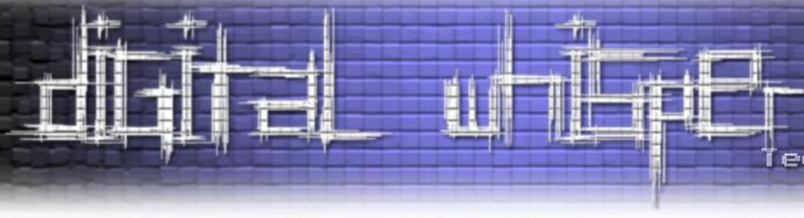


תזמון חוטים ב-Windows

מאת: ששה גולדשטיין

הקדמה

בפעם הקודמת הסתכלנו על המימוש של מנגנוני סנכרון ב-Windows. הצורך במנגנוני סנכרון עולה משימוש שמערכת ההפעלה עושה במספר מעבדים, ומהרצת מספר חוטים (Threads) על מעבד אחד או כמה מעבדים.

במאמר זה נסקור כיצד Windows מחליטה איזה חוט יש להריץ, על איזה מעבד יש להריץ אותו, כיצד ניתן להשפיע על החלטות אלה, ואילו אופטימיזציות ושיקולים משפיעים על ההחלטות האלה כתוצאה משינויי החומרה של השנים האחרונות.

מדוע בכלל להשתמש בחוטים?

מבלי להיכנס לפרטים שחורגים מגבולות המאמר, ניתן לומר שיש שלוש סיבות עיקריות לשימוש בחוטים הרלוונטיות הן בכתיבת תוכניות משתמש והן בפיתוח מערכת ההפעלה עצמה:

1. **ניצול מספר מעבדים** - חוט אחד יכול לרוץ על מעבד אחד בלבד בכל רגע נתון, ולכן על מנת להשתמש בצורה מיטבית במספר גדול של מעבדים יש ליצור מספר גדול של חוטים ולדאוג לספק להם עבודה בכל רגע. (כמובן, מטרתנו ברכישת מחשב עם מספר רב של מעבדים היא להביא אותם לניצולת של 100%, ולא להניח להם להתבטל).

2. **שילוב של פעולות קלט/פלט ופעולות חישוב** - בזמן שרכיב חומרה מסוים (למשל, דיסק קשיח) מטפל בבקשת קלט/פלט, המעבד פנוי ויכול להמשיך לבצע חישובים. תוכנית שמבצעת הן קלט/פלט והן חישובים יכולה להשתמש במספר חוטים. למשל, בעת שחוט מחכה לתוצאות של פעולת קלט/פלט, חוט אחר יכול לבצע חישובים ולהמשיך לנצל בצורה מיטבית את המשאבים.
3. **תגובתיות למשתמש** - שימוש במספר חוטים (אפילו במערכת בעלת מעבד אחד) מאפשר לתוכנית משתמש להישאר תגובתית, ע"י כך שאת הטיפול בפעולות המשתמש מבצעים בחוט אחד ואת הפעולות הדורשות עיבוד כבד מבצעים "ברקע", בחוט אחר.

מובן שהשימוש בחוטים מרובים כרוך גם בבעיות מרובות, שאת מקצתן ראינו במאמר הקודם - בעיות כגון חֲבֶק (deadlock), גישה לא מסונכרנת לזיכרון משותף, הפרות סדר פעולות הגישה לזיכרון ועוד. בכל זאת, היתרונות עולים על החסרונות, וכמעט כל תוכנית משתמשת היום במספר חוטים - לפעמים באמצעות תשתיות תוכנה מחוכמות המנהלות את יצירת החוטים והשמת עבודה לחוטים בעצמן.

כיצד מחליפים בין חוטים?

Windows מחליפה בין חוטים לעתים קרובות מאוד. כך נשמרת האשליה, אפילו במחשב בעל מעבד אחד, כאילו כמה תוכניות יכולות לרוץ במקביל. בהמשך נסקור את הזמן שניתן לכל חוט כאשר הוא מקבל את ההזדמנות לרוץ, אבל קודם יש לתת את הדעת לסוגיה הבאה: בכל פעם שמערכת ההפעלה בוחרת להחליף בין חוטים, עליה "להסיר" מן המעבד את החוט שמתבצע כעת, ו"להשים" על המעבד את החוט החדש. מה המשמעות של "להסיר" ו"להשים"?

