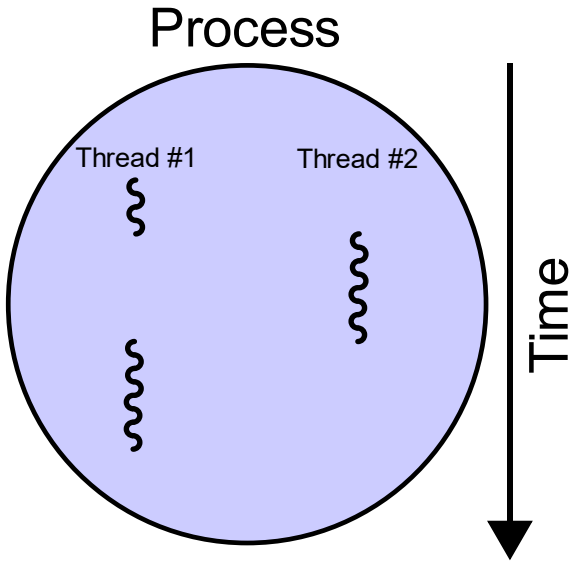
תרגול 6

חוטים (threads) בלינוקס
תמיכת גרעין לינוקס בחוטים
תכנות מקבילי באמצעות חוטים
מנגנוני סינכרון: מנעולים

TL;DR



- מעבדים מודרניים הם **מרובי ליבות**.
  איך אפשר לנצל אותם כדי לשפר ביצועים?
  - לדוגמה, מיון מערך גדול ע"י: חלוקה לשניים, מיון כל חצי מערך בנפרד, ולבסוף מיזוג.
- **תהליכים** לא משתפים זיכרון, ולכן הם פחות מתאימים לתכנות מקבילי.
- **חוטים** (threads), לעומת זאת, פועלים **במרחב זיכרון משותף**.
  - חוט הוא יחידת ביצוע עצמאית בתוך תהליך ("Lightweight process").
  - כל תהליך יכול להכיל מספר חוטים שירוצו במקביל.
- אבל שיתוף זיכרון בין חוטים יוצר גם בעיות, שאחת הנפוצות בהן היא: **היעדר אטומיות בגישה למשתנים משותפים**.
- נלמד איך להתגבר על הבעיה באמצעות מנעולים (mutex/spinlock).

חוטים (THREADS) בלינוקס

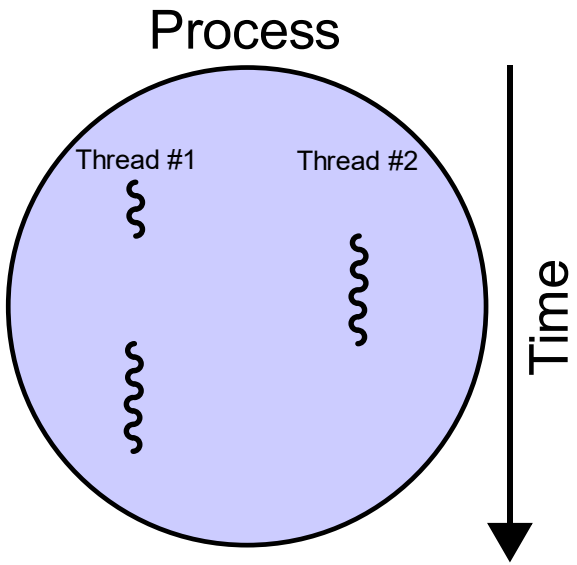
אנטי–דוגמה: מיון מקבילי של מערך

```
void quick_sort(int a[], int length);

int* parallel_sort(int a[], int length) {
  int* b = (int*) malloc(length * sizeof(int));
  pid_t p = fork();
  if (p == 0) { // son
    quick_sort(a, N / 2);
  } else { // father
    quick_sort(a + N / 2, N / 2);
    wait(NULL);
    // now merge the two subarrays into b
  }
  return b;
}
```

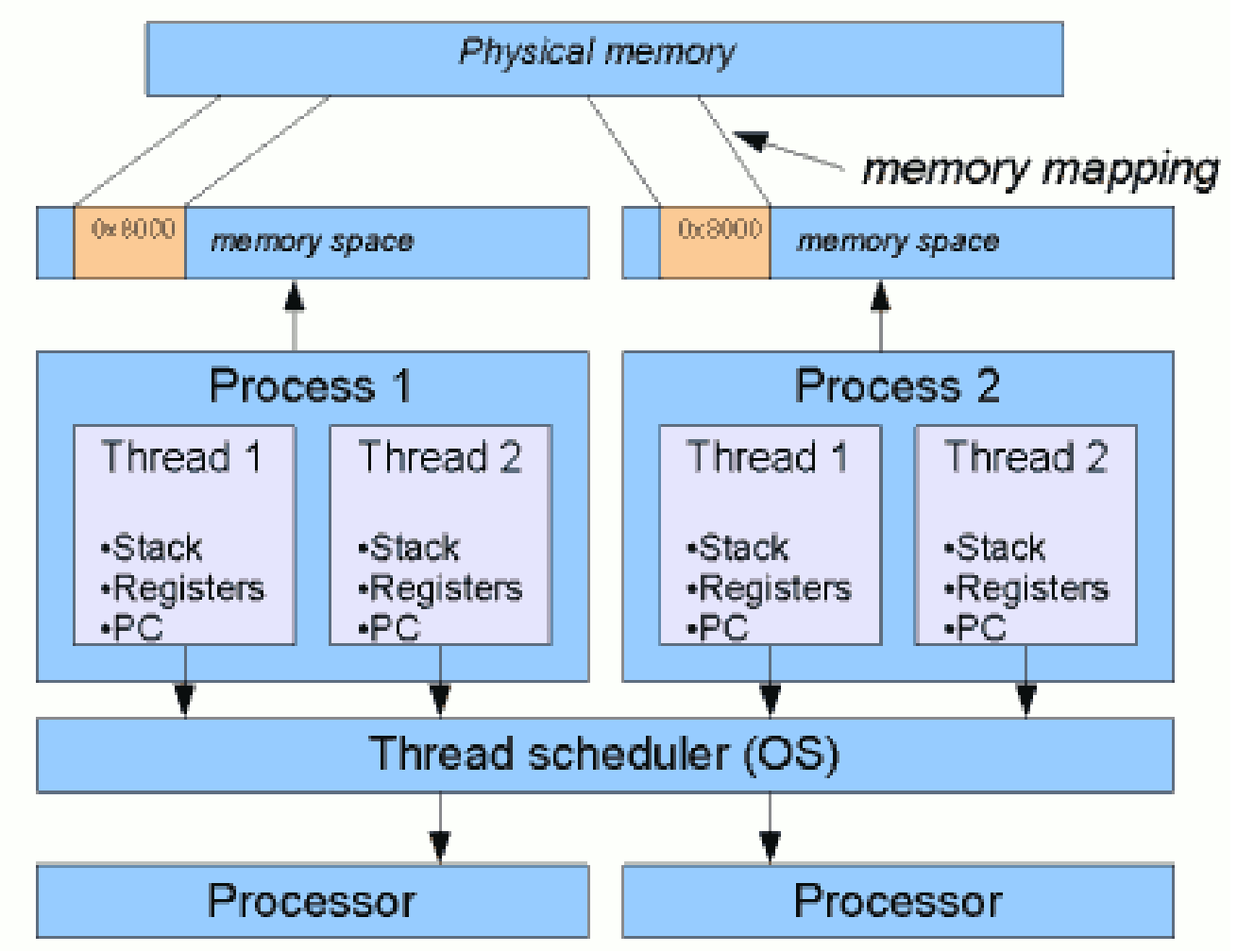
למה המימוש הזה לא יעבוד?

