

Processor (CPU)

- הרכיב הכי חשוב במחשב
- המעבד מריץ את התוכניות שרצות על המחשב

המודול

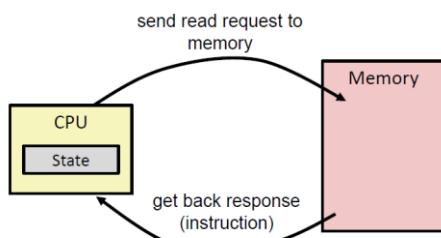
- **CPU** = מכונת מצבים שמחוברת לדיסקון
- **זיכרון** = מערך של בתים (האינדקסים של המערך = כתובות)
- המטריה של המעבד: להריץ תוכנית
- **תוכנית** = אוסף של פקודות (instructions)
- כל פקודה מגדרה איך לשנות את המצב של המעבד באשר הפקודה מורצת
- ניתן להקביל את המודול **למבנה טירוגן**. הפענקציה של מכונת טירוגן מקבלת: מצב הנוכחי, קלט מהсрט ומחזירה כפלט את המצב הבא, עדכון לסרט (במקום בו הראש המצביע), כיון אליו הראש צריך לבצע עית.
- **ה-CPU עובד בצורה דומה:**
מקבל פקודה -> ניגש למקום המתאים בזיכרון -> בהתאם למידע שנמצא בזיכרון ולמצב הנוכחי -> משנה את המצב הנוכחי של המחשב וمعدכן את הזיכרון.

ה מצב של המעבד

- מוגדר על ידי רצף של ביטים.
- לצרכי נוחות (ויעילות) הביטים מחולקים לרגיסטרים.
- הרגיסטרים מחולקים ל-2 קבוצות:
 1. **GPR** (General Purpose Registers) –
a. הפקודות של המעבד **יכולות לשנות את הערך שלהם**.
 - b. אבל התוכן שלהם **לא משפיע על פעולה המעבד עצמו**.
 - למשל: ..., RAX, RBX (ב-x86)
 2. **רגיסטרים** שניצנים לשנות את התוכן שלהם **והם גם משפיעים על פעולות המעבד**.
 - למשל: הרגיסטר ששמור את כתובת הפקודה הבאה: PC. נקרא IP או RIP (ב-x86).

ISA (Instruction Set Architecture)

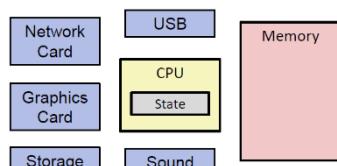
- מגדיר את הפקודות שהחומרה יודעת למש.
- הפקודות מחולקות ל-2:
 1. פקודות שימושות את המצב של המכונה. למשל: חיבור 2 רגיסטרים או קפיצה לפקודה אחרת.
 2. פקודות שמתקשירות עם ה"עולם שמחוץ למוכנה": W1, SW.



Program Execution

1. קרא את הפקודה הבאה מהזיכרון
2. קדם את PC-לפקודה הבאה
3. בצע את הפקודה שקרהת ב-1 (שינוי המצב של המעבד)
4. חזר ל-1

(assembly) = תוכנית בשפת מוכנה (assembly) = Program



התקנים = Devices

- תפקידם: לחבר את המעבד לעולם החיצוני (הפיזי)
- לمثال: חיבור לדיסקון, USB, שמע וכו'

זיכרון = DRAM (Random Access Memory). זהו הזיכרון הקרוב למעבד.

אחסון. למשל: Hard Disk, SSD card וכו'.

הסיבה להפרדה בין השניים:

אחסון	זיכרון
לא נגיש ישירות למעבד	נכיש ישירות למעבד
נכישות ברמת הבlok	נכישות ברמת הביט'
נדיף (volatile) – נמחק כאשר המחשב נכבה	נדיף (non volatile) – נמחק כאשר המחשב נכבה
איטוי פי 10 - 100	מהיר
גדול פי 10 - 1000	קטן

אילו בעיות מערכת הפעלה פוטרת?

1. איך לגרום לתוכנית לרוץ?

התוכנית לא יכולה להריץ את עצמה (bootstrapping).

עם הפעלת המעבד הוא מרים תוכנית שמנמצאת במקום קבוע מראש – **booting**.

הчисרון – המעבד יכול להריץ רק תוכנית אחת – התוכנית שאוותה מריצ' בעת ה-booting.

