

Synchronization



© 2022 מרכז הפעלה

Operating Systems

Lecture 4 – Synchronization

Dr. Marina Kogan-Sadetsky

ניבור לפרק הרביעי - סינכרונייזציה.

הערה: הקוד המוצג במצגות - פסודו קוד בלבד.

הקוד שמאյע מהספר של טננbaum - שפת C.

Course Syllabus

- 1. Introduction
- 2. Process Management
- 3. Scheduling algorithms
- 4. **Synchronization**
 - Synchronization primitives and their equivalence
 - Deadlocks
- 5. Memory Management
- 6. File Systems
- 7. Virtualization



syn - "together"

chronos - "time"

synchronoz = "at the same time"

The mutual exclusion problem (Dijkstra, 1965)

Race Condition: system behaves differently for different timings.

How to avoid race conditions ?

We need **protocol** that guarantees
mutually exclusive access
to **critical section**.



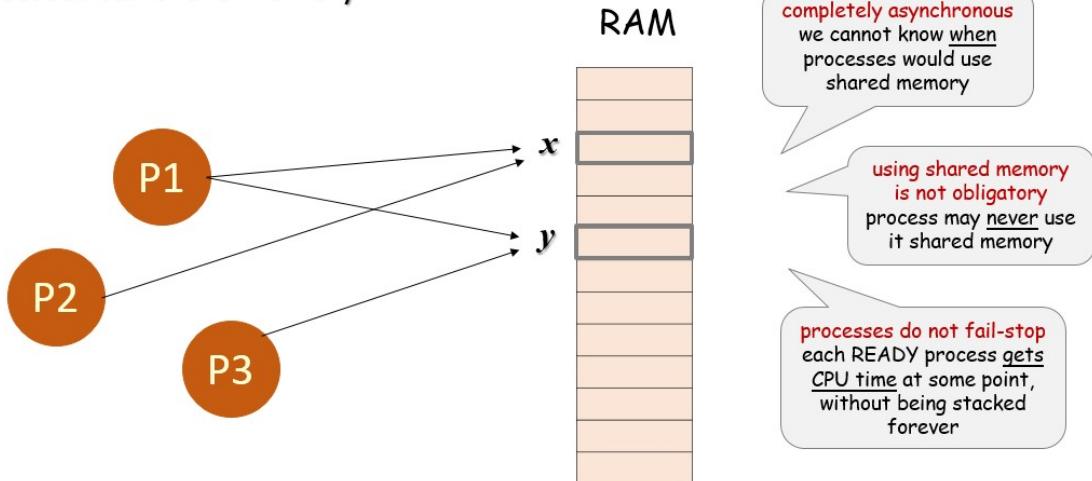
Race Condition

במערכות הפעלה מסוימות, תהליכיים שעובדים ביחד אם יכולים להיות עם זיכרון משותף כר שכל אחד מהתהליכים מסוגל לקרוא ולכתוב לזכרון המשותף. "הזכרון" יכול להיות בזיכרון הראשי (עם עדיפות לכך שהוא יהיה מבנה נתונים בקרנל) או קובץ משותף - זה לא משנה את מהות עורך התקשרות או הביעות שצפות בעזרת עורך תקשורת זה.

הבעיה אינה נתמודד בחלק זה היא בסיטואציות בהן 2 תהליכיים או יותר כתובים וקוראים דاطא המשותף להם והותצר הסופי תלי **בסדר הריצה של הפקודות**. דיקטורה האדריך סיטואציות אלו בשם **(מרוץ) race condition** (ההליכים). כמו כן מרכיב תenthalga בצורה שונה עברו זמן ריצה שונים. לשם כך נדרש בפרוטוקול המגדיר מניעה הדדית לאזורים קריטיים בקוד.

The problem model definition

- each process has unique PID
- processes can share memory



Critical Regions

ראשית נגדיר את המודל:
יש לנו מספר תהליכיים, כך שלכל תהליך יש PID משל עצמו והם יכולים לשתף זיכרון - החלק הקרייטי.
הגישה לזכרון זה אסינכרונית, כלומר אנחנו לא יודעים מתי כל אחד מהתהליכיים הולך לשמש בזיכרון משותף,
או אם בכלל תהליך ירצה לאשת לזכרון המשותף. בנוסף, אנו מניחים כי לכל תהליך במצב של ready, תהיה נקודת
זמן בה הוא יקבל זמן ריצה (do not fail-stop)

Mutual exclusion, Deadlock, Starvation

Mutual exclusion:

no two processes are at the critical section at the same time

guaranty of
correctness

Deadlock freedom:

if several processes are trying to enter their critical section,
then one of them eventually enters the critical section

guaranty of
global progress

Starvation freedom:

If a process is trying to enter its critical section,
then this process must eventually enter its critical section

guaranty of
individual progress

כיצד נמנע מרוץ תהליכיים?

המפתח למניעת הבעיה ובסייעות נוספת נוספת בהן יש משבבים משותפים למספר תהליכיים היא למצוא דרך בה 2 תהליכיים או יותר לא יוכל לבצע פעולות קרייה ופעולות כתיבה בו בזמן בזיכרון המשותף. במקרה אחרה – מניעה הדדיות (mutual exclusion).

הweeney הוא שבחלך מן הזמן תחילת מבצע תהליכי חישוב פנימיים שאינם קשורים לזכרן המשותף ורק קטעי הקוד המתעסקים בזיכרון המשותף מובילים למורות תהליכי - אלו הם קטעי הקוד הクリיטיים (critical section). כאשר נרצה שתתהליך ירצה להיכנס לקטע הクリיטי הוא יעבור דרך קוד אשר יתמוך במניעה הדדית על מנת למנוע מרווח תהליכי - הקוד הזה הוא למעשה מעשה מגנון הסינכרונייזציה.

עתה אנו נראה מספר מגנוני סינכרון. על מנת לנתח בצורה מדוייקת, להעיר לעומק ולמדוד את טיבם בשימוש במודדים הבאים:

Mutual Exclusion:

אין שני תהליכי בקטע קוד קרייטי - האחריות שלנו לכך שהקוד יהיה נכון. נכונות.

Deadlock Freedom:

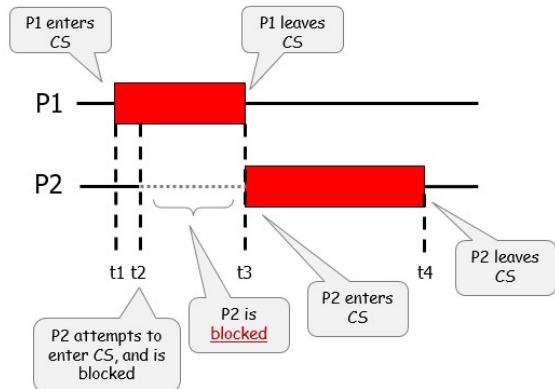
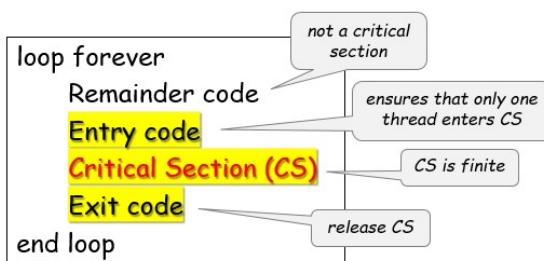
אם כמה תהליכי מנסים לגשת לקטע קרייטי, אז לפחות אחד מהם יצליח לגשת לקטע קרייטי בשלב מסוים. מכך זה מבטיח התקדמות גלובלית של האפליקציה.

Starvation Freedom:

אם תהליכי מנסה להיכנס לקטע קוד קרייטי, הוא חייב בשלב מסוים להיכנס אליו. מבטיח התקדמות אישית של כל תהליכי.

באפליקציות מסוימות אפשר לספוג פגעה בחופש מהרעה, אך השאר חייבים להתקיים. בנוסף, אנו נראה בהמשך תוכנה נוספת - FIFO.

Critical Section definition



Blocked means:

- busy wait
- sleep
- BLOCKED by OS till some event occurs

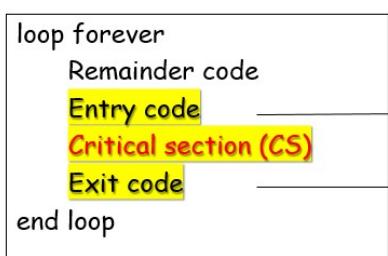
נגידו את קטע הקוד הקרייטי להיות קטע קוד סופי המערב זיכרון המשותף למספר תהליכי.

אם תהליכי רוצח לא-CS הוא חייב להריץ קוד כניסה ולבסוף קוד יציאה.
אנו נראה מספר מימושים לקטע קוד הכניסה, זה יכול לעירב wait busy, מעבר למחב שינה ועוד.

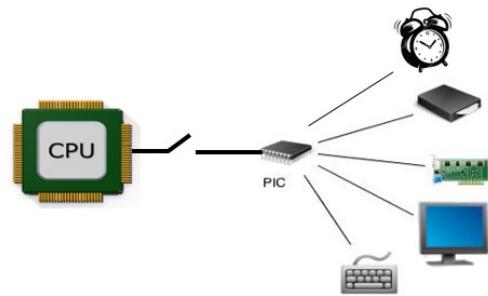
הweeney בגודל הוא שאם יש 2 תהליכי אשר מנסים לגשת ל-CS אחד מהם יכנס במצב של blocked עד שהשני יצא מה-CS.

נבחן כי מצב Blocked במודול שלנו לא מחייב התערבות של מערכת הפעלה (העברת התהליכי במצב blocked ב-PCB).

Mutual Exclusion by disabling interrupts



Disable interrupts
Enable interrupts



Since this
prevents context
switches on CPU.

Problems

- ❑ User processes should not be allowed to disable interrupts
- ❑ Disabling interrupts must be done for a very short period
- ❑ Disabling interrupts is done per-CPU

Some other processes
running on different CPU
might also enter CS.

Mutual Exclusion with Busy Wait

כעת אנו רוצים למשך קוד ל-mutual exclusion. אנו נתחילה מהפשטיטם, נתקדם לסמפורים ונס"ט עם המוניטורים (לא בהכרח כמו בג'אווה). בהתחלה ננסה לתזמן רק עם משתנים פשוטים. ראשית, נבחן כי ניתן למשך קוד למניעת הדדיות אך ורק עם משתנים פשוטים אך האלגוריתמיקה תהיה מסובכת יותר.

כיבוי פסיקות

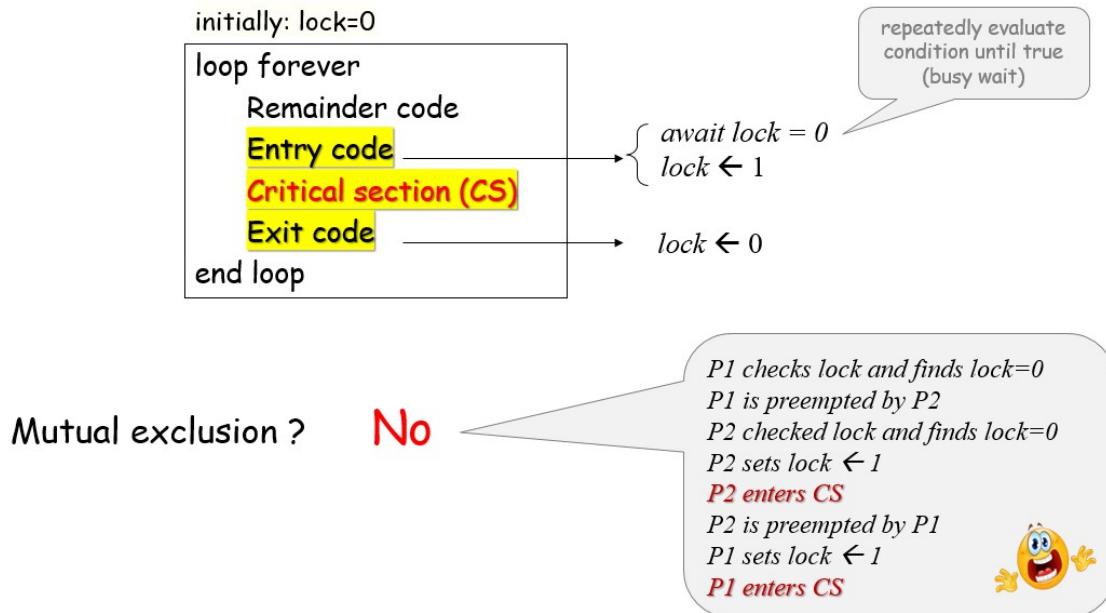
במערכת שבה קיימן מעבד אחד, ניתן להשתמש בפתרון פשוט: כל תהליך יכבה את היכולת לקבל פסיקות לפני שהוא נכנס לקוד קרייטי, ויחזר את היכולת זהו כאשר הוא יוצא משם. במקרים אחרים, אנו מנצלים את היכולת לשולט בקבלת פסיקות כדי להבטיח מניעת הדדיות.

כאשר אנו מכבים פסיקות, המעבד לא יכול לעבור לטיפול בתהליכיים אחרים, ולכן רק מעבד אחד יכול לעבוד על איזור קוד קרייטי בכל פעם.

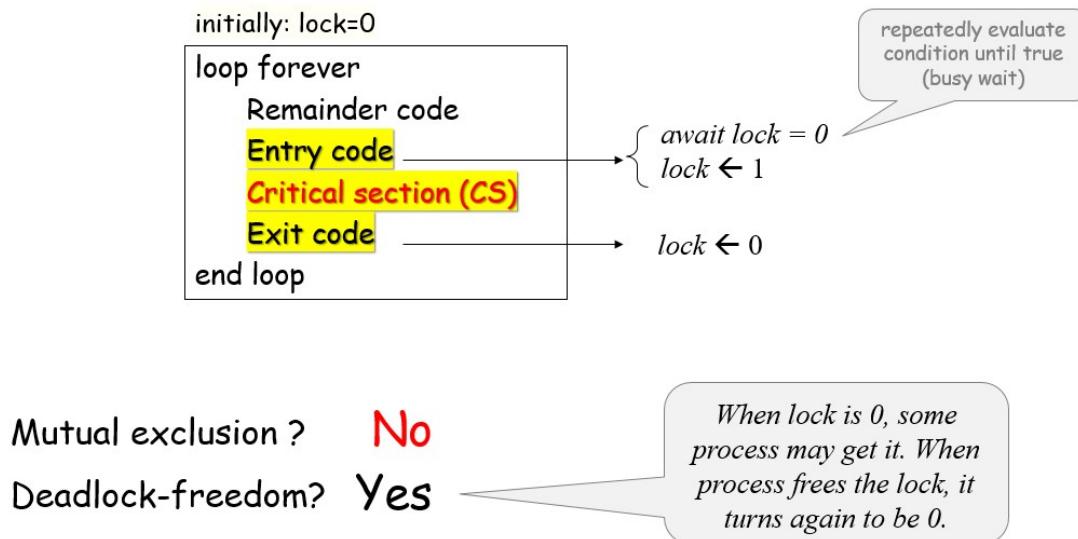
אף על פי שהמנגן זהה עובד, הוא אפשרות לא ראוייה, מכיוון שהוא מאפשר למשתמש לבצע פעולה שאמורה להיות שמורה למרחב הליבה של המערכת. בנוסף כאשר מערכת הפעלה מבצעת זאת זה, היא עשויה זאת רק לאחר זמן קצר מאוד, ולא לקטעי קוד גודלים.

בנוסף, הרעיון הזה לא יעבוד כאשר מדובר במחשב עם מספר מעבדים. זה מכיוון שכיבוי הפסיקות מתבצע רק עבור מעבד אחד. לעומת זאת אם מעבד אחד מכבה את הפסיקות כדי להכנס לאיזור קוד קרייטי, מעבד אחר עדין יכול להיכנס לאזורי איזור.

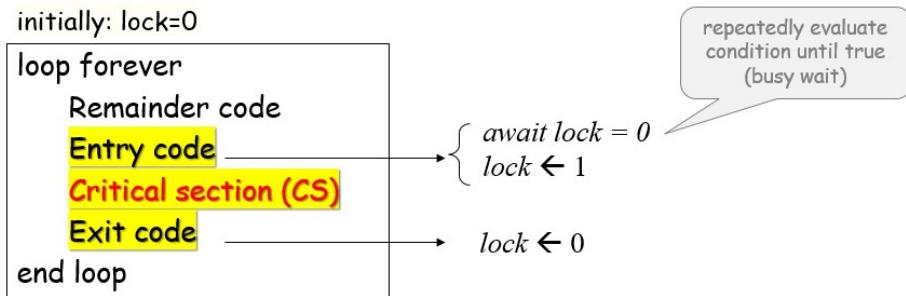
Mutual Exclusion by lock variable



Mutual Exclusion by lock variable



Mutual Exclusion by lock variable



Mutual exclusion? **No**
 Deadlock-freedom? **Yes**
 Starvation-freedom? **No**

while (true)
 P1 checks lock and finds lock=0
 P1 sets lock ← 1
P1 enters CS 😢
 P2 checks locks and finds lock=1
 P1 exits CS and sets lock ← 0

Lock Variables

```
initially: lock=0
loop forever
```

Remainder code:

```
  await lock = 0
  lock <- 1
```

Entry code

Critical section (CS)

Exit code:

```
  lock <- 0
```

end loop

כעת נבחן ניסיון נוסף לפתרון באמצעות תוכנה: נaddir משתנה גלובלי בשם lock, שמתחל ערך 0. כאשר תחיל רצחה להיכנס לאזור קוד קרייטי, הוא בודק את המניעול - אם הערך שלו הוא 0, התחיל מעדכן את המניעול ל-1 ונכנס לאזור. אם המניעול הוא 1 מלכתחילה, זה מעיד שכבר קיים תחיל אחר באזור הקוד הקרייטי, ולכן תחיל הנוכחי ייחכה עד שהמניעול יתאפס.

אף על פי שהרעיון נראה כאילו הוא עובד, הוא נופל מאותה הסיבה שדיברנו עליה לעיל. נניח שתחליך 1 הגיע, בדק את המניעול ומצא שהוא 0, כלומר אין תחיל אחר באזור. כתוצאה מהחלטה של מערכת הפעלה, בדיק בנקודה זו תחליך 1 מוחלף בתחליך 2, שאם הוא בודק את המניעול ומצא שהוא 0 (כי תחליך 1 לא הספיק לשנות אותו). תחליך 2 מעדכן את המניעול ל-1 ונכנס לאזור הקוד הקרייטי. ברגע מסוים, בעת שתחליך 2 נמצא באזור הקוד הקרייטי, מתבצעת החלפת הקשר עם תחליך 1. לאחר ותחליך 1 כבר עבר את שלב הבדיקה, גם הוא נכנס לאזור הקוד הקרייטי.

במילים אחרות, אין לנו כאן מניעה הדדית. עם זאת, יש לנו חופש מדלוקים, מכיוון שלפחות תהילך אחד יוכל להיכנס לאזרור הקרייטי. אך אין לנו חופש מהרעהה, מכיוון שהוא שתהילך ימתון לנצח, בעוד שתהילך אחר יוכל לנקח את המניעול, להיכנס לאזרור הקרייטי ולעדכן את המניעול להיות 0 אחרי שהתהליך השני בדק את הערך של המניעול.

2 נקודות ראשונות בהן אלגוריתמים אשר מנסים לתמוך במניעה הדדית נופלים:

א. מערכת הפעלה עשתה החלפת הקשיים בבדיקה בנקודה אחריה שתהילך יוצא-m-waiting ובא להיכנס לקטע קוד קרייטי ואין שום דרך לסתור לදעת שתהילך אחר כבר נכנס. בשלב זהה צריך לחסוד בכיר שאין מניעה הדדית!

ב. מצב שבו תהילך אחד יכול כל הזמן רק לצאת ולהיכנס בזמן שתהילך אחר ממתין - מצב שבו בטוח אין חופש מהרעהה.

Mutual Exclusion by strict alternation

initially: turn=0

process P0:

1. await turn=0
2. CS
3. turn \leftarrow 1

process P1:

1. await turn=1
2. CS
3. turn \leftarrow 0

Mutual exclusion? Yes
 Deadlock-freedom?
 Starvation-freedom?

We need to show that it is impossible that both P0 and P1 are in CS. We suppose that one of the processes is in CS, and proof that the other one cannot be in CS at the same time.

Note that the value of turn is 0 or 1 at any point of time. Suppose, without limitation of generality, that turn is 0. Suppose P0 and P1 both try to enter CS. Since Entry code condition is true only for P0, P1 cannot enter CS. Since P1 awaits for turn=1, and this would happen only when P0 leaves CS, then P0 and P1 could not be in CS together.

Mutual Exclusion by strict alternation

initially: turn=0

process P0:

1. await turn=0
2. CS
3. turn \leftarrow 1

process P1:

1. await turn=1
2. CS
3. turn \leftarrow 0

Mutual exclusion? Yes
 Deadlock-freedom? No
 Starvation-freedom?

suppose P0 never tries to enter CS
 then P1 would never succeed to enter CS
 while (true)
 P1 checks turn and finds turn = 0



Mutual Exclusion by strict alternation

initially: turn=0

process P0:
1. await turn=0
2. CS
3. turn \leftarrow 1

process P1:
1. await turn=1
2. CS
3. turn \leftarrow 0

May we say, that since there is no Deadlock-freedom, it means that there is no Starvation-freedom as well ?

Mutual exclusion? Yes

Deadlock-freedom? No

Starvation-freedom?

YES.

*no Deadlock-freedom => no Starvation-freedom
Starvation-freedom => Deadlock-freedom*

Mutual Exclusion by strict alternation

initially: turn=0

process P0:
1. await turn=0
2. CS
3. turn \leftarrow 1

process P1:
1. await turn=1
2. CS
3. turn \leftarrow 0

Mutual exclusion? Yes

Deadlock-freedom? No

Starvation-freedom? No

*suppose P0 never tries to enter CS
then P1 would never succeed to enter CS*

while (true)

P1 checks turn and finds turn = 0



Strict Alternation:

process P0:

1. await turn=0
2. CS
3. turn \leftarrow 1

process P1:

1. await turn=1

2. CS

3. turn <- 0

כעת נעברו בדרך שלישית, אשר יכולה להיות רלוונטיות בשפות כמו C או C++, אך לא באואה לדוגמה Ci יש לאואה garbage collector.

במנגנון זה ישנו משתנה מטיפוס integer אשר משותף ל-2 תהליכיים ושמו turn, כאשר בשלב האתחול הוא מאותחל ל-0 ותפקידו הוא לסמן את תורו של אחד התהליכים אשר יכולים להיכנס לקטע הקוד הクリיטי.

אם P0 ירצה להיכנס לקטע קרייטי הוא יdagם את turn על מנת לבדוק האם הוא 0, מה שמעיד כי זהו תורו, אחרת זהו לא תורו ולכן הוא ימתין עד שהערך יתעדכן אחרת. באותו האופן אם P1 ירצה להיכנס לקטע הקרייטי הוא יdagם את turn וימתין שערכו יהיה 1.

מימוש ב-C (מתוך ספר הקורס):

```
/* P0 */
while (TRUE) {
    while (turn != 0) /* loop */
        critical region( );
    turn = 1;
    noncritical region( );
}

/* P1 */
while (TRUE) {
    while (turn != 1) /* loop */
        critical region( );
    turn = 0;
    noncritical region( );
}
```

יש כאן mutual exclusion כי בכל נקודת זמן רק תהליך אחד מבין השניים יוכל להיכנס. אין כאן deadlock-freedom: אם turn מאותחל להיות 0, יכול להיות ש-P1 לא יכנס לקטע הקרייטי אם P0 בכלל לא מנסה להיכנס לקטע הזה. אין כאן starvation-freedom, מאותה סיבה.

משפט: אם אין deadlock-freedom אז גם אין starvation-freedom.
הערה:

בדיקות חוזרת ונשנית של משתנה נקראת wait busy ויש להימנע ממנה לאחר והיא מבזבזת זמן CPU. יש להשתמש במנגנון זה כאשר יש ציפייה הגיונית שההמתנה תהיה קצרה. מנעול המקיים תוכנה זו נקרא lock spin.

נקודה נוספת בה אלגוריתמים יכולים ליפור:

מקורה שבו קוד הכניסה בכלל לא בוחן את האפשרות שתהליך מסוים בכלל לא רוצה להיכנס לקטע קוד קרייטי - זה מתכוון לדಡלאק. ואם יש הרעה, מהמשפט אוטומטית יש הרעה (כי זה יפול על אותו תרחיש זמן

Mutual Exclusion by binary flag array

initially: bool $b[2] = \{\text{false}, \text{false}\}$

process P0:

1. $b[0] \leftarrow \text{true}$
2. await $b[1]=\text{false}$
3. CS
4. $b[0] \leftarrow \text{false}$

process P1:

1. $b[1] \leftarrow \text{true}$
2. await $b[0]=\text{false}$
3. CS
4. $b[1] \leftarrow \text{false}$

*$b[i]$ is true when
Pi wants to
execute CS*

Mutual exclusion? Yes

Deadlock-freedom?

Starvation-freedom?

We need to show that it is impossible that both P0 and P1 are in CS. We suppose that one of the processes is in CS, and proof that the other one cannot be in CS at the same time.

Note: flags[i] is only changed by Pi.

Note: Pi first changes $b[i]$, and only then waits for $b[1-i]$ to be false.

Suppose, without limitation of generality, P0 is in CS. When P0 enters CS, $b[0]=\text{true}$ and $b[1]=\text{false}$. So P1 at that moment still did not set $b[1]$ to be true. This means that P1 starts waiting only after P0 is already in CS, and would wait till $b[0]$ would be false to enter CS. Since turn becomes 1 only after P0 exits CS, P0 and P1 cannot be together in CS.

Mutual Exclusion by binary flag array

initially: bool $b[2] = \{\text{false}, \text{false}\}$

process P0:

1. $b[0] \leftarrow \text{true}$
2. await $b[1]=\text{false}$
3. CS
4. $b[0] \leftarrow \text{false}$

process P1:

1. $b[1] \leftarrow \text{true}$
2. await $b[0]=\text{false}$
3. CS
4. $b[1] \leftarrow \text{false}$

*$b[i]$ is true when
Pi wants to
execute CS*

Mutual exclusion? Yes

Deadlock-freedom? No

Starvation-freedom?

*P0 sets $b[0] \leftarrow \text{true}$
P1 sets $b[1] \leftarrow \text{true}$
P0 and P1 would wait forever*



Mutual Exclusion by binary flag array

initially: bool $b[2] = \{\text{false}, \text{false}\}$

process P0:

1. $b[0] \leftarrow \text{true}$
2. await $b[1]=\text{false}$
3. CS
4. $b[0] \leftarrow \text{false}$

process P1:

1. $b[1] \leftarrow \text{true}$
2. await $b[0]=\text{false}$
3. CS
4. $b[1] \leftarrow \text{false}$

$b[i]$ is true when
 P_i wants to
execute CS

Mutual exclusion? Yes

Deadlock-freedom? No

Starvation-freedom? No

P_0 sets $b[0] \leftarrow \text{true}$
 P_1 sets $b[1] \leftarrow \text{true}$
 P_0 and P_1 would wait forever



binary flag array

initially: bool $b[2] = \{\text{false}, \text{false}\}$

process P0:

1. $b[0] \leftarrow \text{true}$
2. await $b[1]=\text{false}$
2. CS
3. $b[0] \leftarrow \text{false}$

process P1:

1. $b[1] \leftarrow \text{true}$
2. await $b[0]=\text{false}$
2. CS
3. $b[1] \leftarrow \text{false}$

באו נוסיף מערך בגודל של שניים ונציג את התהילכים הבאים:

תהליך 0:

תהליך זה מעדכן את הדגל שלו להיות "אמת", מה שמשמעו שהוא מעוניין להיכנס לאזור הקוד הקרייטי. התהליך מתבצע עד שתתהליך 1 יעדכן את הדגל שלו להיות "שקר", מה שמשמעו שהוא לא מעוניין להיכנס לאזור הקוד הקרייטי. ברגע שתתהליך 1 מעדכן את הדגל שלו להיות "שקר", תהליך 0 מבצע את האזור הקוד הקרייטי ולאחר מכן מעדכן את הדגל שלו להיות "שקר".

תהליך 1:

תהליך זה מתנהל באופן סימטרי לתהליך 0.

במנגנון זה, יש לנו מניעה הדדית, מכיוון שבכל נקודת זמן רק תהליך אחד יכול להיכנס לאזור הקוד הקритי. אך אין לנו חופש מדלוקים, מכיוון שהוא חייב ששני התהליכים מבצעים את השורה הראשונה בו בזמןית, אז מתקעים בשורה השנייה. מכאן שגם לנו חופש מהרעה.

נקודת נוספת בה אלגוריתמי מנעה הדדית יכולים ליפול:

קוד הכנסה לא בוחן את האפשרות שכמה תהליכי כניסה נוכנים בו בזמןית ויכולם להיכנס ל'busy wait'.

Peterson's 2-process algorithm (Peterson, 1981)

combines flag array and strict alternation together

initially: $b[0]=\text{false}$, $b[1]=\text{false}$
initial value of turn immaterial

process P0:

1. $b[0] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 0$
3. **await ($b[1]=\text{false}$ or $\text{turn}=1$)**
4. **CS**
5. $b[0] \leftarrow \text{false}$

$b[i]$ is true when process P_i wants to execute CS

$\text{turn} = 1-i$ means P_i may enter CS



Larry L. Peterson

process P1:

1. $b[1] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 1$
3. **await ($b[0]=\text{false}$ or $\text{turn}=0$)**
4. **CS**
5. $b[1] \leftarrow \text{false}$

Peterson's 2-process algorithm (Peterson, 1981)

initially: $b[0]=\text{false}$, $b[1]=\text{false}$
initial value of turn immaterial

process P0:

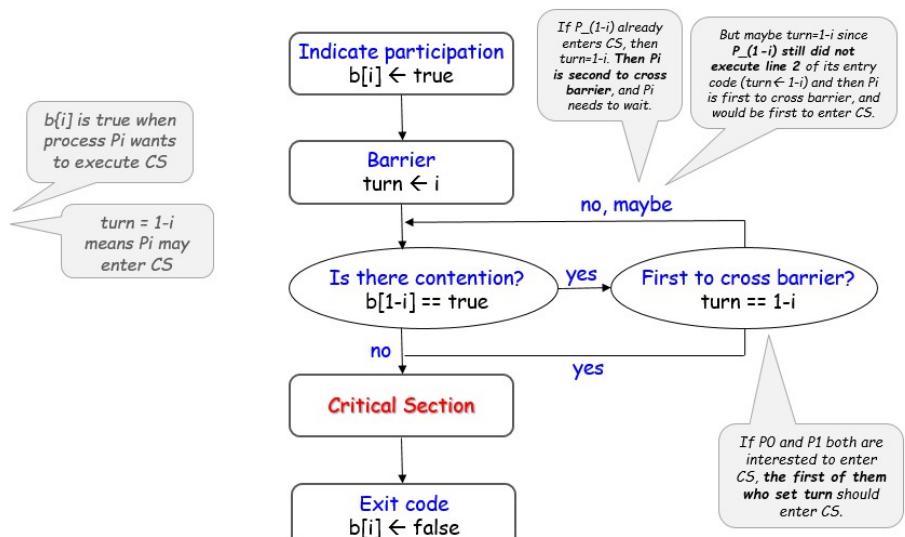
1. $b[0] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 0$
3. **await ($b[1]=\text{false}$ or $\text{turn}=1$)**
4. **CS**
5. $b[0] \leftarrow \text{false}$

$b[i]$ is true when process P_i wants to execute CS

$\text{turn} = 1-i$ means P_i may enter CS

process P1:

1. $b[1] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 1$
3. **await ($b[0]=\text{false}$ or $\text{turn}=0$)**
4. **CS**
5. $b[1] \leftarrow \text{false}$



Peterson's 2-process algorithm (Peterson, 1981)

initially: $b[0]=\text{false}$, $b[1]=\text{false}$
initial value of turn immaterial

process P0:

1. $b[0] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 0$
3. **await ($b[1]=\text{false}$ or $\text{turn}=1$)**
4. **CS**
5. $b[0] \leftarrow \text{false}$

Suppose P_0 is first to enter CS. Then $b[0]=\text{true}$ and $\text{turn}=1$. This means that P_1 should wait till $b[0]$ becomes false, or till turn becomes 0 - this happens when P_0 exits CS and still did not re-entered it.

Intuitively, since **turn value is 1 or 0**, only one of the processes can enter CS.

process P1:

1. $b[1] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 1$
3. **await ($b[0]=\text{false}$ or $\text{turn}=0$)**
4. **CS**
5. $b[1] \leftarrow \text{false}$

Mutual exclusion? Yes

Deadlock-freedom?

Starvation-freedom?

Peterson's 2-process algorithm (Peterson, 1981)

initially: $b[0]=\text{false}$, $b[1]=\text{false}$
initial value of turn immaterial

process P0:

1. $b[0] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 0$
3. **await ($b[1]=\text{false}$ or $\text{turn}=1$)**
4. **CS**
5. $b[0] \leftarrow \text{false}$

Suppose P_0 and P_1 both are interested to enter CS. Since **value of turn is 1 or 0**, one of them could enter CS.

If only P_i is interested to enter CS, then P_i can enter since $b[1-i]=\text{false}$.

process P1:

1. $b[1] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 1$
3. **await ($b[0]=\text{false}$ or $\text{turn}=0$)**
4. **CS**
5. $b[1] \leftarrow \text{false}$

Mutual exclusion? Yes

Deadlock-freedom? Yes

Starvation-freedom?

Peterson's 2-process algorithm (Peterson, 1981)

initially: $b[0]=\text{false}$, $b[1]=\text{false}$
initial value of turn immaterial

process P0:

1. $b[0] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 0$
3. **await ($b[1]=\text{false}$ or $\text{turn}=1$)**
4. **CS**
5. $b[0] \leftarrow \text{false}$

If P_i is waiting, then $P(1-i)$ is in CS. When $P(1-i)$ exits CS and tries to re-enter CS, $P(1-i)$ sets turn to be $1-i$. In order to let $P(1-i)$ to re-enter CS, $b[1-i]$ must be false, or turn must be i . This is possible only if P_i would execute CS, exit it, and try to enter CS once again.

process P1:

1. $b[1] \leftarrow \text{true}$
2. $\text{turn} \leftarrow 1$
3. **await ($b[0]=\text{false}$ or $\text{turn}=0$)**
4. **CS**
5. $b[1] \leftarrow \text{false}$

Mutual exclusion? Yes

Deadlock-freedom? Yes

Starvation-freedom? Yes

אלגוריתם פטרסון ל-2 תהליכיים

initially: $b[0]=\text{false}$, $b[1]=\text{false}$

initial value of turn immaterial

process P0:

```
b[0] <- true  
turn <- 0  
await (b[1]=false or turn=1)  
CS  
b[0] <- false
```

process P1:

```
b[1] <- true  
turn <- 1  
await (b[0]=false or turn=0)  
CS  
b[1] <- false
```

זהו הרעיון הראשון שהוא מוצלח. הרעיון הוא בעצם שילוב של הרעיונות הקודמים. הוא כולל 2 פרוצדורות עבור 2 תהליכיים, כאשר הרעיון הוא לקחת בחשבון גם את ניהול תור וגם את רצונו של התהילה להיכנס לקטע קוד קרייטי.

אתחול:

זכרון משותף: מערך בגודל 2 אשר כל תא בו מקבל את הערך 0 או 1 (כລומר boolean) שתפקידו לייצג את הרצון של כל תהליך להיכנס לקטע קוד קרייטי או לא. בנוסף ישנו משתנה גלובלי שנקרא turn שהערך ההתחלתי אינו חשוב.

תיאור הפתרון:

נניח כי בהתחלה אף אחד מהתהליכיים אינו נמצא בקטע קוד קרייטי. לאחר זמן מה, תהליך 0 מעוניין לגשת לקטע קוד קרייטי. לכן הוא מרים את הדגל שלו במערך d , מעדכן שהטור הוא שלו (כלומר מעדכן את turn להיות 0). מאחר וכל התנאים המבוקשים מתקיימים הוא נכנס לקטע הקוד הקרייטי. ברגע שתהליך 1 ירצה להיכנס לקטע זה, הוא יכנס למצב המתנה עד שהערך $[\text{d}[0]]$ יהיה שלילי, או במקרה אחרות כאשר תהליך 0 כבר אינו מעוניין להיכנס לקטע הקוד קרייטי.

