נושא: מערכות הפעלה
מרצה: ד"ר אלעד חורב
אתר הקורס: www.elad-horev.org/os15
מבנה הקורס: ציון: 70% מבחן, 30% עבודות (הגשה - יום לפני סוף הסמסטר)

חומר הקורס: בקורס זה ישנם הרבה נושאים, בפועל ניגע רק ב-2 הראשונים:
א. Synchronisation- עיקר הקורס
ב. ניהול זיכרון (RAM)

מצגת מס' 1

השיעור הראשון, ואף השני יעסקו בהגדרת מושגים, ובעיקר נרצה לדבר על system call, כפי שיוסבר להלן.

- Unix - מערכות מאוד כבדות, מול Linux - שהיא קלה יותר.

- Bus - קו תקשורת עליו יושבים המרכיבים. הרכיבים שולחים בינם נתונים.

- נבחין בין 2 סוגי תוכניות: א. System (Resorce managers), ז"א: מנהלת משאבים.
ב. User - פתרונות לבעיות של משתמש (כמו word וכו')

הגדרה 1: מע' ההפעלה - OS - זוהי תוכנית system (המשתמש לא ניגש לתוכנית זו), המהווה software abstraction ומספקת ממשק תכנותי עבורם (בעצם זו מעין "מגן" למשתמש - שיוכל לתכנת את הרכיבים עצמם).

(שקף 7) כל תוכנה בנויה בשלבים - ראה טבלה, השלב הראשון - אלו תוכנות שהמשתמש נוגע בהם, לעומת ה: kernel mode - שם אין אפשרות לגעת (=מוגן מה: user).
Shell - תוכנת system, אך היא יושבת ב: apps שיש להם גישה למשתמש! (user mode) ולכן הם לא חלק ממערכת ההפעלה.

שכבת החומרה (שקף 8) - אנחנו לא נתעסק בחומרה, אך בכמה מילים:
השלב הראשון (מלמעלה למטה) - הוא שפת המוכנה.
השלב השני - מיקרו-פרוגרמינג - זהו קיבוץ היחידות הפיזיות של המחשב - ליחידות לוגיות פונקציונליות.
השלב השלישי - החומרה עצמה.
שקף 9 - מראה לנו איך הדברים עובדים בצורה ברורה יותר.

הגדרה 2: שפת מכונה - החומרה + פקודות המיקרו דלעיל - החשופות לרמת האסמבלי (ISA), נקראות ביחד: 'machine lang'.

נתחיל לעבור על הגדרות בסיסיות (שקף 12)

Multiprogramming - הגדרה 4: בעבר OSs הריצו task אחר task, ללא קשר/התחשבות לכמות הזמן הנצרך ל: task על I\O וכיוצא בזה. על מנת להגביר את ניצול ה: CPU הוחלט על חלוקת תשומת הלב של ה: CPU בין task-ים שונים, והחילוף בינהם נקרא: spooling.

הגדרה 5: האקט של מעבר בין task-ים נקרא: context switch. זו פעולה יקרה יחסית, ולכן נעדיף הרבה פעמים - לעבור בין משימות בעזרת thread-ים, שהם לא מצריכים context-switch.

Processes - המרכיב המרכזי של כל הקורס - תהליכים. הגדרה 6: כך נקראת תוכנית בהרצה.
יש לנו שלושה מרכיבים: קוד (קוד בינארי שרץ), data ומחסנית. בעצם הפרוסס הוא קונטיינר הנדרש בשביל ה: CPU בשביל להריצו.
הגדרה 7: רשימת המקומות הפיזיים המשויכים ל: processes על מנת לבצע את פעולותו נקראים address space. (עמוד 15 במצגת)

שקף 16 - המודל - לביצוע פעולות, הגדרה 8: העיבוד של פקודה אחת ע"י ה:cpu במודל הנ"ל נקרא: instraction cycle. השאלה היא - מה קורה כאשר ישנם כמה תהליכים בבת אחת... בנוסף: כאשר יש תהליכים - אנחנו נרצה שהם יתקשרו ביניהם, ולכן ניצור "העברת הודעות" הנקרא pipe, וזהו סוג מיוחד של קובץ, שניתן להעביר בין תהליך לתהליך (בדרך אחרת - זה היה מאוד קשה).

שקף 19 - הנושא המרכזי, כאשר יש לנו כמה תהליכים - אנחנו עלולים להגיע באחד מהם ל:"מבוי סתום" (deadLock), כאשר אחד התהליכים מגיע למקום שממנו הוא אינו יכול להמשיך.

נפרט מעט יותר על ה:address space: כל כתובת וירטואלית צריכה להיות מומרת לכתובת פיזית, והדברים מסתבכים כאשר מגדירים דברים דינאמיים. ובעצם אנחנו מחלקים בין - לפני ריצה, לזמן ריצה. הסיבה לעצם החלוקה - תוסבר לקראת סוף הסמסטר.

שקף 21 - threads - דיברנו על CPU בודד שעושה כמה משימות. כעת לא נרצה לעבוד ב: Proscess שונה, אלא באותו פרוסס - נרצה לחלק את המשימות, באופן בלתי תלוי אחד בשני, אחרת יש חשש ל"מבוי סתום", מעבר לכך - מחירו של פרוסס חדש - יקר בהרבה ממחירו של טרד נוסף.

שקף 23 - ניתן לראות פרוסס שיש בו כמה טרדים. כמובן שבדבר זה יש סכנה, כי חלק מהנתונים משותפים (מסומן בוורוד), ואנו נתמודד בשביל למנוע בעיות מסוג זה.

לא נלמד על "קבצים" (מפאת חוסר זמן).

נעבור לדבר על system calls - הגדרה 9: זהו הממשק דרכו ה:user מבקש שירותים מה:kernel. זהו מעין switch בין מצב "משתמש" למצב 'ker. בפועל יש הרבה מעברים בין 2 המצבים (שקף 28).

נעבור לדבר על I\O (שקף 30) - יש כמה סוגים, הראשון: המחשב מבקש בקשה וממתין עד להשלמה - Blocking, ויש סוג שאינו ממתין Non, ויש את הסוג המסובך יותר - שממשיך בשאר התהליכים תוך כדי שהוא מחכה למשתמש: Asynchronous.

פסיקות - interrupts - בקשה מה:CPU לעצור את שאר המשימות - ולתת לאירוע מסוים את המקום הראשון. ראינו זאת בקוד, אך יותר יעניין אותנו בחומרה. הגדרה מדויקת יותר: זהו אות למע' הפעלה שהתרחש אירוע - שיש לטפל בו באופן מיידי. למשל: exceptions, זוהי סוג של פסיקה, ואנו יכולים להחליט - מה לעשות כאשר מקבלים אותה.

מה קורה כאשר מתרחשת פסיקה?

א. הפרוסס שרץ כעת מושהה
ב. ה:os מקבלת את תשומת הלב של ה:CPU.
ג. מפנה את האות ל:interrupt handler
ד. הפרוסס חוזר.

טיפול במספר פסיקות בו זמנית. יש 2 דרכים לטפל במצב:

א. נטפל בפסיקה פסיקה ללא עדיפות
ב. נטפל בפסיקה פסיקה עם עדיפות (בדומה לתור עדיפויות)

חשוב לשים לב כי בדרך כלל אין חפיפה בין פסיקה לפסיקה, ולא חייבים לעצור את כולם, זה בעצם מה שיכול הטרד לעשות (הובא ממצגת מס' 2).

- Shell - תוכנית System שיושבת ב:UserMode שאיננה חלק מה:OS (מבחינתנו אין הבדל בין Shell ל:Word).

- OS Design - יש 2 דרכים קיצוניות:
  א. "אבן גדולה" - כל המע' כתובה בקובץ אחד, כך היו המע' של Unix הראשונות.
  ב. וירטואל משאין - כך בנויות המע' בימנו, החסרון - אין גישה בין פרוסס לפרוסס אחר.

- Posix - פרוטוקול שמגדיר כיצד צריכה להראות OS. חשוב לזכור - שלא כל המע' עומדות בתנאים הללו.

מצגת מס' 2

נכנס יותר לעומק לנושא הפרוסס מול הטרד.

- מטרות ה:OS לגבי הפרוססים:
  א. החלפה, מה שעולה מחיר יקר.
  ב. זמן תגובה הגיוני
  ג. תקשורת וסנכרון בין הפרוססים

- הסיבות ליצירת פרוסס:
  א. נוצרים בהדלקת המחשב (boot)
  ב. תהליך אחד יוצר תהליך אחר
  ג. המשתמש מבקש ליצור תהליך (ראה תרגול).

- הסיבות לסיום תהליך:
  א. יציאה נורמלית (כיבוי המערכת למשל).
  ב. טעות הגורמת לפרוסס לסיים (Error)
  ג. טעות הגורמת לקריסה (Fatal Error)
  ד. פרוסס אחד "מחסל" פרוסס אחר.

- מצב הפרוסס (כמובן שהכל קורה באופן אוטומטי, אך מה קורה לפרוסס במהלך חייו)
  א. running - התהליך רץ
  ב. ready - התהליך לא רץ, אך יכול לרוץ (לוגית - אינו ממתין לשום אירוע או משאב שיאפשר לו לרוץ)
  ג. blocked - תהליך לא רץ וגם לא יכול לרוץ כי הוא ממתין ל:event שיתרחש.

כיצד נעבור ממצב למצב?
  א. running <-> ready - החלטה של ה:OS (כלומר: scheduling algo).
  ב. running -> blocked - הפרוסס מחכה ל:event שיתרחש.
  ג. blocked -> ready - תהליך לא רץ אך ה:event אליו הוא חיכה - אכן התרחש (כנ"ל לגבי המשאבים - שכעת הם התפנו).

- optimization - הצעת ייעול - בגלל שה:RAM מוגבל, אי אפשר לשים את כל הפרוססים (Blocked Ready) על גביו, ולכן יש צורך להעביר חלק מהם אל HHD. כמובן שהצעה זו עלולה להיות יקרה מאוד.

- סוגים שונים של פרוססים
  א. orphan - יתום, ז"א: הפרוסס שיצר אותו "חוסל"
  ב. zombie - הפרוסס סיים את תפקידו, אך הוא עדיין מופיע ב:table, או פרוסס שאביו לא חיכה לו.

- ההבדל בין פרוסס לטרד: הפרוסס כולו ייחודי ואינו חולק מידע, לעומת הטרד שחלקו ייחודי, אך חלקו משותף, כגון: data, code. מתוך כך: switch בין פרוסס לפרוסס יקר בהרבה מאשר switch בין טרד לטרד.

- ישנם 3 סוגי שטחי עבודה עבור טרדים: user-space מול kernel-space, כאשר user בטוח יותר ואילו kernel חזק יותר, וניתן גם לשלב בין השנים (שקופית 25-27).

- מצבי הטרד זהים למצבי הפרוסס (שלושה מצבים).

- tread pool - מאגר של טרדים, כאשר הם מתוזמנים ע"י ה:manager של המאגר.

מצגת מס' 3

מצגת זו תעסוק ב: synchronization בין תהליכים.

- חשוב לשים לב - יש הבדל גדול בין 2 תהליכים לבין n תהליכים, והדבר יוסבר בהמשך המצגת.

- בפועל - אנחנו לא יכולים להתערב בתזמון של התהליכים, אך נוכל ליצור מציאות בה כל התהליכים יצליחו לרוץ.

