוון שהפרוצדורה נמצאת במקום שונה בזיכרון). כעת קוד הפרוצדורה נמצא לפניכם:

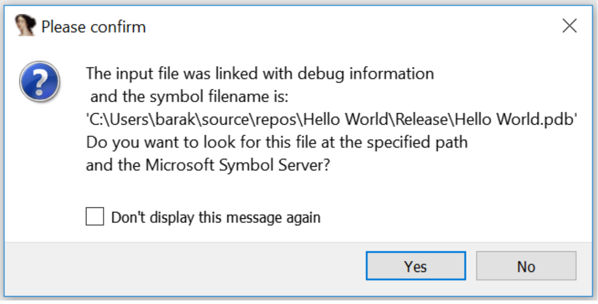


אפשר לראות שבתוך הקוד יש פקודת קפיצה, המיוצגת גרפית על ידי חץ, וכמו כן יש קריאה ל- printf. ואכן, מטרת הקוד הזה היא להדפיס את הערך ששמור בתוך הרגיסטר eax.

2.3.2 מבצעים Disassembly לקוד של עצמנו

בשלב הראשון ניצור קובץ הרצה מסוג EXE. קמפלו את הקוד הבא ב-visual studio, במצב **debug** (חשוב – לא release, הדבר ישנה את המשך התהליך):

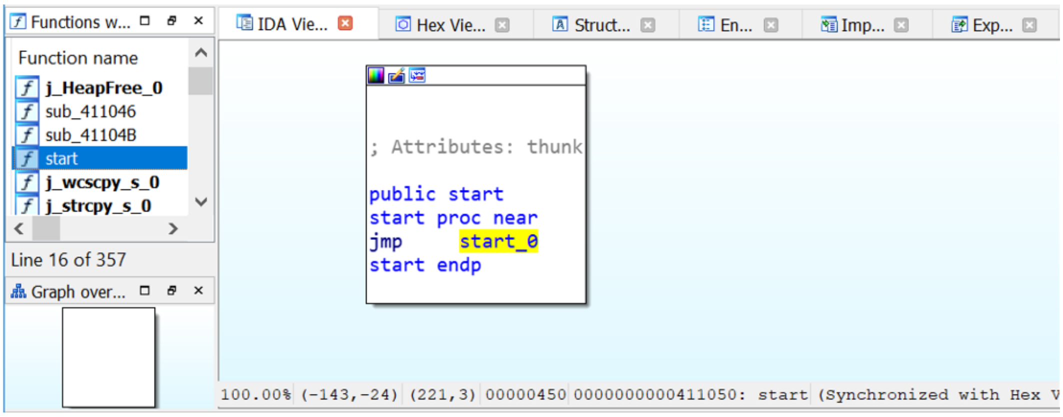
```
#include <stdio.h>

int main()
{
    printf("Hello!");
    return 0;
}
```



לאחר שנטען את קובץ ה-exe לתוך IDA נקבל הודעה דומה להודעה הבאה:

כעת נבחר באפשרות "**No**". הבחירה הזו משמעותית- IDA מזהה שיש קובץ מסוג pdb עם מידע מהקומפיילר (שכולל לדוגמה שמות של פונקציות). כאשר נקבל קוד שקומפל על ידי מישהו אחר, לא סביר שיהיה לנו קובץ pdb שייקח עלינו, לכן נבחר No.

הפתעה! אם ציפינו לכך שריצת התוכנית תחל ב-main, הדבר רחוק מכך. לפני שרץ main מתרחשים דברים רבים. אנחנו נמצאים כרגע במקום שנקרא start. ה-start היא נקודה חשובה מאד, משום שמשם מתחילה ריצת התוכנית שלנו. אם נרצה להגיע אליו שוב- מצד שמאל יש את שמות כל הפונקציות בקוד שלנו, לחיצה

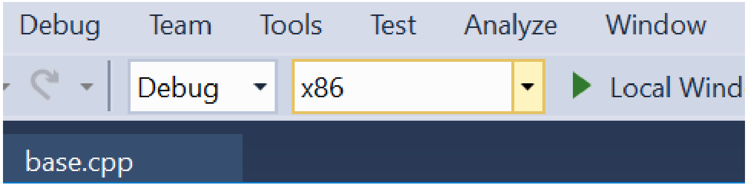


כפולה על start תחזיר אותנו לשם. מה גורם לכך שהתכנית מתחילה דווקא מ-start, ואיך אפשר לשנות את זה? התשובה לכך קשורה לדרך שבה קובץ הרצה מאורגן, נלמד על כך בפרק "פורמט PE", אך בינתיים נסתפק בתשובה זו.

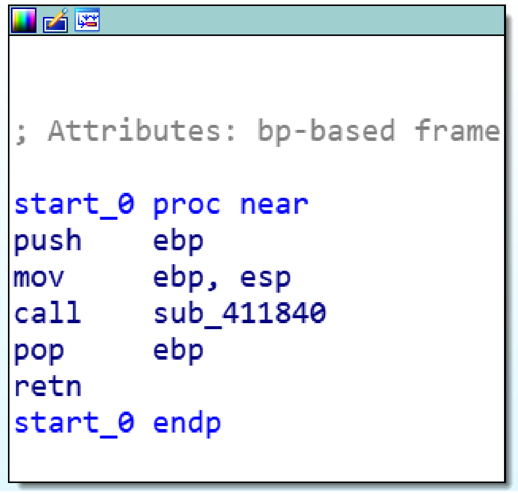
2.3.3 מציאת ה-main

משימתנו הראשונה היא למצוא את ה-main שלנו. לפנינו פקודת אסמבלי, שהיא הפקודה הראשונה שהמעבד מריץ כאשר הוא מריץ את התכנית שלנו, הפקודה jmp start_0. פקודה זו גורמת לקפיצה של הקוד אל עבר המקום בזיכרון שמסומן על ידי התווית start_0. ליתר דיוק, הקפיצה היא שינוי בערכו של הרגיסטר EIP, קיצור של Extended Instruction Pointer. רגיסטר זה הוא ההרחבה ל-32 ביט של הרגיסטר IP המוכר לנו מעולם ה-16 ביט (למי שלמד מספר האסמבלי של המרכז לחינוך סייבר).

נזכיר, שאנו נמצאים כרגע בעולם ה-32 ביט משום שהגדרות הפרוייקט שלנו ב-Visual Studio היו לקמפל לקוד שתומך ב-32 ביט, שנקרא x86. באותו מסך הייננו יכולים לבחור באפשרות x64 ולקבל תמיכה ב-64 ביט:

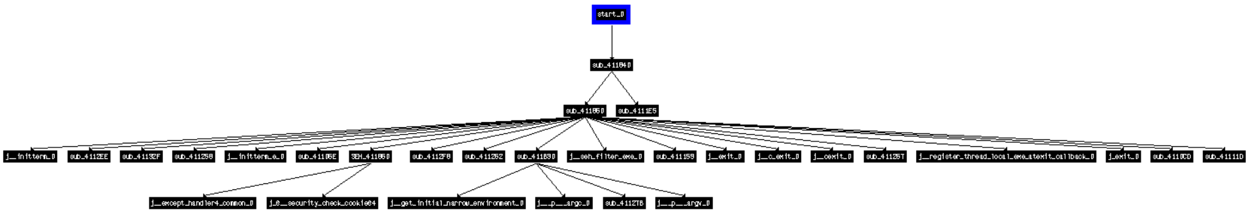


נחזור לפקודת ה-jmp שלנו: לידה רשומה כתובת בזיכרון. אם נקליק עליה נגיע לכתובת זו בזיכרון. ננסה זאת:



הגענו לכתובת start_0, יש אחריה מספר פקודות. איך כל זה עוזר לנו להגיע ל-main?

אם נעשה קליק ימני ונבחר "Xrefs graph from" נקבל את גרף הקריאות לפונקציות בתוכנית-



במגבלות העמוד אי אפשר להראות את שמות הפונקציות, אבל זו לא באמת מגבלה. אפשר לראות שהכל מתחיל למעלה, ב-start שמוקף בכחול. Start קורא לפונקציה, שקוראת לשתי פונקציות והן כבר קוראות לכמות די נכבדה של פונקציות. היכן שהוא בגרף הזה יש את פונקציית ה-main שלנו, אבל היא אינה קרויה main. אם כך, חישבו- איך נמצא אותה?

דרך אחת אפשרית היא היא לעבור על כל הפונקציות בצד שמאל של IDA, היכן שראינו שנמצא start, ולהקליק עליהן אחת אחת עד שנמצא קוד שנראה כמו main. זה אפשרי אבל ארוך ומתיש.

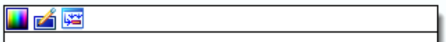
אפשרות אחרת היא לחשוב היכן נכון לחפש את main? באיזה מקום בקוד צפויה להיות קריאה ל-main?

אפשר לצפות שהקריאה תהיה לאחר קריאות לפונקציות של אתחולים ובדיקות, ושאחרי ה-main תיקרא רק פונקציה של יציאה מהתוכנית, משהו שיכלול את השם exit בשם הפונקציה. לכן מה שנעשה זה נתקדם עם הקריאות לפונקציות, בכל פעם שיש לנו הסתעפות בקריאות (פונקציה שקוראת ליותר מפונקציה אחת) נבחר להתקדם עם הפונקציה האחרונה שנקראת, כל עוד היא אינה כוללת בתוכה את השם "exit".

אם כך, נתחיל. הפקודה הראשונה שיש לנו שקוראת לפונקציה היא call sub_411840. כזכור call היא קריאה לפונקציה ואחריה מגיעה תווית שהיא הכתובת של הפונקציה בזכרון. במקרה זה IDA אומר לנו שהכתובת של הפונקציה בזכרון היא 0x411840.

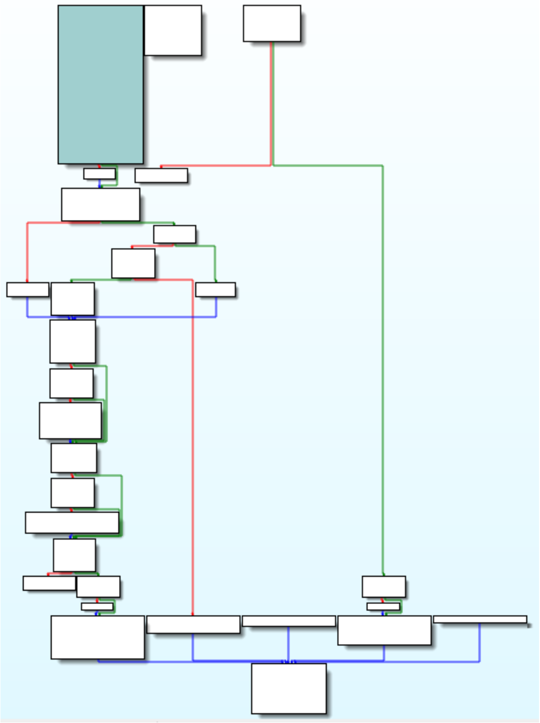
הערה חשובה: הכתובת לא חייבת להיות 0x411840 כמו בדוגמה. זאת מכיוון שבכל קומפילציה הקומפיילר יכול לבחור כתובת שונה. כמו כן יכול להיות שאצלכם מותקנת גרסה שונה של הקומפיילר. לכן צפוי שיהיו הבדלים מסויימים בקוד זהה שקומפל על ידי שני מחשבים שונים.

נקליק על שם הפונקציה ונעבור אליה:



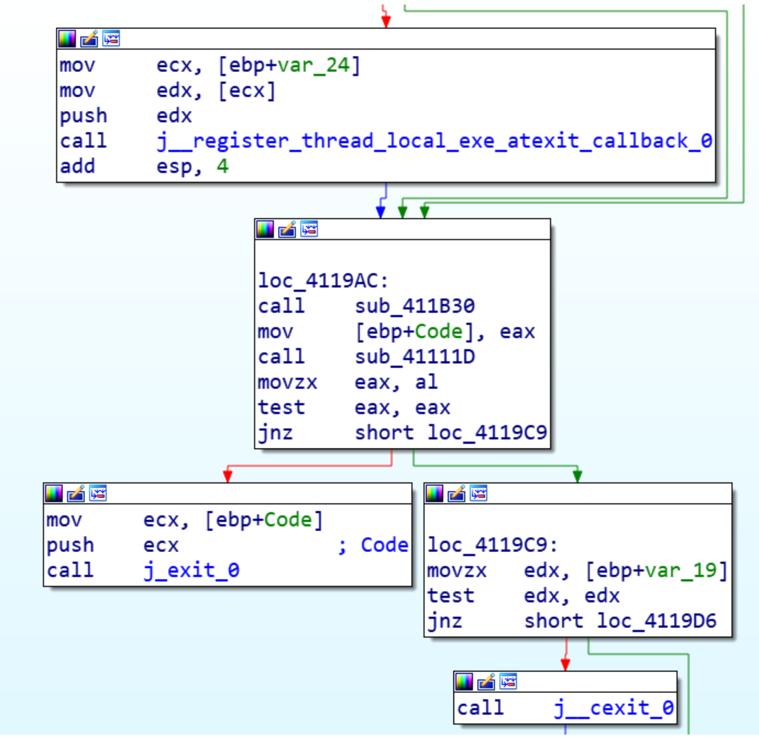
```
; Attributes: bp-based frame

sub_411840 proc near
push    ebp
mov     ebp, esp
call    sub_4111E5
call    sub_411860
pop     ebp
retn
sub_411840 endp
```

לפנינו הפונקציה שנמצאת בזכרון בכתובת 0x411840, כפי שניתן לראות בשורה הראשונה. בתוכה יש שתי קריאות לפונקציות. בהתאם לשיטה שלנו, נבחר בנתיב של הפונקציה האחרונה, ונקליק על sub_411860, נגיע למסך הבא (שכאן מוצג ב-zoom out מקסימלי עקב היקף הקוד הרב):

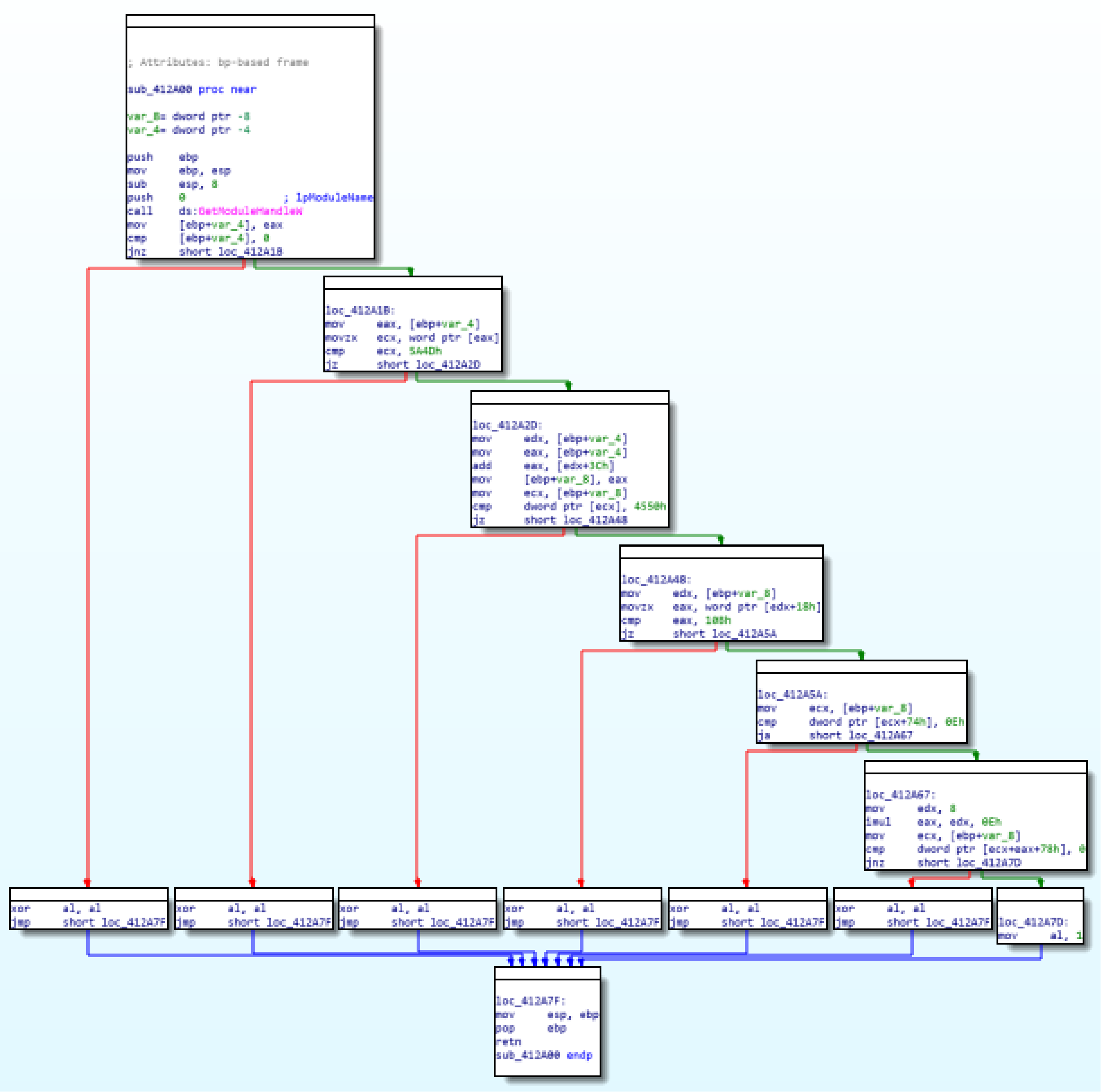


זה אכן נראה לא מעט קוד, אבל אין מה לחשוש- אנחנו לא צריכים לפענח את כולו. רק למצוא את מה שאנחנו מחפשים. כעת נשתמש בהנחת העבודה שלנו שהקריאה ל-main תהיה אחרונה לפני ה-exit. לכן במקום לקרוא את הקוד מלמעלה למטה, נקרא אותו מלמטה למעלה ונחפש פונקציה שנקראת לפני ה-exit.

ואמנם, אם נתמקד בקטע הקוד שנמצא בחלק השמאלי התחתון של הגרף שלנו, נמצא קריאות לפונקציות שיש בשם שלהן את המילה "exit":



הפונקציה שנקראת אחרונה היא sub_41111D. נקליק עליה ונקפוץ יחד עם ה-jmp אל כתובת 0x412A00. כעת נקבל קוד שנראה כך:

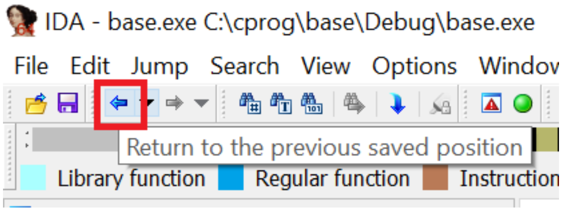


צורה כזו של קוד אופיינית לאוסף של תנאי if שיש ביניהם and ולאחר מכן else. רק אם כל התנאים מתקיימים אז מתבצע משהו, ואם אחד מהם אינו מתקיים אז קורה דבר מה אחר. אפשר לראות שכל תנאי מוביל אותנו במסלול הירוק אל התנאי הבא, או במסלול האדום הישר אל ה-else. בכל אופן, זה לא נראה כמו ה-main שלנו. כרמז לעתיד, בו נלמד על פורמט PE, שימו לב למספר ערכים שנמצאים בקוד ושמהווים חלק מתהליך הבדיקה, אם אחד מהערכים הללו אינו נמצא אז הפונקציה תחזיר 0:

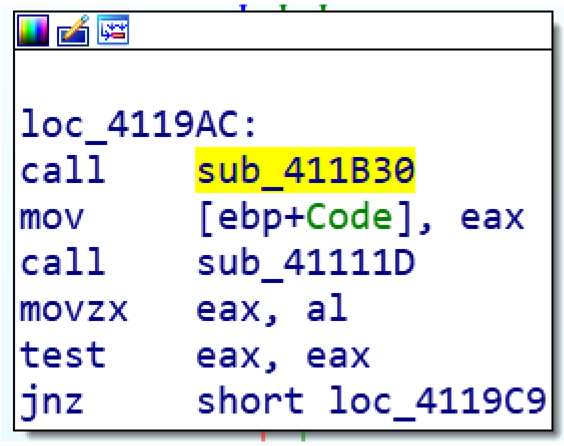
- 0x5A4D (בהמשך נראה שערך זה שייך ל-DOS Header)

- 0x4550 (בהמשך נראה שערך זה שייך ל-NT Header)
- 0x10B (בהמשך נראה שערך זה ייחודי ל-Optional Header)
- 0x0E (בהמשך נראה שזהו מספר של ה-Data Directories)

המסקנה היא שהגענו רחוק מדי בקוד שלנו ואנחנו נמצאים בחלק שבו ה-main כבר סיים לרוץ, לכן צריך לחזור אחורה. כדי לחזור אחורה, נשתמש בלחצן החזרה אחורה של IDA, המודגש פה באדום:



נחזור עד לקטע הקוד הבא-



הפעם נבחר ב-sub_411B30. הקלקה עליו תוביל אותנו אל הקוד הבא:

```
; Attributes: bp-based frame

sub_411B30 proc near
push    ebp
mov     ebp, esp
call    j__get_initial_narrow_environment_0
push    eax
call    j___p___argv_0
mov     eax, [eax]
push    eax
call    j___p___argc_0
mov     ecx, [eax]
push    ecx
call    sub_41127B
add     esp, 0Ch
pop     ebp
retn
sub_411B30 endp
```

נפלא. לפנינו קוד שבו רואים קריאה לפונקציה של קבלת משתני סביבה ולאחריה קריאות לפונקציות שמקבלות את argv ו-argc. הפרמטר argv מחזיק את הערכים שהמשתמש ביקש להעביר לפונקציית ה-main ואילו argc מחזיק את כמות הערכים. אלו דברים שצריכים להתרחש רגע לפני הקריאה ל-main. מיד אחרי השגת הפרמטרים ומשתני הסביבה יש קריאה ל-sub_41127B. לחיצה עליה ודילוג על ה-jmp שבדרך יובילו אותנו אל הקוד הבא:

```
; Attributes: bp-based frame

sub_412320 proc near

var_C0= byte ptr -0C0h

push    ebp
mov     ebp, esp
sub     esp, 0C0h
push    ebx
push    esi
push    edi
lea     edi, [ebp+var_C0]
mov     ecx, 30h
mov     eax, 0CCCCCCCCh
rep stosd
push    offset aHello   ; "Hello!\n"
call    sub_411366
add     esp, 4
xor     eax, eax
pop     edi
pop     esi
pop     ebx
add     esp, 0C0h
cmp     ebp, esp
call    sub_41134D
mov     esp, ebp
pop     ebp
retn
sub_412320 endp
```

זהו, סיימנו את הבסיס שיאפשר לנו לקחת קובץ הרצה, לפתוח אותו ב-IDA, למצוא את ה-main שלו ולהתבונן בקוד האסמבלי שלו.

2.3.4 הדרך הקלה

לאחר שלמדנו כיצד למצוא לבד את ה-main של פונקציה, אפשר לגלות לכם שבמקרים מסויימים אפשר לעבוד בדרך הקלה.

קמפלו את הקוד שלכם במצב Release.

במסך הבחירה הבא, בחרו "Yes" על מנת להעלות את קובץ ה-pdb.

כיוון שכעת IDA יודע שיש פונקציה בשם main ומה הכתובת שלה, הוא מביא אותנו ישירות ל-main בלי צורך לעבור דרך start. כמו כן, IDA מזהה באמצעות קובץ ה-pdb שהכתובת של הפונקציה שאנחנו קוראים לה היא בעצם printf ולכן מקל עלינו. כעת מסך הפתיחה נראה כך:

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

push    offset aHello   ; "Hello!\n"
call    printf
add     esp, 4
xor     eax, eax
retn
main endp
```

דרך קלה נוספת לאתר את קוד האסמבלי של קוד שכתבתם היא פשוט להריץ אותו ב-Debugger של Visual Studio. בזמן שאתם מריצים את הקוד במצב Debug, תוכלו לפתוח חלון Disassembly ולצפות בקוד האסמבלי ממש לצד הקוד בשפת C או CPP, בצורה מאד נוחה.

מצוידים בכלים שקיבלתם, אתם יכולים להמשיך ולחקור קוד. מומלץ להמשיך לקמפל קוד בשפת C ולראות איך התוצאה נראית בזיכרון המחשב.

המשך מחקר מוצלח :)

פרק 3 – תפקידי מערכת ההפעלה

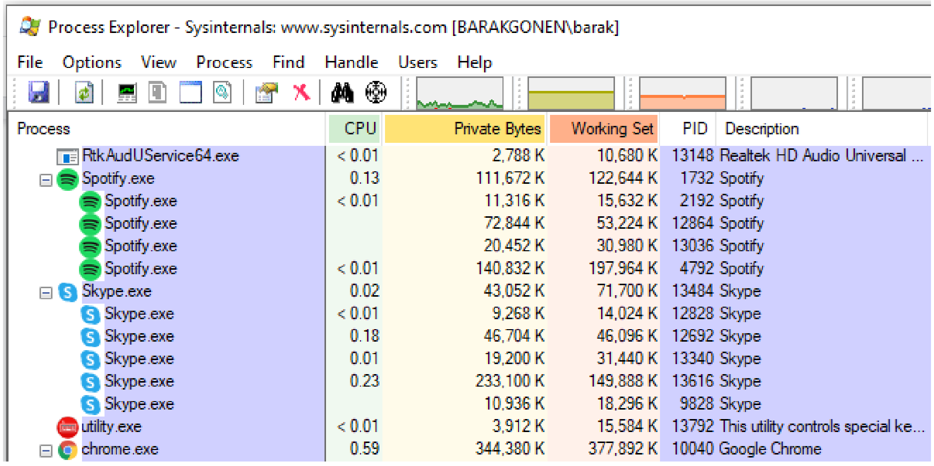
בפרק זה נלמד מהם ארבעת התפקידים של מערכת ההפעלה, נסקור בקצרה סוגים שונים של מערכות הפעלה ונבין במה הן נבדלות זו מזו. לאחר מכן נתקדם אל נושא אבטחה ונבין את מושג מעגלי ההרשאה של מערכת ההפעלה, מהו Userland ומהו ה-Kernel. לסיום נבצע תרגיל בו נחקור בעצמנו כיצד מטופלת קריאה של המשתמש להתקן חומרה.

לפני שניגש להסבר אודות תפקידי מערכת ההפעלה, נבין שני מושגים שיעלו בהמשך- Process ו-Driver. למרות שקיימים למושגים אלו תרגומים לעברית, נהוג לומר את המושגים הלועזיים ולכן גם ספר זה ישתמש ב"Process" ולא ב"תהליך", ב"Driver" ולא ב"מנהל התקן" .

3.1 מהם Processים

כפי שאנחנו יודעים כל תוכנה מורכבת משורות קוד. נוכחנו גם ששורות הקוד הללו נשמרות באזורים שונים של הזיכרון. כדי שתוכנה תפעל, שורות הקוד צריכות להיות מורצות על ידי המעבד, ואזורי הזיכרון של התוכנה צריכים להיות נגישים עבור המעבד. כלומר, אנחנו מבינים שצריך משהו שיקשור ביחד את שלושת המרכיבים הדרושים להרצת תוכנה: קוד, זיכרון, מעבד.

את החיבור הזה ממלא Process. כל תוכנה שרצה, עושה זאת בתוך Process שמטפל בה. לדוגמה, אם נריץ את הדפדפן כרום, ייפתח Process שמטפל בכרום- מכיל את הזיכרון הנדרש עבור כרום, שומר את שורות הקוד של כרום, מקבל זמן ריצה של המעבד עבור כרום. אם נפתח מסמך אקסל, ייפתח Process שמטפל באקסל. ה-Process של אקסל ייתן לאקסל את אותם שירותים שה-Process של כרום נותן לכרום. פיתחו את Process Explorer והביטו בכל ה-Processים שרצים על המחשב שלכם כרגע.



מה יקרה אם נפתח פעמיים את תוכנת אקסל, לדוגמה? האם ה-Process שמטפל באקסל יטפל בשתי תוכנות האקסל שרצות במקביל? נסו זאת. הקליקו פעמיים על אקסל. אם אין לכם אקסל, תוכלו לבצע את אותו ניסוי על כל תוכנה שתרצו.

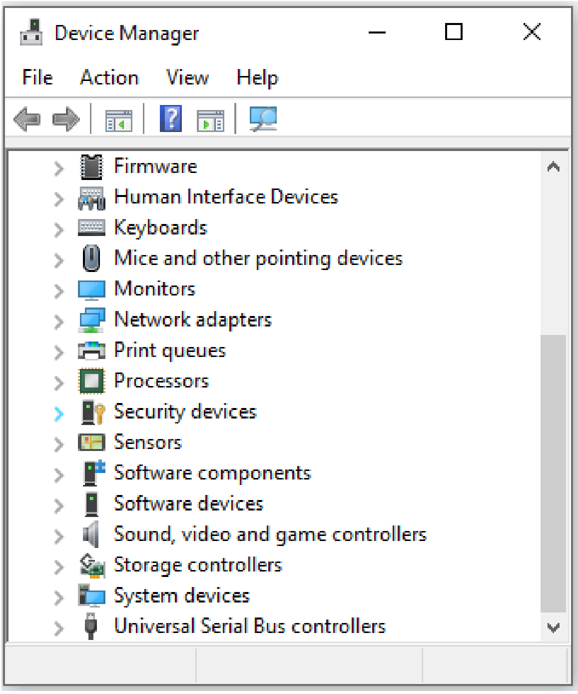
כפי שראיתם, נוצר Process חדש של אקסל. המונח המקצועי הוא Instance (בדומה ל-Instance של Class, מונח מוכר למי שמכיר תכנות מונחה עצמים). כלומר כל הפעלה של תוכנה יוצרת Instance של התוכנה, וכל Instance רץ תחת Process שמטפל בו.

בהמשך נתעמק ב-Processים, אך לטובת מה שאנחנו צריכים בשלב זה, הידע שיש לנו מספק.

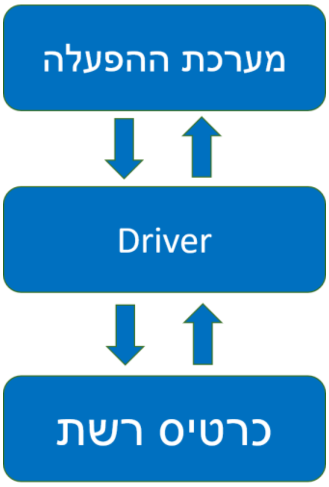
3.2 מהם Driverים

למחשב שלנו יש התקני חומרה שונים. לדוגמה, מקלדת, עכבר. אילו עוד התקני חומרה אתם מכירים? נסו למנות את כל התקני החומרה שאתם יכולים לחשוב עליהם שקיימים במחשב שלכם.

כעת לחצו על מקש ה- WinKey במקביל ללחיצה על מקש ה-X. (מקש ה-WinKey הוא המקש עם צורת הלוגו של Windows, שנמצא משמאל למקש הרווח). מבין האפשרויות בתפריט, בחרו "Device Manager", או "מנהל ההתקנים" אם אתם בגרסה עברית. כעת פרושים לפניכם כל התקני החומרה במחשב שלכם. אם הצלחתם לחשוב אפילו על שליש מקבוצות ההתקנים, אתם ממש טובים.



כדי שמערכת ההפעלה תוכל לתקשר עם התקני החומרה, נדרשות תוכנות ניהול התקנים, אלו ה-Driverים. ה-Driverים אחראים להעברת המידע בין התקני החומרה למערכת ההפעלה. לדוגמה, Driver של כרטיס רשת יקלוט מכרטיס הרשת את כל המידע שנשלח למחשב שלנו ויעביר אותו למערכת ההפעלה. מערכת ההפעלה תוכל, בין היתר, להעביר את המידע הלאה לדפדפן. בקשות שנשלח לדפדפן, אל אתרים שאנחנו רוצים לגלוש אליהם, יועברו ממערכת ההפעלה דרך ה-Driver של כרטיס הרשת אל כרטיס הרשת.

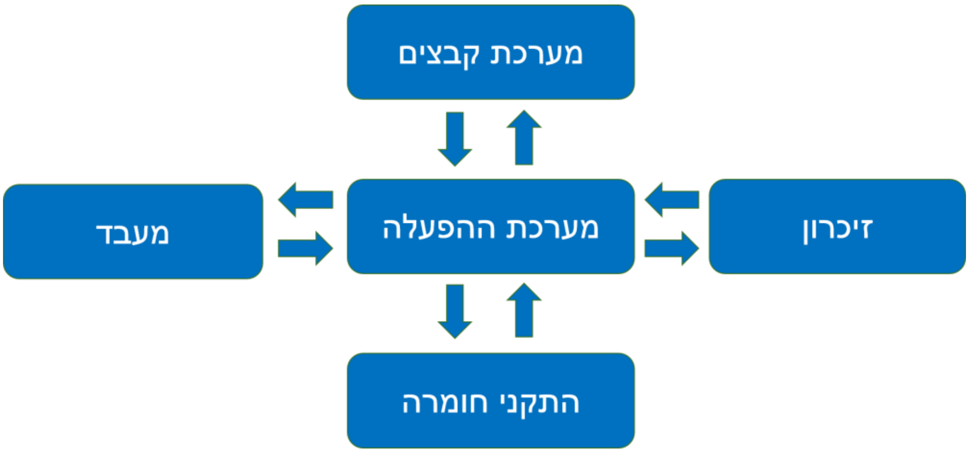


את ה-Driverים להתקני החומרה כותבים יצרני החומרה, לכן כאשר אנחנו מוסיפים למחשב חומרה חדשה, אנחנו צריכים להתקין תוכנה. זהו בעצם ה-Driver של החומרה שהתקנו. ה-Driverים שונים מתוכנות רגילות בשני דברים מרכזיים. ההבדל הראשון הוא שהם נכתבים באמצעות פונקציות מיוחדות שמערכת ההפעלה חושפת בפניהם, ואשר רמת ההרשאות שלהן גבוהה יותר. נדון בכך בהמשך כאשר נבין את מושג ה-Kernel. ההבדל השני הוא שה-Driverים רצים מאחרי הקלעים בלי שהמשתמש יצטרך לעשות משהו מיוחד כדי להפעיל אותם. העובדה ש-Driverים רצים מאחרי הקלעים באופן אוטומטי ללא פקודה מפורשת של המשתמש, ושיש להם הרשאות גבוהות במיוחד, הופכת אותם למעניינים מאד עבור מי שיש לו רצון להזיק.

3.3 תפקידי מערכת ההפעלה

מערכת ההפעלה היא הגשר שמקשר בין התוכנות שאנחנו מריצים לבין ארבעה גורמים:

- המעבד
- הזיכרון
- מערכת הקבצים
- התקני החומרה



המעבד: מחשב תמיד מריץ יותר מתוכנית אחת, ולכן חלוקת משאבי המעבד בין Processים היא הכרחית (כזכור אמרנו, בלי להכנס לפרטים, שכל תוכנה רצה תחת Process). נכון, יש מחשבים עם מספר מעבדים, אך גם בהם כמות ה-Processים שרצים עולה בהרבה על כמות המעבדים. התפקיד הראשון של מערכת ההפעלה הוא לקבוע איזה Process יקבל את משאבי המעבד בזמן שיתר ה-Processים ימתינו לתורם.

הזיכרון: כדי להריץ Process של תוכנה כלשהי יש לטעון את הקוד של התוכנה אל הזיכרון. תוכנות שמורות באופן שגרתי על הדיסק הקשיח, אולם הדיסק הקשיח הוא זיכרון איטי למדי ואילו המעבד היה נאלץ להמתין לו אז חלק משמעותי מזמן העיבוד של המעבד היה מתבזבז. לכן תוכנות ניטענות מהדיסק הקשיח אל זיכרון מהיר יותר. בפרק על הזיכרון נפרט בהרחבה. תהליך הטעינה לזיכרון מתבצע על ידי מערכת ההפעלה. לזיכרון המהיר יש גודל מוגבל ויש סיכוי טוב שהוא אינו מספיק עבור כל ה-Processים שפועלים ברגע מסויים על המחשב. לכן על מערכת ההפעלה לא רק להקצות לכל Process מקום בזיכרון אלא גם לדאוג לחלוקת משאבי הזיכרון בין כל ה-Processים.

כתוצאה מכך שכל ה-Processים חולקים את הזיכרון, למערכת ההפעלה יש גם תפקיד אבטחה- לדאוג לכך שכל Process כותב רק לאזור הזיכרון שמוקצה לו, ולא לאזור זיכרון של Process אחר. דמיינו מצב שבו Process תוכנה א', לדוגמה כרום, כותב לאותו אזור בזיכרון שבו שמורות פקודות ההרצה הבאות של תוכנה ב', לדוגמה אקסל. סביר שכאשר המעבד ינסה להריץ את הקוד של אקסל, התוכנה תקרוס. אפילו חמור מכך- ניתן יהיה לתכנת תוכנה זדונית שמשנה את הקוד שמריצים Processים אחרים, כך שבמקום לבצע את המשימה המקורית שלהם הם יבצעו פעולות אחרות, זדוניות, וכך לעקוף מנגנוני אבטחה. אחרי הכל, מי יחשוד שתוכנת אקסל או כרום מבצעות דברים זדוניים?

מערכת הקבצים: תפקידה של מערכת ההפעלה דומה לתפקיד של ספרן בספריה. ראשית יש לקבוע שיטת קיטלוג שתאפשר מציאה קלה של כל ספר, כדי שלא נצטרך לחפש את הספר שאנחנו מעוניינים בו בין כל הספרים בספריה. באופן דומה, יש לקבוע שיטת קטלוג שתאפשר למצוא כל קובץ שאנחנו רוצים. מערכת ההפעלה צריכה לדעת לא רק היכן הקובץ שמור, אלא גם היכן הקובץ מתחיל והיכן הוא מסתיים. כמו שספר בספריה עשוי להיות מושאל, גם קובץ עשוי להיות בשימוש על ידי תוכנה כלשהי. עשו ניסוי קטן- פיתחו באמצעות IDA את הקובץ hello.exe שיצרתם קודם לכן. כעת פיתחו Visual Studio, פיתחו את ה-solution שנקרא hello.sln ונסו לקמפל את הפרוייקט וליצור קובץ הרצה. כאשר תעשו זאת תקבלו הודעת שגיאה- מערכת ההפעלה הבחינה בכך שהקובץ hello.exe פתוח על ידי IDA, והיא אינה מאפשרת לתוכנה אחרת לשנות את הקובץ עד ש-IDA לא תשחרר אותו. שמות של מערכות קבצים לדוגמה הן FAT, NTFS ו-EXT.

התקני חומרה: ראינו שיש תוכנות שנקראות Driverים ואשר תפקידן הוא לקשר בין החומרה לבין מערכת ההפעלה. אם כך, מערכת ההפעלה צריכה להיות בקשר מול כל ה-Driverים ולדאוג להעברה של המידע מהם ואליהם. מערכת ההפעלה צריכה גם לדאוג לעדיפויות- מה קורה אם גם ה-Driver של כרטיס הרשת וגם ה-Driver של המקלדת מבקשים שירות ממערכת ההפעלה? האם לתת למשתמש להמתין, או לתת לכרטיס הרשת להמתין? כמו כן, במקרים רבים, מספר תוכנות חולקות התקן חומרה אחד. לדוגמה, תוכנת אקסל ותוכנת word מנסות לשמור מידע לדיסק הקשיח. מי תקבל שירות ראשונה?

ננסה לדמיין מצב שבו למחשב שלנו אין מערכת הפעלה. איך הדברים יתנהלו?

ראשית, המעבד שלנו יוכל להיות מוקצה רק לProcess אחד. כלומר בכל רגע נוכל להריץ רק תוכנה אחת ונצטרך לסגור אותה כדי להריץ תוכנה אחרת. אפשר אולי לעבוד ככה, לדוגמה במערכות פשוטות שיש להן רק משימה אחת, אבל מערכת הפעלה כזו אינה מתאימה למשתמש שיש לו בכל רגע מספר חלונות פתוחים.
שנית, נצטרך לטעון בעצמנו את התוכנה אל הזיכרון, כי אף אחד לא יעשה זאת בשבילנו. נצטרך לוודא שכל הכתובות בזיכרון שהתוכנה משתמשת בהן אכן פנויות לשימוש, ואם יש בעיה- נצטרך באופן ידני לשנות את הכתובות.

שלישית, נתקשה מאד לעבוד עם התקני חומרה, מכיוון שלדרייברים לא יהיה מול מי לעבוד. לדוגמה, אם נקליד את האות "א" היא לא בהכרח תודפס למסך, כיוון שלא יהיה מי שיאסוף אותה מהדרייבר של המקלדת ויעביר אותה לדרייבר של כרטיס המסך.
ולסיום, סידור הקבצים בדיסק הקשיח יהיה לגמרי באחריות שלנו. נצטרך להחזיק תיעוד כלשהו שאומר באיזה כתובות בדיסק הקשיח נמצא כל קובץ. תהיה לנו בעיה לשמור את התיעוד הזה בקובץ, מכיוון שאם נשכח את הכתובת של קובץ התיעוד נאבד את הגישה לכל הקבצים שיש לנו בדיסק הקשיח... בסופו של דבר נצטרך לתעד את הדברים עם נייר ועט.
אם קבלתם את התחושה שבלי מערכת הפעלה המחשב שלכם היה לא שמיש לחלוטין- מעולה, זו היתה המטרה ☺

3.4 סוגים של מערכות הפעלה

לכל צרכן יש צרכים שונים ממערכת ההפעלה: הצרכים של מחשבים אישיים או לפטופים שונים מהצרכים של שרתי אינטרנט. הצרכים של סמארטפונים שונים מהצרכים של מערכות משובצות מעבדים שנמצאות לדוגמה במכוניות.
מהן הדרישות העיקריות של כל אחד מהצרכנים שהוזכרו?
מחשבים אישיים ולפטופים זקוקים למערכת הפעלה ידידותית למשתמש, שמאפשרת תגובה מהירה של תוכנות והרצה שלהן בקלות. אבטחה היא חשובה, אבל במידה סבירה ובלי להקשות יותר מדי על המשתמש.
שרתים ברשת האינטרנט צריכים מערכת הפעלה שתעבוד מסביב לשעון ללא הפרעות. בניגוד למחשב אישי, שסביר שיבצע איתחול מחדש לצורך התקנת תוכנות אחת לכמה ימים, שרת אינטרנט חייב לפעול ברציפות. כמו כן נושא האבטחה הוא קריטי, ולעומת זאת שרת האינטרנט מטופל על ידי מומחה מחשבים שנושא הידידותיות של מערכת ההפעלה פחות חשוב לו מאמינות יציבות ואבטחה.
סמארטפונים, כידוע לכל משתמש, תמיד זקוקים לעוד זמן סוללה ויש להם כמות זיכרון קטנה יותר מאשר מחשב אישי. אם תשוו את גודל זכרון ה-RAM (קיצור של Random Access Memory, זיכרון מהיר שאינו נשמר בין כיבויים של המחשב) של המחשב שלכם לזה של הסמארטפון שלכם, סביר שתמצאו שבמחשב יש פי כמה זיכרון RAM, אלא אם כן יש לכם סמארטפון מתקדם מאד ומחשב אישי מיושן יחסית. לכן למערכת ההפעלה יש תפקיד חשוב בשמירת הסוללה של סמארטפונים, תוך הסתפקות בכמות זיכרון מועט יחסית.
מערכות משובצות מעבדים, או Embedded systems כמו שהם נקראים באנגלית, זקוקות בדרך כלל למערכת הפעלה פשוטה ביותר, והממשק של המשתמש הוא מוגבל למדי. כמות הזיכרון שיש במערכות

משובצות צפויה להיות קטנה למדי, ולכן הגודל של מערכת ההפעלה עצמה הוא קריטי, כדי שלא תתפוס לבדה את כל הזיכרון הפנוי.

3.5 התפתחות המעבדים ומערכות ההפעלה

מערכות ההפעלה לא נוצרו מראש כפי שהן כיום אלא עברו גלגולים שונים. הבעיות שעיצבו את מערכת ההפעלה הן גם מעניינות וגם מקילות את הלימוד וההבנה של מערכת ההפעלה כפי שהיא כיום.

אחד הדברים המרכזיים ביותר שמשפיעים על מערכת ההפעלה הוא כמובן המעבד. באופן כללי מערכת ההפעלה Windows נבנתה סביב משפחת מעבדי ה-x86 של אינטל, ועד היום אפשר למצוא לכך שרידים בולטים בכל מחשב שמריץ Windows. תחת התיקיה "C:\" תמצאו את התיקיה "Program files(x86)"

3.5.1 מעבד ה-8086, Real Mode

נחזור בזמן לשנת 1978, שנה קסומה בה להקת אבבא ולהקת כוורת היו פופולריים ואינטל הוציאה את מעבד ה-8086 ההיסטורי.

מעבד ה-8086 עבד בארכיטקטורה של 16 ביט, כלומר הרגיסטרים שלו היו בגודל 16 ביט. באופן עקרוני באמצעות רגיסטר של 16 ביט ניתן להגיע לכמות של 2 בחזקת 16 כתובות, כלומר 64KB זיכרון בלבד, אולם אינטל יצרה פס כתובות (Address Bus) בגודל 20 ביט ובאמצעות תכסיס מחוכם (שילוב של segment+offset) הצליחה לאפשר למעבד גישה לכמות מרשימה של 1MB זיכרון. כל הפרטים על כך נמצאים בספר לימוד האסמבלי של המרכז לחינוך סייבר.

ממש כמו היום, קוד של תוכנה כלל כתובות שהמעבד צריך לגשת אליהן. כאשר התוכנה ביקשה לגשת לכתובת מסויימת, המעבד ניגש אליה. צורה זו של גישה לזיכרון נקראת Real Mode. האם אתם יכולים לחשוב על בעיית אבטחה כלשהי ב-Real Mode? מיד נענה על השאלה.

מערכת ההפעלה שמיקרוסופט פיתחה על בסיס מעבד ה-8086 נקראה DOS, קיצור של Disk Operating System. הגרסה הראשונה של DOS יצאה בשנת 1981. פרט טריוויה מעניין: למעשה, מיקרוסופט רכשו את בסיס הקוד של DOS תמורת 75,000$. במקור התוכנה נקראה QDOS, ראשי תיבות של Quick and Dirty Operating System. כפי שהשם מרמז, אפשר לחשוד שמפתח ה-QDOS לא חשב שזו תוכנה איכותית במיוחד...

כפי שזכור לכם מספר לימוד האסמבלי, מערכת ההפעלה DOS כללה פסיקות, או באנגלית Interrupts, אשר הינן פונקציות של מערכת ההפעלה, כלומר קוד שמערכת ההפעלה מריצה בתגובה לארועים שונים. הפסיקות הופעלו או בעקבות בקשות של התקני חומרה שונים או בעקבות קוד של המתכנת. דוגמה לפסיקה שמופעלת בעקבות בקשה של התקן חומרה: אם המשתמש הקיש על תו כלשהו במקלדת, הופעלה פסיקה שאספה את התו מהמקלדת והעבירה אותו לתוך הזיכרון הנגיש למעבד. פסיקות שונות פותחו עבור פעולות נפוצות, לשימוש המתכנתים. דוגמה לפסיקה שמופעלת באמצעות קוד של המתכנת: הדפסה של מחרוזת תווים למסך.

כל תוכנה יכולה לקרוא בלי מגבלה לפסיקות של מערכת ההפעלה באמצעות הפקודה "int". הפסיקה המפורסמת ביותר של DOS נקראת int 21h, והיא מקבלת באמצעות הרגיסטר ah את הסוג המדויק של הפעולה המבוקשת.
להלן תוכנית שמיועדת למעבד ה-8086. התוכנית מדפיסה למסך את הטקסט השמור במשתנה msg, ואז יוצאת בצורה מסודרת:

```
mov     dx, msg
mov     ah, 9
int     21h

mov     ax, 4c00h
int     21h
```

אם ברצונכם להבין את התוכנית לעומק, ספר לימוד האסמבלי של המרכז לחינוך סייבר ידריך אתכם. אולם גם בלי לדעת אסמבלי אפשר לראות שהתוכנה משתמשת בשתי פסיקות, האחת בשביל פעולת ההדפסה והשניה בשביל יציאה.

צריך להדגיש שלמרות שבמקרה של הקוד המצורף נעשה שימוש בפסיקות של DOS, אפשר לבצע את אותן פעולות באמצעות פסיקות של המעבד עצמו (פסיקות BIOS, שפותחו על ידי אינטל) ואפילו באמצעות גישה ישירה לחומרה. מערכת ההפעלה DOS לא מונעת את זה מאיתנו. האם אתם יכולים לחשוב על בעיית אבטחה שעלולה להיות כתוצאה מהשימוש ב-Real Mode?

נסקור מקצת מבעיות האבטחה.

ראשית, צורת הגישה לזיכרון Real Mode מאפשרת מצב שבו Process מסויים מבקש מהמעבד שייגש לאזור זיכרון ב-RAM שכלל לא שייך לו אלא ל-Process אחר ששמור בזיכרון. אפשר לשנות את הזיכרון, לשנות ערכים של משתנים ואולי גם לשנות פקודות הרצה: במצב Real Mode אין את המגבלות על Readable, Writeable, Executable. כלומר אפשר לקחת Process תמים ולשנות את קוד ההרצה שלו כך שהוא יהפוך לנוזקה שתשלח לתוקף את כל המידע שיש במחשב שלנו.

שנית, גישה לא מוגבלת להתקני החומרה עלולה לאפשר ל-Process לשמור אותם לעצמו במקום לחלוק אותם עם Processים אחרים. אמנם מערכת ההפעלה דואגת להקצות את משאבי החומרה לפי סדר

ועדיפויות, אבל מי שיודע לפנות ישירות להתקני החומרה, "מעל הראש" של מערכת ההפעלה, עוקף את מנגנוני ההקצאה של מערכת ההפעלה.

שלישית, האפשרות לשנות כל קוד עלולה לאפשר לתוקף לשנות גם את הקוד של מערכת ההפעלה, שגם הוא שמור בזיכרון ה-RAM. ברגע שמשנים את קוד מערכת ההפעלה- המשחק נגמר. התוקף יכול להשיג כל מה שהוא רוצה, אין מי שימנע ממנו.

כאמור אלו רק מקצת מבעיות האבטחה, אלו שקשורות באופן הדוק ל-Real Mode. במרוצת השנים נתגלו עוד בעיות אבטחה ונוספו מנגנונים הן במעבדים והן במערכות ההפעלה כדי לסגור פרצות. בחלק מבעיות האבטחה נדון בהמשך.

בשלב זה אפשר לקבוע, שמעבד האינטל 8086 ומערכת ההפעלה DOS היו הישג טכנולוגי נפלא, אך מבחינת אבטחה ישנן בעיות, שדורשות דור חדש של מעבדים ומערכת הפעלה.

3.5.2 מעבד ה-386

שנת 1985. ישראל שולחת לאירוויזיון את "עולה עולה" ואילו אינטל מציגה את הפיתוח החדש והמהפכני- מעבד ה-386. כדי להבין עד כמה מהפכני היה המעבד הזה, הוא כלל לא רק שדרוג מהירות משמעותי, אלא גם שדרוג רגיסטרים, הכפלה פי 4000 של אזור הזיכרון הנגיש ומערכת אבטחה משוכללת. כל אחד מהשדרוגים האלה בפני עצמו הוא קפיצת מדרגה, השילוב של הכל ביחד הוא כל כך משמעותי שבמשך 20 שנה, עד שהוצגה ארכיטקטורת ה-64 ביט, כל המעבדים עבדו פחות אותו יותר באותה דרך. אפילו היום, למרות עשרות השנים שחלפו, ניתן להריץ תוכנות בגרסת 32 ביט שמעבד ה-386 הציג לעולם.

נפרט אודות השיפורים החשובים שהוצגו במעבד ה- 386.

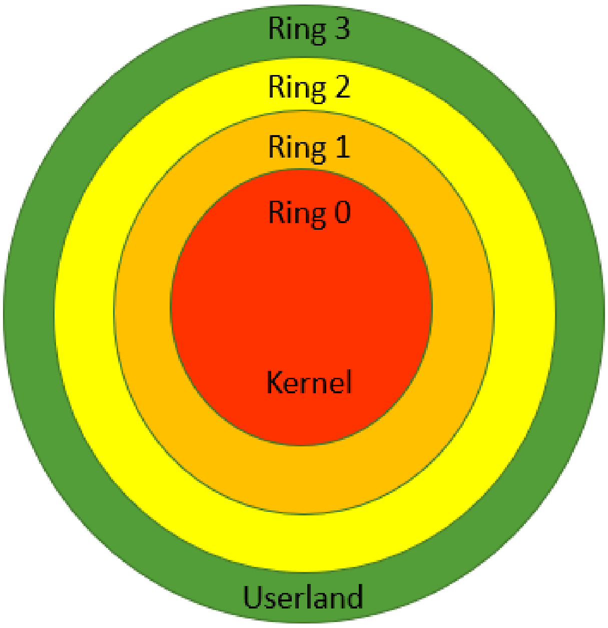
ראשית, הכפלה של הרגיסטרים- מ-16 ביט ל-32 ביט. רגיסטרים של 32 ביט יכולים לקרוא ולכתוב 4 בתים בכל פעולה של העתקת זיכרון, לכן למעבד 32 ביט יש יתרון משמעותי על פני מעבד 16 ביט, גם אם שניהם מבצעים את אותה כמות של פעולות בשניה.

שנית, הכפלת גודל הרגיסטרים לוותה בהגדלת ה-Address Bus ל-32 ביט. כלומר באמצעות רגיסטר יחיד ששומר כתובת בזיכרון ניתן לגשת ל-2 בחזקת 32 כתובות נפרדות. דבר זה מגביל את גודל זיכרון ה-RAM לכ-4GB, כמות זיכרון שלקח עשרות שנים עד שהפכה למגבלה מעשית.

שלישית, אינטל החליפה את Real Mode ב-Protected Mode. כיוון שזהו רעיון המשמש גם כיום, המקום לדון בו הוא אינו סקירה היסטורית. כעת נסיים את הסקירה ההיסטורית שלנו, הרקע שקבלנו יסייע לנו להבין את הנושא החשוב הזה.

3.5.3 מעגלי הרשאה ו-Protected Mode

במצב Protected Mode לכל תוכנה יש רמת הרשאות שהיא נמצאת בה, ורק רמת ההרשאות העליונה ביותר יכולה לגשת ישירות לזיכרון או להתקני חומרה. שיטה זו נקראת "מעגלי הרשאה", או באנגלית "Rings of Protection", כפי שממחיש האיור הבא:



המונח "Rings of Protection" נהוג כשמדברים על מעבדי אינטל, אולם גם למעבדים אחרים יש מנגנונים דומים. לדוגמה במעבדי ARM יש Exception Levels.

תוכנות שהמשתמש מריץ תמיד ירוצו במעגל 3. מעגל זה נקרא Userland. לדוגמה, אם תפעילו דפדפן כרום, ה-Process שלו ירוץ ב-Userland. לתוכנות אלו אין אפשרות לגשת באופן ישיר לזיכרון או להתקני חומרה. אם כך, איך תוכנה שהמשתמש מריץ יכולה לגשת לזיכרון או לפנות לחומרה? בקרוב נענה על שאלה זו.

מעגל 0 נקרא ה-Kernel, הוא הכינוי לליבת מערכת ההפעלה. תוכנות שרצות בהרשאה של Kernel יכולות לעשות ככל העולה על רוחן, אין שום דבר שמגביל אותן. על כן, ה-Kernel יועד לשימוש רק על ידי מערכת ההפעלה.

מעגלים 1 ו-2 תוכננו על ידי אינטל לשימוש על ידי תוכנות שזקוקות להרשאות רבות יותר מאשר תוכנות רגילות, לדוגמה- Driverים.

נסכם מהם הדברים שצריך למנוע מ-Process שרץ ב-Userland לבצע:

1. לכתוב בלי בקרה לזיכרון של Process אחר: כתיבה לזיכרון של Process אחר עלולה לשנות את הדרך בה ה-Process האחר רץ, על כן זו פעולה רגישה מאד. חשוב לציין כי במקרים מסויימים Process צריך לכתוב לזיכרון של Process אחר, אולם פעולה כזו צריכה להתבצע תוך בקרה של מערכת ההפעלה, שמספקת פקודות מתאימות.
2. לכתוב לחומרה באופן ישיר: בצורה זו ניתן להשתלט על משאבי החומרה, למנוע מ-Processים אחרים גישה לחומרה ובאופן כללי להזיק לפעולה התקינה של המחשב.

3. לכתוב לדיסק הקשיח: כתיבה לדיסק הקשיח בלי מגבלה עלולה לשנות את הקוד של תוכנות שונות ששמורות עליו, וכך כאשר התוכנות הללו יורצו הן יבצעו דברים שונים ממה שהן אמורות לעשות.
4. שינוי טבלת הפסיקות של מערכת ההפעלה: ראינו שלמערכת ההפעלה יש אוסף של פונקציות, Interrputים, שמופעלים כתוצאה מארועים חומרתיים. אם Process יוכל לשנות את הקוד של הפונקציות הללו הוא יוכל לגרום למערכת ההפעלה להריץ את הקוד המבוקש במעגל ההרשאות של ה-Kernel.
5. שינוי ההרשאות של ה-Process: כמובן שאם מאפשרים ל-Process לגשת למקום בו נשמרות ההרשאות שלו ולשנות אותן, ה-Process יכול להעביר את עצמו למעגל ההרשאות של ה-Kernel ומשם לבצע כל פעולה.

3.6 הרחבה – מעגל ההרשאה של Driverים

למרות שמעבדי אינטל תומכים במעגלי ההרשאה, הם מספקים את השירות הזה לכותבי מערכת ההפעלה אך אינם כופים על מתכנתי מערכת ההפעלה את השימוש במעגלי ההרשאה. מיקרוסופט, משיקולים שונים, בחרה שלא להשתמש בכל ארבעת מעגלי ההרשאה, אלא רק ב-Userland וב-Kernel, מעגל 3 ומעגל 0. מה שמעלה את השאלה- באיזו הרשאה רצים Driverים? והתשובה- גם במערכת ההפעלה Linux וגם ב-Windows, הדרייברים רצים ב-Kernel. כן, אם הנכם כותבים Driverים ל- Windows או ל- Linux, אתם מקבלים הרשאה של Kernel.

3.6.1 פרצות אבטחה ב-Driverים

כיצד מונעים מחברה זדונית כלשהי לפתח Driver מרושע, שנראה תמים אך למעשה הוא תוכנת ריגול או נוזקה?

Driverים שמופצים על ידי Linux נמסרים כקוד פתוח לקהילת מפתחי Linux, מתוך הנחה שהגשת הקוד לקהילה תסייע במניעת ואיתור פרצות אבטחה. רק Driver שעבר ביקורת של הקהילה מופץ יחד עם Linux.

ב-Windows התהליך הוא שונה. מי שרוצה לפתח Driver ל-Windows יכול לעשות זאת ללא קושי כל עוד הוא אינו מבקש שה-Driver יהיה בעל אישור של מיקרוסופט (האישור נקרא WHQL – Windows Hardware Quality Labs). כלומר כל Driver שאינו מאושר על ידי מיקרוסופט הוא בסיכון גבוה למשתמש, כיוון שאף אחד אינו בודק מה הוא באמת עושה.

עבור Driverים שמוגשים לאישור WHQL של מיקרוסופט מתקיימת בדיקה שפרטיה ידועים לחברת מיקרוסופט, ולכן ניתן להעריך שצריך יותר מאמץ בשביל ליצור קוד זדוני. אולם העובדה שמידי פעם פעם מיקרוסופט פוסלת Driverים שמתגלות בהן בעיות אבטחה, אינה אומרת שאף Driver עם בעיות אבטחה אינו מצליח לעבור את הבדיקות של מיקרוסופט.

הסיפור הבא, מחודש מרץ 2019, הוא על אודות פרצת אבטחה שנתגלתה על ידי מיקרוסופט ב-Driver של חברת Huawei, ענקית החומרה הסינית.

https://arstechnica.com/gadgets/2019/03/how-microsoft-found-a-huawei-driver-that-opened-systems-up-to-attack/

הדברים שלמדנו עד כה מאפשרים לנו להבין היטב את מהות הפרצה: מפתחי ה-Driver של Huawie יצרו מנגנון שמאפשר לקחת זיכרון מכל מקום, גם מה-Kernel, להעתיק אותו לאזור זיכרון בעל הרשאות קריאה וכתיבה, ולהעתיק אותו חזרה לכל מקום- גם ל-Kernel. לכאורה ה-Driver אינו מבצע שום דבר זדוני בפני עצמו, אולם תוכנה זדונית עלולה להשתמש בפרצה. לפני שתמשיכו לקרוא, בידקו את עצמכם- חישבו איך אפשר להשתמש בפרצה זו?

- קוד של מערכת ההפעלה, שנמצא ב-Kernel ללא הרשאות גישה למשתמש, יועתק באמצעות הפרצה אל אזור זיכרון שנמצא ב-Userland
- הקוד שנמצא ב-Userland נמצא תחת הרשאות כתיבה, ולכן ניתן לשנות אותו ולהפוך אותו לקוד זדוני
- לאחר השינוי, באמצעות פרצת האבטחה של ה-Driver, ניתן להעתיק את הקוד חזרה אל ה-Kernel. למעשה שינינו את הקוד של מערכת ההפעלה, ברגע שאנחנו כותבים את מערכת ההפעלה, השגנו שליטה מלאה על המחשב המותקף.

נניח שהייתם מעוניינים לתקוף מחשבים ושהיה ביכולתכם לפתח Driverים, סביר שלא הייתם גורמים ל-Driver עצמו לבצע דברים זדוניים. זה שקוף מדי, ואילו הקוד הזדוני שלכם היה נחשף הייתם עלולים להיות חשופים לסנקציות משפטיות, פליליות וכלכליות. יותר סביר שהייתם משאירים ב-Driver שלכם פרצה, שידועה רק לכם, כך שה-Driver עצמו אינו עושה שום דבר זדוני – חוץ מזה שהוא מקל עליכם לתקוף מחשבים שבהם הוא מותקן. ומה תעשו אם בדיקת אבטחה תגלה את הפרצה? פשוט תתנצלו ותוציאו מיד עדכון שמתקן את הפרצה, עדכון שייתכן שהכנתם אותו מראש ואשר, אולי, כולל פרצה אחרת.

ובכן, מה עשתה חברת Huawie לאחר שהפרצה התגלתה? הוציאה עדכון תוכנה שמתקן אותה ובכך הסיפור הסתיים. הכתבה מסתיימת באמירה ש"התופעה מדגישה כמה מהתופעות הנוראות שיצרני חומרה עושים כשהם מנסים לכתוב תוכנה". הקורא נותר להחליט לבד אם הטעות היתה בתום לב או לא.

3.7 ביצוע Syscall ופסיקת Sysenter

כאשר סקרנו את נושא מעגלי האבטחה, השארנו שאלה פתוחה: אם כל Process שרץ ב-Userland מנוע מלגשת לחומרה ולזיכרון, איך Processים בכל זאת מצליחים לבצע פעולות כלשהן? לדוגמה, הרי כל אחד יכול לפתוח מסמך word ולשמור את הקובץ לדיסק הקשיח. ברור ש-word רץ ב-Userland, כי הוא מופעל על ידי המשתמש. אם כך, איך אין פה סתירה לרעיון מעגלי ההרשאה?

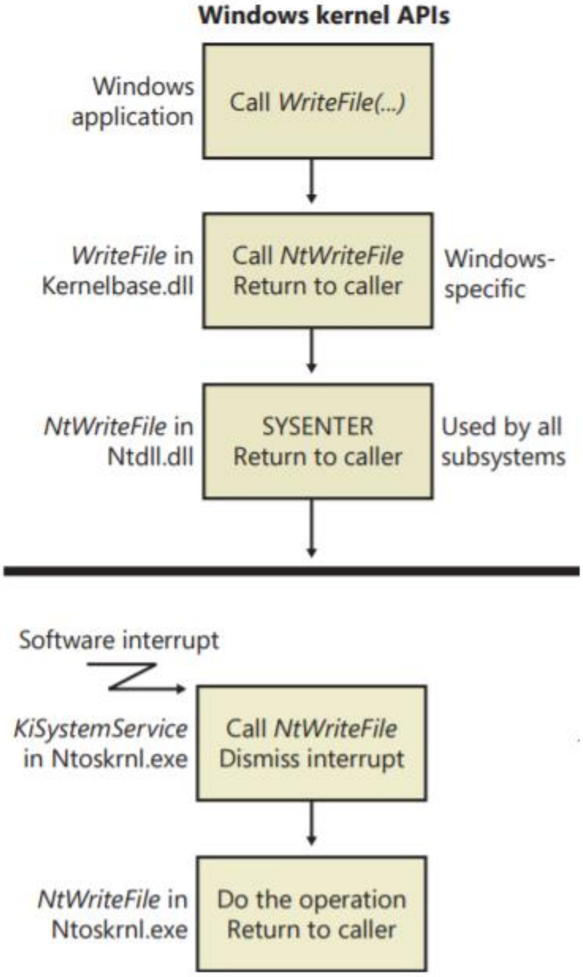
התשובה היא Syscall, קיצור של System Call.

כאשר Process רוצה לגשת למשאב כלשהו, הוא יבצע Syscall, שהינו שם כללי לפניה אל מערכת ההפעלה לצורך ביצוע פעולה כלשהי. מערכת ההפעלה חושפת למשתמש אוסף של פונקציות, שמיועדות לקריאה מ-

Userland. במערכת ההפעלה Windows, פונקציות כאלו קיימות לדוגמה בקבצים kernel32.dll, kernelbase.dll ו-gdi32.dll.

כדי להבין בדיוק את התהליך, נבדוק מה שרשרת הקריאות שמתבצעת כאשר Process שרץ ב-Userland קורא לפונקציה WriteFile. התיעוד על כך נמצא בספר שנקרא Windows Internals.

1. בשלב הראשון מתבצעת קריאה לפונקציה WriteFile, שנמצאת בתוך Kernelbase.dll (למרות שמו, זהו אינו ה-Kernel, אלא אוסף של פונקציות שרצות בהרשאה של Userland, והן מתועדות על ידי מיקרוסופט כדי שכל מי שמעוניין יוכל להשתמש בהן)
2. בשלב השני, קריאה לפונקציה NtWriteFile, שנמצאת בתוך Ntdll.dll (קוד נוסף של מיקרוסופט, שרץ גם הוא ב-Userland, אך אינו מתועד. מתכנתים אינם אמורים לקרוא לפונקציות מתוכו)
3. בשלב השלישי, מבוצעת פסיקת תוכנה שנקראת SYSENTER. פסיקת תוכנה זו מכניסה אותנו לעולמו של ה-Kernel, אנחנו עוברים למעגל אבטחה 0 ומקבלים מצביע חדש ל-Stack, כדי לקיים הפרדה מה-Stack שמשמשת את הפונקציות שרצות ב-Userland. נדון על כך עוד בהמשך. פונקציית ה-Kernel הראשונה שמתבצעת נקראת KiSystemService והיא נמצאת בתוך Ntoskrnl.exe.
4. בשלב הרביעי, נקראת הפונקציה NtWriteFile מתוך Ntoskrnl והיא זו שמבצעת את הגישה לחומרה וקריאת תוכן הקובץ המבוקש.



מקור: *Windows Internals*

כמובן, שה-Kernel לא חייב לעשות כל מה שה-Process שנמצא ב-Userland מבקש ממנו. ה-Kernel יבדוק שכל התנאים שצריכים לקרות כדי שהבקשה תתבצע אכן קיימים. קודם כל, של-Process שרץ ב-Userland יש הרשאה לבצע את הפעולה הנדרשת. לאחר מכן, שאין בעיה אחרת שמונעת להשלים את הבקשה. לדוגמה- כתיבה לכתובת בזיכרון שמוגדרת לקריאה בלבד, נסיון להשתמש במשאב שתפוס על ידי Process אחר וכו'. רק אם כל התנאים מתקיימים, ה-Kernel ייתן את השירות המבוקש.

3.8 שימוש ב-Objects וב-Handles

נעסוק כעת בשני מושגים שיסייעו לנו להבין טוב יותר את מערכת ההפעלה Windows.

דברים רבים במערכת ההפעלה מאורגנים במבנה נתונים שנקרא אובייקט, באנגלית Object. מיד נראה כיצד נראה מבנה הנתונים הזה. חלק מהמושגים שהכרנו עד כה הם למעשה אובייקטים של מערכת ההפעלה. לדוגמה, Process הוא אובייקט, קובץ הוא אובייקט.

כפי שאתם אולי מכירים משפות תכנות, לאובייקט יש מתודות (Methods) ומאפיינים (Attributes) שמוגדרים עבורו. לדוגמה, לאובייקט מסוג קובץ מוגדרת מתודה CreateFile. לאובייקט קובץ יש מאפיינים כגון שם, אפשרויות שיתוף ועוד.

נוסף ל-Process ול-File קיימים עוד סוגים רבים של אובייקטים של Windows. בין היתר:

Thread, Mutex, Registry key, Event, Memory section.

אלו דוגמאות לאובייקטים שטרם הכרנו אך נכיר בהמשך הלימוד.

כל האובייקטים מנוהלים על ידי ה-Object Manager של Windows. ישות זו אחראית ליצור אובייקטים, להחזיק את המצב שלהם ולשנות אותם במקרה הצורך. תפקידי ה-Object Manager הם:

- לאפשר לתת לאובייקטים שמות קריאים, שאנשים יכולים להבין
- לשתף אובייקטים בין Processים
- להגן על אובייקטים מגישה לא מורשית
- להחזיק מנגנון שבודק אם האובייקט בשימוש

אפשר לחשוב על ה-Object Manager בתור רוקח בבית מרקחת. הרוקח משרת את הלקוחות שפונים אליו. ישנן תרופות שאי אפשר סתם כך לבקש מהרוקח אלא צריך להציג לו מרשם של רופא. הרוקח יבדוק את הפרטים במרשם, לדוגמה אם המרשם בתוקף ואם המרשם אכן מיועד לאדם שמשתמש בו, ורק אם הכל בסדר הרוקח יעניק את התרופה ללקוח.

בדומה לרוקח, גישה לאובייקט צריכה לעבור דרך ה-Object Manager. הבקשה תיבדק על ידי ה- Object Manager בהתאם לרמת ההרשאות של ה-Process המבקש וסוג הגישה המבוקשת. אם בקשת הגישה תיענה בחיוב, התוצאה תהיה Handle שבאמצעותו אפשר לבצע פונקציות שונות. בואו נתבונן בדוגמה מתוך MSDN. כאשר אנחנו מבקשים גישה לקובץ נשתמש בפונקציה CreateFile. כפי שאפשר לראות, היא מחזירה משתנה מטיפוס Handle:

```
HANDLE CreateFileA(
  LPCSTR                lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

