היכרות עם WinDbg

מאת סשה גולדשטיין

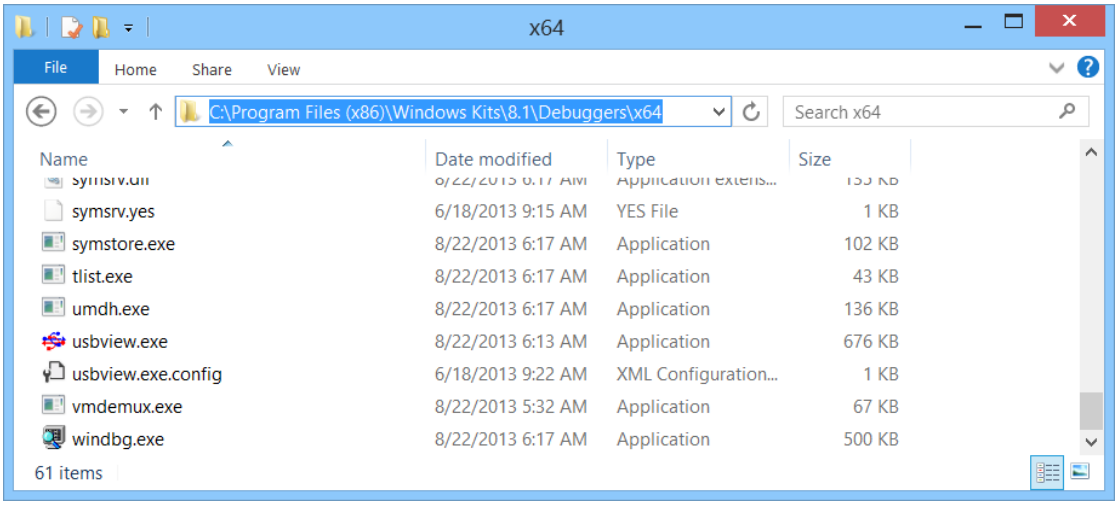
הקדמה

בגיליונות קודמים של Digital Whisper הצגתי את יכולות האוטומציה של WinDbg ואת האפשרויות שעומדות לרשותנו כדי לחלץ פרמטרים במוסכמת הקריאה של x64. מטרת המאמר הנוכחי (**שאולי יהפוך לסדרת מאמרים - זה תלוי גם בתגובות שלכם!**) היא להציג מבוא ל-WinDbg למי שלא מכיר את הכלי בכלל. אנחנו נסקור את היכולות הבסיסיות של WinDbg ונראה כיצד להשתמש בשורת הפקודה שלו. למרות שהדוגמה העיקרית שנשתמש בה היא תוכנית user mode, רוב הדברים שנלמד יהיו שימושיים גם ב-kernel mode ובניתוח של system dumps.

לפני שנתחיל, אני מאוד ממליץ לכם לעקוב אחרי הצעדים המתבצעים במאמר באמצעות תוכנית דוגמה פשוטה. על ידי התנסות עצמית בפקודות של WinDbg תוכלו להכיר את הכלי הרבה יותר טוב מאשר על ידי קריאה של חומר תיאורטי ומשעמם. את תוכנית הדוגמה תוכלו למצוא כאן ולקמפל אותה בעצמכם בעזרת Visual C++ 2013 (ניתן להוריד גרסה חינמית שלו מכאן), או שתוכלו להשתמש ב-exe. מוכן שניתן להוריד מכאן.

התקנת WinDbg

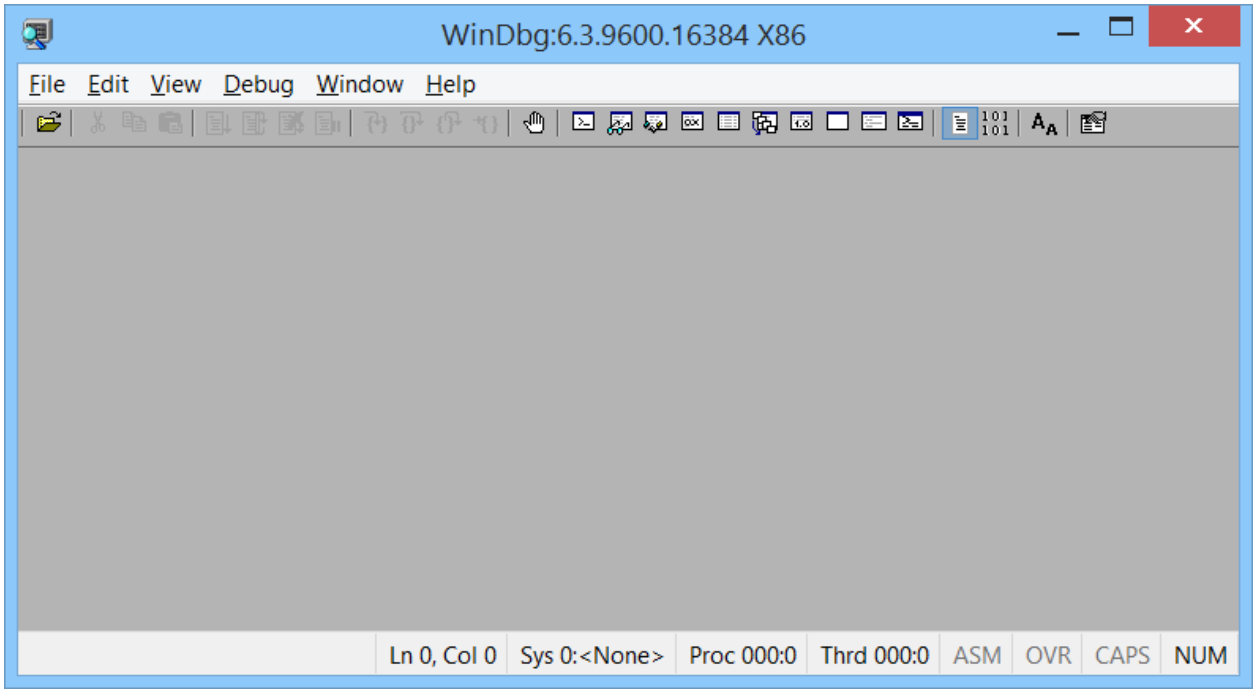
WinDbg הוא כלי של מיקרוסופט שניתן להתקין חינם כחלק מה-Windows SDK. ההתקנה לא עושה שום דבר מלבד העתקת קבצים, כך שאם יש לכם כבר WinDbg במחשב מסוים, תוכלו פשוט להעתיק את הספריה כולה ללא צורך בהתקנות נוספות.



