פרק #5: פתילים Threads

5.1 מוטיבציה

נניח שעלינו לכתוב תכנית המבצעת מספר פעילויות במקביל: קוראת קלט מהמקלדת, מהעכבר, ממספר קבצים, מהרשת (ממספר שרתים שונים במקביל); מציגה תמונה על המסך, וכן הלאה.

הרצתה של התכנית כתהליך יחיד תפגע ביכולתה של התכנית לבצע את המשימות הללו *במקביל*: עת ממתינים לקלט מהמקלדת התכנית *חסומה* (blocked), ואינה יכולה בינתיים לבצע פעולה אחרת, באופן דומה עת ממתינים למידע ממקור אחר, או עת מציגים תמונה על המסך.

פתרון אפשרי: האפליקציה (תכנית) שלנו תשריץ מספר תהליכים, כל-אחד מהם יהיה אחראי על תת-משימה אחרת של התכנית. עת אחד התהליכים ממתין\חסום האחרים יכולים להמשיך בפעולתם.

מגבלות הפתרון:

א. בהנחה שהתהליכים השונים מבצעים משימה שלמה אחת יהיה עליהם לחלוק מידע. אומנם ראינו כלים לשם כך, אך לכל אחד מהכלים יש את המגבלה שלו; ודאי שאלה ביניהם שמחייבים ק.מ. יאטו את ביצוע התכנית.

ב. ייצור תהליך הוא פעולה יקרה, שכן לתהליך מוקצים משאבי מערכת רבים (מרחב כתובות, PCB).

ג. בהנחה שחלק מהתהליכים שנייצר יבצעו פעולות דומות (כמה מהם פונים במקביל לרשת, או לקבצים שונים) יהיה להם אותו מקטע קוד\טקסט, ובכך יש משום בזבוז זיכרון.

ד. אם התכנית שלנו תייצר תהליכים רבים כנ"ל (היא שרת) אזי הבזבוז בסעיפים ב', ג' יגדל, ואנו עלולים להציף את המערכת בתהליכים.

מסקנה: נשמח לפתרון פחות יקר.

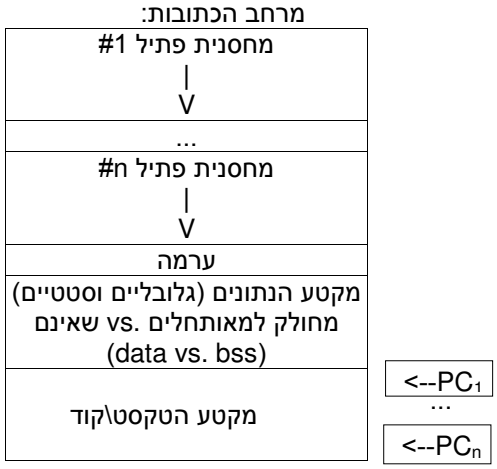
הישועה: פתיל (thread), נקרא גם תהליכון או lightweight process (LWP).

הרעיון העקרוני: תהליך יחיד, הכולל מרחב כתובות יחיד, ושמוקצה לו PCB יחיד, יוכל להריץ פונקציות שונות הנכללות בו במקביל זו לזו. במילים אחרות: עת התהליך (או ליתר דיוק: הפתיל הראשי בתהליך) מזמן פונ' f(), הפתיל הראשי לא יושהה עד לסיום פעולתה של f() כפי שהכרנו עד כה, אלא ימשיך לרוץ במקביל לפונ'.

(כמובן, שאם המחשב שלנו כולל מעבד יחיד אזי המקביליות היא רק ברמה שכעיקרון חסימת הפתיל המבצע את f() אינה צריכה לגרום לחסימתו של הפתיל הראשי. רק אם המחשב שלנו כולל מספר מעבדים המקביליות תוכל להיות 'אמיתית').

התוצאה: על אותם נתונים (מרחב כתובות [בפרט מקטע קוד ומקטע נתונים=משתנים גלובליים], קבצים פתוחים, טפלי סיגנלים) מתבצעות במקביל סדרות פעולות שונות (כלומר מורצות פונ' שונות). כל סדרה מהווה פתיל.
כלל הפתילים המרכיבים אפליקציה= קבוצת פתילים Thread Group.

כדי שכל פתיל יוכל לבצע את הפונ' 'עליה הוא מופקד' מוקצה לכל פתיל PC משלו, וקטע משלו ע"ג מחסנית התהליך (קטע בו יישמרו פרמטרי הפונ', משתני הלוקליים, וכל המידע הנשמר עת מזמנים פונ').



המידע אודות כל פתיל יישמר ב- Thread Control Block.

<u>סיבור העין:</u> ב- Solaris 2: יצירת תהליך לוקחת פי שלושים זמן מיצירת פתיל. החלפת הקשר בין תהליכים לוקח פי חמש זמן מהחלפת הקשר בין פתילים.

כמובן שנקל על הפתילים: (א) לחלוק מידע: בעזרת משתנים גלובליים, (ב) לפנות לאותם קבצים (יש להם טבלת מתארי קבצים אחת).

<u>אתגרים\קשיים הכרוכים בשימוש בפתילים:</u>

א. יש להגן על פתיל מפני משנהו.
ב. יש צורך לסנכרן את פעולת הפתילים עת הם מבצעים משימה משותפת, או מטפלים במשתנים משותפים.
ג. יש חשש ממצב תחרות (race condition): תוצאת ריצתה של התכנית תקבע (גם) על-פי האופן המקרי בו מ.ה. תזמנה את הפתילים השונים, ועל-כן עלולה להשתנות מהרצה להרצה, או לא להיות בהתאם למצופה\נדרש (דוגמה בהמשך).

עד כאן תיארתי את מושג הפתיל בצורה עקרונית, עתה אציג מספר אופנים תיאורטיים שונים בהם ניתן לממש פתילים בפועל. לבסוף, אציג מימוש מסוים נפוץ של פתילים: פתילי POSIX.

5.2 פתילי משתמש לעומת פתילי גרעין
User and Kernek Threads

עד כה הצגתי את הפתילים כמקבילים\חלופיים לתהליכים. למעשה לא תמיד זה המצב: קיימת אבחנה בין פתילי משתמש (שאינם מוכרים לגרעין מ.ה.) לפתילי גרעין המוכרים לגרעין.

עת תכניתנו מייצרת פתילי משתמש, משמעות הדבר שהתכנית מזמנת פונקצית ספריה *במרחב המשתמש*, המייצרת פתיל *שאינו מוכר למ.ה.*. מ.ה. ממשיכה להתייחס לתכנית שלנו כמורכבת מסדרת צעדים אחת.

כיצד אם כן אנו בכל אופן 'נהנים' מהפתילים? עת תכניתנו מתוזמנת לרוץ ע"י מ.ה., מוקצה לכל פתיל בתכנית, על-פי החלטת ספריית הפתילים, חלק מחריץ הזמן שהוקצה לתכניתנו ע"י המערכת. החלפת ההקשר בין הפתילים שקופה לגרעין, הינה באחריות הספרייה, אשר תממש בעצמה את הפעולה: שמירת מצבו של פתיל א' (תוכן האוגרים), המופסק לרוץ, בזיכרון, וטעינת המצב בו הופסק פתיל ב' למעבד לשם הרצתו של ב'.

פתילי גרעין הם החלופה האחרת: הם נתמכים ע"י גרעין מ.ה. אשר מייצרם, מתזמנם, ומנהלם.

כפי שכבר אמרנו הם איטיים יותר ביצירה ובניהול (שכן מחייבים קבלת שירות מהגרעין), אך בהם חסימת פתיל א' של האפליקציה אינה מונעת מפתיל ב' להמשיך לרוץ, ואם המחשב כולל מספר מעבדים ניתן להריץ פתילים שונים במקביל.

המונח lightweight process מתייחס פעמים רבות לפתילי גרעין דווקא.

<u>המשמעות:</u>

א. עת אחד הפתילים בתכניתנו מזמן ק.מ. חוסמת (לדוגמה: כדי לקרוא נתון), מ.ה., שאינה מודעת לכך שתכניתנו מורכבת ממספר פתילים, חוסמת את (כלל) התהליך, וגם פתילים אחרים בתכנית לא יוכלו להמשיך בפעולתם.
ב. במערכת רבת מעבדים, פתילים שונים הנכללים באותה תכנית לא יוכלו לרוץ במקביל.
ג. ומנגד: פעולת ייצור פתיל, או החלפת הקשר בין פתילים, אינן מצריכות שרות של מ.ה., ועל כן הינן יחסית מהירות.
<u>לדוגמה:</u> לינוקס, פנטיום, 700MHz: יצירת\סיום תהליך 251 מיקרו-שנייה, פתיל גרעין: 94 מיקרו-שנייה, פתיל משתמש 4.5 מיקרו-שנייה.
ד. לפחות לכאורה, ספריית הפתילים עשויה לחלק את חריץ הזמן בין הפתילים השונים הנכללים בתהליך בצורה מותאמת יותר מאשר מ.ה..
ה. ספריית פתילי משתמש אינה בהכרח תלוית מ.ה. וע"כ הינה נשיאה (portable) יותר.

5.3 מודלים של ריבוי פתילים

בסעיף הקודם הבחנו בין פתילי משתמש, השקופים לגרעין, לפתילי גרעין המנוהלים על-ידו.

שתי אפשרויות 'קיצוניות' אלה נקראות:

א. מודל של רבים ליחיד (Many-to-One Model), שכן פתילי משתמש רבים בה ממופים לפתיל גרעין יחיד. ו:

ב. יחיד ליחיד (One-to-One Model), שכן כל פתיל משתמש בה ממופה לפתיל גרעין יחיד.

מעבר לשתי האפשרויות הנ"ל קיימת אפשרות שלישית הנקראת: מודל של רבים לרבים (many-to-Many Model). על-פי מודל זה n פתילי משתמש ממופים ל: m (≥ n) פתילי גרעין. (m עשוי להיות תלוי מכונה, או תלוי אפליקציה).

