תרגול 7

מנגנוני סנכרון: משתני תנאי
מנגנוני סנכרון: סמפורים
דוגמה: מימוש מנעול קוראים-כותבים
סינכרון בגרעין לינוקס

TL;DR

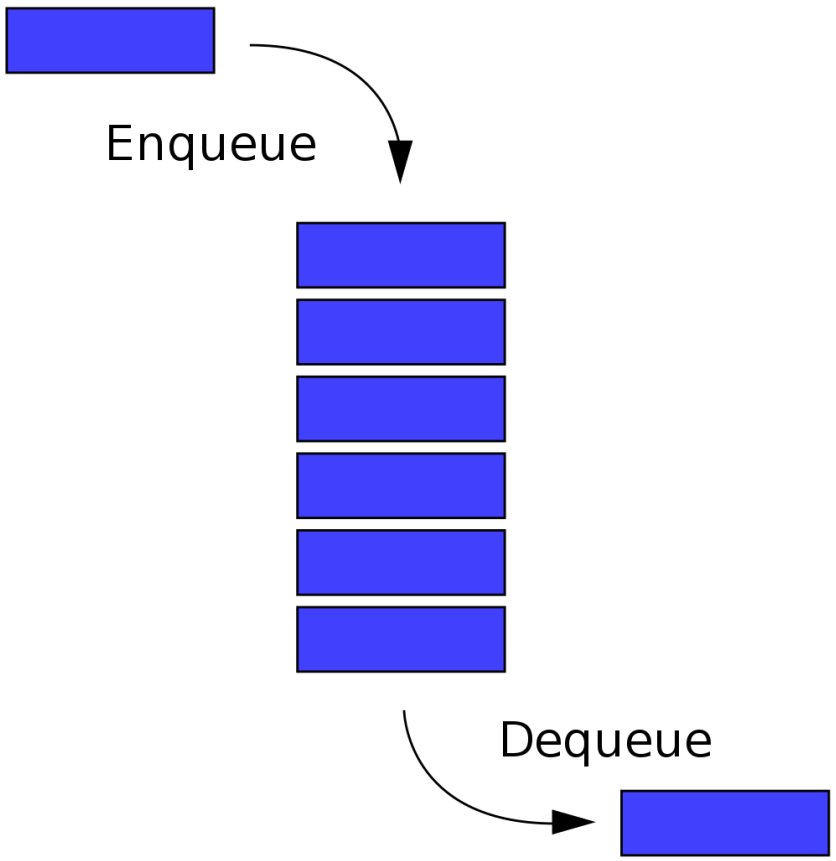
- בתרגול הקודם למדנו לכתוב קוד מקבילי באמצעות חוטים.
  - ראינו שבכל בעיה לא טריוויאלית יש צורך **בסנכרון** בין החוטים.
- היום נלמד על מנגנוני סנכרון נוספים של ממשק pthreads :

להבטחת אטומיות	להבטחת סדר
מנעולים (mutexes)	משתני תנאי (condition variables)
סמפורים (semaphores)	

- לבסוף, נלמד דוגמה נוספת של קוד מקבילי חשוב: גרעין לינוקס.
  - קוד הגרעין לא משתמש בחוטים, אבל **ניגש לזיכרון משותף מתוך מספר מסלולי בקרה שרצים במקביל**, ולכן העקרונות שלמדנו תקפים גם עבורו.

מנגנוני סנכרון: משתני תנאי

הצגת הבעיה: תור מקבילי



- מבנה נתונים עם שתי פעולות:
  - enqueue – הכנסת איבר לתור.
  - dequeue – הוצאת איבר מהתור. **אם התור ריק, הפעולה תיחסם עד שיכנס איבר חדש.**
- יש להגן על התור ע"י מנעולים.
  - כי חוטים שונים יכולים להכניס לתור או להוציא מהתור במקביל.
- בנוסף צריך להבטיח **סדר**.
  - חוט שרוצה להוציא איבר יצטרך להמתין לתנאי "התור אינו ריק".

ניסיון ראשון לפתרון

```
mutex_t m;
int queue_size = 0;

void enqueue(item x) {
  mutex_lock(&m);
  /* add x to tail */
  queue_size++;
  mutex_unlock(&m);
}

item dequeue() {
  mutex_lock(&m);
  while (queue_size == 0);
  /* remove from head */
  queue_size--;
  mutex_unlock(&m);
}
```

- מה הבעיה במימוש המוצע?
- קיפאון (deadlock).

1. חוט #1 מנסה להוציא איבר אבל התור עדיין ריק ← נתקע בלולאת while.
2. חוט #2 מנסה להכניס איבר לתור ← נתקע כי חוט #1 עדיין מחזיק את המנעול.

- אם חוט #1 היה מנסה לוותר על המנעול ולתפוס אותו לסירוגין, היינו נתקלים בבעיה אחרת, של יעילות: בדיקה חוזרת ונשנית על גודל התור מבזבזת זמן מעבד.

משתנה תנאי (condition variable)

- משתנה תנאי הוא אובייקט סנכרון המאפשר לחוט **לצאת להמתנה** בתוך **קטע קריטי**.
  - כלומר, לפנות את המעבד ולצאת לתור המתנה.

- ההמתנה תתבצע **עד לקיום תנאי** כלשהו.
  - ההמתנה מאפשרת **לאכוף סדר** בביצוע של החוטים.

- שימוש תכנותי נכון במשתני תנאי מחייב להגדיר גם:

1. **משתנה מצב** – החוט עובר להמתנה או חוזר מהמתנה בהתאם לערכו של משתנה המצב.
2. **מנעול mutex** – מבטיח לנו אטומיות והגנה על הקטע הקריטי.

סכימה כללית למשתני תנאי

```
cond_t c; // should be initialized
mutex_t m; // should be initialized
int state_var = 0;
```

- החוט הממתין לאירוע יקרא ל:

```
while (!condition_holds(state_var))
    cond_wait(&c, &m);
```

מדוע cond_wait() מקבלת גם את המנעול?

- החוט שמסמן לחוטים הממתינים להמשיך יקרא ל:

```
if (condition_holds(state_var))
    cond_signal(&c);
```

מדוע cond_wait מקבלת גם את המנעול?