- Race condition - מידע משותף שעליו ישנו "מירוץ" בין התהליכים שחולקים אותו. ברור כי תוכנית עם RC - איננה נכונה, ואנחנו נרצה למנוע את הבעיה הזו בעזרת אלגו' שונים.

- הדרך הקלאסית להתמודד עם הבעיה הנ"ל - Critical Section (CS) - הגדרה: חלקי התוכנית שמכילים את המידע המשותף למספר תהליכים.

דרישות מה-CS:

א. בתוך ה:CS יש לכל היותר תהליך אחד.
ב. לא ניתן להניח דבר בעניין ה:CPU, או מהירות הביצוע של התהליך
ג. תהליכים שמחוץ ל:CS לא גורמים להשהייה של תהליכים שבתוך ה:CS.
ד. אף תהליך לא נתקע לנצח בנסיון להיכנס ל:CS (מה שנקרא starvation - הרעבה).

הנחה: אף פרוסס לא תקוע לנצח בתוך ה:CS.

- Mutual exclusion (ME) - דחיה הדדית, ז"א: אם קיים פרוסס בתוך ה:CS - הוא מונע מפרוססים אחרים להיכנס פנימה.

- Deadlock (DL) - קבוצת תהליכים שכל אחד מהם ממתין לאירוע שיכול להתרחש רק ע"י תהליך אחר בקבוצה, נאמר עליהם שהם ב:DL.

תכונות שנרצה שכל אלגו' יקיים:

א. Mutual exclusion - יש להוכיח כי בתוך ה:CS יש פרוסס אחד בלבד.
ב. DL freedom (DLF) - אם מספר תהליכים מנסים להיכנס ל:CS, לפחות אחד מצליח.
ג. Starvation freedom (SF) - אם תהליך p מנסה להיכנס ל:CS, הרי שמתישהו - הוא אכן יצליח להיכנס.

חשוב לזכור: SF גורר DLF. אך DLF לא גורר SF. בנוסף: אם האלגו' אינו DLF ברור שאינו SF.

- פעולות אטומיות - זוהי פעולה שלא ניתן להפסיקה באמצע (ז"א: לא ניתן להפריע בעת ביצועה) לדוגמא: כתיבה של משתנים - נעשית בצורה אטומית, למשל: int x = 1. במבחן - אין להגדיר פעולה אטומית על דעת עצמנו.

לפני שנעבור לאלגו' עצמם, במהלך ההרצאה - ראינו 2 מתוך 3 אלוג' המטפלים ב-2 פרוססים (בתחילה הראה ד"ר חורב אלגו' נאיבי - שעוצר את כל הפסיקות ברגע שתהליך נכנס ל:CS, כמובן שפתרון זה אינו טוב כי: מסוכן לתת ל:USER את האפשרות לעצור את כל הפסיקות, וכן - לא מותאם עבור ריבוי תהליכים.

- Nonpreemprive \ preemptive - מע' הפעלה עובדות היום בצורה כזאת שניתן לעצור תהליך בתוך ה:CS.

- busy-waiting - זוהי בעצם לולאת while ללא body - שרצה עד שהאירוע המבוקש בה - יתרחש. מתורגם ל:await.

נעבור כעת על מספר אלגו' ונבדוק האם הם מקיימים את התכונות שהבאנו לעיל.

Algorithm: lock variables

Shared: lock variable

Initially: lock = 0

```
Program for both processes
1. await  lock=0
2. lock:=1
3. cs
4. lock:=0
```

א. naive 1 - יש משתנה משותף: lock עבור 2 הפרוססים, מאותחל ב:0. כל עוד lock אינו 0 - אנחנו מחכים (await). כאשר הוא 0 - נכנסים ל:CS, וכאשר יוצאים מה:CS מחזירים אותו לאפס.

נוכיח/נסתור את שלושת הטענות:

a. האלגו' אינו ME - נוכיח זאת: נגדיר 2 פרוסיסים A, B.

* A רוצה להיכנס ל:CS - וקורא lock=0

לפני ש:A משנה את lock ל:1 הוא נעצר ע"י ה:OS ו:B מקבל זמן ריצה

* B רואה lock = 0, משנה את lock ל:1 ונכנס ל:CS

A מקבל זמן ריצה, זוכר כי lock = 0 ונכנס גם כן ל:CS, יוצא ש:A,B נמצאים באותו זמן ב:CS

b. האלגו' הינו DLF - נניח בשלילה כי A, B מנסים להיכנס ל:CS אך תקועים ב:Entry, היות ו:lock=1 עבור שניהם.

* A רואה lock = 1 כי B ביצע lock = 1, ז"א: B נמצא בתוך ה:CS.

* B רואה lock = 1 כי A ביצע lock = 1, ז"א: A נמצא בתוך ה:CS, סתירה להנחה!

c. האלגו' אינו SF - A,B נמצאים ב:RC על lock. אם B מהיר יותר מ:A יוכל B לנצח את A בכל מרוץ על lock ויגרום לכך ש:A יראה lock = 1 בכל זמן שיקבל זמן ריצה.

חשוב לשים לב: בהמשך נראה שישנן 3 טענות שיש לבדוק בשביל להוכיח שהאלגו' אכן SF, במקרה זה הוא אינו - ולכן לא דיברנו על כך (ראה להלן).

Algorithm:

Shared: turn variable

Initially: turn = 0

Strict alternation

```
Program for process 0
1. await turn=0
2. CS
3. turn:=1
```

```
Program for process 1
1. await turn=1
2. CS
3. turn:=0
```

ב. naive 2 - יש משתנה משותף turn, עבור 2 הפרוסיסים, מאותחל ב:0 (באופן שרירותי). עבור פרוסס A - כל עוד turn = 1 אנחנו מחכים, אחרת - נכנסים ל:CS. עבור פרוסס B - בדיוק הפוך - מחכים - כל עוד turn = 0, כאשר turn = 1 נכנסים ל:CS. אותו הדבר לגבי היציאה.

נוכיח את שלושת הטענות:

a. האלגו' הינו ME - נוכיח זאת: נגדיר 2 פרוסיסים A, B.

נניח בשלילה כי A,B נמצאים בתוך ה:CS.

* משום ש:A נכנס ל:CS הרי ש:turn = 0

* משום ש:B נכנס ל:CS הרי ש:turn = 1, וזו סתירה להנחה (היות ו:turn לא יכול להיות 0 וגם 1 בו זמנית).

b. האלגו' הינו DLF - נניח בשלילה כי A, B מנסים להיכנס ל:CS אך תקועים ב:Entry:

* A רואה turn = 1 לנצח

* B רואה turn = 0 לנצח, וזו סתירה להנחה

(כנ"ל, כי קיימת נקודה בזמן שבה turn = 0 וגם turn = 1).

c. האלגו' אינו SF - ניתן לשים לב ששני הפרוסיסים תלויים אחד בשני, ולכן אם אחד הפרוסיסים תקוע ב:Reminder-code (יוסבר להלן) הרי שהתהליך השני לא יוכל להיכנס.

נעבור לאלגו' הבא

(שלב מקדים לאלגו' של Peterson)

Algorithm:

Shared: bool interested[2]

Initially: interested[0]=interested[1] = false

```
Program for process 0
1. interested[0]:=true
2. await interested[1]=false
3. CS
4. interested[0]:=false
```

```
Program for process 1
1. interested[1]:=true
2. await interested[0]=false
3. CS
4. interested[1]:=false
```

ג. יש משתנה משותף: interested עבור 2 הפרוסיסים. משתנה זה הינו מערך (בעל 2 איברים) בוליאני. מאותחל ב:false, מה שאומר - כרגע - אף אחד מהפרוסיסים אינו מעוניין להיכנס ל:CS.

שיטת העבודה כדלקמן:

כל פרוסס - מגדיר לפני ה:CS כי הוא מעוניין לקבל גישה, ומלבד זאת - הוא מחכה - עד שהפרוסס השני אינו מעוניין בגישה (ע"פ המערך המשותף להם).

נוכיח את שלושת הטענות:

a. האלגו' אכן ME - נניח בשלילה כי A,B נמצאים בתוך ה:CS, ולכן:
חשוב לשים לב: בהתכתבות עם ד"ר חורב - הוכחה זו אינה מספיקה, להלן הובאה ההוכחה מהשיעור:
* פרוסס A כתב `inter[0] = true`, וראה `inter[1] = false` ולכן הוא נכנס לתוך ה:CS
* פרוסס B כתב `inter[1] = true`, וראה `inter[0] = false` ולכן הוא נכנס לתוך ה:CS, מה שגורם לסתירה, שכן קיימת נקודה בזמן שבה `inter[0], inter[1]` שווים בו זמנית גם ל:true וגם ל:false).

במהלך ההרצאה הראה ד"ר חורב הוכחה ארוכה יותר, כאמור - ההוכחה לעיל - לא מספקת:
נאמר כי תהליך A (נסמנו ב-0) נמצא בתוך ה:CS, ז"א, נראה את הדרך בה הגענו לכך: (מימין לשמאל)

$$1.\ write_1(inter[1] = false) \rightarrow read_0(inter[1] = false) \rightarrow CS_0$$

נסביר: מה גרם לכניסת A לתוך ה:CS? תהליך B (מסומן ב-1) כתב שאינו מעוניין, תהליך A קרא זאת ולכן נכנס לתוך ה:CS.

כאמור, גם B נמצא בתוך ה:CS, וזה התרחש באופן הבא:

$$2.\ write_0(inter[0] = false) \rightarrow read_1(inter[0] = false) \rightarrow CS_1$$

כעת - נפרש את הקטעים המסומנים בצהוב, על פי ה:entry:

$$1.2.\ write_0(inter[0] = true) \rightarrow read_0(inter[1] = false)$$
$$2.2.\ write_1(inter[1] = true) \rightarrow read_1(inter[0] = false)$$

משלב זה לצערנו לא הבנו את ההוכחה, ומהו המעגל המדובר:
על פי 1.2,2.2 ניתן לומר באופן ישיר כי:

$$read_0(inter[1] = false) \rightarrow write_1(inter[1] = true)$$
$$read_1(inter[0] = false) \rightarrow write_0(inter[0] = true)$$

בעצם נוצרה גרירה בצורת מעגל, מה שאומר כי קיימת סתירה להנחה.

b. האלגו' אינו DLF - ולכן גם לא SF.
* A קובע `inter[0] = true`
* B מקבל זמן ריצה וקובע `inter[1] = true`.
* תהליך B תקוע כי הוא מחכה ל:A, וכן להפך: תהליך A תקוע, כי הוא מחכה ל:B.

ד. נעבור לאלגו' של פטרסון: גם כאן יש משתנה משותף: `interested` עבור 2 הפרוססים. משתנה זה הינו מערך (בעל 2 איברים) בוליאני. מאותחל ב:false, מה שאומר - כרגע - אף אחד מהפרוססים אינו מעוניין להיכנס ל:CS. <u>מעבר לכך</u> - קיים משתנה משותף מסוג int בשם `turn_to_wait` - המייצג תור איזה פרוסס **להמתין**, האתחול שלו אינו קריטי (שרירותי).

