סיכום OS

20 ביוני 2022

1 שבוע 1:

1.1 הרצאה 1:

- **מטרות הקורס:**
  הכרת מערכות הפעלה. להבין מזה מערכת מחשב.

- **מערכת מחשב:** כוללת חומרה, אמצעי אחסון - SSD, מקלדת, עכבר, מעבד, זכרון.

- **מערכת ההפעלה:** היא תכנה שמשמשת כסביבת עבודה עליה יעבדו האפליקציות, למעשה היא מתווכת בין החומרה לתכנה.
  נשתמש בה כדי לנהל קבצים ולהשתמש בחומרה פרפיריאלית (מדפסת, מסך וכו), שאר העבודה של האפליקציות יקרו ישירות על המעבד.

- **הערות:**
  כשנכתוב תוכנית נשים אותה במיקום ההתחלתי של הדיסק $RAM[0]$.
  אם נרצה לכתוב למקום ספציפי בדיסק נצטרך להגיד זאת מפורשות.
  אם יוחלף הדיסק או החומרה, נצטרך לכתוב את התוכנית מחדש, כי התוכנית ספציפית לחומרה.
  בכל פעם יכולה לרוץ תוכנית אחת בלבד.

- **אבסטרקציה:** ממשק נקי כך שהמשתמש או האפליקציה יוכלו לדבר עם החומרה, תוך התעלמות מפרטים לא חשובים. (לדוג' - נותנת את האופציה להתייחס לקובץ בגודל משתנה, למרות שבזכרון הבלוק הינו בגודל קבוע).

- $SPOOL$**:** מבנה נתונים הנקרא $job\ pool$ שמנהל את המשימות כך שנוכל להשתמש במדפסת המסך והמחשב בו זמנית במקביל, וכך לשפר ביצועים.

- **יעילות המערכת:** נמדדת בעזרת זמן, מהו הזמן שלוקח לדברים להסתיים.
  דרך נוספת - מספר המגה ביט שלחנו בשניה (תפוקה).

- **ניצולת** $Utilizasion$**:** מספר בין 1-0 שמודד את נצילות המעבד וכמה זמן אחוז מהזמן הוא עובד ועושה **פעילות שימושיות** (הרצת פקודות של התכנית) .

- **תקורה** $Overhead$**:** מספר בין 1־0 שמודד פעולות שהמעבד מבצע **כהכנה לפעולה,** ולא הפעולה עצמה (הרצת מערכת ההפעלה).

- **מכונה וירטואלית:** בהמשך תכננו מערכת הפעלה כך שמערכת אחת תוכל לבצע כמה פעולות ולהריץ כמה תוכניות (לאו דווקא במקביל). **היו כמה שיטות:**
$multy\ program$**:** מערכת ההפעלה עבדה עד שקראנו להתקן פלט קלט (התקן פרפריאלי) ולאחר מכן עבדה לתוכנית הבאה.
$Time\ Sharing$**:** מערכת ההפעלה נכנסת לפעולה אחת כמה זמן, ומחליטה על איזו תוכנית לעבוד (מחלקת את הזמן בין תכניות באופן לא רציף).

- **בכדי להצליח במכונה וירטואלית הנחנו כמה הנחות:**
נניח כי תמיד קיים CPU, והוא תמיד יכול לעבוד וניתן להשתמש בו.
ה CPU תמיד יותר מהיר מ IO (התקני קלט פלט).
התקנים פרפיריאלים יכולים לגשת באופן ישיר לזיכרון ולקרוא ממנו.
ניתן לעשות יותר מפעולה אחת בזמן נתון.

- **דרישות ממערכת ההפעלה:**
יכולה לגשת להתקני קלט פלט.
יכולה לנהל את הזכרון (גודל, ואבטחה).
תזמון ־ מערכת ההפעלה צריכה להחליט איזה קובץ ירוץ כעת.
$spooler$ ־ ניהול איזה תכנית תשתמש באיזה התקן חיצוני.

- **משאבים שמערכת ההפעלה צריכה לנהל לפי חשיבות:** מעבד, זכרון, IO.

- $multy\ core$**:** שיטה של ריבוי מעבדים, כך יש יותר מעבדים לבצע כמה פעולות במקביל.

- $Virtualization$**:** יש לנו משהו מדומה שאותו אנו רוצים לשים על החומרה הממשית.
מרחיב או מחליף את הממשק העכשוי, כדי לממש פעולות שלא באמת קיימות.

- $Hyproisor$: שכבה שעובדת על החומרה הפיזית, ומעליה עובדות חומרות וירטואליות שמעליהן עובדת מערכת ההפעלה. כלומר ־ מערכת ההפעלה עובדת על חומרה וירטואלית שמעדכנת את החומרה הפיזית.

1.2 תרגול 1:

- **סייקלים של פקודה:** כל פקודה מבוצעת במחשב תוך חמישה סיבובים לערך. לכן בכל פעם שפקודה מסיימת את הסיבוב הראשון נכנסת אחריה פקודה חדשה. לכן ניתן לבצע על אוו CPU 5 פקודות "במקביל".

- **דיבוג עם** $Valgrind$**:**
1: הוולגריינד ידפיס לנו את הפלט שלנו, אם ניתקל בהדפסות שאינן הפלט באמצע הפלט אזי במקום זה מתרחשת הדליפה בזכרון.

2: כשיש דליפה נראה אותה ב $leak\ summery$.

3: לפעמים נרצה לדבג בעזרת פרינטים, לכן לאחר שנוסיף $print$ נוסיף את הפקודה $fflush(stdout)$ פקודה זו מכריחה את $C++$ להדפיס במקום ולא לשמור את ההדפסה לסוף הריצה.

שיטה נוספת: היא להוסיף "$\backslash n$" בסוף ההדפסה.

4: $segmentation\ fault$: תקרה כאשר ניגש למקום בזכרון שאנינו מורשים לגשת אליו.

1.2.1 $GDB - Debugger$:

- בכדי שנוכל להשתמש בדיבאגר אנו צריכים להוסיף ל $make\ file$ את הדגל $-g$.
- כדי להריץ את הדיבאגר מהטרמינל נכתוב את הפקודה הבאה: $gdb\ programName$.
- **פקודת** $watch\ x$: עבור משתנה x נוכל להגדיר לגדיבאגר לעקוב אחר המשתנה ולדווח לנו שערכו משתנה.
  למתקדמים: ניתן לשים תנאי והדיבאגר יעצור כשהתנאי מתקיים.
- $conditional\ breakpoint$: נשים ברייקפויינט ונקיש על מקש ימני. יפתח לנו פריט שניתן לבחור ממנו תנאי, כך הדיבאגר יעצר כשהתנאי יתקיים (שימושי ללולאות).
- $Exeption\ breakpoint$: ניתן להגדיר ברייקפויינט כך שהדיבאגר ייעצר כשיזרק אקספשן.
- $symbolic\ breakpoint$: ברייקפויינט שתיעצר כשנגיע לפונקציה ספציפית.
- $Tread$: כשנפצל את התכנית ל $thread$ נוכל לקפוץ בדיבאגר בין התכניות גם כן.

1.2.2 $Virtualization$:

- $Doker$ - **קונטיינר:** משמש כ VM רך יותר, כך שלא צריך להגדיר לו רכיבי חומרה מראש.

2 שבוע 2:

2.1 הרצאה 2 - ממשקים:

- $User\ mood$: המצב בו המערכת נמצאת כשהמשתמש עושה שימוש באפליקציות.
  $Kernel\ mood$: המצב בו נמצאת מערכת ההפעלה כשהיא פועלת.
  ייצוג: נייצג את המצב הנוכחי בעזרת ביט בודד - $mood\ bit$.
- $System\ calls$: שירות שמערכת ההפעלה מספקת דרכו האפליקציות מדברות עם מערכת ההפעלה כשהן צריכות להשתמש ברכיבי חומרה.
  מה מערכת ההפעלה תבצע: מערכת ההפעלה תבדוק שהפרמטרים תקינים, ותעבור למצב $Kernel\ mood$ (ע"י שינוי $mood\ bit$).

- $Privilege\ instraction$: גישה שמערכת ההפעלה עושה כדי לגשת אל החומרה כאשר אפליקציות דורשות שימוש בחומרה.
  מערכת ההפעלה בלבד יכולה לגשת לפקודות $Privilege$.

- **פסיקות** - $Interrupts$: החומרה מתקשרת עם מערכת ההפעלה בעזרת שיטה זו, גורמת למעבד להפסיק את פעולו ולעשות את פעולת מערכת ההפעלה.

- **פקודות שהתוכנה יכולה לעשות:**
  1: פקודות אריתמטיות. **2:** פקודות שרק מערכת ההפעלה יכולה לעשות - גישה לחומרה או למבני נתונים של מערכת ההפעלה (קבצים).

- **סדר הפעולות:**
  כשהאפליקציה מבקשת גישה למערכת ההפעלה (פעולות שהמערכת בלבד יכולה לבצע). האפליקציה תשמור ברגיסטר מסויים את מספר הפקודה ואחכ מערכת ההפעלה תיכנס לרגיסטר הזה ותבדוק מה הפעולה אותה היא צריכה לבצע.
  לאחר מכן נבצע את פקודת $trap$ - נעבור למצב $kernel$.
  אח"כ מערכת ההפעלה תבצע את הפקודה אותה היא נקראה לבצע.
  לבסוף מערכת ההפעלה תכניס את ערך ההחזרה לרגיסטר מסויים, והאפליקציה תקרא ממנו.

- **חריגות** - $Exeption$: כאשר תתבצע חריגה מערכת ההפעלה עשויה לעבור למצב $Kernel$.
  למעשה ה CPU מעביר את האחריות למערכת ההפעלה שתתמודד עם החריגה.

- **מערכת ההפעלה:**
  $System\ utility$: הרשאות שמערכת ההפעלה נותנת לנו כדי שהאפליקציות יוכלו לבצע כל מיני פעולות על מערכת ההפעלה.
  דרייברים: חלק ממערכת ההפעלה, אך מי שכתב אותם לא בהכרח כתב את מערכת ההפעלה.

- **טבעות** - $Rings$: שיטה שמחליפה את $kernel\backslash user$, אלא מתחלקת לטבעות כך שטבעת 0 היא $kernel\ mood$ וטבלת 3 היא $user\ mood$ וטבעות 1,2 משמשות לדרייברים. ככל שמערכת ההפעלה רואה בדרייבר יותר אמין היא תאפשר לו יותר גישה לחומרה.

2.1.1 התקני קלט פלט:

- $controler$: מעבד קטן שנמצא על כל אחד מההתקנים ו"מדבר" עם הזכרון וה CPU.

- **תקשורת:** התקשורת עם ה CPU נעשית באמצעות Bus. התקשורת דרכו מהירה וזולה, אך החסרון שלו הוא שהוא מהווה צוואר בקבוק.

- **פסיקה** - $interrupt$: כשהתקן צריך לשתמש במעבד הוא שולח פקודת $interrupt$, וכך הוא מפנה את תשומת לב המעבד אליו. בסיום הוא מדווח סיום.

- $interrupt\ vector$: רשימה שיש ל CPU, והוא מסתכל עליה ופועל לפי הפקודות כאשר נכנסת פקודת $interrupt$ מהתקן מסויים.
  בנוסף ה $mood$ ישתנה למצב $kernel$ אוטומטית.

- $maskable$: מה קורה כששתי פסיקות נכנסות בו"ז?
לא נעצור, ונמשיך עם פסיקה אחת בלבד. החומרה בלבד יכולה לשנות תהליכים ולכן החומרה היא $maskable$. יש פסיקות חשובות שלהן כן יש הרשאה לדרוס הליכים אחרים, והן מוגדרות גם כן $maskable$.

- **לסיכום - שלשת הדרכים לתקשם עם החומרה:** חריגות, פסיקות, $system\ call$.

- **יתרונות של קוד** $kernel$ **קטן:** חוסך בזכרון, חוסך קמפול מחדש בזמן עדכון של תכנות נוספות.

- $Modular\ Monolithic$: השיטה של שילוב הין ה $moods$, בשיטה זו משתמשות רוב מערכות ההפעלה כיום.

2.2 תרגול 2:

- **זכרון:**
  $cache$: יחידת ביניים ששומרת משתנים שנעשה בהם שימוש תדיר.
  דיסק: מה שנשמר בדיסק לא נמחק עד שנמחק אותו.
  הירכיה: הוצאת מידע מהזכרון מחייב מעבר **בכל** יחידות הזכרון.

- PIC: מחובר לכל התקני $I\backslash O$ ומדווח למעבד כשהתקן צריך להשתמש במעבד או הפסיק להשתמש במעבד.

- **סוגי** $interrupts$:
  חיצוני: הקריאה נעשית ע"י התקן חומרה חיצוני (עכבר, מקלדת, דיסק, טיימר ועוד).
  פנימי: התכנית הנוכחית דורשת התערבות של מערכת ההפעלה כדי להמשיך לעבוד ($system\ call$).

- **שלבי הטיפול ב** $External\ Interrupt$:
  1: קבלת האינטרפט - ה CPU בודק בסוף כל סייקל אם הגיע אינטרפט חדש.
  2: שמירת המצב הנוכחי - כמו שמירת מצב בקריאה לפונקציה.
  3: העברת השליטה למערכת ההפעלה - מעבר ל $kernel\ mood$.
  4: שחזור המצב הקודם - כמו בחזרה מפונקציה.
  5: החזרת השליטה לתהליך הקודם - מעבר למצב $User\ mood$.

- $Internal\ Interrupt$: ה CPU יודע בגלל איזו פקודה קרה האינטרפט.
סוגי $Internal\ Interrupt$: חריגות, $Segmentation\ fault$, פקודה לא חוקית, הוראה שאין למשתמש סמכות לבצע.

- **שלבי הטיפול ב** $Internal\ Interrupt$: למעט השלב הראשון, כל השלבים דומים.

3 שבוע 3:

3.1 הרצאה 3 - ניהול תהליכים $Process\ Management$ (20.3):

- **ההקשר של הביצוע - $context$:** אם נסתכל על ה CPU והזיכרון, נוכל לדעת היכן התכנית נמצאת בכל רגע נתון.
- **תהליך $Process\backslash job$:** ביצוע של תכנית, החלק במערכת ההפעלה שאחראי על הרצת התכנית. כל תהליך לא יידע על תהליך אחר והם לא יתנגשו אחד בשני, התהליך נגמר בסוף ריצת התכנית.
- $resident, non-resident$: נאמר שתהליך הוא $resident$ על ה CPU אם כשמסתכלים על הרגיסטרים של ה CPU ניתן לראות את התכנית מתבצעת.
  בכל פעם יכולה רק תכנית אחת להתבצע על ה CPU לכן רק תכנית אחת תהיה $resident$ ושאר התכניות $non-resident$.
- **תכנית $Progrsm$:** קובץ שכתובות עליו פקודות. לאחר שנעלה אותו לזיכרון יתחיל ה $prosses$ שיוסיף לתכנית $context$, מתכנית אחת ניתן להריץ הרבה $prosses$.
  בהקבלה לאפיה - תכנית היא מתכון, $prosses$ הוא תהליך האפיה עצמו.
- **מרחב הכתובות של התהליך $Address\ space$:** מכיוון שמספר תכניות יכולות לרוץ בקביל אנחנו צריכים להגדיר מרחב זיכרון לכל $prosses$.
  יש מקום מוגדר בזכרון שמוגדר מרחב הכתובות של $prosseses$. כתובות אלו מסומנות ב $[base, base + bound]$ במקרה שנחרוג מטווח זה ה CPU ישלח $exeption$. מערכת ההפעלה היא שמקצה את הזכרון.
  תרגום כובות: כשצריך לגשת לכתובת x מסויימת ה CPU מתרגם את הכתובת ל $base + x$.
- **מרחב הכתובות של מערכת ההפעלה:** נקרא $kernels\ memmory$, כל השאר נקרא $user\ memmory$.
- **מעבר בין $prosses$:** כשנעבור מתהליך אחד לאחר, נשמור את כל הערכים שנמצאים ברגיסטרים של ה CPU ונשמור אותם ב PCB (מערכת ההפעלה מעתיקה את המידע).
  לכל $prosses$ נשמור $stack$ משל עצמו.
- $ProcessControlBlock(PCB)$: מבנה נתונים של מערכת ההפעלה שמייצג את התהליך, בחלק זה נשמור את הרגיסטרים של התהליכים שסיימנו ($context$).
  בנוסף יישמר מספר התהליך, $user$, מצב ועוד.
- **מחזור חיים של תהליך:** $ready\ queue$ $\Leftarrow$מערכת ההפעלה תבחר איזה תהליך להתחיל $\Leftarrow$אם יש הפרעה (שימוש במדפסת) התהליך יעבור להמתנה ($blocked$). אם התהליך השתמש יותר מידיי זמן במעבד מערכת ההפעלה תחזיר אותו בחזרה ל $ready\ queue$.
- $Scheduling$: החלק במערכת ההפעלה שאחראי על העברת התהליכים מ $ready\ queue$ ל CPU, ומחליט איזה תהליך צריך להיכנס. (החלק שאחראי על ההחלטות)

- $Dispatcher$: החלק במערכת ההפעלה שמבצע את הפעולות בפועל. הוא מעביר את הרגיסטרים ושומר אותם ב PCB.

- **מתי נבצע מעבר בין תהליכים:**
$interrupts$: פסיקת שעון - מערכת ההפעלה מגדירה שעון שיעשה פסיקה פעם בכמה זמן כדי לנהל את התהליכים.
$memmory\ fault$. $System,\ call$. $Exception$.

- **איך נוצר תהליך:** כשאנו מעלים את המערכת נוצר לנו תהליך.
דרך נוספת ליצור תהליך - כאשר תהליך רוצה לקרוא לתהליך אחר הוא יכול ליצור תהליך ע"י הפקודה $fork()$.
לעיתים פרוסס האב יהרוג את הבן.

- $fork()$: כאשר נקרא ל $fork$ התהליך מתפצל לשני תהליכים זהים, ההבדל בניהם הוא ה id. האב יקבל את הערך של pid של הבן, והבן יקבל ערך 0.
אין לנו דרך לדעת איזה תהליך יתבצע קודם - האב או הבן וזה תלוי במערכת ההפעלה.

- **רגיסטר** EAX: רגיסטר שנמצא בכל PCB והוא הרגיסטר שמחזיר ערכים למשתמש.

- **שיתוף פעולה בין תהליכים** IPC: מערכת ההפעלה מספקת את השירות של קשר בין תהליכים מבלי למחוק אחד את השני כמובן.
דרך אחרת: להגדיר מקום בזכרון שכולם יכולים לראות.
דרך שניה: שליחת הודעות בין תהליכים (דומה לתקשורת בין מחשבים מרוחקים).

3.1.1 $THREADS$:

- **תהליכון -**$THREADS$: מסלול חישוב של התהליך, לכל תהליך יש $THREAD$ משלו. בנוסף בניגוד לתהליך הוא מדמה וירטואליזציה של כמה מעבדים.
$Multithreading$: הרצה של כמה $threads$ על אותו התהליך, במקרה זה יש שיתוף מלא בין כל ה $threads$ בניגוד ל $multiprocesing$.
שיתוף מידע יעיל יותר מאשר IPC.

- $Kernel-LevelThreads$: ב $kernel\ mood$ האחריות היא על מערכת ההפעלה. למעשה היא תנהל את כל ה $threds$ ואת השימוש במעבד וכו. המעבר בניהם ייעשה ע"י ה OS.

- $User-LevelThreads$: ספריות שעל ידי שימוש בהם ניהול ה $threads$ עובד למשתמש, ומערכת ההפעלה לא יודעת עליהם כלל.

- **יתרונות של** $THREAD$: יצירה, מחיקה, מעבר, ושיתוף מהירים הרבה יותר מאשר $prosses$.

3.2 תרגול 3:

3.2.1 סיגנלים $Signals$:

- $signal$: הודעה שנשלחת **לתהליך** כדי להודיע לו על אירוע שקרה, הוא עוצר את הפעולה הנוכחית שלו מטפל באירוע ואחכ חוזר לתהליך.
  בשונה מ $interrupts$ התהליך מקבל את האירוע ולא ה OS.
  יש רשימה מוגדרת של סיגנלים ואי אפשר הוסיף עליהם.

- **הבדלים:**
  $signals$: מערכת ההפעלה שולחת את ההודעות.
  $interupts$: מערכת ההפעלה מקבלת אותם.

- **מתי נשלח סיגנלים:**
  כשיש קלט אסינכרוני מהמשתמש (קלט שלא ציפינו לו).
  סיגנלים שנשלחים בין תהליכים שונים, כל תהליך יכול לשלוח סיגנל לאחר.
  $softwareinterrupt$ - הודעות שננשלחות ממערכת ההפעלה, הן גורמות לסיגנל בצורה עקיפה.

- **שליחת סיגנל דרך שורת הפקודה:** הפקודה $kill\ sigName\ pid$ תשלח סיגנל לתהליך שה id שלו צויינה.

- **פונקציית** $kill$: פונקציה שמופיעה בקוד c שתשלח סיגנל לתהליך במהלך ריצת הקוד.

- $strace$: עוקב אחרי סיגנלים.

- **התמודדות עם סיגנל:** ההתנהגות הדיפולטיבית: יציאה, התעלמות, השהיה, המשך הריצה אחרי עצירה. או כתיבת פונקציה שתרוץ במקרה של סיגנל.