נוכל להשתמש ב-Handle שהשגנו כדי לבצע פעולות שונות על אובייקט הקובץ שלנו. לדוגמה, אם נרצה לקרוא ממנו. נספק את ה-Handle בתור הפרמטר הראשון לפונקציה ReadFile:

```
BOOL ReadFile(
  HANDLE       hFile,
  LPVOID       lpBuffer,
  DWORD        nNumberOfBytesToRead,
  LPDWORD      lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

כעת הדברים מתחברים: כל האובייקטים נמצאים תחת מערכת ההפעלה ומנוהלים על ידי ה- Object Manager . אם Process רוצה גישה לאובייקט הוא צריך לבקש גישה, ואם היא ניתנת לו הוא מקבל Handle.

ישנן ארבע דרכים שבהן Process יכול להשיג Handle לאובייקט:

- יצירה. לדוגמה CreateFile. יווצר אובייקט חדש, מן הסתם למי שיצר אותו יהיה Handle אליו
- פתיחה. לדוגמה OpenFile. בפתיחה של אובייקט קיים ה-Object Manager יבדוק את רמת ההרשאות ויחזיר Handle אם ההרשאות תקינות
- ירושה. כפי כנראה בהמשך, Process יכול ליצור בנים. בתהליך היצירה הוא יכול לבחור באפשרות InheritHandles ואז הבנים מקבלים את ה-Handleים ששייכים לאב
- שכפול. אפשר לשכפל Handle באמצעות הפונקציה DuplicateHandle.

3.8.1 מבנה נתונים של אובייקט

אובייקט של Windows מתואר על ידי מיקרוסופט כ-"Opaque data structure". כלומר, פרטי המימוש המדוייקים שלו ידועים רק למיקרוסופט. הסיבה לכך היא ש-Thread לא אמור לגשת לאובייקט בעצמו, אלא רק דרך Handle שניתן לו. הדבר נמנע משתי סיבות, כפי שמיקרוסופט כותבת כאן:

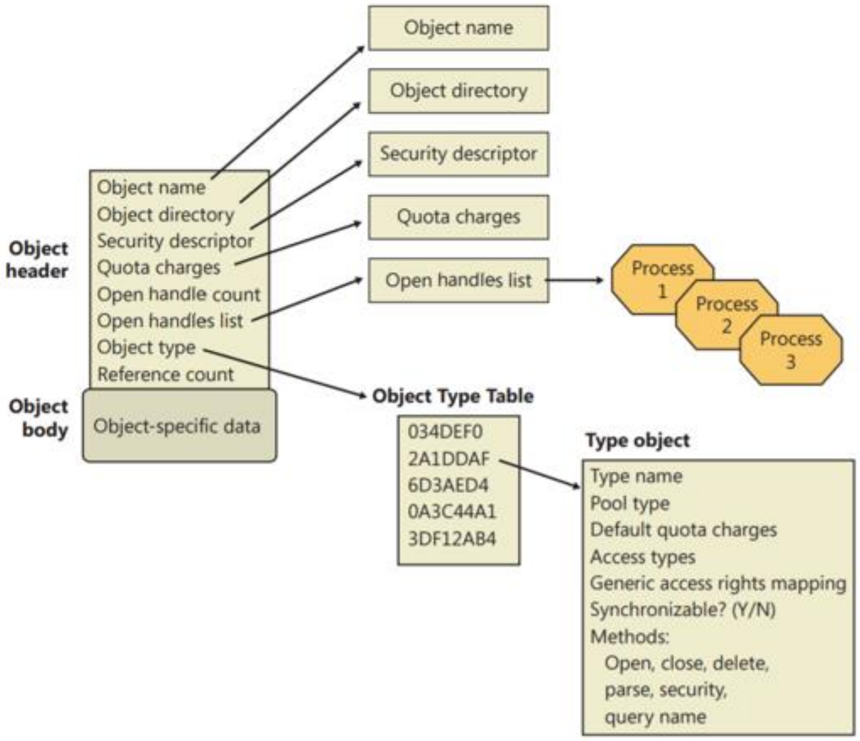
The system uses objects and handles to regulate access to system resources for two main reasons. First, the use of objects ensures that Microsoft can update system functionality, as long as the original object interface is maintained. When subsequent versions of the system are released, you can use the updated object with little or no additional work.

Secondly, the use of objects enables you to take advantage of Windows security. Each object has its own access-control list (ACL) that specifies the actions a process can perform

on the object. The system examines an object's ACL each time an application creates a handle to the object.

כלומר גם מתוך רצון של מיקרוסופט לעדכן בחופשיות את מבני הנתונים שלה, וגם מתוך צורך לשמור על אבטחה.

להלן מבנה של אובייקט, כפי שמופיע ב-Windows Internals:



רואים שאובייקט הוא מבנה נתונים שכולל שני חלקים. החלק הראשון, ה-Object Header, זהה לכל סוגי האובייקטים. כלומר גם לאובייקט מסוג קובץ וגם לאובייקט מסוג Process יהיו את אותם שדות ב-Header. החלק השני הוא ה-Object Body, והוא מכיל מידע ספציפי לגבי האובייקט. כאן צפוי להיות כמובן שוני בין אובייקטים מסוגים שונים.

ה-Header מכיל מצביעים שונים. סביר שנתקלתם במושג ה-Header בספר רשתות מחשבים. כמו בפרוטוקולים שיש להם שדה שאומר מה הפרוטוקול בשכבה הבאה, כך גם ה-Header של האובייקט מצביע על מבני נתונים שמאפשרים ל-Object Manager להבין מה יש בכל אובייקט ולנהל את האובייקטים בקלות. נעבור על ה-Object Header ונסקור כמה מצביעים מעניינים במיוחד:

1. Object name - כשמו כן הוא. אפשר לייצר אובייקטים שיש להם שם, ואז אפשר לפנות אליהם ולפתוח אותם באמצעות השם.
2. Object type - אובייקטים מסוג זהה חולקים מאפיינים זהים. לדוגמה קובץ, שני קבצים יכולים להיות שונים זה מזה אך לשניהם יש אותן מתודות של פתיחה, קריאה, כתבה וכו. באמצעות השדה הזה מערכת ההפעלה יכולה לגשת אל שדות שמשותפים לאובייטקים מאותו סוג.

3. Open Handle List - רשימת כל ה-Handleים הפתוחים לאובייקט.
4. Open Handle Count - כל Handle שנפתח לאובייקט מעלה את המספר ב-1, וכל סגירה של Handle מפחיתה 1.
5. Reference Count - קוד של מערכת ההפעלה גם יכול להשתמש באובייקטים, אך הוא לא זקוק ל-Handleים, אלא מקבל כתובת בזיכרון באופן ישיר. כל מצביע לכתובת מעלה את הספירה ב-1.

יכול להיות מצב שבו הן ה-Handle Count והן ה-Reference Count מתאפסים. המשמעות היא שאין אף שימוש באובייקט. במקרה כזה, ה-Object Manager ימחק את האובייקט וישחרר את הזיכרון.

תרגיל 3.1 תרגיל מסכם מודרך - Syscall

בתרגיל מודרך זה נמצא בעצמנו את שרשרת הקריאות שמתבצעת כאשר Process קורא ל-ReadFile, כל הדרך מ-Userland אל תוך ה-Kernel. על הדרך, נתקין תוכנות שנזדקק להן בהמשך ונתנסה בהפעלה שלהן.

התקנת תוכנות

1. התקינו את Visual Studio. בחרו בגרסה החינמית המיועדת לתלמידים, הנקראת Visual Studio Community

https://visualstudio.microsoft.com/downloads/

2. הורידו את כלי Sysinternals מהקישור הבא, צרו ספריה בשם Sysinternals והעתיקו לתוכה את כל הכלים

https://download.sysinternals.com/files/SysinternalsSuite.zip

יצירת תוכנית

כדי להציג את שרשרת הקריאות, אנחנו צריכים תוכנית שקוראת ל-ReadFile.

אם אין לכם נסיון ב-WinAPI, קיראו את מדריך ה-WinAPI שנמצא בנספח לפרק זה. לאחר שסיימתם, עבדו לפי סדר הפעולות הבא:

1. ראשית קיראו לפונקציה CreateFile והשתמשו בה על מנת לפתוח קובץ טקסט שיצרתם מראש. ההסבר על שימוש ספציפי ב-CreateFile לא נמצא במדריך ה-WinAPI, אך ההדרכה תאפשר לכם להסתדר בכוחות עצמכם.
2. לאחר מכן קיראו לפונקציה ReadFile.
3. לבסוף קיראו ל- CloseHandle וזהו. התיעוד של כל הפונקציות הללו נמצא ב-MSDN. חפשו בגוגל את שם הפונקציה ותגיעו ישר לתיעוד.

אל תשכחו ליצור קובץ txt שהתכנית תקרא ותדפיס למסך. הזינו לתוך הקובץ טקסט קצר כלשהו.

לאחר שסיימתם, הגיע הזמן לחקור את הקוד.

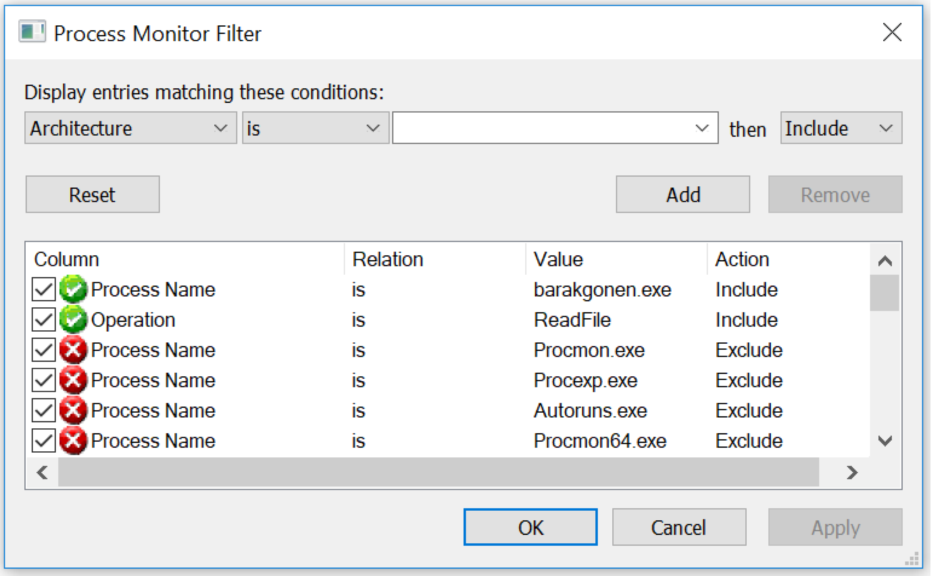
הרצת Procmon

זהו כלי חשוב שמאפשר לעקוב אחרי פעולות שמבצעים Processים. יש בו יכולות חיפוש וסינון והוא משמש גם לאיתור נוזקות שמבצעות דברים זדוניים על המחשב שלנו.

הריצו את Procmon מתוך ספריית Sysinternals. הריצו אותו כ-Administrator (קליק ימני על שם הקובץ ואז בחרו באפשרות הרצה כמנהל). מהר מאד יופיעו לפניכם אירועים רבים. מהו אירוע? כל קריאה של Process לפונקציה של ה-Kernel היא אירוע, שנמצא בעמודה בשם Operation.

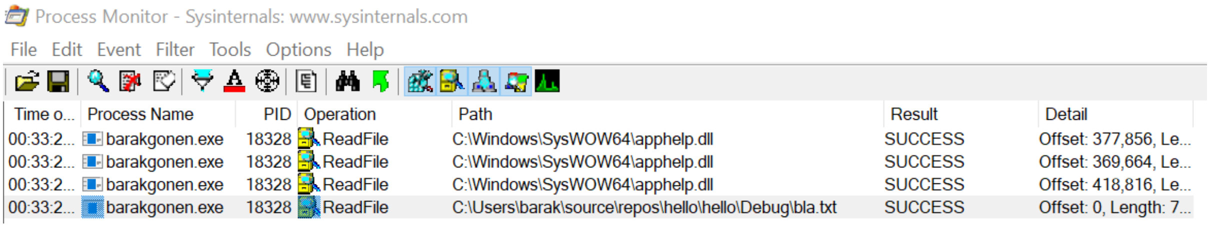
באפשרותנו למקד את הארועים שאנחנו רוצים לצפות בהם באמצעות הגדרת פילטר. לחצו על הטאב של הפילטר ובחרו באפשרות עריכת הפילטר (או לחצו Ctrl+L). לפילטר שאנחנו רוצים להגדיר יש שני מאפיינים:

1. Process name is ולאחריו יבוא שם קובץ ההרצה שיצרתם, לדוגמה hello.exe
2. Operation is read file – אנחנו רוצים לפלטר רק את הקריאה מהקובץ



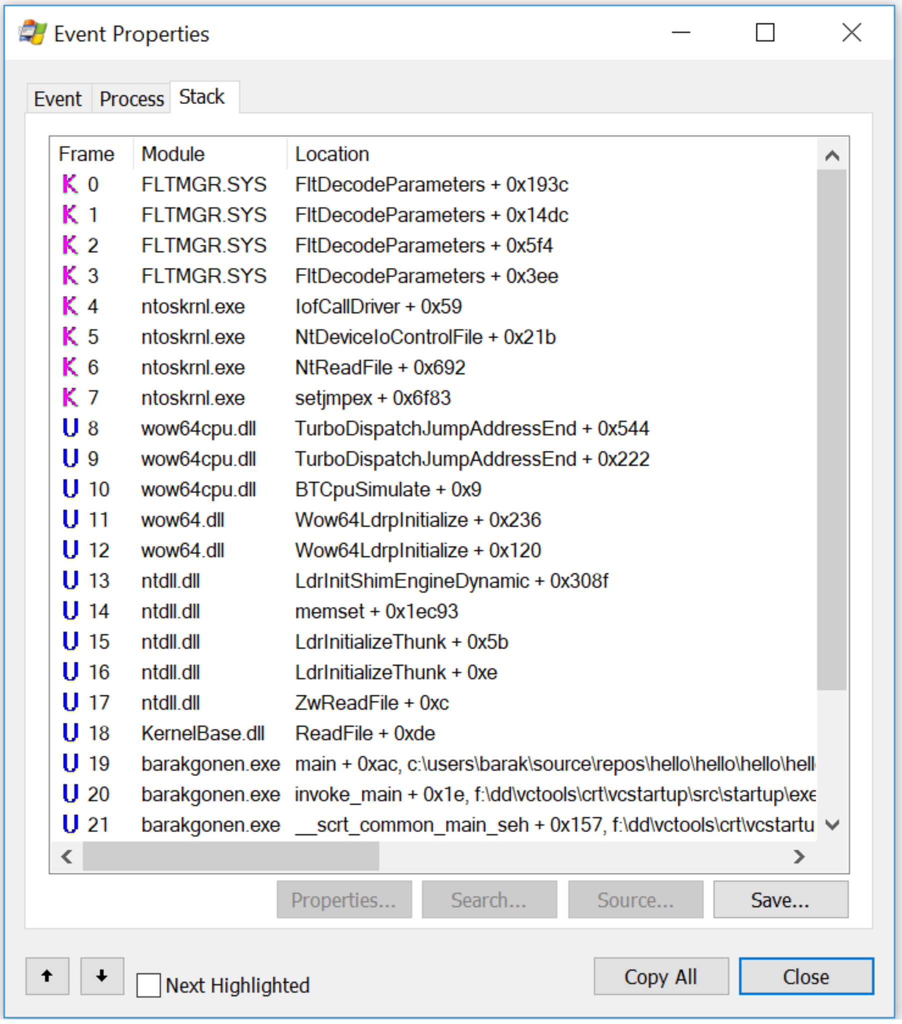
הרצת התכנית וצפיה בתוצאות

כעת כשהפילטר מסודר נריץ את התוכנית (לחיצה כפולה על שם הקובץ שיצרנו). התוכנית מסיימת את ריצתה מהר למדי, אך Procmon קלט את הדברים שהיא ביצעה ומציג לנו מספר ארועים:



מעניין אותנו הארוע האחרון, שבו ה-Path תואם את מיקום קובץ הטקסט שלנו. נעמוד עליו עם העכבר, נבצע קליק ימני ונבחר Stack (מחסנית).

מדוע המחסנית מעניינת אותנו? כזכור מלימודי האסמבלי, המעבד צריך לדעת לאיזו כתובת לחזור בסיום ריצת כל פונקציה. כתובות החזרה נשמרות על המחסנית לפי הסדר. כלומר אם נעבור על המחסנית נוכל לאתר את כל כתובות החזרה. תוכנת Procmon יודעת לבצע מיפוי בין כתובות החזרה לשם הפונקציה שקראה לה, וכך אנחנו יכולים לקבל את שרשרת הקריאות עד ל-Kernel. נתבונן במחסנית של התוכנית שלנו, כפי שהוא נשמר ברגע שה-Kernel הגיע להריץ את ReadFile:



בשורה 20 ניתן לראות שהתוכנית שלנו מריצה את main.

בשורה 19 main קורא ל-ReadFile.

בשורה 18 אנחנו בתוך ReadFile של KernelBase.dll, הוא קורא לפונקציה ZwReadFile של ntdll.

וכך הלאה עד שבשורה 7 מגיעים אל ntoskrnl.exe. שימו לב שליד שורה זו כתוב "K" בזמן שכל השורות מתחתיה הן "U", כלומר עברנו מ-Userland ל-Kernel.

למעשה התמונה של המחסנית שמציג לנו Procmon היא "שקר" מסויים, הקריאות של הפונקציות ב-Kernel וב-Userland לא נמצאות במחסנית אחת. יש יותר ממחסנית אחת שמחזיקה את שרשרת הקריאות. רמזנו על זה כבר, אתם מוזמנים להזכר, נסקור את הנושא בפרק הבא.

בשורה 6 הקרנל קורא ל-NtReadFile. היזכרו, שבתעוד של מיקרוסופט שמובא בתחילת התרגיל, הבלוק התחתון ביותר הוא קריאה ל-NtWriteFile. עד כה מה שראינו מאד תואם לתיעוד.

בשורות 5 ומעלה יש קריאות נוספות של ה-Kernel, שכפי שניתן להסיק מהשמות שלהן הינן קריאות להתקן החומרה, במקרה זה הדיסק הקשיח.

בצעו את הפעולות הללו עם הקוד שקימפלתם וחיזרו על אותם הצעדים, עד שתמצאו את שרשרת הקריאות ל-ReadFile. בהצלחה!

3.9 נספח - מבוא ל-WinAPI

3.9.1 רקע

מערכת ההפעלה Windows כוללת ממשק, שמאפשר למתכנתים לקרוא לפונקציות שונות של מערכת ההפעלה. מדוע להשתמש בממשק זה? יש לכך שתי סיבות.

הסיבה הראשונה, היא חיסכון בעבודה. לדוגמה, במקום לכתוב פונקציה שפותחת קובץ לקריאה, או בודקת ערך ברגיסטרי, אפשר להשתמש בפונקציות המוכנות CreateFileA ו- RegQueryValueEx.

הסיבה השניה היא שאין (כמעט) ברירה אחרת. כפי שלמדנו, עקב מעגלי האבטחה שלמערכת ההפעלה, אפליקציה שרצה ב-userland לא יכולה לגשת באופן חופשי לחומרה, כגון הדיסק הקשיח או הזיכרון. הכל חייב לעבור דרך הקרנל של מערכת ההפעלה. ה-API, קיצור של Application Programming Interface, הוא דרך נפוצה לאפשר למתכנתים גישה לתוכנה שגורם אחר פיתח, וספציפית WinAPI היא הדרך שבה מתכנתי מערכת ההפעלה Windows מאפשרים לתוכנה חיצונית לפנות לקרנל ולבקש ממנו שירותים שונים.

הממשק של WinAPI מתועד כך שכל מתכנת יכול להכנס ולבדוק אילו פונקציות קיימות וכיצד יש לקרוא להן. כאן חשוב להזכיר שהפונקציות המוגדרות ב-WinAPI אינן פונות ישירות אל הקרנל אלא קוראות קודם כל לפונקציות מתוך ntdll – זהו קוד נוסף של מיקרוסופט, שגם הוא רץ ב-Userland ושם נמצאות הפונקציות שקוראות ישירות אל ה-Kernel. ה-ntdll נקרא גם Native API והוא כולל פונקציות של מיקרוסופט שרובן מוכרות רק למיקרוסופט. תיאורטית אפשר לקרוא מתוך הקוד לפונקציות של ntdll, לדוגמה במקום לקרוא ל-WriteFile מתוך WinAPI אפשר לקרוא ל- NtWriteFile מתוך ה-NativeAPI, אך הקושי הוא שבמרבית המקרים הפונקציות הללו לא יהיו מתועדות. מיקרוסופט גם אינה מתחייבת לשמור על הפונקציות והממשקים של ה-Native API שלה, רק של WinAPI.

כדי להפעיל את הפונקציות של WinAPI יש צורך לבצע include לקובץ windows.h. כדי לחסוך זמן קימפול מומלץ לבצע את ה-include בתוך הקובץ pch.h וכך להנות מהיתרון של pre-compilation. המשמעות היא שבפעם הראשונה מבוצע תהליך קימפול ארוך יותר אך בפעמים הבאות הקימפול מתקצר. זו פרקטיקה טובה עבור קבצי header גדולים שאינם משתנים לעיתים קרובות, ובפרט windows.h.

3.9.2 טיפוסי משתנים בסיסיים

כאשר תכנתנו ב-C הכרנו אוסף של טיפוסי משתנים. char, float, int ועוד. ב-windows.h יש שמות אחרים של טיפוסי משתנים. לדוגמה BYTE הוא משתנה בגודל 8 ביט, PCHAR הוא מצביע ל-char. האם אנחנו חייבים להשתמש בטיפוסי המשתנים של windows.h? לא חייבים, אפשר לבחור אם להשתמש בהגדרות של windows.h או בהגדרות המוכרות משפת C. למעשה, הדברים זהים לחלוטין, מכיוון שכל טיפוסי המשתנים של windows.h הם פשוט קריאה לטיפוסים ב-C בשמות אחרים, באמצעות ההנחיה typedef. כך לדוגמה:

```
typedef CHAR     *PCHAR;
```

בעוד CHAR עצמו מוגדר כך:

```
typedef char     CHAR;
```

כך שלמעשה PCHAR הוא בעצם-

```
typedef char     *PCHAR;
```

באופן דומה מוגדרים כל יתר הטיפוסים של windows.h.

למרות שלא חייבים להשתמש בטיפוסי משתנים אלו, מומלץ לעבוד איתם מכיוון שכל הפונקציות של WinAPI מתועדות עם טיפוסי משתנים אלו. בלי לכל הפחות להכיר אותם, יהיה מאד קשה לקמפל את הקוד ללא שגיאות.

טיפוסי משתנים בסיסיים מרכזיים ב-windows.h:

```
typedef int                BOOL;
typedef unsigned char      BYTE;
typedef char               CHAR;
typedef unsigned long      DWORD;
typedef float              FLOAT;
typedef int                INT;
```

המידע המלא על טיפוסי משתנים אלו ועל כל יתר טיפוסי המשתנים של WinAPI נמצא בלינק הבא, באתר MSDN של מיקרוסופט:

https://docs.microsoft.com/en-us/windows/desktop/winprog/windows-data-types

3.9.3 מצביעים

ל-WinAPI יש טיפוסי משתנים גם עבור מצביעים. לכל טיפוסי המשתנים שלעיל ניתן להוסיף P על מנת לציין pointer. לדוגמה PCHAR PWORD PINT וכו'.

בדרך כלל נראה את האות L כתובה לפני מצביעים. לדוגמה LPINT. מהי ה-L והאם היא הכרחית? עבור מעבדים ישנים של 16 ביט, הוגדרו מצביעים מסוג near ו-far, כאשר מצביע near שומר כתובת בגודל 16 ביט ואילו מצביע far שומר כתובת בגודל 32 ביט. עבור מעבדים של 32 ביט ומעלה אין הבדל עם או בלי L. בדרך כלל נראה את ה-L מקדימה את טיפוס המשתנה, מטעמי שמירה על תאימות תוכנה, אבל עבור קוד שאינו מיועד למעבדים עתיקים אין לכך משמעות.

סוג מיוחד של מצביע הוא PVOID או LPVOID, כלומר מצביע על אובייקט שהוא VOID, כלום. מה ההגיון בלהצביע על אובייקט שהוא כלום? זהו בעצם מצביע שהוא כמו ג'וקר בחבילת קלפים- כמו שג'וקר יכול להיות כל קלף, לפי רצון השחקן, כך מצביע ל-VOID יכול להצביע על כל אובייקט שהמתכנת קובע עבורו. דבר זה מאפשר גמישות תכנותית, אך מאידך המתכנת צריך לזכור על איזה סוג של אובייקט המצביע מצביע בכל שלב. לדוגמה, אם המתכנת חושב שהמצביע מצביע על מחרוזת, אך למעשה הוא מצביע על INT, לא תהיה שגיאת קומפילציה אך עלולה להיגרם שגיאה בזמן ריצה.

קטע הקוד הבא ממחיש כיצד מצביע מסוג PVOID מקבל פעם אחת ערך של מצביע מסוג PINT ופעם אחרת מסוג PFLOAT, דבר שאי אפשר לעשות עם מצביע אחר.

```
PVOID joker = NULL;
```

```
INT num = 2;
PINT pnum = &num;
joker = pnum;
printf("Joker: %d\n", *(PINT)joker);

FLOAT pi = 3.1415;
PFLOAT pfl = &pi;
joker = pfl;
printf("Joker: %f\n", *(PFLOAT)joker);
```

3.9.4 מחרוזות

ה-API תומך בשני סוגי מחרוזות. הסוג הראשון הוא תווי ASCII, בהם כל תו נשמר ב-char, 8 ביט. כיוון שתווי ASCII מוגבלים ל-256 ערכים ואינם מסוגלים לייצג את התווים שקיימים בכל השפות, פותחו תווי UNICODE, בהם כל תו נשמר ב-word, 16 ביט. כעת יש ברשותנו כ-65536 צירופי תווים אפשריים, וזה כבר מספיק טוב.

לדוגמה, קוד ה-ASCII של התו 'a' הוא 61 (בהקסדצימלי). ניתן לשמור אותו ב-8 ביט. ב-UNICODE הצירוף מורחב ל-16 ביט ולכן הקוד הוא 0061. לעומת זאת האות העברית 'א', לה אין קוד ASCII, יש ערך UNICODE של 0D5.

איך המחשב יודע לפרש את 0D5 בתור האות א' ולא בתור שני קודי ASCII, קוד 5 וקוד 0D? צריך להגדיר לו עם איזה סוג מחרוזת אנחנו עובדים. מחרוזות שהן מסוג UNICODE יכילו את האות W – קיצור של wide, ואילו מחרוזות ASCII לא יכילו W. דוגמאות לסוגי מחרוזות ומצביעים אליהן:

```
typedef CONST CHAR     *LPCSTR;
typedef CONST WCHAR    *LPCWSTR;
typedef WCHAR          *LPWSTR;
```

ואותן המחרוזות ללא תחילית ה-L (שכאמור במעבדי 32 ביט והלאה- המשמעות זהה):

```
typedef CONST CHAR     *PCSTR;
typedef CONST WCHAR    *PCWSTR;
typedef WCHAR          *PWSTR;
```

בחלק מהמקרים של מצביעים למחרוזות, נמצא את התו C, שמציין Const. כלומר אי אפשר לשנות את המחרוזת.

הדרך להגדיר מחרוזת מסוג W היא לכתוב לפני המחרוזת את התו L. לדוגמה:

```
LPCWSTR my_str = L"Hello";
```

במקרה זה יש צורך להגדיר C מכיוון שזו מחרוזת קבועה. כלומר my_str מצביע לאזור בזכרון שם נשמר Hello כקבוע. אם, לעומת זאת, אנחנו רוצים להגדיר מצביע למחרוזת שיכולה להשתנות:

```
WCHAR my_str [6] = L"Hello";
LPWSTR pstr = my_str;
```

שימו לב שיש צורך להגדיר מחרוזת בגודל 6 איברים, למרות שהמילה "Hello" היא בת 5 תווים, מכיוון שבסוף המחרוזת צריך לשמור מקום לתו שמציין סיום מחרוזת. תו זה מיוצג על ידי קוד הקסדצימלי 00, כך שבכל סוף מחרוזת נמצא קוד 00. תו סיום המחרוזת מאפשר לבצע פעולות שונות על מחרוזות, כגון לבדוק אורך של מחרוזות ולהדפיס מחרוזת. ללא תו סיום מחרוזת, הפונקציה שבודקת את אורך המחרוזת לא תדע היכן להפסיק את הספירה ופונקציית ההדפסה לא תדע היכן לעצור את ההדפסה.

כפי שראינו, ישנם תווים שה-UNICODE שלהם כולל את הקוד 00. לדוגמה התו 'a' הוא 0x0061. במילים אחרות, אם נשלח מחרוזת UNICODE לפונקציה שבודקת אורך מחרוזת או לפונקציה שמדפיסה מחרוזת, אנחנו עלולים לקבל תוצאה לא תקינה. לכן יש סט מיוחד של פונקציות, שמטפלות במחרוזות מסוג UNICODE.

לדוגמה, במקום printf שמתאימה לקוד ASCII, הפונקציה wprintf מקבלת מחרוזת UNICODE. לדוגמה, במקום strlen שמתאימה ל-ASCII, הפונקציה wcslen מוצאת אורך של מחרוזת UNICODE.

ומה אם אנחנו רוצים לכתוב קוד שיתאים הן ל-UNICODE והן ל-ASCII?

במקרה זה במקום האות W נכתוב את האות T. המנגנון עובד כך: כאשר מוגדרת מחרוזת עם T, הקומפיילר בודק אם יש הגדרה של define UNICODE. אם לא- נעשה שימוש בהגדרות של ASCII, ולהיפך. כך נראית הגדרה של טיפוס מחרוזת PTSTR:

```
#ifdef UNICODE
     typedef LPWSTR PTSTR;
#else typedef LPSTR PTSTR;
#endif
```

כפי שרואים, אם מוגדר UNICODE אז הטיפוס הינו בעצם LPWSTR, אחרת- LPSTR.

3.9.5 פונקציות עם סיומת A לעומת סיומת W

לאחר שהבנו את ההבדלים בין קוד ASCII לבין UNICODE, אפשר להבין את ההבדל בין גרסאות שונות שיש לפעמים לאותה פונקציה. עבור פונקציות שמקבלות מחרוזת בתור אחד הפרמטרים, יש חשיבות לדעת בדיוק את הפורמט של המחרוזת, כדי למנוע טעויות. פונקציות שמסתיימות ב-A יצפו לקבל מחרוזת ASCII, פונקציות שמסתיימות ב-W יצפו לקבל מחרוזת UNICODE.

לדוגמה CreateFile, גרסה אחת היא CreateFileA ואילו גרסה אחרת היא CreateFielW. חפשו את ההבדל בפרמטרים שלהן:

```
HANDLE CreateFileA(
  LPCSTR                lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

```
HANDLE CreateFileW(
  LPCWSTR               lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

כפי שאתם רואים, כל ההבדל הוא תו אחד: בפונקציה CreateFileA הפרמטר הראשון הוא מסוג LPCSTR, ואילו בפונקציה CreateFileW הפרמטר הראשון הוא מסוג LPC**W**STR. הפרמטר הזה הוא מצביע על שם הקובץ אותו מבקשים ליצור.

3.9.6 קריאה לפונקציות של WinAPI

התיעוד של כל הפונקציות נמצא באתר MSDN, קיצור של MicroSoft Developers Network. אם אנחנו יודעים מה שם הפונקציה שאנחנו מחפשים, ניתן לגגל msdn ושם הפונקציה, לדוגמה "msdn readfile", וצפוי שהתוצאה העליונה תהיה הדף ב-MSDN. נבצע קריאה מודרכת בעמוד של MSDN ReadFile. מצד שמאל אפשר לראות את כל שמות הפונקציות מסודרות לפי הסדר המילוני שלהן.

LockFileEx function

QueryDosDeviceW function

ReadFile function

ReadFileEx function

ReadFileScatter function

נראה של-ReadFile יש וריאציה שנקראת ReadFileEx. לא נעסוק כרגע בוריאציה הזו, רק חשוב לשים לב שכדאי לעבור על וריאציות של פונקציות שאנחנו מחפשים, לעיתים נמצא פונקציה יותר מתאימה ממה שחיפשנו.

החלק המרכזי של ReadFile כולל הסבר קצר על מה שהפונקציה מבצעת ומיד לאחריו את הפרמטרים שהיא מקבלת ומחזירה:

Syntax

```
BOOL ReadFile(
  HANDLE       hFile,
  LPVOID       lpBuffer,
  DWORD        nNumberOfBytesToRead,
  LPDWORD      lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

לאחר מכן יש הסבר על כל פרמטר. לדוגמה:

`nNumberOfBytesToRead`

The maximum number of bytes to be read.

לפונקציה ReadFile יש כמה פרמטרים שברור שהיא חייבת לקבל. Handle לקובץ, מצביע לזיכרון שאליו אנחנו רוצים שהמידע מהקובץ יועתק, וכמובן כמה בתים להעתיק. אלו שלושת הפרמטרים הראשונים. לגבי שני הפרמטרים הנוספים יש לנו שתי אפשרויות. הראשונה היא להזין שם ערכים מתאימים, אחרי שבדקנו מה הפרמטרים הללו אומרים, או לנצל את האפשרות שכתובה בתיעוד ולהזין פשוט NULL.

לאחר פירוט הפרמטרים, מוסבר ערך החזרה של הפונקציה. כפי שאפשר לראות מה-Syntax שלה, ערך החזרה הוא BOOL, כלומר יש רק שתי אפשרויות- 0 או 1. אנחנו עדיין צריכים הבהרה מה מציין 0 ומה מציין 1.

Return Value

If the function succeeds, the return value is nonzero (**TRUE**).

If the function fails, or is completing asynchronously, the return value is zero (**FALSE**). To get extended error information, call the GetLastError function.

3.9.7 שימוש יעיל בקודי שגיאה

אם יש בעיה והפונקציה נכשלה, בשלב ראשון כדאי לבדוק את קוד השגיאה. קוד השגיאה יכול להסביר לנו מה היתה הבעיה הספציפית שגרמה לפונקציה לחזור עם FALSE. כדי לבצע זאת, כל מה שאנחנו צריכים לעשות זה לקרוא בקוד שלנו ל-GetLastError. לדוגמה:

```
HANDLE      hfile;
INT         num_bytes = NUM_BYTES;
CHAR        data[NUM_BYTES];
LPVOID      pbuffer = data;
BOOL result = ReadFile(hfile, pbuffer, num_bytes, NULL, NULL);
if (result == FALSE)
{
      printf("%d", GetLastError());
      return 0;
}
```

כעת יודפס ל-Console מספר כלשהו. כדי להבין מה המשמעות שלו ניגש לדף קודי השגיאה ב-MSDN:

https://docs.microsoft.com/he-il/windows/desktop/Debug/system-error-codes

3.10 סיכום

זה היה רקע בסיסי על תכנות WinAPI, שמטרתו לתת "נחיתה רכה" למי שעושים את הצעדים הראשונים בכתיבת תוכנות שקוראות לפונקציות של מערכת ההפעלה. התחלנו מהסבר על טיפוסים שונים שמוגדרים ב-WinAPI וסיימנו בהסבר על איך לקרוא תיעוד ב-MSDN. אלו הכלים הבסיסיים שצריך בשביל להתחיל לעבוד.

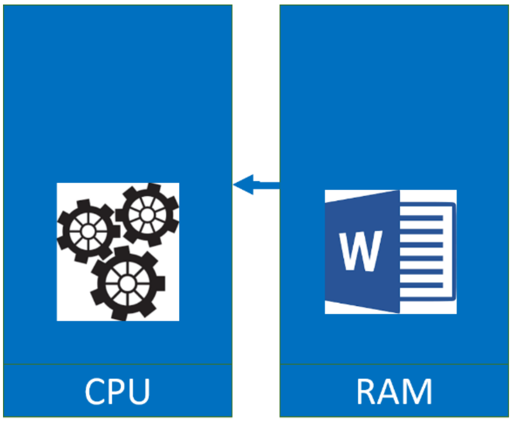
פרק 4 – תהליכים ותהליכונים - Processes and Threads

בפרקים הקודמים ראינו כי תוכנות שרצות במחשב שלנו נמצאות תחת Processים. Processים הם חלק מרכזי בהבנת אופן הפעולה של מערכות הפעלה. בחלק זה נבין לעומק מהם Processים, מהם Threadים ונלמד לכתוב תוכנית ב-C שיוצרת Processים וThreadים.

נתחיל ממושגי היסוד – Processים וThreadים.

4.1 תהליכונים - Threads

המושג Thread, או בעברית "תהליכון", מייצג רצף של פקודות שמבצעות משימה אחת. לדוגמה, ספר זה נכתב באמצעות תוכנת Word. אוסף כל הפקודות ש-Word מריץ הוא Thread של התוכנה Word. פקודות המכונה המרכיבות את התוכנה Word שמורות בזיכרון ה-RAM ומשם הן מועברות למעבד, שמריץ אותן.

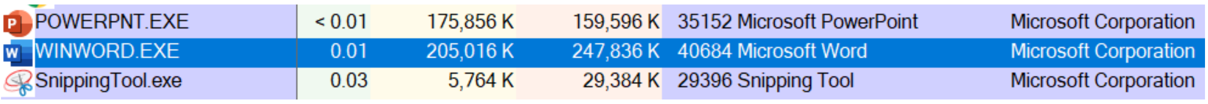


זהו המודל הפשוט ביותר של עבודת המחשב. מעבד שמריץ פקודות מהזיכרון. לו היינו מוסיפים לאיור יחידת קלט / פלט, היינו מקבלים את מודל המחשב של פון נוימן. אולם, למודל עבודה זה יש בעיות והוא אינו יעיל.

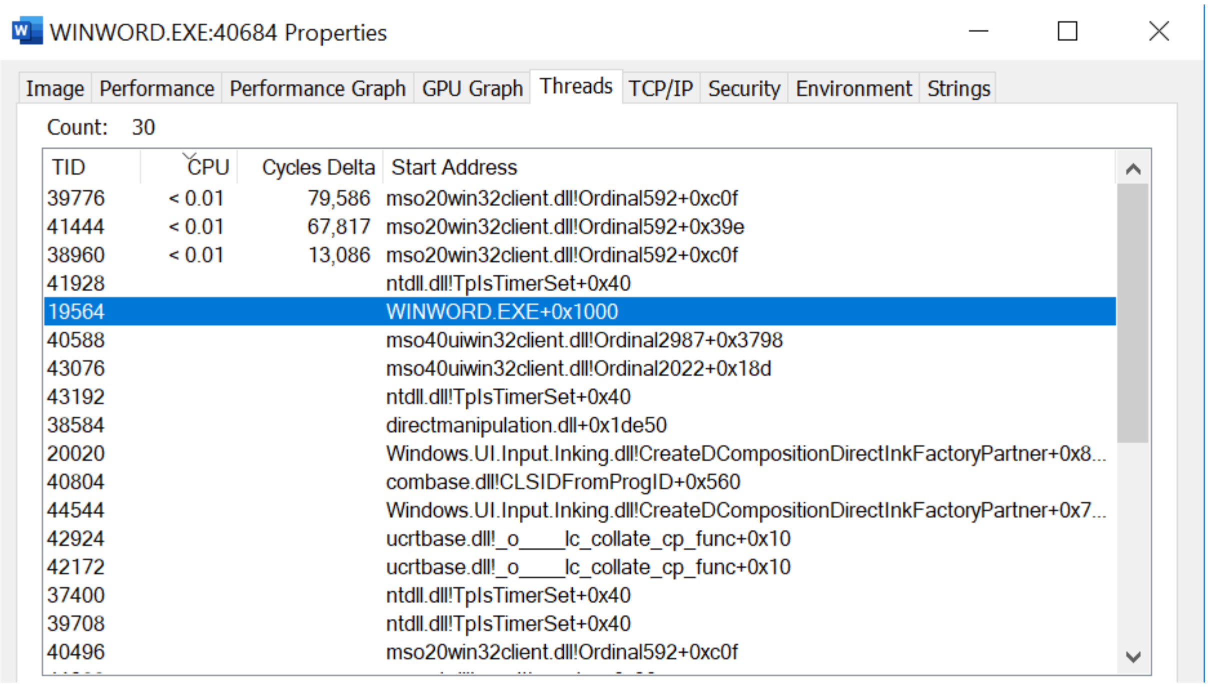
4.2 ריבוי תהליכונים - Multi Threads

בפרקים הקודמים הורדנו את חבילת הכלים Sysinternals Suite. לפני שניגש להסבר תאורטי על Mutli Threads, נעשה ניסוי קטן. מתוך חבילת Sysinternals, הפעילו את התוכנה "Process Explorer". יחד איתה הפעילו את תוכנת Word. תוכנת Process Explorer מציגה לכם סטטוס עדכני של כל הProcessים שרצים על המחשב. נחפש את הProcess של Word. כדי לעשות זאת, נשתמש באייקון של המטרה שנמצא במסך ה-Process Explorer. המטרה היא האייקון הימני ביותר בשורה:

לחצו עליה לחיצה ארוכה עם העכבר ואל תשחררו. האייקון של העכבר הופך למטרה. גררו את המטרה אל מסמך ה-Word שפתוח אצלכם ושחררו את העכבר. ברגע שתעשו זאת, Process Explorer מיד יציג לכם את השורה המבוקשת, ה-Process של Word. כפי שאתם שמים לב, שם ה-Process הוא WINWORD.EXE.

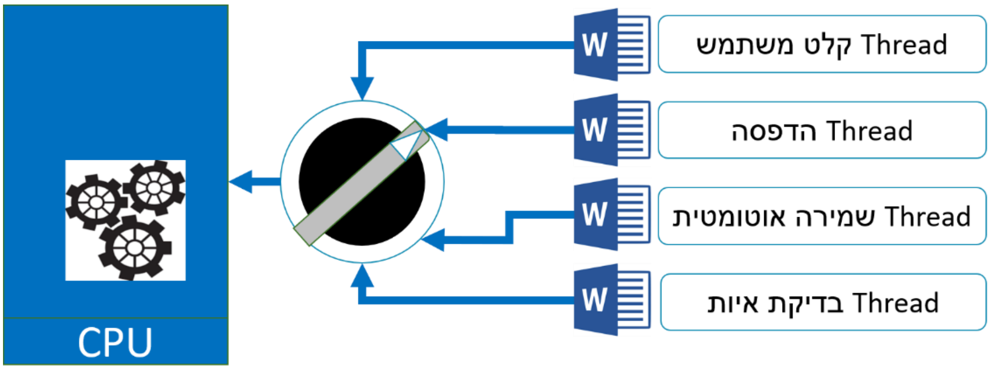


הקליקו על WINWORD הקלקה כפולה, וכעת עברו לטאב שנקרא Threads.



הפתעה. מסתבר שה-Word הצנוע שלנו משתמש בלא פחות מאשר 30 Threadם. מה הסיבה לכך? ובכן, לכל תוכנה, וגם ל-Word, יש "רגעים מתים". רגעים בהם התוכנה לא מבצעת דבר אלא ממתינה למשהו שצריך להתרחש. לדוגמה, אחת לכמה זמן תוכנת Word מבצעת שמירה אוטומטית של המסמך בדיסק הקשיח, ולכן עליה להמתין לסיום השמירה בדיסק. לדוגמה, התוכנה ממתינה להקלדות המשתמש על המקלדת. יחסית לקצב העבודה של המעבד, למשתמש לוקח זמן עצום בין הקלדה להקלדה.

אילו Word היה מריץ רק Thread אחד, אי אפשר היה לעשות הרבה כדי לשפר את היעילות. אולם, למה שלא נריץ מספר Threadם של Word? יהיו לנו Thread שקולט את הקלדות המשתמש. Thread נוסף יבצע שמירה אוטומטית אחת לזמן מה. Thread שלישי יבצע בדיקת שגיאות הקלדה ו-Thread רביעי יבצע משימות של הדפסה ברקע, כך שנוכל להמשיך להקליד בעוד המסמך שלנו מודפס. הרבה יותר יעיל. כמובן שהחלוקה הזו ל-Threadים היא רק דוגמה עבור Word, יש תוכנות רבות שיכולות להרוויח מחלוקת משימות ל-Threadים.



איך נאפשר עבודה של מספר Threadים במקביל? הרי כל Thread מעביר למעבד פקודות מכונה שונות. נשים ביניהם לבין המעבד רכיב תזמון, נקרא לו "Scheduler". רכיב התזמון יעניק את משאבי העיבוד כל פעם ל-Thread אחר. בקרוב נראה שה-Thread האחר לא חייב להיות אפילו שייך לאותה התוכנית. כך, כאשר ה-Thread שמקבל קלט מהמשתמש יהיה חסר עבודה, יוענקו משאבי המעבד ל-Thread אחר שזקוק להם. באופן זה ניתן יהיה לבצע את כלל המשימות בצורה יעילה ומהירה יותר. הידד!

אלא שכעת נוצרה לנו בעיה חדשה: לכל הThreadים שלנו יש דברים משותפים. מה משותף לכולם?

הם ניגשים לאותו קובץ, לכן אם Thread מסויים לא חולק את הקובץ יתר הThreadים לא יכולים לבצע את העבודה שלהם.

הם ניגשים לאותו חלון של המשתמש, כך שאם המשתמש מקליד על תו מסוים או מבצע פעולה כלשהי, כל הThreadים צריכים לדעת כך.

בנוסף המסמך בגרסתה העדכנית צריך להיות בזיכרון של כל הThreadים, כי אחרת הם לא יעבדו על אותה גרסה של המסמך.

אנחנו נציג כעת מושג חדש- Process. ההסבר עליו יגיע בקרוב, אך כרגע לצורך הדיון חישבו על Process בתור תוכנה שרצה במחשב שלכם. לדוגמה כשמקליקים על תוכנת Word אז נפתח Process של Word.

חישבו: אילו משאבים צריכים להיות משותפים לכלל הThreadים של אותו Process?

Threadים של אותו Process חולקים ביניהם:

כתובות בזיכרון: כל הThreadים רואים את אותם ערכים באותן כתובות בזיכרון. לדוגמה, אם Thread א' רואה בכתובת 0x402000 את הערך 5, אז Thread ב' יראה את הערך 5 באותה הכתובת.

זיכרון Heap: לכל הThreadים יש זיכרון Heap יחיד. כלומר, כאשר Thread כלשהו מבצע הקצאת זיכרון malloc, כל הThreadים מקבלים גישה לזיכרון הזה.

קוד הרצה: כל הThreadים רצים מאותו אזור זיכרון. כיוון שמדובר באותה תוכנה, אין צורך להעתיק את אותו קוד שוב ושוב לזיכרון עבור כל Thread בנפרד. הקוד נמצא במקום יחיד וכל Thread מריץ את שורות הקוד שהוא זקוק להן.

משתנים סטטיים וגלובליים: כפי שראינו משתנים סטטיים וגלובליים נשמרים באותו אזור בזיכרון, ה- data section. כיוון שThreadים חולקים כתובות בזיכרון, הם גם חולקים ביניהם את כל המשתנים הסטטיים והגלובליים.

אילו משאבים צריכים להיות ייחודיים לכל Thread?

רגיסטרים: הThreadים אינם חולקים ביניהם את הרגיסטרים. במילים אחרות, כאשר Thread ב' מחליף את Thread א', הערכים שנמצאים בכל הרגיסטרים צריכים להתחלף.
מכך שכל הרגיסטרים משתנים, אפשר להבין שבאופן ספציפי משתנים גם הרגיסטרים EIP ו-ESP (נשתמש בשמות המוכרים עבור רגיסטרים של 32 ביט, הכוונה זהה עבור מערכות של 64 ביט עם הרגיסטרים RIP ו-RSP).

מצביע הקוד: כזכור הרגיסטר EIP מצביע על הפקודה הבאה שצריכה להיות מורצת על ידי המעבד. הגיוני שהרגיסטר EIP יתחלף בין Threadים, כי הרי כל Thread מריץ חלק אחר של הקוד.

המחסנית: הרגיסטר ESP מצביע על מיקום ה-Stack, המחסנית של הThread. הגיוני שלכל Thread תהיה מחסנית משלו, מכיוון שהמחסנית כוללת את שרשרת הקריאות לכל הפונקציות, וכיוון שכל Thread מריץ קוד שונה יש לו שרשרת קריאות שונה על המחסנית.

למעשה, לכל Thread יש מחסנית נוספת, שמשרתת את ה-Kernel. ההפרדה בין מחסנית של ה-Thread למחסנית של ה-Kernel נחוצה לטובת שמירה על מעגלי האבטחה. אילו היתה רק מחסנית אחת שמשרתת את שניהם, היה אפשר להשתמש בטכניקות שונות כדי להריץ קוד ברמת הרשאה של Kernel. לדוגמה, לערוך את הערכים שנמצאים על המחסנית כך שפונקציות של ה-Kernel יקבלו ערכים שונים מכפי שהן היו אמורות לקבל, או לשתול כתובות חזרה כך ששרשרת הקריאות לפונקציות של ה-Kernel תריץ בשלב מסויים קוד של ה-Thread, וכך קוד שאמור להיות מורץ ב-Userland יורץ בהרשאות Kernel.

רגע אחד, איך יכול להיות לאותו Thread שתי מחסניות, אחת לו ואחת ל-Kernel, אם יש ל-Thread רק רגיסטר ESP אחד?

כפי שאתם זוכרים מהפרק הקודם, בכל פעם שיש צורך לקרוא לפונקציות של ה-Kernel, מבוצע Syscall. בשלב מסויים בשרשרת הקריאות, מבוצע Sysenter אל פונקציות של ה-Kernel. תוך כדי ביצוע ה-Sysenter, מוחלף ערכו של ESP כך שהוא מצביע על המחסנית של ה-Kernel. הטריק הקטן הזה מאפשר את הפרדת המחסניות וכך נשמרים מעגלי האבטחה.

משתנים לוקליים: כתוצאה מכך שכל Thread הוא בעל מחסנית אחרת, ניתן להסיק שבניגוד למשתנים גלובליים וסטטיים, משתנים לוקליים – שגם הם נשמרים על המחסנית - אינם משותפים לThreadים. כלומר, אם Thread כולל לדוגמה פונקציה שבתוכה מוגדר משתנה מקומי i המשמש בתור אינדקס ללולאת for, אף Thread אחר אינו מכיר את המשתנה i.

אפשר לתהות- הרי אמרנו שהזיכרון הוא משותף לכל הThreadים, והמחסנית היא בסך הכל אזור בזיכרון, אם כך איך יכול להיות שהמחסנית אינה משותפת לכל הThreadים? אמנם המחסניות של כל הThreadים נמצאות באותו מרחב זיכרון, ולכן תיאורטית Thread יכול לשנות את המחסנית של אחר, אך באופן מעשי כדי

לשנות ערכים במחסנית צריך לדעת היכן נמצאת המחסנית. מצביע המחסנית, ESP, משתנה בין Threadים ולכן לא פשוט ל-Thread לגשת למחסנית שאינה שלו.

אזור זיכרון Thread Local Storage (בקיצור TLS): דמיינו שיש לנו Thread כלשהו, שמבצע חישוב מסויים באמצעות משתנה גלובלי, או סטטי. המשתנה הזה נמצא באזור הזיכרון של כל ה-Threadים ששייכים לאותו Process ויש סכנה ש-Thread כלשהו ישנה את ערכו של המשתנה. פתרון אפשרי הוא להגדיר אותו כמשתנה לוקלי, על המחסנית, אבל לפעמים מאד נוח לנו להגדיר משתנה גלובלי. לדוגמה, הדבר מאפשר לכל הפונקציות שבקוד לגשת אליו. קיימת אפשרות נוספת, והיא להגדיר את המשתנה בתוך TLS, וכך למנוע מ-Threadים אחרים לגשת אליו.

עדיפות: לכל Thread יש עדיפות כלשהי, שה-Scheduler משתמש בה כדי להקצות זמן עיבוד. ב-Windows העדיפות היא מספר בין 0-31, כאשר מספר גבוה יותר משמעותו עדיפות גבוהה יותר. העדיפויות מתחלקות לשני תחומים: התחום בין 0-15 מיועד לעדיפויות רגילות, התחום בין 16-31 מיועד ל-Threadים שצריכים לרוץ בזמן אמת. המשמעות של ריצה בזמן אמת היא שחייב להיות מובטח שה-Thread יקבל זמן מעבד כל פרק זמן מוגדר, אחרת עלול להיות כשל. לדוגמה, נחשוב על שרת אינטרנט. השרת מחובר לכרטיס רשת, שמעביר לו מידע שהשרת צריך לפענח ולהשיב לו. המידע שמגיע מכרטיס הרשת מועבר לאזור זיכרון בגודל מסויים. אם השרת לא ימשוך את המידע מכרטיס הרשת כל פרק זמן מוגדר, אזור הזיכרון יתמלא, המידע החדש שמגיע יילך לאיבוד והשרת לא יוכל לטפל בו, לא משנה כמה משאבי מעבד יוקצו לעיבוד המידע. כלומר בעוד Threadים שרצים באופן רגיל צריכים לקבל *בממוצע* זמן מעבד מספק, Threadים שרצים בזמן אמת צריכים לקבל זמן מעבד על פרק זמן מוגדר.

יש קשר בין העדיפות של Thread לעדיפות של Process אליו ה-Thread שייך, אולם מכיוון שטרם הסברנו מהו Process, נמתין מעט עם ההסבר.

בשורה התחתונה, אנחנו צריכים שחלק מהמשאבים של הThreadים יהיו משותפים. אם הם לא יוכלו לשתף המשאבים ביניהם, במקום לקבל ריצה חלקה של Threadים שחולקים ביניהם את העבודה, נקבל אוסף Threadים שמתנגשים זה בזה, מחכים זה לזה ולא עובדים על אותה גרסה של המסמך.

משאבים שאינם משותפים בין Threadים של אותו Process	משאבים משותפים בין Threadים של אותו Process
ערכי רגיסטרים. ספציפית מצביע הקוד ומצביע המחסנית	זיכרון (יתר המשאבים בטבלה נגזרים מכך שהזיכרון הוא משותף)
Thread Local Storage	זיכרון Heap
Thread ID	קוד הרצה
עדיפות	משתנים גלובליים וסטטיים

כאשר אנחנו מבינים ש- Multi Threads הוא רעיון נפלא אך זקוק למנגנון שידאג לכך שחלק המשאבים יהיו משותפים, זה הזמן לדון בProcessים.

4.3 תהליכים - Processes

ה-Process מכיל את כל המשאבים המשותפים לכלל ה-Threads של אותה התוכנה. לדוגמה, כל הזיכרון ששייך לכל ה-Threadים של Word שפתוחים על אותו מסמך, יהיה תחת Process אחד. שימו לב לכך, שאם נפתח מסמך Word נוסף יהיה לו Process אחד עם Threadים אחרים.

אפשר לדמות את ה-Process למיכל של Threadים. המיכל מכיל את כל מה שצריכים Threadים בשביל לרוץ- כל המשאבים המשותפים יהיו מרוכזים אצלו.

כיוון ש-Process הוא רק מיכל, אין לו אפשרות להריץ קוד בעצמו. כל Process חייב Thread אחד לפחות שיריץ את הקוד. במילים אחרות, כאשר אנחנו מקליקים על תוכנה כדי להריץ אותה, נפתח Process חדש שמכיל לפחות Thread אחד. זהו ה-Main Thread, ממנו ניתן ליצור Threadים נוספים.

כל Process חדש שנפתח מקבל אוסף משאבים, אותם הוא חולק עם כל הThreadים שיש לו. אך מהם אותם משאבים?

קוד הרצה שמועתק ל-RAM: כאשר אנחנו מפעילים Process מסויים, כל ה-Threadים שהוא מפעיל חולקים את אותו הקוד. כל Thread יכול להשתמש בחלק אחר של הקוד, לדוגמה עבור Word יהיה Thread שמשתמש בקטע הקוד שמבצע בדיקת איות ו-Thread אחר ישתמש בקטע הקוד שמבצע שמירה של המסמך לדיסק הקשיח.

Thread אחד לפחות: כאמור Process הוא רק מיכל, אוסף של משאבים, ואינו יכול להריץ בעצמו קוד. על כן כל Process נפתח עם Thread אחד לפחות. כל ה-Threadים של Process הם משאבים של ה-Process.
הקצאת זיכרון: כל Process מקבל כמות של זיכרון, שהוא חולק עם כל ה-Threadים שנמצאים תחתיו. כיוון שהנושא מכוסה בפרק מיוחד, לא נרחיב על כך כעת.

עדיפות: נושא העדיפות הוסבר כאשר דיברנו על עדיפויות של Threadים. ל-Processים שונים עשויה להיות עדיפות שונה, לדוגמה Processים שמבצעים משימות ניהול עבור מערכת ההפעלה צריכים לקבל עדיפות גבוהה יותר מאשר, לדוגמה, Process של כרום שמציג סרטון יוטיוב. כל ה-Threadים שנמצאים תחת אותו Process יורשים ממנו את רמת העדיפות הבסיסית, אך תוך כדי ריצה ניתן לשנות את העדיפות שלהם. לדוגמה ניתן להפוך Thread לרמת עדיפות של ריצה ברקע, או אפילו ל-IDLE (אינו מבצע דבר, העדיפות

הנמוכה ביותר), אך גם ניתן להעלות את העדיפות של Threadים במידה מסויימת אפילו מעבר לעדיפות הבסיס של ה-Process.

מזהה ייחודי PID: לכל Process יש מספר מזהה ייחודי- Process ID, או בקיצור PID. אולי הבחנתם בכך כאשר עבדתם עם Process Explorer, ה-PID מוצג באחד הטורים.

טבלת Handle Table: נניח ש-Process רוצה לגשת למשאב מערכת או אובייקט של מערכת ההפעלה, כגון קובץ (בהמשך נכיר דוגמאות נוספות). אי אפשר לעשות זאת ישירות, אלא דרך אובייקט שנקרא Handle. נתקלנו כבר ב-Handle כאשר למדנו לתכנת WinAPI, הפונקציה CreateFile היא דוגמה לפונקציה שמחזירה Handle. לכל Process יש טבלה בה שמורים כל ה-Handleים שיש ברשותו.

הרשאת אבטחה Security Token: מערכת ההפעלה שומרת רשימה של הרשאות האבטחה שנדרשות על מנת לגשת לכל משאב. לכל Process יש Security Token, שצריך להיות לפחות לפי רמת ההרשאה של משאב ברשימה כדי לגשת אליו.

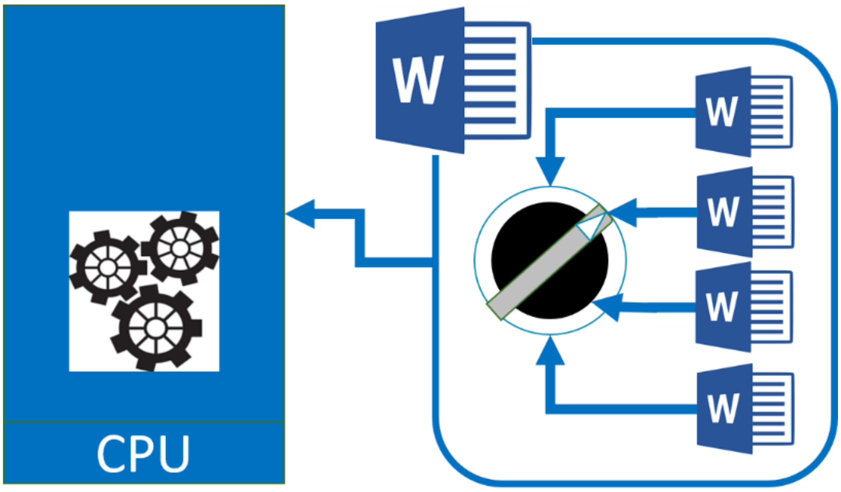
חשוב לציין שהדברים שנמצאים ברשימה הזו הם רק דברים ששייכים ספציפית ל-Process מסויים. ישנם משאבים שמספר Processים יכולים לחלוק ביניהם והם אינם נמצאים ברשימה זו. לדוגמה, Processים יכולים לחלוק זיכרון, נלמד על כך בהמשך. לדוגמה, קוד מסוג DLL הוא קוד שמיועד לשימוש של מספר Processים במקביל. נקדיש לנושא פרק מיוחד. לדוגמה, משתני סביבה, שהינם משתנים שמוגדרים לא בתוך התוכנה אלא מחוץ לה, על ידי המשתמש, ומאפשרים לקבוע דברים שונים באופן עבודת התוכנה.

תרגיל 4.1 מחקר משאבים של Process

פיתחו Process Explorer, בחרו Process כלשהו ומיצאו עבורו את המשאבים הבאים:

- ה-PID שלו
- העדיפות שלו
- כל ה-Handleים שיש לו (מסך תחתון, CTRL+H)
- קובץ ההרצה (קובץ ה-exe שנטען ל-RAM ושמשויך ל-Process). הקליקו על ה-Process ומיצאו תחת טאב "Image"
- כל ה-Threadים שלו
- בחרו Thread ספציפי ומיצאו את העדיפות שלו.

כעת נוכל לשדרג את ההסבר שלנו על אופן פעולת מערכת ההפעלה. עצרנו במודל של Multi-Threads, שבו מספר Threadים צריכים לפעול במקביל תוך שיתוף משאבים זה עם זה. מושג ה-Process עונה על השאלה כיצד מתבצע שיתוף המשאבים.



כאשר אנחנו מריצים Process, הוא מקבל את כל המשאבים הנדרשים ממערכת ההפעלה. הProcess פותח מספר Threadים כפי שנדרש לו, וכולם חולקים את המשאבים. כל Thread מקבל משאבי CPU בתורו, שנקבע על ידי Scheduler של מערכת ההפעלה.

נסכם את ההבדלים בין Process ו-Thread:

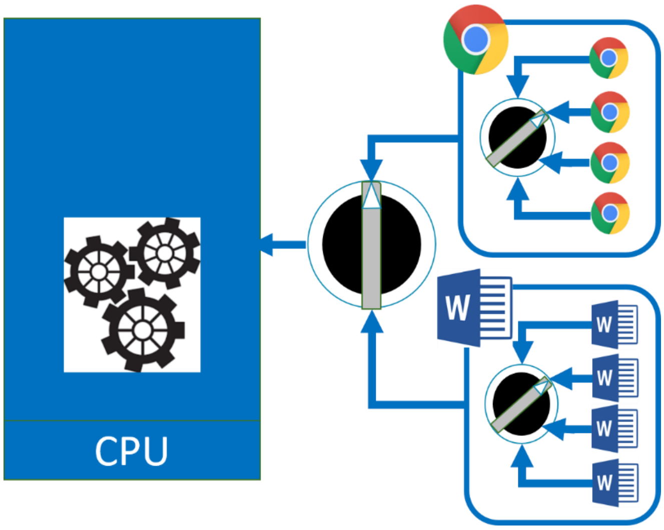
Thread	Process
יורש משאבים מה-Process	מקבל משאבים (זיכרון וכו') ממערכת ההפעלה
משתף משאבים עם Threadים אחרים תחת אותו Process	לא משתף משאבים עם Processים אחרים
מריץ קוד	אין לו קוד שרץ, חייב לפתוח Thread
חייב שיהיה לו Process אחד שייצור אותו	יכול להכיל כמה Threadים

4.4 ריבוי תהליכים - Multi-Process

אנחנו מבינים ש-Threadים ששייכים לאותו Process חולקים ביניהם את המשאבים של ה-Process, אולם Process יחיד לא פותר את כל הבעיות, כיוון שתוכנות רבות צריכות להיות מורצות בו זמנית.

הפתרון הוא כמובן עבודה עם Processים רבים בו זמנית. כל Process יקבל את המשאבים הנחוצים לו וירוץ באופן נפרד מה-Processים האחרים.

האיור הבא ממחיש את אופן הפעולה של Processים ו-Threadים תחת מערכת ההפעלה. במערכת שבאיור קיימים שני Processים: אחד עבור כרום ושני עבור Word. לכל Process יש מספר Threadים. כדי לבחור מי מקבל את זמן המעבד, ה-Scheduler של מערכת ההפעלה בורר את הProcess ובתוכו את הThread.



נסכם בהבדל בין ריבוי Threadים לריבוי Processים: בריבוי Threadים כל הThreadים ניגשים למרחב זיכרון בו קיים אותו קוד הרצה. ההבדל בין ה-Threadים הוא בעיקר בנקודה בקוד שבה הם נמצאים. כמו כן מרבית המשאבים שלהם הם משותפים, הדבר מתבטא לדוגמה בכך ש-Thread יכול לשנות משתנה גלובלי שהוגדר על ידי Thread אחר.

בריבוי Processים, לא לכל ה-Processים יש בהכרח את אותו הקוד. כלומר, יכול להיות מצב ששני Processים מריצים כרום, ואז יש להם את אותו קוד, אך זהו המצב החריג. בדרך כלל Processים שונים מריצים קוד שונים. כמו כן לכל Process יש משאבים שונים, כאשר הכוונה בעיקר למרחב זיכרון שונה. אך האם הדבר אומר ש-Processים רצים לחלוטין בנפרד זה מזה ואינם מסוגלים לשתף זיכרון? על שאלה זאת נענה בקרוב.

4.5 ביצוע Context Switch

המשמעות של המונח Context Switch היא "כל הדברים שצריכים להתבצע על מנת לעבור ממשימה למשימה אחרת". לדוגמה, אם יש לנו בילוי עם חברים לאחר אימון ספורט, נצטרך לעשות Context Switch שיכלול מקלחת והחלפת בגדים. כמו שלוקח זמן להתקלח ולהחליף בגדים, לוקח גם זמן למערכת ההפעלה לבצע Context Switch בין משימות שהיא צריכה לבצע. בחלק זה נדון בדברים שמערכת ההפעלה צריכה לבצע כדי לעשות Context Switch בין Threadים.

נניח שמערכת ההפעלה מבצעת Context Switch בין Threadים של אותו Process. מה הדברים שמערכת ההפעלה צריכה לשנות?

ראינו שעיקר ההבדל בין Threadים הוא הערכים שיש ברגיסטרים שלהם. בזמן שמערכת ההפעלה מבצעת החלפה בין Threadים של אותו Process, כל מה שהיא צריכה לעשות זה לשמור בצד את הערכים שיש ברגיסטרים ולטעון אל הרגיסטרים את הערכים המתאימים ל-Thread החדש. ברגע שהערכים החדשים טעונים, גם הערך של EIP משתנה כך שהוא מצביע על הפקודה הבאה שה-Thread צריך להריץ

ושה-Thread עצר בה בפעם הקודמת שזמן המעבד ניתן לו. הערך של ESP גם הוא משתנה וכך ה-Thread החדש מקבל חזרה את הגישה למחסנית שלו, שנותרה ללא שינוי מאז הפעם הקודמת שה-Thread רץ. כלומר, ביצוע Context Switch בין Threadים של אותו Process הוא פעולה שדורשת זמן אך למרות זאת אינה מסובכת במיוחד.

לעומת זאת, ביצוע Context Switch בין Threadים שאינם באותו Process היא פעולה מורכבת הרבה יותר. הסיבה לכך היא ש-Threadים שנמצאים ב-Processים שונים אינם חולקים זיכרון זה עם זה. עדיין לא למדנו על זיכרון וכיצד הוא פועל, אבל לצורך הפשטות נניח שיש לנו זיכרון RAM בגודל 1MB ושתי תוכניות, שכאשר כל אחת מהן טעונה לזיכרון בנפרד היא תופסת 1MB, כלומר את כל זיכרון ה-RAM. לכן, על מנת לבצע Context Switch בין ה-Processים שמריצים את שתי התוכניות, מערכת ההפעלה תצטרך למעשה להחליף את כל זיכרון ה-RAM של המעבד. בשלב הראשון מערכת ההפעלה תעתיק את כל זיכרון ה-RAM של Process א' אל הדיסק הקשיח. היא חייבת לעשות זאת מכיוון שאחרת יאבדו כל הערכים של המשתנים שנוצרו והשתנו בזמן הריצה. לאחר מכן להעתיק מתוך הדיסק הקשיח אל ה-RAM את הזיכרון של Process ב', כפי שנשמר בדיסק הקשיח בפעם האחרונה שProcess ב' סיים את הריצה שלו.

אפשר כמובן לטעון ששלב א', ההעתקה של כל הזיכרון של Process א' אל הדיסק הקשיח, יכול לעבור שיפור ביצועים ניכר. אפשר להעתיק חזרה לדיסק הקשיח רק את השינויים שה-Process ביצע תוך כדי ריצה. אך הדבר אינו משנה את השורה התחתונה, ככל שיש יותר העתקות זיכרון נדרש זמן רב יותר לביצוע Context Switch והמשתמש עלול להרגיש שהמחשב "איטי". זו גם הסיבה לכך שכדי שמחשב ירוץ במהירות, חשוב לא רק מהירות המעבד אלא גם גודל ה-RAM.

מה בכל זאת אפשר לעשות כדי שביצוע Context Switch בין Processים יתבצע במהירות? נלמד על כך בחלק שיוקדש לזיכרון.

4.6 שימוש ב-Multi-Process לעומת Multi-Thread

מתי תוכנה תשתמש ב-Multi-Processing ומתי ב-Multi-Threading?

נפתח Process Explorer. אם יש לכם דפדפן כרום שרץ עם כמה טאבים, תוכלו לראות תמונה שדומה לתמונה הבאה- תחת כרום יש Sub Processes שגם הם נושאים את השם כרום. כלומר, הProcess כרום יצר Processים בנים באותו השם.

Process	CPU	Private Bytes	Working Set	PID	Description
chrome.exe	0.01	416,744 K	354,032 K	36680	Google Chrome
chrome.exe		1,796 K	2,348 K	29276	Google Chrome
chrome.exe		1,988 K	2,328 K	35228	Google Chrome
chrome.exe		766,112 K	281,088 K	37704	Google Chrome
chrome.exe	0.05	46,668 K	50,056 K	30740	Google Chrome
chrome.exe		47,848 K	15,056 K	19236	Google Chrome
chrome.exe		256,964 K	223,592 K	36740	Google Chrome
chrome.exe		32,332 K	16,716 K	25012	Google Chrome
chrome.exe		90,700 K	29,764 K	36900	Google Chrome

כמובן שלא רק כרום יוצר Sub Processes כאשר הוא פועל. דוגמה נוספת היא שירות המוזיקה Spotify:

Process	CPU	Private Bytes	Working Set	PID	Description
Spotify.exe	0.90	140,100 K	93,776 K	11440	Spotify
Spotify.exe	< 0.01	32,812 K	4,276 K	11576	Spotify
Spotify.exe	0.18	120,932 K	66,496 K	11820	Spotify
Spotify.exe	< 0.01	44,440 K	15,672 K	11944	Spotify
Spotify.exe	0.57	174,108 K	125,728 K	12208	Spotify

ספוטיפיי פותח Processים נוספים כדי לחלק את המשימות שעליו לבצע. לדוגמה, משימה אחת היא להציג למשתמש את הממשק הגרפי הכולל רשימת שירים ואפשרויות חיפוש. משימה זו דורשת עבודה מול הכרטיס הגרפי של המחשב. משימה אחרת כוללת עבודה מול כרטיס הקול והשמעה של המוזיקה. משימה אחרת היא תקשורת מול השרת של ספוטיפיי והזרמה של השירים אל המחשב.

אך חלוקת משימות יכולה להתבצע גם על ידי Threadים. למה ספוטיפיי – ותוכנות רבות נוספות – מתעקשות לפתוח דווקא Processים נוספים? למה לא לבצע הכל רק על ידי Multi-Threading?

ראשית, במקרים רבים חלוקת משימות אכן מתבצעת על ידי פתיחת Threadים נוספים. הנה לדוגמה כרום, שלמרות שראינו שפותח Processים נוספים, הוא משתמש גם בריבוי Threadים. מתוך Process Explorer, הקליקו על הProcess המבוקש ובתוכו הקליקו על הטאב Threads. תוכלו לראות את כל ה-Threadים שה-Process עושה בהם שימוש:

chrome.exe:29276 Properties

Image | Performance | Performance Graph | GPU Graph | Threads | TCP/IP | Security | Environment | Strings

Count: 7

TID	CPU	Cycles Delta	Start Address
31904			chrome.exe!GetHandleVerifier+0xa6da0
26808			chrome.exe!GetHandleVerifier+0x5ae90
36524			chrome.exe!GetHandleVerifier+0x5ae90
34868			chrome.exe!GetHandleVerifier+0x5ae90
36352			chrome.exe!GetHandleVerifier+0x2db60
36124			chrome.exe!GetHandleVerifier+0x2db60
16212			ntdll.dll!RtlInitializeResource+0x410

יש יתרונות רבים לחלוקת המשימות ל-Threadים. כיוון שהם חולקים משאבים, קל יחסית לנהל משאבים משותפים. וכפי שראינו, ביצוע Context Switch בין Threadים של אותו Process הוא משימה חסכונית יותר מאשר בין Processים שונים.

אך גם לחלוקה ל-Processים יש מספר יתרונות חשובים. היתרון הראשון הוא חוסר תלות- אם Process בן קורס עקב באג כלשהו, יתר ה-Processים ממשיכים לפעול. באופן ספציפי במקרה של כרום, הדפדפן מריץ תוספים שונים שמותקנים עליו. התוספים יכולים להיות חוסם פרסומות, תוספים שהאנטיוירוס התקין, או תוספים שפותחו לטובת שיפור חוויית המשתמש באתרים ספציפיים. התוספים הללו יכולים לבוא ממקורות שונים וגוגל מפתחת הכרום לא יכולה להבטיח שהתוספים הללו לא יקרסו תוך כדי ריצה. מבחינת כרום, הגיוני לתת לכל תוסף לרוץ ב-Process משלו, כך שקריסה של תוסף לא תגרום לסגירת כל פעולת הגלישה שלנו. כמו כן, גם גלישה לדף אינטרנט עלולה "להתקע" או במקרים נדירים אפילו לקרוס. לכן גם לטאבים שונים שפתוחים בכרום מוקצים Processים שונים.

היתרון השני של Multi Processing הוא בכך שרוב המחשבים הביתיים המודרניים כוללים מספר מעבדים. בדרך כלל ארבעה או שמונה. כאשר Process פותח מספר Threadים, יותר יעיל להריץ אותם על אותו מעבד. מיד נבין מדוע. לכן חלוקה ל-Processים מאפשרת לנצל בצורה יותר טובה את ריבוי המעבדים שיש במחשב.

אם כך, מדוע יש יתרון בהרצת Threadים של אותו Process על אותו מעבד? כיוון שטרם למדנו על הזיכרון, נסקור באופן כללי איך הדברים עובדים, והדברים יפורטו בפרק שעוסק בזיכרון. לכל מעבד יש זיכרון קטן ומהיר שצמוד אליו. בניגוד ל-RAM שהוא משאב משותף לכל מעבדים, הזיכרון המהיר של כל מעבד שייך לו בלבד, נקרא לו באופן זמני "זיכרון פרטי". יש כל הזמן חילופי מידע בין ה-RAM המשותף לזיכרון הפרטי של כל מעבד. לב הבעיה בחלוקה של מספר Threadים של אותו Process בין כמה מעבדים, היא בכך שאם כל מעבד משנה דברים בזיכרון הפרטי שלו, ולאחר מכן השינויים מועתקים אל ה-RAM המשותף, אז עלולה להיות התנגשות. לדוגמה בזכרון הפרטי של מעבד א כתוב שערכו של המשתנה הגלובלי X הוא 5, ואילו בזכרון הפרטי של מעבד ב כתוב שערכו של X הוא 4. מהו הערך ה"נכון" אותו יש לשמור ב-RAM? יש צורך ליצור מנגנון שימנע בעיות כאלה. קיים מנגנון כזה, אך פעולת התיאום לא מגיעה חינם אלא יש לה עלויות של זמן עיבוד. לכן, כאשר רוצים לחלק משימה באופן מיטבי בין מעבדים, יש יתרון לריבוי Processים על פני ריבוי Threadים.