הרעיון: מצד אחד חסימת פתיל כלשהו באפליקציה לא תחסום אותה לחלוטין, m-1 פתילים אחרים בה יוכלו לרוץ; מצד שני: גם אם האפליקציה תייצר שפע של פתילים הדבר לא 'יציף' את הגרעין שמגביל את מספר הפתילים מהאפליקציה אליהם הוא מוכן להתייחס (בפרק 10# נלמד מדוע הגרעין רוצה להגן על עצמו מפני 'הצפה' שכזאת, במילים אחרות: מה רע בכך שהמערכת תריץ מספר רב של פתילים?)

5.4 פתילים, fork() ו- exec()

כזכור, ק.מ. fork() יוצרת תהליך חדש, המהווה עותק של האב.

נניח שתכניתנו מורכבת ממספר פתילים ואחד מהם ביצע fork() מה ייוולד?

תשובות אפשריות:

א. תהליך המריץ את אותם פתילים כפי שהיו באב. כלומר כלל פתילי האב ישוכפלו.

ב. תהליך המריץ רק את הפתיל שביצע את ה-fork()

במקומותינו (לינוקס): ה- fork() משכפל רק את הפתיל שביצע אותו. בפרט, אם ה- fork() בוצע ע"י פתיל משני המריץ פונ' f(), יהיו במערכת:

א. תהליך א' הכולל את הפתיל הראשי (המריץ, למשל את main) ופתיל נוסף (המריץ את f()), ו

ב. תהליך ב' שכולל פתיל יחיד (המריץ את f()).

עתה נניח שאחד הפתילים מבצע exec ומחליף את מרחב הכתובות (שלו?). מה יהיה האפקט על התכנית?

תשובות אפשריות:

א. רק הפתיל הנוכחי מחליף את מרחב הכתובות שלו, ועובר להריץ תכנית חדשה; כל יתר הפתילים ממשיכים לרוץ כרגיל.

ב. כל הפתילים הנכללים בתהליך מופסקים, והתהליך בכללותו 'משנה עורו', ועובר להריץ את התכנית החדשה.

במקומותינו חל מקרה ב', על-כן אם פתיל ראשי הוליד פתיל משני, והפתיל המשני ביצע exec אזי גם ביצועו של הפתיל הראשי ייקטע (שכן 'השטיח' = מרחב הכתובות, 'נשמט' גם מתחת 'לכפות רגליו' = *כל* מרחב הכתובות מוחלף).

לקראת סוף הפרק נראה דוגמה לכך.

5.5 ביטול פתילים Cancellation

ביטול פתיל שקול להריגת תהליך = סיום הפתיל לפני שהוא השלים את ריצתו.

לדוגמה: אם כמה פתילים מחפשים מידע במקביל (בקבצים או ברשת), ואחד מהם מצאו, אזי ניתן לבטל את כל יתר הפתילים.

אם פתיל מסוים טוען דף אינטרנט, והמשתמש מבקש לקטוע את טעינת הדף, ניתן לסיים את הפתיל מיידית.

הפתיל שיש לסיים נקרא פתיל מטרה target thread.

ביטול פתיל עשוי להתבצע באחד משני אופנים:

א. ביטול אסינכרוני (Asynchronous Cancellation): הפתיל מופסק *מיידית* בעת קבלת בקשת הביטול.

ב. ביטול דחוי (Deferred Cancellation): הפתיל יסיים רק עת הוא יגיע 'למקום מתאים'= נקודת ביטול (Cancellation Point), (למשל אחרי שחרור משאבים, או השלמת משימה כלשהי, בפרט עדכון מבנה נתונים).

5.6 טיפול בסיגנלים Signal Handling

נניח שסיגנל נשלח לתהליך הכולל מספר פתילים.

<u>שאלה</u>: מי יקבל את הסיגנל?
תשובות אפשריות:

1. הפתיל לו הסיגנל רלוונטי (זה שחילק באפס).
2. כלל הפתילים (פרט לאלה שהגדירו שהם מתעלמים מהסיגנל, אם אנו מאפשרים הגדרה שכזאת רק לחלק מהפתילים).
3. לפתיל יעודי בתהליך שתפקידו לתפוס סיגנלים.

בלינוקס:

א. סיגנל סינכרוני (שנשלח בשל פעולה שביצע המעבד, למשל חלוקה באפס), נשלח רק לפתיל שביצע את הפעולה. הפתיל עשוי לתפוס את הסיגנל ולטפל בו (כולל למשל לבצע pthread_exit()). אם הסיגנל לא ייתפס, ותבוצע פעולה המחדלית, פעמים רבות היא תסיים את כלל התהליך (כתלות באופייה).
ב. סיגנל אסינכרוני שנשלח לתהליך (למשל c^) ישלח לפתיל כלשהו, בד"כ הראשי, בתהליך. (תוצאתו תהיה דומה למקרה מעל).
ג. הטיפול בסיגנלים הינו אחיד לכלל התהליך, ע"כ אם, למשל, פתיל משני יתעלם מסיגנל, הסיגנל יתעולם לכלל הפתילים בתהליך.

ד. חסימת\מיסוך סיגנל עשויה להיות פר פתיל. במקום להשתמש ב- sigprocmask() נשתמש בפונ' מעט שונה: pthread_sigmask() (שלא אתאר במדויק). באופן כזה קבוצת פתילים עשויה לחסום סיגנל, אחד הפתילים ימתין לַסיגנל (בעזרת sigwait()), וכך יוגדר להיות פתיל יעודי לטיפול בסיגנל.

בהמשך נראה דוגמה לטיפול בסיגנלים.

5.7 מאגר פתילים Thread Pool

אמרנו ששרת רשת עשוי לכלול מספר פתילים (כל פתיל ישרת פניה יחידה). ציינו שיצור פתיל זול יותר מייצור תהליך (ובמקרה זה גם עדיף שכן לכל הפתילים יהיה אותו מקטע טקסט\קוד).

אולם גם ייצור וחיסול פתיל הינו תקורה נוספת, overhead, שנשמח לחסוך; מעבר לכך ייתכן שנרצה להגביל את מספר הפתילים שייוצרו.

<u>הפתרון</u>: מאגר פתילים.

<u>מהות הרעיון</u>: עת התהליך מתחיל הוא מייצר n פתילים שיושמו במאגר פתילים, בו הם ימתינו. עת מגיעה פניה לשרת, הוא מעיר פתיל פנוי מהמאגר, ומעביר לו את הבקשה. עת הפתיל גומר לטפל בבקשה הוא חוזר לשון במאגר (אך אינו מסתיים).

עת מגיעה פניה ואין פתיל פנוי במאגר הפניה תמתין.

מספר הפתילים במאגר.

1. ייקבע סטטית כתלות במספר המעבדים, גודל הזיכרון, מספר הפניות בהן נרצה לטפל במקביל.
2. ייקבע דינאמית ע"פ העומס במערכת (ככל שהיא יותר עמוסה נאפשר לייצר פחות פתילים).

5.8 נתונים פרטיים, 'גלובליים' לפתיל (Thread-Specific Data, TSD)

כזכור, כלל הפתילים בתהליך חולקים את מקטע הנתונים של התהליך – כך נקל עליהם לחלוק מידע.

לעתים נרצה *שלכל* פתיל יהיה עותק משלו של משתנים כלשהם, אך כזה שלא ישמר ע"ג המחסנית, ועל-כן יהיה מוכר רק בפונ' הראשית של הפתיל, אלא יוכר בכל הפונ' שהפתיל מריץ— יהיה 'גלובלי' לפתיל, אך שונה מפתיל לפתיל. מידע זה נקרא ספציפי או פרטי (private).

לדוגמה: *לכל* פתיל נרצה שיהיה עותק משלו של errno כך ששינוי בערכו של המשתנה עבור פתיל א', לא יראה בפתיל ב'. או
אם כל פתיל שולח פלט לקובץ או לחלון נפרד, אזי מתאר הקובץ או החלון צריך להיות פרטי וייחודי לכל פתיל.

האחריות לכך ש- errno הוא ספציפי לכל פתיל הינה של מ.ה., האחריות לכך שמתאר הקובץ\חלון יהיה פרטי לכל פתיל מוטלת עלינו, וכדי להשיג זאת נזדקק ל- TSD.

מכיוון שכלל הפתילים רשאים לפנות לכלל מרחב הכתובות של התהליך, אזי בעיקרון כל פתיל יכול לפנות לנתונים של פתיל עמית, אולם לא ניתן לו 'קצה חוט' לכך.

ברוב ספריות הפתילים, בפרט pthread אותה נכיר, ניתן להקצות זיכרון כך *שלכל* פתיל יהיה *עותק נפרד* של זיכרון זה. נראה את המימוש הטכני של הדברים בשלהי הפרק.

5.9 Pthreads (פתילי POSIX)

כמו הסטנדרטים האחרים הנכללים ב- POSIX, גם pthread הוא תקן המגדיר API ליצירת וסינכרון פתילים. התקן אינו קובע כיצד ימומשו הפתילים, בפרט באיזה מידה הם יוכרו ע"י הגרעין. בלינוקס, בעזרת pthread אנו מייצרים פתילי משתמש.

בניגוד לפקודת ה- fork() בה *התהליך* החדש מתחיל את ריצתו בפקודה העוקבת לפקודת ה- fork(), עת נשתמש ב- pthread_create()
כל פתיל חדש מתחיל את ריצתו בפונ' שמצביע אליה, והארגומנטים לה, מועברים ל- pthread_create().

