**מערכות הפעלה – תרגיל בית מספר 3 – תיעוד החלק הרטוב**

במסגרת התרגיל התבקשנו לממש ThreadPool. נתאר את אופן המימוש שנבחר, את השימוש שעשינו ב-Posix threads וה-synchronization primitives בהם השתמשנו על מנת להשיג את המטרה של מקבול וסנכרון.

מבנה ה-thread pool:

מבנה זה הוא בעצם המבנה העיקרי במסגרת המימוש. הוא מכיל את השדות הבאים:

1. מצביע לתור מהממשק אשר סופק לנו שנקרא taskQueue ומטרתו בעצם לשמור את כלל ה-tasks שה-pool צריך לטפל בהם. בהמשך יוגדר המבנה המייצג task ומהווה איבר בתור זה.
2. מנעול mutex שנקרא mutex\_taskQueue\_lock. מתלווה ל-condition variable שיוגדר בהמשך
3. משתנה תנאי (condition variable) שנקרא cond\_taskQueueNotEmpty אשר מטרתו להיות אחראי על התנאי שבודק אם תור המשימות ריק בדומה לדוגמאת התור שנלמדה בתרגול. משתנה זה משולב עם מנעול ה-mutex שתואר כשדה השני של מבנה זה. הסיבה לצורך ב-mutex מפורטת בחלק היבש של תרגיל בית זה.
4. משתנה integer השומר את כמות ה-threads שיהיו ב-pool (מועבר כפרמטר בעת האתחול).
5. סמפור שנקרא sem\_tpDestroyWasInvoked ומטרתו לסמן האם פונקציית הממשק tpDestroy הופעלה כבר שכן הדבר משפיע על פעולות אחרות שעלולות להתרחש במקביל (ראה גם פירוט בהמשך). נדגיש כי הפעלת tpDestroy מתפקדת כמשאב יחיד ולכן הסמפור בו בחרנו להשתמש הינו סמפור בינארי.
6. מערך של מצביעים לטיפוס pthread\_t כאשר המערך שומר בעצם את מזהי ה-threads ב-pool במטרה שנוכל לבצע הריסה של החוטים הללו בעת הפעלת tpDestroy.

מבנה המייצג task (נקרא funcAndParam):

תחילה הגדרנו מבנה אשר מכיל 2 שדות כאשר הראשון הוא מצביע לפונקציה כלשהי וכן משתנה void\* אשר שומר בתוכו פרמטר מסוים בו ייעשה שימוש במסגרת הפונקציה אותה אנו שומרים במבנה זה. בעצם, תור ה-tasks אותו נגדיר בהתאם לדרישת התרגיל יכיל איברים כאשר כל איבר הוא מטיפוס מבנה זה.

פונקציית ההריסה העצמית (self-destruct):

הגדרנו פונקציה שמהותה הריסת חוט מסוים. כאשר נרצה לבצע השמדה מוחלטת של ה-thread pool אנו נזין את הפונקציה הזו כמות פעמים כמספר ה-threads ב-pool כ-tasks בתור המשימות ובכך נגרום להשמדת כל אחד מהחוטים. כלומר,פונקציה זו בעצם תהיה פרמטר לשדה הפונקציה במבנה funcAndParam שהוגדר לעיל.

פונקציית לולאת ביצוע ה-tasks (נקראת getandexecutetasksforever):