מערכת ההפעלה מתחזקת מבנה נתונים הנקרא Context, שמתאר חוט שמתבצע כרגע על המעבד. ה-Context מכיל את כל האוגרים (Registers) של המעבד. למשל, במעבד x86, ה-Context מכיל את האוגר EIP המתאר את הפקודה הבאה שהמעבד יבצע, ואת האוגר ESP המתאר את ראש המחסנית. להלן ההגדרה של מבנה הנתונים CONTEXT עבור מעבדים ממשפחת x86[1]:

```
typedef struct _CONTEXT {
    DWORD ContextFlags;
    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD   SegGs;
    DWORD   SegFs;
    DWORD   SegEs;
    DWORD   SegDs;
    DWORD   Edi;
    DWORD   Esi;
    DWORD   Ebx;
    DWORD   Edx;
    DWORD   Ecx;
    DWORD   Eax;
```

[1] מבנה הנתונים CONTEXT עבור חוטים שאינם פעילים כרגע נשמר על המחסנית של החוט. ניתן גם להשפיע על תוכנו ישירות באמצעות הפונקציות GetThreadContext ו-SetThreadContext.

```
    DWORD   Ebp;
    DWORD   Eip;
    DWORD   SegCs;
    DWORD   EFlags;
    DWORD   Esp;
    DWORD   SegSs;
    BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
} CONTEXT;
```

(מבנה הנתונים CONTEXT עבור מעבדי x86, לקוח מתוך WinNT.h)

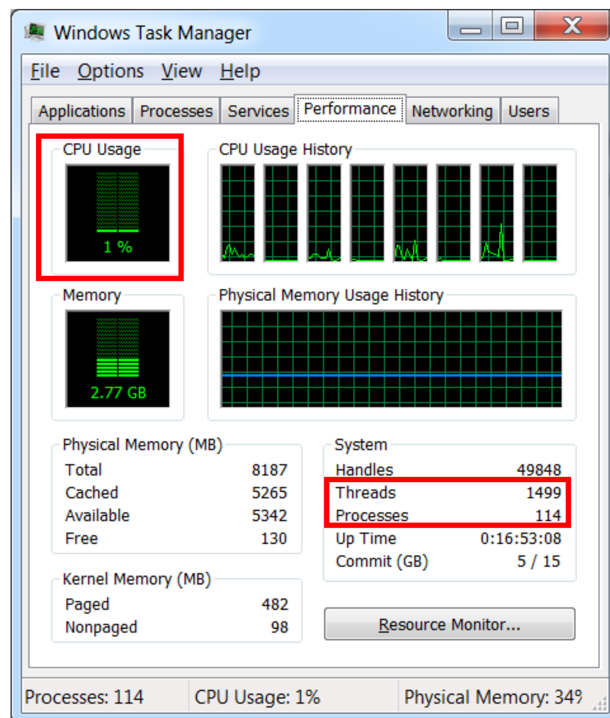
לכן, החלפה בין חוטים משמעותה שמירת האוגרים מהמעבד בתוך ה-Context של החוט המתבצע כרגע, ואחזור האוגרים של המעבד מתוך ה-Context של החוט החדש. זה מספיק בדיוק כדי להחליף בין החוטים - למשל, החלפת האוגר EIP תגרום לכך שהחוט החדש יריץ קוד שונה מהחוט הקודם, והחלפת האוגר ESP תגרום לכך שפעולות של החוט החדש על המחסנית לא ישפיעו על המחסנית של החוט הקודם.

ההבטחה הגדולה: החוט האחד בעל העדיפות הגבוהה ביותר שמוכן לריצה ירוץ תמיד

סכמת התזמון של מערכת ההפעלה מבוססת על רעיון אחד: בכל רגע נתון בוחרים מבין החוטים המוכנים לריצה את החוט בעל העדיפות הגבוהה ביותר, ומריצים אותו. אין זה משנה האם החוט הזה רץ כבר בעבר או שהוא נוצר ברגע זה, והאם העדיפות שלו השתנתה או מאז ומעולם הייתה הגבוהה ביותר. ההחלטה מבוססת אך ורק על המצב הנוכחי, מה שהופך את המימוש של המתזמן (Scheduler) לפשוט יחסית.[2]

יש לציין שבמערכת טיפוסית, ישנם מאות חוטים בכל רגע, אפילו אם המשתמש לא הפעיל עדיין אף תוכנית משלו. אם כל החוטים האלה היו מוכנים לריצה, לא ניתן היה לעבוד בצורה סבירה על המערכת. מכאן, רוב החוטים אינם מוכנים לריצה בכל רגע נתון (כי הם ישנים, ממתינים על מנגנון סנכרון, ממתינים לתוצאות של פעולת קלט/פלט, ועוד) - מה שהופך את מלאכתו של המתזמן לפשוטה עוד יותר, שכן הבחירה היא בין מספר מצומצם יחסית של חוטים.