חוטים (threads)



- חוט הוא יחידת ביצוע בתוך תהליך.
  - באנגלית נקרא גם: Lightweight Process.
  - בעברית נקרא גם: תהליכון.
- תהליך בלינוקס יכול לכלול מספר חוטים המשתפים ביניהם את כל משאבי התהליך: מרחב הזיכרון, גישה לקבצים והתקני חומרה, ועוד.
- למרות שחוט הוא רכיב של תהליך, כל חוט הוא יחידת ביצוע עצמאית שניתן להריץ על מעבד ללא קשר לשאר החוטים.
  - כל חוט מהווה הקשר ביצוע נפרד – לכל חוט מחסנית ורגיסטרים משלו.
  - ה-scheduler מזמן לריצה חוטים ולא תהליכים.

חוטים מול תהליכים



חוטים יכולים לשפר ביצועים

- במערכת מרובת מעבדים: כל חוט יפתור חלק מהמשימה של אותו תהליך, והחוטים ירוצו במקביל על מעבדים שונים.
  - אבל זה דורש לפרק את הבעיה לתתי–בעיות בלתי תלויות – משימה לא טריוויאלית למתכנת...
- גם במחשב עם מעבד יחיד ניתן לשפר ביצועים באמצעות שימוש בריבוי חוטים: חוט אחד יכול לוותר על המעבד (למשל כדי להמתין לנתונים מהרשת) וחוט אחר ימשיך בביצוע חלק אחר של התוכנית.
  - יתרון נוסף לחוטים: יצירת חוט זולה מעט מיצירת תהליך חדש, מפני שאינה כרוכה בהעתקת משאבים כמו טבלת הדפים וטבלת הקבצים הפתוחים.

מתי כדאי/לא כדאי להשתמש בחוטים?

מתי כדאי?

- תהליכים חישוביים "כבדים" (למשל חיזוי מזג האוויר) יכולים לנצל יותר ליבות כדי לשפר את הביצועים.
- חוטים יכולים להריץ משימות בלתי תלויות, למשל הדפסת מסמך במקביל לעריכתו.
- אפליקציות שרת יכולות ליצור חוט חדש לכל בקשה כדי לטפל במספר בקשות בו-זמנית.

מתי לא כדאי?

- תוכניות קטנות ופשוטות עלולות לסבול מתקורה מיותרת:
  עלות יצירת חוטים חדשים.
- תהליכים חישוביים במערכת מעבד יחיד עלולים לסבול מתקורה מיותרת:
  עלות החלפות הקשר.

תקשורת בין חוטים

- חוטים צריכים בדרך-כלל לתקשר ולהחליף ביניהם מידע. אמצעי התקשורת הוא פשוט ביותר: קריאה וכתיבה למשתנים משותפים.
- המשתנים המשותפים צריכים להיות **גלובאליים** או להיות **מועברים כפרמטרים לחוטים**.
- משתנים משותפים הם גם חסרון: יש לתאם את הגישות אליהם על-מנת למנוע את שיבוש הנתונים.
- ישנם מצבים שבהם חוטים לא נדרשים לשתף ביניהם מידע.
  - לדוגמה: חיפוש מילה ספציפית במספר גדול של קבצים. כל חוט יקרא קובץ וידפיס למסך את המופעים של המילה בקובץ שהוא בדק.
  - בעיות שלא דורשות תקשורת בין החוטים נקראות embarrassingly parallel.

ספריית pthreads

- בשנת 1995 הוצג **תקן pthreads** (POSIX Threads), המגדיר אוסף טיפוסים ופונקציות המאפשרים עבודה עם חוטים במערכות תואמות Unix.
  - הטיפוסים והפונקציות מוגדרים בקובץ /usr/include/pthread.h , וממומשים בספריית pthreads (ספריית משתמש כמו libc).