- $sigaction(sid, action)$: מגדירה $heandle$ לסיגנל. פונקציה שמקבלת מספר סיגנל ו $action$ שזה למעשה פויינטר לפונקציה שתרוץ במקרה שנקבל סיגנל.
  בנוסף היא מגדירה איזה סיגנלים אנו נרצה לחסום בזמן הביצוע של ה $heandler$ עד לסיומו. בדיפולט היא חוסמת רק סיגנל שקשור ל $heandler$.

- **התמודדות עם סיגנלים:**
  SIG_IGN: מגדיר לתהליך להתעלם מהסיגנל.
  SIG_DEF: שיחזור הדיפולט של הסיגנל, אם שינינו אותו.

- $sigprocmask()$: פונקציה שחוסמת סיגנלים ב $prosses$ עד לסיום הטיפול בסיגנל הנוכחי.

3.2.2 $THREADS$:

- $THREADS$: חלק מה $prossess$, לכל $prossess$יש לפחות $thread$ אחד. והם ב"ת אחד בשני, אך יכולים לגשת אחד למידע של השני.

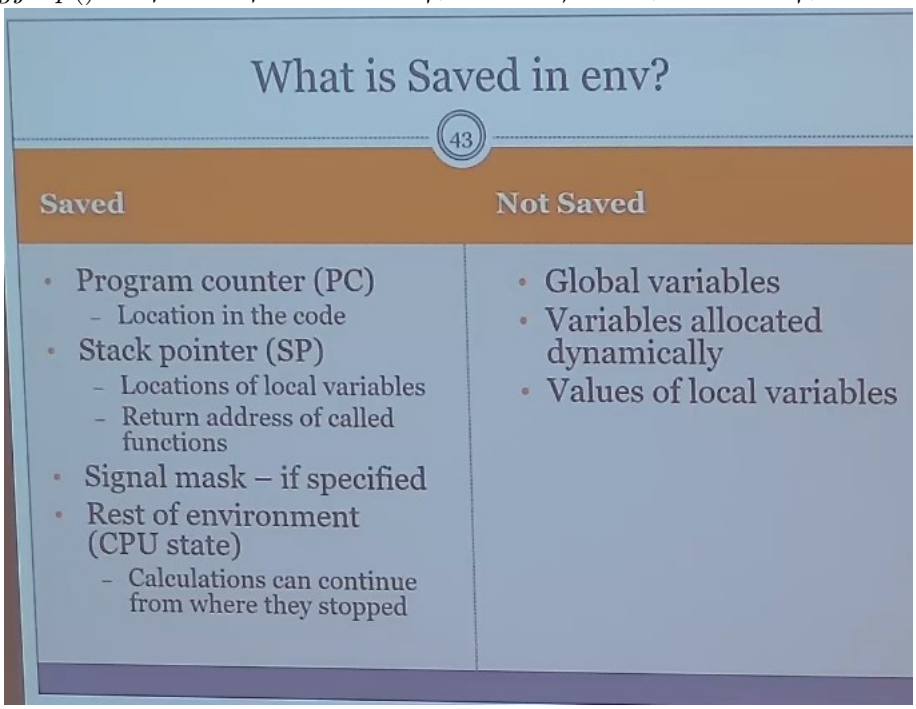
- **מה** $thread$ **שומר:** לכל $thread$ יש מחסנית משלו, ערכי רגיסטרים משלו (PC), אך בניגוד לתהליך אין לו $heap$ משלו.

- **מתי נשתמש:** כשיש צורך לפצל את העבודה של התהליך, נותנים לכל $thread$ לבצע חלק מהעבודה. (אלגוריתם $merge\ sort$ מממש $threads$)

- $UserLevelThreads$**:** סיפרייה שניתן לייבא ולנהל את ה $threads$, מערכת ההפעלה לא יודעת עליהם ומהמבט שלה רץ רק $thread$ אחד.
חסרונות: אם $thread$ אחד נחסם, אזי כולם נחסמים איתו יחד, כי מערכת ההפעלה מתייחסת אליהם כ $thread$ אחד. חיסרון נוסף: כולם רצים על אותו CPU.
יתרונות: אין צורך לקרוא למערכת ההפעלה עבור כל $thread$ והתהליך מתבצע ב $user\ mood$.

- $threaddescriptor$**:** מידע ששייך לכל $thread$, נקצה בתוך ה $heap$ מקום לכל $thread$ כולל מצביע למחסנית של כל $thread$
בנוסף: נצטרך לדאוג להחלפה בין $threads$ - עצירת ה $thread$ הראשון, שמירת מידע שלו, טעינת ה $thread$ החדש.

- $sigsetjmp()$**:** פונקציה ששומרת את המידע של ה $thread$ הנוכחי.
ארגומנטים: מקבלת סטראקט (env) - שומרת את הערכים שצריך לשמור בתוך הסטראקט. ומשתנה int שמקבל ערכי 0,1 - האם אנו רוצים לשמור את הסיגנל הנוכחי (1) או לא (0).
כדי לטעון את המידע מחדש, נוכל לטעון את הסטראקט לפונקציה $siglongjmp()$.



- $siglongjmp()$**:** טוענת את המידע שנשמר, ממשיכה את הפעולה של ה $thread$ מהמקום שהפסקנו.
ארגומנטים: סטראקט (env) - ממנו נטען את המידע. int - ערך החזרה.

פעולה: טוענת את הפרמטרים לרגיסטרים ואחכ קופצת לשורה שמופיעה בה הפונקציה $sigsetjmp()$ ונותנת לה את ערך ההחזרה int.

4 שבוע 4:

4.1 הרצאה 4 - סינכרוניזציה (27.3):

4.1.1 סינכרוניזציה:

- **מוטיבציה:** שני $prossess$שרוצים לכתוב ל $buffer$ משותף, יכולים לגרום לנזקים לא צפויים. לכן נצטרך את הסינכרוניזציה שתנהל לנו משאבים משותפים.

- **הגדרה - קטע קוד קריטי:** קטע קוד שבו שני $prossess$ או יותר משתתפים במשאב משותף.
לכן כותב הקוד יגדיר את הקטע כקריטי, ומערכת ההפעלה - $Mutualexclusion$ תטפל ותוודא שבכל שלב לא יכנסו שני $prossess$ יחד.
הנחה: הקוד באורך סופי ואין בו לולאות אינסופיות.

- $Mutual\ exclusion$**:** שיטה לפיה נוכל לשמור על קטע הקוד הקריטי.
קוד שאינו קריטי יסומן כ $reminder$ - שארית.
פתרון הבעיה: נוסיף לפני הקוד הקריטי $Entry\ code$ ו $Exit\ code$ שישמרו על הקוד הקריטי.

- **מה נרצה להבטיח:**
1 - מניעה הדדית **Mutual exclusion :** שלא יהיו שני $prossess$ שמבצעים את הקוד הקריטי.
2 - $Progres$**:** נוסיף תנאי התקדמות ב $Enter$ - לולאה שלא תיתן להיכנס אם הקוד הקריטי בריצה.
3 - Starvation fiee **:** אם $prosses$ רוצה להיכנס לקטע קוד קריטי הוא יכנס ראשון.
4 - $Generality$**:** פתרון כללי לכל מספר של $threads$.
5 - No blocking in the reminder : נדאג לא לעשות $block$ בקוד שארית.

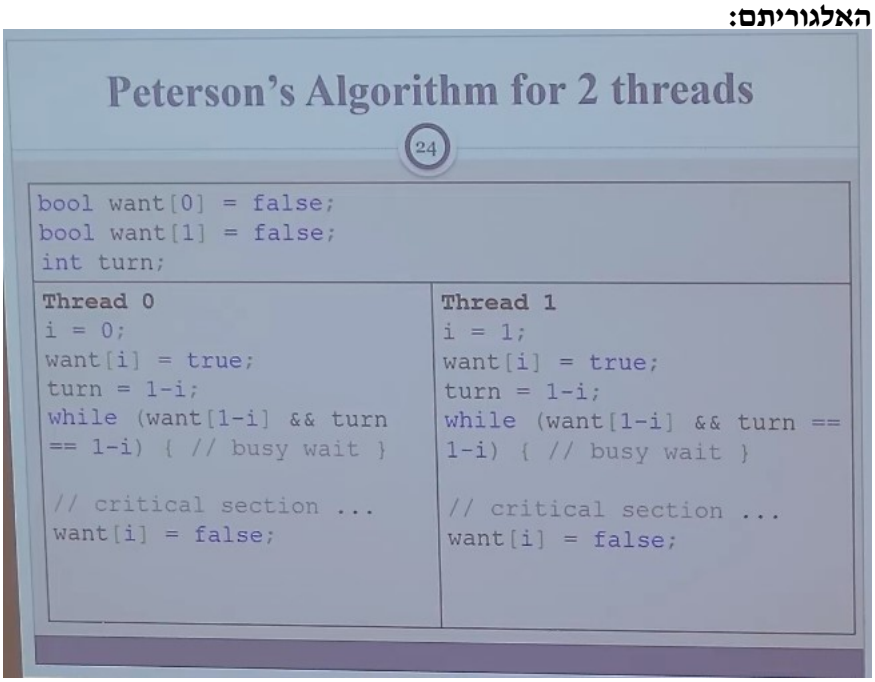
- **פתרון ראשון:** ב $Enter$ נוסיף $Disable\ interrupts$, ובסוף נשחרר.
חיסרון: שיטה זאת לא תעבוד ב $user\ mood$. לא תעבוד עם $multy\ core$. ניתן להשתלט על המעבד בצורה זו דרך ה $user$.

- **פתרון שני:** נוסיף תנאי בקטע הכניסה שבו נחליט איזה $thread$ יכול לעבוד בכל פעם. לאחר ש $thread$ אחד מסיים הוא משנה את הביט לביט של ה $thread$ השני, ולמעשה מוסר לו את התור.
החסרון: אחרי ש $thread\ 1$ מסר את התור הוא לא יוכל לחזור גם אם הוא יצטרך שוב את קטע הקוד הקריטי, אלא הוא יצטרך לחכות ש $thread\ 2$ יפתח לו את התור מחדש.

- **פתרון שלישי:** נוסיף $flag$ לכל $thread$ שיסמן האם ה $thread$ רוצה את קטע הקוד הקריטי. כך לא יהיה מצב שהתור לא נפתח, כי $thread$ שלא רוצה את הקוד לא יקבל גישה אליו.
החסרון: במקרה של שני $thread$ שמחכים אחד לשני יקרה $deadv, bth\ lock$ ואף אחד מהם לא יכנס אף פעם.

- **פתרון רביעי:** נשים את התנאי הבא - כל $thread$ לא נכנס אם לא תורו $(flag = 0)$ ו $thread$ אחר גם רוצה להיכנס.
  החסרון: שני $threads$ יוכלו להיכנס לקוד יחד.

- **הפתרון - האלגוריתם של פיתרסון:** נעלה את הדגל שמסמן כי ה $thread$ רוצה להיכנס. לאחר מכן נשנה את המשתנה של התור. ואחכ נשים את התנאי מפתרון 4.
  האלגוריתם:



  החסרון: האלגוריתם לשני $threads$ בלבד.

- **הגדרה -** $Race\ condition$**:** מירוץ בין שני $threads$ כך שכניסה של אחד מהם לפני השני תשפיע על התוצאה.

- **הגדרה - פקודה אטומית:** פקודה שצריכה להתבצע מהתחלה עד הסוף בפעם אחת ורק לפניה או אחריה יכול להגיע $contect\ swich$ (כתיבה וקריאה מהזכרון).

- $busy\ waiting$**:** לולאות שמבזבזות משאבים $(while\ True)$.

- **אלגוריתם המאפיה (לכמה** $threads$**):** נחלק מספרים לכל תהליך, והם יכנסו לפי תורם. כל מספר נמצא בשימוש פעם אחת.
  החסרון: הפעולה אינה אטומית לכן שני $threads$ יוכלו לקחת את אותו המספר.
  הפתרון: כל $thread$ שנכנס לתור לוקח את המספר המקסימלי 1+ ומסמן שהוא בוחר מספר, בנוסף נסתכל על id של ה $thread$, כך שני $threads$ שונים שלקחו בטעות את אותו מספר לא יכנסו יחד אלא לפי id.

- $Test\&Set$**:** טיפול של מערכת ההפעלה, הגדירו את הפעולה $Test\&Set$ להיות אטומית. בשיטה זו יהיה משתנה אחד שמסמן שער - כשהוא פתוח הכניסה תתאפשר.
  החסרון: לא בהכרח שיתקיים $FIFO$.

- **האלגוריתם של** $Burns$: באותו האופן של $Test\&Set$ אך עם רשימה שמסדרת את ה $threads$ לפי הסדר.
חסרון: יש $threads$ שיחכו בתור על אף שהם לא צריכים להיכנס - בזבוז משאבים.

4.2 תרגול 4 - $Creating\ Threads$:

- $pthread\ create()$: כל תוכנית מהרגע שהיא נוצרת יש לה $thread$ הראשי, והוא יכול ליצור עוד $threads$ ע"י הפונקציה $pthread\ create()$ עם הארגומנטים:
p: המערכת תשמור בו את ה id של ה $thread$.
$attr$: אטריביוטים, $NULL = defoult$.
$func(void)$: מצביע לפונקציה שמקבלת ומחזירה $void$, שאותה ה $threads$ יכולים להפעיל.
$args$: ארגומנטים של הפונקציה.

- **סיום** $threads$:
1: ניתן לקרוא ל $pthread\ exit$, שיכולה להחזיר סטטוס יציאה $void^*$.
2: ה $thread$ עושה $return$ מהפונקציה שהוא מריץ.
3: ביטלו אותו מבחוץ ע"י $ptread\ cancle(id)$.
4: אם ה $main$ הראשי מסיים לרוץ או מסיים את התכנית.

- $pthread\ Join()$: פונקציה שה $thread$ שקורא לה נכנס ל $sleep$ עד שה $thread$ שהוא שלח כפרמטר יסתיים. מקבלת כפרמטר id וסטטוס יציאה $void^{**}$ פרמטר פלט.

4.2.1 סינכרוניזציה - $Mutex$:

- $Mutex$: מנגנון שמנהל משאפים משותפים, משמש כ "מפתח גישה" למשאב כך שרק בעל המפתח יוכל להשתמש במשאב.

- **שימוש ב** $Mutex$:
1: ניצור אובייקט ונאתחל אותו - ניתן ליצור באופן סטטי עם מאקרו קבוע. או באופן דינאמי - ע"י הפונקציה $pthread\ mutex\ init$.
2: כל $thred$ שרץ צריך לנעול את ה $mutex$ עם $pthread\ mutex\ lock(*mutex)$. ולשחרר אותו בגמר השימוש - $pthread\ mutex\ unlock(*mutex)$, כדי שאחרים יוכלו להשתמש
3: בסוף התכנית נשחרר את האובייקט עם - $pthread\ mutex\ destroy()$.

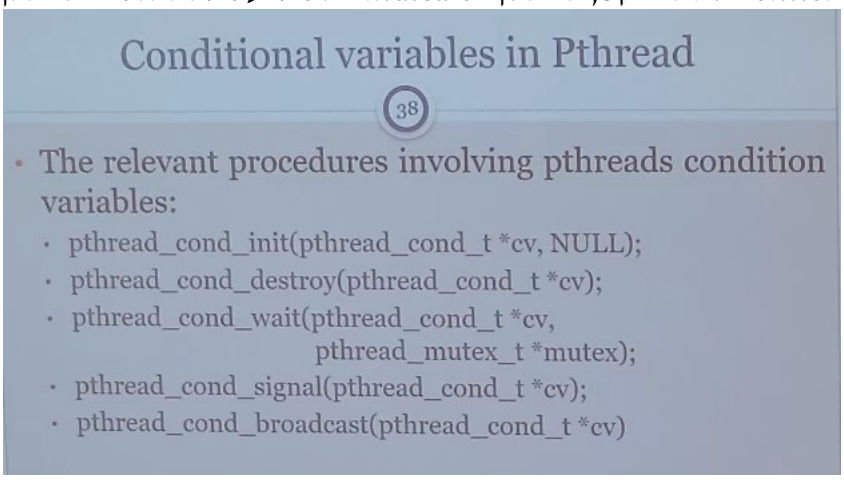
- $Deadlock$: שני $threads$ יחסמו אחד את השני בזמן שהם צריכים אחד את השני ויובילו ללולאה אינסופית. לכן צריך לנהל את ה $mutex$ בצורה זהירה.

4.2.2 סינכורניזציה $Monitor$:

- **בעיות סינכרוניזציה נוספות:**
ראינו - ניהול גישה לאותו מיקום בקטע קוד קריטי.

תיזמון ריצת $threads$ כך שאחד יוכל לעקוף את השני.
$barrier$ - נקודה בקוד שרק ברגע שכל התרדים יגיעו אליה הם יוכלו להמשיך לרוץ.
תיזמון כתיבה וקריאה לקובץ באותו הזמן.
$bounded\ buffer$: תיזמון הכנסת והוצאת איברים מ $buffer$.

- $Monitor$: אובייקט, שבנוסף ל $mutex$ יכול לייעל לנו תהליכי סנכרון בין תרדים. הפונקציות דומות ל $mutex$



$broadcast$: פונקציה שמעירה את כל התרדים ולא רק תרד אחד.

- $pthread\ cond\ wait(monitur, mutex)$: פונקציה שנשתמש בה אחרי $mutex\ lock$, קריאה לפונקציה ע"י תרד שרץ ,תשחרר את ה $mutex$ ותכניס אותו למצב שינה עד ש $thred$ אחר ישחרר אותו, לאחר מכן היא תחזיר את ה $mutex$.

- **הערה:** התרד שקורא ל $pthread\ cond\ wait$ גם צריך לשחרר את ה $mutex$.

4.2.3 סינכרוניזציה - $Semaphores$:

- $Semaphores$: אובייקט, שבנוסף ל $mutex$ יכול ליעל לנו תהליכי כתיבה קריאה מקובץ. הוא מנהל קיבולת מקסימלית בכניסה.
  הערה: אם נאתחל אותו ל 1 הוא יתנהג כמו מוניטור.

- **שימוש:** צריך לאתחל את האובייקט. הפונקציה $init$ מגדירה פרמטרים של - שיתוף בין תהליכים, ומספר שמייצג את הקיבולת המקסימלית. לבסוף נשחרר את האובייקט.

4.2.4 משתנים אטומים:

- **משתנה אטומי:** משתנה של שפת C, נכתוב כך: $std::atomic<type>$ ניתן לעשות עליו שתי פעולות - $load$ ו $stor$, בנוסף ניתן להשוות עם =. והוא משמש כ $monitor$.

5 שבוע 5:

5.1 הרצאה 5 - סינכרוניזציה (4.4):

5.1.1 *Semaphore*:

- **אינטואציה:** ה *Semaphore* משמש כמארחת של מסעדה, שאחראית על וויסות כניסת הלקוחות כך שהמסעדה לא תתמלא מעבר לתפוסה.

- **אובייקט** *Semaphore*: מחלקה עם שתי שדות, השדה הראשון הוא ערך ה *Semaphore*, והשדה השני רשימה של תהליכים או תרדים.

- **פעולות על** *Semaphore*: איתחול עם ערך, הורדת והעלאת ערך. (פעולות אלו קורות בצורה אטומית).

Down(S)	Up(S)	Init(S,v)
S.*value*= S.*value* - 1 if S.*value* < 0 then { add this thread to S.*L*; sleep();}	S.*value*= S.*value* + 1 if S.*value*≤0 then {remove a thread T from S.L; Wakeup(T);}	S.*value*= v

- **כיצד נשמור על הקטע הקריטי:** יש *Semaphore* אחד משותף עבור כל התהליכים, נאתחל אותו עם ערך התחלתי - מספר התרדים שיכולים להיכנס בו זמנית לקטע הקריטי. כל תרד שרוצה להיכנס מוריד את הערך וכל תרד שיוצא מעלה את הערך. אם הערך קטן שווה ל 0 אזי התרד יכנס לרשימת המתנה.

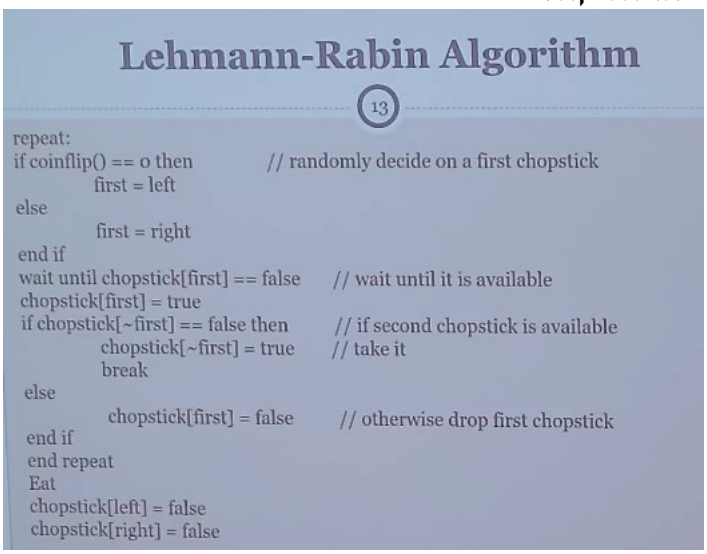
- **החסרון:** אנו משתמשים במערכת ההפעלה ועוברים בין *user* ל *kernel*, אך זאת השיטה הטובה ביותר שנמצאת כיום בשימוש.

