

היכרות עם multithreading

עד כה כאשר פיתחנו קוד בפייטון – נדרשנו לקוד שרץ בצורה רציפה ומבצע "פעולה אחר פעולה".

עם זאת, במקרים רבים הדבר אינו מספק. אם ניקח לדוגמה את Word – בזמן שאנו עובדים על מסמך מסוים מתבצעות במקביל פעולות רבות נוספות כגון בדיקת שגיאות כתיב, שמירה אוטומטית ברקע ועוד.

כפי שלמדנו – התהליך (Process), כלומר התוכנה אותה אנו מריצים, אחראי על מסגרת המשאבים (למשל מרחב הזיכרון המשויך לתוכנה) ובמסגרתו רצים ה - תהליכונים (threads) שהם בסופו של דבר מריצים את הקוד. לכל תהליך חייב להיות לפחות תהליכון אחד ובמידה ולא יצרנו במהלך הקוד תהליכונים נוספים (אם בצורה ייעודית ואם מתוך שימוש בחבילה חיצונית המשתמשת בהם) – תהליכון זה יהיה היחיד שירוצץ. נקרא לו – התהליכון הראשי (main thread). כל התהליכונים רצים מבחינתנו "במקביל" ומאפשרים לנו לבצע מספר לוגיקות בתוכנה שלנו באותו הזמן.

[למרות שאנו דוגלים בשימוש מלא בעברית, יתכן ובחלק מהמקרים נשתמש במושג thread (ת'רד) בשמו באנגלית, על מנת למנוע בלבול].

אם נסתכל על הכל בצורה סכמתית פשוטה:



אבל... איפה אנחנו נתקלים בצורך בשימוש במספר thread-ים??

בטח חלקכם זוכרים מרשתות את התרגיל של כתיבת צ'אט. במסגרת התרגיל היה עליכם לממש שרת אשר תומך בחיבור ממספר לקוחות במקביל. אחת השיטות שבהן היה ניתן להשתמש כדי לטפל בחיבורים הללו הייתה יצירת thread לכל חיבור.

בנוסף, נתקל בהמשך בתוכנות הכוללות GUI (ממשק משתמש גרפי), ובהן גם נזדקק לשימוש במספר thread-ים, שהרי היינו רוצים שמשתמש יוכל ללחוץ על כפתורים ולהשתמש בממשק באותו הזמן שלמשל הצ'אט שלנו מחכה להודעה ממשתמשים אחרים / שולח הודעה וכו'. למזלנו, ברוב המקרים כאשר ניצור GUI – החבילה שבה נשתמש כדי ליצור אותו תטפל ביצירת ה-thread-ים בעצמה.

דוגמה נוספת ניתן למצוא במקרה בו נרצה להאיץ תהליך. יכולות להיות לכך מספר סיבות:

- התהליך אותו אנו מבצעים כולל זמן המתנה ארוך לתגובה של צד אחר. כך למשל, אם נרצה לכתוב כלי שמוריד את כל התוכן של אתר מסוים – נצטרך להוריד קובץ אחר קובץ, כאשר לכל אחד מהם אנו מבצעים בקשת GET מול השרת, מחכים לתשובה, שומרים את התוצאה לדיסק ורק אז עוברים לבא. באמצעות ריבוי thread-ים – ניתן לבצע את התהליך מספר פעמים במקביל, ולמקבל את זמן ההמתנה לתשובת השרת.

- התהליך שלנו כולל מאמץ חישובי רב, ולכן נרצה להיעזר בכלל ליבות המעבד הזמינות. ברגע שנריץ את הקוד במקביל מספר פעמים (במידה כמובן שהתאמנו אותו לריצה במקביל, כך שבכל thread מבוצע חלק אחר של החישוב) – אז מספר ליבות של המעבד יוכלו לפעול במקביל ונוכל להגיע לניצול יותר יעיל של משאבי המחשב. (** בפייתון בו אנו משתמשים, אשר המימוש שלו הינו CPython – לא נוכל לקבל ביצוע מקבילי "אמיתי" באמצעות thread-ים – הם יראו לנו כרצים במקביל, אך בכל רגע זמן נתון רק אחד מהם יבוצע על המעבד. במידה ואכן נרצה להביא לשיפור ביצועים במקרה זה – נצטרך להשתמש בריבוי תהליכים – process-ים. המתעניינים מוזמנים לקרוא על מגבלת ה GIL ב-Python).

אז... איך יוצרים thread-ים בפועל?

לצורך כך – נשתמש בחבילה threading בפייתון. החבילה מאפשרת יצירת thread-ים בצורה פשוטה ונוחה. את קטע הקוד שניתן ל-thread להריץ – אנו מגדירים בפונקציה. ניתן גם, במידת הצורך, להעביר אליה פרמטרים (למשל – כשמדובר על שרת צ'אט – נעביר כפרמטר את מזהה החיבור שבו אותו thread הולך לטפל).