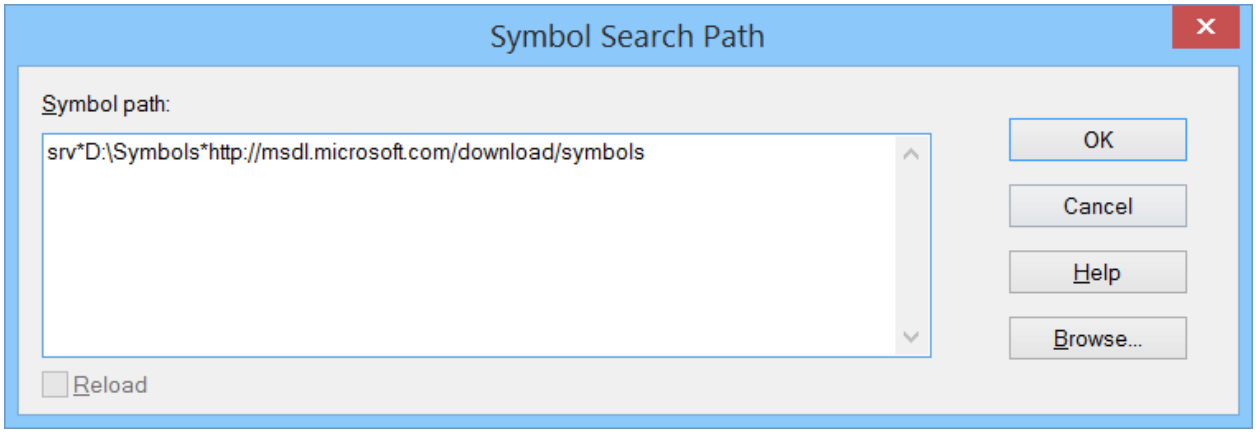
ספריית ברירת המחדל של ההתקנה היא C:\Program Files (x86)\Windows Kits\8.1\Debuggers, ושם תוכלו למצוא את WinDbg x64 ו-WinDbg x86. בניגוד ל-Visual Studio, כאן יש להקפיד להשתמש בגרסה המתאימה של WinDbg כתלות בתוכנית שעליה מסתכלים.

צעדים ראשונים

לאחר שבניתם (או הורדתם) את תוכנית הדוגמה, צריך להיות לכם קובץ הפעלה בשם WinDbgIntro.exe. כעת נרצה להריץ את התוכנית הזאת ב-WinDbg ולהתחיל להשתמש בכלי. הריצו את WinDbg. זהו החלון הראשי:



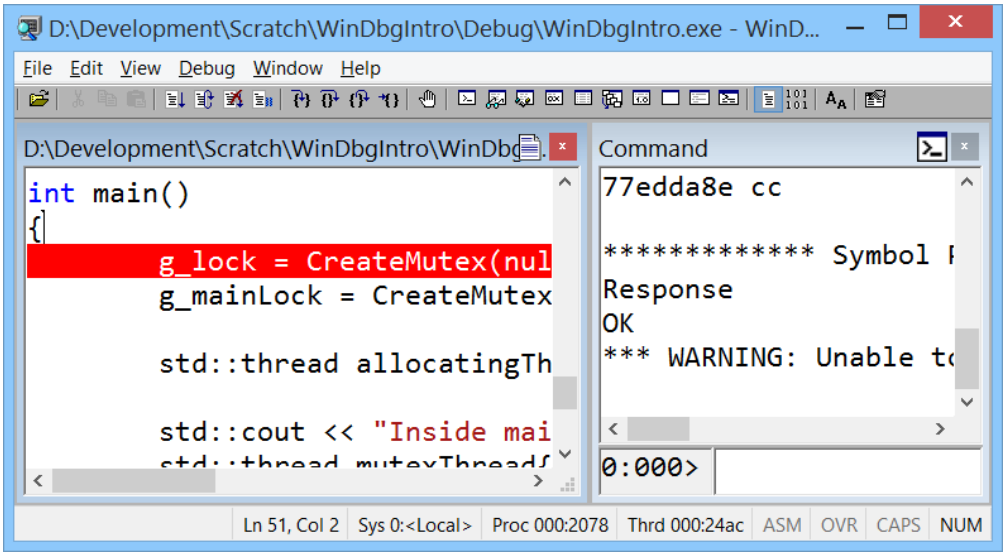
לפני הכל, יש להגדיר ל-WinDbg את הנתיב שבו הוא יחפש את קבצי הסמל (debugging symbols). את קבצי הסמל של התוכנית שלנו הוא ימצא באופן עצמאי בספריה של התוכנית, אבל את הנתיב לקבצי הסמל של מערכת ההפעלה יש להגדיר במפורש. בחרו ב-File > Symbol File Path, והכניסו את הטקסט הבא:



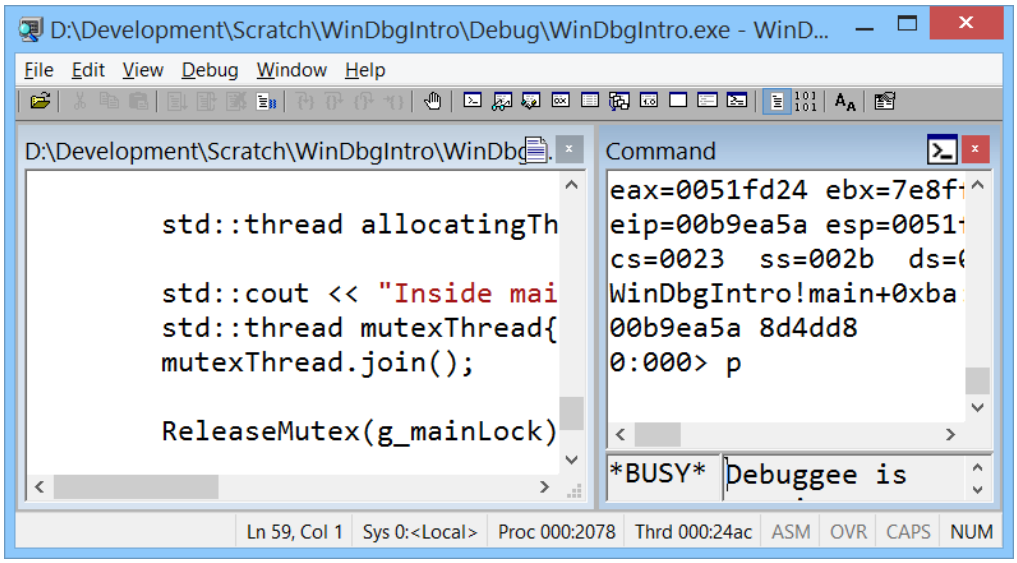
שורה זו מגדירה ל-WinDbg את שרת הסמל של מיקרוסופט, שממנו ניתן לקבל את קבצי הסמל של מערכת ההפעלה (וגם ספריות נוספות כגון ה-CRT ו-MFC). הנתיב המקומי D:\Symbols נתון, כמובן, לשיקולכם - זהו המקום אליו WinDbg יוריד את קבצי הסמל שהוא זקוק להם בפעם הראשונה בה הוא ניגש אליהם.

כעת אנו מוכנים להתחיל: בחרו את האפשרות File > Open Executable, ונווטו לתוכנית WinDbgIntro.exe. לפני שהתוכנית מתחילה לרוץ באופן מלא, WinDbg עוצר בנקודה שנקראת initial breakpoint, שהיא הרבה לפני נקודת הכניסה של התוכנית (main). נקודת עצירה (breakpoint) זו היא שימושית בין היתר כדי להגדיר נקודות עצירה נוספות.