היתרון השלישי של ריבוי Processים הוא אבטחה. כיוון שהתחלנו את ההסבר שלנו מדפדפן כרום, נסקור בקצרה את מנגנון ה-Chrome Sandbox. אחת הבעיות שכרום נתקל בהן היא שאתרי אינטרנט עלולים להיות זדוניים ולנסות לגרום לדפדפן לבצע פעולות זדוניות. אם האקר מוצא באג בדפדפן הוא יכול לנצל אותה כדי לגרום לדפדפן להריץ קוד כרצונו. לדוגמה, להוריד תוכנה מסויימת לרשום אותה ב-Registry בלי לשאול את המשתמש. מוקדם יותר בפרק הזכרנו שלכל Process יש Security Token ושכאשר Process מבקש לגשת למשאב מסויים שדורש הרשאה, מערכת ההפעלה משווה את ה-Security Token לרמת ההרשאה של המשאב. עלול להיות מצב שהקוד הזדוני ינצל את ה-Security Token של כרום כדי לגשת למשאבים מאובטחים, כגון ה-Registry, כיוון שמבחינת מערכת ההפעלה מי שמבקש לגשת למשאב המאובטח הוא כרום. למערכת ההפעלה אין מושג שכרום מריץ קוד זדוני שדף אינטרנט גרם לו להריץ. הפתרון של כרום הוא

לייצר Sandbox- "ארגז חול". זהו Process שיש לו את ההרשאות הנמוכות ביותר, כך שהוא חסר תועלת בשביל האקר בעל מטרות זדוניות.

לסיכום, כפי שאנחנו רואים שמתרחש בפועל, תוכנות משלבות בין יתרונות של Multi-Threading ו-Multi-Processing על מנת לנצל בצורה המיטבית את משאבי העיבוד שנתונים לרשותם.

4.7 מהם Services / Daemons (הרחבה)

המונחים Service ו Daemon הינם שמות נרדפים, כאשר Service מקובל בסביבת Windows ואילו Daemon בסביבת Linux. המונח מציין סוג מיוחד של Process שיש לו שלושה מאפיינים:

א. מופעל אוטומטית על ידי מערכת ההפעלה

ב. מנוהל על ידי מערכת ההפעלה

ג. אין לו מסך משתמש, כלומר הוא רץ מאחרי הקלעים.

בסביבת Windows הניהול של כל ה-Services מתבצע באמצעות Process שנקרא Service Control Manager, או בקיצור SCM. ה-SCM דואג בין היתר להפעיל אוטומטית את ה-Service, להחזיק מידע על הסטטוס של ה-Service ולהעביר אליו וממנו מידע.

נתעמק מעט בנושא ההפעלה האוטומטית. ברור שזהו הבדל משמעותי בין Service ל- Process"רגיל", כגון Word. כדי להפעיל Word צריך שהמשתמש יקליק על התוכנה של Word. לעומת זאת, Service רץ אוטומטית באמצעות ה-SCM. אם כך אפשר לשאול, מה ההבדל בין Service לבין Process שרץ בעזרת ה-Scheduled Tasks? נזכור, שכאשר למדנו על תוכנת Autoruns ראינו שאחת הדרכים לגרום לתוכנה לרוץ באופן אוטומטי היא להכניס אותה ל-Scheduled Tasks. אם כך, האם גם תוכנה שרצה אוטומטית באמצעות Scheduled Tasks היא בעצם Service?

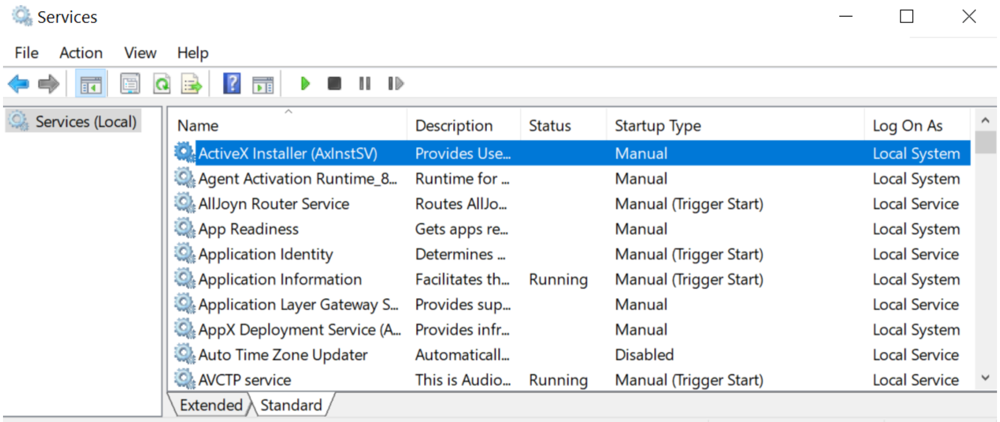
כאשר תוכנה רצה באמצעות Scheduled Task, אחת לזמן המוגדר לה היא "מתעוררת" ומבצעת את המשימה שלה, לאחר מכן נסגרת. יש בכך יתרון, מכיוון שבזמן שהתוכנה סגורה אין לה Process שדורש משאבים. זה מתאים למשימות מסוג מסויים. לדוגמה, לבדוק אחת ליום אם יש עדכון אנטיוירוס. לעומת זאת, Service רץ כל הזמן ויכול להגיב לארועים שאי אפשר לדעת מראש מתי הם יקרו. נסקור מספר דוגמאות ל-Serviceים, נסו לחשוב מה היה קורה אם התוכנות הללו היו "מתעוררות" רק אחת לזמן מה, ולא מסוגלות להגיב באופן מיידי לארועים שמתרחשים, כגון משתמש שמבצע Log-In או חיבור של המחשב לרשת:

דוגמאות ל-Serviceים:

- DHCP Client: כזכור מלימודי הרשתות, פרוטוקול DHCP מאפשר למחשב לקבל כתובת IP, כתובת IP של שרת DNS, מסכת רשת וראוטר ברירת מחדל. המחשב נדרש לפעול מול שרת DHCP. על כן נחוץ שהמחשב יפעיל עם העליה שלו DHCP Client. אין טעם להטריח את המשתמש, לכן הפעולה מבוצעת מאחרי הקלעים כ-Service
- User Profile Service: כאשר אנחנו מבצעים Log In למחשב, באופן אוטומטי התצוגה עולה כפי שהיא שמורה בפרופיל שלנו ב-Registry, ויש שימוש בהרשאות ההרצה שלנו לתוכנות השונות

שמותקנות על המחשב. הדברים מתבצעים על ידי Service, שהרי אין הגיון לבקש מהמשתמש להריץ תוכנה שתבדוק את ההרשאות שלו ותחסום גישה לתוכנות מסויימות...

קיימים עוד Serviceים רבים שרצים ממש כרגע על המחשב שלכם. מעוניינים לחקור אותם? כדי למצוא אותם, הקישו על WinKey+R ובחלון הרצת התוכנות שנפתח כיתבו services.msc.

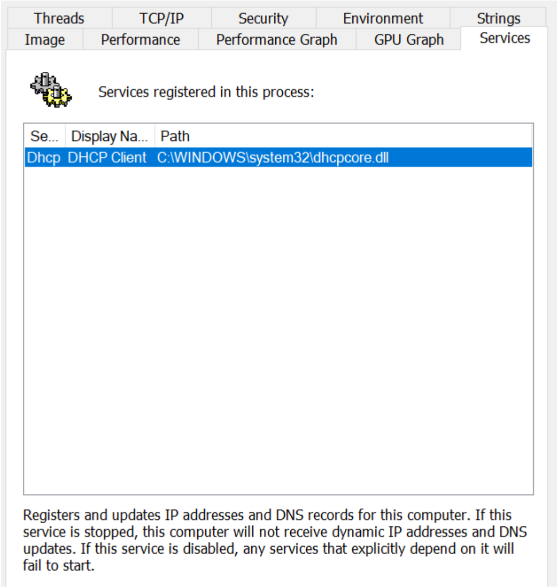


תוכלו לבחור כל Service שרץ אצלכם ולקרוא עליו פרטים, כגון התפקיד שלו.

איך אפשר למצוא את ה-Serviceים הללו ב-Process Explorer? פיתחו את Process Explorer וחפשו את svchost.exe. תמצאו לא מעט Processים שזה השם שלהם. מדוע?

כפי שנראה בקרוב, יצירה של Process חדש דורשת משאבים לא מעטים. אילו כל Service של Windows היה רץ בתוך Process, משאבים רבים היו מתבזבזים. לכן, מתכנתי Windows איפשרו למערכת ההפעלה ליצור Processים בשם svchost.exe, שכל אחד מהם מרכז מספר Serviceים של מערכת ההפעלה, ומכאן גם מגיע השם svchost- קיצור של service ו-host. זהו Process ש"מארח" Serviceים.

אילו services פועלים תחת כל Process של svchost.exe? הקליקו על אחד הProcessים ובחרו את טאב Services. תוכלו לקבל הסבר על תפקידו של כל Service באמצעות בחירת Service מסויים, ההסבר ניתן בתחתית המסך:



תרגיל 4.2 מחקר Services

בחרו שני services שנמצאים במצב running, באמצעות Process Explorer חיקרו מה הם מבצעים.

שימו לב שכל השמות של ה- services מסתיימים בסיומת dll. קבצי DLL, קיצור של Dynamic Link Library, הם קבצים מושלמים למשימה הזו, מכיוון שהם מיועדים להרצה על ידי Processים שמריצים דברים שונים. ניתן כרגע רקע קצר כיצד הדברים עובדים, ונפרט על כך בפרק מיוחד שיוקדש ל-DLLים.

קובץ DLL כולל פונקציות שונות, שמי שמתכנת את ה-DLL מאפשר לתוכנות אחרות להשתמש בהן. לדוגמה, מתכנת יכול ליצור פונקציה בשם hello ולכלול אותה ב-DLL. פורמט ה-DLL מאפשר לתוכנה אחרת, שכתב מתכנת אחר, לקרוא לפונקציה hello ולהשתמש בה. כמו כן DLLים כוללים פונקציה ראשית בשם DllMain, שנקראת אוטומטית ברגע שתוכנה כלשהי מבקשת להשתמש ב-DLL המבוקש. התוצאה היא מנגנון מאד נוח, שמאפשר לProcess להיות פשוט מיכל של משאבים, שמפעיל DLLים.

בגרסאות ישנות של Windows, במקום שכל DLLיקבל Process משלו, כמה DLLים חלקו את המשאבים שלהם זה עם זה, כדי להשיג חסכון במשאבים. החל מגרסה 1703 של Windows 10, יש נטיה להריץ כל Service באמצעות Process משלו, משיקולי אבטחה.

תרגיל 4.3 תרגיל מודרך – יצירת Thread

בתרגיל זה נשתמש ב-WinAPI שלמדנו בפרק הקודם. התכנות ב-WinAPI ילווה אותנו כל משך לימוד מערכת ההפעלה Windows, כיוון שזו הדרך הטובה ביותר להתנסות בחומר הנלמד.

בשלב זה, נלמד ליצור Thread.

הפונקציה CreateThread יוצרת Thread חדש, תחת ה-Process שמריץ את CreateThread. במילים אחרות, אם אנחנו מריצים את התוכנית Hello.exe והיא מפעילה את הפונקציה CreateThread, יווצר Thread חדש תחת הProcess שנקרא Hello.exe.

בעת יצירת הThread, אנחנו קובעים שני דברים. הראשון- מה הפונקציה הראשית שתרוץ בThread. נניח שהקוד שלנו כולל את הפונקציות bla ו-stam. כאשר אנחנו יוצרים Thread חדש, אפשר לקבוע שהוא יריץ את הפונקציה bla או את הפונקציה stam. הדבר השני – אנחנו יכולים להעביר פרמטר (כן, פרמטר אחד) שיועבר לפונקציה הנבחרת. מה עושים אם רוצים להעביר לפונקציה מספר פרמטרים? כאשר נגיע לגשר, נעבור אותו. בינתיים אתם מוזמנים לחשוב על פתרון אפשרי להעברה של יותר מפרמטר אחד.

שלב א'- הגדרת הפונקציה הנקראת

בשלב זה נלמד כיצד להגדיר פונקציה באופן ש-CreateThread יכול לקרוא לה.

הפונקציה חייבת להיות מוגדרת כך:

```
DWORD WINAPI function_name(LPVOID lparam);
```

ההנחיה WINAPI מגדירה את ה-Calling Convention של הפונקציה. הבנה מהו Calling Convention לא הכרחית לטובת התכנות, אבל זהו ידע חשוב לטובת משימות אחרות, כגון קריאת קוד אסמבלי. בתמצית, הכוונה לדרך שבה פרמטרים מועברים לפונקציה, לדרך בה מוחזר ערך מהפונקציה ולדרך שבה המחסנית מנוקה מהפרמטרים שהועברו אליה. אפשר לקרוא פרטים נוספים בספר האסמבלי של המרכז לחינוך סייבר, בפרק אודות המחסנית ופרוצדורות.

להלן דוגמה לקוד שכולל פונקציה שמוגדרת לפי הדרישות של CreateThread. הקוד כולל העברה של פרמטר מסוג LPVOID ושימוש בו:

```
#include "pch.h"

DWORD WINAPI bla(LPVOID lparam) {
    for (INT i=1; i<=*(PINT)lparam; i++) {
        printf("bla\n");
    }
    return 1;
}

int main()
{
    DWORD count = 2;
    LPVOID pcount = &count;
    bla(pcount);
}
```

בקוד זה העברנו פרמטר יחיד. כפי שרואים, הגדרת הפונקציה מחייבת שליחה של פרמטר יחיד. אך זה לא אומר שאי אפשר להעביר יותר מפרמטר אחד- נוכל לעקוף את המגבלה אם ניצור Struct שיכלול את כל הפרמטרים שלנו, ונעביר אותו.

תרגיל 4.3.1 העברת Struct ל-Thread

שנו את קוד התוכנית כך שיועברו לפונקציה 2 מספרים, באמצעות Struct שכולל משתנים מטיפוס INT. הפונקציה תחשב את הסכום שלהם ותבצע איתו פעולה לפי רצונכם.

פתרון:

```
#include "pch.h"

struct MyStruct
{
    INT a;
    INT b;
};

DWORD WINAPI bla(LPVOID lparam) {
    struct MyStruct *nums = (MyStruct*)lparam;
    INT count = nums->a + nums->b;
    for (INT i = 1; i <= count; i++) {
        printf("bla\n");
    }
    return 1;
}

int main()
{
    struct MyStruct *my_struct =
       (MyStruct*)malloc(sizeof(MyStruct));
    my_struct->a = 3;
    my_struct->b = 4;
    LPVOID pstruct = my_struct;
    bla(pstruct);
}
```

שלב ב' - קריאה ל- CreateThread

קיים על הפונקציה CreateThread תיעוד מקיף ב-MSDN. בכל פעם שאתם צריכים להשתמש בפונקציה כלשהי של WinAPI, פשוט חפשו בגוגל את שם הפונקציה ו-MSDN.

על סמך התעוד ב-MSDN, הפונקציה CreateThread מקבלת שישה פרמטרים, כאשר מתוכם רק שני פרמטרים (השלישי והרביעי) חייבים להקבע על ידי המשתמש, ואילו ארבעת הנותרים יכולים להיות NULL או אפס.

אם כך, שני הפרמטרים היחידים שאנחנו צריכים לקבוע הם שם הפונקציה שה-Thread יריץ, והפרמטר שהפונקציה תקבל. כל מה שנותר לנו לעשות הוא למלא את הפרמטרים השונים לפי התבנית. מומלץ לשמור על ההערות לגבי משמעות כל פרמטר- הן עוזרות בתהליך הדיבוג. לדוגמה, כדי לוודא שלא התבלבלנו ושינינו את סדר הפרמטרים המועברים.

תרגיל 4.3.2 שימוש ב-CreateThread

נסו להוסיף את הקריאה ל-CreateThread בעצמכם, בלי להעזר בקוד הבא.

פתרון:

```
int main()
{
    struct MyStruct *my_struct =
          (MyStruct*)malloc(sizeof(MyStruct));
    my_struct->a = 3;
    my_struct->b = 4;
    LPVOID pstruct = my_struct;
    HANDLE hThread = CreateThread(
        NULL,       //default security attributes
        0,          //default stack size
        bla,        //thread function
        pstruct,    //thread param
        0,          //default creation flags
        NULL        //return thread identifier
    );
}
```

כפי שבטח שמתם לב, הפונקציה שאנחנו קוראים לה לא מדפיסה למסך דבר. אם לדייק, הסיכוי לכך שיודפס משהו הוא אמנם קיים אבל קלוש. מיד נבין למה. נראה כאילו דבר לא מתרחש. האם זה מכיוון שיש לנו באג כלשהו בקוד? אולי לא נוצר Thread חדש? אולי יש בעיה עם ההדפסה מה-Thread החדש אל המסך?

כדי לנסות להגיע לבד אל התשובה, הוסיפו לאחר הקריאה ל-CreateThread מספר שורות קוד שלא מבצעות כלום. לדוגמה:

```
for (INT i = 0; i < 100000; i++)
        continue;
```

הפתעה! לפתע מודפס למסך הפלט של הפונקציה שקראנו לה (אם זה לא קורה, הריצו פעם נוספת או הגדילו את כמות הפעמים שהלולאה מתבצעת). חישבו- מדוע זה קורה? מדוע תוספת של לולאה שלא מבצעת כלום גורמת לכך שה-Thread שקראנו לו מדפיס פלט? כיצד הדברים קשורים? לאחר שחשבתם על כך, עיברו לחלק הבא.

4.7.2 שלב ג' – WaitForSingleObject

נסביר מה התרחש וכיצד שורות הקוד שהוספנו השפיעו. זה התהליך שהתרחש לפני הוספת הלולאה המיותרת:

1. עם הרצת התוכנית נוצר Process חדש ובתוכו Thread ראשי. הפונקציה main שייכת לThread הראשי
2. ה-Thread הראשי ביצע פתיחה של Thread נוסף והעביר לו פונקציה ופרמטר
3. מיד לאחר פתיחת ה-Thread הנוסף, ה-Thread הראשי הגיע לסוף פונקציית ה-main. כאשר ה-Thread הראשי מגיע לסוף ה-main, ה-Process נסגר וסוגר איתו את כל ה-Thread'ים שעדיין פתוחים. במילים אחרות, ה-Thread הנוסף בכלל לא קיבל הזדמנות לרוץ

לאחר הוספת הלולאה ה"מיותרת", ה-Thread הראשי ממשיך לרוץ ולבצע את הלולאה. הוא אינו נסגר. הזמן בו ה-Thread הראשי ממשיך לרוץ, נותן הזדמנות ל-Scheduler של מערכת ההפעלה להקצות זמן מעבד ל-Thread הנוסף. ה-Thread הנוסף מסיים את העבודה שהגדרנו לו, ה-Scheduler חוזר ומקצה זמן מעבד ל-Thread הראשי עד גמר הריצה שלו וסגירה שלו ושל ה-Process.

למעשה, בזכות תוספת הלולאה המיותרת קיבלנו הצצה אל מנגנון התזמון של ה-Scheduler. אפשר לשחק עם המספרים בלולאה ולקבל הערכה לגבי הקצב שבו ה-Scheduler מחליף בין ה-Thread'ים.

לאחר שהבנו את הבעיה, נתקן אותה. התיקון הוא פשוט ומתבצע על ידי הפונקציה WaitForSingleObject. פונקציה זו מקבלת אובייקט כלשהו- רשימת האובייקטים נמצאת בתעוד הפונקציה ב-MSDN והיא כוללת בין היתר אובייקטים מסוג Process, מסוג Mutex (סבלנות, ממש בקרוב נכיר) ומסוג Thread, כמובן. יחד עם האובייקט, מקבלת הפונקציה את כמות הזמן המקסימלי שהתכנית צריכה להמתין לאובייקט.

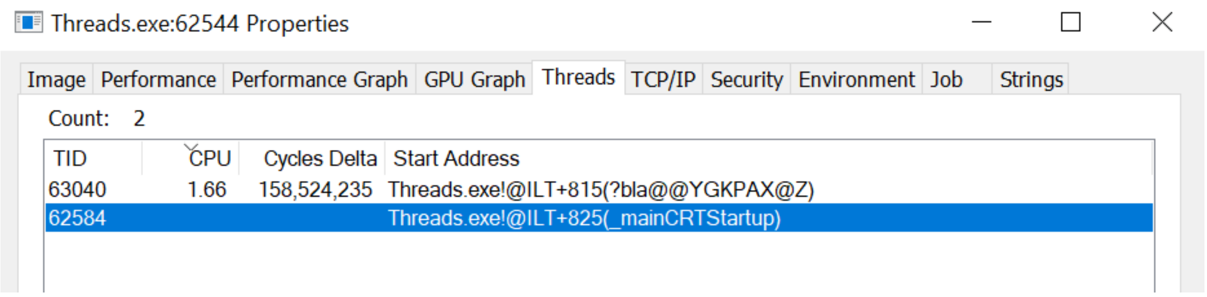
לכן כל מה שאנחנו צריכים לעשות הוא להוסיף את הקריאה הבאה:

```
WaitForSingleObject(hThread, INFINITE);
```

לבסוף, בואו נבדוק איך רואים את התוכנית שלנו ב-Process Explorer!

כדי שריצת התוכנית לא תסתיים מהר מדי, ניצור בתוך הThread החדש לולאה, או עדיף- נקרא לפונקציה Sleep (חפשו אותה ב-MSDN). שימו לב לשים את הקוד בתוך הקוד שמריץ הThread, ולא בתוך ה-main. אחרת, ריצת הThread תסתיים מהר מדי בשביל להבחין בפרטים שאנחנו רוצים לבדוק.

בתוך Process Explorer נקליק על הProcess של התכנית, ומבין הטאבים השונים נבחר את Threads (שימו לב שבדוגמה זו שם התוכנית שמורצת הוא Threads.exe):



ה-Thread התחתון הוא הראשי. אפשר לראות שכתובת ההתחלה שלו היא _mainCRTStrartup.
ה-Thread העליון הוא ה-Thread שנוצר בזמן הריצה, ואפשר לראות שכתובת ההתחלה שלו היא הפונקציה שהעברנו לו.

תרגיל 4.4 **תרגיל מודרך - יצירת** Process

בחלק זה נלמד להשתמש בפונקציה CreateProcess. לפני שנכנס לפרטים של הפרמטרים שלה, יש דבר כללי שצריך להבין: הפונקציה הזו מקבלת כפרמטר קובץ exe שירוץ על ידי ה-Process החדש שיווצר. לכן נחלק את התרגיל שלנו לשני שלבים. בשלב הראשון ניצור קובץ exe שמבצע משהו קטן, ובשלב השני נלמד להשתמש ב-CreateProcess כדי להריץ אותו.

שלב א' – יצירת קובץ הרצה

למעשה ברגע שנקמפל את תוכנית ברירת המחדל שנוצרת על ידי Visual Studio כבר יש בידינו קובץ הרצה, אבל אנחנו נעשה לו שדרוג קטן על ידי הוספת ארגומנטים (תזכורת- ארגומנט הוא המידע שנשלח לתוך פונקציה, פרמטר הוא המשתנה המקומי בתוך הפונקציה שמקבל את הערך של הארגומנט) בכניסה ל-main. צרו פרוייקט בשם HelloWorld, העתיקו אליו את הקוד הבא ושימו לב לשורה הראשונה:

```
int main(int argc, char *argv[])
{
     if (argc != 2) {
     printf("Incorrect number of arguments\n");
     return 0;
     }
     int id = atoi(argv[1]);
     printf("Hello World! Process got argument %d\n", id);
     return 1;
}
```

הפרמטר הראשון argc הוא קיצור של Argument Count, כלומר מספר הארגומנטים שהפונקציה מקבלת.
הפרמטר השני argv הוא קיצור של Argument Vector, כלומר מערך של מצביעים לארגומנטים.

אם ההסבר הזה נראה לכם לא מובן, במקרה זה קל הרבה יותר לראות את הדברים קורים מאשר לקרוא הסבר תיאורטי.

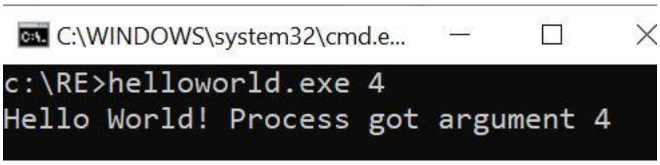
א. קמפלו את התוכנית. אל תשכחו לעשות-

`include <iostream>`

ב. פיתחו cmd

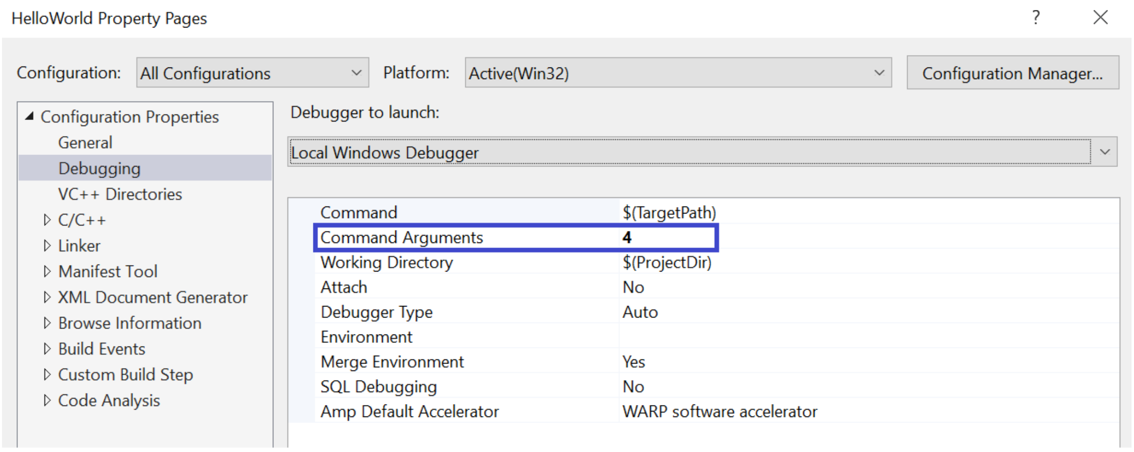
ג. באמצעות פקודות cd הגיעו אל התיקיה שבה נשמר קובץ ההרצה HelloWorld.exe

ד. כיתבו בשורת הפקודה HelloWorld.exe, לאחר מכן רווח, ואז מספר כלשהו, ולחצו `enter`. אתם צפויים לראות תוצאה דומה לתוצאה הבאה:



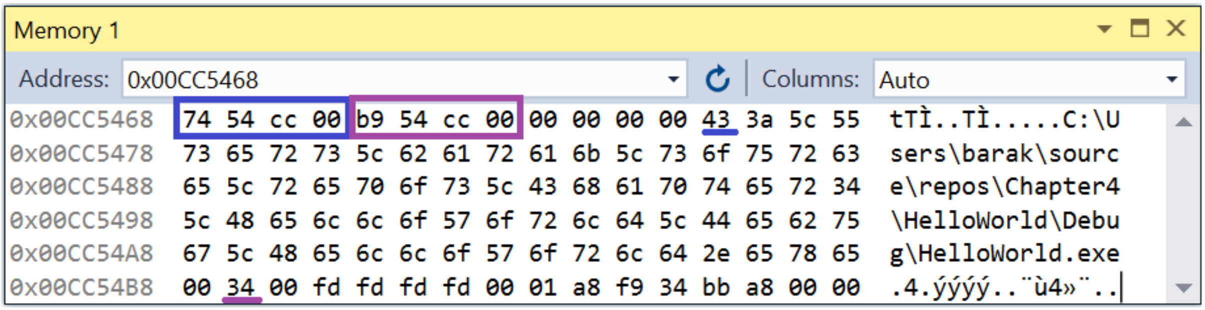
יפה. רואים שהתוכנית הדפיסה את הארגומנט שהעברנו לה, המספר 4.

בואו נבחן מה קורה בתוך התכנית עצמה. בתוך Visual Studio הכנסו אל הגדרות הפרוייקט והוסיפו מספר כלשהו בשורה של Command Arguments:



שימו Breakpoint בתחילת ה-main והריצו במצב Debug. בחלון המשתנים אתם מבחינים בכך שהערך של argc הינו 2. למה דווקא 2? הרי העברנו רק מספר אחד כארגומנט?

הכתובת בזיכרון של argv כתובה גם היא בחלון המשתנים. פיתחו חלון זיכרון (לחיצה בו זמנית על Ctrl+Alt+m ולאחר מכן בחירה של מספר בטווח 1 עד 4) והעתיקו לשדה הכתובת את הכתובת של argv. לפניכם יופיע מבנה הזיכרון של argv, כמובן שהנתונים של HelloWorld והכתובת בזיכרון של argv שונים מאשר בדוגמה:



רואים ש- argv מתחיל בשתי כתובות בזיכרון. הכתובת הראשונה 0x00cc5474 היא הכתובת שבה מתחילה המחרוזת שמבטאת את מיקום קובץ ההרצה, ואילו הכתובת השניה 0x00cc54b9 היא הכתובת שבה נמצא הארגומנט שהעברנו. כעת ברור למה argc הינו 2, זאת כיוון שהמערך argv מכיל שני מצביעים לזיכרון. עכשיו יתר הקוד בתוכנית כבר ברור מאליו. התנאי הראשון בודק שאכן יש שני ארגומנטים, אם לא זו שגיאה. בהמשך, הפונקציה atoi מתרגמת את המחרוזת למספר והפונקציה printf מסיימת את העבודה. השימוש ב-atoi היה ממש לא הכרחי בקוד הזה. היינו יכולים לבצע הדפסה באמצעות-

```
printf(“%s\n”, argv[1])
```

בלי להמיר את argv באינדקס 1 למספר. אך חשוב שנכיר את atoi, הוא יסייע לנו בפרק הבא.

שלב ב' – יצירת תוכנית שקוראת לקובץ ההרצה שיצרנו

הפונקציה CreateProcess כוללת לא פחות מ-10 פרמטרים. התיעוד המלא נמצא כמובן ב-MSDN. עם זאת, מבין כל הפרמטרים יש למעשה רק פרמטר אחד שהוא חשוב לנו כרגע, כל יתר הפרמטרים יכולים לקבל ערכי ברירת מחדל. הפרמטר החשוב לנו הוא השני, כיוון שהוא מייצג את "שורת הפקודה" שמריצה את הפרוסס שאנחנו יוצרים. כלומר נרצה ליצור מחרוזת שתכיל את כל מה שייכנס ל-argv בפרוסס החדש שיווצר.

קטע הקוד הבא משמש ליצירת מחרוזת זו. שימו לב לשלבים:

א. הגדרת קובץ ההרצה והערך שנמסר כארגומנט בתור קבועים

ב. חישוב גודל הזיכרון שנצטרך להקצות, שהוא סכום המחרוזות שהגדרנו ועוד 2, לטובת תו רווח ותו סיום מחרוזת Null Termination

ג. הקצאת מערך באמצעות malloc

ד. העתקת הערכים למערך באמצעות sprintf_s

```
#define EXE_FILENAME  "..\\..\\HelloWorld\\Debug\\HelloWorld.exe"
#define PROCESS_ARG   "4"

int main()
{
	// create argument string
	CHAR exe_filename[] = EXE_FILENAME;
	CHAR process_arg[] = PROCESS_ARG;
	INT size = strlen(exe_filename) + strlen(process_arg) + 2;
	size += 2  // space + null termination
	PCHAR param = (PCHAR)malloc(size * sizeof(CHAR));
	sprintf_s(param, size, "%s %s", exe_filename, process_arg);
```

שימו לב שהמחרוזת שיצרנו היא של תווי ASCII ולכן נקרא ל-CreateProcessA, גרסת ה-ASCII, ולא ל-CreateProcessW.

לטובת הקריאה יש סידור נוסף שאנחנו צריכים לבצע. יש צורך לאתחל את שני הפרמטרים האחרונים שמועברים ל-CreateProcessA.

- הערך הראשון הוא משתנה מטיפוס STRATUPINFOA. זהו מבנה נתונים שמכיל את הגדרות החלון שייפתח על ידי ה-Process בזמן הריצה, אם תחפשו עליו ב-MSDN תמצאו את מלוא השדות שהוא מכיל
- הערך השני הוא מבנה נתונים שקרוי Process Information. זהו מבנה פשוט למדי שכולל בסך הכל ארבעה שדות:
  - המספר המזהה של הפרוסס, ה-PID, שנקבע על ידי מערכת ההפעלה בזמן יצירת הפרוסס
  - המספר המזהה של ה-Main Thread, אותו Thread ראשון שמופעל עם יצירת ה-Process.
  - אובייקט Handle ל-Process
  - אובייקט Handle ל-Main Thread

בתהליך יצירת ה-Process אנחנו שולחים לפונקציה CreateProcess שני מבני נתונים מאופסים, שיתמלאו על ידי מערכת ההפעלה לפני שהיא תיצור את ה-Process.

בסיום הריצה, נשתמש בנתונים שנמצאים ב-Process Information על מנת לסגור את ה-Handle ל-Process שיצרנו ול-Main Thread שלו.

להלן כלל הקוד של התרגיל, כולל יצירת מבני הנתונים הנדרשים והאיפוס שלהם. הריצו אותו וודאו שהקובץ HelloWorld נקרא באופן תקין:

```
#define EXE_FILENAME   "..\\..\\HelloWorld\\Debug\\HelloWorld.exe"
#define PROCESS_ARG   "4"

int main()
{
      STARTUPINFOA si;
      PROCESS_INFORMATION pi;
```

```
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create argument string
    CHAR exe_filename[] = EXE_FILENAME;
    CHAR process_arg[] = PROCESS_ARG;
    INT size = strlen(exe_filename) + strlen(process_arg) + 2;
    PCHAR param = (PCHAR)malloc(size * sizeof(CHAR));
    sprintf_s(param, size, "%s %s", exe_filename, process_arg);

    // Start the child process.
    CreateProcessA(NULL,
    param,     // Command line
    NULL,      // Process handle not inheritable
    NULL,      // Thread handle not inheritable
    FALSE,     // Set handle inheritance to FALSE
    0,         // No creation flags
     NULL,         // Use parent's environment block
    NULL,      // Use parent's starting directory
    &si,       // Pointer to STARTUPINFO structure
    &pi);      // Pointer to PROCESS_INFORMATION structure
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
    free(param);
    return 0;
}
```

תרגיל 4.5 **תרגיל מסכם**

בתרגיל זה נחבר את הידע שצברנו, ועל הדרך נמחיש את אופן תזמון הThreadים של ה-Scheduler.

כיתבו תוכנית שיוצרת 4 Threadים חדשים, כל Thread רץ בלולאה 1000 פעם, ובכל פעם מדפיס למסך את המספר הסידורי של הThread ואת מונה הלולאה.

שימו לב, שבתכנית הזו ישנם שני אתגרים. הראשון- להעביר ל-Thread מספר סידורי. היזהרו ממצב שבו אתם מעבירים ל-Threadים מצביע לכתובת בזיכרון שהינה אותה כתובת לכל ה-Threadים. לדוגמה, אם יש לכם משתנה i ואתם מעבירים בכל פעם את הכתובת של i, כל ה-Threadים ייגשו לאותה הכתובת בזיכרון. כאשר ערכו של i ישתנה, הערך ישתנה עבור כל ה-Threadים.

האתגר השני, הוא שהפעם צריך לחכות לסיום הריצה לא של Thread יחיד אלא של מספר Threadים. קיראו ב-MSDN על הפונקציה WaitForMultipleObjects.

דוגמה לקטע מתוצאת ריצה:

```
Thread 1, var = 0
Thread 3, var = 0
Thread 3, var = 1
Thread 1, var = 1
Thread 2, var = 0
Thread 3, var = 2
Thread 4, var = 0
Thread 2, var = 1
Thread 3, var = 3
Thread 1, var = 2
Thread 2, var = 2
Thread 4, var = 1
Thread 3, var = 4
```

4.8 סיכום

בפרק זה למדנו שכל תוכנה שרצה במחשב נמצאת בתוך Process, שמעניק לה את כל המשאבים שהיא זקוקה להם. ראינו ש-Process יחיד יכול לחלק משימות למספר Threadים, שחולקים בתוכו משאבים משותפים. כדי להמחיש לעצמנו את הדברים, יצרנו בעצמנו תוכניות שפותחות מספר Threadים וחקרנו דרך Process Explorer את החלוקה ל-Processים ו-Threadים. בפרק הבא נדון בבעיות שנגרמות כתוצאה מחלוקה זו, וכיצד מתגברים עליהן.

פרק 5 – סנכרון Processים ו-Threadים

בפרק זה נדון בבעיות שונות שעלולות להתרחש כתוצאה מהרצה בו זמנית של מספר Threadים, בין אם הם שייכים לאותו Process או ל-Processים שונים. נסקור מנגנונים שונים שפותחו על מנת לפתור בעיות אלו, נלמד על מנגנוני הסנכרון בין Processים ו-Threadים, מנגנונים שקרויים Mutexes ו-Critical Sections. לבסוף נפתור בעיה קלאסית בתחום מדעי המחשב- בעיית הפילוסופים הסועדים.

5.1 מהו Race Condition

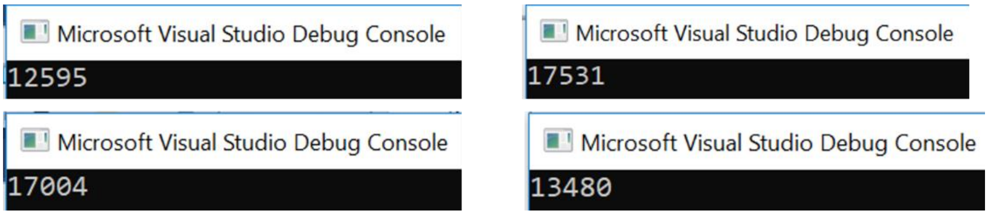
המושג הראשון אותו נלמד הוא Race Condition. כדי להבין אותו, נבצע ניסוי קטן. לפנינו קוד, שפותח שני Threadים ומעביר לכך אחד מהם כתובת של אותו משתנה. כל אחד מה-Threadים מקדם את ערכו של המשתנה 10000 פעם. עיברו על הקוד הבא וענו על השאלה הבאה (בלי להריץ את הקוד): מה יהיה ערכו של המשתנה param בסיום הריצה?

```
#define NUM_THREADS 2
#define MAX 10000

VOID inc(PINT param) {
    INT temp = *param;
    temp++;
    *param = temp;
}

DWORD WINAPI thread_func(LPVOID param) {
    for (INT i = 0; i < MAX; i++) {
        inc((PINT)param);
    }
    return 1;
}

int main()
{
    INT val = 0;
    LPVOID param = &val;
```

```
    HANDLE hThread[NUM_THREADS];
    for (INT i = 0; i < NUM_THREADS; i++) {
        hThread[i] = CreateThread(
            NULL,       //default security attributes
            0,          //default stack size
            thread_func,//thread function
            param,      //thread param
            0,          //default creation flags
            NULL        //return thread identifier
        );
    }

    WaitForMultipleObjects(NUM_THREADS, hThread, TRUE, INFINITE);
    printf("Param value: %d\n", *(PINT)param);
}
```

באופן טבעי, היינו מצפים שבסיום הריצה ערכו של param יהיה שווה ל-20000. כעת הריצו את התוכנית מספר פעמים- מה קבלתם?

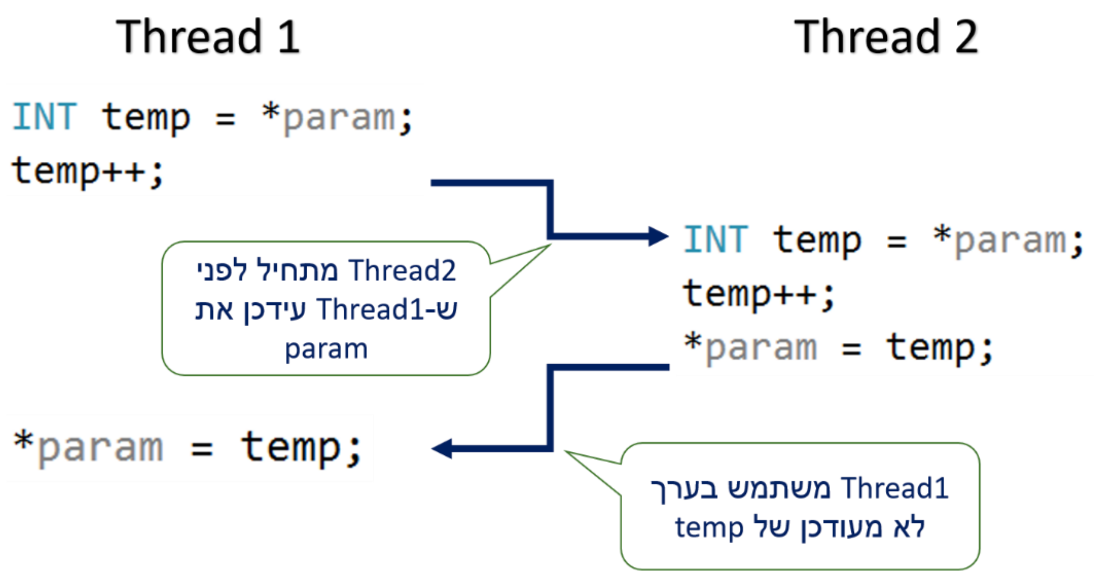
הנה תוצאות של ארבע הרצות- באף אחת מהן התוצאה אינה קרובה לתוצאה הצפויה.



לא רק שהתוצאות אינן 20000, יש גם שונות רבה מאד בין התוצאות. מחשבים לא אמורים לתת תוצאה שונה בכל פעם שמריצים את אותה תוכנה. מה מתרחש כאן?

לשני ה-Threadים יש את אותה כתובת של param, כלומר שניהם קוראים וכותבים לאותה כתובת בזיכרון. הבעיות מתרחשות כאשר מתבצע Context Switch בזמן לא טוב. (תזכורת: Context Switch הוא ארוע שמתוזמן על ידי ה-Scheduler של מערכת ההפעלה, ובעקבותיו המעבד עובר מהרצה של קוד של Thread כלשהו להרצה של קוד של Thread אחר)

האיור הבא ממחיש מהו זמן <u>לא טוב</u> לביצוע Context Switch. שימו לב מתי הוא מתבצע- בדיוק אחרי ש-Thread1 העלה את ערכו של temp, ולפני שהוא העתיק את temp העדכני לתוך param.



ניקח דוגמה, בה param שווה 5. במקרה זה, Thread1 קורא את הערך מהזיכרון, ומחשב את temp להיות 6. ערכו של temp עדיין לא נשמר בחזרה לתוך param.

ברגע זה מתבצע Context Switch אל Thread2, שגם הוא קורא את param, שעדיין ערכו הוא 5. לכן Thread2 יעדכן את ערכו של param להיות 6.

כאשר Thread1 יחזור, הוא ישמור ב-param את temp. המשתנה temp הוא משתנה מקומי, כלומר ל-Thread1 יש משתנה temp שונה מאשר ל-Thread2. ערכו של temp שבתוך Thread1 הוא 6. לכן Thread1 ישמור 6 לתוך param.

התוצאה תהיה שלמרות ש-param התעדכן על ידי שני ה-Thread-ים, מה שלכאורה היה צריך להעלות את ערכו ל-7, למעשה ערכו נותר 6.

התופעה שסקרנו כרגע נקראת Race Condition. ישנה "תחרות" בין שני ה-Thread-ים מי יעדכן את ערכו של param, והתוצאה היא שהם מפריעים זה לזה לבצע את המשימה.

קטע קוד נקרא "קריטי", או באנגלית Critical Section, אם אסור שבמהלכו יבוצע Context Switch. כפי שאנחנו רואים, קטע הקוד שסקרנו הוא קטע קוד קריטי.

שאלה למחשבה:

התבוננו בשורת קוד הבאה. שורת הקוד מעדכנת את ערכו של המשתנה ש-param מצביע עליו ועושה זאת בפקודה אחת בלבד. מה דעתכם, האם ביצוע העדכון באופן הזה ימנע את ה-Race Condition, כך שלא תתבצע החלפה בין Thread-ים בין פקודת ההעתקה לפקודת העדכון והשמירה חזרה של הערך העדכני?

```
*param += 1;
```

ובכן, התשובה היא לא, למרות שכעת פקודת העדכון נכתבת בשורה אחת הבעיה לא נפתרה.

כדי להבין את הסיבה לכך, נבדוק איך הפקודה הזו מתורגמת לקוד אסמבלי, קל לעשות זאת באמצעות טאב ה-Disassembler של Visual Studio, אשר מופיע בכל הרצה במצב Debug:

```
*param += 1;
00DF18DF  mov       eax,dword ptr [param]
00DF18E2  mov       ecx,dword ptr [eax]
00DF18E4  add       ecx,1
00DF18E7  mov       edx,dword ptr [param]
00DF18EA  mov       dword ptr [edx],ecx
```

מסתבר כי הפקודה הבודדת שכתבנו בשפת C מתורגמת ל-5 פקודות אסמבלי שונות. העמודה השמאלית היא הכתובת בזיכרון שבה נמצאת כל פקודה, הכתובות עשויות להיות שונות מהכתובות בדוגמה. הפקודה הראשונה מעתיקה את ערכו של param לתוך רגיסטר eax, כך ש-eax שווה לכתובת ש-param מצביע עליה. הפקודה השניה מעתיקה את הערך שנמצא בכתובת של eax לתוך רגיסטר אחר, ecx. כך נראה קוד אסמבלי של עבודה עם מצביעים. לאחר מכן מתבצע עדכון של ecx והשמה שלו באמצעות שתי פקודות נוספות אל הכתובת ש-param מצביע עליה.

כפי שאנחנו רואים, גם אם קוד כתוב בפקודה אחת בשפה עילית, אין הדבר אומר שהוא יתורגם לפקודה בודדת בשפת אסמבלי. לכן, עלול להתבצע Context Switch במהלך ריצה של פקודה בודדת בשפה עילית.

5.2 מהו Lock

בתוכנית שלנו, ישנו קטע קוד קריטי:

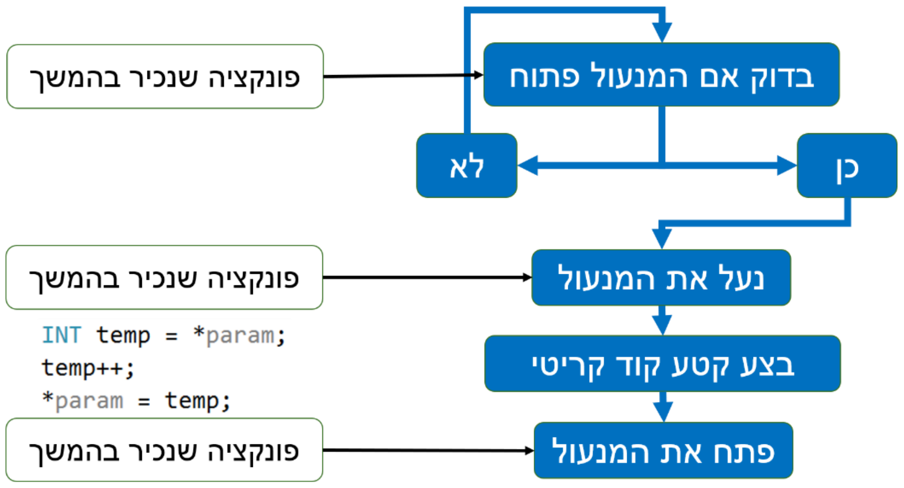
```
INT temp = *param;
temp++;
*param = temp;
```

הפתרון הוא אובייקט שנקרא מנעול, Lock.

כזכור, אובייקט הוא מבנה נתונים שכולל משתנים ופונקציות שפועלות על המשתנים. לאובייקט מסוג Lock יוגדרו:

- משתנה "מצב מנעול"- 0 אם המנעול סגור, 1 אם המנעול פתוח
- פונקציות שבודקות את מצב המנעול
- פונקציות שמשנות את מצב המנעול

תרשים הזרימה הבא מתאר את אופן הפעולה של Lock. שימו לב, שאת השמות של הפונקציות שמבצעות את הבדיקות השונות, אנחנו עדיין לא מכירים. כרגע אנחנו רק מתארים באופן כללי איך Lock מופעל לצד קטע קוד קריטי.



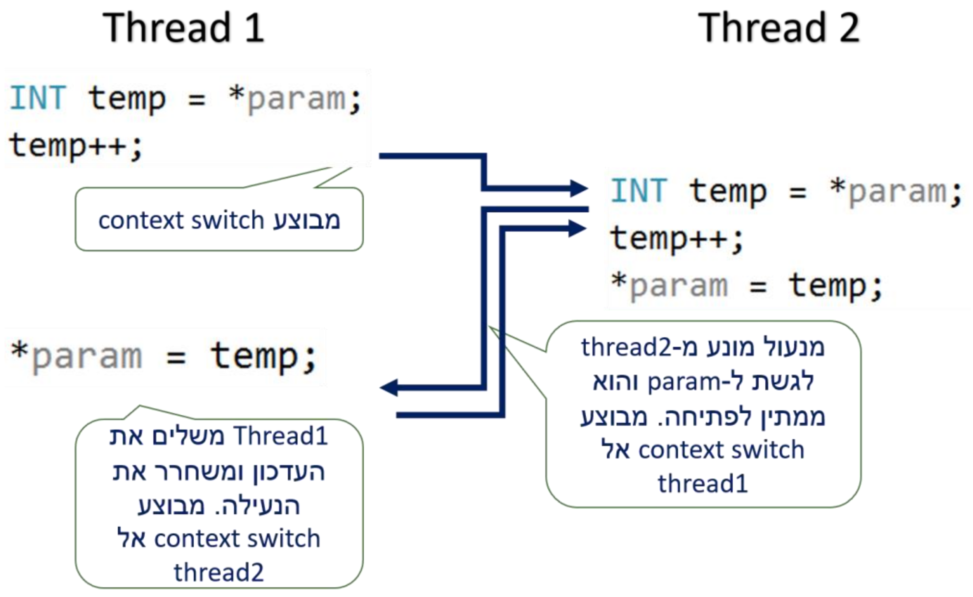
Thread שמגיע לקטע קוד קריטי יצטרך קודם כל לבדוק אם המנעול פתוח. אם לא, הוא יצטרך להמתין לפתיחת המנעול. יכול מאד להיות שתוך כדי ההמתנה, יבוצע Context Switch ל-Thread אחר, שביצע את הנעילה.

אם המנעול פתוח, לפני הכל נועלים אותו. המטרה היא למנוע מ-Thread אחר לנעול את המנעול לפנינו. פעולת נעילת המנעול היא פקודה אחת בשפת מכונה, כך שלא יכול להיות שהמעבד יבצע Context Switch במהלכה. אם להשתמש במונח שלמדנו זה עתה, נעילת מנעול היא פעולה אטומית.

לאחר שהנעילה בוצעה, ניתן לגשת בשקט לביצוע קטע הקוד הקריטי. במקרה הכי גרוע, בו יבוצע Context Switch ל-Thread אחר, ה-Thread האחר ייתקל בנעילה ויישאר על המנעול עד שמשאבי המעבד יחזרו אל ה-Thread שנמצא בתוך קטע הקוד הקריטי.

לבסוף, עם סיום קטע הקוד הקריטי, ה-Thread משחרר את המנעול ובכך הוא מאפשר בנדיבות גם ל-Threadים אחרים לבצע את קטע הקוד הקריטי.

האיור הבא ממחיש איך מנגנון ה-Lock פותר את הבעיה של Context Switch תוך כדי ביצוע קטע קוד קריטי.



כעת, לאחר שהבנו את התפקיד של מנעולים, נסקור שני סוגים של מנעולים: CriticalSection ו- Mutex. לכל סוג של מנעול יש שימוש משלו ופונקציות מיוחדות לו במערכת ההפעלה.

5.3 מנעול מסוג CriticalSection

המנעול CriticalSection משמש לסנכרון בין Threadים בתוך אותו Process. שימו לב, שהשם של המנעול זהה למושג ה-Critical Section, קטע (קוד) קריטי. לכן כדי להבחין ביניהם נקרא למנעול CriticalSection, ללא רווח בין המילים.

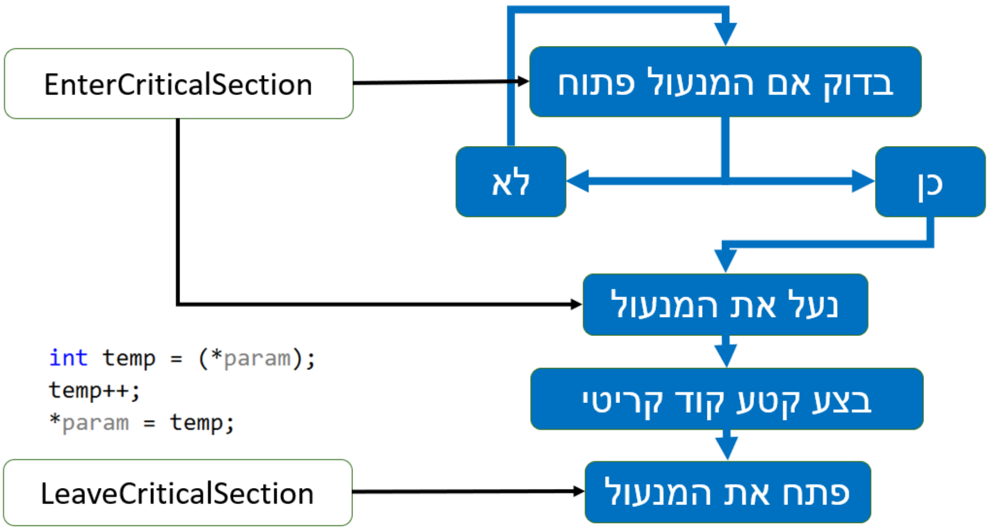
אנחנו צריכים ליצור אובייקט תוכנה כלשהו שכל ה-Threadים יכירו. לפני שתמשיכו לקרוא עיצרו וחישבו- איך הייתם מייצרים אובייקט שכל ה-Threadים יכירו?

אחד הדברים הבסיסיים שלמדנו על Threadים של אותו Process, הוא שיש להם זיכרון משותף. לכן, הפתרון קל: את מצב המנעול נשמור בתוך משתנה גלובלי. משתנה גלובלי מוכר על ידי כל ה-Threadים.

להלן הסבר השלבים של יצירת מנעול CriticalSection ושימוש בו.

1. נגדיר משתנה גלובלי מסוג CRITICAL_SECTION. ה-type שנקרא CRITICAL_SECTION מוגדר על ידי Windows ולכן כל מה שאנחנו נדרשים לעשות כדי שה-type יהיה מוכר זה לבצע include ל-Windows.h.
2. מתוך main, נקרא לפונקציה InitializeCriticalSection. כעת המנעול שלנו מוכן לשימוש.
3. כאשר Thread כלשהו רוצה לקחת בעלות על ה-CriticalSection, הוא יכול לעשות זאת רק אם המנעול אינו נעול. קיימות שתי פונקציות שמבצעות זאת, כל אחת שימושית למקרה אחר:
   a. EnterCriticalSection- פונקציה זו ממתינה עד שהמנעול פתוח, ואז תופסת עליו בעלות. במילים אחרות, אם המנעול סגור, ה-Thread שלנו ימתין ויחכה, גם אם זה לנצח.
   b. TryEnterCriticalSection – פונקציה זו תופסת בעלות על מנעול פתוח. בניגוד לפונקציה הקודמת, אם המנעול סגור הפונקציה לא ממתינה אלא מחזירה True / False. כלומר ה-Thread אינו ממתין למנעול והוא יכול להמשיך לבצע פעולות אחרות.
4. כאשר ה-Thread סיים להשתמש במנעול, הוא יבצע LeaveCriticalSection, כדי לשחרר את המנעול לThreadים אחרים שאולי זקוקים לו.
5. לאחר שכל ה-Threadים סיימו את עבודתם (ראינו כבר שיש פונקציה שממתינה להם), פונקציית ה-main מוחקת את המנעול ומשחררת את המשאבים הקשורים אליו, באמצעות הפונקציה DeleteCriticalSection.

כעת כשאנחנו מכירים את הפונקציות הללו, אפשר לעדכן את תרשים הזרימה של מנעול מסוג CriticalSection, שמגן על קטע הקוד הקריטי שניתן כדוגמה מוקדם יותר:



להלן שלד של קוד שמבצע את שלבי העבודה עם critical section. זה אינו קוד שמתקמפל בצורה תקינה, אך כולל את המבנה הכללי שלתוכו צריך להוסיף את הקוד הקריטי:

```
CRITICAL_SECTION ghCriticalSection;

int main(void)
  {
      InitializeCriticalSection(&ghCriticalSection)

      ...

      DeleteCriticalSection(&ghCriticalSection);
  }

DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    EnterCriticalSection(&ghCriticalSection);

    // Place critical code here
    ...

    LeaveCriticalSection(&ghCriticalSection);

    return 1;
}
```

שימו לב לכך שבתוכנית הזו חסר כל מנגנון ניהול ה-Threadים. שלד התוכנית לא יוצר Threadים ולא ממתין לסיום ריצת ה-Threadים לפני שהוא מסיים את הריצה של main. נושאים אלו נלמדו מוקדם יותר, ואם אתם זקוקים לדוגמאות הקוד שלהם דפדפו מעט אחורה.

5.4 מצב Deadlock

דמיינו צומת שבה יש אור ירוק בכל הכיוונים. איך תראה זרימת התנועה בצומת? סביר שכפי שנראית התנועה בתמונה הבאה- הרכבים חוסמים זה את זה כך שאף נתיב אינו מקבל זרימה חופשית.

מקור: http://www.livingreal.com.au

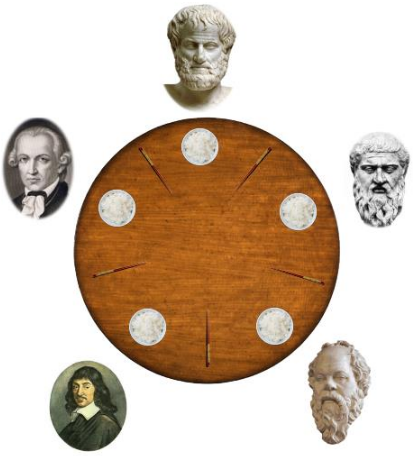
תופעה דומה עלולה להתרחש גם בעולם המחשבים, בעקבות תכנון לא נכון. נניח שיש לנו שני Threadים ושני משאבים, וכל Thread מקבל פקודה לתפוס את שני המשאבים, לבצע פעולה מסויימת ואז לשחרר את המשאבים לרשות ה-Thread השני. הבעיה תיגרם כאשר Thread אחד יתפוס משאב אחד ואילו ה-Thread השני יתפוס את המשאב השני. בשלב זה כל אחד מה-Threadים ממתין לשחרור המשאב השני, ולכן גם אינו משחרר את המשאב שברשותו. אין פתרון למצב זה, הוא יכול להמשך לנצח. לכן התופעה נקראת- גם בעולם המחשבים וגם בעולם האמיתי- בשם Deadlock.

5.5 מצב Starvation

נדמיין שלצומת העמוס שלנו הגיע שוטר תנועה. כדי למנוע מצב של Deadlock, השוטר עוצר לחלוטין את זרימת התנועה משלושה כיוונים ומאפשר רק למכוניות שמגיעות לצומת מכיוון אחד להמשיך באופן חופשי. המפקד של שוטר התנועה שואל אותו בקשר מה הסטטוס, והשוטר עונה "התנועה זורמת". אכן, התנועה זורמת למי שהתמזל מזלו להגיע מהכיוון הנכון, אולם כל מי שהגיע לצומת משלושת הכיוונים החסומים יכול להמשיך לחכות… לתופעה זו קוראים Starvation, הרעבה. בניגוד ל-Deadlock, כאן יש מי שמקבל את המשאבים, אולם יש כאלה שלעולם לא יזכו לקבל אותם.

תרגיל 5.1 **תרגיל מסכם** CriticalSection

בעיית הפילוסופים הסועדים היא בעיה קלאסית מתחום מדעי המחשב. חמישה פילוסופים יושבים ליד שולחן, במרכז השולחן יש קערת אורז וליד כל פילוסוף יש מקל אכילה אחד בלבד. כמובן שכל סועד נזקק לשני מקלות אכילה כדי לאכול.



עליכם לכתוב תוכנה שתמצא דרך שבה כל פילוסוף יזכה לזמן אכילה. כאשר פילוסוף כלשהו אוכל, התוכנה תדפיס הודעה על מספר הפילוסוף שאוכל, בסיום האכילה הפילוסוף ישחרר את מקלות האכילה לסועד הבא.
היזהרו מ-Deadlocks! אילו כל פילוסוף יחזיק במקל אכילה אחד בלבד, כל הפילוסופים יישארו רעבים.
מאידך, היזהרו גם מ-Starvation. אילו שני פילוסופים יחזיקו בשני מקלות אכילה כל אחד, הם יאכלו לשובע אך כל יתר הפילוסופים ירעבו.

לאחר שוידאתם שהתוכנית רצה באופן שבו כל הפילוסופים אוכלים, בצעו מדידת זמנים- תנו לכל פילוסוף לאכול מיליון פעם, ומדדו כמה זמן לוקח עד שכל הפילוסופים מסיימים את האכילה. לטובת מדידת הזמנים, הסירו את קטע הקוד שמדפיס איזה פילוסוף אוכל בכל רגע.

הדרכה לפתרון

מומלץ להתבסס על דוגמת הקוד של שימוש ב-CriticalSection. עליכם להוסיף שלושה חלקים:

- טיפול ביצירת Threadים, באמצעות CreateThread. כל פילוסוף צריך לדעת ליד אילו מקלות אכילה הוא יושב, כלומר כל Thread צריך לדעת אילו שני CriticalSections עליו לתפוס. מבחינה תכנותית, ניתן לבצע זאת באמצעות העברת פרמטר דרך CreateThread, שמציין את המספר של ה-Thread.
- כתיבת הפונקציה שמריצים ה-Threadים כאשר הם נוצרים. הפונקציה תכיל את הלוגיקה של שימוש ב-Critical Section. כל Thread ישתמש בפרמטר שהועבר אליו כדי לדעת אילו CriticalSections עליו לתפוס. כדי למנוע מצב של Deadlock או Starvation, השתמשו בתבונה ב-EnterCriticalSection בצירוף TryEnterCriticalSection. תוכלו להסתייע בשלד הקוד הבא, שמבצע TryEnterCriticalSection, ולהתאים אותו לצרכיכם:

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    BOOL success =
    TryEnterCriticalSection(&ghCriticalSection);
    if success
    {
        // Place critical code here
        ...

        LeaveCriticalSection(&ghCriticalSection);
    }
    return 1;
}
```

- המתנה לסיום ריצת כל ה-Threadים לפני סיום ריצת התכנית, באמצעות WaitForMultipleObjects.

5.6 מנעול מסוג Mutex

המנעול Mutex משמש לסנכרון בין Threadים שנמצאים ב-Processים שונים. כאשר דנו ב- Multi-Processing לעומת Multi-Threading, ראינו דוגמאות לכך שתוכנות פותחות Processים רבים לטובת ביצוע המשימות שלהן.

לעיתים קרובות Processים צריכים לתקשר זה עם זה, בפרט כאשר מדובר ב-Processים שונים של אותה התוכנה. לדוגמה, Processים שכרום פתח לטובת הרצת תוספים צריכים לתקשר עם Processים שנוצרו לטובת טאבים שהמשתמש פתח בדפדפן. דוגמה נוספת, ה-Process שאחראי על הממשק הגרפי של ספוטיפיי צריך לתקשר עם ה-Process שאחראי על השמעת השירים, כדי להעביר פקודות כמו "עצור" ו"נגן".

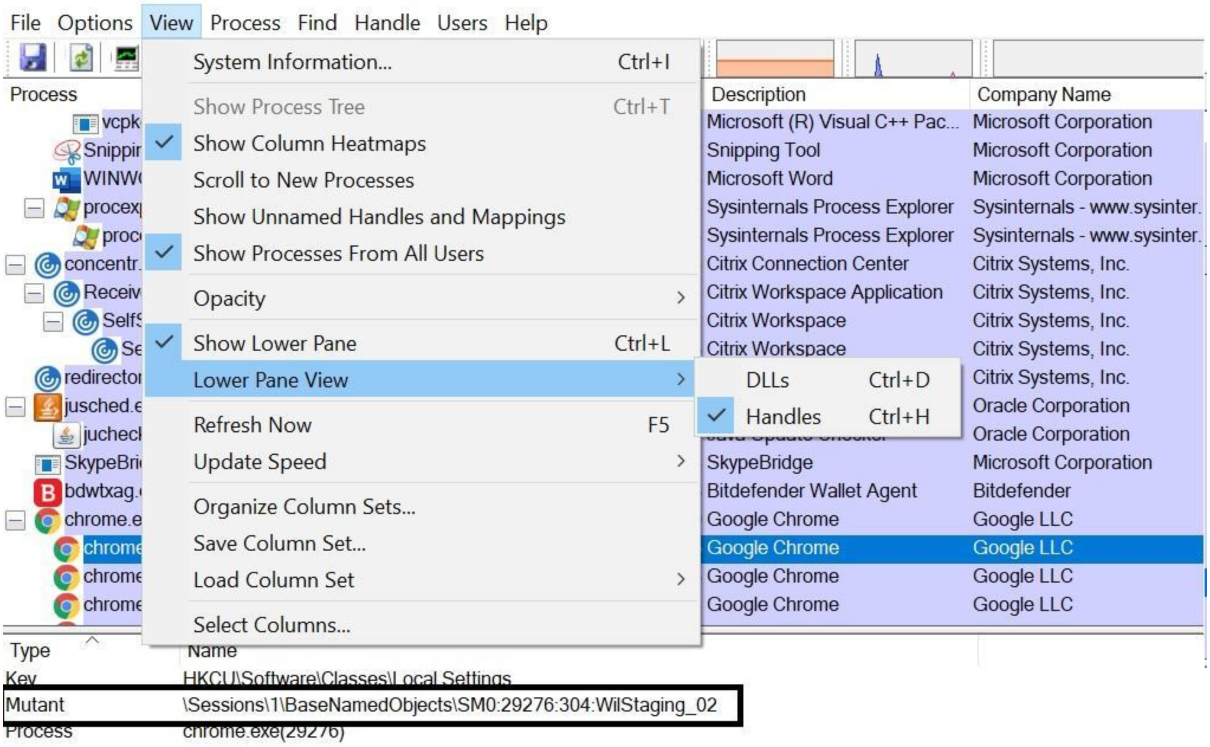
לכן, הכרחי של-Processים שונים יהיו דרכי תקשורת זה עם זה. אחת מדרכי התקשורת הנפוצות בין Processים היא שימוש במשאב משותף, כגון זיכרון. נלמד בהמשך אודות זיכרון משותף, כרגע כל מה שצריך לדעת הוא שאפשר לגרום לאזור זיכרון להיות "שייך" ליותר מ-Process אחד. כמובן, שפעולות הכתיבה והקריאה לזיכרון המשותף צריכות להיות מתואמות, אחרת עלולות להיווצר בעיות קשות, כפי שראינו כשסקרנו Race Conditions. איכשהו, כל Process צריך לקבל בעלות על הזיכרון המשותף בזמן שיתר ה-Processים לא יכולים לגשת אל הזיכרון המשותף. וכדי לבצע את הסינכרון הזה, CriticalSection כבר אינו מתאים. חישבו- מדוע?

התשובה היא, שמשתנים גלובליים אינם מוכרים מחוץ ל-Process שבו הם מוגדרים. כשמערכת ההפעלה מבצעת Context Switch בין Processים, כל המשתנים הגלובליים שהיו שייכים ל-Process הקודם פשוט אינם קיימים יותר מבחינת ה-Process החדש. אם כך, אנחנו צריכים למצוא פתרון אחר.

הפתרון הוא שימוש באובייקטים של מערכת ההפעלה. כזכור, ב-Windows מוגדר טיפוס מסוג Object. כל האובייקטים של Windows מנוהלים על ידי ה- Object Manager. אפשר לפנות אליו ולבקש Handle לאובייקט כלשהו, ואם ה-Object Manager ימצא שיש למבקש הרשאה מתאימה, הוא יעניק לו Handle לאובייקט המבוקש.

אם כך, המתכנת מגדיר אובייקט מסוג Mutex וקובע את השם שלו (בקרוב נבצע זאת בעצמנו) ואילו מערכות ההפעלה שומרות את האובייקט בתוכן ומספקות Handle שמאפשר גישה אליו. אם כך, Mutex ו-CriticalSection דומים בכך ששניהם מבצעים את אותה פונקציונליות של סינכרון, אך בעוד ש-CriticalSection נמצא בזיכרון של Process כלשהו, Mutex הוא אובייקט של מערכת ההפעלה ומי שמשתמש בו מקבל רק Handle אליו.

באמצעות Process Explorer ניתן לצפות בשמות ה-Mutexים שProcessים מסויים משתמש בהם. בחרו את טאב View, ובתוכו אפשרו את "Show Lower Pane". לאחר מכן וודאו שמה שמוצג הוא ה-Handles:



למטה, מוקף לצורך ההמחשה במסגרת שחורה, ניתן למצוא את שם ה-Mutex שה-Process של כרום משתמש בו. מסיבות היסטוריות, הקרנל של Windows קורא Mutant ל-Mutex, זה אותו דבר.

5.6.1 שימוש ב-Mutex

השימוש ב-Mutex דומה למדי לשימוש ב-CriticalSection, אך שמות הפונקציות והשימוש בהם שונה מעט. להלן שלבי העבודה עם Mutex.

שלב א' – <u>CreateMutex</u>

יצירת Mutex מתבצעת על ידי CreateMutex. הפונקציה מקבלת שם קבוע, שיהיה השם של ה-Mutex. השם הזה יכול להיות כל שם שהמתכנת בוחר. כעת יש שתי אפשרויות. האפשרות הראשונה, אם ה-Mutex הזה אינו קיים, הוא נוצר. היכן הוא נוצר? כפי שאמרנו, כל ה-Mutexים הם משאבים של מערכת ההפעלה. זה הכרחי כדי להבטיח שה-Mutexים יהיו מוכרים על ידי Threadים של Processים שונים, שאינם חולקים זיכרון משותף. האפשרות השניה, אם ה-Mutex הזה כבר קיים, הוא לא יווצר שוב אלא רק יתקבל אליו Handle, שיאפשר לעשות עליו פעולות שונות כמו לבדוק מה המצב שלו והאם הוא תפוס.

דוגמה לקריאה ל-CreateMutex:

```
HANDLE hMyMutex = CreateMutexA(
    NULL,       // default security attributes
    FALSE,      // initially not owned
    "Hello");   // named mutex
```

שימו לב שהקוד הזה צריך להיות בשימוש גם ב-Process האב, שיוצר את ה-Mutex, וגם בכל ה-Processים הבנים, שצריכים להשיג אליו Handle.

שלב ב' - <u>WaitForSingleObject</u>

שלב זה מתרחש רק ב-Processים הבנים. כדי להשיג בעלות על ה-Mutex, משתמשים ב-WaitForSingleObject. הפונקציה הזו מקבלת את השם של ה-Mutex המבוקש ומנסה להשיג עליו בעלות.

הפונקציה מקבלת גם פרק זמן מקסימלי כדי לנסות להשיג את ה-Mutex. אם פרק הזמן הוא INFINITE, אז הפונקציה תמתין כמה זמן שצריך עד שה-Mutex יהיה פנוי (בדומה ל-EnterCriticalSection, שמחכה כמה זמן שצריך ובינתיים התוכנית עוצרת).

לעומת זאת, אם פרק הזמן המקסימלי נקבע ל-0, אז במידה וה-Mutex אינו זמין הפונקציה תמשיך הלאה בלי המתנה (בדומה ל-TryEnterCriticalSection).

דוגמאות לקריאה ל-WaitForSingleObject:

```
DWORD waitResult = WaitForSingleObject(
    hMyMutex,
    INFINITE);  // no time-out interval
```

```
DWORD waitResult = WaitForSingleObject(
    hMyMutex,
    0);          // zero time-out interval
```

בנוסף על הבדיקה האם ה-Mutex פנוי, הפונקציה גם תופסת אותו אם הוא פנוי, לכן אין צורך להשתמש בפונקציה נוספת לתפיסת ה-Mutex.

הפונקציה מחזירה אחד מתוך ארבעה ערכים המוגדרים בתיעוד ב-MSDN (חפשו בגוגל msdn waitforsingleobject, היכנסו לתיעוד וקיראו עליהם). הערך שמעניין אותנו הוא WAIT_OBJECT_0 שהמשמעות שלו היא שה-Mutex היה פנוי וכעת הוא ברשותנו.

קוד דוגמה לבדיקת ערך החזרה:

```
if (waitResult != WAIT_OBJECT_0) {
   // Mutex not free
   Bad();
}
else {
   // Mutex free
   Good();
```

שלב ג' – ReleaseMutex

Process שסיים את השימוש ב-Mutex ומבקש לשחרר אותו לשימוש Processים אחרים, משתמש ב-ReleaseMutex. זוהי פונקציה פשוטה ביותר שמקבלת רק Handle ל-Mutex המבוקש.

דוגמת קוד:

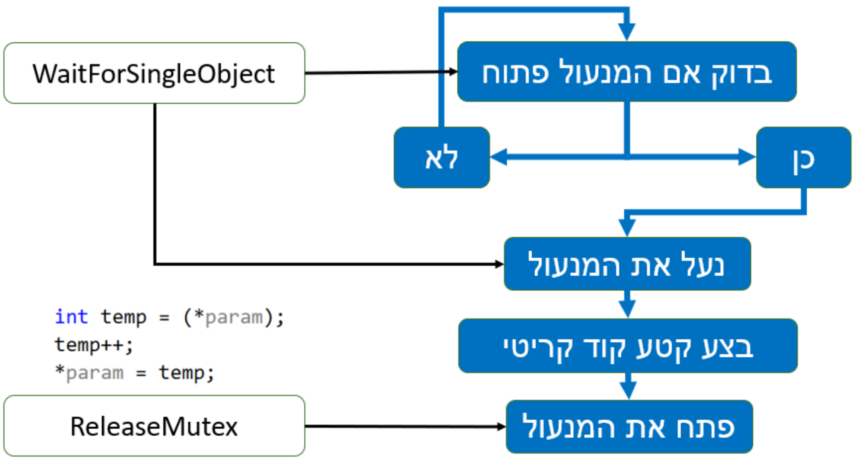
```
ReleaseMutex(hMyMutex);
```

שלב ד' -CloseHandle

בסיום השימוש ב-Mutex מומלץ לסגור את ה-Handle שעושה בו שימוש

```
CloseHandle(hMyMutex);
```

כעת כשאנחנו מכירים את הפונקציות הללו, אפשר לעדכן את תרשים הזרימה גם של מנעול מסוג Mutex, שמגן על קטע הקוד הקריטי שניתן כדוגמה מוקדם יותר:



תרגיל 5.2 תרגיל ביניים Mutexים

בתרגיל זה ניצור תוכנה בשם HelloWorldMutex שמדפיסה את המספר שהיא קיבלה כארגומנט משורת הפקודה. אבל, התוכנה לא מדפיסה מיד את המספר, אלא מחכה בנימוס כדי לוודא ששום תוכנה אחרת לא מבצעת את אותה פעולה כרגע.

בתרגיל ניצור גם תוכנה בשם MutexParentProcess. תוכנה זו תיצור שני Processים שיריצו במקביל את HelloWorldMutex. אנחנו רוצים לראות הדפסה "מנומסת" שבה Process מדפיס רק אם הוא היחיד שמדפיס כרגע, ואם לא הוא ממתין.

הרצה תקינה של MutexParentProcess תוצג למסך באופן הבא:

Hello World! Process got argument 0

(המתנה של 5 שניות)

Hello World! Process got argument 1

(המתנה של 5 שניות)

סיום ריצה

מיד נעבור לפתרון מודרך של הקוד, אך יש לכם כבר כעת את כל הכלים לפתור בעצמכם. נסו זאת לפני שתעברו אל הפתרון המודרך.

כבסיס לקוד של HelloWorldMutex נשתמש בקוד של HelloWorld שפיתחנו בפרק הקודם, התוכנה שמדפיסה למסך את הפרמטר שהיא מקבלת מה-Process האב.

אנחנו צריכים להוסיף בפתיחה CreateMutex, כדי לקבל Handle. לאחר מכן נבצע בדיקה פשוטה אם ה-Mutex פנוי. אם כן- נבצע הדפסה ונשחרר את ה-Mutex.

