**שו"ת**

1. **מהו Thread?**

Thread ("תהליכון") הוא מנגנון המאפשר לתוכנית (program) לבצע יותר מפעולה אחת (כאילו) בו-זמנית, ע"י יצירת יחידות-ביצוע (=פונקציות) עצמאיות.

מבחינה היררכית, תהליך מכיל תהליכון – כאשר מריצים תוכנית, הקרנל יוצר תהליך חדש, שבתוכו קיים\* תהליכון אחד, ובו רצה התוכנית. תהליכון יכול ליצור תהליכון אחרים, וכולם יריצו את אותה התוכנית – באותו התהליך (כלומר, הם מוכלים בתוך התהליך, ולא בתוך התהליכון שיצר אותם).

* הביטוי "בתוכו קיים" פירושו שהתהליכון והתהליך שבו הוא נוצר חולקים את אותם **"משאבים"** (למשל, אותו מרחב זיכרון וירטואלי ואותם סגמנטים). בוויקיפדיה יש הגדרה יפה:

**A process is a unit of resources, while a thread is a unit of (scheduling and) execution**

1. **מהם ההבדלים בין process לבין thread?**

|  |  |  |
| --- | --- | --- |
|  | **process** | **thread** |
| **1. מהות** | **מופע של תוכנית (או: יחידת משאבים)** | **רכיב בתוכנית הניתן לתזמון בפני עצמו  (או: יחידת ביצוע)** |
| **2. Resource Sharing** | **לכל תהליך יש מרחב כתובות ומשאבים נפרדים** | **תהליכונים חולקים את אותו מרחב כתובות** (**וסגמנטי זיכרון - מלבד ה-stack**, אך ראה [כאן](http://stackoverflow.com/questions/1762418/process-vs-thread?rq=1))**, ולכן - את אותם משאבים** (מלבד לוקאליים ו-thread-specific) |
| **3. תקשורת** | **באמצעות מנגנוני IPC** | **באמצעות מנגנוני נעילה וסנכרון** |
| **4. context switch** (נובע מסעיף 2) | **איטי** | **מהיר** |

1. **מתי נשתמש ב-process, ומתי ב-thread?**
2. כאשר אין צורך/סיבה שיחידות-ביצוע יחלקו את אותם המשאבים**,** נשתמש במספר תהליכים שונים. אם יש צורך/סיבה שיחידות-ביצוע יחלקו את אותם המשאבים (וזה קורה בעיקר כאשר מיקבול העבודה על משימות יהפוך את התוכנית ליעילה יותר מבחינת זמני עבודה, או נוחה יותר עבור משתמש שעושה שימוש ב-GUI, ראה שאלה 5), נשתמש במספר תהליכונים שונים עבור התהליך/תוכני.
3. אם יש מעבד אחד, יהיה שגוי להשתמש בתהליכונים, מכיוון שהזמן שלוקח לבצע context switch יגרום לתוכנית להיות איטית יותר לעומת אי-שימוש בתהליכונים.
4. **מהי multi-thread program?**

תוכנית המורכבת מיחידות-ריצה הפועלות במקביל זו לזו (ולא באופן סדרתי), אשר חולקות ביניהן משאבים (מלבד משאבים שהם לוקאליים לפונקציות שהן מריצות ומשאבים מסוג thread-specific data).

1. **כתוב לפחות 3 סוגי בעיות אשר שימוש ב-multi-threading יאפשר לפתור אותן** ([מקור](https://docs.oracle.com/cd/E18752_01/html/816-5137/mtintro-68348.html))
2. **שיפור התגובתיות** - כל תוכנית המכילה פעולות שאינן תלויות זו בזו יכולה להיבנות כך שיחידת-פעילות עצמאית תתפקד כתהליכון.

**למשל,** שימוש בתהליכונים בתוכנית עם GUI יאפשר עבודה של משימות ברקע בעוד משימה אחרת הרלוונטית למשתמש תוכל להמשיך לפעול בחזית (למשל, שמירה אוטומטית של מסמך וורד יכולה להתרחש ברקע בעוד המשתמש ממשיך לכתוב במסמך. ראה טננבאום עמ' 98-99).