בשביל להריץ במאה תוכניות סימולטנייה המעבד מבצע time-sharing: מחלקים את זמן המעבד בין כל התוכניות שאמורות להיות מופרחות. חלונות הזמן שבוחן כל תוכנית שעבודת יכולים להיות מאוד קטנים כך נראה אילו התוכניות עוסקות במקביל על אף שבפועל הריצה היא טורית.

בצד למש time-sharing על ידי המעבד?

2. איך לבדוק תוכנית כך ששימוש בתוכנית אחת לא יפגע באחרות (למשל: על ידי דרישת זיכרון, תוכניות שלא עוצרות)

לא ניתן להסתמך על התוכניות שהן ייצרו זאת בעצמן, אלא צריך מישחו שינהל את זה מלמעלה.

יתרונן נוספת: התוכניות עצמן לא יצטרכו לדאוג לחלוקת המשאים ביניהן אלא רק לתוחום האחוריות שלהם מבלתי לדעת איך הדברים מתבצעים מאחורי הקלעים.

3. איך להשתמש בכל התקנים שמחוברים למחשב?

תפקידו של מערכת הפעלה

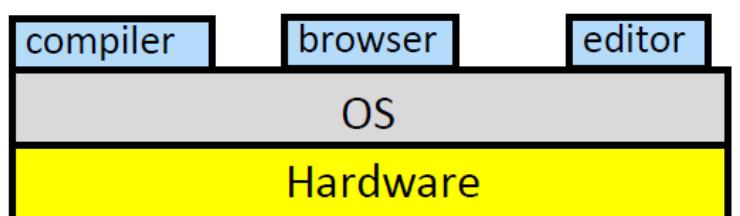
1. ניהול משאבי החומרה – לאפשר ולנהל את השיתוף של משאבי החומרה בין שאר התוכניות שרצות על המעבד.

2. הגנה על התוכניות שרצות מפני תוכניות אחרות ו**בידוד** התוכניות האחת מהשנייה.

- בידוד שגיאות ותקלות שישפיעו רק על התוכנית שגרמה להן.
- הגנה על המידע של תוכניות.
- וידוא שימושיים משותפים בצדורה הוגנת.

3. אבסטרקציה לתוכנים

אבסטרקציה = הגדרה של מודל שמתאר את הפונקציונליות שאנו רוצים מבל' לציין איך היא תמומש.
הגדרת API-ים ל-OS.



system

- תוכנית שמספקת אבסטרקציות (ממשקים) לתוכניות אחרות עבור שימוש במשאב כלשהו.
 - סוג של מתווך.
 - בוללת:
 1. עיצוב (design) ומימוש אבסטרקציות.
 2. מנגנונים ו מדיניות (mechanisms and polices) לניהול המשאים המשותפים.
- (מע' הפעלה היא דוג' **קלאסית ל-system**)
- דוגמאות נוספת: database, browser, JVM
 - לרוב ימומשו ב-C/C++.

Monolithic OS

- מע' הפעלה מונוליטית היא מע' הפעלה שמורכבת מתוכנית אחת ב-C.
 - דוג' למע' הפעלה מונוליטיות: Microsoft Windows, Linux, UNIX, Android.
 - הביעה של מע' הפעלה מונוליטית היא שאם יש באג בקוד הוא יכול להפיל את מע' הפעלה, לאחר וכל מערכת הפעלה מתבססת על תוכנית יחידה.
 - כדי לפתור את זה, המציאו מע' הפעלה לא מונוליטיות שמורכבת מכמה תוכניות שמתקשנות אחת עם השניה.
 - ○ OS max הן מערכות הפעלה היברידיות שימושה עקרונית מ-2 העולמות (מונוליטיות ולא מונוליטיות).
-
- The diagram illustrates a monolithic operating system architecture. It shows a single vertical stack of components. At the bottom is the **Hardware** layer, represented by a yellow rectangle. Above it is the **OS = Kernel** layer, represented by a grey rectangle. At the top of the stack are three application layers: **compiler**, **browser**, and **editor**, each represented by a blue rectangle. All these components are contained within a single monolithic structure.
- Kernel = OS**
- **Kernel** - התוכנית שהיא מערכת הפעלה.
 - **user-level** – שאר התוכניות.
 - מערכת הפעלה היא רק ה-**Kernel** ללא כל שאר הספריות, תוכנות והשירותים הנלוויים.