עבור המקרה ש-2 תהליכיים מעוניינים באותו זמן להיכנס לקטע הקוד קרייטי, הם שניים ינסו לשנות את הערך של turn ואז זה שהוא מספיק Zariz לעדכן אותו יכנס קודם.

יש כאן חסימה mutual exclusion כי באופן אינטואיטיבי, מכיוון שהערך של turn הוא 1 או 0, רק אחד מהתהליכיים יכול להיכנס לקטע הקרייטי. נניח שתהליך 0 הוא הראשון שנכנס לקטע הקוד הקרייטי. אז $[\text{d}[0]]$ הופך לאמת ו- turn הופך ל-1. זה אומר שתהליך P1 צריך להמתין עד ש- $[\text{d}[0]]$ יהיה לשקר, או עד ש- turn יהיה 0 - זה קורה כאשר 0 P יצא מהקוד הקרייטי ועודין לא נכנס אליו שוב.

יש כאן deadlock-freedom: נניח ש-0 P ו-1 P שניים מעוניינים להיכנס לקטע הקוד הקרייטי. מכיוון שערך ה- turn הוא 1 או 0, אחד מהם יכול להיכנס לאזרור הקוד הקרייטי.

יש כאן starvation-freedom כי כאשר תהליך 0 יצא מקטע הקוד קרייטי הוא מבצע קוד יציאה שבכרכח יכנס את התהליך השני לקטע הקוד הקרייטי. ככלומר, אם רק P_i מעוניין להיכנס לקטע הקוד הקרייטי, אז P_i יכול להיכנס מכיוון ש- $[\text{d}[i]]$ הוא שקר. אם P_i ממתין, אז $(\neg 1)P$ נמצא בקוד הקרייטי. כאשר $(\neg 1)P$ יוצא מהקוד הקרייטי ומנסה להיכנס שוב, $(\neg 1)P$ מגדיר את ה- turn להיות 1-. כדי לאפשר ל- $(\neg 1)P$ להיכנס שוב לקטע הקוד הקרייטי, $[\text{d}[1]]$ חייב להיות שקר, או שה- turn חייב להיות ?. זה אפשרי רק אם P_i יבצע את הקוד הקרייטי, יצא ממנו, ונסה להיכנס שוב.

זה נראה פתרון מעולה, אך יש בו בעיה אחת: ה-entry code הוא שונה לכל תהליך. לכן יש צורך במימוש קוד כניסה ייחודי לכל תהליך שיש סיכון ברגע שעוברים לארסה כללית יותר.

על כן, מסיבה זו ומכך שיש פתרונות עילים יותר, אין שימוש בפתרון זה.

מימוש ב-C:

```
#define FALSE 0
#define TRUE 1
#define N 2          /* number of processes */
int turn;           /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */
void enter region(int process); /* process is 0 or 1 */
{
    int other;        /* number of the other process */
    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process; /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */
;
```

```

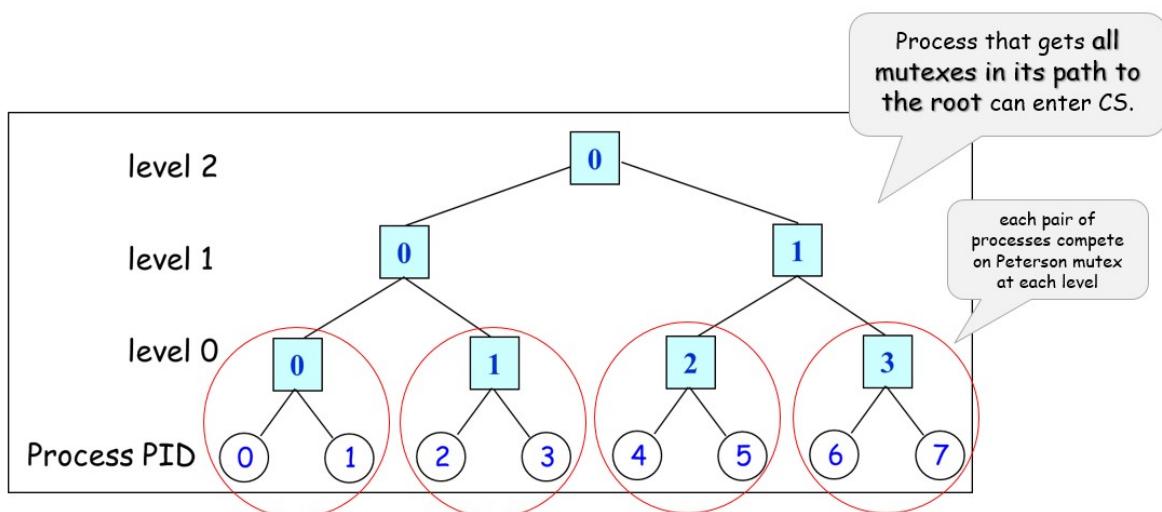
}

void leave_region(int process) /* process: who is leaving */
{
interested[process] = FALSE; /* indicate departure from critical region
*/
}

```

נקודות בהן אנו יכולים לראות כי אלגוריתמים מצלחים:
א. יש מניעה הדדית בגלל שיש כאן בינהarity: או שתהילר אחד לוקח את turn או שתהילר השני לוקח את turn, אבל לא יכול להיות שניהם יקחו אותו בו זמן.
ב. מאותה סיבה לא יכולה להיות כאן דלקרים או הרובוט - הבחירה כאן היא בינהarity בנויגוד לעיל.

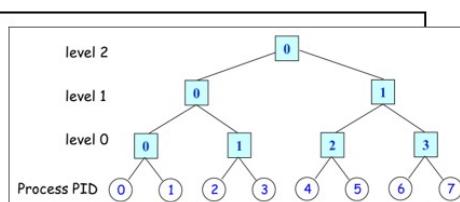
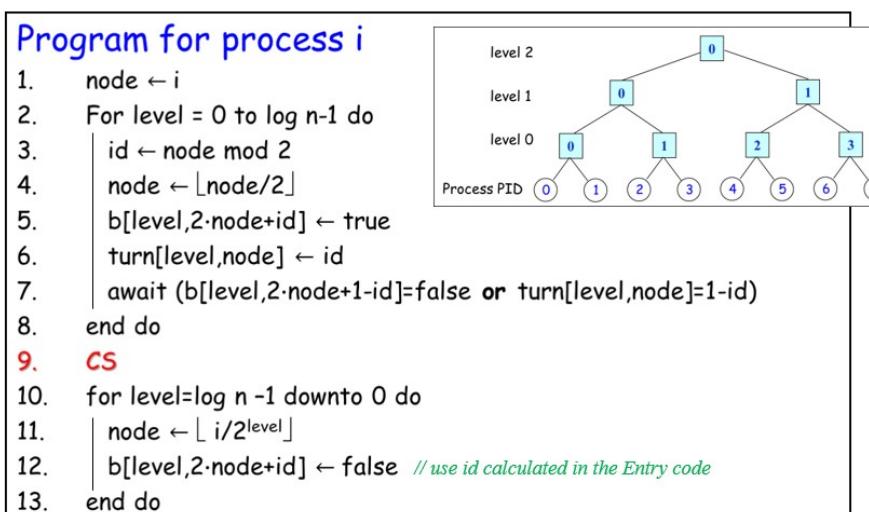
N-process mutual exclusion: tournament tree



N-process mutual exclusion: tournament tree

Shared variables: $b[\log n-1, n] \leftarrow \{\text{false}\}$, $\text{turn}[\log n-1, n/2]$

Per process local variables: **level**, **node**, **id**.



level - current level the process
node - process node number in its current level
id - 0 means left in the pair, 1 means right in the pair

	0	1	2	3	4	5	6	7
level 0	F	F	F	F	F	F	F	F
level 1	F	F	F	F	F	F	F	F
level 2	F	F	F	F	F	F	F	F

	0	1	2	3
level 0				
level 1				
level 2				

N-process mutual exclusion: tournament tree

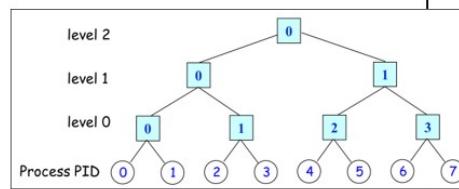
Shared variables: $b[\log n-1, n] \leftarrow \{\text{false}\}$, $\text{turn}[\log n-1, n/2]$

Per process local variables: **level**, **node**, **id**.

Program for process i

```

1. node ← i
2. For level = 0 to log n-1 do
3.   id ← node mod 2
4.   node ← ⌊node/2⌋
5.   b[level,2·node+id] ← true
6.   turn[level,node] ← id
7.   await (b[level,2·node+1-id]=false or turn[level,node]=1-id)
8. end do
9. CS
10. for level=log n-1 downto 0 do
11.   node ← ⌊i/2level⌋
12.   b[level,2·node+id] ← false // use id calculated in the Entry code
13. end do
    
```



P5:

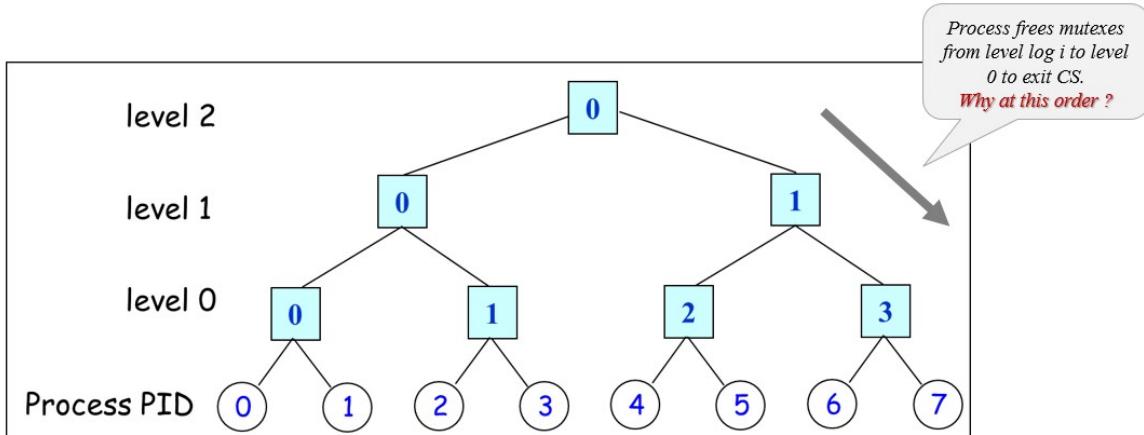
```

node ← 5, level = 0
id ← 5%2 = 1
node ← ⌊5/2⌋ = 2
b[0, 5] ← true
turn[0, 2] ← 1
await(b[0,4]=false or turn[0,2]=0)
    
```

	0	1	2	3	4	5	6	7
level 0	F	F	F	F	T	F	F	
level 1	F	F	F	F	F	F	F	
level 2	F	F	F	F	F	F	F	

	0	1	2	3
level 0		1		
level 1				
level 2				

N-process mutual exclusion: tournament tree



הגרסה עברית תהליכי - עצ תחרות:

Shared variables: $b[\log n-1, n] \leftarrow \{\text{false}\}$, $\text{turn}[\log n-1, n/2]$

Per process local variables: **level**, **node**, **id**.

Program for process i

```

1. node <- i
2. For level = 0 to log(n-1) do
    3. id <- node mod 2
        
```

```

4. node <- [node/2]
5. b[level,2.node+id] <- true
6. turn [level,node] <- id
7. await (b[level,2*node+1-id]=false or turn[level,node]=1-id)
8. end do
9. CS
10. for level=log(n-1) downto 0 do
11. node <- [i/(2^level)]
12. b[level,2.node+id] <- false // use id calculated in the Entry code
13. end do

```

הארסה הכללית של אלגוריתם פטרסון מתייחסת ל蹶ה שבו יש ח תהליכיים, כאשר ח הוא חזקה של 2. האלגוריתם משתמש במבנה של עץ תחרות, שבו כל זוג תהליכיים מתחברים זה לזה על הכניסה לרמה הבאה בעץ, עד שמאגים לשורש של העץ. התהיליך שמאגים לשורש הוא זה שמנnis לקטע הקוד הקרייטי.

הקוד כניסה מתחילה מהתחתית של העץ ומטקדם למעלה, והקוד יצאה מתחילה מהשורש של העץ ומטקדם למטה.

כאשר תחיליך רוצה להיכנס לקטע הקוד הקרייטי, הוא מבצע את הפעולות הבאות:

א. מעדכן את המשתנה `node` להיות ה-ID שלו.

ב. מעבר על כל הרמות בעץ. בכל רמה, התהיליך מעדכן את המזהה שלו להיות שארית החלוקה של `node` ב-2, ושומר ב-`node` את הערך השלים התיכון של תוצאה החלוקה.

ג. התהיליך מעדכן את הדגל שלו במערך `b` להיות `true`, מה שמצוין שהוא מעוניין להיכנס לקטע הקוד הקרייטי ברמה הנוכחית. לאחר מכן, הוא מעדכן את המשתנה `turn` במקום המתאים להיות ה-ID שלו, וממתין עד שהטהיליך השני לא מעוניין יותר בקטע הקוד הקרייטי, או שהטור מתעדכן להיות שלו.

לאחר הלולאה, התהיליך מבצע את קטע הקוד הקרייטי.

בקוד הייצאה, התהיליך מעדכן את כל הערכים במערך `b` להיות `false` מהו `true` היה. הוא עובר על כל הרמות מהשורש למטה, מעדכן את המשתנה `node` וمعدכן את המשתנה `b` במקום המתאים להיות `false`, מה שמצוין שהטהיליך לא מעוניין יותר להיכנס לקטע הקוד הקרייטי.

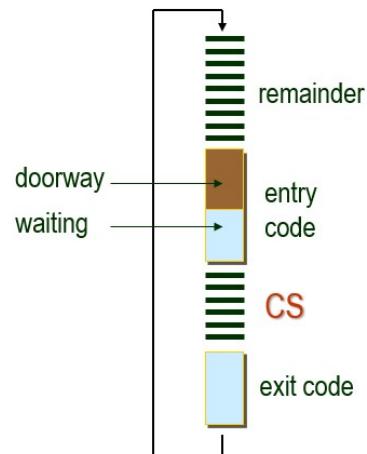
השימוש בשאריות החלוקת מאפשר לנו לומר אם מדובר בתהליך הימני או השמאלי בזוג.

Fairness: First in First Out (FIFO)

- + Mutual exclusion
- + Deadlock-freedom
- + Starvation-freedom

Fair algorithm ??

FIFO - if **p** started waiting to enter CS, and **q** has not yet started the doorway, then **q** will not enter CS before **p**.

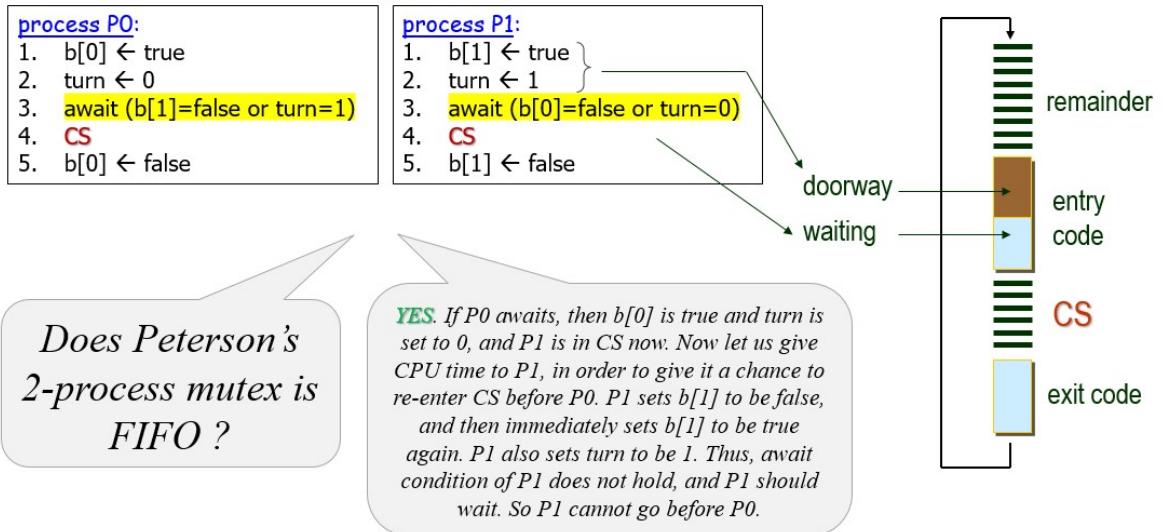


Fairness: First in First Out (FIFO)

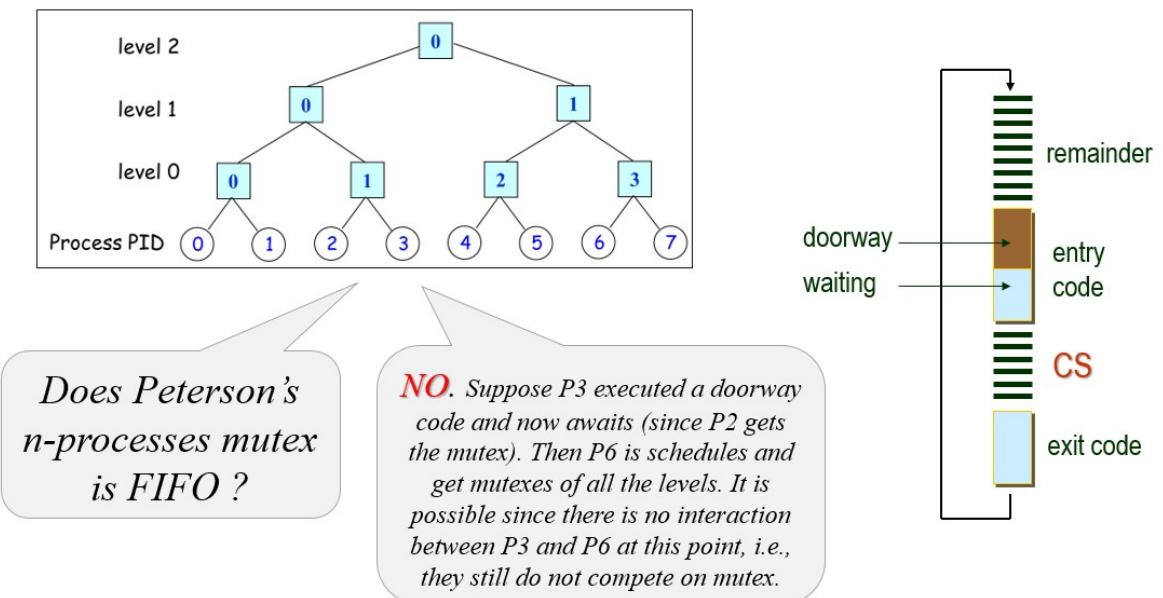
FIFO - if **p** started waiting to enter CS, and **q** has not yet started the doorway, then **q** will not enter CS before **p**.



Fairness: First in First Out (FIFO)



Fairness: First in First Out (FIFO)



Fairness: First in First Out (FIFO)

What is the relation between **FIFO** and **Starvation-freedom** ?

1. No one implies the other
2. FIFO => Starvation-freedom
3. Starvation-freedom => FIFO

FIFO guarantees order of enters, but we should also guaranty ability of enters to CS, i.e., guarantee deadlock-freedom.

Suppose an algorithm which does not allow to enter CS. It is FIFO by definition, but not Starvation-free.

שאלה: האם שלושת המדרדים הראשונים מספיקים? התשובה היא כן, אך אנו מעוניינים גם במנגנון שיהיה הוגן, זהה תלוי באפליקציה שאנו מפתחים.

הגדירות: אנו מפרקим את קוד הכניסה לשני חלקים:
א. "Doorway": זהו השלב בו התחילה מחליט שהוא רוצה להיכנס לקטע הקוד הקритי, אך עדין לא מתחילה להמתין.
ב. "Waiting": זהו השלב של ההמתנה עצמה.

הגדרת התכונה: אם ישנים שני תהליכי P-Q, שהוצאים להיכנס לקטע הקוד הקритי, ו-P מתחילה להמתין בשלב הכניסה ו-Q עדין לא הצהיר שהוא מוכן להיכנס (כלומר, עדין לא נכנס לשלב ה-"Doorway"), אז Q לא יכול להיכנס לקטע הקוד הקритי לפני P.

אבחנות:

- א. ההגדירה לא מספרת متى כן מתקיים FIFO, היא מספרתمتיאין FIFO.
- ב. המשמעות של FIFO היא שאחד חייב לסיים את ה-doorway לפני השני בכל חשב להיכנס.
- ג. אינטואיציה: בתור בית מרקחת, אנחנו רוצים שהוא הגיע והמתין בתור עד שקיבל שירות, קיבל אותו לפני מי שהוא עתה הגיע בכלל לבית מרקחת.

במקרה הפרטי של פטרסון יש FIFO:

אם P0 ממתין, אז [0] הוא אמת וה-turn מוגדר להיות 0, ו-P1 נמצא כעת בקטע הקוד הקритי. נניח שאנו חנכו מעניקים זמן מעבד ל-P1, כדי让他 להזמין להיכנס שוב לקטע הקוד הקритי לפני P0. P1 מעדכן את [1] בלא להיות שקר, והוא מיד מעדכן את [1] להיות אמת שוב. P1 אם מעדכן את ה-turn להיות 1. כך שתנאי ההמתנה של P1 לא מתקיים, ו-P1 צריך להמתין. לכן, P1 לא יכול להיכנס לפני P0.

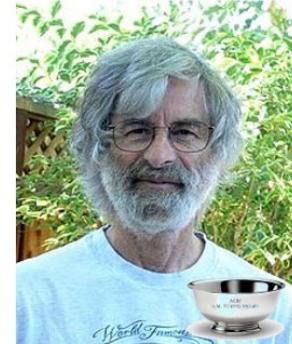
במקרה הכללי של פטרסון אין FIFO:

נניח ש-P3 ביצע את קוד ה-"doorway" וכעת ממתין (מכיוון ש-P2 מחזיק במנועול). אז P6 מתזמן ומקבל את המנגולים של כל הרמות. זה אפשרי מכיוון שאין כל קשר בין P3 ל-P6 בנקודה זו, כלומר, הם עדין לא מתחברים כלל המנועול.

הערות:

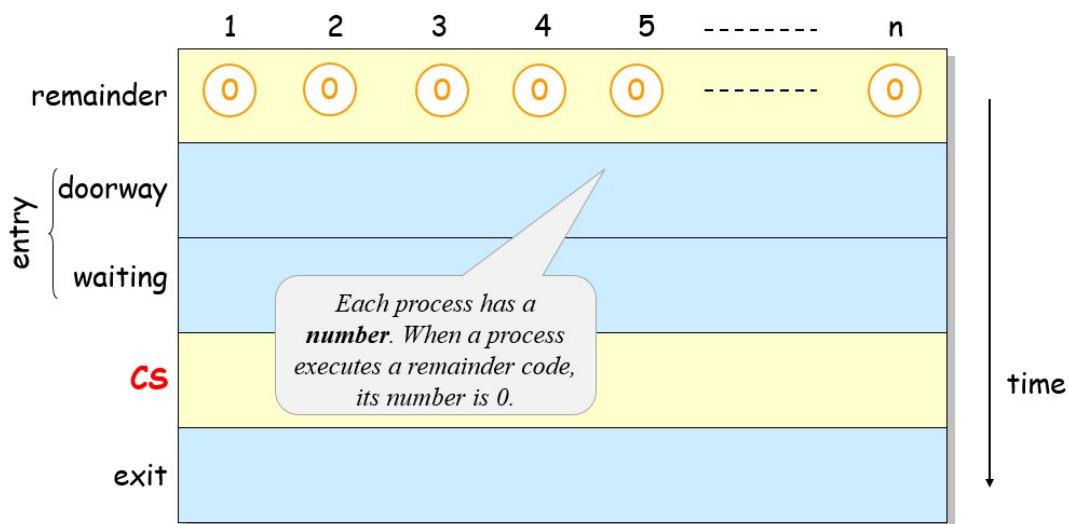
- א. כדי להראות שהאלגוריתם תומך ב-FIFO, יש להראות את 2 המקרים (מי התחליל קודם).
- ב. אין קשר בין FIFO לבין starvation-freedom! קיום האחד לא גורר את קיומ השני.

Lamport's Bakery Algorithm (1974) [the original paper](#)

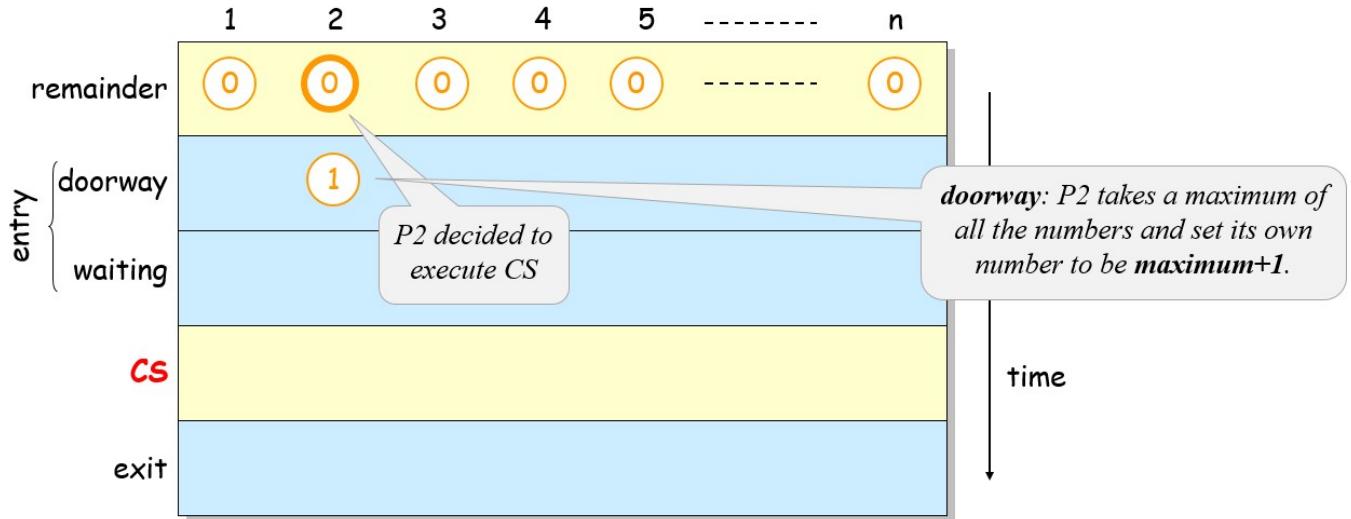


Leslie Lamport

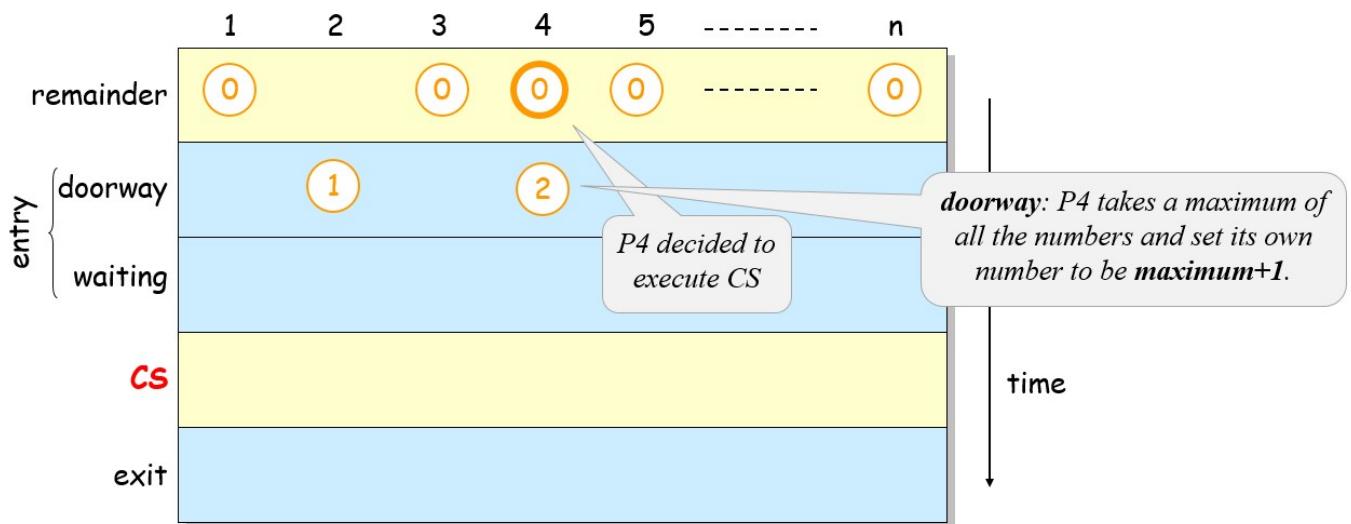
Lamport's Bakery Algorithm (1974)



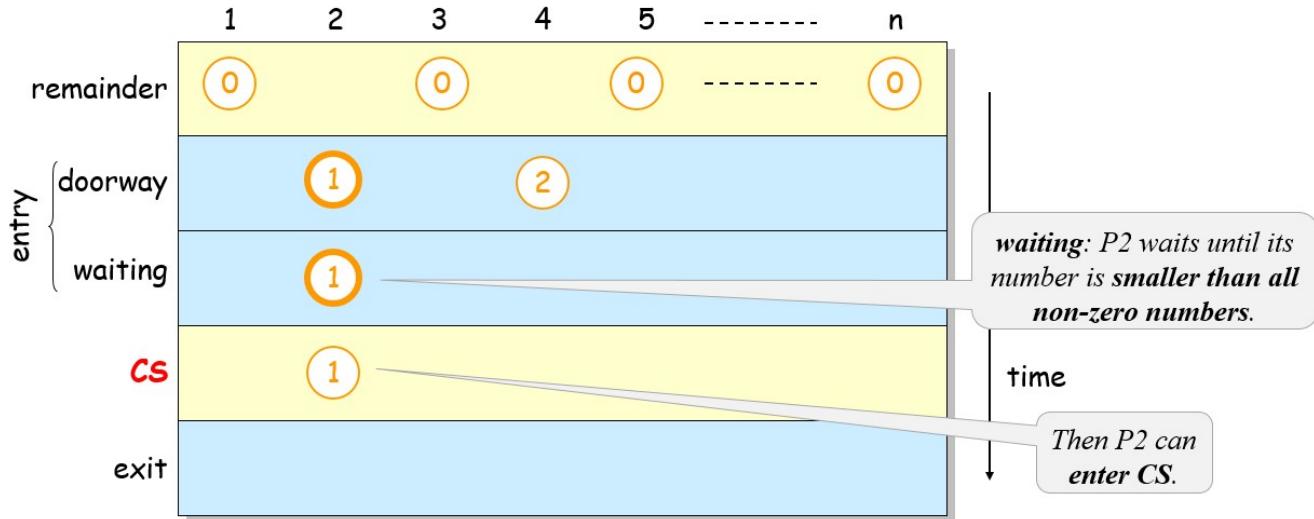
Lamport's Bakery Algorithm (1974)



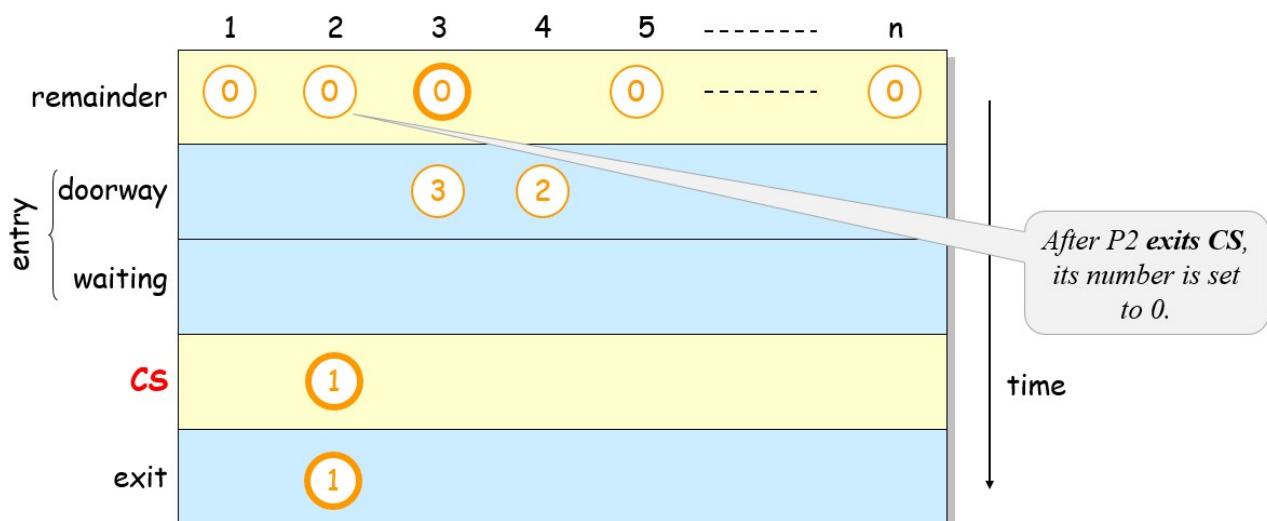
Lamport's Bakery Algorithm (1974)



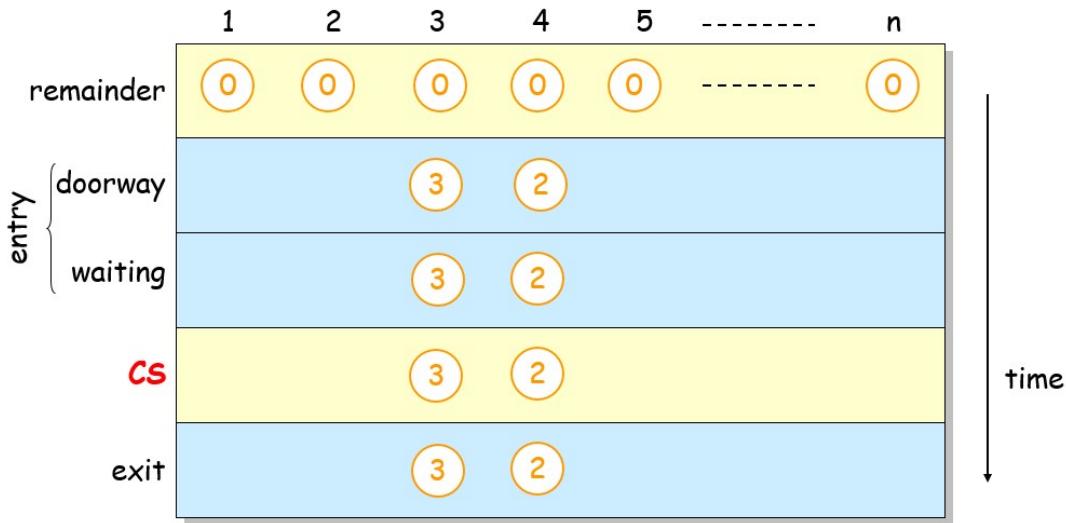
Lamport's Bakery Algorithm (1974)



Lamport's Bakery Algorithm (1974)



Lamport's Bakery Algorithm (1974)



Lamport's Bakery Algorithm

אלגוריתם זה הוא אלגוריתם מניעה הדדית מרכזי שנוצר על ידי למפורט בשנת 1974. האלגוריתם נוצר עם המחשבה לתרום במספר תהליכיים כללי ולא רק במקרה המיעוד של שני תהליכיים. הוא מבוסס על מערכת כרטיסים פשוטה שבה לכל תהליך מוקצה מספר כרטיס, ובכל שלב שבו הכנסה לקטע הקרייטי פתוחה, התהילר עם המספר הכי נמוך מקבל עדיפות והוא הראשון שנכנס לקטע הקרייטי.

