Digital Whisper
Technologic Papers

מימוש מנגנוני סנכרון ב-Windows ומעבר להם

מאת: סשה גולדשטיין

הקדמה

מבוא למנגנוני סנכרון ב-Windows

מנגנוני סנכרון ב-Windows משמשים למטרות רבות מאוד. צורך אחד ברור של מערכת ההפעלה הוא להחצין כלפי תוכניות user-mode את מנגנוני הסנכרון המוכרים והמתועדים, המאפשרים מניעה הדדית (Mutex, Critical Section), המתנה וקבלת הודעה על שינויים (Event), המתנה מותנית (Condition Variable), ועוד.

```
HANDLE hMutex;
DWORD WINAPI FirstThread(LPVOID)
{
    DoSomePrivateWork(1);
    WaitForSingleObject(hMutex, INFINITE);
    ModifyImportantSharedData();
    ReleaseMutex(hMutex);
}
DWORD WINAPI SecondThread(LPVOID)
{
    DoSomePrivateWork(2);
    WaitForSingleObject(hMutex, INFINITE);
    ModifyImportantSharedData();
    ReleaseMutex(hMutex);
}
```

(מקטע של תוכנית המשתמשת ב-Mutex על מנת לבצע מניעה הדדית בין שני חוטים.)

אולם גם ה-Kernel זקוק למנגנוני סנכרון לצרכיו הפנימיים - הן כדי לממש את מנגנוני הסנכרון האחרים, והן כדי לאפשר מניעה הדדית המגנה על קטעי קוד או מידע שניגשים אליהם בקצב גבוה במיוחד של מיליוני פעמים בשניה. במקצת מהמקרים ניתן להגן על מידע משותף גם ללא ביצוע מניעה הדדית ע"י שימוש בהוראות מיוחדות של המעבד, אולם מקרי הקצה והשימוש בהוראות אלה אינם טריוויאליים.

במאמר זה נסקור את המימוש של מנגנוני הסנכרון ב-Windows המוחצנים לתוכניות, את חלק ממנגנוני הסנכרון הנמצאים בשימוש של ה-Kernel, את הרעיונות הבסיסיים של "סנכרון ללא סנכרון", ובדרך נתבונן בקצרה באבחון של בעיות נפוצות הקשורות בסנכרון.

אזהרה: חלק מהפרטים שנראה אינם מתועדים ועשויים להשתנות מגרסה לגרסה של מערכת ההפעלה, אך בכל זאת, הרעיונות הבסיסיים רלוונטיים ונכונים כמו לפני עשר שנים ב-Windows XP כך גם היום ב-Windows 7. אם כך, כיצד נגלה את הפרטים החבויים והלא מתועדים?

ראשית, מארק רוסינוביץ' ודיוויד סלומון כתבו כבר חמש מהדורות של הספר המצוין Windows Internals שניתן למצוא בו פרטים רבים על האופן שבו המערכת בנויה ומתנהגת, לרבות חלק מהפרטים על מנגנוני סנכרון שנדון בהם בהמשך. שנית, באמצעות Kernel Debugger ו-Debugging Symbols מתאימים ניתן לנבור בנבכי המערכת, בעיקר עם IDA פתוח על ה-Checked Build של מערכת ההפעלה, המקומפל ללא אופטימיזציות[1]. (את ה-Checked Build של המערכת ניתן למצוא ב-WDK, או להוריד מאתר מיקרוסופט אם יש לכם מנוי MSDN.)

מימוש מנגנוני סנכרון המוחצנים לתוכניות המשתמש

אובייקטי הסנכרון של מערכת ההפעלה המוחצנים לתוכניות משתמש נגישים באמצעות HANDLE, שהוא אינדקס לטבלה[2] הנשמרת במבנה נתונים של המערכת המשויך לתהליך ספציפי. תוכניות משתמש לא יכולות לגשת לאובייקטי הסנכרון ישירות - כלומר, התוכנית לא מקבלת גישה ישירה למבנה הנתונים שמחזיק מידע על Mutex או על Event - אלא מעבירה HANDLE לשירותים של מערכת ההפעלה, כמו WaitForSingleObject ו-ReleaseMutex.

כל אובייקטי הסנכרון המוחצנים לתוכניות משתמש ממומשים במערכת ההפעלה באמצעות מבנה נתונים שנקרא DISPATCHER_HEADER. האחריות על ניהולו מוטלת על ה-Kernel, שמאחסן בו פרטים על סוג האובייקט, האם הוא Signaled או לא, ועוד (נאמר שאובייקט סנכרון הוא Signaled - מסומן - אם כאשר חוט יבצע עליו פעולה הדורשת המתנה, החוט לא יצטרך לחכות אלא יקבל מיד את אובייקט הסנכרון; סימון האובייקט נעשה בצורה ספציפית לסוגו, ומתועד ב-MSDN).

