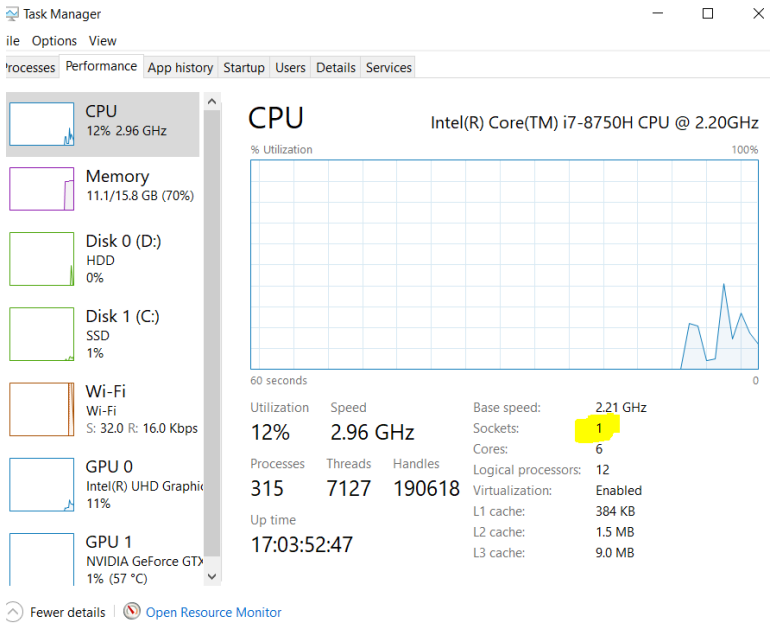
Multithreading Programming

הקדמה

המעבד (Central Processing Unit) CPU או Processor זה המוח של המחשב, אם יש לנו מעבד עם שתי ליבות נוכל לחשוב עליהם כעל המוח השמאלי והמוח הימני.

המעבד מכיל רכיבים אלקטרוניים ומעגלים חשמליים שיכולים לבצע חישובים, פעולות לוגיות, ופעולות קלט פלט. הליבות הן יחידות חישוב שתפקידם להריץ קטעי קוד בתוך המעבד.

בעיקרון ניתן להתקין מחשב מרובה מעבדים בלוחות אם שתומכים בכך, זה מקובל לפעמים בשרתים, פחות במחשבים הביתיים (אם כי ישנם). אבל בימינו בדרך כלל נקנה מחשב עם מעבד מרובה ליבות. בווינדויס נוכל לבדוק כמה ליבות למעבד שלנו ב Task Manager=> Performance.



Process היא ישות וירטואלית, שמערכת ההפעלה מתחזקת על מנת להריץ קטעי קוד. בדרך כלל Process עבור כל אפליקציה שרצה. מערכת ההפעלה מפרידה את האפליקציות באמצעות הישות Process ומקצה עבור כל Process אזור מסוים בזיכרון. הזיכרון לא שותף לשני Processים, וכך נמנע מכל אפליקציה או תוכנית לגעת בקוד או נתונים של Process אחר.

בתמונה למעלה אנחנו רואים שיש 315 Process שרצו במחשב בזמן שהתמונה נלקחה. ניתן לגשת לרשימת ה Process שמערכת ההפעלה מריצה מתוך קוד התוכנית שלנו. זהו שירות של מערכת ההפעלה שחשוף לנו אם נרצה ב #C.

כאמור Process אין להם מידע משותף, אבל כמובן שProcess יכול לרשום לקובץ X וProcess אחר יקרא את הנתונים וישנה אותם. שני הקבצים יכולים לגשת לקובץ X שהוא משאב חיצוני.

כמו"כ ישנו מנגנון שנקרא Mutex שמאפשר לשני Processים לגשת לאזור מסוים בזיכרון שנגדיר באמצעות מערכת ההפעלה. שוב, תוכנית לא ניגשת לאזור הזיכרון או למידע שיושב בזיכרון של התוכנית השנייה. אבל שניהם יכולים לגשת לאזור שלישי בזיכרון, ולתת למערכת ההפעלה לנהל את הגישה לאזור השלישי ל Mutex. קיים עוד מנגנון דומה נוסף שנקרא Semaphore (ההבדלים בגרסה הבאה של מסמך זה).

במקרה ויש לנו כמה ליבות מערכת ההפעלה יכולה להריץ כל Process בליבה אחרת במקביל. כך כמובן המחשב שלנו יהיה מהיר יותר, כל Process מקבל זמן עיבוד במעבד אחר. בתמונה למעלה ראינו שיש 6 ליבות ויש לנו 315 Processים, האם הProcessים רצים יחדיו או שרצים רק שישה והשביעי ייכנס למעבד ויתחיל לרוץ כאשר אחד הProcessים שרצים כעת יסתיים? אם הProcessים היו מחכים להיכנס בתורם עד סיום הProcessים שקדמו להם, היינו מחכים שעות רבות לדברים לרוץ, היינו מקלידים A ורואים אותו מצויר רק עוד שעתיים, מחשבים לא היו מתפתחים למה שהם היום, והיינו מתייאשים (אולי היינו הולכים לים במקום –לא רע כל כך אולי 😊, מצד שני לא היינו יכולים לשבת על החוף ולהתכתב בוואטסאפ כי המעבד של הפלאפון היה מחכה שעות להודעה, כי יש לו עוד Processים להריץ והוא מחכה שהם יסתיימו).