ה- include הדרוש:

```
#include <pthread.h>
```

ומה שחשוב מכך, פקודת הקומפילציה:

```
gcc –Wall –o my_prog –l pthread
                my_prog.c
```


5.9.1 צעדים ראשונים: ייצור וסיום פתיל

כדי לייצר פתיל נשתמש בפונ':

```
pthread_t thread_data ;
int a = 1,
    b = 2 ;
int status ;

status = pthread_create(&thread_data,
                        NULL,
                        my_func,
                        (void *) &a) ;
if (status != 0) {
    perror("pthread_create failed in main") ;
    exit(EXIT_FAILURE) ;
}
```

נסביר:

א. pthread_t thread_data ; הוא טיפוס (מספר שלם או מבנה) שמכיל את מזהה הפתיל יחסית לתהליך (ולא כללית במערכת).
ב. בפרמטר השני יש להעביר דגלים המציינים כיצד יש לייצר את הפתיל. נהוג להעביר NULL בארגומנט זה, כלומר לבחור במחדלים (זהו פתיל משתמש, שניתן להמתין לסיומו). בסעיף 5.9.5 נציג אפשרויות אחרות.
ג. בפרמטר השלישי מועבר מצביע לפונ' שתהווה את קוד הפתיל: ממנה הפתיל מתחיל את דרכו.

ד. הפרמטר האחרון הוא מצביע מטיפוס void * לַארגומנטים לַפונ' המהווה את קוד הפתיל. כלומר פונ' זו מקבלת בהכרח מצביע יחיד, מטיפוס void *, לו היא, כמובן, עושה cast לטיפוס הרצוי, ופונה לארגומנטים שהוסכם שיועברו לה. בדוגמה שלנו מועבר לפונ' int יחיד. (הפונ' המהווה את קוד הפתיל מחזירה בהכרח void *, כפי שעוד נראה, כלומר הפרוטוטיפ שלה יהיה: void *f(void *params))

הפונ' pthread_create מחזירה אפס בהצלחה, ערך שונה מאפס עת יצירת הפתיל נכשלה.

בשלב זה, קצת כמו אחרי ביצוע fork(), בתכנית שלנו רצים במקביל שני פתילים: פתיל ראשי, הממשיך בביצוע ה- main (או הפונ' שהורצה קודם לכן), ופתיל משני המריץ את my_func.

שני הפתילים נכללים באותו תהליך ולכן בד"כ (כפי שנסביר) יש להם אותו pid, לכל אחד מהם יש מספר פתיל שונה (שאינו מוכר למ.ה., אלא רק לספריית הפתילים).

הפקודה:

pthread_exit(NULL) ;

מסיימת את ביצוע הפתיל שזימן אותה (כולל הפתיל הראשי, אם הוא ביצָעה). אם כל הפתילים בתכנית מבצעים פקודה זאת אזי ערך ההחזרה של התהליך יהיה בהכרח אפס.

כפי שאמרנו, הפונ' הראשית של כל פתיל מחזירה בהכרח void*, ובפקודת ה- pthread_exit אנו גם מחזירים את המצביע הדרוש (בפרט NULL אם אין לנו מה להחזיר).

נראה דוגמה קטנה של תכנית שלמה, והרצתה:


```
// file:  pthread1.c
// a first thread example.
// compile: gcc -Wall -lpthread pthread1.c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>


void* my_func(void *) ;
//-------------------------------------------------------

int main() {
   pthread_t thread_data ;
   int a = 1,
       b = 2 ;
   int status ;

   srand((unsigned)time(NULL)) ;

   status = pthread_create(&thread_data,
                           NULL,
                           my_func,
                           (void *) &a) ;
   if (status != 0) {
      fputs("pthread_create failed in main", stderr) ;
      exit(EXIT_FAILURE) ;
   }

   my_func( (void *) &b) ;

   puts("we do not reach this place") ;
   return(EXIT_SUCCESS) ;
}
```

למה fputs(), ולא perror()?

גם הפתיל הראשי פונה לביצוע my_func אך מעביר לה מצביע ל- b (שערכו 2)

ומייד נשאל: למה?


```
//-------------------------------------------------------
void* my_func(void * args)  {
   int i,
       *val = (int *) args ;

   for (i= 0; i< 5; i++) {
      printf("process = %d thread= %d (%u) i= %d\n",
             getpid(),
             *val,
             (unsigned int) pthread_self(),
             i) ;
      sleep(rand()%18) ;
   }

   pthread_exit(NULL) ;
}
//-------------------------------------------------------

/* A run:
 ========
<219|1>yoramb@inferno-05:~/os$ !g
gcc -Wall pthread1.c -lpthread
<220|0>yoramb@inferno-05:~/os$ !a
a.out
process = 21937 thread= 2 (3086653120) i= 0
process = 21937 thread= 1 (3086650256) i= 0
process = 21937 thread= 1 (3086650256) i= 1
process = 21937 thread= 2 (3086653120) i= 1
process = 21937 thread= 2 (3086653120) i= 2
process = 21937 thread= 1 (3086650256) i= 2
process = 21937 thread= 2 (3086653120) i= 3
process = 21937 thread= 2 (3086653120) i= 4
process = 21937 thread= 1 (3086650256) i= 3
process = 21937 thread= 1 (3086650256) i= 4
<221|0>yoramb@inferno-05:~/os$
*/
```

ערכו 2 בפתיל הראשי, 1 בפתיל הנוסף

השקול ל- getpid()

מסיים הפתילים (במקרה שלנו, גם את הראשי)

<u>הערות</u>

1. אמרנו שהטיפוס pthread_t המחזיק את מזהה הפתיל עשוי להיות מבנה או פרימיטיבי (שלם). לעתים עלינו להשוות מספרי פתילים (למשל כדי לדעת האם הפְּנִיה הנוכחית היא אלי). הפונ':

```
int pthread_equal(pthread_t tid1,
                  pthread_t tid1) ;
```

משווה מספרי פתילים.
מחזירה אפס עת הם שווים, ערך $\neq$ 0 עת שונים.

2. בדוגמה שראינו לשני הפתילים היה אותו מספר תהליך. בלינוקס זה לא חייב להיות כך. מימוש pthread_create עשוי להיות באמצעות ק.מ. clone() הייחודית ללינוקס, עליה נדבר בהמשך, ואשר עשויה לצור תהליך חדש החולק משאבים עם אביו כך שהינו למעשה פתיל.

3. exit() מסיימת תהליך שלם. על כן אם אחד הפתילים יבצע אותה כלל התהליך יסתיים, כולל כל הפתילים הנכללים בו. שימו לב שבדוגמה שלנו גם הפתיל הראשי מבצע pthread_exit() עת הוא מריץ את my_func, בפרט הוא לא מגיע לבצע את פקודת הפלט שבסופו, ואת ה- return(EXIT_SUCCESS);

4. ראינו את pthread_exit() המסיימת פתיל (תוך החזרת ערך מטיפוס void *).
לחילופין: הפתיל רשאי לבצע פקודת return מהפונ' עימה הוא נוצר, תוך שהוא מחזיר ערך מטיפוס void * .
לדוגמה: return((void *) &num)
)
יש רק לתת את הדעת היכן הוקצה num ומה קורה לו אחרי סיום הפתיל. אך זה נכון גם לגבי pthread_exit(). לכן אולי סביר שנרצה להגדיר:

```
int *ret_val = (int *) malloc(sizeof(int))
```

להכניס ל: ret_val* את ערך ההחזרה של הפתיל:

```
*ret_val = 0 ;
```

ולהחזיר את ret_val (void *):

```
pthread_exit((void *) ret_val) ;
```

).

<u>5.9.2 ביטול פתילים</u>

כפי שראינו שתהליך א' יכול ל(נסות ו)להביא לסיומו של תהליך ב', באמצעות הפקודה:

```
int kill(pid_t pid, int sig);
```

(כמובן שלשם כך על ההורג לדעת מהו ה- pid של המחוסל)
כך גם בין פתילים: פתיל א' יכול להביא לסיומו של פתיל ב'. הפקודה המתאימה:

```
pthread_cancel(thread_data) ;
```

(בהנחה ש: pthread_t thread_data מחזיק את מזהה הפתיל שמעוניינים לסיים).

כמו שבין תהליכים, פקודת ה- kill לא בהכרח תביא לסיומו של התהליך, כך בין פתילים יתכנו מספר אפשרויות (דומות אך שונות מכפי שקורה בין תהליכים):

א. הפתיל יסיים מיידית, אסינכרונית.
ב. הפתיל יתעלם מהבקשה לסיים.
ג. הפתיל יענה לבקשה לסיים, אך יבצע אותה באופן דחוי, עת הוא יגיע ל<u>נקודת ביטול</u> (cancellation point). (מייד נרחיב על כך).

כיצד נקבע מה תעולל בקשת סיום לַעולַל?
באמצעות הפונ': pthread_setcancelstate() אותה יכול להריץ פתיל, וכך לקבוע מה תהיה תגובתו לבקשת ביטול.
האפשרויות:

א. כדי לגרום לו לסרב לבקשות ביטול עתידיות יבצע הפתיל:

```
int old ;
pthread_setcancelstate(
    PTHREAD_CANCEL_DISABLE,
    &old) ;
```

ל- old יוכנס מצב הביטוליות הקודם.

ב. כדי לגרום לו להיענות לבקשות ביטול (באופן סינכרוני או אסינכרוני, כפי שנקבע בפקודה אחרת, אותה נראה מייד) יבצע הפתיל:

```
pthread_setcancelstate(
    PTHREAD_CANCEL_ENABLE,
    NULL) ;
```

(זוהי גם ברירת המחדל.)

כדי לקבוע האם הסיום יהיה מיידי (אסינכרוני) או דחוי (סינכרוני) נשתמש בפקודה:

```
pthread_setcanceltype()
```

באופן הבא:

```
pthread_setcanceltype(
    PTHREAD_CANCEL_ASYNCHRONOUS, NULL) ;
```

או:

```
pthread_setcanceltype(
    PTHREAD_CANCEl_DEFERRED, NULL) ;
```

המחדל: אם לא נקבע את תגובת הפתיל לבקשות בטול (באמצעות: pthread_setcancelstate()), ולא נקבע באיזה אופן יבוצע הביטול (באמצעות: pthread_setcanceltype()), אזי הפתיל ניתן לביטול דחוי (סינכרוני).

נניח שטיפוס הביטוליות של הפתיל נקבע (מחדלית, או על-ידינו) להיות דחוי\סינכרוני. מה תהינה נקודות הביטול בהן הפתיל יסיים עת הוא קיבל בקשת ביטול?