במרכז המימוש שלנו עומדת פונציה האחראית בעצם על קבלת task מסוים כפי שהוגדר במסגרת דרישות התרגיל ועל ביצועו. אנו שומרים במסגרת המימוש את ה-tasks במבנה של תור ולכן, מכיוון שאנו משתמשים במקבול יש צורך לסנכרן את פעולות ההוצאה וההכנסה לתור כפי שראינו בתרגול. נדגיש כי הלולאה הכללית הזו שאחראית על לקיחת ה-task מהתור וביצועו היא לולאה אינסופית וכי בכל איטרציה מתבצעת נעילה של התור באמצעות mutex ושימוש ב-condition variable באופן דומה לחלוטין לבעיית התור כפי שהוצגה בתרגול. כלומר, אנו משתמשים בלולאת while פנימית (הסיבה לשימוש ב-while גם היא נלמדה בתרגול וזאת מכיוון שגם לאחר החזרה מ-wait של ה-condition אנו צריכים לבדוק את התנאי פעם נוספת שכן מרגע קבלת הסיגנל ועד הריצה עובר זמן מה) שבודקת התנאי ובמקרה של קיומו מבצעת את ההמתנה ל-task. רק לאחר שהממתנה הושלמה אנו אכן מוציאים את ה-task מהתור, משחררים את הנעילה ומבצעים את ה-task. נבחין כי במסגרת פונקציה זו אנו כבר עושים שימוש ב-synchronization primitives ובפרט בקונספט של condition variable (המשולב עם mutex). נדגיש כי החלק של הלולאה שבודקת אם תור ה-tasks ריק זהה לנלמד בתרגול ולכן מקפיד על סנכרון כך שלא ננסה לבצע הוצאה מתור ריק.

פונקציית הממשק tpCreate:

פונקציה זו בעצם אחראית בהתאם גם להגדרתה בממשק לאתחול של thread pool עם כמות threads נתונה כפרמטר. במסגרת פונקציה זו דאגנו להקצאת הזכרון הדרושה ל-thread pool כמבנה (כפי שהוגדר בקובץ ה-h) וכן לאתחול כל אחד מהשדות שמחזיק ה-thread pool. במסגרת פונקציה זו אנו דואגים גם לאתחול כלל המנעולים שהגדרנו במסגרת ה-tp וכן אתחול של attribute לmutex על מנת להגדירו כ-mutex בודק שגיאות כפי שהגדרנו בתרגול והסברנו מדוע זו הדרך הטובה ביותר.

פונקציית הממשק tpDestroy:

על מנת לבדוק אם אנחנו כבר במסגרת תהליך של הריסת ה-thread pool אנו מחזיקים בין שדות ה-tp שדה שהוא בעצם מצביע ל-semaphore הנקרא sem\_tpDestroyWasInvoked. בתחילת הפונקציה אנו בודקים אם אם ניתן לבצע wait עם ה-semaphore. אם אנו נכנסים אל התנאי, הרי שלא ניתן לבצע wait עבור ה-semaphore ומשמעות הדבר שהמשאב עליו מגן ה-semaphore לא פנוי, כלומר כבר ביצענו קריאה ל-tpDestroy ובעצם semaphore זה נועד להגן מהפעלה של tpDestroy לאחר שכבר הופעל במסגרת המקבול. אנו גם מקפידים במסגרת פונקציה זו לבצע נעילה של התור עם ה-mutex וזאת על מנת למנוע מצב של שינויים במצב התור במהלך תהליך ההריסה של ה-thread pool. נדגיש גם כי אנו מבצעים בדיקה על הפרמטר המועבר שנקרא shouldWaitForTasks בהתאם לדרישת התרגיל על מנת להפריד בין המצבים השונים. במידה וערכו 0 נרוקן את כלל תור ה-tasks. לאחר מכן, כפי שכבר צוין קודם לכן, אנו מכניסים לתור ה-tasks מספר פעמים את הפונקציה self-destruct שהגדרנו לעיל וזה כמספר ה-threads שקיימים ב-pool וזאת על מנת לבצע את ההריסה של כל אחד מהחוטים. רק לאחר מכן נבדוק את האפשרות של ביצוע broadcast למשתנה התנאי שתפקידו לבדוק אם התור ריק (נרצה לנסות "להעיר" את כלל החוטים בהתאם). נדגיש כי במסגרת הפונקציה אנו דואגים גם להריסת המנעולים המתאימים והריסת כל ה-threads תוך שימוש ב-ID שלהם שנשמרו במערך כפי שתואר לעיל.