1. **שיפור מבנה התוכנית -** תוכנית יכולה להיבנות בצורה טובה יותר אם יחידות-ביצוע בה יוגדרו כעצמאיות (כלומר, ככאלו שירוצו כתהליכונים), מכיוון שבנייתה באופן מונוליטי יצריך הקדשת חלקים בקוד לתיאום המשימות השונות, ובמקרה שיתעורר הצורך באדפטציות - יהיה קשה יותר לבצע אותן.
2. **ייעול השימוש במשאבי מערכת -** מכיוון שלכל תהליך יש מרחב כתובות משלו (תזכורת: בשכפול תהליך, כל סגמנטי הזיכרון משוכפלים) ו-[state](https://en.wikipedia.org/wiki/Process_state) משלו, העלות שביצירת ובניהול כמות המידע על התהליך יקרה בהרבה מתהליכון, הן בזמן והן במקום (בזיכרון).
3. **מהן הבעיות ב-multi-threading?**
4. Raceconditioning
5. Deadlock (ראה שאלה 10 ו-19)
6. Starvation (כנ"ל)
7. הרצת יותר מדי תהליכונים במקביל תאט את התוכנית ותפספס את מהות השימוש בתהליכונים (=להפוך את התוכנית למהירה יותר).
8. **כיצד נעביר מידע בין שני threads?**

כל שימוש במשאב שאיננו לוקאלי לפונקציית-תהליכון (כלומר, שלא יושב ב-stack השייך לתהליכון) ואיננו משאב מסוג thread-specific data (שנ"ל שהוא גם כן מעין stack) הוא משאב שמספר תהליכונים יכולים לעשות בו שימוש.

1. **מהו lock?**

**א. מהות**

מנגנון המיועד לסנכרון העבודה והגישה למשאבים (למשל: למשתנים גלובליים, ל-buffer (רשימה מקושרת, circular buffer, ועוד)) בין תהליכונים בתוכניות שהן multi-threaded, המונע התנגשות ו-race conditions שעלולים לגרום לתוצאות שגויות.

השימוש במנעול פוגם, למעשה, בעבודה המקבילית שמאפשרת עבודת multi-threading, ולכן יש להשתמש במנעולים רק בנקודות הכרחיות - ב-critical sections, ואותם-עצמם יש לצמצם ככל הניתן מאותה הסיבה.

* חשוב לשמור על כך שכל מנעול יקושר באופן ברור עם data שהוא נועל:

lock data, not code

למשל: "the strcut foo is locked by foo\_lock".

**ב. צורת פעולה**

בעת הגעה לקטע קריטי, התהליכון קורא לפונקציית נעילה ושולח כפרמטר משתנה-מנעול כלשהו.

אם המנעול פנוי, הנעילה תצליח והתהליכון יוכל להתקדם. אם המנעול לא פנוי - תוצאת בקשת הנעילה תלויה בסוג המנעול (ב-mutex, ב-semaphore וב-condition variable תתבצע חסימה (blocking) של התהליכון הקורא, אך גם זה תלוי-מקרה).

נעילת קטע הקוד הקריטי משמעה: רק התהליכון הנועל יכול להשתמש (קריאה/כתיבה) במשאבים אליהם מתבצעת פנייה בקטע הקוד.

עם סיום הקטע הקריטי, יבצע התהליכון שחרור של המנעול.

1. **מהו spinlock?**

**מהות**

נעילה (זאת יותר "נעילה" מאשר [סוג של] "מנעול") הגורמת לתהליכון המנסה לבצע אותה להמתין בצורת busy-waiting: להמתין בלולאה (spin) הבודקת-וחוזרת-ובודקת אם המנעול פנוי.

**יעילות**

שימוש ב-spinlock יכול להיות יעיל במקרה שהתהליכון צפוי להיות חסום (מנעילת המנעול) לפרק-זמן קצר. במקרים כאלה, שימוש במנעול mutex ובמנעולים מורכבים אחרים יהיה יקר יותר מכיוון שהם מרדימים את התהליכון המבקש לבצע עליהם נעילה ומעירים אותו כשהמנעול נפתח - פעולות יקרות מבחינת משאבי מערכת, הדורשות כמות רבה יחסית של הוראות ל-CPU ולפיכך גם דורשות יותר זמן.