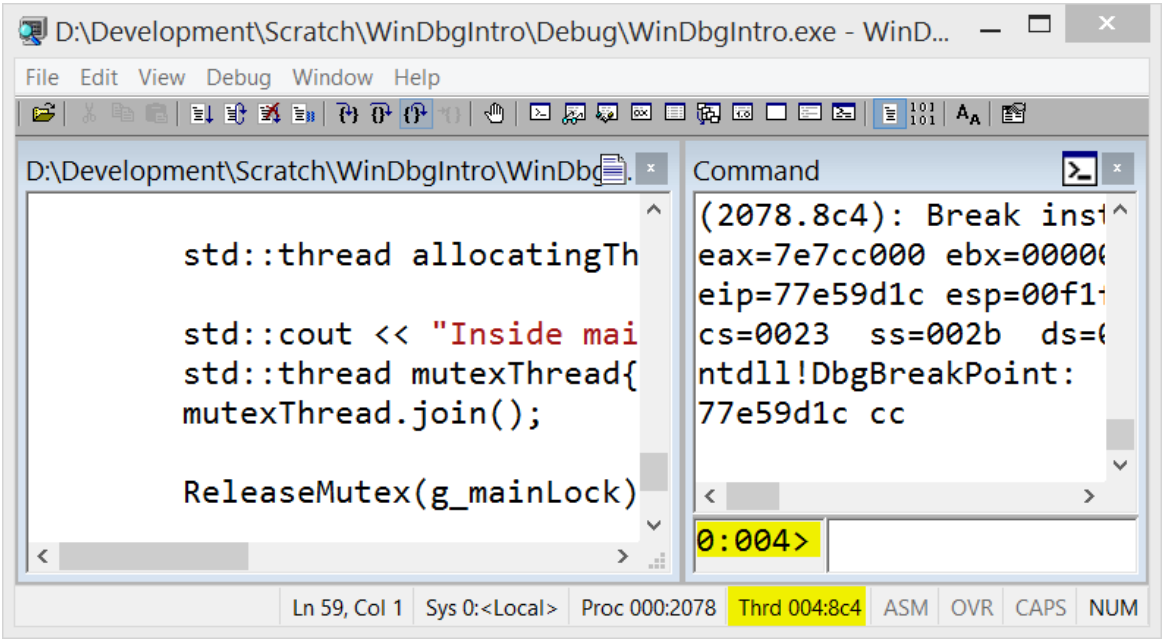
בחרו את האפשרות File > Open Source File, ונווטו לקובץ main.cpp שממנו בניתם את התוכנית. גללו למטה עד הפונקציה main, לחצו על השורה הראשונה של הפונקציה, ולחצו F9. כעת הגדרתם נקודת עצירה בכניסה לתוכנית.



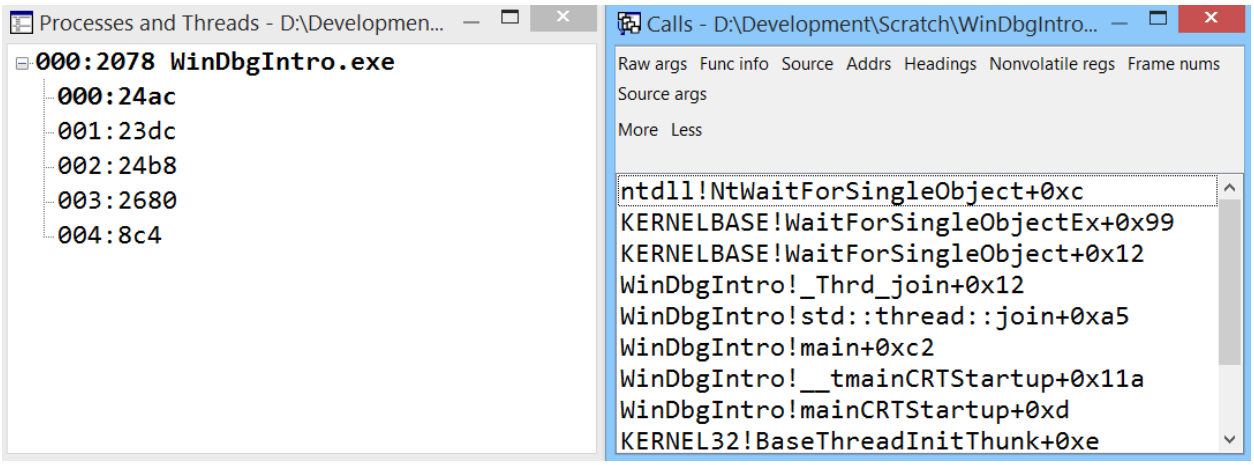
כעת אנו רוצים שהתוכנית תוכל להמשיך לרוץ ולהגיע לנקודת העצירה. לשם כך אפשר ללחוץ F5 או להקליד לשורת הפקודה של WinDbg את הפקודה g (קיצור של "go"). כעת אנו עוצרים בכניסה ל-main ויכולים לעבור על הקוד בצורה סדרתית, בדיוק כמו ב-Visual Studio, באמצעות המקשים F10 ו-F11. לחצו על F10 עד שתגיעו לשורה mutexThread.join();. כעת התוכנית רצה ותקועה, ולכן WinDbg נמצא במצב *BUSY*.



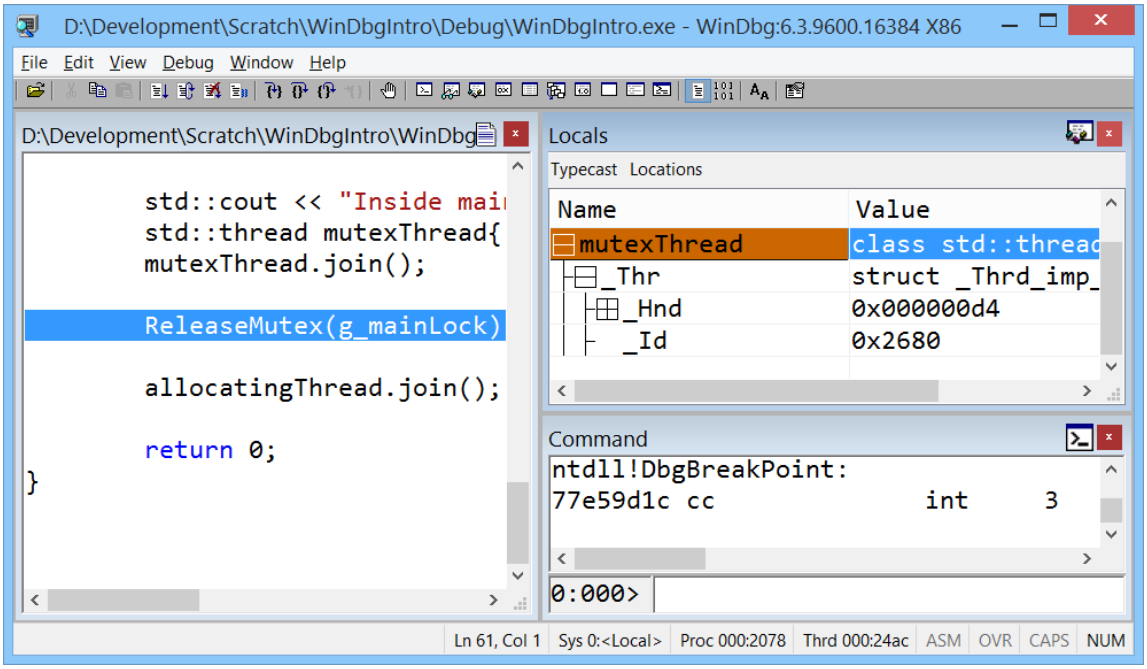
כדי לעצור את התוכנית בכל זאת, לחצו על Ctrl+Break (או השתמשו באפשרות התפריט Debug > Break). חשוב להבין שכאשר אנו עוצרים את התוכנית בצורה כזו, לא מובטח לנו שאנחנו שוב נמצאים באותו חוט (thread) שבו היינו בפעם האחרונה. למעשה, מספר החוט שאנחנו נמצאים בו כרגע מופיע בשורת הפקודה של WinDbg, וגם בשורת הסטטוס שמופיעה בתחתית המסך:



הגיע הזמן, אם כן, לעבור בין החוטים השונים ולהסתכל על מחסנית הקריאות שלהם. לשם כך נוכל לפתוח חלונות נוספים של WinDbg - את החלון View > Processes and Threads, ואת החלון View > Call Stack. לחצו פעם אחת על החוט שמספרו 000. שימו לב שמחסנית הקריאות מתעדכנת בהתאם.



כעת לחצו לחיצה כפולה על WinDbgIntro!main במחסנית הקריאות. WinDbg מביא אותנו לקוד של הפונקציה main, ומציג את השורה הבאה שתתבצע (ReleaseMutex(g_mainLock);). בהרבה מקרים מעניין גם להסתכל על המשתנים המקומיים של התוכנית, ולשם כך ניתן להיעזר בחלון View > Locals:



עד כה, השתמשנו בעיקר בממשק המשתמש הגרפי של WinDbg. יש לו יכולות מרשימות והוא בהחלט יכול לספק את הסחורה ולאפשר ניתוח וניפוי שגיאות רק על ידי לחיצות על חלונות וכפתורים. עם זאת, היכולות האמיתיות של WinDbg טמונות בשורת הפקודה שלו, שמאפשרת לראות את כל מה שניתן לקבל מהממשק הגרפי ועוד הרבה יותר, וכן לבצע אוטומציה של פעולות מורכבות שלא סביר לבצע בעזרת הממשק הגרפי.

שורת הפקודה של WinDbg

בשלב זה אני ממליץ לכם לסגור את כל החלונות של WinDbg למעט חלון שורת הפקודה (שכותרתו "Command"). כעת ננסה לקבל את אותה אינפורמציה כמו קודם באמצעות שורת הפקודה בלבד. נתחיל מרשימת כל החוטים - הריצו את הפקודה הפשוטה `~`:

```
0:000> ~
.  0  Id: 2078.24ac Suspend: 1 Teb: 7e8fe000 Unfrozen
   1  Id: 2078.23dc Suspend: 1 Teb: 7e8f8000 Unfrozen
   2  Id: 2078.24b8 Suspend: 1 Teb: 7e8f5000 Unfrozen
   3  Id: 2078.2680 Suspend: 1 Teb: 7e7cf000 Unfrozen
#  4  Id: 2078.8c4 Suspend: 1 Teb: 7e7cc000 Unfrozen
```

אנו רואים רשימה של כל החוטים שיש בתהליך. הבה נעבור לחוט מספר 3, באמצעות הפקודה `~3s`:

```
0:000> ~3s
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000002 edi=00000002
eip=77e6d72c esp=00e1f450 ebp=00e1f5d0 iopl=0         nv up ei pl nz ac pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000216
ntdll!NtWaitForMultipleObjects+0xc:
```

```
77e6d72c c21400          ret     14h
```

לאחר ביצוע המעבר, WinDbg מדפיס את ערכי האוגרים במעבד ואת הפקודה הבאה שתבוצע בחוט שבחרנו. בדרך כלל, המידע הזה הרבה פחות מעניין מאשר מחסנית הקריאות, שנוכל להציג באמצעות הפקודה kpn:

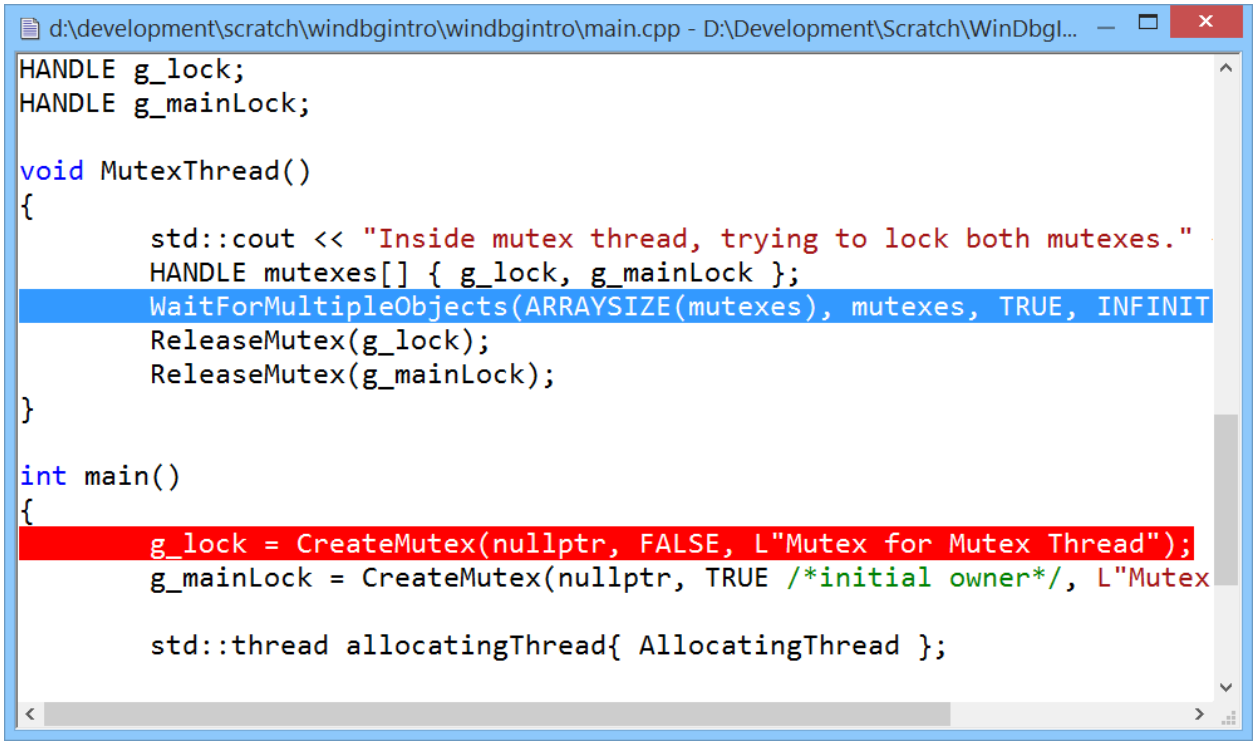
```
0:003> kpn
 # ChildEBP RetAddr
00 00e1f44c 7717ea7f ntdll!NtWaitForMultipleObjects+0xc
01 00e1f5d0 77609188 KERNELBASE!WaitForMultipleObjectsEx+0xdc
02 00e1f5ec 00b9038e KERNEL32!WaitForMultipleObjects+0x19
03 00e1f6e0 00b8c73c WinDbgIntro!MutexThread(void)+0x5e
[d:\development\scratch\windbgintro\windbgintro\main.cpp @ 45]
04 00e1f7c0 00b8c1d5 WinDbgIntro!std::_Bind<1,void,void (class std::tuple<> _Myfargs = class std::tuple<>,
struct std::_Arg_idx<> __formal = struct std::_Arg_idx<>)+0x2c [c:\program files (x86)\microsoft visual studio
12.0\vc\include\functional @ 1149]
05 00e1f8c0 00b90ee6 WinDbgIntro!std::_Bind<1,void,void (void)+0x45 [c:\program files (x86)\microsoft visual
studio 12.0\vc\include\functional @ 1138]
06 00e1f9a4 00b91a2c WinDbgIntro!std::_LaunchPad<std::_Bind<1,void,void (class
std::_LaunchPad<std::_Bind<1,void,void (__cdecl*const)(void)> > * _Ln = 0x0051fb18)+0x46 [c:\program files
(x86)\microsoft visual studio 12.0\vc\include\thr\xthread @ 196]
07 00e1fa88 00b98b06 WinDbgIntro!std::_LaunchPad<std::_Bind<1,void,void (void)+0x2c [c:\program files
(x86)\microsoft visual studio 12.0\vc\include\thr\xthread @ 187]
08 00e1fabc 00bf0411 WinDbgIntro!_Call_func(void * _Data = 0x0051fb18)+0x46
[f:\dd\vctools\crt\crtw32\stdcpp\thr\threadcall.cpp @ 28]
09 00e1faf8 00bf0671 WinDbgIntro!_callthreadstartex(void)+0x51 [f:\dd\vctools\crt\crtw32\startup\threadex.c @
376]
0a 00e1fb04 7760919f WinDbgIntro!_threadstartex(void * ptd = 0x00802f48)+0xb1
[f:\dd\vctools\crt\crtw32\startup\threadex.c @ 359]
0b 00e1fb10 77e7a8cb KERNEL32!BaseThreadInitThunk+0xe
0c 00e1fb54 77e7a8a1 ntdll!__RtlUserThreadStart+0x20
0d 00e1fb64 00000000 ntdll!_RtlUserThreadStart+0x1b
```