- כדי להשתמש בספריית pthreads יש:
  - להוסיף קובץ header בתחילת קוד C:

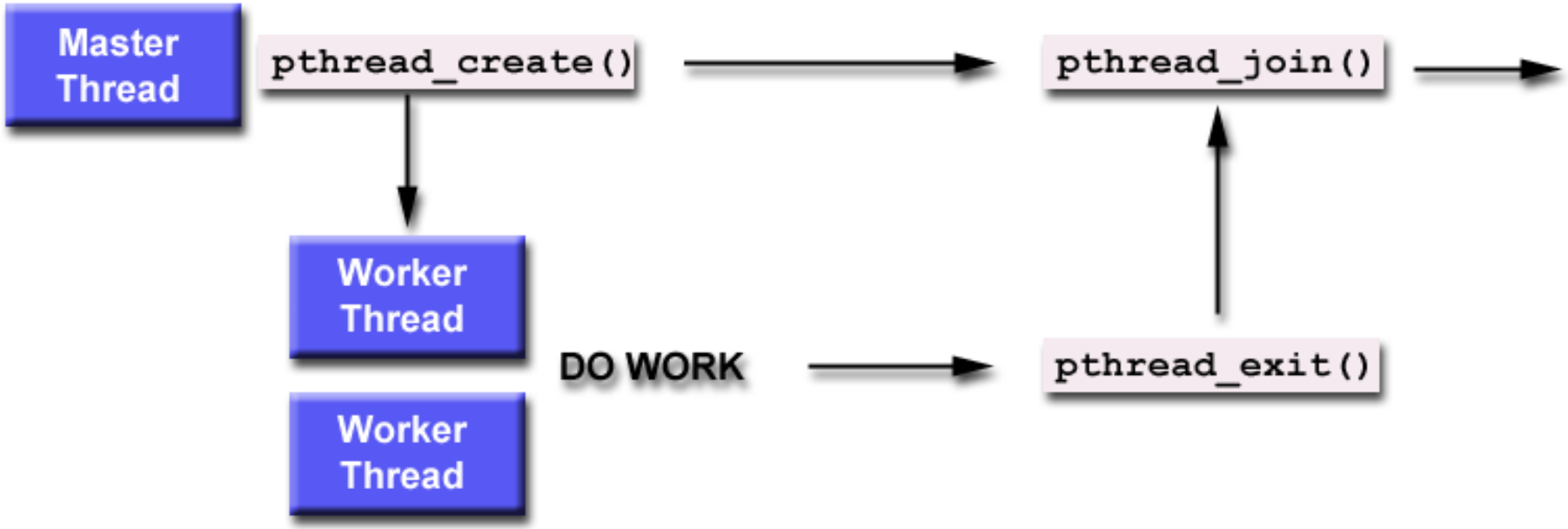
```
#include <pthread.h>
```

  - להוסיף את הדגל pthread- במהלך ההידור:

```
gcc myprog.c -pthread -o myprog
```

  - הדגל מגדיר מספר macros ומקשר את התוכנית עם הספרייה הנחוצה.

סכמת עבודה



- שימו לב: שילוב בין תהליכים לחוטים הוא אפשרי אך אינו מומלץ.

דוגמא ליצירת חוטים

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

void* print_m(void* ptr) {
    char* m = (char*)ptr; // cast the thread argument
    printf("%s PID = %d, pthread ID = %ld\n",
           m, getpid(), pthread_self());
    return NULL;
}

int main() {
    pthread_t t1;
    const char* m = "I'm Thread 1!";
    pthread_create(&t1, NULL, print_m, (void*)m);
    // we assume pthread_create() didn't fail
    pthread_join(t1, NULL);
    return 0;
}
```

מה ידפיס הקוד הבא?

יצירת חוט חדש

```
int pthread_create(pthread_t *thread,
      pthread_attr_t *attr,
      void* (*start_routine)(void*),
      void *arg);
```

- <u>פעולה:</u> יוצרת חוט חדש המתבצע במקביל לחוט הקורא בתוך אותו תהליך. החוט החדש מתחיל לבצע את הפונקציה המופיעה בפרמטר start_routine ומת בסיום ביצוע הפונקציה.
- <u>ערך מוחזר:</u>
  - 0 במקרה של הצלחה. כמו כן, מזהה החוט החדש נכתב למשתנה המוצבע ע"י thread.
  - ערך שגיאה שונה מ-0 במקרה של כישלון.

יצירת חוט חדש

- <u>פרמטרים:</u>
  - thread – מצביע למקום בו יאוחסן מזהה החוט החדש במקרה של סיום הפונקציה בהצלחה.
  - attr – מאפיינים המתארים את תכונות החוט החדש, כגון האם החוט הוא חוט גרעין או חוט משתמש, האם ניתן לבצע לו join, כלומר להמתין לסיומו, וכו'. בד"כ נספק ערך NULL המציין חוט מערכת שניתן להמתין לסיומו.
  - `void* (*start_routine)(void*)`
  - מצביע לפונקציה שתהווה את קוד החוט. הערך המוחזר מפונקציה זו במקרה של סיומה הטבעי הינו **ערך הסיום של החוט**.
  - arg – פרמטר שיסופק לפונקציה עם הפעלתה.

קבלת מזהה החוט

```
pthread_t pthread_self();
```

- פעולה: מחזירה לחוט הקורא את המזהה של עצמו. מזהה זה הוא פנימי לספרייה pthreads **ואינו קשור** ל-PID של החוט.
- מתוך ה-man page:
  - "Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results."

סיום חוט

```
void pthread_exit(void *retval);
```

- פעולה: מסיימת את פעולת החוט הקורא. ערך הסיום יוחזר לחוט שימתין לסיום חוט זה.
- פרמטרים:
  - retval – ערך סיום (בדומה לזה של exit()).

הריגת חוט

```
int pthread_cancel(pthread_t thread);
```

- פעולה: מסיימת את ביצוע החוט thread בעזרת סיגנל ייעודי.
  - ערך סיום הביצוע של החוט שנהרג יהיה PTHREAD_CANCELED.
- פרמטרים:
  - thread – מזהה החוט המיועד לסיום.
- ערך מוחזר:
  - 0 במקרה של הצלחה.
  - ערך שגיאה שונה מ-0 במקרה של כישלון.

המתנה לסיום חוט