```
#include "pch.h"

int main(int argc, char *argv[])
{
	if (argc != 2) {
		printf("Incorrect number of arguments\n");
		return 0;
	}
	int id = atoi(argv[1]);
	HANDLE hMyMutex = CreateMutexA(NULL, FALSE, "Hello");
	DWORD waitResult = WaitForSingleObject(hMyMutex, 10000);
	if (waitResult == WAIT_OBJECT_0)
	{
		printf("Hello World! Process got parameter %d\n", id);
		Sleep(5000);
		ReleaseMutex(hMyMutex);
	}
	else
		printf("Mutex could not be obtained for too long\n");
	CloseHandle(hMyMutex);
	return 1;
}
```

איך צריך להראות הקוד של MutexParentPrcoess?

בפרק הקודם ראינו איך נראה קוד שיוצר Process ומעביר אליו פרמטר. השדרוגים שאנחנו צריכים לבצע הם שניים:

- להגדיר Mutex
- ליצור שני Processים, לא אחד כמו בפרק הקודם

ההגדרה של Mutex היא בדיוק כפי שראינו בדוגמת הקוד למעלה. כדי ליצור שני Processים פשוט ניצור לולאה שבכל איטרציה תיצור מחרוזת חדשה שתועבר כשורת פקודה. בנוסף, הפרמטר pi שאנחנו מעבירים ל-CreateProcess צריך להפוך למערך, כדי שתהיה לנו גישה ל-Handleים של כל ה-Processים שיצרנו. אנחנו חייבים לשמור גישה ל-Handleים כדי לנוכל לקרוא ל-WaitForSingleObject ולהבטיח שריצת ה-Processים הבנים הסתיימה לפני שנסיים ריצה.

לכן הקוד לאחר השינוי נראה כך:

```
#include "pch.h"
#define EXE_FILENAME    "..\\HelloWorldMutex.exe"
#define NUM_PROCESSES   2


int main()
{
	HANDLE hMyMutex = CreateMutexA(NULL, FALSE, "Hello");
	INT size = strlen(EXE_FILENAME) + 3; // length is increased by 3:
	                                     // space character- 1 byte,
	                                     // one digit number- 1 byte,
	                                     // string NULL terminator- 1 byte
	PCHAR param = (PCHAR)malloc(size*sizeof(CHAR));
	STARTUPINFOA si;
	PROCESS_INFORMATION pi[NUM_PROCESSES];

	for (int i = 0; i < NUM_PROCESSES; i++) {
		sprintf_s(param, size, "%s %d", EXE_FILENAME, i);
		ZeroMemory(&si, sizeof(si));
		si.cb = sizeof(si);
		ZeroMemory(&pi[i], sizeof(pi[i]));
		CreateProcessA(
			NULL,
			param,
			NULL,
			NULL,
			FALSE,
			0,
			NULL,
			NULL,
			&si,
			&pi[i]);
	}
	// Wait for every process to finish, close everything before return
```

```
	for (int i = 0; i < NUM_PROCESSES; i++) {
		WaitForSingleObject(pi[i].hProcess, INFINITE);
	}

	ReleaseMutex(hMyMutex);

	for (int i = 0; i < NUM_PROCESSES; i++)
	{
		CloseHandle(pi[i].hProcess);
		CloseHandle(pi[i].hThread);
	}

	return 0;
}
```

5.6.2 הרחבה: שימוש ב-Mutexים בנוזקות

אחת הבעיות שיש למפתחי נוזקות היא כיצד למנוע מצב שבו הנוזקה מותקנת יותר מפעם אחת במחשב המותקף. כאשר מחשב כבר נגוע בנוזקה, ביצוע התקנה נוספת של הנוזקה הוא לא רק מיותר אלא גם בעייתי עבור הנוזקה, מכיוון שתהליך ההתקנה עלול לחשוף אותה, או להפריע לעבודה שלה.

הפתרון שכותבי נוזקות רבים בוחרים הוא שימוש ב-Mutex. כל התקנה של הנוזקה בודקת אם קיים Mutex מסויים ואם הוא אינו קיים היא מתקינה את עצמה ומגדירה אותו. אם הוא קיים- הנוזקה לא תתקין את עצמה כלל.

עם הזמן, למדו מערכות ההגנה שבאמצעות ה-Mutexים ניתן לזהות את הנוזקות וכך לדעת שהמחשב נגוע, ואף להשתמש במידע כדי למנוע מהנוזקות לרוץ, כמו חיסון. במאגרי מידע על נוזקות ניתן למצוא את רשימת ה-Mutexים שהגרסאות השונות של הנוזקות עושות בהן שימוש. לדוגמה, להלן רשימת Mutexים חשודים של הנוזקה Qakbot, כפי שמופו על ידי האתר: https://blog.talosintelligence.com

Win.Trojan.Qakbot-6327689-0

INDICATORS OF COMPROMISE

Registry Keys

- <HKCU>\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN
    - **Value:** kddds
- <HKCU>\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\INTERNET SETTINGS
    - **Value:** ProxyOverride
- <HKLM>\SYSTEM\CONTROLSET001\SERVICES\DBRNOCX
    - **Value:** DisplayName

Mutexes

- wawrtxtguelkunm
- \BaseNamedObjects\Global\uhtvtft
- eioigs
- \BaseNamedObjects\Global\ubrjqsxr
- knsoonoa

כתוצאה מכך, הנוזקות נעשו מתוחכמות יותר והן אינן משתמשות בשמות קבועים של Mutexים. אם השמות אינם קבועים, איך בכל זאת הנוזקות יכולות לדעת שהן כבר מותקנות על המחשב? הפתרון של מתכנתי הנוזקות הוא ליצור שמות של Mutexים שמבוססים על דברים שמשתנים בין מחשבים, אך נשארים קבועים באותו מחשב. לדוגמה, המספר הסדרתי של רשיון מערכת ההפעלה Windows, או כתובת ה-MAC של כרטיס הרשת. בצורה זו, לא ניתן לספק רשימת Mutexים שנותנים אינדיקציה לכך שנוזקה התקינה את עצמה על המחשב, מכיוון שרשימה כזו תכלול מיליארדי רשומות.

תרגיל 5.3 **תרגיל מסכם** Mutex

פיתרו שנית את בעיית הפילוסופים הסועדים, אך הפעם הגדירו את מקלות האכילה בתור Mutexים.

הדרכה לפתרון:

התוכנית הראשית שלכם צריכה ליצור חמישה Processים, אחד עבור כל פילוסוף. השתמשו בפונקציה CreateProcess, שתיעוד עליה תמצאו ב-MSDN. כפי שתמצאו, הפונקציה מקבלת כפרמטר שם של קובץ exe שעליה להריץ עבור הProcess החדש. לצורך ההסבר, נקרא לקובץ זה "SinglePhilosopher.exe".

שלב א- יצירת הקובץ SinglePhilosopher.exe

הפרוייקט הראשון שתפתחו ייצור את הקובץ המבוקש, שמבצע את הלוגיקה עבור פילוסוף יחיד. ראשית, על התוכנית לזהות את המיקום של הפילוסוף, כדי לקבוע אילו שני Mutexים ("מקלות אכילה") עליו לתפוס כדי לאכול. זיהוי מיקום הפילוסוף יכול להתבצע באמצעות מספר שיועבר אליו מהתוכנית שיוצרת את הProcess, באמצעות הפונקציה CreateProcess.

לאחר שהProcess שלנו מבין אילו שני Mutexים הוא צריך להחזיק, נמתין לאחד מהם, וברגע שהוא מתפנה ננסה לתפוס את השני. אם נצליח, נדפיס הודעה מתאימה ונשחרר את ה-Mutexים. אם השני תפוס, נצטרך לשחרר את השני כדי למנוע מצב של Deadlock.

שלב ב- יצירת התוכנית הראשית

הפרוייקט השני שתפתחו ינהל את ה-Processים. הפרוייקט יגדיר חמישה Mutexים, כמספר מקלות האכילה, וכן יפתח חמישה Processים חדשים, כמספר הפילוסופים.

הגדרת ה-Mutexים מתבצעת על ידי הפונקציה CreateMutex, שמקבלת בפשטות את השם שבחרתם עבור כל Mutex. העזרו בתיעוד הפונקציה:

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexa

פתיחת ה-Processים מתבצעת באמצעות הפונקציה CreateProcess. בפרק הקודם ראינו כיצד קוראים לפונקציה הזו וכיצד מעבירים לה פרמטרים. עליכם להעביר לכל Process את המספר הסידורי שלו, כדי שיידע באילו "מקלות אכילה" עליו להשתמש. השתמשו בקוד הדוגמה שניתן בפרק הקודם כדי ליצור מחרוזת שכוללת את הפרמטר שמועבר לכל Process.

כאשר ביצעתם את התרגיל עם Critical Sections מדדתם את משך הזמן שלוקח להאכיל כל פילוסוף מיליון פעם. כיצד לדעתכם ישתנה משך הזמן אם תבצעו את אותה המדידה עם Mutexes? בצעו אתת הניסוי וודאו שאתם מבינים את הסיבה לשוני בזמנים.

5.7 מנגנון איתות Semaphore

עד עכשיו דנו בשני סוגים של מנעולים, שהיו בינאריים במהותם. כלומר, אם ניקח מנעול מסוג Mutex או מסוג CriticalSection, יש בכל רגע נתון רק Thread אחד שמחזיק במנעול.

המנעולים הבינאריים שלנו עבדו יפה כל עוד היה הגיון בכך שרק Thread יחיד יהיה הבעלים של המנעול, כי רצינו למנוע מ-Threadים אחרים גישה לקטע קוד קריטי. אך לפעמים המקרה הוא שונה. נשתמש בדוגמה

מהחיים שבוודאי מוכרת לכולם- שירותים ציבוריים. השירותים מכילים מספר תאים. בכל פעם שנכנס אדם הוא תופס תא וכמות התאים הפנויים יורדת באחד, בכל פעם שאדם מסיים התא שלו מתפנה וכמות התאים הפנויים עולה באחד. אם הגעתם לשירותים שבהם כמות התאים הפנויים היא אפס, אתם נאלצים להמתין לתא פנוי.

באופן דומה גם במחשב יש לפעמים משאבים שיש להם מאפיינים של "שירותים ציבוריים". נניח לדוגמה שאנחנו רוצים להגביל גישה לקובץ כלשהו לכמות משתמשים מקסימלית, או להגביל את כמות החלונות שתוכנה פותחת. לשם כך נשתמש ב-Semaphore.

ה-Semaphore אינו מנעול אלא מנגנון איתות (באנגלית Signalling), שמאפשר ל-Thread לאותת ל-Threadים אחרים כאשר אירוע מוגדר מתרחש, אך ניתן להשתמש בו באופן דומה למנעול. Semaphore מאפשר ליותר מ-Thread אחד להיות בעלים של משאב משותף. הדרך שבה הדברים מתבצעים היא שבזמן יצירת ה-Semaphore מוגדר מספר מקסימלי כלשהו. תפיסת משאב מורידה ב-1 את ערכו של ה-Semaphore, שחרור משאב מעלה את ערכו ב-1. אם הערך של ה-Semaphore שווה לאפס, לא ניתן לתפוס בעלות על המשאב. תכונה חשובה של Semaphore היא שהעלאת או הורדת הערך היא פעולה אטומית, כלומר פעולה שבמהלכה לא יכול להתבצע Context Switch, שאחרת הספירה היתה עלולה להיות שגויה.

לא נתעכב על הפונקציות השונות שמשמשות ליצירת ושימוש ב-Semaphore, הן די דומות לפונקציות שלמדנו עבור Critical Section ו-Mutex, וניתן למצוא אותן באתר MSDN בלינק

https://docs.microsoft.com/en-us/windows/win32/sync/using-semaphore-objects

ב-Windows קיימים שני סוגים של Semaphore. בתהליך יצירת ה-Semaphore נבחר אם לתת לו שם או לא. אם לא ניתן לו שם, הוא יהיה אובייקט מקומי של ה-Process ורק Threadים באותו ה-Process יוכלו להשתמש בו, בדומה ל-CriticalSection. אם ניתן לו שם, מה שנקרא "Named Semaphore", הוא יהיה אובייקט של מערכת ההפעלה. לכן גם Threadים מ-Processים אחרים יכולים לקבל Handle ל- Named Semaphore. הפעולה מתבצעת באמצעות הפונקציה OpenSemaphore, שמקבלת את שם ה-Semaphore בתור פרמטר. כפי שראינו עם Mutexים, כאשר Semaphore הוא אובייקט של מערכת ההפעלה יש לכך משמעות, כל קריאה אליו צריכה לעבור דרך מערכת ההפעלה. לכן הוא יותר איטי מאשר CriticalSection.

לסיכום, להלן טבלת השוואה בין שלושת המנגנונים שהכרנו:

	CriticalSection	Mutex	Named Semaphore
סנכרון **Thread**ים שנמצאים באותו **Process**	V	V (אפשרי אך פחות יעיל מאשר Critical Section)	V
סנכרון **Thread**ים ש*אינם* נמצאים באותו **Process**	X	V	V
כמות גישות מקסימליות בו זמנית למשאב	1	1	מוגדר על ידי המשתמש

מהתבוננות בטבלה נראה כאילו Mutex הוא מקרה פרטי של Named Semaphore, שבו המשתמש מגדיר את כמות גישות מקסימלית להיות 1. מקרה זה נקרא Binary Semaphore ואכן, מבחינה מעשית יש דמיון רב ל-Mutex. עם זאת, כאמור Semaphore הוא מנגנון איתות ולא מנעול. לכן אין Thread שהוא הבעלים שלו וכל Thread יכול להעלות או להוריד את הערך שלו, בניגוד ל-Mutex.

5.8 סיכום

התחלנו את הפרק בהבנה של מושג ה-Race Condition, שעלול לגרום למצב שבו שני Processים או Threadים מפריעים זה לזה לעבוד עקב בעיות של שימוש במשאב משותף. ראינו, שכדי לעבוד עם משאב משותף יש צורך במנגנון של Lock, שימנע את תופעת ה-Race Condition.

ראינו ששימוש לא מושכל במנעול עלול לגרום למצב של Deadlock, בו המערכת מחכה לנצח למשאב שלעולם לא ישתחרר מכיוון ששחרור המשאב תלוי בעצמו בצורה מעגלית.

לאחר מכן דנו בשני סוגים של מנעולים- הראשון הוא CriticalSection, מנגנון שמשמש לסנכרון בין Threadים של אותו Process. זהו מנגנון קל יחסית לשימוש, כיוון שהוא דורש רק הגדרה של משתנה גלובלי. המנגנון השני הוא Mutex, שמשמש לסנכרון בין Processים. מנגנון זה מצריך שימוש במשאבי מערכת ההפעלה וקריאה לפונקציות שפועלות עם Mutexים דורשת לכן פניה אל הקרנל. הפניה אל הקרנל דורשת זמן רב יותר מאשר פניה אל משתנה גלובלי כמו CriticalSection, וזו הסיבה לכך שבתרגיל הפילוסופים השימוש ב-CriticalSections נתן מדידת זמן קצרה משמעותית.

לבסוף למדנו על מנגנון האיתות Semaphore, שמאפשר להקצות למשאב כמות מקסימלית של משתמשים שחולקים אותו.

פרק 6 – זיכרון

נושא זיכרון המחשב היה שזור עד כה כמעט בכל נושא שכיסינו עד כה. בפרק הראשון ראינו כיצד נראה מרחב זיכרון של Process יחיד. לאחר מכן כשסקרנו את תפקידי מערכת ההפעלה, ניהול הזיכרון היה אחד מארבעת התפקידים, והזכרנו את העובדה שלמערכת ההפעלה יש אזור זיכרון מוגן ונפרד משלה, אזור שאינו נגיש ל-Processים שרצים ב-Userland. כאשר דנו בProcess-ים ו-Threadים יצאנו מנקודת הנחה שלכל Process יש מרחב זיכרון משלו ואילו Threadים חולקים את אותו מרחב זיכרון. הזכרנו גם ש-Processים יכולים לשתף זיכרון ביניהם. קשה להעלות על הדעת משהו שהמעבד יכול לבצע ללא שימוש כלל בזיכרון, אפילו הרצה של פקודת מכונה בודדת מורכבת משלבים שהראשון ביניהם הוא הבאה (Fetch) של פקודה מהזיכרון אל יחידת העיבוד של המעבד.

אך יש שאלות רבות שנותרו פתוחות ובפרק זה נענה עליהן. נתחיל מפירוט התקני הזיכרון ונסביר מדוע בכלל יש צורך בסוגי זיכרון שונים. לאחר מכן נדבר על כתובות וירטואליות, נבין מהן, מדוע יש בהן צורך וכיצד מתבצע התרגום בין כתובות פיזיות ווירטואליות.

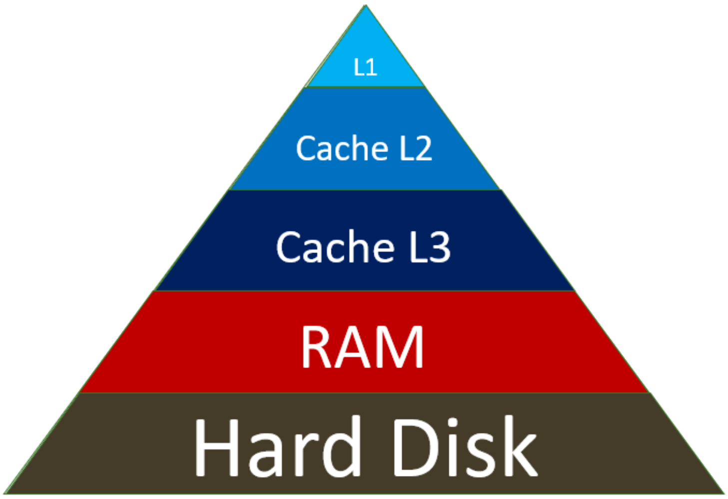
6.1 זיכרון פיזי

זיכרון פיזי הוא התקן חומרה שמאפשר שמירה של רצפים של אחדות ואפסים. כל אחד מאיתנו מכיר התקני חומרה שמשמשים כזיכרון: לדוגמה הדיסק הקשיח, USB ודיסק אופטי. כמו כן בפרקים הקודמים הזכרנו שתוכנות מחשב רצות מתוך זיכרון שנקרא "RAM". ייתכן ששמעתם גם על המושג "זיכרון Cache". ובכן, מדוע יש צורך בסוגים שונים של זיכרונות ומה תפקידם? בתור משתמשים, ישנם ארבעה דברים שהיינו רוצים מזיכרון של מחשב:

- מהירות- זמן הגישה (קריאה וכתיבה) של המעבד לזיכרון הוא בעל חשיבות קריטית לריצה מהירה של תוכניות. חישבו על מערכת דמיונית שבה יש מעבד מהיר, שמסוגל לבצע פעולת חיבור במיליארדית שניה, ואילו לצד המעבד יש זיכרון איטי שכל גישה אליו לוקחת שניה. פעולת חיבור של שני משתנים כוללת שתי גישות לזיכרון לטובת קריאת המשתנים, וקריאה נוספת לזיכרון לטובת כתיבת התוצאה. כמות הזמן הכללית שתיקח למערכת לבצע את החיבור תהיה לכן 3 שניות ועוד מיליארדית שניה... המהירות הרבה של המעבד פשוט לא באה לידי ביטוי עקב הזיכרון האיטי
- גודל- האם קרה לכם שרציתם לשמור קובץ כלשהו על דיסק קשיח או על USB וקבלתם הודעת שגיאה שהדיסק מלא? נראה שאין צורך להסביר למה יותר זיכרון עדיף על פחות זיכרון (בהנחה שמשתנה רק גודל הזיכרון, ולא דברים אחרים כגון מהירותו)
- יציבות – אנחנו רוצים זיכרון שיאגור בתוכו את המידע גם כאשר הוא אינו מקבל חשמל. דמיינו שהדיסק הקשיח שלכם היה נמחק בכל פעם שהייתם מנתקים את המחשב מהחשמל, ושכל הדלקה של המחשב היתה כרוכה בהתקנה מחדש של מערכת ההפעלה...
- מחיר – כמובן, עדיף לשלם כמה שפחות!

הבעיה היא, שאין זיכרון שהוא אופטימלי מכל ארבעת הבחינות האלה. זיכרון גדול יותר ומהיר יותר גם יהיה יקר יותר. זיכרון קבוע יהיה יותר יקר מאשר זיכרון נדיף (זיכרון שנמחק עם הפסקת זרם החשמל, שנקרא

Volatile, לעומת זיכרון Non-Volatile, שנשמר גם ללא אספקת זרם חשמל). במילים אחרות, אין זיכרון אחד שעדיף על כל הזיכרונות. לכל זיכרון יש את נקודות העדיפות והנחיתות שלו על פני זיכרונות אחרים.
כדי לבנות מערכת בעלת ביצועים אופטימליים אנחנו צריכים לשלב סוגי זיכרונות שונים. נשתמש בהרבה זיכרון איטי וזול, ומעט זיכרון מהיר ויקר. התרשים הבא מייצג את ההיררכיה של התקני זיכרון- איטי וזול בתחתית הפירמידה, מהיר ויקר בקצה.



התפקיד של מערכת ההפעלה הוא לנהל בצורה יעילה את משאבי הזיכרון שיש לה. בתור כלל אצבע היינו רוצים שקוד שאנחנו זקוקים לו לעיתים קרובות יהיה בזיכרון מהיר, וככל שהצורך שלנו בקוד הוא לעיתים יותר רחוקות, כך נוכל להתפשר ולשמור אותו בזיכרון יותר איטי. הבאת מידע מזיכרון איטי עלולה לקחת זמן רב בצורה ניכרת, כפי שמיד נראה, ולכן מערכת ההפעלה צריכה לנסות לצמצם ככל האפשר את הסיכוי שנצטרך מידע שנמצא בזיכרון איטי. כמובן, שככל שיש לנו יותר זיכרון מהיר, כך עבודתה של מערכת ההפעלה קלה יותר והסיכוי שנמצא את מה שאנחנו צריכים בזיכרון המהיר הוא גדול יותר.

כדי לפשט מעט את התמונה, נתחיל בתיאור של מחשב שיש בו רק שני סוגים של זיכרון- דיסק קשיח ו-RAM. הדיסק הקשיח הוא כלי קיבול זול וענקי. במאה דולר אפשר לרכוש נכון למועד כתיבת שורות אלה (אמצע 2020) דיסק בנפח דמיוני של 1 טרה בתים. כמה בתים יש ב-1 טרה בתים?

1KB (Kilo Bytes) = 1024B (Bytes)

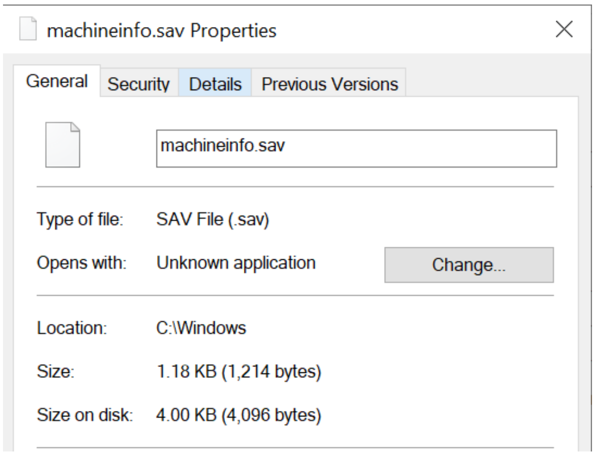
1MB (Mega Bytes) = 1024KB = 1,048,576B

1GB (Giga Bytes) = 1024MB = 1,073,741,824B

1TB (Tera Bytes) = 1024GB = 1,099,511,627,776B

למעלה ממליון-מליוני בתים. דיסק כזה יכול לשמור בקלות מאות אלפי תמונות דיגיטליות. מעניין לציין שיצרני הדיסקים נוטים לאמץ שיטת ספירה אחרת, בה הכפולות אינן של 1024 כפי שצריך להיות אלא של 1000 בלבד. כך לדוגמה אם תרכשו דיסק בגודל 1TB תקבלו "רק" 1,000,000,000,000 בתים. כפי שרואים, שיטת הספירה הזו מאפשרת ליצרנים למכור לכם 10% פחות מנפח הדיסק שחשבתם שאתם רוכשים...
מובן מאליו שהדיסק הקשיח אינו נמחק כאשר הוא מנותק מחשמל, כלומר הוא Non-Volatile. המידע על הדיסק הקשיח נשמר בכפולות של 4 קילו בתים, גודל שנקרא "בלוק". אם יש קובץ קטן מ-4KB הוא יישמר בבלוק שזה גודלו, ואם יש קובץ שגדול מבלוק יחיד, הגודל שלו יעוגל מעלה אל כפולת הבלוקים השלמה

הקרובה. לטובת ההמחשה, בחרו קובץ כלשהו ובידקו את ה-Properties שלו, תגלו שהגודל על הדיסק הוא כפולה של גודל בלוק:



כיוון שזמן הגישה אל הדיסק הקשיח הוא כה גדול, המעבד אינו ניגש ישירות אל כתובות בזיכרון של הדיסק הקשיח. במקום זאת, מערכת ההפעלה מפעילה Driver, שמביא בכל פעם לפחות בלוק אחד של מידע מהדיסק. היתרון בכך שמביאים בלוק שלם מהדיסק הקשיח הוא שכמעט תמיד נצטרך להשתמש לא רק במידע שנמצא בבית מסויים אלא גם בבתים הקרובים לו. כך, זמן הגישה הארוך אל הדיסק הקשיח מתחלק בין כל הבתים שנמצאים בבלוק. לדוגמה, אם במערכת כלשהי גישה לדיסק הקשיח לוקחת 4 מילישניות, ובגישה הזו מובא בלוק שכאמור גודלו הוא 4KB (4096 בתים), אז הבאה של כל בית מהדיסק הקשיח לוקחת בערך מיליונית שניה בלבד.

כיוון שאין למעבד יכולת לגשת לכתובת ספציפית בדיסק הקשיח, אלא רק לבלוק של מידע, המשמעות היא שאי אפשר להריץ תוכניות ישירות מהדיסק הקשיח (הערת צד: יש שיטה בשם Execute In Place שמאפשרת זאת בתנאים מסויימים, לא נסקור אותה). זאת מכיוון שכאשר מעבד מריץ תוכנה, הוא חייב גישה לכתובות ספציפיות. דמיינו מצב שבו המעבד מבקש להביא אליו את הפקודה הבאה להרצה, ובמקום לקבל פקודה יחידה הוא מקבל בלוק של 4096 בתים. יש צורך בהתקן זיכרון אחר שמאפשר למעבד לגשת לכתובות ספציפיות.

לעומת הדיסק הקשיח, גודלו של ה-RAM הוא קטן למדי ויקר הרבה יותר. תמורת מאה דולר ניתן לרכוש RAM שגודלו אחוזים בודדים מגודלו של דיסק קשיח במחיר זהה. אם זה לא גרוע מספיק, ה-RAM הוא גם זיכרון מסוג Volatile, שיימחק בכל פעם שנכבה את המחשב. היתרונות של ה-RAM על פני הדיסק הקשיח הם מהירותו והעובדה שהמעבד יכול לגשת באופן ישיר לכל כתובת ב-RAM. זו הסיבה שכדי להריץ תוכניות ששמורות על הדיסק הקשיח, ראשית כל יש צורך להעתיק אותן אל ה-RAM.
אם נסכם עד כה, יש זיכרון ענק, איטי, ובלתי מחיק, אשר חלקים ממנו מועתקים לפי הצורך אל זיכרון קטן, מהיר ומחיק. באופן זה אפשר לקבל נפח אחסון כמעט אינסופי, אך במקביל לדאוג שבזמן שהמעבד מריץ קוד יעמוד לרשותו זיכרון מהיר.

מנגנון ה-Cache הוא שיפור של מודל העבודה שסקרנו עד כה. הרי לא כל המידע שמגיע ל-RAM משמש את המעבד בתכיפות זהה. לדוגמה, נדמיין תוכנית מסויימת שמציגה למסך סרטון וידאו. בעוד כל פריים בסרטון מוצג למסך פעם אחת בלבד, ייתכן שקטע קוד מסויים בתוכנית מורץ שוב ושוב. ניקח את אותו קטע קוד ונשים

אותו בזיכרון מהיר עוד יותר. כך נוכל להאיץ את הגישה לזיכרון בדיוק לאותו אזור בזיכרון שהמעבד מרבה לפנות אליו. כיצד יוצרים זיכרון מהיר יותר? לדוגמה, במקום לשים אותו על לוח האם (Motherboard) כמו את ה-RAM, יצרניות המעבדים שמות את ה-Cache ממש על הסיליקון של המעבד. גם בסיליקון של המעבד יש אזורים "קרובים" יותר ליחידת העיבוד המרכזית, וכך נוצרים למעשה שלושה זיכרונות Cache, הידועים בכינויים L1, L2, L3, כאשר L1 הוא המהיר והקטן ביותר.

מצב שבו המעבד זקוק למידע מסוים והמידע הזה נמצא ב-Cache, נקרא Cache Hit. המצב ההפוך, בו המידע אינו נמצא ב-Cache ויש להביא אותו מה-RAM או חס וחלילה מהדיסק הקשיח, נקרא Cache Miss. חלק מהחוכמה שבקוד מערכת ההפעלה היא הלוגיקה של מתי להעביר מידע ל-Cache ומתי להחליף מידע שב-Cache במידע אחר.

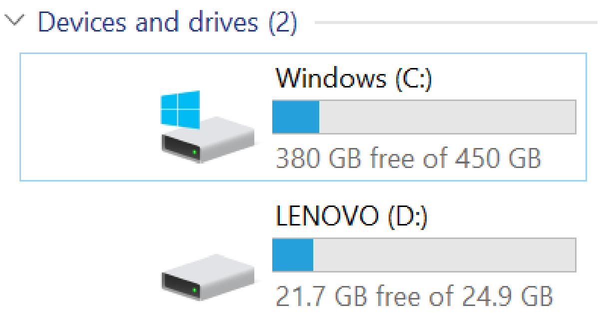
עד כה דנו באופן תיאורטי בזיכרון מהיר יותר ופחות. כדי להבין את סדרי הגודל המדוברים, נסקור את כמות הזמן שלוקחת גישה לזיכרון מסוגים שונים. כל הזמנים המצוטטים כאן הם מהאתר http://norvig.com/21-days.html#answers, חשוב לציין הזמנים משתנים בין חומרה לחומרה אך היחס ביניהם מספק סדר גודל. ובכן:

- זמן גישה לרגיסטר: 0
- זמן גישה ל- L1 Cache – 0.5 ננו שניות (ננו שניה – אחד חלקי מיליארד של שניה)
- זמן גישה ל- L2 Cache – 7 ננו שניות
- זמן גישה ל-RAM – 100 ננו שניות
- זמן גישה לדיסק קשיח – זמן הגישה תלוי מאד בסוג הדיסק ובממשק הפיזי בינו לבין המעבד. לדיסק קשיח מגנטי זמן הגישה הוא מסדר גודל של מיליון ננו שניות (אלפית שניה), כלומר פי 2 מיליון מאשר זמן הגישה ל-L1. לדיסק בטכנולוגיית SSD זמן הגישה הוא מסדר גודל של 5000 ננו שניות.

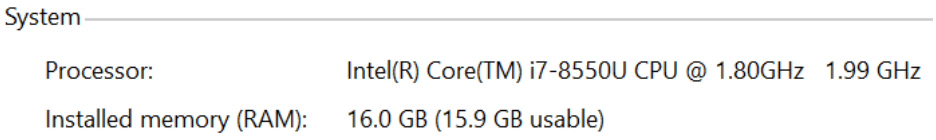
נשווה זאת לחיים האמיתיים, (קרדיט לחברי יותם גיגי על רעיון ההמחשה): אנחנו יושבים ליד שולחן כתיבה וזקוקים לעט. פניה לרגיסטר היא כמו שימוש בעט שנמצא אצלנו ביד, אנחנו לא צריכים להקדיש זמן כדי להרים אותו. גישה ל- L1 Cache היא כמו להרים את העט משולחן הכתיבה, נניח שפעולה זו לוקחת כחצי שניה. גישה ל- L2 Cache היא כמו להוציא עט ממגירה שבשולחן הכתיבה, סדר גודל של מספר שניות. גישה ל-RAM היא כמו לקום מהשולחן וללכת לקחת עט שנמצא בחדר אחר בבית, סדר גודל של דקה. ומה לגבי גישה לדיסק הקשיח, בפרט אם הוא מגנטי? גישה זו, בהתאם ליחסי הזמנים שבדוגמאות הקודמות, היא כמו להקדיש כ- 250 שעות להשגת העט. כלומר גישה להארד דיסק נראית למעבד כמו איסוף של עט מחנות שנמצאת באוסטרליה. גם אם נשתמש בדיסק SSD, זמן הגישה אליו מקביל לאיסוף העט מחנות שנמצאת במרחק שעה נסיעה.

כדי להשלים את התמונה, נבדוק בעצמנו כמה זיכרון מכל סוג ישנו במחשב האישי שלנו.

1. את גודלו של הדיסק הקשיח תוכלו למצוא באמצעות ה-Explorer. הצביעו על האייקון של המחשב האישי ובידקו את ה-Properties



2. את גודלו של ה-RAM תוכלו למצוא באמצעות Control Panel -> System and Security -> System



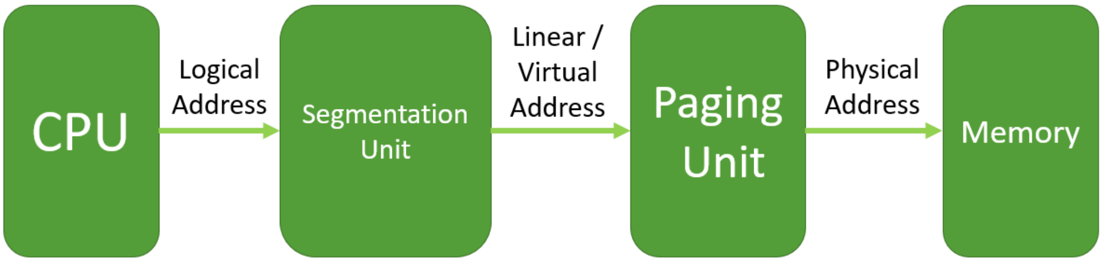
3. את גודלו של L2, L3 Cache תוכלו למצוא באמצעות ה-cmd. הריצו את הפקודה הבאה:

wmic cpu get L2CacheSize, L3CacheSize

4. כדי למצוא את גודלו של L1, תצטרכו לאתר את דגם המעבד הספציפי שלכם (מצאנו אותו כאשר חיפשנו את גודל ה-RAM), ולאחר מכן חפשו בגוגל את דגם המעבד יחד עם הביטוי L1 Cache.

6.1.1 גישה לכתובות בזיכרון הפיזי

בפרק בו דנו בתפקידי מערכת ההפעלה, מתחנו קו שמחבר בין המעבד לבין הזיכרון. זו כמובן הפשטה של המציאות. בפועל, החיבור בין המעבד לזיכרון הפיזי כולל כמה וכמה שלבים בדרך, לפי תרשים הזרימה הבא:



נפרט ונסקור שלב שלב, מה הוא מבצע וכיצד.

6.1.2 חלוקת הזיכרון ל-Segmentים, כתובות לוגיות

נתחיל מהמעבד. כרגיל, נתמקד במעבד ממשפחת ה-86x. המעבד קיבל מספר פקודות אסמבלי לבצע:

```
mov   eax, [ebx]
mov   eax, [ebp]
jmp   label
```

כל אחת מהפקודות הללו ניגשת לאזור זיכרון שונה. בפקודה הראשונה הרגיסטר ebx מצביע על משתנה בזיכרון הנתונים, הפקודה תטען לתוך eax את הערך שהמצביע מצביע עליו. בפקודה השניה המעבד יזהה שהשימוש ברגיסטר ebp אומר שיש צורך לפנות אל אזור הזיכרון של המחסנית. פקודת ה-jmp קובעת שיש לקפוץ למקום מסוים באזור הזיכרון בו נמצא קוד התוכנית.

כל אחד מאזורי הזיכרון נקרא Segment, ולמעבד יש רגיסטרים מיוחדים ששומרים את המיקום בזיכרון של כל ה-Segmentים. בפקודות האסמבלי שהובאו בתור דוגמאות נעשה שימוש ברגיסטרים הבאים:

DS – Data Segment

SS – Stack Segment

CS – Code Segment

Segmentים נוספים, שלא נעשה בהם שימוש בדוגמה הזו, הם ES, FS, GS.

כך שהדרך בה כתבנו את הפקודות היא למעשה קיצור כתיבה של הפקודות הללו:

```
mov   eax, [DS:ebx]
mov   eax, [SS:ebp]
jmp   CS:label
```

צורת ההצגה הזו של כתובת נקראת כתובת לוגית, או Logical Address. הכתובת הלוגית מחולקת לSegment ול-Offset, שהוא ההיסט מתחילת ה-Segment. אם המונחים הללו נראים לכם מבלבלים, חישבו על ספרים. אתם יכולים להחזיק מספר ספרים, בכל ספר יש עמודים רבים. אם מישהו יבקש ממכם לפתוח ספר בעמוד 20, המידע לא יספיק לכם כדי לדעת לאיזה ספר הוא מתכוון. לכן צריך קודם כל לציין על איזה ספר מדובר (ה-Segment) ולאחר מכן מה העמוד הספציפי (ההיסט מתחילת הספר).

כל עוד מדובר בספרים ועמודים החישוב פשוט. אבל איך מתרגמים Segment והיסט לכתובת פיזית? בעבר היתה לכך שיטה פשוטה. במעבד 16 ביט צורת חישוב הכתובת הפיזית נקראת Real Mode. בשיטה זו רגיסטר ה-Segment מחזיק את כתובת ההתחלה של ה-Segment, ויש לה שלושה שלבים פשוטים:

1. הוצא את כתובת הבסיס של ה-Segment מתוך רגיסטר ה-Segment
2. כפול את כתובת הבסיס של ה-Segment ב-16 (קל לבצע באמצעות הזזה של 4 ביטים שמאלה, היזכרו בספר האסמבלי)
3. הוסף לתוצאה את ההיסט.

לדוגמה, אם רגיסטר ה-DS Segment שווה לערך 0x2000 המשמעות היא ש-Segment הנתונים מתחיל בכתובת 0x2000 כפול 16, כלומר 0x20000. אם נרצה להגיע להיסט מסוים פשוט נוסיף אותו לכתובת ההתחלה. לדוגמה, אם bx=0x3012, אז השילוב של DS:bx יביא אותנו לכתובת 0x23012.

במעבדי 32 ביט, ומאוחר יותר גם 64 ביט, אימצו שיטת חישוב כתובות שנקראת Protected Mode והיא מורכבת הרבה יותר, אך הבסיס לכל חישובי הכתובות עדיין נמצא בכתובת של ה-Segment ובהיסט בתוך ה-Segment. מצד שני, ב- Protected Mode כבר אי אפשר להוציא את כתובת הבסיס של ה-Segment ישירות מתוך רגיסטרי ה-Segment. במקום זאת רגיסטרי ה-Segment כוללים אלא אוסף של שדות, כאשר אחד השדות הוא בגודל שני ביטים והוא מייצג את מעגל ההרשאה Protection Ring שה-Process שלנו

נמצא בו. לכן, החל ממערכות של 32 ביט כבר לא קוראים להם Segment Registers אלא Segment Selectors, כדי להדגיש שהמידע שהם שומרים השתנה.

נסכם את מה שלמדנו עד כה בנוגע לכתובת לוגית של Protected Mode:

מיד נכנס לפרטי החישובים שמתבצעים ב-Protected Mode, עד שנפענח את כל הדרך שמתבצעת מהכתובית הלוגית עד לכתובת הפיזית.

6.1.3 יחידת ה-Segmentation והמרה ל-Linear Address

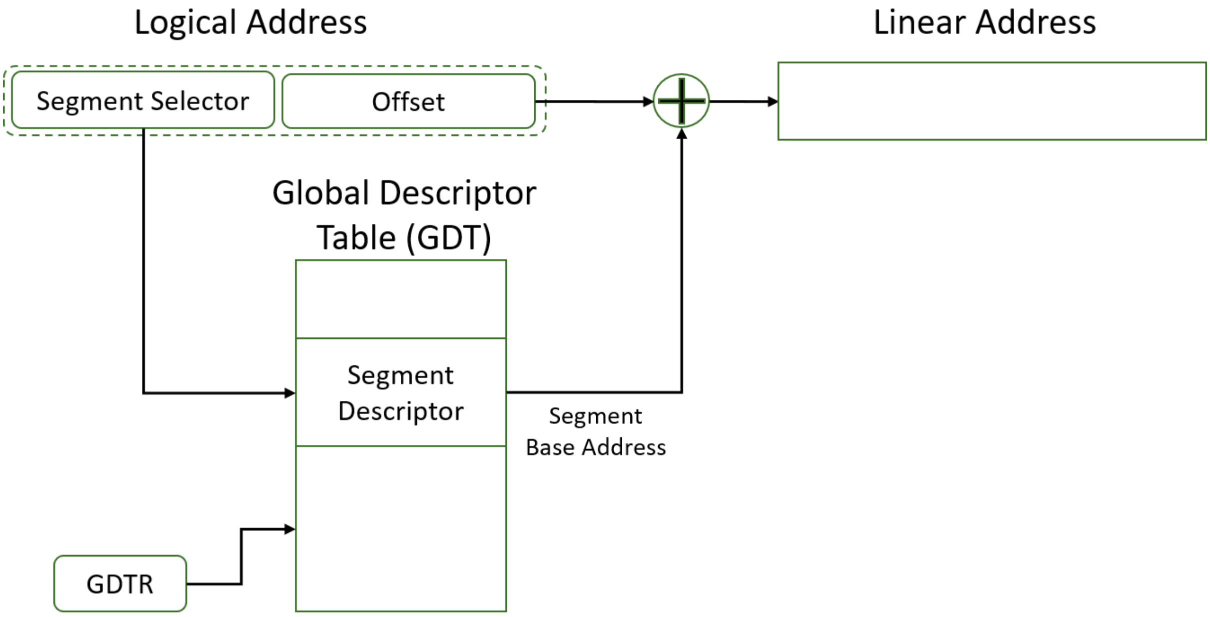
סגמנטציה היא פעולה היסטורית במקצת, אחד התפקידים המרכזיים שלה - מניעה מ-Process לגשת לאזור זיכרון של Process אחר – מיושם כיום על ידי מנגנון ה-Paging שנדון בו בהמשך. על כן, נסתפק בתיאור תמציתי יחסית של יחידת הסגמנטציה.

כיום נותרו ליחידת הסגמנטציה שני תפקידים:

1. ליישם את מנגנון ה-Protection Rings וכך למנוע גישה לא מבוקרת של Thread לאזורי זיכרון שמעבר להרשאה שלו.
2. לתרגם Logical Address ל-Linear Address. כתובת לינארית היא כתובת שאינה מבוססת על שילוב של Segment והיסט בתוך ה-Segment, אלא מספר יחיד שבאמצעותו אפשר לגשת אל הזיכרון הפיזי. שימו לב שכתובת לינארית אינה כתובת פיזית, יש ביניהן את מנגנון ה-Paging.

כל כתובת לוגית מורכבת כאמור מ-Effective Address ומ-Segment Selector. ה-Segment Selector מצביע לתוך טבלה שנקראת GDT, או Global Descriptor Table (קיים גם LDT, Local Descriptor Table, שהוא גם טבלה אך פרטית לכל Thread). בתוך ה-GDT יש רשומות שמאפשרות למצוא את הכתובת שבה נמצאת כתובת הבסיס של ה-Segment. זוכרים איך ב-Real Mode מוצאים את כתובת הבסיס של כל Segment פשוט בעזרת רגיסטר ה-Segment? בעצם מה שאנחנו עושים פה זה אותו דבר, רק שההמרה מתבצעת באמצעות טבלת חיפוש, שה-Segment Selector מצביע בה על הרשומה המתאימה. התוצאה של החיפוש בטבלה הזו היא כתובת הבסיס של ה-Segment.

לכתובת הבסיס של ה-Segment צריך להוסיף את ה- Effective Address וכך מגיעים אל הכתובת הלינארית.



מפאת חשיבות ה-GDT, למעבד יש רגיסטר מיוחד שנקרא GDTR שכל תפקידו הוא להחזיק את הכתובת של ה-GDT. ה-GDTR הוא רגיסטר שתוכניות שרצות מחוץ למעגל 0 אינן יכולות בכלל לגשת אליו.

כעת הגענו בדיוק לנקודה שבזכותה קוראים ל-Protected Mode בשמו. כאשר דנו ב-Segment Selectors הזכרנו שהם כוללים שני ביטים שמבטאים את ה-Protection Ring של ה-Process. לכל רשומה ב-GDT צמוד מעגל ההרשאה שיש לה, מה שמאפשר למערכת ההפעלה להשוות בין ההרשאה של ה-Process לבין ההרשאה של Segment הקוד שהוא מנסה להריץ (או להרשאה של Segmentים אחרים שהוא מנסה לפנות אליהם). ההרשאה של ה-Process קרויה CPL, קיצור של Current Protection Level. ההרשאה הנדרשת קרויה DPL, קיצור של Desired Protection Level. כיוון שכל גישה לזיכרון מתחילה מ- Segment Selector, אם Process מנסה לגשת למעגל הרשאה נמוך יותר (בעל הרשאה גבוהה יותר, כיוון ש-0 הוא ה-Kernel) מאשר מעגל ההרשאה שהוא נמצא בו, הגישה תיחסם ותתקבל שגיאה.

אם כך, איך מערכת ההפעלה מאפשרת לקוד לקרוא לפונקציות של ה-Kernel? בפרקים הקודמים הזכרנו שהדבר מבוצע באמצעות Syscall. כזכור אחד השלבים של ביצוע Syscall הוא ביצוע Sysenter, שמעלה את רמת ההרשאה לרמת הרשאה של Kernel. לאחר ביצוע הפעולות הנדרשות, מבוצע Sysexit ורמת ההרשאה חוזרת לרמה המקורית.

6.2 זיכרון וירטואלי

כדי להבין מדוע בכלל יש צורך בזיכרון וירטואלי, נסקור כיצד נראית מערכת המבוססת על מה שלמדנו עד כה:

1. מרחב הזיכרון של כל Process נמצא ב-RAM.
2. לכל Process יש מרחב זיכרון משלו.
3. המעבד עובד ב-Protected Mode, עם מעגלי הרשאה שונים עבור Threadים שרצים ב- Userland לעומת Kernel Threads, כלומר לכל אזור זיכרון מוגדר אם הוא שייך למעגל הרשאה User או

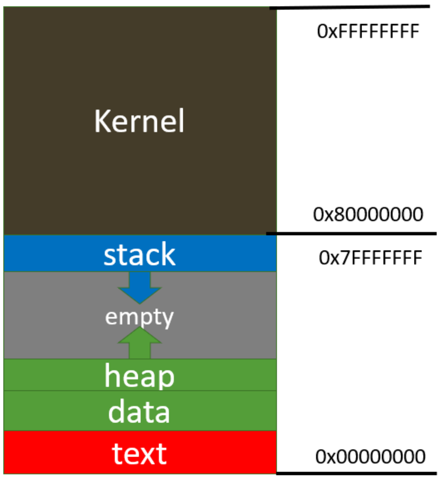
Thread .Kernelים של User יכולים לפנות רק לאזור זיכרון של User (עד כדי באגים, כמו Meltdown או Spectre).

כל אחד מהעקרונות הללו הוא נכון, אבל הם אינם מספיק טובים. הנה שלוש בעיות, המתייחסות לשלושת העקרונות הללו:

1. מרחב הזיכרון של כל Process נמצא ב-RAM: מה קורה אם ה-RAM קטן מלהכיל את מרחב הזיכרון של ה-Process? במערכות הפעלה של 32 ביט, ל-Process יש מרחב זיכרון של 4GB (היזכרו- 2 בחזקת 32 כתובות). בעבר זיכרון RAM בגודל כזה היה לא מציאותי. גם כיום, במערכות הפעלה של 64 ביט, אין RAM שגודלו 2 בחזקת 64. במילים אחרות, קורה באופן שגרתי מצב שבו Process פונה לכתובת חוקית בזיכרון שלו, אולם אין כזו כתובת פיזית ב-RAM.
2. לכל Process יש מרחב זיכרון משלו – נניח שהצלחנו להכניס את כל מרחב הכתובות הפיזיות של Process אל תוך ה-RAM. ברגע שיבוצע Context Switch אל Thread מתוך Process אחר, יהיה צריך להעביר ל-RAM את כל הזיכרון של ה-Process החדש. כמובן שהחלפה של כל המידע ב-RAM לוקחת זמן משמעותי למדי ואינה פתרון הגיוני.
3. יחידת הסגמנטציה שמבצעת את בדיקת מעגלי ההרשאה עבור ה-Protected Mode אכן מוצלחת למדי בשביל למנוע מ-Process להגיע באופן לא מבוקר לאזורי זיכרון בעלי רמות הרשאה גבוהות ממנו, אולם כבר אינה מבצעת את תפקידה ההיסטורי הנוסף, למנוע מ-Process לגשת לכתובות פיזיות ששייכות ל-Process אחר בעל אותה רמת הרשאה ולשנות את המידע שנמצא בהן, בטעות או באופן מכוון. יש צורך למנוע מ-Process שרץ ב-Userland לגשת באופן לא מבוקר לזיכרון של Process אחר שרץ ב-Userland.

זיכרון וירטואלי הוא שיטה שמאפשרת להתגבר על הבעיות הללו. הרעיון הכללי הוא שכאשר Process יפנה לכתובת הלינארית שנוצרה על ידי יחידת הסגמנטציה , זו לא תהיה כתובת פיזית אלא כתובת וירטואלית, כתובת לא אמיתית, שרק דרך מנגנון של תרגום כתובות ניתן להפוך לכתובת פיזית אליה המעבד ניגש.
נמחיש את הרעיון: הרגיסטר EIP מצביע על הפקודה הבאה בזיכרון, וכמובן שהוא מיוחס ל-Segment הקוד, כך שהכתובת הלוגית היא מסוג CS:EIP. הכתובת הלוגית הזו מתורגמת על ידי יחידת הסגמנטציה לכתובת לינארית, נניח 0x401300 (אפשר לכתוב אותה גם כ-0x00401300, האפסים המובילים אינם משנים, לעיתים נשתמש בצורת כתיבה עם אפסים מובילים כשנרצה לעשות חישובים של 32 ביט).
מה שה-Process כלל אינו מודע אליו, הוא שהוא לא באמת ניגש לזיכרון שבכתובת 0x401300. זוהי כתובת וירטואלית. במקום זאת, יש מנגנון שמחליף את הכתובת 0x401300 בכתובת אחרת, נניח 0x7145300 איך מתבצעת ההחלפה? ואיך מערכת ההפעלה דואגת לכך שלאחר ההחלפה ה-Process אכן יגיע לזיכרון הפיזי שהוא חושב שהוא נמצא בו? מי שמבצע זאת הוא מנגנון ה-Paging.

הבסיס של כתובות וירטואליות הוא שלכל Process יש מרחב כתובות וירטואליות משלו. בפרק המבוא, כאשר חקרנו את מרחב הכתובות של Process, למעשה חקרנו את מרחב הכתובות הוירטואליות של ה-Process.
חלק מהזיכרון של Process מוקצה למערכת ההפעלה ,ב-Windows נמצא את מערכת ההפעלה לרוב בכתובות שמעל 0x80000000, כלומר מרוחב הכתובות מחולק חצי-חצי בין מערכת ההפעלה ל-Process.
לכן במקרה הנפוץ של Windows 32 bit מרחב הכתובות הוירטואליות של Process נראה כך:

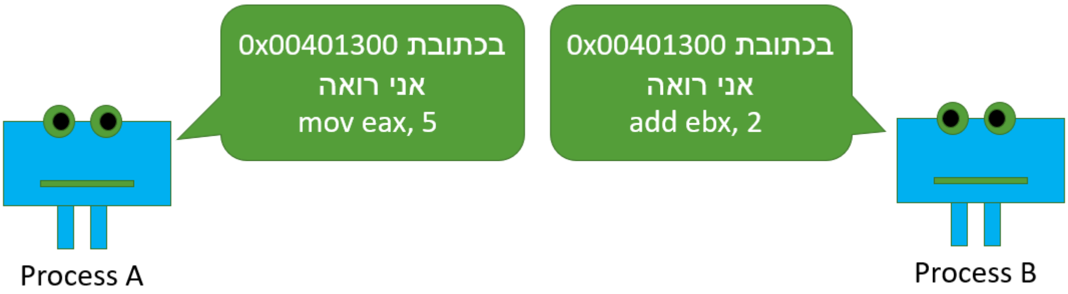


לחלוקה של מרחב הזיכרון בין ה-Process לבין מערכת ההפעלה צריך להוסיף שתי הסתייגויות:

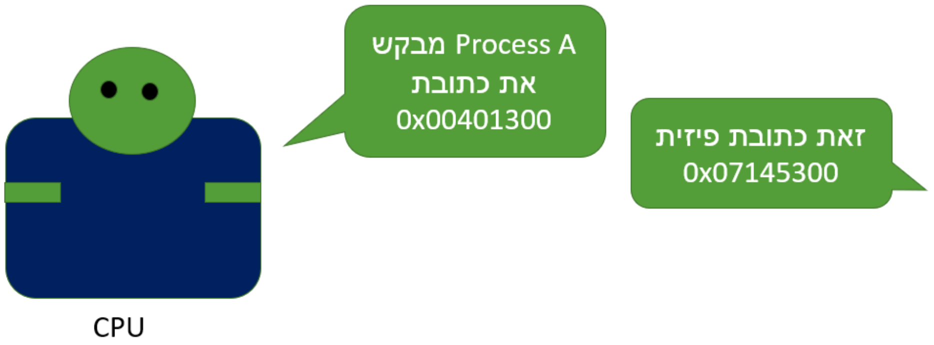
א. ישנה גם חלוקה אחרת, שבה 3GB זיכרון וירטואלי שייכים ל-Process ורק 1GB למערכת ההפעלה.

ב. בעקבות באג Meltdown, שאיפשר ל-Thread שרץ ב-Userland לגשת ולשנות את מרחב הזיכרון של מערכת ההפעלה, פותחו צעדי התמודדות שונים שמגבילים את היכולת של Thread לגשת לכל מרחב הכתובות הוירטואלי שתיאורטית עומד לרשות ה-Process שהוא נמצא בו. לדוגמה, ב-Linux מיושם מנגנון Kernel Page Table Isolation (בקיצור KPTI), וב-Windows יש מנגנון דומה במהותו שנקרא Kernel Virtual Address Shadow (בקיצור KVA Shadow). הרעיון הכללי של הפתרונות הללו הוא ליצור ל-Process מרחב כתובות וירטואלי שמכיל רק חלק קטן ממרחב הכתובות של ה-Kernel, כך שאין ל-Thread ששייך לו כל אפשרות לגשת למקומות בלתי רצויים. כאשר מבוצע Sysenter אז מתבצע מעבר מרחב זיכרון אחר, שבו כן ניתן לגשת למלוא מרחב הזיכרון של ה-Kernel.

נשוב להסבר אודות כתובות וירטואליות. מבחינת כל Process, כל הכתובות בזיכרון שייכות לו ורק לו. ה-Process לא מודע לכך שאותה הכתובת הוירטואלית קיימת גם עבור Process אחר, ומערכת ההפעלה מבצעת המרות והחלפות כתובות מאחרי הקלעים. כך לדוגמה, אם יש שני Processים על המחשב, לשניהם תהיה כתובת וירטואלית 0x401300, אך יש סיכוי גבוה שכל Process יראה בכתובת הזו מידע שונה, מכיוון שלכל Process מבוצע תרגום אחר בין הכתובות הוירטואלית לכתובות הפיזיות. מיד נבין כיצד התרגום מבוצע.

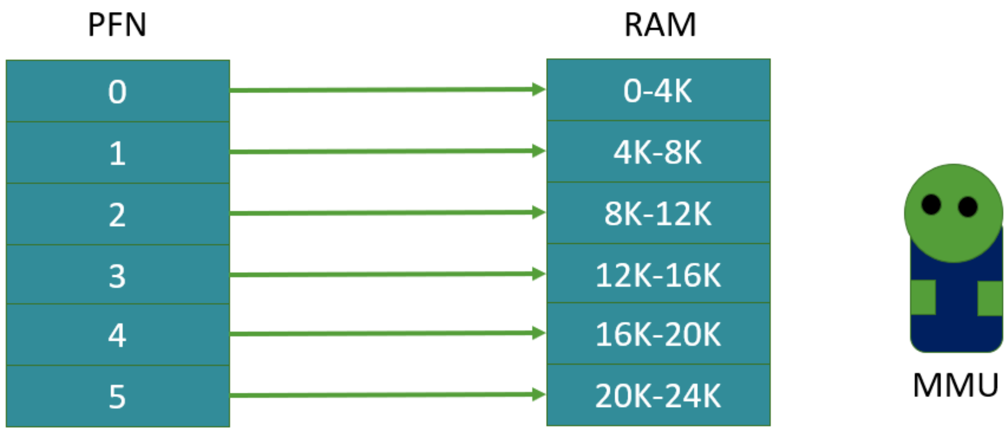


ההמרה בין כתובת וירטואלית ופיזית מתבצעת באמצעות רכיב בשם MMU, קיצור של Memory Management Unit. כאשר ה-CPU נתקל בכתובת וירטואלית כלשהי שעליו לגשת אליה, הוא מעביר את הכתובת לפענוח ב-MMU.



6.2.1 יחידת ה-Paging והמרה ל-Physical Address

ה-MMU מחלק את הזיכרון הפיזי למקטעים. כל מקטע נקרא "Page Frame", לעיתים קרובות פשוט קוראים להם "Frames". ה-Page Frames יכולים להיות בגדלים שונים, חזקות של 2. האפשרויות הן 4KB, 2MB או 4MB. אנחנו נשתמש בדיון שלנו ב-Page Frames בגודל הנפוץ של 4KB. כל Page Frame מקבל מספר מזהה שנקרא PFN, קיצור של Page Frame Number. האיור הבא ממחיש את חלוקת הזיכרון ל-Page Frames כאשר לכל Page Frmae ישנו מספר מזהה PFN.



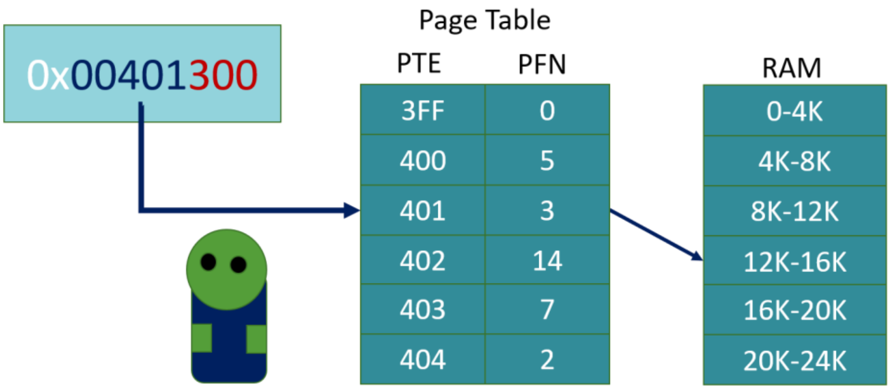
ל-MMU יש טבלה, שמאפשרת לו להמיר בקלות כתובת וירטואלית לכתובת פיזית. כל שורה בטבלה כוללת PFN שמצביע על זיכרון ה-RAM, ו-PTE שהוא פשוט מספר השורה בטבלה (PTE הוא קיצור של Page Table Entry). האיור הבא הוא המחשה לקטע מהטבלה:

Page Table

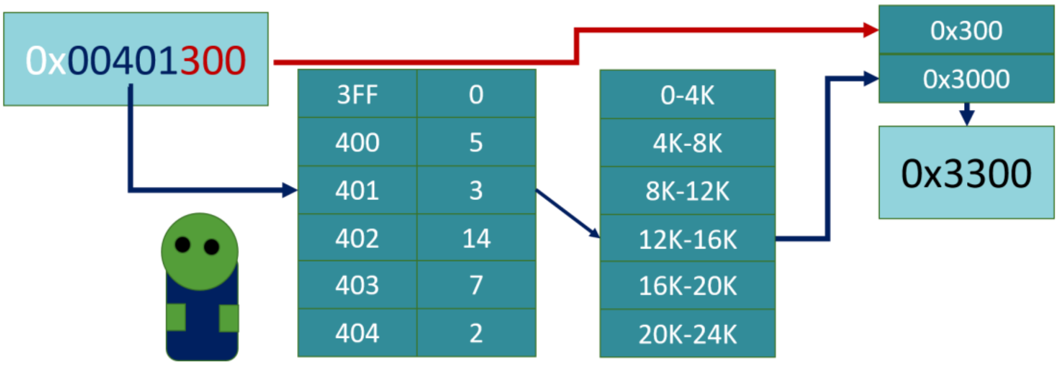
PTE	PFN
3FF	0
400	5
401	3
402	14
403	7
404	2

עד כה לא נראה שעשינו משהו מיוחד. יצרנו טבלה שכוללת אינדקס (PTE) ומצביע למקום בזיכרון ה- RAM (PFN). כעת נראה איך זה מסייע להמיר כתובת וירטואלית לפיזית. נחלק את הכתובת הוירטואלית לשני חלקים: החלק הראשון ישמש אותנו כדי להכנס למקום הנכון בטבלה, וכך להגיע ל-Page Frame הנכון ב-RAM. החלק השני יהיה כמה בתים אנחנו צריכים להוסיף מתחילת ה-Page Frame כדי להגיע בדיוק לכתובת המבוקשת.

האיור הבא ממחיש כיצד החלק הראשון של כתובת וירטואלית של 32 ביט מסייע לנו להגיע ל-Page Frame הנכון ב-RAM:



בדוגמה זו, חמשת הספרות הראשונות של הכתובת הוירטואלית מייצגות את ה-PTE שערכו 0x00401. ה-PTE מצביע על PFN שערכו 3, כלומר על ה-Page Frame ב-RAM שכולל את הכתובות בין 12K ל- 16K. במילים אחרות ה-Page Frame מתחיל בכתובת הקסדצימלית 0x3000 (אם זה לא ברור לכם, השתמשו באפליקציית calc במצב Programmer והמירו 12 כפול 1024 למספר הקסדצימלי). שלושת הספרות האחרונות של הכתובת מייצגות את ההיסט של הכתובת מתחילת ה-Page Frame, במקרה זה 0x300, ולכן הכתובת הפיזית שנגיע אליה היא 0x3300.



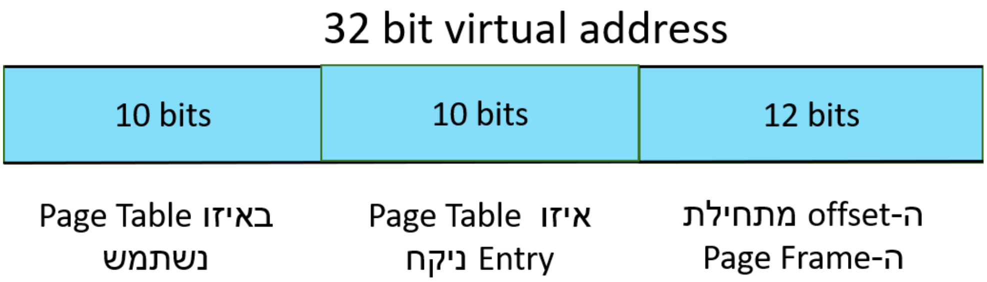
מדוע הכתובת הוירטואלית מחולקת דווקא ל-5 ספרות הקסדצימליות (20 ביטים) שמייצגות PTE ו-3 ספרות (12 ביטים) שמייצגות היסט בתוך ה-Page Frame? נזכור שהנחת המוצא שלנו היא שבחרנו באפשרות שבה גודל Page Frame הוא 4096 בתים, או 0x1000, כך שההיסטים האפשריים בתוך Page Frame הם בין 0x000 ל- 0x999, שלוש ספרות הקסדצימליות. אילו היינו חורגים משלוש ספרות הקסדצימליות עבור ההיסט, היינו מקבלים היסט גדול מגודל Page Frame ואז היה ניתן לבטא את אותה כתובת בזיכרון באמצעות כמה כתובות וירטואליות. כיוון שאותה כמות ביטים משמשת לייצוג של כתובות פיזיות ושל כתובות וירטואליות, כל כתובת פיזית חייבת להיות מיוצגת על ידי כתובת וירטואלית יחידה. אם יש כתובות פיזיות שמיוצגות על ידי מספר כתובות וירטואליות, אז יש כתובות פיזיות שלא ניתן להגיע אליהן על ידי כתובת וירטואליות כלל.

6.2.2 חלוקה ל-Page Directories

בשיטה שסקרנו עד כה להמרת כתובות וירטואליות לפיזיות יש בעיה: חישבו: מה גודל ה-Page Table שצריך ה-MMU לטובת ביצוע ההמרות? בכתובת הוירטואלית מוקצים לטובת ההיסט 12 ביט, לכן במערכת של 32 ביט יוותרו 20 ביטים שמבטאים את ה-PTE. ה-20 ביטים הללו מיתרגמים ל-2 בחזקת 20 כניסות לטבלה, או למעלה ממיליון כניסות. כלומר ה-MMU צריך ליצור טבלה בת מיליון רשומות רק לטובת ההמרה, כפי שאתם יכולים להניח טבלה זו צריכה להשתנות עבור כל Process. התוצאה היא בזבוז עצום של זיכרון, רק לטובת שמירת ה-Page Tables של כל ה-Process‏ים. במערכת של 64 ביט גודל ה-Page Table יהיה כה עצום שהוא עלול להיות יותר גדול מהזיכרון הפיזי שעומד לרשותנו כדי לשמור אותו.

כדי למנוע את הבעיה הזו, במקום שיהיה לנו Page Table אחד ענק, ניצור הרבה Page Tables קטנים יותר. לכאורה הרבה טבלאות קטנות תופסות את אותו מקום בזיכרון של טבלה אחת גדולה ולכן לא חסכנו זיכרון, אבל למעשה ברוב המקרים ל-Process יש מרחבי זיכרון לא מנוצלים ורק לעיתים נדירות Process תופס את כל הזיכרון הוירטואלי שעומד לרשותו. התכונה הזו מאפשרת לנו ליצור עבור Process חדש Page Table אחד קטן, ואם ה-Process צריך עוד זיכרון נוסיף לו עוד Page Table ועוד אחת, עד שנשביע את דרישות הזיכרון שלו. כעת כשיש לנו הרבה Page Tables קטנים, מספרי ה-PTE צפויים לחזור על עצמם. לדוגמה ב-Page Table מספר 0, ה-PTE שמספרו 0x401 יכול להצביע על זיכרון פיזי שמתחיל בכתובת 0x00003000, ואילו ב-Page Table מספר 1, ה-PTE שמספרו הוא גם כן 0x401 יצביע על כתובת אחרת לגמרי בזיכרון הפיזי, לדוגמה 0x02100000. אין קשר בין הכתובות הפיזיות.

לכן מערכת ההפעלה צריכה לדעת לאיזה Page Table בדיוק מתייחס ה-PTE שיש בכתובת הוירטואלית, ובשביל זה יש לה טבלה נוספת: Page Directory. ה- Page Directory קובעת לאיזה Page Table צריך להגיע כדי לפענח את הכתובת ה-PTE. כך נראית החלוקה עבור כתובות של 32 ביט:



תרגיל 6.1 פירוק כתובת וירטואלית למרכיביה

פרקו את הכתובת הוירטואלית 0x01234567 למרכיביה השונים:

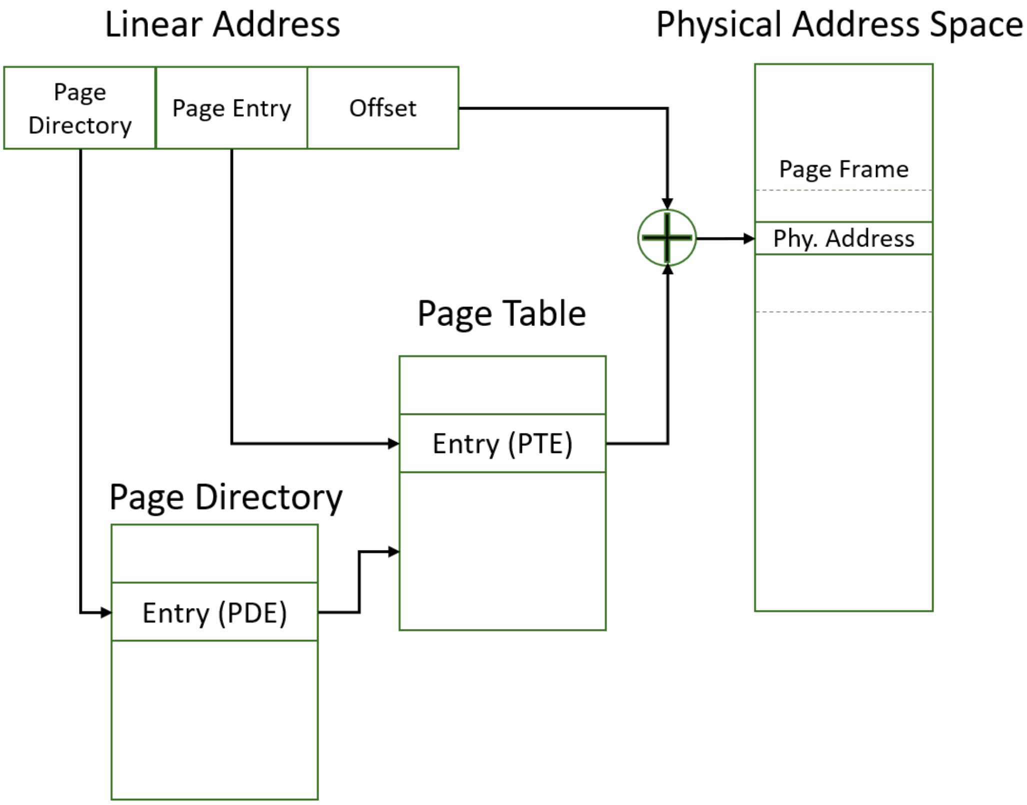
```
Page Directory Entry