ככל שתהליכון מחזיק מנעול לאורך יותר זמן, ה-polling שמתרחש ב-spinlock יגזול יותר ויותר זמן CPU, ובמקרים כאלה עדיף להרדים את התהליכון.

* Interrupt handlers עושים שימוש רק ב-spinlocks, מכיוון שלינוקס לא מאפשרת ל-handlers להירדם.

הסיבה היא שבלינוקס interrupt handlers אינם processes העומדים בפני עצמם, אלא רצים בתוך קונטקסט ה-process שגרם לקריאתם, ולעיתים פועלים על אותו ה-stack שלו. ולכן אין מנגנון scheduling שיתמוך בהם ויאפשר להם "לישון" ו"להתעורר". (מקורות: [כאן](https://www.quora.com/Why-shouldnt-one-acquire-a-mutex-in-an-interrupt-service-routine), [כאן](http://stackoverflow.com/questions/1053572/why-kernel-code-thread-executing-in-interrupt-context-cannot-sleep) ו[כאן](http://stackoverflow.com/questions/12758182/is-interrupt-handler-running-like-this-and-for-how-long)).

1. **מהו try-lock?**

נעילה (זאת יותר "נעילה" מאשר [סוג של] "מנעול") המתבצעת רק אם המנעול הנקרא הינו פנוי ברגע הקריאה לנעילתו.

לרוב, השימוש בפועל בפונקציה כדוגמת pthread\_mutex\_lock מתבצע כחלק מהתניה, כך שבמידה והמנעול פנוי – הקריאה לפונקציה תבצע נעילה על קטע קריטי, אך במידה והמנעול תפוס – יוחזר ערך שגיאה כלשהו (למשל, EBUSY), ופעולות אחרות יתבצעו (עד לנסיון הנעילה הבא).

1. **מהו sequential lock (seqlock)?** (מקור: הספר understanding the linux kernel, וויקיפדיה)

מנגנון נעילה המתבסס על access counter, המשמש לצורך גישת writer & reader למשאב משותף (למשל, משתנה גלובלי. ראה [כאן](http://www.hpl.hp.com/techreports/2012/HPL-2012-68.pdf) דוגמא קצרצרה, בעמ' 2. אני לא יודע איך שימוש ב-seqlock יכול להיות רלוונטי למשאב מורכב יותר, כמו circular buffer).

כאשר מבצעים שימוש במנגנוני spinlock לקריאה ולכתיבה, לפעולת הכתיבה והקריאה יש רמת עדיפות זהה, דהיינו: הקריאה יכולה להתחיל רק כאשר הכתיבה הסתיימה, והכתיבה יכולה להתחיל רק כאשר הקריאה הסתיימה.

מנגנון ה-seqlock מאפשר לתת רמת עדיפות גבוהה יותר לפעולת הכתיבה: הכתיבה יכולה להתבצע גם אם מתבצעת קריאה, וקריאה יכולה להתבצע גם אם מתבצעת כתיבה, אולם במידה והתבצעה כתיבה במשאב בו מתבצעת קריאה, הקריאה תתבצע-מחדש כדי לקבל valid copy של המשאב.

מנעול seqlock מכיל בתוכו מונה-רצף (sequence counter): כל כותב (writer) מעדכן את המונה בתחילת הכתיבה ובתום הכתיבה - מה שמבטיח ערך אי-זוגי במהלך כתיבה, וערך זוגי כשלא מתרחשת כתיבה. על כל קורא (reader) לבדוק את המונה פעמיים - לפני ואחרי קריאת המשאב, ולקבוע האם ערכיו זהים. אם הם שונים, המשמעות היא שהתבצעה פעולת כתיבה תוך כדי הקריאה - והיא זו שהעלתה את המונה - ולכן מה שקרא הקורא איננו תקף עוד.