יציאה להמתנה ואז שחרור המנעול?

```
item dequeue() {
  mutex_lock(&m);
  while (queue_size == 0) {
    cond_wait(&c);
    mutex_unlock(&m);
    mutex_lock(&m);
  }
  /* remove from head */
  queue_size--;
  mutex_unlock(&m);
}
```

שחרור המנעול ואז יציאה להמתנה?

```
item dequeue() {
  mutex_lock(&m);
  while (queue_size == 0) {
    mutex_unlock(&m);
    cond_wait(&c);
    mutex_lock(&m);
  }
  /* remove from head */
  queue_size--;
  mutex_unlock(&m);
}
```

שני המימושים שגויים ← מימוש תקין של משתני תנאי חייב לשחרר את המנעול ולצאת להמתנה **באופן אטומי**.

מימוש תקין

```
cond_t c; // should be initialized
mutex_t m; // should be initialized
int queue_size = 0;

void enqueue(item x) {
  mutex_lock(&m);
  /* add x to tail */
  queue_size++;
  cond_signal(&c);
  mutex_unlock(&m);
}

item dequeue() {
  mutex_lock(&m);
  while (queue_size == 0) {
     cond_wait(&c, &m);
  }
  /* remove from head */
  queue_size--;
  mutex_unlock(&m);
}
```

האם ניתן להוציא את signal מחוץ לנעילה?

אתחול ופינוי משתני תנאי

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond,
      pthread_condattr_t *cond_attr);
```

- ערך מוחזר: הפעולה תמיד מצליחה ומחזירה 0.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- ערך מוחזר: 0 בהצלחה, ערך שונה מ-0 בכישלון (למשל, אם יש עדיין חוטים הממתינים על משתנה התנאי).

- פרמטרים:
  - cond – משתנה התנאי עליו מבוצעת הפעולה.
  - cond_attr – מגדיר את תכונות משתנה התנאי.
    - תמיד נעביר ערך NULL בקורס זה.

המתנה על משתני תנאי

```
int pthread_cond_wait(pthread_cond_t *cond,
      pthread_mutex_t *mutex);
```

<u>פעולה:</u>

1. משחררת את המנעול ומעבירה את החוט להמתין על משתנה התנאי **באופן אטומי** (ראינו קודם מדוע זה הכרחי).
   - החוט הממתין **חייב** להחזיק במנעול mutex לפני הקריאה.
2. בחזרה מהמתנה על משתנה התנאי, **החוט עובר להמתין על המנעול**. החוט יחזור מהקריאה ל-pthread_cond_wait() רק לאחר שינעל מחדש את ה-mutex.

- <u>ערך מוחזר:</u> הפעולה תמיד מצליחה ומחזירה 0.

שחרור חוטים ממתינים

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- משחררת את **אחד** החוטים הממתינים (הגינות לא מובטחת).

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- משחררת את **כל** החוטים הממתינים.
  - כל החוטים מפסיקים להמתין על משתנה התנאי ועוברים להמתין על המנעול. החוטים יחזרו לפעילות בזה אחר זה (בסדר כלשהו, לאו דווקא הוגן) לאחר שינעלו מחדש את ה-mutex.

- שימו לב: אם אין אף חוט שממתין באותו רגע על משתנה התנאי cond, הפעולות חסרות השפעה (הסיגנל הולך לאיבוד ואינו נזכר הלאה).

- ערך מוחזר: הפונקציות תמיד מצליחות ומחזירות 0.

מימוש שגוי #1

הסבירו מדוע הקוד שגוי, כלומר תארו במדויק תרחיש מסוים שבו הקוד לא יעל כנדרש.

אם החוט הראשון יתבצע לפני השני, האיתות ילך לאיבוד והחוט השני ייתקע לנצח.

```
cond_t c; // should be initialized
mutex_t m; // should be initialized

void enqueue(item x) {
  mutex_lock(&m);
  /* add x to tail */
  cond_signal(&c);
  mutex_unlock(&m);
}

item dequeue() {
  mutex_lock(&m);
  cond_wait(&c, &m);
  /* remove from head */
  mutex_unlock(&m);
}
```

מימוש שגוי #2

הסבירו מדוע הקוד שגוי, כלומר תארו במדויק תרחיש מסוים שבו הקוד לא יפעל כנדרש.

```
cond_t c; // should be initialized
mutex_t m; // should be initialized
int queue_size = 0;

void enqueue(item x) {
  mutex_lock(&m);
  /* add x to tail */
  mutex_unlock(&m);
  cond_signal(&c);
  queue_size++;
}

item dequeue() {
  mutex_lock(&m);
  while (queue_size == 0) {
     cond_wait(&c, &m);
  }
  /* remove from head */
  queue_size--;
  mutex_unlock(&m);
}
```

אם תתרחש החלפת הקשר בנקודה הזו, האיתות שוב ילך לאיבוד.

מימוש שגוי #3

מה הבעיה במימוש הזה?

```
cond_t c; // should be initialized
mutex_t m; // should be initialized
bool is_signal_caught = false;

void enqueue(item x) {
  mutex_lock(&m);
  /* add x to tail */
  mutex_unlock(&m);
  while (!is_signal_caught) {
    cond_signal(&c);
  }
  is_signal_caught = false;
}

item dequeue() {
  mutex_lock(&m);
  cond_wait(&c, &m);
  /* remove from head */
  mutex_unlock(&m);
  is_signal_caught = true;
}
```

הפתרון הזה בזבזני כמו ה- busy-wait שראינו בהתחלה.

מימוש שגוי #4

```
cond_t c;
mutex_t m;
int queue_size = 0;

void enqueue(item x) {
  mutex_lock(&m);
  /* add x to tail */
  queue_size++;
  cond_signal(&c);
  mutex_unlock(&m);
}
item dequeue() {
  mutex_lock(&m);
  if (queue_size == 0)
     cond_wait(&c, &m);
  /* remove from head */
  queue_size--;
  mutex_unlock(&m);
}
```

t2

t1

- אם נשתמש בתנאי if במקום בלולאת while ייתכן מצב של הוצאת איבר מתור ריק:

1. בהתחלה התור ריק.
2. חוט t1 קורא ל-dequeue() ולכן משחרר את המנעול וממתין.
3. חוט t2 קורא ל-enqueue(), מכניס איבר לתור ומבצע cond_signal().

- חוט t1 מתעורר ועובר להמתין לשחרור המנעול.

מימוש שגוי #4