הערה: המציג מתארת חמשה שלבים. ראיתי במקומות אחרים שארבעה שלבים מסוימים: בקשה, המתנה, קטע קרייטי, יציאה.

ניתן לתאר את האלגוריתם בחמשה מצבים בצורה כללית:
א. מצב ה-"remainder" הוא המצב ההתחלתי. במצב זה, כל תהליך מגדר את מספר הcarteis שלו ומצהיר שהוא מתכוון להיכנס לקטע הקרייטי. בחירות המספר מתבצעת על ידי מציאת המקסימום מבין הערכים הקיימים והוספת 1 אליום.

ב. שלב ה-"doorway": מציג את הנקודה שבה התהילר ממתין לפני שהוא נכנס לקטע הקרייטי.

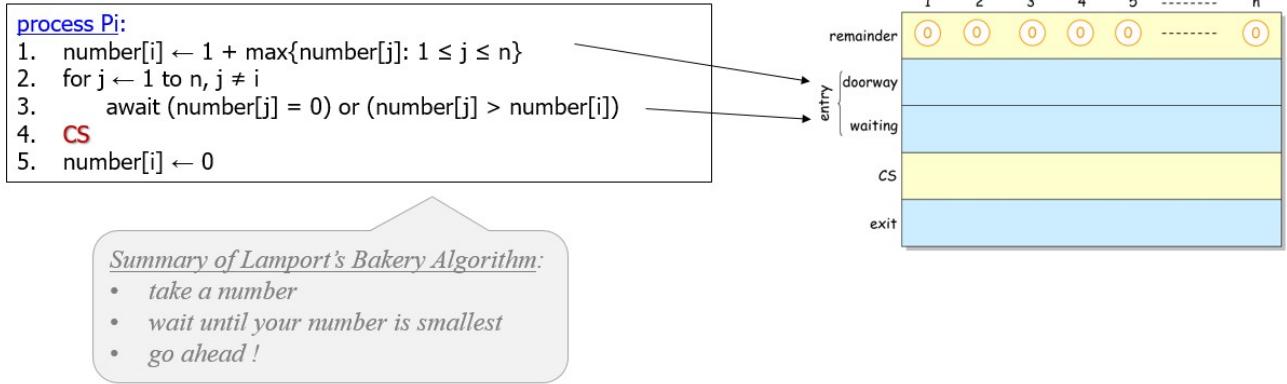
ג. שלב ה-"waiting": בשלב זה התהילר ממתין עד שmagiu התור שלו (על ידי בדיקה חוזרת ונשנית של מספר הcarteis שלו).

ד. CS - הקטע הקרייטי.

ה. שלב היציאה: לאחר יציאה מהקטע הקרייטי, התהילר מאפס את מספר הcarteis שלו, מעדכן שהוא סיום לרצף ואין לו כוונה להיכנס שוב לקטע הקרייטי.

כל תהליך יש מונה. אם הם לא מעוניינים להיכנס לאזור הקרייטי, אז הערך הוא 0. בכל שלב של האלגוריתם, הוא מתקדם במספר הגבואה ביותר ומוסף 1. לדוגמה, אם המספר gaboa ביותר היה 0, אז התהילר הבא יתקדם ל-1 ויקדם אותו ב-1. לאחר שהגיע התהילר נוסף שקיבל 4, הוא מוסיף 1 למונה, מקבל את הערך 2 וועובר במצב המתנה, מכיוון שיש תהליך אחר עם כרטיס שערכו נמוך יותר.

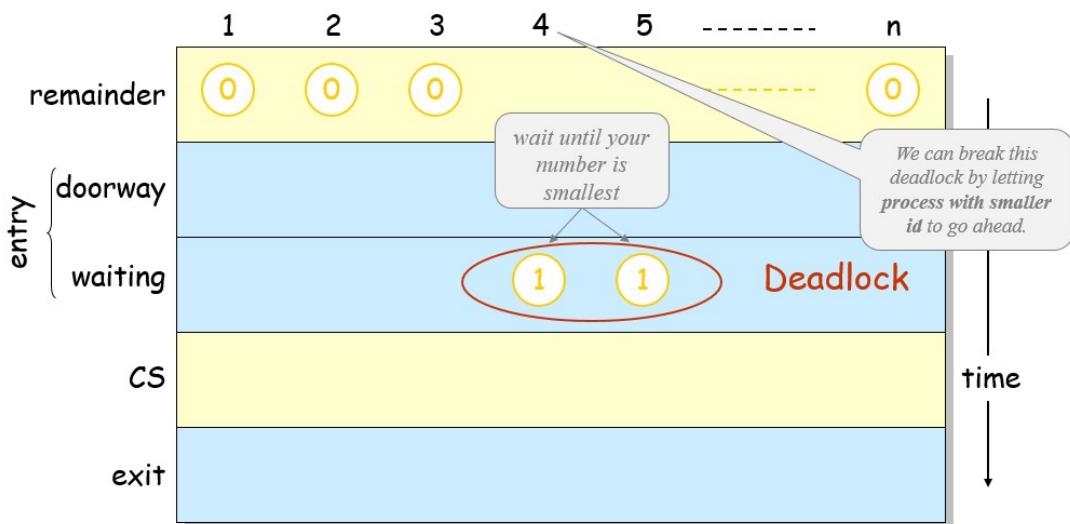
Implementation 1



Does this implementation work ?

NO. It can deadlock !

Implementation 1 – deadlock



מימוש ראשון

אלגוריתם זה מתרגם את השלבים של התהליך ה-2 באלגוריתם. נתחיל בקוד:

1. $\text{number}[i] \leftarrow 1 + \max(\text{number}[j] : 1 \leq j \leq n)$

בשלב זה, התהליך 2 בוחר את מספר הקרטיס שלו על ידי מציאת המספר הגדול ביותר מבין כל התהליכיים האחרים והוספת 1 לו.

2. for $j \leftarrow 1$ to n , $j \neq i$

בשלב זה, התהליך 2 עובר על כל התהליכיים האחרים, חוץ מעצמו.

3. await(number[j] = 0) or (number[j]>number[i])

בשלב זה, התחילה ממתין עד שמתקיים אחד משני התנאים הבאים:

א. מספר הכתובת של התהילה \neq 0, כלומר התהילה \neq עדין לא לzech כרטיס.

ב. מספר הcartis של התהילה j גדול ממספר הcartis של התהלה i. זה מאפשר הוגנות בין כל התהלים שיש להם אותו מספר Cartis - רק אחד מהם יוכל להיכנס.

4. CS

בשלב זה, התחילה נכנס לקטע הקוד הקרייטי.

5. number[i] <- 0

לאחר שהתחליר ז' יוצא מהקטע הקוד הクリיטי, הוא מעדכן את מספר הכרטיס שלו להיות 0, מה שמצוין שהוא סימן את הריצה שלו ואינו מעוניין להיכנס שוב לקטע הקוד הクリיטי.

במילימ אחרות, כאשר התהלהר רוצה להיכנס לקטע הקוד הクリיטי, הוא מחשש את המספר האגדל ביותר, מוסיף 1 לו, ומציב את התוצאה במשתנה שלו. לאחר מכן, הוא עובר על כל התהליםיהם קטנים ממנו וממתין להם - או שהם יתאפסו או שהמספר שלהם יהיה אגדל יותר ממנו.

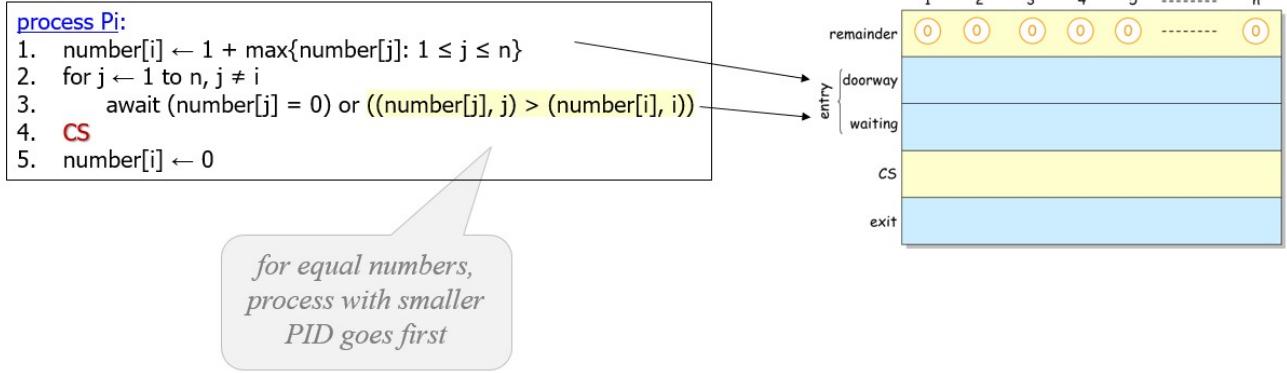
המימוש זהה יכול להוביל למצב של דילוק. למעשה זאת נקודה שכבר ראינו כבר: 2 תהליכי מגעים בו בזמןית ולקחים ביחס את אותו הערך.

לדוגמה, שני תהליכי מגעים בשלב 1 בו זמנית ומבצעים אותו. כתוצאה מכך, שני התהליכי מקבלים את המספר 1. בשלב 3, כל אחד מהתהליכי מגע לבדיקה האי-שוויון ונכשל בה. כתוצאה מכך, תהליך 1 מחליף 2 ולהפך.

לשכם כך, למפורט הציג מודל לפיו נ.labour לעבוד לפי יחס הסדר של ה-ID. לעומת התהילה ש-ID שלו קטן יותר, קיבל להיכנס קודם.

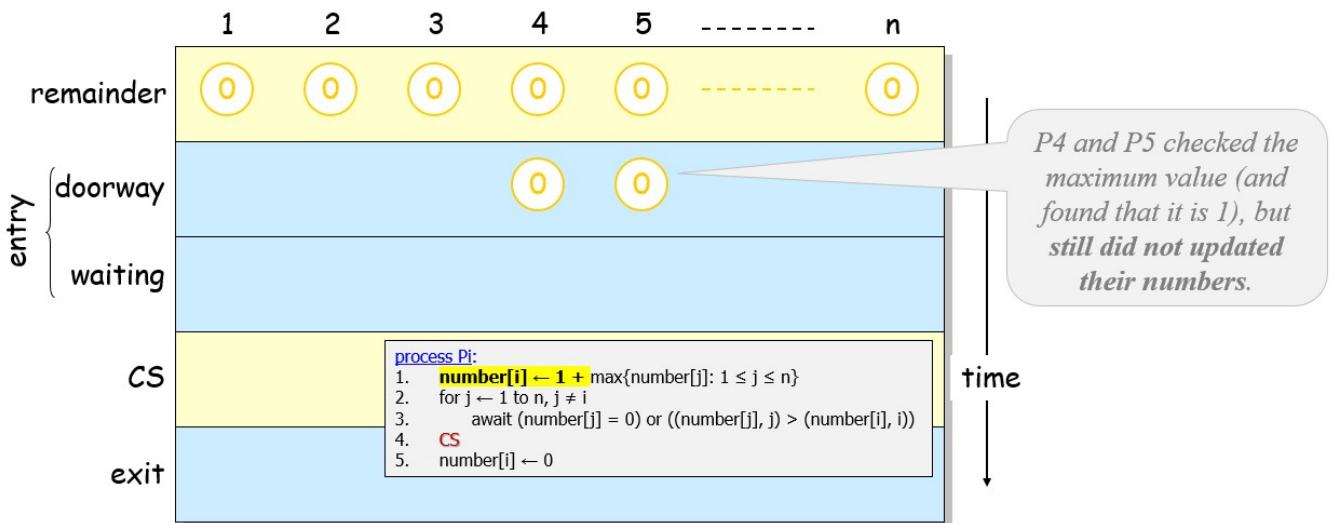
במימוש הנוכחי, יתכן גם שהייה תרחש בו תהיה הרעבה, אך בהמשך, באלאgorיתם המלא, נראה איך ניתן לפתרו את הבעיה.

Implementation 2

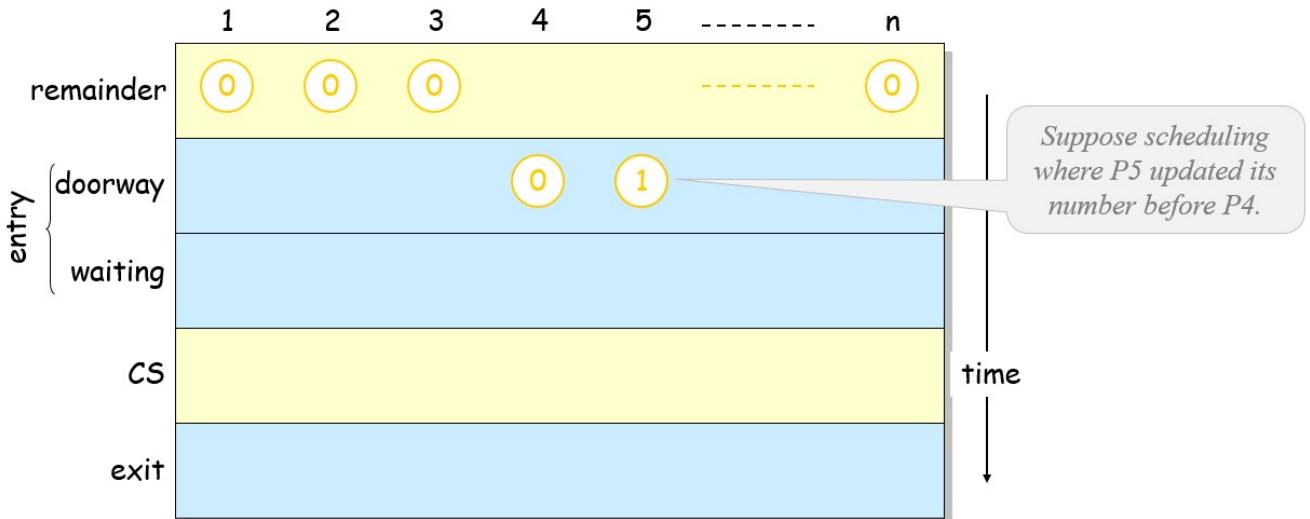


Does this implementation work?
NO. It does not satisfy mutual exclusion!

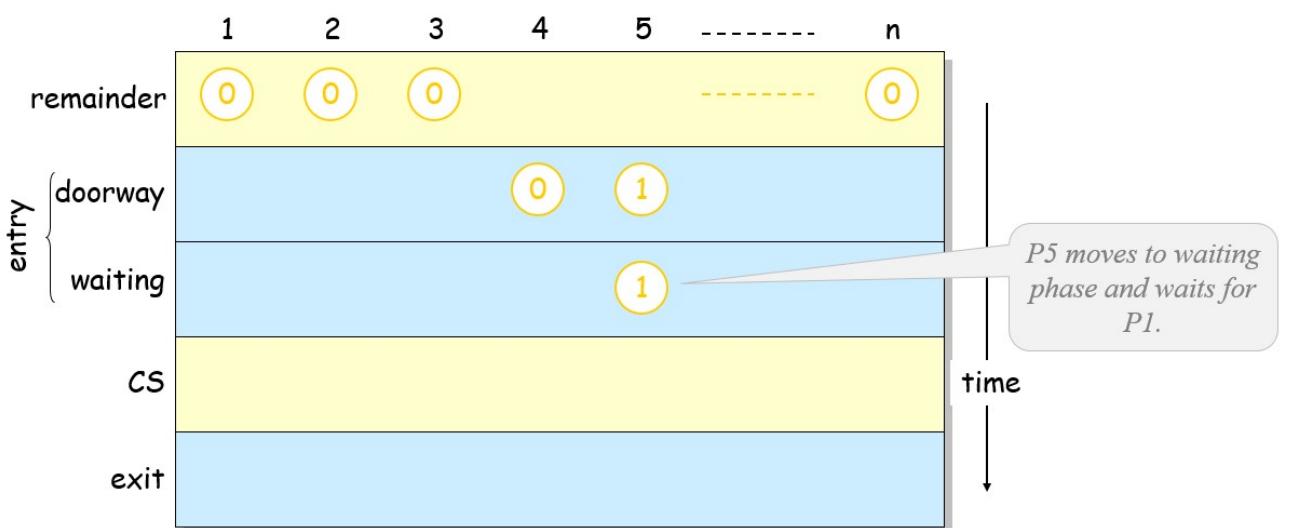
Implementation 2 – no mutual exclusion



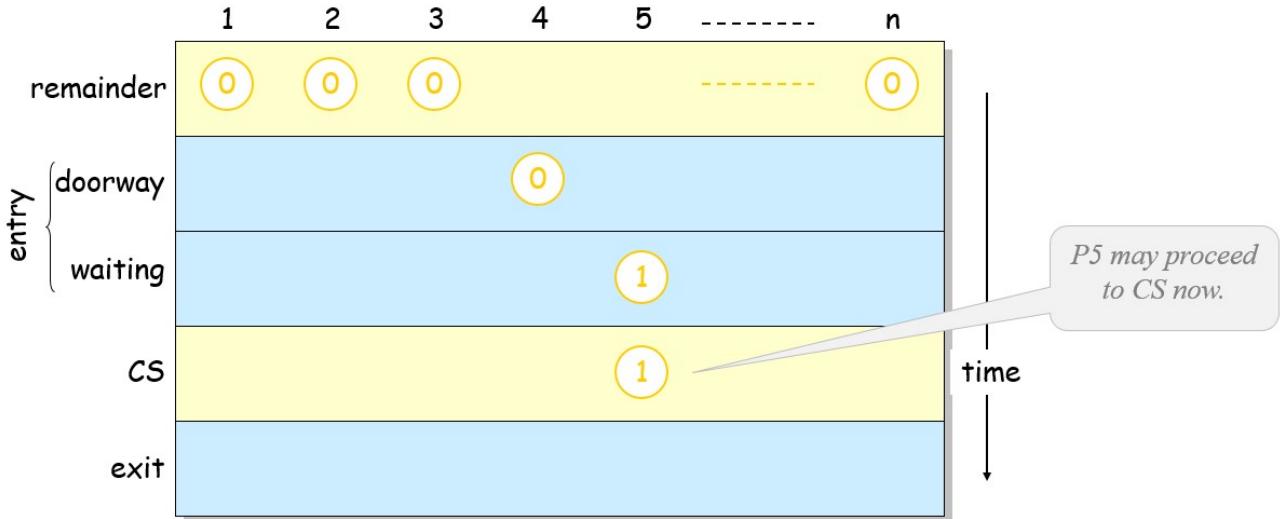
Implementation 2 – no mutual exclusion



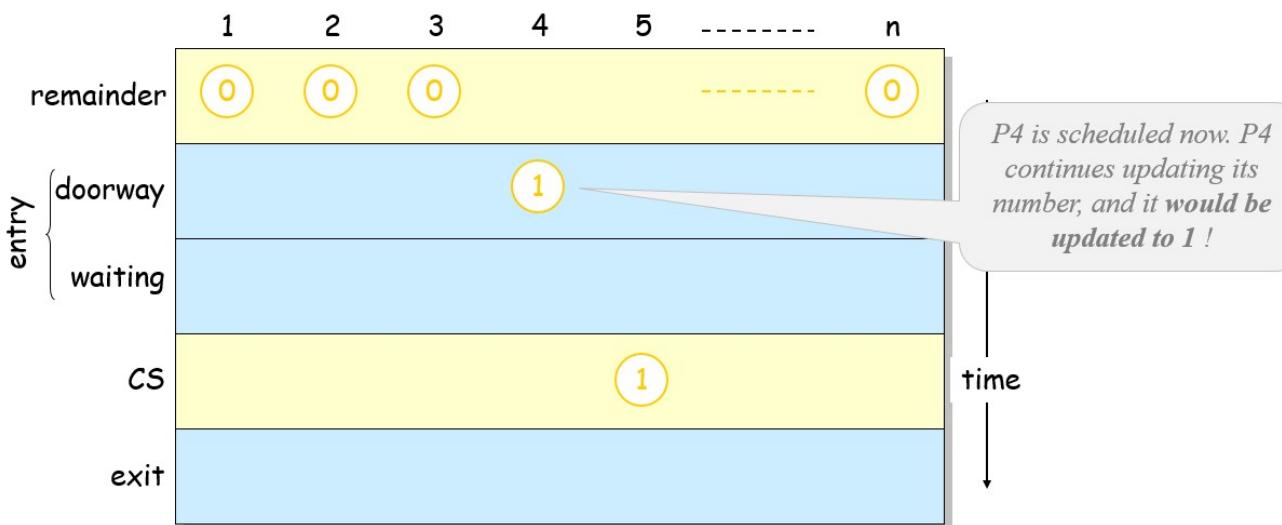
Implementation 2 – no mutual exclusion



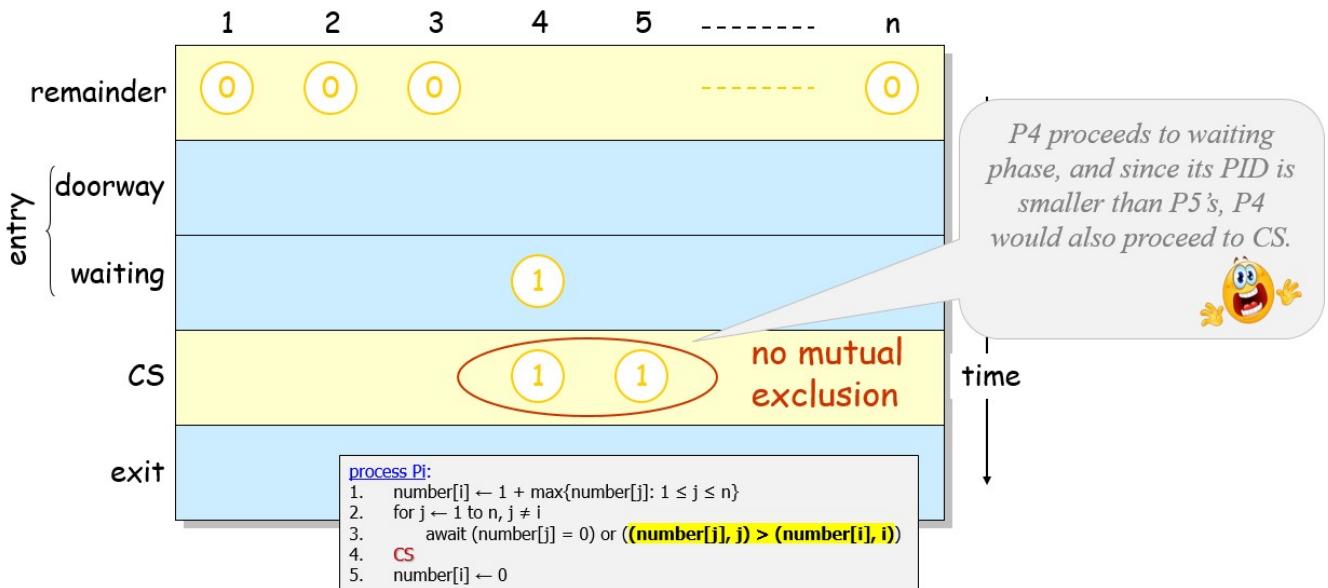
Implementation 2 – no mutual exclusion



Implementation 2 – no mutual exclusion



Implementation 2 – no mutual exclusion



אלגוריתם המאפיין של למפורט - מימוש שני

השינוי העיקרי בIMPLEMENTATION 2 הוא שטרם תחילת הפעלה, השוואת המספרים בarray נורא בזווית ימינה. אם השוואת המספרים נורא בזווית ימינה, אז תחילת הפעלה מוגדרת כרטיית.

3. `await(number[j] = 0) or ((number[j], j) > (number[i], i))`

בשלב זה, התהיליך יתבצע עד שאחד משני התנאים הבאים מתקיים:

- מספר crt[i] של התהיליך j הוא 0, כלומר התהיליך j עדין לא בחר מספר crt[i].
- הזוג (j, [j]num) גדול מהזוג (i, [i]num) לפי היחס הלקסיקוגרפי.

בנסיבות השינוי הזה, אם שני התהיליכים יש להם את אותו מספר crt[i], התהיליך עם ה-ID הקטן יותר יקבל עדיפות.

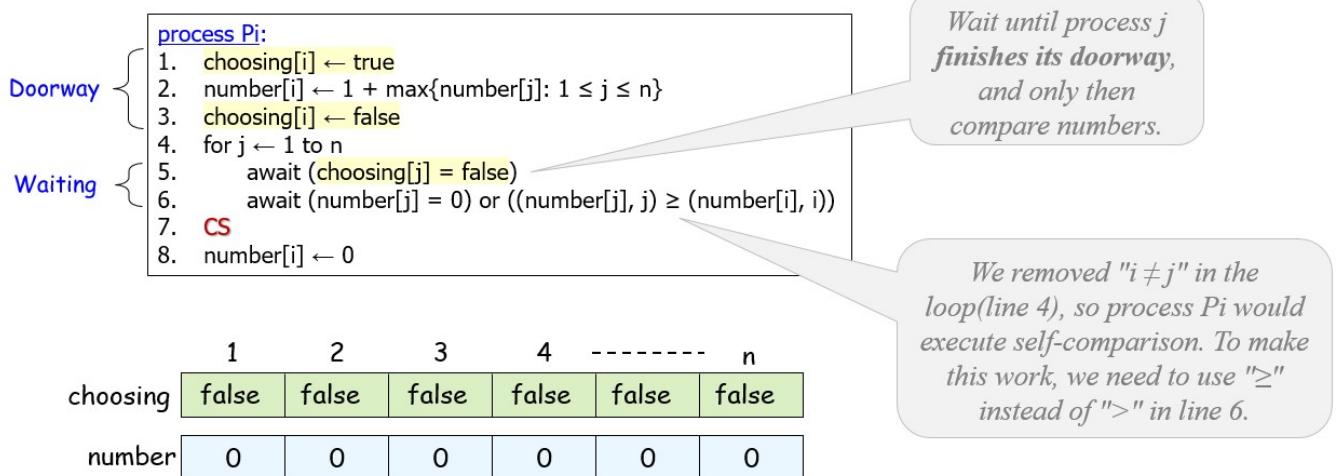
אך, מימוש מציג בעיה של הפרת המוניה ההדדיות. נסביר זאת באמצעות דוגמה:
נניח שני תהיליכים מעוניינים להיכנס לקטע קוד קרייטי. בשלב הראשון, שני התהיליכים מגיעים לבחירת crt[i]. לאחר ויהי crt[i] המקסימלי הוא 0.

נניח שמשמש לפנייהם מוסיפים 1 למספר crt[i], מערך הפעלה מחליט לבצע החלפת הקשרים. מערכת הפעלה יכולה לעצור את שני התהיליכים בכל שלב, כולל בשלב שימוש לפני ההשמה, לאחר וזה פעולה שהיא יכולה להחליט שהיא מורכבת.

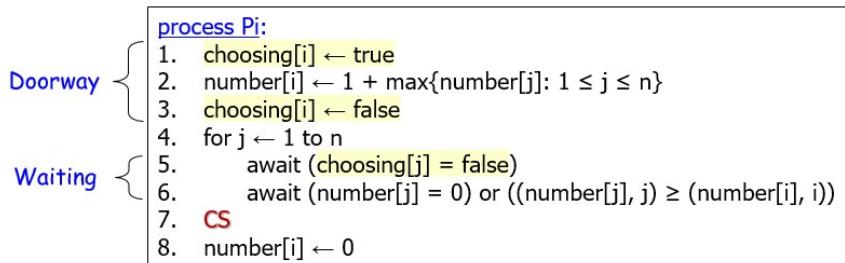
נניח שמערכת הפעלה מחליטת לתת זמן ריצה קודם ל-P5. הוא מעודכן את המספר שלו ל-1 ואז בשלב 3 הוא עובר על כל התהיליכים, רואה שהוא הcy קטן ולכן נכנס לקטע הקוד הקרייטי. בעצם, P4 מקבל זמן ריצה ומעודכן את המספר שלו ל-1. כתום, מבין כל התהיליכים שהמספר שלהם אינו 0, הוא לא הcy קטן, אלא שווה. לכן, הוא עובר לבדיקה לפי ה-ID, ומאתה-ID שלו קטן יותר, הוא נכנס. כתוצאה לכך, ישנו שני תהיליכים שנמצאים בקטע הקוד הקרייטי בו זמןנית.

פתרון אחד שהוצע הוא להוסיף אינדיקטור. ישנו גם פתרונות אחרים שנייתן לשקלול.
כאן מופיע עיקרון חשוב:
 קריאה וכתיבה: מערכת הפעלה יכולה לעזור תהליכיים לאחר קריאה לפני כתיבה. זה עיקרון שיכל להפיל הרבה אלגוריתמי סנכרוניזציה, ולא רק הם - זה יכול לבוא לידי ביטוי בכל קוד מקבילי!

The Bakery Algorithm



The Bakery Algorithm



Is it possible that two processes would get same number ?

YES. `choosing[]` does not prevent it.

The Bakery Algorithm

```

Doorway {  

process Pi:  

  1. choosing[i] ← true  

  2. number[i] ← 1 + max{number[j]: 1 ≤ j ≤ n}  

  3. choosing[i] ← false  

  4. for j ← 1 to n  

     5.   await (choosing[j] = false)  

     6.   await (number[j] = 0) or ((number[j], j) ≥ (number[i], i))  

  7.   CS  

  8.   number[i] ← 0
}

Waiting {  

}

```

Does this code hold Mutual exclusion property ?

YES. Suppose $P_i < P_j$. If P_i is smaller, then P_i enters CS. If they have same number, then P_i enters CS since $P_i < P_j$. When P_i starts comparing its (number, PID) to P_j , P_j already finished doorway and updated its number. Thus, one of the processes P_i or P_j would find its (number, PID) bigger, and would wait the other process to exit CS.

The Bakery Algorithm

```

Doorway {  

process Pi:  

  1. choosing[i] ← true  

  2. number[i] ← 1 + max{number[j]: 1 ≤ j ≤ n}  

  3. choosing[i] ← false  

  4. for j ← 1 to n  

     5.   await (choosing[j] = false)  

     6.   await (number[j] = 0) or ((number[j], j) ≥ (number[i], i))  

  7.   CS  

  8.   number[i] ← 0
}

Waiting {  

}

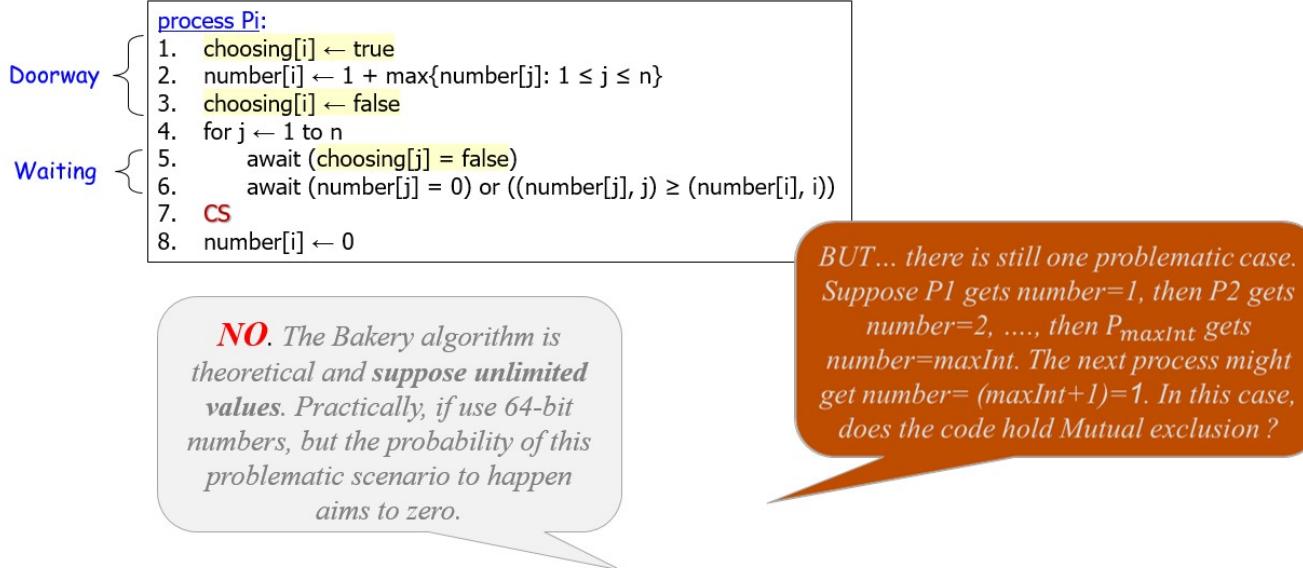
```

Does this code hold Mutual exclusion property ?

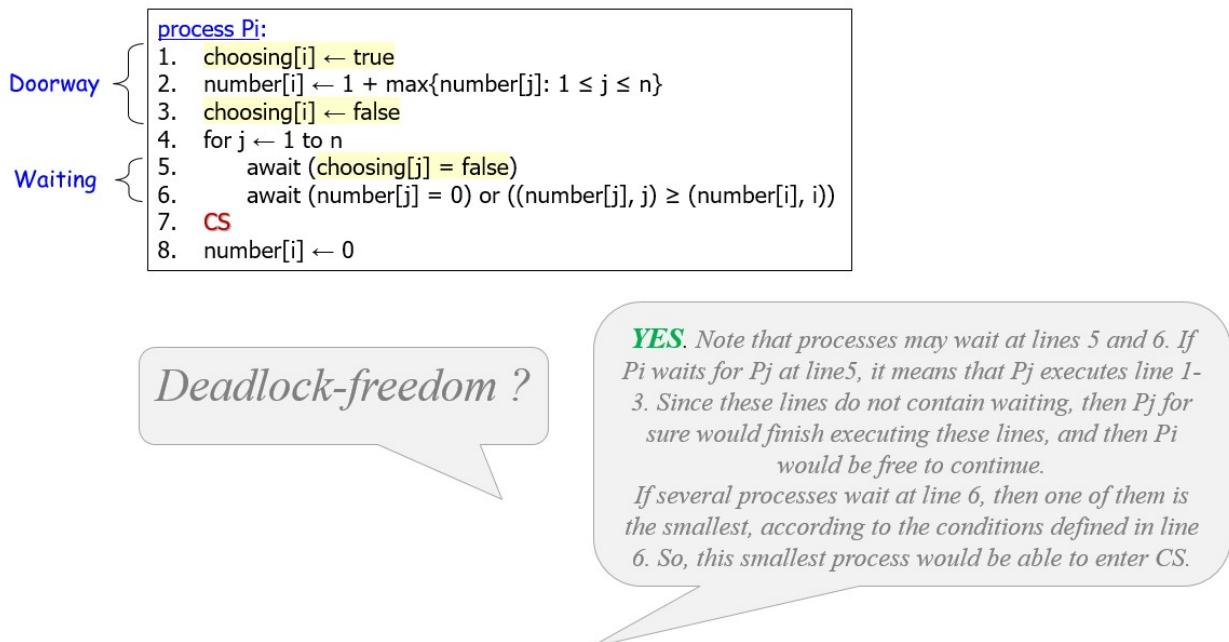
BUT... there is still one problematic case. Suppose P_1 gets number=1, then P_2 gets number=2, ..., then P_{\maxInt} gets number= \maxInt . The next process might get number= $(\maxInt+1)=1$. In this case, does the code hold Mutual exclusion ?

YES. Suppose P_i gets number=1, then P_{i+1} enters CS since $P_{i+1} < P_i$. When P_{i+1} starts comparing its (number, PID) to P_i , P_i already finished doorway and updated its number. Thus, one of the processes P_i or P_{i+1} would find its (number, PID) bigger, and would wait the other process to exit CS.

The Bakery Algorithm



The Bakery Algorithm



The Bakery Algorithm

```

Doorway { process Pi:
1. choosing[i] ← true
2. number[i] ← 1 + max{number[j]: 1 ≤ j ≤ n}
3. choosing[i] ← false
4. for j ← 1 to n
5.     await (choosing[j] = false)
6.     await (number[j] = 0) or ((number[j], j) ≥ (number[i], i))
7.     CS
8.     number[i] ← 0
}

Waiting { 
```

FIFO ?