PETERSON'S ALGORITHM

Algorithm:
Shared: bool inter[2], turn_to_wait

Initially: inter[0]=inter[1] = false, turn_to_wait = Don't care

Program for process 0

```
1. inter[0]:=true
2. turn_to_wait:=0
3. while (inter[1]=true and
   turn_to_wait=0);
4. CS
5. inter[0]:=false
```

Program for process 1

```
1. inter[1]:=true
2. turn_to_wait:=1
3. while (inter[0]=true
   and turn_to_wait=1)
4. CS
5. inter[1]:=false
```

שיטת העבודה כדלקמן:
כל פרוסס (למשל פרוסס 0) - מגדיר לפני ה:CS כי הוא מעוניין לקבל גישה (`inter[0] = true`), ומלבד זאת - הוא מחכה - כל עוד הפרוסס השני מעוניין בגישה (ע"פ המערך המשותף להם), <u>וגם</u> שתורו (הפרוסס 0) לחכות.
אם אחד התנאים אינו מתקיים - הוא נכנס ל:CS, זה בעצם יפתור לנו את הבעיה באלגו' הקודם.

נוכיח את שלושת הטענות:

a. האלגו' אכן ME - נניח בשלילה כי A,B נמצאים בתוך ה:CS...:
כאן סמך ד"ר חורב על ההוכחה של האלגו' הקודם. ואותו צ"ע צריך גם כאן.

b. האלגו' הינו DLF - נניח בשלילה כי 0,1 מנסים להיכנס ל:CS אך תקועים ב:Entry (ז"א: ב:while)
על מנת ששניהם יהיו תקועים ב:while צריך להתקיים
* 0 רואה כי 1 מעוניין להיכנס, וגם תורו של 0 לחכות

* 1 רואה כי 0 מעוניין להיכנס, וגם תורו של 1 לחכות. זו סתירה, שכן קיימת נקודת זמן בה המשתנה turn_to_wait הינו גם אפס וגם אחד.

c. האלגו' הינו SF, על מנת להוכיח זאת - נסתכל על 3 מקרים, וכך יש להוכיח כל SF בהמשך הדרך.
תחילה נניח בשלילה כי תהליך 0 תקוע לנצח ב:entry שלו (בתוך ה:while). מה שאומר:

```
inter[0] = inter[1] = true
turn_to_wait = 0
```

עבור תהליך 1 יתכנו שלושה מקרים:

1. תהליך 1 נשאר ב:remainder code שלו לנצח - במקרה זה inter[1] = false, כי אנחנו לא מנסים להיכנס ל:CS, וזו כמובן סתירה להנחה דלעיל.
2. תהליך 1 תקוע ב:entry code שלו לנצח - דבר זה אינו אפשרי - משום שכבר הוכחנו שהאלגו' הינו DLF.
3. תהליך 1 יוצא ונכנס מה:CS באופן מהיר יותר מתהליך 0 - גם זה לא יגרום לתהליך 0 להיתקע, שכן כאשר תהליך 1 יוצא מה:CS הוא מגדיר inter[1] = false, אך בגלל היותו מהיר יותר - מייד הוא מגדיר inter[1] = true. אך יחד עם זאת הוא מגדיר turn_to_wait = 1, ומנסה להיכנס ל:CS שוב, אך הפעם הוא לא יכול - שכן - תורו לחכות וגם תהליך 0 אכן מעוניין. כעת תהליך 0 יקבל זמן ריצה - ויכנס.

Shared: lock

Initially: lock = 0

```
while (Test-and-Set(lock) == 1);
CS
lock:=0
```

```
Test-and-Set(w){
 do atomically:
  prev:= w
  w:= 1
  return prev
}
```

נעבור לאלגו' הבא, באלוג' זה נשתמש כבר בפעולות אטומיות:

ה. testAndSet. חשוב לשים לב - בכל שלב של הוכחה - חייבים להזכיר שמדובר בפעולה אטומית, אחרת ההוכחה איננה נכונה (ז"א: "היות והפעולה אטומית...)

נסביר תחילה את הפעולה האטומית - TAS. אנחנו מקבלים את w (משתנה בינארי - אפס או אחד),
מגדירים משתנה prev השווה לו, מאתחלים את w ב-1,
ומחזירים את prev, ז"א:
עבור w=1 דבר לא ישתנה, אך עבור w=0, הערך המוחזר הינו 0, אך w הפך להיות 1.

כעת נראה את השימוש של TAS כאלגו' המקיים:ME: נאמר וישנם 2 תהליכים 0,1 - ולהם משתנה משותף lock המוגדר להיות 0.
כעת - נכנס ללולאה - כל עוד TAS(lock)==1 נמתין, אחרת - ניכנס ל:CS, וביציאה נבצע lock = 0.

נוכיח את שלושת הטענות:

a. האלוג' אכן ME - נניח בשלילה כי A,B נמצאים בתוך ה:CS, כלומר: שניהם קיבלו 0 מ:TAS(lock).
* היות ו:TAS פעולה אטומית - הרי זה אומר שלא ניתן להפריע לה באמצע התהליך, אם כן - כאשר תהליך 0 (שרירותית) נכנס לתוך ה:TAS ומשנה את lock = 1 הרי שתהליך 1 לא יוכל להיכנס כל עוד תהליך 0 לא יצא (שרק לאחר היציאה התהליך 0 משנה את lock להיות שוב שווה לאפס).

b. האלגו' הינו DLF - נניח בשלילה כי 0,1 מנסים להיכנס ל:CS אך תקועים ב:Entry (ז"א: ב:while), משמע ששניהם מקבלים 1 מ:TAS. היות והייתה נקודה זמן בה lock = 0, הרי שאחד התהליכים כן נכנס לתוך ה:CS, והיות ו:TAS הינה אטומית, זה אומר שאותו תהליך נכנס וגם יצא, משמע ששוב lock = 0, סתירה.

c. האלגו' אינו SF, על מנת להוכיח זאת - נסתכל על 3 מקרים:
תחילה נניח בשלילה כי תהליך 0 תקוע לנצח ב:entry שלו (בתוך ה:while). מה שאומר:

```
TAS(lock) = 1
```

עבור תהליך 1 יתכנו שלושה מקרים:

1. תהליך 1 נשאר ב:remainder code שלו לנצח - במקרה זה lock = 0, כי אנחנו לא מנסים להיכנס ל:CS, וזו כמובן סתירה להנחה דלעיל.
2. תהליך 1 תקוע ב:entry code שלו לנצח - דבר זה אינו אפשרי - משום שכבר הוכחנו שהאלגו' הינו DLF.
3. תהליך 1 יוצא ונכנס מה:CS באופן מהיר יותר מתהליך 0 - ייתכן ותהליך 0 יהיה מהיר יותר מתהליך 1, ובעצם לאחר הפיכת lock = 0, מייד ינסה שוב להיכנס ויעקוף את תהליך 0.

אם כן, "נפלנו" על הטענה השלישית - ולכן האלגו' אינו SF.

נעבור לאלגוריתם שנלמד בתרגול (לא הובא בשיעור עצמו)

ו. אלגו' זה דומה מאוד לפטרסון, וכפי שיוסבר להלן.

Dekker's algorithm

Process = 0

```
bool* flag = {false, false};
int turn = 0 ;

Entry code

flag[0] = true;
while (flag[1]) {
        if (turn==1) {
                flag[0] = false;
                while (turn == 1) { }
                flag[0] = true;
        }
}

/* critical section */

Exit Code

turn = 1;
flag[0] = false;

/* non-critical section */
```

Process = 1

```
bool* flag = {false, false};
int turn = 0 ;

Entry code

flag[1] = true;
while (flag[0]) {
        if (turn==0) {
                flag[1] = false;
                while (turn == 0) { }
                flag[1] = true;
        }
}

/* critical section */

Exit Code

turn = 0;
flag[1] = false;

/* non-critical section */
```

15

אלגו' דקר: גם כאן יש משתנה משותף: interested עבור 2 הפרוססים. משתנה זה הינו מערך (בעל 2 איברים) בוליאני. מאותחל ב:false, מה שאומר - כרגע - אף אחד מהפרוססים אינו מעוניין להיכנס ל:CS. מעבר לכך - קיים משתנה משותף מסוג int בשם turn - המייצג תור איזה פרוסס להיכנס, האתחול שלו אינו קריטי (שרירותי), נאתחל בשביל הדוגמא ב-0.

שיטת העבודה כדלקמן: כל פרוסס (למשל פרוסס 0) - מגדיר לפני ה:CS כי הוא מעוניין לקבל גישה: (inter[0] = true), כעת הוא נכנס ל:while אם גם פרוסס 1 מעוניין ב:CS, אחרת פרוסס 0 נכנס מיידית ל:CS.

בתוך ה:while פרוסס 0 בודק האם התור שייך לפרוסס 1, אם כן - הוא מגדיר את עצמו (פרוסס 0) כך שאינו מעוניין ב:CS, ונכנס ללולאה עד שמגיע תורו (ז"א: עד ש turn == 0). לאחר שפרוסס 1 סיים, והפך את turn להיות אפס - פרוסס אפס חוזר ומגדיר את עצמו כך ש: inter[0] = true, ובעצם נכנס ל:CS, כי turn = 0 וגם inter[1] = false, כי פרוסס 1 סיים את התהליך.

נוכיח את שלושת הטענות:

a. האלוגו' אכן ME - נניח בשלילה כי 0,1 נמצאים בתוך ה:CS:

1. $write_1(inter[1] = false) \rightarrow read_0(inter[1] = false) \rightarrow SC_0$
2. $write_0(inter[0] = false) \rightarrow read_1(inter[0] = false) \rightarrow SC_1$

חסר כאן ההמשך... אם מישהו יודע לעשות זאת בצורה נכונה כמו שהייתה בהרצאה - נשמח לדעת...

b. האלגו' הינו DLF - נניח בשלילה כי 0,1 מנסים להיכנס ל:CS אך תקועים ב:Entry (ז"א: ב:while)
על מנת ששניהם יהיו תקועים ב:while צריך להתקיים
* 0 רואה כי 1 מעוניין להיכנס, וגם תורו של 0 לחכות

* 1 רואה כי 0 מעוניין להיכנס, וגם תורו של 1 לחכות. זו סתירה, שכן קיימת נקודת זמן בה המשתנה turn הינו גם אפס וגם אחד.

c. האלגו' הינו SF, על מנת להוכיח זאת - נסתכל על 3 מקרים:.

תחילה נניח בשלילה כי תהליך 0 תקוע לנצח ב:entry שלו (בתוך ה:while). מה שאומר - 2 אופציות:

```
1. inter[0] = inter[1] = true
   turn = 0
```

או:

```
2. inter[0] = inter[1] = true
   turn = 1
```

עבור תהליך 1 יתכנו שלושה מקרים, נשים לב כי עבור 2 האופציות שלושת הטענות מתקיימות באותו אופן:

1. תהליך 1 נשאר ב:remainder code שלו לנצח - במקרה זה inter[1] = false, כי אנחנו לא מנסים להיכנס ל:CS, וזו כמובן סתירה להנחה דלעיל.
2. תהליך 1 תקוע ב:entry code שלו לנצח - דבר זה אינו אפשרי - משום שכבר הוכחנו שהאלגו' הינו DLF.
3. תהליך 1 יוצא ונכנס מה:CS באופן מהיר יותר מתהליך 0 - גם זה לא יגרום לתהליך 0 להיתקע, שכן כאשר תהליך 1 יוצא מה:CS הוא מגדיר inter[1] = false, אך בגלל היותו מהיר יותר - מייד הוא מגדיר inter[1] = true. אך יחד עם זאת הוא מגדיר turn = 0 (בשלב היציאה), ומנסה להיכנס ל:CS שוב, אך הפעם הוא לא יכול - שכן אם פרוסס 0 מעוניין, ותורו של פרוסס אפס לעבוד - פרוסס 1 מוותר על הרצון שלו להיכנס כל עוד פרוסס אפס לא סיים את עבודתו ושינה את turn להיות 1.