- *Binary Semaphore*: *Semaphore* שמאוחל ל 1 ולא שומר ערכים מתחת ל 0.

- **פתרון בעיות סינכרוניזציה נוספות:** עם *Semaphore* אנו יכולים לפתור בעיות נוספות, לדוגמה אם נרצה שפעולה A תתבצע לפני פעולה B.
  נוכל לעשות זאת כך - נאתחל את הערך ההתחלתי ל 0, ונגדיר שפעולה B לא תתבצע אלא אם כן הערך ההתחלתי יהיה שווה ל 1.

- **בעיית הפילוסופים האוכלים:** הפילוסופים (תהליכים) יושבים במעגל כך שליידם מכל צד מונח צ'ופסטיק (*Semaphore*), כל פילוסוף צריך שני צ'ופסטיקס כדי לאכול, אך מספר הצ'ופסטיקס שווה למספר הפילוסופים כל שלא כולם יכולים לאכול יחד.
  כיצד נימנע מ *dedlock*: אנו יכולים הגיע למצב של *deadlock* אם כל פילוסוף ירים את הצ'ופסטיק שנמצא לשמאלו ויחכה לימני. לכן הפתרון הוא לשבור את הסימטריה - כל הפילוסופים למעט אחד ירימו את השמאלי ויחכו לימני, אך אחד מהם ירים את הימני ויחכה לשמאלי.

פסואודו קוד:



- **מתי קורה** $deadlock$**:** אם כל התנאים הבאים מתקיימים
  1: יש לנו משאב שנעלנו.
  2: אם אחד מהמשאבים מחכה לאחר שיפתח, בכדי לבצע את הפעולה שלו.
  3: אין מצב שתרד אחר ישחרר משאב שננעל.
  4: מעגליות, כל אחד מחכה לבא בתור אחריו, והאחרון מחכה לראשון.

- **התמודדות עם** $dedlock$**:** מערכת ההפעלה לא מסוגלת להתמודד עם דדלוק, לכן קיים תוסף שמזהה שנכנסנו למצב דדלוק ומאלץ את אחד מהתרדים לשחרר את המשאב.

- **התמודדות עם** $dedlock$ **בשלב התכנון:**
  1: נסתכל על כל המקורות שיש לנו, ניתן לכל אחד מספר ונסדר אותם לפי סדר.
  2: ננעל אותם מהגבוה לנמוך ונשחרר מהנמוך לגבוה, כך נצליח להימנע ממעגל.

5.1.2 $Bounded\ Buffer - Producer\ Costumer$:

- **הבעיה:** תרד אחד (או יותר) מייצר עבודה $(producer)$, ותרד אחר (או יותר) צורך את העבודה $(cosumer)$.
  לדוגמה - תרדים שמייצרים עבודות למדפסת, ותרדים של המדפסת שצריכים להדפיס.

- **מימוש:** התרדים שמייצרים עבודה יכניסו אותה לבאפר, והתרדים שצריכים לבצע את העבודה ימשכו אותה מהבאפר.

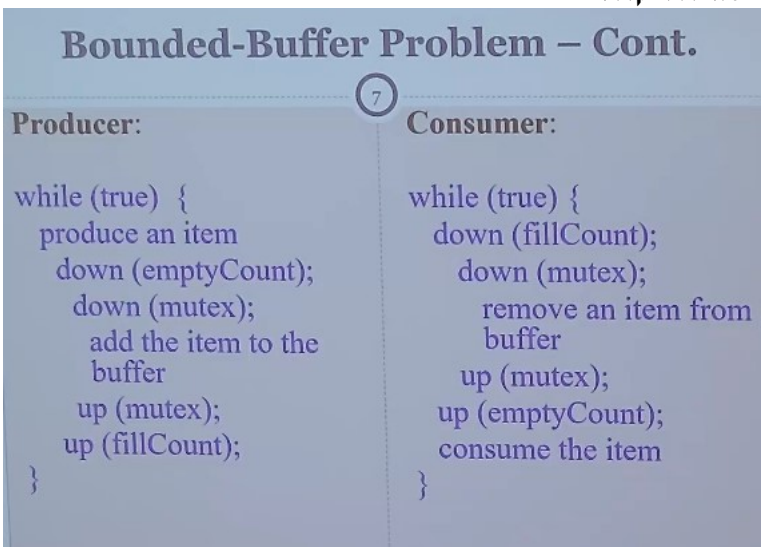
- **הערה:** ההתנגשות (דריסת תאים, דילוג או חזרה) יכולה לקרות גם בהכנסה וגם בהוצאה.

- **הגדרה - באפר ציקלי:** באפר שכשנמלא אותו עד סופו, נחזור להתחלה ונמלא שוב במקומות הפנויים.

- **כיצד נדע אם הבאפר מלא:** נאתחל $counter$ שמסמן את מספר התאים המלאים בבאפר. בכל שלב כל תרד יעדכן את הבאפר אם הוא כתב או מחק ממנו (נממש את המונה כמשתנה אטומי או קטע קריטי).

- **התמודדות עם מספר** $Costumers$ **או** $Producers$**:** נשתמש ב $mutex$ שיגן על הבאפר כך שרק תרד אחד יוכל להיכנס ולעדכן בכל פעם (לכתוב או למחוק).
בנוסף נגדיר שני $Semaphores$ - $empty = n$ $fill = 0$, שיעדכנו את מספר התאים הריקים והמלאים בכדי שלא ננסה לבצע עבודה על תא ריק, או לכתוב לתא מלא.

- **פסואודו קוד:**



- **למה צריך שני** $Semaphores$**:** אם נמחק אחד מהם, לא נוכל לדאוג שלא נעבוד על תא ריק, או נכתוב לתא מלא.

5.1.3 $Monitor$:

- $Monitor$**:** מבנה נתונים שמתפקד כאובייקט. יש לו דאטה ומתודות שניתן לבצע על הדאטה.

- **מימוש:** עבור מבנה נתונים משותף - באפר מסויים, ננעל אותו ונגדיר אותו כמוניטור, כך שניתן לבצע עליו מספר פעולות מסוימות בלבד שהמוניטור דואג לסינכרוניזציה שלהן (שרק תרד אחד יבצע את הפעולה בכל שלב).

5.2 תרגול 5 - סינכרוניזציה ומקביליות:

5.2.1 בעיית $Readers\ Writers$:

- **הבעיה:** יש לנו מבנה נתונים משותף, קבוצה של תרדים שקוראים מהקובץ (קריאה סטטית מבלי לשנות את הקובץ) וקבוצה נוספת שכותבת לקובץ (משנ אותו).

- **מה אנו רוצים:** לא רוצים ששני כותבים יכתבו לקובץ וישנו אותו יחד. בנוסף אנו לא רוצים שיהיו יחד קוראים וכותבים. נאפשר - או רק קוראים או כותב בודד.

- **פתרון ראשון:** נגדיר מונה שסופר את מספר הקוראים $read = 0$, ונגדיר $Semaphore = 1$ שיגן אליו. בנוסף נגדיר עוד $Semaphore = 1$ שידאג לכך שלא יהיו כותב וקורא יחד בקובץ.
החסרון: הכותבים יכולים לחכות לנצח ולא לכתוב אף פעם.

- **פתרון שני:** נשנה את הפתרון כך שכותבים יקבלו ייתרון על פני קוראים.
בנוסף נוסיף עוד $Semaphore$ שיקרא $read = 1$, תפקידו למנוע מקוראים להיכנס כשכותב רוצה לכתוב.
נוסיף מונה שסופר את מספר הכותבים שרוצים לכתוב $write = 0$ ונגן עליו בעזרת $Semaphore = 1$.
נוסיף $Semaphore$ נוסף $Queue = 1$ שתפקידו הוא לשמור על תחרות הוגנת, כך שמול כל כותב יתחרה רק קורא אחד.

פסואודו קוד:

מהצד של הכותב:

```
The writer:

while (true) {
  down (writeCount_mutex )
    writeCount++;  //counts number of waiting writers
    if (writeCount ==1)
      down (read)
  up(writeCount_mutex)

  down (write) ;
    // writing is performed – one writer at a time
  up (write) ;

  down (writeCount_mutex)
    writeCount--;
    if (writeCount ==0)
      up (read)
  up (writeCount_mutex)
}
```

מהצד של הקורא:

```
The reader:

while (true) {
  down (queue)
    down (read)
      down (readCount_mutex) ;
        readCount ++ ;
        if (readCount == 1)
          down (write) ;
      up (readCount_mutex)
    up (read)
  up (queue)

  reading is performed

  down (readCount_mutex) ;
    readCount - - ;
    if (readCount  == 0)
      up (write) ;
  up (readCount_mutex) ;
}
```

5.2.2 תרגיל 3:

- ניצור תרדים ונריץ עליהם פונקציות - $kernel\ mood$, נצטרך לדאוג למשאבים משותפים - $mutex$ וכו.
התרדים יריצו פונקציות $map\ reduce$ - מחלקת את המשימה לתתי משימות כך שכל תרד יריץ תת משימה.
map - מקבלת פונקציה ומערך ומפעילה את הפונקציה על המערך.
$reduce$ - סוכמת את האיברים במערך איך שנחליט (חיבור כפל וכו).
הקלט - סדרה של קלטים שיכולים להיות כל דבר, ואנו מפצלים את הסדרה כך שכל תרד מקבל איבר ומריץ עליו

את map.
הפונקציה map יכולה להוציא סדרת פלטים לכל תרד.
שלב ביניים סינגל תרד (נממש עם $barier$ - קודם כל התרדים יסיימו את שלב ה map) ואז נתקדם - לאסוף את כל תוצאות ה map ולמיין אותן.
שלב ה $reduce$ - כל תרד מקבל סדרת ביניים (פלט של map הממויין) ומפעיל עליה את $reduce$ שמחזיר פלט בודד, הפלט הסופי הוא הפלטים של כל ה $reduces$ של כל התרדים.
הערה: נשתמש באותה כמות תרדים גם ל map וגם ל $reduce$, אין צורך ליצור תרדים חדשים לכל שלב.
הערה 2: הקלט הוא סדרת קלטים $map, reduce$ ואנו לא צריכים לייצר את הפונקציות הללו.

6 שבוע 6:

6.1 הרצאה 6 (10.4) - $Sceduler$:

6.1.1 $Sceduler$:

- **ציר הזמן של תהליך:** כשנכנס תהליך הוא יוגדר כ $arrive$, לאחר מכן הוא יהיה ב $wating\ time$, כשהוא יתחיל לרוץ הוא יוגדר $running$ ויהיה ב $running\ time$. לבסוף הוא יוגדר כ $terminate$. התהליך כולו מוגדר כ $response\ time$.

- **מהו ה** $Sceduler$**:** למעשה זהו אלגוריתם, הקלט שלו הוא רשימה של עבודות, הפלט שלו הוא החלטה - עבודה אחת שתרוץ. מטרת האלגוריתם היא יעילות המעבד וביצועים טובים. בנוסף הוא יוכל להפסיק ריצה של עבודה אחרת כדי להכניס עבודה חדשה ל $running$.

- **מתי נקרא ל** $Sceduler$**:** כשריצה של עבודה מסתיית או כשנכנסת עבודה חדשה.

- **כיצד נגדיר** $Sceduler$ **טוב:**
  1: נדאג שזמן ה $response\ time$ לכל עבודה יהיה כמה שיותר נמוך.
  2: נדאג בעיקר שזמן ההמתנה יהיה כמה שיותר נמוך.
  3: מדד נוסף הוא $slowdown = \frac{response\ time}{running\ time}$.

- **הוגנות:** בקורס שלנו נניח שאנו מדברים על חלוקות שוות.
  1: שכל תהליך יקבל את מה שמגיע לו באופן הוגן (לא בהכרח שווה).
  2: אנו רוצים למנוע מצב שבו עבודה מסויימת לא תיכנס לעולם ל $running$, אלא כל מי שמחכה נכנס.
  3: נרצה למנוע העדפות שנקבעות מבחוץ, השיקולים לבחירה יהיו שיקולי מערכת ולא שיקולים זרים.

- **השיקולים:** נשים לב כי אם אנו רוצים לחסוך בזמן מעבד ולהריץ כל עבודה עד סופה אנו נשלם ב $wait\ time$, ומתקיים טריידאוף בניהם.

- **הגדרה -** $online$ **אלגוריתם:** אלגוריתם שלא מקבל את כל הקלט מראש, אלא מקבל אותו בחלקים ובכל שלב הוא צריך להגיב לחלק החדש שמגיע.

- **הגדרה -** $preemption$**:** מאפשר לבטל החלטו קודמות של האלגוריתם. לדוגמה - לעצור באמצע ריצת עבודה ולהחליף את העבודה בעבודה אחר.

- $offline\ Sceduler$**:** כל המידע על העבודות מתקבל בהתחלה, וה $Sceduler$ צריך לבחור מי מהן תרוץ.
  הפתרון הנאיבי הוא להריץ אותן לפי הסדר.
  החסרון: אם יש עבודה ארוכה היא תוקעת את כל המערכת.
  הפתרון - $Shorted\ Job\ first$**:** נסדר את העבודות בסדר עולה מהקטנה לגדולה. (נשים לב שהאופטימיזציה כאן היא על זמן ההמתנה).

6.1.2 $online\ Sceduler$:

- $online\ Sceduler$**:** המידע מתקבל בחלקים.

- **הפתרון הנאיבי:** נריץ את העבודות בשיטת $FIFO$.

- **פתרון -** $Shorted\ Job\ first$**:** נבצע בכל פעם את העבודה הקצרה ביותר, אך **רק לאחר** שנסיים את הריצה הנוכחית. (א"א להפסיק באמצע ריצה ולהחליף עבודות).

- **פתרון -** $Priority\ Sceduling$**:** נקבע לכל עבודה $priority$ ונגדיר ל $Sceduler$ לקחת את העבודה שה $priority$ שלה הגבוה ביותר.

- **פתרון בשימוש** $(SRPT) - preemption$**:** אם נכנסה עבודה קצרה יותר, נחליף את הנוכחית בחדשה. נשים לב כי אנו מבזבזים כאן זמן על החלפה בין עבודות - $overhead$.

- **פתרון משופר:** אנו לא יודעים כמה זמן כל עבודה צריכה לרוץ לכן בכדי להשתמש בפתרון הקודם נצטרך לשערך מה זמן הריצה של כל עבודה.

- **שיערוך זמן ריצה של עבודה:** נתבסס על ההנחה כי העבודה שאנו רוצים להעריך תלויה בעבודות שכבר ביצענו, והיא למעשה חלק משאר העבודות ולכן הריצה של כולן אורכת סביב הממוצע.
  עבור t_n - זמן האמיתי של הפעילות ה n, ועבור τ_{n+1} - שערוך של זמן הריצה של העבודה הבאה. נגדיר משקל $0 \leq \alpha \leq 1$ ונשערך באופן שיתן משקל גבוה יותר לעבודות האחרונות שסיימנו.

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

החסרון: הערכה שלנו לא תמיד נכונה, ויכול להיות שנריץ מספר עבודות מהירות אך החלק האחרון יהיה ארוך מאד.

- **פתרון נוסף - שיתוף המעבד** $proccesor\ shearing$**:** נניח כי עבודות יכולות לרוץ יחד על המעבד, ולא רק עבודה בודדת בכל פעם. ועבור k עבודות יקח למעבד לבצע כל עבודה זמן של $\frac{1}{k}$. כך אף עבודה לא מחכה אך זמן התגובה ארוך מאד, אך כך עבודה קצרה תסתיים מהר.
  החסרון: הקצב נהיה איטי וכל עבודה רצה יותר זמן. לכן נשתמש בשיתוף רק אם יש לנו מעט עבודות באורכים שונים (סטיית התקן יותר גדולה מהממוצע).

- $Round\ Robin(RR) - proccesor\ shearing$: כיצד נדמה שיתוף מעבד - נחלק את הזמן של המעבד לחלקים קטנים של מילי שניות, כל חלק ייקרא $time\ quantum$, ונחלק אותם בין התהליכים. כל תהליך ירוץ $time\ quantum$ ולאחר מכן נחליף לעבודה הבאה בעזרת $interrupt$ של השעון.

- **זמנים עם** $time\ quantum$:
  עבור n עבודות בתור הזמן שכל עבודה מחכה הוא $(n-1)q$.
  עבור $contact\ swich$ שלוקח c זמן ה $overhead$ יעלה לנו $\frac{c}{q+c}$.

6.2 תרגול 6:

- לא התקיים תרגול.

7 שבוע 7:

7.1 הרצאה 7 (24.4) - $Sceduler$:

7.1.1 האלגוריתם $Multi - LevelFeedbackQueues$:

- **הרעיון:** שיפור האגוריתם של RR. כאשר עבודה מסיימת לרוץ את ה $quantum$ שלה ואנו עושים לה $preemption$, אנו יודעים שהעבודה הזו רצה יותר זמן ולכן היא עבודה ארוכה. כעת נוכל להפריד בין העבודות הארוכות לקצרות, וכשתיכנס עבודה חדשה ניתן לה עדיפות על עבודות שאנו יודעים שהן ארוכות.

- **המימוש:** יהיו לנו מספר תורים, תור אחד עבור תהליכים חדשים שנכנסים, ותורים נוספים עבור תהליכים שסיימו $quantum$. כל תור ייצג מספר $quantums$ שהתהליך עבר. כל תהליך שסיים את התור שלו יכנס לתור של תהליכים שכבר סיימו $quantum$. ואנו נעדיף להכניס תחילה תהליכים מהתור של התהליכים החדשים.

- **כיצד נדאג שלא יהיה** $starvation$: אם בכל שלב נכניס רק עבודות חדשות אזי יכול להתקיים מצב שעבודות בתורים אחרים לא יכנסו כלל.
  פתרון: נוכל להגדיר את הכללים כך שלא תמיד נכניס תהליכים מהתור הראשון אלא מידיי פעם ניתן עדיפות לשאר התורים, כלומר - נחלק את הזמן בין התורים כך שהתור הראשון יקבל אחוז מירבי מהזמן, וככל שהתור ישן יותר הוא יקבל עדיפות פחותה.
  פתרון נוסף - משוב שלילי: כמו הפתרון הקודם, אך נגדיר בנוסף שככל שתהליך מחכה יותר זמן העדיפות שלו עולה.

- **גישה נוספת ל** $starvation$: לא אכפת לנו שמתרחש $starvation$, כי אם מתרחש מצב כזה שבכל פעם נכנסות מלא עבודות קצרות, אזי אנו לא במצב אידיאלי אלא ב $overload$ - המערכת מוצפת בבקשות. לכן $starvation$ הוא תוצאה של הצפה, ותהליכים מצטברות בתור כי המערכת לא יכולה לעבד את כל התהליכים שנכנסו, לכן ניתן עדיפות לתהליכים קצרים ונייבש את התהליכים הארוכים.

7.1.2 $Unix\ Sceheduler$:

- **כיצד הוא עובד:** עובד באופן הבא, יש 128 תורים כאשר 50 התורים הראשונים מוקצים לתהליכי $kernel$ והשאר ל $user$. תמיד הוא בוחר את התור הראשון שלא ריק.
- **החלוקה של $kernel$ תעבוד באופן הבא:** מי שחיכה לדיסק קיבל עדיפות 20 (כי הוא איטי יותר) ולטרמינל - 28.
- **עדיפות תהליך ה $user$:** נבדוק כמה הוא רץ עד עכשיו $(cpu\ useage)$ + 50 של $kernel$.
- **כיצד נחשב את $cpu\ useage$:** השעון שולח $interapt$ 100 פעמים בשניה, ואנו נספור לכל תהליך כמה טיקים של השעון קיבלנו בזמן שהוא רץ. אם התהליך לא סיים, אזי בסוף ה $quantum$ אם יש תהליך עסצ עדיפות גבוה יותר נריץ אותו, אחרת - נריץ את התהליכים שנותרו עם RR. פעם בשניה נסתכל על $cpu\ useage$ של כולם ונחלק אותו ב 2 (כדי להעלות את הסיכוי של כל התהליכים שמחכים מלא זמן).

7.1.3 הערכת ייעילות של $Scheduler$:

- **שיטה ראשונה:** נשתמש בתורת התורים, ננתח מעבר מתור לתור ונראה מה זמן תגובה ממוצע.
- **שיטה שניה:** נסמלץ ריצה שלו ונראה כיצד הוא מתנהג.
- **שיטה שלישית:** נממש, נשווק ונראה אם יש תלונות.
- **אנו ננתח את המקרה הפשוט:** יש לנו מעבד בודד. העבודות מגיעות בתהליך פואסוני בקצב λ, כלומר מרווחי הזמן בין כניסת העבודות מתפלגים אקספוננציאלית. קצב העיבוד של המעבד שווה ל μ.
  בכדי שלא נגיע ל $overload$ נרצה שיתקיים $\lambda \leq \mu$.
- **משפט קטן - זמן התגובה הממוצע:** נסמנו ב $\bar{r}$, עבור מספר העבודות הממוצע - $\bar{n}$ שווה ל: $\bar{n} = \lambda \cdot \bar{r}$.
- **כיצד נמצא את $\bar{n}$:** נשתמש בשרשראות מרקוב, עבור i מצבים כאשר $i = 0, 1, 2..$ מתאר מצב של משפר העבודות במערכת. עבור כל מצב i יש התפלגות π_i המסמנת את ההסתברות להיות במצב ה i. ומתקיים:

$$\pi_i = \left(\frac{\lambda}{\mu}\right)^i \pi_0 = \rho^i \pi_0$$

עבור ρ^i = הניצול של המערכת.
ומתקיים:

$$1 = \sum_{i=0}^{\infty} \pi_i = \pi_0 \sum_{i=0}^{\infty} \rho^i = \frac{\pi_0}{1-\rho}$$

ולכן:

$$\pi_i = \rho^i (1-\rho), \quad \rho = \frac{\lambda}{\mu}$$

וכעת מתקיים כי מספר התהליכים הממוצע במערכת שווה ל:

$$\bar{n} = \sum_{i=0}^{\infty} i \cdot \pi_i = \sum_{i=0}^{\infty} i(1-\rho)\rho^i = \frac{\rho}{1-\rho}$$

ומחוק קטן נובע:

$$\bar{r} = \frac{\bar{n}}{\lambda} = \frac{\rho_0}{\lambda(1-\rho)} = \frac{1/\mu}{1-\rho}$$

- **מסקנה:** אם נגיע למערכת עם ניצול שמתקרב ל 100% זמן ההמתנה ישאף לאינסוף כי נגיע ל $overload$.