YES. If P_i is waiting (at lines 5 or 6), then P_i already updated its number. If P_j is starting the doorway just now, P_j 's number would be bigger than P_i 's number, according to the algorithm. So, P_i would enter CS first.

Deadlock-freedom + FIFO = Starvation-freedom 😊

The Bakery Algorithm

```

Doorway { process Pi:
1. choosing[i] ← true
2. number[i] ← 1 + max{number[j]: 1 ≤ j ≤ n}
3. choosing[i] ← false
4. for j ← 1 to n
5.     await (choosing[j] = false)
6.     await (number[j] = 0) or ((number[j], j) ≥ (number[i], i))
7.     CS
8.     number[i] ← 0
}

Waiting { 
```

Suppose P_i checks $\text{choosing}[j]$ and finds it false. Right after this P_j starts to choose its number. Then P_i starts executing line 6.

Does this scenario lead to problem ?

NO. In this situation, P_j would choose bigger number than of P_i . Two possibilities – P_i checks P_j 's number before P_j finishes to choose it. Then P_i sees $\text{number}[j]=0$ and ignores P_j candidature. Otherwise, P_i checks P_j 's number after P_j finished to check it. In this case P_i would see that P_j 's number is bigger and would ignore P_j 's candidature as well.

האלגוריתם המלא של למפורט

האלגוריתם המלא מכיל מערך של משתנים בוליאניים בשם `choosing`. כאשר הערך במקום 1 הואאמת, זה אומר שהתחליר או בחירת מספר crtis שלו. השינויים באלגוריתם הם כדלקמן:

```

process Pi:
1. choosing[i] <- true 
```

התחליר או מצין שהוא רצה לבחור מספר crtis ולהיכנס לקטע הקוד הקרייטי.

```
2. number[i] <- 1+max(number[j]: 1<=j<=n)
```

התהיליך בוחר את מספר הcrcטיס.

3. choosing[i] <- false

התהיליך מצין שהוא סיים לבחור את מספר הcrcטיס.

4. for j <-1 to n

עבור כל התהיליך j,

5. await(choosing[j] = false)

התהיליך ממתין עד שהטהיליך j יסיים לבחור crcטיס אם הוא מעוניין להיכנס לقطع הקוד הקרייטי.

6. await(number[j] = >(j, [j]number) or (0 = [i]number))

שים לב שכאן אנחנו בודקים אם הזוג גדול או שווה, לאחר ואנחנו עוברים בלולאה על כל התהיליכים, כולל התהיליך j.

7. CS

התהיליך מכניס לقطع הקוד הקרייטי.

8. number[i] <- 0

לאחר שהטהיליך יצא מקטע הקוד הקרייטי, הוא מעדכן את מספר crcטיס שלו להיות 0.

תיאורטית, האלגוריתם מקיים מניעה הדדיות:

נניח ש- $P_i < P_j$. אם מספר P_i קטן יותר, אז P_i ייכנס לقطع הקוד הקרייטי. אם להם יש את אותו המספר, אז P_i ייכנס לقطع הקוד הקרייטי לאחר ש- $P_j < P_i$. כאשר P_i מתחילה להשוו את הזוג (מספר, PID) שלו ל- P_j , P_j כבר סיים את ה-doorway ועדכן את המספר שלו. כך שאחד מהטהיליכים P_i או P_j ימצא שהזוג (מספר, PID) שלו גדול יותר, וימתין לתהיליך האخر לצאת מקטע הקוד הקרייטי.

פרקטיית, בהסתברות נמוכה האלגוריתם לא מקיים מניעת הדדיות:

יש עדין מקרה בעייתי אחד. נניח $P_1 < P_2$ מקבל מספר=1, אז P_2 מקבל מספר=2, אז P_{maxint} מקבל מספר= $maxInt$. התהיליך הבא עשוי לקבל מספר=1. במקרה זה, האם הקוד מקיים אגנה הדדיות? לא. האלגוריתם זה הוא תיאורטי ומחייב ערכים בלתי מוגבלים. מבחינה מעשית, אם משתמשים במספרים בין 64 סיביות, אז ההסתברות שהמקרה הבועייתי הזה יתרחש שואף לאפס.

האלגוריתם זהו **חופשי מדלקים**. אם יש תהיליך שמתinan לתהיליך j ב-choosing, אז אנחנו יודעים שבשלב כלשהו הוא יתעדכן כי j ממתין לקבל זמן ריצה כדי לסייע את העדכו ולא לשום דבר אחר. במקרה השני - ראיינו שהוא עובד. אם שני הטהיליכים ממתינים או אחד מהם יהיה בעל id קטן יותר, אז במקרה האגרען תמיד יהיה משווה לשיננס לقطع הקוד הקרייטי.

אם האלגוריתם מקיים FIFO? התשובה היא כן. אם יש תהיליך אחד שכבר נמצא בלולאה ויש תהיליך אחר שעוד לא התחיל את doorway, אז הם לא יקבלו את אותו המספר, והשני יקבל מספר בעל ערך גבוה יותר. מכאן שהאלגוריתם חופשי מהרעות, כי תמיד יהיו לכל היוטר 1-ה איטרציות בלולאה.

It would be much **easier** to implement mutual exclusion if we use a single **atomic Read-And-Write instruction**

נראה סוגים שונים של כלים.

נתחיל מהכל שיש לו תמייה בחומרה - הוראה אוטומטית.

ראינו כבר S בקורס SPL, נראה CUT עד אחד.

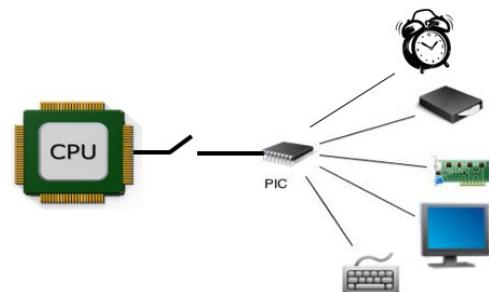
Test-And-Set atomic instruction

Test-And-Set (m)

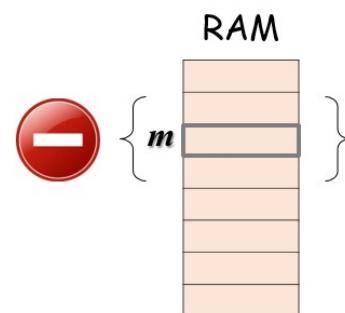
do atomically

- $prev \leftarrow m$
- $m \leftarrow 1$
- return $prev$

Test-And-Set is implemented by calling TSL (Test-and-Set Lock) Assembly atomic instruction.



This atomicity is supplied by hardware, which disables interrupts on the appropriate CPU and disables access to m (by locking data bus), till Test-And-Set is fully executed.



ההוראה TSL

למימוש ההצעה הבאה, נדרש תמייה מהחומרה. ישנו מעבדים, במיוחד אלו בהם מרובי ליביות, שבהם קיימת הוראה מיוחדת שמתוארת בדרך כלל כך:

TSL RX, LOCK

ההוראה מקריאה את התוכן של הכתובת שנמצאת ב-LOCK לטור הרגיסטר AX, ואז שומרת ערך שונה מ-0 בזיכרון בכתובת של LOCK. מובטח שהפעולה הזאת תבוצע באופן בלתי נתנה להפרעה, כמובן, אף תהילך אחר לא יוכל לגשת ל זיכרון באותו מקום של lock.

אפשר לתאר את ההוראה באמצעות הפסאודוקוד הבא:
תהייה משתנה (lock) w. בצע באופן אוטומי (כלומר, תוך ביטול פסיקות וכל גישה ל זיכרון של w): עדכן את השדה prev עם הערך של w, לאחר מכן עדכן את w להיות 1, ולבסוף החזר את prev.

זה מאפשר למשתמש לדגום את המשתנה שהוחזק.
כדי לבצע את הפעולות באופן אוטומי, המעבד חייב לבטל פסיקות כדי למנוע הפרעות, וגם לנעול חומרתית את החלק הזה ברם - נעה זה יכולה להיות חלקית או מלאה.

שני התנאים האלה הם תנאים הכרחיים ומשמעותיים לביצוע ההוראה באופן אוטומי.
שאלה:

מהו נדרש שני התנאים לאוטומיות?
תשובה:

כפי שראינו, כיבוי פסיקות במעבד אחד לא משפיע על מעבד שני, ולכן כדי להבטיח מניעה הדדית יש לדרש שגם האפיק של הנתונים (the bus) יהיה נעול.

Mutex by using Test-And-Set

Test-And-Set (m)

do atomically
 • $prev \leftarrow m$
 • $m \leftarrow 1$
 • *return prev*

initially: $m \leftarrow 0$

Pi:

1. await Test-And-Set(m)=0
2. CS
3. $m \leftarrow 0$

Mutual exclusion? Yes

Deadlock-freedom?

Starvation-freedom?

Suppose several processes try to enter CS. Since Test-And-Set is atomic, only one of them succeeded to get 0 as previous value of m. Thus, all other processes must wait.

הכלי הפשטוני לזמן תהליכיים

הכלי שנדון בו הוא הכלי הפשטוני לזמן תהליכיים. התהילך הנוכחי ימתין עד שיקבל 0, ואז יכנס לאוזור הקרייטי. לאחר מכן, התהילך יעדכן את המשתנה w להיות 0. הכלי הזה פשוט ויעיל להפליא, והוא הכלי הטוב ביותר שיש לנו כרגע.

החסרון היחיד של הכלי הזה הוא שהוא מבצע "busy wait" - ממתין בלאלה עד שהערך של המשתנה הופך להוית 0. אף על פי זאת, הוא עדין כל מוציא בזכותו הפשטות והיעילות שלו. נראה בהמשך מתי הכלי הזה אינו יעיל או לא מתאים.

כל לראות שהכל מקיימים את תנאי ה"mutual exclusion": נניח שיש מספר תהליכיים שרצוים להיכנס לאזור הקרייטי - רק תהליך אחד יכול להתקדם, מכיוון שהפעולה "test and set" היא אוטומית - רק תהליך אחד יכול לבצע את הפעולה בכל פעם.

הכלי גם חופשי מDDLוקים - בכל מצב, תמיד יהיה תהליך שיוכל לבצע את "Test and Set" ולהתקדם.

אך הכל*י* אינו חופשי מהרעות! האלגוריתם מאד תלוי בסדר הגעה של התהליכיים! יכול להיות מצב שתהליך מסוים נכנס ויוצא מהאזור הקרייטי, משנה את המשטנה, ואז מערכת הפעלה מאפשרת לו לבצע שוב ושוב את "Test-And-Set", ובכך כל תהליך אחר ישרר רעב.

הכלי מתאים אם אנחנו רוצים התקדמות גלובלית של התוכנית.

המימוש הנוסף באסמבלי:

```
enter_region:  
    TSL REGISTER,LOCK      ;copy lock to register and set lock to 1  
    CMP REGISTER,#0          ;was lock zero?  
    JNE enter_region         ;if it was not zero, lock was set, so loop  
    RET                      ;return to caller  
  
; critical region entered  
  
leave_region:  
    MOVE LOCK,#0             ;store a 0 in lock  
    RET                      ;return to caller
```

Mutex by using Test-And-Set

Test-And-Set (m)

do atomically

- $prev \leftarrow m$
- $m \leftarrow 1$
- return $prev$

initially: $m \leftarrow 0$

Pi:

1. await Test-And-Set(m)=0
2. CS
3. $m \leftarrow 0$

Mutual exclusion? Yes

Deadlock-freedom? Yes

Starvation-freedom?

Suppose several processes want to enter CS. If $m=0$, then the first of them would enter CS. If $m=1$, the processes should wait, since CS is occupied right now. Since m changes to be 0 at exits, this would let next entrance to CS.

Mutex by using Test-And-Set

Test-And-Set (m)

do atomically

- $prev \leftarrow m$
- $m \leftarrow 1$
- return $prev$

initially: $m \leftarrow 0$

Pi:

1. await Test-And-Set(m)=0
2. CS
3. $m \leftarrow 0$

Mutual exclusion? Yes

Deadlock-freedom? Yes

Starvation-freedom? No

initially: $m:=0$

while (true)

P1 TAS ($m:=1$ and returns 0)

P1 enters CS

P2 TAS (returns 1)

P1 exits CS

Starvation-free mutex by using Test-And-Set

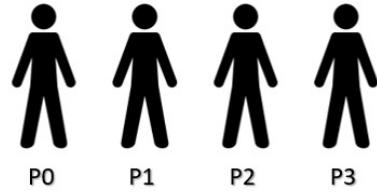
initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} (\text{Test-And-Set}(m) = 0 \text{ or } \text{interested}[i] = \text{false})$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& !interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j=i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

$m : 0$

interested false false false false



P0 P1 P2 P3

Starvation-free mutex by using Test-And-Set

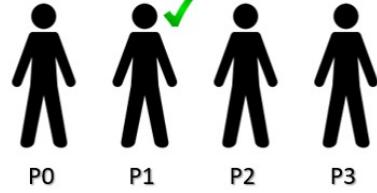
initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. **interested**[i] $\leftarrow \text{true}$
2. $\text{await} (\text{Test-And-Set}(m) = 0 \text{ or } \text{interested}[i] = \text{false})$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& !interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j=i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

$m : 1$

interested false true false false



P0 P1 P2 P3

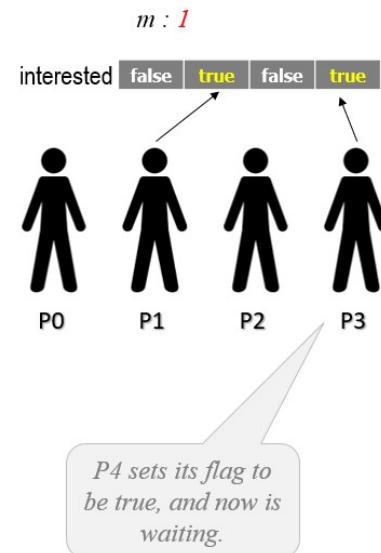
If P_i wants to enter CS, it sets $\text{interested}[i]$ to be true. If then $\text{interested}[i] = \text{false}$, it means that some other process changed it to be false.

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ && } !\text{interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j==i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

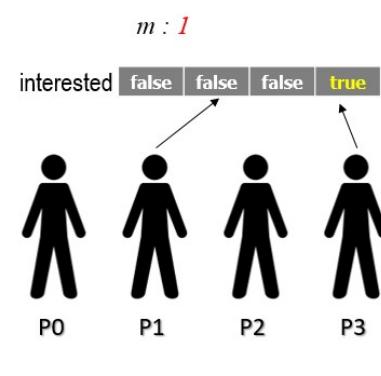


Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ && } !\text{interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j==i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$



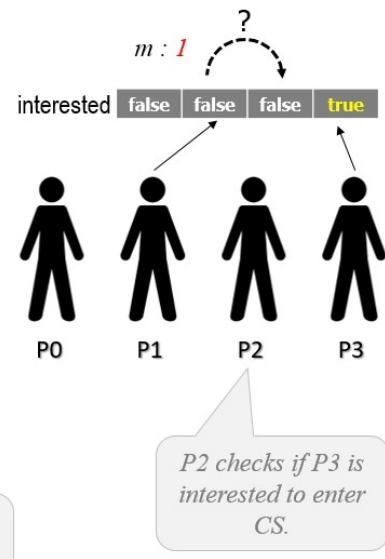
Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} (\text{Test-And-Set}(m) = 0 \text{ or } \text{interested}[i] = \text{false})$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& !interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j==i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

Starting from the next PID (i.e., $i+1$), P_i checks if there is some process P_j that wants to enter CS.



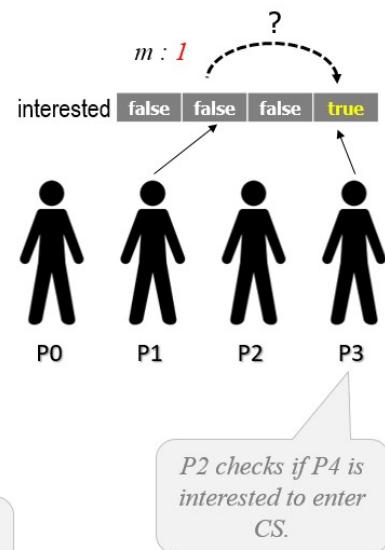
Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} (\text{Test-And-Set}(m) = 0 \text{ or } \text{interested}[i] = \text{false})$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& !interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j==i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

Starting from the next PID (i.e., $i+1$), P_i checks if there is some process P_j that wants to enter CS.



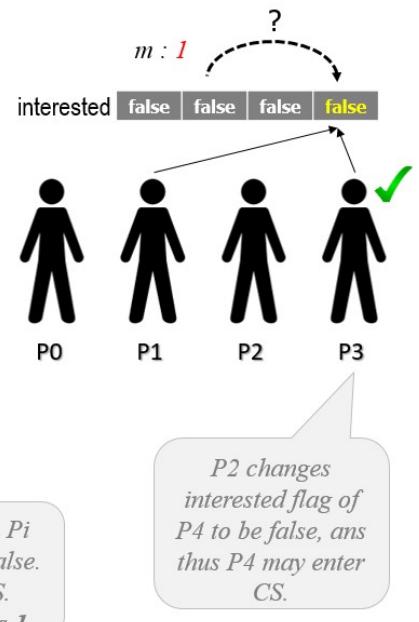
Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} (\text{Test-And-Set}(m) = 0 \text{ or } \text{interested}[i] = \text{false})$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& !interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j=i)$
8. $m \leftarrow 0$
9. else
10. **$\text{interested}[j] \leftarrow \text{false}$**

If such process P_j exists, P_i turns $\text{interested}[j]$ to be false.
This lets P_j to enter CS.
Note: in this case m stays 1.



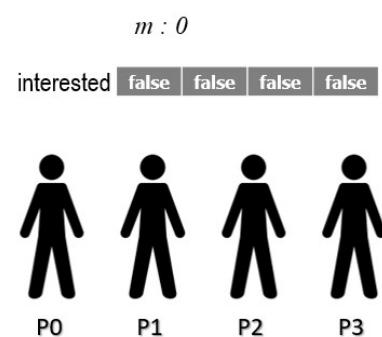
Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} (\text{Test-And-Set}(m) = 0 \text{ or } \text{interested}[i] = \text{false})$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& !interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j=i)$
8. **$m \leftarrow 0$**
9. else
10. **$\text{interested}[j] \leftarrow \text{false}$**

If nobody is interested to enter CS, P_i frees the mutex.



Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& } \text{!interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j == i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

Suppose several processes try to enter CS. Since Test-And-Set is atomic, **only one of them succeeded to get 0 as previous value of m . Thus, all other processes must wait.**

Mutual exclusion? Yes

Deadlock-freedom?

Starvation-freedom?

Suppose P_i is in CS. If P_j wants to enter CS, P_j would wait till P_i sets $\text{interested}[j]$ to be false. It happens only when P_i exits CS. Note: m in this case stays 1, so no other process may enter CS together with P_j .

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& } \text{!interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j == i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

Mutual exclusion? Yes

Deadlock-freedom? Yes

Starvation-freedom?

If P_i and P_j want to enter CS, one of them would succeed if $m=0$. Otherwise, suppose some other process is in CS. When this process exits, it would find both P_i and P_j $\text{interested} = \text{true}$. In this case one of them would be set to true, and this would let one of them to enter CS.

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& } \text{!interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j == i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

Mutual exclusion? Yes

Deadlock-freedom? Yes

Starvation-freedom? Yes

Suppose P_i is in CS, and P_j is waiting to enter CS. Suppose $P_i < P_j$. Two scenarios are possible – or P_i unlocks P_j , and let it in, or P_i unlocks some other process P_k , and let it in. In the second scenario exists: $P_i < P_k < P_j$ (otherwise, P_j would be unlocked before P_k). Thus, P_j gets closer to enter CS. More precisely, P_j should wait at most $|P_j - P_i|$ processes to enter CS.

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& } \text{!interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j == i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

FIFO? No



lock

False	False	False
-------	-------	-------

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& } \text{!interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j == i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

FIFO ? **NO**



lock

True	False	False
------	-------	-------

- p_0 performs lines 1-2, enters CS

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& } \text{!interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j == i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

FIFO ? **NO**



lock

True	False	True
------	-------	------

Spin pic means
“busy wait”

- p_0 performs lines 1-2, enters CS

- p_2 performs line 1 and starts waiting in line 2

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& } \text{!interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j==i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

FIFO ? **NO**



lock

p_0

p_1

p_2

True

True

True

- p_0 performs lines 1-2, enters CS
- p_2 performs line 1 and starts waiting in line 2
- p_1 performs line 1 and starts waiting in line 2

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i

1. $\text{interested}[i] \leftarrow \text{true}$
2. $\text{await} ((\text{Test-And-Set}(m) = 0) \text{ or } (\text{interested}[i] = \text{false}))$
3. **CS**
4. $\text{interested}[i] \leftarrow \text{false}$
5. $j \leftarrow (i+1) \% n$
6. $\text{while } (j \neq i \text{ \&& } \text{!interested}[j]) \quad j \leftarrow ++j \% n$
7. $\text{if } (j==i)$
8. $m \leftarrow 0$
9. else
10. $\text{interested}[j] \leftarrow \text{false}$

FIFO ? **NO**



lock

p_0

p_1

p_2

False

False

True

- p_0 performs lines 1-2, enters CS
- p_2 performs line 1 and starts waiting in line 2
- p_1 performs line 1 and starts waiting in line 2
- p_0 exits and sets $\text{interested}[1]$ in line 10 to false

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i	
1.	$\text{interested}[i] \leftarrow \text{true}$
2.	await ((Test-And-Set(m) = 0) or ($\text{interested}[i] = \text{false}$))
3.	CS
4.	$\text{interested}[i] \leftarrow \text{false}$
5.	$j \leftarrow (i+1) \% n$
6.	while ($j \neq i \&& \text{!interested}[j]$) $j \leftarrow ++j \% n$
7.	if ($j == i$)
8.	$m \leftarrow 0$
9.	else
10.	$\text{interested}[j] \leftarrow \text{false}$

FIFO ? **NO**



lock



p₁

p₂

False False True

- p_0 performs lines 1-2, enters CS
- p_2 performs line 1 and starts waiting in line 2
- p_1 performs line 1 and starts waiting in line 2
- p_0 exits and sets $\text{interested}[1]$ in line 10 to false
- p_1 enters CS before p_2 , violates FIFO

Starvation-free mutex by using Test-And-Set

initially: $m \leftarrow 0$, $\text{interested}[i] \leftarrow \text{false}$ for all processes

Program for process i	
1.	$\text{interested}[i] \leftarrow \text{true}$
2.	await ((Test-And-Set(m) = 0) or ($\text{interested}[i] = \text{false}$))
3.	CS
4.	$\text{interested}[i] \leftarrow \text{false}$
5.	$j \leftarrow (i+1) \% n$
6.	while ($j \neq i \&& \text{!interested}[j]$) $j \leftarrow ++j \% n$
7.	if ($j == i$)
8.	$m \leftarrow 0$
9.	else
10.	$\text{interested}[j] \leftarrow \text{false}$

If we remove line 4,
would the algorithm
be correct ?

NO.

P1	P2
$\text{interested}[1] \leftarrow \text{true}$	
Test-And-Set(m) returns 0	
enter CS	
	$\text{interested}[2] \leftarrow \text{true}$
	Test-And-Set(m) returns 1
	await...
exit CS	
set $j \leftarrow 2$	
P2 is interested ? Yes.	
set $\text{interested}[2] \leftarrow \text{false}$ (note that m is still 1 !!)	
	enter CS
	exit CS
	Deadlock...

Even if P1 wants to re-enter CS, it could not since nobody can set its $\text{interested}[1]$ to be false, and m is 1.

וראייה אחרת בה יש טיפול במקרה של הרעה בה משלמים בסיבוכיות זמן ריצה. ניתן להסתכל על אלגוריתם זה כעל דוגמה לאלגוריתם מניעה הדדית של דיקסטרה:

אתחול:

המשתנה m , אשר נמצא בזיכרון משותף, מאותחל ל-0. לכל תהליך P_i , אתחל את התא ה- i במערך $\text{interested}[i]$ להיות שקר. המערך מבטא את הרצון של כל תהליך להיכנס לקטע קוד קרייטי.

ריצת האלגוריתם עבורי התהליך ה- i :

1. $\text{interested}[i] \leftarrow \text{true}$

התהליך מעדכן כי ברצונו להיכנס לקטע קוד קרייטי!

```
2. await(Test-And-Set(m) = 0) or (interested[i] = false)
```

התהיליך ה-*m* ממתין עד שהוא מצליח לשים את הערך במשתנה *m* בעזרת ההוראה האוטומטית או שתהיליך אחר הביע רצון להיכנס לקטע הקוד והוא יהיה זריז יותר. במקרה השני *interested* יהיה שקר כתוצאה משורה 10.

3. CS

כניסה לקטע הקוד הקרייטי

```
4. interested[i] <- false
```

לאחר יציאה מקטע הקוד הקרייטי, התהיליך ה-*m* מעדכן כי אין ברצונו להיכנס לקטע הקרייטי.

```
5. j <- (i+1)%n
```

הגדיר את *j* להיות האינדקס של התהיליך הבא בצורה מעגלית.

```
6. while(j!=i && !interested[j]) j++
```

רוץ בולולאה עד שכל התהיליכים נבדקו או שיש תהיליך שמעוניין להיכנס.

```
7. if(j==i)
```

במקרה שבו כל התהיליכים נבדקו, אתחל מחדש את הערך של *m* להיות 0 כדי שתהיליכים אחרים יוכלו להיכנס.

```
8. m<-0
```

```
9. else
```

```
10. interested[j] <- false
```

אחרת יש תהיליך שמעוניין להיכנס ולכן השדה *interested* שלו יעדכן להיות שקר (ובכך הוא יוכל לצאת משורה 2).

באלגוריתם זה, מתקיימת מניעה הדדית, האלגוריתם אינו מאפשר דಡוקים, ומונענו את הרעבות התהיליכים מכיוון שאם שדה *interested* של תהיליך הוא true, הוא יփוך להיות false בסופו של דבר כאשר אנחנו מזווים בצורה מעגלית. לABI FIFO: לא תוכנן אך גם מתקיים - בשל הבדיקה המוגבלת.

דוגמת ריצה לכך שהאלגוריתם לא מקיים FIFO:

התחל עם המשתנה *m* ומשני *interested* בגודל שווה למספר התהיליכים. נניח שהטהיליך 1 רוצה להיכנס: הוא מעדכן *l*-true. אם הוא מצליח לבצע את הפעולה האוטומטית, מעולה, הוא ממשיר לפעולה הבאה. התהיליך 3 גם רוצה להיכנס - הוא מעדכן את שדה *interested* ומתחיל להמתין בשורה 2. התהיליך 1 מבצע שורות 2-1, מכניס את האזור הקרייטי, יצא, ומגדיר את *interested[1]* לשורה 10 ל-false. התהיליך 1 מכניס את האזור הקרייטי לפני התהיליך 2, מפרק את FIFO.

שאלה בסגנון מבחנים:

נניח שאנו מתלבשים על אלגוריתם מסוים ומשנים אותו מעט, מה קורה? האם זה נכון? האם מאפיינים מסוימים מתקאים או לא? כמו שחקנו את שורה 4, מה יקרה? בסבירות מאד אבואה, זה יוביל לשזה לא טוב. הבדיקה אינה שם סתם. שורה זו מתרחשת לאחר האזור הבעייתי. נניח שנכנסנו להמתנה בשל TAS, ונניח שלאחר שיצאנו לא עדכנו ל-false. חיפשנו מי מעוניין ונניח שמצאנו מישחו מעוניין, שינו ל-1=m=false ו-JP מכניס. הוא גם יוצא

ומבחן שאם IC מעוניין - אך הוא בכלל לא מעוניין. במקום זה, זה PK שאם רצה להיכנס, לא יכול לבצע TAS והתנאי השני לא מתקיים, ולכן הוא נתקע - יש דילוק כאן!



סיכום ביןיהם:

אחד האתגרים החשובים בסנכרון הוא בעיית מניעה הדדית, שהוצאה לראשונה על ידי דיקסטרה בשנת 1965. מנעה הדדית היא העיקרי שלפיו שני תהליכי יכולם להיות בחלק הקרייטי, חלק מהקוד שבו משאבים משותפים מונגים באותו הזמן. המונח זהה חינוי כדי למנוע מצבים של תחרות, שבהם התנהלות המערכת עשויה להשנותה בשל הזמן השונים של התהליכים.

ופש מדולקים, וחופש מהרעה הם שני מאפיינים נוספים חיוניים שפרוטוקולי הסנכרון צריכים לשמור עליהם. חופש מדולוק מבטיח שאם ישנים תהליכי מרובים שמנסים להיכנס לחלק הקרייטי, לפחות אחד מהם יצליח בסופו של דבר. חופש מהרעה, מצד שני, מבטיחה שאם תהליך מנסה להיכנס לחלקו הקרייטי, עליו להיאפשר לעשות זאת בסופו של דבר.

שיטות שונות הוצאו כדי להשיג המנגנון הדדית, חופש מדולוקים, וחופש מהרעה. שיטה מוכרת במיעוד היא אלגוריתם ה-2-תהליכי של פיטרסון. אלגוריתם של פיטרסון משלב את העקרונות של מערכת דגלים וחילופים כדי לספק הדדי הבנה לשני תהליכי. אף שהוא מספק מנעה הדדית וחופש מדולוקים, הוא לא בהכרח מבטיח חופש מהרעה.

אלגוריתם משמעותי נוסף הוא אלגוריתם למפורט. כל תהליך במערכת מקבל מספר, ורק התהליך שהמספר שלו הוא הקטן ביותר מבין כולם יוכל להיכנס לחלק הקרייטי. האלגוריתם זהה הוא פשוט ויעיל, ומספק מנעה הדדית, חופש מדולוקים, וחופש מהרעה.

יחד עם זאת בסביבה של תהליכי מרובים המשימה הופכת להיות יותר מורכבת. אין פתרון שמתאים לכלום, והבחירה של האלגוריתם תלויות בדרישות והמגבליות הספציפיות של המערכת. למרות האתגרים, המרדף אחר



© 2022 מערך הפעלה

Operating Systems

Lecture 5 – Synchronization: Semaphores

Dr. Marina Kogan-Sadetsky

Semaphores

ראינו בಗאואה מימוש של סטטוס של סטטוס על ידי מוניטור. הסטטוס האמתי לא באמת משתמש במוניטור. כמו כן כל הכלים שראינו עד כה השתמשו ב-*wait*-*busy*.

עד כה הכלים שהיו לנו ביד זה רק אוטומטיים עם *wait* *busy*. עכשו אנחנו רוצים מנגנון שידע לזמן תהליכים.

ראו לציין כי יש עוד כלים, וגם גרסאות שונות של סטטוס אבל המטרה המרכזית של סטטוס היא להיפטר מה-*busy* *wait*.

Semaphores: outline

- ❑ Semaphores Introduction
- ❑ Producer-Consumer problem
- ❑ Counting semaphore by Binary semaphores
- ❑ Event counters
- ❑ Message passing synchronization

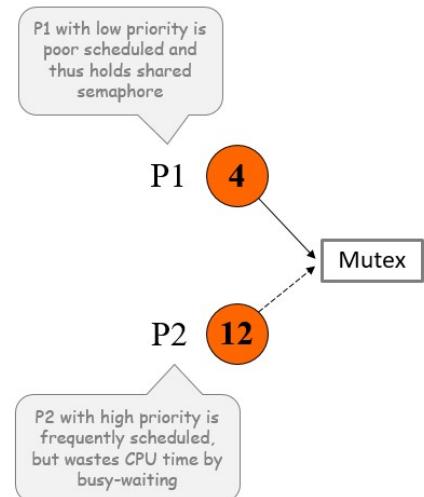
The mutual exclusion algorithms we saw
use busy-waiting.
What's wrong with that?

❑ Waste of CPU time

- but may be efficient if waiting-time is short

❑ Doesn't make sense for single core

- may cause (long but sufficient) **deadlock** – when low-priority process holds lock but is poor scheduled, and high-priority process gets (almost) all CPU time to execute busy-waiting (but does not progress)



הרעון של סמפור הוצג מדיםטרה בשנת 1965. דיקטורה הציע טיפוס חדש, שנקרא סמפור שיכל להיות עם הערך 0 או עם ערך חיובי. הרעיון שלו היה להקליל את המנגנון של sleep ו-wakeup עבור מספר תהליכיים. השימוש ב-sleep ו-wakeup מאפשר להיפטר מ-busy wait.

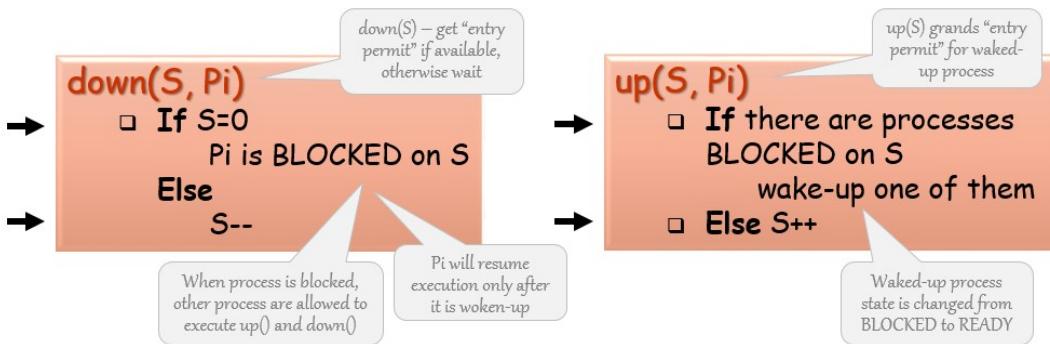
על מנת להמיץ את הבעיה אפשר להסתכל על הדיאגרמה משמאלי:
אם אנחנו בעצם גורמים לכך שתהיליך 2 עם עדיפות גבוהה יהיה ב-busy wait בעוד שתהיליך עם עדיפות נמוכה יכול להיכנס. אנחנו בעצם נרצה שתהיליך 2 יכנס למחבש של blocked במערכת הפעלה.

Semaphore

is synchronization primitive with
no busy-wait

Semaphores - definition

- $S=n$ is initially non-negative
 - $up()$ and $down()$ are **synchronized operations**
- n* is number of entry permissions
- synchronization of $up()$ and $down()$ is supported by semaphore implementation



הסמןורי לפי דיקסטרה

יש כמה דרכים להגדיר סמןורי, נתמקד בהצעה של דיקסטרה:
ישנן 2 פעולות על הסמןורי. הפעולות אלו הן אוטומטיות,
במובן שאם מתבצעת פעולה down במעבד - יש רק פעולה אחת צו, כך שהן מסונכרנות אחת על השניה.
פעולות אלו הן down ו-up (הכללות ל-sleep ו-wakeup). תחת פעולה אחת, הסמןורי בעצם שומר את
מספר ה-wakeup-down.

פעולה DOWN

מטרת הפעולה היא לבדוק האם הערך של הסמןורי S הוא גדול מ-0. אם כן, היא מורידה את הערך שלו ב-1

(כלומר משתמש ב-wakeup אחד) וממשיכה. אחרת, התחליף הולך לשונן מבלי להשלים את הפעולה.

פעולות UP

תפקידה להגדיל את הערך של הסמפור. אם יש תחליף אחד או יותר אשר ישנו על הסמפור, כלומר לא הצלחן להשלים את פעולה down, אז מערכת ההפעלה תבחר תחליף תחליף כזה כדי שיישלים את פעולה down.

נשים לב כי כל הפעולות המתבצעות על הסמפור, כולל דגימתו, שינויו והטלתו וככינסה למצב שונה מאשר חיבור להיות פעולות אוטומטיות. זאת כדי להבטיח שכאשר פעולה מוגדרת מוגדרת, אף תחליף אחר לא יוכל לגשת אליו.