[2] יש לציין שרעיון "פשוט" זה עובד היטב במערכת בעלת מעבד אחד. ישנה הרחבה מתבקשת למספר מעבדים, אבל נראה בהמשך שמסיבות של ביצועים וסקאלאביליות לא משתמשים בה.



(תצלום מסך ממערכת בעלת 8 מעבדים, שעליה כרגע 1499 חוטים. בכל זאת, צריכת המעבד היא 1% - מרבית החוטים אם לא כולם אינם מוכנים לריצה כרגע)

למעשה, בעיקר במערכות קצה, במשך מרבית הזמן אין אפילו חוט אחד המוכן לריצה - ובמקרה כזה המתזמן מריץ קוד המכונה Idle Loop, שבין היתר יכול להוריד את צריכת החשמל של המערכת או אפילו לכבות מעבדים שאינם בשימוש.

מטבע הדברים ישנן גם מערכות שבהן עשרות חוטים המוכנים לריצה בכל רגע נתון. יכול להיות אפילו שיש חוט אחד או שניים שכל הזמן מוכנים לריצה, מאחר שכל עבודתם היא חישובית. במקרה כזה, ייתכן שהרעיון של הרצת החוט החשוב ביותר נראה קצת לא הוגן - ייתכן שקבוצה קטנה של חוטים מסגלת לעצמה את כל משאבי החישוב של המערכת, ולא מאפשרת לחוטים אחרים לרוץ בכלל. מסיבה זו, בין היתר, Windows מממשת מנגנון של שינויי עדיפות דינאמיים, כפי שנראה בהמשך.

סכימת העדיפויות של Windows

עדיפות של חוט ב-Windows מורכבת משני חלקים: **עדיפות הבסיס** (Process Priority Class) של התהליך שלו, **והעדיפות היחסית** (Relative Priority, Thread Priority) של החוט. כל אחד מהחלקים נקבע בנפרד באמצעות פונקציות שונות - SetPriorityClass עבור עדיפות של תהליך, ו-SetThreadPriority עבור עדיפות יחסית של חוט. אולם בסופו של דבר, המערכת לא מתעניינת בעדיפות היחסית של החוט או בעדיפות הבסיס של התהליך, אלא רק בעדיפות האבסולוטית של החוט - מספר בין 0 ל-31. העדיפות 0 היא העדיפות הנמוכה ביותר, והעדיפות 31 היא העדיפות הגבוהה ביותר.

הטבלה הבאה מסכמת כיצד עדיפות הבסיס של התהליך והעדיפות היחסית של החוט משתלבות יחד ליצירת העדיפות האבסולוטית:

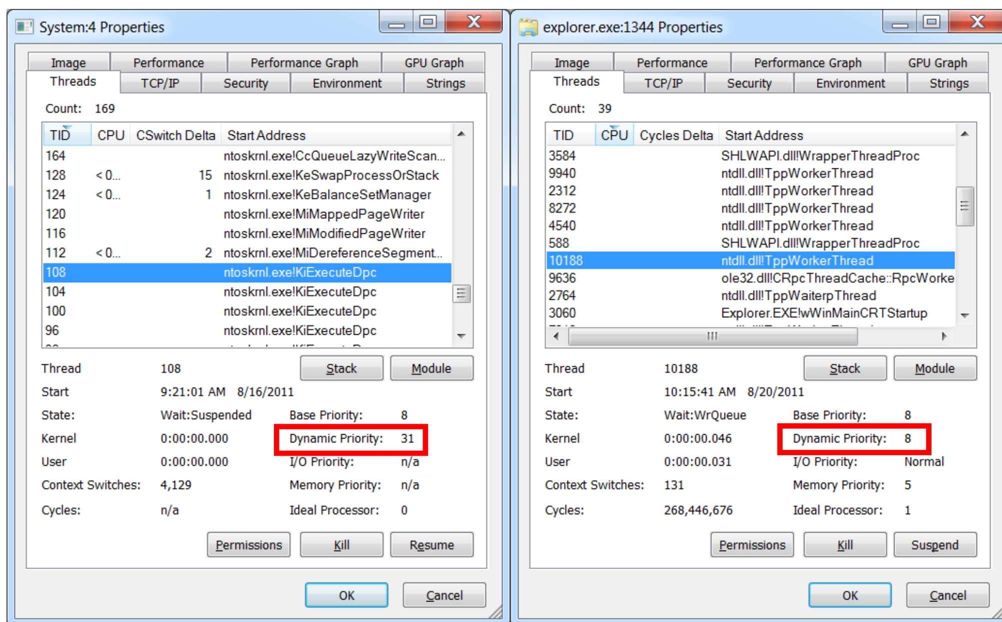
Process Priority Class	Thread Priority Level	Absolute Priority
Idle	Idle	1
	Lowest	2
	Below Normal	3
	Normal	4
	Above Normal	5
	Highest	6
	Time Critical	15
Below Normal	Idle	1
	Lowest	4
	Below Normal	5
	Normal	6
	Above Normal	7
	Highest	8
	Time Critical	15
Normal	Idle	1
	Lowest	6
	Below Normal	7
	Normal	8
	Above Normal	9
	Highest	10
	Time Critical	15