- **מערכת פתוחה וסגורה:**

מערכת פתוחה: לדוגמה שרת ווב. המערכת מקבלת תהליכים מבחוץ מטפלת בהם ומשחררת אותם.
מאפיינים: אין פידבק מהביצועים של המערכת להגעת עבודות חדשות. מספר העבודות משתנה כל הזמן. בכדי שהמערכת תהיה יציבה צריך להתקיים $load < 100\%$. נמדוד ביתועים לפי זמן תגובה.

מערכת סגורה: לדוגמה מחשב פרטי. המערכת מטפלת כל פעם באותם תהליכים שחוזרים למערכת.
מאפיינים: כאשר המערכת מאטה אנו יודעים ששלחנו יותר מידיי עבודות, - יש פידבק. מספר העבודות קבוע. מקיים $load = 100\%$. נמדוד ביצועים לפי מספר עבודות פר יחידת זמן.

- **צוואר בקבוק:** במערכת יכולים להווצר לנו צווארי בקבוק בכמה מוקדים.
הדיסק - אם נעשה פעולות שמערבות מלא $I\backslash O$ ללא CPU אזי יהיה לנו צוואר בקבוק בדיסק.
המעבד - אם תהליכים יבצעו מלא חישובים ולא ירצו גישה לדיסק אזי יהיה לנו צוואר בקבוק במעבד.
לכן נצטרך לייעל תהליכים המרומות שבהם מתרחשים צווארי בקבוק כי אחרת הבעיה לא תיפתר.

- $Long - Term\ Scheduling$**:** כאשר יש לנו תהליכים שצריכים להשתמש במלא זיכרון, לעיתים אף יותר זכרון ממה שהמערכת מציעה. לכן נצטרך להוציא חלק מההליכים מהתור בכדי לפנות זכרון לתהליכים אחרים.
הפתרון: נשמור על עבודות אינטרקטיביות - עבודות שהמשתמש מחכה להן. ונשמור על $job\ mix$ טוב - שיהיו לנו תהליכים משני הסוגים בכדי שכל רכיבי המערכת יהיו בניצול מקסימלי כך שלא יתרחש צוואר בקבוק.

- $Fair\ Share\ Scheduling$**:** הוגנות, אך לא ברמת השוויון אלא שכל תהליך יקבל את מה שהוא צריך.
אלגוריתם $virtual\ time$**:** ככל שתהליך רץ יוצר זמן הוא מקבל עדיפות נמוכה יותר, אך הקצב שבו נספור את השימוש ב CPU ימדד אחרת לכל תהליך.
אלגוריתם $Lottery$**:** נחלק לכל תהליך כרטיסי הגרלה, כך שמספר הכרטיסים שהוא מקבל משקף את העדיפות שלו, וההחלפות בין התהליכים נעשות ע"י הגרלה.

7.2 תרגול 7 (27.4) - $Sceduling$:

7.2.1 תזמון של המעבד - $CPU\ Sceduling$:

- בחלק זה נעסוק במספר אלגוריתמים שפותרים את בעיית התזמון במעבד - איזה תהליך נכניס למעבד בכל שלב.
נרצה למקסם את: ניצול המעבד - $CPU\ utilization$ הזמן שהמעבד עסוק בהרצת **תהליכים** (ולא משימות של מערכת הפעלה).
נרצה למקסם גם את: תפוקת המעבד - $Throughput$ מספר התהליכים שמסתיימים פר יחידת זמן.
נרצה למזער את: זמן ההמתנה - $Waiting\ time$ הזמן שכל תהליך מחכה עד שהוא מתחיל לרוץ.
נרצה למזער גם את: $Turnaround\ time$ הזמן הממוצע שכל תהליך חי במערכת.

- **הגדרה - אלגוריתם** $non-preemptive$: אלגוריתם שמכניס עבודה למעבד ולא מוציא אותה עד שהיא מסיימת את ריצתה.

- **האלגוריתם** $first\ come\ first\ served-FCFS$: עובד בשיטת $FIFO$ התהליך שנכנס ראשון יכנס למעבד ראשון.
החסרון: אם זמן הריצה של התהליך הראשון שהגיע ארוך, זמן ההמתנה הממוצע של התהליכים יהיה גבוה.
האלגוריתם הוא $non-preemptive$.

- **האלגוריתם** $shortest\ job\ first$: נמיין את העבודות לפי הגודל מהקטנה לגדולה ונקצה את המעבד למשימה הקצרה ביותר.
האלגוריתם הוא $non-preemptive$.
הוא אופטימלי מבחינת זמן ההמתנה הממוצע תחת ההנחה שאנו במערכת $offline$.
החסרון: עובד על מערכות $offline$ כשזמן הריצה נתון מראש. בנוסף אם יש לנו תמיד עבודות קטנות העבודות הגדולות לא ירוצו אף פעם (במצב $online$).

- **האלגוריתם** $shortest\ ramaining\ time\ first-SRTF$: נריץ את העבודה שנשאר לה הכי פחות זמן.
האלגוריתם הוא $preemptive$.
החסרונות: צריכים לדעת את זמן הריצה מראש. בנוסף אם יש לנו תמיד עבודות קטנות העבודות הגדולות לא ירוצו אף פעם.

- **האלגוריתם** $priority$: נריץ את העבודות לפי העדיפות שלהן, נמיין לפי עדיפות ונריץ. מערכת ההפעלה יכולה לקבוע את העדיפות לפי מידע שהיא שמרה על העבודה, או שהמשתמש יגדיר עדיפויות למערכת ההפעלה.
ניתן לממש את האלגוריתם בשתי הווריאציות $preemptive$ ו $non-preemptive$.
החסרון: אם יגיעו תמיד עבודות עם עדיפות גבוה אזי משימות עם עדיפות נמוכה לא ירוצו, או משימה ארוכה עם עדיפות גבוהה שתחסום את שאר המשימות.
ניתן לפתור את הבעיה ע"י עדכון העדיויות לכל עבודה בכל שלב.

- **האלגוריתם של** $round\ robin-RR$: נסדר את המשימות בתור מעגלי ונגדיר $quantum$, ברגע שנגמר $quantum$ נחליף לתהליך הבא בתור.
החסרון: נבצע מלא cs והניצולת של המעבד תרד. בנוסף זמן ההמתנה הממוצע יעלה.
האלגוריתם הוא $preemptive$.

- **האלגוריתם** $Multylevel\ feedback\ queue$**:** נחלק את התור לתתי תורים כך שלכל תת תור יהיה אלגוריתם תזמון משלו. בנוסף יהיה אלגוריתם תזמון חיצוני שמתזמן את תתי התורים. ניתן לעדכן את התורים ולהזיז תהליכים בין התורים, כך שאם תהליך בזבז יותר מידיי משאבים או ירד לעדיפות יותר נמוכה, נעביר אותו לתור מתאים יותר. ניתן גם לחלק את זמן המעבד בין התורים באופן לא שוויוני, כך שתור מועדף יקבל יותר זמן.

7.2.2 תזמון משימות במקביל - $Parallel\ System$:

- $super\ computer$**:** מספר מחשבים שמחוברים יחד ברשת פנימית, ומריצים משימות ללא הפסקה במקביל. עבור כל משימה נגדיר מספר מעבדים, זכרון ו $timeout$ והם מוקצים לנו עד סוף המשימה.

- **המשימה:** אנו רוצים לבנות מתזמן שיכול להריץ משימות על כמה מחשבים המקביל. אנו מניחים כי המשתמש מגדיר את זמן הריצה ואת מספר המעבדים הנדרשים מראש.

- **האלגוריתם** $first\ come\ first\ served - FCFS$**:** נכניס את העבודות למחשבים לפי סדר הגעתן.
  החסרון: עבודות קטנות יכולות לחכות למרת שיש לנו מעבדים פנויים שהן יכולות לרוץ עליהם.

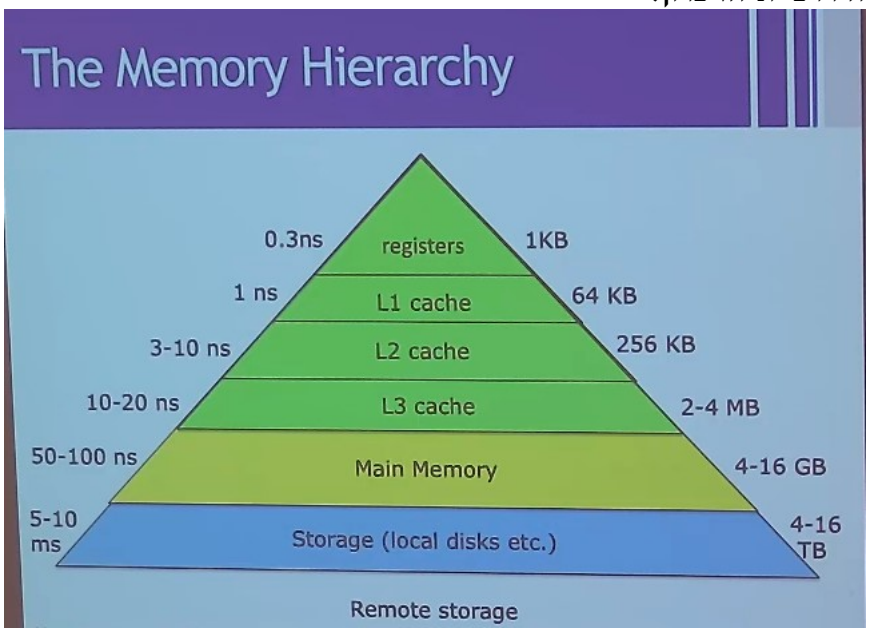
- **האלגוריתם** $Backfilling$**:** נשאף לתזמן משימות לפי סדר הגעתן, אך אם יכנסו עבודות שדורשות מספר קטן של מעבדים ויש לנו מעבדים בשבילן נכניס אותן לפני שתורן הגיע.
  החסרון: אם יגיעו מלא עבודות דורשות מספר קטן של מעבדים אך רצות מלא זמן, הן יתפסו לנו את המעבדים ולא נוכל להריץ את שאר העבודות.

- **האלגוריתם** $EASY$**:** נשמור שני מבני נתונים - אחד למשימות שרצות כרגע, והשני למשימות שממתינות ונסדר אותן בסדר ההגעה. האלגוריתם יעבוד כך:
  1: נתזמן את המשימות לפי $FCFS$.
  2: אם אין מקום למשימה מסויימת, נשמור לה מקום מראש.
  3: אם ניתן לשבץ את כל המשימות מבלי לגעת במקומות השמורים נעשה זאת.
  מתי נכניס משימות בין החורים: אם ניתן להשלים מבחינת הזמן - המשימה שנכניס תסתיים לפני שהמשימה ששמרנו לה מקום תגיע. או אם המשימה שאנו רוצים להכניס לא תופסת מעבדים שמורים.

8 שבוע 8 - זיכרון:

8.1 הרצאה 8 - זיכרון (1.5):

- **היררכיית הזיכרון:**



8.1.1 זכרון המטמון - $Cach$:

- **העיקרון:** נרצה לשמור בחלק הזכרון שקרוב אלינו (המהיר), את המידע שנמצא בשימוש באחוז הגבוה ביותר. והוא ימוקם בין ה CPU לזכרון המרכזי ויחולק לכמה רמות (בד"כ 3).

- **שאלות שנרצה לענות עליהן:** כיצד נמצא דברים במטמון? מה צריך לשמור במטון? ומה נעשה כשהוא יתמלא - איזה מידע נפנה?

- **מה נשמור:** נעבוד לפי עיקרון הלוקאליות - דברים שקרובים אנו ניגשים אליהם בתדירות גבוהה יותר, כאשר קרובה מתייחס לזמן ומיקום. לכן נשמור מידע קרוב - לוקאליות בזמן: מידע שהשתמשנו בו לאחרונה. לוקאליות במקום: בקריאת פקודות, נרצה לשמור במטמון את שורות הקוד שקרובות לשורת הקוד הנוכחית שאנו קוראים.

- **היררכיית המטמון:** כל רמה בזכרון משמשת כמטמון של הרמה שמעליה.

- **זכרון נדיף:** זכרון שהמידע עליו נמחק ברגע שיש נפילת חשמל.
  הזכרון הראשי הוא זכרון נדיף, בעוד ה $storage$ הוא זכרון שאינו נדיף.

- **הבדלים בהיררכיית הזכרון:** כאשר אנו עובדים עם כמה ליבות, אזי לכל מעבד יש רמה $L2$ פרטית, בעוד רמה $L3$ משותפת בין כל המעבדים.

- **תחומי אחריות:** מערכת ההפעלה אחראית על הזכרון המרכזי ועל ה $storage$, אנו נתעסק בחלקים אלו. החומרה אחראית על $L1, L2, L3$. הקומפיילר אחראי על הרגיסטרים.

8.1.2 הזכרון המרכזי - $Main\ Memory$:

- **זכרון וירטואלי \ לוגי:** זכרון שלא נמצא אצלנו בפועל, אך נחלק אותו בין התהליכים ונדמה לכל אחד מהם מצב כאילו כל הזכרון של המחשב נמצא אצלו. כדי שכל התליך יוכל לבצע את עבודתו בלי לקחת בחשבון תהליכים אחרים שרצים ברקע.

- **מרחב הכתבות:** המקום בו אנו מאתרים את המידע, כשאנו כותבים תכנית אם אנו עובדים עם מערכת הפעלה 32 ביט יש לנו $4G$ של זכרון לשימוש **לכל תהליך**. חלקו חסום ע"י מערכת ההפעלה ונשאר לנו לשימוש חלק מהכמות. לרוב, הזכרון שכל תהליך יקבל $(4G)$ לא יוקצה לו באמת בכרון המרכזי, אלא ינתן לו זכרון וירטואלי.

- $Address\ translation$: תפקידו לתרגם לתהליך את הכתובות המשותפות בין התהליכים.

- MMU: רכיב במעבד שמתרגם את הכתובות הלוגיות (הכתובת הוירטואלי לכל מעבד) לכתובות הפיזיות (הכתובות בפועל). לרוב יש לו זכרון משל עצמו.

- **דרך ראשונה - הקצאת זכרון לתהליכים:** נקצה לתהליכים את הזכרון באופן רציף, ועבור כל תהליך נשמור את כתובת הבסיס - $base$, ונמצא את הכתובת הפיזית ע"י $base + x$. לעיתים מוסיפים משתנה $bound$ שמגדיר את גבול הזכרון לתהליכים.
  חסרונות: מתקיים $fragmentation$.

- **שיברור - $fragmentation$:** בכל פעם שנשמור מידע באופן רציף בזכרון יגרם לנו $fragmentation$. הוא מחולק לשני חלקים:
  $internal\ fragmentation$: זכרון פנוי **בתוך** תהליכים. נצטרך לשמור זכרון ריק עבור כל תהליך כדי שיהיה לו מקום לגדול אליו.
  $external\ fragmentation$: זכרון פנוי **בין** תהליכים. אם נשחרר תהליך שסיים את עבודתו ויגיע תהליך גדול יותר הוא לא יוכל להיכנס, מכיוון שאין מקום לגודל שלו, ובנוסף המידע צריך להישמר באופן רציף בזכרון.

- **אלגוריתמים למציאת מקום פנוי לתהליך חדש:**
  האלגוריתם $best\ fit$: נעבור על כל הזכרון ונמצא את המקום המתאים ביותר. **חסרון** - מעבר על כל הזכרון.
  האלגוריתם $first\ fit$: נעבור על הזכרון, נמצא את המקום הפנוי הראשון ונאחסן בו. **חסרון** - $external\ fragmentation$.
  האלגוריתם $next\ fit$: גישת האלגוריתם היא שנשאף להשאיר את החורים בזכרון שכמה מקומות, נתחיל מהמקום שיצאנו ממנו בזכרון, נמצא את המקום הפנוי הראשון ונאחסן בו. **חסרון** - $external\ fragmentation$.

- **פתרון ל** $external\ fragmentation$: פעם בכמה זמן נעבור על הזכרון ונעשה לו $compaction$ - נדחוס את הזכרון. **החסרון** - פעולת דחיסת הזכרון יקרה לאללה (צריך להעיק את הזכרון למקום שלישי).

8.1.3 דרך שניה להקצאת זכרון לתהליכים - $Paging$:

- **הרעיון:** בשיטה זו לא נקצה את הזכרון באופן רציף אלא נחלק כל תהליך לדפים $pages$ **בגודל קבוע**, ואת הזכרון לתבניות - $frames$ וכל דף נכניס במסגרת אחרת לפי הסדר שנחליט.

- **החסרון:** כיצד ה MMU יוכל לגשת לכתובת הפיזית?

- **כיצד תתבצע השמירה לזכרון:** תתבצע עם מילון. נחלק את הזכרון לתבניות ממוספרות, ונמספר את הדפים של כל תהליך, ועבור כל דף נשמור במילון את התבנית שאליה הוא נשמר. נעשה זאת עבור כל תהליך בנפרד. בנוסף מערכת ההפעלה מחזיקה רשימה של כל המסגרות הפנויות, וכשיגיע תהליך חדש תקצה לו מספר מסגרות בשבילו.

- **כיצד ה MMU יבצע את המרת הכתובות:** נשמור בכתובת לוגית אחת ונחלק אותה כך - הביטים השמאליים יגדירו את הדף ($segment\ number$), הביטים שאחריהם - $offset$ יגדירו את השורה בדף אותה אנו צריכים. לאחר מכן הוא יגש למילון ולפי מספר הדף ימצא את התבנית המתאימה, וישתמש בביטים $offset$ בכדי לגשת לשורה המתאימה בתבנית.

- **מהו גודל הדף האופטימלי:** ככל שהוא יהיה קטן יותר אנו נקטין את ה $internal\ fragmentation$, אך ככל שהוא יהיה גדול יותר - נצטרך לשמור פחות רשומות על הדפים.
  עבור גודל דף - p, ועבור s - גודל הממוצע של תהליך, ו e - רוחב הרשומה בטבלת הדפים. אזי

$$Overhead = (s \cdot e/p) + p/2$$

  כדי למזער אותו נרצה גודל דף של $\sqrt{2s \cdot e}$, כיום - $4KB$.

- **המטמון של המילון - TLB:** גודל הטבלה ששומרת את המילון יכולה להגיע לגודל גדול מאד כך שנצטרך לשמור אותה בזכרון המרכזי. לכך נועד המטמון - והוא ישמור בכל שלב את החלק הרלוונטי בטבלה אותו נמצא לפי מיקום וזמן.

8.2 תרגול 8:

- אין תרגול.

9 שבוע 9 :

9.1 הרצאה 9 - זכרון וירטואלי (8.5):

- **הרעיון:** אם סך כל הזכרון הלוגי גדול יותר מהזכרון הפיזי, נצטרך ליצור זכרון ווירטואלי עבור כל תהליך. מכיוון שאין מספיק מקום על ה RAM חלק מהזכרון שיוקצה לתהליכים יוקצה בהארד דיסק - זיכרון וירטואלי.

- **הערה:** תהליך **לא יכול** לרוץ על הדיסק, כי המעבד לא יכול לגשת ישירות לדיסק.

- **מה נשמור בהארד דיסק:** נעשה הבחנה בין זכרון שאנו צריכים ונמצא בשימוש תדיר לבין זכרון שנמצא פחות בשימוש. נשתמש בעיקרון הלוקאליות כדי להחליט מה ישמר בדיסק, ומה ישמר קרוב יותר.
  כשנקצה זיכרון לתהליך חדש נשמור את הדפים החשובים בזכרון הראשי, ואת שאר הזכרון נשים בדיסק.

- **איזה מידע נשמור בזכרון -** $DEMAND\ PAGING$**:** נשמור רק את המידע הנחוץ לנו. ברגע שתהליך רוצה מידע מהזכרון, והמידע לא נמצא בזכרון הראשי אלא בדיסק, נעתיק את כל הדף מהדיסק לזכרון הראשי.
נצטרך לעדכן את טבלת הדפים שהדף עבר מהדיסק לזכרון הראשי, בנוסף יתכן ונצטרך למחוק דפים אחרים כדי לפנות מקום בזכרון.

- **שיטה נוספת -** $PRE\ PAGING$**:** כשנעלה מהדיסק דף, נשתמש בעיקרון הלוקאליות במרחב ונעלה את כל הדפים שקרובים אליו.