הערך ההתחלתי של הסמפור:

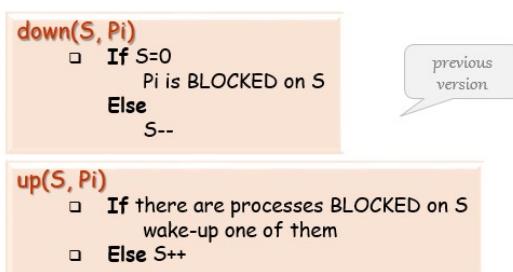
הערך ההתחלתי יכול להיות 0, ואז ניתן יהיה להשתמש בסמפור כמכניזם לשילוח סיגナル, או כדי לציין שהסמפור מלכתחילה נועל. הוא יכול להיות 1, ואז זה אומר שרק תחליף אחד יוכל לבצע down ואז יש מניעה הדדית (כלומר לא שומר wakeup כלל). הוא יכול להיות יותר מ-1 ואז אין מניעה הדדית, ככלומר שומר wakeup.

מסקנה: סמפור לא בהכרח שומר על מניעה הדדית. סמפור שימוש נכון מקיים מניעה הדדית אם הוא אוחROL עם הערך 1.

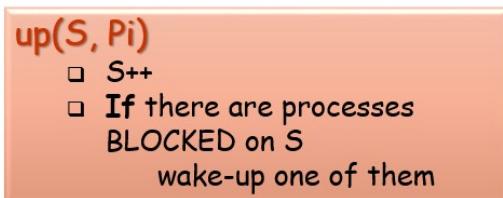
הערה: הכליה זהה הוא לא עצמאית ודורש עוד מנגנון סינכרון בשבייל הפעולות up ו-down, כראע יש לנו אוטומטיים.

האלגוריתמיקה בשקף נוכחנו, אנחנו נראה כיצד שינוי קטן במניטור יכול להוביל לבעיות:

Semaphores: is the following code correct ?



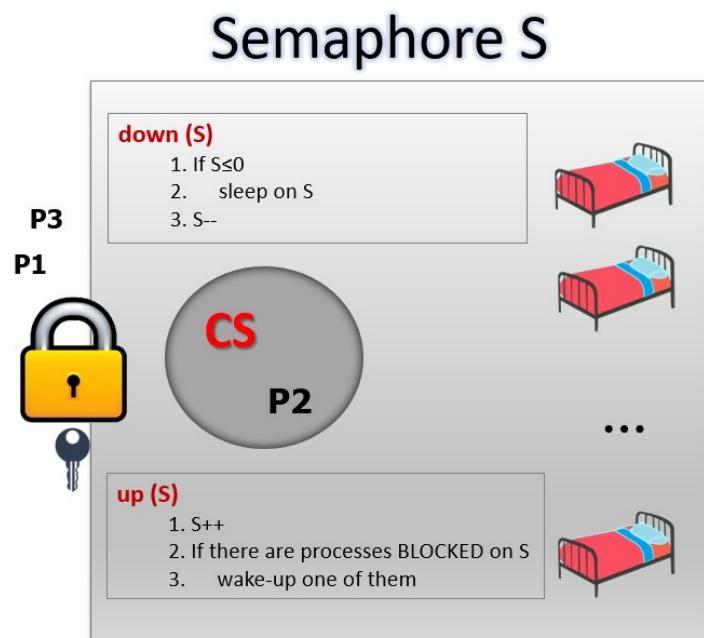
NO. Mutual exclusion might be violated. Let's demonstrate this by example.



כאן אנו מושנים את סדר הפעולות של UP, כך שקודם מגדילים את S ואז מעיררים תחליף אחר שישן על הסמפור. כתוצאה לכך נפגע ה-mutual exclusion.

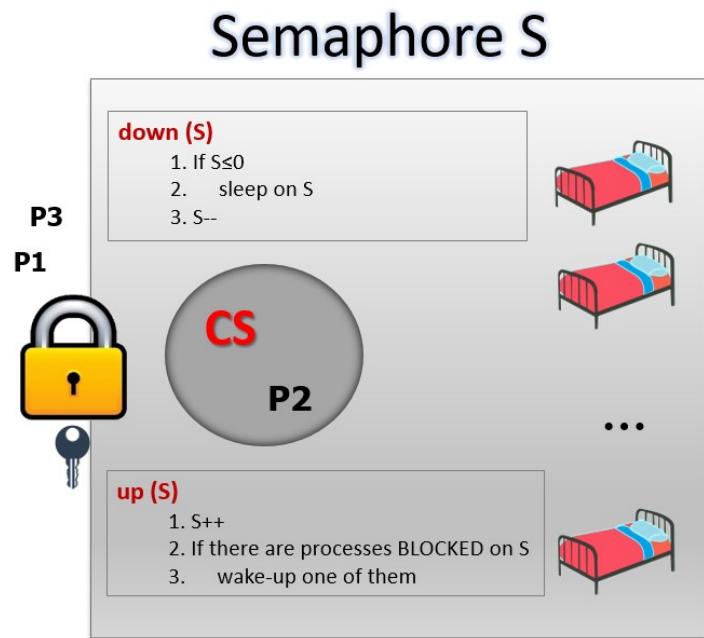
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
----	----	----



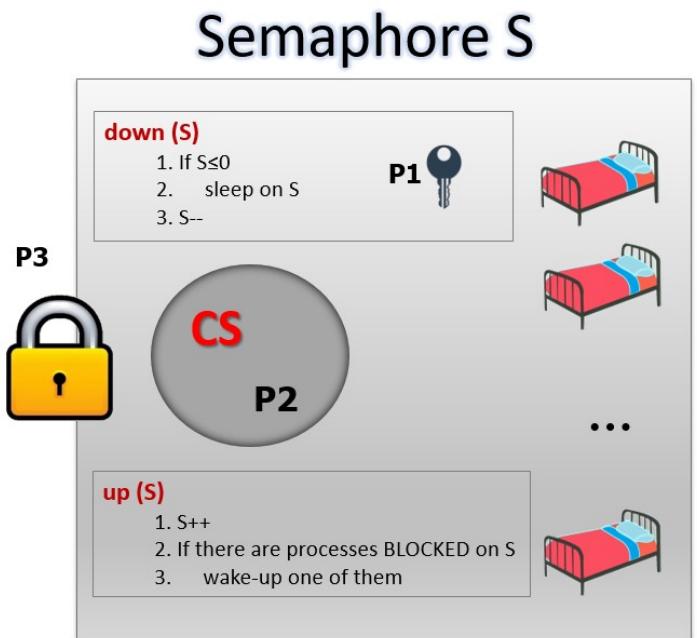
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S)		



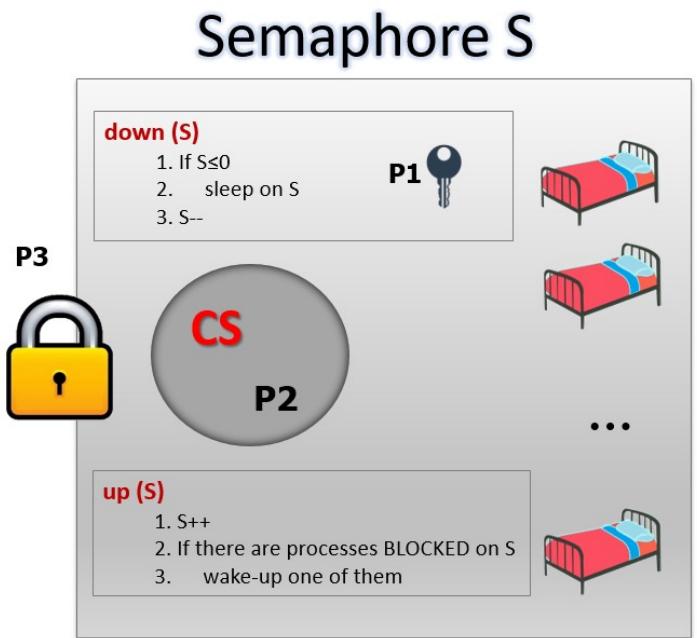
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S)		



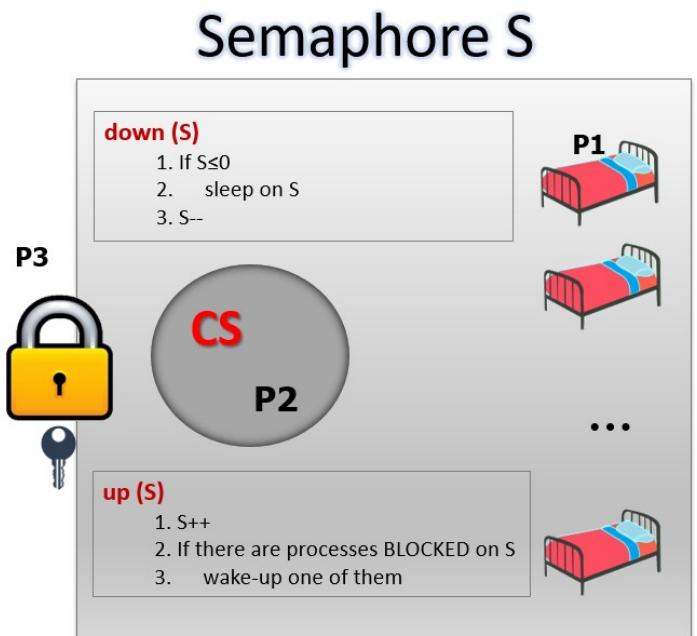
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:		



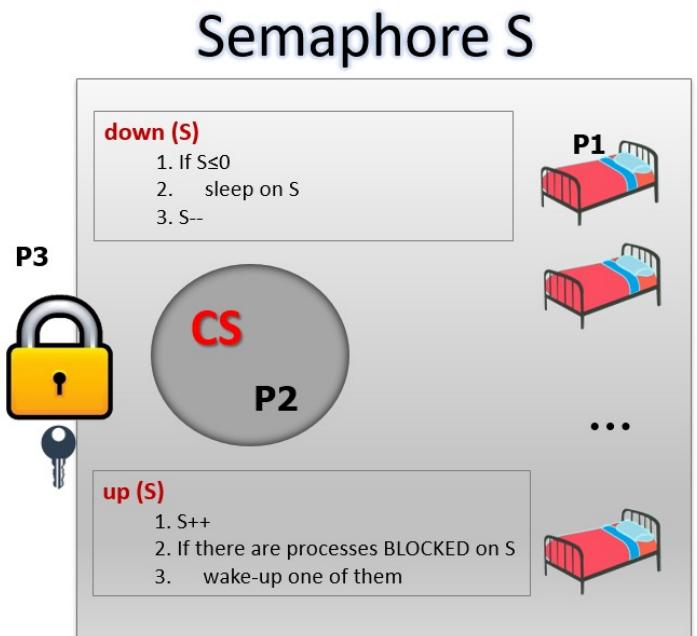
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:		



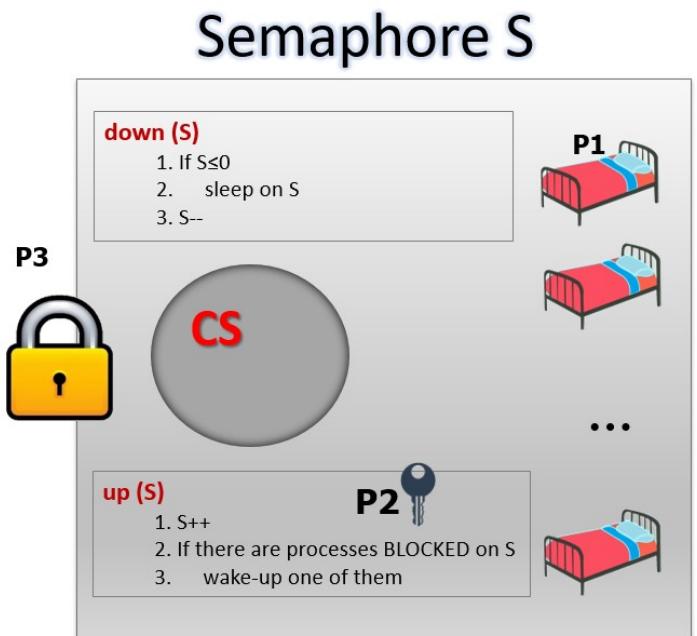
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:	exits CS up(S)	



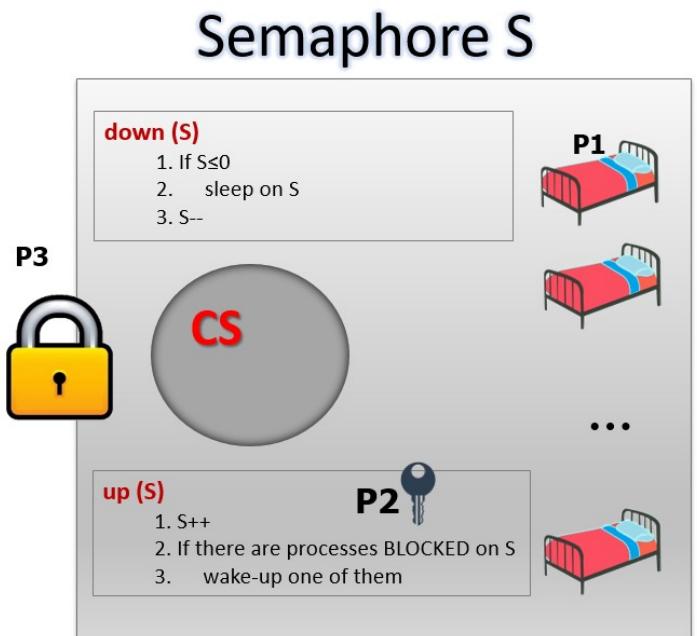
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:	exits CS up(S)	



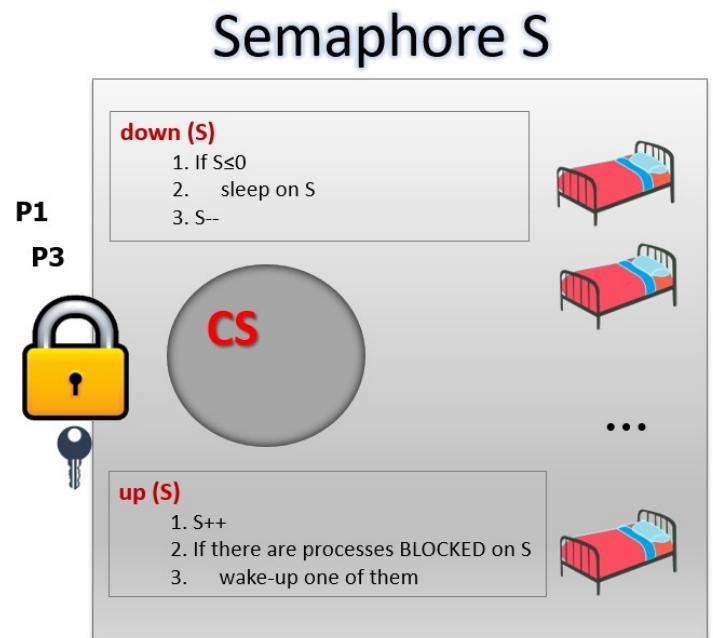
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED: exits CS up(S) $S \leftarrow 1$ wake-up P1		



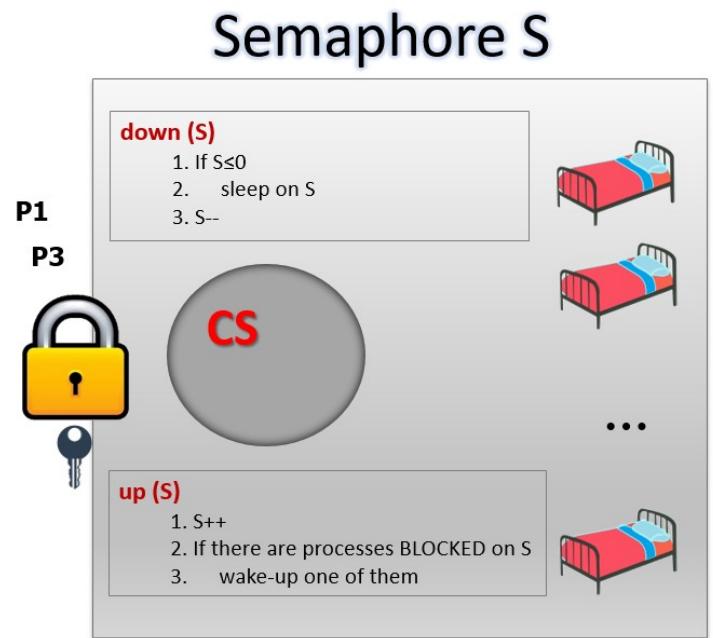
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:	exits CS up(S) $S \leftarrow 1$ wake-up P1	



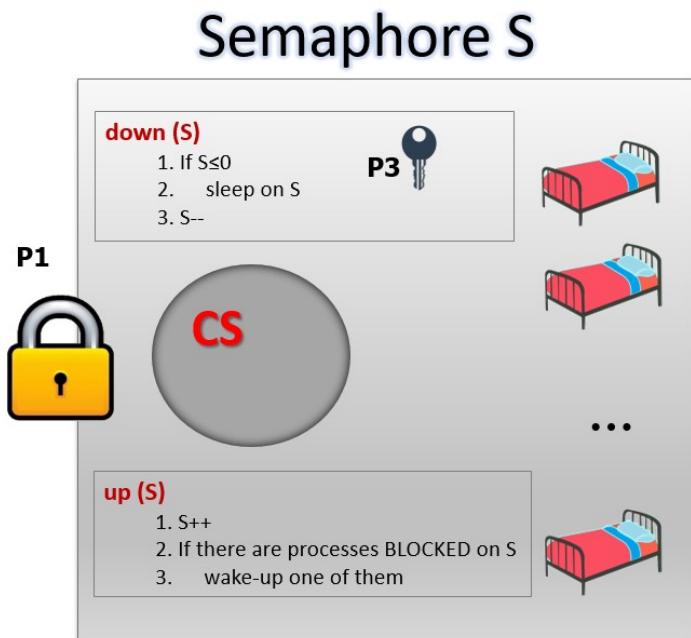
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:	exits CS up(S) $S \leftarrow 1$ wake-up P1	down(S)



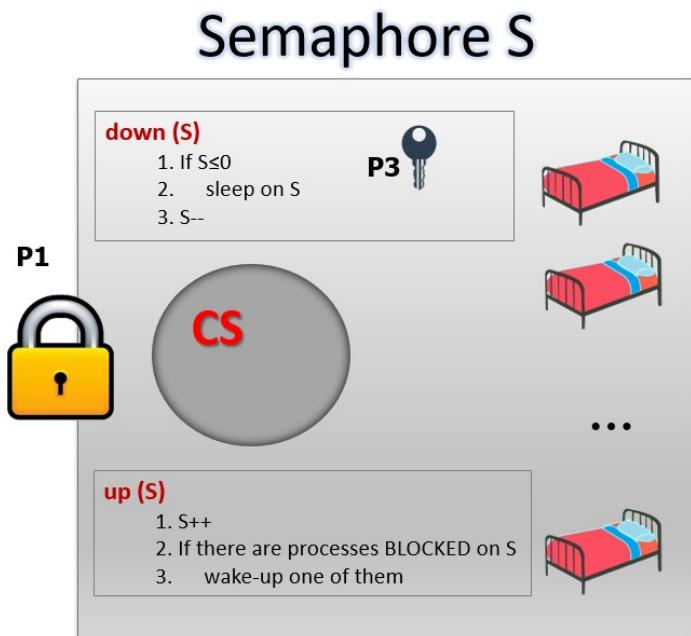
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:	exits CS up(S) $S \leftarrow 1$ wake-up P1	down(S)



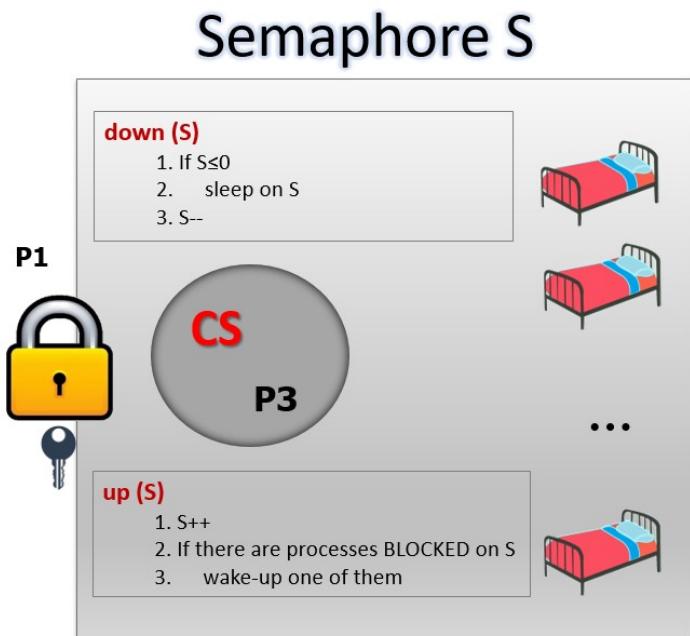
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:	exits CS up(S) $S \leftarrow 1$ wake-up P1	down(S) $S \leftarrow 0$ enter CS



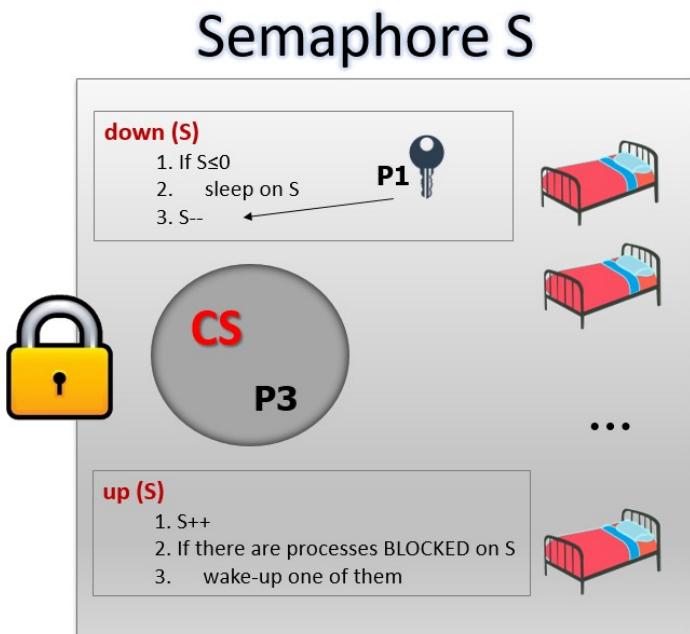
Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:	exits CS up(S) $S \leftarrow 1$ wake-up P1	down(S) $S \leftarrow 0$ enter CS



Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

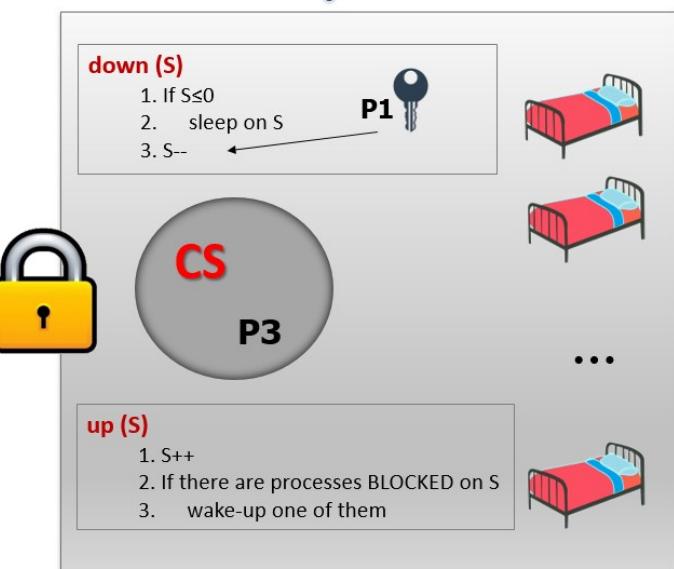
P1	P2	P3
down(S) BLOCKED: continues executing down(S)	exits CS up(S) $S \leftarrow 1$ wake-up P1	down(S) $S \leftarrow 0$ enter CS



Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED: continues executing down(S) S ← -1	exits CS up(S) S ← 1 wake-up P1	down(S) S ← 0 enter CS

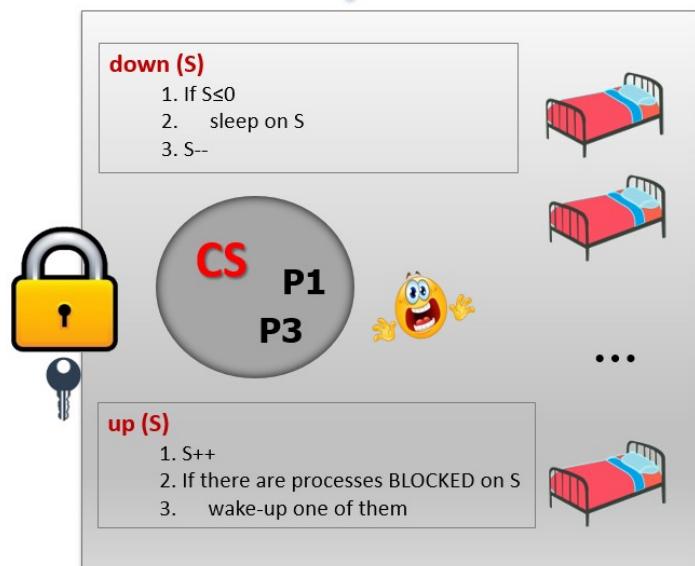
Semaphore S



Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED: continues executing down(S) S ← -1 enter CS	exits CS up(S) S ← 1 wake-up P1	down(S) S ← 0 enter CS

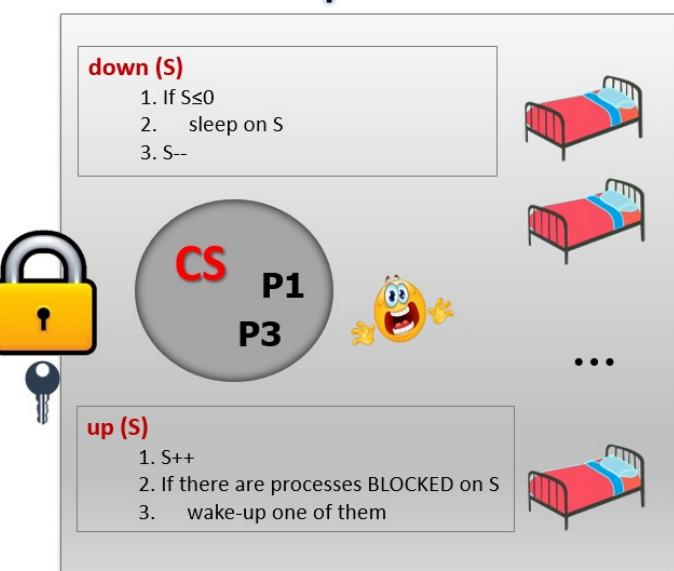
Semaphore S



Suppose: S=0, P2 in CS, P1 and P3 wants to enter CS

P1	P2	P3
down(S) BLOCKED:	exits CS up(S) $S \leftarrow 1$ wake-up P1	
continues executing down(S)		down(S) $S \leftarrow 0$ enter CS
$S \leftarrow -1$ enter CS		
single up() freed 2 down(s)		

Semaphore S



נניח שיש מפתח אחד על UP ועל DOWN ו

- S=0
- I-1

.

P1:

מבצע down והולך לישון.

P2:

ויצא מהקטע הקרייטי וביצע UP. מעלה את S ומעיר את P1

P1:

מתעורר ויבצע S-- ויכנס פנימה אבל P3 עוקף אותו אז יצליח לבצע DOWN

יארום ל-S להיות 0 ולהיכנס לקטע קרייטי.

P1:

ממשיר מבצע chown ומצליח להיכנס לקטע הקרייטי.

קיבלנו כי 2 תהליכי הצלicho להיכנס לקטע קוד קרייטי.

עיקנון שבגלו סמפורים יכולים ליפול:

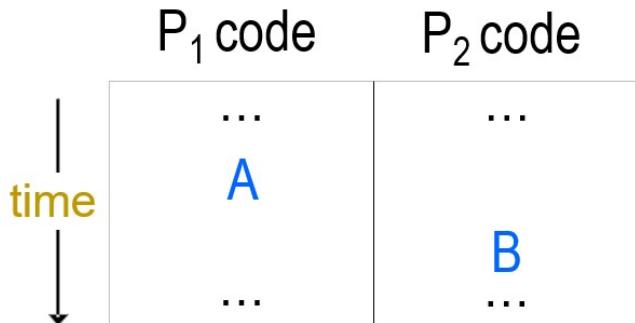
תהליך אחד כבר עשה chown ונכנס בפנים, ואז תהליך נוסף מגיע ועושה chown, כתוצאה לכך הוא הולך לישון. הבעיה היא שבפעולה ה-chown לא נלקחה בחשבון האפשרות שזמן אחד בדיק התעורר, תהליך אחר יכול לבוא ולבצע את העדכון (או שניהם באים ומבצעים את העדכון). שוב - הנקודה שהיא בדיק אחרי שתהליך יוצא מ-

Semaphore as synchronizing tool: example 1

Processes: P_1, P_2

Semaphore: S

Task: code A should be executed before code B



How to use S in
order to guarantee
such an order of
execution?

שימוש בסמפור ככלי סנכרון

דוגמה 1:

בהתinan 2 תהליכי P_1 ו- P_2 אשר רצים במקביל, סמפור S יש לדאוג כי קטע קוד A יתבצע לפני קטע קוד B.

פתרון:

ניתן להשתמש בסמפור 1 עם ערך התחלתי 0. נתנו ל P_1 קטע B נבצע down, כתוצאה לכך P_2 יהיה חייב למתן לישון לפניו שייגש ל-S רק אם P_1 עדין לא ביצע את קטע קוד A.

אחרי קטע קוד A נבוצע P_2 , כתוצאה לכך P_1 יעיר את P_2 לאחר שביצע את קטע קוד A.

הערה: חיבים להיות עדינים מאוד עם השימוש שלו. כאן זה מכוון למטרה.

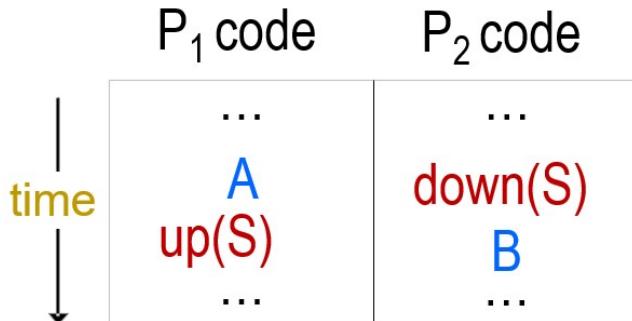
הלכה למעשה סדר ההרצה האפשרי הוא AB

Semaphore as synchronizing tool: example 1

Processes: P_1, P_2

Semaphore: $S = 0$

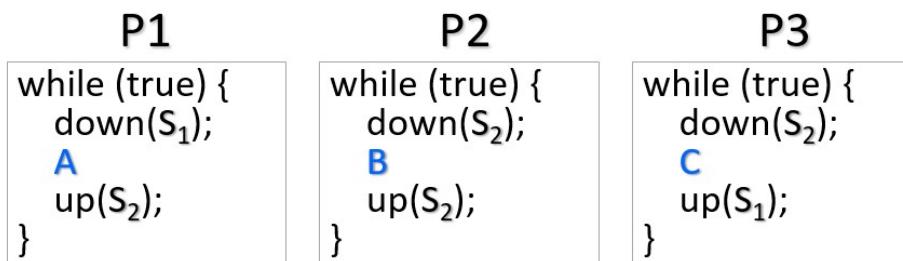
Task: code A should be executed before code B



Semaphore as synchronizing tool: example 2

Processes: P_1, P_2, P_3

Semaphore: $S_1 = 1, S_2 = 0$



Which execution orders of A, B, C, are possible? $(A \ B^* \ C)^*$

דוגמה 2:

בහינתן 3 תהליכיים אשר רצים במקביל ושני סמפורים כך שהראשון מאותחל להיות 1 והשני מאותחל להיות 0.

בහינתן סדר המפתחות לעיל מהם סדרי הריצה האפשריים של A, B, C, S_1, S_2 ?
נשים לב כי P_1 לוקח מנגול- S_1 זה הוא מאותחל להיות 1 תוך כדי ש- P_2 וה- P_3 התחליכים האחרים מבצעים $down$ על S_2 .
לכן בהכרח יתקיים A ללא תלות בשאר התהליכיים. לאחר מכן הוא מרים את S_2 .
마וחר והסمفורה השני מאותחל להיות 0, P_2 ו- P_3 הולכים לישון ישן. ככלומר רק לאחר קטע קוד A הם יתעוררו, ולכן
בכל ריצה אפשרית, A תמיד יהיה הראשון בסדר.

מכאן אם P2 הטעורר ראשון הוא יבצע את B ואז ירים את S2 - כתוצאה לכך B יבוצע פעם אחת או יותר עד שP3 הטעורר, יבצע את C ואז יעיר את P1 שיבצע את A.

אם P3 הטעורר ראשון, אז הוא יבצע את קטע C ויעיר את A וחזרנו להתחלה. כאן אין הכרה ש-B בכלל יבוצע. לכן סדרי ההרצה האפשריים הם למעשה כל המילימ' בשפה של הביטוי הרגולרי שבşekף.

Negative Semaphore

If S is negative, then there are $|S|$ blocked processes.

down(S, Pi)

- S--
- If $S < 0$
Pi is BLOCKED

up(S, Pi)

- S++
- If there are processes
BLOCKED on S (i.e., $S \leq 0$)
wake-up one of them

Answer: If there is a single waiting process,
then it would not be waked-up.

- S=1
- P1 down(S), S=0
- P1 enters CS
- P2 down(S), S=-1
- P2 goes sleep on S
- P1 exits CS and up(S), S=0
- P1 would not wake-up P2 since if($S < 0$) does not hold
- **Deadlock...**

If we change " \leq "
to "<", what
would happen ?

סمفורה שלילי:

מדובר בסمفורה שהערך שלו יכול להיות גם שלילי, בניגוד לסמפורים הכלליסיים. תכונה חשובה של סمفורה שלילי, היא שאם S שלילת או הערך המוחלט שלו הוא מספר התהליכים שি�שנים על הסمفורה.

שימושים עיקריים:

- א. שמירה על משאבם.
- ב. הצלת עדוף
- וועוד ...

לשם כך יש להגדיר מחדש הפעולות UP ו-DOWN

down:

ראשית מורידים ב-1 את הערך של הסمفורה - כלומר התהליך ה-1 שולח בקשה גישה למשאב משותף. אם S שלילי, זה אומר שלא היו מפתחות זמינים (או באופן מקביל, יש כבר תהליכיים אחרים ששמתיינים למשaab), אז הוא יכנס ל-BLOCK.

אחרת, אם S אינו שלילי, אז התהליך יוכל המשיך.

UP:

שחרר מפתח. תחילת הוא מעלה את הערך של S, כלומר התהליך ה-1 שחרר את המשאב המשותף. אם S עדין אי חיובי, זה סימן לכך שיש תהליכיים אשר ישנים על S, ולכן הוא מעיר את אחד התהליכים ויצא.

למה קטן/שווה? מה זה משנה? אם ה-S הוא 0 אז עשינו ++ לערך שלילי, כלומר היה מישו אחד שהמתין למפתח.
אם לא היונו עושים זאת היינו מפספסים אותו - DEADLOCK!

הdagש החשוב הוא שהערך השלילי אומר שיש ערך מוחלט של S תהליכיים שמתינים למפתח - זה קאונטר מייד!

נקודה שבה מנגנים יכולים ליפול:

בנוקודה סביר בה הערכים של הסמפור הם בסביבה אבוליטית של ערכים בסמפור (במקרה זה גיד 0), אלו נקודות קטנות שיכלות להוביל אלגוריתמים. כאן זה הפיל את האלגוריתם כי ברגע שתהליך אחד הגיע ל-UP יהיה תהליך אחד שישן. אם לא היה את השינוי, הסמפור קיבל את הערך 0 וזה לא יכוליק את התנאי ב-UP - מה שבוביל לכך שתהליך לא יתעורר.