Above Normal	Idle	1
	Lowest	8
	Below Normal	9
	Normal	10
	Above Normal	11
	Highest	12
	Time Critical	15
High	Idle	1
	Lowest	11
	Below Normal	12
	Normal	13
	Above Normal	14
	Highest	15
	Time Critical	15
Realtime	Idle	16
	Lowest	22
	Below Normal	23
	Normal	24
	Above Normal	25
	Highest	26
	Time Critical	31

(רשימת העדיפויות האבסולוטיות האפשרויות כתלות בעדיפות היחסית של החוט ועדיפות הבסיס של התהליך)

יש לציין שקביעת עדיפות הבסיס Realtime דורשת מהמשתמש הרשאות מיוחדות, שלרוב נתונות רק למנהל המערכת (Administrator). שאר העדיפויות זמינות לכל התוכניות, אולם רוב התוכניות אינן טורחות לשנות הן את עדיפות הבסיס של התהליך והן את העדיפויות היחסיות של החוטים. האם זה כדאי? על פניו נראה שכדאי לשנות את רמת העדיפות בהתאם למשימה, או להעלות את העדיפות של התוכנית שלנו על חשבון תוכניות של משתמשים אחרים, או של יצרני תוכנה אחרים. אבל אם כל התוכניות היו משנות את העדיפות שלהן, למשל לעדיפות High, כי אז רמת העדיפות הזאת הייתה הופכת למעשה לרמת ברירת המחדל.

באילו רמות עדיפות משתמשת מערכת ההפעלה עבור החוטים שלה? ובכן, התשובה תלויה בהגדרה של "מערכת ההפעלה". למשל, החוטים של התהליך Explorer.exe רצים בעדיפויות רגילות, הגם שעבור משתמשים רבים הוא מערכת ההפעלה, כיוון שהוא מציג את שולחן העבודה ואת שורת המשימות. לעומת זאת, ישנם חוטים של המערכת שרצים בעדיפויות גבוהות מאוד, למשל חוטים של מנהל הזיכרון הווירטואלי.



(שני חוטים לדוגמה מתוך Explorer.exe והתהליך System. העדיפויות הן 8 ו-31 בהתאמה)

מה ההבדל בין Base Priority ל-Dynamic Priority? Windows מבצעת שינויים דינאמיים של העדיפות לפי פרמטרים שונים. בתצלומי המסך מוצגת העדיפות של החוט כפי שנקבעה לעומת העדיפות של החוט כפי שהמערכת החליטה שהיא צריכה להיות באותו רגע. עוד על שינויי עדיפות דינאמיים בהמשך.

אם אתם מכירים את המושג IRQL (רמת עדיפות של פסיקות), אל תתבלבלו: אין קשר בין עדיפויות של פסיקות לבין עדיפויות של חוטים. למעשה, ניתן לומר שכל פסיקה היא יותר חשובה מאשר כל חוט - ולראיה, אפילו אם כרגע במערכת מתבצע חוט בעדיפות 31, ומתרחשת פסיקה, המערכת תפסיק את ביצוע החוט ותבצע את הפסיקה. מצד שני, במהלך טיפול בפסיקה לא מתרחשת (בדרך כלל) החלפת Context של חוטים, כך ששני המושגים - עדיפות של פסיקות ועדיפות של חוטים - הם אורתוגונאליים לחלוטין.