- **שגיאת** $page\ fault$**:** כשננסה לגשת לדף שלא הוקצה לו מקום ($valid\ bit = 0$) נקבל $exeption$ שנקרא $page\ fault$ שיזרוק $interupt$ שיגרום למערכת ההפעלה להתערב. מערכת ההפעלה תפעיל את $DEMAND\ PAGING$ ותעלה את הדף מהדיסק. בנתיים מערכת ההפעלה תעשה $contect\ swich$ ותעלה תהליך אחר ותחכה עד שהדיסק יסיים (יעשה $interupt$). כשהדיסק יסיים, מערכת ההפעלה תעדכן את טבלת הדפים ותעיר את התהליך הראשון שחיכה לדף, והתהליך יחזור ויבקש את הדף שוב.

9.1.1 אלגוריתמי בחירת קורבן - $Replacement\ algorithes$:

- **איזה דפים נפנה מהזכרון:** הדף שנרצה לפנות מהזכרון לטובת דף חדש שיכנס ייקרא הקורבן - $victim$. אנו רוצים למצוא את הקורבן האופטימלי, כך שלא נפנה דף שנצטרך בקרוב.

- $WORKING\ SET$**:** בעולם אוטופי בו אנו יודעים על איזה דפים אנו עובדים כרגע ועל אילו נעבוד בעתיד ובאיזה סדר, נגדיר את סט הדפים הזה להיות ה $WORKING\ SET$. אך מכיוון שאנו לא יודעים את העתיד, ננסה לקרב את ה $WORKING\ SET$ שלנו.
נרצה שדפים שנמצאים ב $WORKING\ SET$ ישארו בזכרון, ואלו שלא יירדו לדיסק לטובת דפים אחרים.

- **הגדרה פורמלית -** $WORKING\ SET$**:** אם אנו מסתכלים על k גישות אחרונות, נבדוק מהם k הדפים האחרונים שניגשנו אליהם ב k הגישות האחרונות. אם הגישות של כל תהליך לדפים הן רנדומיות אזי ב k הגישות האחרונות יהיו לנו בערך k דפים.

אך במציאות מכיוון שיש לוקאליות, כמות הדפים שניגשנו אליהם בפועל תהיה קטנה ממש מ k. וגודל ה $WORKING\ SET$ יהיה 1.

- **כיצד נשערך את ה** $WORKING\ SET$**:** נסתכל על הדפים האחרונים שניגשנו אליהם ונקח אותם. כל האלגוריתמים שלפנינו מנסים לשערך את ה $WORKING\ SET$ לפי הדפים **שהשתמשנו** בהם לאחרונה.

- **מי יעדכן את הביטים:** ה MMU יצטרך במקביל לעדכן מידע על הדפים - $reference\ bit$: האם ניגשנו לדף הנוכחי. $dirty\ bit$: האם עדכנו את הדף הנוכחי.

- **הגדרה -** $Fault\ rate$**:** מספר הדפים שהיינו צריכים להעלות מהדיסק לזכרון, חלקי מספר הדפים הכללי שהשתמשנו בהם (עם כפילויו).

- **האלגוריתם האופטימלי (האלגוריתם של בלאדי):** בהנחה ואנו יודעים את העתיד, ומתי נצטרך כל אחד מהדפים שבזכרון, נפנה את הדף שלא נצטרך בזמן הקרוב. האלגוריתם הזה יהיה הטוב ביותר, והוא ישמש בעיקר להשוואה לאלגוריתמים אחרים ולמדידת יעילות.

- **האלגוריתם הרנדומי:** מפנה דפים באופן רנדומלי מבלי לבדוק אם אנו צריכים אותם. ישמש בעיקר להשוואה לאלגוריתמים אחרים ולמדידת יעילות.

- **אלגוריתם** $FIFO$**:** נפנה את הדף שהיה הכי הרבה זמן בזכרון, מי שנכנס ראשון ייצא ראשון.
  כיצד נשמור מידע: נחזיק רשימה מתי כל דף נכנס לזכרון.
  החסרון: לרוב הדף שנשאר הכי הרבה זמן בזכרון זה כי השתמשנו בו הרבה. חסרון נוסף - אם נגדיל את הזכרון הפיזי יהיו לנו **יותר** $page\ faults$.

- **האלגוריתם** LRU**:** נפנה את הדף שמאז הפעם האחרונה שהשתמשנו בו עבר הכי הרבה זמן. קירוב של האלגוריתם הזה נמצא כיום בשימוש באופן הנרחב ביותר, קשה לממש את החומרה שלו לכן יש נסיונות לקרב אותו.
  מימוש: עיקרון הלוקאליות בזמן. נותן קירוב טוב של ה $WORKING\ SET$.
  החסרון: נצטרך לשמור ביט שייצג מתי השתמשנו בכל דף, ולעדכן מבנה נתונים בהתאם.

- **האלגוריתם** NRU**:** אלגוריתם שמקרב את האלגוריתם LRU. נגדיר זמן $recent$ וכל דף שעבר את הזמן הזה ייצא החוצה.
  מימוש: האלגוריתם משתמש ב $reference\ bit$, פעם בכמה זמן נמחק את רשימת הביטים ונאתחל אותה מחדש להיות 0. כשנרצה להחליף דפים נסתכל על הרפרנס ביט וכל דף שהביט שלו נשאר 0, נוכל להוציא אותו (כי הוא לא היה בשימוש מאז האיפוס).

- **האלגוריתם** LFU**:** נסתכל על התדירות של שימוש בדף, נפנה את הדף שהשתמשנו בו הכי פחות.

- **אלגוריתם השעון (**$Second\ Chance\ FIFO$**)- קירוב ל** NRU**:** נחזיק את ה $frames$ במעגל (מבחינה רעיונית), ויהיה פויינטר שיצביע על אחד מה $frames$ שלכל אחד מהם יש רפרנס ביט. כשנבוא לפנות דף נסתכל על ה $frame$ שעליו מצביע הפויינטר, אם הרפרנס ביט שלו הוא 0 - נפנה אותו, אחרת - נאפס את הביט ונעבור ל $frame$ הבא.

- **אלגוריתם השעון - קירוב ל** LRU**:** בדומה לאלגוריתם השעון ל NRU רק שכאן נוסיף שעון וירטואלי, והוא יהיה אחראי על איפוס הרפרנס ביט. בנוסף - נוסיף פרמטר k שיגדיר מחזורי שעון, וביט נוסף לכל $frame$ - $time\ step$ שישמור את הזמן מהשימוש האחרון. פעם בכמה זמן נעבור על כל ה $frames$ שהרפרנס ביט שלהם שווה ל 1, נאפס אותו ונעדכן את ביט ה $time\ step$ שלהם להיות הזמן שמופיע בשעון הוירטואלי.
  כלומר - הרפרנס ביט יגיד לנו אם השתמשנו בדף לאחרונה, וה $time\ step$ ביט יגיד לנו מתי השתמשנו בו לאחרונה.
  כיצד האלגוריתם יעבוד: כשצריך להוציא דף, נסתכל על הדף שהפויינטר מצביע עליו. אם הרפרנס ביט שלו שווה ל 1 - ז"א שהשתמשנו בו לאחרונה ולכן נתקדם לדף הבא (לא נאפס את הביט). אם הביט שווה ל 0 - ז"א שלא השתמשנו בו לאחרונה, נבדוק מה ערך ה $time\ step$ ביט שלו, אם $virtual\ time - time\ step > k$ נוציא אותו, אחרת - נתקדם לדף הבא.
  אם אף דף לא גדול מ k - נפנה את הדף שהכי קרוב ל k.

- *dirty and clean pages*: כשנפנה דפים נקיים - שלא שינינו יהיה לנו קל יותר כי לא נצטרך לכתוב אותם מחדש לזכרון. לעומת זאת דף ששינינו יצטרך להיכתב מחדש לזכרון ויקח לנו יותר זמן. לכן נרצה לתת עדיפות להורדת דפים שמוגדרים כ *clean page*.

  עדכון אלגוריתם השעון: נוכל להוסיף תנאי באלגוריתם כך שאם הדף *dirty* נדלג עליו ולא נוציא אותו. אם אין אף דף שעומד בנבלים נפנה את הדף עם k הגבוה ביותר.

- **פתרון נוסף:** עבור דפים שלא השתמשנו בהם הרבה זמן וגם מוגדרים כ *dirty* - פעם בכמה זמן נעבוד להעתיק את כל הדפים האלה לדיסק (*lazy copy*), ונשנה את הביט שלהם ל *clean*. כך פעם הבאה שנבוא לפנות דף, הדף הזה יהיה מועמד ראוי כי הוא *clean* ו *old*.

  מכיוון שההעתקה שלהם לדיסק לא קוראת בזמן ההחלפה, אנו לא מבזבזים זמן אא עובדים במקביל.

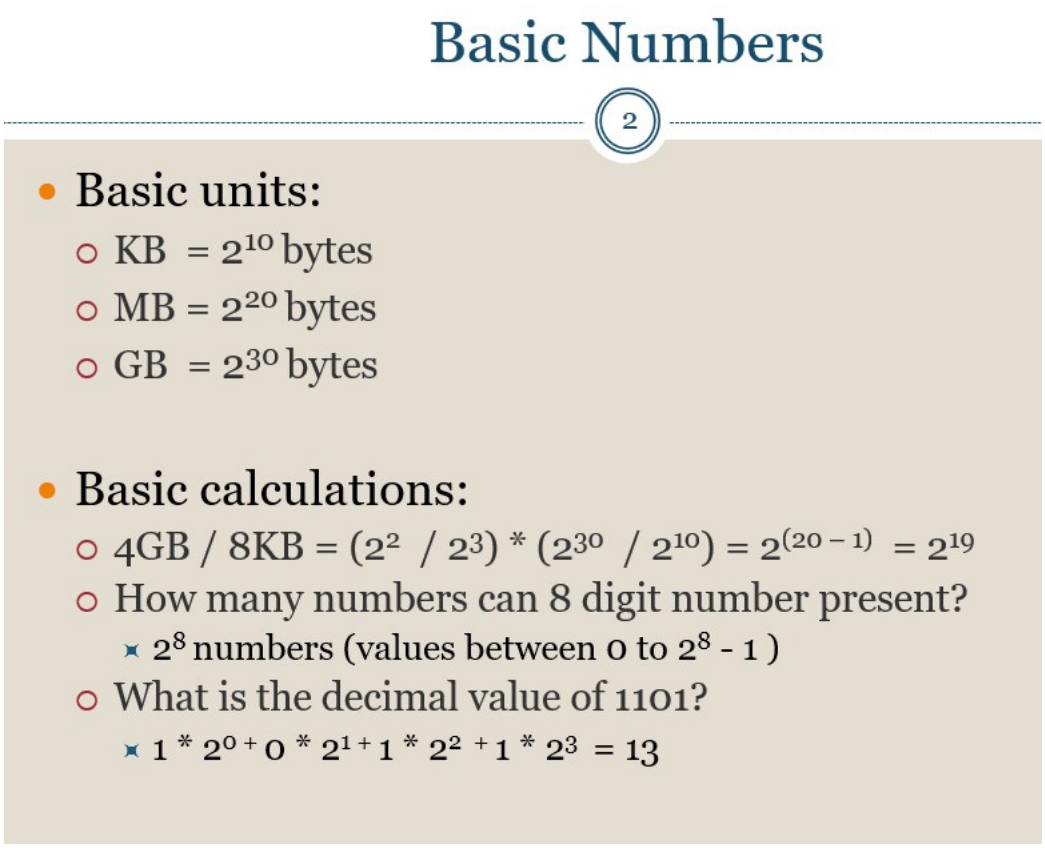
- *local and global paging*: גישות שונות לאחריות שמוטלת על כל תהליך.

  local: כל תהליך מפנה רק את הדפים שהוא אחראי עליהם. **יתרון:** מספר הדפים לכל תהליך קבוע, בנוסף אלגוריתם יותר יעיל יש פחות מקומות לשנות.

  global: תהליך יכול לפנות גם *frames* של תהליכים אחרים. **חסרון:** אפשר לפגוע בתהליכים אחרים. **יתרון:** יש יותר מקומות ולכן נחסוך ב *page flouts*.

9.2 תרגול 9 - זכרון:

- **לגזור ולשמור למבחן:**



- **גודל הזכרון ומספר הכתובות:** עבור זכרון בגודל nGB, כמות הכתובות שיש לנו היא $log(n)$. נניח כי כתובת תצביע לבייט בודד.

- **מערכת הפעלה n ביט:** הגודל של מערכת ההפעלה של המחשב מתייחס לגודל הזכרון הוירטואלי שכל תהליך מקבל. עבור מערכת הפעלה של n ביטים, כל תהליך יקבל 2^n ביטים של זכרון וירטואלי.

9.2.1 דרך שניה להקצאת זכרון לתהליכים - $Segmentetion$:

- **הרעיון:** בשיטה זו לא נקצה את הזכרון באופן רציף אלא נחלק כל תהליך לסגמנטים $(segments)$ **בגודל משתנה** נגדיר $segment$ לכל חלק שנרצה (המחסנית הערימה וכו), ואת הזכרון לתבניות - $frames$ וכל דף נכניס במסגרת אחרת לפי הסדר שנחליט.

- **כיצד תתבצע השמירה לזכרון:** תתבצע עם מילון. נחלק את הזכרון לתבניות ממוספרות, ונמספר את הסגמנטים של כל תהליך, ועבור כל סגמנט נשמור במילון את התבנית שאליה הוא נשמר. נעשה זאת עבור כל תהליך בנפרד. בנוסף מערכת ההפעלה מחזיקה רשימה של כל המסגרות הפנויות, וכשיגיע תהליך חדש תקצה לו מספר מסגרות בשבילו.

- $segmentetion\ fault$**:** נשמור עבור כל $segment$ את כתובת הבסיס שלו - $base$ ואת הגבול - $limit$. בכל פעם שהרגיסטר MMU יתרגם כתובת וירטואלית לכתובת פיזית הוא יבדוק שהכתובת לא חורגת מה $limit$, אם היא חורגת - נקבל שגיאת $segmentetion\ fault$.

- **מידע נוסף שישמר על ה** $segment$**:** $validation\ bit$: ביט שיסמן האם הסגמנט נשמר ל RAM או לדיסק. ביטים נוספים ישמרו מידע האם הסגמנט לקריאה כתיבה וכו.

- **כיצד ה** MMU **יבצע את המרת הכתובות:** נשמור בכתובת לוגית אחת ונחלק אותה כך - הביטים השמאליים יגדירו את הסגמנט ($segment\ number$), הביטים שאחריהם - $offset$ יגדירו את השורה בסגמנט אותה אנו צריכים. לאחר מכן הוא יגש למילון ולפי מספר הסגמנט ימצא את התבנית המתאימה, וישתמש בביטים $offset$ בכדי לגשת לשורה המתאימה בתבנית. הכתובת תימצא ב $base + offset$.

- **החסרון:** $external\ fragmentation$**.**

- **יתרון:** כל תהליך מקבל את הסגמנטים שלו ואף תהליך אחר לא יכול לגשת אליהם - מתקיים עיקרון ה $pritaction$.

9.2.2 $Paging$:

- **אינטואיציה:** מכיוון ש $segmentetion$ גורמת לנו ל $external\ fragmentation$ אנו נרצה שיטה אחרת שתתמודד עם הבעיה ביעילות.

- **הרעיון:** בדומה ל $segmentetion$, אך כאן נחלק את הזכרון לדפים **בגודל קבוע**, כך נפתור את הבעיה של $external\ fragmentation$.

- **שמירה לזכרון:** יהיה לנו מילון ($page\ table$) עבור **כל תהליך** שיכיל לכל $page$ את ה $frame$ שלו. אין צורך לשמור משתנה $limit$ כי לכל הדפים יש את אותו הגודל.
  בנוסף מערכת ההפעלה תשמור רשימה של כל ה $frames$ הפנויות.

- **כיצד ה** MMU **יתרגם את הכתובות:** בכתובת תתחלק לשני חלקים - הביטים השמאליים ישמרו $page\ number$, והביטי הימניים $offset$ ישמרו את הכתובת אליה אנו רוצים להגיע בדף.
  עבור מערכת הפעלה בגודל m, וגודל הדף בגודל 2^n אזי: עבור כתובת בגודל m - $m-n$ הביטים השמאליים ייצגו את מספר הדף, ו n ביטים הימניים ייצגו את ה $offset$.

- $valid\ bit$: ביט שייצג האם הדף שמור כרגע ב RAM או בדיסק.

- $Modified$: ביט שייצג האם הדף השתנה מאז הפעם האחרונה שקראנו אותו מהדיסק.
  למה זה משנה לנו: כך נוכל לדעת האם צריך לקרוא את הדף מחדש או שנוכל להשתמש בדף שכבר שמור לנו בזכרון, וכך לחסוך גישות לדיסק.

- $Used\ bit$: ביט שמייצג חותמת זמן של הדף. רלוונטי לאלגוריתמים שמביאים דפים חדשים לזכרון ומפנים דפים ישנים.

- $Access\ premition$: ביט שמייצג הרשאות - קריאה כתיבה וכו.

- **שגיאת** $Page\ fault$: כשנרצה לגשת לדף אך הוא לא נמצא ב RAM, ($valid\ bit = 0$).
שגיאה זו לא מקריסה את התכנית, אלא מפעילה את אלגוריתם ההחלפה ומעלה את הדף מהזכרון על חשבון דף לא רלוונטי.

- **החסרון:** טבלת הדפים תבזבז לנו מלא זכרון והיא לא תיכנס כולה בזכרון הפיזי.
הפתרון: טבלת דפים היררכית או טבלת דפים הפוכה.

9.2.3 טבלת דפים היררכית - $Multy\ lavel\ page\ table$:

- זאת השיטה שנמצאת קיום בשימוש הרחב ביותר.

- **הרעיון:** לא נשמור את הטבלה כולה בזכרון המרכזי, אלא נפרק אותה לעץ טבלאות. תהיה טבלת שורש שכל שורה בה מצביעה לטבלה אחרת ברמה 2. כל טבלה ברמה השניה תהיה דף.

- **הפתרון** מתבסס על כך שלא נצטך את כל תתי הטבלאות מהרמה השניה, כי לא כל תהליך תופס את כל המקום שהוקצה לו.

- **כיצד הכתובת תישמר:** הכתובת תתחלק לשניים - מספר דף ו $offset$ כמו בטבלה שטוחה. אך מספר הדף יתחלק גם הוא לשניים - כאשר החלק הראשון ייצג את המספר בטבלה ברמה הראשונה, והחלק השני יסמן את מספר הדף בטבלה השניה.

- TLB: זכרון קטן שבו נשמור חלק מהטבלה כדי שלא נצטרך לבצע בכל פעם גישות לזכרון, משמשת מעין מטמון ל $page\ table$.

9.2.4 טבלת דפים הפוכה - $Inverted\ page\ table$:

- **הרעיון:** תהיה לנו טבלת דפים אחת לכל התהליכים. ואנו נבצע מיפוי הפוך - בכל מסגרת נשמור איזה דף של איזה תהליך שמור בה.

- **כיצד יתבצע תרגום הכתובות:** כל כתובת לוגית תתחיל עם $prosses\ ID$, לאחר מכן ישורשר אליה מספר דף ו $offset$.

- **גישה לכתובת הפיזית:** נצטרך לבצע חיפוש לינארי על כל $prosses\ ID$ ולמצוא את התהליך שלנו ולאחר מכן למצוא את הכתובת שממופה לו.

- **החסרון:** אמנם חסכנו מקום, אך אנו מבצעים מלא גישות לזכרון.

10 שבוע 10:

10.1 הרצאה 10 - (15.5):

- **ביצועים:** נגדיר את p להיות ההסתברות ל $page\ fault$. כשנרצה לחשב את העלות הממוצעת של הגישה לזכרון נצטרך לחשב את הזמן עם $page\ fault$ ובלי $page\ fault$ ולשב ממוצע בניהם.
חישוב העלות:

$$\text{Effective access time } = p(\text{ page fault time }) + (1-p)(\text{ memory access time })$$

נחשב את האטת המערכת כך:

$$\text{Slowdown} = \text{Effective access time / memory access time}$$

לכן נשאף להקטין את p, כדי לעשות זאת נצטרך אלגוריתם החלפה יעיל.

- **כתובות פופולריות:** נרצה לשמור בזכרון כתובות שיוגדרו כפופולריות - כתובות שהשימוש בהן הוא תדיר. פופולריות נמדדת עם הסתברות $Zipf$.

- **מדידת לוקאליות -** $Stack\ distance$**:** דרך למדידת לוקאליות. נחזיק מחסנית שתייצג גשות לכתובות, כשנצטרך לגשת לכתובת מסויימת נבדוק האם ניגשנו אליה בעבר, אם כן - היא נמצאת במחסנית, נסתכל על המיקום שלה במחסנית ונעלה אותה לראש התור. המיקום שלה במחסנית יסמן לכמה כתובות ניגשנו מאז הגישה האחרונה לכתובת הנוכחית.
הערה: אם יהיה לנו זכרון מרכזי שקרוב לגודל הסטאק אנו נפחית מאד את ה $page\ fault$.

- $Stack\ distance$ **עם** LRU**:** אם גודל הזכרון המרכזי $= k$, והמחסנית גדולה ממש מ k. נוכל למדוד את ההסתברות ל $page\ fault$ ע"י ההסתברות של דף להיות גדול מ k

$$p = \text{ probability of } > k \text{ in the stack}$$

- **הגדרה -** $Streaming$**:** מידע שאנו משתמשים בו פעם אחת בלבד. נשים לב כי אין טעם לשמור אותו ולכתוב אותו מחדש לדיסק.