הפלט הארוך הזה הוא אכן מחסנית הקריאות של החוט שעברנו אליו. ליד כל שורה מופיע מספר סידורי (המתחיל למעלה ב-00), ערך האוגר EBP באותה פונקציה, כתובת החזרה של הפונקציה, השם של הפונקציה, פרמטרים אם ניתן להציגם, ומידע על קובץ המקור (.c/.cpp) שממנו הפונקציה מגיעה.

כך למשל, אם נסתכל על השורה המודגשת למעלה בצהוב, נראה שהחוט הנוכחי מריץ את הפונקציה MutexThread שמגיעה מהקובץ main.cpp, שורה מספר 45. כדי לגרום ל-WinDbg לעבור לפונקציה זו, נשתמש בפקודה frame 03. (המספר הוא המספר הסידורי המופיע בתחילת השורה, וכמובן יכול להיות שונה במקצת בתוכנית שלכם):

```
0:003> .frame 03
03 00e1f6e0 00b8c73c WinDbgIntro!MutexThread+0x5e
[d:\development\scratch\windbgintro\windbgintro\main.cpp @ 45]
```

בנוסף לביצוע המעבר בשורת הפקודה, WinDbg גם פותח באופן אוטומטי את קוד המקור של התוכנית:



מהתבוננות בקוד, כנראה נרצה לקבל את ערכיהם של המשתנים הגלובאליים g_lock ו-g_mainLock, וכן את ערכו של המשתנה המקומי mutexes. את כל אלה ניתן לעשות על ידי שימוש באופרטור ??, שמקבל שם של משתנה או ביטוי מורכב מעט יותר, ומדפיס את ערכו:

```
0:003> ?? g_lock
void * 0x0000003c
0:003> ?? g_mainLock
void * 0x00000044
0:003> ?? mutexes
void *[2] 0x00e1f6d4
0x0000003c
Void
0:003> ?? mutexes[0]
void * 0x0000003c
0:003> ?? mutexes[1]
void * 0x00000044
```

אנו רואים שהחוט הנוכחי קרא לפונקציית המתנה של מערכת ההפעלה, WaitForMultipleObjects, והעביר לה את שני המשתנים שראינו קודם (g_lock ו-g_mainLock).

מאחר שהם מסוג HANDLE, נוכל לקבל עליהם פרטים נוספים על ידי שימוש בפקודה !handle (שימו לב שהמספרים עצמם יכולים להיות שונים בהרצה שלכם!):

```
0:003> !handle @@c++(mutexes[0]) 8
Handle 3c
  Object Specific Information
    Mutex is Free
0:003> !handle @@c++(mutexes[1]) 8
Handle 44
  Object Specific Information
    Mutex is Owned
    Mutant Owner 2078.24ac
```

השימוש באופרטור המוזר @@c++ דרוש כדי להסביר ל-WinDbg שהביטוי בסוגריים הוא ביטוי בתחביר C++, שאינו תחביר ברירת המחדל של WinDbg. אבל המידע שקיבלנו גורם לתחביר המסורבל להיות מאוד משתלם: אנו יודעים כעת שהידית (handle) הראשונה מצביעה ל-mutex פנוי[1], והידית השנייה מצביעה ל-mutex שתפוס על ידי חוט מסוים שהמזהה שלו הוא 2078.24ac. החלק הראשון הוא מזהה התהליך, והחלק השני הוא מזהה החוט עצמו.

כיוון שסביר שהחוט התופס נמצא באותו תהליך כמו שלנו (אם כי זה לא תמיד המצב), אנו יכולים להשתמש בפקודה מיוחדת כדי לעבור לחוט התופס לפי מספרו:

```
0:003> ~~[24ac]s
eax=00000000 ebx=7e8ff000 ecx=00000000 edx=00000000 esi=00000000 edi=000000d4
eip=77e6d1bc esp=0051fa98 ebp=0051fb04 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
ntdll!NtWaitForSingleObject+0xc:
77e6d1bc c20c00          ret     0Ch
0:000> ~.
.  0  Id: 2078.24ac Suspend: 1 Teb: 7e8fe000 Unfrozen
      Start: WinDbgIntro!ILT+22805(_mainCRTStartup) (00b8191a)
      Priority: 0  Priority class: 32  Affinity: ff
```

אנו רואים אם כן שעברנו לחוט מספר 000, שהוא החוט הראשי של התוכנית. הוא תופס את ה-mutex שהחוט 003 מנסה להיכנס לתוכו. מקריאה של הקוד אכן ניתן לראות שזה המצב (שימו לב לקריאה ל-CreateMutex עם הפרמטר TRUE במקום השני, שגורם ל-mutex להיווצר כאשר הוא כבר תפוס על ידי החוט הנוכחי).