```
cond_t c;
mutex_t m;
int queue_size = 0;

void enqueue(item x) {
  mutex_lock(&m);
  /* add x to tail */
  queue_size++;
  cond_signal(&c);
  mutex_unlock(&m);
}
item dequeue() {
  mutex_lock(&m);
  if (queue_size == 0)
     cond_wait(&c, &m);
  /* remove from head */
  queue_size--;
  mutex_unlock(&m);
}
```

t2

t3

t1

4. חוט t3 קורא ל–dequeue(), ונחסם בהמתנה למנעול בתחילת הקוד.
5. חוט t2 משחרר את המנעול ומסיים את enqueue().
6. חוט t3 מקבל את המנעול, נכנס, מוציא איבר ומסיים.
   - כלומר חוט t3 משחרר את המנעול.
7. חוט t1 מקבל את המנעול, ממשיך לבצע את הקוד ומנסה להוציא איבר מתור ריק!

מימוש שגוי #4

- ממה נבעה הבעיה?
  - בסמנטיקה הנוכחית (Mesa) פעולת cond_signal() לא בהכרח גורמת לחוט הממתין להמשיך מיד, מפני שהחוט צריך לתפוס קודם את המנעול.
  - אבל ייתכן שלפני שהחוט הממתין ינעל את ה-mutex מחדש, חוט נוסף ירוץ וישנה את הנתונים כך שהמצב הרצוי כבר לא מתקיים.
- **שאלה:** האם עדיין הייתה בעיה אם המנעול היה **הוגן**? (סדר FIFO)
- כיצד ניתן לפתור את הבעיה?
  - ע"י בדיקה נוספת של תנאי האירוע לאחר החזרה מ-cond_wait והמתנה נוספת לפי הצורך. לדוגמה:

```
while (queue_size == 0)
      cond_wait(…)
```

מנגנוני סנכרון: סמפורים

סמפור (Semaphore)

- סמפור הוא אמצעי סנכרון אשר מאפשר להבטיח אטומיות (כמו מנעול) או סדר (כמו משתנה תנאי) – בהתאם לערך ההתחלתי שלו.
  - יכול לממש גם פעולות סנכרון אחרות – נראה בהמשך.

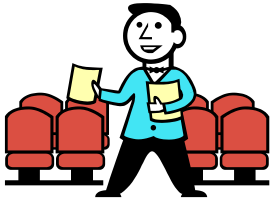
- סמפור ממומש כמונה אי-שלילי עם שתי פעולות עליו:

```
int sem_wait(sem_t *sem);
```

- <u>פעולה:</u> אם המונה גדול מ-0, מקטינה אותו ב-1. אחרת, מעבירה את החוט לתור המתנה.

```
int sem_post(sem_t *sem);
```

- <u>פעולה:</u> אם תור הממתינים לא ריק, מוציאה ומעירה את החוט הראשון בתור. אחרת, מגדילה את המונה ב-1.



סמפור (Semaphore)

- <u>פעולות נוספות על סמפורים:</u>

```
int sem_trywait(sem_t *sem);
```

- גרסה לא-חוסמת של wait. אם המונה של הסמפור אינו גדול מ-0, חוזרת מיד ונכשלת.

```
int sem_getvalue(sem_t *sem, int *sval);
```

- קוראת את ערך מונה הסמפור למקום אליו מצביע sval.
  - תמיד מצליחה ומחזירה 0.

אתחול ופינוי סמפור

- יש לכלול קובץ header נוסף מעבר ל- pthread.h :

```
#include <semaphore.h>
```

- וכמובן לקשר לספרייה pthread עם דגל הקומפילציה pthread- .

- אתחול סמפור לפני השימוש:

```
int sem_init(sem_t *sem, int pshared,
      unsigned int value);
```

- <u>פרמטרים:</u>
  - sem – הסמפור עליו מבוצעות הפעולות.
  - pshared – אם ערכו גדול מ-0, מציין שהסמפור יכול להיות משותף למספר תהליכים. תכונה זו אינה נתמכת, ולכן תמיד נציב בו 0.
  - value – ערכו ההתחלתי של מונה הסמפור.
- <u>ערך מוחזר:</u> 0 בהצלחה, (-1) בכישלון.

- פינוי סמפור בתום השימוש:

```
int sem_destroy(sem_t *sem);
```

דוגמה: סמפור בתור מנעול

```
sem_t sem;

sem_init(&sem, 0, 1);

sem_wait(&sem);
// critical section
sem_post(&sem);
```

- סמפור עם ערך התחלתי 1 נקרא סמפור בינארי.
- סמפור בינארי יכול לשמש להגנה על קטע קריטי (ע"י מניעה הדדית בין החוטים הניגשים).
- **<u>דגש:</u> סמפור בינארי שונה ממנעול mutex**, משום שכל חוט יכול לבצע post על סמפור, גם אם לא ביצע wait על הסמפור קודם לכן (אין "בעלות" על הסמפור).

דוגמה: סמפור בתור מנעול "משוכלל"

- סמפור יכול לממש הגנה על קטע קריטי מפני הרצה של יותר מ-N חוטים במקביל, אם נתאחל אותו לערך $N > 1$.
- דוגמה: שרת שיכול לשרת עד 10 משתמשים.
  - במידה ויהיו 10 משתמשים במערכת, המשתמשים הבאים שינסו להתחבר יחכו בפקודה wait().
  - רק לאחר שמשתמש כלשהו יתנתק, יורשה להיכנס המשתמש הבא.