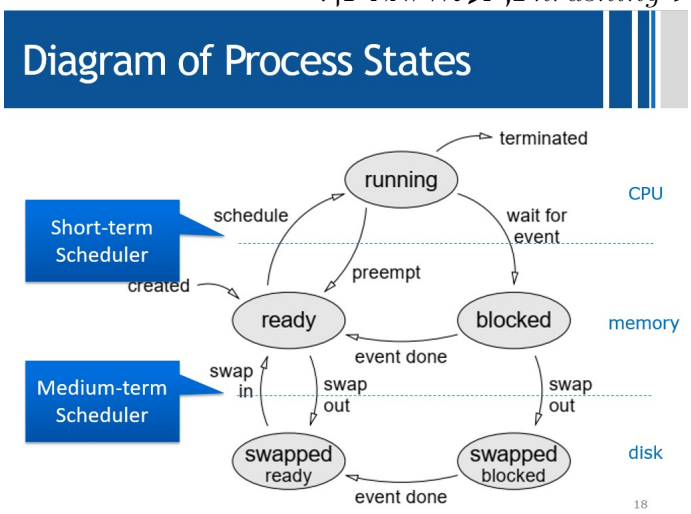
- **ייעילות עם דאטה שונה:** נוכל למפות את הדאטה לשני סוגים ולטפל בהם בהתאם. מידע שיוגדר $Streaming$ יטופל בדרך אחת, ומידע נחוץ יטופל עם אלגוריתם אחר.

10.1.1 $Thrashing$:

- **הגדרה -** $Thrashing$**:** ייעילות המעבד נמצאת בעליה ככל אנו מוסיפים תהליכים שירוצו במקביל, אך בשלב מסויים עם ריבויי התהליכים המערכת מוצפת כי מתרחשים מלא $contact\ swich$ ואף אחד מהתהליכים לא מצליח לתפוס את המעבד. תכונה זאת נקראת $Thrashing$.

- **מתי יתרחש** $Thrashing$: כאשר הגודל של ה $working\ set$ גדול יותר מהזיכרון הפיזי.

- **הפתרון:** נגביל את כמות התהליכים שיכולים לרוץ בו זמנית, אך לא את כמות התהליכים שיכולים להיווצר במערכת. נעשה זאת ע"י הוספת תורים חדשים שייצגו תהליכים שאנו רוצים להשאיר בתור החדש (המתנה) כדי שלא יגרמו לנו ל $Thrashing$, נעשה זאת כך:



- **תזמון:** נחזיק שני $Schedulers$, אחד שיתזמן את התורים הרגילים, ואחד נוסף שיתזמן את תורי ההמתנה ויעבוד במקצבים ארוכים יותר מהמקצבים של ה $Scheduler$ הרגיל.

- $Big\ memory$

נחשב את גודל ה $page\ table$: עבור מעבד של $32\ bit$ כל דף בגודל $4G$, כל תהליך צריך 2^{20}דפים, בהנחה שייצוג כל דף בטבלה תופס 4 בתים, אנו צריכים סה"כ 2^{22} בתים כדי לייצג את הטבלה. זאת אומרת שרוב הטבלה לא תוכל לשבת על הזכרון.

10.1.2 טבלת דפים היררכית:

- **הרעיון:** נחלק את טבלת הדפים לכמה טבלאות, כמו בעץ. את הטבלה הראשונה נחלק לשורות, כך שכל שורה תפנה אותנו ל $frame$ שמכיל תת טבלה ברמה השניה.

- **ייתרונות:** נוכל לפרק את הטבלה לתתי טבלאות בגודל דף ולהחזיק אותן בנפרד (לא באופן רציף).
בנוסף, כמו דפים נוכל להחזיק אותן בדיסק.
רוב הטבלאות יהיו ריקות ולא נצטרך לשמור אותן כלל, כך נחסוך בזכרון.

- **כיצד נקרא מהזכרון:** נקבל כתובת לוגית, ונחלק אותה באופן הבא: עבור $frame$ בגודל n נקח $log(n)$ ביטים בכל פעם והם ייצגו לנו את הטבלה ברמה ה i.
נקח את הכתובת מהטבלה האחרונה בעץ ונשרשר לה את ה $offset$ כך נגיע לכתובת המדוייקת אותה אנו מחפשים.

- **מספר הגישות לזכרון:** עבור כל גישה לזכרון מספר הגישות הכללי יהיה גובה העץ + 1.

- **הערה:** בגישה אחת לזכרון יכול להתרחש יותר מ $page\ fault$ אחד. נתמודד עם זה ע"י TLB שישמור את הטבלאות שבשימוש נפוץ.

10.1.3 טבלת דפים עם האש:

- **הרעיון:** נמיר את הבעיה לפונקציית האש שאיתה אנו רוצים לשמור מפתחות p והערכים הם r (ה $frame$ בו בכתובת הפיזית נמצאת). נפעיל פונקציית האש על p ונקבל ערך שממופה אליו.
- **חסרונות:** המקרה הגרוע יכול לעלות לנו הרבה.

10.1.4 טבלת דפים הפוכה - $Inverted\ page\ table$:

- **הרעיון:** כשהזכרון הפיזי קטן, נשמור באופן הפוך, טבלה אחת לכל התהליכים. עבור כל כתובת בזכרון הפיזי נשמור איזה כתובת לוגית שמורה בה ולאיזה תהליך היא שייכת.
- **החסרון:** נצטרך לעבור על כל הטבלה כדי למצוא את התהליך שלנו ואחכ לחפש את הדף הרלוונטי. זמן הריצה הוא גודל הזכרון הפיזי.

10.1.5 הגנה - $Protaction$, ושיתוף זכרון:

- **הגנה:** כשאנו משתמשים בטבלת דפים נפרדת לכל תהליך אנו ממשים את עיקרון ההגנה. כך אתף תהליך לא יכול לגשת לדפים של תהליך אחר, והוא בכלל לא מודע לקיומם של דפים אלו.
- **שיתוף זכרון:** כשתהליכים ירצו לשתף זכרון, אנו נשתף דפים בין טבלאות דפים של כל תהליך. (טבלאות הדפים של כל התהליכים שמשתפים את הזכרון, יצביעו על אותו ה $frame$ בזכרון הפיזי).

10.2 תרגול 10 - אלגוריתמי החלפה (18.5):

- **מה אנו מחפשים:** נרצה אלגוריתם שיבחר לנו את החלפת הדפים בין ה RAM לדיסק, בצורה האופטימלית - כמה שפחות החלפות.
- שאר התרגול מופיע בהרצאה 9.

11 שבוע 11:

11.1 הרצאה 11 - קבצים (22.5):

11.1.1 קבצים:

- אבסטרקציה שמערכת ההפעלה מדמה לנו לאחסון מידע במחשב.
- **הגדרה - קובץ:** אחסון מידע בצורה רציפה עם חשיבות לסדר, בזכרון שאינו נדיף וניתן לגשת אליו בעזרת שם.

- **הגדרה - שם** $(Naming)$**:** מטא דאטה, כדי שנוכל לזהות את הקובץ ולגשת בקלות (השם לא נשמר במטא דאטה של הקובץ, אלא בתוכן של התקיה שמכילה את הקובץ).

- **הגדרה - אינו נדיף** $(Persistence)$**:** הקובץ נשמר בזכרון שאינו נדיף, ולכן נשמר לאורך זמן, הוא שורד גם לאחר שהתהליך נגמר.

- **מטא דאטה של קובץ\ אטרביוטים:** דאטה שנשמר בנוסף לקובץ ומכיל מידע על הקובץ, כגון - גודל, בעלים, הרשאות, חותמות זמן, מיקום בדיסק, סוג ועוד.
מתוחזק ע"י מערכת ההפעלה, והמשתמש יכול לקרוא ולשנות אותם. בנוסף מערכת ההפעלה תחזיק אטרביוטים שמיועדים רק לה והמשתמש לא יכול לגשת אליהם, אטרביוטים אלו ייצגו שם ומיקום קובץ בשיטה יעילה יותר למחשב (מספרים) מאשר אותיות. המידע נשמר במבנה הנקרא $inode$.

- **סיומת קובץ:** בחלק ממערכות ההפעלה סיומת הקובץ תגדיר באיזו אפליקציה נפתח את הקובץ ($linux$ לא מתייחסת לסיומת).

- **חלוקת האחריות על קבצים בין האפליקציות למערכת ההפעלה:**
מערכת ההפעלה: מספק את התשתית ואת התמיכה בקבצים. אחראית לכך שהנתונים יישמרו ויהיה ניתן לקרוא אותם, בנוסף היא מספקת את המטא דאטה - שם קובץ. היא מנסה להתערב כמה שפחות בפעילות האפליקציות.
האפליקציות: אחראית על יצירת הנתונים, פירוש הנתונים, שימוש בנתונים, שמירת הנתונים והגדרה באיזה פורמט הם יישמרו.

- **הגישה בפועל:** כשנרצה לגשת למערכת הקבצים, האפליקציה תבצע $system\ call$ למערכת ההפעלה, לאחר מכן יתבצע $contect\ swich$ ומערכת ההפעלה תטפל בפניה. בגישה להתקני $I\backslash O$ יתבצע $interupt$ בסיום הפעולה.

- API **של מערכת הקבצים:** יצירה, מחיקה, פתיחה, סגירה, כתיבה, קריאה, $seek$ - קריאה החל ממקום מסויים, חיתוך, עדכון אטרביוטים, שינוי שם.

- **ספריות\תקיות וארגון קבצים:** נאחסן את הקבצים בתקיות מכמה סיבבות:
יעילות: נוכל לאתר את הקובץ במהירות בתקיות שמדמות עץ.
שמות: מאפשר לתת שמות דומים לקבצים שמאוחסנים שתיקיות שונות
הקבצה: נוכל לקבץ קבצים מאותה קטגוריה יחד בתקיה אחת.

- **ספריות\ תקיות:**
התוכן של כל תקיה יכול להיות תת תקיה או מצביע לקובץ מסויים.
ניתן להגיע לקובץ ע"י נתיב אבסולוטי - נתיב מהשורש עד התקיה עצמה, או נתיב רלטיבי - מיקום ביחס לתקיה הנוכחית .

- **הערה:** מערכת ההפעלה מנהלת את שמות הקבצים בנפרד מניהול הדאטה.

- **מבנה הנתונים של הספריות ולינקים:** הספריות לא חייבות להישמר במבנה עץ, אלא יכול להתרחש לינק.
$hard\ link$**:** שני ענפים (ספריות) שונים יצביעו על אותו עלה (אותו דאטה).

$symbolic\ link$: שני ענפים מצביעים לשם של אותו הקובץ.
הבדלים: במקרה שנשנה את שם התקיה - $hard\ link$ יצליח לגשת לקובץ מהתקיה ששמה לא שונה. במקרה של $symbolic\ link$ אנו נאבד את הקובץ מכל המקומות שהצבענו עליו.

- **יתרונות של לינקים:** ניתן לשמור קובץ כמה פעמים בתקיות שונות עם מצביע וכך לחסוך במקום. בנוסף שינוי של הקובץ ישנה אותו בכל תקיה שמצביעה עליו.
חסרונות: אם נמחק קובץ שיש לו מצביע ממקום אחר, הוא ימחק מכל המקומות. הפתרון הוא לשמור מצביע לשם לאחר השינוי.

- **הפקודה** $mount$: הוספת מערכת קבצים חדשה למערכת קבצים קיימת. לדוגמה הוספת קבצים מהתקן USB למחשב. ניתן לבצע את הפקודה גם ב $remote$ מהאינטרנט - למעשה נשלחת פקודה למחשב שיעדכן את התקיות.

11.1.2 הגנה על קבצים:

- **הגנה על קבצים:** כשיוצרים קובץ אנו קובעים מי יכול לגשת אליו ומה הוא יוכל לבצע בקובץ, בעזרת הפקודה $change\ mode(chmod)$ שיוצרת $system\ call$ ומשנה את האטרביוטים של הקובץ.

- **ייעול:** כדי לייעל את התהליך במקום להגדיר הרשאות לאינדיבידואלים, נוכל להגדיר הרשאות על קבוצת משתמשים. (ניתן לתת הרשאות למשתמש ספסיפי).
ההרשאות הן - $RWX(read, write, execute)$ ומיוצגות בעזרת שלשה ביטים.

- **הרשאות בלינוק ווינדוס:** בלינוקס: לכל קובץ יש רק קבוצה או בעלים אחד.
בווינדוס: ממשים את שיטת $access\ controler$ ניתן להגדיר לכל קובץ ולכל משתמש מה ההרשאות שלו על הקובץ.

- $Access\ controler$: מגדירים לכל קובץ $access\ controler\ list\ (LCA)$ משלו, שמגדירה קבוצה של משתמשים שיש להם הרשאות לקובץ ומה הם יכולים לבצע בקבץ. אם אין LCA - כולם יכולים לבצע הכל על הקובץ. אם ה LCA ריקה - אסור לעשות עליו כלום. אם יש לו כמה משמשים ב LCA - נעבור אחד אחד ונראה מה ההרשאות שלהם.

11.1.3 מימוש מערכת הקבצים:

- **גישה לקבצים:** ניתן לגשת באופן סדרתי (מההתחלה), או גישה רנדומלית מאמצע הקובץ. לרוב קבצים הם סדרתיים.

- **שברור -** $Fragmentation$: נרצה להימנע מחלקים ריקים בזכרון שאנו לא יכולים להשתמש בהם, הפתרון הוא לחלק את הקובץ לבלוקים (כמו $pages$ בזכרון).

- **הסתכלות על קובץ מנקודות מבט שונות:**
משתמש: הדאטה רציף. API: סדרה של בתים שניתן לקרוא ולכתוב אותם. **מערכת הקבצים:** אוסף של בלוקים $4K$. **דיסק:** סקטורים - יחידות קטנות מבלוקים ($512b$).

- **קריאה וכתיבה מקובץ:** כשהמשתמש מבקש לקרוא טווח מסויים מקובץ - מערכת ההפעלה תעלה את כל הבלוק מהזכרון ותציג למשתמש את הטווח הרצוי.

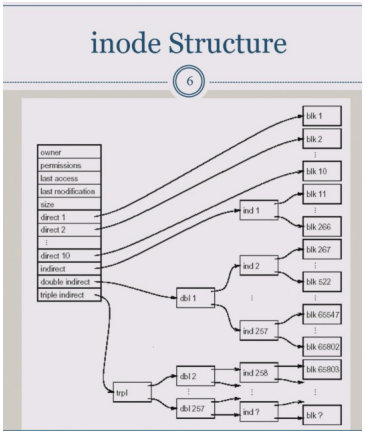
כתיבה לטווח רצוי - מערכת ההפעלה תעלה את הבלוק, תשנה את הטווח הרצוי ותכתוב מחדש את כל הבלוק לדיסק. מכיוון שמערכת ההפעלה יכולה לגשת רק לבלוקים שלמים.

- **שמירת קבצים שיטה ראשונה - רציפות:** נעדיף לשמור את הקבצים בדיסק בצורה רציפה בסקטורים צמודים, ועל אותה הטבעת כדי לחסוך זמן.
החסרונות: במקרה של שינוי הקובץ והוספה אליו נצטרך לפצל לכמה בלוקים שלא ישמרו ברציפות. בנוסף יכול להיגרם $Fragmentation$.

- **שמירת קבצים שיטה שניה - בלוקים:** נשמור רשימה מקושרת של בלוקים, נוסיף לכל בלוק מצביע שיצביע לבלוק הבא. הבלוק האחרון יסומן ב eof.
חסרונות: יהיה קשה מאוד לגשת רנדומלית לאמצע הקובץ כי אנו חייבים הלתחיל מהבלוק הראשון. בנוסף גודל הבלוק יהיה שונה מחזקה של 2 בגלל הפויינטר.

- **שמירת קבצים- FAT:** נשמור את כל המצביעים בבלוק נפרד (טבלה)מספר מצביעים כמספר הבלוקים בדיסק. בקובץ נשמור מצביע לבלוק הראשון ולאחר מכן נתקדם לפי הטבלה. כך לא נצטרך לעבור על כל הבלוקים אלא רק על הטבלה, בנוסף נפתרה הבעיה של חזקת 2 כי האטרביוטים יישמרו בבלוק נפרד.

- **שמירת קבצים - $inodes$:** $inodes$ הוא מבנה ששומר מידע על המטא דאטה של הקובץ בניהם מיקום הבלוקים בדיסק. פתרון זה מבוסס על ההנחה כי רוב קובצים צורכים 12 בלוקים, לכן עבור כל קובץ נשמור מצביעים ל 12 בלוקים בגישה מהירה $(direct)$, ונשמור מצביע לטבלת פויינטרים לשאר הבלוקים במיקום מרוחק יותר - טבלת בלוקים משלימים $(indirect, double\ indirect, triple\ indirect)$. סה"כ נשמור טבלאות עד עומק 3.
בשיטה זו לכל קובץ ניתן להגיע תוך לכל היותר שלש גישות לדיסק.
תרשים של $inode$**:**



- $Maemmory\ maped\ files$: אופציה לשים קובץ מסויים במיקום מסויים בזכרון של התהליך הספציפי שרץ, כך שלא נצטרך ללכת כל פעם לדיסק. זהו מיקום נפרד בזכרון מה $buffer\ cach$.

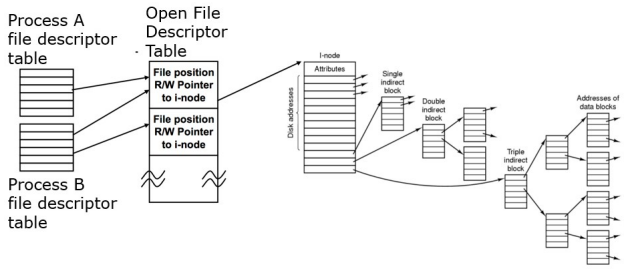
11.2 תרגול 11 - היררכיית זיכרון (24.5):

- **תקיה:** קובץ שה $type$ שלו מוגדר להיות תיקיה. גם לה יש מטא דאטה, ותוכן \ מדריך. התוכן של התקיה מכיל מיפוי של כל שמות הקבצים שנמצאים בתיקיה, למצביע ל $inode$ של הקובץ.

11.2.1 מימוש קבצים במערכת ההפעלה:

- **שמירת שם קובץ:** שמות קבצים לא נשמרים ב $inode$ של הקובץ, אלא הם נשמרים בתוכן של התקיה ששומרת אותם.
- $SperBlock$: אובייקט שמערכת ההפעלה שומרת ב RAM, והוא מכיל מידע על מערכת הקבצים - רשימה חלקית של בלוקים פנויים ורשימה חלקית של ה $inodes$ הפנויים, ועוד. כך ניתן ליצור קובץ חדש ולעדכן קובץ קיים.
- **הקצאת** $inodes$: כשמערכת הקבצים מאותחלת לראשונה אנו יוצרים רשימה סופית של $inodes$ שנשמרת בדיסק.
  כאשר $inode$ משוחרר זה אומר שהקובץ נמחק, לכן נוסיף אותו לרשימת הפנויים.
  כשניצור קובץ נקח $inode$ פנוי מהרשימה, אם הרשימה ב $SperBlock$ ריקה, נלך לדיסק להביא רשימה חדשה של $inodes$ ריקים.
- **הקצאת בלוקים בדיסק:** באותו האופן כמו עם $inodes$ רק בבלוקים.
- **ייצירת קובץ:** כשנפתח קובץ מערכת ההפעלה תקצה לו בלוקים בדיסק ותקשר אותם ל $inode$ של הקובץ, לאחר מכן נוכל לכתוב אליהם או לקרוא מהם. כשנקצה את הבלוקים מערכת ההפעלה תשמור אותם בזכרון ותכתוב אותם לדיסק מאוחר יותר, משום שאנו רוצים לחסוך גישות לדיסק.
- **פתיחת קובץ:** כשנפתח קובץ fd =open("myfile" , R) יחזור לנו מספר אי שלילי שנוכל לשמור ולבצע עליו פעולות קריאה וכתיבה.
  ערך ההחזרה בפתיחת קובץ הוא אינדקס של מערך שנקרא $file\ descriptor\ Table$.
  בשלב פתיחת הקובץ יחזור לנו מצביע ל $inode$ של הקובץ, והבלוקים עדיין לא ייטענו.
- $file\ descriptor\ Table$: טבלה שנשמור **עבור כל תהליך בנפרד**. תשמור את רשימת כל הקבצים שפתוחים עבור תהליך הנוכחי, עבור כל קובץ נשמור מצביע לטבלת $open\ files\ table$ שתשמור $inode$ של הקובץ. ת'רדים של תהליך מסויים ישתפו בניהם את הטבלה.
- **קבצים ותהליכים:** בערכת הפעלה לינוקס בכל פעם שתהליך נוצר נפתחים איתו שלשה קבצים. קובץ אחד ייצג קלט מהמשתמש, קובץ שני ייצג פלט, ושלישי ייצג $errors$.
- $open\ files\ table$: טבלה שתשמור לנו מצביע ל $inode$ לכל קובץ שפתוח **מכל התהליכים שרצים** (מצביע לשורה ב $inode\ Table$). בנוסף נשמור ביטים שייצגו הרשאות, ו $offset$ שיצביע על המיקום שלנו בתוך הקובץ ($offset = 0$ מצביע לתחילת הקובץ).
- $inode\ Table$: טבלה ששומרת את כל ה $inodes$, אחד לכל קובץ.