Negative Semaphore Implementation

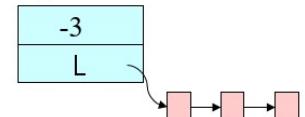
Initially: $S.value = n$, $S.L$ is a list of sleeping processes, $S.flag = 0$

```
down(S, Pi):
    while (Test-And-Set(S.flag)==1):
        S.value--
        if (S.value < 0) {
            add Pi to S.L
            sleep (Pi)
            S.flag=0
        }
    else S.flag=0
```

```
up(S, Pi):
    while (Test-And-Set(S.flag)==1):
        S.value++
        if (S.value <= 0) {
            remove some process P from S.L
            wakeup (P)
        }
    S.flag=0
```

Test-And-Set() atomic instruction is used to get mutual exclusion of down() and up() execution.

Semaphore struct:



This implementation uses **busy-wait** spin-lock, since while(Test-And-Set(S.flag)) is busy-wait loop.

Test-And-Set(S.flag):

- $prev \leftarrow S.flag$
- $S.flag \leftarrow 1$
- $return prev$

מימוש סמפור שלילי בעזרת פעולה אוטומית

הכלי היחידי - אוטומיים:

אתחול:

cutת הסמפור הוא בעצם מבנה בעל 3 שדות:
הערך $S.value$ אשר מעודכן להיות ח, כלומר מספר המפתחות בסמפור (או מספר המשאבים הפנויים) הוא ערך מוחלט של ח.

הרשימה L מאותחלת להיות רשימה שעוקבת אחרי תהליכיים אשר ישנים על התהליך.
השדה $S.flag$ נדרש לסייע פועלות בעזרת TaS .

Down:

```
down(S,Pi):
    while(Test-And-Set(S.flag) == 1);
```

מעין תפיסת מפתח על הגישה לסמפור. מבטיח כי בכל זמן מתיון רק תהליך אחד יוכל לבצע את הקוד של פעולה זו.

```
S.value--
```

הורדה ב-1 - משמעותה שההתהיליך תפס מפתח.

```
if (S.value < 0){  
    add Pi to S.L  
    sleep(Pi)  
    S.flag = 0  
}
```

אם יש כבר תהליכיים מסוימים לקחת מפתח אז הוסיף את התהיליך לרשימה, דבר המעיד על כך שההתהיליך כרגע ישן, הכנס את התהיליך למצב שונה ושחרר את המניעול על מנת לאפשר לתהליכיים אחרים ל תפס אותו.

```
else S.flag = 0
```

אחרת, התהיליך לאלקח את המניעול האחרון, ככלומר יש עוד תהליכיים שיכולים ל תפס מניעול (או לשחרר ב-UP), ולכן שחרר את המניעול.

שאלה: אם אנחנו קודם יישנים, איך נוכל לשחרר את flag? מה עושים בזה?

תשובה:

עדין זה אפשרו עובד כי כל זה בסופו של יומם מהוות קריית מערכת! תהיליך בפני עצמו לא יכול להריז את הקוד הזה. אם אנחנו בכל מקרה משתמשים במערכת הפעלה, אז אפשר להأدיל בראש ולאפשר ל kernell להריז את הכל קריית מערכת. במקרים אחרים, הקוד במצב רץ במרחב הקרנל. ברגע שמבצעים את ה-sleep מקריית המערכת, אפשר לבצע רוטינת sleep במצב צורה כזו שמתוייחסת לנקודה העדינה.

שאלה: מה אם מחליפים בין השורות?

תשובה: מערכת הפעלה עלולה לפגוע במנגנון וליצור מצב של דזדון.

UP:

```
up(S, Pi):  
    while(Test-And-Set(S.flag) == 1);
```

סנכרון על האישה ל פעולה.

```
S.value++
```

mbtata שחרור מפתח.

```
if(S.value <= 0){  
    remove some process P from S.L  
    wakeup(P)  
}
```

אם יש תהליכיים אשר מסוימים קיבל אישה ל משאב (רוצים מפתח):
הסר תהיליך מהרשימה והעיר אותו

S.flag = 0

אפשר לתהליכים אחרים לגשת לפעולות up-down.

עוד שאלת מכשילה:

בנינו את הכלי כדי להיפטר מה-WAIT BUSY. אבל אנחנו משתמשים באוטומטיים שכן אמורים להוביל ל-busy wait, ועודין זה מימוש סבבה: במצב שבו משתמשים רק באוטומטיים - יהיה wait עד שייכנס לקטע הקרייטי. פה תהליך מקבל flag בודק אם הוא יכול להיכנס, וממתין רק ב-SLEEP או יש WAIT רק בollowה הראשונית - המתנה לפעולה מאוד קצרה, ככל קרצה שלא נחשיב אותה C-wait busy.

באופן כללי: משתמש ב-wait busy רק בשילוב דברים קצריים.

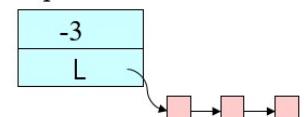
Negative Semaphore Implementation

Initially: S.value = n, S.L is a list of sleeping processes, S.flag = 0

```
down(S, Pi):  
    while (Test-And-Set(S.flag)==1);  
    S.value--  
    if (S.value < 0) {  
        add Pi to S.L  
        sleep (Pi)  
        S.flag=0  
    }  
    else S.flag=0
```

```
up(S, Pi):  
    while (Test-And-Set(S.flag)==1);  
    S.value++  
    if (S.value <= 0) {  
        remove some process P from S.L  
        wakeup (P)  
    }  
    S.flag=0
```

Semaphore struct:



NO. Suppose P1 executes down(S), and P1 must go sleep since P2 is currently in CS. If P1 gets asleep first, P1 could not set flag to be 0. Now, nobody can execute nor down(S) neither up(S). **Deadlock.**

Is the code of down() and up() may be executed by the process itself (i.e., in user mode)?

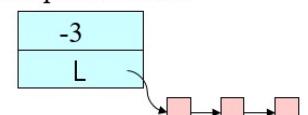
Negative Semaphore Implementation

Initially: S.value = n, S.L is a list of sleeping processes, S.flag = 0

```
down(S, Pi):  
    while (Test-And-Set(S.flag)==1);  
    S.value--  
    if (S.value < 0) {  
        add Pi to S.L  
        S.flag=0  
        sleep (Pi)  
    }  
    else S.flag=0
```

```
up(S, Pi):  
    while (Test-And-Set(S.flag)==1);  
    S.value++  
    if (S.value <= 0) {  
        remove some process P from S.L  
        wakeup (P)  
    }  
    S.flag=0
```

Semaphore struct:



And what if P1 first sets flag to be 0, and then goes asleep?

STILL NO. If P1 first set flag to be 0, then OS scheduler might stop P1, and schedule P2 before P1 gets asleep. P2 might finish CS, and execute up(S). P2 would try to wakeup P1 since P2 finds S.value to be negative. BUT P1 STILL DOES NOT SLEEP !!! So P2 would not wake up P1. When P1 is re-scheduled, P1 would continue executing down(S), and gets asleep. Now there might be nobody who can wake up P1. **Deadlock.**

So, down(S) and up(S) should not be executed by the process - they should be or **system calls**, and then executed by OS, or be executed by some other (manager) thread.

Semaphores: outline

- ❑ Semaphores Introduction
 - ❑ Producer-Consumer problem
 - ❑ Counting semaphore by Binary semaphores
 - ❑ Event counters
 - ❑ Message passing synchronization

בעה קלאסית שראינו כבר ב-SPL שאפשר לפתור עם סמפורים.

Bounded Producer-Consumer Problem

Producer produces items that are consumed by consumer.



יש 2 אגרסואות אך נטמקד באחת מהן. האחת שבה ה-buffer לא מוגבל והשנייה בה הגודל מוגבל.
נדבר על הארסה השנייה.
יש לנו לפחות יצרך אחד ולפחות צרך אחד.
היצרך מייצר למערך והצרך לוקח מהמערך.

אנו צריכים לסנן את תהליך הצורך כי יש כאן צורך משותף וגם אנחנו לא רוצים לייצר יותר ממה שאפשר
לצורך ולהפוך, שלא יהיה ניתן לצרוך יותר ממה שאפשר לייצר.

Bounded Producer-Consumer Problem

```

semaphore mutex = 1           // binary semaphore for access control to critical section
semaphore empty = N            // empty semaphore - counts empty buffer slots
semaphore full = 0             // full semaphore - counts full slots
    
```

full semaphore stops Consumer, **empty** semaphore stops Producer

mutex (binary) semaphore is to synchronize usage of shared buffer

Producer:

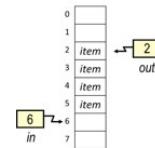
```

while(TRUE)
    produce_item(&item) // generate item
    down(&empty) // is there free space in buffer ?
    down(&mutex) // enter critical section
    insert_item(item) // insert item into buffer
    up(&mutex) // leave critical section
    up(&full) // there is an item to consume
    
```

Consumer:

```

while(TRUE)
    down(&full) // is there items to consume ?
    down(&mutex) // enter critical section
    remove_item(&item) // take item from buffer
    up(&mutex) // leave critical section
    up(&empty) // update count of empty
    consume_item(item) // consume item
    
```



הצעת פתרון עם 3 סמפורים:

mutex:

מה שיספק את הגישה בלעדית למערך, כלומר המניעה ההדדיות.

empty:

ערך התחלתי הוא N - האודל של הבאפר (בדוגמה בציור: 8)

full:

הוא יהיה 0.

השניים האחרים הם לא mutex.

ה-ח מקומות ב-EMPTY זה בדיק מספק התאים הריקים במערך בהתאם. ברגע שייצר רצח ליצר מוצר אז הוא לוקח מפתח, כלומר מבצע S-- ואז מייצר אותו. אם הייצן ירצה לייצר מוצר אחד יותר מגודל הבאפר, הוא יתקע בסמפור של empty.

הכרך עושה הפור: full אומר כמה תאים מלאים יש. אם יש תאים מלאים, אז full הוא לא 0 ואז יהיה אפשר לחת מפתח, אחרת הוא יתקע עד שייהיו מוצרים.

Bounded Producer-Consumer Problem

```

semaphore mutex = 1           // binary semaphore for access control to critical section
semaphore empty = N            // empty semaphore - counts empty buffer slots
semaphore full = 0             // full semaphore - counts full slots

```

Producer:

```

while(TRUE)
    produce_item(&item) // generate item
    down(&empty) // is there free space in buffer ?
    down(&mutex) // enter critical section
    insert_item(item) // insert item into buffer
    up(&mutex) // leave critical section
    up(&full) // there is an item to consume

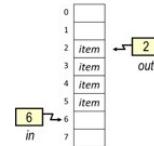
```

Consumer:

```

while(TRUE)
    down(&full) // is there items to consume ?
    down(&mutex) // enter critical section
    remove_item(&item) // take item from buffer
    up(&mutex) // leave critical section
    up(&empty) // update count of empty
    consume_item(item) // consume item

```



NO. Suppose two producers P1 and P2 want to insert items into the buffer. Suppose in=6. If P1 and P2 insert their items simultaneously, both would insert into 6'th index of the buffer, and one of the items would be discarded...

May we **not use** the **mutex** ?

לו הינו מօותרים על mutex, "יתכן כי צריך מסויים היה יכול לנסוטות "לצורך" משהו שעדין לא ייצור"

Bounded Producer-Consumer Problem

```

semaphore mutex = 1           // binary semaphore for access control to critical section
semaphore empty = N            // empty semaphore - counts empty buffer slots
semaphore full = 0             // full semaphore - counts full slots

```

Producer:

```

while(TRUE)
    produce_item(&item) // generate item
    down(&empty) // is there free space in buffer ?
    down(&mutex) // enter critical section
    insert_item(item) // insert item into buffer
    up(&mutex) // leave critical section
    up(&full) // there is an item to consume

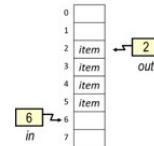
```

Consumer:

```

while(TRUE)
    down(&full) // is there items to consume ?
    down(&mutex) // enter critical section
    remove_item(&item) // take item from buffer
    up(&mutex) // leave critical section
    up(&empty) // update count of empty
    consume_item(item) // consume item

```



YES. The reason is that Consumer and Producer never try to use same index in the buffer concurrently. Suppose buffer with i items, buffer[0] till buffer[i-1]. Then Consumer will use buffer only at these indexes, and then would be BLOCKED on full semaphore. Producer will use only [i,n] indexes in the buffer, and then would be BLOCKED on empty semaphore.

If there are only **single** Producer and **single** Consumer, may we **not** use the **mutex** ?

ומה אם יש יצרן אחד וצרן אחד?
אז אין צורך בmutex כי אם זה ייפול, זה יהיה רק כאשר צריך ירצה לצורק משהו שלא נוצר עדין, אבל אז הצורך היה נתפרק בסמפור של full.

Bounded Producer-Consumer Problem

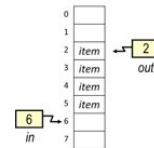
```
semaphore mutex = 1           // binary semaphore for access control to critical section
semaphore empty = N            // empty semaphore - counts empty buffer slots
semaphore full = 0             // full semaphore - counts full slots
```

Producer:

```
while(TRUE)
    produce_item(&item) // generate item
    down(&empty) // is there free space in buffer ?
    down(&mutex) // enter critical section
    insert_item(item) // insert item into buffer
    up(&mutex) // leave critical section
    up(&full) // there is an item to consume
```

Consumer:

```
while(TRUE)
    down(&mutex) // enter critical section
    down(&full) // is there items to consume ?
    remove_item(&item) // take item from buffer
    up(&mutex) // leave critical section
    up(&empty) // update count of empty
    consume_item(item) // consume item
```



NO. Suppose empty buffer. Suppose consumer comes first, down the mutex, and then goes sleep on full semaphore. In this case no producers can put items into buffer since mutex is occupied. **Deadlock.**

We switched two first lines in consumer code.
Is the code correct ?

מה קורה אם היינו מחליפים את הסדר?
יש דלוק כי אם הבאפר ריק, הצריך יכול לחת את המפתח של mutex ולהיתקע על full. כתוצאה לכך היצרא לא יכול להמשיך ונתקע גם הוא.

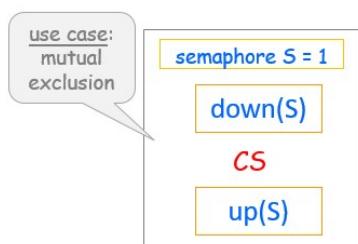
Semaphores: outline

- ❑ Semaphores Introduction
- ❑ Producer-Consumer problem
 - ❑ Counting semaphore by Binary semaphores
 - ❑ Event counters
 - ❑ Message passing synchronization

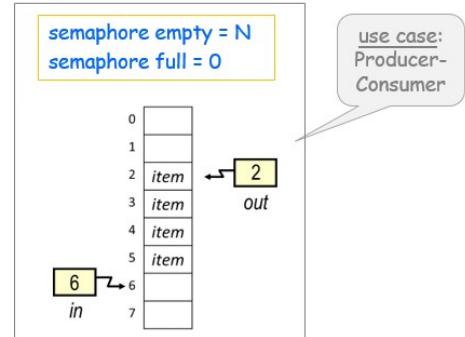
Binary Semaphore vs. Counting Semaphore

- $S \in \{0, 1\}$ single permit
- $\text{down}(S, P)$ blocks P if $S=0$
- $\text{up}(S)$ either wakes up some process,
or sets $S \leftarrow 1$

- $S \in \{-\infty, n\}$ n permits
- $\text{down}(S, P)$ blocks P if $S < 0$
- $\text{up}(S)$ increments S , and wakes up some process if $S \leq 0$



How to implement a counting semaphore by using binary semaphores ?



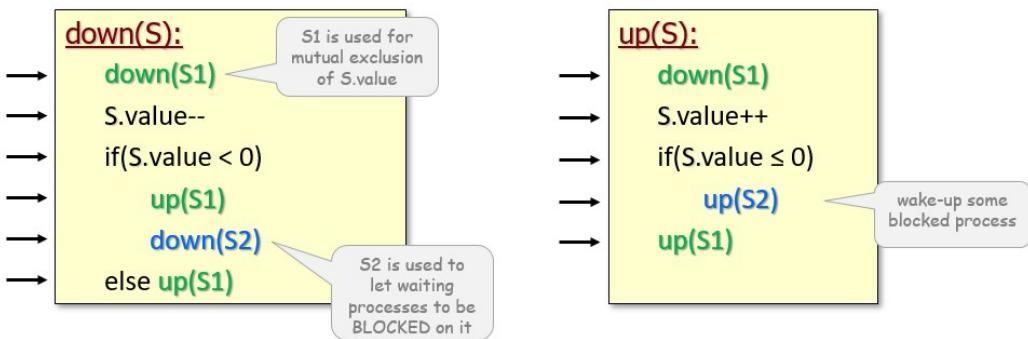
סמנפור בינהרי הוא סמנפור אשר מקבל רק את הערכים 0 ו-1.
 בפעולת down , תהליך הולך לישון אם $S=0$
 סמנפור מסוג counting הוא סמנפור שהערך שלו הוא בתחום של 0 ועד מינוס אינסוף (כלומר מוגבל על ידי מספר התהליכים שמנסים לחתת מפתח, ומספר זה אינו חסום).

בפעולת down , תהליך הולך לישון אם S אינו בעל חיובי.

אנו רוצחים למשתמש סמנפור counting בעזרת סמנפור בינהרי. לשם כך אנו נראה סדרה של מימושים שבהתחלתה לא יהיו נכונים ולאחר מכן נראה מימוש נכון.

Counting semaphore by binary semaphores – try #1

Initially: binary semaphore $S1 \leftarrow 1, S2 \leftarrow 0, S.value \leftarrow n$



This code does not work. Why?

It may deadlock.

ניסיון סתמי של מימוש כושל:

down(s1):

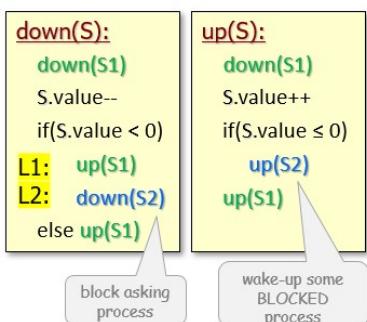
הוא המנעה החדדית על ה- $S.value$:

s2:

המספר שאחראי על הנעילה כאשר S שלילי.

Counting semaphore by binary semaphores – try #1

Initially: $S1 \leftarrow 1, S2 \leftarrow 0, S.value \leftarrow n$



Problematic scenario:

Suppose $S.value = 0$

P1 $\text{down}(S)$, $\rightarrow S.value = -1$

P1 is preempted between lines L1 and L2

P2 $\text{down}(S)$, $\rightarrow S.value = -2$

P2 is preempted between lines L1 and L2

P3 performs $\text{up}(S)$
 $\rightarrow S.value = -1, S2.value=1$

there is still nobody to wakeup

P4 performs $\text{up}(S)$
 $\rightarrow S.value = 0, S2.value=1$

there is still nobody to wakeup

P1 re-scheduled and perform L2

$\rightarrow P1$ enters CS

P2 re-scheduled and perform L2

P2 is BLOCKED on S2. Deadlock....

המקרה שבו זה נופל:

נניח וכרגע $S.value = 0$

P1:

מצlich לגשת ל- $\text{down}(S)$ אבל הוא נעצר על ידי מערכת הפעלה בין L1 ו-L2.

P2:

אותו דבר.

P3:

בא לבצע UP, קיבלנו 1 כי 2 עדין לא התחבוצה כלל.

P4:

אם מנסה לעשות UP אבל אף אחד לא ישן על S2 והוא בכל מקרה 1 אז הוא מתפספס.

ברגע ש-P1 ו-P2 ייאשו לבצע את L2, אחד מהם יכנס לדילוק.

למה זה נפל?

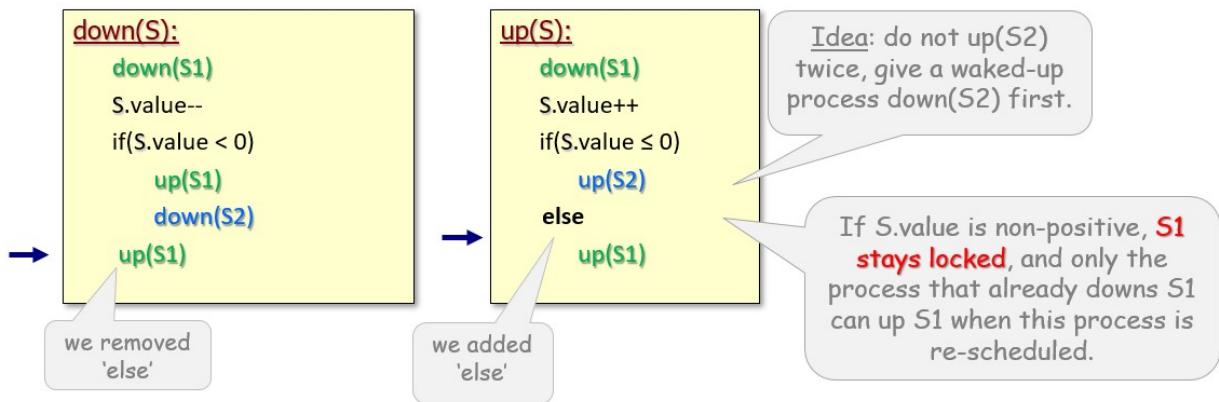
הרגע שבו מבצעיםrup לסמפור אחד dowm לסמפור אחר - זאת נקודה שבה מערכת הפעלה יכולה להתעורר בה. דילוק יכול להיות או כאשר יש (S1-down) - S2 הוא כבר 0. אבל אז זה אומר שתהליך אחר נמצא כבר באחת הפונקציות ובשלב מסוים מערכת הפעלה מעירה אותן (אמרנו במודול שלנו שתהליכיים כן יקבלו זמן ריצה בשלב צזה או אחר). אז שם אין בעיה.

עוד מקום אפשרי לדילוק הוא כאשר מבצעים (S2-down). אם הוא נכנס לדילוק שם, זה אומר שכאשר תהליך אחר עשה (S)rup או שלא ביצע (S2)rup כי הערך היה חיובי (לא יתכן כי אז אין שום אינטראס לתהליך ב-down להיכנס לתנאי). נשאר רק המצב שבו שתהליך אחד ביצע (S2)rup אבל תהליך אחר לא תפס אותן. זה יתכן כאשר מערכת הפעלה ביצעה החלפת הקשר של תהליך גע לפני ה-down. מכאן אפשר לבנות את הדוגמה.

Counting semaphore by binary semaphores – try #2

Hemmendinger, 1988

Initially: $S1 \leftarrow 1, S2 \leftarrow 0, S.value \leftarrow n$



Does this code work?

Let's try to run previous problematic scenario.

ניסוי נוסף של מדען מ-1988.

P1:

מבצע (S)down ונכנס.

P2:

אם מבצע זאת, עושה (S1)rup והולך לשון על אינטראפט שעון.

P3:

אותו דבר.

P1:

עושה UP, משחררת את S2 אבל לא שחררנו את S1 - לא שחררנו את המונע הדדי!

P2,P3:

עוד יכולם להיכנס.

P4:

רוצה לעשות down ונתקע על S1

זה פתרון לבעה הקודמת.

Counting semaphore by binary semaphores – try #2

Hemmendinger, 1988

Initially: $S1 \leftarrow 1, S2 \leftarrow 0, S.value \leftarrow n$

down(S):
 down(S1)
 S.value--
 if(S.value < 0)
 L1: up(S1)
 L2: down(S2)
 up(S1)

up(S):
 down(S1)
 S.value++
 if(S.value ≤ 0)
 up(S2)
 else
 up(S1)

Does this code work?

Let's try to run previous problematic scenario.

Initially: $S1 \leftarrow 1, S2 \leftarrow 0, S.value \leftarrow n$

down(S):
 down(S1)
 S.value--
 if(S.value < 0)
 L1: up(S1)
 L2: down(S2)
 up(S1)

up(S):
 down(S1)
 S.value++
 if(S.value ≤ 0)
 up(S2)
 else
 up(S1)

Does this code work?

Now let's try to use this counting semaphore for producers-consumers.

Problematic scenario of try #1:

Suppose $S.value = 0$

P1 down(S), → $S.value = -1$
P1 is preempted between lines L1 and L2

P2 down(S), → $S.value = -2$
P2 is preempted between lines L1 and L2

P3 performs up(S)
→ $S.value = -1, S2 = 1, S1 is still 0$

P4 performs up(S)
→ P4 is BLOCKED on S1

P1 is re-scheduled and performs L2
→ $S2 = 0$
P1 up(S1) → P4 wakes-up, S1 is still 0
P2 re-scheduled and perform L2
P2 is BLOCKED on S2. But...
P2 would be later waked-up by P4.



Counting semaphore by binary semaphores – try #2

Hemmendinger, 1988

Initially: $S1 \leftarrow 1, S2 \leftarrow 0, S.value \leftarrow n$

down(S):
 down(S1)
 S.value--
 if(S.value < 0)
 L1: up(S1)
 L2: down(S2)
 up(S1)

up(S):
 down(S1)
 S.value++
 if(S.value ≤ 0)
 up(S2)
 else
 up(S1)

Does this code work?

Now let's try to use this counting semaphore for producers-consumers.

Problematic scenario of try #1:

Suppose $S.value = 0$

P1 down(S), → $S.value = -1$
P1 is preempted between lines L1 and L2

P2 down(S), → $S.value = -2$
P2 is preempted between lines L1 and L2

P3 performs up(S)
→ $S.value = -1, S2 = 1, S1 is still 0$

P4 performs up(S)
→ P4 is BLOCKED on S1

P1 is re-scheduled and performs L2
→ $S2 = 0$
P1 up(S1) → P4 wakes-up, S1 is still 0
P2 re-scheduled and perform L2
P2 is BLOCKED on S2. But...
P2 would be later waked-up by P4.



try #2 for Producers-Consumers

```
producer:
while(TRUE)
produce_item(&item)
down(&empty)
down(&mutex)
insert_item(item)
up(&mutex)
up(&full)
```

```
consumer :
while(TRUE)
down(&full)
down(&mutex)
remove_item(&item)
up(&mutex)
up(&empty)
consume_item(item)
```

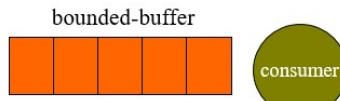
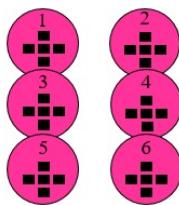
Problematic Scenario:

Consider a bounded-buffer of 5 slots.
Consider 6 producers P1, P2, P3, P4, P5, P6, with five items each.
Consider a single consumer.
Initially: mutex = 1, empty.value = N, full.value = 0

```
down(S):
down(S1)
S.value--
if(S.value < 0)
up(S1)
down(S2)
up(S1)
```

```
up(S):
down(S1)
S.value++
if(S.value ≤ 0)
up(S2)
else
up(S1)
```

producers



כעת לאחר שהבנו שהקוד עובד, ננסה לפתרור אותו את בעיית צרכן-יצwan.
מה שצבעו - הקריאות לסמפור שלנו.

הפרודוסרים עדין לא קיבלו זמן ריצה כדי לשים את המוצריים שלהם בבאפר, שהוא בהתחלה ריק.
נניח כי יש צרכן אחד.

P1:

נכns פנימה ומקבל מספיק זמן ריצה כדי להכניס את כל הפריטים שלו - קלומר מלא את הבאפר.
כל שאר התהליכים מנסים גם לכתוב והולכים לישון.
כתוצאה לכך הערך של empty הוא 5.

לאחר מכן הצרכן מגיע וצריך אחד, הוא מצליח לhattakdm כי full אינו 0. בשלב מאוחר יותר הוא מנסה לעשות דע על empty:

כל לראות שהוא יצליח לבצע (S1),
הערך הוא אי שלילי ולכן מעיר מישוה שישן על S2.
נניח בה"כ כי P2 הטעור, אך עדין לא מקבל זמן ריצה, ובמקרה הצרכן קיבל זמן ריצה.
הצרכן צריך להיות מסוגל להוריד את הפריטים ואז לhattakdm. בפועל הוא לא יצליח להוריד את השני.

down(full)->up(empty):

מסוגל, אך הוא לא מצליח לעשות up על empty כי S1 הוא 0 ואז נתקע על (S1)down.

לא קיבלנו דילוק, כי אומנם הצרכן לא נתקע לנצח, אבל אני לא מאפשר לצרכן לhattakdm ואז איבדו את המקבילות בין הצרכנים ליצרנים.

זאת לא בעיה של נוכנות, כי מן המשתמע שהסምפור כן מבטיח שא יהיו דילוקים וmbtih שיש מניעה הדדיות, זה

נשים לב לנקודת חשובה: זהו כלי לתזמון, אנחנו משתמשים בו רק בנקודת הקритיות, כי אנחנו לא רוצים
שהזה יפגע בריצה של הקוד.

מה הבעיה כאן?

הבעיה היא לא בהכרח קשורה לזה שאין דಡוקים. הבעיה כאן היא שהסמן מומש בצורה כזאת שהוא יכול לפחות במקבילות. שכן המימוש של הסמן מוגבל רק לתוך אחד להיכנס בכלל לאחת מן הפעולות של הסמן, וברגע שתתלהר אחד יבצע UP ל-S2, הוא לא יעשה UP ל-S1. מאוחר זהו סמן בינהר ומאחר שיש כאן מרוחץ זמן שלא חייב להיות קצר בכלל, מערכת הפעלה יכולה להתעורר. עצם זה שסמן בינהר הוא חסר זיכרון (לא זוכר כמה המתינו) והמרוחץ בין הרגע שבו תתלהר אחד ביצוע (S2)UP לבין הרגע שתתלהר שייתעורר יבצע (S1)up יכול להרום את כל המקבילות.

try #2 for Producers-Consumers

producer:

```
while(TRUE)
produce_item(&item)
down(&empty)
down(&mutex)
insert_item(item)
up(&mutex)
up(&full)
```

consumer :

```
while(TRUE)
down(&full)
down(&mutex)
remove_item(&item)
up(&mutex)
up(&empty)
consume_item(item)
```

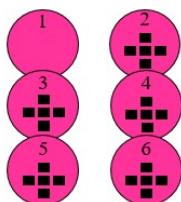
down(S):

```
down(S1)
S.value--
if(S.value < 0)
up(S1)
down(S2)
up(S1)
```

up(S):

```
down(S1)
S.value++
if(S.value ≤ 0)
up(S2)
else
up(S1)
```

producers



bounded-buffer



consumer

Problematic Scenario:

Consider a bounded-buffer of 5 slots.

Consider 6 producers P1, P2, P3, P4, P5, P6, with five items each.

Consider a single consumer.

Initially: mutex = 1, empty.value = N, full.value = 0

P1 inserted all its items
→ empty.value = 0, full.value = N

try #2 for Producers-Consumers

producer:

```
while(TRUE)
produce_item(&item)
down(&empty)
down(&mutex)
insert_item(item)
up(&mutex)
up(&full)
```

consumer :

```
while(TRUE)
down(&full)
down(&mutex)
remove_item(&item)
up(&mutex)
up(&empty)
consume_item(item)
```

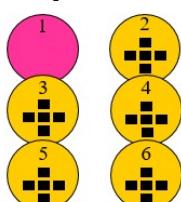
down(S):

```
down(S1)
S.value--
if(S.value < 0)
up(S1)
down(S2)
up(S1)
```

up(S):

```
down(S1)
S.value++
if(S.value ≤ 0)
up(S2)
else
up(S1)
```

producers



bounded-buffer



consumer

Problematic Scenario:

Consider a bounded-buffer of 5 slots.

Consider 6 producers P1, P2, P3, P4, P5, P6, with five items each.

Consider a single consumer.

Initially: mutex = 1, empty.value = N, full.value = 0

P1 inserted all its items
→ empty.value = 0, full.value = N
P2, P3, P4, P5, P6 tried to insert their items, and are **BLOCKED**
→ empty.value = -5

try #2 for Producers-Consumers

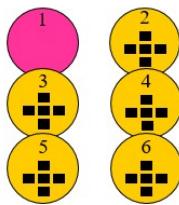
```
producer:
while(TRUE)
produce_item(&item)
down(&empty)
down(&mutex)
insert_item(item)
up(&mutex)
up(&full)
```

```
consumer :
while(TRUE)
down(&full)
down(&mutex)
remove_item(&item)
up(&mutex)
up(&empty)
consume_item(item)
```

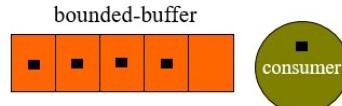
```
down(S):
down(S1)
S.value--
if(S.value < 0)
up(S1)
down(S2)
up(S1)
```

```
up(S):
down(S1)
S.value++
if(S.value ≤ 0)
up(S2)
else
up(S1)
```

producers



bounded-buffer



Problematic Scenario:

Consider a bounded-buffer of 5 slots.

Consider 6 producers P1, P2, P3, P4, P5, P6, with five items each.

Consider a single consumer.

Initially: mutex = 1, empty.value = N, full.value = 0

P1 inserted all its items

→ empty.value = 0, full.value = N

P2, P3, P4, P5, P6 tried to insert their items, and are **BLOCKED**

→ empty.value = -5

Consumer removes one item

→ Consumer executes **up(empty)**

→ since **(empty.value≤0)** holds, Consumer up(empty.S2) (this would wake-up one of the blocked producers), but **does not up(empty.S1)**

→ empty.value = -4

try #2 for Producers-Consumers

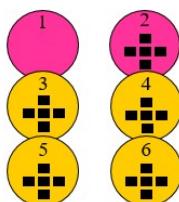
```
producer:
while(TRUE)
produce_item(&item)
down(&empty)
down(&mutex)
insert_item(item)
up(&mutex)
up(&full)
```

```
consumer :
while(TRUE)
down(&full)
down(&mutex)
remove_item(&item)
up(&mutex)
up(&empty)
consume_item(item)
```

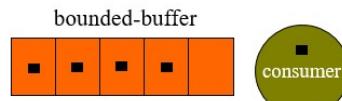
```
down(S):
down(S1)
S.value--
if(S.value < 0)
up(S1)
down(S2)
up(S1)
```

```
up(S):
down(S1)
S.value++
if(S.value ≤ 0)
up(S2)
else
up(S1)
```

producers



bounded-buffer



Problematic Scenario:

Consider a bounded-buffer of 5 slots.

Consider 6 producers P1, P2, P3, P4, P5, P6, with five items each.

Consider a single consumer.

Initially: mutex = 1, empty.value = N, full.value = 0

P1 inserted all its items

→ empty.value = 0, full.value = N

P2, P3, P4, P5, P6 tried to insert their items, and are **BLOCKED**

→ empty.value = -5

Consumer removes one item

→ Consumer executes **up(empty)**

→ since **(empty.value≤0)** holds, Consumer up(empty.S2) (this would wake-up one of the blocked producers), but **does not up(empty.S1)**

→ empty.value = -4

→ **P2 wakes up**

→ empty.S1 = 0



try #2 for Producers-Consumers

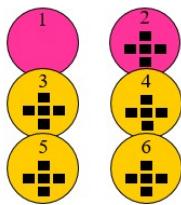
```
producer:
while(TRUE)
produce_item(&item)
down(&empty)
down(&mutex)
insert_item(item)
up(&mutex)
up(&full)
```

```
consumer :
while(TRUE)
down(&full)
down(&mutex)
remove_item(&item)
up(&mutex)
up(&empty)
consume_item(item)
```

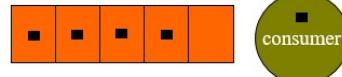
```
down(S):
down(S1)
S.value--
if(S.value < 0)
up(S1)
down(S2)
up(S1)
```

```
up(S):
down(S1)
S.value++
if(S.value ≤ 0)
up(S2)
else
up(S1)
```

producers



bounded-buffer



Problematic Scenario:

Consider a bounded-buffer of 5 slots.

Consider 6 producers P1, P2, P3, P4, P5, P6, with five items each.

Consider a single consumer.