[1] אין במשפט זה משום עידוד לביצוע Reverse Engineering לקוד של מערכת ההפעלה, מה שיכול להיות אסור על פי החוקים הרלוונטיים או הסכם המשתמש (EULA).

[2] פרטי המימוש של הטבלאות בזיכרון המתרגמות HANDLE לאובייקט של מערכת ההפעלה חורגים ראויים למאמר נפרד.

מבנה הנתונים הזה משתנה בין גרסאות של מערכת ההפעלה, וב-Windows XP SP2 נראה כך:

```
0: kd> dt nt!_DISPATCHER_HEADER
   +0x000 Type             : UChar
   +0x001 Absolute         : UChar
   +0x002 Size             : UChar
   +0x003 Inserted         : UChar
   +0x004 SignalState      : Int4B
   +0x008 WaitListHead     : _LIST_ENTRY
```

(מבנה הנתונים DISPATCHER_HEADER המתאר אובייקט סנכרון.)

השדות המשמעותיים לדיון הם שדה ה-Type שמאפיין את סוג האובייקט (Mutex, Event וכו'), שדה ה-SignalState המציין האם האובייקט מסומן, ושדה ה-WaitListHead המכיל אינפורמציה על חוטים הממתינים לאובייקט הסנכרון. כש-Windows מסמנת אובייקט סנכרון מסוים, למשל כתוצאה מקריאת API, המידע הנ"ל מאפשר לה להעיר את החוטים שממתינים לסימון האובייקט. הדבר מתרחש בפונקציה KiWaitTest שקוראת ל-KiUnwaitThread במידה ואכן יש להעיר חוט כלשהו[3].

כמובן, ישנו גם הכיוון ההפוך - המערכת זקוקה למקום שבו יאוחסן המידע על אובייקטי הסנכרון שחוט מסוים מחכה שיסומנו. מידע זה מאוחסן במבנה נתונים נוסף שנקרא KWAIT_BLOCK, המקשר בין חוט לבין אובייקט סנכרון. מבנה נתונים זה נראה כך:

```
0: kd> dt nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY
   +0x008 Thread           : Ptr32 _KTHREAD
   +0x00c Object           : Ptr32 Void
   +0x010 NextWaitBlock    : Ptr32 _KWAIT_BLOCK
   +0x014 WaitKey          : Uint2B
   +0x016 WaitType         : Uint2B
```

(מבנה הנתונים KWAIT_BLOCK המתאר חוט הממתין לאובייקט סנכרון.)

השדה Thread מצביע לחוט שמחכה לאובייקט הסנכרון, השדה Object מצביע לאובייקט הסנכרון, השדה NextWaitBlock מצביע ל-KWAIT_BLOCK הבא שהחוט מחכה לו, השדה WaitListEntry מצביע ל-KWAIT_BLOCK של החוט הבא שמחכה על אובייקט הסנכרון, והשדות WaitKey ו-WaitType מחזיקים מידע על סוג ההמתנה - האינדקס של אובייקט הסנכרון במערך ה-HANDLE-ים המועבר ל-WaitForMultipleObjects והאם לחכות לכל האובייקטים במערך או רק לאחד מהם. במבנה הנתונים הפרטי של כל חוט (KTHREAD) שמורה רשימה מקושרת של KWAIT_BLOCK-ים שהוא ממתין להם.

[3] ב-Windows 7 הפונקציה KiWaitTest חולקה ל-2: KiSignalNotificationObject ו-KiSignalSynchronizationObject.

המשמעות היא שבהינתן חוט, נוכל לדעת בקלות (יחסית) לאילו אובייקטי סנכרון הוא ממתין. על מנת לעשות זאת נעקוב אחר הצעדים הבאים[4]:

1. בהינתן הכתובת של ה-KTHREAD המשויך לחוט, נמצא בתוכו את השדה WaitBlockList המכיל מצביע ל-KWAIT_BLOCK הראשון שעליו החוט ממתין.
2. מה-KWAIT_BLOCK נוציא את הכתובת של אובייקט הסנכרון, ומשם גם את הסוג שלו, ישירות או באמצעות פקודת דיבאגר כמו !object.
3. מאותו KWAIT_BLOCK נוציא גם את הכתובת של ה-KWAIT_BLOCK הבא, אם יש כזה, כדי לראות לאילו אובייקטים נוספים החוט מחכה.

```
0: kd> dt nt!_KTHREAD 0x892a64f8
   +0x000 Header           : _DISPATCHER_HEADER
   ...
   +0x05c WaitBlockList    : 0x892a6568 _KWAIT_BLOCK
   ...
   +0x1b8 FreezeCount      : 0 ''
   +0x1b9 SuspendCount     : 0 ''
   +0x1ba IdealProcessor   : 0x1 ''
   +0x1bb DisableBoost     : 0 ''

0: kd> dt nt!_KWAIT_BLOCK 0x892a6568
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x897ade48 - 0x897ade48 ]
   +0x008 Thread           : 0x892a64f8 _KTHREAD
   +0x00c Object           : 0x897ade40
   +0x010 NextWaitBlock    : 0x892a6568 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1

0: kd> dt nt!_DISPATCHER_HEADER 0x897ade40
   +0x000 Type             : 0x1 ''
   +0x001 Absolute         : 0xc3 ''
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0x89 ''
   +0x004 SignalState      : 0
   +0x008 WaitListHead     : _LIST_ENTRY [ 0x892a6568 - 0x892a6568 ]

0: kd> !object 0x897ade40
Object: 897ade40  Type: (898c42f8) Event
    ObjectHeader: 897ade28 (old version)
    HandleCount: 1  PointerCount: 2
```

(דוגמא מתוך הדיבאגר שבאמצעותה גילינו לאיזה אובייקט סנכרון חוט מסוים מחכה.)

[4] לקורא המדייק: בימי KD ו-WinDbg אין צורך לבצע פעולות אלה באופן ידני כל כך. למשל, בהינתן KTHREAD ניתן להעבירו לפקודה !thread המציגה מיד את אובייקטי הסנכרון שהחוט מחכה להם וגם את הסוג שלהם.

אם ברצוננו לבצע אבחון של חֶבֶק (Deadlock) שמעורבים בו אובייקטים וחוטים רבים, אנו זקוקים לפיסת מידע נוספת. עלינו לדעת מי החוט **המחזיק** אובייקט סנכרון מסוים, ולא רק מי החוטים הממתינים לו. לא תמיד מידע מסוג זה קיים בכלל - למשל, ל-Mutex אכן יכול להיות חוט המחזיק אותו עד שיקרא ל-ReleaseMutex, אבל ל-Event לא בהכרח יש חוט ה"אחראי" לבצע SetEvent בצורה בלעדית (לאיזו מטרה המערכת צריכה מידע מסוג זה? למשל, אסור לחוט לבצע ReleaseMutex על Mutex שלא שייך לו.).

מידע ספציפי על שייכות של אובייקט סנכרון נשמר אף הוא במבנה הנתונים של אותו אובייקט. אלא שכעת השדות של DISPATCHER_HEADER לא מספיקים, ואכן לרוב אובייקטי הסנכרון מבני נתונים ספציפיים עבורם המחזיקים מידע נוסף. למשל, עבור Mutex זהו מבנה הנתונים KMUTANT; השדה OwnerThread זהו מצביע לחוט המחזיק את ה-Mutex.

```
0: kd> dt nt!_KMUTANT
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListEntry  : _LIST_ENTRY
   +0x018 OwnerThread      : Ptr32 _KTHREAD
   +0x01c Abandoned        : UChar
   +0x01d ApcDisable       : UChar
```

(מבנה הנתונים KMUTANT המתאר Mutex.)

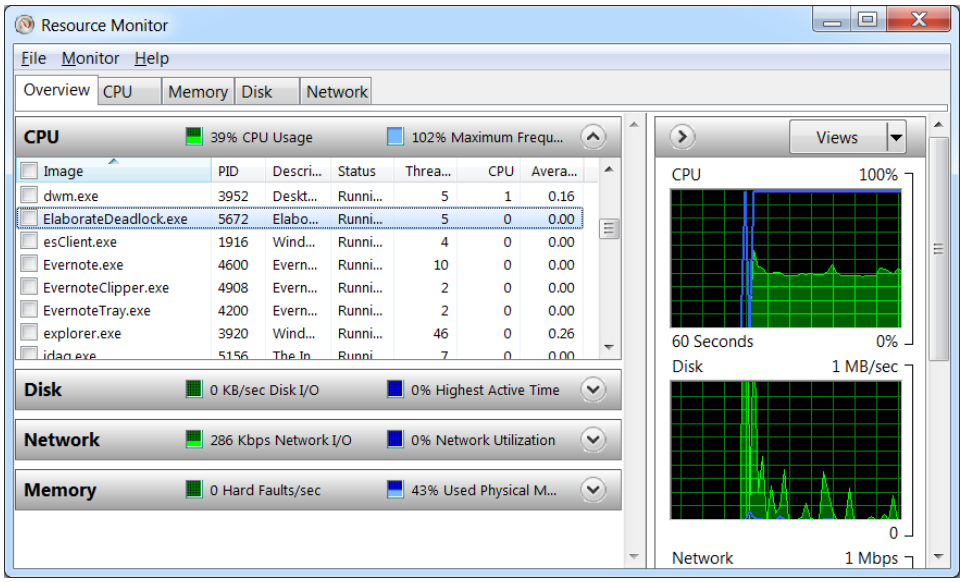
כעת עומד לרשותנו תהליך דטרמיניסטי לאבחון חבק שבו משתתפים חוטים ו-Mutex-ים בלבד: לכל חוט נבצע את התהליך המתואר לעיל כדי למצוא את אובייקטי הסנכרון שהוא ממתין להם, ולכל אובייקט סנכרון נתבונן במבנה הנתונים KMUTANT ונגלה מיהו החוט הבא ברשימה שעלינו להסתכל עליו. תוצאת הניתוח הנ"ל מובילה לשרשרת המתנה, כמו למשל השרשרת הבאה. מעגל בשרשרת הוא תנאי הכרחי לחבק, ומבנה השרשרת משרה הבנה מלאה של אופי הבעיה.