```
sem_t sem;

sem_init(&sem, 0, 10);

void login() {
  sem_wait(&sem);
}

void logout() {
  sem_post(&sem);
}
```

דוגמה: סמפור להבטחת סדר

אם נאתחל את הסמפור ל-0, החוט השני יחכה לראשון.

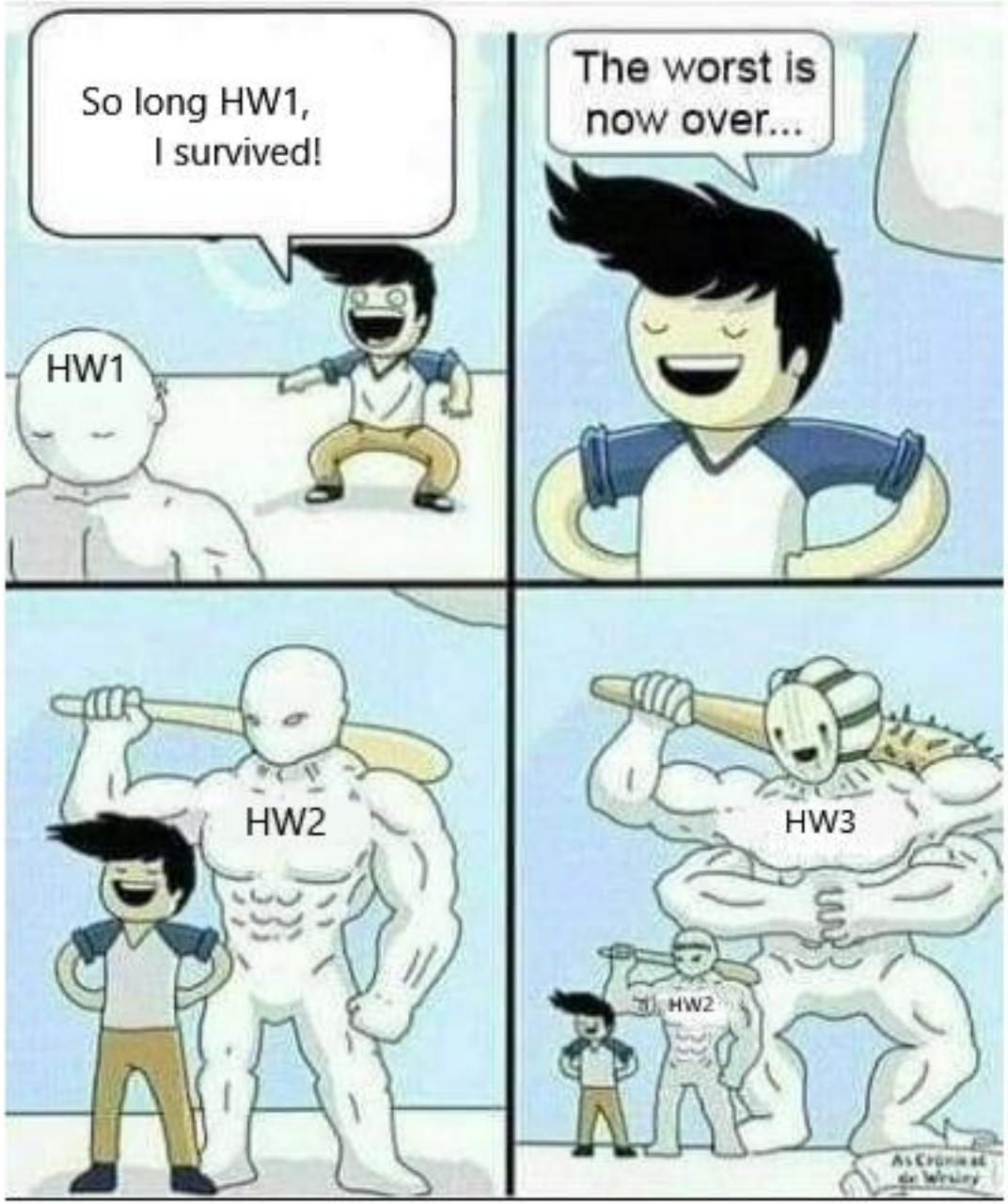
השימוש בסמפור פשוט יותר מאשר במשתנה תנאי כי post() של סמפור לא הולך לאיבוד.

```
mutex_t m;
sem_t queue_size;
sem_init(&queue_size, 0, 0);

void enqueue(item x) {
  mutex_lock(&m);
  /* add x to tail */
  mutex_unlock(&m);
  sem_post(&queue_size);
}

item dequeue() {
  sem_wait(&queue_size);
  mutex_lock(&m);
  /* remove from head */
  mutex_unlock(&m);
}
```

הפסקה



דוגמה: מימוש מנעול קוראים–כותבים

מנעול קוראים–כותבים

- מנגנון סנכרון המאפשר להגן על מבנה נתונים באופן הבא:
  - מספר חוטים יכולים לקרוא את המידע (מבלי לשנות אותו) בו–זמנית.
  - כאשר חוט רוצה לעדכן את המידע, הוא צריך גישה בלעדית למבנה הנתונים.
- לדוגמה, כדי להגן על משתנה x הנגיש ממספר חוטים:

reader thread	writer thread

read_lock();
y = 2*x;
read_unlock();
	write_lock();
	x = 5*x + 1;
	write_unlock();

- בשקפים הבאים נדגים כיצד ניתן לממש מנעול קוראים–כותבים באמצעות משתני תנאי.
  - בהרצאה ראיתם איך להשתמש בסמפור למטרה זו.

1

שאלה ממבחן

מנעול קוראים-כותבים

- ממשו מנעול קוראים-כותבים בעזרת משתני תנאי ומנעולי mutex **בלבד** (בשונה מהמימוש שראיתם בהרצאה באמצעות סמפורים).

- יש לממש את 5 הפונקציות הבאות:

1. reader_lock()
2. reader_unlock()
3. writer_lock()
4. writer_unlock()
5. readers_writers_init()

מימוש מנעול קוראים-כותבים (1)

```
int readers_inside, writers_inside;
cond_t read_allowed;
cond_t write_allowed;
mutex_t global_lock;

void readers_writers_init() {
    readers_inside = 0;
    writers_inside = 0;
    cond_init(&read_allowed, NULL);
    cond_init(&write_allowed, NULL);
    mutex_init(&global_lock, NULL);
}
```

מה ערכו המקסימלי של writers_inside?

מימוש מנעול קוראים–כותבים (2)

```
void reader_lock() {
    mutex_lock(&global_lock);
    while (writers_inside > 0)
        cond_wait(&read_allowed, &global_lock);
    readers_inside++;
    mutex_unlock(&global_lock);
}

void reader_unlock() {
    mutex_lock(&global_lock);
    readers_inside--;
    if (readers_inside == 0)
        cond_signal(&write_allowed);
    mutex_unlock(&global_lock);
}
```

למה משתמשים בלולאת while ולא תנאי if?

מימוש מנעול קוראים–כותבים (3)