נעבור לדבר על אלגוריתמים עבור n תהליכים (סך אלגו' - 4, וכולם נלמדו בשיעור)

אלגו' מס' 1: 'Tournament algo - משתמשים בכל אלוג' של 2 תהליכים (כמו שלמדנו לעיל), ומפעילים מעין "תחרות" - עבור כל 2 יבחר אחד, וכן הלאה (כמו עץ בינארי) - עד שנגיע לשורש העץ - זהו התהליך שיכנס ל:CS.

אלגו' מס' 2: The filter algo - אלגו' זה הינו הכללה ישירה של פטרסון. כל פרוסס חייב לעבור n-1 "חדרי המתנה" על מנת להיכנס ל:CS. בכל רמה יש מספר מסויים של תהליכים, כאשר ה:level הגבוה ביותר הינו ה:CS.

דוגמא: עבור n=8, יש 7-0 levels. ב:0 level יש 8 תהליכים. ב:7 level - תהליך בודד וזהו ה:CS.

THE FILTER ALGORITHM

Shared:

```
level[n] ={0}
turn_to_wait[n] = Don't care
```

Code for Process i:

```
for (int L = 1; L < n; L++)   {
     level[i]  = L;
     turn_to_wait[L] = i;
     while ((∃ k != i s.t. level[k] >= L)
             and turn_to_wait[L] == i ) {}
     }
critical section
level[i] = 0;
```

איך האלגו' עובד? יש משתנה משותף: מערך בשם level בגודל n, כאשר הוא מאותחל ב-0, ז"א: רמת ההתעניינות של כל תהליך. בנוסף: ישנו מערך turn_to_wait בגודל n, האתחול הינו שרירותי, והוא מייצג את התהליך שמחכה בכל שלב.

עבור כל תהליך - נרוץ בלולאה מ-1 ועד n שזה בעצם הרמות. נאמר כי level[i] = L, כאשר L הוא האינדקס של הלולאה, מה שאומר - הרמה המבוקשת עבור אותו תהליך. מיד לאחר מכן - יגדיר התהליך שתורו לחכות באותו level L. אלא שנחכה רק בתנאי ששני התנאים הבאים יתקיימו:

א. קיים תהליך כלשהו שאינו i, כך שה:level אליו הוא רוצה להגיע הינו ה:level ש:i רוצה להגיע, או גדול ממנו.

ב. תורו של i לחכות.

אחרת, תהליך i יעלה לרמה הבאה, עד שיגיע בסופו של דבר ל:CS.

חשוב לשים לב, כאשר תהליך i יוצא מה:CS, הוא מגדיר את ה:level שלו לאפס, ז"א: לא מעוניין יותר להיכנס.

כעת אנחנו רוצים להוכיח את שלושת הטענות דלעיל:

א. הוכחת ME - במהלך השיעור הוכיח ד"ר חורב טענה מקדימה בעזרת אינדוקציה (claim 2). בסופו של דבר לא סיכמנו זאת. כעת נשתמש בטענה ונאמר שהיא גוררת ME, כי בעצם יש תהליך בודד בתוך ה:CS.

ב. ד"ר חורב לא הוכיח DLF, אלא SF מה שגורר DLF. גם כאן הדבר נעשה באינדוקציה (הפוכה) על הרמות, גם זה לא סוכם, מפאת חוסר זמן.

■ Fairness - הגינות, האלגו' שראינו הינם SF, אך עדיין יש בעיית "הגינות", ז"א: ייתכן ותהליכים יעקפו תהליך אחר מספר פעמים. כעת המטרה היא: להפוך את האלגו' להיות הוגן, כאשר ישנן רמות שונות להגינות, הטובה ביותר - FIFO, ולאחריה n Bounded.

■ doorway - קטע קוד ב: entry code שאף תהליך לא נתקע בו. לעומת waiting code, שם התהליכים ממתינים לפי התנאים (בדרך כלל while כלשהו).

■ R-Bounded waiting - עבור אלגו' ME - נאמר כי הוא R-Bounded אם לכל זוג תהליכים A,B מרגע ש: A מסיים את ה: doorway שלו, B לא עוקף את A ביותר מ: R פעמים בכניסות ל: CS. לכן bounded 0 = FIFO.

טענה: האלגו' Filter הוא הכי לא הוגן שאפשר, וזאת משום שאי אפשר להגדיר R-Bounded עבורו.
נוכיח זאת: נאמר כי A סיים את ה: DW שלו, ונמצא כעת ב: Waiting. נאמר כי B הצטרף וקבע שתורו לחכות. כעת נכנס תהליך P_1 וקובע שתורו לחכות. ה: CPU נותן ל: B זמן ריצה. B מבצע, ושוב חוזר לרמה של A וקובע שתורו לחכות. כעת נכנס P_2 וקובע שתורו לחכות, ושוב ה: CPU נותן ל: B להמשיך לרוץ וחוזר חלילה. נעשה זאת $r+1$ פעמים, מ.ש.ל.

נעבור לאלגוריתם הבא: Lamport.
הרעיון הכללי: כל תהליך "לוקח" מספר בתור, חשוב לשים לב: ייתכן כי 2 תהליכים ויותר יקבלו את אותו התור.
איך האלגו' עובד? יש משתנה משותף: מערך בשם inter בגודל n, כאשר הוא מאותחל ב: false עבור כל תהליך, ז"א: אף תהליך לא מעוניין להיכנס. בנוסף: ישנו מערך number בגודל n, והוא מייצג את ה"תור" של תהליך i כלשהו.

Code for process i

```
Doorway {inter[i]=true;
         number[i] = max(number[0], …,number[n-1])+1;
         while (∃ k!= i such that:
                    inter[k] && (number[i],i) > (number[k],k)) {};

         critical section

         inter[i] = false;
```

עבור כל פרוסס i - נגדיר שהוא מעוניין להיכנס, ניתן לו מספר בתור (מס' עוקב), כל זה נמצא ב: DW. כעת - התהליך i ימתין כל עוד 2 דברים מתקיימים גם יחד:

א. קיים תהליך כלשהו שרוצה להיכנס
ב. התהליך המתחרה - קיבל כרטיס "טוב" יותר, ד"ר חורב עשה זאת בעזרת זוגות סדורים, ז"א:

$$(a,b) > (c,d) \rightarrow (a > c) \quad or \quad (a = c \;\; and \; b > d)$$

הנקודה הזו חשובה במקרה ששני פרוססים יקבלו את אותו מספר בתור - מה שיקבע - יהיה ה: index שלהם.
חשוב לשים לב: לאחר שתהליך יוצא מה: CS מתרחש inter[i] = false, ז"א: הוא כבר לא מעוניין להיכנס.

נוכיח את שלושת הטענות עבור האלגו' הנ"ל:

א. האלגו' הינו ME - נניח בשלילה שלא, ולכן קיימים 2 תהליכים A,B הנמצאים ב: CS בו זמנית.
נסמן: numA,numB את "תור התהליך" בזמן הכניסה שלו לתוך ה: CS.
נניח (בה"כ) ש: $(numA, A) < (NumB, B)$. B הצליח להיכנס ל: CS, כלומר: הוא ראה כי inter[A] = false או $(numA, A) > (NumB, B)$ - ז"א הכרטיס של B טוב יותר, אך זה מנוגד להנחה.
כלומר: נקבל את שרשרת האירועים הבאה:

$$NumB\ getTicket \rightarrow read_B(inter[A] = false) \rightarrow write_A(inter[A] = true) \rightarrow NumA\ getTicket$$

כלומר: $NumA > NumB$ סתירה להנחה.

ב. נוכיח DLF: עבור n תהליכים הממתינים להיכנס, התהליך עם ה"כרטיס" הטוב ביותר - ייכנס ל: CS. כך הוכיח ד"ר חורב את הטענה בשיעור, תימא...

במקום להוכיח SF - נחזור לדבר על R-bounded, ונאמר כי האלגו' הינו 1-bounded (בניגוד לספרים הטוענים כי האלגו' הנ"ל הינו FIFO), לכן בוודאי הוא שהוא מקיים SF.
נראה שהאלגו' אינו FIFO: נניח כי ה: doorway של A מסתיים לפני ה: doorway של B, ז"א: $NumA < NumB$.
כעת: נכנסים $P_1 \dots P_{15}$ תהליכים. כאשר כל התהליכים מקבלים את אותו "כרטיס" (משום שאין כאן פעולה אטומית), אם כן - כל התהליכים - יכולים להיכנס לפני A, היות וה"כרטיס" שלהם נמוך יותר. חשוב לשים לב שכל אחד מהם לא יוכל לעקוף את A פעם נוספת, היות ובכניסתם הבאה - יקבלו תור גדול ממש מ: A.

■ busy waiting - אמרנו בעבר כי השימוש ב: while ב: entry - הוא לא דבר טוב, משום שהוא מבזבז כח ל: CPU, מה עוד שהוא יכול לגרום לבעיות כמו: DL.

מצגת מס' 4

הנושא של המצגת הינו Semaphores.
מה היעוד של סמפור? לפתור את בעיית ה: busy waiting שדיברנו עליה במצגת הקודמת.

נייצג את הנושא של סמפורים בעזרת בעיית ה: Producer-Consumer (PC) (עמ' 3). חשוב לשים לב - מדובר בבעיה קלאסית, במצגת הבאה נעסוק בבעיות נוספות.
בעצם ישנו pro שממלא buffer משותף, ו: con שמרוקן את אותו ה: buffer. הנקודה הבעייתית - מקרי הקצה - כאשר ה: buffer מרוקן או מלא לחלוטין.

ד"ר חורב הראה דוגמא של PC כך שהאלגו' נכנס ל: DL (בשביל להבין טוב יותר - מומלץ לעבור על האלגו' בעמ' 4).
תרחיש: buffer ריק, ה: con קרא count = 0 ונעצר.
pro מקבל זמן ריצה, מכניס אלמנט ל: buffer, וקובע count = 1.
היות ו: count=1 הוא שולח wakeup ל: con.
con מקבל זמן ריצה, זוכר כי count = 0 ומיד חוזר לישון.
ה: pro ממלא את ה: buffer והולך לישון. -> DL.
אם כן - הבעיה במקרה הנ"ל: ה: signal WU נשלח בזמן לא נכון והלך לאיבוד.

נשתמש בסמפור לפתרון הבעיה:

Two atomic operations are supported by a semaphore S:

סמפור הינו משתנה מסוג int, ולכל סמפור ישנן 2 פעולות אטומיות:
Up, Down. כאשר:
down - אנחנו שואלים אם הסמפור $s \leq 0$ נכנסים ל: block, אחרת מחסירים 1 מ: s.
up - אנחנו שואלים - אם ישנם תהליכים "תקועים" - נעיר אחת מהם, אחרת - תוסיף 1 ל: s.
ניתן לשים לב כי s תמיד חיובי בהגדרה הנ"ל, וכמות ה: sleep זהה לכמות ה: wakeUp (הכוונה - כמה פעמים לקחנו מטבע מול כמה פעמים שהחזרנו אותו).

down(S) [the 'P' operation]
- **If** S ≤ 0 then block
- **Else** S-= 1

up(S) [the `v' operation]
- **If** there are blocked processes, wake-up one of them
- **Else** S+=1

1. S ≥ 0 always (= pile of coins)
2. Blocked processes released by up(S)

Pile of coins means:
ups = # downs

לכל סמפור יש "גבול", למשל: סמפור עם מטבע בודד - הוא סמפור בינארי, בדיוק כמו שראינו לעיל. הבינארי סמפור נקרא גם mutex.