- **תרשים טבלאות הקבצים במערכת ההפעלה:**



- **קריאה מקובץ:** נשלח לפונקציה $read(fd, buff, bytesToRead)$ את המצביע שקיבלנו מ $open$, באפר ריק אליו ייקרא המידע מהקובץ, מספר הבתים אותם אנו רוצים לקרוא.
מערכת ההפעלה תלך לטבלה $open\ files\ table$ ותוציא ממיקום fd את המצביע ל $inode$ של הקובץ, ותטען את כל הבלוק המתאים אל זכרון מערכת ההפעלה ל $buffer\ cach$. לאחר מכן תעתיק לבאפר את מספר הבתים הנדרש.

- $buffer\ cach\backslash disk\ cach$**:** מקום בזכרון שמערכת ההפעלה תשמור בו את הבלוקים שהעלנו מהדיסק ועליהם אנו רוצים לעבוד. אחרי שנקרא מה $inode$ הוא ישאר בבאפר, ורק כשנקרא בלוק חדש נרד לזיכרון.
מספר חסרונות:
1: מכיוון שהכתיבה לדסק מתרחשת ב $buffers$ ולא פר בלוק, לפעמים קורה שאין סנכרון בין הדיסק לקאש.
2: המידע על הבאפר נדיף. ולכן יכול להימחק במקרה של הפסקת חשמל, או הוצאת דיסק און קי באופן לא מבוקר.

- **כתיבה לקובץ:** מערכ ההפעלה תיגש את הבלוקים של הקובץ באותו האופן שניגשנו בקריאה, ותתחיל לכתוב לתוך הבלוק הנדרש. אם הבלוק יתמלא ונשאר עוד בתים לכתוב, מערכת ההפעלה תפתח בלוק חדש, תקשר אותו ל $inode$ של הקובץ ותכתוב עליו את מה שנשאר לכתוב.

- **גישות לדיסק:** כשאנו ניגשים לתקיה אנו ניגשים לבלוק של התקיה ולאחר מכן קוראים וניגשים לבלוק של ה $inode$ של התקיה או הקובץ שאנו צריכים לגשת אליו. לכן עלות גישה לכל תקיה היא 2.

12 שבוע 12:

12.1 הרצאה 12 - וירטואליזציה (29.5):

- **הרעיון:** נרצה להפריד בין התכנה שאנו מריצים לבין החומרה עליה אנו מריצים את התכנית. כך ניתן לחסוך בשרתים וליצור שרת וירטואלי, ולהריץ שני מערכות הפעלה שונות על אותו המחשב (אותה חומרה) בו זמנית.

- **תהליך:** תהליך שרץ במחשב הוא סוג של VM, הוא מדמה לאפליקציה וירטואליזציה כאילו היא רצה לבד על המחשב. יש לה זיכרון וירטואלי, מערכת קבצים אבסטרקטית, וגישה למעבד.

12.1.1 מכונות וירטואליות (VM):

- **מכונה וירטואלית** (VM)**:** מכונה וירטואלית אמיתית היא תוכנה שגם מהצד שלה **וגם מצד מערכת ההפעלה** היא חיה לבד בעולם. מבחינת מערכת ההפעלה המכונה הוירטואלית היא החומרה עליה אנו רצים. VM יושבת מעל מערכת

ההפעלה, ומדמה חומרה שעליה רצות כמה מערכות הפעלה.

- $Hypervisor\backslash VMM$: השכבה שמתאמת בין המכונה הוירטואלית לחומרה הפיזית. תפקידה גם לנהל את חלוקת החומרה בין המכונות הוירטואליות השונות שרצות יחד על המחשב.
  למעשה היא מנהלת את המכונות הוירטואליות כמו שמערכת ההפעלה מנהלת את התהליכים שרצים במערכת.

- $Multiplexing$: היכולת להריץ על אותה החומרה כמה מכונות וירטואליות שונות.

- **בידוד** - $isulation$: בידוד המכונות הוירטואליות כך שאחת לא תדע על קיומה על האחרת, על אף שהן משתפות את החומרה.

- **תכונות שמכונה וירטואלית צריכה לקיים:**
  $Equivalence$: מכונה וירטואלית צריכה להיות מסוגל לבצע כל מה שמכונה פיזית יכולה לבצע.
  $Safety$: מכונה וירטואלית מבודדת מהמכונה הפיזית עליה היא רצה, וממכונות וירטואליות אחרות הרצות איתה על אותו מחשב.
  $Preformance$: נרצה שההפחתה במהירות תהיה קטנה.

- **סוגי וירטואליזציה עם** $Multiplexing$:
  זיכרון וירטואלי: חלוקת זיכרון בין תהליכים.
  חלוקת הדיסק: חלוקת הזכרון בין תהליכים כך שמבחינתן כל הדיסק שלהם.
  רשת וירטואלית: מדמה מצב של רשת פנימית על אף שאנו משתמשים ברשת הגלובלית.

- **וירטואליזציה עם** $Aggrigation$:
  $RAID$: מספר דיסקים שמחזיקים העתק של אותו הדיסק כגיבוי. כך תמיד המידע ישאר זמין והמשתמש לא חשוף לכך שיש מספר דיסקים, ומבחינתו יש דיסק אחד.

- **למה אנו צריכים מכונות וירטואליות:**
  1: בכדי שנוכל להריץ תכנות על מערכות הפעלה שונות, באותה מערכת.
  2: ניצולת של שרתים, בעולם הרגיל כל שרת מריץ את המידע שלו, אך זה לא יעיל, וירטואליזציה נותנתלנו את היכולת לאגד דברי ולעבוד על שרת בודד לכמה מערכות..
  3: אבטחה - יש לנו בידוד בין המכונות ואין שיתוף תהליכים.
  4: זמינות - נוכל להריץ את אותה המכונה על מחשבי שונים, ניתן להעביר אותה בקלות.

- **אנקפסולציה:** היכולת להעתיק, לשכפל ולהעביר מכונת וירטואלית בקלות עם כל המידע והאפליקציות ששמורות בתוכה.

- **שלשת סוגי** $Hypervisor\backslash VMM$: יש שלשה $types$ -
  $Bare\ metal$: מריצים את ה $Hypervisor$ על החומרה ממש.
  $Hosted$: ה $Hypervisor$ הוא אורח במערכת הפעלה אחרת, ורץ במקביל אליה.
  $Container$: ה $Hypervisor$ רץ מעל מערכת ההפעלה.

12.1.2 כיצד נעשה התיאום בין המכונה הוירטואלית לחומרה הפיזית:

- יש ארבע שיטות לתרגום החומרה הפיזית לוירטואלית, ודימוי המצב למכונה הוירטואלית.

- *Trap and Emulate*: מתייחס ל *Hypervisor* *type* 1, המכונות הוירטואליות רצות ב *user mood* על החומרה. בכל פעם שאפליקציה רוצה לבצע *system call* ה *Hypervisor* מקבל את הקריאה ומבצע אותה או מעביר אותה למערכת ההפעלה האורחת.
  בשיטה זו למעשה ה *Hypervisor* מתפקד כמערכת ההפעלה.

- *Dynamic binary Translation*: נשתמש בה במצב שבו אין הרבה *traps* - לא בכל גישה יש *trap*. נתרגם את קוד מערכת ההפעלה לקוד בטוח, פקודות רגילות שאינן *privilage instructions* - נתרגם רגיל, פקודות רגישות - נגדיר למערכת ההפעלה לקרוא ל *Hypervisor* שיכנס לפעולה בצורה דינאמית.
  בשיטה זו ה *Hypervisor* נעזר במערכת ההפעלה במקרה הצורך.

- *Paravirtualization*: נשנה את מערכת ההפעלה המאחרת, נשנה את הקוד כך שהיא תקרא ל *Hypervisor* כשנצטרך אותו, אך בשונה מהגישה הבינארית לא נעשה זאת בצורה דינאמית, אלא נקמפל את הקוד בכל פעם מחדש. החסרון הוא שהמכונה הוירטואליית צריכה לקמפל את הקוד, והיא לא באמת מדמה וירטואליזציה, כי היא לא יכולה לרוץ תמיד על כל מערכת כמו שהיא.

- *Hardware Assistance*: עבור *type* 1, המעבד יתמוך ביותר *moods* ולא רק *user* ו *kernel* אלא הוא יוכל להריץ גם מכונות וירטואליות.

וירטואליזציה של הזכרון הוירטואלי: יש לנו כמה שלבים של וירטואליזציה, החל מה *VM* ועד הזכרון שמיוצג בצורה וירטואלית. המעבד יצטרך לתרגם את כל השכבות הוירטואליות הללו. המעבד יעשה זאת באמצעות *shadow page table* - טבלה שמחזיקה את המיפוי האמיתי.

12.2 תרגול 12 - קונטיינרים ורשתות תקשורת (31.5):

12.2.1 קונטיינרים:

- **קונטיינר:** גרסה קלה יותר של וירטואליזציה, דורש פחות משאבים מאשר *VM*. הוירטואליזציה היא ברמת מערכת ההפעלה ולא ברמת החומרה. תכנית שרצה בקונטיינר רואה את מה שיש בתוך הקונטיינר, אך לא את מה שיש מחוצה לו.
  כל התכניות משתפות את אותה מערכת הפעלה, אך מבחינת התכניות כל אחת מהן משתמשת במערכת הפעלה אחרת.

- *Linux Containers*: קונטיינרים שרצים על מערכת לינוקס ומסמלצים מערכת הפעלה של לינוקס. בלינוקס מוגדרים שני אלמנטים שאנו צריכים להגדיר עם יצירת קונטיינר:
  namespace: מגדיר את רמת הבידוד של הקונטיינר, מה הקונטיינר יראה מבחוץ (*host*).
  Cgroups: מה הקונטיינר יכול לעשות.

12.2.2 $Namespace$:

- $Namespace$: מגדיר לאלו משאבים של ה $host$ התהליכים שרצים בתוך הקונטיינר יכולים לגשת, קיימת רשימה עם כמה $namespaces$ שניתן להגדיר:
  pID: ניתן להגדיר שיש להם גישה ל pID של התהליכים החיצוניים, מנגד ניתן להגדיר שה pID של התהליכים בקונטיינר מתחילים מ 0.
  $Hostname$: מגדיר מה השם של המחשב המארח.
  $user\ ID, File\ system$ ועוד..

- **יצירת תהליכים חדשים:** תהליכים יכולים ליצור תהליכים חדשים עם $namespaces$ חדשים, או להצטרף עם $namespaces$ קיימים.
  בלינוקס - כשאנו מפעילים את המערכת הוא יוצר $namespace$ מכל הסוגים שמשותפים ע"י כולם. לאחר מכן שאר התהליכים שנוצרים מגדירים $namespaces$ לפי הצורך.

- **כיצד תהליך יוצר תהליך חדש:** ע"י הפונקציה

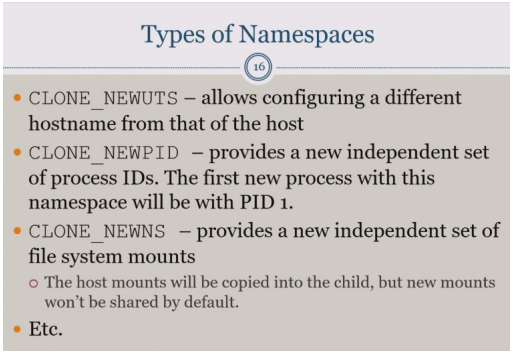
$$int\ clone(int(*fn)(void*), void*stack, int\ flags, void*arg);$$

  שמקבלת: פונקציה שהבן יריץ, מצביע למחסנית (שיצביע לסוף המחסנית), דגלים שייצגו את ה $namespaces$, ופרמטרים לפונקציה.
  ערך ההחזרה של הפונקציה הוא pid של הבן.

- **הדגל** $SIGCHLD$: סיגנל שהבן שולח לאב כשהוא מסיים את ריצתו. (ניתן לשלוח את הסיגנל הזה בתור דגל בפונקציה $clone$)

- **פונקציות להרצת תכנית אחרת:** כשנרצה תהליך שיריץ תכנית אחרת, נעשה זאת באמצעות הפונקציות $exec$ (כאשר מייצג אותיות המייצגות פונקציות שונות).
  לדוגמה $execvp$: מקבלת קובץ ורשימת ארגומנטים (שם הארגומנט הראשון צריך להיות שם הקובץ, והארגומנט האחרון צריך להיות "0") ומריצה את הקובץ.
  הערה: כל שורת קוד שתופיעה לאחר הפונקציה $execvp$ לא תרוץ, כי אנו עוברים לקובץ אחר.

- **סוגי** $namespaces$:



- **הערה:** כאשר תהליך הבן מוגדר לשנות משו מסויים (לדוגמה שם ה $host$), זה משתנה רק לבנים שלו, ולא במעלה העץ. מבחינת האב הדם נשאר כמו שהוא.

- **ביצירת תהליכים ושימוש ב** $namespace$ **-** $CLONE\ NEWPID$**:** כאשר תהליך יוצר תהליך חדש, מבחינת האב ה id של הבן יהיה $parent\ pid.1$, אך מחינת הבן הוא מהתחיל מ 1 (בגלל ה $namespace$ שהוגדר).

- **התיקיה** $proc$ **במערכת לינוקס:** מייצגת את כל התהליכים שנוצרו במערכת עד כה.

- **יצירת תהליך עם** $Filesystem$ **חדש:** כשנרצה ליצור תהליך שיגדיר $Filesystem$ חדש, נבצע את הפעולות הבאות:
1: נצטרך תחילה לשלוח את הדגל שמגדיר זאת בפונקציית היצירה - $CLONE\ NEWNS$.
2: בנוסף הבן יצטרך לרוץ על עותק של מערכת הקבצים. נוריד עותק של מערכת הקבצים ונאחסן אותו בתיקיה, אח"כ נגדיר לבן שזאת מערכת הקבצים שלו, נעשה זאת עם הפונקציה

```
int chroot(const char *path);
```

3: נבצע פקודת $mount$ בתיקיה $proc$ (ב $root$ החדש), בעזרת הפונקציה הבאה

```
mount ("proc", "/proc", "proc", 0, 0);
```

בסוף האב יקרא לפונקציה $umount$ שתבטל את הפונקציה $mount$.

12.2.3 $CGroups$:

- $CGroups$**:** מגדיר מה הקונטיינר יכול לעשות, למשל - להגביל את מספר התהליכים שהוא יכול לייצר, כמה זכרון וכח מעבד הוא יכול להשתמש.

- **יצירת** $CGroup$ **חדש:**
1: נגדיר $CGroup = "pids"$ שמגביל את מספר התהליכים שהתהליך הבן יכול ליצור.
2: תחת העותק שיצרנו בתהליך ה $namespace$, ניצור תיקיה חדשה שנקראת $pids$ באמצעות הפקודה $mkdir$ שנמצאת בתוך $/sys/fs/cgroup/pids$, כך נוצר $CGroup$ חדש.
3 מה קורה לאחר שיצרנו את התקיה $pids$**:** השם של התיקיה הוא שם שמור, ולאחר ייצירת התקיה התהליך מבין שחלות עליו הגבלות. הוא ימלא את התקיה בכל מיני קבצים, אחד מהם הוא הקובץ $cgroup.procs$ נכתוב לתוכו את pid שאליו ה $cgroup$ מקושר ($pids$ שה $CGroup$ משפיע עליהם).
4 לאחר מכן: ניגש לקובץ $pids.max$ ונעדכן אותו שיכיל את המספר המקסימלי של תהליכים שיכולים להיווצר.
5 לבסוף: נכתוב לתוך הקובץ "notify on release" את הספרה 1 . קובץ זה אחראי על שחרור המשאבים כשהקונטיינר מסיים את עבודתו.

12.2.4 רשתות תקשורת - $Networking$:

- **הגדרה - פרוטוקול:** כששני מחשבים רוצים לתקשר בניהם הם צריכים להסכים על פרוטוקול מוגדר היטב ולהשתמש בו.
פרוטוקול צריך להגדיר סינטקס, סמנטיקה וסינרוניזציה של תקשורת.

- **בעיות של הודעות ארוכות:** במחשבים שמחוברים ישירות בכבל, כשאנו שולחים הודעות ארוכות בין שני מחשבים יכולות להתרחש מספר בעיות הקשורות לפיסיקה, למשל חלק מהביטים ישתנו או יאבדו בדרך.

- **הפתרון - פקטים:** נעביר הודעו בפקטים קטנים, כך הסיכוי לכך שביטים ישתנו או ימחקו יקטן. לשם כך שני המחשבים יצטרכו להסכים על פרוטוקול שמגדיר את גודל החבילות ועוד.

- **בעיות של פקטים:** כשאנו משתמשים בפקטים יכול להיות שסדר שליחתם וקבלתם במחשב המקבל יהיה שונה, בנוסף פקט יכול לאבד לנו באמצע.
הפתרון - $End\ to\ end\ control$**:** נגדיר מערכת שתנהל את השליחה בין שני המחשבים, היא תדאג שהפקטים יסודרו לפי הסדר, ואם פקט נאבד היא תבקש מהמחשב השולח לשלוח אותו שוב.

- **שליחת הודעות בעזרת האינטרנט:** כנשלח הודעות באופן מקוון הודעות יעברו בכמה מרכזיות, החל מהראוטר וספק התקשורת ועד היעד.
כעת, בנוסף למספר הפקט נצטרך לשלוח גם את כתובת המקור וכתובת היעד.
בנוסף, מצטרף להודעה בלוק נוסף - ECC המשמש לתיקון שגיאות, למקרה וההודעה תשתבש בדרך.

- $Protocol\ Stack$**:** מחסנית שתייצג את כל הפרוטוקולים בהם השתמשנו בכדי לשלוח את ההודעה, וכל רמה במחסנית אחראית על רמה אחרת בתקשורת (פיצול ההודעה לפקטים, בדיקת שגיאות, הוספת כתובות וכו..). כמובן שנדרשת הסכמה בין שני הצדדים.

- **שכבות הפרוטוקולים:**

Internet protocol suite (TCP/IP) –
The protocol stack that is used by the Internet

35

Layer name	Description (Layer's goal)	Protocols
Application	process-to-process communications	HTTP/S, SSH, FTP, DNS
Transport	End-to-end communication services for applications	TCP, UDP
Network / Internet	Transport datagrams (packets) from the originating host across network boundaries, if necessary, to the destination host specified by a network address	IP
Link / Physical	Communications protocols that only operate on the link that a host is physically connected to.	802.11 WiFi, Ethernet

- **הפרוקטוקול** $Network$**:** אחראי על להעביר את ההודעה ליעד, אך הוא שולח אותה בצורה לא אמינה - בלי טיפול בשגיאות ובדיקה שכל החבילות הגיעו. אלא רק העברה ליעד בלבד.

- **הפרוטוקול** $Transport$**:** פרוטוקול שיושב מעל $Network$ פרוטוקול. נועד לפתור את הבעיות ש $Network$ לא מטפל בהן. יש שני סוגים של פרוטוקולי $Transport$ - TCP, UDP.

- $UDP - user\ datagram\ protocol$**:** בעיקר להעברת מידע ללא טיפול בשגיאות. הוא יושב מעל ה IP, ומוסיף לכל פקט:
1: את ה $port$ - מייצג את שם התכנה במחשב היעד שצריכה לקבל את המידע.
2: אורך ובדיקה, $length + chechsum$. בדיקה מינימלית של טיפול בשגיאות אך לא מעבר (כפילויות, אובדן חבילות).

- $TCP - Transmission\ control$: מטפל בכל השגיאות ש UDP לא מטפל בהן.
  1: דואג לתקשורת אמינה שמתועדת - הצדדים מתואמים.
  2: בקרת זרימה - ווידוא שהפקטים מגיעים בסדר הנכון.
  3: בקרת עומס - במקרה של הצפה הוא יתריע למחשב השולח שיאט את קצב השליחה.
  4: מטפל בפקטים אבודים. ומסדר אותן לפי הסדר.

- **הבדלים בין** TCP, UDP:

Transport Layer Summary

40

Property	UDP	TCP
Reliable	no	yes
Connection type	Connectionless	Connection oriented
Flow control	No	Yes
Latency	Low	High
Applications	VOIP, Most games	HTTP, HTTPs, FTP, SMTP, Telnet, SSH

TCP ישמש להודעות קצרות, ו UDP ישמש להעברת קבצים גדולים - יוטיוב זום ועוד...

13 שבוע 13:

13.1 הרצאה 13 - $I\backslash O$ (12.6):

- **במה נתמקד:** בהרצאה הנוכחית נתמקד בקשר בין הקבצים למערכת ההפעלה מהצד של החומרה. כיצד מערכת ההפעלה מנהלת את התקני הקלט פלט החומרתיים.

- **החיבור הפיזי:** רכיבי הקלט פלט מחוברים למערכת בעזרת bus שמצד אחד שלו יושב המעבד, והרכיבים מחוברים ל bus בעזרת $controlers$.

- **דרייברים:** רכיבי תכנה שהם החלק שמחבר בין ליבת מערכת ההפעלה לבין החומרה.