```
void writer_lock() {
    mutex_lock(&global_lock);
    while (writers_inside + readers_inside > 0)
      cond_wait(&write_allowed, &global_lock);
    writers_inside++;
    mutex_unlock(&global_lock);
}

void writer_unlock() {
    mutex_lock(&global_lock);
    writers_inside--;
    if (writers_inside == 0) {
        cond_broadcast(&read_allowed);
        cond_signal(&write_allowed);
    }
    mutex_unlock(&global_lock);
}
```

האם יש צורך ב-if?

למה לא להשתמש ב-broadcast
כדי להעיר את כל הכותבים?

חסרונות של המימוש

- **<u>הרעבת כותבים וחוסר הוגנות:</u>** כל עוד המנעול אצל הקוראים, קורא חדש שמגיע יצליח להיכנס ויעקוף כותבים שהגיעו לפניו.

- **<u>חוסר סדר</u>**: לא ניתן לדעת האם הקוראים או הכותב יכנסו לקטע הקריטי.
  - תלוי מי יצליח לתפוס ראשון את המנעול `global_lock` שמשתחרר בסיום `writer_unlock()`.

- איך אפשר לפתור בעיות אלו?

2

שאלה ממבחן

מועד א', אביב 2008, שאלה 1

- נרצה לממש מנעול קוראים/כותבים **עם עדיפות לכותבים.**
  - בעדיפות לכותבים הכוונה שאם יש גם קוראים וגם כותבים המחכים להיכנס לקטע הקריטי, הכותבים מקבלים עדיפות – יכנסו **תמיד** לפני הקוראים.

- סעיף א: סמנו בעיגול את כל הדרישות מפתרון הבעיה החדשה.

X יכול להיות לכל היותר קורא אחד בקטע קריטי.
V יכול להיות לכל היותר כותב אחד בקטע קריטי.
V יכולים להיות מספר קוראים בקטע קריטי.
X יכולים להיות מספר כותבים בקטע קריטי.
V אסור לכותבים וקוראים להיות בקטע קריטי בו זמנית.
X אסור להרעיב קוראים שמנסים להיכנס לקטע קריטי.
V אסור להרעיב כותבים שמנסים להיכנס לקטע קריטי.
X יתכן מצב שקורא שהגיע אחרי כותב ייכנס לקטע הקריטי לפניו.
V יתכן מצב שכותב שהגיע אחרי קורא ייכנס לקטע הקריטי לפניו.

מועד א', אביב 2008, שאלה 1

- <u>סעיף ב:</u> להלן הצעה לפתרון בעיית קוראים/כותבים עם עדיפות לכותבים, המשתמשת בסמפורים.

- תארו 3 בעיות שונות של נכונות ו/או יעילות שיש בפתרון הנ"ל. הניחו כי הסמפור הינו הוגן.

```
sem_t sem;   // Global semaphore,
with initial value 1

int writer_lock() {
  sem_wait(sem);
}

int writer_unlock() {
  sem_post(sem);
}

int reader_lock() {
  while(sem_getvalue(sem) <= 0)
    sleep(1);
  sem_wait(sem);
}

int reader_unlock() {
  sem_post(sem);
}
```

מועד א', אביב 2008, שאלה 1

```
sem_t sem;   // Global semaphore,
with initial value 1

int writer_lock() {
  sem_wait(sem);
}

int writer_unlock() {
  sem_post(sem);
}

int reader_lock() {
  while(sem_getvalue(sem) <= 0)
    sleep(1);
  sem_wait(sem);
}

int reader_unlock() {
  sem_post(sem);
}
```

1. **בעיית נכונות:** הפתרון לא מאפשר ליותר מקורא אחד להיכנס לקטע קריטי.
2. **בעיית נכונות:** אם יש גם קוראים וגם כותבים, הכותבים לא בהכרח יקבלו עדיפות ועלולים להיות מורעבים בניגוד לדרישה.
3. **בעיית יעילות:** קוראים מבצעים busy wait.

מועד א', אביב 2008, שאלה 1

- <u>סעיף ג:</u> כתבו קוד הפותר את בעיית קוראים/כותבים עם עדיפות לכותבים, המשתמש במנעולים ומשתני תנאי.

- ניתן להגדיר משתנים גלובלים ואמצעי סנכרון כרצונכם, (מנעולים, ומשתני תנאי) אבל יש לזכור כי יעילות הפתרון מהווה חלק מהציון (כלומר מיעוט אמצעי הסנכרון עדיף וקטעים קריטיים קצרים עדיפים). ניתן להניח שעדיפות כל החוטים זהה ואמצעי הסנכרון הינם הוגנים.

- **<u>רמז</u>**: מומלץ להיעזר בפתרון הבעיה של מנעול קוראים/כותבים עם עדיפות לקוראים, כפי שהוצגה בתרגול.

מימוש מנעול קוראים-כותבים (1)

```
int readers_inside, writers_inside, writers_waiting;
cond_t read_allowed;
cond_t write_allowed;
mutex_t global_lock;

void readers_writers_init() {
    readers_inside = 0;
    writers_inside = 0;
    writers_waiting = 0;
    cond_init(&read_allowed, NULL);
    cond_init(&write_allowed, NULL);
    mutex_init(&global_lock, NULL);
}
```

מועד א', אביב 2008, שאלה 1

```
void reader_lock() {
    mutex_lock(&global_lock);
    while (writers_inside > 0 || writers_waiting > 0)
        cond_wait(&read_allowed, &global_lock);
    readers_inside++;
    mutex_unlock(&global_lock);
}

void reader_unlock() {
    mutex_lock(&global_lock);
    readers_inside--;
    if (readers_inside == 0)
        cond_signal(&write_allowed);
    mutex_unlock(&global_lock);
}
```

מועד א', אביב 2008, שאלה 1