Non monolithic OS

- מערכת הפעלה שמורכבת מכמה תוכניות.
- מרכיבים ב-**privileged mode** תוכנית הרבה יותר קטנה שבכל מה שהוא עשו זה להוות אחריות על:
 - **scheduling** של תהליכיים
 - **תקשורת בין תהליכיים – IPC** = Inter processes communication
 - את כל שאר התפקידים של מערכת הפעלה ממשים בעדרת תהליכיים.
- כבה מקטינים את כמות הקוד שרצה ב-**privileged mode**, אך מצד שני, מאבדים ביצועים, כי כל קראיה של **call system** יהיה ברוך ב-**context switch** כדי לעבור לתהילך שאחראי על הטיפול ב-**call system** (זה מוקד המחקר למערכות הפעלה לא מונוליטיות, לאחר וזהו ה-**bottle neck**).

Processes

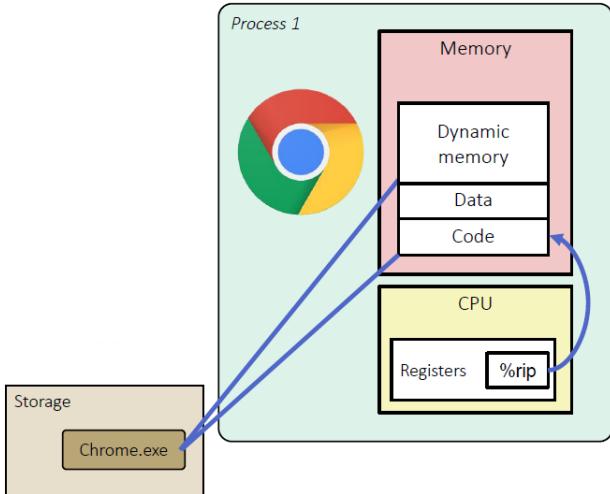
- system מספקת אבסטרקציות לתוכניות אחרות לשימוש במשאב כלשהו. למשל: חומרה (CPU, זיכרון, devices).
- המשאב היכי חשוב שמערכת הפעלה מספקת עבורו אבסטרקציה הוא **process = תהליך**.
- **האבסטרקציה של התהיליך מוחלקת ל-2:**

 1. **אבסטרקציה של המעבד (CPU):** – מתארת פונקציונליות של תוכנית שרצה בלבד על המעבד. כל מה שהמעבד עשו זה להריץ את הפקודות של אותה תוכנית ללא תלות בתוכניות אחרות.
 2. **אבסטרקציה של הזיכרון:** – מתארת פונקציונליות של זיכרון פרטי. לא מודעת שיתכן שיש עוד תוכניות שמשתמשות אליו בזכרון.

system call interface – שירותים המספקים ע"י מערכת הפעלה:

 - I/O
 - (Inter Process Communication = IPC) communication between processes
 - .process management
 - **running program = process** = תוכנית שרצה
 - חשוב להבדיל בין תוכנית שרצה (process = תהליך) לבין התוכנית עצמה (program executable) = קובץ עם פקודות של התוכנית).

- תהליך הוא כבר חישוב עצמו שמתבצע. זה בא לידי ביטוי ב:
 1. יש לו כבר state (ערכי הרגיסטרים באשר החישוב מתבצע).
 2. יש לו זיכרון דינמי.



- ב כדי להפוך תוכנית לתהליך שרצן:
 1. מע' הפעלה צריכה להעתיק את data של התוכנית מהאחסון (storage) אל הדיזרין (memory).
 2. לגרום למעבד להריץ את התוכנית:
 - ה-PC של המעבד מציביע לפקודה הבאה לביצוע.
 - הרגיסטרים שלו מटעדכנים לפי פקודות התוכנית.
 - כארה התהליך צריך זיכרון דינמי הוא ישמש בכתובות חדשות בזיכרון.