נסתכל על דוגמא פשוטה:

```
import threading

def thread_func(num1, num2):
    print num1 + num2

t = threading.Thread(target=thread_func, args=(1,2))
t.start()
```

במקרה זה הגדרנו פונקציה – `thread_func` שאותה הגדרנו כ"קוד שירוץ בתוך ה-thread". ניתן לראות בדוגמא זו כיצד משתמשים באובייקט Thread בכדי ליצור את ה-thread ולהריץ אותו (`start`).

במידה ונרצה להחזיר מידע מה-thread שהרצנו – נוכל להשתמש למשל ב-list. מכיוון ש list הינו משתנה mutable (חפשו באינטרנט על משמעות הדבר!) אנו יכולים להעביר אותו ל-thread כפרמטר, וכל שינוי שיבוצע עליו ב-thread ישמר במשתנה שהעברנו (כלומר – אם נעביר לו את המשתנה a ב-args לאחר שאתחלנו אותו לרשימה ריקה, ובתוך ה-thread יתווסף איבר לרשימה – נוכל לגשת לאיבר הזה דרך a).

פונקציה שימושית נוספת שיתכן ונזדקק לה בהמשך הינה `join` (שלה התנהגות שונה מה-`join` שאנו מכירים):

```
t = threading.Thread(target=thread_func, args=(1,2))
t.start()
t.join()
```

הפונקציה מחכה לסיום הביצוע של ה-thread הנתון ורק לאחר מכן ממשיכה הלאה. דבר זה יכול להיות שימושי במיוחד כאשר נרצה להריץ מספר thread-ים, ורק לאחר שהם יסתיימו – להמשיך את ריצת הקוד שלנו ב-thread הראשי.

תרגיל Fast Getter

קיבלתם טלפון בהול ממר שטרול, מנהל אתר החדשות האהוב YMAT.

מר שטרול סיפר כי קיבל בבוקר איום מהאקרים שאם לא ישלם להם עד הערב מליון באט (מטבע תאילנדי), ימחקו לו את כל מאגרי המידע. מר שטרול שמע על יכולותיכם וביקש מכם לכתוב לו משהו שידע להוריד סט של קבצים מהר, כדי שיוכל לגבות את המאגרים. אחד הדברים שהכי חשובים לו הינו רשימת הסרטים האהובים עליו מכל הזמנים. רשימה זו יושבת בתוך סט קבצים עם מזהה סידורי. מכיוון שהרשימה משתנה, והוא רוצה לגבות אותה כמה שיותר קרוב לדד-ליין שהציבו לו ההאקרים – עליכם לכתוב כלי שיאפשר להוריד אותה בצורה הכי מהירה.

שמות הסרטים ממוקמים ב- <http://cyber.org.il/os/demo/movieXX.txt> כאשר XX מהווה מספר בין 1 ל 32.

כך, למשל, הסרט הראשון יושב בקובץ <http://cyber.org.il/os/demo/movie1.txt> (במקרה מדובר על הסרט המפורסם "חומות של תקווה").

1. כתבו תחילה קוד המוריד את הקבצים באופן סדרתי (אחד אחרי השני) ומדפיס אותם למסך ללא שימוש ב thread-ים. מומלץ להיעזר בחבילה urllib ובה בפונקציה urlopen אשר מתחברת ישירות לכתובת url וחוסכת לכם את ההתעסקות עם בניית בקשת HTTP.
2. כעת, כתבו קטע קוד נוסף המשתמש ב thread-ים בכדי לבצע הורדה במקביל של כל הקבצים יחד. כל קובץ למעשה ירד ב thread נפרד. גם כאן יש להדפיס את שמות כל הסרטים שהורדתם.

האם שמתם לב להבדל בביצועים בין שני קטעי הקוד?

במידה ובחרתם ב- (2) להדפיס את שמות הסרטים מתוך ה- thread - יתכן ותתקלו בבעיה. נסו לחשוב כיצד לפתור אותה (רמז: היעזרו בנקודות המופיעות בעמוד הקודם).

בהצלחה!

הרחבה: שימוש ב multiprocessing

כפי שהזכרנו – כאשר נרצה לנצל את כל הליבות של המעבד למשל לטובת חישוב כבד, השימוש ב thread-ים לא יעזור לנו. למקרה זה, אנו יכולים להיעזר בחבילה שמאפשרת ריבוי תהליכים בפיתון – כך שבמקום שקטע הקוד שלנו יורץ ב thread נפרד, הוא ירוץ ב process חדש של פיתון. (תוכלו לראות ממש תהליך חדש של python.exe שיווצר ויהיה קיים בזמן ריצת קטע הקוד).

על מנת להקל על המפתחים – לחבילה זו (multiprocessing) ממשק דומה מאוד לזה שראינו ל-thread-ים. כך, עבור קוד זהה למה שראינו מקודם (רק הפעם עם שימוש בריבוי תהליכים) - נקבל:

```
import multiprocessing

def process_func(num1, num2):
    print num1 + num2

if __name__ == '__main__':
    t = multiprocessing.Process(target=process_func, args=(1,2))
    t.start()
```