[1] תזכורת: mutex הוא מנגנון סנכרון המספק מניעה הדדית. כאשר חוט אחד תופס את ה- mutex, חוטים אחרים אינם יכולים לתפוס אותו ולהריץ את הקוד המוגן על ידיו. רק חוט אחד יכול להימצא בתוך ה- mutex בכל עת.

במקרים מסוג זה מעניין לעתים קרובות להבין מה עושה כרגע החוט התופס. אם נבין מה עושה החוט התופס (000), נוכל להבין מתי יש סיכוי שהוא ישחרר את ה-mutex, ועל ידי כך יאפשר התקדמות גם של החוט הממתין (003). נתבונן במחסנית הקריאות של החוט התופס על ידי שימוש בפקודה ~0 kpn:

```
0:000> ~0 kpn
 # ChildEBP RetAddr
00 0051fa94 771410fd ntdll!NtWaitForSingleObject+0xc
01 0051fb04 7714103d KERNELBASE!WaitForSingleObjectEx+0x99
02 0051fb18 00b98232 KERNELBASE!WaitForSingleObject+0x12
03 0051fb30 00b938a5 WinDbgIntro!_Thrd_join(struct _Thrd_imp_t thr = struct
_Thrd_imp_t, int * code = 0x00000000)+0x12
[f:\dd\vctools\crt\crtw32\stdcpp\thr\cthread.c @ 71]
04 0051fc3c 00b9ea62 WinDbgIntro!std::thread::join(void)+0xa5 [c:\program files
(x86)\microsoft visual studio 12.0\vc\include\thread @ 214]
05 0051fd4c 00be8b2a WinDbgIntro!main(void)+0xc2
[d:\development\scratch\windbgintro\windbgintro\main.cpp @ 61]
06 0051fd98 00be8d0d WinDbgIntro!__tmainCRTStartup(void)+0x11a
[f:\dd\vctools\crt\crtw32\startup\crt0.c @ 255]
07 0051fda0 7760919f WinDbgIntro!mainCRTStartup(void)+0xd
[f:\dd\vctools\crt\crtw32\startup\crt0.c @ 165]
08 0051fdac 77e7a8cb KERNEL32!BaseThreadInitThunk+0xe
09 0051fdf0 77e7a8a1 ntdll!__RtlUserThreadStart+0x20
0a 0051fe00 00000000 ntdll!_RtlUserThreadStart+0x1b
```

התוצאה נראית מעניינת - גם החוט התופס ממתין למשהו, וניתן להבין זאת מהקריאה ל-WaitForSingleObject. עוד לפני כן ניתן לראות קריאה לפונקציה std::thread::join, שממתינה לסיום של חוט מסוים לפני שהיא חוזרת. כדי להבין מיהו החוט הבא בתור שהחוט שלנו ממתין לו, אנחנו יכולים לנקוט במספר גישות. הראשונה היא פשוט להסתכל על הקוד שלנו (בפונקציה main) ולהבין מה אנחנו עושים:

d:\development\scratch\windbgintro\windbgintro\main.cpp - D:\Development\Scratch\WinDbgI...

```
{
        g_lock = CreateMutex(nullptr, FALSE, L"Mutex for Mutex Thread");
        g_mainLock = CreateMutex(nullptr, TRUE /*initial owner*/, L"Mutex

        std::thread allocatingThread{ AllocatingThread };

        std::cout << "Inside main thread, starting wait on mutex thread."
        std::thread mutexThread{ MutexThread };
        mutexThread.join();

        ReleaseMutex(g_mainLock);

        allocatingThread.join();

        return 0;
}
```

השורה הרלוונטית היא אחת מעל השורה המסומנת - `mutexThread.join();`. מיהו mutexThread? נוכל לגלות על ידי שימוש באופרטור המוכר לנו כבר:

```
0:000> ?? mutexThread
class std::thread
   +0x000 _Thr             : _Thrd_imp_t
0:000> ?? mutexThread._Thr
struct _Thrd_imp_t
   +0x000 _Hnd             : 0x000000d4 Void
   +0x004 _Id              : 0x2680
0:000> ~~[2680]
   3  Id: 2078.2680 Suspend: 1 Teb: 7e7cf000 Unfrozen
      Start: WinDbgIntro!_threadstartex (00bf05c0)
      Priority: 0  Priority class: 32  Affinity: ff
```

כעת הכל ברור: החוט הראשי ממתין לחוט שהמזהה שלו הוא 2680, או כמו שאנחנו מכירים אותו, חוט מספר 003. היינו כבר בחוט הזה ואנחנו מכירים את הסיפור: חוט 003 ממתין ל-mutex שתפוס על ידי החוט 000, שבתורו ממתין לחוט 003. זהו חֶבֶק (deadlock) שאין דרך לצאת ממנו.

בשלב זה כבר רכשנו כלים לביצוע ניתוח בסיסי ב-WinDbg, מעבר בין חוטים, התבוננות במחסנית ובמשתנים מקומיים, ואפילו ראינו כיצד לקבל פרטים על ידיות של מערכת ההפעלה. בתחילת המאמר גם ראינו כיצד להגדיר נקודת עצירה על ידי לחיצה על שורה מסוימת בקוד. גישה זו סבירה לנקודות עצירה

פשוטות, אבל לא מאפשרת להגדיר נקודת עצירה בקוד שאינו שלנו (כמו קוד של מערכת ההפעלה) או להגדיר נקודת עצירה שעושה משהו מתוחכם יותר מאשר פשוט... לעצור.

נקודות עצירה מתקדמות

שורת הפקודה של WinDbg מאפשרת להגדיר נקודות עצירה מתקדמות בכל מקום בתוכנית. כאשר מגיעים לנקודת העצירה, אנו יכולים להריץ באופן אוטומטי כל פקודה אחרת. כך למשל, אנחנו יכולים להדפיס את מחסנית הקריאות בהגיענו לנקודת העצירה; להציג ערך של משתנה מעניין; להגדיר נקודות עצירה נוספות; ואפילו פשוט להורות לתוכנית להמשיך לרוץ ובכלל לא לעצור.