האלגו' הבא, שמשתמש בסמפור: עומד בשלושת הדרישות: ME, DLF, וגם SF עבור 2 תהליכים (עבור n תהליכים - יש צורך במסגרת של fairness).

```
Shared: Semaphore lock = 1
DOWN(lock)
<CS>
UP(lock)
```

נוכיח זאת:

א. האלגו' הינו ME - משום ש: up, down הן פעולות אטומיות, ולכן ברגע שתהליך אחד יבצע down - שום תהליך אחר לא יוכל להיכנס ל: CS, עד שאותו תהליך יבצע up ביציאה.
ב. האלגו' הינו DLF - משום ש: up, down הן פעולות אטומיות, הרי שהתהליך הראשון שינסה להיכנס - אכן יצליח, משום ש: down אטומית, ומכאן ולהבא - התהליכים יכנסו ויצאו בזה אחר זה.
ג. האלגו' הינו SF - נניח שלא, ז"א: A ביצע down על lock, ולעולם אינו מצליח לעבור. ישנן 3 אופציות עבור תהליך B:
   a. B תקוע לנצח ב: RC. כיצד הדבר אפשרי? אפשרות א': B ביצע up לפני down של A - הרי ש: lock הינו 1, וזו סתירה לכך ש: A תקוע. אפשרות ב': B ביצע up אחרי down של A - הרי שב: up B מעיר את A, וגם זו סתירה לכך ש: A תקוע.

b. B תקוע ב: entry code - כבר הוכחנו DLF, ולכן הדבר לא ייתכן.
c. תהליך B מהיר יותר מ: A. גם לא אפשרי. שהרי ברגע ש: B עושה up הוא משחרר את A, אז A נכנס אוטומטית ל: CS (אין while שמפריע לו להיכנס).

במהלך ההרצאה - הציע ד"ר חורב אלטרנטיבה ל: up,down של הסמפור, אך הצעה זו איננה נכונה, משום שמספר ה: up אינו זהה למספר ה: down. נוכיח זאת בדוגמא שלילית:

down(S) [the 'P' operation]
- **If** S ≤ 0 then block
- **Else** S-= 1

up(S) [the `v' operation]
- S+=1
- If there are blocked processes, wake-up one of them

נאמר ואנחנו באמצע הרצה: s=0, כעת:
A מבצע down ונחסם.
B מבצע up, אזי: s=1, ו: B מעיר את A.
C מבצע down, ז"א: הוא מעדכן את s להיות 0, והוא לא נחסם.
ז"א: up אחד של B שקול ל: down 2, ומכאן יוצא שגם A וגם C נמצאים בתוך ה: CS.

במהלך ההרצאה נתן ד"ר חורב דוגמא ל-3 תהליכים עם 2 סמפורים, כאשר סמפור s1 מוגדר להיות 1, ו: s2 מוגדר להיות 0.
ראינו שסדר העבודה של התהליכים חייב להיות באופן הבא: $(AB^*C)^*$ (שפה - כמו שלמדנו באוטומטים).

Three processes p1; p2; p3

semaphores s1 = 1, s2 = 0;

p1	*p2*	*p3*
down(*s1*);	down(*s2*);	down(*s2*);
A	B	C
up(*s2*);	up(*s2*);	up(*s1*);

מימוש של הדוגמא הנ"ל: תחילה נממש את מחלקת pro המייצגת את הפרוססים p1,p2,p3:

```
public class pro extends Thread {

	String index;
	Semaphore down;
	Semaphore up;

	public pro(String index, Semaphore down, Semaphore up) {
		this.index = index;
		this.down = down;
		this.up = up;
	}

	public void run() {
		int cur = 0;
		while (cur < 100) {
			try {
				down.acquire();
				System.out.println(index);
				Thread.sleep(500);
				up.release();
			} catch (Exception e) {}
			cur++;
		}
	}
}
```

חשוב לשים לב לשליחת הסמפורים ב: main:

```
public static void main(String[] args) {
	Semaphore s1 = new Semaphore(1);
	Semaphore s2 = new Semaphore(0);
	Thread p1 = new pro("A", s1, s2);
	Thread p2 = new pro("B", s2, s2);
	Thread p3 = new pro("C", s2, s1);
	p1.start(); p2.start(); p3.start();
}
```

נעבור לדבר על <u>סמפורים שליליים</u> (counter semaphore (CS)). ניתן לשים לב שעד עכשיו המשתנה s לא היה שלילי. מה שמאפשר לרדת מתחת לאפס. השיטה דומה לאלטרנטיבה שהביא ד"ר חורב לעיל, אך הפעם - היא עובדת נכון, משום שגם ב:up וגם ב:down מוסיפים ולוקחים מטבע בקביעות:

```
down(s) {
      s = s-1;
      if (s<0) block
}

up(s) {
      s = s + 1;
      If there are blocked processes wake-up one of them
}
```

אם s שלילי - מספר התהליכים החסומים הינו ABS(s).

נחזור לבעיית ה:PC שדבירנו עליה לעיל: ניתן לראות דוגמא לפתרון בעזרת שימוש בסמפור בינארי ושני סמפורים שליליים. כאשר הסמפור הבינארי - אחראי על הכניסה ל:CS, ושני הסמפורים השליליים אחראים על חסימה וביטול החסימה של ה:Con וה:Pro.

PRODUCER-CONSUMER WITH SEMAPHORES

```
#define N        100          /* Buffer size */
semaphore        mutex = 1;   /* access control to critical section */
semaphore        empty = N;   /* counts empty buffer slots */
semaphore        full = 0;    /* counts full slots */
```

```
void producer(void) {
   int   item;
   while(TRUE) {
        produce_item(item);
        down(empty);          /* dec. empty */
        down(mutex);          /* enter cs */
        enter_item(item);     /* insert*/
        up(mutex);            /* leave CS */
        up(full);             /* inc. full */
        }
   }
```

```
void consumer(void)
   {
        int          item;

        while(TRUE)    {
          down(full);          /* dec. full */
          down(mutex);         /* enter cs */
          remove_item(item);   /* take */
          up(mutex);           /* leave cs */
          up(empty);           /* inc. empty */
          consume_item(item);
        }
   }
```

נעצור לרגע עם בעיית ה:PC, <u>ננסה לממש סמפור שלילי בעזרת סמפור בינארי</u>: תחילה נגדיר מחדש את הסמפור הבינארי: (השינוי הינו ב:up בלבד), ולכן אם בוצע up:
value = 1, אם value כבר שווה ל:1 - הרי ש:up "התבזבז".
בנוסף: אם ישנו תהליך חסום - הוא מתעורר.

Initially:

Binary Sem. Binary Sem.

```
mutex = 1, delay = 0, S.value = 1
```

לאחר הגדרה זו - נממש את הסמפור השלילי בעזרת 2 סמפורים בינאריים, ולכן יהיה לנו משתנה משותף s המייצג את הסמפור השלילי, ו:mutex, delay סמפורים בינאריים (על פי ההגדרה החדשה). כאשר mutex אחראי על הגישה ל:s, ו:delay, הוא מעין תור - לכל התהליכים שלא יכולים להיכנס ל:cs (צריך לשים לב - מדובר ב:CS "פנימי", ז"א: ה:CS שנותן גישה לשנות את s.value).

```
down(S): (NOT ATOMIC)
   down(mutex); // enter cs
   S.value--;
   if(S.value < 0){ // if no coins
        up(mutex);  // release cs
        down(delay); //join blocked queue
   }
   else up(mutex); // release cs
```

```
up(S): (NOT ATOMIC)
   down(mutex);     //enter cs
   S.value++;
   if(S.value ≤ 0){ // need 0 for -1
        up(delay); //wakeup one
   }
   up(mutex) //exit cs
```

אך כרגע המימוש הנ"ל אינו נכון, ניתן תרחיש: ההגדרה הראשונית:

$s.val = 1, mutex = 1, delay = 0$

כעת מגיעים $p_1 \dots p_4$ תהליכים ומבצעים down(s), נניח כי $p_2 \dots p_4$ מושהים ע"י ה:OS בין $line_1, line_2$ (ז"א: נעשה up(mutex) אך עדיין לא נעשה down(delay). וכעת $s.val = -3$.

כעת מגיעים $p_5 \dots p_7$ תהליכים המבצעים up(s). היות ו: $s.val = -3$ כולם מבצעים up(delay) ו-2 מהם הולכים לאיבוד (רק הראשון ישנה את ערכו ל-1).
ה:OS מחזיר לפעולה את p_2, ומבצע down(delay), כעת p_3, p_4 מבצעים down(delay) ונחסמים לנצח.
בעצם קיבלנו ש: $\#ups \neq \#down$ שהרי 2 ups הלכו לאיבוד.

במהלך השיעור הראה ד"ר חורב אופציה נוספת - אך גם היא לא עלתה יפה. מכיוון שהוכחת ה:DL הינה ארוכה, לא סיכמתי אותה כאן, באופן כללי: הבעיה של הצעה זו - שנוצר מצב שאף אחד לא עושה up(mutex) וכולם נחסמים בחוץ (מומלץ לנסות ולהבין - עמ' 22-31).

אופציה שלישית - נספור את כמות ה:delay, ז"א: אם באופציה 2 עשינו "חסימות", אך לא ידענו כמה אנחנו חוסמים, כאן - אנחנו נספור את כמות החסומים. לשם כך נשתמש במשתנה נוסף: wake. אך גם אופציה זו אינה טובה, נעשה תרחיש:

$s.val = 0, mutex = 1, delay = 0, wake = 0$

$p_1 \dots p_7$ מבצעים down(s), הם נחסמים ב:$line2$ (ז"א: down(delay))
$p_8 \dots p_{11}$ מבצעים up(s), ונניח כי ה: ups משחררים את $p_1 \dots p_4$. כלומר: $p_5 \dots p_7$ עדיין תקועים ב:$l2$.
גם כאן $\#ups \neq \#downs$.

FROM BINARY TO COUNTING - ATTEMPT 5

Barz, 1983

mutex=1, delay=min(1, init_value), value=init_value

```
down(S): (not atomic)
  down(delay);
  down(mutex);
    S.value--;
    if (S.value>0){
      up(delay);
    }
    up(mutex);
```

```
up(S): (not atomic)
    down(mutex);
    S.value++;
    if(S.value == 1) {
        up(delay);
    }
    up(mutex);
```

כעת נגיע למימושים שכן עובדים, בשיעור הובאו 2, אך בפועל התייחסנו רק לאפשרות הקלה יותר. בנוסף: לא הוכחנו את המימוש, אלא רק הבאנו דוגמת הרצה.

חשוב לשים לב - המימוש מאוד דומה למימוש הראשון, רק הפעם - אנחנו דואגים לכך שלא ניכנס ל:DL.
עיקר העניין - אנחנו מתייחסים לתהליך בודד, ולא לקבוצה, ולכן לא יקרה מצב בו נאבד up או down כלשהו.

מצגת מס' 5

מצגת זו עוסקת במוניטורים.

בגלל שראינו שסמפורים הם עניין "מסוכן" (למשל: החלפת סדר ב:down-up עלול לגרום ל:DL), ננסה להעביר את האחריות לשפת התכנות בעצמה.

מוניטור עובד בצורה הבאה - ישנו class המוגדר להיות מוניטור, ובו פרוצדורות (למשל - pro,con).