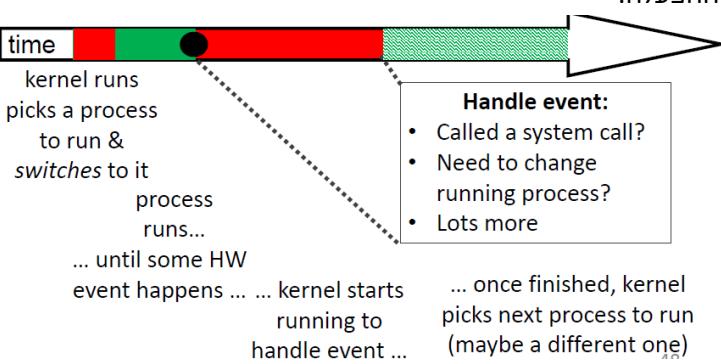
Mechanisms implementing process abstractions

private address space	logical control flow
<ul style="list-style-type: none"> • Achieved using virtual memory 	<ul style="list-style-type: none"> • Achieved by time sharing the CPU • The Mechanism that is responsible for that – context switching

CPU time sharing

- מערכת הפעלה היא תוכנית המונעת מאירועים (event-driven) שמטורחת להריץ תהליכים.
- ברוב הזמן מערכת הפעלה לא רצאה כי היא נותנת לתהליכים אחרים להשתמש במעבד.
- איך זה עובד?

1. מע' הפעלה מנעה את המערכת:



- בזמן-boot המעבד מתחילה להתחיל להריץ את מערכת הפעלה.
- אחרי עדכונים וקונפיגרציות היא בוחרת את התהליך הראשון שצריך להריץ.
- משתמשת במנגנון שבו היא עוצרת את עצמה וגורמת למעבד להניע את התהליך שנבחר.

2. התהליך רץ (המעבד לרשותו) עד שקרה אירוע (event)

- חומרתי (למשל: אם התהליך מבצע קריאת מערכות – System call, או אם הזמן שימושה הפעלה הגדרה בעבר התהליך נגמר)
- המעבד יודע להחזיר את מערכת הפעלה למצב שבו היא רצתה מחדש.
- היא שוב בוחרת איזה תהליך יתחל לוחץ.
- משתמשת במנגנון שבו היא עוצרת את עצמה וגורמת למעבד להניע את התהליך שנבחר.

4. וחזר חלילה...

- מערכת הפעלה רצתה על המעבד רק אם התהליך הנוכחי גורם ל-/חוטף exception.
- הרעיון של זיכרון וירטואלי מאפשר למערכת הפעלה להריץ הרבה תהליכים במקביל, אחריו ולכל תהליך יש את מרחב הקטובות שלו (רחוב תוכניות לא צריכות את כל הזיכרון הפיזי הקיים כך שבפועל התהליכים חולקים את הזיכרון הפיזי על אף שנדמה שלא, לפיה מרחב הזיכרון הוירטואלי (שיכולים לחפות בין תהליכים שונים).

- מעבר בין ההליכים כולל 2 חלקים:
 1. **שינוי מצב ה-CPU – State Switch**
 - א. שמיון התוכן הנוכחי של כל הרגיסטרים של התוכנית הנוכחי (שבධוק מפסיקה להרץ)
 - ב. לטען לתוך הרגיסטרים של המעבד את הערכים השמורים שלהם
 2. **שינוי מרחב הכתובות**

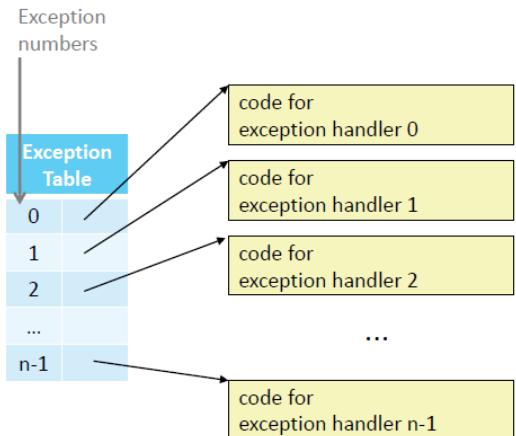
מאחר ומצב המעבד מרכיב גם מרגיסטר שקובע את מרחב הכתובות, מספיק רק לשנות את מצב המעבד (CPU).

בכדי למנוע מצב שבו תהליך מסוים משתלט על המעבד וטוען בעצמו תהליכים אחרים שיוציאו אחריו במקומות שונים שמערכת הפעלה עשויה זאת, יש שני מצבים למעבד:

unprivileged/user mode	privileged/kernel mode/"ring 0"
באן נמצאים כל התהליכים שאינם ה-OS.	רק ה-OS נמצא כאן.
רק חלק מסוּם של הפקודות ניתן לביצוע.	המעבד מבצע בלבד פקודה.
ניתן לגשת ולשנות את המצב של כל רגיסטר במעבד.	רק חלק מרגיסטרים של המעבד נגיש.