תוכנית הדוגמה שאנו מריצים מכילה קוד המבצע הקצאות זיכרון רבות. לעתים הקצאות זיכרון רבות מובילות לדליפת זיכרון (אם חלקן לא משוחררות), ולעתים לבעיות ביצועים. לכן לפעמים זה חשוב להבין באילו מקומות בתוכנית מתבצעות הקצאות בגודל מסוים, או סתם הקצאות גדולות יחסית. מכיוון שזה לא מעשי להגדיר נקודת עצירה בכל שורה בתוכנית שמקצה זיכרון, אנו זקוקים לגישה חלופית. כל הקצאות הזיכרון הדינאמיות (מהערימה - heap) מתבצעות דרך פונקציה אחת בסופו של דבר, הפונקציה ntdll!RtlAllocateHeap. אנו יכולים להגדיר בה נקודת עצירה, וכך לגלות מיידית את המקומות בתוכנית שמבצעים הקצאת זיכרון. נגדיר את נקודת העצירה באמצעות הפקודה `bp ntdll!RtlAllocateHeap`, ניתן לתוכנית להמשיך לרוץ (F5 או הפקודה `g`), ונעצור בנקודת העצירה:

```
0:004> bp ntdll!RtlAllocateHeap
0:004> g
Breakpoint 0 hit
eax=00000041 ebx=00802308 ecx=00000041 edx=007e0000 esi=007afa70 edi=007ae5a4
eip=77e70821 esp=007ae3c8 ebp=007ae3dc iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
ntdll!RtlAllocateHeap:
77e70821 8bff            mov     edi,edi
0:001> kpn
 # ChildEBP RetAddr
00 007ae3c4 00c08891 ntdll!RtlAllocateHeap
01 007ae3dc 00be5a6f WinDbgIntro!_heap_alloc_base(unsigned int size = 0x41)+0x51
[f:\dd\vctools\crt\crtw32\heap\malloc.c @ 58]
02 007ae424 00be612d WinDbgIntro!_heap_alloc_dbg_impl(unsigned int nSize = 0x1d, int nBlockUse = 0n1, char *
szFileName = 0x00000000 "", int nLine = 0n0, int * errno_tmp = 0x007ae468)+0x1ff
[f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 431]
03 007ae444 00be60ca WinDbgIntro!_nh_malloc_dbg_impl(unsigned int nSize = 0x1d, int nhFlag = 0n0, int nBlockUse
= 0n1, char * szFileName = 0x00000000 "", int nLine = 0n0, int * errno_tmp = 0x007ae468)+0x1d
[f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 239]
04 007ae46c 00c024b9 WinDbgIntro!_nh_malloc_dbg(unsigned int nSize = 0x1d, int nhFlag = 0n0, int nBlockUse =
0n1, char * szFileName = 0x00000000 "", int nLine = 0n0)+0x2a [f:\dd\vctools\crt\crtw32\misc\dbgheap.c @ 302]
05 007ae48c 00be05cf WinDbgIntro!malloc(unsigned int nSize = 0x1d)+0x19
[f:\dd\vctools\crt\crtw32\misc\dbgmalloc.c @ 56]
06 007ae4a8 00b95fdc WinDbgIntro!operator new(unsigned int size = 0x1d)+0xf
[f:\dd\vctools\crt\crtw32\heap\new.cpp @ 59]
```

```
07 007ae4b4 00b8e5bc WinDbgIntro!operator new[](unsigned int count = 0x1d)+0xc
[f:\dd\vctools\crt\crtw32\stdcpp\newaop.cpp @ 6]
08 007ae5a4 00b90c00 WinDbgIntro!DynamicArray::DynamicArray(unsigned int len = 0x1d)+0x2c
[d:\development\scratch\windbgintro\windbgintro\main.cpp @ 13]
09 007afa68 00b8c73c WinDbgIntro!AllocatingThread(void)+0x90
[d:\development\scratch\windbgintro\windbgintro\main.cpp @ 32]
0a 007afb48 00b8c1d5 WinDbgIntro!std::_Bind<1,void,void (class std::tuple<> _Myfargs = class std::tuple<>,
struct std::_Arg_idx<> __formal = struct std::_Arg_idx<>)+0x2c [c:\program files (x86)\microsoft visual studio
12.0\vc\include\functional @ 1149]
0b 007afc48 00b90ee6 WinDbgIntro!std::_Bind<1,void,void (void)+0x45 [c:\program files (x86)\microsoft visual
studio 12.0\vc\include\functional @ 1138]
0c 007afd2c 00b91a2c WinDbgIntro!std::_LaunchPad<std::_Bind<1,void,void (class
std::_LaunchPad<std::_Bind<1,void,void (__cdecl*const)(void)> > * _Ln = 0x0051fb18)+0x46 [c:\program files
(x86)\microsoft visual studio 12.0\vc\include\thr\xthread @ 196]
0d 007afe10 00b98b06 WinDbgIntro!std::_LaunchPad<std::_Bind<1,void,void (void)+0x2c [c:\program files
(x86)\microsoft visual studio 12.0\vc\include\thr\xthread @ 187]
0e 007afe44 00bf0411 WinDbgIntro!_Call_func(void * _Data = 0x0051fb18)+0x46
[f:\dd\vctools\crt\crtw32\stdcpp\thr\threadcall.cpp @ 28]
0f 007afe80 00bf0671 WinDbgIntro!_callthreadstartex(void)+0x51 [f:\dd\vctools\crt\crtw32\startup\threadex.c @
376]
10 007afe8c 7760919f WinDbgIntro!_threadstartex(void * ptd = 0x00802308)+0xb1
[f:\dd\vctools\crt\crtw32\startup\threadex.c @ 359]
11 007afe98 77e7a8cb KERNEL32!BaseThreadInitThunk+0xe
12 007afedc 77e7a8a1 ntdll!__RtlUserThreadStart+0x20
13 007afeec 00000000 ntdll!_RtlUserThreadStart+0x1b
```

מחסנית הקריאות נראית מסובכת, אך לאמיתו של דבר החלק המעניין הוא המודגש בצהוב. הפונקציה AllocatingThread גרמה לקריאה לבנאי (constructor) של מחלקה בשם DynamicArray. בנאי זה קרא ל-[]operator new שמקצה זיכרון, והעביר לו בתור גודל את המספר 0x1d (שהוא 29 בבסיס עשרוני). מכאן ואילך אנו רואים שרשרת של קריאות שבסופן עצירה ב-RtlAllocateHeap, שבה הגדרנו את נקודת העצירה שלנו.

זה לא סביר, כמובן, שנעצור באופן ידני ונתבונן בגודל ההקצאה עבור כל הקצאה שהתוכנית עושה. אנו צריכים להגדיר את נקודת העצירה כך שתדפיס אוטומטית את גודל ההקצאה, ואולי פרטים נוספים אם ההקצאה מעניינת אותנו (למשל גדולה במיוחד). באופן עקרוני ניתן לעשות זאת גם כאשר עוצרים ב-RtlAllocateHeap, אבל מכיוון שאנחנו לא מכירים את הפרמטרים של הפונקציה הזאת, יהיה צריך לחלץ אותם בצורה ידנית מהמחסנית. זה אפשרי אבל לא נוח כל כך[2]. לחלופין, אנו יכולים לשים לב שאת הפרמטר של malloc אנחנו רואים בקלות. נוכל להגדיר נקודת עצירה שם, וכך להבין מה ההקצאות שעושה התוכנית.

[2] למתקדמים: גודל ההקצאה הוא הפרמטר השלישי של RtlAllocateHeap, ובכניסה לפונקציה הוא נמצא בהיסט של 12 בתים מראש המחסנית. נוכל לקבל את ערכו מתוך נקודת העצירה באמצעות הביטוי dwo(@esp+0n12).