Thread	waits	**Mutex**	owned	**Thread**	waits	**Mutex**	owned	**Thread**	waits	**Mutex**
2106	for	“A”	by	1204	for	“B”	by	424	for	“A”

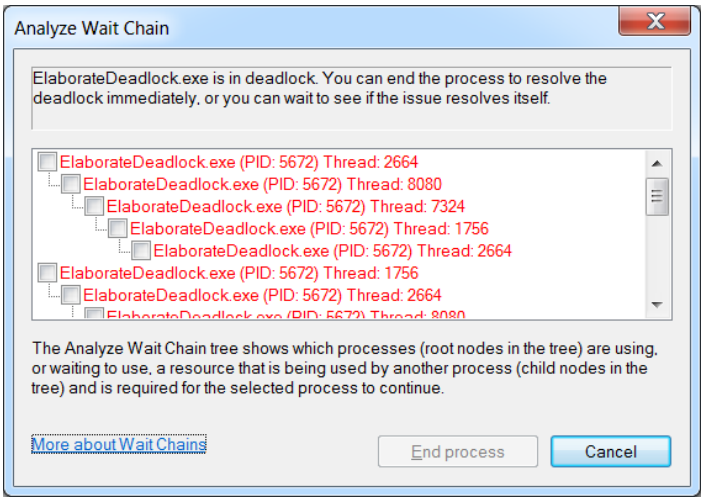
(שרשרת המתנה שבה שלושה חוטים ושלושה Mutex-ים, וגם מעגל המצביע על חבק)

אלה הנתונים שמערכת ההפעלה מחזיקה אודות אובייקטי סנכרון והחוטים שממתינים להם. כאשר מדובר באובייקטים רבים ובחוטים רבים, מעקב ידני אחר הנתונים יכול להיות קשה למדי. החל מ- Windows Vista, קיים API בשם Wait Chain Traversal (WCT) שאפשר להשתמש בו כדי לקבל שרשרת המתנה מוכנה עבור חוט מסוים, המכילה חוליות מסוגים רבים (Mutex, Critical Section, Window Message ועוד). ישנו כלי מובנה של מערכת ההפעלה המשתמש בשירות זה לניתוח חבק, וזהו ה- Resource

Monitor. למרבה הצער, הכלי לא מדווח את שמות אובייקטי הסנכרון המעורבים, אלא רק את החוטים ויחסי ההמתנה ביניהם.



(החלון הראשי של Resource Monitor.)



(שרשרת ההמתנה של תהליך הנמצא במצב של חבק.)

אם תרצו להשתמש בשירות שרשראות ההמתנה של המערכת מתוך התוכנית שלכם, או מתוך הדיבאגר, תוכלו להיעזר בתיעוד של מיקרוסופט וכן בהרחבה לדיבאגר שכתבתי ופרסמתי יחד עם קוד המקור שלה.

```
0:001> .load D:\Development\DebuggingExtensions\Release\wct.dll
0:001> !wct_thread
>>> Begin wait chain for thread 4784:6016 with 5 nodes
    Thread [4784:6016:blocked WT=7287568] ==>
    Mutex [\Sessions\1\BaseNamedObjects\SecondMutex] ==>
    Thread [4784:5924:blocked WT=7286947] ==>
    Mutex [\Sessions\1\BaseNamedObjects\FirstMutex] ==>
    Thread [4784:6016:blocked WT=7287568] ==>
DEADLOCK FOUND
>>> End wait chain for thread 4784:6016
```

(דוגמת הרצה של הרחבה לדיבאגר המשתמשת ב-WCT ומדפיסה את התוצאות.)