יחידת הזמן הניתנת לחוט (Quantum)

לפעמים המתזמן של המערכת מופעל כאשר מתרחשים שינויים בחוטים - למשל, העדיפות של חוט מסוים משתנה, חוט נכנס להמתנה, חוט "חוזר" מהמתנה לתוצאות של פעולת קלט/פלט. לעומת זאת, לעתים קרובות לא מתרחשים שינויים בקבוצת החוטים שמוכנה לריצה. במקרה כזה, דרוש מנגנון חיצוני שיגרום למתזמן לרוץ כדי שלא לאפשר לחוט אחד (שאולי תקוע בלולאה אינסופית) להשתלט על כל משאבי החישוב של המערכת.

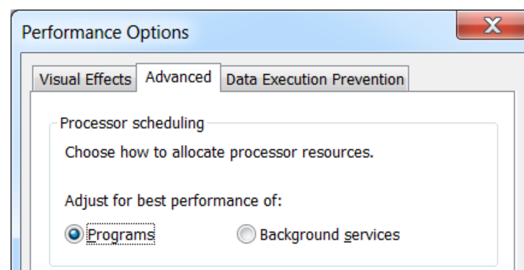
כאשר חוט נבחר לריצה והמערכת מאפשרת לו לרוץ על מעבד מסוים, הוא מוגבל בזמן. זמנו של החוט קצוב אפילו אם החוט מוכן להמשיך לרוץ לנצח ולא מתרחשים אירועים מעניינים אחרים במערכת הגורמים לתזמון. לאחר שזמן זה פג, המתזמן מקבל מחדש את ההחלטה על החוט שירוץ ביחידת הזמן הבאה.

משך הזמן שניתן לחוט (הנקרא Quantum) קשור בשני גורמים. הגורם הראשון הוא קצב השעון המחובר למערכת (אין מדובר על השעון של המעבד, אלא על רכיב חומרה חיצוני היושב על לוח האם). השעון מייצר פסיקות במרווחי זמן קבועים, והמתזמן יכול להשתמש בפסיקות האלה כדי להחליט מתי פגה יחידת הזמן שהוקצבה לחוט מסוים. קצב השעון נמדד בדרך כלל במילישניות - 15 מילישניות הן ערך נפוץ.[3]

```
C:\Windows\system32\cmd.exe

D:\Programs\Sysinternals>clockres

ClockRes v2.0 - View the system clock resolution
Copyright (C) 2009 Mark Russinovich
SysInternals - www.sysinternals.com

Maximum timer interval: 15.600 ms
Minimum timer interval: 0.500 ms
Current timer interval: 1.000 ms

D:\Programs\Sysinternals>
```

(בדיקת קצב השעון של המערכת באמצעות הכלי Clockres.exe של Sysinternals)

הגורם השני המשפיע על ה-Quantum הוא הגדרה שנמצאת בידי המשתמש. Quantum קצר גורם לכך שהחלפה בין חוטים מתרחשת לעתים קרובות יותר, מה שיכול לשפר את התגובתיות של המערכת - כל חוט מקבל הזדמנות לרוץ לעתים קרובות יותר, מה שמתאים לעתים קרובות לתוכניות המבצעות אינטראקציה עם המשתמש. לעומת זאת, Quantum ארוך מגדיל את הסיכוי שחוט יסיים את עבודתו (או לפחות חלק ניכר ממנה) לפני שיבקשו ממנו לפנות את המעבד, מה שמתאים לעתים קרובות לתוכניות המבצעות עבודה ברקע. לא תמיד ברור מה האפשרות העדיפה, ולכן ניתן לשלוט עליה באמצעות ה-Registry או הגדרות "המחשב שלי":



(הגדרה המשפיעה על ה-Quantum. האפשרות המסומנת משמעותה Quantum קצר יותר)

---

[3] בהרבה מקרים, יש אפשרות להשפיע על קצב השעון. למשל, בתצלום המסך להלן, קצב השעון הנוכחי הוא מילישניה אחת, אולם הקצב המקסימלי הוא 15.6 מילישניות והקצב המינימלי הוא 0.5 מילישניות (500 מיקרושניות). שינוי קצב השעון בזמן ריצה גורם לכך שמנגנוני תזמון כמו Timers יכולים להיות מדויקים יותר, אבל הוא לא משפיע על תדירות הריצה של מתזמן החוטים, שמסתמך על נתון ה"מקסימום" בתצלום המסך לעיל.