- **נרצה שיתקיימו מספר דברים:**
  אבסטרקציה: נרצה שמערכת ההפעלה תעבוד עבור כל ההתקנים, ושהיא לא תצטרך לטפל בכל אחד מהם אחרת. טיפול בשגיאות והפרטים התכנים ינוהלו ע"י הדרייברים.
  יעילות: נרצה ניהול יעיל וחפיפה של המעבד והתקני הקלט פלט.
  שיתוף: הגנה כאשר יש שיתוף משאבים, ותזמון השימוש בין מספר התקנים.

- **ההבדל בין** $controler$ **לדרייבר:** לכל דרייבר יש את ה $controler$ שלו,והוא משמש כמיני מעבד שמפעיל את ההתקן. מבחינת מערכת ההפעלה ה $controler$ הוא חלק מהחומרה של המערכת, והדרייבר הוא החלק שמתקשר בין ה $controler$ למערכת ההפעלה.

- **דרייברים:** כל $controler$ צריך דרייבר שיידע לחשב בינו לבין מערכת ההפעלה, הם רצים ב $kernel\ mood$ על המעבד. הם נכתבים ע"י ייצרני ההתקן.

- **הקשר בין הדרייבר ל** $controler$**:** ה $controler$ מקבל פקודות מהדרייבר, הממשק בניהם הוא דרך הרגיסטרים שנמצאים ב $controler$. הדרייבר יודע לקחת את המידע מהרגיסטרים. מבחינת המעבד הרגיסטרים האלה הם רגיסטרים של $controlers$ והוא יודע כיצד לתקשר איתם.

- $USB - universal\ serial\ bus$**:** כיום רוב הרכיבים מתחברים עם USB, שמעביר חשמל ונתונים. הוא למעשה מחליף את כל המחברי השונים שהיו באחד קטן וסטנדרטי. קיים פרוטוקול שמערכת ההפעלה צריכה להבין איזה התקן מחובר ל USB, ובאיזה $mood$ הוא יעבוד. **יש 4 סוגי תקשורת:**
$interrupt$**:** עבור התקנים שיוזמים את התקשורת ומעבירים מעט נתונים ולא באופן רציף. לדוגמה עכבר ומקלדת.
$bluk$**:** עבור התקנים שמעבירים כמויות גדולות של מידע, $64\ byte$ - מדפסות. עם תיקון טעויות כך שמובטח לנו שבמידע יעבור נכון.
$isochronos$**:** מתואם בזמן, משמע התקני אודיו. מעביר מידע בקצב קבוע, אין תיקון טעויות.
$control$**:** תקשורת בין המעבד, מערכת ההפעלה וההתקנים כדי לתאם דברים.

- **האופרציה של ה** USB**:** בחיבור, מערכת ההפעלה מזהה איזה התקן התחבר. אין צורך להפעיל את המערכת מחדש כדי לחבר אותם.
רכיבים שעובדים במוד של $interrupt, isochronos$ מגדירים מה הכמות המקסימלית אותה הם רוצים להעביר. מערכת ההפעלה תחבר אותם רק אם הם תופסים פחות מ 90% מה bus.
המידע עובר ביחידות של $1500\ bytes$. אחוז מסויים מוקצה עבור התקנים שעובדין ב $interrupt, isochronos$. ה $bytes$ הנותרים ישמשו עבור התקנים רוצים לעשות $bluk, control$.

- **מי שולט בהעברת המידע:** יש כמה שיטות -
1 המעבד ומערכת ההפעלה אחראים: מערכת ההפעלה כותבת לרגיסטרים את הפרמטרים ואח"כ את הפקודה. לאחר מכן ההתקן יקח את הנתונים לתוך באפר פנימי שלו, וידליק את הביט של $busy$. כשהוא יסיים הוא יכבה את הביט ומערכת ההפעלה תיכנס לפעולה ותקח את הנתונים. במשך הזמן הזה מערכת ההפעלה תבדוק האם ההתקן סיים או לא.
2 $interupt$**:** השיטה הזו שונה, בכך שמערכת ההפעלה לא צריכה לוודא שההתקן סיים, אלא ההתקן שולח $interupt$ למעבד כשהוא מסיים. לאחר מכן מערכת ההפעלה נכנסת לפעולה ומעתיקה את הקבצים.
3 $Direct\ Memory\ Acces(DMA)$**:** השיטה השלישית שונה בכך שההתקן מעתיק את הנתונים בעצמו לזכרון המרכזי ע"י DMA, ורק אח"כ שולח $interupt$ למעבד. כך המעבד לא עובד על העתקת הנתונים.

- DMA**:** הארכיטקטורה נראית כך שהמעבד לא יושב ישירות עלה bus, אלא הוא עובר דרך ה DMA. ה DMA יושב על אותו bus עם המעבד והזכרון המרכזי. כך ה DMA מעביר את המידע מההתקן לזכרון הראשי מבלי לערב את המעבד.

- **חלוקת הדרייברים לרמות:** נרצה להקל על המבנה של מערכת ההפעלה ולאחד דרייברים. מערכת ההפעלה מפוצלת לשלשה סוגים שונים של התקנים, ולכל אחד מהם יש ממשקים שונים:
$Block$**:** התקנים שעובדים בבלוקים וצריך לעשות להם תיקון שגיאות - דיסק $CD - ROM$.
$stream$**:** התקנים שעובדים ביחידות מידע קטנות - מחברת ועכבר.
$network$**:** סוג שלישי של התקנים - תקשורת.

- $Buffering$: תחנת ביניים למידע שיכולה לעזור לנו לאחסן את המידע עד שההתקן שאנו מחכים שיתפנה, יתפנה.
- **טיפול בשגיאות:** יכולות להתרחש תקלות חומרה או תוכנה. תחילה נצטרך לזהות שמתרחשת תקלה ולתקן בהתאם. כדי לעלות על כך שהתרחשה תקלה, אנו שולחים את הדאטה עם מידע נוסף, כך כשהוא מתקבל אנו בודקים את הפונקציה שמגדירה את תקינות הקלט.

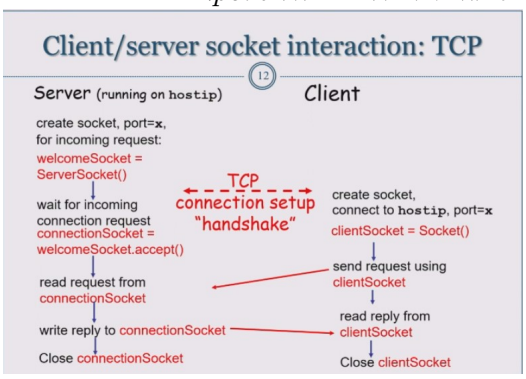
13.2 תרגול 13 - $Sockets$ (7.6):

- **הרעיון:** נרצה לאפשר לשתי אפליקציות שיושבות בשני מחשבין מרוחקים לתקשר אחת עם השניה. הן יעשו זאת בעזרת שמדמה שער דרכו הן יוכלו לתקשר - להעברי ולקבל מידע.
- פרדיגמת $Client\backslash server$: צורת תקשורת, בו חשב אחד הוא השרת שמבצע פקודות של הלקוחות.
- $TCP\ service$: תקשורת בשיטת TCP

1: כשה $server$ יתחיל לרוץ הוא ייצור $socket$ דרכו הוא יקבל בקשות של $clients$.

2: ה $client$ יפתח $socket$ ויתחבר אל ה $server$, בחיבור הוא יציין מה כתובת ה IP של ה $server$ ואת מספר ה $port$ (מייצג את מספר התהליך אליו הוא רוצה להתחבר).

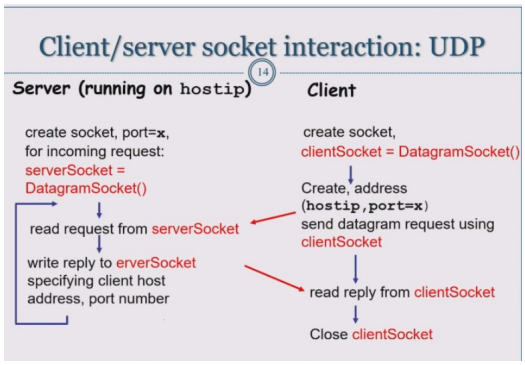
3: השרת יקבל את החיבור, ויקצה לו $socket$ חדש ייחודי רק לו, דרכו הוא יתקשר עם ה $client$ הספציפי. השרת ישמור את ה IP וה $port$.



- $Streams$: סדרת תווים שזורמים לתוך תהליך או החוצה ממנו (מישע שנכנס ויוצא מתהליך).

$input\ stream$ - מידע שמגיע לשרת מה $client$. $output\ stream$ - מישע שיוצא מהשרת ל $client\backslash$.

- $UDP\ service$: תקשורת בשיטת UDP. בדומה לשיטת TCP, אך כאן אין את התיאום בין השרת ל $client$. השרת מציין להיכן הוא רוצה לשלוח את המידע (כתובת IP) ושולח אותו, ללא שום תיאום או בקרה האם המידע הגיע. בנוסף אין $socket$ פרטי לכל $client$.



- **הסטראקט** $sockaddrin$: נייבא את הסטראקט $sockaddrin$ שמייצג כתובת של $socket$ לתקשורת על גבי האינטרנט. יש לו כמה שדות: **השדה הראשון** מייצג את סוג הכתובת, כדי לייצג כתובת IP נשתמש ב $AFINET$. **השדה השני** מייצג את ה $port$. **השדה השלישי** מייצג את כתובת ה IP.

- **שיטות לשמירת מספרים** $big\backslash litle\ Endian$: קיימות שתי שיטות לשמירת מספרים - מתחילת המספר או מסוף המספר. האינטרנט $(network\ byte\ order)$ והמארח משתמשים בשתי שיטות שונות ויש פונקציות להמרה בין שתי השיטות.

- **הפונקציה** $inet\ aton()$: פונקציה שמקבלת סטרינג של IP וממירה אותו לכתובת IP. בנוסף היא שומרת אותו ב $network\ byte\ order$.
  הפונקציה $inet\ ntoa()$: מתרגמת בכיוון ההפוך.

- **הפונקציה** $getpeername$: כך השרת מקבל את המידע שה $client$ שלח אליו.

$$int\ getpeername(int\ sockfd, struct\ sockaddr^{*}addr, int * addrlen);$$

  מקבלת $socket$, סטראקט $sockaddrin$, ואת גודל הסטראקט.

- $DNS - domain\ name\ service$: פרוטוקול שמתרגם לנו שם של אתר לכתובת IP.

- **הפונקציה** $gethostname()$: מחזירה את השם של המחשב שהתכנית שלנו רצה עליו.

- **הפונקציה** $gethostbyname()$: מחזירה את כתובת ה IP של המחשב לפי שם. ערך ההחזרה שלה היא $struct$ שנקרא $hostent$ שמייצג את ה IP של ה $host$.

- **כתובת** IP **של מחשב מקומי:** היא 127001 ונקראת $local\ host$.

13.2.1 שלבים לחיבור $server - client$:

צד שרת:

- **1 - ייצירת** $socket$ **של שרת:** נקרא לפונקציה $socket()$ -

$$\text{int socket(int domain, int type, int protocol);}$$

$domain$ - **סוג הכתובת:** $AFINET$ כדי לייצג תקשורת ע"ג האינטרנט.
$type$ - **סוג הפרוטוקול איתו נרצה לעבוד:** $SOCK\ STREAM$ - מייצג תקשורת של TCP.
$protocol$: נשתמש בדיפולטיבי ע"י שליחת 0 .

- **2 - חיבור ה** $socket$ **לכתובת:** נקרא לפונקציה $bind()$ -

$$int\ bind(int\ sockfd, const\ struct\ sockaddr * addr, socklent\ addrlen);$$

הפרמטר הראשון: ה $socket$ שיצרנו ע"י הפונקציה $socket()$. **הפרמטר השני:** הסטראקט $sockaddrin$. **הפרמטר השלישי:** גודל הסטראקט.

- **3 - פתיחת ה** $socket$ **לקבלת** $cients$: נשתמש בפונקציה $listen()$, שמקבל את ה $socket$, ומספר שמייצג את מספר ה $clients$ המקסימלי שימתינו במידה והשרת תפוס.

- **4 - השרת יחכה ל** $clients$: ע"י הפונקציה $accept()$, פונקציה שמכניסה את השרת ללולאה וגורמת לו לחכות ל $clients$.

$$int\ accept(int\ sockfd, struct\ sockaddr * cliaddr, socklent^{*}cliaddrlen)$$

מקבלת את ה $socket$. סטראקט שלתוכו ייכנס המידע של ה $client$ (פרמטר פלט). ואת אורך הסטראקט.
הפונקציה מחזירה את ה $socket$ של ה $client$.

צד $client$:

- **1 - ייצירת** $socket$: ה $client$ ישתמש בפונקציה $socket()$ כמו השרת וייצור $socket$.

- **2 - חיבור לשרת:** יקרא לפונקציה $connect()$ כדי להתחבר.

$$int\ connect(int\ sockfd, const\ struct\ sockaddr^{*}servaddr, socklent\ addrlen);$$

מקבלת את ה $socket$. הסטראקט שמכיל את המידע לאן להתחבר - $ID, port$. ואת אורך הסטראקט .

- **3 - שליחת מידע:** מכיוון ש $socket$ הוא $file\ descriptor$ לכן נקרא ונשלח מידע עם הפונקציות $read()$ - כדי לקרוא מידע מהצד השני. $write()$ - כדי לשלוח מידע לצד השני.
הערה: מכיוון שהמידע מגיע בפאקטים נצטרך להריץ את הפונקציה $read()$ בלולאה על מספר הפאקטים.

חיבור של כמה $clients$:

- בעולם האמיתי לכל שרת יש כמה לקוחות, לכן הם עובדים עם ת'רדים , ת'רד לכל $client$.

- **הפונקציה** $select()$: שימושית לא רק לתקשורת .

$$int\ select(int\ nfds, fdset * read-fds, fdset^{*}write-fds, fdset * except-fds, struct\ timeval * timeout)$$

מקבלת קבוצה של fd שמהם נקרא מידע, עוד קבוצה שאליהם נכתוב מידע.

14 שבוע 14:

14.1 הרצאה 14 - אבטחה (19.6):

- **הגדרה -מערכת אמינה:** מערכת שיודעת לטפל בשגיאות. במידה ואנו נכנסים למצב של דדלוק או שחסר לנו קובץ מהדאטה היא תדע להשלים אותו ולטפל בתקלה.

- **הגדרה - מערכת בטוחה:** מערכת שתעזור לנו להגן על עצמנו מתוקף. היא תעשה זאת ע"י הפרדה בין ה $user\ space$ ל $kernel$.

- **סוגי מתקפות:** הסוג הראשון של המתקפה היא מתקפה על משתמש ספציפי. מתקפה נוספת היא על כל המערכת, מתקפה זו היא מסוכנת יותר ויכולה לשבש יותר דברים.

- **דליפות ופריצות:** במערכת יכולים להתרחש שני סוגי מפגעים. **דליפות** של מידע - כתוצאה מתכנון לקוי של המערכת משתמשים נחשפים למידע שהן לא צריכים להיחשף אליו. **פריצה** למערכת - משתמש זדוני פורצל מערכת בכוונה תחילה.

14.1.1 אימות - $Authentication$:

- **משתמשים במערכת:** במערכת יש מספר חשבונות כך שכל משתמש מיוצג ע"י $user\ ID$, לכל משתמש יש הרשאות גישה שונות לקבצים שונים. בנוסף יש במערכת משתמש שמוגדר כמנהל מערכת שלו יש הרשאות גישה נרחבות, פריצה לחשבון זה חמורה יותר.
  נרצה ללמוד כיצד המערכת יודעת מי המשתמש שנכנס אל המערכת, וכיצד היא מבטיחה שרק המשתמש הנכון ישתמש בחשבון שלו.

- **כניסה למשתמש:** כשמשתמש רוצה להיכנס לחשבון שלו הוא צריך להכניס סיסמא, המערכת שומרת את הסיסמאות ומשווה את הסיסמא שהמשתמש הזין, לסמאות השמורות. הסיסמאות יישמרו באופן מוצפן כדי שלא ייחשפו במקרה של פריצה למערכת.

- **מתקפת** $brute\ force$**:** מתקפה בה התוקף מנסה מספר רב של סיסמאות עד שהוא מוצא את הסיסמא הנכונה. הצלחת המתקפה תלויה בגיוון הסיסמא (אותיות גדולות, קטנות וסימנים) ואורכה.

- **מתקפות נוספות:** האקרים ינסו לפרוץ את החשבון ע"י הכנסת סמאות נפוצות עם מילים נפוצות, או סמאות של אנשים אחרים שכבר נחשפו בעבר. תבניות מסויימות של סמאות שנפוצות. כך הם מצמצמים את מרחב החיפוש.

- **כיצד משתמש יכול לבחור סיסמא טובה:** ככל שהסיסמא יותר ארוכה ומגוונת יותר קשה לפרוץ אליה. לכן שיטה טובה היא לבחור מספר מילים אקראיות ולבסס עליהן את הסיסמא.
  פתרון נוסף הוא מנהל הסיסמאות שמג'נרט אותיות ותווים באופן אקראי.

- **כיצד מערכת ההפעלה יכולה להתגונן מפריצות:**

1: מערכת ההפעלה צריכה להצפין את הסמאות שהיא שומרת, וההצפנה צריכה להיות חזקה דייה כדי שלא יפרצו

אותה.

2: בנוסף המערכת תוסיף לסיסמא סטרינג אקראי ותצפין אותו עם הסיסמא. כך שהתוקף לא יידע איזה חלק הוא הסיסמא המקורית ואיזה חלק הוא התוספת של מערכת ההפעלה.

3: המערכת תחביא את הקובץ ותשמור אותו ללא שם, ורק הפונקציה של ה $login$ תדע היכן הוא נמצא ומה מספר ה $inode$ שלו.

4: המערכת יכולה להגביל את מספר הנסיונות.

5: שימוש באמצעי הגנה ביומטרים - זיהוי פנים, טביעת אצבע.

14.1.2 הרשאות $Permissions$:

- **הרשאות:** ניתן לתאר אותן בתור מטריצה - טבלה, שכלשורה מייצגת משתמש, וכל עמודה מייצגת אובייקט שיש למשתמש גישה אליו.
  קיימת האפשרות להריץ תכנית מסויימת תחת הרשאות של משתמש אחר.

- **הגנה על מובייל:** אנו צריכים להפריד בין מערכות שונות. מחשב עם $multy\ users$ יצטרך הגנה אחרת מאשר מובייל שיש לו רק משתמש אחד, ויש לו את כל הרשאות המערכת.

- **מתקפות על מובייל:** המערכת תצטרך להגן עליו מפני אפליקציות זדוניות. יש כמה סוגי מתקפות - וירוס ונוזקה.
  וירוס - החדרת וירוס שמטרתו לגרום לנזק במערכת.
  נוזקה - השתלת דלת אחורית באפליקציה שתגרום למערכת להיות חשופה לתוקף.

- **הגנה על מובייל:**
  ניתן להתקין אנטי וירוס שיגן מפני וירוסים.
  אפשרות נוספת היא לעקוב אחר התנהגות המערכת ולראות אם מתרחשת התנהגות חשודה.
  התקנת $fire\ wall$ שמונע גישה של המערכת לאינטרנט, ומאפשר רק לתקשורות מסויימות לעבור.

- **מתקפת באפר אוברפלואו:** מתקפה בה התוקף דורס את ערך החזרה של הפונקציה ע"י הזרקת קוד עויין למחסנית. כך התכנית תחזור לקוד המוזרק ותריץ אותו. התוקף יכול להגדיל את סיכויי המתקפה ע"י שורות $noop$.

- **כיצד מערכת ההפעלה תתגונן ממתקפת באפר אוברפלואו:**
  1 מנגנון הקנרית: המערכת תוסיף סטרינג רנדומלי במחסנית הנקרא ערך קנרית, ולפני שהתכנית תרוץ היא תבדוק אם ערך הכנרית נשמר, אם הוא נדרס היא לא תריץ את הקוד.
  2 הזזת המחסנית: אם כתובת תחילת המחסנית תשונה ולא יהיה קבוע, התוקף לא יוכל לדעת איזו כתובת לדרוס.
  3 ערבול הכתובת: המערכת יכולה להצפין את הכתובת, כך שהתוקף לא יוכל להכניס כתובת ולהגיע אליה ללא הצפנת הכתובת.
  4 ביט $exeutable$: נגדיר בקטע הטקסט ביט שמגדיר שהקוד הוא $exeutable$ וביט נוסף שיסמן שהוא לא $writeble$. ובמחסנית נגדיר ההפך. כך שקוד לא יוכל להיכנס למחסנית כי היא לא מוגדרת $exeutable$.

- **מתקפת** $libc$: מתקפה שמבוססת על הספריה $libc$ שמיובאת כמעט לכל קובץ C, ובפרט הפונקציה $system$. כך ניתן לדרוס את ערך ההחזרה של הפונקציה ולשנות אותו לכתובת של הפונקציה $system$ שהיא פונקציה שמריצה כל קטע

קוד שנבחר. כך התוקף יוכל להגדיר לה להריץ קוד זדוני.

- $Sandbox$: הגנות בפני אפליקציות שירדו מהרשת ואינן בטוחות בוואות. לכן המערכת תריץ אותן בקופסה שתמנע מהן להריץ פקודות מערכתף ותגביל את הגישה שלהן.
ניתן לעשות זאת עם וירטואליזציה - VM. או הרצת האפליקציה בתוך תהליך אך עם $user\ ID$ שאין לו הרשאות מערכת.

14.2