זהו מנגנון שימושי אם מתקיימים התנאים הבאים:

1. ישנם יותר קוראים מכותבים.
2. הכתיבה תדירה הרבה פחות מאשר קריאה.
3. רק כותב אחד יכול לפעול (יכולים להתקיים כותבים רבים, אך לא לפעול במקביל)
4. אנו מעוניינים לתת עדיפות לכותבים.
5. ה-data איננו פוינטר, מכיוון שייתכן שכותב יכוון-מחדש את הפוינטר בעוד הקורא כבר נסע לכתובת הישנה (כלומר, תתבצע אינוולידציה של הפוינטר).

* אם כן קיים פוינטר, נכון להשתמש במנגנון read-copy update (שאיננו מנגנון נעילה, ולכן לא קראנו עליו).

1. **מהו mutex?**

מיוטקס הוא כלי סנכרון המשמש כמנעול, המציב מגבלה על גישה למשאב בסביבת multi-threading, כך שרק תהליכון אחד יכול לעשות שימוש במוכל לעשות שימוש במשאב.

במידה ותהליכון אחר מנסה אף הוא להינעל במיוטקס, התהליכון השני נכנס להמתנה (blocked), ורק כאשר התהליכון הראשון יפתח את המנעול, התהליכון האחר ייצא מהחסימה ויינעל במיוטקס.

1. **מהו deadlock וכיצד הוא נגרם?**

מצב בו תהליכון אחד או יותר ממתינים לפתיחת מנעול שלעולם לא תקרה.

קיפאון יתרחש רק אם ארבעת התנאים הבאים יתרחשו במקביל:

**1. מניעה הדדית (mutual exclusion) של משאב:** המשאב ניתן לשימוש ע"י תהליך אחד ברגע נתון.

2. **החזק והמתן (hold and wait):** תהליך מחזיק (לפחות) במשאב אחד וממתין למשאב אחר (שכרגע בשימוש על ידי תהליך אחר).

3. **אין הפקעה (no preemption):** המשאב יכול להשתחרר רק על ידי התהליך שמחזיק בו (אחרי שאותו תהליך השלים את עבודתו) [תהליכים אחרים לא יכולים "להפקיע" את המשאב].

4. **המתנה מעגלית (circular wait):** קיימת קבוצת תהליכים ממתינים {p0,p1,...,pn} , כך שכל תהליך pi ממתין למשאב שמוחזק על ידי תהליך pi+1. תהליך pn ממתין למשאב שמוחזק על ידי p0. באופן כזה, קבוצת התהליכים יוצרת מעגל בו כל תהליך מחכה למשאב המוחזק על ידי תהליך אחר.

סוג פשוט של deadlock הוא מצב בו תהליכון נעול על מיוטקס, ואז מבקש להינעל עליו בשנית. במידה וזהו fast mutex, הרי שהתהליכון נכנס להמתנה עד להשתחררות הנעילה הראשונה, אך הוא-הוא זה שביצע את הנעילה הראשונה...לכן הוא תקוע.

דוגמא אחרת היא מצב בו ישנם שני משאבים ושני תהליכונים, וכל תהליכון זקוק לשני המשאבים לשם השלמת פעולתו. כעת, אם תהליכון 1 נועל את משאב א בעוד תהליכון 2 נועל את משאב ב, ותהליכון 1 ממתין לשחרור משאב ב בעוד תהליכון 2 ממתין לשחרור משאב א, הם לא ישתחררו לעולם.

1. **כיצד נימנע מ-deadlock?**
2. **ע"י שימוש ב-lock hierarchy:** טכניקה בה ממספרים את המיוטקסים, וכל התהליכונים מבצעים נעילה ושחרור של כ"א מהמיוטקסים לפי אותו סדר. כמו כן, אם תהליכון מחזיק במנעול שמספרו n, הוא לא יכול לבצע נעילה על מנעול הגדול מ-n.
3. **ע"י שימוש בפונקציית trylock.**
4. **כיצד ממומש mutex?**

נעילת מיוטקס ממומשת באמצעות פקודת Test & Set Lock (פקודת TSL), הבנויה מ-2 שלבים (הדוגמא למטה היא שילוב של טננבאום עמ' 133 והקישור [הזה](http://stackoverflow.com/questions/7863657/what-happens-if-two-process-in-different-processors-try-to-acquire-the-lock-at-e))