כיצד ההגדרות הנ"ל מתורגמות ליחידות זמן? הבחירה ב-"Programs" גורמת ל-Quantum להיות באורך כפול מתדירות השעון, למשל 31.2 מילישניות אם תדירות השעון היא 15.6 מילישניות. לעומת זאת, הבחירה ב-"Background services" גורמת ל-Quantum להיות כפולת-12 של תדירות השעון, למשל 187.2 מילישניות אם תדירות השעון היא 15.6 מילישניות.

גם כאן אין מנוס מדקויות: ראשית, לא כל חוט יוכל לרוץ בדיוק את משך הזמן שהוקצב לו מלכתחילה, שכן במהלך ריצתו יכול להיווצר (או להתעורר) לפתע חוט חדש, חשוב יותר, והמתזמן יחליף אותו מיד. שנית, ייתכן שבסוף ה-Quantum, המתזמן יחליט להמשיך להריץ את החוט, מאחר שהוא עדיין החוט החשוב ביותר המוכן לריצה. שלישית, ייתכן שבמהלך ריצת החוט מתרחשות פסיקות רבות הגוזלות מזמן הביצוע של החוט, ובמקרה כזה מערכת ההפעלה תתחשב בכך ותאריך את משך הזמן המוקצב לו[4].

שינויי עדיפויות דינאמיים

מחד, השימוש בעדיפויות מאפשר לתוכניות לתעדף משימות חשובות יותר ופחות, ולמערכת ההפעלה להחליט בקלות איזה חוט להריץ בכל רגע. מאידך, שימוש עיוור בעדיפויות מוביל לבעיות של אי-הוגנות, כאשר הנפוצות ביותר מביניהן הן הרעבה והיפוך עדיפויות.

הרעבה היא מצב שבו חוט לא מקבל אפשרות לרוץ מכיוון שכל זמן שהוא מוכן לריצה, יש חוט חשוב יותר שמוכן גם הוא לריצה. היפוך עדיפויות הוא מצב עדין יותר, שבו משתתפים לפחות שלושה חוטים. נניח ש-A, B, C הם חוטים בעלי עדיפויות 1, 2, 3 בהתאמה. נניח גם שהחוטים A, C משתפים ביניהם משאב, ולכן משתמשים במנגנון סנכרון M למניעה הדדית. כעת ייתכן המצב הבא: החוט A מתחיל לרוץ, ותופס את ה"מנעול" M. מיד לאחר מכן החוט C מתעורר, מה שגורם למתזמן החוטים לסלק את A מהמעבד ולהחליפו ב-C. החוט C זקוק גם הוא ל-M, ולכן הוא נכנס למצב המתנה. כעת מתעורר החוט B ומתחיל לרוץ - הוא חשוב יותר מ-A, ואם כעת הוא מוכן לריצה למשך זמן רב, החוט A מורעב. אבל לא זו בלבד ש-A מורעב ולא מקבל הזדמנות לרוץ, אלא שבגלל M גם C לא מקבל הזדמנות לרוץ (כיוון שהוא זקוק ל-M ש-A מחזיק). נוצר היפוך עדיפויות - חוט בעדיפות 2 רץ בעוד שיש חוט בעדיפות 3 שהיה יכול לרוץ, אבל אינו מוכן לריצה באותו רגע.

על מנת לטפל במצבים מעין אלה Windows מבצעת שינויי עדיפות דינאמיים של חוטים במהלך ריצתם. רק חוטים בעלי עדיפויות 1-15 נהנים משינויי עדיפות דינאמיים אלה, וכל שינויי העדיפות הם זמניים,

---

[4] התחשבות כזו קיימת רק בגרסת NT 6.0 ומעלה, כלומר החל מ-Windows Vista.

ובדרך כלל דועכים ברמת עדיפות אחת מדי Quantum עד חזרתם לרמה המקורית. כמו כן, כל השינויים הם כללי אצבע - כלומר, המערכת לא יכולה לדעת בוודאות ששינוי העדיפות ישפר את התגובתיות, ההוגנות, או פרמטרים חיוביים אחרים. התקווה היא שבמרבית המקרים, שינויי העדיפות לא מזיקים, ולפעמים יכולים גם להועיל.