נראה כעת איך ניתן לפתור את בעיית ה:PC בעזרת מוניטור, הבעיה היא (למשל): אם ה:pro נמצא ב:block בגלל שה:buffer מלא, כיצד מעירים את con? לכן נכריח את ה:pro לשחרר את המוניטור עד שיהיה מקום פנוי, con יעיר את pro כאשר התפנה מקום. pro מבקש שוב את המוניטור, וכן הלאה.

למדנו 2 גישות למוניטור: הראשונה של Hoare - ברגע שפרוצדורה "מתעוררת" - היא זו שתרוץ. כמובן שמאוד קשה להגיע למצב שכזה.
גישה שניה - נקראת: Hansen, והיא קובעת שלא מוכרח - שמי שהתעורר - הוא זה שירוץ ראשון, אלא התהליך ששלח wakeup - יהיה חייב לצאת מהמוניטור, מכאן - ששליחת ה:wakeup תתרחש <u>רק בסוף התהליך</u>.

ב:Java ישנו class הנקרא condition, והוא משמש כמוניטור. אך בשביל שנוכל להשתמש בו יש צורך להגדיר משתנה מסוג Lock, באופן הבא (הסבר מפורט יותר בהמשך הסיכום):

```
Lock lock = new ReentrantLock();
```

עבור כל תהליך נגדיר condition באופן הבא:

```
Condition first = lock.newCondition();
Condition second = lock.newCondition();
```

בתוך הקונדישין ישנם 2 אפשרויות - wait או signal\signalAll, הדומה ל:wakeup שראינו לעיל. בעמ' 11 ניתן לראות דוגמת פסאדו לבעיית PC עם מוניטור (pro ו:con בודד), אך חשוב לשים לב - כי עבור k פרודיוסרים, ו-m קנוסיומרים - הפתרון אינו נכון, שכן ראינו בכיתה תרחישים שגורמים ל:DL (2 תרחישים - $m < k$, בנוסף: pro תקוע, ועוד לפני שהוא נכנס לפעולה - pro אחר עוקף אותו).

דוגמא למימוש הבעיה הנ"ל בעזרת Condition:

נתחיל עם מחלקת consumer:

```
public class consumer extends Thread{
	private final static int size = 5;
	Lock lock;
	Condition thisCon;
	Condition secondCon;
	LinkedList<Integer> buffer;
	int sp[];

	public consumer(Lock lock,
			Condition thisCon,
			Condition secondCon,
			LinkedList<Integer> buffer,
			int sp[]) {
		this.lock = lock;
		this.thisCon = thisCon;
		this.secondCon = secondCon;
		this.buffer = buffer;
		this.sp = sp;
	}
```

```
        public void run() {
                int cur = 0;
                while (cur < 50) {
                        lock.lock();
                        try {
                                while (sp[0] == 0) thisCon.await();
                                buffer.removeFirst();
                                sp[0] = buffer.size();
                                Thread.sleep(300);
                                System.out.println("print Buffer in con -> " + buffer.toString());
                        }
                        catch(Exception e) {}
                        finally {
                                if (sp[0] == 1) secondCon.signalAll();
                                lock.unlock();
                        }
                        cur++;
                }
        }
}
```

מחלקת ... מאוד דומה למחלקה הנ"ל - ההבדל הוא שאנחנו מכניסים איברים לרשימה (ההבדלים מסומנים בצהוב):

```
public class producer extends Thread{

        private final static int size = 5;
        Lock lock;
        Condition thisCon;
        Condition secondCon;
        LinkedList<Integer> buffer;
        int sp[];

        public producer(Lock lock,
                        Condition thisCon,
                        Condition secondCon,
                        LinkedList<Integer> buffer,
                        int sp[]) {
                this.lock = lock;
                this.thisCon = thisCon;
                this.secondCon = secondCon;
                this.buffer = buffer;
                this.sp = sp;
        }

        public void run() {
                int cur = 0;
                while (cur < 50) {
                        lock.lock();
                        try {
                                while (sp[0] == size) thisCon.await();
                                buffer.add(cur);
                                sp[0] = buffer.size();
                                Thread.sleep(300);
                                System.out.println("print Buffer in pro -> " + buffer.toString());
                        }
                        catch(Exception e) {}
                        finally {
                                if (sp[0] == size) secondCon.signalAll();
                                lock.unlock();
                        }
                        cur++;
                }
        }
}
```

כל מה שנשאר - להפעיל את התהליכים, חשוב לשים לב - לסדר הכנסת ה:Condition-ים.

```
public static void main(String[] args) {

	Lock lock = new ReentrantLock();
	Condition conCon = lock.newCondition();
	Condition proCon = lock.newCondition();

	LinkedList<Integer> buffer = new LinkedList<Integer>();
	int sp[] =new int[1];
	sp[0] = buffer.size();

	Thread con = new consumer(lock, conCon, proCon, buffer, sp);
	Thread pro = new producer(lock, proCon, conCon, buffer, sp);

	pro.start();
	con.start();
}
```

באותו אופן ניתן לממש את הבעיה בעזרת פונקציות synchronized, בעצם ניצור מחלקה בשם מוניטור שתכיל את כל הפונקציות המסונכרנות, בהן התהליכים ישתמשו, נתחיל במחלקת Monitor:

חשוב לשים לב - שכאן אין צורך להגדיר את sp כאובייקט, משום שהוא נמצא באותה מחלקה.

```
public class Monitor {

	public static final int size = 5;
	LinkedList<Integer> buffer;
	int sp;

	public Monitor() {
		buffer = new LinkedList<Integer>();
		sp = buffer.size();
	}

	public synchronized void insert(int n) {
		try {
			if (sp == size) this.wait();
			Thread.sleep(300);
			buffer.add(n);
			sp = buffer.size();
			System.out.println("insert Print -> " + buffer.toString());
			if (sp == 1) this.notifyAll();
		} catch (Exception e) {}
	}

	public synchronized void remove() {
		try {
			if (sp == 0) this.wait();
			Thread.sleep(300);
			buffer.removeFirst();
			sp = buffer.size();
			System.out.println("remove Print -> " + buffer.toString());
			if (sp == size -1) this.notifyAll();
		} catch (Exception e) {}
	}
}
```

נעבור לטרדים עצמם:

```
public class consumerThread extends Thread{

	Monitor monitor;

	public consumerThread(Monitor m) {this.monitor = m;}

	public void run() {
		int cur = 0;
		while (cur < 100) {
			monitor.remove();
			cur++;
		}
	}
}

public class producerThread extends Thread{

	Monitor monitor;

	public producerThread(Monitor m) {this.monitor = m;}

	public void run() {
		int cur = 0;
		while (cur < 100) {
			monitor.insert(cur);
			cur++;
		}
	}
}
```

הרצת התוכנית: כל תהליך - יקבל את המוניטור:

```
public static void main(String[] args) {

	Monitor monitor = new Monitor();
	Thread cons = new consumerThread(monitor);
	Thread prod = new producerThread(monitor);

	cons.start();
	prod.start();
}
```

- פרוצדורה המפעילה פרוצדורה בתוך אותו מוניטור - החשש: DL, הרי אין אפשרות ששתי פרוצדורות ירוצו בו זמנית, ובעצם הפרוצדורה תכניס את עצמה ל:DL. כל זה ב:C\C++, אך ב:Java דאגו לעניין, ואפשר לקרוא לפרוצדורה מתוך פרוצדורה. באותה מידה אפשר לשאול - עבור 2 מונטורים נפרדים, שאחד קורא לשני (פרוצדורה לפרוצדורה).

כעת נחזור לחומר הקודם דרך מוניטורים - נממש מוניטור בעזרת סמפורים. ז"א:

```
Semaphore mutex = 1 /* control access to monitor */
Semaphore c = 0 /* A queue to hold waiting processes */
int count = 0      /*  number of processes waiting for monitor */
enter_monitor():
   down(mutex)        /*  only one-at-a-time */

leave():
   up(mutex)           /*  allow other processes in */

leave_with_signal()
   if(count == 0):
         up(mutex)    /*  no waiting, just leave.. */
   else:
         count-=1
         up(c)   /* no need to release mutex we wake up the next process */

wait(): /*  block on a condition  */
   count+=1     /*  count waiting processes */
   up(mutex)   /*  allow other processes  */
   down(c)      /*  block  */
```

- simple lock - משתנה שאפשר להסתכל עליו כמשתנה בינארי אטומי, בעל 2 מצבים: lock, unlock. ו-2 פעולות אטומיות:

acquire - אם המצב "פתוח" - תגדיר את המצב ל:lock
- אם המצב "סגור" - תחסום עד שהמצב ישתנה ל"פתוח".

release - אם המצב הינו "סגור" - תגדיר את המצב ל:unlock
- אם המצב "פתוח" - אל תעשה דבר.

דוגמא לשימוש:

```
l = lock()
l.acquire()
<CS>
l.release()
```

כמובן שיש צורך לעטוף את הדברים ב:try, catch, וגם finally. בעצם simple lock דומה מאוד לסמפור בינארי.

יכולה להיווצר בעיה - כאשר פרוסס יפעיל ()lock פעם שניה. על מנת למנוע בעיה זו נוצר Reentrant locks, שלא מבצע שום דבר אם קוראים יותר מפעם אחת ל:lock.

נקודות חשובות בשימוש של condition variables:

א. חייבים לעשות finally ובתוכו release.
ב. כנ"ל לגבי notify השייך ל:condition

בעמ' 26 ניתן לראות דוגמא הממחישה מדוע יש צורך ב"בעלות", ז"א: ההפעלה של wait,notify יהיו מתוך ה:CS, אחרת - ניתן להגיע ל:DL.

ההבדל בין notify\notifyAll - אם ידוע שיש רק תהליך בודד שמחכה - מספיק notify. בנוסף: בקריאה ל:notifyAll - אין אפשרות לבחור איזה תהליך יתעורר. ישנם מצבים בהם אנחנו חייבים להשתמש דווקא ב:notifyAll, שאם לא נעשה זאת - יש אפשרות להיכנס ל:DL, במהלך ההרצאה נתן ד"ר חורב תרחיש של PC, עם buffer בגודל 1 (ריק), ו-2 cos, ו:pro בודד. בעצם - אם נשלח notify ייתכן ש:con1 ינסה להעיר את con2 ונקבל DL, כאשר ה:buffer ריק.

- events - משמעותו - להמתין לדבר מסוים שיקרה. בדומה ל:condition - גם כאן יש wait, אך ה:wait תלוי במשתנה בולאני - בעצם נחכה עד שהמשתנה יהפוך ל:ture (המשתנה מאותחל ב:false), ויש 2 פונקציות לשנות את המצב של המשתנה: set - הופך ל:true.
clear - הופך ל:false.
אובייקט מסוג זה אינו קיים ב:Java, אך כן ב:C++,python.

חשוב לשים לב: condition, event הם 2 דברים שונים לחלוטין, שכן condition הינו הגנה ל:CS, מה שאין כן ב:event, שרק ממתין למאורע, אך לא מעבר.

סוג נוסף של condition הקיים ב:Java הינו הגדרה של synchronized - או של קטע קוד או של משתנה, אך צריך להיזהר עם משתנים, שכן מחרוזות נמצאות בזיכרון באותו מקום, ויש חשש שמחרוזות זהות יוגדרו להיות synch, ואנו לא מעוניינים בכך (עמ' 34).

כללים לגבי synch:

א. אין להגדיר synch בתוך synch, כאשר אומרים wait על הפנימי.
ב. ניתן לעשות synch בתוך synch, אך יש לשמור על סדר קבוע - מי חיצוני ומי פנימי.