Page Table Entry

Page Frame Offset
```

פתרון:

0000 0001 0010 0011 0100 0101 0110 0111

Page Directory Entry	–	0000 0001 00	= 0x4
Page Table Entry	–	10 0011 0100	= 0x234
Page Frame Offset	–	0101 0110 0111	= 0x567

כלומר כדי להמיר את הכתובת לכתובת פיזית, יש לגשת אל רשומה מספר 4 ב-Page Directory ברשומה זו נמצאת הכתובת של ה-Page Table, ולאחר שנגיע אל ה-Page Table נוציא את כתובת תחילת ה- Page Frame שנמצאת ברשומה מספר 0x234 ונוסיף לה את ההיסט 0x567 כדי לחשב את הכתובת הפיזית.



שאלה למחשבה: במקרים רבים כשחוקרים תוכנה באמצעות IDA, מגלים שהקוד שלה מתחיל בכתובת וירטואלית 0x401000. זוהי כתובת ברירת המחדל לתחילת אזור הקוד בקבצי EXE ו-DLL, נלמד על כך בהמשך. מה מיוחד דווקא בכתובת זו?

בואו נפרק את הכתובת הזו כפי שלמדנו. כיוון שזו כתובת של 32 ביט, למעשה "חסרים" שני אפסים בתחילת הכתובת. בצורה המלאה הכתובת היא 0x00401000. כעת נכתוב אותה בתור רביעיות של ביטים

0000 0000 0100 0000 0001 0000 0000 0000

נפריד את החלקים השונים. עשרת הביטים הראשונים:

0000 0000 01

עשרת הביטים הבאים:

00 0000 0001

שנים עשר הביטים האחרונים:

0000 0000 0000

כלומר אנחנו רואים שברירת המחדל עבוד כתובת תחילת אזור הזיכרון שבו נמצא הקוד היא לגשת ל- Page Directory Entry הראשון בתוך ה-Page Directory, לאחר שהגענו באמצעותו ל-Page Table צריך לגשת בתוכו ל-Page Table Entry הראשון, ולאחר שהגענו באמצעותו ל-Page Frame צריך להוסיף לו היסט אפס.

6.2.3 תהליך Context Switch בין Processים

זה הזמן להכיר את הרגיסטר CR3. במעבדי אינטל 32 ביט ישנם שמונה Control Registers, או בקיצור CR, הממוספרים מ-0 עד 7 (מעבד ה-64 ביט כולל גם CR8). רגיסטרים אלו שולטים אלו על מגוון פעולות הקשורות להפעלה של המעבד עצמו. לדוגמה:

- משפחת מעבדי אינטל ידועה בכך שיש תאימות לאחור בין דורות המעבדים שלה. כלומר, גם מעבד מודרני יכול להריץ תוכנות שנכתבו עבור מעבד מוקדם. כפי שהזכרנו, בראשית משפחת ה-x86 המעבדים תמכו ב-Real Mode. החל ממעבד ה-286 החלה להיות תמיכה ב-Protected Mode והחל ממעבד ה-386 החלה להיות תמיכה במנגנון ה-Paging שאנחנו סוקרים כרגע. גם כיום, כאשר מחשב נדלק, המעבדים שלו עולים קודם כל במצב Real Mode ורק לאחר מכן עוברים ל-Protected Mode. פעולת הבקרה הזו מבוצעת על ידי הביט הראשון ברגיסטר CR0.
- עד כה כשדנו על Page Frame הנחנו שגודלו הוא 4KB. אך גודל ה-Page יכול להיות שונה. הביט האחרון ברגיסטר CR0 קובע אם ניתן לשנות את גודל ה-Page.

אחד מהרגיסטרים הללו, CR3, כולל שדה בשם Page Directory Base Address. זוהי הכתובת של ה-Page Directory, היכן שמתחיל כל תהליך התרגום מכתובת לינארית (וירטואלית) לכתובת פיזית. כל מה שצריך לעשות כדי ליצור תרגום שונה לגמרי הוא רק להחליף את ה-Page Directory שמשמש לתרגום.

שאלה למחשבה: מה דעתכם, האם הכתובת שנמצאת ב-CR3, הכתובת של ה-Page Directory, היא כתובת וירטואלית או פיזית?

ובכן, זו חייבת להיות כתובת פיזית. הרי כל תהליך תרגום כתובת וירטואלית לפיזית מתחיל מהכתובת הזו, שצריכה להצביע על ה-Page Directory הנכון. לו הכתובת הזו היתה וירטואלית, המצב שהיה נוצר הוא שהכתובת ב-CR3 מתרגמת את עצמה...

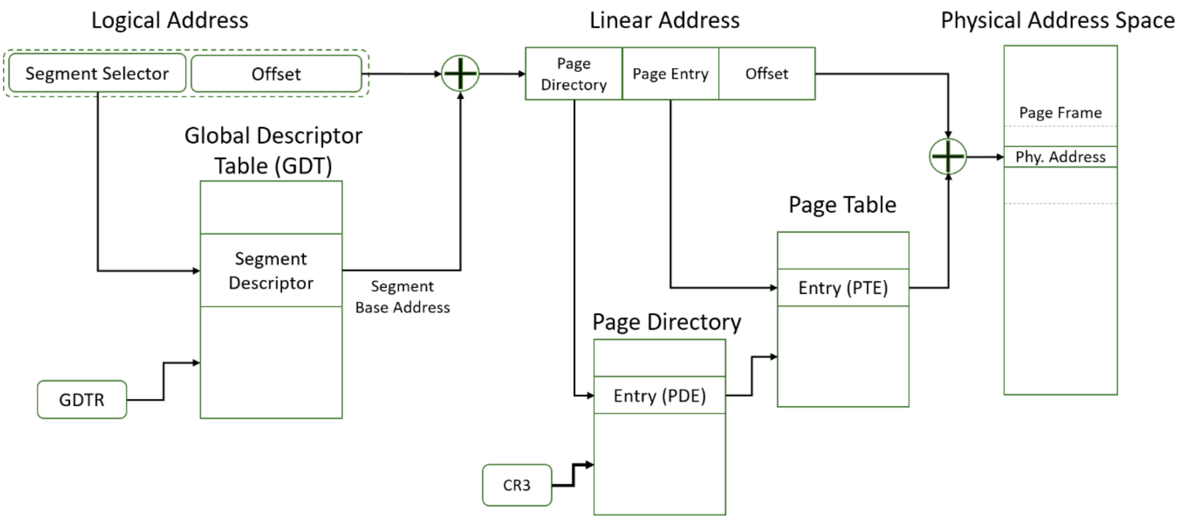
עכשיו אפשר לחבר יחד את כל הידע שלנו. למערכת ההפעלה יש Page Directory עבור כל Process שהיא מריצה, בכל פעם ש-Process חדש נוצר, נוצר עבורו Page Directory, שיאפשר ל-Process לגשת לאזורים שמערכת ההפעלה הקצתה לו ב-RAM. כל מה שמערכת ההפעלה צריכה לעשות כשהיא מבצעת Context Switch הוא להחליף את הרגיסטרים של המעבד. בעקבות ההחלפה הזו יתרחשו שני דברים:

1. הרגיסטר EIP שמחזיק את מיקום הפקודה הבאה יצביע על המקום בו ה-Process עצר את ריצתו בפעם האחרונה שמערכת ההפעלה נתנה לו זמן מעבד.
2. הרגיסטר CR3 שמחזיק את המפתח לכל הזיכרון של ה-Process יעבור להצביע על דפי הזיכרון של ה-Process החדש. כך אין צורך "להחליף" את הזיכרון הפיזי ב-RAM, דבר שיכול לקחת זמן ניכר. מספיק רק להחליף רגיסטר אחד.

חישבו- מה קורה כאשר מבוצע Context Switch בין Threadים ששייכים לאותו Process?

במקרה זה יש צורך להחליף את ערכי הרגיסטרים השונים אך לא את CR3. כך, ה-Threadים מבצעים כל אחד משימה אחרת (יש להם EIP שונה) אך הם חולקים מרחב זיכרון זהה (יש להם CR3 זהה).

לסיכום, הנה התמונה המלאה של תהליך המרת הכתובות, החל מ-Logical Address וכלה ב- Physical Address (מבוסס על תיעוד IA-32 Processor Architecture של אינטל):



6.2.4 מנגנון ה-TLB

אם כל ההבדל בין Context Switch של Processים ל-Context Switch של Threadים הוא שמירת או החלפת הרגיסטר CR3, מדוע ביצוע Context Switch בין Processים נחשב לפעולה שלוקחת זמן רב יותר?

התשובה נעוצה ברכיב שטרם הכרנו. כפי שראינו, פעולת ההמרה בין כתובת לוגית לפיזית דורשת שלבים רבים, כוללת מספר גישות לזיכרון ופעולות חיבור. השלבים הללו לוקחים זמן. לכן למעבד יש טבלה שבה נשמרים תרגומים של כתובות לוגיות נפוצות ישירות לכתובות פיזיות. הטבלה הזו, שנקראת TLB, קיצור של Translation Lookaside Buffer, היא כמו פתקים צהובים שהמעבד מניח לעצמו על שולחן העבודה. מה שנמצא בפתקים חוסך זמן חיפוש בספר עבה. ה-TLB הוא כמו Cache, אבל רק של כתובות, והוא מאיץ באופן משמעותי את הרצת הקוד.

כאשר מתבצע Context Switch בין Processים, המנגנון הזה נעצר בחריקת בלמים, או ליתר דיוק- נמחק ומתחיל מאפס. כיוון של-Process החדש יש מיפוי שונה לגמרי בין כתובות לוגיות לפיזיות, כל מה שרשום על ה"פתקים הצהובים" לא נכון יותר .

כדי למנוע שימוש ב-TLB של Process אחר, מה שכמובן יגרום לשגיאות, יש צורך להחליף את ה-TLB בכל פעם שמחליפים Process. כדי לבצע זאת יש למעבד טריק נחמד: ברמת החומרה הרגיסטר CR3 מחובר חשמלית אל ה-TLB, וכל שינוי שיש ב-CR3 שולח זרם חשמלי ומוחק את ה-TLB. לכן התשלום הכבד של Context Switch בין Processים אינו החלפת הזיכרון, אלא הצורך ליצור מחדש את הכתובות השימושיות ב-TLB. כמובן, שכל רכיב Cache שיש לנו בהמשך הדרך, ספציפית L1, L2, L3 Cache, יצטרכו גם הם לעבור ריענון לפני שיהיו שימושיים.

הבעיה של השפעת מחיקת ה-TLB על הביצועים לא נעלמה מעיניהם של יצרני המעבדים ומעבדים החלו לכלול מנגנון שמאפשר למנוע מחיקה של חלק מה-TLB. לדוגמה, לסמן Page Entries שהשימוש בהם נפוץ כך שיישארו ב-TLB ולא יימחקו. החיסרון הוא שהדבר עלול לחשוף את המעבדים לבעיות אבטחה, כיוון שאם ה-TLB לא נמחק במלואו אז נפתח פתח, ולו תיאורטי, לשימוש בו על מנת לגשת ל-Page Frames של Process אחר ואף של ה-Kernel. כדי לנקות באופן מוחלט את ה-TLB אפשר להשתמש בפקודת האסמבלי INVLPG.

לקריאה נוספת אודות בעיות אבטחה שעלולות להיווצר בעקבות אי מחיקה מלאה של ה-TLB:

https://www.usenix.org/conference/usenixsecurity18/presentation/gras

6.2.5 תפקידים נוספים של Page Table ו-Page Directory

כפי שאפשר לראות באיור הבא, בנוסף לחלק שלהם בפענוח כתובות וירטואליות, ל-Page Directory ול-Page Table יש גם תפקידים נוספים. לכל רשומה שם מתווספים כמה שדות מידע מעניינים.

32-Bit Page Directory Entry (PDE)

20 bit page table base address	Avail	G	PS	0	A	PCD	PWT	U/S	W	P

32-Bit Page Table Entry (PTE)

20 bit page base address	Avail	G	PAT	D	A	PCD	PWT	U/S	W	P

רוב השדות דומים למדי בשמות שלהם, כאשר ההבדל העיקרי הוא אם מה שהם מסמנים תקף עבור Page ספציפי או עבור אוסף של Pages. אנחנו נתמקד בשלושת השדות האחרונים- U/S, W ו-P.

שדה ה- User/Supervisor (או בקיצור U/S) מבצע בדיקה נוספת של מעגלי ההרשאה. שדה זה כולל רק ביט אחד – 0 מציין שהדף דורש הרשאה של Kernel, ו-1 מציין את כל יתר האפשרויות. כלומר בשלב זה אין הפרדה בין מעגלים 1,2,3 אלא רק בין מעגל 0 לכל היתר.

שדה ה- Write (או בקיצור W) מציין אם ניתן לכתוב לזיכרון או שכל הזיכרון שה-Page Table מצביע עליו הוא Read Only.

שדה ה- P, קיצור של Present, מציין אם הזיכרון אכן קיים ב-RAM. נחשוב על מצב שבו מסיבה כלשהי ה-Process דורש יותר זיכרון מאשר קיים ב-RAM. דוגמה פשוטה היא אם גודלו של ה-RAM במערכת 32 ביט הוא 1GB, ואילו Process כלשהו מנסה לנצל את מלוא טווח הכתובות המקסימלי שיכול להיות לProcess של 32 ביט, 4GB (כולל האזור שמוקדש למערכת ההפעלה). במקרה זה כמובן שאין אפשרות להעמיד כתובת פיזית מאחרי כל כתובת וירטואלית.

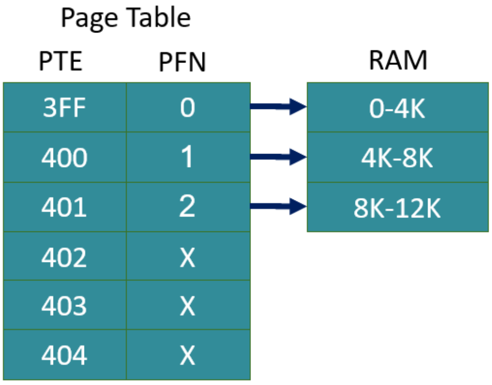
כיום גודל RAM של 4GB הוא מובן מאליו, אך בו זמנית גדלה גם כמות הזיכרון הוירטואלי של Processים של 64 ביט, כלומר בעיית כמות ה-RAM היתה ונותרה. בנוסף, אף אחד לא מבטיח שכל ה-RAM ינוצל לטובת Process אחד בלבד. להיפך, כדי להריץ בצורה מהירה ככל שניתן מספר Processים, רצוי שהם יחלקו ביניהם את ה-RAM. התוצאה היא שהניצול בפועל של ה-RAM עבור כל ה-Processים צפוי לעיתים לעלות על גודל הזיכרון הפיזי. הפתרון של מערכות ההפעלה: להקצות מקום על הדיסק הקשיח ולשמור שם את כל הדפים שאין להם מקום ב-RAM. כמובן שגישה אל Page שנמצא על הדיסק הקשיח תיקח זמן רב, אך אם אין עוד RAM הרי שזו האפשרות הכי פחות גרועה.

השדה P מאפשר למערכת ההפעלה לדעת אם ה-Page המבוקש אכן נמצא ב-RAM או שהוא הועתק לדיסק הקשיח.

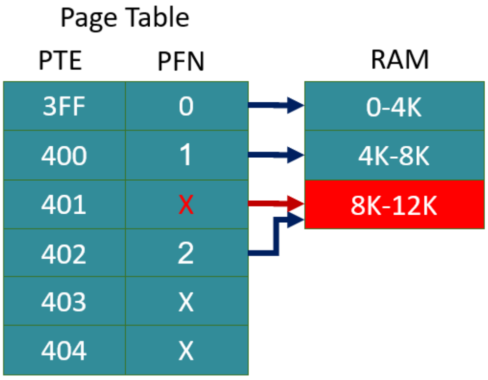
6.2.6 שגיאת Page Fault

מצב שבו ה-Page Frame לא נמצא ב-RAM נקרא Page Fault. למרות ש-Page Fault גורם לבזבוז זמן רב, זהו הליך שגרתי ושכיח מאד.

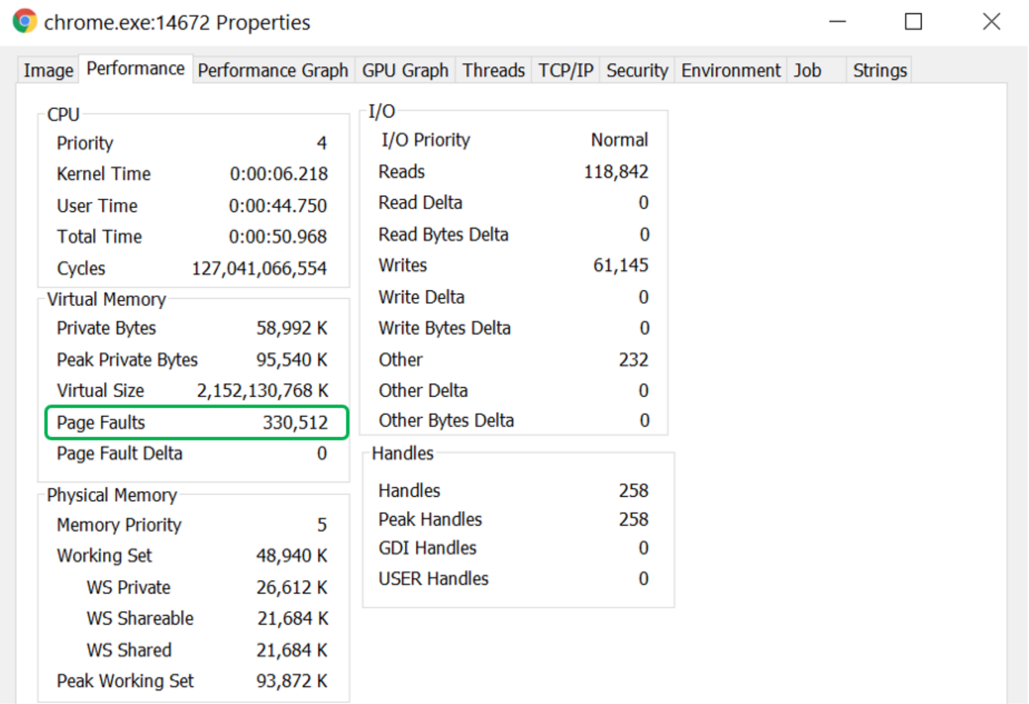
הדוגמה הבאה ממחישה Page Fault. ניקח לצורך הדוגמה מצב תיאורטי בו גודל הזיכרון הפיזי הוא 12KB בלבד, כלומר כולל שלושה Page Frames בלבד. הזיכרון הפיזי קטן מהזיכרון הוירטואלי ולכן חלק מה-PTEים אינם ממופים ל-PFNים ואינם מצביעים על מיקום כלשהו ב-RAM. ה-Process לא מודע לכך, ומבקש כתובת שנמצאת ב-PTE מספר 402.



שדה ה-P יאפשר למערכת ההפעלה לדעת שישנה בעיה שיש לתקן אותה. מערכת ההפעלה תצטרך לבחור באופן כלשהו Page Frame מתוך ה-RAM ולהחליף את התוכן שלו בתוכן של ה-Page Frame המבוקש. לאחר מכן יעודכן ה-Page Table כך ש-PTE מספר 402 יצביע על ה-Page Frame המבוקש, ואילו PTE מספר 401 כבר אינו עדכני ויצטרך להימחק:



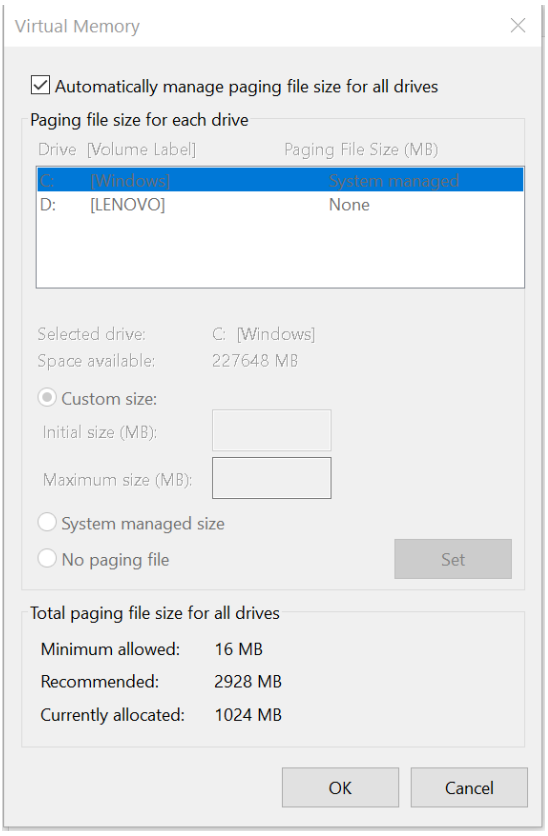
הכנסו אל Process Explorer, בחרו Process כלשהו ובידקו את ה-Properties שלו. תחת טאב Performance תוכלו לראות את כמות ה-Page Faults שיש לו. בהזדמנות זו, תוכלו להתרשם מההבדל בין גודל הזיכרון הוירטואלי שה-Process חושב שעומד לרשותו, לבין גודל הזיכרון הפיזי ב-RAM שעומד לרשותו בפועל (הקרוי Working Set).



6.2.7 הקובץ Pagefile.sys

כדי לשמור את ה-Page Frames שאין להם מקום ב-RAM, מערכת ההפעלה Windows משתמשת בקובץ בשם pagefile.sys, זהו קובץ שנמצא בתיקיית הבסיס של כונן C אולם הוא מוגדר כקובץ נסתר.
יש מצבים שבהם אין צורך להשתמש ב-pagefile.sys כדי לשמור מידע שצריך להוציא מה-RAM. אם לדוגמה הרצנו תוכנית כלשהי ששמורה על הדיסק הקשיח שלנו, הרי שכל הקוד שלה כבר נמצא ממילא על הדיסק הקשיח. הגיוני שנרצה לשמור ל-pagefile.sys אזורי זיכרון שיש להם הרשאות קריאה וכתיבה, כיוון ששם עשוי להיות הבדל בין מה שנמצא ב-RAM לבין מה שנמצא על הדיסק. מערכת ההפעלה עשויה לשמור אינדיקציה ש-Page Frames מסויימים הם "Dirty", כלומר שלא רק שיש להם הרשאות כתיבה אלא שבוצעה אליהם כתיבה בפועל ולכן הם שונים מהמידע שקיים בקובץ בדיסק הקשיח. במקרים אלו אכן אין ברירה ואם אין מקום ב-RAM חייבים לשמור אותם ה-pagefile.sys.

אפשר לשנות את ההגדרות של pagefile.sys (לדוגמה את הגודל המקסימלי המוקצה לקובץ) באמצעות גישה ל- Control Panel ומשם System Properties, לאחר מכן Advanced, משם Performance Settings, שוב פעם Advanced ולבסוף נגיע אל תפריט Virtual Memory. מחיקת הקובץ הזה עלולה לגרום לבעיות קשות וחוסר יציבות ומיקרוסופט עבדו קשה כדי להקטין את הסיכוי שמישהו יגיע אל הגדרות הקובץ וישנה אותן...



6.2.8 תוכנת VMMap

כעת כשאנחנו מבינים מהו זיכרון וירטואלי אנחנו יכולים להכיר כלי נוסף של Sysinternals, שנקרא VMMap, קיצור של Virtual Memory Map.

ראשית גשו לשורת החיפוש של Windows, והקלידו

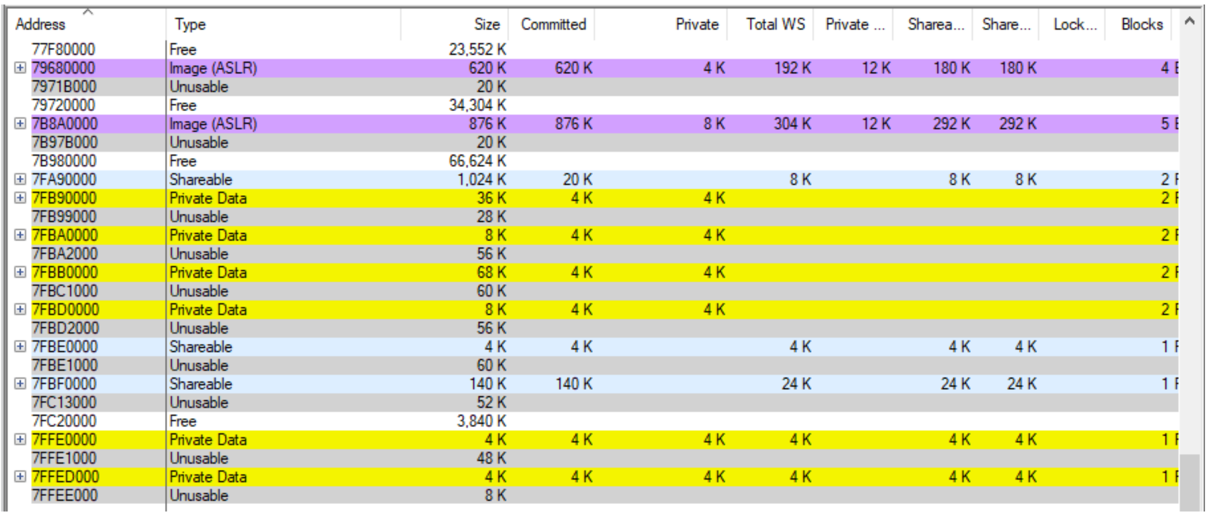
\windows\syswow64\notepad.exe

זוהי גרסת ה-32 ביט של תוכנת notepad. כל התוכנות בגרסת 32 ביט שמורות בתיקיה זו של Windows. זו הזדמנות לספר מעט על המעבר מ-32 ל-64 ביט. למרות שבמבט ראשון נראה שזו דווקא תיקיה של תוכנות 64 ביט, זו אכן תיקיה של תוכנות 32 ביט. המשמעות של SysWow64 היא "Windows 32 bit on 64 bit". מיקרוסופט בחרה בשם המבלבל הזה מכיוון שטרם פיתוח מערכת הפעלה 64 ביט, תוכנות עשו שימוש בתיקיית System32 ובחלק מהתוכנות השם של התיקיה הזו נכתב בצורה קשיחה בתוך הקוד. כיוון שאוסף של תוכנות קומפלו כך שהן מחפשות קבצים שימושיים בתיקיית System32, הדרך לגרום להן להשתמש בקבצים הללו בגרסאות של 64 ביט היתה לשתול את גרסאות ה-64 ביט באותה תיקיה שלפני כן שימשה ל-32 ביט. אחרת, היה צורך לקמפל מחדש את כל התוכנות שעושות שימוש בתיקיה הזו. לכן מיקרוסופט בחרה להכניס את קבצי ה-64 ביט לתוך System32 ואת קבצי ה-32 ביט "להגלות" אל תיקיית SysWow64.

בכל אופן, אין חובה לעבוד דווקא עם תוכנת notepad בגרסת 32 ביט, אולם העובדה שבחרנו תכנית של 32 ביט תאפשר לנו לחוש יותר בנוח עם הצגת הזיכרון הוירטואלי.

כעת פיתחו את VMMap, ובחרו במסך בחירת התוכניות את notepad.

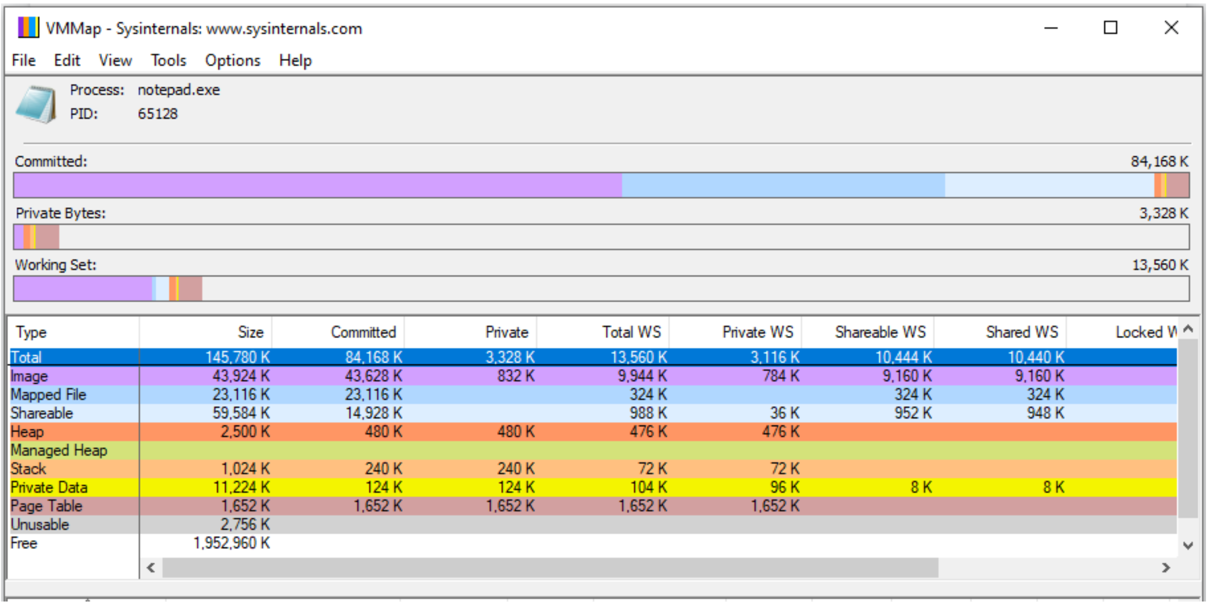
נתחיל מהחלק התחתון של המסך. אנחנו רואים לנגד עינינו את החלוקה של הזיכרון הוירטואלי של notepad לאזורי זיכרון שונים. בטור השמאלי נמצאת הכתובת הוירטואלית שבה מתחיל אזור הזיכרון.



Address	Type	Size	Committed	Private	Total WS	Private ...	Sharea...	Share...	Lock...	Blocks
77F80000	Free	23,552 K								
79680000	Image (ASLR)	620 K	620 K	4 K	192 K	12 K	180 K	180 K		4 E
7971B000	Unusable	20 K								
79720000	Free	34,304 K								
7B8A0000	Image (ASLR)	876 K	876 K	8 K	304 K	12 K	292 K	292 K		5 E
7B97B000	Unusable	20 K								
7B980000	Free	66,624 K								
7FA90000	Shareable	1,024 K	20 K		8 K		8 K	8 K		2 F
7FB90000	Private Data	36 K	4 K	4 K						2 F
7FB99000	Unusable	28 K								
7FBA0000	Private Data	8 K	4 K	4 K						2 F
7FBA2000	Unusable	56 K								
7FBB0000	Private Data	68 K	4 K	4 K						2 F
7FBC1000	Unusable	60 K								
7FBD0000	Private Data	8 K	4 K	4 K						2 F
7FBD2000	Unusable	56 K								
7FBE0000	Shareable	4 K	4 K		4 K		4 K	4 K		1 F
7FBE1000	Unusable	60 K								
7FBF0000	Shareable	140 K	140 K		24 K		24 K	24 K		1 F
7FC13000	Unusable	52 K								
7FC20000	Free	3,840 K								
7FFE0000	Private Data	4 K	4 K	4 K	4 K		4 K	4 K		1 F
7FFE1000	Unusable	48 K								
7FFED000	Private Data	4 K	4 K	4 K	4 K		4 K	4 K		1 F
7FFEE000	Unusable	8 K								

אם תגללו את המסך למטה, תראו שהכתובות מגיעות עד 0x7FFEE000. ישנו אזור זיכרון קטן שאינו בר שימוש ולאחר מכן יתחילו כתובות שמתחילות ב-0x8000000. הכתובות הללו כזכור שייכות ל-Kernel, כך שהן אינן מוצגות לפנינו.

נעבור לחלק העליון של המסך.



Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked W
Total	145,780 K	84,168 K	3,328 K	13,560 K	3,116 K	10,444 K	10,440 K	
Image	43,924 K	43,628 K	832 K	9,944 K	784 K	9,160 K	9,160 K	
Mapped File	23,116 K	23,116 K		324 K		324 K	324 K	
Shareable	59,584 K	14,928 K		988 K	36 K	952 K	948 K	
Heap	2,500 K	480 K	480 K	476 K	476 K			
Managed Heap								
Stack	1,024 K	240 K	240 K	72 K	72 K			
Private Data	11,224 K	124 K	124 K	104 K	96 K	8 K	8 K	
Page Table	1,652 K	1,652 K	1,652 K	1,652 K	1,652 K			
Unusable	2,756 K							
Free	1,952,960 K							

זיכרון שהוא Committed הוא כל מרחב הזיכרון הוירטואלי שקיים ל-Process ושיש מאחריו כתובות "אמיתיות". לכאורה, מרחב הזיכרון הוירטואלי של Process ב-32 ביט צריך להיות 4GB, אולם יש אזורים לא מנוצלים ולכן מערכת ההפעלה לא מגבה אותם בזיכרון פיזי. זיכרון Committed הוא זיכרון וירטואלי שמערכת ההפעלה מתחייבת לגבות או ב-RAM או ב-pagefile.sys שמגבה את ה-RAM. אין זה אומר שלכל הזיכרון

שהוא Committed יש גיבוי של זיכרון פיזי בכל רגע נתון. יכולה להיות כתובת וירטואלית שאין לה גיבוי פיזי ברגע מסוים, כלומר היא לא נמצאת לא על ה-RAM ולא ב-pagefile, אך אם ה-Process יבקש לגשת אליה מערכת ההפעלה תעתיק ל-RAM את המידע שהכתובת הוירטואלית מצביעה עליו. לסיכום, זיכרון Committed הוא זיכרון וירטואלי שמערכת ההפעלה מבטיחה שבמידת הצורך היא תגבה אותו עם זיכרון פיזי.

זיכרון Private Bytes הוא הזיכרון שה-Process ביקש ממערכת ההפעלה להקצות לו. מוזר שהזיכרון הזה קטן מה-Committed, הלא כן? ובכן, זיכרון Private הוא זיכרון שאינו כולל קבצים שעברו מיפוי (נלמד על כך בפרק הבא) ואינו כולל קבצי DLL (נלמד גם על כך בהמשך, כרגע לצורך הפשטות נחשוב עליהם בתור ספריות תוכנה מקומפלות שתוכנה יכולה לייבא). זיכרון ה-Committed כן כולל את הקבצים הללו. כך, אם תוכנית עושה שימוש ב-DLL, אז הזיכרון שתופס ה-DLL לא יופיע ב-Private Bytes אך כן יופיע ב-Committed.

זיכרון Working Set הוא הזיכרון שנעשה בו שימוש בפועל. אם ניגש אל ה-RAM ונבדוק כמה מהזיכרון שנמצא שם שייך ל-Process מסויים, נקבל את ה-Working Set שלו. כשדברנו על זיכרון Committed אמרנו שזהו זיכרון שמערכת ההפעלה מבטיחה לתת לו גיבוי של זיכרון פיזי. ה-Working Set הוא הזיכרון שמערכת ההפעלה לא רק מבטיחה אלא גם "מקיימת" ברגע מסויים.

נעבור על מפת הזיכרון:

- Image הוא הזיכרון המוקצה לקוד התוכנה
- Mapped File הינו זיכרון שמוקצה לקבצים שעברו מיפוי, כאמור על כך בפרק הבא
- Shareable הוא זיכרון שניתן לשיתוף. בפרק הבא נלמד שיתוף זיכרון, אפשר להגדיר זיכרון שמשותף ליותר מ-Process אחד. אם זיכרון מוגדר כ"בר שיתוף" הוא יופיע בתור Shareable
- Heap הוא זיכרון שמוקצה באופן דינמי במהלך הריצה, לדוגמה באמצעות הפונקציה malloc או VirtualAlloc
- Stack הוא המחסנית, האזור שמשמש לשמירת משתנים מקומיים, פרמטרים שמועברים לפונקציות וכתובות חזרה. זו הזדמנות להרחיב מעט על ריבוי מחסניות. כפי שהזכרנו בעבר, לכל Thread יש שתי מחסניות: אחת שנמצאת ברשותו ואחת שמשמשת את ה-Kernel כאשר ה-Thread מבקש לקרוא לפונקציות של ה-Kernel ומתבצע Sysenter.

בצעו ניסוי קטן: לפני כן וודאו שב-VMMap בחרתם Process של 32 ביט ושמערכת ההפעלה שלכם היא 64 ביט. הקליקו הקלקה כפולה על ה-Stack ב-VMMap. אתם יכולים לראות שלא רק שכל Thread יש מחסנית משלו, ישנה גם אזורי זיכרון שנקראים "64 bit thread stack". מערכת ההפעלה נותנת ל-Thread הדמיה שהוא רץ ב-32 ביט, אך לפעמים יש קריאה לפונקציות שהמימוש שלהן הוא בקוד של 64 ביט. לכן, מערכת ההפעלה זקוקה למחסנית נוספת לכל Thread, כזו שתשמור משתנים וכתובות בגודל 64 ביט. אם תפתחו ב-VMMap איזשהו Process שרץ ב-64 ביט, לדוגמה chrome.exe, לא תראו במחסנית אזורים שנקראים 64 ביט, שהרי כל המחסניות הן 64 ביט.

- Private data הוא הזיכרון המוקצה למשתנים של התוכנית שאינם במחסנית או ב-Heap. במילים אחרות, נמצא שם קבועים, משתנים גלובליים, משתנים סטטיים ומבנה זיכרון בשם Thread Environment Block. לא נתעמק במבנה הנתונים הזה, רק נציין שהוא כולל מידע שונה שה-Thead צריך, לדוגמה באמצעותו ה-Thread יכול לגשת אל אזור ה-Thread Local Storage שדנו עליו כשסקרנו מהם המשאבים הייחודיים לכל Thread.
- Page Table: ובכן, אם הגעתם עד כאן אתם כבר יודעים היטב מהו... מעניין לראות את גודלו של הזיכרון שמוקצה עבור ה-Page Table. כזכור, אלמלא היה נעשה שימוש ב-Page Directores היה נדרש זיכרון רב הרבה יותר. למרות שניתן להתרשם מ-VMMap כאילו הטבלאות נמצאות בזיכרון הוירטואלי של ה-Process, למעשה הן נמצאות במרחב הזיכרון של ה-Kernel. אחרת, היה ניתן לגשת באופן חופשי לטבלאות, לשנות אותן וכך להגיע לכל כתובת פיזית, גם כתובת שנמצאת במרחב הכתובות של Process אחר.

6.3 סיכום

התחלנו את הפרק בסקירת סוגים שונים של זיכרון פיזי. הצבענו על יתרונות וחסרונות של סוגי הזיכרון הפיזי- גודל, מחיר, מהירות גישה, נדיפות. למדנו שProcess אינו יכול לרוץ מדיסק קשיח אלא רק מתוך זיכרון ה-RAM וראינו את היתרון של שילוב של דיסק קשיח עם RAM.

למדנו איך פועל מנגנון המרת הכתובות מכתובות לוגיות אל כתובות לינאריות (וירטואליות) ועד לכתובות פיזיות ב-RAM. למדנו על מנגנון הסגמנטציה שמאפשר את מעגלי האבטחה השונים ומונע גישה למקומות לא רצויים. למדנו על מנגנון ה-Paging שמאפשר הפרדה בין זיכרון של Processים וביצוע מהיר של Context Switch במעבר לProcess אחר. לאחר מכן ראינו מה מבצעת מערכת ההפעלה כאשר היא נתקלת ב- Page Fault וראינו באיזה קובץ Windows שומרת את הדפים שאין להם מקום ב-RAM.
השאלה שתלווה אותנו בפרק הבא: כיצד, למרות שיש הפרדה בין הזיכרון שלהם, Processים יכולים לשתף ביניהם זיכרון?

פרק 7 – שיתוף זיכרון

את הפרק הקודם סיימנו בשאלה, כיצד, למרות שיש הפרדה בין הזיכרון שלהם, Processים יכולים לשתף ביניהם זיכרון?

כדי להבין איך יכול להיות מצב שבו שני Processים ניגשים לאותו מקום ב-RAM יש צורך להבין מהו מיפוי זיכרון. לכן חלק ניכר נקדיש למנגנון מיפוי הזיכרון, ולאחר מכן המשך ההסבר על שיתוף הזיכרון יהיה פשוט.

מיפוי זיכרון הוא תהליך שבו נוצר קישור בין זיכרון פיזי ב-RAM לבין זיכרון של התקן חומרה כלשהו. התקן חומרה שבוצע עבורו מיפוי זיכרון ל-RAM נקרא Memory Mapped I/O, כאשר ה-I/O הם קיצור של Input/Output. מה המשמעות של קישור בין שני זיכרונות? הכוונה ליצירת מנגנון שבו כל שינוי בזיכרון אחד יוצר את אותו השינוי בזיכרון אחר. היתרון בשיטה הזו היא שהיא מאפשרת למעבד לפנות בקלות להתקני חומרה, בלי צורך להפעיל Interruptים (פסיקות חומרה) ובלי צורך בפניה ל-Portים ושימוש בפקודות מיוחדות. אם למדתם אסמבלי באמצעות ספר הלימוד של המרכז לחינוך סייבר, ייתכן ואתם זוכרים את פקודות האסמבלי IN ו-OUT, שמאפשרות למעבד לקבל ולשלוח מידע אל פורטים ומהם.

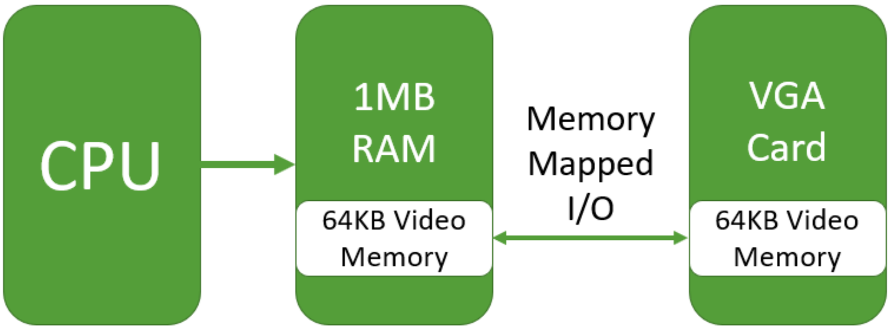
מבין התקני החומרה השונים שניתן לבצע אליהם מיפוי זיכרון, התקן חומרה ספציפי הוא התקן לשמירת קבצים כגון הדיסק הקשיח. אם המיפוי מבוצע לקובץ ולא למקרה הכללי יותר של התקן חומרה, הקובץ נקרא פשוט Memory Mapped File.

כדי להבין את העיקרון, נקדיש הסבר קצר ל-Memory Mapped I/O ומשם נמשיך ל-File Mapping.

7.1 מיפוי זיכרון קלט/פלט - Memory Mapped I/O

השימוש ב- Memory Mapped I/O קלאסי לכרטיסי גרפיקה, עקב מהירות העדכון הרבה שהם דורשים, לדוגמה במשחקי מחשב, מיפוי זיכרון הוא אפשרות מקובלת. ניקח מערכת שיש בה מעבד, זיכרון RAM וכרטיס גרפי. כל מה שמוצג למסך המשתמש חייב להיות בזיכרון של הכרטיס הגרפי. נמחיש את העקרון באמצעות מעבד ה-8086 ההיסטורי, אך העקרון תקף גם כיום.

למעבד ה-8086 היתה גישה ל-1MB של זיכרון RAM. לצד המעבד היה מותקן כרטיס גרפי VGA שמסוגל להציג 256 צבעים על מסך שגודלו 320X200 פיקסלים. כל פיקסל דרש בית אחד, לכן גודלו של הזיכרון של כרטיס ה-VGA היה 64KB. כל הזיכרון הגרפי מופה אל אזור מסוים בזיכרון ה-RAM, שהתחיל בכתובת 0xA0000 וגודלו היה 0x10000 (כלומר 64KB).



היתרון של מיפוי הזיכרון הוא הפשטות והמהירות: כדי לפנות לזיכרון הוידאו, המעבד פשוט מבצע פקודות העתקה אל אזור הזיכרון שממופה אצלו ל-RAM. אין צורך בפסיקות חומרה איטיות. החיסרון- המיפוי גוזל חלק מזיכרון ה-RAM של המעבד, שלמעשה הפך להיות בלתי שמיש לכל מטרה שאינה תקשורת עם כרטיס הוידאו.

7.2 מיפוי קובץ- Memory Mapped File

לפעולה של מיפוי קובץ, או קטע מקובץ, אל ה-RAM יש שני יתרונות:

1. חיסכון במקום ב-RAM
2. אפשרות לשיתוף זיכרון בין Processים שונים

נתחיל בחיסכון ב-RAM, וכדי להבין את הנושא היטב נבצע תרגיל מודרך.

תרגיל 7.1 תרגיל מודרך מיפוי קובץ לזיכרון, שלב א

הורידו את הקובץ http://data.cyber.org.il/os/gibrish.bin (קרדיט לאייל אבני על הקוד ליצירת הקובץ). זהו קובץ טקסט הכולל רצף אקראי של תווים, לדוגמה gH9rnA5Zzf. עליכם למנות את כמות התווים "A" המופיעים בקובץ.

בצעו את המשימה באמצעות פונקציות של WinAPI, אך ללא MapViewOfFile (אם אינכם מכירים את הפונקציה הזו, הכל בסדר, זה חלק מהתרגיל המודרך ויוסבר בהמשך).

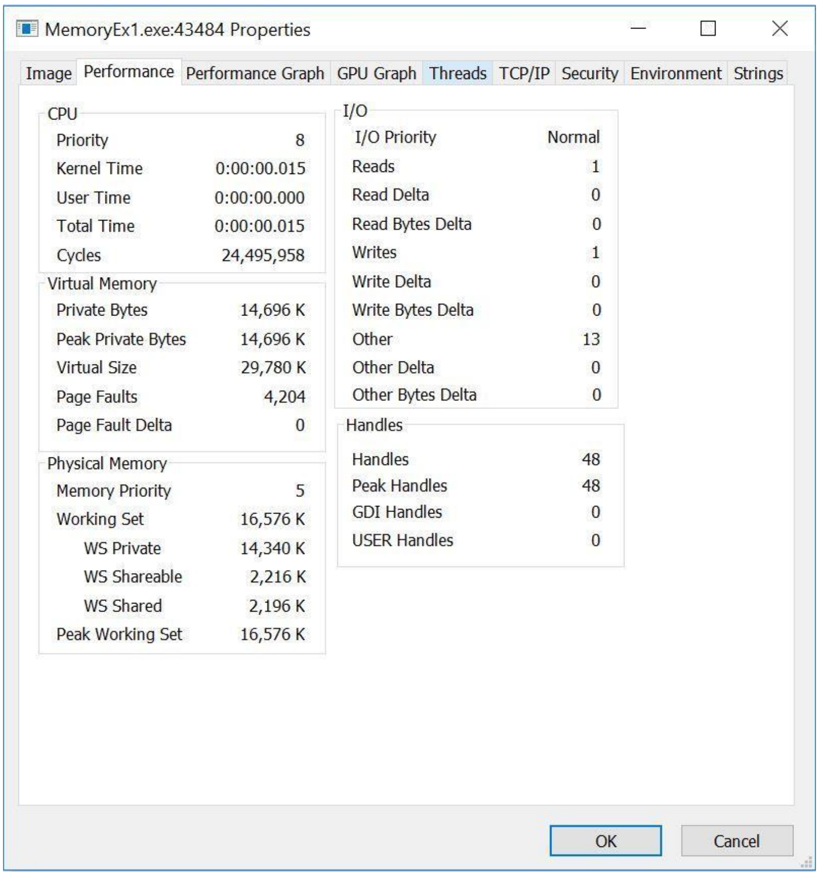
כיצד לבצע את התרגיל? תוכלו להעזר בשלבים הבאים. אם אתם יכולים להסתדר בלי הדרכה- עדיף! דלגו מעל ההדרכה.

1. על מנת לפתוח את הקובץ השתמשו בפונקציה CreateFileA. ה-A בסיום הפונקציה מציין שהיא עושה שימוש בתווי ASCII בגודל של בית, ולא בתווים מורחבים בגודל שני בתים. אנחנו מניחים ששם הקובץ והנתיב הם כולם תווי ASCII, אם יש תווים בעברית לדוגמה אז תצטרכו להשתמש ב-CreateFileW.
2. כדי לדעת כמה מקום נצטרך להקצות בזיכרון, נבצע GetFileSize.
3. כעת נבצע malloc של כמות הזיכרון הנחוצה כדי לשמור את כל הקובץ. כמובן שמספר זה יכול להיות גדול מאד, זה בדיוק העניין אותו אנחנו מבקשים להציג פה ובהמשך לבצע בצורה חסכונית יותר.
4. הצעד הבא הוא ביצוע ReadFile לתוך הבאפר שהגדרנו ב-malloc.
5. כעת נותר לכתוב פונקציה שעוברת על הבאפר ומונה את כמות התווים "A" שנמצאים בו.
6. לסיום התוכנית, נבצע free לזיכרון ו-CloseHandle לקובץ.

להלן התוכנית, למעט החלק של ספירת כמות ה-"A":

```
#include "pch.h"
#define FILENAME "output_file.bin"

int main()
{
    HANDLE hFile;
    CHAR file_name[] = FILENAME;
    LPCSTR pFileName = file_name;

    hFile = CreateFileA(
        pFileName,              // file name
        GENERIC_READ,           // access type
        0,                      // other processes can't share
        NULL,
        OPEN_EXISTING,          // open only if file exists
        FILE_ATTRIBUTE_NORMAL,
        NULL);

    DWORD file_size = GetFileSize(hFile, NULL);
    LPVOID pBuf;
    LPSTR data;
    pBuf = malloc(file_size + 1);
    ReadFile(hFile, pBuf, file_size, NULL, NULL);
    data = (LPSTR)pBuf;
    Sleep(10000); // place the character count here, instead of
                  // Sleep
    free(pBuf);
    CloseHandle(hFile);
    return 0;
}
```

הריצו את התוכנית מתוך ה-command line (לא מתוך visual studio) והפעילו process explorer. בזמן שהתוכנית רצה, הקליקו עליה קליק ימני ובחרו properties.



תחת סעיף Physical Memory- Working Set אפשר למצוא את הזיכרון הפיזי שהוקצה ב-RAM עבור ה-process שלנו. היזכרו בהסבר על Working Set מהפרק הקודם.

בדוגמה זו השתמשנו בקובץ טקסט שתופס 14,327,808 בתים על הדיסק (כפולה מדוייקת של גודל Page-4096 בתים- בידקו זאת בעצמכם!). הקוד שלנו טען ל-RAM את כל התווים בקובץ בזה אחר זה, ולכך נוספה כמות מסויימת של RAM שהתוכנית שלנו היתה צריכה על מנת לרוץ בעצמה. בסך הכל, בשיא התוכנית שלנו ניצלה כ-16,576KB של RAM. לו הקובץ שלנו היה גדול יותר, היינו נדרשים לכמות RAM גדולה יותר. אם אנחנו רוצים לחסוך RAM, נצטרך להתאמץ קצת.

הרעיון הכללי של הקטנת כמות הזיכרון הנדרשת עובד כך: במקום לטעון את כל הקובץ מהדיסק הקשיח ל-RAM, וממנו אל הזיכרון של ה-Process, נבצע בכל פעם טעינה של קטע מידע אחר מהדיסק אל ה-RAM. כל עוד המידע מהדיסק יהיה בשימוש על ידי ה-Process, נשמור אותו ב-RAM. כאשר לא נזדקק יותר לקטע המידע נזרוק אותו ונשתמש באותו אזור ב-RAM לטובת קטע מידע אחר.

בתהליך שנקרא "מיפוי" אנחנו מייצרים קשר בין ה-RAM לדיסק הקשיח, כך שכל שינוי באחד-משפיע על השני. נסקור את הפונקציות שמבצעות את הפעולות הנדרשות.

7.2.2 פונקציית CreateFileMapping

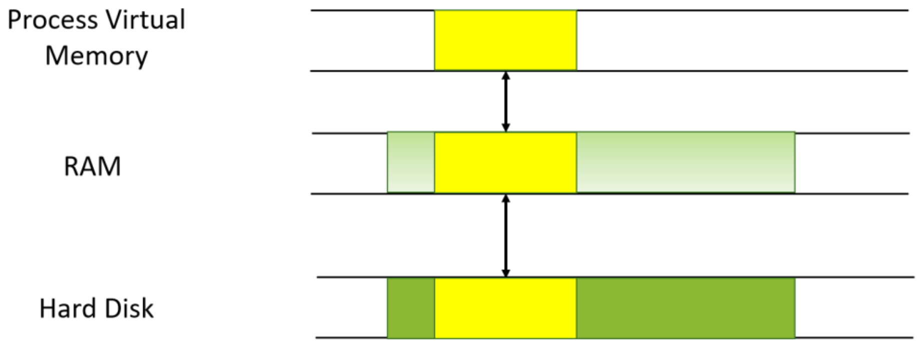
הפונקציה שיוצרת את המיפוי היא CreateFileMapping. יצירת המיפוי עדיין לא טוענת את הקובץ ל-RAM, היא רק מחזירה לנו Handle ל"אובייקט מיפוי", מעין מצביע על קובץ, בו נוכל להשתמש על מנת לטעון חלקים מהקובץ כאשר נזדקק להם.

מיפוי קובץ מהדיסק הקשיח אל ה-*RAM*,

שימוש ב-*CreateFileMapping* יוצרת אובייקט מיפוי בין הדיסק הקשיח אל ה-*RAM*

7.2.3 פונקציית MapViewOfFile

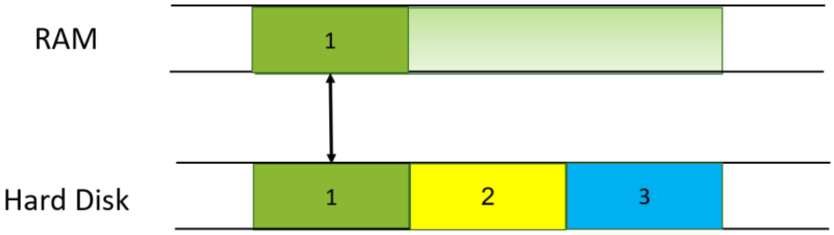
כאשר ה-Process צריך להשתמש בקובץ הוא יבצע MapViewOfFile לחלק מהקובץ שברצונו להשתמש בו. רק בשלב זה ייטען חלק מהקובץ אל ה-RAM. להלן המחשה לתמונת הזיכרון לאחר ביצוע MapViewOfFile:



תמונת הזיכרון אחרי ביצוע *MapViewOfFile* לחלק הקובץ שצבוע בצהוב

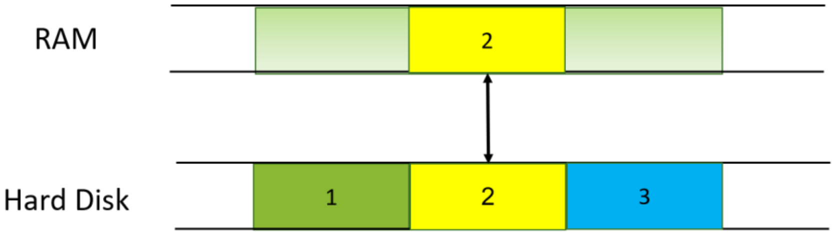
האם אנחנו יכולים לבצע מיפוי של קטע זיכרון בכל גודל שנבחר? זוהי נקודה עדינה שצריך להתייחס אליה לפני שנוכל לתכנת תוכנית עובדת. עקרונית- כן, אנחנו יכולים לבצע MapViewOfFile לכל חלק של הקובץ, בהנחה שגודלו לא עולה על גודל החלק שעשינו לו לפני כן CreateFileMapping. עם זאת, אם נרצה לעשות MapViewOfFile לחלק הבא של הקובץ, ניתקל באילוץ של מערכת ההפעלה, שהאזור בזיכרון צריך להתחיל בכתובת "עגולה". מבחינת המחשב, כתובת שמתחלקת יפה בחזקות של 2 היא כתובת "עגולה", כמו שמבחינתנו בני האדם מספר שמתחלק יפה בחזקות של 10 הוא מספר עגול. אילוץ זה הוא בדרך כלל 65536 בתים (2 בחזקת 16).

נמחיש את העניין: בדוגמה הבאה, אנחנו מבצעים מיפוי לחלק 1 ולאחר מכן לחלק 2, שנמצא בקובץ מיד אחרי חלק 1. כדי שנוכל לבצע מיפוי לחלק 2 בלי להיתקל בשגיאה של מערכת ההפעלה, אנחנו צריכים לדאוג לכך שהמיפוי של 2 יתחיל בכתובת "עגולה" שמתחלקת ב-65536.



מיפוי חלק מספר 1, יכול להיות בכל גודל שנרצה

כלומר, כדי שנוכל לבצע את המיפוי בקלות, רצוי מאד שהגודל של חלק מספר 1 והגודל של כל החלקים שאנחנו ממפים יהיו כפולות שלמות של 65536.



מיפוי חלק מספר 2 חייב להתחיל בכתובת "עגולה", לכן כדאי שחלק מספר 1 יהיה בגודל "עגול" מספר זה ידוע כ- "Alignment Granularity" וניתן למצוא אותו באמצעות הקוד הבא:

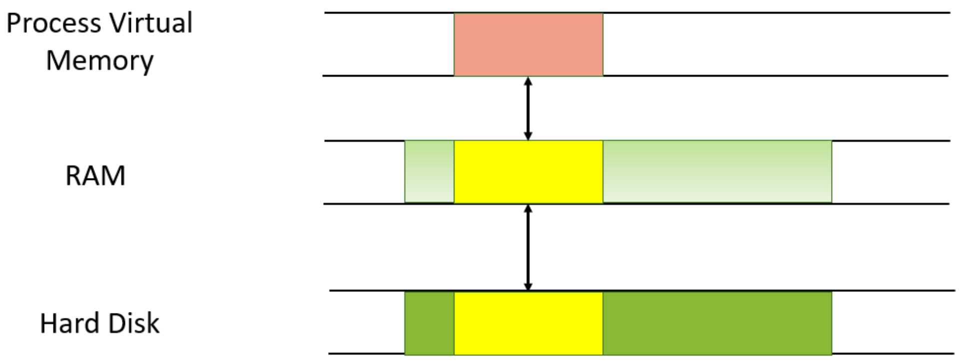
```
// get system memory alignment granularity (usually 65536)
SYSTEM_INFO sys_info;
GetSystemInfo(&sys_info);
int mem_buffer_size = sys_info.dwAllocationGranularity;
```

7.2.4 פונקציית UnmapViewOfFile

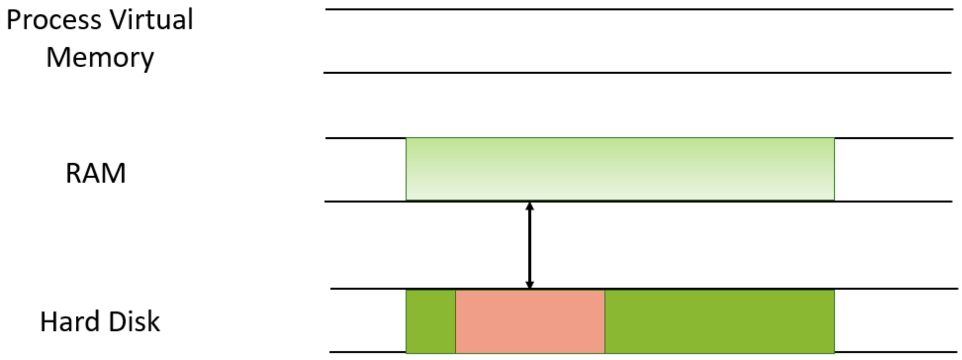
פעולות המיפוי שראינו עד כה לא סייעו לנו לחסוך בזיכרון. זאת מכיוון שלא עשינו שימוש חוזר בזיכרון. אפשר לנצל את העובדה שכאשר אנחנו מסיימים לעבור על קטע מהקובץ אנחנו לא צריכים אותו יותר: לאחר שנסיים לעבור על הקטע שמיפינו, נוכל להסיר את המיפוי שלו מהזיכרון ואז נטען את הקטע הבא בדיוק לאותו מקום בזיכרון. בצורה זו, הגודל של הזיכרון שנשתמש בו יהיה די קבוע ודומה לגודלו של הקטע שמיפינו אל הזיכרון. גם אם הקובץ שלנו מאד גדול, לא נצטרך להגדיל את כמות הזיכרון, רק להחליף יותר פעמים את הקטע הממופה לזיכרון.

הפונקציה UnmapViewOfFile מבצעת את הפעולה ההפוכה ל-MapViewOfFile: היא מבטלת את המיפוי בין ה-RAM לבין קטע הקובץ שביצענו לו מיפוי. שימו לב שהפונקציה אינה מבטלת את אובייקט המיפוי שיצרנו. האובייקט עדיין קיים ואפשר להשתמש בו למיפויים נוספים.

כחלק מתהליך ביטול המיפוי של קטע הקובץ, כל השינויים שבוצעו לקטע הקובץ ב-RAM נשמרים חזרה לתוך הקובץ בדיסק הקשיח.



סכמת הזיכרון לפני ביצוע *UnmapViewOfFile*. הקטע האדום מסמן קטע קובץ שהשתנה בזיכרון



סכמת הזיכרון לאחר ביצוע *UnmapViewOfFile*. השינוי נשמר בדיסק

7.2.5 פונקציית CloseHandle

הפונקציה CloseHandle היא הפונקציה ההפוכה לפונקציה CreateFileMapping. כשם ש-CreateFileMapping יוצרת אובייקט מיפוי, הפונקציה CloseHandle סוגרת את האובייקט. פונקציה זו יכולה לשמש כמובן גם לסגירת הקובץ שפתחנו בדיסק.

תרגיל 7.2 תרגיל מודרך מיפוי קובץ לזיכרון, שלב ב

חיזרו על התרגיל הקודם. הפעם עליכם למנות את כמות התווים "A" בקובץ gibrish.bin, אך תוך שימוש מינימלי בזיכרון RAM.

יש בידיכם את ההבנה התיאורטית שנחוצה כדי לפתור את התרגיל. תזדקקו לפונקציות ***CreateFile***, ***CloseHandle***, ***UnmapViewOfFile***, ***MapViewOfFile***, ***CreateFileMapping***. המשך הסעיף ידריך

אותכם עד צעד בכתיבת הקוד עצמו, אך נסו להסתדר בלעדיו! קיראו את התעוד של הפונקציות באתר MSDN, הוא יעזור לכם לבצע את המשימה גם בלי עזרה.

בשלב הראשון אנחנו צריכים לבצע CreateFile על מנת לפתוח את קובץ הטקסט. אין לנו צורך לשנות משהו בקובץ, לכן נפתח אותו לקריאה בלבד. אנחנו גם לא מעוניינים ש-Processים אחרים יהיו שותפים לקובץ שלנו. שימו לב שמה שנכון למשימה שלנו כרגע לא בהכרח יהיה נכון גם בהמשך.

בהזדמנות זאת נבדוק גם את גודל הקובץ, נתון שישמש אותנו כאשר נרצה לבצע לולאה שתקרא את כל המידע בקובץ בכפולות של ה-System Granularity.

```
hFile = CreateFileA(
    pFileName,                  // file name
    GENERIC_READ,               // access type
    0,                          // other processes can't share
    NULL,                       // security
    OPEN_EXISTING,              // open only if file exists
    FILE_ATTRIBUTE_NORMAL,
    NULL);

DWORD file_size = GetFileSize(hFile, NULL);
```

בשלב השני נשתמש ב-handle אל הקובץ כדי ליצור מיפוי שלו לזיכרון.

```
hMapFile = CreateFileMappingA(
    hFile,                      // file handle
    NULL,                       // default security
    PAGE_READONLY,              // read access
    0,                          // maximum object size (high-order
                                // DWORD)
    0,                          // maximum object size (low-order
                                // DWORD)
                                // 0 means map the whole file
    "myFile");                  // name of mapping object, for future
                                // sharing
```

כמה הערות לגבי הפרמטרים:

1. הרשאת הגישה יכולה להיות שווה או קטנה מההרשאה של CreateFile. במילים אחרות, אם פתחנו את הקובץ לקריאה בלבד, לא נוכל לבצע מיפוי שכולל הרשאת כתיבה.
2. גודל קובץ מקסימלי- הפרמטר 0 משמעותו שאנחנו רוצים את כל הקובץ
3. נתנו למיפוי שם – "myFile". כיוון שאנחנו לא משתפים את הקובץ עם Processים אחרים זה די מיותר, אבל בתוכניות אחרות זה יכול להיות שימושי

בשלב השלישי, נחלק את הקובץ שלנו לחלקים בגודל 65536 בתים ונעבור על הקובץ בלולאה. הלולאה תסתיים כאשר מתקיים תנאי שאם ניקח עוד חלק אחד נעבור את גודל הקובץ. בתוך הלולאה נבצע את הפעולות הבאות:

1. MapViewOfFile לחלק שגודלו 65536 בתים
2. ספירת כמות הפעמים שמופיע התו "A" בחלק
3. UnmapViewOfFile כדי להסיר מה-RAM את הזיכרון שמיפינו וכך להיות חסכוניים יותר בזיכרון

```
while (file_location <= (file_size - mem_buffer_size)) {
    pBuf = (LPSTR)MapViewOfFile(
        hMapFile,                   // handle to map object
        FILE_MAP_READ,              // read/write permission
        0,                          // start point (upper word)
        file_location,              // start point (lower word)
        mem_buffer_size);           // how many bytes to read
    count = CountChar(pBuf, mem_buffer_size, letter);
    printf("%d\n", count);
    buffer_number++;
    file_location = mem_buffer_size * buffer_number;
    UnmapViewOfFile(pBuf);
    Sleep(100);
}
```

בסוף הלולאה קבענו Sleep של עשירית שניה, כדי להאט את הריצה של התוכנית מספיק כך שהיא לא תסתיים לפני שיהיה לנו זמן לבדוק את כמות הזיכרון שהיא דורשת.

שימו לב שהקובץ שלנו לא בהכרח מתחלק ב-65536 בתים, לכן לאחר הלולאה צריך להוסיף עוד מיפוי ובדיקת כמות התו "A" גם לשארית שנותרה.

בשלב האחרון, כל מה שנותר לנו לבצע זה UnmapViewOfFile לשארית, ו-CloseHandle לקובץ ולאובייקט המיפוי.

להלן כלל הקוד. נסו להסתדר בלעדיו!

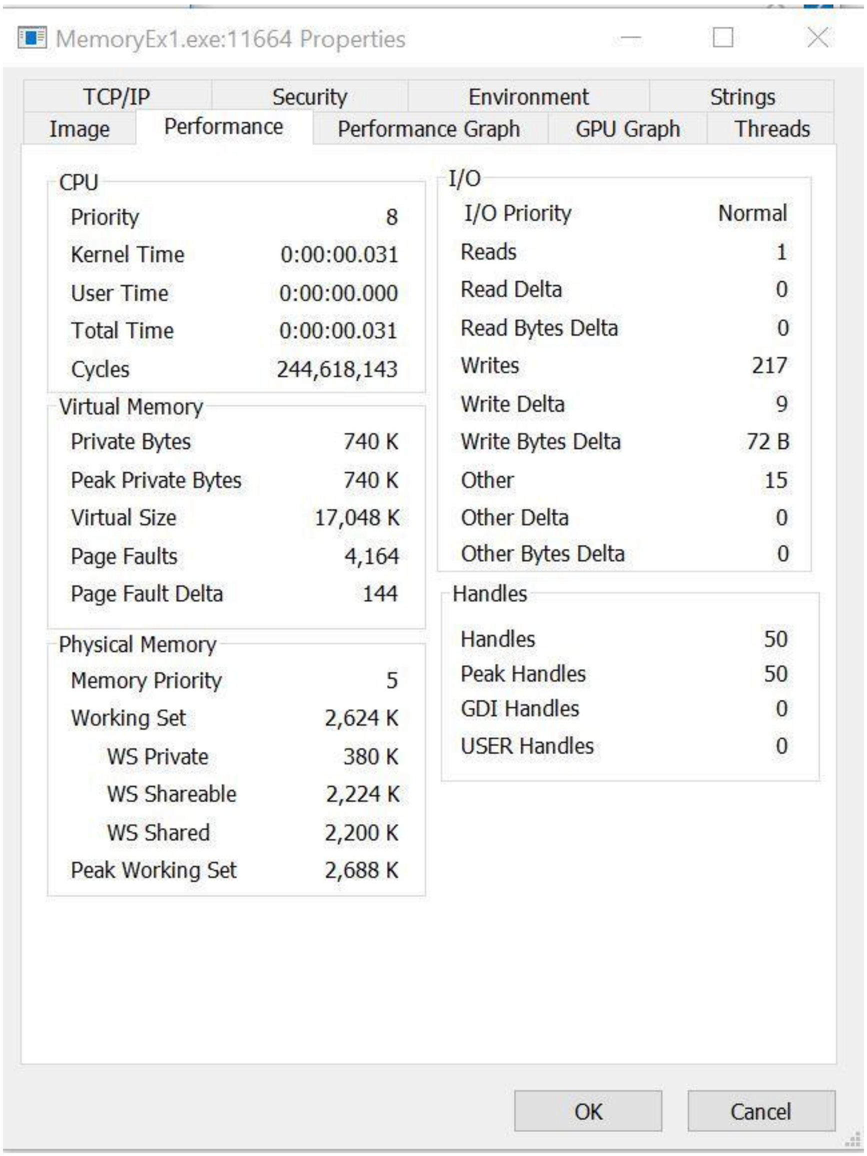
```
#include "pch.h"
#define FILENAME "c:\\os\\gibrish.bin"
#define SEARCH_LETTER "A"

int CountChar(PCHAR pBuf, int buff_size, LPCSTR letter);

int main()
{
    // get system memory allignement granularity (usually 65536)
    SYSTEM_INFO sys_info;
    GetSystemInfo(&sys_info);
    int mem_buffer_size = sys_info.dwAllocationGranularity;

    printf("%d\n", mem_buffer_size);

    // open an existing file for reading
    HANDLE hFile;
    LPCSTR pFileName = FILENAME;

    hFile = CreateFileA(
        pFileName,              // file name
        GENERIC_READ,           // access type
        0,                      // other processes can't share
        NULL,                   // security
        OPEN_EXISTING,          // open only if file exists
        FILE_ATTRIBUTE_NORMAL,
        NULL);
```

```
    DWORD file_size = GetFileSize(hFile, NULL);

    // create file mapping object
    HANDLE hMapFile;

    hMapFile = CreateFileMappingA(
        hFile,                  // file handle
        NULL,                   // default security
        PAGE_READONLY,          // read access
        0,                      // maximum object size (high-order
                                // DWORD)
        0,                      // maximum object size (low-order
DWORD)
                                // 0 means map the whole file
        "myFile");              // name of mapping object, in case we
                                // want to share it

    // read the file, one page at a time
    int buffer_number = 0, count;
    int file_location = buffer_number * mem_buffer_size;
    LPSTR pBuf;
    LPCSTR letter = SEARCH_LETTER;

    while (file_location <= (file_size - mem_buffer_size)) {
        pBuf = (LPSTR)MapViewOfFile(
            hMapFile,                   // handle to map object
            FILE_MAP_READ,              // read/write permission
            0,                          // start point (upper word)
            file_location,              // start point (lower word)
            mem_buffer_size);           // how many bytes to read
        count = CountChar(pBuf, mem_buffer_size, letter);
```

```
        printf("%d\n", count);
        buffer_number++;
        file_location = mem_buffer_size * buffer_number;
        UnmapViewOfFile(pBuf);
        Sleep(100);
    }

    int reminder = file_size - file_location;
    pBuf = (LPSTR)MapViewOfFile(
        hMapFile,                // handle to map object
        FILE_MAP_READ,           // read/write permission
        0,                       // start point (upper word)
        file_location,           // start point (lower word)
        reminder);               // how many bytes to read
    count = CountChar(pBuf, reminder, letter);
    printf("%d\n", count);
    UnmapViewOfFile(pBuf);