קצרה היריעה מכדי למנות את כל שינויי העדיפות הדינאמיים ש-Windows מבצעת, ולכן נתבונן רק בשלושה מהם:

מהות השינוי	גודל השינוי	הסבר
העלאת עדיפות לאחר חזרה מהמתנה על אובייקט סנכרון	1 או 2 רמות עדיפות	התייחסות לחוט שמבצע המתנה בתור חוט ה"מוותר" בחביבות על ה-Quantum שלו
העלאת עדיפות לאחר סיום פעולת קלט/פלט	כתלות במנהל ההתקן (Driver) כאשר הוא קורא לפונקציה IoCompleteRequest	לאפשר לחוט שחיכה לסיום פעולת קלט/פלט לעבד את התוצאות של הפעולה
העלאת עדיפות והארכת ה-Quantum לחוט שלא רץ במשך מספר שניות, ומוכן לריצה	העלאה לרמת עדיפות 15 וריצה במשך שני Quantum-ים	התמודדות עם הרעבה והיפוך עדיפויות

(שלוש דוגמאות לשינויי עדיפות דינאמיים ש-Windows מבצעת במקרים שונים)

Yin and Yang: בחירת מעבד פנוי להרצת חוט ובחירת חוט להרצה על מעבד פנוי

כאשר מערכת ההפעלה צריכה לבחור מעבד להריץ עליו חוט מסוים (למשל, חוט שנוצר כרגע או שהתעורר מהמתנה), היא צריכה להתחשב לא רק בעדיפות של החוט, אלא גם באיכויות היחסיות של המעבד ביחס לחוט. למשל, המערכת צריכה להעדיף את המעבד שעליו החוט רץ לאחרונה, שהרי שם יש עוד סיכוי למצוא במטמון את המידע שהחוט יצטרך בריצה העתידית. אולם גם החוט יכול לקבוע את העדפותיו - באמצעות פונקציות כגון SetThreadAffinity חוטים יכולים לקבוע מעבד או קבוצת מעבדים שרק עליהם יוכלו לרוץ[5]. באופן דומה, כאשר מתפנה מעבד (למשל, כיוון שחוט מסתיים או נכנס להמתנה), ויש לבחור חוט חדש להריץ עליו במקום החוט שהוסר ממנו, מערכת ההפעלה תעדיף חוטים שרצו על המעבד לאחרונה.

[5] מבלי להיכנס לפרטים שאין מקום עבורם במאמר זה, ישנה בעייתיות גדולה בקביעת העדפת מעבדים בצורה נוקשה. למשל, ייתכנו מצבים שבהם קביעת העדפה כזאת תגרום לחוט מסוים להיות מורעב, כיוון שהמעבדים המועדפים עליו תפוסים כל העת על ידי חוטים חשובים אפילו יותר. הדבר נכון עוד יותר כאשר התוכנית צריכה לרוץ על מחשבים בעלי מספר שונה של מעבדים ובעלי ארכיטקטורת חומרה שונה – כי אז קשה מאוד לנחש מראש מהו תעדוף המעבדים המיטבי עבור התוכנית, ולעתים קרובות עדיף להשאיר את ההחלטות למערכת ההפעלה.

כיצד המערכת מתאימה את עצמה לשינויי חומרה

Windows נאלצת להתאים את עצמה לשינויי חומרה המתרחשים בשנים האחרונות. שינויים במעבדים ובמבנה הזיכרון גורמים למתזמן החוטים לקבל החלטות בצורה מורכבת יותר. נביא שלוש דוגמאות מייצגות למצבים כאלה.

מזה כעשור, חברות שונות ובראשן אינטל מייצרות מעבדים בעלי יכולת HyperThreading, כלומר סימולציה של מספר מעבדים לוגיים (בדרך כלל 2) על מעבד פיזי אחד. הדבר נעשה על ידי שכפול האוגרים במעבד השומרים את מצב הביצוע, כגון EIP, ESP שראינו קודם, אך ללא שכפול משאבי החישוב, כמו היחידה האריתמטית-לוגית (ALU). כאשר המעבד אינו משתמש במשאבי החישוב וממתין - למשל כתוצאה מגישה לזיכרון - משתמשים במשאבי החישוב להרצה של חוט אחר. שיפור הביצועים המתקבל לעומת מעבד לוגי אחד על מעבד פיזי אחד הוא בדרך כלל קטן יחסית, ואפילו בתחזיות האופטימיות ביותר מדובר על שיפור של 30% בלבד.