```
void writer_lock() {
    mutex_lock(&global_lock);
    writers_waiting++;
    while (writers_inside + readers_inside > 0)
      cond_wait(&write_allowed, &global_lock);
    writers_waiting--;
    writers_inside++;
    mutex_unlock(&global_lock);
}

void writer_unlock() {
    mutex_lock(&global_lock);
    writers_inside--;
    if (writers_inside == 0) {
        cond_broadcast(&read_allowed);
        cond_signal(&write_allowed);
    }
    mutex_unlock(&global_lock);
}
```

סינכרון בגרעין לינוקס

אנלוגיית המסעדה

- דמיינו מסעדה ובה המלצר מטפל בשני סוגי לקוחות:

לקוחות רגילים	לקוחות VIP
אם המלצר פנוי ומגיע לקוח רגיל, אז המלצר עובר לטפל בו.	המלצר מטפל מיד בכל לקוח VIP שמגיע, גם אם צריך לעזוב באמצע לקוח אחר (רגיל או VIP).
לקוח רגיל יכול לשחרר את המלצר שמטפל בו כרגע לטובת לקוח אחר.	לקוח VIP לעולם לא ישחרר את המלצר שמטפל בו כרגע.
המלצר יכול לעזוב לקוח רגיל לטובת לקוחות VIP שמגיעים. לאחר הטיפול בלקוחות VIP, המלצר יכול לחזור לטפל בלקוח רגיל אחר.	המלצר לא יעזוב לקוח VIP לטובת לקוח רגיל.

הגרעין הוא מלצר

- הגרעין מטפל בשני סוגי בקשות (לקוחות):

קריאות מערכת / חריגות	פסיקות חומרה
אם המעבד מריץ קוד משתמש ומגיעה קריאת מערכת / חריגה אז הגרעין עובר לטפל בה.	הגרעין מטפל מיד בכל פסיקת חומרה שמגיעה, גם אם הוא באמצע טיפול בחריגה / קריאת מערכת / פסיקת חומרה אחרת.
קריאות מערכת יכולות לוותר על המעבד, לדוגמה read(), wait().	שגרת טיפול בפסיקת חומרה לעולם לא תוותר על המעבד.
הגרעין יכול לקטוע טיפול בקריאת מערכת / חריגה לטובת פסיקות חומרה שמגיעות. לאחר הטיפול בפסיקות החומרה, הגרעין יכול לעבור לטפל בתהליך אחר מזה שרץ קודם.	שגרת טיפול בפסיקת חומרה לעולם לא תקרא לקריאת מערכת או תיצור חריגה (למעט חריגת דף - page fault).

איך זה קשור לבעיות סנכרון?

- נסתכל על התרחיש הבעייתי הבא:
  - במסעדה יש ערכת תה אחת המורכבת ממספר חלקים (קנקן, כוסות, ...).
  - לקוח רגיל נכנס למסעדה ומבקש תה.
  - המלצר מתחיל לעבוד ומגיש ללקוח את הקנקן.
  - לפתע נכנס לקוח VIP וגם מבקש תה. המלצר כמובן ניגש לשרת אותו מיד.
  - המלצר מעביר לו את הכוסות, אבל הקנקן עדיין אצל הלקוח הקודם.
  - כל לקוח מחזיק חלק מהערכה בגלל שהמלצר לא הביא אותה **בצורה אטומית**.

- ההקבלה לגרעין המשרת פסיקות: **בעיית אטומיות בגישה למשתנים משותפים.**

מסלולי בקרה בגרעין

- **מסלול בקרה בגרעין** (kernel control path) הוא רצף פקודות שהגרעין מבצע כדי לטפל ב:

1. **קריאת מערכת** – בקשת שירות מצד תהליך משתמש.
   - למשל fork() או getpid() .
   - שימו לב: חלק מקריאות המערכת הן חוסמות, כלומר יכולות לגרום לתהליך לוותר על המעבד ולצאת להמתנה. למשל, wait() מעבירה את תהליך האב לתור המתנה עד לסיום של אחד מבניו.