ראשית, נבטל את נקודת העצירה הקיימת באמצעות הפקודה bc *, ואז נגדיר נקודת עצירה חדשה ב-malloc, שהפעם גם מקבלת פרמטרים נוספים שידפיסו עבורנו את גודל ההקצאה:

```
0:001> bp WinDbgIntro!malloc "?? nSize"
0:001> g
unsigned int 0x222
eax=00000222 ebx=00802308 ecx=007ae6a0 edx=00000218 esi=007afa70 edi=007ae5a4
eip=00c024a0 esp=007ae490 ebp=007ae4a8 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
WinDbgIntro!malloc:
00c024a0 55              push    ebp
```

כעת כאשר אנו עוצרים בנקודת העצירה, ראשית מודפס ערך הפרמטר nSize ולאחר מכן הפרטים הרגילים. בעצם, אין כל כך צורך יותר לעצור בנקודת העצירה, אם אנחנו רק רוצים סטטיסטיקה של גדלי ההקצאות. לכן נוכל להגדיר את נקודת העצירה מחדש כך:

```
0:001> bp WinDbgIntro!malloc "?? nSize; g"
breakpoint 0 redefined
0:001> g
unsigned int 0x241
unsigned int 0x34d
unsigned int 0x24f
unsigned int 0x227
```

כעת, כאשר התוכנית רצה היא מדפיסה באופן אוטומטי את גודל בקשות ההקצאה שמגיעות ל-malloc. אבל אולי נרצה אפילו יותר מזה. למשל, נראה שחלק מההקצאות הן קטנות יחסית ואולי לא מעניינות אותנו. מה אם אנחנו רוצים להציג את מחסנית הקריאות ולעצור את התוכנית רק כאשר מבצעים הקצאה בגודל מסוים, או בגודל שעובר סף מסוים שנקבע? לשם כך אנחנו צריכים אופרטור חדש, if., שמאפשר להוסיף משפטי תנאי לפקודות WinDbg. וכך נוכל להגדיר נקודת עצירה מותנית לפי גודל:

```
0:001> bp WinDbgIntro!malloc "r? $t0 = nSize; .if (@$t0 > 0n300) { .printf
\"Allocating %d bytes\", @$t0; kpn 5 } .else { g }"
breakpoint 0 redefined
```

פקודה זו כבר די מורכבת וכדאי להסתכל עליה בחלקים. ראשית:

```
r? $t0 = nSize;
```

החלק הזה שם את הערך של הפרמטר nSize של הפונקציה malloc במשתנה $t0. WinDbg מעמיד לרשותנו אוסף של 20 משתנים לשימושים זמניים, ששמותיהם $t0, $t1, …, $t19.

```
.if (@$t0 > 0n300) …
```

זהו משפט התנאי שלנו, הבודק שערכו של המשתנה $t0 גדול מקבוע כלשהו שהחלטנו עליו (כדי לקרוא את ערכו של המשתנה יש להוסיף לפניו את התחילית @). הקידומת 0n משמעותה שהמספר ניתן בבסיס עשרוני (ברירת המחדל של WinDbg היא בסיס 16).

```
.printf \"Allocating %d bytes\", @$t0;
```

זוהי הדפסה למסך ששוב משתמשת במשתנה $t0.

```
kpn 5
```

זוהי הצגה של מחסנית הקריאות עם עומק מקסימלי של 5 (פשוט כדי שהפלט יהיה קצר ככל האפשר אבל עדיין יאפשר לנו להבין מי קרא ל-malloc וביקש את הקצאת הזיכרון).

כעת כאשר נריץ את התוכנית והיא תבצע הקצאה גדולה מ-300 בתים, התוכנית תעצור ותדפיס את גודל ההקצאה ואת מחסנית הקריאות:

```
0:001> g
Allocating 887 bytes  # ChildEBP RetAddr
00 007ae48c 00be05cf WinDbgIntro!malloc(unsigned int nSize = 0x377)
[f:\dd\vctools\crt\crtw32\misc\dbgmalloc.c @ 55]
01 007ae4a8 00b95fdc WinDbgIntro!operator new(unsigned int size = 0x377)+0xf
[f:\dd\vctools\crt\crtw32\heap\new.cpp @ 59]
02 007ae4b4 00b8e5bc WinDbgIntro!operator new[](unsigned int count = 0x377)+0xc
[f:\dd\vctools\crt\crtw32\stdcpp\newaop.cpp @ 6]
03 007ae5a4 00b90c00 WinDbgIntro!DynamicArray::DynamicArray(unsigned int len = 0x377)+0x2c
[d:\development\scratch\windbgintro\windbgintro\main.cpp @ 13]
04 007afa68 00b8c73c WinDbgIntro!AllocatingThread(void)+0x90
[d:\development\scratch\windbgintro\windbgintro\main.cpp @ 32]
eax=00000377 ebx=00802308 ecx=007ae6a0 edx=0000036d esi=007afa70 edi=007ae5a4
eip=00c024a0 esp=007ae490 ebp=007ae4a8 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
WinDbgIntro!malloc:
00c024a0 55              push    ebp
```

היריעה קצרה מכדי לדון בפקודות שליטה מורכבות יותר כגון for., while., ו-foreach., אבל תוכלו לקרוא עליהן במאמר שלי על אוטומציה של WinDbg שהופיע בגיליון ה-46.

סיימנו לפעם הזו! כדי לעצור את ה-debugger ולצאת מהתוכנית, השתמשו בפקודה q. לחלופין, אם ברצונכם שהתוכנית תמשיך לרוץ גם לאחר ש-WinDbg מתנתק ממנה, השתמשו בפקודה qd.

סיכום

במאמר זה ראינו כיצד להשתמש ב-WinDbg לניתוח תהליכים חיים. ראינו איך להציג את רשימת החוטים, לעבור בין חוטים ופונקציות, להציג ערכים של משתנים מקומיים וגלובאליים, להגדיר נקודות עצירה, ולהציג ידיות של מערכת ההפעלה. זהו רק קצה הקרחון של יכולותיו של WinDbg. במידה ויתעורר עניין סביב הנושא, אוכל במאמרי המשך לסקור ניתוח של dump files, פקודות נוספות, ופקודות הרלוונטיות רק ל-kernel mode.

למידע נוסף על WinDbg אני ממליץ על הספרים Advanced Windows Debugging של Mario Hewardt ו-Inside Windows Debugging של Tareek Soulami. שני הספרים מביאים דוגמאות ותרחישים אמיתיים לשימוש בפקודות מתקדמות של WinDbg ובסקריפטים פשוטים ומורכבים.

על המחבר

סשה גולדשטיין הוא ה-CTO של קבוצת סלע, חברת ייעוץ, הדרכה ומיקור חוץ בינלאומית עם מטה בישראל. סשה אוהב לנבור בקרביים של Windows וה-CLR, ומתמחה בניפוי שגיאות ומערכות בעלות ביצועים גבוהים. סשה הוא מחבר הספר Pro .NET Performance., ובין היתר מלמד במכללת סלע קורסים בנושא .NET Debugging ו-Windows Internals. בזמנו הפנוי, סשה כותב בלוג על נושאי פיתוח שונים. אפשר למצוא אותו גם בטוויטר.