מערכת ההפעלה צריכה להתחשב ב-HyperThreading במהלך תזמון חוטים. למשל, כיוון שהמטמון משותף במילא לכל המעבדים הלוגיים על אותו מעבד פיזי, אין יתרון להעדפת מעבד לוגי <u>מסוים</u> כיוון שהחוט רץ עליו לאחרונה. כלומר, כאשר בוחרים חוט עבור מעבד לוגי פנוי, יש להסתכל גם על חוטים שרצו לאחרונה על מעבדים לוגיים אחרים על אותו המעבד הפיזי. באופן דומה, המערכת תעדיף לשבץ חוט על מעבד לוגי שהמעבד הפיזי שלו פנוי לגמרי על פני מעבד לוגי שהמעבד הפיזי שלו כבר מריץ חוט אחר (במעבד לוגי אחר).

דוגמה אחרת לשינויי ארכיטקטורה שמכתיבים שינויים במתזמן החוטים היא NUMA, ארכיטקטורת חומרה שבה המעבדים והזיכרון מסודרים בקבוצות הנקראות Nodes. בכל קבוצה נמצאים חלק מהמעבדים וחלק מהזיכרון. למשל, במערכת בעלת 8 מעבדים ו-24 ג'יגה-בתים של זיכרון, יכולות להיות ארבע קבוצות, כאשר בכל קבוצה 2 מעבדים ו-6 ג'יגה-בתים של זיכרון. הסיבה לחלוקה לקבוצות היא הנדסית בעיקרה: ככל שמוסיפים יותר מעבדים ויותר זיכרון למערכת, כך קשה יותר להחזיק את כל המעבדים במרחק זהה (וקטן) מהזיכרון. מכאן שמעבד הניגש לזיכרון בתוך הקבוצה שלו עושה זאת במהירות מירבית, ואילו גישה לזיכרון בקבוצה אחרת היא איטית בהרבה (לעתים פי שניים או פי שלושה).

כמובן, גם במקרה זה מערכת ההפעלה מוכרחה להתחשב ב-NUMA כאשר מתזמן החוטים עושה את עבודתו. למשל, המתזמן ייצא מגדרו על מנת שחוט לא ינדוד בין מעבדים הנמצאים בקבוצות שונות. כמו

כן, המערכת צריכה לדאוג שהקצאות זיכרון המתבצעות על ידי חוט מסופקות מהזיכרון הפיזי הנמצא בתוך הקבוצה של המעבד עליו הוא רץ[6].

לבסוף, בשרתים מודרניים ישנה היכולת לכבות חלק מהמעבדים לחלוטין בזמן שלא נעשה בהם שימוש. יכולת זו נקראת Core Parking והיא מאפשרת חיסכון משמעותי באנרגיה. גם כאן, מערכת ההפעלה נדרשת להתחשב במעבדים כבויים - היא תשבץ חוטים קודם במעבדים פנויים שאינם כבויים, לפני שהיא מדליקה מעבד כבוי.

סיכום

תזמון חוטים הוא משימה מורכבת בהרבה מכפי שנראה אולי בתחילה, במערכות בעלות מעבד אחד ומספר מצומצם של "טריקים" לשינויי עדיפות ומשך זמן עבור ריצת החוטים. בימינו, מערכות ההפעלה השונות מתחרות הן במישור הביצועים והן במישור צריכת החשמל, וכמובן הגידול המעריכי במספר המעבדים אינו מקל את עבודת מתזמן החוטים.

ימים יגידו האם Windows ומערכות הפעלה אחרות הבנויות סביב מודל תזמון החוטים הקלאסי שהוצג כאן יוכלו להתמודד, כפי שהן, עם אלפי או עשרות אלפי מעבדים. בינתיים, Windows מסוגלת לתזמן עשרות אלפי חוטים על 256 מעבדים בהצלחה - ובמורכבות - גדולה.

על המחבר

משה גולדשטיין הוא ה-CTO של קבוצת סלע, חברת ייעוץ, הדרכה ומיקור חוץ בינלאומית עם מטה בישראל. משה אוהב לנבור בקרביים של Windows וה-CLR, ומתמחה בניפוי שגיאות ומערכות בעלות ביצועים גבוהים. משה הוא ממחברי הספר Windows 7 for Developers, ובין היתר מלמד במכללת סלע קורסים בנושא Windows Internals. בזמנו הפנוי, משה כותב בלוג על נושאי פיתוח שונים.

[6] התמיכה המפורשת של Windows ב-NUMA החלה בתקופת Windows Vista והשתפרה עוד ב-Windows 7.