מנגנוני סנכרון הנמצאים בשימוש של ה-Kernel

ה-Kernel עשוי להזדקק בעצמו לסנכרון כשהוא מממש את אובייקטי הסנכרון שתיארנו קודם. למשל, כאשר במערכת מרובת מעבדים (שבה בו-זמנית מעובדים מספר חוטים) מבצעים קריאה לפונקציה כמו WaitForSingleObject, ה-Kernel צריך לסנכרן את הגישה למבני הנתונים KWAIT_BLOCK, KTHREAD ואחרים שראינו קודם. יתר כן - כשהוא מעביר חוט ממצב ריצה למצב המתנה, ה-Kernel מעביר את אובייקט ה-KTHREAD בין תורים פנימיים - שגם הם, מטבעם, דורשים סנכרון.

כיצד איפוא יכול ה-Kernel לדאוג לסנכרון כאשר הוא-עצמו מממש את מנגנוני הסנכרון? הברירה היחידה היא להשתמש במנגנוני סנכרון אחרים, שאינם נסמכים על מערכת ההפעלה אלא על שירותי חומרה המגנים על איזורי זיכרון מפני שינויים לא-בטוחים. לפני שנוכל לדון בפרטים, נצטרך הקדמה קצרה בענייני שיתוף מידע בין מעבדים.

מערכות המחשב ש-Windows פועלת עליהן היום עובדות תחת הנחות המכונות ccNUMA - זיכרון שניתן לגישה מכל המעבדים וצריך להימצא במצב קונסיסטנטי כאשר מספר מעבדים מבצעים עליו שינויים בו-זמנית. החומרה מבטיחה למערכת ההפעלה שפעולות כמו קריאה או כתיבה של מילת זיכרון (למשל, 32 ביט) היא פעולה אטומית. למשל, אם שני מעבדים מנסים לכתוב את המילה, אזי מובטח שאחד מהם "ינצח", ולא ייתכן שהתוצאה בזיכרון היא "ערבוב" או "איחוד" של הביטים משני העדכונים.

למעט הפעולה הפרימיטיבית של כתיבה או קריאת מילה, ארכיטקטורות חומרה מודרניות מאפשרות גם פעולות מורכבות יותר שעדיין מתבצעות בצורה אטומית, כגון קידום מונה בצורה אטומית. פעולות אלה

קריטיות למימוש הסנכרון ב-Kernel, ולמעשה גם למערכות סנכרון הפועלות בקצב גבוה (למשל, מנגנון מניעה הדדית שמתבצעות עליו מיליוני נעילות בשניה). אחת הפעולות הבסיסיות הנ"ל היא InterlockedCompareExchange, הנקראת לפעמים Compare-and-Swap (CAS), והמאפשרת לכתוב ערך לזיכרון בצורה מותנית:

```
//Pseudo-code - in actuality, this is a single atomic instruction
LONG InterlockedCompareExchange(LONG* Loc, LONG Exchg, LONG Cmpnd)
{
    LONG OldVal = *Loc;
    if (*Loc == Cmpnd)
        *Loc = Exchg;
    return OldVal;
}
```

(פסאודו-קוד של InterlockedCompareExchange; בפועל, הפונקציה ממומשת ע"י פקודת מעבד בודדת, למשל LOCK CMPXCHG במעבדי 32-ביט של אינטל.)

העובדה שפעולה מסוג זה מתבצעת בצורה אטומית אינה טריוויאלית כלל, ובמעבדי אינטל דורשת נעילה מלאה של ה-Bus המוביל לזיכרון הראשי (כלומר, כאשר מעבד מסוים מבצע CAS, מעבדים אחרים אינם יכולים לגשת לזיכרון כלל). אולם בהינתן פעולה זו, ניתן לבנות מנגנוני סנכרון ולהגן על נתונים ללא צורך באבסטרקציות של מערכת ההפעלה כמו DISPATCHER_HEADER.

מנגנון הסנכרון המרכזי שה-Kernel משתמש בו, בין היתר כדי להגן על מבני הנתונים של אובייקטי סנכרון "סטנדרטיים", נקרא Spinlock. זהו מנעול, כלומר מנגנון המאפשר מניעה הדדית, הממומש באמצעות CAS[5].