    CloseHandle(hMapFile);
    CloseHandle(hFile);
    return 0;
}

int CountChar(PCHAR pBuf, int buff_size, LPCSTR letter) {
    static int count = 0;
    for (int i = 0; i < buff_size; i++) {
        if (pBuf[i] == *letter) count++;
    }
    return count;
}
```

7.2.6 סיכום תרגיל מודרך

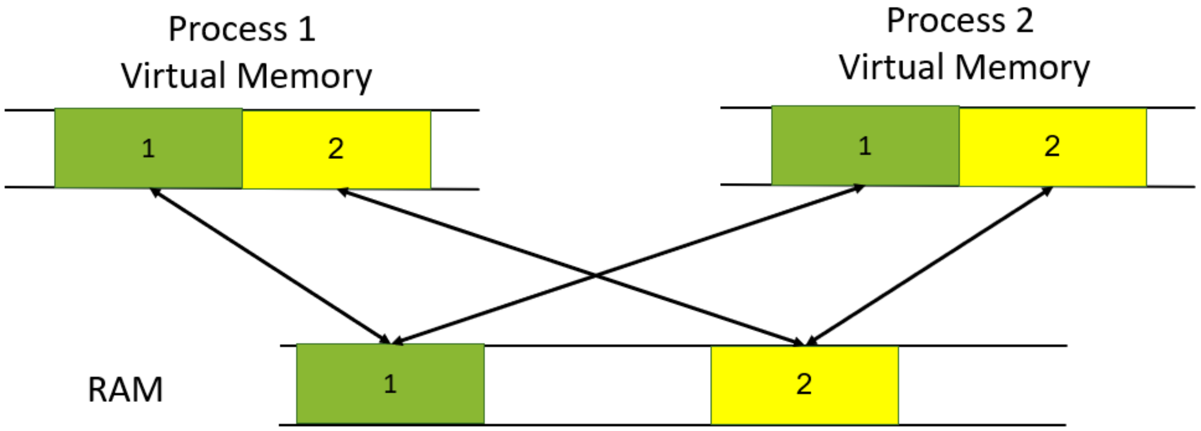
נשתמש שוב ב-Process Explorer כדי לחזור על בדיקת כמות הזיכרון הנדרשת, הפעם עם הקוד היעיל שלנו. ראו כמה חיסכון של Physical Memory השגנו יחסית לקוד הקודם, ה-Working Set הוא רק 2624KB וזה עבור קובץ לא מאד גדול של כ-10 מגה. לו היינו מבצעים את הפעולה על קובץ של מספר ג'יגה, ההבדל היה ניכר עוד הרבה יותר.



7.3 זיכרון משותף - Shared Memory

מיפוי זיכרון יכול להפוך בקלות לשיתוף זיכרון. במצב של שיתוף זיכרון, אותו אזור ב-RAM ממופה לתוך הזיכרון הוירטואלי של שני Processים.

האיור הבא ממחיש את שיתוף הזיכרון, שני קטעי זיכרון ב-RAM ממופים אל שני Processים. במצב זה, הזיכרונות של שני ה-Processים קשורים ביניהם באותם קטעי זיכרון ב-RAM. אם אחד מה-Processים ישנה את הזיכרון בקטעים המשותפים, ה-Process השני יראה את השינוי.



כדי להשיג Shared Memory, יש לפעול לפי השלבים הבאים:

Process 1:

1. פותח קובץ- CreataFile
2. יוצר מיפוי בין הקובץ לזיכרון – CreateFileMapping. כחלק מתהליך המיפוי, נותן שם לאזור בזיכרון. היזכרו בפרמטר האחרון בפונקציה CreateFileMapping, שבדוגמת הקוד נקרא "MyFile"
3. משתמש במיפוי עבור חלק מהקובץ - MapViewOfFile

Process 2:

1. לוקח גישה לאזור הממופה – OpenFileMapping. כדי להשיג גישה, צריך להכיר את שם האזור (בדוגמה- "MyFile")
2. משתמש במיפוי עבור חלק מהקובץ – MapViewOfFile

לקריאה נוספת, מתוך הדרכה של מיקרוסופט:

https://docs.microsoft.com/en-us/previous-versions/ms810613(v=msdn.10)?redirectedfrom=MSDN

7.3.1 דוגמאות ל-Shared Memory

נסקור שתי דוגמאות קלאסיות לשיתוף זיכרון: ה-Kernel וה-DLLים.

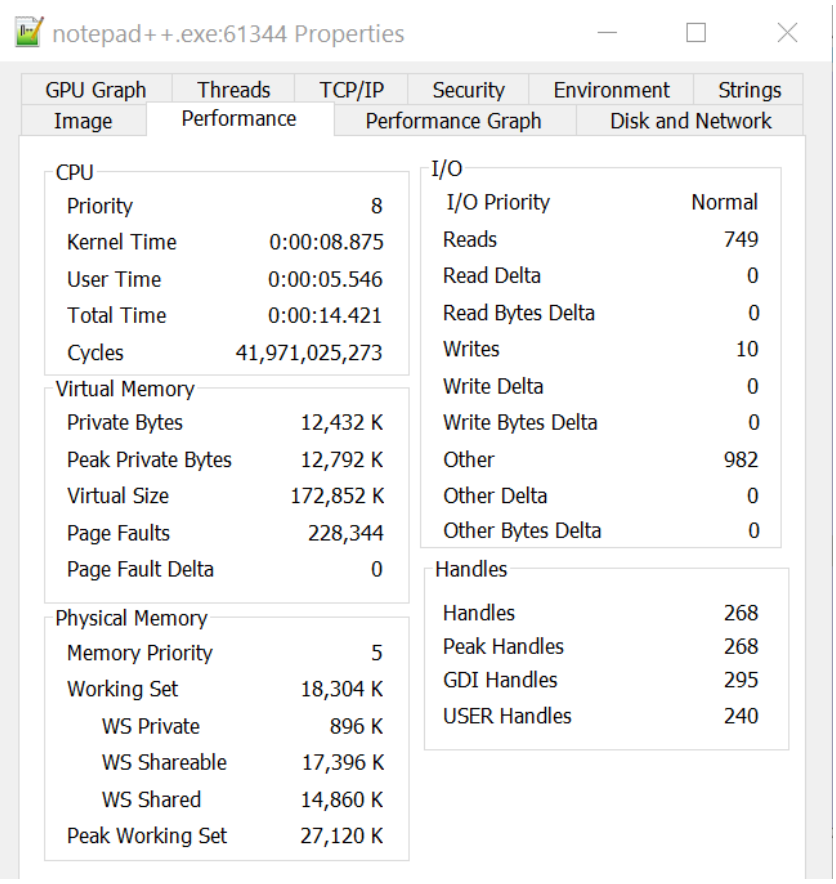
כפי שלמדנו, חלק ממרחב הזיכרון של כל Process מוקדש לקוד של ה-Kernel של מערכת ההפעלה. דבר זה מאפשר ל-Process לפנות לפונקציות של מערכת ההפעלה.

מה קורה כאשר מבוצע Conext Switch אל Process חדש? כיוון שגם ה-Process החדש צריך את אותו הקוד של מערכת ההפעלה במרחב הזיכרון, זה יהיה בזבוז עצום להחליף את כל הדפים ב-RAM שיש בהם את הקוד של מערכת ההפעלה. הרבה יותר פשוט לבצע מיפוי של קוד מערכת ההפעלה שנמצא כבר ב-RAM אל הזיכרון של ה-Process החדש, ובאופן כללי אל כל ה-Processים שרצים. התוצאה היא שקוד של מערכת ההפעלה נמצא Shared Memory בין כל ה-Processים.

בניגוד לקובץ הטקסט שעליו ביצענו את התרגיל, שבזמן השיתוף שלו הענקנו לשני ה-Processים שחלקו בו הרשאות כתיבה, כמובן שכל אזור הזיכרון של מערכת ההפעלה נמצא בהרשאות שאינן מאפשרות ל-Process ב-Userland לשנות אותו, שלמרות שה-Kernel נמצא בזיכרון משותף האבטחה נשמרת. כשסקרנו את ניהול הזיכרון ראינו איך מנגנון המרת הכתובות מכתובת לוגית לכתובת פיזית מממש את מעגלי האבטחה ומונע גישה לזיכרון שמכיל את ה-Kernel.

דוגמה נוספת לשיתוף זיכרון היא השימוש ב-DLLים, קיצור של Dynamic Link Library. נפרט אודות DLLים בהמשך, בינתיים כל מה שצריך לדעת הוא ש-DLL הינו חבילת קוד מקומפל שאפשר לייבא לתוך הקוד שלנו. מה קורה כאשר תוכניות רבות מייבאות את אותה חבילת קוד? אין הגיון בכך שכל Process יעתיק את חבילת הקוד לתוך ה-RAM. לכן DLLים, כמו קוד מערכת ההפעלה, ממופים לכל הזיכרון של כל ה-Processים שביצעו להם יבוא. נצפה בזיכרון המשותף של Process לדוגמה, ה-Process הנבחר יהיה ++notepad.

אם נבדוק את הזיכרון הפיזי של ++notepad (באמצעות Process Explorer, כמובן) נמצא כמה זיכרון ה-Process דורש רק לעצמו וכמה זיכרון פיזי הוא משתף עם Processים אחרים:



נתבונן בנתוני צריכת הזיכרון הפיזי. הקיצור WS מסמן Working Set. הנתון WS Private הוא כמות הזיכרון הפיזי של ה-Process שנמצא בפועל ב-RAM והוא שייך רק ל-Process, אינו ממופה ל-Processים אחרים. הנתון WS Shared הוא כמות הזיכרון שעבר מיפוי ומשותף עם Processים אחרים והנתון WS Shareable הוא כמות הזיכרון שעבר מיפוי כך שהוא נגיש ל-Processים אחרים, אך כרגע אין אף Process אחר שעושה בו שימוש. מי אחראים לכל הזיכרון המשותף של ה-Process?

הפעילו את תוכנת VMMap . תוכלו לראות את כל מרחב הכתובות הוירטואלי של ה-Process שבחרנו. בצד ימין של הטבלה כתובים השם של חבילת הקוד שהועלתה לזיכרון. כפי שאפשר לראות, הזיכרון הוירטואלי מכיל DLLים רבים שמופו אליו:

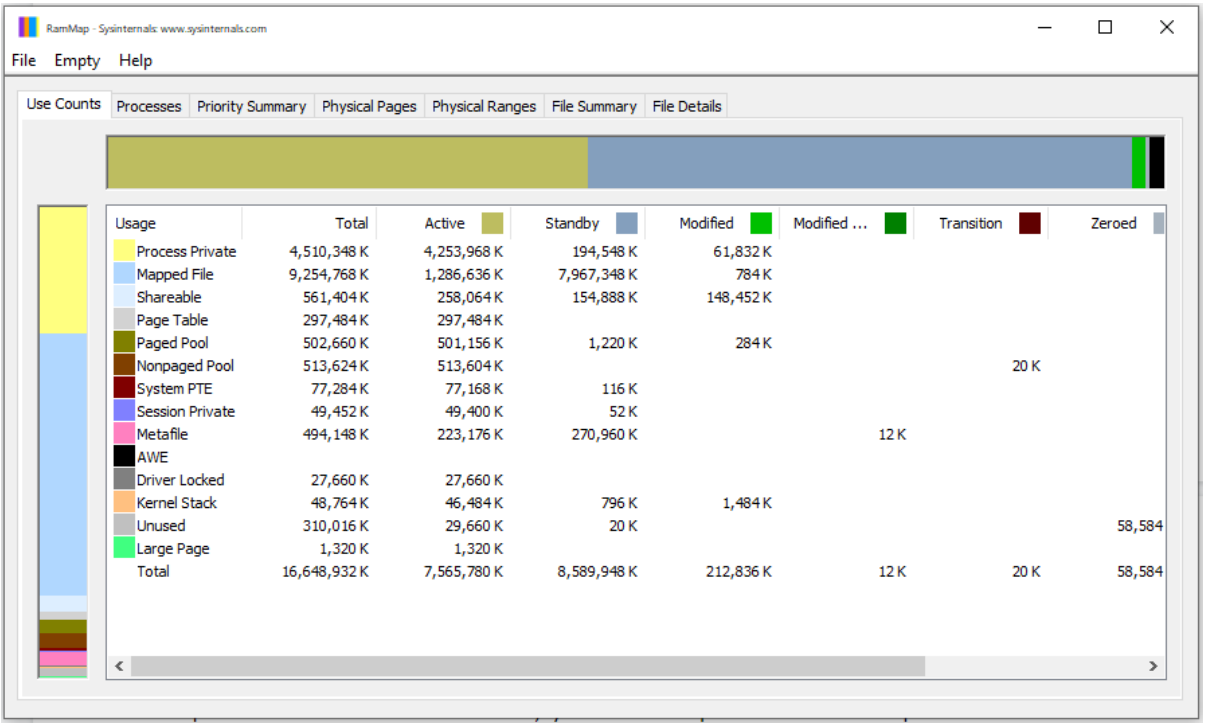
Address	Type	Size	Com...	Private	Total ...	Priva...	Shar...	Sh...	Loc...	Blo...	Protection	Details
35550000	Private Data	64 K	64 K	64 K	64 K	64 K				1	Execute/Read	
5C220000	Image (ASLR)	1,04...	1,044 K	24 K	140 K	20 K	120 K	96 K		7	Execute/Read	C:\Program Files (x86)\Notepad++\SciLe...
66FA0000	Image (ASLR)	32 K	32 K	4 K	28 K	12 K	16 K	16 K		4	Execute/Read	C:\Windows\SysWOW64\SensApi.dll
6B430000	Image (ASLR)	24 K	24 K	4 K	20 K	8 K	12 K	12 K		4	Execute/Read	C:\Windows\SysWOW64\msimg32.dll
6C900000	Image (ASLR)	124 K	124 K	16 K	92 K	20 K	72 K			5	Execute/Read	C:\Program Files (x86)\Sysinternals\Trac...
6F6C0000	Image (ASLR)	488 K	488 K	12 K	84 K	16 K	68 K	68 K		5	Execute/Read	C:\Windows\SysWOW64\uxtheme.dll
72230000	Image (ASLR)	1,59...	1,596 K	116 K	188 K	44 K	144 K	144 K		10	Execute/Read	C:\Windows\SysWOW64\dbghelp.dll
723D0000	Image (ASLR)	640 K	640 K	36 K	224 K	16 K	208 K	208 K		8	Execute/Read	C:\Program Files\Bitdefender\Bitdefend...
72880000	Image (ASLR)	2,10...	2,108 K	12 K	116 K	20 K	96 K	96 K		4	Execute/Read	C:\Windows\WinSxS\x86_microsoft.wind...
73440000	Image (ASLR)	636 K	636 K	8 K	240 K	12 K	228 K	228 K		4	Execute/Read	C:\Windows\SysWOW64\apphelp.dll
74DB0000	Image (ASLR)	32 K	32 K	4 K	24 K	8 K	16 K	16 K		4	Execute/Read	C:\Windows\SysWOW64\version.dll

אזור הזיכרון שתופסים ה-DLLים השונים מסתיים כצפוי לפני הכתובת 0x80000000, שבה מתחיל ה-Kernel.

7.4 תוכנת RAMMap

לא נסיים את הפרק בלי להשתמש בכלי נוסף של Sysinternals, שכעת יש לנו את הידע הנדרש כדי להבין.

פיתחו את RAMMap. בניגוד ל-VMMap, שמציג לכם מיפוי של מרחב הזיכרון של Process ספציפי, הכלי RAMMap נותן צילום מצב של כל שנמצא כרגע ב-RAM שלכם.



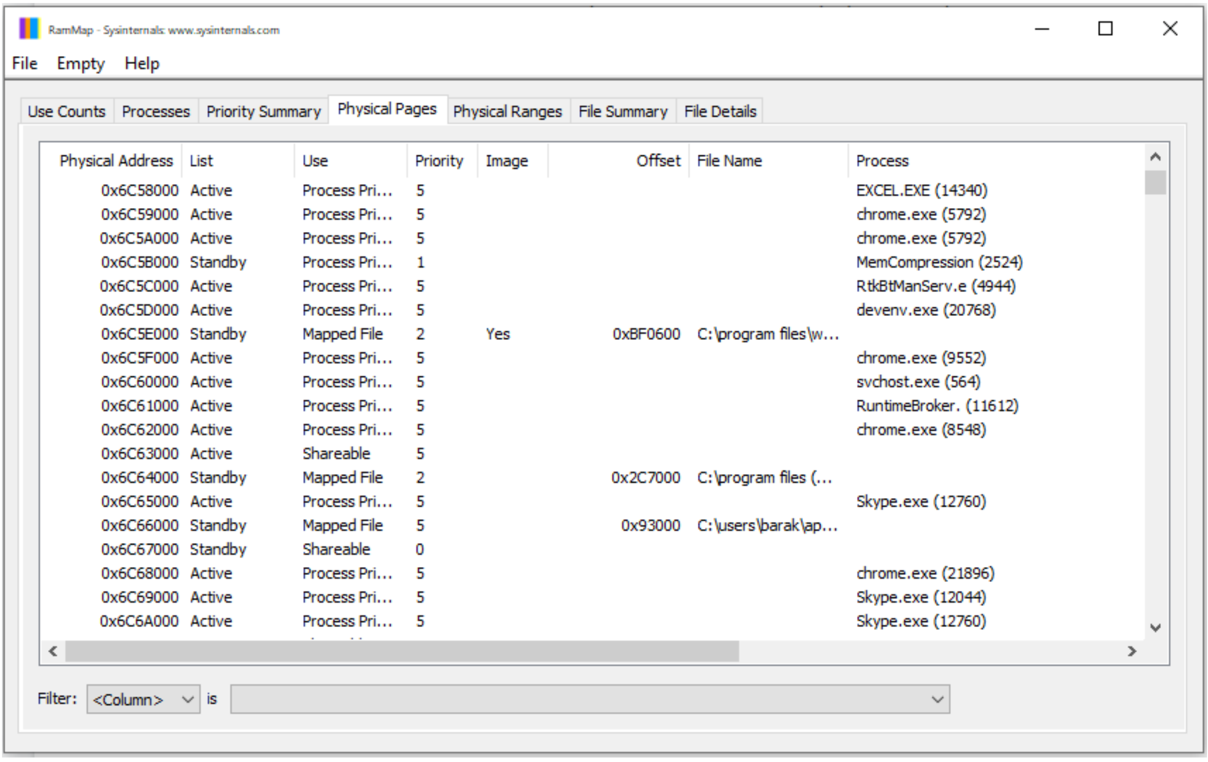
Usage	Total	Active	Standby	Modified	Modified ...	Transition	Zeroed
Process Private	4,510,348 K	4,253,968 K	194,548 K	61,832 K			
Mapped File	9,254,768 K	1,286,636 K	7,967,348 K	784 K			
Shareable	561,404 K	258,064 K	154,888 K	148,452 K			
Page Table	297,484 K	297,484 K					
Paged Pool	502,660 K	501,156 K	1,220 K	284 K			
Nonpaged Pool	513,624 K	513,604 K				20 K	
System PTE	77,284 K	77,168 K	116 K				
Session Private	49,452 K	49,400 K	52 K				
Metafile	494,148 K	223,176 K	270,960 K		12 K		
AWE							
Driver Locked	27,660 K	27,660 K					
Kernel Stack	48,764 K	46,484 K	796 K	1,484 K			
Unused	310,016 K	29,660 K	20 K				58,584
Large Page	1,320 K	1,320 K					
Total	16,648,932 K	7,565,780 K	8,589,948 K	212,836 K	12 K	20 K	58,584

נתחיל מהשורה התחתונה- סכום אזורי הזיכרון מסתכם בגודלו של ה-RAM. במקרה שמוצג לפניכם, 16GB.

ניתן לראות את ההפרדה בין זיכרון פרטי ומשותף.
זיכרון Process Private, כלומר כל המידע ששייך ל-Process ספציפי. שימו לב שזהו סיכום, כלומר אם ל-Process א' יש 200MB מידע פרטי ול-Process ב' יש MB300 מידע פרטי, אז יוצג לפנינו 500MB מידע פרטי.

בשורה מתחתיו מפורטת כמות הזיכרון שמוקדשת ל-Mapped Memory. בשלב זה כבר אין צורך להסביר. הכוונה ב-Shareable Memory היא לזיכרון שיש לו אפשרות להיות משותף, לדוגמה קובץ שביצעו לו מיפוי עם הרשאות שיתוף, אך הוא עדיין לא שותף בפועל עם Process אחר.

מי שרוצה לראות מה בדיוק יש בכל Page Frame של זיכרון ב-RAM, יכול להכנס לטאב Physical Pages:

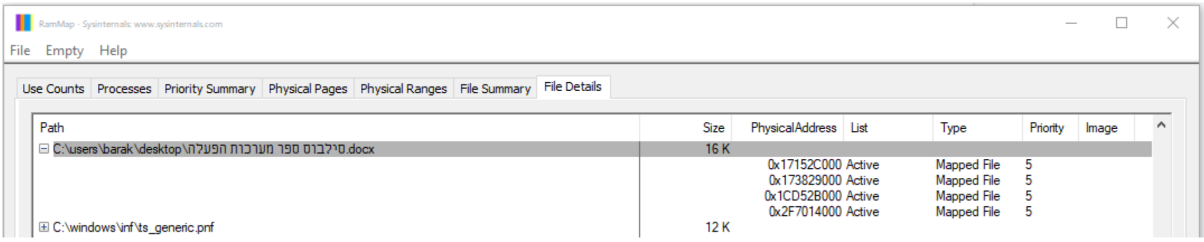


Physical Address	List	Use	Priority	Image	Offset	File Name	Process
0x6C58000	Active	Process Pri...	5				EXCEL.EXE (14340)
0x6C59000	Active	Process Pri...	5				chrome.exe (5792)
0x6C5A000	Active	Process Pri...	5				chrome.exe (5792)
0x6C5B000	Standby	Process Pri...	1				MemCompression (2524)
0x6C5C000	Active	Process Pri...	5				RtkBtManServ.e (4944)
0x6C5D000	Active	Process Pri...	5				devenv.exe (20768)
0x6C5E000	Standby	Mapped File	2	Yes	0xBF0600	C:\program files\w...	
0x6C5F000	Active	Process Pri...	5				chrome.exe (9552)
0x6C60000	Active	Process Pri...	5				svchost.exe (564)
0x6C61000	Active	Process Pri...	5				RuntimeBroker. (11612)
0x6C62000	Active	Process Pri...	5				chrome.exe (8548)
0x6C63000	Active	Shareable	5				
0x6C64000	Standby	Mapped File	2		0x2C7000	C:\program files (...	
0x6C65000	Active	Process Pri...	5				Skype.exe (12760)
0x6C66000	Standby	Mapped File	5		0x93000	C:\users\barak\ap...	
0x6C67000	Standby	Shareable	0				
0x6C68000	Active	Process Pri...	5				chrome.exe (21896)
0x6C69000	Active	Process Pri...	5				Skype.exe (12044)
0x6C6A000	Active	Process Pri...	5				Skype.exe (12760)

אפשר לראות שבמקרה זה גודלו של כל Page Frame הוא 1KB. אפשר לראות שהזיכרון ב-RAM כלל לא מזכיר את הסדר והארגון שנדמה שיש בזיכרון הוירטואלי. קשה למצוא שני Page Frames סמוכים ששייכים לאותו Process...

עמודת ה-Priority הינה העדיפות של אותו הזיכרון, כלומר אם מידע שנמצא ב-Page הוא בעדיפות נמוכה אז המידע עלול להיות מוחלף על ידי מידע אחר. לעומת זאת, Page שהמידע שמסומן בו נמצא בעדיפות גבוהה צפוי להשאר ב-RAM גם אם יהיה מחסור ב-RAM. אם תקליקו על עמודת העדיפות היא תסתדר מהנמוך לגבוה ואז תוכלו לראות איך קבצי DLL של מערכת ההפעלה נמצאים בעדיפות הגבוהה ביותר.

הטאב האחרון, File Details, מאפשר לנו לראות את הדפים הפיזיים שהוקצו לכל קובץ שלנו שנמצא ב-RAM. בחרו קובץ כלשהו, לדוגמה מסמך word, ופיתחו אותו. באמצעות CTRL+F הכנסו לאפשרות החיפוש וחפשו את שם הקובץ שלכם. תוכלו לראות ממש את הדפים הפיזיים שהוא תופס ב-RAM :)



תרגיל 7.3 **מסכם**

שנו את התו הראשון בקובץ gibrish.bin לתו "*". אך עליכם לפעול לפי המגבלה הבאה, שתאלץ אתכם להשתמש ב-Shared Memory: Process לא יכול לבצע הן קריאה של קטע קובץ והן שינוי שלו, אלא רק אחת משתי הפעולות. במילים אחרות Process מספר 1 צריך לבצע את הקריאה לזיכרון המשותף, ואילו Process מספר 2 צריך לבצע את השינוי של התו.

איך Process יכול לדעת אם הוא מספר 1 או מספר 2? אפשר להשתמש בטריק הבא: הדבר הראשון ש-Process עושה הוא לבדוק אם מישהו כבר מיפה זיכרון בשם MyFile או כל שם שתבחרו. אם בוצע כבר מיפוי, הרי שה-Process יודע שהוא מספר 2.

7.5 סיכום

בפרק זה למדנו כיצד שני Processים יכולים לשתף ביניהם זיכרון. התחלנו בהסבר כללי על מיפוי זיכרון I/O כפי שמבוצע לעיתים בזיכרון של כרטיסי מסך. לאחר מכן עברנו לדון במיפוי של קבצי מהדיסק הקשיח אל זיכרון ה-RAM. התנסינו בשימוש בפונקציות של WinAPI שמבצעות את הפעולות השונות הנדרשות לטובות המיפוי: CreateFileMapping, MapViewOfFile, UnmapViewOfFile וכמובן CloseHandle.

תוך כדי הלימוד, ראינו כיצד השימוש במיפוי זיכרון מאפשר לנו לחסוך RAM באופן משמעותי, כך שניתן לעבוד על קבצים גדולים בלי לשנות את כמות ה-RAM הנדרש לעבודב.

הבנת הטכניקה של מיפוי הזיכרון איפשרה לנו להבין במהירות את תהליך שיתוף הזיכרון, בו שני Processים משתמשים בזיכרון פיזי זהה שממופה אל שניהם, לדוגמה ה-Kernel ו-DLLים.

תרגיל 7.4 **תרגיל מסכם לפרקי הלימוד עד כה – "זמן של מספרים"**

עד כה התרגיל המסכם של כל פרק כלל תרגיל ספציפי לנושא שהפרק עסק בו. תרגיל זה ישלב ביחד חלק ניכר מהנושאים שלמדנו, הן בפרק זה והן בפרקים הקודמים:

- Processים
- Threadים
- מנעולים
- שיתוף זיכרון

המשימה בתרגיל- נכתוב קוד שמוצא את כל המספרים הראשוניים עד 1,000,000 (מיליון). עלינו לתכנת קוד כך שהמשימה תבוצע בזמן הקצר ביותר.

הרעיון הכללי של המשימה הוא לחלק את העבודה בין Processים על מנת להאיץ את מהירות החישוב. תוך כדי, נחקור את ההשפעה של החלוקה לכמות משתנה של Processים ונמצא את המספר האופטימלי להאצת החישוב.

להלן שלבי הדרכה לביצוע המשימה.

שלב א – חלוקה ל-Processים

נתחיל בכתיבת הקוד של ה-Process האב.

הרעיון הכללי הוא שכל Process בן יקבל תחום של מספרים שבתוכו עליו לחפש מספרים ראשוניים. כאשר Process בן יסיים את העבודה על התחום המבוקש, הוא ימשיך לתחום המספרים הבא וכך יעשו כל ה-Processים עד סיום כל תחום המספרים עד 1,000,000.

מה יעשה Process בן שמגלה מספר ראשוני? עליו לכתוב אותו לזיכרון משותף. לכן, ה-Thread הראשי של ה-Process האב יגדיר אזור זיכרון משותף, שלתוכו כל Process בן שמוצא ראשוני יכתוב אותו.
במהלך הלימוד ראינו כיצד ניתן לבצע מיפוי זיכרון ושיתוף שלו, ספציפית של קובץ מהדיסק הקשיח. יש אפשרות כמובן גם לשתף אזור זיכרון שאינו מיפוי של קובץ מהדיסק, תוך שימוש בקוד זהה כמעט לחלוטין, כאשר ההבדל הוא שהפונקציה CreateFileMappingA לא מקבלת את המצביע לקובץ על הדיסק בתור פרמטר, אלא INVALID_HANDLE_VALUE. הפרטים נמצאים באתר msdn

https://docs.microsoft.com/en-us/windows/win32/memory/creating-named-shared-memory

יש לנו מושג כללי לגבי גודל הזיכרון המשותף שניצור, הוא צריך להיות קצת מעל כמות המספרים הראשוניים שאנחנו מצפים למצוא. כמה מספרים כאלה ישנם? לפי התאוריה צריכים להיות בערך 1,000,000 חלקי ln של 1,000,000. כלומר בערך 72,000 מספרים ראשוניים. ניקח ספייר, נשמור מקום ל-80,000 מספרים.

כיוון שכל ה-Processים הבנים כותבים בו זמנית לזיכרון המשותף, עלול להיות מצב של התנגשות. נתייחס לזיכרון המשותף כמערך של intים. נניח שעד כה מצאנו n מספרים. המספר הראשוני הבא צריך להיכתב לאינדקס n במערך. האינדקס n צריך להיות מוגן באמצעות מנעול, כדי שלא יהיה מצב שבו שני Processים

מנסים לכתוב לאותו אינדקס במערך. שימו לב כמובן לכך שגם האינדקס n צריך להיות משתנה שנשמר בזיכרון משותף. נגדיר שהאיבר הראשון של הזיכרון המשותף יהיה האינדקס הבא אליו יש לכתוב מספר ראשוני.

המחשה – כך נראה אזור הזיכרון המשותף:

5 (מספר שהוא האינדקס הבא אליו יש לכתוב מספר ראשוני שיימצא)

1, 2, 3, 5, 7 (המספרים הראשוניים שנמצאו עד כה, שמגיעים עד אינדקס 4)

כדי להקל את התכנות מומלץ להגדיר struct שכולל משתנה בשם index ומערך.

שלב ב – כתיבת קובץ EXE שמוצא מספרים ראשוניים בטווח מוגדר

כדי להריץ את התוכנית שלנו, נצטרך לספק ל-CreateProcess את קובץ ה-EXE שכל אחד מה-Processים הבנים צריך להריץ. בשלב זה ניצור אותו, אך נבצע אותו בצורה פשוטה יחסית רק כדי לבדוק שהדברים עובדים היטב עד כה.

כל Process יקבל בתור פרמטר את המספר שלו. נגדיר באופן קשיח בקוד ש-Process מספר 1 מחפש מספרים בתחום 0-249999, ש-Process מספר 2 מחפש בתחום 250000-499999 וכך הלאה (אנחנו מניחים שה-Process האבא יוצר ארבעה Processים בנים, כמובן שבהמשך נשדרג את הקוד כך שמספר זה יוכל להשתנות).

בשלב זה נכתוב את הקוד שמוצא מספרים ראשוניים בתחום מוגדר, ומשתמש ב-Mutex על הזיכרון המשותף בשביל לכתוב את המספרים שהתגלו על ידו. הקוד הפשוט יחסית שכתבנו בשלב זה יעזור לנו לוודא שעד כה הדברים עובדים היטב.

שלב ג – שדרוג הקוד לניהול חכם יותר של ה-Processים הבנים

בשלב זה נשדרג את הקוד שמריצים ה-Processים הבנים, כך שהם לא תמיד ירוצו על אותו תחום מספרים אלא יעבדו בגישת "קח תחום מספרים קטן וכשתסיים אותו תחזור לקבל עוד". אך לפני כן אנחנו צריכים לשדרג את ה-Process האב כך שיאפשר חלוקת עבודה חכמה בין ה-Processים הבנים.

הרעיון הכללי הוא שהזיכרון המשותף שלנו יכלול גם משתנה שאומר מה תחום המספרים הבאים שצריך לבדוק. כל Process בן שיסיים לעבור על תחום המספרים שהוקצה לו, ייגש אל הזיכרון המשותף וייקרא מה תחום המספרים הבא שמוקצה לו. לאחר מכן, ה-Process הבן ישנה את התחום, לדוגמה בקפיצה של 10000 מספרים, כך שכאשר ה-Process הבא יקרא את התחום הוא ימצא תחום חדש שעליו לחקור.

המחשה – כך נראה אזור הזיכרון המשותף לאחר השיפור:

1352 (מספר שהוא האינדקס הבא אליו יש לכתוב מספר ראשוני שיימצא)

340000 (תחילת התחום הבא ש-Process שסיים חיפוש צריך לקבל)

1, 2, 3, 5, 7... (מספרים ראשוניים, שמגיעים עד אינדקס 1351)

לאחר שה-Process שסיים חיפוש קיבל את המשימה לחפש 10,000 מספרים מהמספר 340000, הוא יעדכן את הזיכרון המשותף כך שה-int השני במערך יהיה 350000.

כל מה שנצטרך לעשות הוא לשדרג את ה-struct שהגדרנו כך שיכלול משתנה של startingPoint.

שלב ד' – מדידת זמן

הוסיפו מנגנון מדידת זמן ובידקו כיצד זמן הריצה הכולל משתנה עבור כמות שונה של Processים, בין 1 ל-10. הסבירו את הגרף המתקבל.

בונוס - חלוקה ל-Threadים

בידקו את נושא שיפור הביצועים כאשר כל Process משתמש ביותר מאשר Thread יחיד על מנת למצוא מספרים ראשוניים. החלוקה בין ה-Threadים יכולה להתבצע באופן הפשוט ביותר שעולה על דעתכם, לדוגמה חלוקת תחום המספרים לכמה תתי תחומים.

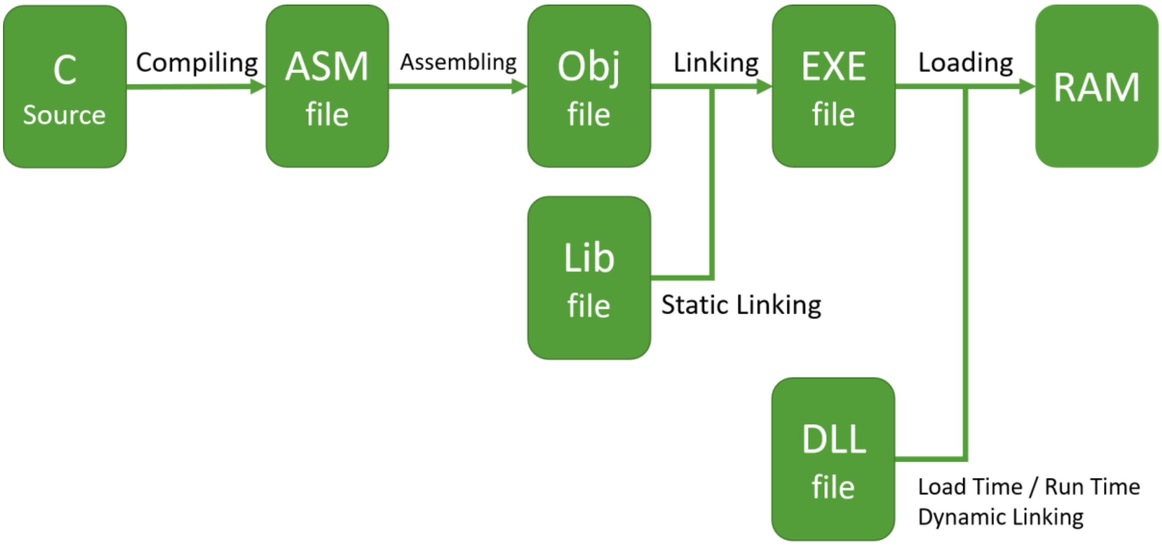
https://www.youtube.com/watch?v=vmV3IETpiQU

פרק 8 – תהליך ה-Linking ויצירת DLLים

בפרקים שדנו בזיכרון התבוננו בזיכרון מנקודת מבט של המעבד. הבעיה שהצגנו היתה ניתנת לתיאור באופן הבא – "המעבד רוצה לפנות לכתובת מסויימת במרחב הכתובות הוירטואלי שלו, כיצד הכתובת הזו מאותרת ב-RAM?". בשלב זה אנחנו הולכים להחליף את נקודת המבט ולשאול את השאלה "כיצד בכלל יודע המעבד לאיזו כתובת וירטואלית עליו לפנות כדי לגשת לפקודה או למשתנה כלשהם? מה שרשרת התהליכים שגורמת לכך שמשתנה או פקודה כלשהי יופיעו בכתובת מסויימת בזיכרון ב-RAM, ואיך המעבד יודע לאן לגשת?".

עקב מורכבות וריבוי הפרטים, נחלק את התשובה שלנו לשני פרקים. בפרק הנוכחי נפרט אודות תהליך יצירת קבצי הרצה כאשר תוך כדי הלימוד של קבצי הרצה נלמד גם על יצירת קבצי DLL, שכפי שנראה יש להם פורמט זהה. בפרק הבא "פורמט PE ותהליך ה-Loading" נצא מתוך הנחה שיש בידינו קובץ הרצה או קובץ DLL ונבין כיצד הם נטענים לזיכרון.

סיכום התהליך מתואר בדיאגרמה הבאה. נתחיל מצד שמאל של הדיאגרמה ונתקדם שלב שלב עד שנבין את התהליך כולו.

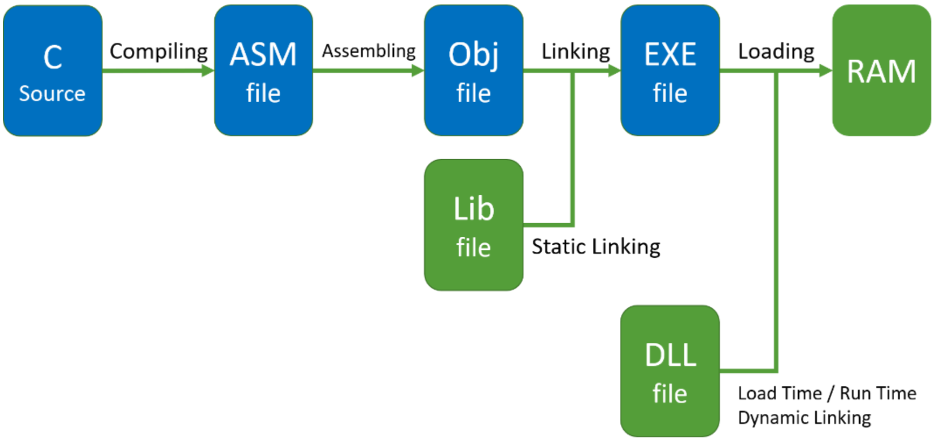


תוך כדי הלימוד נענה על השאלות הבאות:

1. מהם שלבי ההמרה של קבצי מקור בשפת C לקבצי הרצה exe
2. מהי library ומהו Static Linking
3. מהו DLL
4. כיצד לייבא DLL ב-Load Time
5. כיצד לייבא DLL ב-Run Time
6. מהו DLL Injection ואיך אפשר לבצע פעולה זו באמצעות הרגיסטרי
7. לבסוף, נסקור שיטה שמבצעת DLL Injection באמצעות פתיחת thread בתוך process אחר

8.1 תהליך המרת קוד משפת C לקובץ הרצה exe

המטרה שלנו היא להבין את השינויים שעובר הקוד שלנו בדרכו להפוך לשפת מכונה, נתמקד בכתובות בזיכרון בהן נשמרים המשתנים והקוד שלנו. ההסבר שלנו יכלול את השלבים הצבועים בכחול:



נתחיל את התהליך משני קבצי C. הקובץ הראשון, main_file.c, כולל את הקוד הבא:

```
#include <stdio.h>
#include "my_funcs.h"

int main(){
    static int a = 2;
    static int b = 3;
    int c;
    c = my_add(a, b);
    printf("%d", c);
    return 0;
}
```

המשתנים a ו-b מוגדרים כ-static בכוונה, על מנת שיישמרו ב-data section ולא במחסנית.

8.1.1 תהליך ה-Compiling

נוח לצפות בתוצרי תהליך ה-Compiling כאשר משתמשים בקומפיילר gcc של MinGW (סביבת קוד פתוח עבור windows). להלן הקוד של פונקציית ה-main, הפעם בשפת אסמבלי:

```
movl    _b.2064, %edx
movl    _a.2063, %eax
movl    %edx, 4(%esp)
movl    %eax, (%esp)
call    _my_add
movl    %eax, 28(%esp)
movl    28(%esp), %eax
movl    %eax, 4(%esp)
movl    $LC0, (%esp)
call    _printf
movl    $0, %eax
leave
```

למי מכם ששולט באסמבלי, שימו לב לכך שסדר האופרנדים הפוך מהרגיל. לדוגמה בשורה הראשונה _b.2064 מועתק לתוך edx, ולא להיפך. זהו תחביר אסמבלי של חברת AT&T. למי מכם שאינו שולט באסמבלי, הסבר קצרצר. הקוד מחולק ל-3 חלקים. החלק הראשון כולל את הקוד עד call my_add, החלק השני כולל את הקוד עד call printf והחלק השלישי הוא סיום ריצת ה-main וביצוע return 0.

בכל אחד משני החלקים הראשונים מתבצעת העברה של הפרמטרים לתוך הפונקציה. לדוגמה במקרה של my_add, יש להעביר לה את הפרמטרים a, b. לאחר ריצת כל אחת מהפונקציות, ערך החזרה של הפונקציה נכנס לתוך הרגיסטר eax. השימוש ברגיסטר eax להחזרת ערך התוצאה של פונקציה הוא מוסכמה מקובלת בקריאה לפונקציות, מה שנקרא Calling Convention (הסבר מפורט בספר האסמבלי של המרכז לחינוך סייבר). כך, לאחר סיום ריצת הפונקציה my_add, רואים שמופיעות שורות קוד עם הרגיסטר eax, שמעביר את תוצאת החישוב אל המשתנה המקומי שהוגדר.

כפי שבטח שמתם לב, שמות הפונקציות והמשתנים כוללים קו תחתי (Underscore). הסיבה לכך היא שיכול להיות שמתכנת בשפה עילית קרא לפונקציה או למשתנה בשם של פקודת אסמבלי. לדוגמה אפשר לקרוא לפונקציה בשם "call" או "leave". אילו הקומפיילר היה משאיר את השמות כפי שהם, הדבר היה עלול לגרום בלבול בשלב הבא, בו מבוצעת ההמרה משפת אסמבלי לשפת מכונה. לכן הקומפיילר מוסיף _ לפני כל משתנה או פונקציה, כך שיהיה ברור שזו אינה פקודת אסמבלי.

המשתנים a ו-b הוחלפו בסימבולים _a.2063 ו- _b.2064. הסימבולים הללו מציינים שיש להם כתובת בזיכרון, אך כרגע הכתובת אינה ידועה. בהמשך התהליך יהיה צורך לפענח את הסימבולים הללו ולהחליף אותם בכתובות בזיכרון. שימו לב לכך שהמשתנה c, שלא הוגדר כסטטי ולכן נשמר על המחסנית, לא דורש

סימבול. המשתנה הזה נשמר בכתובת שהיא מיקום יחסי מתחילת המחסנית ולכן הכתובת שלו מבוטאת יחסית לרגיסטר המחסנית esp.

הקובץ השני, my_funcs.c, כולל את ההגדרה של הפונקציה my_add:

```
int my_add(int a, int b)
{
    static int one = 1;
    return a+b+one;
}
```

כאשר נבצע לו Compiling, נקבל את קוד האסמבלי הבא:

```
movl     8(%ebp), %edx
movl     12(%ebp), %eax
addl     %eax, %edx
movl     _one.1554, %eax
addl     %edx, %eax
```

גם במקרה זה, המשתנה הסטטי one מקבל סימבול ולא כתובת בזיכרון. כדי שהמעבד יוכל לפנות למשתנה הזה, הוא יהיה צריך לקבל כתובת.

8.1.2 תהליך ה-Assembling

נעבור לשלב ה-Assembling. בשלב זה נוצר מקוד האסמבלי קוד בשפת מכונה, ששמור בקובץ עם הסיומת .obj. זהו קובץ בפורמט בינארי, שמתאים בדרך כלל למערכת ההפעלה שיצרה אותו, אך עדיין אינו בשל להרצה, כפי שמיד נראה.

נפתח את הקובץ באמצעות תוכנת IDA, אשר כזכור לנו מפרק המבוא ממירה מקבצים בשפת מכונה לקוד אסמבלי.

```
seg000:00000112 8B 15 00 00 00 00      mov     edx, ds:dword_0
seg000:00000118 A1 04 00 00 00         mov     eax, ds:dword_4
seg000:0000011D 89 54 24 04            mov     [esp+4], edx
seg000:00000121 89 04 24               mov     [esp], eax
seg000:00000124 E8 00 00 00 00         call    $+5
seg000:00000129 89 44 24 1C            mov     [esp+1Ch], eax
seg000:0000012D 8B 44 24 1C            mov     eax, [esp+1Ch]
seg000:00000131 89 44 24 04            mov     [esp+4], eax
seg000:00000135 C7 04 24 00 00 00 00   mov     dword ptr [esp], 0
seg000:0000013C E8 00 00 00 00         call    $+5
seg000:00000141 B8 00 00 00 00         mov     eax, 0
seg000:00000146 C9                     leave
```

מהן הכתובות בזיכרון של סגמנט הקוד שלנו? בשלב זה הכתובות נראות מוזרות למדי, בין הכתובות 0x112 ל-0x146 מתחילת סגמנט הקוד. חשוב להבין על מה אנחנו מתבוננים כרגע. פתחנו קובץ שנמצא על הדיסק

הקשיח, אנחנו מתבוננים בפקודות בשפת מכונה שהכתובות שלהן בזיכרון לא מייצגות את מה שיקרה כאשר תבוצע טעינה של הקוד לזיכרון. בשפה המקצועית, מה שאנחנו מבצעים הינו Static Analysis, כלומר ניתוח קובץ קוד שנמצא על הדיסק. זאת בניגוד ל-Dynamic Analysis, שהוא ניתוח של קוד שנטען ל-RAM. בהמשך נגיע גם לכך.

האזורים המודגשים באדום הם הכתובות בזיכרון אליהם תורגמו המשתנים הסטטיים והפונקציות. מה מופיע לפנינו? המשתנה a נמצא בכתובת בזיכרון 0. המשתנה b נמצא בכתובת 4. הפונקציות my_add ו-printf נמצאות שתיהן בכתובת 0 בזיכרון. המחרוזת "%d" נמצאת אף היא בכתובת 0 (כפי שרואים בשורת הקוד בכתובת 0x135). כמובן שהדבר אינו אפשרי. אפשר להסיק מכך שאומנם הקוד שלנו עבר לשפת מכונה, אבל אף אחת מהכתובות שרשומות למעבד לגשת אליהן איננה נכונה.

איזה מידע יש בקובץ obj?

1. מידע כללי לגבי הקובץ- שם, גודל, זמן יצירה וכו'
2. קוד בינארי – פקודות בשפת מכונה, שהינן התרגום של פקודות האסמבלי שהתקבלו לאחר הקומפילציה
3. מידע לגבי Relocation- זהו מונח חדש שנשתמש בו גם בהמשך. כפי שראינו, הקוד הבינארי עדיין לא מכיל כתובות הגיוניות בזיכרון. כל מקום שיש בו כתובת בזיכרון צריך להיות מתוקן בשלב ה-Linking, והתהליך של שינוי כתובת בזיכרון נקרא Relocation, קצת בדומה לשינוי כתובת מגורים בעולם האמיתי. פעולת תיקון של כתובת ספציפית נקראת לעיתים גם "fix up". כדי שהלינקר יידע אילו כתובות צריך לעבור Relocation, קובץ ה-obj מכיל טבלה עם המידע.
4. טבלת סימבולים Symbols- הטבלה כוללת את שמות הסימבולים הגלובליים שהוגדרו בקובץ המקור ואת שמות הסימבולים שצריך לייבא מקבצים חיצוניים. במקרה שלנו, המשתנה a הוא סימבול שמוגדר בקובץ המקור, ואילו הפונקציה my_add היא סימבול שמיובא לקוד שלנו.
5. מידע לטובת דיבוג – לדוגמה, שמות של משתנים ושל פונקציות. כפי שראינו בעבר ומיד נראה שוב, כאשר אנחנו פותחים באמצעות תוכנת ida קובץ exe שיצרנו, מוצגים לפנינו השמות של הפונקציות. לעומת זאת, ללא מידע לטובת דיבוג, נקבל רק כתובות בזיכרון.

באותו אופן, גם הקובץ הבינארי של my_funcs מכיל את פקודות האסמבלי, אך עם כתובות לא מפוענחות בזיכרון:

```
seg000:000000DF 8B 55 08          mov     edx, [ebp+8]
seg000:000000E2 8B 45 0C          mov     eax, [ebp+0Ch]
seg000:000000E5 01 C2             add     edx, eax
seg000:000000E7 A1 00 00 00 00    mov     eax, ds:dword_0
seg000:000000EC 01 D0             add     eax, edx
```

שימו לב לכתובות בזיכרון של הקוד, הכתובות היחסיות לתחילת הסגמנט נמצאות בין 0xDF לבין 0xEC. למרות שבמקרה אין חפיפה עם הכתובות של הקובץ השני, יכל בהחלט להיות מצב שבו שני הקבצים מקבלים אזורי כתובות חופפים עבור הקוד שלהם. אי אפשר לטעון את הקבצים הבינאריים לזיכרון כפי שהם.

8.1.3 תהליך ה-Linking

מי שתפקידו לספק את הכתובות החסרות הוא ה-Linker. תהליך ה-Linking מאחד את כל אזורי הקוד לאזור אחד ואת כל אזורי הנתונים לאזור אחד. כעת, כאשר ברור איזו פקודה מגיעה אחרי איזו פקודה ומהם כל המשתנים שצריך להקצות להם זיכרון, אפשר לחלק כתובות וירטואליות לכל פונקציה ומשתנה.

```
.text:00401410 55                               push    ebp
.text:00401411 89 E5                            mov     ebp, esp
.text:00401413 83 E4 F0                         and     esp, 0FFFFFFF0h
.text:00401416 83 EC 20                         sub     esp, 20h
.text:00401419 E8 62 05 00 00                   call    ___main
.text:0040141E 8B 15 04 40 40 00                mov     edx, _b_2064
.text:00401424 A1 08 40 40 00                   mov     eax, _a_2063
.text:00401429 89 54 24 04                      mov     [esp+4], edx
.text:0040142D 89 04 24                         mov     [esp], eax
.text:00401430 E8 1F 00 00 00                   call    _my_add
.text:00401435 89 44 24 1C                      mov     [esp+1Ch], eax
.text:00401439 8B 44 24 1C                      mov     eax, [esp+1Ch]
.text:0040143D 89 44 24 04                      mov     [esp+4], eax
.text:00401441 C7 04 24 44 50 40+               mov     dword ptr [esp], offset aD ; "%d"
.text:00401448 E8 73 26 00 00                   call    _printf
.text:0040144D B8 00 00 00 00                   mov     eax, 0
.text:00401452 C9                               leave
.text:00401453 C3                               retn
.text:00401453                  _main           endp
.text:00401453
.text:00401454
.text:00401454                  ; =============== S U B R O U T I N E ===========================
.text:00401454
.text:00401454                  ; Attributes: bp-based frame
.text:00401454
.text:00401454                                  public _my_add
.text:00401454                  _my_add         proc near               ; CODE XREF: _main+20↑p
.text:00401454
.text:00401454                  arg_0           = dword ptr  8
.text:00401454                  arg_4           = dword ptr  0Ch
.text:00401454
.text:00401454 55                               push    ebp
.text:00401455 89 E5                            mov     ebp, esp
.text:00401457 8B 55 08                         mov     edx, [ebp+arg_0]
.text:0040145A 8B 45 0C                         mov     eax, [ebp+arg_4]
.text:0040145D 01 C2                            add     edx, eax
.text:0040145F A1 0C 40 40 00                   mov     eax, _one_1554
.text:00401464 01 D0                            add     eax, edx
.text:00401466 5D                               pop     ebp
.text:00401467 C3                               retn
.text:00401467                  _my_add         endp
```

נדון בשלושת החלקים שקבע ה-Linker: הכתובות של הפקודות, הכתובות של המשתנים, הכתובות של הפונקציות.

8.1.3.1 כתובות הפקודות

כפי שרואים, הפקודה

```
mov     edx, _b_2064
```

נמצאת בכתובת 0x0040141E. זהו אזור זיכרון שמוגדר בתור text, כלומר קוד. יש לו הרשאות מתאימות (קריאה והרצה). הפקודה הבאה נמצאת בכתובת 0x00401424, כלומר 6 בתים לאחר תחילת הפקודה

הקודמת. אם תיזכרו בתרגיל הקטן שביצענו כאשר למדנו על מבנה כתובות וירטואליות, בו פירקנו את הכתובת 0x00401000 למרכיביה השונים, תשימו מיד לב לכך שה-Linker קבע לפקודות כתובות וירטואליות.

אנחנו עדיין מסתכלים בקוד כפי שהוא שמור בקובץ על הדיסק הקשיח. יש אפשרות שלאחר שהקוד ייטען ל-RAM יוקצו לו כתובות וירטואליות אחרות. כלומר יכול להיות שהפקודה הראשונה תיטען לכתובת 0x0050141E. אך דבר אחד יישאר קבוע- ההפרש של הבתים בין הפקודות. תהליך הטעינה לוקח את כל הפקודות וקובע להן כתובות וירטואליות חדשות או מותיר אותן במקום שבו ה-Linker המליץ לשים אותן, אך בכל מקרה כל הפקודות זזות יחד כקבוצה.

8.1.3.2 כתובות המשתנים

נסקור את הכתובות הוירטואליות שקבע ה-Linker למשתנים שלנו (אם אינכם מבינים את הקשר בין הכתובות בקוד לבין הטבלה, קיראו על "Little Endian" בספר האסמבלי של המרכז לחינוך סייבר) :

המשתנה	כתובת וירטואלית
b	0x00404004
a	0x00404008
one	0x0040400C

חשוב להבין שלמרות שה-Linker העניק כתובות לכל המשתנים, אנחנו עדיין לא יודעים מה הכתובות שייעשה בהן שימוש כאשר הקובץ ייטען ל-RAM. אפשר להתייחס אל הכתובות שה-Linker קבע בתור "המלצות". מה שכן יישמר הוא היחסיות ביניהן. כלומר אם ה-Linker קבע שמשתנה a נמצא בכתובת 0x00404008 ואילו המשתנה b נמצא בכתובת 0x00404004, אז הפער ביניהם בגודל 4 בתים יישמר גם אם הן יוזזו ממקומם בהמשך.

8.1.3.3 כתובות הפונקציות

הכתובות שהוענקו לפונקציות הן כתובות יחסיות לפקודה שמבצעת להן call, כלומר הן בעצם מבטאות כמה בתים צריך לקפוץ כדי להגיע לפונקציה:

הפונקציה	כתובת יחסית
my_add	0x0000001F
printf	0x00002673

לדוגמה, הקריאה לפונקציה my_add מתבצעת בפקודה שנמצאת בכתובת 0x00401430 וגודלה 5 בתים. כלומר כשהפקודה הזו מורצת הרגיסטר EIP כבר מצביע על הכתובת של הפקודה הבאה אותה צריך להריץ, 0x00401435. הקריאה לפונקציה אומרת "הזז את EIP כ-0x1F בתים מהמקום בו הוא נמצא כרגע". ההזזה הזו, אם תבצעו את החישוב, מעלה את ערכו של EIP להיות 0x00401454. זו בדיוק הכתובת הוירטואלית בה נמצאת הפונקציה my_add.

הבנו את התהליך שעבר הקוד שכתבנו עד להפיכתו לקובץ exe, אולם יש קטע קוד אחד שלא אנחנו כתבנו ולמרות זאת מופיע בקוד שלנו- הפונקציה printf.

הפונקציה printf אמורה להיות כ-0x2673 בתים מהמקום בו היא נקראת, כלומר בכתובת 0x00403AC0. נבדוק מה יש בכתובת זו:

```
.text:00403AC0                    ; =============== S U B R O U T I N E =======================================
.text:00403AC0
.text:00403AC0                    ; Attributes: thunk
.text:00403AC0
.text:00403AC0                    ; int printf(const char *, ...)
.text:00403AC0                                    public _printf
.text:00403AC0                    _printf         proc near               ; CODE XREF: _main+38↑p
.text:00403AC0 FF 25 D8 81 40 00                  jmp     ds:__imp__printf
.text:00403AC0                    _printf         endp
.text:00403AC0
.text:00403AC0                    ; ---------------------------------------------------------------------------
```

אנחנו רואים קפיצה נוספת. הפעם זו אינה קפיצה יחסית אלא קפיצה ממש, לכתובת 0x004081D8

ניגש לכתובת הזו (אפשר לבצע זאת בקלות אם נקליק ב-ida על שם הפונקציה), אבל שוב לא נמצא את הקוד שלה. במקום זאת, נגיע אל אזור זיכרון בשם idata ונמצא את המידע הבא:

```
.idata:00408180 ;
.idata:00408180 ; Imports from msvcrt.dll
.idata:00408180 ;

.idata:004081D8 ; int _printf(const char *, ...)
.idata:004081D8                 extrn __imp__printf:dword
```

כלומר הפונקציה printf מיובאת מתוך קובץ DLL בשם msvcrt.dll. הקובץ הנ"ל הוא קוד מקומפל של מיקרוסופט, שבתוכו יש פונקציות שימושיות כגון הדפסה, הקצאה ושחרור של זיכרון, טיפול במחרוזות. אפשר למצוא בתוך IDA את כל הפונקציות שייבאנו מתוך msvcrt.dll. בהמשך הפרק נעסוק בקבצי DLL, נלמד ליצור אותם ונראה מה קורה לקבצי DLL בזמן תהליך ה-Loading ל-RAM. בשלב זה מה שאנחנו מבינים, הוא שה-Linker לא יודע מה המיקום בזיכרון של הפונקציה printf, אלא הוא סומך על מישהו אחר שישלים את הכתובת הזו. את הפרטים המלאי נבין בפרק הבא, כאשר נלמד על תהליך ה-Loading ועל ה- Import Address Table.

נסכם את מה שלמדנו על ה-Linker: הוא מסדר כתובות וירטואליות לקוד, משתנים ופונקציות, אך עם שני סייגים:

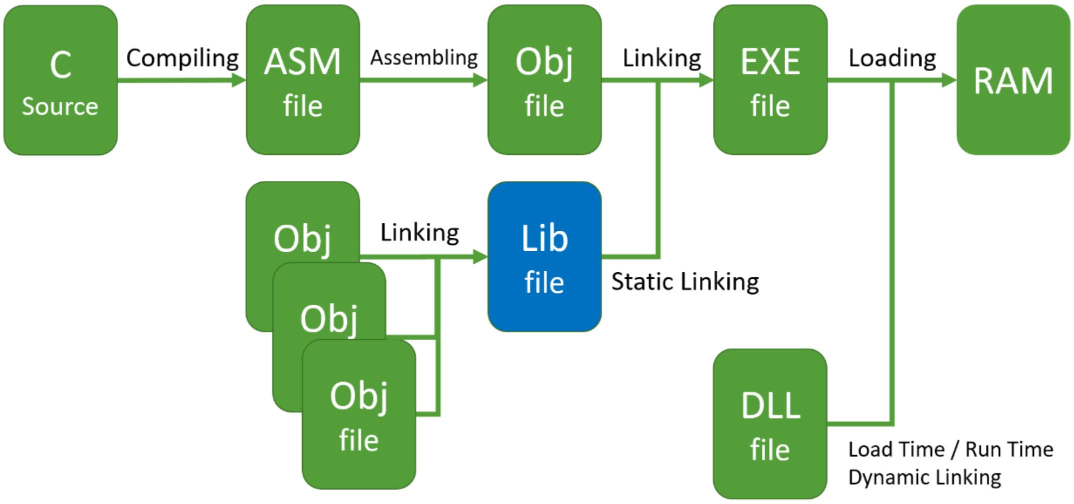
א. הכתובות הללו יכולות לזוז בהמשך, כאשר הקובץ ייטען ל-RAM.

ב. ה-Linker לא מסדר כתובות וירטואליות לפונקציות שמיובאות לקוד שלנו מתוך קבצי DLL.

8.2 יצירת Library וביצוע Static Linking

לפני הכל, חשוב להדגיש שהמונחים והשרטוטים מכאן והלאה נכונים ל-Windows. במערכות הפעלה אחרות יש הבדלים, לדוגמה ב-Linux אין קבצי lib אלא קבצים בסיומת a ובמקום קבצי DLL יש קבצים בסיומת so. אך באופן כללי העקרונות דומים ומי ששולט בתהליכים שעובר קוד ב-Windows יוכל לגשר על ההבדלים ולהבין גם מערכות הפעלה שונות.

ב-Windows, קבצי lib (קיצור של Library) מציינים קוד מקומפל, שעדיין לא עבר את השלב האחרון הנדרש להפיכתו לקוד הרצה exe. בדומה לקובץ obj שסקרנו בסעיף הקודם, הקוד של ה-Library הוא בשפת מכונה, אך המעבד לא יכול להריץ אותו כיוון שלא בוצע לו Linking: הכתובות שרשומות בקוד (פונקציות, משתנים) אינן נכונות. אם יש ברשותנו Library כלשהי, אנחנו יכולים להפוך אותה לקוד הרצה באמצעות פעולת Linking עם קוד אחר שיש לנו. כך יהפוך ה-Library לחלק מקוד התוכנית שלנו, ממש כאילו כתבנו אותו בעצמנו.

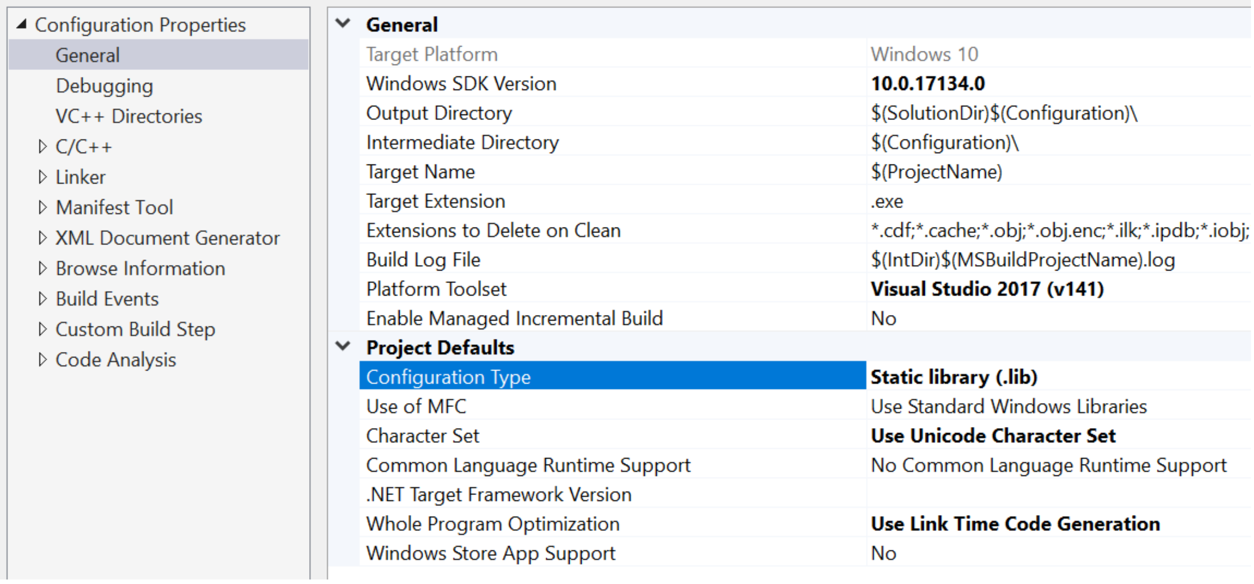


אם כך, מה ההבדל בין Library לקובץ obj? ה-Library הוא תוצר של אחד או יותר קבצי obj שבוצע להם Linking זה עם זה. היתרון של קובץ lib יחיד על פני מספר קבצי obj הוא בכך שישנה טבלה אחת של סימבולים לכל הקובץ. כאשר ה-Linker מבצע את השלב האחרון של המרה לקובץ exe הוא אינו צריך לחפש כל סימבול שאינו מוכר לו בכל קבצי ה-obj האפשריים, אלא רק בטבלה אחת.

בעבר, טרם פיתוח ה-DLLים, הצורך ב-Library נבע מכך שיש קטעי קוד שהם שימושיים למספר רב של תוכנות. לדוגמה, קוד שיודע לקלוט היכן המשתמש הקליק עם העכבר על המסך, יהיה שימושי לכל תוכנה שיש לה חלון משתמש. לכן כל מי שמתכנת תוכנה שיש לה חלון משתמש ירצה לשלב את הקוד הזה בתוכנה שלו. אפשרות אחת היא לקחת את הקוד שמטפל בעכבר ולקמפל אותו יחד עם קוד התוכנה. כך ניצור קובץ הרצה exe שיש בו כבר את כל הפונקציות שמטפלות בעכבר.

אולם בגישה זו יש בעיה: אנחנו צריכים את קוד המקור, ואילו חברות תוכנה אינן מספקות את קוד המקור שלהן, בדרך כלל, מסיבות של סודיות מסחרית. לכן נדרשת שיטה שתאפשר לקמפל קוד שלנו יחד עם קוד של מישהו אחר – מבלי להפר או לפגוע באותה סודיות.

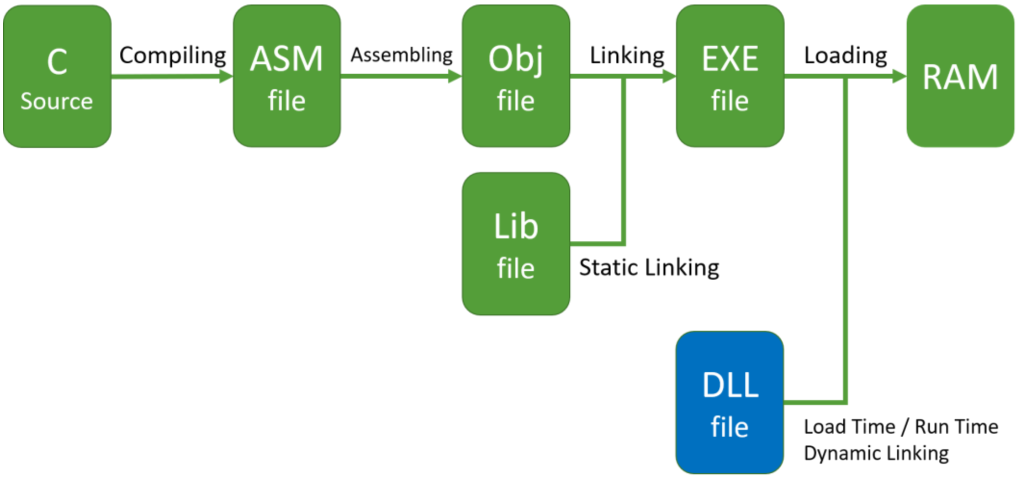
על הבעיה הזו ענה שימוש בקבצי lib. היכנסו אל Visual Studio ופיתחו פרוייקט כלשהו. שנו את הגדרות הפרוייקט כך שהתוצר שלו יהיה קובץ lib. כאשר תבצעו build, תיצרו קובץ lib. את קובץ ה-lib שיצרתם אתם יכולים למסור לכל מי שתרצו שיעשה בו שימוש, והוא יבצע Linking אל הקוד שלו. צורה זו של Linking נקראת **Static Linking**. היתרון שלה הוא שהוא מאפשר לנו לחלוק חבילות קוד, בלי לחלק את קוד המקור שעמלנו רבות על פיתוחו. בעבר, זו היתה הצורה היחידה של שיתוף קוד.



שינוי הגדרות הפרוייקט כך שיווצר קובץ *lib*

8.3 קובץ DLL

בחלק זה נענה על השאלה מהו DLL ומדוע יש בו צורך, ונסביר מה ההבדל בין Load Time Dynamic Linking ו- Run Time Dynamic Linking.



השם DLL הוא ראשי תיבות של Dynamic Link Library. הפורמט של קבצי DLL הומצא על ידי מיקרוסופט, אם כי הרעיון הכללי נמצא גם במערכות הפעלה אחרות. ב-Linux יש קבצים שמשמשים לתכלית דומה, אך הם נקראים קבצי SO, קיצור של Shared Object. הצורך ב-Dynamic Linking נובע מכך שישנן בעיות ש-Static Linking אינו פותר.

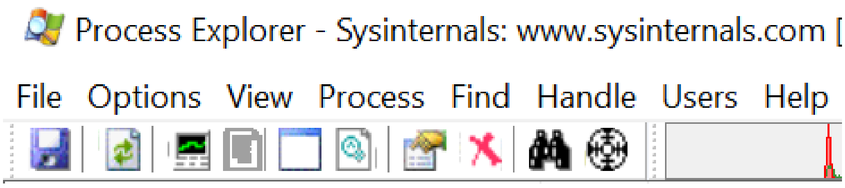
הבעיה הראשונה ב-Static Linking היא קושי בתחזוקה של קוד. נניח שחברת התוכנה שכותבת קוד שמטפל בעכבר שיפרה את הקוד. כעת כל מי שרוצה ליהנות מהשיפור צריך לבצע Linking מחדש של התוכנה שלו עם ה-library המשודרגת ולהפיץ לכל המשתמשים שלו את העדכון. תהליך מסובך ולא יעיל. בנוסף, ביצוע Linking לספריות קוד מגדיל את גודלו של קובץ ההרצה, ה-exe, ועלול לבזבז משאבים. לדוגמה מקום בדיסק הקשיח או כמה טוב היה אם היה מספיק רק לעדכן קובץ, שכולל את הקוד המשודרג של העכבר, וכל התוכנות במחשב שמשתמשות בקובץ הזה יקבלו מיידית את היכולות המשודרגות.

הבעיה השניה היא בזבוז של זיכרון. אם עשר תוכנות נמצאות בזיכרון שלנו, וכולן עושות שימוש בעכבר, אז הפונקציות של העכבר משוכפלות עשר פעמים בזיכרון. מדוע שהקוד שמטפל בעכבר לא ייטען פעם אחת לזיכרון, וכל מי שזקוק לו יקרא אותו משם?

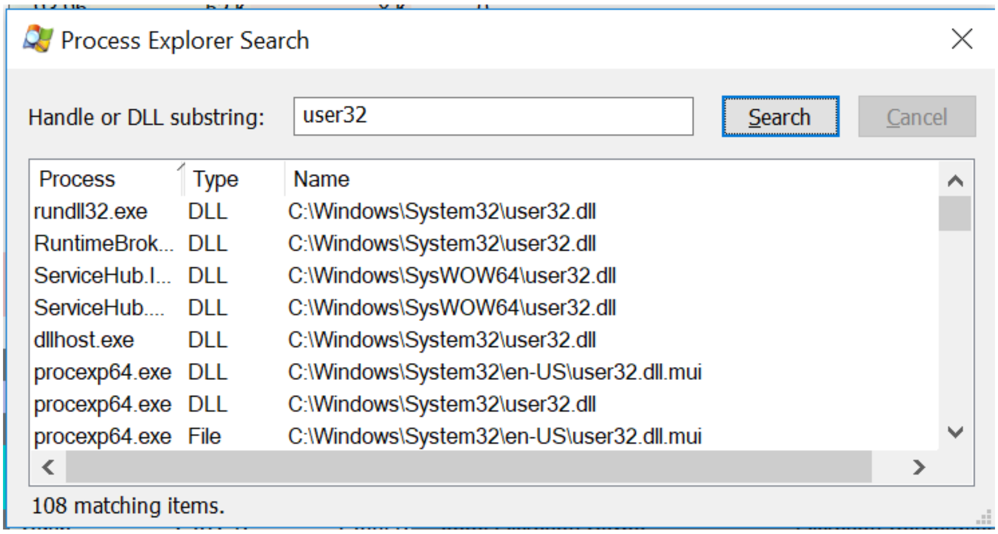
זהו היתרון של DLL. זהו קוד, שמכיל פונקציות שונות. הפונקציות הללו נטענות לזיכרון פעם אחת, וכל התוכנות שרצות במקביל משתמשות באותו עותק שנמצא בזיכרון. לכן, מספיק שיש לנו עותק אחד של ה-DLL על המחשב, שנטען לזיכרון פעם אחת. אם יש לנו עדכונים לתוכנה, נפיץ DLL חדש, כל מי שמשתמש בתוכנה שלנו יחליף את ה-DLL הישן, וכל התוכנות שעושות בו שימוש יקבלו מיד את היכולות החדשות.

הדוגמה שניתנה לפני כן, של קוד שמפעיל את העכבר, לא היתה מנותקת ממה שמתרחש באמת. פונקציות שמנהלות את העכבר, כמו פונקציות נוספות שעובדות מול חלונות פתוחים, נמצאות בקובץ user32.dll. נפתח Process Explorer ונבדוק מי משתמש בקובץ זה.

היכנסו ל-Process Explorer והקליקו על Find



כעת הזינו לשורת החיפוש user32:



כפי שאתם רואים, בזמן כתיבת מסמך זה, נמצאו לא פחות מ-108 התאמות לחיפוש. חלק מהתוצאות הן גרסת ה-64 ביט של ה-DLL, חלקן גרסת ה-32 ביט (כזכור מה שנמצא ב-SysWow64) וחלק הן קובץ mui, שהוא התאמה של התצוגה למשתמש לשפות שונות. בכל מקרה אפשר לראות שכל אחד מהקבצים הללו משרת עשרות Processים ולכן יותר ברורה התועלת בשימוש ב-DLL שנטען פעם אחת לזיכרון.

קיימות שתי דרכים לטעון DLL לתוך תוכנה- Load Time Dynamic Linking ו- Run Time Dynamic Linking.

ב- **Load Time Dynamic Linking** ה-DLL המבוקש נטען עם טעינת התוכנה. למה הכוונה? בניגוד ל- Static Linking, הקוד של ה-exe שנוצר אינו כולל את קוד ה-Library. קוד ה-Library נשאר בקובץ נפרד, מסוג DLL. יכול להיות exe שמשתמש ב-DLL שגודלו 1MB, ועדיין גודלו של ה-exe יהיה מספר KB בודדים.

איך מתבצע ה-Linking בין ה-exe וה-DLL, שהינם שני קבצים נפרדים? לכל קובץ הרצה exe יש חלק, שבו כתובים כל ה-DLLים שצריך לטעון לזיכרון לפני שהמעבד מתחיל להריץ את התוכנה עצמה. לכן השיטה בה הטעינה של ה-DLLים שרשומים בקובץ ה-exe מתרחשת בזמן טעינת ה-exe, נקראת Load Time Dynamic Linking. בזמן הטעינה, ה-DLL ממופה לתוך הזיכרון הוירטואלי של ה-Process שנוצר, כך שניתן יהיה לקרוא לפונקציות שלו. אם יש כבר עותק של ה-DLL המבוקש ב-RAM לא יהיה צורך לטעון אותו שוב, אלא רק לגרום להעביר ל-Process את הכתובות של ה-DLL ושל הפונקציות הנדרשות. בכל מקרה ה-DLL ייטען אל ה-RAM רק פעם אחת, וכל מי שמשתמש בו יעשה שימוש בעותק היחיד הטעון ל-RAM.

ב-**Run Time Dynamic Linking** הרעיון מאד דומה, אך בהבדל אחד מרכזי: אמרנו שקובץ ה-exe כולל את רשימת ה-DLLים שיש למפות לזיכרון בזמן הטעינה. כצפוי, DLL שאינו נמצא ברשימה זו לא ייטען. אבל אפשר שבזמן הריצה ייקראו פונקציות שיטענו בעצמן את ה-DLL הנדרש. השיטה שבה ה-DLL נטען בזמן ריצה נקראת Run Time Dynamic Linking.

בסעיפים הבאים נלמד כיצד יוצרים DLL וכיצד טוענים אותו, בשתי השיטות.

8.3.1 יצירת DLL

נפתח פרוייקט חדש, בדוגמה בחרנו לקרוא לו mydll.

בתור התחלה, נוסיף לפרוייקט שלנו קובץ h חדש (בתוך Solution Explorer קליק ימני על Header Files ואז Add, בחרו New Item ולאחר מכן Header File). נקרא לקובץ mydll.h. נכתוב לתוכו את הקוד הבא:

```
#pragma once

#ifdef DLL_EXPORT
#define DECLDIR __declspec(dllexport)
#else
#define DECLDIR __declspec(dllimport)
#endif

extern "C"
{
    DECLDIR void Share();
    void Keep();
}
```

מומלץ לקרוא על ההנחיות לקומפיילר, זהו ידע חשוב אך הוא מחוץ למוקד של מסמך זה ולכן הוא לקריאה עצמית:

pragma once -

https://docs.microsoft.com/en-us/cpp/preprocessor/once?view=vs-2019

ifdef - https://docs.microsoft.com/en-us/cpp/preprocessor/hash-ifdef-and-hash-ifndef-directives-c-cpp?view=vs-2019

extern “C” -

https://embeddedartistry.com/blog/2017/05/01/mixing-c-and-c-extern-c/

נתמקד בשתי הפונקציות שמוגדרות- Share ו-Keep.

לפני הכל, חשוב להבין שכאשר אנחנו מפיצים DLL הכולל פונקציות שונות, ישנן פונקציות שנרצה שלא ייעשה בהן שימוש על ידי מי שאנחנו מוסרים לו את ה-DLL. אם הקוד שלנו כולל פונקציות בעלות יכולות מיוחדות, שאנחנו רוצים שיישארו רק ברשותנו, נצטרך דרך לסמן אילו פונקציות פתוחות לשימוש של כולם ואילו לא.

לפונקציה Share קודמת ההגדרה DECLDIR, שהפרוש שלה משתנה אם מוגדר DLL_EXPORT או לא. בהנחה שכן, אז הקידומת תפורש כ- __declspec(dllexport) כלומר הפונקציה תיוצא על ידי ה-DLL. המשמעות של זה היא שה-DLL יחשוף את הפונקציה הזו לשימוש של כל מי שמבצע linking ל-DLL.

לעומתה, הפונקציה Keep היא בשום מקרה לא מיוצאת, ולכן הקוד היחיד שיכול להשתמש בה הוא קוד של ה-DLL עצמו.

לאחר שהגדרנו את קובץ ה-header, נעבור לקובץ הראשי- .mydll.cpp

```
#include "pch.h"
#define DLL_EXPORT
#include "mydll.h"

extern "C"
{
    DECLDIR void Share()
    {
        printf("I am an exported function, can be called outside
                the DLL\n");
    }

    void Keep()
    {
        printf("I am not exported, can be called only within the
                DLL\n");
    }
}

BOOL APIENTRY DllMain(HANDLE hModule,        // Handle to DLL
module
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved)     // Reserved
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        // A process is loading the DLL.
        Share();
        Keep();
        break;
```

```
    case DLL_THREAD_ATTACH:
        // A process is creating a new thread.
        break;

    case DLL_THREAD_DETACH:
        // A thread exits normally.
        break;

    case DLL_PROCESS_DETACH:
        // A process unloads the DLL.
        break;
    }
    return TRUE;
}
```

בתחילת הקובץ ניתנות מספר הנחיות לקומפיילר:

1. Include לקובץ pch.h- אותו כבר הכרנו (משמש ל-Include לקבצים שאינם משתנים וכך חוסך זמן קומפילציה)
2. הגדרת DLL_EXPORT, שהיא משמעותית לטובת ההגדרות בקובץ mydll.h ולכן הכרחי שתופיע לפניו. שימו לב לא לשים את ההגדרה לפני ה-Include לקובץ pch.h, איננו יודעים באיזה מקום עלול להיות תנאי שקשור להנחיה זו!
3. Include לקובץ mydll.h

לאחר מכן מוגדרות הפונקציות Share ו-Keep, שמה שהן מבצעות מובן מאליו.

לאחר מכן מגיע החלק המעניין- dllmain. זוהי נקודת הכניסה לתוך ה-DLL שלנו. אם יש קוד שאנחנו רוצים שירוץ אוטומטית, זה המקום להכניס אותו. חשוב להבין: קוד שנמצא ב-dllmain ירוץ גם בלי שהתוכנה הראשית, שטוענת את ה-DLL, תקרא לו. מערכת ההפעלה קוראת ל-dllmain והיא עושה זאת במספר מצבים:

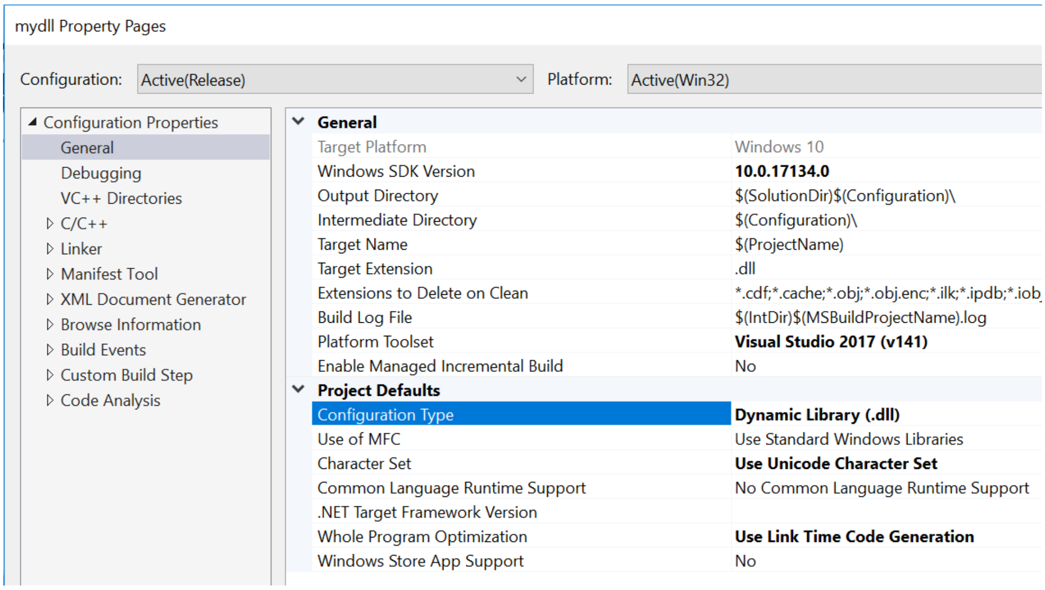
1. Process טוען את ה-DLL שלנו
2. ה-Process שטוען את ה-DLL שלנו יוצר Thread חדש
3. Thread שה-DLL שלנו היה טעון בו, מסיים ריצה
4. Process שטוען את ה-DLL שלנו מבצע Unload, הסרת ה-DLL שלנו

כל אחד מהמקרים האלה מעביר את השליטה ל-dllmain. ההגיון בכך, הוא שאם תוכנה מסויימת טוענת את ה-DLL שלנו, ייתכן ונרצה להריץ פעולות מקדימות כלשהן, שיאפשרו לפונקציות שלנו לרוץ היטב. לדוגמה,

בדיקה מהי החומרה שאנחנו רצים עליה, הקצאת זיכרון וכו'. גם בסיום ריצה או הסרת ה-DLL שלנו ייתכן שיש פעולות שאנחנו צריכים לבצע, לדוגמה שחרור זיכרון, שמירת מידע לקובץ ועוד. חשוב מאד לשים לב שמיקרוסופט מגבילים מאד את הקוד שניתן להריץ אוטומטית מ-dllmain, דברים מסויימים עלולים לגרום לקריסת ה-Process שטען את ה-DLL. מידע נוסף על מה אפשר ואי אפשר לבצע, נמצא בלינק https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-library-best-practices

בקוד הדוגמה שלנו, אנחנו משתמשים רק באפשרות להריץ קוד עם טעינת ה-DLL. מתבצעת קריאה לפונקציה Share ולאחר מכן קריאה לפונקציה Keep, שנועדה להמחיש שהקוד של ה-DLL יכול לקרוא ל-Keep ללא קושי. הפונקציה Keep חסומה רק לקריאות מקוד חיצוני.

נותר לנו רק לייצר את ה-DLL. כדי לעשות זאת, נשנה את הגדרות הקונפיגורציה של הפרוייקט שלנו, כך שבמקום EXE יווצר DLL:



שינוי הגדרות הפרוייקט כך שיווצר DLL

הידד! יש לנו DLL. אם נפתח את תיקיית ה-Release של הפרוייקט mydll, נוכל למצוא שם את הקובץ mydll.dll וכן את הקובץ mydll.lib. מדוע נוצר לנו קובץ lib? הרי אנחנו מבצעים Dynamic Linking, ואמרנו שקבצי lib משמשים עבור Static Linking? התשובה היא, שהסיומת היא אותה סיומת אך השימוש שונה. בניגוד ל-Static Linking, קובץ ה-lib שנוצר לנו אינו כולל קוד, אלא רק אזכור של הפונקציות שה-DLL מייצא. כאשר אנחנו יוצרים קובץ EXE שקורא לפונקציות מתוך DLL, ה-linker משתמש בקובץ ה-lib כדי לשתול בקובץ ההרצה הפניות מתאימות אל ה-DLL, כך שכל פעם שנקרא לפונקציה מתוך ה-DLL תבוצע קפיצה אל הקוד שב-DLL. תהליך זה מאפשר לנו לקרוא לפונקציות שב-DLL ממש כאילו הן פונקציות של הקוד שלנו. מיד נראה כיצד משתמשים בקובץ ה-lib.

8.3.2 יבוא DLL ב-Load Time

ניצור פרוייקט חדש, בדוגמה שלנו usemydll.

נערוך את הקובץ usemydll.cpp באופן הבא:

```
#include "pch.h"

extern "C" __declspec(dllimport) void Share();
// Other option would be-
// #include "mydll.h"

int main()
{
    Share();
    return 0;
}
```

מבין שתי הפונקציות המוגדרות ב-DLL, אפשר לקרוא מהקוד שלנו רק לפונקציה Share. כזכור, הפונקציה Keep אינה מיוצאת. כדי לשלב את הפונקציה Share אנחנו צריכים למסור לקומפיילר מידע על המבנה שלה. את המידע הזה אפשר למסור בשתי דרכים. דרך אחת היא לבצע include לקובץ mydll.h. זוהי הדרך הקלה, אך היא דורשת שהקובץ mydll.h יהיה ברשותנו. לא בטוח שזה יהיה המצב. הדרך השניה היא, כפי שנכתב בקוד שלנו, לתת לקומפיילר את המבנה של Share ולומר לו שזוהי פונקציה מיובאת.

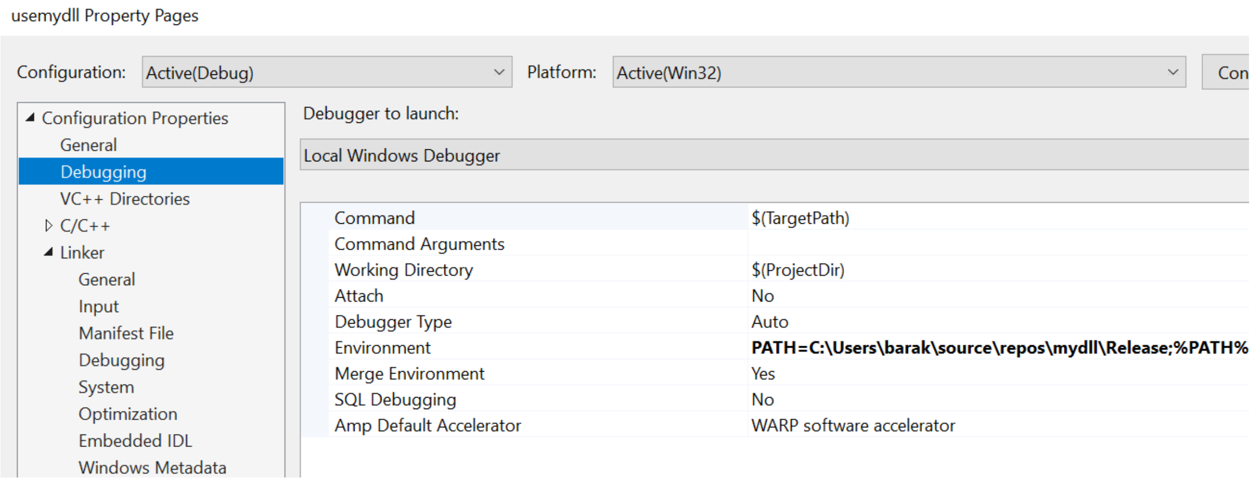
נותר לנו רק לשנות את הגדרות הפרוייקט.

כדי להודיע ללינקר היכן הוא יכול למצוא את mydll.dll, נכניס את התיקיה של mydll.dll לתוך משתנה הסביבה PATH. בתוך Environment נכתוב:

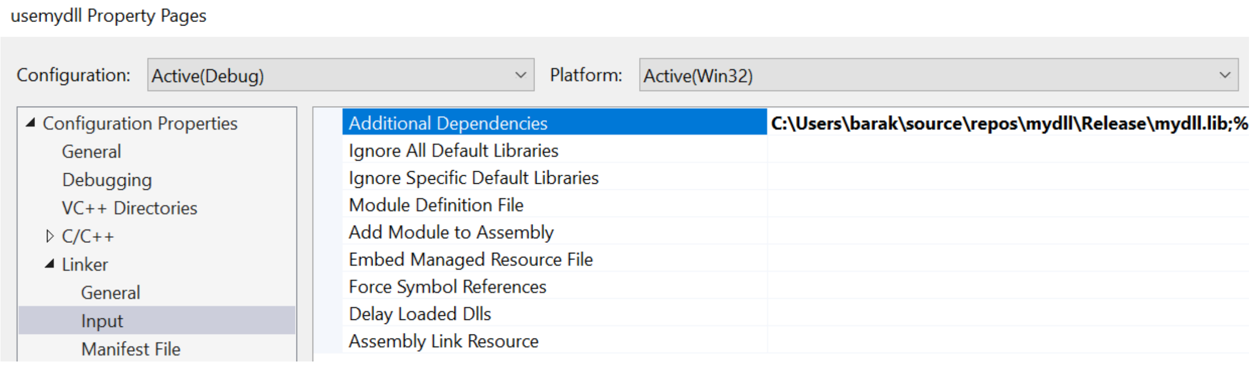
PATH=mydll_path;%PATH%

כאשר mydll_path הוא הנתיב המלא אל ה-DLL שלנו. לדוגמה

PATH=c:\software\dlls;%PATH%



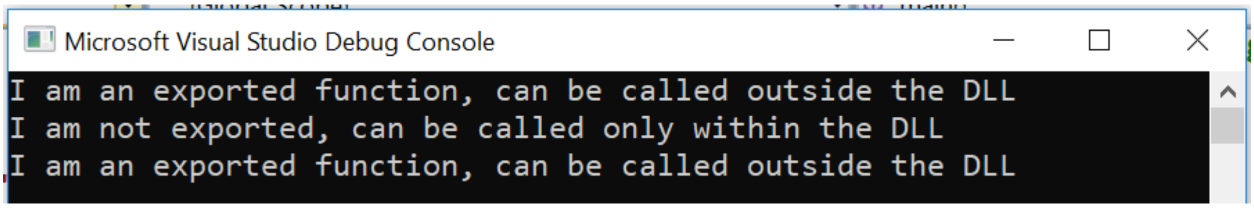
בשלב האחרון אנחנו צריכים לסייע ללינקר למצוא את קובץ ה-lib mydll.lib. נערוך את השדה Additional Dependencies כך שיכלול את שם ה-lib ואת הנתיב המלא אליו:



זהו!

כעת נריץ את התוכנית usemydll. חישבו, מה יודפס למסך? התשובה בעמוד הבא.

בעקבות הרצת usemydll תופיע ההדפסה הבאה:



מה גרם להדפסה הפלט הזה?

שתי השורות הראשונות הן תוצאה של טעינת ה-DLL. כזכור, בתוך dllmain יש קריאה ל-Share ול-Keep. השורה השלישית היא תוצאה של הקריאה ל-Share מתוך הקוד של usemydll.

סיימנו ללמוד על Load Time Dynamic Linking. כעת נוכל ללמוד על ביצוע Run Time Linking.

8.3.3 יבוא DLL ב- Run Time

בניגוד ל-Load Time Dynamic Linking, כעת אנחנו לא מוסרים ללינקר שלנו פרטים על ה-DLL. אילו פרטים העברנו ללינקר ב-Load Time וכעת חסרים? העברנו לו את:

1. מיקום ה-DLL שאנחנו רוצים לטעון
2. קובץ ה-lib שכולל את שמות הפונקציות

כתחליף ל-א', נשתמש בפונקציה LoadLibrary. הפונקציה מקבלת נתיב מלא הכולל שם של DLL, טוענת את ה-DLL לזיכרון ומחזירה מצביע על ה-DLL המבוקש (אם לא אירעה שגיאה).

כתחליף ל-ב', נשתמש בפונקציה GetProcAddress. כפי שאפשר להבין משמה, היא מחזירה את הכתובת בזיכרון בה נמצאת הפרוצדורה שביקשנו (בשפת אסמבלי, פרוצדורה הינה פונקציה).

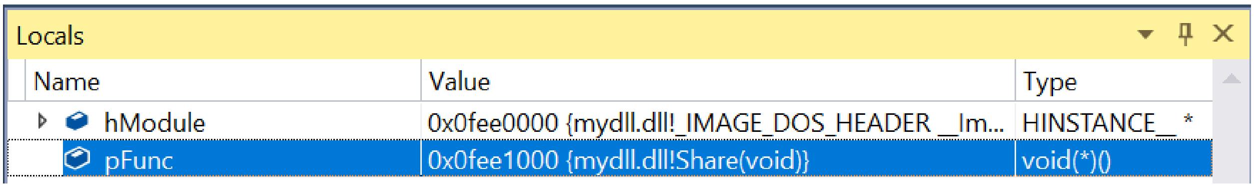
```
#include "pch.h"
#define LIBRARY "c:\\mydll\\Release\\mydll.dll"

typedef void(*PFUNC)(void);

int main()

{
    HMODULE hModule = LoadLibraryA(LIBRARY);
    if (NULL == hModule) {
        printf("Failed to load DLL\n");
        return 0;
```

```
    }
    PFUNC pFunc = (PFUNC)GetProcAddress(hModule, "Share");
    if (NULL != pFunc) {
        (*pFunc)();
    }
    else {
        printf("Failed to load function\n");
    }
    return 0;
}
```

שימו לב לטכניקה שבה אנחנו משתמשים בכתובת החזרה. אנחנו מודיעים לקומפיילר, באמצעות typedef, ש- PFUNC הוא מצביע לפונקציה, שמקבלת void ומחזירה void. זו צורת הכתיבה המקובלת ב-C.

כאשר אנחנו קוראים ל-GetProcAddress, אנחנו מבצעים Casting של ערך החזרה: ערך החזרה הוא כאמור כתובת בזיכרון. ה-Casting מסביר לקומפיילר, שהכתובת הזו היא מצביע לפונקציה, ושהפונקציה הזו מקבלת void ומחזירה void.

הנה, כך נראים המשתנים שהגדרנו, תוך כדי הרצה:



המשתנה hModule מקבל את הכתובת בזיכרון הוירטואלי של ה-Process, כתובת שאליה נטען ה-DLL המבוקש. ה-DLL מתחיל בחלק שנקרא IMAGE_DOS_HEADER ולכן נראה ש-hModule מצביע עליו. המשתנה pFunc מקבל את הכתובת של הפונקציה Share מתוך mydll.dll. ניתן לראות שמיקום הפונקציה הוא כ-0x1000 בתים אחרי ההתחלה של ה-DLL, היכן שאזור הקוד בדרך כלל מתחיל.

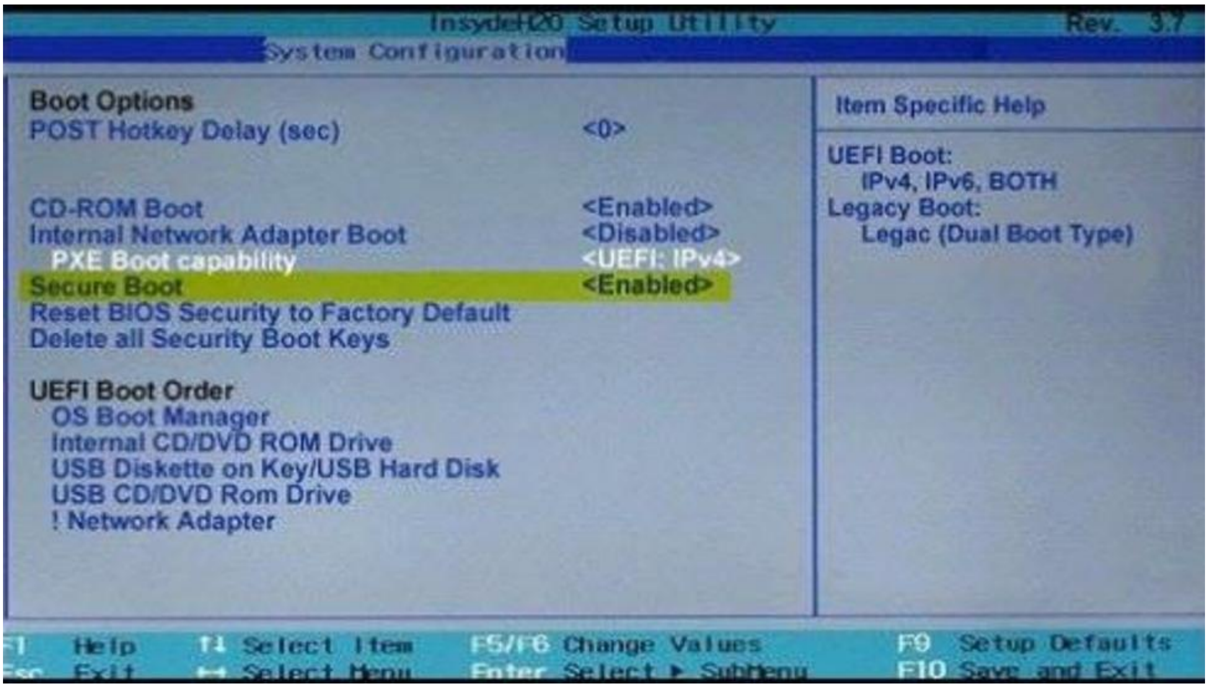
8.4 שימוש ברגיסטרי ל-DLL Injection

ל- DLL Injection יש שימושים שונים. מצד אחד, היכולת פותחת על מנת לדבג קוד של תוכנות: באמצעותה ניתן לגרום לתוכנה להוציא פלט של כל הקבצים שהיא פתחה, הערכים שהיא ניגשה אליהם ברגיסטרי ועוד, ולהיעזר במידע זה כדי לוודא שהיא עובדת כשורה. מצד שני, האפשרות לקחת תוכנה תמימה שאינה מעוררת חשד ולהפוך אותה לנוזקה הפכה את השיטה הזו לפופולרית גם אצל כותבי נוזקות. מן הסתם, אנחנו סומכים עליכם שתשתמשו בידע שתרכשו בתרגיל אך ורק לטובה.

בחלק זה נלמד אודות AppInit וכיצד באמצעות שימוש בשדה זה אפשר לגרום לכך שה-DLL שיצרנו יטען אוטומטית על ידי תוכנות שונות. לאחר שלמדנו ליצור DLL, זהו חלק קל מאד.
פיתחו את Autoruns. אחד מהטאבים שלמעלה הוא AppInit. אנחנו מקווים שתחת הטאב הזה אין לכם כלום על המחשב, אם כן זה חשוד ומומלץ לבדוק ב-Virus Total את ה-DLL שמופיע אצלכם. השדה הזה ברגיסטרי עובד כך: כל תוכנה שנטענת לזיכרון ושמשתמשת ב-user32.dll, בודקת את הערך של AppInit וטוענת כל DLL שנמצא שם. קשה לרדת לעומק דעתה של מיקרוסופט שיצרה את המנגנון הזה, אך הוא בהחלט שם וניתן להשתמש בו. כפי שראינו, יש אוסף לא קטן של Processים שמריצים את user32.dll.

להלן השלבים אותם צריך לבצע:

א. בשלב מסוים מיקרוסופט החליטו לפעול לתיקון בעיית האבטחה וכיום לא ניתן לעשות שימוש ב-user32.dll לביצוע DLL Injection בלי לעשות Disable למנגנון ה-Secure Boot. כדי לבצע זאת, בצעו אתחול מחדש למחשב שלכם, ותוך כדי האתחול לחצו על מקש ה-F2 כדי להכנס למסך ההגדרות של מערכת ההפעלה.



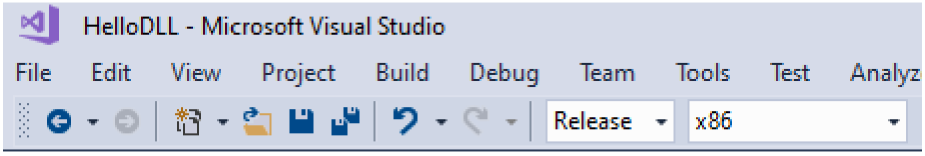
ב. שנו את קוד ה-DLL כך ש-dllmain יפתח חלון משלו ויכתוב הודעה לבחירתכם. לדוגמה:

```
#include "pch.h"
#define DLL_EXPORT
#include "HelloDLL.h"

extern "C"
{
    DECLDIR void Share()
    {
        int msgboxID = MessageBoxA(NULL,
```

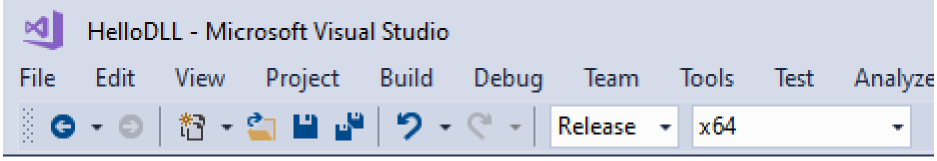
```
            (LPCSTR)"Hacked by BG\n",
            (LPCSTR)"DLL Injection Example",
            MB_DEFBUTTON2);
    }
}

BOOL APIENTRY DllMain(
    HANDLE hModule,      // Handle to DLL module
    DWORD ul_reason_for_call,
    LPVOID lpReserved)      // Reserved
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        // A process is loading the DLL.
        Share();
        break;

    case DLL_THREAD_ATTACH:
        // A process is creating a new thread.
        break;

    case DLL_THREAD_DETACH:
        // A thread exits normally.
        break;

    case DLL_PROCESS_DETACH:
        // A process unloads the DLL.
        break;
    }
    return TRUE;
}
```

ג. קמפלו את הקוד שלכם או עבור גרסת 32 ביט או עבור גרסת 64 ביט. שימו לב- תצטרכו להיות עקביים. אם קמפלתם ל-32 ביט, תצטרכו שמעתה כל השלבים שלכם יתאימו ל-32 ביט, כנ"ל אם קמפלתם ל-64 ביט.

קמפול ל-32 ביט (אין זה משנה אם Debug או Release):



קמפול ל-64 ביט (אין זה משנה אם Debug או Release):



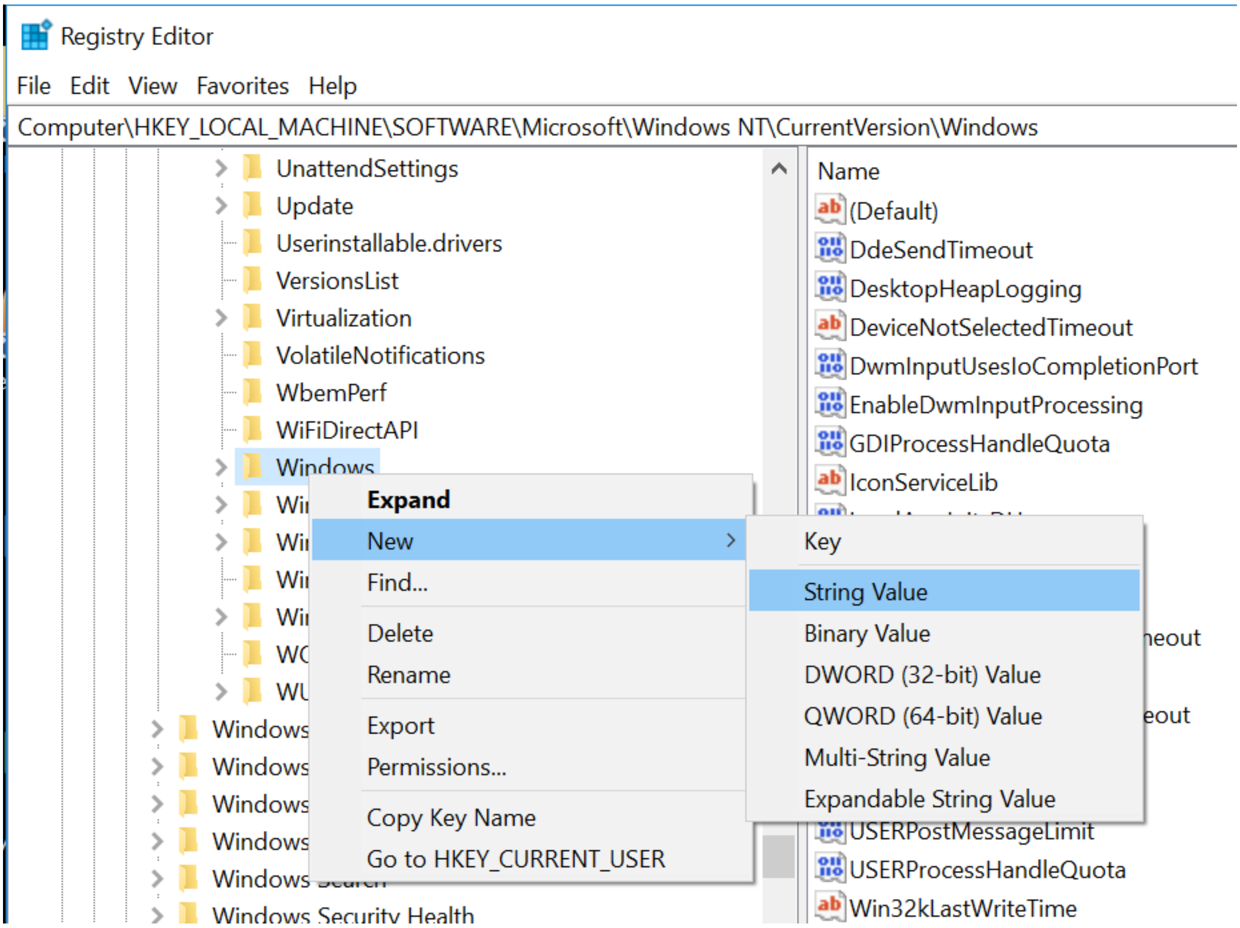
ד. פיתחו את Regedit. אם קמפלתם את הקוד שלכם ל-64 ביט, הכנסו לרגיסטרי במיקום:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows

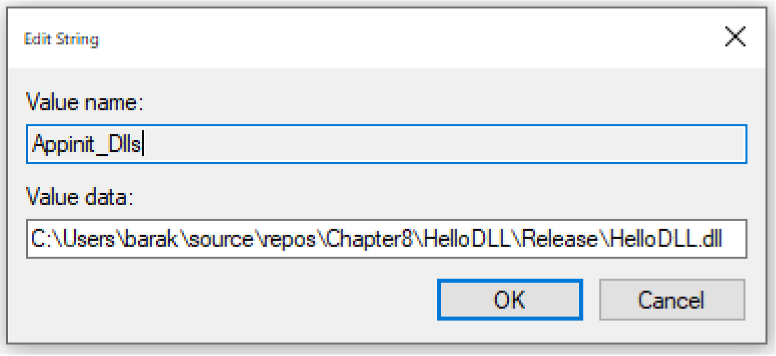
מאידך, אם קמפלתם את הקוד ל-32 ביט, הכנסו לרגיסטרי במיקום

HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows

ה. צרו מפתח בשם AppInit_DLLs. כדי לבצע זאת, לחצו על קליק ימני ובחרו ליצור מפתח רגיסטרי חדש:



השם צריך להיות AppInit_DLLs, והערך שבתוכו צריך להיות ה-DLL שאתם מבקשים להריץ. קריטי לוודא שה-DLL בגרסה הנכונה שלו (32 או 64 ביט) אכן נמצא בתיקיה שהזנתם בתור ערך.



ו. כמעט סיימנו. נותר לשנות שני ערכי רגיסטרי:

1. הערך של LoadAppInit_DLLs צריך להיות 1 (כלומר True)
2. אם קיים אצלכם מפתח בשםRequireSignedAppinit_DLLs הערך שלו צריך להיות 0 (כלומר False), כיוון שה-DLLים שאנחנו מייצרים אינם חתומים על ידי יצרן תוכנה אותנטי

כעת פיתחו חלון של תוכנה כלשה, שרצה או ב-32 או ב-64 ביט בהתאם לסוג ה-DLL שיצרתם (שימו לב שאם יצרתם DLL של 32 ביט והכנסתם אותו לרגיסטרי במיקום של 32 ביט והפעלתם תוכנה שעובדת ב-64 ביט, ולהיפך, זה לא יעבוד. צריך תאימות מלאה בכל השלבים) ותמצאו שהחלון שלכם צץ ☺

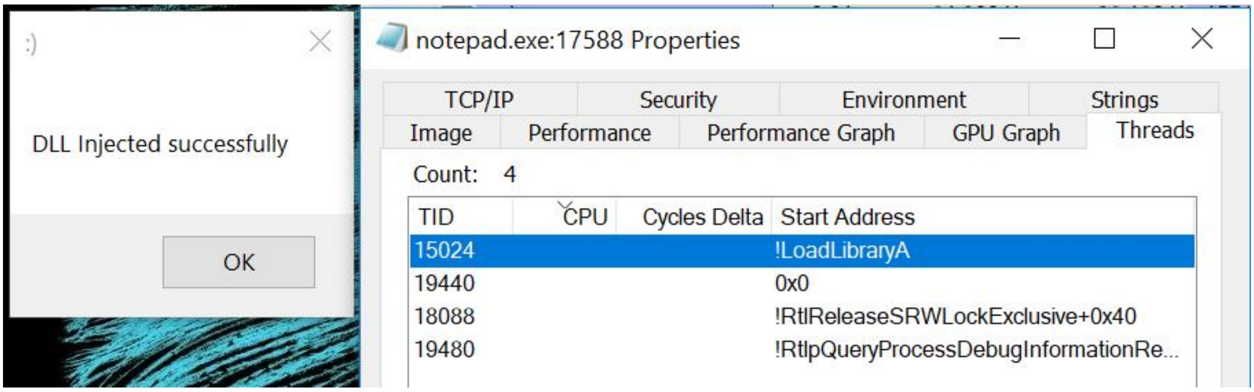
תרגיל 8.1 תרגיל מסכם - ביצוע DLL Injection

להלן הסבר כללי על שיטה נוספת לביצוע DLL Injection:

1. נבחר Process קיים (מרוחק, כלומר לא בן של ה-Process שלנו) ונפתח אותו
2. ב-Process המרוחק, ניצור Thread חדש
3. כזכור כאשר פותחים Thread חדש, מעבירים לו את הפונקציה שממנה יש להתחיל את הריצה. במקרה שלנו, נעביר לו את הכתובת של LoadLibrary, יחד עם הנתיב של ה-DLL שאותו ברצוננו לטעון
4. שלב 3 מעלה בפנינו שני קשיים:
   a. אנחנו צריכים לדעת מה הכתובת של LoadLibrary
   b. אנחנו צריכים שהמחרוזת שמתארת את הנתיב של ה-DLL תהיה שמורה בזיכרון של ה-process המרוחק (העובדה שהמחרוזת שמורה בזיכרון של ה-Process שלנו לא מספיק טובה, מכיוון שאנחנו חולקים זיכרון וירטואלי עם ה-Process המרוחק)
5. כדי לפתור את הקושי הראשון, נבדוק מה הכתובת של LoadLibrary ב-Process שלנו. הכתובת תהיה זהה לכתובת של LoadLibrary ב-Process המרוחק, מכיוון שהפונקציה הזו נמצאת בתוך DLL שמשותף לכל ה-Processים (איזה DLL? מיצאו אותו באמצעות MSDN)
6. כדי לפתור את הקושי השני, נקצה זיכרון ב-Process המרוחק ונשמור לתוכו את המחרוזת עם שם ה-DLL שעלינו להעביר ל-LoadLibrary

שימו לב שה-DLL וקוד התוכנית שמזריקה אותו צריכים להיות באותה ארכיטקטורה (32 / 64 ביט) של התוכנית אליה ה-DLL מוזרק, אחרת הפונקציה CreateRemoteThread תחזור עם קוד שגיאה Access_Denied.

דוגמה לפעולת ההזרקה: במסך ה- Process Explorerאפשר לראות של-notepad.exe יש Thread שנקודת ההתחלה שלו היא LoadLibrary. לצד זה, מופיע חלון שהוקפץ על ידי notepad.exe.



לפניכם שלד של תוכנית שמבצעת DLL Injection לפי השלבים שהוסברו. התוכנית כוללת תיעוד של השלבים לפי סדרם. חלקים מהקוד הושמטו ובמקומם יש שלוש נקודות. השלימו את החסר, וגירמו ל- notepad.exe להריץ חלון נוסף עם הודעה אישית ממכם!

```
// Barak Gonen 2019
// Skeleton code - inject DLL to a running process

#include "pch.h"

int main()
{
    // Get full path of DLL to inject
    DWORD pathLen = GetFullPathNameA(...);

    // Get LoadLibrary function address –
    // the address doesn't change at remote process
    PVOID addrLoadLibrary =
          (PVOID)GetProcAddress(GetModuleHandle(...),
          LoadLibraryA");
```

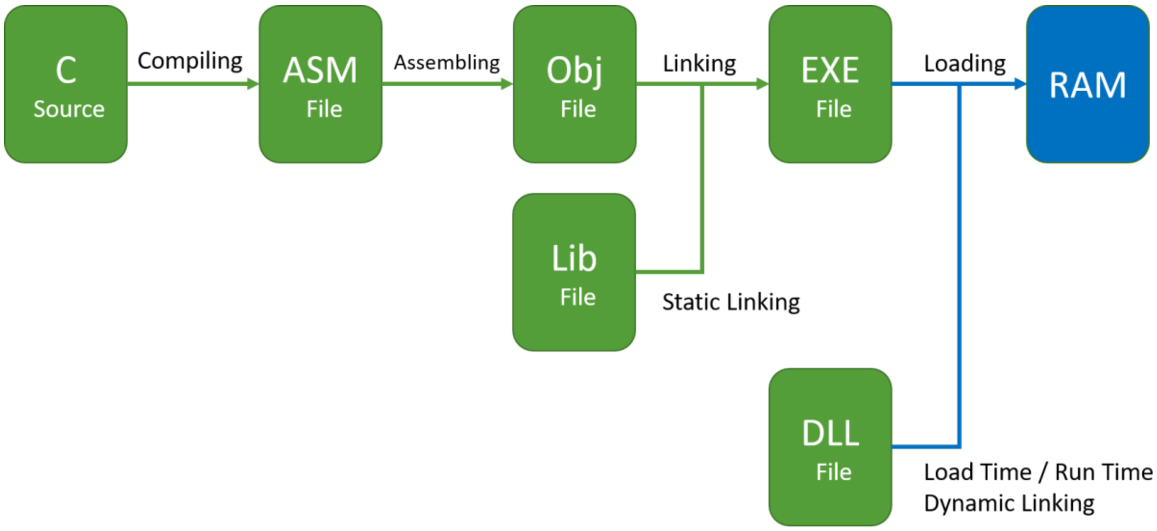
```
    // Open remote process
    proc = OpenProcess(...);

    // Get a pointer to memory location in remote process,
    // big enough to store DLL path
    PVOID memAddr = (PVOID)VirtualAllocEx(...);
    if (NULL == memAddr) {
        err = GetLastError();
        return 0;
    }

    // Write DLL name to remote process memory
    check = WriteProcessMemory(...);
    if (0 == check) {
        err = GetLastError();
        return 0;
    }

    // Open remote thread, while executing LoadLibrary
    // with parameter DLL name, will trigger DLLMain
    HANDLE hRemote = CreateRemoteThread(...);
    if (NULL == hRemote) {
        err = GetLastError();
        return 0;
    }

    WaitForSingleObject(hRemote, INFINITE);
    check = CloseHandle(hRemote);
    return 0;
}
```

8.5 סיכום

סיימנו את הבסיס של DLLים והזרקת DLLים. ישנן עוד שיטות רבות להזריק קוד שלנו לתוך Process קיים, יש לכם את הבסיס לחקור וללמוד אותן בכוחות עצמכם. יחד עם זאת, חשוב שתדעו: ביצוע DLL Injection לתוכנה אשר ישפיע על אדם אחר המשתמש בה בלי שיש הסכמה מפורשת מצדו עלול להיחשב לעבירה על חוק המחשבים. ובלי קשר לחוק, שימו לב שאתם חוקרים בטוב טעם, ולא עושים שום דבר מפוקפק. בהצלחה!

פרק 9 – פורמט PE ותהליך ה-Loading

בפרק הקודם למדנו על תהליך היצירה של קבצי exe ו-DLL מתוך קבצי מקור בשפת C. עברנו על שלבי ה-Compiling, ה-Assembling וה-Linking. הזכרנו באופן כללי שיש שלב נוסף, שלב ה-Loading, שבו הקבצים נטענים לזיכרון ה-RAM. עקב המורכבות של תהליך ה-Loading ראוי להקדיש לו פרק נפרד. האיור הבא ממחיש את השלב אותו נלמד בפרק זה, לעומת שלבים שכיסינו בפרק הקודם.



בפרק זה נעסוק בתפקידי ה-Loader, נלמד כיצד קבצים מאורגנים בפורמט שנקרא PE, קיצור של Portable Executable. זהו פורמט שמאגד את המידע שה-Loader צריך בשביל לבצע את משימותיו. כחלק מתהליך הלימוד נבצע משימה מאתגרת. המשימה היא לשנות את אופן הפעולה של תוכנית, כך שהיא תבצע דברים שונים ממה שהיא תוכננה לבצע במקור. נלמד איך לממש תהליך שנקרא IAT Hooking, כך שהקוד יריץ פונקציות שאנחנו כתבנו במקום הפונקציות שהוא תוכנן להריץ.

9.1 ה-Loader

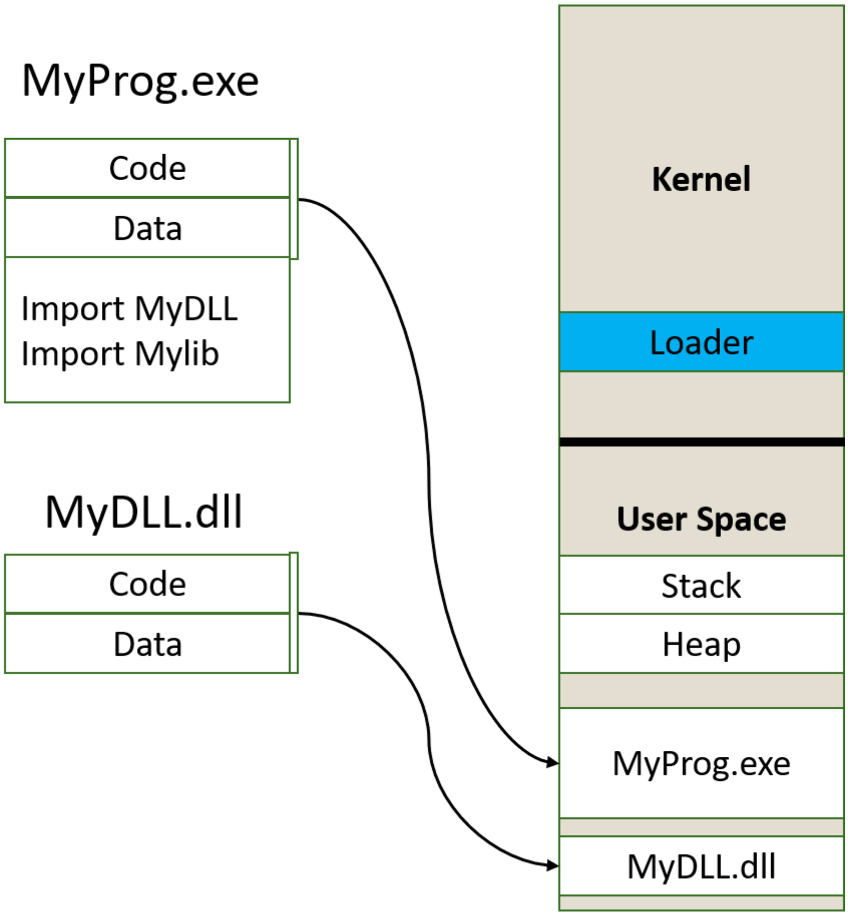
9.1.1 תפקידי ה-Loader

נניח ויצרנו קובץ exe. הקובץ כולל פקודות בשפת מכונה, כתובות בהן הפקודות נמצאות, משתנים, כתובות בהן המתשנים נמצאים, פונקציות והכתובות שלהן. האם המעבד יכול להריץ את הקובץ? חישבו על כך לפני שתמשיכו, מה חסר?

ובכן, הדברים שחסרים הם:

1. טעינה לזיכרון. קובץ ה-exe שמור בהתקן זיכרון כגון דיסק קשיח, תקליטור או USB. המעבד לא יכול להריץ קבצים מהתקנים אלו, מכיוון שכפי שראינו אין לו יכולת לגשת לכתובת ספציפית בזיכרון ההתקנים הללו אלא רק לבלוק שמכיל כתובות רבות. לכן, תפקידו הראשון של ה-Loader הוא לטעון את הקובץ לזיכרון ה-RAM, ומכאן שמו- Loader, טוען. האיור הבא ממחיש את אופן פעולת ה-

Loader. יש ברשותנו קוד שמבצע Static Linking לקובץ MyLib.lib ובנוסף מבצע Dynamic Linking לקובץ MyDLL.dll. הקוד שבוצע לו Static Linking הופך לחלק בלתי נפרד מקובץ ה-exe. את קובץ ה-DLL, לעומת זאת, ה-Loader יטען בנפרד.



המחשת האופן בו ה-*Loader* מבצע טעינה לזיכרון

2. פתרון כתובות (Address Resolution). בזמן ביצוע ה-Linking, לתוך הקוד בשפת מכונה נכנסו כמעט כל הכתובות שצריך לגשת אליהם בזיכרון. אך יש משהו שה-Linker לא ידע: לאילו כתובות ייטענו פונקציות שאנחנו מייבאים מתוך DLLים. היזכרו בכך ש-DLLים נמצאים בזיכרון משותף ויכולים לשמש Processים רבים. כאשר קובץ הרצה נטען לזיכרון, עליו להשתמש בכתובות של פונקציות שכבר נמצאות בזיכרון. ה-Linker לא יכל לחזות מה יהיה מיפוי הזיכרון בזמן שיוחלט להריץ את התוכנית, בין היתר עקב מנגנון רנדומלי בשם ASLR שיורחב עליו בהמשך. לכן ה-Linker משאיר את המלאכה הזו ל-Loader. מיד נראה, שלעיתים ה-Loader לא רק פותר כתובות שה-Linker לא ידע לפתור, אלא גם משנה כתובות שה-Linker פתר.

1. הקצאת זיכרון למשתנים לא מאותחלים. היזכרו בסגמנט ה-BSS, קיצור של Block Started by Sysmbol. נניח שהגדרנו משתנה גלובלי שהינו מערך בגודל מליון בתים, ולא מילאנו את המערך בערכים התחלתיים. אין כל סיבה שבזמן שקובץ ה-exe שלנו שמור

בדיסק, יוקצו בדיסק מליון בתים לטובת שמירת מידע שעדיין אינו קיים. לעומת זאת, בזכרון ה-RAM המערך חייב להופיע ולהיות בגודל הנכון, אין ברירה. ה-Loader מטפל בכך.

2. הפניית המעבד לנקודת תחילת הריצה (Entry Point). מהיכן להתחיל להריץ את הקוד? כל הקוד טעון לזיכרון, כל הכתובות מסודרות ובמקום הנכון. עכשיו, צריך להורות למעבד מהיכן להתחיל את הרצת הקוד. מישהו צריך להעתיק לתוך EIP (רגיסטר הפקודות) את הכתובת של שורת הקוד הראשונה ב-main. ובכלל, שורת הקוד הראשונה שיריץ המעבד לא חייבת להיות בתחילת main. אפשר לכתוב קוד אסמבלי שבו שורת הקוד הראשונה תהיה בכל מקום בקובץ. גם בכך ה-Loader צריך לטפל, כדי שהרצת הקוד תפעל בהצלחה.

לכן, אם נסכם עד כה, תפקידי ה-Loader הם: טעינה לזיכרון, פתרון כתובות, טעינה של DLLים שנטענים ב-Load Time (בניגוד ל-DLLים שנטענים ב-Run Time, כפי שלמדנו בפרק הקודם), הקצאת זיכרון למשתנים לא מאותחלים והפניית המעבד לנקודת תחילת הריצה.

9.1.2 המחשה ל-Address Resolution על ידי ה-Loader

קטע הקוד הבא מבוסס על תכנית שכתבנו בפרק הקודם. הקוד קורא לפונקציה Share, אשר נמצאת ב-dll שיצרנו. נפתח את קובץ ה-exe באמצעות IDA. שימו לב לכתובת של הפונקציה Share כפי שה-Linker קבע:

```
.text:004116FE B9 03 B0 41 00     mov     ecx, offset unk_41B003
.text:00411703 E8 FB FA FF FF     call    sub_411203
.text:00411708 8B F4              mov     esi, esp
.text:0041170A FF 15 E0 A0 41 00  call    ds:Share
```

כלומר זוהי הכתובת 0x0041A0E0 (אם אינכם מבינים מדוע הכתובת "מבולבלת", היזכרו ב-Little Endian). כדי לצפות במרחב הכתובות הוירטואלי של פרוסס לאחר טעינתו ל-RAM אנחנו צריכים לעבוד עם תוכנה שמסוגלת להראות לנו את הזיכרון בזמן ריצה. נשתמש בתוכנה שטרם הכרנו, WinDbg. לא חייבים לדעת להשתמש בתוכנה בשביל להבין את ההסברים הבאים, אבל מי שרוצה – בהמשך יש פירוט של כל השלבים שהתבצעו כדי להגיע להסברים הבאים.

להלן ה-main כפי שנמצא בזיכרון (שימו לב לכך שזו תוצאה של בחירת כתובת בהרצה ספציפית אחת, בהרצות אחרות הכתובות צפויות להשתנות):

```
usemydll!main:
009c16e0 55              push    ebp
009c16e1 8bec            mov     ebp, esp
009c16e3 81ecc0000000    sub     esp, 0C0h
009c16e9 53              push    ebx
009c16ea 56              push    esi
009c16eb 57              push    edi
009c16ec 8dbd40ffffff    lea     edi, [ebp-0C0h]
009c16f2 b930000000      mov     ecx, 30h
009c16f7 b8cccccccc      mov     eax, 0CCCCCCCCh
009c16fc f3ab            rep stos dword ptr es:[edi]
009c16fe b903b09c00      mov     ecx, offset usemydll!_AD1F52AD_usemydll.cpp (009cb003)
009c1703 e8fbfaffff      call    usemydll!ILT+510(__CheckForDebuggerJustMyCode (009c1203)
009c1708 8bf4            mov     esi, esp
009c170a ff15e0a09c00    call    dword ptr [usemydll!_imp__Share (009ca0e0)]
009c1710 3bf4            cmp     esi, esp
009c1712 e8f6faffff      call    usemydll!ILT+520(__RTC_CheckEsp) (009c120d)
009c1717 33c0            xor     eax, eax
009c1719 5f              pop     edi
009c171a 5e              pop     esi
009c171b 5b              pop     ebx
009c171c 81c4c0000000    add     esp, 0C0h
009c1722 3bec            cmp     ebp, esp
009c1724 e8e4faffff      call    usemydll!ILT+520(__RTC_CheckEsp) (009c120d)
009c1729 8be5            mov     esp, ebp
009c172b 5d              pop     ebp
009c172c c3              ret
```

איך נראית עכשיו הקריאה לפונקציה Share? יש קריאה מסוג

dword ptr[009CA0E0]

בשפת אסמבלי המשמעות היא שזהו פשוט מצביע לזיכרון. צריך לגשת לכתובת 0x009CA0E0 ושם נמצא את הכתובת של הפונקציה המבוקשת, Share. אם כך נעשה זאת.

```
Memory
Address: 009ca0e0
00000000`009CA0E0  00 10 F5 77 00 00 00 00 00 00 00 00 00 00 00 00
00000000`009CA0F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000`009CA100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

בכתובת זו נמצאת הכתובת 0x77F51000.

בסופו של התהליך הגענו למסקנה ש-Share נמצאת בכתובת 0x77F51000. מסתבר שאפשר להגיע אל התשובה הזו בדרך פשוטה הרבה יותר, WinDbg מספק לנו את הכתובות של הפונקציות השונות:

```
0:000> x mydll!Share
77f51000          mydll!Share (void)
```

ורואים שזו אכן הכתובת.

אם תחזרו אל הקוד של פונקציית ה-main תשימו לב לכך שכל הכתובות זזו. הכתובת של המשתנה הגלובלי , שמועתק לתוך ecx לפני הקריאה לפונקציה, זזה מ- 0x0041B003 ל- 0x009CB003. אפילו הפקודות זזו: שורת הקוד של ההעתקה הנ"ל עברה מהכתובת 0x004116FE אל הכתובת 0x009C16FE.

בטח שמתם לב לכך שיש תבנית קבועה להזזות הכתובות. נראה שה-Loader החליף את כל הכתובות באותו אופן- במקום להתחיל ב-0x0041, הכתובות מתחילות ב-0x009C. יתר הבתים שמייצגים את הכתובת נותרו זהים.

מה ה-Loader לא שינה? ובכן, יש שני דברים שלא השתנו. הראשון, הפקודות עצמן. כלומר היכן שהיתה פקודה להעתיק כתובת לרגיסטר ECX, לדוגמה, הפקודה נותרה העתקה ל-ECX. היכן שהיתה פקודה להעתיק את הרגיסטר ESP לתוך ESI, הפקודה נותרה ללא שינוי. הדבר השני שלא השתנה היא הפקודה השניה, שמבצעת Call לפונקציה כלשהי. שפת המכונה היתה E8FBFAFFFF ונותרה ללא שינוי. במחשבה ראשונה זה נראה מוזר, מכיוון שאם כל הכתובות זזו אז גם הפונקציה הזו היתה צריכה לעבור כתובת. אולם, הקריאה לפונקציה הזו מתבצעת באמצעות קפיצה *יחסית*. את נושא הקפיצות היחסיות סקרנו במהלך ההסבר על תהליך ה-Linking, כאשר סקרנו כיצד קוד בשפת C מומר לשפת מכונה. בקצרה, הפקודה אומרת למעבד לקרוא לפונקציה שנמצאת כמות מסויימת של בתים מסוף פקודה זו. כיוון שכל הקוד מוזז באותה כמות בתים, הפונקציה נותרה במרחק זהה של בתים מהשורה שקוראת לה, לכן ה-Loader לא היה צריך לבצע תיקון כתובת.

9.1.3 ביצוע Relocation

נכיר מונח חדש- Relocation. במקרה שלנו, לא מדובר על לקחת את המשפחה ולנסוע לחו"ל מטעם העבודה, אלא בתהליך שינוי הכתובות בזיכרון. כל כתובת שה-Linker שם במקום מסויים וה-Loader העביר לכתובת אחרת, עברה Relocation. כפי שראינו, ה-Relocation מבוצע בהזזה קבועה לכל הכתובות.
בהמשך הפרק נבין יותר לעומק את התהליך, אולם העקרון הוא פשוט למדי: ישנה כתובת בסיס, שנקבעת על ידי ה-Linker, ויש טבלה בה ה-Linker כותב את כל הכתובות שצריכות לעבור Relocation אם ה-Loader מחליט לא להשתמש בכתובת הבסיס של ה-Linker אלא לבצע הזזת כתובות.

נסקור דוגמה שהתרחשה לנגד עינינו. כך נראתה פקודת ההעתקה של משתנה גלובלי כלשהו לתוך הרגיסטר ecx, כפי שקבע ה-Linker:

```
.text:004116FE B9 03 B0 41 00      mov     ecx, offset unk_41B003
```

ה-Linker מכניס לטבלת ה-Relocation את הכתובת 0x004116FF (כן, בית אחד לאחר תחילת הפקודה, כיוון ששם מתחיל החלק של הכתובת, שרק אותו צריך לשנות). כל כניסה בטבלה נקראת "Fix Up", כיוון שהיא מצביעה על מקום שצריך "לתקן" בקוד.

ה-Loader החליט לבצע Relocation, במקרה שלנו להזיז אותן בהיסט של 0x005B0000 (זה ההפרש בין 0x00410000 לבין 0x009C0000). המשתנה שנמצא בכתובת 0x0041B003 זז יחד עם כל אזור הזיכרון של ה-data. הבעיה היא שה-Loader צריך גם לתקן את כל הפניות אל המשתנה שיש בקוד. הוא נכנס

לטבלה ורואה את ה-Fix Up, שבכתובת 0x004116FF יש פניה למשתנה. כעת ה-Loader יודע איפה לבצע את התיקון.

הנה, כך נראה הקוד לאחר התיקון, כפי שמצאנו אותו בזמן הריצה, לאחר שה-Loader שינה אותו:

```
009c16fe b903b09c00      mov     ecx, offset usemydll!_AD1F52AD_usemydll.cpp (009cb003)
```

כפי שרואים, הכתובת של המשתנה עברה תיקון והיא שונתה בזמן הטעינה מהדיסק ל-RAM.

9.1.4 הרחבה- שימוש בסיסי ב-WinDbg

התוכנה WinDbg דורשת הדרכה, שהינה מחוץ למיקוד של ספר זה. אך לטובת מי שמכירים את התוכנה או שמוכנים למאמץ נוסף כדי לראות בעצמם את הכתובות בזיכרון, הצעדים הבאים הם מה שנדרש כדי לשחזר את הדוגמאות שבצילומי המסך:

1. הורידו WinDbg Preview מאתר מיקרוסופט. שימו לב- נדרש Windows 10.
2. העתיקו את ה-DLL אל התיקיה שבה נמצא קובץ ההרצה (לחילופין אפשר להכניס את הנתיב של קובץ ה-DLL אל משתנה הסביבה PATH, אולם זו השקעה מיותרת בשביל קובץ שנמצא בשימוש רק פעם אחת)
3. באמצעות WinDbg בצעו טעינה לקובץ ההרצה שטוען את ה-DLL
4. הכניסו נקודת עצירה בתחילת ריצת התוכנית, באמצעות כתיבת bp $exentry בשורת הפקודה
5. בשורה הפקודה כיתבו g, התוכנית תעצר בנקודת העצירה
6. במסך הפקודה של WinDbg כיתבו את הפקודה הבאה x dllname!main כאשר אתם מחליפים את dllname בשם שבו קראתם ל-dll. בעקבות זאת תודפס הכתובת של פונקציית main.
7. מבין הטאבים שבראש המסך, בחרו View ואז Disassembly
8. יופיע לפניכם קוד האסמבלי של התוכנית, החל מנקודת העצירה בה התוכנית נמצאת כרגע. בשדה הכתובת שלמעלה, הזינו את הכתובת של main שמצאתם
9. מבין הטאבים שבראש המסך, בחרו View ואז Memory
10. הזינו לתוך חלון ה-Memory את הכתובת שנמצאת בתוך המצביע לזיכרון, ש-main קורא לו כדי להפעיל את הפונקציה שב-DLL.
11. במסך הפקודה של WinDbg כיתבו את הפקודה הבאה x dllname!funcname כאשר אתם מחליפים את dllname בשם שבו קראתם ל-DLL ואת funcname בשם של הפונקציה שאתם מחפשים, לדוגמה Share

9.2 פורמט PE

כדי שה-Loader יוכל לבצע את עבודתו, הוא צריך לקבל את כל המידע שנחוץ לו והמידע צריך להיות מאורגן באופן מסודר. צריך לדעת היכן נמצא הקוד, היכן הנתונים, היכן יש כתובות שצריך לבצע להן Relocation, היכן יש פונקציות שצריך לייבא, היכן נקודת ההתחלה של הריצה ועוד. במילים אחרות, צריך פורמט לקבצים מסוג קבצי הרצה וקבצי DLL . הפורמט הזה נקרא במערכת Windows פורמט PE, קיצור של Portable Executable. בפורמט PE נעשה שימוש גם בסוגים אחרים של קבצים שאינם מעניינו כעת, כגון קבצי SCR שומרי מסך, קבצי SYS, OCX, CPL ועוד. במערכת לינוקס יש פורמט אחר, הקרוי ELF, קיצור של Executable and Linkable Format.

לפני שנצלול פנימה לתוך פורמט PE, צריך להדגיש שזהו פורמט מורכב הכולל מגוון עצום של פרטים. אנחנו נתמקד בפרטים החשובים ביותר, ואף על פי כן הדיון צפוי להיות ארוך. מצד שני, הבנת הפורמט תאפשר לנו לבצע דברים מעניינים ולא שגרתיים.

לפני שתתקדמו, הורידו את תוכנת CFF Explorer מהקישור הבא:

https://ntcore.com/files/ExplorerSuite.exe

אגב, כלים מוצלחים נוספים ניתן למצוא כאן:

https://blog.malwarebytes.com/threat-analysis/2014/05/five-pe-analysis-tools-worth-looking-at/

9.2.1 אזור ה-DOS Header

הקטע הראשון בקובץ PE נקרא DOS Header. הסיבה שקוראים לו כך תתברר מיד.

פיתחו קובץ הרצה exe כלשהו באמצעות CFF Explorer. אתם יכולים לראות שה-DOS Header כולל מגוון שדות. בפועל, יש רק 2 שדות חשובים:

- שדה ה-Magic, השדה הראשון
- שדה ה-lfanew, השדה האחרון

<u>**9.2.2.1 שדה ה-**Magic</u>

שדה זה חייב להכיל את הקוד "5A4D", שמיתרגם בקוד ASCII לאותיות "MZ". זוהי מחווה למי שהמציא את פורמט PE, ששמו Mark Zbikowski.

תרגיל 9.1 שדה ה-Magic

פיתחו קובץ exe או DLL כלשהו באמצעות CFF Explorer. גשו למסך Hex Editor ושנו את MZ למשהו אחר. שימרו את הקובץ (בתפריט יש אופציה Save As). נסו להריץ את הקובץ (אם זה קובץ הרצה) או לבצע לו Loading כפי שלמדנו לעשות לקבצי DLL. מה קרה?

<u>**9.2.2.2 שדה ה-**lfanew</u>

שדה זה כולל את המיקום של ה-Header הבא, ה-NT Header.

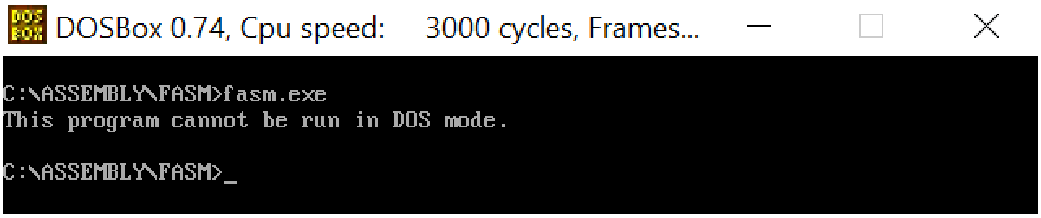
תרגיל 9.2 שדה ה-lfanew

באמצעות CFF Explorer, בידקו מהו הערך שנמצא ב-lfanew. הכנסו ל-NT Header ובידקו היכן הוא מתחיל (ההתחלה נמצאת בשדה שנקרא Signature), וודאו שהמיקום הוא אותו מיקום.

לאחר שביצעתם את התרגיל, ראוי להדגיש שהערך שמצאתם יכול להיות שונה, אפשר לערוך אותו ועדיין הפורמט יהיה תקין. זאת בניגוד לערך MZ, שאם נשנה אותו הקובץ לא יהיה שמיש. מדוע מישהו ירצה להזיז את מיקום ה-NT Header? ובכן, התשובה טמונה בדרך שבה פועלות תוכנות אנטי וירוס. תוכנות אלו סורקות קבצים במחשב ומחפשות קבצים זדוניים. כדי לעשות זאת הן מנתחות קבצי הרצה, אך בלי להריץ אותם כמובן. לכותבי נוזקות יש ערך רב במציאת דרכים ליצור קבצי הרצה שמצד אחד יהיו תקינים לפי הגדרות הפורמט, אך מצד שני יכשילו את האנטי וירוסים באמצעות סידורים מורכבים ולא סטנדרטיים של קובצי ההרצה.

9.2.2.3 DOS Stub

זהו שדה היסטורי שכבר אינו חשוב, ונעסוק בו בעיקר מכיוון שהוא ממש מגניב. ובכן, מדוע קוראים ל- DOS Header כך? כפי שבטח שמתם לב, כאשר פותחים קובץ PE באמצעות עורך HEX, אפשר לקרוא בתחילת הקובץ את הטקסט: "This program cannot be run in DOS mode". מיקרוסופט הציגה את פורמט PE בשנת 1993, באותה תקופה היה עדיין צורך לתמוך לאחור במערכת ההפעלה DOS. לכן בחלק זה יש קוד שמורץ אם מערכת ההפעלה היא DOS. כדי לראות את הקוד בפעולה, יש צורך להתקין אמולטור של DOS שרץ ב-16 ביט Real Mode. אם ננסה להריץ תוכנה שקומפלה ל-Windows, נקבל את ההודעה הבאה:



אם אתם חובבי אסמבלי Real Mode כמוני, תוכלו לעקוב אחרי הפקודות, חלקן מודגשות כאן:

```
00000040 | 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 | ..º..´.Í!¸.LÍ!Th
00000050 | 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F | is.program.canno
00000060 | 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 | t.be.run.in.DOS.
00000070 | 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 | mode....$.......
```

אפשר לזהות פעמיים את הרצף CD 21, שהוא הפקודה int 21. בפעם הראשונה, אחרי B4 09, שהיא הפקודה-

mov ah, 9

קוד זה אומר להדפיס מחרוזת למסך. בפעם השניה, אחרי B8 01 4C, שהיא הפקודה-

mov ax, 4C01

זהו קוד שאומר לסיים את ריצת התוכנית.

תרגיל 9.3 תרגיל אתגר לחובבי אסמבלי - DOS Stub

עירכו קובץ exe כלשהו, כך שהרצה במצב Real Mode תבצע דבר מה שונה. התוכנית תבקש בתור קלט סיסמה מהמשתמש, ותשתמש בה על מנת לבצע XOR להמשך הנתונים השמורים בקובץ ה-exe. בעקבות ביצוע ה-XOR, ייחשף קוד חוקי ב-Real Mode אשר המעבד יריץ.

9.2.2 אזור ה-NT Header

ה-NT Header מכיל 3 דברים:

1. שדה בשם Signature. שדה זה חייב להיות 50 45, שהוא קוד ה-ASCII של "PE".
2. מבנה נתונים בשם File Header
3. מבנה נתונים בשם Optional Header

מעבר לכך אין מה להרחיב על ה-NT Header, המידע המעניין נמצא בכל אחד ממבני הנתונים שהוא כולל בתוכו

9.2.3 אזור ה-File Header

ה-File Header הוא מבנה נתונים שנראה כך:

Member	Offset	Size	Value	Meaning
Machine	00000104	Word	014C	Intel 386
NumberOfSections	00000106	Word	0005	
TimeDateStamp	00000108	Dword	5D443FFC	
PointerToSymbolTable	0000010C	Dword	00000000	
NumberOfSymbols	00000110	Dword	00000000	
SizeOfOptionalHeader	00000114	Word	00E0	
Characteristics	00000116	Word	0102	Click here

ה-Offset שכתוב ליד שם של כל שדה, הוא המקום בקובץ ה-exe שבו נמצא השדה הנ"ל. לדוגמה, שדה ה-TimeDateStamp נמצא 0x108 בתים מתחילת קובץ ה-exe. נדגיש, כי באופן כללי שדה ה-Offset מתייחס למיקום על הדיסק הקשיח, לא הכתובת בזיכרון ה-RAM שאליו נטען הקובץ.

אפשר לראות שבין ה-Offsetים יש הפרשים של מספר בתים. לדוגמה שדה ה-Machine נמצא 0x104 בתים מתחילת הקובץ, גודלו כפי שנראה מיד הוא 2 בתים, ולכן השדה הבא יתחיל ב-Offset של 0x106.

מבין השדות ב-File Header נעבור על אלו שמובלטים בכחול.

9.2.3.1 Machine

שדה זה מציין על איזה סוג ארכיטקטורה של מעבד אמור לרוץ הקובץ. השדה יכול להכיל אחד משני ערכים: הערך 014C מציין מעבד 32 ביט. בהמשך נראה שזה אינו השדה היחיד שבו מוגדר סוג המעבד.

תרגיל 9.4 שדה ה-Machine

באמצעות CFF Explorer, שנו את הערך כך שיתאים למעבד 64 ביט (הקליקו על Meaning ובחרו בערך המתאים ל-AMD64. כן, זה ערך ברירת המחדל של תוכניות 64 ביט שמקומפלות ב-Visual Studio).

9.2.3.2 NumberOfSections

שדה זה, כשמו כן הוא. מציין את מספר ה-Sections בקוד.

תרגיל 9.5 שדה ה-NumberOfSections

השוו את מספר ה-Sections שנמצא בשדה זה, למספר ה-Sections שנמצאים תחת הטאב "Section Headers".

9.2.3.3 TimeDateStamp

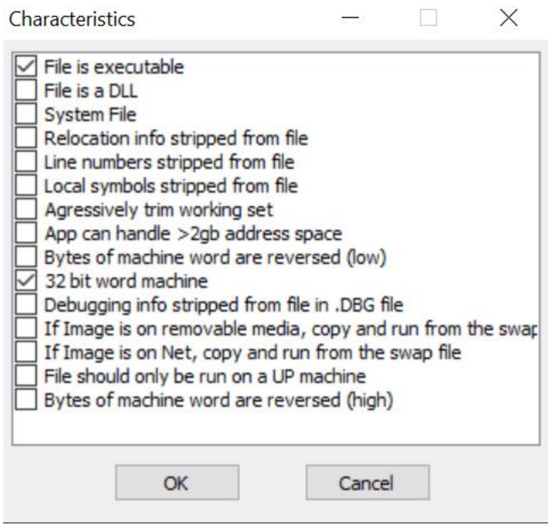
שדה זה מציין את הזמן בו נוצר הקובץ על ידי הקומפיילר, לפי שיטת הספירה של UNIX. בשיטה זו, סופרים שניות החל מהשעה 00:00:00 בתאריך ה-1 לינואר 1970, לפי שעון UTC (זמן לונדון, ללא התחשבות בשעון קיץ). האם תוכלו למצוא בפורמט PE מקום נוסף שיש בו שדה של זמן?

תרגיל 9.6 מכונת הזמן

אנחנו עדיין לא יודעים לבנות מכונת זמן שתאפשר לנו לשנות את ההיסטוריה, אבל כן נוכל לשנות את מועד היצירה של קבצי ה-PE שלנו. שנו את הזמן בקובץ שיצרתם, כך שהוא יצביע על 5 בנובמבר 1995 בשעה 21:44. חפשו בגוגל TimeDateStamp Converter.

9.2.3.4 Characteristics

הקליקו על "Click Here" בשורה של Characteristics. תקבלו טבלה כזו:



השדות שנעסוק בהם הם:

- File is executable- ללא הסימון לא ניתן להריץ את הקובץ
- File is a DLL- יש בסך הכל ביט אחד שמבדיל בין קובץ exe ו-DLL!
- Relocation info stripped from file- כזכור, ה-Loader עשוי לתקן כתובות בזיכרון, אם הוא אינו משתמש בכתובות שה-Linker יצר. לטובת התיקון, יש צורך בטבלה שאומרת אילו כתובות צריך לתקן. אם שדה זה מסומן, הטבלה הזו לא קיימת בקובץ ולכן ה-Loader לא יוכל לבצע תיקון כתובות. המשמעות המעשית היא שה-Loader יטען לזיכרון בדיוק באותן כתובות שה-Linker קבע.
- 32 bit word machine- שוב, שדה שקובע אם הקובץ הוא 32 ביט או 64.

9.2.4 אזור ה-Optional Header

השם Optional הוא שם מטעה ביותר, מכיוון שהוא ממש לא אופציונלי. השם נקבע להיות כך בזמן שפורמט הקבצים תוכנן להיות מותאם לסוגים שונים של מערכות הפעלה, ולכן הנתונים של Windows היו אופציונליים עבור מערכות הפעלה אחרות. אבל עבור Windows, המידע שנמצא שם הוא הכרחי להרצה.

להלן השדות של ה-Optional Header של 32 ביט, אותם שדות קיימים גם ב-64 ביט אך הגדלים של חלקם שונים כמובן. מסומנים השדות בהם נעסוק.

הערה: השדה האחרון של ה-Optional Header הוא Data Directories, אשר ב-CFF Explorer מופיע בתור טאב נפרד. נתייחס אליו בקרוב, לאחר שנעבור על כל השדות המשמעותיים.

Member	Offset	Size	Value	Meaning
Magic	00000118	Word	010B	PE32
MajorLinkerVersion	0000011A	Byte	0E	
MinorLinkerVersion	0000011B	Byte	0F	
SizeOfCode	0000011C	Dword	00000E00	
SizeOfInitializedData	00000120	Dword	00001400	
SizeOfUninitializedData	00000124	Dword	00000000	
AddressOfEntryPoint	00000128	Dword	00001338	.text
BaseOfCode	0000012C	Dword	00001000	
BaseOfData	00000130	Dword	00002000	
ImageBase	00000134	Dword	00400000	
SectionAlignment	00000138	Dword	00001000	
FileAlignment	0000013C	Dword	00000200	
MajorOperatingSystemVers...	00000140	Word	0006	
MinorOperatingSystemVers...	00000142	Word	0000	
MajorImageVersion	00000144	Word	0000	
MinorImageVersion	00000146	Word	0000	
MajorSubsystemVersion	00000148	Word	0006	
MinorSubsystemVersion	0000014A	Word	0000	
Win32VersionValue	0000014C	Dword	00000000	
SizeOfImage	00000150	Dword	00006000	
SizeOfHeaders	00000154	Dword	00000400	
CheckSum	00000158	Dword	00000000	
Subsystem	0000015C	Word	0003	Windows Console
DllCharacteristics	0000015E	Word	8140	Click here
SizeOfStackReserve	00000160	Dword	00100000	
SizeOfStackCommit	00000164	Dword	00001000	
SizeOfHeapReserve	00000168	Dword	00100000	
SizeOfHeapCommit	0000016C	Dword	00001000	
LoaderFlags	00000170	Dword	00000000	
NumberOfRvaAndSizes	00000174	Dword	00000010	

9.2.4.1 Magic

קבצי הרצה שמיועדים למעבדים של 32 דורשים טיפול שונה מה-Loader לעומת קבצי הרצה שמיועדים למעבדי 64 ביט. לדוגמה, הכתובת שבה המעבד צריך להתחיל את הריצה: בקובץ שמיועד ל-32 ביט הכתובת תהיה שמורה בגודל של 32 ביט ואילו בקובץ של 64 ביט צריך כמובן לבטא את הכתובת כ-64 ביט. לכן, ה-Optional Header של 64 ביט צריך להיות שונה בכמה מקומות מאשר בגרסה של 32 ביט.