```
int pthread_join(pthread_t thread,
void **thread_return);
```

- <u>פעולה:</u> גורמת לחוט הקורא להמתין לסיום החוט המזוהה ע"י thread.
  - ניתן להמתין על סיום אותו חוט פעם אחת לכל היותר – ביצוע pthread_join() על אותו חוט יותר מפעם אחת ייכשל.
  - כל חוט יכול להמתין לסיום כל חוט אחר באותו תהליך.
  - ההמתנה על סיום החוט משחררת את מידע הניהול של החוט ברמת הספריה pthreads וברמת הגרעין.

המתנה לסיום חוט

- <u>פרמטרים:</u>
  - thread – מזהה החוט שממתינים לסיומו.
    - לא ניתן להמתין ל"סיום חוט כלשהו" בדומה ל-wait().
  - thread_return – מצביע למקום בו יאוחסן **ערך הסיום** של החוט עבורו ממתינים.
    - ניתן לציין NULL כדי להתעלם מערך הסיום.

- <u>ערך מוחזר:</u> 0 במקרה של הצלחה, וערך שונה מ-0 במקרה כישלון. כמו כן, במקרה של הצלחה ערך הסיום נכתב למשתנה המוצבע ע"י thread_return (אם אינו NULL).

סיום חוטים ותהליכים

- חוט יכול להסתיים במספר דרכים שונות:

סיבת הסיום	האם החוט מסתיים?	האם התהליך מסתיים?
קריאה ל- pthread_exit() בתוך קוד החוט	כן	לא בהכרח (רק אם החוט שהסתיים היה האחרון)
קריאה ל- pthread_cancel() מחוט אחר		
חזרה מהפונקציה start_routine (הפונקציה המבצעת של החוט)		
קריאה לקריאת מערכת exit() ע"י חוט כלשהו בקבוצה של החוט המדובר		
פעולה לא חוקית באחד החוטים (למשל, חלוקה באפס)		
חזרה מהפונקציה main() של החוט הראשי (שקול לקריאה ל-exit())		

תזכורת:

```
int __libc_start_main(…)
{
        ……
        exit(main(…));
}
```

תמיכת גרעין לינוקס בחוטים

חוטים בגרעין לינוקס

- בגרעין לינוקס, חוטים ממומשים למעשה כתהליכים רגילים המשתפים ביניהם משאבים כגון זיכרון, גישה לקבצים וחומרה.

- כל תהליך נוצר עם חוט יחיד – **החוט הראשי** (**primary thread**) – באמצעות קריאת המערכת ()fork.
- חוטים נוספים נוצרים באמצעות **קריאת המערכת** **()clone**.
  - קריאת מערכת זו היא הבסיס לתמיכה בחוטים.

- בניגוד לתהליכים, אין קשרי משפחה בין החוטים.
  - אין חוט אב וחוט בן.
  - כל חוט יכול להמתין לסיום של חוט אחר כלשהו של אותו תהליך.
  - כל חוט יכול להרוג חוט אחר כלשהו של אותו תהליך.

קבוצת חוטים (thread group)

- לכל חוט, בהיותו תהליך רגיל, יש PCB משלו ו–**PID משלו**.

- עם זאת, המתכנת מצפה שלכל החוטים השייכים לאותו תהליך ניתן יהיה להתייחס דרך PID יחיד – של התהליך המכיל אותם.
  - פעולת getpid() תחזיר את אותו PID בכל החוטים של אותו תהליך.
  - קריאות מערכת כמו kill() הפועלות על ה–PID של התהליך צריכות להשפיע על כל החוטים בתהליך.

- לכן, לינוקס מאחדת את כל החוטים של תהליך מסוים ל**קבוצת חוטים** (**thread group**) כדי שאפשר יהיה להתייחס אליהם יחד.

קבוצת חוטים (thread group)

- השדה **tgid** במתאר התהליך מכיל את ה-**PID המשותף** לכל החוטים באותה קבוצה.
  - למעשה, **זהו ערך ה-PID של החוט הראשון** (הראשי) של התהליך.
  - חוטים חדשים יקבלו ערך PID חדש וערך TGID זהה לחוט הראשון.
  - קריאת המערכת getpid() מחזירה למעשה את current→tgid.
- השדה thread_group במתאר התהליך (task_struct) הוא ראש הרשימה המקושרת של כל החוטים באותה קבוצה.
- פעולות על ה- PID המשותף מתורגמות לפעולה על קבוצת החוטים המתאימה ל-PID.
  - למשל: kill(pid, SIGKILL) הורגת את כל החוטים עבורם tgid==pid.

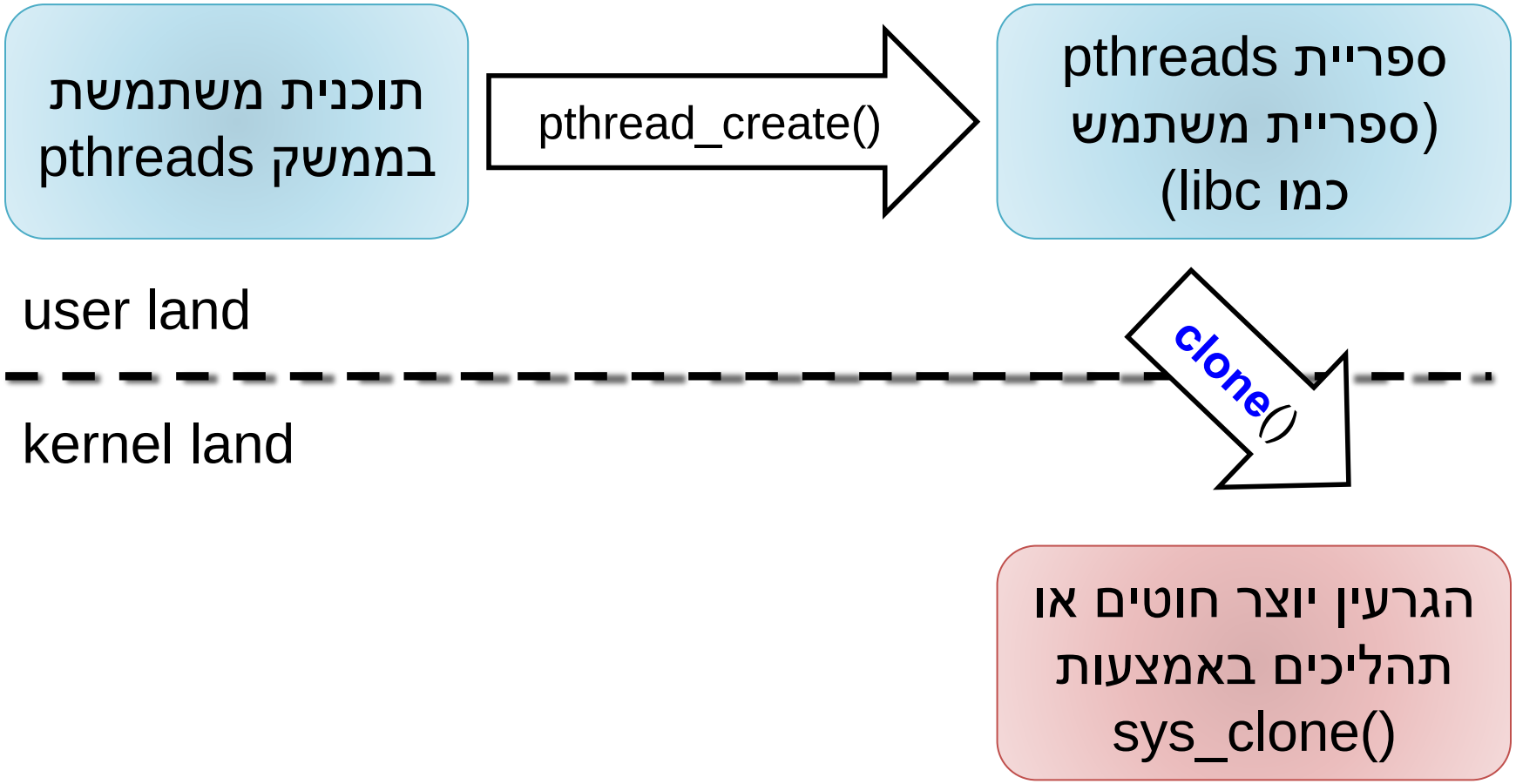
סיכום: TID מול PID

- המשתמש ומערכת ההפעלה מסתכלים על חוטים באופן שונה:

- **המשתמש:**
  - תהליך מרובה חוטים מורכב מחוט ראשי וחוטים משניים המרוכזים תחת PID יחיד (שהוא בעצם גם ה–TGID).
  - ניתן לקבל TGID זה באמצעות getpid().
  - לכל חוט מזהה נוסף הנקרא TID או Thread ID (זהו בעצם ה–PID האמיתי של החוט).
  - ניתן לקבל TID זה באמצעות **gettid()** .

- **מערכת ההפעלה:**
  - מסתכלת על כל חוט וחוט כתהליך **נפרד**, בעל PID משל עצמו. קיימת "קשירת גורלות" של כל התהליכים המרוכזים תחת אותו TGID.

תרשים: חוטים במבט מערכתי



קריאת המערכת clone()

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

- <u>פעולה:</u> יוצרת תהליך בן המשתף עם תהליך האב משאבים ונתונים לפי בחירה.
- <u>פרמטרים:</u>
  - fn – מצביע לפונקציה שתהווה את הקוד הראשי של התהליך החדש.
  - arg – הפרמטר המועבר לפונקציה fn() בתחילת ביצוע התהליך החדש.
    - כשביצוע הפונקציה fn(arg) מסתיים, נגמר התהליך החדש.
  - child_stack – מצביע לראש המחסנית של התהליך החדש
    - תזכורת: המחסנית גדלה לכיוון הכתובות הנמוכות.
    - לכן child_stack צריך להצביע לסוף בלוק הזיכרון המוקצה לטובת המחסנית.

איזו מחסנית?
משתמש / גרעין?

קריאת המערכת clone()

- flags – מסכת דגלים הקובעת את צורת השיתוף בין התהליך הקורא והתהליך החדש. להלן מספר דגלים אופייניים:

CLONE_VM	שיתוף מרחב הזיכרון
CLONE_FILES	שיתוף טבלת הקבצים הפתוחים
CLONE_FS	שיתוף טבלת נתוני עבודה עם קבצים, המכילה נתונים כגון ספרית העבודה הנוכחית ועוד
CLONE_PARENT	לתהליך החדש יהיה אותו אב כמו התהליך הקורא (אחרת החדש יהיה הבן של הקורא)
CLONE_THREAD	התהליך החדש הוא חוט באותה קבוצת חוטים כמו התהליך הקורא (אותו tgid). גורר גם CLONE_PARENT

- ניתן לשלב מספר דגלים יחד באמצעות OR לוגי ביניהם, לדוגמה: CLONE_FS | CLONE_VM .

- <u>ערך מוחזר:</u> במקרה של הצלחה מוחזר ה-PID של התהליך החדש, אחרת 1-.

מימוש קריאת המערכת clone()

- בתוך הגרעין, sys_clone() משתמשת ישירות בפונקציה do_fork() עליה למדנו בתרגולים קודמים, ומעבירה לה את הדגלים על-מנת לקבוע לכל משאב אם לשתף אותו או ליצור אותו כחדש.

הפסקה

תכנות מקבילי באמצעות חוטים

וגם: מה קורה בגישה לא מתואמת למשתנים משותפים?

תכנית לדוגמה

מה התכנית הזו עושה?

מה היא מנסה למקבל?

```
#include <pthread.h>
#include <stdio.h>
#define N 1000
int i = 0;
int a[N];

void* f(void* arg) {
  a[i] = i;
  i++;
  return NULL;
}

int main() {
  pthread_t threads[N];
  for (unsigned int i=0; i<N; i++)
    pthread_create(&threads[i], NULL, f, NULL);
  for (unsigned int i=0; i<N; i++)
    pthread_join(threads[i], NULL);
  printf("a[999] = %d\n", a[999]);
  return 0;
}
```

תכנית לדוגמה

הפונקציה f() תרוץ 1000 פעמים **במקביל**, ובכל פעם תמלא איבר נוסף של המערך.

מה יהיה ערכו של a[999] בסיום התכנית?