א. בכל מקום בו הפתיל מבצע ק.מ. חוסמת.
ב. עת הפתיל מבצע: pthread_testcancel(); המערכת בודקת האם ממתינה לפתיל בקשת ביטול, ואם כן מסיימת אותו.
ג. עת הפתיל מזמן פונ' שונות לטיפול בפתילים כדוגמת pthread_joint (שנראה בקרוב).

5.9.3 'destructor' לפתיל

כפי שעבור תהליכים ביכולתנו לקבוע, באמצעות הפונ' atexit() שעת התהליך מסיים יש להריץ פונקציה\יות מסוימות (אם נכתוב: atexit(f); עבור f שהינה פונ' void f() , אזי לפני סיומו של התהליך שלנו תזומן הפונ' f)

כך גם עבור פתיל ביכולתנו לקבוע 'destructor' לפתיל. המנקה הוא פונ' המקבלת כפרמטר מצביע מסוג void * ומחזירה void.

ה-'מנקה' יקרא עת:

א. הפתיל מבצע pthread_exit() (אך לא אם הוא מבצע return).
ב. הפתיל מגיב לבקשת ביטול.
ג. הפתיל קורא מפורשות לפונ' ניקוי עם ארגומנט $\neq 0$.

נדגים:

בתחילת הפתיל נכתוב, לדוגמה:

```
pthread_cleanup_push(
                cleanup_msg,
                (void *) "thank you, and come again");
pthread_cleanup_push(cleanup_malloc, (void *) p);
```

ובהמשך, באותו קינון סוגריים (ומעבר לכך לא משנה היכן) נכתוב:

```
pthread_cleanup_pop(0);
pthread_cleanup_pop(0);
```

התוצאה: עת הפתיל יסיים באחד האופנים שתיארנו מעל, ראשית תקרא הפונ' cleanup_malloc ויועבר לה: p (void *) ; ושנית תקרא: cleanup_msg ולה יועבר: "thank you, and come again" (void *) .

נראה תכנית שלמה לדוגמה:

```
// file: pthread_cleanup2.c
// a program that creates a thread.
// The thread runs until the main thread cancels it.
// The secondary thread pushes and then pops functions
// that are executed when it terminates

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   // for sleep()

void *thread_func(void *parm) ;
void cleanup_malloc(void *arg) ;
void cleanup_msg(void *arg) ;
//-----------------------------------------------------------------
int main(int argc, char **argv)
{
  pthread_t            thread;
  int                  rc;

  srand(time(NULL)) ;

  puts("creating the thread") ;
  rc = pthread_create(&thread, NULL, thread_func, NULL);
  if (rc) {
     fputs("pthread_create() failed", stderr) ;
     exit(EXIT_FAILURE) ;
  }
  sleep(2);

  puts("cancelling the thread");
  rc = pthread_cancel(thread);

  sleep(2);
  puts("main finishing") ;
  return EXIT_SUCCESS;
}
```

הפתיל המשני מבוטל בשל בקשה מהפתיל הראשי. (האם ומדוע הבקשה תכובד?)


```
//-----------------------------------------------------------------
void *thread_func(void *parm)
{
  puts("in secondary thread");
  char *p = (char *) malloc(10 *sizeof(char)) ;

  pthread_cleanup_push(cleanup_msg,
                 (void *) "thank you, and come again");
  pthread_cleanup_push(cleanup_malloc, (void *) p);

  if (p == NULL) {
     puts("malloc failed") ;
     exit(EXIT_FAILURE) ;
  }

  while (1) {
    puts("secondary thread run") ;
    if (rand() % 10 < 1)
      return NULL;

    //pthread_testcancel();

    sleep(1);
  }
  pthread_cleanup_pop(0);
  pthread_cleanup_pop(0);
  return NULL;
}
```

קביעת המהרסים והארגומנטים להם

אם הפתיל מסיים ע"י return לא מורץ המהרס

אם נשלחה בקשת ביטול, אזי ביצוע של אחת משתי פקודות אלה יגרום לפתיל לסיים, תוך הרצת המהרס

נדרש מצרכים תחביריים. חסר משמעות בהרצה.


```
//-----------------------------------------------------------------
void cleanup_msg(void *arg)
{
  puts("In the cleanup_msg");
  puts((char *) arg) ;
}
//-----------------------------------------------------------------
void cleanup_malloc(void *arg)
{
  puts("In the cleanup_malloc");
  free( (char *) arg ) ;
}
//-----------------------------------------------------------------
/* a run:
   ======
<219|1>yoramb@inferno-05:~/os$ !g
gcc -Wall pthread_cleanup2.c -l pthread
<220|0>yoramb@inferno-05:~/os$ a.out
creating the thread
in secondary thread
secondary thread run
secondary thread run
cancelling the thread
In the cleanup_malloc
In the cleanup_msg
thank you, and come again
main finishing
<221|0>yoramb@inferno-
05:~/os$<219|0>yoramb@inferno-05:~/os$ a.out
creating the thread
in secondary thread
secondary thread run
cancelling the thread
main finishing
*/
```

עת הפתיל המשני מסיים בשל בקשה מורצים המהרסים

כאן הפתיל המשני מסיים מיוזמתו עם return, ואין מהרס

5.9.3 המתנה לסיום פתיל ופתילים מנותקים

כמו שתהליך אב יכול לחכות לבנותיו (ע"י wait(), waitpid()), כך גם בפתילים, פתיל א' עשוי להמתין לפתיל ב' (שאינו בהכרח ילדו).

בפרט, אם פתיל א' הוא הפתיל הראשי, אזי ייתכן שלפני שהוא מבצע return(0); ובכך מסיים את כל התהליך, הוא ממתין לילדיו, כדי לא לקטוע את פעולתם.

המְתנה לפתיל נעשית ע"י הפונ': pthread_join() המקבלת את הפרמטרים הבאים:

1. מזהה הפתיל לו יש להמתין (מטיפוס pthread_t).
2. מצביע מטיפוס void ** שהמצביע עליו הוא מורה יעודכן ע"י הפונ' כך שהוא יופנה למקום בו יאוחסן ערך ההחזרה של הפתיל לו ממתינים (ניתן להעביר כאן NULL, או &ret_val, עבור: int *ret_val; . אחר-כך ret_val* יכיל את ערך ההחזרה של הפתיל).

הפונ' מחזירה אפס בהצלחה, ערך ≠ 0 בכישלון.

כלומר ההמתנה היא לפתיל מסוים (ולא לכל פתיל שהוא, כמו ב- wait()).

כמו ב- wait(), ההמתנה משחררת את המידע המוחזק אודות הפתיל *בספריית הפתילים* (שיצרה את הפתיל, ושומרת את הנתונים אודותיו, שכן אנו דנים בפתילי משתמש. המידע אודות הפתיל הוא בדומה לנתונים שמ.ה. שומרת אודות כל תהליך, ושיש למחזר).

כדי שניתן יהיה להמתין על פתיל הוא חייב להיות במצב joinable (בר-צירוף), כפי שנסביר מייד (כמו שעבור תהליך, ממתינים עליו רק אם אביו לא ניער חוצנו ממנו. להזכירכם, בתהליכים, האב יכול לקבוע שהוא מתעלם מ- SIG_CHLD ואז אין אומרים אחריו קדיש. אולם בתהליכים, בניגוד לפתילים, ההתעלמות היא גורפת לכל הילדים).

המתנה על פתיל יותר מפעם יחידה מחזירה כישלון (ערך שונה מאפס).

<u>ניתוק פתיל</u>

אמרנו שעל פתיל שאינו מנותק ניתן, וגם ראוי להמתין. עתה נתאר כיצד ניתן לנתק פתיל: להפכו <u>למנותק</u> (detached), כך שהוא לא יהיה <u>בר-צירוף</u> (joinable), ולא יהיה לא צורך ולא אפשרות להמתין עליו.

פקודת הניתוק:

pthread_detach(a_thread) ;

עבור pthread_t a_thread שמחזיק את מזהה הפתיל שיש לנתק.

ניתוק של פתיל יכול להיעשות ע"י הפתיל עצמו, אימו, או כל פתיל שמכיר את מזההו. (הזכרנו את הפונ': pthread_self() אשר מחזירה את מזהה הפתיל שזימן אותה [מקבילה ל: getpid()], ואת הערך המוחזר על-ידה יכול הפתיל להעביר ל: pthread_detach()).

נראה דוגמה:


```
// file:  pthread3.c
// an example of pthread_detach()
// detached thread continues to run after main thread
// made pthread_exit.
// output in file pthread3.txt
// compile: gcc -Wall -lpthread pthread3.c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int arr[5] = {0, 0, 0, 0, 0} ;

void* my_func(void *) ;
//-------------------------------------------------------

int main() {
   pthread_t thread_data[5] ;
   int ids[5] = {0, 1, 2, 3, 4} ;
   int i ;
   int status ;

   for (i = 0; i < 5 ; i++) {
         status = pthread_create(&(thread_data[i]),
                                 NULL,
                                 my_func,
                                 (void *) &(ids[i])) ;
         if (status != 0) {
            perror("pthread_create failed in main") ;
            exit(EXIT_FAILURE) ;
         }
         if (i == 0)
             pthread_detach(thread_data[0]) ;
   }
```

מערך גלובלי, השמור במקטע הנתונים, ומוכר לכל הפתילים (בניגוד למידע השמור על מחסנית כל פתיל)

הפתיל הראשי מייצר חמישה פתילים, 'מזהה' כל אחד מהם הוא 0, 1, 2, ... על-פי סדר יצורם, וערכם של תאי המערך. מייד נראה מה עושות הבנות.

למה שלא נעביר כאן פשוט את &i?

לשם ההדגמה, את פתיל 0#, ורק אותו, הפתיל הראשי מנתק.