ניתן להשתמש בו ממש כמו ב-Mutex, כאשר כדי לתפוס את המנעול קוראים ל-KeAcquireSpinlock וכדי לשחררו קוראים ל-KeReleaseSpinlock:

```
//Pseudo-code - the actual implementation may be different
void KeAcquireSpinlock(LONG* Spinlock)
{
    //Try to swap the spinlock bit from 0 to 1, atomically.
    //If an attempt fails, try again until success.
    while(1)
    {
```

[5] למעשה, במעבדי אינטל 32-ביט למשל, מערכת ההפעלה משתמשת בפקודת המעבד LOCK BTS על מנת לממש Spinlock. אין הבדל של ממש בין פקודה זו לבין המימוש של CAS, אלא שפקודה זו פועלת על ביט אחד בלבד.

```
        if (InterlockedCompareExchange(Spinlock, 1, 0) == 0)
            break;
    }
}
void KeReleaseSpinlock(LONG* Spinlock)
{
    *Spinlock = 0;
    MemoryBarrier();
}
```

(פסאודו-קוד של מימוש Spinlock; במציאות, המימוש יכול להשתמש בפקודות אחרות או לבצע אופטימיזציות מסוימות שלא מוצגות כאן)

הרעיון הכללי פשוט יחסית - משתמשים בביט אחד של מידע כדי לסמן האם המנעול תפוס או לא. כדי לתפוס את המנעול, מנסים להחליף את הביט מ-0 ל-1 - המשמעות של הצלחה בהחלפה כזאת היא שהמנעול היה פנוי, ואז תפסנו אותו. אם ההחלפה נכשלת, מנסים שוב ושוב עד שמצליחים. מספר הניסיונות אינו חסום, ולכאורה ייתכן שחוט מסוים (מעבד מסוים) יכול לחכות זמן לא חסום; אף על פי כן, במציאות זמן ההמתנה נוטה להתפזר בצורה אחידה על פני כל המעבדים.
במימוש האמיתי של Spinlock ישנו סיבוך נוסף, הנובע מהאופן שבו פסיקות (Interrupts) מתועדפות ע"י מערכת ההפעלה. לכל Spinlock משויכת רמת עדיפות פסיקות (IRQL) שחובה על המעבד להיות בה על מנת שיוכל לתפוס את המנעול. נושא הטיפול בפסיקות ראוי למאמר בפני עצמו.

בכל אופן, נשים לב להבדל עצום בין Spinlock לבין מנגנוני סנכרון קלאסיים של מערכת ההפעלה: כשחוט רוצה לתפוס Spinlock, הוא "מבלה" את זמנו בלולאה ששורפת זמן מעבד, עד שהוא מצליח לתפוס את המנעול. כלומר, כאשר המנעול תפוס וחוטים אחרים "מתחרים" על המנעול, הם ממשיכים לרוץ על המעבד. לעומת זאת, כשחוט רוצה לתפוס Mutex או מנגנון סנכרון קלאסי אחר ונאלץ לחכות, מערכת ההפעלה מעבירה את החוט לרשימת המתנה ומריצה חוט אחר על אותו מעבד. מחד, "לשרוף CPU" נשמע כמו רעיון גרוע; מאידך, הזמן שלוקח למערכת ההפעלה לבצע החלפת חוטים אינו זניח בכלל, וכאשר התחרות על המנעול מגיעה לקצב של מיליוני פעולות בשניה, בהחלט יש טעם לשקול Spinlock על פני סנכרון קלאסי גם באפליקציות משתמש.

אחד החסרונות של המימוש לעיל הוא חוסר הוגנות בין מעבדים[6]. ייתכן שמעבד 2 התחיל לבצע את הלולאה מוקדם יותר ממעבד 2, אבל במקרה מעבד 2 הוא הראשון לראות את ה-Spinlock מקבל את הערך 0, ולכן תופס אותו ראשון. בנוסף, כאשר מספר מעבדים ממתינים ל-Spinlock ודוגמים כולם את איזור הזיכרון המשותף שבו המנעול נמצא, שחרור המנעול (כתיבה למשתנה של המנעול) גורמת לאינוולידציה של המטמון בכל המעבדים הממתינים, פעולה יקרה יחסית. מסיבות אלה ואחרות, החל מ-Windows XP, כמעט שלא משתמשים במערכת במימוש לעיל, אלא באופטימיזציה המכונה Queued Spinlock, שהיא מורכבת יותר. מעבד שמבקש לקבל מנעול מסוג זה מכניס את עצמו לרשימת המתנה, וכל מעבד ברשימת ההמתנה דוגם משתנה פרטי (ולא את המשתנה המשותף של המנעול). המעבד שמשחרר מנעול כזה "מעיר" את המעבד הראשון ברשימת ההמתנה ע"י כתיבת ערך מתאים למשתנה הפרטי שלו. כך מתקבלת הוגנות בקבלת המנעול, ואין צורך באינוולידציית מטמון במעבדים שלא מקבלים את המנעול.