```
#include <pthread.h>
#include <stdio.h>
#define N 1000
int i = 0;
int a[N];

void* f(void* arg) {
  a[i] = i;
  i++;
  return NULL;
}

int main() {
  pthread_t threads[N];
  for (unsigned int i=0; i<N; i++)
    pthread_create(&threads[i], NULL, f, NULL);
  for (unsigned int i=0; i<N; i++)
    pthread_join(threads[i], NULL);
  printf("a[999] = %d\n", a[999]);
  return 0;
}
```

פלט התכנית לדוגמה

```
>> gcc -pthread -O2 main.c
>> ./a.out
a[999] = 999
>> ./a.out
a[999] = 999
>> ./a.out
a[999] = 0
```

?

מה קורה כאן?

- החלפת הקשר בין החוטים יכולה לקרות בכל נקודת זמן, בפרט באמצע הפונקציה של חוט מסויים.

- פלט התכנית תלוי בתזמון של החוטים ובסדר בו הם מתבצעים – מצב שנקרא **race condition** (תנאי מרוץ).

- מכיוון שאנחנו לא שולטים בתזמון של תהליכים (או חוטים), תכנית המכילה race condition נחשבת תקולה (buggy).
  - ליתר דיוק, תכנית כזאת היא בעלת התנהגות לא מוגדרת.
  - קשה לדבג תכניות כאלה כי קשה לשחזר את ההתנהגות הבעייתית.

תרחיש אפשרי #1 (תקין)

```
a[i] = i;
i++;
```

thread #662		thread #257
	i = 0	
a[i] = i; // a[0]←0		
i++;		
	i = 1	
	context switch	
		a[i] = i; // a[1]←1
		i++;
	i = 2	

תרחיש אפשרי #2 (בעייתי)

```
a[i] = i;
i++;
```

thread #662 | thread #257

i = 0

```
a[i] = i; // a[0]←0
```

context switch

```
a[i] = i; // a[0]←0
i++;
```

i = 1

context switch

```
i++;
```

i = 2

מי יכתוב
a[1]←1 ?

race condition בשורת C אחת

- שימו לב: **החלפת הקשר יכולה לקרות גם באמצע שורת C**, כי הקומפיילר עשוי לתרגם שורת C אחת למספר פקודות אסמבלי.
- לכן, race condition יכול לקרות גם במקומות לא צפויים.
- לדוגמה, הקומפיילר יכול להדר את השורה i++ לקוד האסמבלי הבא:

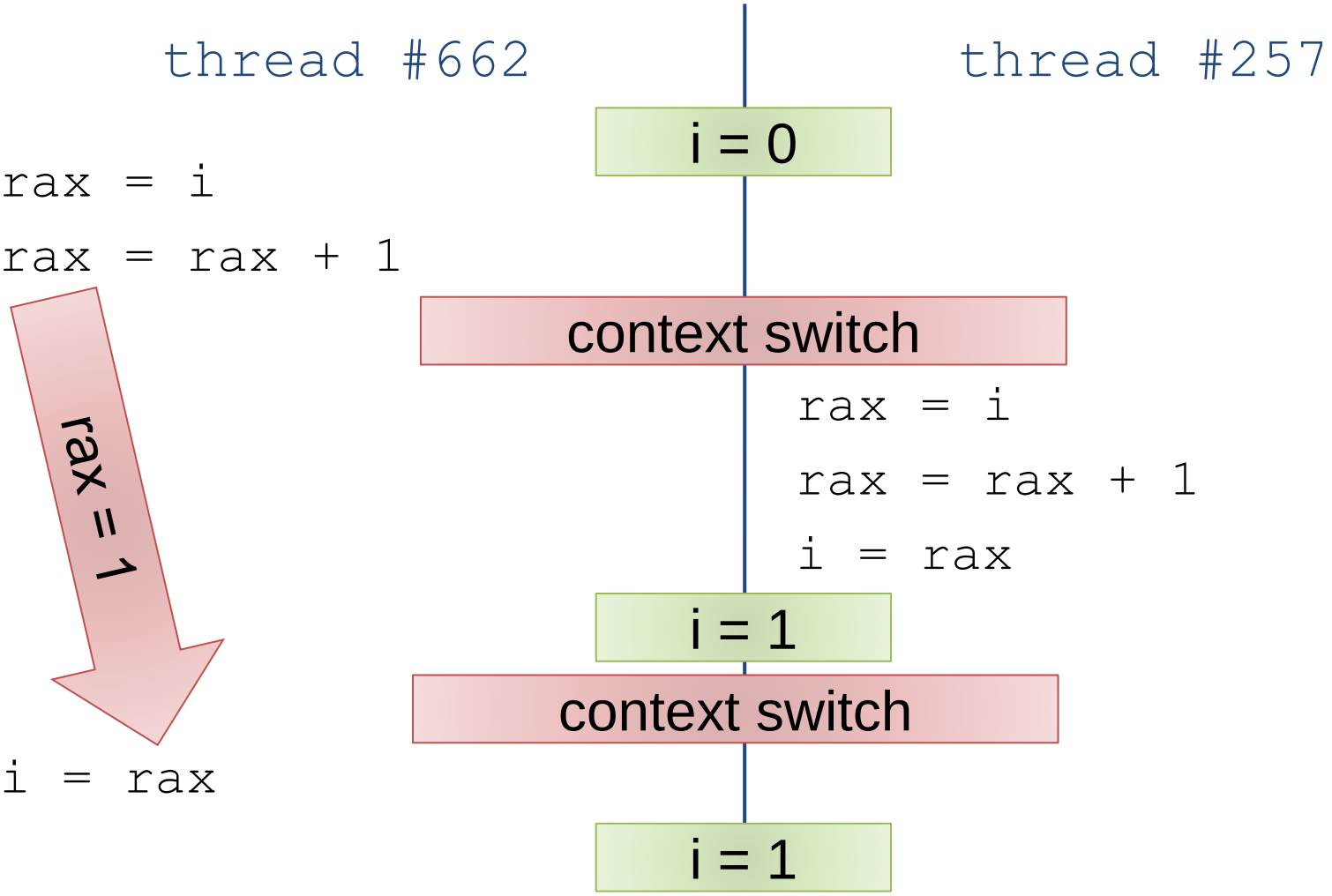
```
f:
    …
    movl (%rbx), %rax
    addl $1, %rax
    movl %rax, (%rbx)
    …
```

פסאודו-קוד:

```
rax = i
rax = rax + 1
i = rax
```

תרחיש אפשרי #3 (בעייתי)

```
i++ { rax = i
      rax = rax + 1
      i = rax
```



קטע קריטי

- קטע קוד הניגש למשאב משותף (למשל משתנה בזיכרון).
- ביצוע קטע קריטי במקביל עלול לגרום להתנהגות לא רצויה.
- תכנית תקינה צריכה להבטיח **מניעה הדדית** – **לכל היותר חוט אחד** יריץ את הקטע הקריטי בכל רגע נתון.
  - במילים אחרות: הפקודות בקטע הקריטי צריכות להתבצע בצורה **אטומית** ביחס לגישות אחרות למשאב המשותף – או שכל הפקודות בקטע הקריטי ירוצו יחד ויסתיימו, או שהן לא ירוצו כלל.

```
void* f(void* arg) {
  a[i] = i;
  i++;
  return NULL;
}
```

בדוגמה הקודמת: שתי השורות האלו הן קטע קריטי

מנגנוני סינכרון: מנעולים

מנעולים

- מנעולים הם אחד המנגנונים הבסיסיים למימוש מניעה הדדית.

- האנלוגיה: קטע קריטי ⇔ חדר עם דלת מוגנת ע"י מנעול עם מפתח בפנים.
- כדי להיכנס לחדר (הקטע הקריטי) צריך לנעול את המנעול ולשים את המפתח בכיס.
- ביציאה מהחדר יש לפתוח את המנעול ולהשאיר את המפתח בדלת (לטובת החוטים האחרים).

- **<u>שימו לב:</u>**
  - בכל רגע נתון, לכל היותר חוט אחד יכול לתפוס / להחזיק / לנעול את המנעול.
  - רק החוט המחזיק במנעול אמור לשחרר אותו (בעלות על המנעול).

תכנית לדוגמה

אתם יכולים לזהות את הקטע הקריטי בתכנית זו?

איפה אפשר להוסיף מנעול(ים) כדי לתקן את הבעיה הקיימת בתכנית זו?

```
#include <pthread.h>
#include <stdio.h>
#define N 1000
int i = 0;
int a[N];

void* f(void* arg) {
  a[i] = i;
  i++;
  return NULL;
}

int main() {
  pthread_t threads[N];
  for (unsigned int i=0; i<N; i++)
    pthread_create(&threads[i], NULL, f, NULL);
  for (unsigned int i=0; i<N; i++)
    pthread_join(threads[i], NULL);
  printf("a[999] = %d\n", a[999]);
  return 0;
}
```

מנעולים ב-pthreads

- בתקן pthreads מנעולים נקראים **mutex**.
  - קיצור של mutual exclusion (מניעה הדדית).

- נתקן את הדוגמה הקודמת בעזרת נעילה:

```
pthread_mutex_t m;

void* f(void* arg) {
  pthread_mutex_lock(&m);
  a[i] = i;
  i++;
  pthread_mutex_unlock(&m);
  return NULL;
}
```

- אם המנעול אינו נעול, החוט נועל אותו ונכנס לקטע הקריטי.
- אם המנעול כבר נעול, החוט נחסם עד אשר המנעול ישוחרר.

איך התרחיש הבעייתי נמנע?

```
mutex_lock(&m);
a[i] = i;
i++;
mutex_unlock(&m);
```

thread #662	thread #257
`mutex_lock(&m);`	
`a[i] = i;`	
context switch	
	`mutex_lock(&m);`
	➔ `blocks…`
context switch	
`i++;`	
`mutex_unlock(&m);`	

דוגמת קוד עם mutex

```
// shared var
// must be protected
long long count;

pthread_mutex_t m;

void update_count() {
 pthread_mutex_lock(&m);
 count = count * 5 + 2;
 pthread_mutex_unlock(&m);
}

long long get_count() {
 long long c;
 pthread_mutex_lock(&m);
 c = count;
 pthread_mutex_unlock(&m);
 return c;
}
```

- **שאלה:** מדוע צריך להגן על הגישה ל-count בתוך update_count()?
  - **תשובה:** כדי למנוע שיבוש ערך count בעדכונים מחוטים שונים.
- **שאלה:** מדוע צריך להגן על הגישה ל-count בתוך get_count()?
  - **תשובה:** כדי למנוע קבלת תוצאות חלקיות הנוצרות במהלך העדכון.
- **שאלה:** נניח שפעולת העידכון הייתה count++. האם עדיין צריך להגן על הפונקציה update() באמצעות מנעול?
  - **תשובה:** כן, כי לא מובטח שהקוד הנפרש באסמבלי הינו אטומי.

מנעולים עם/בלי המתנה

mutex

- כאשר חוט מנסה לתפוס מנעול נעול, מערכת ההפעלה תעביר אותו **לתור המתנה** ותבצע החלפת הקשר.
  - כאשר המנעול ישוחרר, מערכת ההפעלה תעיר את אחד החוטים המחכים למנעול.
- **יתרון:** תהליך חדש יכול לרוץ מיד, וכך לא מתבזבז זמן מעבד יקר.
  - עדיף כאשר זמן ההמתנה המשוער גבוה יחסית (כלומר, כאשר הקטע הקריטי ארוך, כפי שקורה לרוב בקוד משתמש).

spinlock

- כאשר חוט מנסה לתפוס מנעול נעול, הוא בודק את ערך המנעול ללא הפסקה ולא מוותר על המעבד.
  - טכניקה כזו נקראת גם: polling, busy waiting, busy looping.
- **יתרון:** התהליך יכול להמשיך לרוץ עוד בקוונטום הנוכחי, וכך לחסוך את התקורה על החלפת הקשר.
  - עדיף כאשר זמן ההמתנה המשוער נמוך יותר מהמחיר של החלפת הקשר (כלומר, כאשר הקטע הקריטי קצר, כפי שקורה לרוב בקוד גרעין).

אתחול מנעול mutex