2. **פסיקת תוכנה (חריגה)** – שגיאה שנוצרת ע"י קוד משתמש.
   - למשל חלוקה באפס.
3. **פסיקת חומרה** – פסיקה אסינכרונית מהתקן חומרה חיצוני.
   - למשל פסיקת שעון.

מסלולי בקרה נחתכים

- מסלולי בקרה עלולים לחתוך זה את זה או להשתלב (interleave) זה בזה. לדוגמה:
  - לפני סיום הביצוע של קריאת המערכת fork(), התקבלה פסיקת שעון, אשר גרמה לביצוע של scheduler_tick() ← פסיקת חומרה חתכה קריאת מערכת.
  - תהליך A קרא לקריאת המערכת wait(), יצא להמתנה והעביר את המעבד לתהליך B. תהליך B קורא בינתיים לקריאת המערכת getpid() ← קריאת מערכת חתכה קריאת מערכת.
  - שני מעבדים שונים מטפלים בו-זמנית בשתי חריגות שיצרו התהליכים שרצו עליהם ← חריגה חתכה חריגה.

- **יש להגן על מבני נתונים בגרעין הנגישים למסלולי בקרה נחתכים.**
  - גישה למבני נתונים של הגרעין מהווה קטע קריטי שחייב להתבצע בשלמותו ע"י מסלול הבקרה שנכנס אליו לפני שמסלול בקרה אחר יוכל להיכנס אליו.

אילו חיתוכים אפשריים?

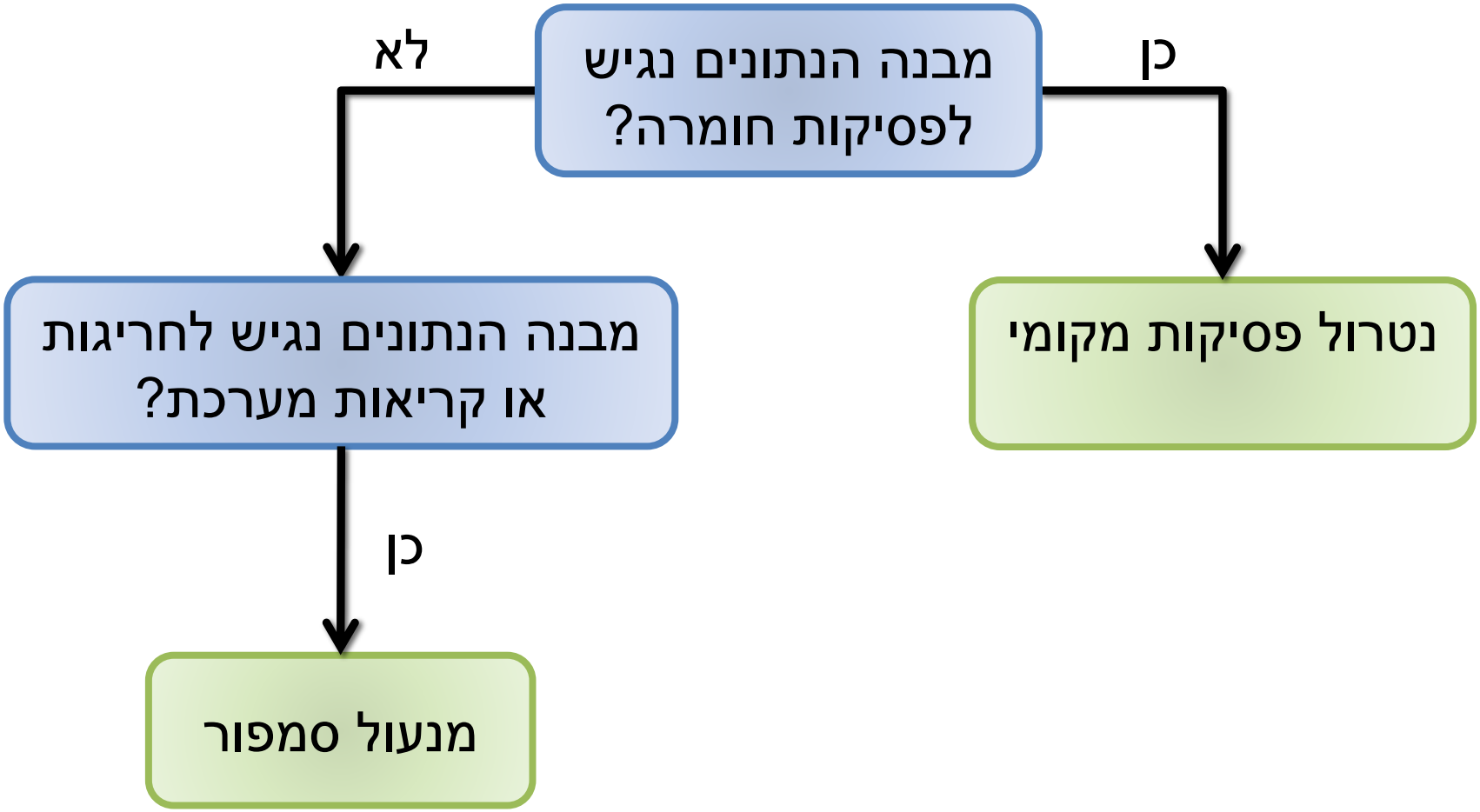
- **<u>במערכת מעבד יחיד:</u>**

סוג	יכולה להיחתך ע"י
קריאת מערכת או חריגה	כל מסלול בקרה
פסיקות חומרה	פסיקות חומרה בלבד

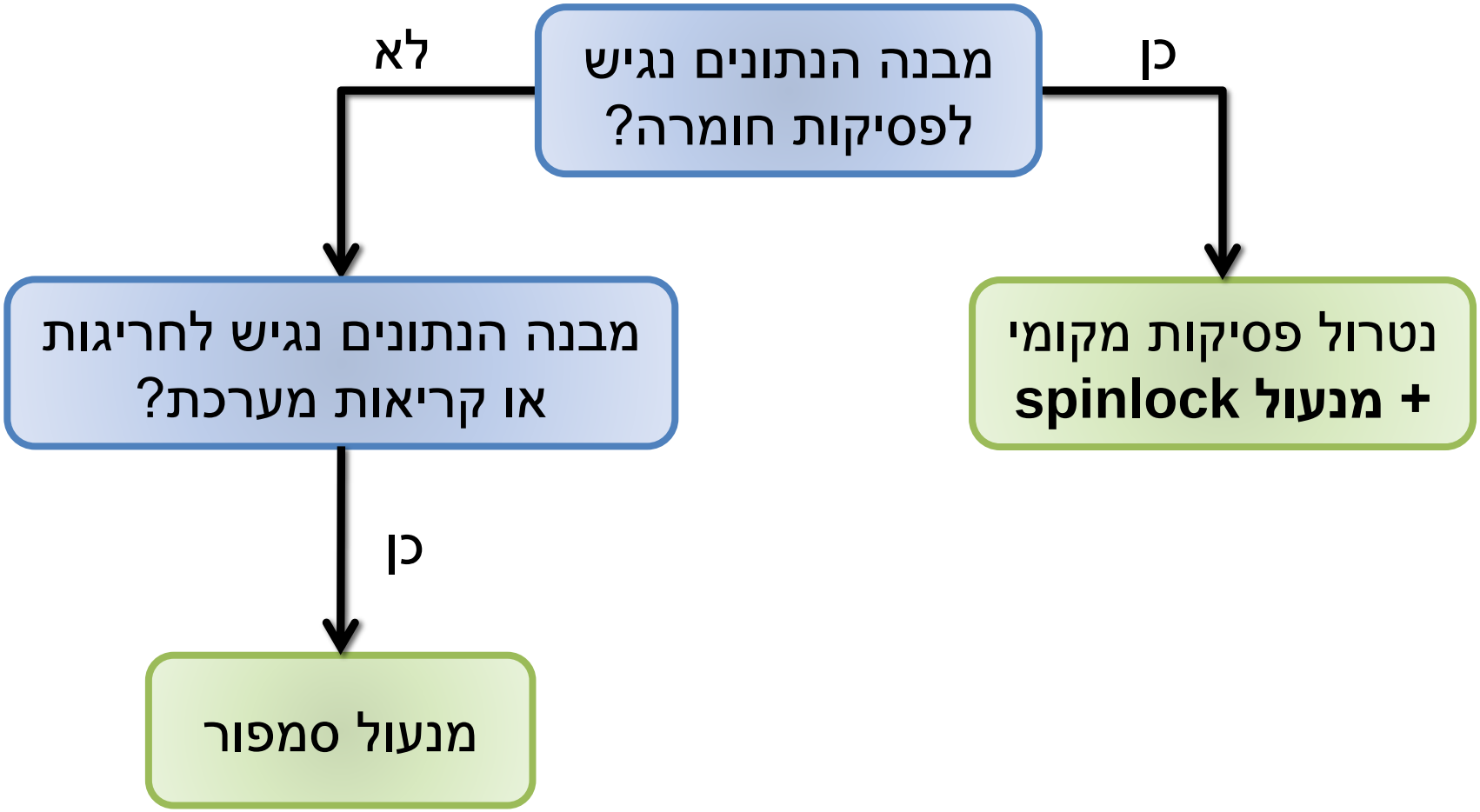
- מסלולים שאינם יכולים להיחתך:
  - פסיקת חומרה אף פעם לא קוראת לקריאת מערכת.
  - פסיקת חומרה אף פעם לא גורמת לחריגה (למעט חריגת דף).
  - קריאת מערכת / חריגה אף פעם לא גורמת לחריגה נוספת (למעט חריגת דף).

- **<u>במערכת מרובת מעבדים:</u>** כל מסלולי הבקרה יכולים להיחתך כי מעבדים שונים יכולים להריץ בו–זמנית מסלולי בקרה שונים.

אמצעי הגנה במערכת מעבד יחיד



אמצעי הגנה במערכת מרובת מעבדים



פסיקות חומרה במערכת מעבד יחיד

- פסיקת חומרה חדשה יכולה להגיע תוך–כדי ביצוע טיפול בפסיקת חומרה אחרת (קינון פסיקות חומרה).
- פסיקת חומרה יכולה להגיע גם תוך כדי טיפול בחריגה.
- לכאורה, יש להגן על מבני נתונים הנגישים לפסיקות חומרה באמצעות מנעולים. בפועל, נעילה היא בעייתית במערכת עם **מעבד יחיד**. נדגים באמצעות התרחיש הבא:
  - מסלול בקרה #1 מטפל בפסיקת חומרה כלשהי ומחזיק מנעול.
  - פסיקת חומרה אחרת מגיעה ומטופלת מיד במסלול בקרה #2 (אשר קוטע את מסלול בקרה #1).
  - מסלול בקרה #2 מנסה לתפוס את המנעול, ולכן הוא ממתין לסיום מסלול #1.
  - אבל גם מסלול בקרה #1 ממתין לסיום מסלול #2 לפני שיחזור לרוץ.
  - **קיבלנו deadlock.**

נטרול פסיקות מקומי

- כאמור, תפיסת מנעול במהלך טיפול בפסיקות חומרה עלולה להוביל ל-deadlock במערכת עם מעבד יחיד.
- לכן יש להשתמש באמצעי סנכרון אחר: **נטרול פסיקות מקומי** (Local Interrupt Disabling).
- כדי לנטרל פסיקות יש לכבות את הדגל IF של רגיסטר RFLAGS.