"סנכרון ללא סנכרון"

בסקירתנו את הסנכרון ב-Kernel ראינו כיצד ניתן לממש מנעול באמצעות CAS. מתבקשת השאלה האם ניתן לממש באמצעות פעולה פרימיטיבית זאת אלגוריתמים או מבני נתונים מורכבים יותר שאינם נדרשים לסנכרון כלל - אפילו לא ל-Spinlock.

בתור חימום, נסו לממש באמצעות CAS פונקציה הכופלת ערכי שני משתנים, מחלקת את התוצאה בערכו של משתנה שלישי, ומחזירה את התוצאה למשתנה הראשון:

```
A = (A*B)/C
```

באופן עקרוני, פונקציה מסוג זה דורשת סנכרון בגישה למשתנה A, שכן ייתכן שתוך כדי החישוב (לאחר שקראנו את ערכו של A) מעבד נוסף מנסה לבצע את אותה פעולה, וכך אחד העדכונים של A הולך לאיבוד:

CPU 1	CPU 2
Read A	Read A
Read B	Read B

[6] כדאי לציין בהזדמנות זו שהוגנות בין מעבדים או בין חוטים אינה תמיד תכונה רצויה. למשל, מנגנוני סנכרון הוגנים נוטים להוביל לתופעות כמו Lock Convoy, שגורמות לפגיעה נואשת בביצועים. מכאן שמערכות הפעלה מודרניות לעתים נמנעות בכוונה ממימוש מנגנוני סנכרון הוגנים, ונוקטות במנגנוני תיעדוף או אף משאירות את ה"מנצח" ליד הגורל.

Temp1 = A*B	Temp2 = A*B
Read C	Read C
Temp1 = Temp1/C	Temp2 = Temp2/C
A = Temp2	A = Temp2

(עדכון הולך לאיבוד מאחר ששני המעבדים קוראים את A וכותבים את A במקביל.)

והנה פתרון אפשרי המשתמש ב-CAS, שרוחו דומה מאוד למימוש של Spinlock שראינו קודם:

```
void MultiplyAndDivide(LONG* A, LONG B, LONG C)
{
    LONG OldA, NewA;
    do
    {
        OldA = *A;
        NewA = (OldA*B)/C;
    }
    while(InterlockedCompareExchange(A, NewA, OldA) != OldA);
}
```

ביצוע הפעולה A = (A*B)/C באופן אטומי באמצעות CAS; שימו לב להקפדה לקרוא את A רק פעם אחת במהלך כל איטרציה, ולביצוע החישוב באמצעות עותק מקומי (OldA)

לאחר החימום, ברור שיש עוד פעולות רבות שניתן לבצע בתבנית דומה - נבצע את החישוב על עותק של הנתונים שעומדים לשנות, וננסה להחליף את הנתון בצורה אטומית בגרסתו החדשה. ישנם גם רעיונות מתוחכמים יותר המאפשרים לשלב מספר עדכונים ל"חבילה" של CAS-ים על מספר אזורי זיכרון (MCAS). נגביל את עצמנו כרגע לדוגמה פשוטה יחסית - נממש מחסנית (Stack) ללא נעילות קלאסיות אבל בצורה שתאפשר להשתמש בה ממספר חוטים (על מספר מעבדים) ללא צורך בסנכרון נוסף.

הרעיון דומה מאוד למה שעשינו כבר. המחסנית תמומש באמצעות רשימה מקושרת חד-כיוונית. הכנסה ושליפה מהמחסנית הן פעולות על ראש המחסנית (ראש הרשימה המקושרת), והעדכון הנדרש עומד בתנאים של CAS. מכאן הקוד של הכנסה למחסנית - את השליפה אשאיר לכם כתרגיל (מימוש נגיש יחסית עם הסברים נוספים אפשר למצוא כאן):

```
typedef struct _NODE
{
    struct _NODE* Next;
    void* Item;
} NODE;

NODE* Head = NULL;

void Push(void* Item)
{
    NODE* New = (NODE*)malloc(sizeof(NODE));
    New->Item = Item;
    while(1)
    {
        NODE* OldHead = Head;
        New->Next = OldHead;
        if (InterlockedCompareExchange(&Head, New, OldHead) == OldHead)
            break;
    }
}
```

(מימוש חלקי של הכנסה למחסנית ללא סנכרון ע"י שימוש ב-CAS)

גם ה-Kernel משתמש ברעיונות דומים של "סנכרון ללא סנכרון" לצרכים מסוימים. מערכת ההפעלה אפילו מייצאת לתוכניות משתמש פונקציונאליות של רשימה מקושרת חד-כיוונית שלא דורשת סנכרון (SList). למעשה, ככל שמספר המעבדים עולה, עלות הסנכרון הקלאסי נעשית בלתי-נסבלת, ואין מנוס משימוש ב"טריקים" מקוריים על מנת להימנע מהמתנה כמעט בכל מחיר.