Initially: mutex = 1, empty.value = N, full.value = 0

P1 inserted all its items

→ empty.value = 0, full.value = N

P2, P3, P4, P5, P6 tried to insert their items, and are **BLOCKED**

→ empty.value = -5

Consumer removes one item

→ Consumer executes **up(empty)**

→ since **(empty.value≤0)** holds, Consumer up(empty.S2) (this would wake-up one of the blocked producers), but **does not up(empty.S1)**

→ empty.value = -4

→ **P2 wakes up**

→ empty.S1 = 0

Consumer removes another item

→ Consumer executes **up(empty)**

try #2 for Producers-Consumers

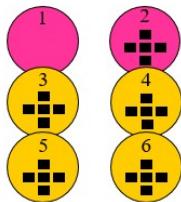
```
producer:
while(TRUE)
produce_item(&item)
down(&empty)
down(&mutex)
insert_item(item)
up(&mutex)
up(&full)
```

```
consumer :
while(TRUE)
down(&full)
down(&mutex)
remove_item(&item)
up(&mutex)
up(&empty)
consume_item(item)
```

```
down(S):
down(S1)
S.value--
if(S.value < 0)
up(S1)
down(S2)
up(S1)
```

```
up(S):
down(S1)
S.value++
if(S.value ≤ 0)
up(S2)
else
up(S1)
```

producers



bounded-buffer



Problematic Scenario:

Consider a bounded-buffer of 5 slots.

Consider 6 producers P1, P2, P3, P4, P5, P6, with five items each.

Consider a single consumer.

Initially: mutex = 1, empty.value = N, full.value = 0

P1 inserted all its items

→ empty.value = 0, full.value = N

P2, P3, P4, P5, P6 tried to insert their items, and are **BLOCKED**

→ empty.value = -5

Consumer removes one item

→ Consumer executes **up(empty)**

→ since **(empty.value≤0)** holds, Consumer up(empty.S2) (this would wake-up one of the blocked producers), but **does not up(empty.S1)**

→ empty.value = -4

→ **P2 wakes up**

→ empty.S1 = 0

Consumer removes another item

→ Consumer executes **up(empty)**

→ Consumer executes **down(empty.S1)** and is **BLOCKED**, since empty.S1 = 0

try #2 for Producers-Consumers

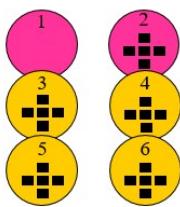
```
producer:
while(TRUE)
    produce_item(&item)
    down(&empty)
    down(&mutex)
    insert_item(item)
    up(&mutex)
    up(&full)
```

```
consumer :  
while(TRUE)  
    down(&full)  
    down(&mutex)  
    remove_item(&item)  
    up(&mutex)  
    up(&empty)  
    consume_item(item)
```

```
down(S):
    down(S1)
    S.value--
    if(S.value < 0)
        up(S1)
        down(S2)
    up(S1)
```

```
up(S):
    down(S1)
    S.value++
    if(S.value ≤ 0)
        up(S2)
    else
        up(S1)
```

producers



bounded-buffer



Problematic Scenario:

Consider a bounded-buffer of 5 slots.
Consider 6 producers P1, P2, P3, P4, P5, P6, with five items each.
Consider a single consumer.
Initially: mutex = 1, empty.value = N, full.value = 0

- empty.value = 0, full.value = N
- P2, P3, P4, P5, P6 tried to insert their items, and are **BLOCKED**
- empty.value = -5
- Consumer removes one item
 - Consumer executes up(empty)
 - since **(empty.value≤0) holds**, Consumer up(empty.S2) (this would wake-up one of the blocked producers), but **does not up(empty.S1)**
 - empty.value = -4
 - **P2 wakes up**
 - empty.S1 = 0
- Consumer removes another item
 - Consumer executes up(empty)
 - Consumer executes down(empty.S1) and is **BLOCKED**, since empty.S1 = 0

Now, only P2 would up(S1) when P2 inserts its item into the buffer.
So, Consumer must wait until P2 inserts its item, and cannot consume the items in the buffer.

Counting semaphore by binary semaphores – try #3

P.A. Kearns, 1988

Initially: binary semaphore $S1 \leftarrow 1$, $S2 \leftarrow 0$, $S.value \leftarrow n$, $wake \leftarrow 0$

down(S):

→ down(S1)
→ *S.value--*
→ if(*S.value < 0*)
→ up(S1)
→ down(S2)
→ down(S1)
→ *S.wake--*
→ if(*S.wake > 0*)
→ up(S2)
→ up(S1)

```

sequenceDiagram
    participant S
    participant S2
    S->>S2: down(S1)
    activate S2
    S2->>S: S.value++
    deactivate S2
    S->>S2: if(S.value <= 0) S.wake++
    activate S2
    S2->>S: up(S2)
    S->>S2: up(S1)

```

'wake' variable
remembers number of
wakeup, and thus let
them not to get lost

Is this code correct ?

NO. Note that each up(S) both wake-up somebody, and increments wake.

Let's see a problematic scenario.

ניסיון נוסף - שנה אחרי הניסיון הקודם:
בנוסף ל-2 סמפורים ולערך יש גם את ע

wake:

מיצג את כמות הפעמים שתהיליך בא להגדיל ב-1 וgilah שיש מישחו שאפשר להעיר.

P1:

ועשו chosop. הביטוי בתנאי הוא שקר ולכן רק מבצע (1s) up.

P2:

אם עושה `chopd` הולך לישון על `down(S2)`

P3:

הסיפור שחוור על עצמו:
השורות 1-2 - בעייתית ():

P1:
עושה UP.
נניח שP4 גם עושה UP. הוא מעדכן את wake להיות 2 ואז מבצע (s)down ומס'ים.

P2:
מקבל זמן ריצה ועושה (s)down ואז S2 מתעדכן להיות 0.
הוא מנסה לחת בלהדיות על .S1 (wake(s3))

P3:
אם מנסה לעשות (S2)down ואז הוא נכנס למצב block על S2.

P2:
מקבל זמן ריצה וממשיר.
הוא עושה (s)up ואז משחרר את P3.

הניסיונו הזה הוא ניסיון לא לפספס wakeup.
הweeneyון לא עובד

Counting semaphore by binary semaphores – try #3

P.A. Kearns, 1988

Initially: binary semaphore S1 $\leftarrow 1$, S2 $\leftarrow 0$, S.value $\leftarrow n$, wake $\leftarrow 0$

<u>down(S):</u>
down(S1)
S.value--
if(S.value < 0)
up(S1)
down(S2)
down(S1)
S.wake--
if(S.wake > 0)
up(S2)
up(S1)

<u>up(S):</u>
down(S1)
S.value++
if(S.value ≤ 0)
S.wake++
up(S2)
up(S1)

<u>Problematic scenario:</u>
Assume semaphore state: S.value = 1
P ₀ , ..., P ₇ perform down(S)
→ P ₀ goes through, P ₁ , ..., P ₇ are BLOCKED
→ S.value = -7
P ₈ , ..., P ₁₁ perform up(S) and wakeup P ₁ , ... P ₄
→ S.wake = 4
→ S.value = -3
P ₁ performs: S.wake--, up(S2) → wakeup P ₅
P ₂ performs: S.wake--, up(S2) → wakeup P ₆
P ₃ performs: S.wake--, up(S2) → wakeup P ₇

4 up(S) operations have released 7 down(S) operations...

P0:
מגיע ונוועל את שאר התהליכיים.
ניקח את הסטטוס בזיכרון בו S.value = 1.
הגיעו 8 תהליכיים לביצוע (S)down.

הערך של wake משתנה ל-4 ו-S.value מינוס 3.
עכשו מגיעים עוד 4 תהליכיים כדי לבצע פעולה ק�לה.
כרגע 7.value =

כתחזאה מכך תהליכי 1 עד 4 מתעוררים.

- P1: מבצע wake-- ומעיר את כל מי שישן על S2 - העיר את P5.
- P2: מקבל זמן ריצה ועשה אותו הדבר ומעיר את P6.
- P3: אותו דבר ומuir את P7.

השלב הזה בדייתי כי השתחררו 4 permits אבל התעוורנו 7 תהליכי זה בכלל לא מקיים את ההגדרה של סמפור.

Counting semaphore by binary semaphores

Hemmendinger, 1989

Initially: binary semaphore $S1 \leftarrow 1$, $S2 \leftarrow 0$, $S.value \leftarrow n$, $wake \leftarrow 0$

```
down(S):
    down(S1)
    S.value--
    if(S.value < 0)
        up(S1)
        down(S2)
        down(S1)
        S.wake--
        if(S.wake > 0)
            up(S2)
            up(S1)
```

```
up(S):
    down(S1)
    S.value++
    if(S.value <= 0)
        S.wake++
    if (S.wake == 1)
        up(S2)
        up(S1)
```

only if $wake == 1$, we
wake-up BLOCKED
process

Is this code correct ?

Yes. Let's see the previous
problematic scenario solved...

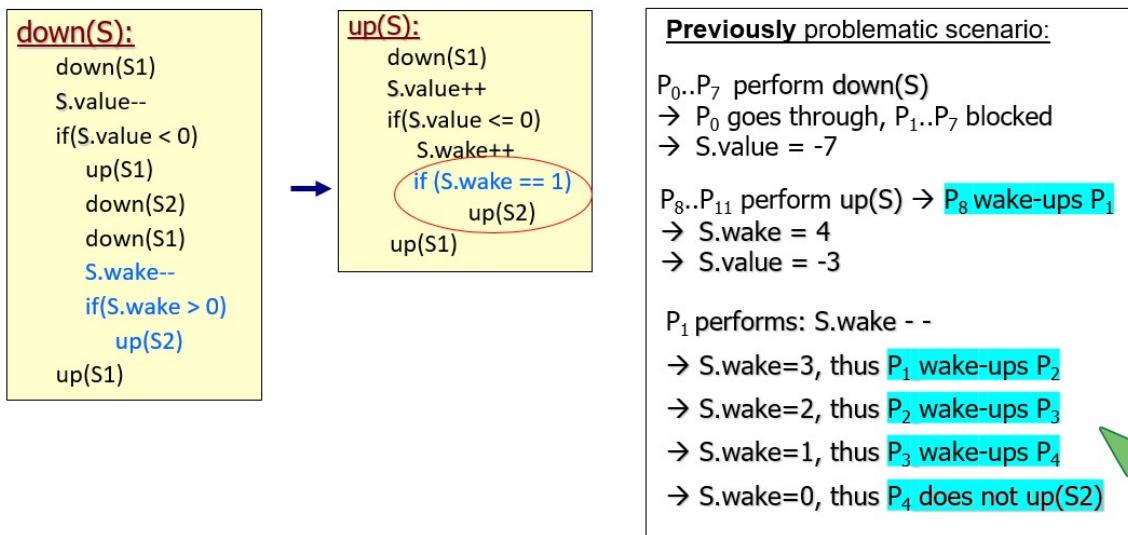
הניסיון הראשון שבאמת עובד.

אני משאיר את הקוד כמו שהוא, אבל הפעם הוא לא מעיר ככה סתם אלא רק כאשר $S.wake == 1$.

Counting semaphore by binary semaphores

Hemmendinger, 1989

Initially: binary semaphore $S1 \leftarrow 1, S2 \leftarrow 0, S.value \leftarrow n, wake \leftarrow 0$



ניקח את אותה הדוגמה ממקודם:

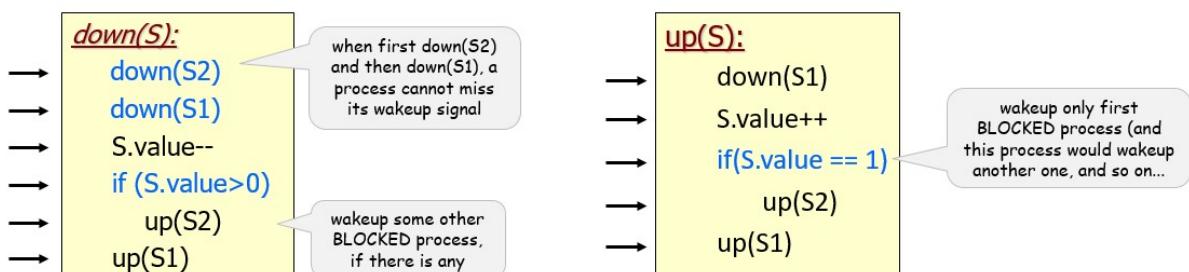
ההבדל הוא שעכשיו יש שרשרת של wakeup, כלומר יש סדר האיגוני על ה-wakeups.

אבל בשלב זהה אפשר לראות שהוא עובד אבל מסובך מדי!

Counting semaphore by binary semaphores

Barz, 1983

Initially: binary semaphore $S1 \leftarrow 1, S2 \leftarrow min\{1, S.value\}, S.value \leftarrow n$



This works, is simpler, and was published earlier(!)...

What if we switch order of downs in $down(S)$? Deadlock...

אלגוריתם שעבוד מ-1983.

יש סמפור $S1$ - $S2$.

הסמפור $S2$ מקבל את המינימום מבין 1 או $S.value$ - סמפור האיגוני.

נניח בשביל הדוגמה ש- $S2$ מקבל את הערך 1.

P1:

מבצע (S2)down ו-(-S1)down ומצליח. כתוצאה לכך הוא מצליח להיכנס לקוד תוך כדי שהוא משחרר את S1.

P2:

אם נכנס ל-(S2)down ומיד נכנס למצב blocked על S2
(למעשה בשלב זהה כל התהליכים שיבואו אחריו S1 ישנו על S2).

P1:

יעשה כן ומשחרר את P2 ומעלה את הערך של S2 ב-1.

P2:

יכול להיכנס פנימה.

למה לפתח מהهو שכבר פתרו?

ההיסטוריה מספרת שהם לא ידעו האחד על המימוש השני.

Semaphores: outline

- Semaphores Introduction
- Producer-Consumer problem
- Counting semaphore by Binary semaphores
 - Event counters
- Message passing synchronization

Event Counters

Each process may **wait for its own specific value**, while semaphore has only a single blocking variable.

<i>Read(E):</i>	<i>return the current value of E</i>
<i>Advance(E):</i>	<i>increment E by 1, wakeup relevant BLOCKED processes</i>
<i>Await(E, v):</i>	<i>BLOCKED until $E \geq v$</i>

מנוי אירועים

כל תהליך יכול להמתין לערך מסוים שלו, בעוד שבסימפור יש רק משתנה חסימה יחיד.

א. **Read (E)**: מוחזיר את הערך הנוכחי של E.

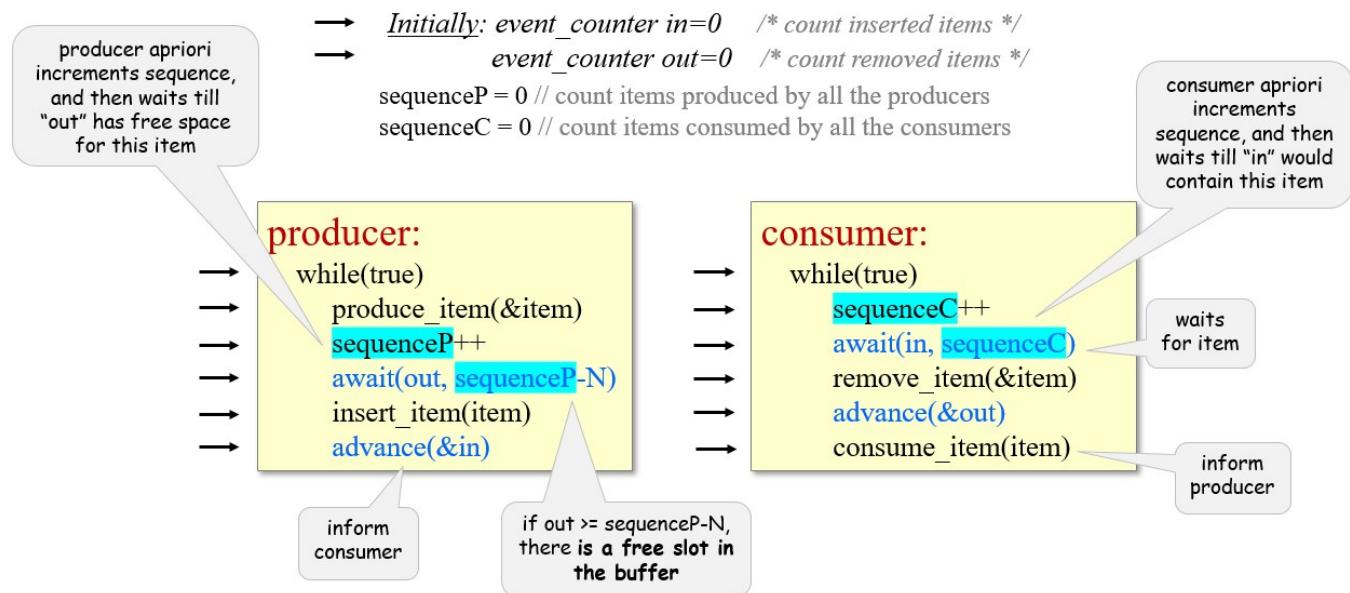
ב. **Advance (E)**: מגדיל את E ב-1, מעיר את התהליכי הרלוונטיים שנחסו.

ג. **(v, E) Await (E, v)**: נחסם עד ש- $E \geq v$.

אנחנו מיעוניים לראות סימפור מסוים, שהוא אם מונה, אך כאן אנחנו מגיעים למצב שבו אנחנו נחסמים על אירוע מסוים. ברגע שהאירוע מתתרחש, כל מי שמתמן לארוע מתעורר.

הערך כאן רק מגדיל (בדוגמה שלנו). ישנה פונקציה של המתנה (`await`) עד שהערך של מונה האירועים מתעדכן.

Single producer-consumer with Event Counters



פתרון בעית יצור-צרכן עם מוני אירועים

אתחול:

- מונה האירועים `0 = in` מבטא את מספר הפריטים שהוכנסו.
- מונה האירועים `0 = out` מבטא את מספר הפריטים שהוסרו.
- `0 = sequenceP` מבטא את מספר הפריטים שהופקו על ידי כל היוצרים
- `0 = sequenceC` // מספר הפריטים שנצרכו על ידי כל הלקוחים

יצר

```

while(true)
  produce_item(&item)
  sequenceP++
  
```

היצר מקדמי בראש הרצף, אז ממשין עד ש-"`out`" יהיה עם מקום פניו לפרט זה

```
await(out, sequenceP-N)
```

אם `out >= sequenceP-N`, יש מקום פניו בבאפר

```

insert_item(item)
advance(&in)      מודיע ללקוח //
  
```

צרכן:

```

while(true)
  sequenceC++
  
```

הצרכן מקדם מראש את הרצף, ואז ממתיין עד ש-"`ho`" יכול את הפריט הזה

```
await(in, sequenceC) // ממתין לפרט  
remove_item(&item)  
advance(&out) // מודיעט ליצahn  
consume_item(item)
```

דוגמיה:

יש לנו בעיה של יצרנים וצרכנים. יש לנו באפר בגודל 5. יש לנו שני מונחים: `in` ו-`out` (בנייה של `empty` ו-`full` ו-`out`) (שaan הם רק מגדילים). יש גם שני משתנים מקומיים - אחד לצרכנים ואחד ליוצרים.

היצאן מייצר פריט, מקדם את `sequenceP` וממתין על `out` כל עוד יש מקום פניו באפר. לאחר מכן, הוא מכניס ומקדם את `in`.

הצרכן, בתחילת, מקדם את המשתנה שלו וממתין ש`in` קיבל את הערך זהה. לאחר מכן, הוא צורק את הפריט ומקדם את `out`.

הרעיון הוא שאנו כאן רק מקדמים את `in` ו-`out`.

אם יש יתרון כאן לטובות מוני האירועים? לא מהו מحوותי מעבר בדרך הקודמת.

Semaphores: outline

- ❑ Semaphores Introduction
- ❑ Producer-Consumer problem
- ❑ Counting semaphore by Binary semaphores
- ❑ Event counters
- ❑ Message passing synchronization

דבר אחרון:

סנכרון בראשת.

ראינו כבר מהו אחר - זה שיש גישה לזכרון.

עכשו אנחנו רוצים לתזמן תהליכי מבלי להשתמש בזכרון משותף.

למה?

זיכרון משותף הוא זיכרון ש-2 תהליכי או יותר משתמשים בו. אנחנו רוצים לפתור את בעיית צרכן-יצאןvr שאי

זיכרון משותף בינם.

זה מאד מזכיר את שרת-לקוח (למרות שלקווח-שרת יכול להשתמש בזכרון משותף, בדרך כלל הוא לא)

Semaphores and Event Counters require access to common shared memory.

In distributed systems this is inapplicable.

*For such cases, we may use
Message Passing.*

Message Passing: no shared memory

*send (destination, &message)
receive (source, &message) /* block while waiting */*

Implementation issues:

- acknowledgements (messages may be lost)
- message sequence numbers required to avoid message duplication
- unique process addresses (domains)
- authentication (validate sender's identity)

we ignore all
these by now

שיטת שליחת הודעות במערכות מבוארות

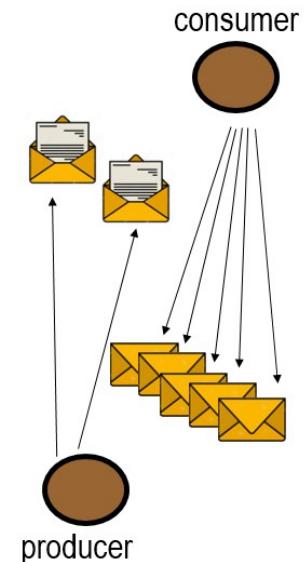
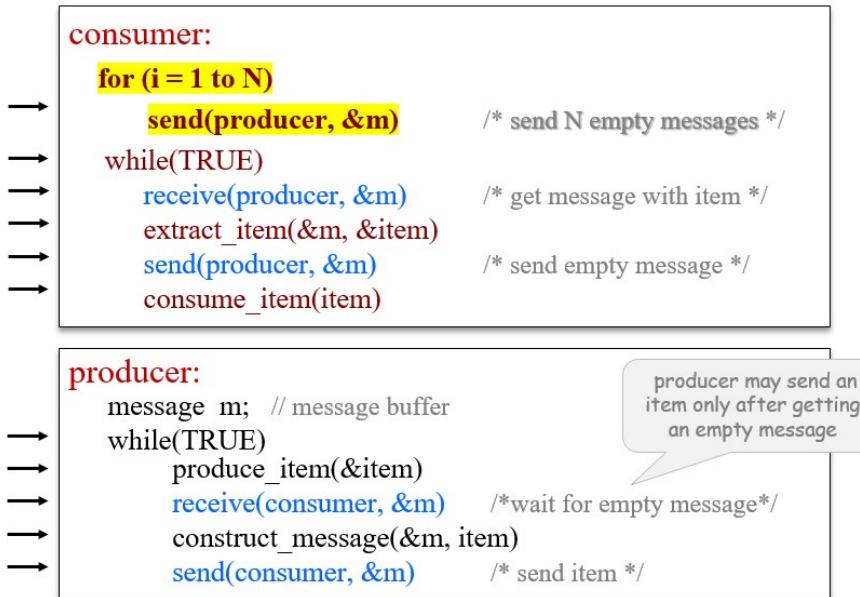
ספורים ומוני איריעים מחיבבים גישה לזכרון משותף. אך במערכות מבוארות, זה לא מתאים. במקרים כאלה, אנו משתמשים בשיטת שליחת הודעות, שבה אין זיכרון משותף.

בשיטת זו, ישנו שתי פעולות פרימיטיביות: שליחה (send) וקבלת (receive). הפעולה send שולחת הודעה לעד מסוים, והפעולה receive מקבלת הודעה ממוקור מסוים. אם אין הודעה זמינה, המქבל יכול להיחס עד שההודעה תגיע, או לחזור מיד עם קוד שגיאה.

השיטה מציגה מספר בעיות ונוסאים לעיצוב, שלא מתכווררים עם סמפורים או מוניטורים, במיעודם התהילכים המתקשרים נמצאים במקומות שונים מחוברות לרשט. לדוגמה, ההודעות יכולות להיבז ברשט. כדי להגן על הודעות שאבדו, השולח והמקבל יכולים להסכים שמיד כשהודעה התקבלה, המקבל ישלח הודעה אישור מיוחדת. אם השולח לא קיבל את האישור תוך פרק זמן מסוים, הוא שולח מחדש הודעה את ההודעה.

שיטת שליחת ההודעות משמשת באופן נרחב במערכות תכונות מקביליות. לדוגמה, שימוש MPI (Message-Passing Interface) הוא מערכת שליחת הודעות מוכרת, המשמשת באופן נרחב בחישובים מדעיים.

Producer-Consumer using message passing



בבית הצרך-יצן ניתן לפתור באמצעות שליחת הודעות ולא זיכרון משותף. הצרך מתחילה על ידי שליחת N הודעות ריקות ליצן. כאשר ליצן יש צורך לתקן, הוא לוקח הודעה ריקה ושולח בהזרה אחת מלאה. בדרך זו, מספר ההודעות במערכת נשאר קבוע בזמן, כך שהן יכולות להישמר בكمות זיכרון מסוימת שנודעה מראש.

אם היצן עובד מהר יותר מהצרך, כל ההודעות יהיו מלאות, ממתינות לצרך; היצן יהיה חסום, וממתין להודעה ריקה לחזור. אם הצרך עובד מהר יותר, אז ההפר קורה: כל ההודעות יהיו ריקות ממתינות ליצן למלא אותן; היצן יהיה חסום, עד שתתקבל הודעה מלאה.

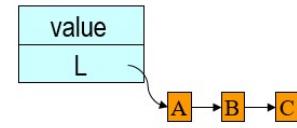
Should semaphore be fair ?

An unfair semaphore does not guarantee that the wakeup order of processes is similar to their falling asleep order.

Because it may conflict with OS scheduler.

NOT SURE.
Why ?

Semaphore struct:



Should we wakeup the process A first ?

Should semaphore be fair ?

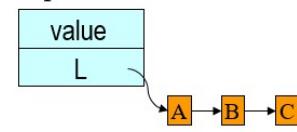
An unfair semaphore does not guarantee that the wakeup order of processes is similar to their falling asleep order.

A's priority=8

B's priority=12

C's priority=4

Semaphore struct:



According to OS scheduler, process C should run first since its (Unix-style) priority is highest !

Fairness of semaphores

If we do prefer fair semaphore, there are several levels of fairness:

- **Weak fairness:** if process A is running on semaphore S, and when A exits S there are $P_1 \dots P_N$ processes BLOCKED on S, at least one of $P_1 \dots P_N$ should enter S before A re-enters S
- **Strong fairness:** if process A is blocked on semaphore S, there is upper bound on number of processes that may re-enter S before A
- **Strongest fairness:** FIFO

אם סמפור חייב להיות הוגן?

במהלך הרצאה שלנו, שאלנו את השאלה הבאה: האם סמפור חייב להיות הוגן? כאשר אנו מדברים על הוגנות בהקשר של סנכרונייזציה, אנו מתיחסים לסדר בו התהליכים מתעוררים מהחסימה שלהם. סמפור לא הוגן לא מבטיח שהסדר של התעוררות התהליכים יהיה דומה בסדר שבו הם התחרסמו.

לדוגמא, נניח שיש לנו שלושה תהליכיים: B, A ו-C, עם עדיפות של 8, 12 ו-4 בהתאם. לפי מזמן מערכת הפעלה, התהליך C צריך לרוץ ראשון מכיוון שלו יש את העדיפויות הגבוהה ביותר.

אם אנחנו מעדיפים סמפור הוגן, ישנו מספר רמות של הוגנות:

א. **הוגנות חלשה:** אם התהליך A רץ על הסמפור S, וכאשר A יוצא-S ישנים תהליכיים P1 עד PN שנחסמו על S. לפחות אחד מהתהליכיים P1 עד PN צריך להיכנס ל-S לפני ש-A יכנס שוב ל-S.

ב. **הוגנות חזקה:** אם התהליך A נחסם על הסמפור S, יש גבול עליון למספר התהליכים שיכולים להיכנס שוב ל-S לפני A.

ג. **הוגנות החזקה ביותר: FIFO.**

חשוב לציין שאף אחת מהשיטות אינה מבטיחה שהתהליכים לא ירבעו, כיון שניתן להיות שהסמפור לא מאפשר אף תהליך להיכנס - אז אף תהליך לא יכול לעקוף את האחרים. זהה המקרה המנון שגורם לחוסר הקשר בין הוגנות לחופש מרעבון.

הערה: אצלנו מערכת הפעלה חייבת בזמן סופי לתת זמן ריצה לכל תהליך.

Monitors and MCS algorithm



© 2022 מערכות הפעלה

Operating Systems

Lecture 6 – Synchronization: Monitors and MCS algorithm

Dr. Marina Kogan-Sadetsky

Semaphores: outline

- Monitors
- The Mellor-Crummey and Scott (MCS) algorithm

Monitors - higher-level synchronization

(Hoare, Hansen, 1974)

```
monitor <monitor-name>
variables declaration
condition variables declaration

  procedure entry P1 ...
    begin ... end

  procedure entry P2 ...
    begin ... end

  .
  .

  procedure entry Pn ...
    begin ... end

  initialization code

end monitor
```

A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.

The monitor's internal data structures may be accessed only from procedures declared in the monitor.



Tony Hoare



Charles Hansen

Monitors - higher-level synchronization

(Hoare, Hansen, 1974)

```
monitor <monitor-name>
variables declaration
condition variables declaration

  procedure entry P1 ...
    begin ... end

  procedure entry P2 ...
    begin ... end

  .
  .

  procedure entry Pn ...
    begin ... end

  initialization code

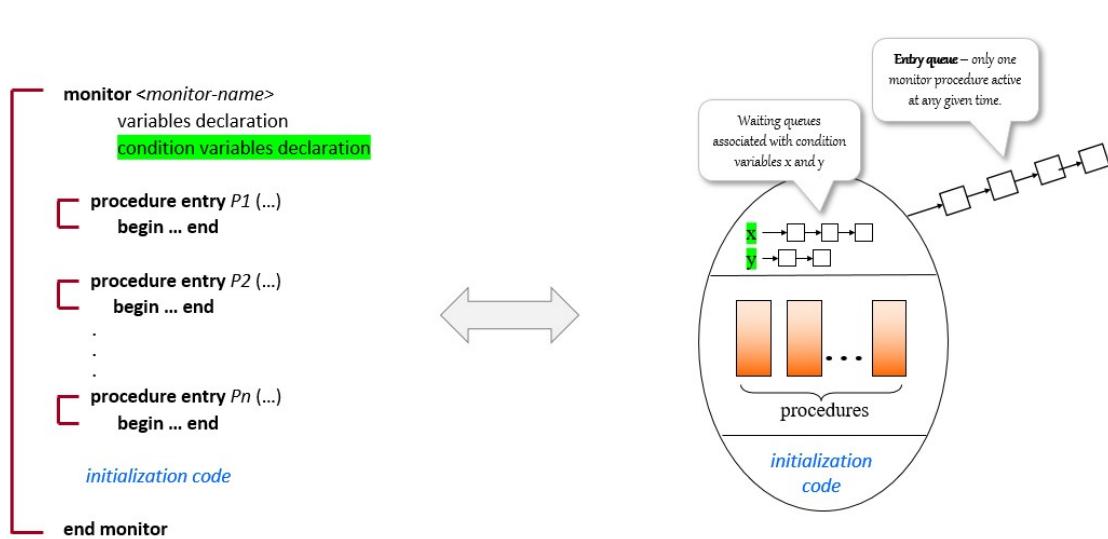
end monitor
```

- Monitor is programming-language structure, generated by compiler
- Only one process is active in a monitor at any given time – mutual exclusion
- Monitors support:
 - condition variables
 - wait and signal operations
- After sending a signal, the thread has two options:
 - keep monitor locked and proceed
 - immediately exit monitor operation, waked-up thread is the only thread that may proceed in the monitor ([Hoare semantics](#))
- Monitor disadvantages:
 - May be less efficient than lower-level synchronization
 - Available only from some programming languages

a signal has no effect if there are no waiting threads!

Monitors - higher-level synchronization

(Hoare, Hansen, 1974)



מונייטרים - כלים סינכרוניים מתקדמים

מונייטור הוא מבנה שנוצר על ידי הקומפיילר, שמאפשר לתהיליך אחד בלבד להיות פעיל בו בכל זמן נתון. זהו כלי שנוצר על ידי Hoare ו-Hansen בשנת 1974, והוא מאפשר לנו להגדיר משתנים, כולל משתני תנאי, ולבצע פעולה של *wait* ו-*signal*.

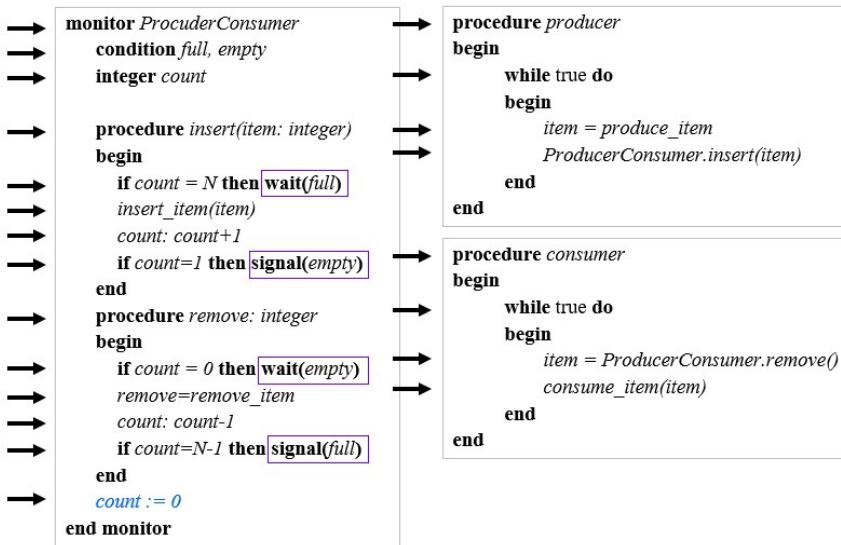
המבנה של מונייטור מורכב ממספר פרוצדורות, משתנים ונתונים שקובצים ייחודי למודול או חבילה מיוחדת. המשתנים ונתוני המונייטור נגשים רק מתוך הפרוצדורות שהוגדרו במונייטור. כל תהיליך יכול לקרוא לפרוצדורות של המונייטור בכל זמן שהוא רוצה, אך הוא לא יכולugasht夷ות לנתוני המונייטור הפנימיים של המונייטור מתוך פרוצדורות שהוגדרו מוחוץ למונייטור.

המונייטורים מאפשרים גם תמייה במשתני תנאי ובפעולות *wait* ו-*signal*. ברגע שתהיליך שולח *signal*, יש לו שתי אפשרויות: להשאיר את המונייטור נועל ולהמשיך, או לצאת מהמונייטור באופן מיידי. אם הוא בוחר באפשרות השנייה, התהיליך שהתעורר מה-*signal* הוא התהיליך היחיד שיכל להתקדם במונייטור. זו הסמנטיקה שהציג Hoare.

אף על פי שהמונייטורים מאפשרים דרך קלה להשיא מניעה הדדית, הם לא מספיקים לבדם. אנחנו צריכים גם דרך לחסום תהילכים כאשר הם לא יכולים להמשיך. למשל, בעיה של צרכן-צרגן, קל לשים את כל הבדיקות של מאגר מלא ומ Lager ריק בפרוצדורות של המונייטור, אבל איך ניתן לבצע כאשר הוא מוצא את המאגר מלא?

הפתרון הוא בעזרת משתני תנאי, יחד עם שתי פעולה עליהם *wait* ו-*signal*. כאשר פרוצדורת מונייטור מגלה שהיא לא יכולה להמשיך (למשל, המפיק מוצא את המאגר מלא), היא מבצעת *wait* על משתנה תנאי מסוים, אם נניח, *full*. פעולה זו אורמת לתהיליך הקורא להיחסם. זה גם מאפשר לתהיליך אחר שהיה מונע בעבר מלהיכנס למונייטור להיכנס עכשו.