  - כל עוד IF==0, המעבד המקומי (שמריץ את הקטע הקריטי) לא יקבל פסיקות חומרה וכך הקטע הקריטי יתבצע בצורה אטומית.

- **שימו לב:** נטרול הפסיקות לזמן רב עלול לגרום לפגיעה בביצועים ולאובדן פסיקות חיוניות, ולכן משתמשים באמצעי זה כמוצא אחרון.

נטרול פסיקות מקומי בפונקציה schedule()

- ישנם מבני נתונים בגרעין הנגישים גם לפסיקות חומרה וגם לקריאות מערכת, למשל תור הריצה (runqueue).
  - קריאת המערכת wait() יכולה להוציא את התהליך הנוכחי מתור הריצה בפונקציה schedule().
  - פסיקת שעון יכולה להעביר את התהליך הנוכחי למקום אחר בתור הריצה בשגרה scheduler_tick().
  - אם מסלולי הבקרה של קריאת המערכת wait() ופסיקת השעון ייחתכו, תור הריצה עלול להגיע למצב לא תקין.

- **<u>במערכת מעבד יחיד:</u>** נטרול פסיקות מקומי בפונקציה schedule() הכרחי ומספיק כדי למנוע חיתוך בין מסלולי בקרה כמו בדוגמה הנ"ל.
- **<u>במערכת מרובת מעבדים:</u>** יש להוסיף מנעול spinlock (ראו בשקף הבא...)

פסיקות חומרה במערכת מרובת ליבות

- במערכת **מרובת ליבות**, מעבדים שונים יכולים לגשת בו-זמנית למבני נתונים משותפים ← יש להוסיף נעילה מעבר לחסימת הפסיקות המקומית.

- שגרות טיפול בפסיקות חומרה עושות שימוש במנעולי spinlock (מנעולים הממומשים כ-busy wait).
  - **<u>שאלה</u>**: מדוע מעדיפים מנעולי spinlock על-פני מנעולי סמפור?

1. busy wait הוא המתנה יעילה יותר כאשר מדובר בנעילות קצרות מאוד כפי שקורה בגרעין, מפני שכך נחסכת התקורה של כניסה ויציאה מהמתנה.
2. בעיית הוגנות, למשל בתרחיש הבא:
   - תהליך רץ, ובאותו הזמן מתקבלת פסיקת חומרה (למשל מהמקלדת).
   - הטיפול בפסיקה מנסה לתפוס את המנעול, אבל המנעול כבר תפוס.
   - התהליך עובר לתור המתנה מסיבה שאינה תלויה בו.

קריאות מערכת + חריגות

- כעת נניח שמבנה נתונים כלשהו נגיש לחריגות וקריאות מערכת בלבד (כלומר אינו נגיש לפסיקות חומרה).

- מה תרחישי הסינכרון הבעייתיים?

- <u>במערכת עם מעבד יחיד:</u> קריאות מערכת וחריגות לא מתבצעות בצורה אטומית (ביחס לקריאות מערכת וחריגות אחרות) בגלל שגרעין לינוקס ניתן להפקעה.

- <u>במערכת מרובת מעבדים:</u> כל קריאות המערכת והחריגות יכולות להתבצע במקביל על מעבדים שונים.

קריאות מערכת + חריגות

- טקטיקות ההגנה האפשריות:

1. להשאיר את מבנה הנתונים במצב תקין לפני הוויתור על המעבד בקריאה ל-schedule() – בדרך כלל בלתי אפשרי.
   - חסרון נוסף: בחזרה לביצוע יש לבדוק שהנתונים לא שונו ע"י מסלול בקרה אחר.

2. להבטיח אטומיות בגישה למבני הנתונים ע"י **נעילת סמפור**.
   - למשל: read() מחזיקה מנעול לכל קובץ שעליו היא עובדת.
   - תהליך שני שינסה לתפוס את הסמפור בקריאת מערכת read() יעבור לתור המתנה. בעתיד, התהליך הראשון ישחרר את הסמפור והתהליך השני יתעורר וימשיך בפעולתו.
   - הסמפור מספק הגנה מפני ביצוע במקביל גם במערכת מרובת מעבדים.