במצגת זו נראה בעיות קלסיות - מילוליות, במצגת הקודמת סגרנו את בעיית ה:PC. נעבור כעת לבעיות הבאות:

בעיית הספר הישן (sleeping barber)
נתונים: ישנה מספרה עם ספר אחד והרבה לקוחות.
מספר הממתינים הינו תור סופי. אם התור מלא - לקוחות נוספים שמגיעים עוזבים.
יש רק כסא אחד שאפשר להסתפר בו.
אם לא קיימים לקוחות שממתינים - הספר ישן, ולהפך - לקוחות שממתינים - ישנים.

בעמ' 4 ישנו קוד שאינו תקין, הראנו זאת - כאשר ישנו מצב ש-2 לקוחות ישבו ביחד על כיסא הספר. וזאת משום שה:OS יכול לחסום לקוח בדרך לכיסא, ובעצם הלקוח לא יסתפר, הלקוח הבא - גם כן יצליח להגיע לכיסא, ובעצם נגיע למצב הבעייתי.

הפתרון: אין אפשרות לספר לצאת - כל עוד הלקוח לא יצא, נעשה זאת בעזרת סמפור בינארי נוסף.

SLEEPING BARBER

```
#define CHAIRS 5
semaphore customers = 0; // number of waiting customers
BinarySemaphore barbers = 0; // number of available barbers: either 0 or 1
int    waiting = 0; // copy of customers for reading
Semaphore mutex = 1;    // mutex for accessing 'waiting'
BinarySemaphore  sync = 0;   // synchronising the service operation
```

```
void  barber(void) {
   while(TRUE)   {
      down(customers); // block if no customers
      down(mutex); // access to 'waiting'
      waiting = waiting - 1;
      up(barbers); // barber is in..
      up(mutex);  // release 'waiting'
      cut_hair();
      down(sync) //wait for customer to leave
   }
}
```

```
void  customer(void) {
   down(mutex);  // access to `waiting'
   if(waiting < CHAIRS)     {
      waiting = waiting + 1; // increment waiting
      up(customers); // wake up barber
      up(mutex); // release 'waiting'
      down(barbers); // go to sleep if barbers=0
      seat_in_chair();
      up(sync); //synchronise service
      }
   else  {
      up(mutex);  /* shop full  .. leave */
   }
}
```

נעבור לבעיה הבעיה: readers - writers
ישנן 2 קבוצות של תהליכים - קוראים וכותבים.
מותר לכמה קוראים להיכנס ביחד ל:CS, אך אסור שהגישה תתאפשר ל:W עם R, וגם לא לשני W גם יחד.
בעמ' 7 ניתן לראות פתרון ראשוני, אך גם הוא בעייתי, משום שאין בו SF: התרחיש שעלול לקרות: זרם בלתי פוסק של readers לא יאפשר לאף writers להיכנס. הפתרון אליו נרצה להגיע - אם קיים W שמחכה - לא יוכלו R חדשים לעקוף אותו. אך גם כאן - אנחנו בבעיה, שכן ה:OS מחליטה מי יחזור לרוץ, וייתכן כי בכל פעם R יקבל זמן ריצה ושוב לא נקבל SF, ולכן יש להכריח שלא יהיו 2 R על rdb, ולכן נוסיף סמפור נוסף mutex2:

```
int readers = writers = 0
BinarySemaphore Rmutex, Wmutex, Mutex2 = 1;
BinarySemaphore Rdb, Wdb = 1;
```

```
void reader(void){
  while(TRUE){
    down(Mutex2)
      down(Rdb);
        down(Rmutex)
          readers+=1;
          if(readers == 1)
            down(Wdb);
        up(Rmutex);
      up(Rdb)
    up(Mutex2)
    read_data_base();
    down(Rmutex);
    readers -= 1;
    if(readers == 0)
      up(Wdb);
    up(Rmutex);
  }
}
```

sole change

```
void writer(void){
  while(TRUE){
    down(Wmutex);
    writers +=1;
    if (writers == 1)
      down (Rdb)
    up(Wmutex)
    down(Wdb)
    write_data_base()
    up(Wdb)
    down(Wmutex)
    writers -=1
    if (writers == 0)
      up(Rdb)
    up(Wmutex)
```

ניתן לפתור את הבעיה הנ"ל בעזרת מוניטורים, ד"ר חורב ביקש שנעשה זאת בבית, לאחר כשעה של נסיונות - הבנו שהדבר לא כ"כ פתיר...

נעבור לבעיה הבאה: בעיית המנהרה.
הבעיה מתארת מצב בו קיימת מנהרה בעלת נתיב אחד בלבד, כאשר יש צורך לעבור משני צדי המנהרה.
אם ישנה תנועה של נתיב אחד - הנתיב השני מחכה, זהו בעצם מקרה מיוחד של RW.

בשלב הזה, נאור ואנוכי התייאשנו מהמצגת, ניתן לשים לב שבכל הבעיות - ידענו היטב לתאר את הבעיה, אך לא את פתרונה... נתפלל כולנו שלא יהיו שאלות על המצגת הזו... זה הזמן לעבור למצגת הבאה...
נ.ב. אם אכן תהיה שאלה על הבעיות הנ"ל - נזכר במשפט "אפילו חרב חדה מונחת על צווארו של אדם - אל ייתיאש מן הרחמים"...

מצגת 7

מצגת זו עוסקת ב: scheduling - לוח זמנים, ובמילים אחרות - תזמון.
נגדיר מספר הגדרות הקשורות לאלגו' תזמון:

א. הגינות - כל פרוסס יקבל נתח זמן שווה מה: CPU
ב. יעילות - לנצל את המשאבים עד תום, אין חוסר מעש כל עוד יש CPU.
ג. זמן תגובה
ד. TurnAround time - זמן ממוצע שפעולה לוקחת (משלב היצירה ועד לסיום שלה)
ה. waiting time - לצמצם כמה שאפשר את זמן ההמתנה ב: ready.
ו. Throughput - הכוונה לכמות המשימות אותם מסוגל ה: CPU ביחידת זמן מסוימת

בעמ' 4 ישנה דוגמא לפרוססים שצריכים לרוץ, והמע' צריכה לתזמן:

The setting:

1. 5 interactive jobs I1,…,I5 and one batch job B.
2. Each Ij:
   1. 10% CPU
   2. 20% Disk I/O
   3. 70% Terminal I/O
   4. 10 sec.
3. B:
   1. 90% CPU
   2. 10% Disk I/O
   3. 50 sec.

נבדוק 2 אופציות:

א. נפעיל תחילה את I ולאחר מכן את B, ולכן:

$$\%\,CPU = \frac{(10_{sec}*0.5_{\%CPU} + 50_{sec}*0.9_{\%cpu})}{60_{totalRunTime}} = 83\%$$

$$TA_{time} = \frac{(10_{sec}*5_{pro'} + 60_{sec}*1_{pro'})}{6_{AllPro'}} = 18.33_{sec}$$

ב. נפעיל את B ובכל מקום שנותר מה: CPU נפעיל תהליך מ: I, ולכן:

$$\%\,CPU = \frac{(50_{sec}*(0.9_{\%cpu}+0.1_{\%cpu}))}{50_{totalRunTime}} = 100\%$$

$$TA_{time} = \frac{(10_{sec}+20_{sec}+30_{sec}+40_{sec}+50_{sec}+50_{sec})}{6_{AllPro'}} = 33_{sec}$$

כמובן שאין אפשרות לומר - שאחד טוב מהשני, אלא תלוי מה העדפה, ז"א: אם אני מעדיף - ניצול CPU הרי שהאפשרות השניה עדיפה, ואם אני מעדיף TA - האפשרות הראשונה עדיפה.

ישנם שני סוגי תזמון:

א. Preemptive - כמעט כל המשימות ניתנות להשהייה
ב. NonPreemptive - לא ניתן להשהות משימות, כמו משימות אטומיות.

אלגוריתם First come first served (FCFS) - הראשון שנכנס - הוא זה שמקבל זמן ריצה, ועד שאינו מסיים - אף אחד לא רץ.
ראה במצגת דוגמא - עמ' 9-10, ההבדל בסדר הגעת הפרוססים המשפיע על זמן ההמתנה הממוצע.

אלגו' משופר יותר - SJF - short job first. אלגו' זה פותר את הבעיה הנ"ל - אך יוצר בעיה אחרת - כיצד נוכל להעריך כמה זמן ייקח לכל פרוסס לרוץ. (בעיה נוספת שעלולה להתעורר - זרם בלתי פוסק של פעולות קצרות יגרום להרעבת הפעולות הגדולות).

הפתרון לבעיה הנ"ל - בניה של שיערוכים, כי בסופו של דבר המע' יחסית צפויה. בשיעור ראינו 2 אופציות לבניית שיערוך

א. שיערוך הזמן של הפרוסס הקרוב - חישוב ממוצע של כל הזמנים שלקחו הפרוססים עד עכשיו.
ב. בשיטת "למידת מכונה" (לא הרחבנו על כך). (עמ' 13-14)

ניקח את האלגו' SJF ונשלב בו את האפשרות - לעצור תהליכים באמצע (Preemptive). חשוב לשים לב: כעת אנחנו בוחנים - כמה זמן נותר לכל תהליך, והקצר ביותר - ירוץ ראשון. ניתן לראות בעמ' 16-17 דוגמאות.

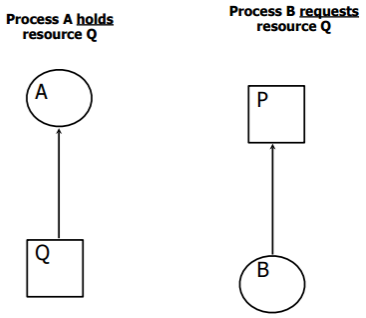
נעבור לאלגו' Round Robin (RR). זהו אלגו' המנסה לספק זמן ריצה לכל התהליכים ללא העדפה, מעין משחק הכיסאות: הפרוססים נשמרים בתוך רשימה, עבור כל פרוסס יש זמן קבוע לרוץ, מתחילים מראש הרשימה ונותנים את הזמן הקבוע לרוץ, כעת ממשיכים עד לזנב, וחוזר חלילה (רשימה מעגלית). כמובן שבאלגו' זה יש לקחת בחשבון את הזמן שלוקח לבצע switch בין התהליכים.

במהלך ההרצאה רצה ד"ר חורב לשכלל את RR, היו כמה נסיונות, אך בפועל לא נגענו בהן (עמ' 21-24), בכללי: לעבוד עם טבלת גיבוב, עדיפות תשתנה בצורה דינאמית, ואפשרות אחרונה - לשלב בין האלגו' דלעיל.

במצגת זו נחזור לנושא ה:DL.

ישנן כמה דרכי התמודדות עם DL:

1. הימנעות
2. הקצאת משאבים בצורה זהירה
3. לא לעשות דבר - זה מה שקורה בדרך כלל, ובעצם זה מה שעושה ה:OS - מחסלת את הפרוסס.



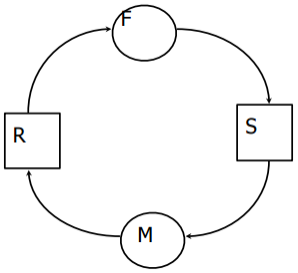
כיצד נזהה DL?, תחילה נגדיר כמה דברים:

עיגול מייצג פרוסס, ריבוע מייצג משאב.

חץ ממשאב לפרוסס אומר כי הפרוסס מחזיק את המשאב.