1. העתקת ערכו של המיוטקס (המילה MUTEX בדוגמא למטה מייצגת פשוט מקום בזכרון) לרגיסטר, ובאותה שורה – הצבת הערך 1 במיוטקס.
2. השוואת הערך שהוכנס לרגיסטר עם המספר 0:
3. אם מתקיים שוויון – המשמעות היא שערך המיוטקס היה (=הינו) 0, כלומר הוא פנוי, ולכן ניתן לחזור (וכבר שינינו את ערכו ל-1 בשלב מס' 1).
4. אם מתקיים אי-שוויון – המשמעות היא שערך המיוטקס היה (=הינו) 1, כלומר הוא לא פנוי, ולכן:
5. אפשר לתהליכון אחר לרוץ.

* כך נמנע busy waiting.

1. כאשר התהליכון הנוכחי יקבל שוב זמן CPU, מתחילים את הכל מחדש.

שחרור המיוטקס מתבצע בשורה אחת פשוטה: פשוט מאחסנים בו 0.

mutex\_lock:

TSL REGISTER, MUTEX ; copy mutex value to register and set mutex to 1

CMP REGISTER, #0 ; compare mutex to zero (= was mutex 0?)

JZE ok ; if it was 0, it means it is unlocked, so return

(note that we set it to 1 in the #1st step)

CALL thread\_yield ; else: mutex is busy, schedule another thread

JMP mutex\_lock ; try again later (when this thread is rescheduled)

ok: RET ; return to caller, critical region entered

mutex\_unlock:

MOVE MUTEX,#0 ; store a 0 in mutex

RET

1. **מהו semaphore?**

סמאפור הוא data structure הכולל בתוכו 3 רכיבים מרכזיים:

1. counter

2. רשימה של תהליכים/תהליכונים ממתינים

3. 2 פעולות אטומיות: down() and up()

ה-counter מאותחל בתחילת הריצה לערך רצוי כלשהו. פעולת ה-down מקטינה את ערך הסמאפור, ובמידה ובעקבות זאת ערך הסמאפור הוא קטן-שווה (או קטן, תלוי מימוש) ל-0, התהליך/תהליכון שגרם ל-down יצורף לרשימת ההמתנה וייכנס לחסימה.

פעולת ה-up מעלה את ערך הסמאפור, ובמידה וערכו גדול (או גדול-שווה, תלוי מימוש) מ-0, אחד הממתינים ברשימת ההמתנה יתעורר מחדש.

1. **כיצד ממומש סמאפור?**

לא שאלה מהדף, זה בשבילי.

1. **מהם ההבדלים בין mutex vs semaphore?** (נעזרתי גם ב[זה](http://www.linuxdevcenter.com/pub/a/linux/2007/05/24/semaphores-in-linux.html?page=6))

|  |  |  |
| --- | --- | --- |
|  | **מיוטקס** | **סמאפור** |
| **1. מהות** | **מנגנון נעילה** | **מנגנון איתות** |
| **2. מטרה** | **exclusive access:** הגנה על קטע קריטי/משאב משותף | **סינכרון** (i.e. "Hey Someone! This occurred!") |
| **3. בעלות** | **בעלים יחיד:** רק התהליכון שנעל את המיוטקס יכול לשחרר אותו | **אין בעלות:** כל תהליכון יכול לשחרר אותו (כלומר לשנות את ערכו מ-0 לערך השונה מ-0) |
| **4. גישה** | **process-wide** רק תהליכונים שנוצרו ע"י התהליך יכולים לגשת למיוטקס | **system-wide** במידה ונעשה שימוש בסמאפור של Sys V, תהליכים שונים יכולים לקבל גישה לסמאפור, והוא איננו מתנקה אוטומטית ביציאה מהתוכנית |
| **5. גודל קובץ** | **קטן** | **גדול/כבד** |

1. **מתי תשתמש בכל אחד?**

מיוטקס נועד להגנה על משאב משותף מפני שימוש בו-זמני של מספר תהליכונים, ומאפשר רק גישה סדרתית למשאב משותף.