עתה הפתיל הראשי ממתין על חמשת ילדיו. מה יקרה עת נמתין על פתיל 0#?

```
   for (i= 0; i< 5; i++) {
      status = pthread_join(thread_data[i], NULL) ;
      printf("join on %d returned %d \n", i, status) ;
   }

   for (i= 0; i< 5; i++)
      printf("cell = %d, val = %d\n", i, arr[i]) ;

   puts("Main thread finishes") ;
   pthread_exit(NULL) ;

   return(EXIT_SUCCESS) ;
}
```

אחרי ההמתנה לילדים, מדפיס הפתיל הראשי את ערכם של תאי המערך arr, ומסיים בעצמו עם pthread_exit (ולא עם exit !)

לכאן איננו מגיעים, וטוב שכך, שכן אז היינו 'מחריבים' גם את המנותקת

```
//-------------------------------------------------------
void* my_func(void * args)  {
   int i,
      id = *((int *) args) ;

   for (i= 0; i< 10000; i++) {
      arr[i % 5] += id ;
   }

   if (val == 0) {
       puts("the detached") ;
       for (i = 0; i < 10; i++) {
          puts("I am still alive") ;
          sleep(1) ;
       }
       puts("The detached finishes") ;
   }

   pthread_exit(NULL) ;
}
```

מה עושה כל פתיל? אלפיים פעם מוסיף לכל אחד מתאי המערך את מזההו (0..4). לכן בכל תא במערך אמור להיות: =2000*(0+1+2+3+4) 20000

הבת הסוררת, מעבר לכך עוד ממשיכה להשתולל (וקצת לשון)

```
/* a run:
  ======
join on 0 returned 22
the detached
I am still alive
join on 1 returned 0
join on 2 returned 0
join on 3 returned 0
join on 4 returned 0
cell = 0, val = 19934
cell = 1, val = 19941
cell = 2, val = 19938
cell = 3, val = 19960
cell = 4, val = 19935
Main thread finishes
I am still alive
I am still alive
I am still alive
I am still alive
I am still alive
I am still alive
I am still alive
I am still alive
I am still alive
The detached finishes

*/
```

המתנה על הבת #0 נכשלת, ועל יתר אחיותיה מצליחה

מסיבה בלתי מובנת, ערכם של תאי המערך אינו 20000

למרות שהפתיל הראשי סיים (עם pthread_exit) הבת #0 ממשיכה בחייה בכייף.

עד סיומה

5.9.4 פתילים ומצב מרוץ

בסעיף זה נרצה להבין את האנומליה שראינו בדוגמה הקודמת: מדוע ערכם של תאי המערך לא היה 20000 כמצופה?

ראשית, מישהו עשוי לטעון שהסיבה לכך היא שהפתיל הראשי לא המתין לבתו המנותקת (#0), והדפיס את ערכם של תאי המערך בטרם עת.
לכאורה זו טענה נכונה, אולם מכיוון שהסוררת הייתה אפס גמור, אזי לא זה ההסבר לתקלה.

ניזכר שוב כי כדי לבצע פעולה: var += val; עושה מחשב למעשה שלוש פעולות:

$reg_0 \leftarrow var$ (ואפילו זו לא בהכרח פעולה אטומית!)
$reg_0 = reg_0 + var$
$var \leftarrow reg_0$

עתה נניח כי ערכו של var הוא אפס, וכי פתיל א' מעוניין להגדילו באחד, ופתיל ב' מעוניין להגדילו בשתיים (כך שערכו הסופי אמור להיות שלוש).

נניח כי ספריית הפתילים מתזמנת את שני הפתילים (השייכים לתהליך יחיד) באופן הבא:

פתיל א'	פתיל ב'
$reg_0 \leftarrow var$ (=0)	
reg_0+= 1	
	$reg_1 \leftarrow var$ (=0)
	reg_1+= 2
$var = reg_0$ (=1)	
	$var = reg_1$ (=2)

התוצאה: ערכו של var הוא 2 (כלומר ערך שגוי)!

<u>הסיבה</u>: הפתילים לא תיאמו\סנכרנו את הגישה לתא הזיכרון var, ועל כן תוצאת הריצה נקבעת על-ידי התזמון *המקרי* בו ספריית הפתילים הריצה את שני הפתילים (התוצאה עשויה הייתה להיות גם נכונה לו ספריית הפתילים הייתה מתזמנת ראשית את שלוש הפעולות של אחד הפתילים, ורק שנית את שלוש הפעולות של משנהו).

למצב כזה אנו קוראים <u>מצב מרוץ</u>: בו תוצאת ההרצה אינה דטרמיניסטית, אלא נקבעת באופן מקרי על-פי התזמון בו הורצו התהליכים\פתילים.

<u>אחריותנו</u>: לכתוב קוד שהינו multi-thread safe כלומר כזה בו לא יחול מצב מרוץ. (בפרק #7 נכיר את הסמפור שיעזור לנו בכך).

5.9.5 קביעת תכונות של פתיל

ראינו כי יצירה של פתיל נעשית באמצעות הפקודה pthread_create(). ציינו כי הפרמטר השני של הפקודה מאפשר לנו לקבוע את תכונותיו של הפתיל הנוצר, וכי מחדלית נהוג להעביר בארגומנט המתאים ערך NULL כדי לצור פתיל עם תכונות מחדליות.

עתה נכיר כמה תכונות שניתן לקבוע באמצעות הפרמטר.

ראשית, נזדקק למשתנה לתוכו נזין את התכונות שברצוננו לקבוע (שתהיינה לפתיל שנייצר):

```
pthread_attr_t attr ;
```

נאתחלו לכדי התכונות המחדליות:

```
pthread_attr_init(&attr);
```

עתה, כדי לקבוע מפורשות שהפתיל יהיה בר-צירוף נכתוב:

```
pthread_attr_setdetachstate(&attr,
                            PTHREAD_CREATE_JOINABLE);
```

ולבסוף נייצר את הפתיל (תוך העברת attr בארגומנט השני, ולא NULL כפי שעשינו עד כה):

```
status = pthread_create(&thread_data,
                        &attr,
                        my_func,
                        NULL) ;
```

תכונה אפשרית אחרת של הפתיל שאנו עשויים לרצות לקבוע היא גודל המחסנית שלו:

```
pthread_attr_setstacksize( &attr,
                           new_size) ;
```

(הפונ': pthread_attr_getstacksize() , שלא אתאר במפורט, מחזירה, בפרמטר המועבר לה, את גודל המחסנית הצפוי לפתיל שייוצר).

הדוגמה הפשוטה למדי בשקף הבא, מציגה שימוש ב- pthread_attr_setdetachstate() .


```
// file: pthread_att.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

void* my_func(void *) ;
//-------------------------------------------------------
int main() {
   pthread_t thread_data ;
   pthread_attr_t attr;
   int status ;

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,
            PTHREAD_CREATE_JOINABLE);
   status = pthread_create(&thread_data,
                           &attr,
                           my_func, NULL) ;
   if (status != 0) {
      fputs("pthread_create failed in main", stderr) ;
      exit(EXIT_FAILURE) ;
   }
   status = pthread_join(thread_data, NULL) ;
   if (status) {
      fputs("pthread_join failed in main", stderr) ;
      exit(EXIT_FAILURE) ;
   }
   puts("Done\n") ;
   return(EXIT_SUCCESS) ;
}
//-------------------------------------------------------
void* my_func(void * args)  {
   puts("I am a secondary thread\n") ;
   return(NULL) ;
}
```


5.9.6 אִתחול יחיד עבור קבוצת פתילים

לעתים ברצוננו לבצע אתחול כלשהו לא עבור פתיל בודד בתכנית, אלא עבור כלל הפתילים הנכללים בתכנית (לדוגמה: הקצאת זיכרון שישמש את כלל הפתילים, או אתחול מנגנון ייצור המספרים האקראיים). את האתחול נרצה לעשות פעם יחידה עבור כלל הפתילים (ולא עבור כל פתיל בנפרד.)

הכלי ש- pthread מעמידה לרשותנו הוא pthread_once() . פונ' זו מאפשרת לנו להריץ פונ' רצויה כלשהי פעם יחידה עבור כלל הפתילים בתכנית.

מעבר לַפונ' שנרצה להריץ פעם יחידה (ושנניח שקראנו לה: my_init) נזדקק למשתנה *גלובלי* (או סטטי) אשר יאותחל כמתואר:

```
pthread_once_t  threads_init =
                PTHREAD_ONCE_INIT;
```

ואשר ידאג לכך שהפונ' אכן תורץ רק פעם יחידה, גם אם היא מזומנת מספר פעמים (למשל, ע"י כ"א מהפתילים).

בפונ' הראשית של הפתילים נזמן:

```
pthread_once(&threads_init, my_init) ;
```

עבור my_init שמבצעת איתחולים שונים, וברצוננו להריצה פעם יחידה עבור כלל הפתילים. my_init אינה מקבלת פרמטרים, ואינה מחזירה ערך. תפקידו של threads_init לדאוג לכך ש- my_init אכן תזומן פעם אחת בדיוק. (כל משתנה מטיפוס pthread_once_t ידאג לכך שפונ' רצויה כלשהי תורץ פעם יחידה.)