```
#include <pthread.h>
int pthread_mutex_init(
      pthread_mutex_t *mutex,
      const pthread_mutex_attr_t *mutexattr);
```

- <u>פרמטרים:</u>
  - mutex – המנעול עליו מבוצעת הפעולה.
  - mutexattr – מגדיר את תכונות ה-mutex .
    - NULL – עבור סוג ברירת המחדל.
    - ניתן לקרוא על סוגים נוספים ב-man pages.
- <u>ערך מוחזר:</u> 0 בהצלחה, ערך אחר בכישלון.

פעולות על מנעולי mutex

- נעילת mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

  - הפעולה חוסמת עד שה-mutex מתפנה ואז נועלת אותו.
- ניסיון לנעילת mutex:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

  - הפעולה נכשלת אם ה-mutex כבר נעול, אחרת נועלת אותו.
- שחרור mutex נעול:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - ניסיון לשחרר מנעול שאינו נעול תביא להתנהגות לא מוגדרת.
- פינוי mutex בתום השימוש:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  - הפעולה נכשלת אם ה-mutex מאותחל אבל נעול.

שימוש תקין במנעולים

- בקוד תקין רק החוט שמחזיק במנעול הוא זה שמשחרר אותו.

- לעומת זאת, הפעולות הבאות יובילו להתנהגות לא מוגדרת:
  1. נעילה חוזרת ע"י החוט המחזיק במנעול.
  2. שחרור המנעול ע"י חוט שאינו מחזיק במנעול.
  3. שחרור מנעול שאינו נעול.

- דיבוג קוד עם מנעולים הוא מאתגר יותר בגלל שהקוד כבר לא דטרמיניסטי – בחלק מהריצות יש באג, בחלק מהריצות אין באג.
- יש כלים שעוזרים בבעיה, לדוגמה helgrind או sanitizers.

פקודות מכונה אטומיות

- פקודת מכונה אטומית (atomic operation) היא פקודה המעדכנת את מצב המערכת (מעבד+זיכרון) בצורה אטומית.
  - כלומר מעבדים אחרים אינם יכולים לראות מצבי ביניים של ביצוע הפקודה ואינם יכולים לעדכן את מצב המערכת תוך כדי ביצוע הפקודה.
- במערכת מעבד יחיד: כל פקודת מכונה היא אטומית מכיוון שהיא אינה יכולה להיקטע ע"י פסיקה.
- במערכת מרובת מעבדים: פקודת מכונה אטומית חייבת לנעול את ערוץ הגישה של כל המעבדים האחרים לזיכרון עד לסיומה.
- הנעילה מתבצעת באמצעות הוספת קידומת lock לפקודה.
  - למשל: lock;inc x היא פקודת מכונה המבצעת x++ בצורה אטומית.

פקודות מכונה אטומיות

- פקודת מכונה אטומית מגדירה למעשה קטע קריטי באורך פקודה אחת, וכך מאפשרת להימנע משימוש במנעול.

- בנוסף, פקודות מכונה אטומיות מאפשרות לממש בצורה פשוטה יחסית אמצעי סינכרון כמו מנעולים.
- לדוגמה: ארכיטקטורת x64 מציעה את פקודה BTS (bit test and set), אשר מדליקה ביט מסוים ברגיסטר או כתובת בזיכרון ומחזירה את ערכו הקודם בדגל CF ברגיסטר הדגלים.
- פונקציית הגרעין test_and_set_bit() משתמשת בפקודת המכונה BTS כדי להדליק משתנה ולהחזיר את ערכו הקודם בצורה אטומית.

1

שאלה ממבחן – מימוש מנעולים

סעיף א'

```
typedef struct lock {
  bool is_locked;
} lock_t;

void init(lock_t* l) {
  l->is_locked = 0;
}

void lock(lock_t* l) {
  while (l->is_locked);
  l->is_locked = 1;
}

void unlock(lock_t* l) {
  l->is_locked = 0;
}
```

- להלן מימוש של מנעול ללא תמיכת הגרעין.
  - הניחו כי השמת ערך למשתנה בזיכרון או קריאה שלו הינן אטומיות.
  - התעלמו מבעיות קוהרנטיות וקונסיסטנטיות.
- האם המימוש מבטיח מניעה הדדית?
- לא. ניתן דוגמה נגדית:
  - חוט A מנסה לתפוס את המנעול, בודק את הערך של is_locked ומגלה שהוא 0 ← עובר את לולאת ה-while.
  - כעת מתבצעת החלפת הקשר וחוט B בודק את הערך של is_locked ומגלה שהוא עדיין 0 ← גם הוא עובר את לולאת ה-while.
  - שני חוטים בקטע הקריטי בו זמנית!

סעיף ב'

```
typedef struct lock {
  bool is_locked;
} lock_t;

void init(lock_t* l) {
  l->is_locked = 0;
}

void lock(lock_t* l) {
  while (test_and_set_bit
    (l->is_locked));
}

void unlock(lock_t* l) {
  l->is_locked = 0;
}
```

- כעת נתון מימוש בעזרת הפקודה האטומית test_and_set_bit(), המדליקה ביט ומחזירה את ערכו הקודם.
- האם המימוש מבטיח מניעה הדדית?
- כן. ההוכחה בעזרת נפנופי ידיים...
- האם המימוש מונע הרעבה?
- לא. ניתן דוגמה נגדית:
  - שני חוטים A,B רצים לסירוגין.
  - חוט A תמיד תופס את המנעול לפני סיום הקוונטום שלו.
  - חוט B תמיד מכלה את הקוונטום שלו בהמתנה למנעול.