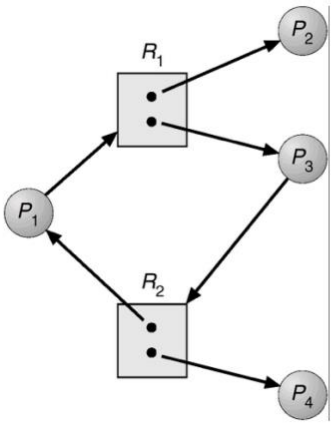
חץ מפרוסס למשאב אומר כי הפרוסס מבקש להשתמש במשאב.

דוגמא ל:DL בקוד:



בעצם אפשר לומר - כשיש מעגל - יש DL.

תחילה ראינו דוגמא עבור resource עם גישה אחת, החל מעמ' 25 התחלנו לדבר על 'res עם יותר מגישה אחת (כמו למשל - קריאה וכתיבה מהדיסק), כאן כבר מעגל אינו בהכרח יוצר DL, ובעצם נצטרך למצוא מעגל עבור כל נקודת גישה במשאב, למשל:



Cycle:

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

But:

P4 may release R2 which will grant P3 its request

ניתן להריץ תהליכים בצורה של מטריצה ולגלות האם יש DL או לא, מספר הגדרות:

E - סך המשאבים הקיימים

A - סך המשאבים הפנויים

C - פרוססים שמשתמשים בחלק מהמשאבים

R - פרוססים שמבקשים להשתמש במשאבים הפנויים (בהתאמה).

(דוגמא בעמ' 36 והלאה)

נעבור להגדרת "מצב בטוח": מצב בטוח הינו אלגו' ללא DL. בנוסף: ישנו "לוח זמנים" מהמצב הנוכחי ועד לסיום כל התהליכים.

אנחנו מניחים כי יש לנו ידע מקדים - מה המקס' בקשות של res עבור כל תהליך (שכמובן הנחה זו אינה מציאותית), ז"א: has - מה שקיים, max - מה שנדרש. free - כמה res פנוי קיים למערכת. בנוסף: תהליך שקיבל את כל הנדרש - מת, ומחזיר את כל ה:res הקיים תחתיו.

'Dijkstra - Banker's algo - זהו אלוג' להגיע למצב בטוח, ז"א: כמו בנק בעיירה קטנה, הבנקאי ייתן הלוואה רק למי שבטוח - יחזיר לו, אך מעבר לכך - הלוואה תינתן, רק בתנאי שנשאר לבנק מספיק כסף בשביל לתת הלוואה מספקת לאדם נוסף.

Coffman - בדומה ל:MC שאנחנו מכירים (אך עם משמעות שונה) - לא ניתן לאף פרוסס משאבים - אם יש פרוסס שכבר לקח חלק. כל שאר הפרוססים מחכים בצורת "מעגל".

במצגת זו ובמצגת הבאה נדבר על Memory Management.

תחילת המצגת הינה הקדמה - דיברנו על סוגי הזיכרון השונים במחשב, ובעצם הגענו למסקנה כי המקום ב:RAM לא תמיד מספיק, ולכן נצטרך להעביר חלק ממנו לדיסק הקשיח, אלא שבכל פעם - נצטרך לעשות Swap וצריך להתחשב במחירו.

בתחילת הדרך - כל מתכנת כתב ישירות ל:RAM, אך זה כמובן בעייתי:

א. כל מתכנת חייב להכיר את מבנה ה:RAM וזה לא פשוט
ב. החלפת RAM תדרוש החלפת הקוד
ג. גישה ל:RAM עלולה להיות מסוכנת.

פתרון: address space לכל פרוסס, וכך כל פרוסס לא תלוי בכל ה:RAM, אלא במה שקיים אצלו.

בעיה חדשה: אין מקום לכל ה:AS ב:RAM, מכאן מגיעים לנושא של swapping.

אם נעשה swap שרירותית - ייתכן וניצור "חורים" ב:RAM, ולכן צריך לחשוב איך להקצות זיכרון כך שנוכל להימנע עד כמה שאפשר ממצב זה.

הצעה ראשונה: חלוקת ה:RAM למחיצות, כשכל מחיצה מגדירה את גודל הפרוסס. בשלב הבא - נרצה להתחשב בפרוסס שיכול גם לגדול, ולכן נעניק לכל מחיצה - מקום בו הפרוסס יוכל להתרחב. אך אופציה זו מסוכנת, שכן יש צורך לעקוב אחרי התהליך ולראות שאינו חורג.

כיום קיימות 2 גישות:
bit maps - יצירה של מערך בצורת מטריצה, המייצג את המקום הפנוי (0) והלא פנוי (1) ב:RAM. מצד אחד - שאילתא על מקום פנוי - זולה, אך מצד שני - טבלה גדולה קשה לתפעול.
linked lists - אותו רעיון רק בצורה של רשימות מקושרות: תחילה נציין בכל רשימה האם מדובר בפרוסס או ב"חור", ולאחר מכן - מהי תחילת הכתובת, וכמה הוא תופס.
עבור שיטה זו - יש כמה שיטות להכנסת פרוסס - השיטה הראשונה - first fit - מציאת הראשון שמתאים, כמובן שזו שיטה מהירה, אך לא מספיק טובה, לכן חשבו לחפש מהמקום בו הכנסנו את הפרוסס האחרון (next fit), אך מתברר שזו שיטה שעובדת פחות טוב מ:first fit. לכן פיתחו את השיטה שמחפשת את החור הכי קטן שיתאים (best fit), כמובן שמחיר החיפוש מאוד יקר, ומעבר לכך - יכולים להיווצר חורים קטנים יותר שיהיה קשה מאוד למלא אותם. לכן חשבו על אלגו' הפוך - worst fit - להכניס את התהליך הקטן ביותר בחור הגדול ביותר, וכך לא יווצרו חורים קטנים.
למסקנה - הדבר הטוב ביותר שאפשר לעשות - רשימה נפרדת לפרוססים ורשימה נפרדת ל"חורים".

חשוב לזכור - העיסוק שלנו הוא ב"חורים שנוצרים בין פרוסס לפרוסס, אנחנו לא מתעסקים עם חורים "פנימיים" למשל. במצגת ישנה דוגמא להעברות של פרוססים ממקום למקום, ניתן לראות שאפשר לשלם מחיר גבוה לעומת מחיר נמוך, תלוי כיצד נבחר להחליף בין "חור" לבין תהליך.

לאור כל הפתרונות שראינו - אף אחד לא נתן לנו תשובה מספיק חזקה - ולכן כיום הפתרון הוא: virtual memory, ובעצם - אנחנו משתמשים בזיכרון וירטואלי, ולא מוגבלים לכמות הזיכרון ב:RAM.

כעת נגדיר את הרעיון:

א. לכל פרוסס יש את ה:AS שלו.
ב. כל AS מחולק לכמה Pages.
ג. ה:Pages ממופים לזיכרון הפיזי, ובעצם לא כל ה:P נמצאים ב:RAM
ד. אם פרוסס מבקש תהליך שלא נמצא ב:RAM, תחילה - יש לטעון אותו לתוך ה:RAM (ע"י OS)

כמובן שכעת המשימה - ליצור קשר בין הזיכרון האמיתי לזיכרון הוירטואלי, הפתרון: MMU - מה שמנהל את הזיכרון.
אם נתבקש להגיע ל:P שאינו טעון - נקבל כשל (page fault), ובעצם המערכת תעבור לטעינת ה:P המבוקש.

מכאן אנחנו עוברים לחלק השני (מצגת מס' 10), כעת נרצה לטעון Page שלא קיים, אם נתבקשנו לקבל אותו. כמובן - שנרצה לעשות זאת בצורה ה"זולה" ביותר.

באופן כללי - כאשר המחשב מתחיל את העבודה - ישנו שינוי דרסטי בתהליכים אותם הוא עושה, אך בהמשך - אנחנו נכנסים לתהליך של התכנסות - ז"א: המערכת מתייצבת, ואין צורך בהרבה שינויים. אם דבר זה לא מתרחש - זה נקרא thrashing.

כעת נרצה למצוא אלגו' שיחליט איזה P להוציא בשביל להכניס P הנדרש.
האלגו' הראשון - אופטימלי, מה שלא קיים במציאות, כי הוא יכול לדעת כמה פעמים המע' נצרכת אליו. הסיבה לאלגו' זה - רק לשם השוואה, בשביל לראות איך שאר האלגו' עובדים.

האלגו' הראשון: not recently used (NRU) - האחרון שהיה בשימוש - יוצא. נגדיר עבור כל page את המשתנים הבאים (ביטים)

R - האם ה:P נקרא

M - האם ה:P השתנה

עבור כל פרוסס - נציין האם הוא R או M או גם שניהם, ונעדיף לזרוק את מי שלא R ולא M.
אלגו' זה אמנם קל למימוש, אך לא כל כך יעיל.

האלגו' הבא: FIFO - כמובן שזה לא הרעיון הכי טוב, והוא דיי נאיבי.
במהלך ההרצאה נתן ד"ר חורב השוואה בין האופטימלי לבין ה:FIFO ויצא כי 4|7 לטובת האופטימלי.

ננסה לשפר את האלגו' של FIFO ולתת צ'אנס עבור כל תהליך, נעשה זאת בעזרת ביטים. בעצם נשלב זאת עם ה:NRU:
נבדוק האם R הינו 1, אם כן - ניתן הזדמנות שניה - ע"י העברה לסוף התור.
חשוב לשים לב: האלגו' צריך לספק פתרון עבור מקרי קצה, כגון: כל ה:R ביט הינם 1. מעבר לכך - תחזוק התור - גם כן עולה כסף.

אלגו' נוסף: 'Clock replacement algo - כל התהליכים ישמרו במבנה מעגלי - וראש הרשימה יצביע על ה:P הכי ישן, אם R של המיקום הינו 0 - מתבצעת החלפה, אחרת - מאפס את R, ועובר לאיבר הבא (מעביר את המחוג קדימה).

נעבור לאלגו' הבא: Least recently used (LRU) - נוציא את הפרוסס שהיה בשימוש בזמן הכי רחוק. כמובן שמחיר של אלגו' מסוג זה יהיה יקר מאוד.

ננסה כעת ליצור Not frequently used (NFU) המתקרב ל:LRU אך לא דורש מחיר גבוה כל כך: תיחזוק של counter עבור כל P. בכל פעם שמתרחש Clock interrupt מע' ההפעלה סורקת את כל ה:P שנמצאים ב:RAM ומוסיפה את ה:R ביט ל:counter. כמובן שנחליט להוציא את ה:counter הנמוך ביותר.

כאן יש לנו בעיה, נמחיש זאת בעזרת דוגמא: נאמר ויש פרוסס שהיה בו שימוש רב בתחילת הפעלת ה:OS, ונאמר שהוא תפס את כל המקום ב:RAM מלבד חלק קטן. כעת מגיע פרוסס נוסף - ותופס את ה:P הפנוי, אך בסופו של דבר החילוף יתבצע רק עליו - כי בו היה השימוש הקטן ביותר.

מתוך בעיה זו - נגיע לאלגו' - Ageing, בעצם לא נעשה counter רגיל, אלא מערך של ביטים, המייצג כל P. מצב זה לא יאפשר לתהליך מסויים - לצבור "כח" - על אף שלא היה בשימוש זמן רב.
השינוי הוא - במקום להוסיף ל: counter את r, אנחנו נעשה shift ימינה, ונוסיף את r. ככל שתהליך יצבור יותר אפסים - הוא יהיה המיועד ליציאה ראשון. (לעבור שוב על הדוגמא מהמצגת).