מערכת ההפעלה, נותנת לכל Process איזשהו זמן מעבד (בדרך כלל מדובר בחלקיקי שניות) ואז מפסיקה אותו ונותנת למספר פקודות של Process אחר לרוץ. כך מערכת ההפעלה מעבירה בין הProcessים וכך נראה לנו כאילו כל הProcessים רצים במקביל.

המעבדים גם של הפלאפונים שלנו קל וחומר של המחשב הנייד או השולחני מהירים מאוד ומבצעים מיליוני (או מיליארדי) פעולות בשניה, מערכת ההפעלה מקצה לעיתים חלקיקי שניה מועטים להרצת כל Process בכדי שנקבל את התחושה של ריצה מקבילית של כולם.

התוכניות עצמם יכולים לחלק את הקוד שלהם לחלקים שונים ולהריץ אותם כל אחד בנפרד או ביחד. הישות הווירטואלית הזו שיכולה להריץ חלקי קוד עצמאיים נקרא. Thread (בעברית חדשה: תהליכון).

אם Process הם תהליכים, תתי התהליכים שמגדירים התוכניות, הם תהליכונים – Threads.

אם כן Thread היא ישות וירטואלית שגם נתמכת ע"י מערכת ההפעלה, המאפשרת לתוכניות להריץ את הקוד שלה ב Threads שונים. אז תוכניות כולל אלו של #C יכולים להגדיר Thread שונים. ה Thread יכילו קטעי קוד (מן הסתם פונקציות/מתודות) שכל אחד רץ בלי קשר לחברו. כך, אם יש לנו במעבד כמה ליבות, ייתכן שכל Thread ירוץ בליבה אחרת וכך הם ירוצו במקביל. וכמו"כ, ייתכן והם ירוצו על מעבד אחד בודד ועדיין ייראה כאילו הם רצים במקביל, כי שניהם יקבלו זמן מעבד לסירוגין.

בתוכניות NET. יש Thread ראשי, הוא בדרך כלל כולל את העבודה עם ממשק המשתמש, וכשרוצים להפעיל פעולות נוספות במקביל ניתן להריצם בתהליכון נפרד. פעמים רבות זה נקרא גם Background Thread.

כל Process מנהל את הזיכרון שלו בפני עצמו, Processים לא חולקים זיכרון ומשאבים משותפים. לעומת זאת, תהליכונים של תוכנית אחת שותפים לאותו אזור בזיכרון. אמנם לכל Thread יש (עקרונית) Stack משלו לניהול ריצת המתודות והפונקציות שרצים בתהליכון, אך יש לנו גישה למשתנים שנמצאים ב stack של Thread אחר. ה Heap משותף לכל התהליכונים (Threads).

אז איך מגדירים מתודות שירוצו בתהליכון נפרד, או איך מגדירים תהליך ונותנים לו להריץ קוד כלשהו?

דוגמה 1:

```
//Create an instance of the calss Thread
//The comiler accepts a void no parameters delegate
//(It's defined in the System.Treading namespace and it's called ThreadStart)
//So in this example we have used the lambda expression to pass
//a void no parameter method to the Thraed's constructor
Thread thread = new Thread(() =>
{
      for (int i = 0; i < 3000; i++)
      {
            Debug.WriteLine(i);
      }
});
```

Race Condition

אם מספר תהליכונים צריכים גישה (לעדכן ולקרא) למשתנים בזיכרון המשותף לכמה תהליכונים, אנחנו חייבים לוודא שאנחנו ממשים קוד שהוא Thread Safe. אחרת אנו עלולים לקבל תוצאות שאינם נכונות ובלתי צפויות, מכיוון שבעוד אנחנו ניסינו לעדכן נתונים ע"י תהליכון אחד, הנתונים נדרסו ע"י תהליכון אחר.

נראה דוגמה לשימוש במשתנים מקומיים בכדי לצמצם את השימוש במשאבים משותפים, כלומר, לצמצם את הסיבוכיות והמורכבות של סנכרון שני התהליכונים בהקשר למשאבים המשותפים שלהם.

```
static long Sum0To300()
{
	long sum = 0;
	for (int i = 0; i <= 300; i++)
	 {
		sum = sum + i;
		Console.WriteLine(sum);
	 }
	return sum;
}

static long Sum300To600()
{
	long sum = 0;
	for (int i = 301; i <= 599; i++)
	{
		sum = sum + i;
	}
	Console.WriteLine(sum);
	return sum;
}


static void Main(string[] args)
{
//Local variables to use, each for a different thread, this way is thread safe
//with no need for synchronization, since there are no shared resources
	long sum1 = 0, sum2 = 0;

//We are returning values from thread1 (into sum 1) and thread2 (into sum2)
	Thread thread1 = new Thread(() => { sum1 = Sum0To300(); });
	Thread thread2 = new Thread(() => { sum2 = Sum300To600(); });

	thread2.Start();
	thread1.Start();

// Wait for thread1/thread2 to finish it's job and terminate, meanwhile current
thread is waiting
// If we don't wait the results of (sum1 + sum2) are not expected, since it may
print before the calculation of sum1/sum2 or while it is still calculating
	thread1.Join();
	thread2.Join();

	Console.WriteLine("Sum = " + sum1 + sum2);
}
```