נראה דוגמה:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

void* my_func(void *) ;
void init() ;

int *arr ;
int counter = 0 ;

pthread_once_t threads_init = PTHREAD_ONCE_INIT;
//-------------------------------------------------------
int main() {
   pthread_t thread_data[5] ;
   int i ;
   int status ;

   for (i = 0; i < 5 ; i++) {
         status = pthread_create(&(thread_data[i]),
                                 NULL,
                                 my_func, NULL) ;
         if (status != 0) {
            fputs("pthread_create failed in main", stderr) ;
            exit(EXIT_FAILURE) ;
         }
   }

   for (i= 0; i< 5; i++)
      pthread_join(thread_data[i], NULL) ;

   for (i= 0; i< counter; i++)
      printf(" %d", arr[i]) ;
   puts("\nDone") ;

   return(EXIT_SUCCESS) ;
}
```

הפתילים ישתמשו ב- arr,
ולכן יש, ראשית, להקצותו,
אך לעשות זאת פעם יחידה


```
//-------------------------------------------------------
void* my_func(void * args)  {
   pthread_once(&threads_init, init) ;

   // SHOULD BE DONE IN A CRITICAL SECTION!!!
   arr[counter++] = rand() % 10 ;

   pthread_exit(NULL) ;
}
//-------------------------------------------------------
void init() {
   puts("In init()") ;
   srand(time(NULL)) ;
   arr = (int *) malloc(5 * sizeof(int)) ;
   if (!arr) {
      perror("malloc() failed\n") ;
      exit(EXIT_FAILURE) ;
   }
}
//-------------------------------------------------------
/*
   a run:
   ======
<242|0>yoramb@inferno-05:~/os$ !a
a.out
In init()
 3 2 0 8 6
Done
<243|0>yoramb@inferno-05:~/os$
*/
```

הפונ' init
נקראת פעם
יחידה

5.9.7 מידע ספציפי לתהליך (Thread Specific Data, TSD)

הזכרנו בתחילת הפרק כי לעתים נרצה להחזיק נתונים שיהיו מצד אחד פרטיים לכל פתיל ופתיל, ומצד שני לא יוחזקו ע"ג המחסנית של כל פתיל, ע"מ שלא רק הפונ' הראשית של הפתיל תכיר אותם—נרצה שהם יהיו 'גלובליים' לפתיל.

נכיר עתה את הכלי אותו מעמידה לרשותנו pthread למשימה.

ראשית, נגדיר משתנה גלובלי:

```
pthread_key_t key ;
```

המשתנה key ידאג לכך שמשתנה אחר כלשהו (בדוגמה שלנו tsd) יהיה פרטי לכל תהליך ותהליך (ולכל תהליך יהיה עותק נפרד שלו).
באמצעות כמה משתנים כאלה נוכל לקבל כמה משתנים ספציפיים.

עתה עלינו לאתחל את המשתנה key.
נעשה זאת באופן הבא:

```
rc = pthread_key_create(&key, NULL) ;
if (rc) {
   fputs("pthread_key_create failed", stderr) ;
   exit(EXIT_FAILURE) ;
}
```

(בארגומנט השני ל: pthread_key_create() ניתן להעביר מהרס למשתנה הספציפי, אנו לא נעשה זאת.)

את האיתחול הנ"ל נרצה לבצע פעם יחידה עבור כל הפתילים בתכנית. מי יעזור לנו לעשות זאת?

המשתנה הגלובלי:

```
pthread_once_t threads_init = PTHREAD_ONCE_INIT;
```

ובתחילת הפתיל נזמן:

```
pthread_once(&threads_init, init_key) ;
```

עבור הפונ' ()init_key שהינה:

```
//-------------------------------------------------------
void init_key() {
   int rc ;

   puts("In init()") ;
   rc = pthread_key_create(&key, NULL) ;
   if (rc) {
      fputs("pthread_key_create failed", stderr) ;
      exit(EXIT_FAILURE) ;
   }
}
```

אחרי האיתחול יוכל כל פתיל להשתמש ב- key כדי לאחסן נתונים שיהיו פרטיים לו. על הנתונים יצביע משתנה מטיפוס void *. לכן בהמשך הפונ' של כל פתיל (אחרי הזימון של pthread_once() נוכל לכתוב:

```
int *tsd = (int *) malloc(2 * sizeof(int)) ;
```

מצביע זה יורה על הנתונים הספציפיים לפתיל.

ונכניס את הערכים הדרושים (הפרטיים) למערך:

```
tsd[0] = id;
tsd[1] = rand() % 100 ;
```

כדי לשמור מידע זה כפרטי וגלובלי לַפתיל נבצע:

```
rc = pthread_setspecific(key, tsd) ;
if (rc) {
   fputs("pthread_setspecific failed", stderr) ;
   exit(EXIT_FAILURE) ;
}
```

עתה באמצעות המשתנה key נוכל בפונ' אחרת, המורצת ע"י פתיל זה, לשלוף את הנתונים.

לכן נניח שזימנו פונ' אחרת, ובה נבצע:

```
void another_func() {
  int  *tsd = (int *) pthread_getspecific(key) ;

  printf("thread #%d, tid = %u, randon num = %d\n",
        tsd[0], (unsigned) pthread_self(), tsd[1]) ;
}
```

הפקודה: pthread_getspecific(key) ; מאפשרת לכל פתיל לשלוף את הנתונים הפרטיים שלו, כפי שנשמרו (קודם לכן, על-ידו) בעזרת הגרסה שלו של המשתנה key.

נראה את התכנית השלמה:

```
// file: pthread_tsd.c
// create a key for a thread specific data,
// and use it to store tsd (the thread 'id',
// and a random value)
// each thread calls another_func() to demonstrate that
// the tsd  is unique to each thread, yet 'global' to it.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>            // for sleep()


void* my_func(void *) ;        // הפונ' הראשית של כל פתיל
void another_func() ;          // פונ' נוספת שכל פתיל מזמן
void init_key() ;              //          הפונ' לאתחול המפתח.
                               // נרצה להריצה פעם יחידה בלבד


                               //          המשתנה הבא יעזור לנו בכך:
pthread_once_t threads_init = PTHREAD_ONCE_INIT;

pthread_key_t key ;            // זהו המפתח בעל הערך הייחודי
                               // לכל פתיל. בעזרתו ישמר ויישלף
                               // המידע הפרטי אך גלובלי לפתיל
```



```
//-----------------------------------------------------
int main() {
   pthread_t thread_data[5] ;
   int i ;
   int status ;

   srand(17) ;

   for (i = 0; i < 5 ; i++) {
         status = pthread_create(&(thread_data[i]),
                                 NULL,
                                 my_func, &i) ;
         if (status != 0) {
            fputs("pthread_create failed in main", stderr) ;
            exit(EXIT_FAILURE) ;
         }
         sleep(2) ;

   }
   for (i= 0; i< 5; i++)
      pthread_join(thread_data[i], NULL) ;

   return(EXIT_SUCCESS) ;
}
```

הסיכוי לאיזו בעיה קטן (אך לא נעלם) תודות לכך?

הפונ' הראשית של הפתילים

```
//-----------------------------------------------------
void* my_func(void * args)  {
   int id = *( (int *) args) ;
   int rc ;
   int *tsd = (int *) malloc(2 * sizeof(int)) ;

   // the key is created only once
   pthread_once(&threads_init, init_key) ;

   if (tsd == NULL) {
      perror("malloc failed") ;
      exit(EXIT_FAILURE) ;
   }

   tsd[0] = id;
   tsd[1] = rand() % 100 ;
   printf("thread #%d, tid = %u, randon num = %d\n",
         id, (unsigned) pthread_self(), tsd[1]) ;
   // in the tsd area store the id of the thread
   rc = pthread_setspecific(key, tsd) ;
   if (rc) {
      fputs("pthread_setspecific failed", stderr) ;
      exit(EXIT_FAILURE) ;
   }

   // call another function that will use this data:
   // the data is 'global' to each thread
   another_func() ;

   pthread_exit(NULL) ;
}
```

מזהה הפתיל == מספרו

המידע הפרטי שברצוננו לשמור עבור כל פתיל

פעולת השמירה של המידע

```
//-------------------------------------------------------
void another_func() {
   // in the other func we use the tsd
   // the key enables the func to retrieve the wanted data
   // as opposed to another data, stored using other keys
   int  *tsd = (int *) pthread_getspecific(key) ;
   printf("thread #%d, tid = %u, randon num = %d\n",
          tsd[0], (unsigned) pthread_self(), tsd[1]) ;
}
//-------------------------------------------------------
void init_key() {
   int rc ;

   puts("In init()") ;
   rc = pthread_key_create(&key, NULL) ;
   if (rc) {
      fputs("pthread_key_create failed", stderr) ;
      exit(EXIT_FAILURE) ;
   }
}
//-------------------------------------------------------
/*
a run:
======
<212|1>yoramb@inferno-05:~/os$ !a
a.out
In init()
thread #0, tid = 3086728080, randon num = 65
thread #0, tid = 3086728080, randon num = 65
thread #1, tid = 3076238224, randon num = 57
thread #1, tid = 3076238224, randon num = 57
thread #2, tid = 3065748368, randon num = 39
thread #2, tid = 3065748368, randon num = 39
thread #3, tid = 3055258512, randon num = 47
thread #3, tid = 3055258512, randon num = 47
thread #4, tid = 3044768656, randon num = 79
thread #4, tid = 3044768656, randon num = 79
```

פעולת השליפה של המידע (תוך שימוש במפתח בעל הערך הייחודי לכל פתיל)

כל פתיל תורם שתי שורות בפלט: האחת ע"י הפונ' הראשית (המאחסנת), השנייה ע"י הפונ' הנוספת (השולפת)


```
<213|0>yoramb@inferno-05:~/os$
*/
```


5.9.8 שליחת סיגנל לפתיל

כפי שניתן לשלוח סיגנל לתהליך, ניתן לעשות זאת גם לפתיל. הפקודה היא:

pthread_kill(<thread id>, <signal num>)

נראה גרסה של התכנית שראינו בעבר בה שני פתילים (בדוגמה הקודמת היו אלה שני תהליכים) שולחים סיגנל זה לזה.

(אתם מוזמנים להשוות בין שתי הגרסות גם כדי לעמוד על הדמיון והשוני בין תהליכים לפתילים: פתילים חלוקים מרחב כתובות, תהליכים לא, ועל-כן פתילים יכולים לפנות למשתנה גלובלי שמכיל את המזהה של שניהם; תהליכים מוכרים ע"י מ.ה. ויכולים ע"י ק.מ. לבקש את המזהה שלהם ושל אביהם.)