Bounded Buffer Producer-Consumer with Monitors



Suppose full buffer. Pi is waiting on full condition variable. A consumer enters the monitor, sends a signal and released Pi. Some other producer Pj enters the monitor and fills the single empty slot of the buffer. When Pi proceeds, Pi would add additional item to already full buffer. **Buffer corruption**.

Suppose full buffer. 3 producers are waiting on full condition variable. 3 consumers enter the monitor one after the other, but only the first consumer send a signal (since count=N-1 holds for the first consumer). Therefore only one producer got a signal, and all others didn't. This is **deadlock**.

Any problem with this code? Yes, several.

נשים לב כי זאת לא הארסה התקינה - אנחנו נראה עוד אחת.
נחוור לבעית צרכן-יצרן:

יש לנו 2 משתני תנאים, כאשר האחד מבטא את המצב שבו הרשימה מלאה והשני מבטא את המצב שהרשימה ריקה. כמו כן יש משתנה גליל שסופר את מספר התאים המלאים "בבאפר".
יש 2 פרוצדורות:

insert:

מקבלת פריט. אם הרשימה מלאה ממתינים לסיגנל ל-full. מכניסים את פריט, מקדמים ב-1 ואם $1 \leq \text{count} < N$ שולחים סיגנל שהרשימה ריקה.

remove:

אם הרשימה ריקה ממתינים לסיג널 על המשתנה empty. מעדכנים את השדה remove מורידים ב-1 את הערך של count

Producer:

לנצח: מייצרת פריט ומנסה להיכנס ל-insert.

consumer

לנצח: מנסה להסיר את הפריט ואז לצרוך אותו.

למה המימוש לא תקין:

א. נניח ויצרן ממתין שהרשימה תתפנה אוז התקבל סיגナル. זה לא מחייב שהיצרן יקבל ישר זמן ריצה ויכול להיות שיצרן אחר יספק להיכנס וימלא. יכול להיות שהיצרן הראשון יתעורר אחרי שיצרן אחר כבר הספיק למלא ואז יתרחש **buffer corruption**.

ב. נניח כי הבאפר מלא. קיימ מצב לפיו יש דדLOCK:
ראשית, 3 יצרנים, P1, P2, P3, מנסים להיכנס לפריט. מאחר והבאפר מלא הם נוכנים למצב המתנה.
לאחר מכן 3 יצרנים מנסים לצורך, כך שאחד נכנס אחריו השני. כאשר הראשון נכנס, הוא מצליח להיכנס ולצרוך מהבאפר. מאחר וכרגע התנאי השני מתקיים, הוא מעיר את אחד התהילכים, נניח בה"כ כי הוא העיר את P1
שהספיק להתעורר אך לא נכנס כי בנסיבות היצרן השני נכנס וכך. מאחר ו-1 לא הצליח להיכנס, התנאי השני

של הצרכן לא מתקיים ולכן הוא לא מעיר את היצרך השני. באותו האופן גם היצרך השלישי לא מתעורר. קיבלנו דדוקן.

ראו לציין שאם הולכים לפי הסמנטיקות של hoare, לפיה יש העברה של המפתח הקוד תקין לאמרי. הכוונה היא לפי כמו שאנחנו מכירים אותו בגיאואה.

Bounded Buffer Producer-Consumer with Monitors

```

→ monitor ProcuderConsumer
→   condition full, empty
→   integer count
→
→   procedure insert(item: integer)
→   begin
→     if count = N then wait(full)
→     insert_item(item)
→     count: count+1
→     if count=1 then signal(empty)
→   end
→
→   procedure remove: integer
→   begin
→     if count = 0 then wait(empty)
→     remove=remove_item
→     count: count-1
→     if count=N-1 then signal(full)
→   end
→
→   count := 0
end monitor

```

```

→ procedure producer
→ begin
→   while true do
→     begin
→       item = produce_item
→       ProducerConsumer.insert(item)
→     end
→   end
→
→ procedure consumer
→ begin
→   while true do
→     begin
→       item = ProducerConsumer.remove()
→       consume_item(item)
→     end
→   end

```

Problem #1: Suppose full buffer. Pi is waiting on full condition variable. A consumer enters the monitor, sends a signal and released Pi. Some other producer Pj enters the monitor and fills the single empty slot of the buffer. When Pi proceeds, Pi would add additional item to already full buffer. **Buffer corruption.**

Problem #2: Suppose full buffer. 3 producers are waiting on full condition variable. 3 consumers enter the monitor one after the other, but only the first consumer send a signal (since count=>N-1 holds only for the first consumer). Therefore, only one producer got a signal, and all others didn't. This is deadlock.

But... this code works if the monitor holds Hoare semantics.

Any problem with this code? Yes, several.

פתרון בעיית צרכן-יצרך בעזרת מוניטורים

בעית הצרכן-יצרך עם מוניטורים מוצגת בצורה מופשטת. יש לנו שני משתנים תנאי, אחד מייצג את המצב שבו הרשימה מלאה והשני מייצג את המצב שבו הרשימה ריקה. יש גם משתנה רגיל שסופר את מספר התאים המלאים בבאפר.

יש לנו שתי פרוצדורות:

.א. insert: מקבלת פריט. אם הרשימה מלאה, ממתינה לסייגל מ-full. מכניסה את הפריט, מגדילה את count ב-1, ואם count שווה ל-1, שולחת סיגנל שההרשימה ריקה.

.ב. remove: אם הרשימה ריקה, ממתינה לסייגל מ-empty. מעדכנת את השדה remove, מפחיתה את count ב-1.

יש לנו גם שתי פרוצדורות נוספות:

.א. producer: בלולאה אינסופית, מייצרת פריט ומנסה להכניס אותו באמצעות insert.

.ב. consumer: בלולאה אינסופית, מנסה להסיר פריט באמצעות remove ואז צורך אותו.

ישנן בעיות מסוימות עם הקוד הזה:

א. בעיה מס' 1: נניח שהבאפר מלא. יצרן מסויים, P_i , ממתין למשתנה התנאי full. יצרן נכנס למוניטור, שולח סיגנל ומשחרר את P_i . יצרן אחר, P_j , נכנס למוניטור וממלא את החלל הריק היחיד בבאפר. כאשר P_i ממשיר, הוא מוסיף פריט נוסף לבאפר שכבר מלא, מה שגורם להשחתת הבאפר.

ב. בעיה מס' 2: נניח שהבאפר מלא. שלושה יצרנים ממתיינים למשתנה התנאי full. שלושה צרכנים נכנסים למוניטור אחד אחרי השני, אך רק היצורן הראשון שולח סיגナル (מאחר שההתנאי $N=1$ count תקף רק לצרכן הראשון). לכן, רק יצרן אחד מקבל סיגナル, וכל השאר לא. זה מוביל לדילוק.

שים לב, זו לא הארסה הנכונה - אנחנו עוד נראה אחת. נחזור לעוית היצורן-יצרן. אם אנחנו הולכים לפי הסמנטיקה של Hoare, שבה יש העברה של המפתח, הקוד הזה הוא תקני לחלוטין. הכוונה היא לפה כמו שאנחנו מכירים אותו ב-Java.

Implementing Monitors from Semaphores – take 1

```
Monitor (c1, c2, ..., ci):
semaphore mutex = 1; // control access to monitor
semaphores c1, c2, ..., c_i = 0; // condition variables
void enter() {
    down(mutex); // mutual exclusion
}
void leave() {
    up(mutex); // allow other processes in
}
void leave_with_signal(c_i) {
    up(c_i); // release the condition variable, mutex not released
}
void wait(c_i) {
    up(mutex); // allow other processes
    down(c_i); // block on the condition variable
}
```

```
Producer:
while(TRUE)
    produce_item(&item) // generate item
    M.enter() // enter monitor
    if(buffer_counter == N) // is there free space in buffer ?
        M.wait(M.full) // wait on full condition variable
    insert_item(item) // insert item into buffer
    buffer_counter++ // increment buffer counter
    if(buffer_counter == 1)
        M.leave_with_signal(M.empty) // signal consumers
    else
        M.leave() // leave monitor
```

```
monitor M(condition variables: full, empty)
buffer_size = N
buffer_counter = 0
```

```
Consumer:
while(TRUE)
    M.enter() // enter monitor
    if(buffer_counter == 0) // are there items in buffer ?
        M.wait(M.empty) // wait on empty condition variable
    item = remove_item() // remove item from buffer
    buffer_counter-- // increment buffer counter
    if(buffer_counter == N-1)
        M.leave_with_signal(M.full) // signal producers
    else
        M.leave() // leave monitor
    consume_item(item) // consume item
```

Any problem with this code?

It may deadlock...

Implementing Monitors from Semaphores – take 1

```
Monitor (c1, c2, ..., ci):
semaphore mutex = 1; // control access to monitor
semaphores c1, c2, ..., c_i = 0; // condition variables
void enter() {
    down(mutex); // mutual exclusion
}
void leave() {
    up(mutex); // allow other processes in
}
void leave_with_signal(c_i) {
    up(c_i); // release the condition variable, mutex not released
}
void wait(c_i) {
    up(mutex); // allow other processes
    down(c_i); // block on the condition variable
}
```

```
Producer:
while(TRUE)
    produce_item(&item) // generate item
    M.enter() // enter monitor
    if(buffer_counter == N) // is there free space in buffer ?
        M.wait(M.full) // wait on full condition variable
    insert_item(item) // insert item into buffer
    buffer_counter++ // increment buffer counter
    if(buffer_counter == 1)
        M.leave_with_signal(M.empty) // signal consumers
    else
        M.leave() // leave monitor
```

```
monitor M(condition variables: full, empty)
buffer_size = N
buffer_counter = 0
```

```
Consumer:
while(TRUE)
    M.enter() // enter monitor
    if(buffer_counter == 0) // are there items in buffer ?
        M.wait(M.empty) // wait on empty condition variable
    item = remove_item() // remove item from buffer
    buffer_counter-- // increment buffer counter
    if(buffer_counter == N-1)
        M.leave_with_signal(M.full) // signal producers
    else
        M.leave() // leave monitor
    consume_item(item) // consume item
```

Producer:
M.enter()
--> M.mutex <- 0
--> buffer_counter <- 1
--> M.leave_with_signal(M.empty)
but no Consumers are sleeping !
--> M.empty <- 1

process has no ability to
check if there are waiting
processes in the monitor

Since M.mutex remains 0, now Monitor is blocked --> **deadlock...**

מימוש מוניטורים בעזרת סטטוסים

נכזה לממש מוניטורים באמצעות סטטוסים. יש לנו סטטוס שנקרא mutex שמשמש לשיליטה בגישה למוניטור, וסטטוסים נוספים שמייצגים משתני תנאי. יש לנו פונקציות של כניסה, יציאה, יציאה עם שליחת סיגナル, והמתנה.

במקרה של היצwan, כאשר הוא מבצע M.enter, הוא מקבל בלעדות על המוניטור.

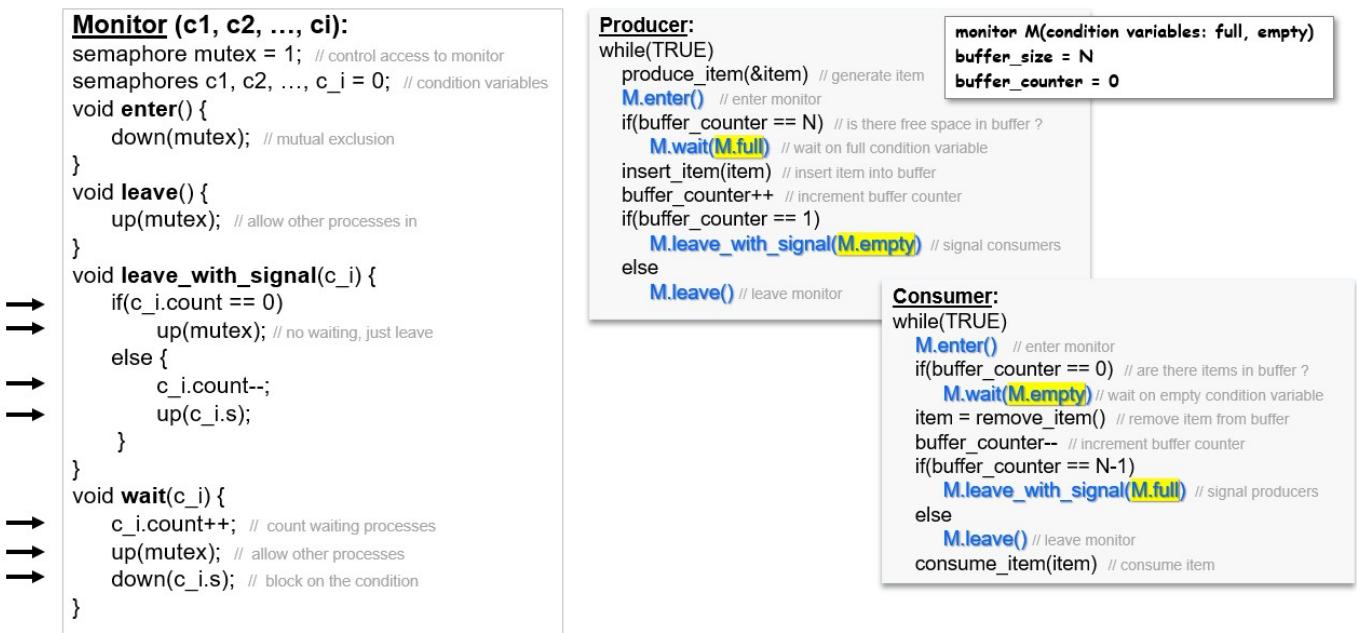
ישנו בעיות עם הקוד הזה:

א. בסמנטיקה של Hoare, אין שחרור של mutex כאשר היצwan יוצא מהמוניטור יחד עם הסיגナル. נניח שהיצwan נכנס כאשר הבאפר ריק. הוא יוצא עם סיגナル, אך אף אחד לא יכול לעבור את המחסום של mutex, וכך

ב. נניח שאין אף תחיל' במחצ' המתנה, חוץ מתחילה אחד שעבר דרך `leave_with_signal`. כתוצאה לכך, המשטנה C מקבל ערך של 1, ובפעם הבאה שתחל' כלשהו ירצה להמתן לאחד ה-C, הוא לא יוכל לעשות זאת, לאחר שהפונקציה `down` לא תחסם אותו.

נשתמש בדוגמת הרצן-יצרן כדי להבין את השימוש. נניח שהיצרן מכניס פריט לבארך כאשר הבארך מלא, הוא ממתיין למשטנה התנאי `full`. אם הבארך ריק, הוא שולח סיגナル לצרכנים. במקרה של הרצן, אם הבארך ריק, הוא ממתיין למשטנה התנאי `empty`. אם הבארך מלא, הוא שולח סיגナル ליוצרים. אך אם אין צרכנים שמתהינים, המוניטור נחכם ומגיבים במצב של דלק.

Implementing Monitors from Semaphores – correct



ונסה שוב למשם מוניטורים באמצעות סמפורים, והפעם נעשה זאת בצורה נכוןה. יש לנו סמפור שנקרא `mutex` שמשמש לשילטה בגישה למוניטור, וסמפורים נוספים שמייצגים משתני תנאי. יש לנו פונקציות של כניסה, יציאה, יציאה עם שליחת סיגナル, והמתנה.

במקרה של היצרן, כאשר הוא מבצע `M.enter`, הוא מקבל בלעדיות על המוניטור. אם הבארך מלא, הוא ממתיין למשטנה התנאי `full`. אם הבארך ריק, הוא שולח סיגナル לצרכנים.

במקרה של הרצן, אם הבארך ריק, הוא ממתיין למשטנה התנאי `empty`. אם הבארך מלא, הוא שולח סיגナル ליוצרים.

השינוי המרכזי בימוש זהה הוא שהסמפור אינו סמפור בינארי. כאשר רוצים לשלוח סיגナル בעת היציאה מהמוניטור, אם המונה של המשטנה התנאי הוא 0, אז אף אחד לא ממתיין ואז משחררים את המוניטור. אחרת, מורידים את המונה ב-1 ומשחררים את המשטנה התנאי.

כאשר הרצן נתקע, הוא מגדיל את המונה. ברגע שמגיבים לשורה של `down`, כל אחד יכול לקחת בלעדיות על המוניטור.

סיכום: המוניטור האמתי תלוי בשפה. הוא יכול להיות ממומש עם סמפורים או לא. זהו כלי שהוא יותר קל לשימוש עבור המשתמש, אך קשה יותר למימוש עבור המתכנת. הסנכרון האולטימטיבי מתרחש באמצעות אוטומטים. אם מיפוי להמתנה ארוכה, מעבירים לשימוש בסמפורים.

MCS Algorithm

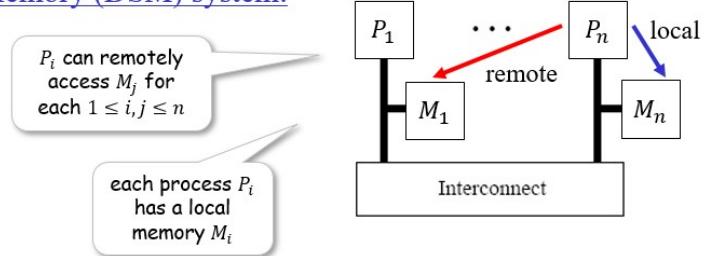
Semaphores: outline

❑ Monitors

❑ The Mellor-Crummey and Scott (MCS) algorithm

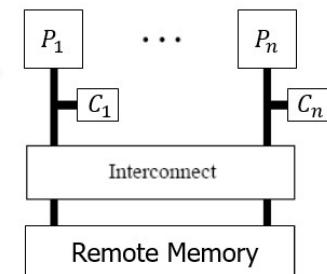
Remote and local memory references

Distributed Shared-Memory (DSM) system:



Cache-Coherent (CC) system:

If P_i needs to use some variable x, P_i first needs to copy x value from the Remote Memory to its cache C_i. Then P_i is allowed to use the copy of x in C_i till x's value is changed in the Remote Memory by some other process. In this case, the value of x is automatically discarded (becomes invalid) by hardware in all the caches it appears.



Distributed Shared-Memory (DSM) system

כאשר שני תהליכיים מתרחשים באותו המעבד, נדרש להתמודד עם מספר אתגרים. אנו משתמש באלגוריתם MCS כדי להתמודד עם התקשרות בין זיכרון מקומי לזכרון מרוחק במערכת של זיכרון משותף מבוזר (DSM), ובמערכת של זיכרון משותף עם עקביות מוגן (CC).

במערכת DSM, לכל תהליך יש זיכרון מקומי מוגבל, והתהליכיים מתקשרים זה עם זה באמצעות רשת התקשרות. כל תהליך יכול לאשת לזכרון שלא שיר לו, מה שמאפשר לנו לשחק עם הזיכרון המשותף.

במערכת CC, אף תהליך אין זיכרון מקומי, אך יש לו מטמון. כל התהליכיים יכולים לאשת לזכרון המרוחק המשותף. כאשר מעודכנים משהו ברםם, הם שלוחים אותן (לא בהכרח פסיקה) לכל המעבדים לבדוק אם היו שינויים בערכיהם.

אלגוריתמים שקשורים לרשאות - המدد הרלוונטי זה מספר ההעתקות הנדרשות. אנחנו נרצה לסנכרן בין 2 תהליכיים מרוחקים.

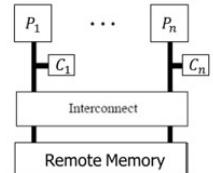
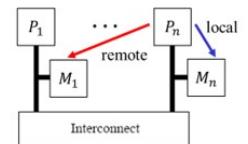
Local Spin algorithm – definition

- ❑ Local Spin algorithm – all busy wait is done by read-only loop of local-accesses

For each $1 \leq i \leq n$, if P_i awaits on x and the algorithm is local spin, then P_i executes limited remote accesses to x during the awaiting.

same algorithm may be local-spin on one architecture (DSM or CC) and non-local spin on the other

to avoid interconnect traffic



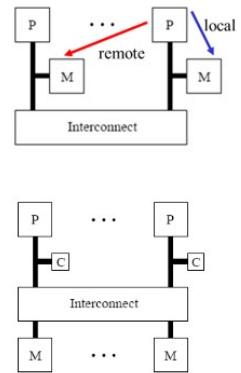
- ❑ complexity of distributed algorithm is calculated by **number of Remote Memory References** (RMRs)

Local Spin algorithm – definition

אלגוריתם מסוים נחשב לאלגוריתם "Local Spin" אם כל הבדיקה שהוא מבצע מתבצעת בלולאה של גישות זיכרון מקומי בלבד. אותו אלגוריתם יכול להיות "Local Spin" בארQUITקטורה אחת (DSM או CC) ולא אחרת. המטרה היא למנוע תנואה ברשת התקשרות. סיבוכיות של אלגוריתם מבואר מחושבת על פי מספר הפניות לזכרון מרוחק.

Peterson's 2-process algorithm

initially: $b[0]=\text{false}$, $b[1]=\text{false}$	
P0:	<ol style="list-style-type: none"> $b[0] \leftarrow \text{true}$ $\text{turn} \leftarrow 0$ await ($b[1]=\text{false}$ or $\text{turn}=1$) CS $b[0] \leftarrow \text{false}$
P1:	<ol style="list-style-type: none"> $b[1] \leftarrow \text{true}$ $\text{turn} \leftarrow 1$ await ($b[0]=\text{false}$ or $\text{turn}=0$) CS $b[1] \leftarrow \text{false}$



Is this algorithm local-spin on a DSM machine? **No**

Is this algorithm local-spin on a CC machine? **Yes**

לפני שנעבור לבחינת האלגוריתם המקורי, נסתכל על אלגוריתם פטרסון בתחום רשות. באורסה הבסיסית של האלגוריתם, יש לנו שני תהליכיים, P0 ו-P1, ושלושה משתנים משותפים שאנו צריכים להחליט איפה למקם אותם.

אם נסתכל על דוגמה רנדומלית למיקום של האלגוריתם בזיכרון, נראה שבמערכת DSM, כל המשתנים ממוקמים אצל P0 (נבחר באופן רנדומלי). עבור P0, זו גישה לokailit, אך עבור P1, זו גישה מרוחקת - הוא לא יכול לשמר עותק מקומי, ולכן כל פעם שהוא רוצה לגשת למשתנה, הוא צריך לגשת ל זיכרון המרוחק. זה אומר שהאלגוריתם אינו "Local Spin" במערכת DSM.

במערכת CC, נניח ש-P0 מבצע גישה ל זיכרון המרוחק ובודק את המשתנה במתמונן שלו. אם המשתנה B משתנה אז יש צורך בגישה ל זיכרון המרוחק, אך זה יקרה רק פעם אחת או שתיים, ככלומר, יש מספר סופי של גישות ל זיכרון המרוחק, ולכן האלגוריתם הוא "Local Spin".

במונח "Local Spin", אנו מתייחסים למצב שבו אנחנו לא מבצעים המתנה על זיכרון מרוחק. בסה"כ, יש לכל תהליך לפחות 5 גישות, ככלומר, יש 10 גישות בסה"כ, שזה מספר סופי, ולכן האלגוריתם הוא "Local Spin".

כאשר אנחנו כתבים קוד ואלגוריתמים, אנחנו לא תמיד חושבים על הקונפיגורציה של הרשות. אך כאשר אנחנו עובדים עם רשות, זה משווה שאנו צריכים לשים לב אליו.

The Mellor-Crummey and Scott (MCS) algorithm (1991)

- Uses Read, Write, Swap, and Compare-And-Swap (CAS) operations
- Provides starvation-freedom and FIFO
- O(1) RMRs per passage in both CC/DSM
- Widely used in practice (also in Linux)



John Mellor-Crummey

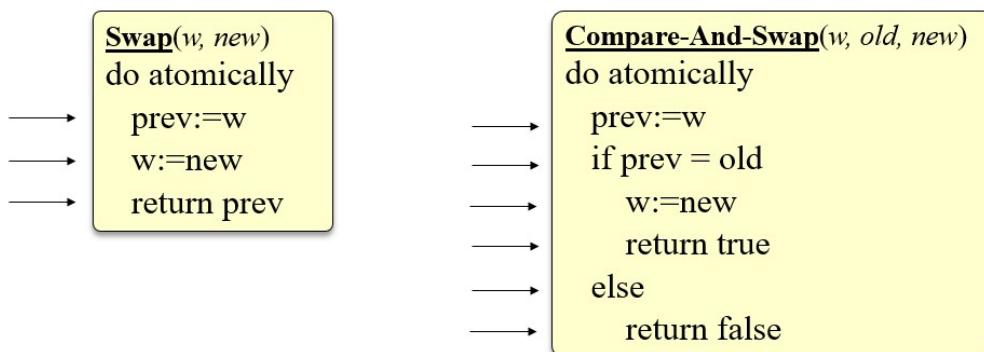


Michael Scott

אלגוריתם מלור-קראמי וסקוט (MCS) משנת 1991

אלגוריתם זה משתמש בפעולות `Compare-And-Swap (CAS)`, `Read`, `Write`, `Swap` ו-`I`. האלגוריתם מספק חופש מהתרעבות (starvation-freedom) ותור FIFO. בנוסף, האלגוריתם מבצע O(1) פניות לזכרון מרוחק (RMRs) בכל חייה, בין אם מדובר במערכת CC או DSM. האלגוריתם משתמש באופן נרחב במערכות מערכות הפעלה לינוקס. האלגוריתם הוא יחסית חדש, אך מאוד אטרקטיבי מבחינה זמן הריצה שלו.

Swap & Compare-And-Swap atomic instructions



הוראות אטומיות Swap & Compare-And-Swap

הוראת Swap מקבלת ערך חדש, שומרת את הערך הישן, מעדכנת את הערך החדש ומוחזירה את הערך הישן. היא בהכרח מבצעת זאת באופן אטומי, בניגוד להוראת TestAndSet .

The Mellor-Crummey and Scott (MCS) algorithm (1991)

Qnode structure: {bit *locked*, Qnode **next*}
 each process has Qnode **myNode*
 shared variable Qnode **tail*
 initially: *tail=NULL*

Program for process i

```

1. myNode->next := NULL // prepare to be last in queue
2. pred=swap(tail, myNode) // tail now points to myNode
3. if (pred ≠ null) // I need to wait for a predecessor
4.   | myNode->locked := true // prepare to wait
5.   | pred->next := myNode // let my predecessor know it has to unlock me
6.   | await myNode->locked := false
7. CS
8. if (myNode->next = NULL) // if not sure there is a successor
9.   | if (compare-and-swap(tail, myNode, NULL) = false) // if there is a successor
10.    |   | await (myNode->next ≠ NULL) // spin until successor lets me know its identity
11.    |   | successor := myNode->next // get a pointer to my successor
12.    |   | successor->locked := false // unlock my successor
13. else // for sure, I have a successor
14.   | successor := myNode->next // get a pointer to my successor
15.   | successor->locked := false // unlock my successor

```



תהליך קבלת המניעול.

כאשר תהליך (thread) רוצה לקבל את המניעול, הוא מוסיף את צומת הסנסקוון שלו בסוף הרשימה המקוורת. אם התור לא היה ריק מלכתחילה, התהליך מעדכן את שדה 'next' של הצומת הקודם להפנות לצומת שלו. לאחר מכן, התהליך מבצע החלפה על שדה הנעילה המקומי בצומת שלו ומכחח שהצומת הקודמת תשנה את השדה לשקר.

תהליך שחרור הנעילה

בעת שחרור הנעילה, התהליך בודק אם יש אחריו עוד תהליך (successor). אם יש כזה, הוא משחרר את הנעילה של הבא בתור. אם אין כזה, הוא מנסה להחליף את הזנב של הרשימה ל-NULL באמצעות הוראת-Compare-And-Swap. אם ההחלפה מצליחה, התהליך משחרר את הנעילה. אם ההחלפה נסלת, זה אומר שיש עוד תהליך שהגיע, וההתהליך ממתין שההתהליך הזה יוסיף את עצמו לרשימה, אז ישחרר את הנעילה.

האלגוריתם הוא Local Spin Lock, מכיוון שבkońפיגורציה הראשונה, אנו ממקמים את הצומת של התהליך בזיכרון של התהליך עצמו. מאחר ואנו לא באמת מביצעים דגימה של הזנב, זה לא משנה איפה הוא יהיה בזיכרון. כמו כן, הנטנה בשורה 10 מתרחשת רק פעם אחת, וכל שאר השינויים לא מעוניינים אותנו כתהליך, מכיוון שאנו לא ממתינים להם. בזכות הדרך שבה הרשימה מורכבת, אין הטרibusות ומתקיים FIFO.

The Mellor-Crummey and Scott (MCS) algorithm (1991)

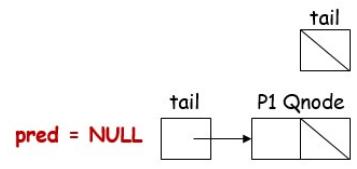
Qnode structure: {bit *locked*, Qnode **next*}
 each process has Qnode **myNode*
 shared variable Qnode **tail*
 initially: tail=NULL

Program for process i

```

1. myNode->next := NULL // prepare to be last in queue
2. pred=swap(&tail, myNode) // tail now points to myNode
3. if (pred ≠ null) // I need to wait for a predecessor
4.   myNode->locked := true // prepare to wait
5.   pred->next := myNode // let my predecessor know it has to unlock me
6.   await myNode->locked := false
7. CS
8. if (myNode->next = NULL) // if not sure there is a successor
9.   if (compare-and-swap(&tail, myNode, NULL) = false) // if there is a successor
10.    await (myNode->next ≠ NULL) // spin until successor lets me know its identity
11.    successor := myNode->next // get a pointer to my successor
12.    successor->locked := false // unlock my successor
13. else // for sure, I have a successor
14.   successor := myNode->next // get a pointer to my successor
15.   successor->locked := false // unlock my successor

```



The Mellor-Crummey and Scott (MCS) algorithm (1991)

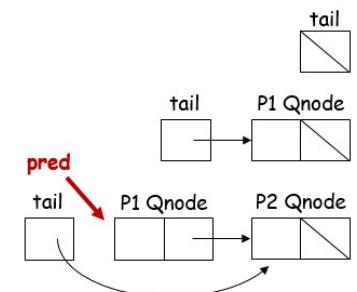
Qnode structure: {bit *locked*, Qnode **next*}
 each process has Qnode **myNode*
 shared variable Qnode **tail*
 initially: tail=NULL

Program for process i

```

1. myNode->next := NULL // prepare to be last in queue
2. pred=swap(&tail, myNode) // tail now points to myNode
3. if (pred ≠ null) // I need to wait for a predecessor
4.   myNode->locked := true // prepare to wait
5.   pred->next := myNode // let my predecessor know it has to unlock me
6.   await myNode->locked := false
7. CS
8. if (myNode->next = NULL) // if not sure there is a successor
9.   if (compare-and-swap(&tail, myNode, NULL) = false) // if there is a successor
10.    await (myNode->next ≠ NULL) // spin until successor lets me know its identity
11.    successor := myNode->next // get a pointer to my successor
12.    successor->locked := false // unlock my successor
13. else // for sure, I have a successor
14.   successor := myNode->next // get a pointer to my successor
15.   successor->locked := false // unlock my successor

```

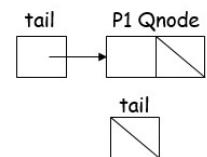


The Mellor-Crummey and Scott (MCS) algorithm (1991)

Qnode structure: {bit *locked*, Qnode **next*}
each process has Qnode **myNode*
shared variable Qnode **tail*
initially: tail=NULL

Program for process i

```
1. myNode->next := NULL // prepare to be last in queue
2. pred=swap(&tail, myNode) // tail now points to myNode
3. if (pred ≠ null) // I need to wait for a predecessor
4.   | myNode->locked := true // prepare to wait
5.   | pred->next := myNode // let my predecessor know it has to unlock me
6.   | await myNode->locked := false
7. CS
8. if (myNode->next = NULL) // if not sure there is a successor
9.   | if (compare-and-swap(&tail, myNode, NULL) = false) // if there is a successor
10.    | | await (myNode->next ≠ NULL) // spin until successor lets me know its identity
11.    | | successor := myNode->next // get a pointer to my successor
12.    | | successor->locked := false // unlock my successor
13. else // for sure, I have a successor
14.   | successor := myNode->next // get a pointer to my successor
15.   | successor->locked := false // unlock my successor
```

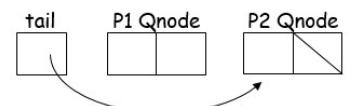


The Mellor-Crummey and Scott (MCS) algorithm (1991)

Qnode structure: {bit *locked*, Qnode **next*}
each process has Qnode **myNode*
shared variable Qnode **tail*
initially: tail=NULL

Program for process i

```
1. myNode->next := NULL // prepare to be last in queue
2. pred=swap(&tail, myNode) // tail now points to myNode
3. if (pred ≠ null) // I need to wait for a predecessor
4.   | myNode->locked := true // prepare to wait
5.   | pred->next := myNode // let my predecessor know it has to unlock me
6.   | await myNode->locked := false
7. CS
8. if (myNode->next = NULL) // if not sure there is a successor
9.   | if (compare-and-swap(&tail, myNode, NULL) = false) // if there is a successor
10.    | | await (myNode->next ≠ NULL) // spin until successor lets me know its identity
11.    | | successor := myNode->next // get a pointer to my successor
12.    | | successor->locked := false // unlock my successor
13. else // for sure, I have a successor
14.   | successor := myNode->next // get a pointer to my successor
15.   | successor->locked := false // unlock my successor
```

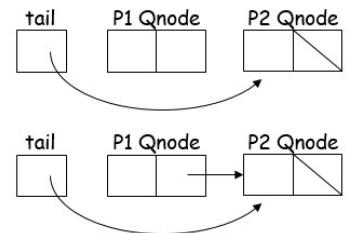


The Mellor-Crummey and Scott (MCS) algorithm (1991)

Qnode structure: {bit *locked*, Qnode **next*}
each process has Qnode **myNode*
shared variable Qnode **tail*
initially: tail=NULL

Program for process i

```
1. myNode->next := NULL // prepare to be last in queue
2. pred=swap(&tail, myNode) // tail now points to myNode
3. if (pred ≠ null) // I need to wait for a predecessor
4.   | myNode->locked := true // prepare to wait
5.   | pred->next := myNode // let my predecessor know it has to unlock me
6.   | await myNode->locked := false
7. CS
8. if (myNode->next = NULL) // if not sure there is a successor
9.   | if (compare-and-swap(&tail, myNode, NULL) = false) // if there is a successor
10.    | | await (myNode->next ≠ NULL) // spin until successor lets me know its identity
11.    | | successor := myNode->next // get a pointer to my successor
12.    | | successor->locked := false // unlock my successor
13. else // for sure, I have a successor
14.   | successor := myNode->next // get a pointer to my successor
15.   | successor->locked := false // unlock my successor
```

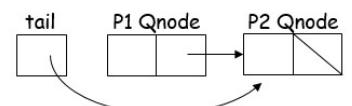


The Mellor-Crummey and Scott (MCS) algorithm (1991)

Qnode structure: {bit *locked*, Qnode **next*}
each process has Qnode **myNode*
shared variable Qnode **tail*
initially: tail=NULL

Program for process i

```
1. myNode->next := NULL // prepare to be last in queue
2. pred=swap(&tail, myNode) // tail now points to myNode
3. if (pred ≠ null) // I need to wait for a predecessor
4.   | myNode->locked := true // prepare to wait
5.   | pred->next := myNode // let my predecessor know it has to unlock me
6.   | await myNode->locked := false
7. CS
8. if (myNode->next = NULL) // if not sure there is a successor
9.   | if (compare-and-swap(&tail, myNode, NULL) = false) // if there is a successor
10.    | | await (myNode->next ≠ NULL) // spin until successor lets me know its identity
11.    | | successor := myNode->next // get a pointer to my successor
12.    | | successor->locked := false // unlock my successor
13. else // for sure, I have a successor
14.   | successor := myNode->next // get a pointer to my successor
15.   | successor->locked := false // unlock my successor
```





© 2022 מתקנים הפעלה

Operating Systems

Lecture 6 – Synchronization: Monitors and MCS algorithm

Dr. Marina Kogan-Sadetsky