אזהרת מסע לגבי כתיבת קוד ללא סנכרון

חשוב מאוד שלא יווצר הרושם כאילו זה קל לכתוב קוד ללא סנכרון באמצעות שימוש ב-CAS, או בכלל להימנע ממנגנוני סנכרון קלאסיים. למעשה, מקרי הקצה והפינות שצריך להכיר הם רבים, ונוטים להיות שונים בין ארכיטקטורות חומרה שונות (למשל, x86 היא ארכיטקטורה מקלה הרבה יותר מאשר Itanium). אחד האתגרים נעוץ במודל הגישה לזיכרון של שפת התכנות ושל המעבד, כלומר ההבטחות שאנו מקבלים כמפתחים לגבי סדר הפעולות על הזיכרון.

לא ניכנס במסגרת זו לדיון מעמיק בנושא, אבל הנה דוגמה פשוטה:

Thread 1 (CPU 1)	Thread 2 (CPU 2)

`while (f == 0) ;`
 	
`printf("%d\n", x);`
	`x = 42;`
	
	`f = 1;`

(תוכנית תמימה המדגימה את הסכנות בעבודה עם זיכרון משותף ללא סנכרון.)

נניח שבהתחלה, `x = f = 0`. כאשר שני החוטים מתבצעים במקביל בשני המעבדים, הפלט יכול להיות 42 אבל הוא יכול גם להיות 0! הסיבה נעוצה בכך שכל מעבד[7] רשאי לבצע את הפקודות שלא לפי הסדר שבו הן כתובות בתוכנית, אלא אם כן הוא מזהה תלויות בין פקודות ובמקרה כזה לא יכול לשנות את הסדר. במקרה לעיל, מעבד 2 לא "מבין" שאסור לכתוב את המשתנה f לפני כתיבת המשתנה x, משום ש**מבחינתו** אין תלות בין שתי פקודות אלה. התלות נוצרת בעקבות העובדה שמעבד 1 משתמש באותם מקומות בזיכרון, ללא מנגנון סנכרון מתאים.

וכי במה יכול לעזור כאן מנגנון סנכרון? מנגנוני הסנכרון של מערכת ההפעלה, וגם פונקציות כמו InterlockedCompareExchange, דואגות להפעיל פקודה מיוחדת של המעבד[8] שיוצרת תלות מאולצת בין פקודות - גדר (fence) שפקודות לא יכולות לחצות אותה. במקרה לעיל, גדר בין שתי הפקודות של מעבד 2 הייתה גורמת לפלט להיות 42 באופן דטרמיניסטי.

המורכבות של הסיטואציה (וכאמור, התלות בארכיטקטורת מעבד ספציפית) גורמת לכך שרק מתכנתים בודדים מרגישים מספיק בנוח עם מודל הזיכרון של המעבד ושפת התכנות כדי לכתוב קוד כמו לעיל או להשתמש בגדרות.

[7] אגב, גם הקומפיילר רשאי לשנות סדר של פעולות בתוכנית, אולם זהו דיון נפרד. ברוב שפות התכנות ניתן לשכנע את הקומפיילר לא לבצע אופטימיזציות מסוג זה על משתנים מסוימים (למשל, באמצעות מילת המפתח volatile).

[8] ב-Windows ניתן להשתמש בפונקציה ()MemoryBarrier.

סיכום

מנגנוני הסנכרון של Windows הם עולם מרתק, ומנגנוני סנכרון בכלל - עוד יותר. במאמר זה נגענו רק במקצת מהנושאים הבוערים היום בתחום מנגנוני הסנכרון והסתכלנו רק על חלק ממה שיש למערכת ההפעלה להציע. אבחון בעיות הקשורות למנגנוני סנכרון, ובין היתר בעיות ביצועים וסקאלאביליות, רלוונטיים כמעט לכל מערכת שנכתבת היום ותרוץ במשך כמה שנים על מחשבים בעלי מספר גדל מעריכית של מעבדים.

לפרטים נוספים, אתם מוזמנים לעיין בספר Windows Internals ובקישורים שפוזרו לאורך המאמר.

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 ממחברי הספר Windows 7 for Developers,ובין היתר מלמד במכללת סלע קורסים בנושא Windows Internals. בזמנו הפנוי, סשה כותב בלוג על נושאי פיתוח שונים.