```
// file:  catch_usr_signal_thread.c
// a dad and a son send signals to each other,
// and thus coordinate their progress.
// An example of an output at the end of the program
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>     // for sleep(), pause()
#include <signal.h>     // for pthread_kill()
#include <sys/types.h>
//---------------------------------
#define        DAD     0
#define        CHILD   1
//---------------------------------
void catch_sigusr1(int sig_num) ;
void* do_son(void *) ;
void do_dad() ;
//---------------------------------
pthread_t thread_data[2] ;
//---------------------------------
int main() {
   int status ;

   signal(SIGUSR1, catch_sigusr1) ;

   thread_data[DAD] = pthread_self() ;
   status = pthread_create(&thread_data[CHILD],
                           NULL,
                           do_son,
                           NULL) ;
   if (status != 0) {
      perror("pthread_create failed in main") ;
      exit(EXIT_FAILURE) ;
   }
   do_dad() ;
   return EXIT_SUCCESS ;
}
```

מערך גלובלי, אליו יפנו האב והבן. מחזיק את מספרי הפתילים של שניהם (לשם שליחת סיגנל זה לזה).

קביעת הטפלית

הזנת ערכים למערך

```
//--------------------------------
void* do_son(void *args) {
   int i ;

   sleep(5) ;          // Very important! Gives dad enough
                       // time to send first printf,
                       // and issue pause().
                       // Otherwise son's puts() intrferes
                       // wth dd's prntf, and son's signal
                       // is lost, and we get a deadlock.
   for (i=0; i < 5; i++) {
      puts("Child's turn") ;
      pthread_kill(thread_data[DAD], SIGUSR1) ;
      pause() ;                // until u get a signal
   }
   return NULL ;
}
//--------------------------------
void do_dad() {
   int i ;

   printf("tid of dad = %u, tid of son = %u\n",
          (unsigned) thread_data[DAD],
          (unsigned) thread_data[CHILD]) ;
   for (i=0; i < 5; i++) {
      pause() ;                // until u get a signal
      puts("Dad's turn") ;
      pthread_kill(thread_data[CHILD], SIGUSR1) ;
   }
}
//--------------------------------
void catch_sigusr1(int sig_num) {
   signal(SIGUSR1, catch_sigusr1) ;
   printf(" thread %u caught signal SIGUSR1\n",
          (unsigned) pthread_self()) ;
}
//--------------------------------
```

שליחת הסיגנל כמעט זהה לפקודה בתהליכים (רק מזהה האב בֵּילד נשלף מהמערך ולא מק.מ.)


```
/* An output:
   ==========
<245|1>yoramb@inferno-05:~/os$ !g
gcc -Wall -lpthread catch_usr_signal_thread.c
<246|0>yoramb@inferno-05:~/os$ !a
a.out
tid of dad = 3085928128, tid of son = 3085925264
Child's turn
 thread 3085928128 caught signal SIGUSR1
Dad's turn
 thread 3085925264 caught signal SIGUSR1
Child's turn
 thread 3085928128 caught signal SIGUSR1
Dad's turn
 thread 3085925264 caught signal SIGUSR1
Child's turn
 thread 3085928128 caught signal SIGUSR1
Dad's turn
 thread 3085925264 caught signal SIGUSR1
Child's turn
 thread 3085928128 caught signal SIGUSR1
Dad's turn
 thread 3085925264 caught signal SIGUSR1
Child's turn
 thread 3085928128 caught signal SIGUSR1
Dad's turn
<247|0>yoramb@inferno-05:~/os$
 */
```


5.9.9 עוד על סיגנלים ופתילים

א. לכל פתיל בתהליך מסכת סיגנלים נפרדת באמצעותה הוא יכול לחסום סיגנלים. הפונ' הדרושה: pthread_sigmask() היא המקבילה לק.מ.: sigprocmask() שראינו בהקשר של תהליכים.

ב. ישנם סיגנלים הנשלחים לתהליך בכללותו (SIGINT ע"י c^), ויש כאלה הנשלחים לפתיל בודד בשל עוון שהוא ביצע (חילק באפס, חרג בזיכרון). סיגנל הנשלח לתהליך בכללותו ישוגר לפתיל מקרי כלשהו שלא חסם את הסיגנל (כך נוכל לייצר פתיל יעודי לטיפול בסיגנלים: הוא ורק הוא לא יחסום אותם). נראה דוגמה:

```
// file:  pthread_catch_signal.c
// compile: gcc -Wall -lpthread pthread_catch_signal.c
// The program catches two signals:
// SIGINT is caught each time
// by a different (random) thread,
// SIGFPE is caught by the thread that divided
// by zero (and exits when it gets it,
// while the other thread continues).
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>

void* my_func(void *) ;
void catch_int(int sig) ;
void catch_fpe(int sig) ;
```



```
//-------------------------------------------------------
int main() {
   int status ;
   pthread_t tid[2] ;

   signal(SIGINT, catch_int) ;
   signal(SIGFPE, catch_fpe) ;

   srand((unsigned) time(NULL)) ;

   status =pthread_create(&tid[0], NULL, my_func, NULL) ;
   if (status != 0) {
      perror("pthread_create failed in main") ;
      exit(EXIT_FAILURE) ;
   }
   status =pthread_create(&tid[1], NULL, my_func, NULL) ;
   if (status != 0) {
      perror("pthread_create failed in main") ;
      exit(EXIT_FAILURE) ;
   }

   fprintf(stderr, "tid0 = %u, tid1 = %u\n",
         (unsigned) tid[0],  (unsigned) tid[1]) ;

   pthread_exit(NULL) ;

   return(EXIT_SUCCESS) ;
}
```

נתפוס את שני הסיגנלים: INT, FPE

הפתיל הראשי רק יוצר שני פתילים, מדפיס את מספריהם, ויוצא.

```
//-------------------------------------------------------
void* my_func(void * args)  {
   int i;

   for (i=0; i< 5; i++) {
      int n, n1 = 10,  n2;

      sleep(rand() % 5) ;

      n2 = rand() % 4 ;
      fprintf(stderr, "tid = %u, div = %d\n",
                     (unsigned) pthread_self(), n2) ;
      n = n1/ n2 ;
   }
   pthread_exit(NULL) ;
}
//-------------------------------------------------------
void catch_int(int sig) {
   fprintf(stderr, "tid = %u catch SIGINT\n",
         (unsigned) pthread_self()) ;
}

//-------------------------------------------------------
void catch_fpe(int sig) {

   fprintf(stderr, "tid = %u catch SIGFPE about to exit\n",
         (unsigned) pthread_self()) ;

   pthread_exit(NULL) ;
}

//-------------------------------------------------------
```

כל פתיל, חמש פעמים מגריל מחלק מקרי ומנסה לחלק (אם המחלק שהוגרל הוא אפס ישלח לו סיגנל).

עת נתפס SIGINT הפתיל שתפסו מודיע לנו מי הוא.

עת נתפס SIGFPE הפתיל שתפסו מודיע לנו מי הוא, ובזאת הוא מסיים.


```
/* A run:
 ========
<291|0>yoramb@inferno-05:~/os$ !a
a.out
tid0 = 3086818192, tid1 = 3076328336
tid = 3076328336, div = 3
tid = 3086818192, div = 3
tid = 3076328336, div = 3
tid = 3086818192, div = 0
tid = 3086818192 catch SIGFPE about to exit
tid = 3076328336, div = 1
tid = 3076328336, div = 3
tid = 3076328336, div = 2
<292|0>yoramb@inferno-05:~/os$

<303|1>yoramb@inferno-05:~/os$ a.out
tid0 = 3086359440, tid1 = 3075869584
tid = 3086359440 catch SIGINT
tid = 3086359440, div = 3
tid = 3086359440 catch SIGINT
tid = 3086359440, div = 3
tid = 3086359440 catch SIGINT
tid = 3086359440, div = 2
tid = 3075869584, div = 1
tid = 3086359440 catch SIGINT
tid = 3086359440, div = 2
tid = 3086359440 catch SIGINT
tid = 3086359440, div = 2
tid = 3075869584 catch SIGINT
tid = 3075869584, div = 1
tid = 3075869584 catch SIGINT
tid = 3075869584, div = 1
tid = 3075869584, div = 1
tid = 3075869584 catch SIGINT
tid = 3075869584, div = 3
<304|0>yoramb@inferno-05:~/os$
*/
```

בהרצה א' לא הוקש ^c.
אנו רואים שהפתיל 192, הגריל אפס, על כן חטף סיגנל, ונזרק לטפלית ושם מסיים.

הפתיל 336 ממשיך לרוץ.

בהרצה ב' התכנית מופצצת ב- ^c.
אשר נתפס בחלק מהמקרים ע"י הפתיל 440,

ובחלק מהמקרים ע"י 584

(באופן אקראי)

(בהרצה זו לא הוגרל אפס)

5.9.10 פתילי Posix, fork() ו- exec()

בתחילת הפרק אמרנו ש:

א. עת פתיל מבצע fork() רק הוא (ולא כל התהליך) משוכפל.

ב. עת פתיל מבצע exec() ביצועם של כלל הפתילים בתהליך מופסק, והתהליך עובר להריץ תכנית חדשה (כל מרחב הכתובות מומר, ולכן גם לפתילים האחרים 'נשמט השטיח מתחת לרגליים').

נראה דוגמה: ראשית ל- exec(), ואחר ל- fork().

דוגמה ל- exec() הקוטעת את ביצוע כלל פתילי התהליך:

```
// file:   pthread_exec.c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>


void* my_func(void *) ;
//-------------------------------------------------------

int main() {
   pthread_t thread_data ;
   int status, i ;

   srand((unsigned)time(NULL)) ;
   status = pthread_create(&thread_data, NULL,
                              my_func,  NULL) ;
   if (status != 0) {
      perror("pthread_create failed in main") ;
      exit(EXIT_FAILURE) ;
   }

   for (i = 0 ; i< 10; i++) {
      puts("main thread") ;
      sleep(1) ;
   }

   pthread_exit(NULL) ;
   return(EXIT_SUCCESS) ;
}
```

הפתיל הראשי אמור להציג 10 פעמים את הפלט שלו.