סמאפור מאפשר שימוש מקבילי במשאב משותף, אך מגביל את השימוש המקבילי למספר תהליכונים מקסימלי מסוים.

1. **מהם ההבדלים בין mutex לבין סמאפור בינארי (semaphore of size 1)?**

הבעלות.

1. **מהו barrier?**

מנגנון סנכרון המאפשר לחלק תהליך לשלבים (phases), כך שהמעבר של תהליכון משלב נוכחי לשלב הבא מתאפשר רק כאשר שאר התהליכונים השלימו את השלב הנוכחי.

1. **מהו race condition?**

מצב בו תוצאת פעולה תלויה בדרך בה יתוזמנו שני תהליכים/תהליכונים (או יותר).

במילים אחרות: מצב בו דרך התיזמון תוביל ל-corruption של דאטה.

1. **כיצד נימנע מ-race condition?**

ע"י שימוש במנגנוני נעילה (spinlock, seqlock, מיוטקס, ויש גם פיוסקס שלא למדנו עליו), מנגנוני סנכרון (סמאפור, משתנה מותנה), ועוד שיטות שלא למדנו עליהן, כמו read-copy-update, ראה טננבאום עמ' 148)

1. **אם התוכנית נתקעת בקטע קוד, מהן הסיבות האפשריות לכך?**
2. **קיפאון (deadlock) –** מצב בו שני תהליכים (או יותר) מחכים האחד לאחר לשחרורו של משאב, או למצב בו יותר משני תהליכים מחכים למשאבים בשרשרת מעגלית (ראה שאלה
3. **הרעבה (starvation) –** מצב בו מתהליך מסוים נמנעת גישה לאחד ממשאבי המערכת כך שהתהליך לעולם לא יצליח לסיים את משימתו.

* הרעבה דומה בתוצאתה לקיפאון (Deadlock), אך בהרעבה אחד מהתהליכים כן מסוגל לגשת למשאבי המערכת, בעוד שבקיפאון אף אחד מהם אינו מסוגל לגשת למשאבים. הרעבה מודגמת באמצעות בעיית הפילוסופים הסועדים.

1. **מהי בעיית הפילוסופים הסועדים?**

המחשה לבעיות תזמון וסנכרון המופיעות בהקשרים של עיבוד מקבילי, כאשר מספר תהליכים/תהליכונים רצים יחד וחולקים משאבים משותפים.

תיאור הבעיה:

5 פילוסופים יושבים סביב שולחן עגול ומבלים את זמנם בחשיבה ובאכילה. במרכז השולחן ישנה קערת ספגטי גדולה, ולכל אחד מהפילוסופים יש צלחת. בין כל שתי צלחות נמצא מזלג, כך שישנם בסך הכול 5 מזלגות.

על מנת לאכול את הספגטי, ישנם 4 כללים:

1. כל פילוסוף זקוק לשני מזלגות.
2. כל פילוסוף יכול להרים רק את שני המזלגות הקרובים אליו (אלו שמימין ומשמאל לצלחת שלו).
3. לא ניתן להרים את שני המזלגות בו-זמנית - כלומר, יש להרים את אחד מהם, ורק לאחר מכן את השני.
4. לא מתקיימת כל תקשורת בין הפילוסופים.

שתי הבעיות הבסיסיות שעלולות להתהוות הן קיפאון והרעבה:

1. קיפאון: מתרחש כאשר כל הפילוסופים נתקעים במצב כלשהו ממנו אינם יכולים לצאת. למשל, אם הכלל שעל פיו הפילוסופים פועלים אומר כי כאשר הם רוצים לאכול עליהם להרים את המזלג שמימינם ולחכות עד שהמזלג משמאלם יתפנה, עלול להיווצר קיפאון אם כל הפילוסופים ירצו לאכול בו-זמנית - כולם ירימו את המזלג שמימינם ולאחר מכן יחכו לנצח למזלג שמצד שמאל.
2. הרעבה: מתרחש כאשר אחד או יותר מהפילוסופים אף פעם אינו מצליח לאכול.