כדי שה-Loader יידע איך לקרוא את הנתונים, האם בשדות בגודל 32 או 64 ביט, הוא צריך לדעת אם ה-Optional Header הוא 32 או 64 ביט. השדה שקובע זאת הוא שדה ה-Magic. עבור 32 ביט הערך יהיה 10B, כמו בתמונה.

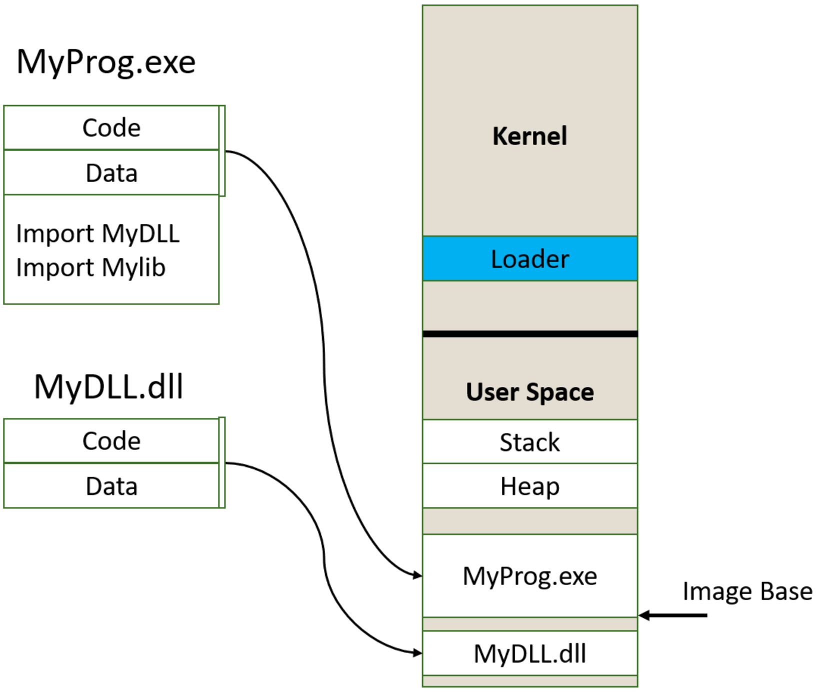
תרגיל 9.7 תרגיל Optional Header Magic

דרך CFF Explorer, גלו מהו ערך שדה ה-Magic עבור קבצי 64 ביט. ניתן לעשות זאת באמצעות הקלקה על שדה ה-"Meaning" ובחירת האפשרות המתאימה.

9.2.4.2 Image Base

אנחנו נדלג לרגע על שדה ה-Address of Entry Point ונתחיל בהסבר על Image Base, מסיבות שתיכף יתבררו.

כזכור, בזמן הרצת הקובץ, ה-Loader יעתיק את המידע מקובץ ההרצה אל מרחב הכתובות הוירטואלי של ה-Process שיווצר. במרחב הכתובות הזה נמצא הקוד של מערכת ההפעלה, של DLLים שנטענים, של זיכרון Stack ו-Heap שמוקצה, וגם כמובן לקובץ ה-exe יוקצה אזור זיכרון, החל בכתובת וירטואלית מסוימת. שדה ה-Image Base אומר מה ה-Linker מצפה שתהיה הכתובת הוירטואלית הזו.



המחשה של שדה ה-*Image Base*

חישבו: למה המידע הזה חשוב? הרי ה-Loader יכול לטעון את הקובץ לזיכרון בכל כתובת וירטואלית, לא רק היכן שה-Linker מצפה לה?

התשובה היא שכפי שראינו ה-Linker מסדר את הכתובות בקובץ ה-exe בהתאם למיקום שבו הוא מבקש שה-Loader יטען אותם לזיכרון. אם ה-Loader רוצה לשנות את כתובת הטעינה לזיכרון, הוא צריך לתקן את הכתובות הללו, ולשם כך הוא צריך לדעת מה הניח ה-Linker. זהו תהליך ה-Relocations שסקרנו לפני כן. שדה ה-Image Base מאפשר ל-Loader להבין בכמה הוא צריך "להזיז" כתובות, כיוון שבלי לדעת מה כתובת הבסיס שקבע ה-Linker לא ניתן לדעת מה גודל ההזזה שצריך לבצע.

לדוגמה: ה-Linker קבע למשתנה מסויים כתובת של 0x00403000. ה-Linker הכניס ל-Image Base ערך של 0x00400000. כעת ה-Loader מבצע טעינה לזיכרון לא בכתובת שה-Linker ביקש, אלא לדוגמה בכתובת 0x00500000. לאיזו כתובת ה-Loader צריך לטעון את המשתנה?

ה-Loader יבצע חיסור של הכתובות, ויגלה כי המשתנה נמצא 0x3000 בתים מתחילת ה-Image Base. את ההיסט הזה הוא יוסיף לכתובת אליה הוא באמת טוען את הקובץ, ולכן המשתנה יהיה בכתובת וירטואלית 0x00503000.

9.2.4.3 Address of Entry Point

שדה משמעותי זה קובע, כפי ששמו אומר, באיזו כתובת במרחב הכתובות הוירטואלי של ה-Process תתחיל ריצת המעבד ברגע שה-Process יווצר. נסביר איך להבין את הערך שרשום בשדה זה. להלן דוגמה:

Member	Offset	Size	Value	Meaning
AddressOfEntryPoint	00000128	Dword	00001338	.text

את שדה ה-Offset הכרנו כבר, הוא מתייחס למקום מתחילת הקובץ שבו נמצא הערך שאנחנו מחפשים.
הערך עצמו הוא- במקרה זה – 0x1338. זהו ערך יחסי מה-Image Base אותו סקרנו זה עתה.
כמעט קלענו למספר יפה, אתם מוזמנים ליצור לעצמכם קובץ PE שבו ערך השדה הוא 1337 :)

https://www.urbandictionary.com/define.php?term=1337

לדוגמה, נניח כי ה-Image Base הוא 0x00400000 וכי ה-Loader לא מבצע שינוי אלא טוען לכתובת זו. במקרה זה, אם ערך ה-Address Of Entry Point הוא כפי שמופיע בדוגמה, אז ריצת הקוד תחל מכתובת 0x00401338 בזיכרון הוירטואלי.

נסיים בהסבר של מושג חדש: **RVA**. אלו ראשי תיבות של Relative Virtual Address. הכוונה לכתובת יחסית מתחילת ה-Image Base. במילים אחרות, בדוגמה שראינו 0x1338 הוא ה-RVA של מיקום תחילת ריצת הקוד. מעתה נשתמש במושג זה.

תרגיל 9.8 תרגיל Entry Point

הורידו את הקובץ https://data.cyber.org.il/os/HiBye.exe. הריצו אותו. הקובץ מדפיס hi ולאחר מכן bye. כעת, באמצעות שינוי ה-Entry Point בלבד, גירמו לקובץ להריץ רק bye. תוכלו להעזר ב-IDA כמובן, כדי לדעת לאיזה ערך לשנות את ה-Entry Point.

9.2.4.4 Section Alignment, File Alignment

כאשר דברנו על מיפוי קבצים לזיכרון נתקלנו כבר בכך שהמיפוי אינו מתבצע סתם כך לכל כתובת בזיכרון, אלא לכתובות "עגולות". ראינו גם, שהזיכרון הוירטואלי מחולק ל-Pages, שגודלם בדרך כלל 4096 בתים (0x1000). גם המידע שאגור בהתקני אחסון כמו דיסק קשיח לא מפוזר סתם כך בכל כתובת, אלא מחולק לאזורי זיכרון שנקראים Sectors, ושגודלם הוא בדרך כלל 512 בתים (0x200).

המונח הטכני המדויק לכתובת "עגולה" הוא "Aligned". כאשר זיכרון הוא "Aligned" למספר מסוים, הכוונה היא שכתובת ההתחלה שלו היא כפולה של המספר. לדוגמה, זיכרון שהוא Aligned לגודל של Page, 0x10000, חייב להתחיל בכתובת שמסתיימת ב-000, כלומר 12 ביטים של אפסים.

לכן אין זו הפתעה שגם בתהליך שבו קובץ ה- exe נטען לזיכרון, יש כתובות Aligned גם על הדיסק הקשיח וגם ב-RAM. שני הסוגים הללו של Alignment מוגדרים בקובץ ה-PE:

- שדה ה-File Alignment שמגדיר את ה-Alignment עבור אזורי הזיכרון על הדיסק הקשיח. בדרך כלל שדה זה יהיה 0x200, בדומה לגודל ההיסטורי של Sector בדיסק (כיום גודל Sector צפוי להיות שונה, לעיתים כ-4KB, אך פורמט PE עושה לעיתים שימוש בגודל ההיסטורי).
- שדה ה-Section Alignment, שמגדיר את ה-Alignment ב-RAM. בדרך כלל שדה זה יהיה 0x1000, בדומה לגודל הנפוץ של Page בזיכרון הוירטואלי.

עיברו לטאב של Section Headers ותוכלו לצפות בארגון אזורי הזיכרון הן על הקובץ בדיסק והן מה שצפוי להיות כאשר הם ייטענו ל-RAM בדוגמה זו אנחנו משתמשים בקובץ HiBye.exe אותו הורדתם:

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.data	0000000B	00001000	00000200	00000200
.text	00000140	00002000	00000200	00000400
.idata	000000D0	00003000	00000200	00000600

כדי לא להקדים את המאוחר, נתמקד כרגע רק בשתי עמודות:

- עמודת ה-Raw Address היא הכתובת בה מתחיל ה-Section בקובץ, על הדיסק. כפי שרואים, כל הכתובות מתחלקות יפה ב-0x200.
- עמודת ה-Virtual Address כוללת את ה-RVA (אנחנו משתמשים במונח החדש שלמדנו) של כל Section ב-RAM. כפי שרואים, כולם מתחלקים יפה ב-0x1000.

כמובן, כאשר אנחנו מתכנתים אף אחד לא מצפה שנכתוב קוד שגודלו בדיוק כפולה של 0x200 או 0x1000.
אם כך, מה קורה בקובץ ה-PE? נמצא את התשובה לבד.
הקליקו על ה-Text Section, בתחתית מסך ה-CFF Explorer תוכלו לראות את הבתים בקובץ על הדיסק, כפי שהם מאורגנים בכתובות יחסיות לתחילת ה-Section. גללו עד לסוף ה-Section, ותוכלו לראות מה שמור ברווח שבין סוף הקוד לסוף ה-Section:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	B9	00	10	00	00	E2	FE	BE	00	10	40	00	E8	B9	00	00	¹.▯..âþ¾.▯@.è¹..
00000010	00	BE	05	10	40	00	E8	AF	00	00	00	6A	00	FF	15	68	.¾▯▯@.è¯...j.ÿ▯h
00000020	30	40	00	55	89	E5	83	EC	14	53	51	52	8D	7D	EC	B9	0@.U▌å▌ì▯SQR }ì¹
00000030	0C	00	00	00	E8	9E	00	00	00	83	C7	0C	4F	C6	07	00	▌...è▌...▌Ç▌OÆ▯.
00000040	6A	10	6A	00	8D	4D	EC	51	FF	15	B4	30	40	00	83	C4	j▯j. MìQÿ▯´0@.▌Ä
00000050	0C	5A	59	5B	83	C4	14	5D	C3	60	EB	05	25	78	0A	0D	▌ZY[▌Ä▯]Ã`ë▯%x..
00000060	00	50	68	5C	20	40	00	FF	15	B0	30	40	00	83	C4	08	.Ph\ @.ÿ▯°0@.▌Ä▯
00000070	61	C3	60	EB	06	25	78	00	0D	0A	00	B9	20	00	00	00	aÃ`ë▯%x....¹....
00000080	D1	C0	89	C2	83	E2	01	51	50	52	68	75	20	40	00	FF	ÑÀ▌Â▌â▯QPRhu @.ÿ
00000090	15	B0	30	40	00	83	C4	08	58	59	E2	E4	68	78	20	40	▯°0@.▌Ä▯XYâähx @
000000A0	00	FF	15	B0	30	40	00	83	C4	04	61	C3	60	EB	0B	3D	.ÿ▯°0@.▌Ä▯aÃ`ë▯=
000000B0	3D	3D	3D	3D	3D	3D	3D	0A	0D	00	68	AF	20	40	00	FF	=======...h¯ @.ÿ
000000C0	15	B0	30	40	00	83	C4	04	61	C3	60	56	FF	15	B0	30	▯°0@.▌Ä▯aÃ`Vÿ▯°0
000000D0	40	00	83	C4	04	61	C3	55	89	E5	83	EC	14	60	89	4D	@.▌Ä▯aÃU▌å▌ì▯`▌M
000000E0	F8	89	4D	F4	89	7D	F0	6A	F6	FF	15	6C	30	40	00	89	ø▌Mô▌}ðjöÿ▯l0@.▌
000000F0	C3	83	7D	F8	00	74	2A	6A	00	8D	4D	FC	51	6A	01	8B	Ã▌}ø.t*j. MüQj▯▌
00000100	4D	F0	51	53	FF	15	70	30	40	00	8B	7D	F0	FF	45	F0	MðQSÿ▯p0@.▌}ðÿEð
00000110	FF	4D	F4	80	3F	0D	74	06	83	7D	F4	00	75	D9	C6	07	ÿMô▌?.t▯▌}ô.uÙÆ▯
00000120	00	6A	00	8D	4D	FC	51	6A	01	8D	4D	EC	51	53	FF	15	.j. MüQj▯ MìQSÿ▯
00000130	70	30	40	00	80	7D	EC	0A	75	E7	61	83	C4	14	5D	C3	p0@.▌}ì.uça▌Ä▯]Ã
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

במקרה הספציפי של הקובץ הנ"ל, גודל ה-Text Section על הדיסק הוא 0x200. אפשר לראות שהחל מכתובת מסויימת ועד הכתובת 0x1FF, הקובץ מלא באפסים.

תרגיל 9.9 תרגיל Padding

פיתחו באמצעות CFF Explorer את הקובץ chrome.exe מהלינק הבא https://data.cyber.org.il/os/chrome.exe (כדי להבטיח שאתם והספר משתמשים באותה גרסה של chrome.exe). אתרו את ה-Text Section, ומיצאו את ה"ריפוד" שיש בסופו. במקרה הזה, אלו אינם אפסים. חפשו את משמעות הקוד בגוגל. חפשו "x86 opcode" יחד עם הקוד שמצאתם.

9.2.4.5 Size Of Image

שדה זה קובע ל-Loader כמה מקום יש להקצות ב-RAM עבור התוכנית כולה. המקום שיש להקצות הוא סכום המקום שצריך להיות מוקצה עבור כל אחד מה-Sections.

נבדוק זאת בעצמנו. בחזרה ל-HiBye.exe, פיתחו את טאב ה-Data Sections ובידקו כמה זיכרון צריך להיות מוקצה עבור התוכנית?

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.data	0000000B	00001000	00000200	00000200
.text	00000140	00002000	00000200	00000400
.idata	000000D0	00003000	00000200	00000600

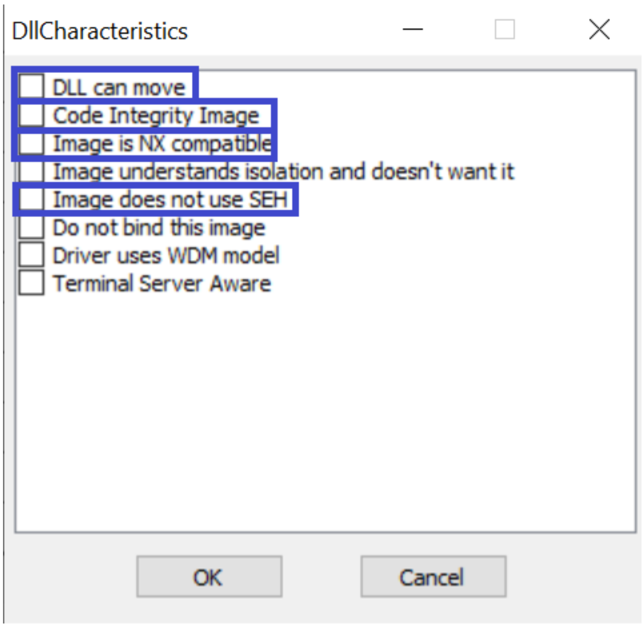
ה-Section האחרון, idata, הוא כפי שראינו בעבר אזור הזיכרון שכולל את כל הקוד שמייבאים ממקורות אחרים, כגון DLLים. אזור הזיכרון מתחיל בכתובת 0x3000 וגודלו הוא 0x1000 (נזכור שגם אם הגודל קטן יותר, מתבצע ריפוד באפסים). לכן יש להקצות 0x4000 בתים עבור התוכנית. ואכן, זה ערכו של שדה Size Of Image:

SizeOfImage	000000D0	Dword	00004000

9.2.4.6 DLL Characteristics

למרות שמו, המאפיינים שנמצאים בשדה זה אינם קשורים רק ל-DLLים אלא גם לקבצי exe. באופן כללי,מדובר באוסף של מאפיינים הקשורים לאבטחה.

להלן אוסף השדות מתוך CFF Explorer. שימו לב לכך ששמות השדות אינם זהים לשמות שמוגדרים באפיון של קבצי PE, לכן לכל שדה מובא בסוגריים השם ה"אמיתי" שלו. מודגשים השדות שנלמד עליהם:



- שדה DLL can move (שם מדויק: Dynamic Base). משמעות: בזמן ה-Loading, אפשר לבצע שינויי כתובות. למעשה, מדובר במנגנון ASLR, עליו נפרט בהסבר נפרד.
- שדה Code integrity image (שם מדויק: Force Integrity). שדה זה מאלץ לבדוק אם החתימה הדיגיטלית של הקובץ תקפה. נסביר בקיצור רב, הסבר מלא הוא מעבר למיקוד של ספר זה: כפי

שראינו, אפשר לערוך קבצי PE באופן ידני. יכולת זו מאפשרת לכותבי תוכנות זדוניות לשתול קוד בתוך תוכנות "טובות" ובעקבות זאת נוצר צורך במנגנון כלשהו שיבדוק אם קובץ PE הוא מקורי או שהקובץ עבר שינוי שיכול לגרום לו להיות זדוני. מנגנון האימות הנ"ל נקרא "חתימה דיגיטלית". בשדה שאנחנו דנים בו מוגדר האם צריך לבדוק שהחתימה הדיגיטלית תקפה לפני הרצה. כמובן שגם את השדה הזה ניתן לשנות ידנית (ולהפוך את הבדיקה ללא הכרחית), אבל החתימה הדיגיטלית מבוססת בין היתר על הערך בשדה הזה ולכן שינוי של השדה יגרום לתוכנות אבטחה להתערב לפני הרצת הקובץ.

לקריאה נוספת:

http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/authenticode_pe.docx

https://docs.microsoft.com/en-us/sysinternals/downloads/sigcheck

- Image is NX compatible (שם מדויק: NX Compat). במעבדים יש דגל שנקרא NX (קיצור של No-Execute), כאשר הדגל דולק לא תתאפשר הרצת פקודות מאזורי זיכרון שאינם קוד. כלומר לא יהיה אפשרי להריץ קוד מהמחסנית, מ-data וכו'. כאשר השדה הזה מופעל בקובץ ה-PE, מערכת ההפעלה מדליקה ביט ב-PTE, שמסמן למעבד שאסור לו להריץ קוד מאזור זיכרון ללא הרשאות מתאימות.
- Image does not use SEH (שם מדויק: NO SEH). ה-SEH הוא קיצור של Structured Exception Handling. בקצרה, כאשר הקוד ביצע משהו חריג הריצה שלו תיעצר. זאת בניגוד למצב שבו הקוד קורא לקוד שתפקידו לטפל בשגיאה, מה שעלול לאפשר ניצול לרעה על ידי נוזקות.

9.2.5 אזור ה-Data Directories

מבנה הנתונים שנקרא Data Directories הוא מערך של מצביעים. אפשר להתייחס אליו כאל מפה, שכתוב בה איפה אפשר למצוא נתונים שימושיים. דוגמאות שימושיות במיוחד הן:

- Imports: פונקציות שהקוד שלנו מייבא
- Exports: פונקציות שהקוד שלנו מייצא (היזכרו בכתיבת DLL, ניתן היה להגדיר אילו פונקציות יהיו זמינות למי שמייבא את ה-DLL)
- Resources: לדוגמה אייקונים שהתוכנה עושה בהם שימוש
- Relocations: רשימת כל הכתובות בקוד שצריך לתקן אם הקוד לא נטען ל-Base Address שה-Linker הניח

מבנה הנתונים כולל 16 איברים, כל איבר נמצא במקום מוגדר מראש. לדוגמה, המידע על Imports תמיד יהיה באינדקס 0 בטבלה והמידע על Relocations תמיד יהיה באינדקס 5 בטבלה. כמובן שאם הנתון אינו קיים, לדוגמה אין טבלת Relocations, אז חלק מהנתונים הם ריקים (אפסים).

חישבו: במקרה שאין טבלת Relocations, מה יהיה הערך “DLL can move” ב-DLL Characteristics?

תשובה: כמובן, שאם אין מידע על Relocations, לא ניתן לבצע שינויי כתובות ולכן ה-DLL (או ה-exe) חייב להטען ב- Base Address שלו.

עבור כל נתון, רשומים הפרטים הבאים:

- ה-RVA שלו
- הגודל שלו

שימו לב לכך שיכולים להיות נתונים שונים בתוך אותו Section. פיתחו באמצעות CFF Explorer את הקובץ Chrome.exe. כפי שתוכלו לראות, ה-Data Directories שלו די מלאה. הן הנתונים על Resource Directory והן ה- Security Directory נמצאים באותו ה- Section. במקרה זה rsrc.

9.2.6 אזור ה-Section Headers

מבנה נתונים זה מרכז את כל המידע לגבי ה-Sections השונים. אין שום מצביע שאומר מה המקום שלו בקובץ ה-PE, אבל קל להגיע אליו: הוא נמצא מיד אחרי ה-NT Header. המיקום של ה-NT Header נמצא בתוך ה-DOS Header, ואילו הגודל של ה-NT Header הוא קבוע, לכן למעשה מיקום ה- Section Headers נמצא ב-

DOS_Header->lfanew + sizeof(NT Header)

להלן דוגמה, של ה-Sections הראשונים של Chrome.exe:

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00117184	00001000	00117200	00000400	00000000	00000000	0000	0000	60000020
.rdata	00031B1C	00119000	00031C00	00117600	00000000	00000000	0000	0000	40000040
.data	0000B118	0014B000	00003800	00149200	00000000	00000000	0000	0000	C0000040

נעבור על השדות המסומנים:

- Virtual Size: הגודל שה-Section תופס בזיכרון הוירטואלי. המספרים כאן אינם מעוגלים ל- Section Alignment, אך בסופו של דבר המקום שנתפס בזיכרון הוא כמובן כפולה של ה- Alignment.
- Virtual Address: ה-RVA של ה-Section, כלומר מה המרחק של ה-Section מכתובת הבסיס שאליו ה-PE נטען. ה-RVA תמיד יהיה כפולה של ה-Section Alignment.
- Raw Size: הגודל שה- Section תופס על הדיסק, כשהוא כבר מעוגל ל-File Alignment
- Raw Address: באיזו כתובת בקובץ ה-PE (כלומר על הדיסק) מתחיל ה-Section.
- Characteristics: אוסף של דגלים שמפרטים את ההגדרות של ה- Section. לדוגמה אם הוא כולל קוד, אם יש לו הרשאות קריאה, כתיבה, הרצה.

תרגיל 9.10 תרגיל Section Headers

התרגילים מתייחסים ל-`chrome.exe` שבקישור https://data.cyber.org.il/os/chrome.exe:

1. באיזו כתובת בקובץ ה-PE מסתיים ה-Text Section?
2. באיזו כתובת ב-RAM מסתיים ה-Text Section?
3. איך יכול להיות שהגודל של ה-Text Section על הדיסק (0x117200) גדול יותר מהגודל שלו בזיכרון הוירטואלי (0x117184)? האם "נעלמו" פקודות?
4. איך יכול להיות שהגודל של ה-data Section על הדיסק (0x3800) הוא קטן יותר מאשר הגודל שלו בזיכרון הוירטואלי? האם "נוספו" נתונים?

תשובות:

1. כיוון שמדובר בכתובת בקובץ, אנחנו מסתכלים על ה-Raw. ה-Text Section מתחיל בכתובת 0x400, וגודלו 0x117200. כלומר הוא מסתיים בית אחד לפני הכתובת 0x117600. שימו לב לכך שזו כתובת ההתחלה של ה- rdata Section.
2. כיוון שמדובר במקום ב-RAM, אנחנו מסתכלים על הכתובות הוירטואליות. ה-Text Section נטען לזיכרון ב-RVA של 0x1000 (אי אפשר לדעת מה הכתובת המדוייקת בשלב זה, רק הכתובת היחסית). גודלו הוא 0x117184, ולכן הוא מסתיים לכאורה בכתובת 0x118184. אבל, עקב ה-Section Alignment שהוא 0x1000, יתבצע עיגול ולכן ה-Text Section יתפוס את כל המקום עד 0x119000 (לא כולל). שימו לב לכך שזו כתובת ההתחלה של ה-rdata Section בזיכרון הוירטואלי.
3. הגודל שמצוין על הדיסק מעוגל ל-File Alignment, זה אינו הגודל האמיתי של סך כל הפקודות. לכן הוא יותר גדול מאשר סך הפקודות שאכן מועתקות לזיכרון הוירטואלי.
4. התשובה נמצאת ב-BSS (היזכרו בהגדרה שלו, מידע לא מאותחל שלא נשמר על הדיסק כדי לחסוך מקום). כאשר מתבצעת הקצאת זיכרון וירטואלי, יש צורך להקצות מקום גם למשתנים שהיו ב-BSS ושעל הדיסק לא תפסו מקום. כלומר הגודל הוירטואלי של ה-data Section כולל גם את הגודל של כל המשתנים ב-BSS, לכן הוא גדול יותר מאשר ה-Raw שלו.

9.2.7 אזור ה-Import Directory

כפי שראינו עד כה, הכתובות שאליהן נטענים DLLים לא יכולות להיות ידועות בזמן תהליך ה-Linking, ואפילו אם הן היו ידועות, עקב מנגנון ה-ASLR הן עשויות להשתנות בכל פעם שמערכת ההפעלה עולה עם הדלקת המחשב. לכן בכל מקרה שבו תוכנה עושה שימוש בפונקציות מתוך DLLים, ה-Loader יצטרך למצוא את הכתובות של הפונקציות ולתקן את הקוד בזמן הטעינה לזיכרון, כך שהפניות יהיו לכתובות הנכונות. ה-Import Directory כולל את כל המידע שה-Loader צריך כדי למצוא את הכתובות של הפונקציות הנדרשות.

כפי שמיד ניווכח, המידע לגבי ה-Importים מאורגן בצורה מורכבת למדי, אבל בסופו של דבר העקרון הוא פשוט: טבלה שרשומים בה כל ה-DLLים שנעשה בהם שימוש, ובכל DLL יש רשימה של כל הפונקציות. בזמן ה-Loading, לצד כל פונקציה נוספת הכתובת שלה בזיכרון. הטבלה הנ"ל נקראת Import Address Table, או בקיצור IAT, והיא משמשת בזמן הריצה לקבוע בדיוק לאיזו כתובת צריך לבצע call כאשר קוראים לפונקציה מתוך DLL.

למרות המורכבות של הפרטים, ההשקעה משתלמת. הבנה של ה-IAT תאפשר לנו לבצע תהליך שנקרא IAT Hooking, שבאמצעותו נשנה את אופן הביצוע של תוכנית תוך כדי ריצה.

נתחיל במשימה. השלב הראשון יהיה למצוא את ה-Import Directory בתוך קובץ ה-PE. שימו לב לכך ש-CFF Explorer נוטה להציג את השדות השונים במיקום שלהם ב-RVA, ולכן אם אנחנו רוצים לראות את מבנה הנתונים בקובץ ה-PE אנחנו צריכים לבצע מעט המרות כתובות. טענו את הקובץ Chrome.exe ובצעו את השלבים הבאים יחד עם ההסבר.

1. מיצאו את ה-RVA של ה-Import Directory

לפי ה-Data Directory, ה-RVA של ה- Import Directory הוא x13F7FD0

Member	Offset	Size	Value	Section
Import Directory RVA	00000108	Dword	0013F7FD	.rdata

2. מיצאו את הכתובת של ה-Import Directory בתוך קובץ ה-PE

כפי שכתוב, ה-Import Directory נמצא בתוך ה- rdata section. נפנה אל ה-Section Headers:

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.rdata	00031B1C	00119000	00031C00	00117600

אפשר לראות שתחילת ה-section הוא 0x117600 בתים מתחילת קובץ ה-PE (מה שנקרא " Raw Address") ושה-Section נטען ל-RAM החל מכתובת וירטואלית 0x119000 (נזכור שזהו מיקום יחסי למיקום שבו ה-Loader בחר לטעון את התוכנה, לדוגמה אם התוכנה נטענה לכתובת 0x00400000, אז ה- rdata Section נטען למקום 0x00519000 בזיכרון הוירטואלי).

אנחנו רוצים להמיר את ה-RVA שערכו 0x0013F7FD לכתובת Raw, בקובץ. הטבלה שלפנינו מראה לנו שכתובת ה-RVA 0x00119000 היא כתובת 0x00117600 בקובץ. אם נחסר אותן נמצא כי ההפרש ביניהן הוא 0x1A00. כלומר, כדי להמיר RVA כלשהו שנמצא בתוך rdata לכתובת בקובץ, צריך להחסיר ממנו 0x1A00. נבצע זאת:

0x13F7FD – 0x1A00 = 0x13DDFD

מעולה, סיימנו שלב נוסף. כעת אנחנו יודעים היכן לחפש את ה- Import Directory בתוך הקובץ.

3. מיצאו את ה-Import Directory בקובץ

ה-CFF Explorer מכיל בתוכו Hex Editor. ניגש אליו, ונקליק על החץ שכתוב עליו "Go To Offset". נזין את הכתובת שחישבנו. האזור המודגש ברקע שחור הוא ה-Import Directory

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0013DDF0	62	6F	78	65	64	50	72	6F	63	65	73	73	00	50	F8	13
0013DE00	00	00	00	00	00	00	00	00	00	10	17	14	00	30	FF	13
0013DE10	00	70	F8	13	00	00	00	00	00	00	00	00	00	1F	17	14
0013DE20	00	50	FF	13	00	10	FF	13	00	00	00	00	00	00	00	00
0013DE30	00	2C	17	14	00	F0	05	14	00	00	00	00	00	00	00	00
0013DE40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

כעת נפענח את המידע שיש בתוכו.

כל רשומה ב-Import Directory מורכבת מ-5 שדות בגודל DWORD כל אחד. מיד נסקור אותם. הרשומות כתובות אחת אחרי השניה, בלי הפרדה. הדרך לדעת שהגענו לסוף ה-Import Directory היא רק כאשר הגענו לרשומה שמתחילה ב-DWORD שכולו אפסים. האיור הבא ממחיש שב-Import Directory של Chrome.exe יש 3 רשומות, ובכל רשומה 5 שדות:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0013DDF0	62	6F	78	65	64	50	72	6F	63	65	73	73	00	50	F8	13
0013DE00	00	00	00	00	00	00	00	00	00	10	17	14	00	30	FF	13
0013DE10	00	70	F8	13	00	00	00	00	00	00	00	00	00	1F	17	14
0013DE20	00	50	FF	13	00	10	FF	13	00	00	00	00	00	00	00	00
0013DE30	00	2C	17	14	00	F0	05	14	00	00	00	00	00	00	00	00
0013DE40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

נפענח את הרשומה הראשונה. הרשומה כוללת את הערכים הבאים:

1. 0x0013F850
2. 0x00000000
3. 0x00000000
4. 0x00141710
5. 0x0013FF30

כיוון שההסבר כולל פרטים רבים, אנחנו נתמקד בשדות 1,2 ו-5. שדה 3 נקרא Time/Date Stamp ושדה 4 נקרא Forwarder Chain . נותיר אותם ללימוד עצמי. מידע נוסף ניתן למצוא כאן:

https://malwology.com/2018/10/05/exploring-the-pe-file-format-via-imports/

שלושת השדות שנתמקד בהם כוללים כתובות RVA. כדי להבין על מה הם מצביעים, נרצה למצוא את הכתובות שלהם בקובץ. לכן, נחסיר מכל אחד מהם את ההיסט המתאים לקובץ הספציפי שלנו, 0x1A00.

1. 0x0013F850 – 0x1A00 = 0x0013DE50
2. 0x00000000
3. 0x00000000
4. 0x00141710 – 0x1A00 = 0x0013FD10

5. 0x0013FF30 – 0x1A00 = 0x0013E530

נתחיל דווקא מהשדה הרביעי. שדה זה נקרא "Name" וכפי שאנחנו רואים הוא מצביע על הכתובת 0x0013FD10 בתוך הקובץ, כתובת שכאמור תהפוך להיות RVA 0x00141710 ברגע שהתוכנית תיטען ל-RAM. חפשו את הכתובת 0x0013FD10 ב- Hex Editor:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
0013FD10	63	68	72	6F	6D	65	5F	65	6C	66	2E	64	6C	6C	00	4B	chrome_elf.dll.K
0013FD20	45	52	4E	45	4C	33	32	2E	64	6C	6C	00	56	45	52	53	ERNEL32.dll.VERS
0013FD30	49	4F	4E	2E	64	6C	6C	00	00	00	00	00	00	00	00	00	ION.dll.........

כפי שרואים, זהו מצביע לשם של ה-DLL. אנחנו רואים שלושה שמות של DLLים, שמופרדים ביניהם ב-00. כעת ברור למה שדה זה נקרא "Name"- הוא מצביע על השם של ה-DLL שנעשה בו שימוש.

אך שם ה-DLL אינו מספיק, אנחנו צריכים לדעת גם אילו פונקציות בדיוק צריך לייבא מתוך ה-DLL. לכן נחקור את השדה הראשון. שדה זה נקרא "Original First Thunk", שם לא מהמוצלחים ביותר. למעשה, באמצעות השדה הזה פשוט מגיעים לרשימה של השמות של הפונקציות הספציפיות שצריך לייבא מכל DLL. נתחיל בתהליך החיפוש.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0013DE50	10	06	14	00	00	00	00	00	2C	06	14	00	00	00	00	00
0013DE60	3E	06	14	00	00	00	00	00	00	00	00	00	00	00	00	00

כפי שאפשר לראות, הכתובת 0x0013DE50 כוללת רשימה של 3 כתובות, שאחריהן יש רק אפסים (אחרי האפסים תתחיל רשימה של שמות הפונקציות של ה-DLL הבא). נמיר את הכתובות ל-Raw:

- 0x00140610 – 0x1A00 = 0x0013EC10
- 0x0014062C – 0x1A00 = 0x0013EC2C
- 0x0014063E – 0x1A00 = 0x0013EC3E

נחפש את הכתובות הללו בקובץ ה-PE, ונמצא שם את שמות הפונקציות:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
0013EC10	00	00	47	65	74	49	6E	73	74	61	6C	6C	44	65	74	61	..GetInstallDeta
0013EC20	69	6C	73	50	61	79	6C	6F	61	64	00	00	00	00	53	69	ilsPayload....Si
0013EC30	67	6E	61	6C	43	68	72	6F	6D	65	45	6C	66	00	00	00	gnalChromeElf...
0013EC40	53	69	67	6E	61	6C	49	6E	69	74	69	61	6C	69	7A	65	SignalInitialize
0013EC50	43	72	61	73	68	52	65	70	6F	72	74	69	6E	67	00	00	CrashReporting..

שימו לב לכך שהכתובות מצביעות לא בדיוק על שמות הפונקציות, אלא שני בתים לפני שמות הפונקציות, שני בתים שבמקרה זה שווים 0x0000. בלי להכנס ליותר מדי פרטים, ניתן לייבא פונקציה לא לפי השם שלה אלא לפי המספר הסידורי שלה ב-DLL. שני בתים אלו הם שדה שנקרא "Hint".

עד כאן הגענו למסקנה, שהנתונים ב-Import Directory הם:

- שמות ה-DLLים שמהם צריך לייבא פונקציות
- עבור כל DLL, השמות של הפונקציות שצריך לייבא מתוכו

מה נותר לנו לחפש? הדבר היחידי שחסר לנו, הוא הכתובות של הפונקציות הללו. ברגע שיהיו לנו גם את הכתובות של כל הפונקציות המיובאות, כל קריאה שלנו לפונקציה מתוך DLL כלשהו תתורגם בקלות לכתובת אליה יש לקפוץ.

נעבור לשדה החמישי והאחרון. שדה זה נקרא IAT, קיצור של Import Address Table, וכפי שאפשר להבין מהשם של טבלה זו, כאן שמורות הכתובות של הפונקציות המיובאות מה-DLLים. כפי שחישבנו לפני כן, הטבלה נמצאת בכתובת 0x0013E530 בקובץ ה-PE. גשו לכתובת זו.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
0013E530	10	06	14	00	00	00	00	00	2C	06	14	00	00	00	00	00	□□□.....,□□.....
0013E540	3E	06	14	00	00	00	00	00	00	00	00	00	00	00	00	00	>□□.............

מה קורה פה? אלו בדיוק הכתובות שנמצאות בשדה הראשון, ה-Original First Thunk. במקום הכתובות של הפונקציות בזיכרון, קיבלנו שוב מצביעים לשמות הפונקציות. חישבו על כך- מדוע קובץ ה-PE לא כולל את הכתובות של הפונקציות?

התשובה היא כמובן, שקובץ ה-PE לא יכול להכיל את הכתובות של הפונקציות מכיוון שהן לא ידועות בזמן ביצוע ה-Linking. אי אפשר לדעת לאן ה-DLLים ייטענו. את הכתובות הללו אפשר לבדוק רק בזמן ביצוע ה-Loading. אם כך, בואו נעשה זאת.

כדי לבדוק את תמונת הזיכרון של תוכנה שעברה תהליך Loading, נשתמש גם הפעם ב- WinDbg. באמצעות הפקודה lm (קיצור של List Modules), נבדוק להיכן נטען הקובץ chrome.exe

```
0:000> lm
start             end                 module name
00007ff7`caf40000 00007ff7`cb0ec000   chrome
00007ff8`1ddd0000 00007ff8`1deb2000   chrome_elf
00007ff8`4a170000 00007ff8`4a17a000   VERSION
00007ff8`4ec30000 00007ff8`4ecbf000   apphelp
00007ff8`513f0000 00007ff8`51693000   KERNELBASE
00007ff8`52f40000 00007ff8`52fde000   msvcrt
00007ff8`537c0000 00007ff8`53872000   KERNEL32
00007ff8`54460000 00007ff8`54650000   ntdll
```

שימו לב לכך שכל הכתובות הינן כתובות של 64 ביט (הקוד הורץ על מעבד ומערכת הפעלה של 64 ביט).

המודול chrome נטען לכתובת 0x00007FF7CAF40000. נתבונן בזיכרון בכתובת זו:

```
Memory
Address: 7FF7CAF40000
00007FF7`CAF40000  4D 5A 78 00 01 00 00 00 04 00 00 00 00 00 00 00
00007FF7`CAF40010  00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
```

ואנחנו אכן רואים את "MZ", מספר הקסם 4D5A, שמציין את תחילת הקובץ שנטען ל-RAM.

נעבור ל-IAT, שנמצא ב-RVA של 0x0013FF30 מתחילת הקובץ בזיכרון.

0x7FF7CAF40000 + 0x0013FF30 = 0x7FF7CB07FF30

```
Memory
Address: 00007ff7cb07FF30
00007FF7`CB07FF20  FE 16 14 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FF7`CB07FF30  74 3B DD 1D F8 7F 00 00 05 10 DD 1D F8 7F 00 00
00007FF7`CB07FF40  00 10 DD 1D F8 7F 00 00 00 00 00 00 00 00 00 00
```

כפי שאפשר לראות, הכתובות שנמצאות ב-IAT הוחלפו בתהליך ה-Loading. נותר לנו רק לוודא שהכתובות אכן מצביעות אל הפונקציות המתאימות בתוך ה-DLLים. הפונקציה הראשונה צריכה להיות GetInstallDetailsPayload מתוך chrome_elf.dll (אם לא ברור לכם מדוע, היזכרו בסדר של ה-DLLים ובסדר של הפונקציות בתוך ה-DLLים, כפי שראינו בתוך ה-Import Table).

ניקח את הכתובת הראשונה, כתובת של 64 ביט:

0x00007FF81DDD3B74

נשווה את הכתובת הזו לכתובת שבה נמצא GetInstallDetailsPayload מתוך chrome_elf.dll.

```
0:000> x chrome_elf!GetI*
00007ff8`1ddd3b74 chrome_elf!GetInstallDetailsPayload (void)
```

וכפי שאפשר לראות, הכתובת שקיימת ב-IAT היא אכן הכתובת של הפונקציה שנמצאת ב-Import Table.

9.3 מנגנון ה-ASLR

ראשי התיבות של ASLR הן Address Space Layout Randomization. כדי להבין מה פירוש הדברים, נתחיל מהבסיס. כפי שראינו, בקובץ ה-PE יש שדה של Base Address, שקובע את הכתובת המועדפת שאליה ייטען קובץ ה-exe או ה-DLL. זו כתובת מועדפת בלבד, מערכת ההפעלה לא חייבת להתחשב בהעדפה. למעשה, הגיוני שקבצי DLL לא ייטענו לכתובת המועדפת, מכיוון שסביר שהיא כבר תפוסה. עבור קבצי exe הכתובת המועדפת ככל הנראה לא תפוסה, משום שעם יצירת Process חדש הם הדבר הראשון שנטען לזיכרון, כשמרחב הזיכרון הוירטואלי עוד פנוי.

בעבר, כאשר מנגנון ASLR לא היה קיים, מערכת ההפעלה Windows היתה מבצעת Loading לכתובות ברירת מחדל: קבצי exe שלא ביקשו כתובת מועדפת היו נטענים לכתובת 0x00400000, ואילו קבצי DLL היו נטענים לכתובת 0x10000000. אם הכתובות הללו היו תפוסות, ה-Loader היה מבצע Relocation לכתובות אחרות.

הדבר הקל על כותבי נוזקות, שיכלו להניח שחלקי קוד, אזורי זיכרון או DLLים יימצאו בכתובות מסוימות בזיכרון.

מנגנון ה-ASLR משפיע על ההיסט שבו ייטענו לזיכרון קבצי exe ו-DLLים ועל ההיסטים של ה-Stack וה-Heap עבור כל מידע שנטען למרחב הזיכרון של Process. באופן זה, לדוגמה, מי שמצליח לכתוב קוד ל-Stack ומעוניין לקפוץ ממנו חזרה אל ה-Code Section, לא יכול לדעת מראש את המרחק שיהיה עליו לקפוץ.

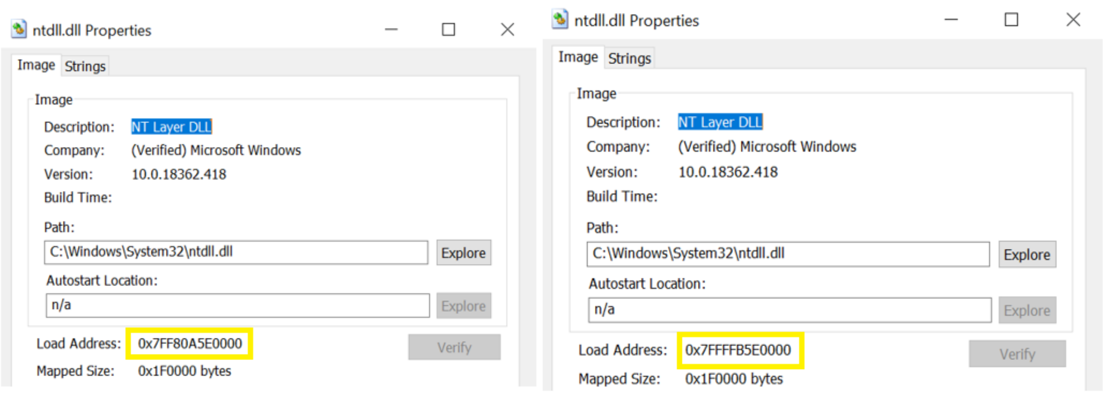
בניגוד ל-Stack ול- Heap, קבצי DLL דורשים שיתוף זיכרון ולכן אי אפשר לטעון אותם בכתובת שונה עבור כל Process. במקרה של DLLים, בזמן תהליך עליית מערכת ההפעלה נוצר על ידי מנהל הזיכרון מספר אקראי, אליו נטען ה-DLL הראשון. לאחר מכן, כל DLL נטען לכתובת גבוהה יותר בזיכרון. מרגע ש-DLL כלשהו נטען לכתובת בזיכרון, זו תהיה הכתובת במרחב הזיכרון עבור כל ה-Processים שישתמשו בו. אם נצפה בכתובת של DLL כלשהו ולאחר מכן נבצע אתחול למחשב שלנו, לאחר העליה הכתובת תשתנה.

תרגיל 9.11 **מנגנון ה-**ASLR

נסו זאת! פיתחו את Process Explorer ובידקו מה הכתובת של ntdll.dll, לדוגמה.

הדרכה:

- תחת תפריט View, בחרו להציג את ה-Lower Pane
- תחת תפריט View, בתוך Lower Pane View, בחרו באפשרות DLLs
- הקליקו על Process כלשהו וחפשו את ntdll.dll, בידקו אם אתם מתבוננים בקובץ שנמצא בתיקיית System32 (כלומר גרסת ה-64 ביט) או SysWow64 (כלומר גרסת ה-32 ביט), כדי להיות עקביים בפעם הבאה שתעשו את החיפוש. הקליקו עליו וחפשו את שדה ה-Load Address.
- חיזרו על הבדיקה לאחר אתחול המחשב, שימו לב לכך שה-Load Address משתנה:

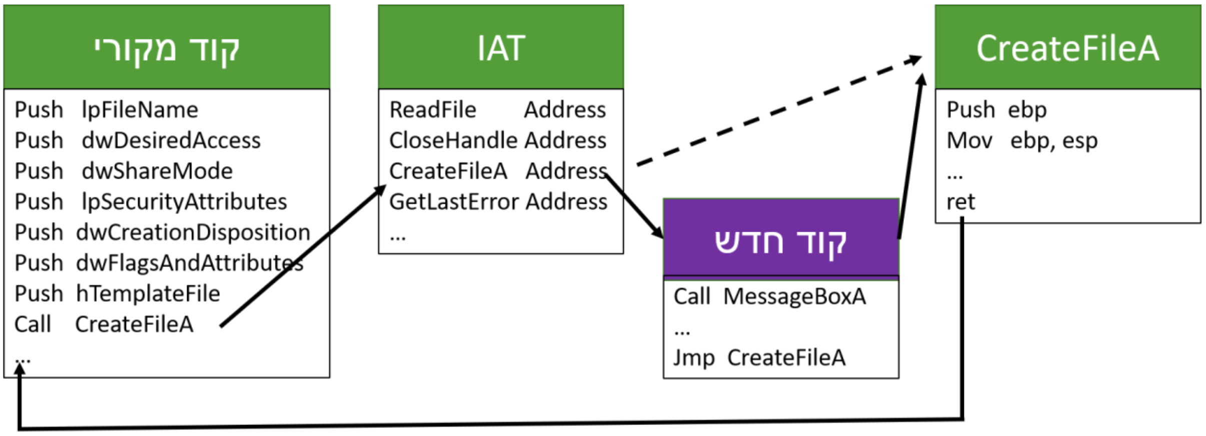


ה-*Load Address* של *ntdll.dll* בשתי עליות מחשב שונות (צפוי שתקבלו כתובות שונות)

תרגיל 9.12 תרגיל מסכם- IAT Hooking

בתרגיל זה נתנסה בהיכרות מעשית עם השדות השונים של פורמט PE, ותוך כדי כך נגרום לתוכנית לבצע דברים שהיא אינה אמורה לבצע.

למדנו שבתוך פורמט ה-PE יש כתובת של טבלה, Import Address Table או בקיצור IAT, ושהטבלה הזו מחזיקה את הכתובות הוירטואליות של כל הפונקציות שאנחנו מייבאים באמצעות DLLים. ה-IAT כולל צמדים של שם פונקציה + כתובת וירטואלית. כאשר אנחנו מבצעים Hooking אנחנו גורמים לקריאה לפונקציה כלשהי להפעיל פונקציה אחרת, לפי בחירתנו. לדוגמה, אם נבצע קריאה ל-CreateFileA, הכתובת של הפונקציה תתברר ב-IAT ואז תתבצע קפיצה אל קוד הפונקציה. בסיום ריצת הפונקציה, פקודת Ret מחזירה את מצביע הפקודות, הרגיסטר EIP, אל הפקודה הבאה בקוד.



הרעיון הבסיסי של IAT Hooking הוא פשוט מאד- נשנה את הטבלה, באופן שליד השם של פונקציה כלשהי (ה-Hooked Function) תופיעה כתובת של קוד חדש. הקוד החדש יעשה מה שיעשה, ובסוף התהליך יקפוץ אל הפונקציה המקורית. כך לדוגמה נראה תהליך של IAT Hooking לפונקציה CreateFileA, כך שקריאה ל-CreateFileA גורמת להרצת קוד שמפעיל את הפונקציה MessageBoxA.

הדגמת התהליך נמצאת בסרטון הבא:

https://data.cyber.org.il/os/IATHooking.mp4

שימו לב לכך, שפעולת ה-IAT Hooking לא תעבוד במקרה של Run Time Linking (כלומר שימוש ב-GetProcAddress ו-LoadLibrary), מכיוון שה-IAT מיועד רק ל-Load Time Linking. פונקציה שנמצאת ב-DLL שנטען ב-Run Time Linking, כלל אינה נשמרת ב-IAT.

שלבי התרגיל

1. כיתבו תוכנית שפותחת קובץ טקסט לקריאה ומדפיסה את תוכנו (שימוש ב- CreateFileA ו-ReadFile).
2. כיתבו פונקציה כלשהי, שתהיה הפונקציה שתחליף את CreateFileA. הפונקציה תקרא ל-MssageBoxA עם הודעה כרצונכם.

3. הוסיפו לקוד את הפונקציה Hook. הפונקציה מקבלת שלושה פרמטרים: השם של ה- Hooked Function, שם ה-DLL שבו היא נמצאת, וכתובת הפונקציה שאנחנו רוצים להריץ במקום ה- Hooked Function. קטע הקוד שבו מתבצע החיפוש אחר הפונקציה שאנחנו רוצים לעשות לה Hooking מבוסס על קוד מתוך הספר המומלץ"The Rootkit Arsenal" מאת Bill Blunden.

להלן קוד שמבצע את התרגיל, אך חסרים בו דברים ובמקומם יש "???".

https://data.cyber.org.il/os/IATHooking.cpp

נעבור על הקוד ונסביר חלק חלק.

הקוד מחולק ל-3 חלקים- פונקציית ה-main, פונקציית ה-hooking ופונקציית ShowMsg, שהיא הפונקציה שאנחנו מנסים לשתול ב-IAT במקום הפונקציה CreateFileA.

פונקציית ה-main כוללת קריאה לפונקציית ה-hooking ולאחר מכן לפונקציות CreateFileA ו-ReadFile המוכרות לנו כבר, כך שאין מה להרחיב עליהן.

הפונקציה ShowMsg היא הקוד שאנחנו רוצים שירוץ במקום CreateFileA. במקרה שלנו, נרצה פשוט להפעיל חלון עם הודעה כלשהי, ואז להמשיך ישר לביצוע של CreateFileA. לפונקציה הזו נתייחס בסוף.

למעט הפונקציה Hook, כל הקוד שלם ותקין. לכן נתמקד בפונקציה זו.

קוד הפונקציה Hook מתחיל ב-GetModuleHandle(NULL).

```
// Get base address of currently running .exe
DWORD baseAddress = (DWORD)GetModuleHandle(NULL);
```

קריאה זו גורמת לכך שתוחזר כתובת הבסיס אליה נטענה התוכנית.

אנחנו מצפים למצוא בכתובת זו את מילת הקסם "MZ". הכתובת נכנסת לתוך המשתנה BaseAddress, ולאחר מכן מבוצע לו Casting למבנה מסוג PIMAGE_DOS_HEADER.

```
// Get the import directory address
dosHeader = (PIMAGE_DOS_HEADER)(baseAddress);
if (((*dosHeader).???) != IMAGE_DOS_SIGNATURE) {
    return 0;
}
```

בשלב זה מומלץ לעצור באמצעות breakpoint, ולבדוק את שמות השדות השונים שיש ב-dosHeader. אחד מהשדות הללו דרוש לטובת בדיקת תקינות, שמוודאת שפורמט קובץ ההרצה הוא אכן תקין. טיפ: אם תכתבו את הקוד הנ"ל ותמחקו את סימני השאלה, תופיע לכם ב-visual studio רשימה עם כל השדות, לבחירתכם. לא צריך לנחש ☺

נמצא את ה-NT Header. כזכור מי שמצביע עליו הוא שדה מסוים ב-DOS Header. הכתובת היא כתובת וירטואלית יחסית, RVA, לכן כדי להגיע ל- NT Header צריך לחבר אותה עם ה-Base Address. לאחר שביצענו זאת, נבדוק שאנחנו אכן ב-NT Header באמצעות שדה מסויים שתמיד מופיע בתחילת ה- NT Header:

```
// Locate NT header
NTHeader = (PIMAGE_NT_HEADERS)(baseAddress + (*dosHeader).???);
if (((*NTHeader).???) != IMAGE_NT_SIGNATURE) {
    return 0;
}
```

איתור ה-Optional Header מתבצע באופן דומה. יש שדה ב-NT Header שמצביע עליו

```
// Locate optional header
optionalHeader = &(*NTHeader).???;
if (((*optionalHeader).???) != 0x10B) {
    return 0;
}
```

השלב הבא מטרתו להביא אותנו אל הטבלה שמחזיקה את שמות כל ה-DLLים שישנם, כדי שנוכל לבדוק אם ה-DLL שאנחנו מחפשים נמצא. אנחנו כבר יודעים היכן נמצא ה- Optional Header, ולמדנו על מבנה נתונים שהוא מחזיק בתוכו ושכולל את המצביעים למבני נתונים מסוג Image Data Directory. אחד מהמצביעים הללו הוא ה- .Import Directory

```
importDirectory =
    (*optionalHeader).???[IMAGE_DIRECTORY_ENTRY_IMPORT];
```

מבנה הנתונים Image Data Directory כולל שני שדות:

- Virtual Address – הכתובת היחסית של אזור הזיכרון
- Size – גודלו של אזור הזיכרון

מבין שני השדות הללו, עלינו לבחור מהו השדה המתאים בשביל לחשב את הכתובת של ה-Import Section:

```
descriptorStartRVA = importDirectory.???;
```

שימו לב, כרגיל חישבנו כתובת שהיא RVA, יחסי. אנחנו כבר יודעים מה צריך כדי להמיר RVA לכתובת בזיכרון. התוצאה היא הכתובת של מבנה הנתונים של ה-Image Import Descriptor. כפי שראינו, מבנה הנתונים הזה מחזיק הן את שם ה-DLL, הן את שמות הפונקציות שמיובאות והן את הכתובות שלהן.

```
importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(??? + ???);
```

ה-importDescriptor מצביע על האיבר הראשון במערך של Image Import Descriptors. האיבר הראשון כולל את הנתונים של ה-DLL הראשון שמיובא, וזה לא בהכרח ה-DLL שאנחנו מחפשים. לכן נצטרך בכל פעם לבדוק ששם ה-DLL שהוא מצביע עליו הוא אכן ה-DLL שאנחנו מחפשים, ואם לא- נמשיך לאיבר הבא במערך. אך מה אם ה-DLL שלנו כלל לא מיובא? נצטרך תנאי עצירה כלשהו. לשמחתנו, אם הגענו לסוף המערך נזהה זאת על ידי שדה ה-Characteristics, שיהיה שווה לאפס.

```
index = 0;
char *DLL_name;
// Look for the DLL which includes the function for hooking
while (???.Characteristics != 0) {
    DLL_name = (char*)(baseAddress + ???.Name);
    printf("DLL name: %s\n", DLL_name);
    if (!strcmp(DLL_to_hook, DLL_name))
        break;
    index++;
}
```

כעת אנחנו נמצאים על מבנה הנתונים שמצביע על ה-DLL הנכון, וצריך לחפש אם הפונקציה שאנחנו רוצים לבצע לה Hooking נמצאת בו. אם נמצא אותה, נצטרך לבדוק גם את השם שלה וגם את הכתובת בה היא נמצאת. לכן נכין מצביעים הן לטבלת שמות הפונקציות והן לטבלת הכתובות של הפונקציות.

```
// Search for requested function in the DLL
PIMAGE_THUNK_DATA thunkILT; // Import Lookup Table - names
PIMAGE_THUNK_DATA thunkIAT; // Import Address Table - addresses
PIMAGE_IMPORT_BY_NAME nameData;

thunkILT = (PIMAGE_THUNK_DATA)(??? + importDescriptor[index].???);
thunkIAT = (PIMAGE_THUNK_DATA)(??? + importDescriptor[index].???);
if ((thunkIAT == NULL) or (thunkILT == NULL)) {
    return 0;
}
```

קטע הקוד הבא מקבל מצביע אל השם של כל פונקציה ופונקציה שה-DLL מייבא, ובודק אם היא שווה לשם הפונקציה שאנחנו מבקשים לבצע לה Hooking:

```
while (((*thunkILT).u1.AddressOfData != 0) &
    (!((*thunkILT).u1.Ordinal & IMAGE_ORDINAL_FLAG)))
{
    nameData = (PIMAGE_IMPORT_BY_NAME)(baseAddress +
        (*thunkILT).u1.AddressOfData);
    if (!strcmp(func_to_hook, (char *)(*nameData).Name))
        break;
    thunkIAT++;
    thunkILT++;
}
```

אם יש התאמה- נשנה את הכתובת ובכך נשלים את תהליך ה-Hooking. שימו לב לכך שעל הדרך אנחנו שומרים גם את הכתובת של ה- Hooked Function, אחרת נאבד את היכולת לקרוא לה. דבר זה יהיה שימושי בהמשך.

```
// IAT Hooking, Barak Gonen 2020

#include "pch.h"

#define MAX 20
#define FILENAME "..\\bla.txt"

int hook(PCSTR func_to_hook, PCSTR DLL_to_hook,
         DWORD new_func_address);
void show_msg();
DWORD saved_hooked_func_addr;

int main()
{
    PCSTR func_to_hook = "CreateFileA";
    PCSTR DLL_to_hook = "KERNEL32.dll";
```

```
DWORD new_func_address = (DWORD)&show_msg;
HANDLE hFile;
hook(func_to_hook, DLL_to_hook, new_func_address);
// open the file for reading
hFile = CreateFileA(FILENAME,    // file name
    GENERIC_READ,           // open for read
    0,                      // do not share
    NULL,                   // default security
    OPEN_EXISTING,          // open only if exists
    FILE_ATTRIBUTE_NORMAL,  // normal file
    NULL);                  // no attr. template


// if file was not opened, print error code and return
if (hFile == INVALID_HANDLE_VALUE) {
    DWORD error = GetLastError();
    printf("Error code: %d\n", error);
    return 0;
}

// read some bytes and print them
CHAR buffer[MAX];
DWORD num;
LPDWORD numread = &num;
BOOL result = ReadFile(hFile,    // handle to open file
    buffer,                      // pointer to buffer
    MAX - 1,                     // bytes to read
    numread,                     // return value - bytes read
    NULL);                       // overlapped
buffer[*numread] = 0;
printf("%s\n", buffer);
```

```
    // close file
    CloseHandle(hFile);
    return 0;
};

int hook(PCSTR func_to_hook, PCSTR DLL_to_hook,
         DWORD new_func_address) {
    PIMAGE_DOS_HEADER dosHeader;
    PIMAGE_NT_HEADERS NTHeader;
    PIMAGE_OPTIONAL_HEADER32 optionalHeader;
    IMAGE_DATA_DIRECTORY importDirectory;
    DWORD descriptorStartRVA;
    PIMAGE_IMPORT_DESCRIPTOR importDescriptor;
    int index;

    // Get base address of currently running .exe
    DWORD baseAddress = (DWORD)GetModuleHandle(NULL);

    // Get the import directory address
    dosHeader = (PIMAGE_DOS_HEADER)(baseAddress);
    if (((*dosHeader).???) != IMAGE_DOS_SIGNATURE) {
        return 0;
    }

    NTHeader = (PIMAGE_NT_HEADERS)(baseAddress + *dosHeader).???);
    if (((*NTHeader).???) != IMAGE_NT_SIGNATURE) {
        return 0;
    }

    optionalHeader = &(*NTHeader).???;
    if (((*optionalHeader).???) != 0x10B) {
        return 0;
```

```
    }

    importDirectory =
        (*optionalHeader).???[IMAGE_DIRECTORY_ENTRY_IMPORT];
    descriptorStartRVA = importDirectory.???;
    importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(??? + ???);

    index = 0;
    char *DLL_name;
    // Look for the DLL which includes the function for hooking
    while (???.Characteristics != 0) {
        DLL_name = (char*)(??? + ???.Name);
        printf("DLL name: %s\n", DLL_name);
        if (!strcmp(DLL_to_hook, DLL_name))
            break;
        index++;
    }

    // exit if the DLL is not found in import directory
    if (importDescriptor[index].??? == 0) {
        printf("DLL was not found");
        return 0;
    }

    // Search for requested function in the DLL
    PIMAGE_THUNK_DATA thunkILT; // Import Lookup Table- names
    PIMAGE_THUNK_DATA thunkIAT; // Import Address Table- addresses
    PIMAGE_IMPORT_BY_NAME nameData;

    thunkILT = (PIMAGE_THUNK_DATA)(??? +
                  importDescriptor[index].???);
    thunkIAT = (PIMAGE_THUNK_DATA)(??? +
```

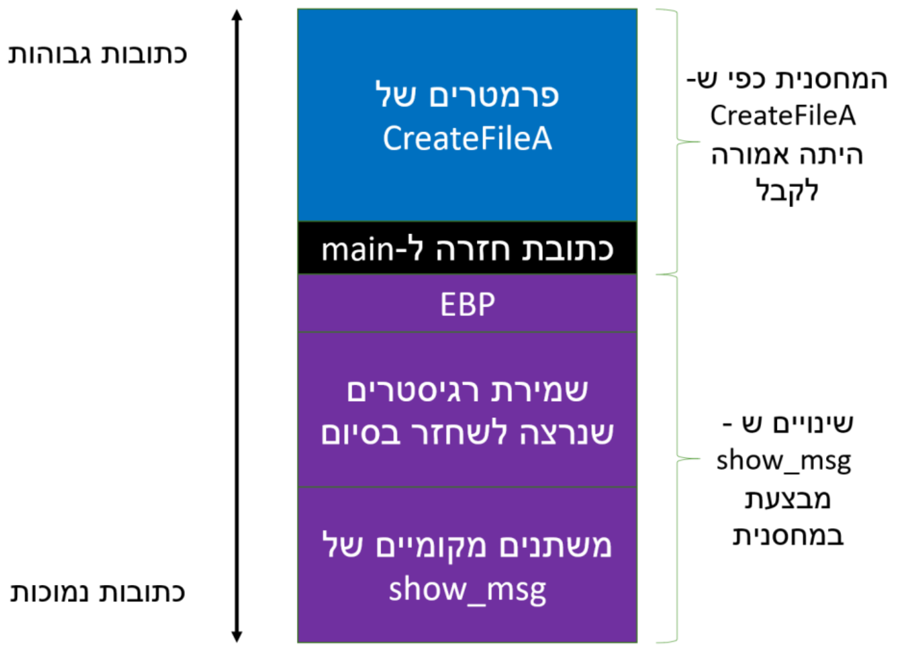
```
                importDescriptor[index].???);
    if ((thunkIAT == NULL) or (thunkILT == NULL)) {
        return 0;
    }

    while (((*thunkILT).u1.AddressOfData != 0) &
          (!((*thunkILT).u1.Ordinal & IMAGE_ORDINAL_FLAG))) {
        nameData = (PIMAGE_IMPORT_BY_NAME)(baseAddress +
            (*thunkILT).u1.AddressOfData);
        if (!strcmp(func_to_hook, (char *)(*nameData).???))
            break;
        thunkIAT++;
        thunkILT++;
    }

    // Hook IAT: Write over function pointer
    DWORD dwOld = NULL;
    saved_hooked_func_addr = (*thunkIAT).u1.Function;
    VirtualProtect((LPVOID)&((*thunkIAT).u1.Function),
                   sizeof(DWORD), PAGE_READWRITE, &dwOld);
    (*thunkIAT).u1.Function = new_func_address;
    VirtualProtect((LPVOID)&((*thunkIAT).u1.Function),
                   sizeof(DWORD), dwOld, NULL);
    return 1;
};

void ShowMsg() {
    MessageBoxA(0, "Hooked", "I Love Assembly", 0);
}
```

<u>שיפור א'</u>

כפי ששמתם לב, התוכנית יוצאת עם שגיאה מיד לאחר הקריאה ל-MessageBox. בשלב זה נבין מדוע זה קורה, ואז תתקנו את הדרוש תיקון.

לפני ההסבר- נסו לחשוב בעצמכם, מדוע יש בכלל בעיה? ומדוע היא מתרחשת מיד לאחר ש-MessageBox מסיימת את ריצתה?

כדי להבין את הבעיה, נסקור את המחסנית בזמן הריצה:



- לפני הקריאה ל-CreateFileA, נדחפים למחסנית 7 פרמטרים (בהתאם למספר הפרמטרים שהפונקציה CreateFile מקבלת). עם הקריאה לפונקציה, נדחפת למחסנית גם כתובת החזרה.
- במקום להריץ את CreateFileA, נקראת הפונקציה ShowMsg. הפונקציה הזו לא מודעת כלל לכך שדחפנו 7 פרמטרים למחסנית ולכן בסיום הריצה שלה היא לא מנקה אותם מהמחסנית.
- כפי שאפשר לראות בקוד האסמבלי הבא, מתבצעת בדיקה אחרי הפונקציה כדי לוודא שהמחסנית נותרה ללא שינוי. באופן טכני, הבדיקה מורכבת מ-3 שלבים:

  א. לפני דחיפת הפרמטרים למחסנית, שומרים את ערכו של מצביע המחסנית לתוך הרגיסטר esi

  ב. לאחר סיום ריצת הפונקציה, בודקים אם מצביע המחסנית עדיין שווה ל-esi

  ג. נקראת פונקציה, שבמקרה שהשוואה נכשלת, עוצרת את ריצת התוכנית עם Exception

איך אפשר לתקן את הבעיה?

אפשרות אחת היא ש-ShowMsg תדאג לנקות את הפרמטרים שנשלחו למחסנית. הביצוע של זה הוא טריקי, כיוון שאם סתם מעלים את ערכו של esp אז מאבדים את כתובת החזרה ששמורה על המחסנית.

האפשרות השניה, אותה נציג כאן, היא שלפני החזרה נבצע גם את הקריאה ל-CreateFileA. האפשרות הזו חביבה יותר, מכיוון שאם נבצע אותה המשך התוכנית יעבוד באופן תקין והפונקציה ReadFile תפעל

בהצלחה. אין הגיון בלבצע IAT Hooking אם התכנית שלנו מפסיקה לעבוד, כיוון שאז גם ה-Hook הולך לאיבוד.

באפשרות זו, נוריד את מצביע המחסנית אל כתובת החזרה, ואז נבצע jmp אל הכתובת המקורית של הפונקציה שעשינו לה Hooking. כיוון שכתובת החזרה כבר נמצאת במחסנית, התוצאה תהיה זהה למצב שבו עשינו Call רגיל לפונקציה זו. לאחר סיום ריצה הפונקציה היא תבצע בעצמה את ניקוי המחסנית מהפרמטרים שהועברו אליה ואת הקפיצה חזרה.

הטכניקה לעשות את הדברים הללו היא להכניס לתוך הקוד שלנו Inline Assembly. הדבר מתבצע באופן טכני פשוט על ידי כך שכותבים _asm ולאחר מכן פותחים סוגריים מסולסלים. להלן דוגמה, שאין בה כמובן שום הגיון מבחינת קוד האסמבלי אך היא ממחישה את צורת הכתיבה:

```
void ShowMsg() {
    MessageBoxA(0, "Hooked", "I Love Assembly", 0);
    _asm {
           xor eax, eax
           mov ebx, ecx
    }
}
```

מה נשים בתוך קוד האסמבלי? נצטרף פשוט לקחת את הסיומת של ShowMsg ולהעביר אותה לפני פקודת ה-jmp אל הפונקציה CreateFileA. הסיומת תלויה בקומפיילר, לכן נפתח את הקוד באמצעות IDA ונבחן מה הסיומת:

```
.text:00411A70
.text:00411A70      loc_411A70:
.text:00411A70      push     ebp          ; Save EBP
.text:00411A71      mov      ebp, esp     ; Set EBP
.text:00411A73      sub      esp, 0C0h    ; Save Memory for Local
                                          ; Variables
.text:00411A79      push     ebx          ; Save Registers
.text:00411A7A      push     esi
.text:00411A7B      push     edi
.text:00411A7C      lea      edi, [ebp-0C0h]
```

```
.text:00411A82      mov     ecx, 30h
.text:00411A87      mov     eax, 0CCCCCCCCh
.text:00411A8C      rep     stosd
.text:00411A8E      mov     esi, esp
.text:00411A90      push    0
.text:00411A92      push    offset aLoveasm ; "LoveASM"
.text:00411A97      push    offset aHooked  ; "Hooked"
.text:00411A9C      push    0
.text:00411A9E      call    ds:MessageBoxA
.text:00411AA4      cmp     esi, esp    ; Check Stack
.text:00411AA6      call    sub_41123A ; If Stack got bad, raise
                                        ; Exception
.text:00411ABD      pop     edi         ; Restore Registers
.text:00411ABE      pop     esi
.text:00411ABF      pop     ebx
.text:00411AC0      add     esp, 0C0h   ; Clear Local Variables
.text:00411AC6      cmp     ebp, esp    ; Check Stack
.text:00411AC8      call    sub_41123A ; If Stack got bad, raise
                                        ; Exception
.text:00411ACD      mov     esp, ebp    ; Restore ESP
.text:00411ACF      pop     ebp         ; Restore EBP
.text:00411AD0      retn
```

ארבעת החלקים המסומנים בירוק הם המקבילים לארבעת החלקים המסומנים באדום. באדום- תפיסת מקום במחסנית ושמירת ערכים, בירוק- שחזור הערכים ושחרור המקום במחסנית. אנחנו נכניס לתוך קוד האסמבלי פקודת jmp אל הכתובת של CreateFileA, ונדאג שכל החלקים שבירוק יתבצעו לפני פקודת ה-jmp שנכניס.

כך, בכניסה ל-CreateFileA המחסנית תראה בדיוק כפי שהיתה נראית אלמלא היינו מבצעים תהליך Hooking. נסו זאת!

<u>שיפור ב'</u>

בצעו IAT Hooking בשילוב DLL Injection. כלומר, הקוד שמבצע IAT Hooking לא יהיה חלק מקוד התוכנית המקורית כמו בדוגמה שעבדנו עליה עד כה, אלא יהיה חלק מה-DLL. טיפ: כפי שראינו, אפשר להשתמש ב- DLLMain כדי להריץ קוד באופן אוטומטי עם הקריאה ל-DLL. הקוד שיורץ יהיה כמובן הקוד של IAT Hooking.

9.4 סיכום

סיימנו את הפרק הקודם כשאנחנו יודעים כיצד ליצור קבצי DLL ו-exe. בפרק זה למדנו כיצד הקבצים הללו נטענים ל-RAM בזמן תהליך ה-Loading. ראינו שכדי שה-Loader יבצע את תפקידו בהצלחה הוא נדרש לקבל כמות רבה למדי של נתונים, אשר נמצאים בקובץ ה-exe וה-.DLL כדי להבין אילו נתונים יש בקבצים הללו, צללנו לתוך פורמט PE. לאחר שהבנו כיצד בנוי פורמט PE על השדות השונים שלו, התמקדנו בשדה מסויים אחד, ה-IAT, וראינו כיצד ניתן לנצל אותו על מנת לשנות את אופן הפעולה של תוכנית תוך כדי ריצה. למי שמעוניין בקורס מקיף אודות פורמט PE, מומלץ הקורס Life of Binaries, בלינק הבא:

https://www.youtube.com/watch?v=AeclzNQ0MxI&list=PLC6652F7766DEE46D

פרקים נוספים, בנושאים הבאים:

- מערכות קבצים ותהליך ה-Boot
- Powershell

צפויים להתפרסם בגרסה הבאה של הספר, בינואר 2021