```
//-------------------------------------------------------
void* my_func(void * args)  {
     sleep(2) ;
     puts("secondary thread about to exec()") ;
     execlp("csh", "csh", "-c",  "date", NULL) ;

     return(NULL) ;
}
//-------------------------------------------------------
```

המשני מבצע exec ויריץ פקודת date

```
/* A run:
 ========
<228|1>yoramb@inferno-05:~/os$ !g
gcc -Wall -lpthread pthread_exec.c
<229|0>yoramb@inferno-05:~/os$ !a
a.out
main thread
main thread
secondary thread about to exec()
Wed Nov 26 12:41:46 IST 2008
<230|0>yoramb@inferno-05:~/os$
*/
```

כל מרחב הכתובות מוחלף, בפרט ביצועו של הפתיל הראשי נקטע בטרם עת.

דוגמה ל- ()fork: רק הפתיל (המשני) שביצע את ה- ()fork משוכפל

```
// file:   pthread_fork.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

void* my_func(void *) ;
//-------------------------------------------------------

int main() {
   pthread_t thread_data ;
   int status, i ;

   srand((unsigned)time(NULL)) ;
   status = pthread_create(&thread_data, NULL,
                                  my_func,  NULL) ;
   if (status != 0) {
      perror("pthread_create failed in main") ;
      exit(EXIT_FAILURE) ;
   }

   for (i = 0 ; i< 3; i++) {
      printf("main thread pid= %d\n", getpid()) ;
      sleep(2) ;
   }

   pthread_exit(NULL) ;
   return(EXIT_SUCCESS) ;
}
```

פלט הפתיל הראשי מופיע מתהליך *יחיד*, 3 פעמים בלבד


```
//-------------------------------------------------------
void* my_func(void * args)  {
   int i, status ;

   status = fork() ;
   if (status < 0) {
      perror("cannot fork()") ;
      exit(EXIT_FAILURE) ;
   }
   for (i=0; i< 3; i++) {
      printf("'secondary' thread pid = %d\n", getpid()) ;
      sleep(2) ;
   }

   pthread_exit(NULL) ;
}
//-------------------------------------------------------
```

הפתיל המשני משתכפל

פלט הפתיל המשני מופיע *משני* תהליכים ועל-כן, 6 פעמים.

```
/* A run:
 ========
<237|1>yoramb@inferno-05:~/os$ !g
gcc -Wall -lpthread pthread_fork.c
<238|0>yoramb@inferno-05:~/os$ !a
a.out
main thread pid= 29701
'secondary' thread pid = 29703
'secondary' thread pid = 29701
main thread pid= 29701
'secondary' thread pid = 29701
'secondary' thread pid = 29703
main thread pid= 29701
'secondary' thread pid = 29701
'secondary' thread pid = 29703
<239|0>yoramb@inferno-05:~/os$
*/
```

בפלט, תהליך 29701 מיוצג על שני פתיליו, 29703 מריץ רק פתיל 'משני'.

5.10 פתילים בחלונות 2000

חלונות 2000 מממשת את ממשק Win32, שהינו ממשק האפליקציה המרכזי של מ.ה. של MS (החל בחלונות 95).

אפליקציית חלונות כוללת תהליך שעשוי להיות מורכב מכמה פתילים. כל פתיל משתמש ממופה לפתיל גרעין. מעבר לכך קיימים בחלונות סיבים (fiber) המאפשרים מיפוי של רבים לרבים.

כלל הפתילים הנכללים בתהליך חולקים את מרחב הכתובות של התהליך.

בהתאמה לכך שהפתילים מוכרים לגרעין, מ.ה. מתזמנת פתילים (ולא תהליכים).

5.11 פקודת ה- clone() של לינוקס (בלבד)

כזכור, לכל תהליך מחזיקה מ.ה. PCB (ביוניקס מכונה: process descriptor). ב- PCB (או ליתר דיוק בשטח זיכרון עליו מורה מצביע מה- PCB) מוחזקים, בין היתר, נתוני הקבצים הפתוחים, טפליות הסיגנלים, ותיאור כתובות הזיכרון המוקצות לתהליך.

עת תהליך מבצע fork() נוצר תהליך חדש הכולל *עותק* של הנ"ל (על כן אם אחד משני התהליכים סגר קובץ פתוח, הדבר אינו משפיע על משנהו).

פקודת ה- clone() ייחודית ללינוקס (אינה קיימת במערכות יוניקסיות אחרות). הפקודה מאפשרת לצור תהליך ילד שיחלוק עם הורהו משאבים במידה בה ננחה את הפקודה.

אם נקבע שהילד יחלוק עם הורהו את מרחב הכתובות, טבלת מתארי הקבצים, וטבלת טפליות הסיגנלים, אזי למעשה קיבלנו פתיל; וזה, למעשה, יעוד הפקודה: לייצר פתילי גרעין (המתוזמנים ע"י מ.ה. ככל 'תהליך' עם ובלי מרכאות). האופן בו הדבר נעשה הוא שה- PCB של הילד מצביע על אותם נתונים כמו של הורהו (כתובות זיכרון, מתארי קבצים, טפליות סיגנלים), וכך הם חולקים את המשאבים. כמו כן בעת יצירת הילד אין צורך להשקיע זמן וזיכרון בשכפול המשאבים.

הילד והורהו עשויים לחלוק חמישה משאבים שונים. עת הם חלוקים אפס משאבים clone() שקולה ל- fork(), עת הם חולקים חמישה clone() מייצרת פתיל.

כלומר בלינוקס ההבחנה בין תהליך לפתיל גרעין היא הדרגתית. המונח בו משתמשים הוא על-כן <u>משימה</u> (task).

הנ"ל אינו בסתירה למימוש ספריית פתילים ברמת המשתמש, כדוגמת pthread.

clone() שונה מ- fork() (ודומה ל- pthread_create()) בכך שהתהליך הנוצר מתחיל את ריצתו בפונ' ייעודית, שמצביע לה ולארגומנטים לה מועבר ל- clone().

תהליך הילד מסתיים עת הפונ' בה הוא נולד חוזרת, וערך ההחזרה שלו הוא ערך ההחזרה של אותה פונ' (בהכרח int).

לפני יצירת הילד, על תהליך האב להקצות לו, במרחב כתובות שלו, שטח מחסנית נפרד, וזאת כדי שמחסניותיהם לא תתנגשנה. מכיוון שבלינוקס מחסניות גדלות כלפי מטה, מועבר ל- clone() מצביע לקצה שטח הזיכרון המיועד.

אתאר חלק מהדגלים באמצעותם נקבעת מידת השיתוף בין ההורה ובתו:

CLONE_FILES מורה שהשניים יחלקו את טבלת מתארי הקבצים (אחרת הבת יורשת *עותק* של הטבלה של אביה).

CLONE_SIGHAND מורה שהשניים יחלקו טבלת טפלי סיגנלים. על כן ביצוע sigaction() באחד ישפיע גם על משנהו. (אך כל תהליך מקבל סיגנלים משלו, שכן הוא תהליך נפרד.)

CLONE_VM מורה שהשניים יחלקו מרחב כתובות, בפרט משתנים גלובליים.

פרוטוטיפ הפונ:

```
#include <sched.h>

int clone( int (*fn)(void *),
           void *child_stack,
           int flags,
           void *args) ;
```

- מצביע לפונ' הראשית של הפתיל
- מצביע *לסוף* מחסנית הפתיל
- דגלים המתארים אילו משאבים יחלוק הילד עם הורהו
- הארגומנטים שיועברו לפונ' הראשית של הפתיל

הפונ' מחזירה לאם את ה- pid של בנה, או 1- בכישלון.

נראה תכנית לדוגמה.

```
// file: clone.c
// an axample of a clone()
// adapted (and fixed) from:
// http://www-128.ibm.com/developerworks/linux/library/l-pow-oprofile/
// the thread uses the global var shared_data.
// Alternatively, the thread my get a
// (single pointer paramm), and handles it.
// ==> indicates places where passing a param is done
// (and is closed now as a documentation)

#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/wait.h>          // for wait()
#include <time.h>              // for srand()
#include <unistd.h>            // for sleep()


struct shared_data_struct {
        unsigned int data1;
        unsigned int data2;
};

struct shared_data_struct shared_data;

static int inc_second(/*==> struct shared_data_struct * */);
//-------------------------------------------------------------
int main(){

        int i, pid;
        void *child_stack;

        srand( (unsigned) time(NULL)) ;
```

מבנה גלובלי אליו יפנו שני התהליכים (החולקים זיכרון). כ"א יפנה לחבר אחר.

מצביע למחסנית הילד


```
        /* allocate memory for other process to execute in */
        if((child_stack = (void *) malloc(4096)) == NULL) {
                perror("Cannot allocate stack for child");
                exit(EXIT_FAILURE);
        }

        // clone process and run in the same memory space
        if ((pid = clone(inc_second,
                        child_stack+4096,     // pointer to END of
                                              // child stack
                        CLONE_VM | SIGCHLD,
                                        // share VM and
                                        // get SIGCHLD when she
                                        // finishes
                        NULL                  // no args to child's func
                        /*==> &shared_data*/)) < 0) {
                perror("clone called failed.");
                exit(EXIT_FAILURE);
        }

        /* increment first member of shared struct */
        for (i = 0; i < 5; i++) {
            sleep(rand() % 6) ;
            shared_data.data1++;
        }

        // only due to SIGCHLD above,
        // the parent can wait to child
        // (otherwise it is not notified, and wait returns -1)
        pid = wait(NULL) ;
        printf("parent pid =  %d, child pid = %d\n",
                (int) getpid(),  pid) ;
        printf(" %d %d\n",
                 shared_data.data1,  shared_data.data2) ;

        return EXIT_SUCCESS ;
}
```



```
//-------------------------------------------------------------

int inc_second(/*==> struct shared_data_struct *sd*/)
{
        int i;

        /* increment second member of shared struct */
        for (i = 0; i < 10; i++) {
            sleep(rand() % 6) ;
            shared_data.data2++ ;
            // ==> sd->data2++;
        }
       fputs("child ends\n", stderr) ;
        return  EXIT_SUCCESS ;
}

/* a run:
   ====
<204|1>yoramb@inferno-05:~/os$ gcc -Wall clone.c
<205|0>yoramb@inferno-05:~/os$ a.out
child ends
parent pid =  12081, child pid = 12082
 5 10
<206|0>yoramb@inferno-05:~/os$
```

קיבלנו שני תהליכים, לכ"א משלו pid