פונקציית הממשק tpInsertTask:

פונקציה זו בהתאם להגדרת הממשק אחראית על הכנסת task חדש לתור ה-tasks. בתחילת הפונקציה אנו בודקים האם tpDestroy הופעל לפני הכניסה לפונקציה זו תוך בדיקת ערך ה-semaphore וזאת מכיוון שעלינו לוודא שלא התחיל תהליך הריסה שלא יאפשר הכנסת tasks חדשים לתור. אנו מבצעים במסגרת הפונקציה גם נעילה של ה-mutex האחראי על נעילת תור ה-tasks שכן אנו מבצעים הכנסה לתור בדומה לנלמד בתרגול. לאחר מכן, נבצע הקצאה של מבנה FuncAndParam כפי שהוגדר לעיל ונדאג להכניס אותו לתור המשימות כנדרש (בעצם מהות פונקציה זו). בדומה לנלמד בתרגול, אנו נבצע לאחר מכן signal של משתנה התנאי הבודק האם התור ריק וזאת מכיוון שלאחר ההכנסה התור בהכרח אינו ריק. לאחר מכן נשחרר גם את ה-mutex המתאים לאותו תור.

מניעת Deadlock:

ציטוט מויקיפדיה:(<http://en.wikipedia.org/wiki/Deadlock>) תנאים היכרחיים לdeadlock

Necessary conditions

A deadlockers situation can arise if all of the following conditions hold simultaneously in a system:

1. [**Mutual Exclusion**](http://en.wikipedia.org/wiki/Mutual_exclusion)**:** At least one resource must be held in a non-shareable mode. Only one process can use the resource at any given instant of time.
2. **Hold and Wait** or **Resource Holding:** A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. **No**[**Preemption**](http://en.wikipedia.org/wiki/Preemption_(computing))**:** a resource can be released only voluntarily by the process holding it.
4. [**Circular Wait:**](http://en.wikipedia.org/wiki/Circular_reference) A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a [set](http://en.wikipedia.org/wiki/Set_(mathematics)) of waiting processes, P = {P1, P2, ..., PN}, such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3 and so on until PN is waiting for a resource held by P1.

These four conditions are known as the **Coffman conditions** from their first description in a 1971 article by [Edward G. Coffman, Jr.](http://en.wikipedia.org/wiki/Edward_G._Coffman,_Jr.) Unfulfillment of any of these conditions is enough to preclude a deadlock from occurring.

אמנם תנאים 1 ו-3 אכן מתקיימים במימוש שלנו אך גם תנאי 2 וגם תנאי 4 לא מתקיימים ודי בכך שתנאי בודד לא יתקיים כדי שלא יתאפשר מצב של deadlock. למעשה המשאב (resource) היחיד שנדרש (ממתינים לו) במקביל הינו מנעול אחד ולכן לא ייתכן מצב שבו אותו thread דורש משאב בזמן שהוא מחזיק במשאב אחר (תנאי 2) כי בקוד שלנו לא מופיעה המתנה למנעול כאשר המנעול כבר מוחזק.

דאגנו במסגרת המימוש שלנו להימנע ממצב של Deadlock (קיפאון), כלומר מצב בו קבוצת תהליכים נמצאת במצב בו כל תהליך מחכה לפינוי משאב ע"י תהליך אחר באותה הקבוצה. על מנת שיווצר deadlock חייב להיות מצב של "המתנה מעגלית" (circular wait) אשר לא ייתכן במסגרת המימוש שסיפקנו. זאת מכיוון שכאשר אנו מבצעים הכנסה והוצאה מתור המשימות אנו דואגים שתתבצע נעילה כך שלא ייתכן מצב בו שני תהליכים ממתינים אחד לשני על מנת לבצע את אחת הפעולות ובסופו של דבר יתבצע שחרור של הנעילה. גם במקרה הנעילה הנוגע להפעלת הפונקציה tpDestroy לא ייתכן מצב כי שני חוטים נמצאים בהמתנה אחד לשני לשחרור המנעול.