למשל, אם דרך הפעולה של הפילוסופים היא להרים את המזלג שמצד ימין, לחכות חמש דקות למזלג שמצד שמאל, ואם הוא טרם התפנה - להניח את המזלג שבצד ימין ולחכות עוד חמש דקות לפני הניסיון הבא, המצב שמתקבל אינו קיפאון (שכן הפילוסופים כל הזמן מורידים ומרימים את המזלגות, ולא נתקעים במצב בודד) אך הוא בבירור הרעבה כי אף פילוסוף אינו מצליח לאכול.

פתרון:

פתרון אחד ידוע (של Dijkstra) הוא להגביל את מספר הפילוסופים היכולים לגשת לשולחן באופן הבא: אם ישנם N מזלגות, אזי נאפשר רק ל-N-1 פילוסופים לגשת לשולחן ולהתחרות עליהם. במצב כזה, לפחות פילוסוף אחד יצליח להרים 2 מזלגות.

פתרון זה ממומש באמצעות סמאפור, המאותחל לערך N-1.

הפתרון נמנע מ-deadlock, אך עדיין תיתכן הרעבה, כאשר פילוסוף אחד לעולם לא יאכל.

(הניסוח נלקח מ[כאן](https://www.cs.indiana.edu/classes/p415-sjoh/hw/project/dining-philosophers/index.htm), שם נטען שגם הפתרון המופיע אצל טננבאום לא נמנע מהרעבה, והאמת היא שטננבאום אכן אומר שהפתרון שלו הוא רק deadlock-free).

1. **האם נעדיף multi-process file system או multi-threaded file system?**

אף אחד לא ידע את התשובה.

1. **מהו critical section?**

קטע קוד העושה שימוש במשאב משותף (שביצועו מוכרח להסתיים לפני שתהליך/תהליכון אחר יוכל להתחילו כדי למנוע race-condition).

1. **אם קיים בקוד critical section, האם יש הבדל בין משתנה מסוג int למשתנה מסוג string?**

לא הצלחתי למצוא הבדל מהותי שקיים דווקא בקטע קוד קריטי, אז אני אעלה השערה כללית: טיפול ב-int הוא מהיר יותר מטיפול ב-string (מקור: [כאן](http://stackoverflow.com/questions/6919380/pointer-vs-variable-speed-in-c), וגם [כאן](http://stackoverflow.com/questions/4904179/why-is-integer-comparsion-faster-then-string-comparion)) מכיוון שגישה למשתנה דרך פוינטר איטית יותר מגישה למשתנה באופן ישיר (בגישה ישירה יש 2 פעולות: חיפוש כתובת המשתנה בזיכרון, והבאתו לרגיסטר; בגישה דרך פוינטר יש 3 פעולות: חיפוש כתובת המשתנה המצביע בזיכרון, הבאתו (כלומר, הבאת הכתובת שהוא מחזיק) לרגיסטר, ואז הבאת הערך המוצבע לזיכרון) ולכן הקטע הקריטי יושלם מהר יותר.

1. **מדוע סמאפור can't be reentrant?**

עדכון: השאלה לא נכונה, והכוונה היא רק לסמאפור בינארי, שגם הוא לא באמת קיים, שהרי הוא יכול לקבל ערכים ששונים מ-1 ומ-0. עם זאת, ניתן לממש סמאפור בינארי באמצעות מיוטקס ו-condition variable (ראה [כאן](http://stackoverflow.com/questions/7478684/how-to-initialise-a-binary-semaphore-in-c)), ואז הוא אכן לא ריאנטרנטי מעצם הגדרתו כבינארי.

1. **מהו reentrant mutex?**

שם נרדף למיוטקס רקורסיבי, המאפשר לתהליכון הנעול עליו (ורק לו) להינעל עליו פעם נוספת אחת או יותר. זהו מיוטקס הכולל בתוכו counter המונה את מספר הנעילות, ולכן יש לפתוח אותו (unlock) בכמות זהה למספר הנעילות. ראה גם שאלה 25, סעיף ד, תת-סעיף 2.

1. **הגדר את המושגים הבאים:**
2. **Condition variable**

כלי סנכרון המאפשר לחסום תהליכונים כאשר תנאי מסוים לא מתקיים, ולאפשר להם להמשיך לרוץ כאשר התנאי מתקיים באמצעות שליחת איתות.

1. **Polling**

פעולה המתרחשת בכל פרק-זמן מסוים ובודקת האם מצב מסוים מתקיים.

אני לא יודע איך זה קשור לנושא (עסקנו בזה כשלמדנו על interrupt vs polling), אבל אם כבר, אפשר [להבהיר](http://programmers.stackexchange.com/questions/193420/best-practise-is-not-to-poll-but-isnt-polling-happening-internally-anyway-whe) שכאשר תהליכון נחסם (blocked) – למשל, כאשר הוא מבקש להינעל על מיוטקס שתהליכון אחר נעול עליו כרגע – אין שום polling שמתבצע ברקע! אלא התהליכון המבקש להינעל מוצב ב-queue, וברגע שהמיוטקס ישוחרר, הוא יוכל להינעל עליו (אם יש תהליכונים אחרים שממתינים, הסדר בו מערכת ההפעלה מוציאה אותם מה-queue אינו צפוי).

1. **Thread-Local Storage (הכוונה היא ל-Thread-Specific Data)**

אזור בזיכרון (שאיננו ה-stack, ולפי התשובה [כאן](http://stackoverflow.com/questions/2459692/linuxs-thread-local-storage-implementation), הוא אזור ייעודי ממש בשם tdata) המאפשר להשתמש במשתנים כך שיהיו לוקאליים לכל התהליכונים שעושים בהם שימוש (= כל תהליכון מקבל רק עותק מהמשתנה).

למשל, אם מספר תהליכונים עושים שימוש במשתנה גלובלי, עלול להיווצר race condition, אותו ניתן למנוע באמצעות נעילה, אך גם באמצעות thread-local variable.

* errno הוא משתנה מסוג זה.

1. **APC**

לא יודע מה הראשי-תיבות.

1. **Reentrancy**

ריאנטרנטיות מתייחסת לאפשרות ביצוע קוד פעמיים או יותר פעמים בו-זמנית מבלי שתיווצר "התנהגות שגויה" (ביטוי זה מתפרש בהתאם להקשר).

1. **Reentrant function**

פונקציה ריאנטרנטית היא פונקציה ש"פועלת בצורה נכונה" כאשר היא נקראת באופן סימולטני ע"י מספר תהליכונים. הביטוי "התנהגות נכונה" מתפרש בהתאם לפונקציה. כך, פונקציות שעושות שימוש במשאבים אשר קריאות סימולטניות לפונקציה עלולה להוביל ל-data corruption צריכות להשתמש במנגנוני הגנה (למשל, מנגנוני נעילה) בכדי להיות ריאנטרנטיות. (ההגדרה הפשוטה להפליא הזאת לקוחה מ[כאן](https://docs.oracle.com/cd/E19120-01/open.solaris/816-5137/guide-3/index.html))

* פונקציה יכולה להיות thread-safe אך לא-ריאנטרנטית. למשל, פונקציה שכל קטע הקוד בה עטוף במיוטקס היא אמנם thread-safe מכיוון שהיא מגנה על משאב משותף כלשהו מפני שימוש של thread אחר בו, אך היא לא-ריאנטרנטית מכיוון שקריאה חוזרת לפונקציה תוביל ל-deadlock.

1. **Reentrant synchronization**

האפשרות להינעל במנעול (mutex) מספר פעמים (למשל, ע"י מיוטקס רקורסיבי, המכונה גם reentrant mutex) מבלי להיכנס לחסימה (וודאי שלא ל-deadlock), ובכך מתאפשרת ריאנטרנטיות.

* האמת היא שזה מונח שרלוונטי בעיקר ל-java, כי שם יש type מיוחד בשם synchronized, אבל כאמור, אצלנו יש מיוטקס רקורסיבי.
* לא ברור המובן של "סנכרון" בהקשר זה, אבל בסדר.