- המעבר בין המצבים הנ"ל מתבצע בעוררת ביט בווד (mode bit) ונקרא **:mode switch**.
- מעבר מ-**unprivileged** ל- **privileged** מתבצע בעזרת הרצת פקודה שמשנה את ביט.
- מעבר מ-**unprivileged** ל- **unprivileged** קורם ורק בעקבות מאורעות חומרה – **exceptions** (חשוב להבחין בין hardware exceptions לבין software exceptions בהם נתקלנו בתוכנות).
- במעבר בין ריצת מערכת הפעלה לבין ריצת תהליך אחר מתרבעים 2 שינויים: **-state switch** ו- **:exceptions**
- 1. **system call** (קריאה מערכת) – הריצה של פקודה מיוחדת שמצוינת שהתהליך צריך איזשהו שירות של מערכת הפעלה.
- 2. **exception – Fault/Trap** – שמייד על תקלה (למשל: חלקה ב-0, גישה לבתו של ללא במרחב הכתובות שמקורה לתהליך).
- 3. **Interrupts** – אירוע (event) חיצוני שהגיע מחומרה המkosherת למחשב (למשל: המשתמש לחץ על בפטור מסוים).
- **exceptions** מחולקים ל-2 סוגים:
 1. **סינכראוני** – נגרם בתוצאה מהרצת של פקודה בלבד.
 2. **אסינכראוני** – נגרם בתוצאה מאירוע חיצוני למעבד.
- **צעד – צעד בעת exception** בזמן ריצת תהליך:
 1. המעבד משנה את מצבו למצב **privileged**.
 2. המעבד **שומר חלק מהמצב (state)** של התהליך הנוכחי. במנימום – חיבר לשומר את ה-PC (program counter), לאחר מכן שה-kernel יכול לזרע הוא יהיה חייב לדרכו את ה-PC. את כל שאר הרגיסטרים הוא יכול לשומר אח"כ תוא"כ ריצה.
 3. המעבד **טוען מצב של מערכת הפעלה** שנועד לטיפול במקרה של **exception handler** (OS handler).
- איך המעבד יודע לאיזה OS handler OS עלוי לקפוץ?
- בחלק השמור (privileged) יש למעבד מצביע לבניינה נתונים (**Exception Table**) שבעת הצורך יודע להגדיר איזה exception handler צריך לטוען עבור כל exception.
- מעדכן את ה-PC להיות הכתובת של ה-**OS handler** המתאים.
- 4. ה-**kernel** מתחילה להרץ על ה-**OS handler** שנסלף ומתחילה לשומר תוא"כ כדי ריצה את הערכים של המצב הקודם, כדי שייחליף אותם בערכיהם שלו.
- 5. בסיסים הטיפול ב-**exception** ה-OS בוחרת תהליך חדש להרץ, ובמצעת תהליך יציאה מהמצב של ה-**exception**:
 - a. מחזירה לרוגיסטרים של המעבד את הערכים שהיו להם לפני ה-**exception**.
 - b. קוראת לפקודת מיוחדת (iret: 86x) כדי לסיים את השחזר.
 - c. **עדכון ה-PC** תליי ב-**exception** שהכנים את ה-**kernel** למצוב טיפול בשגיאה:
 - **next instruction after system call** = PC
 - **fault/trap or interrupt** –-> ה-PC קיבל את אותה פקודת שהיא הול כדי שהפקודה שלא בוצעה (והובילה לשגיאה) תבוצע, מאחר ובפעם הקודמת היא לא בוצעה.
 - בש-OS מגיעה ל-**fault** שהוא לא יכולת לתקן, היא הורגת את התהליך (למשל: כמו ב-**segmentation fault**).

Interrupt Vector Table/Exception Table



- מערך שבו לכל אינדקס exception handler מסוים המתאים.
- הכטובה למערך זהה שמורה בתוך **רגייסטר מסויים** במעבד.
- בר שבעת שגיאה, "השליפה" של exception handler של המתאים מתבצע באופן **חומרתי** ללא עזרה ממערכת הפעלה.
- מערכת הפעלה רק אחראית על עדכון המערך הנ"ל ושמירת כתובתו ברגיסטר המתאים במעבד.

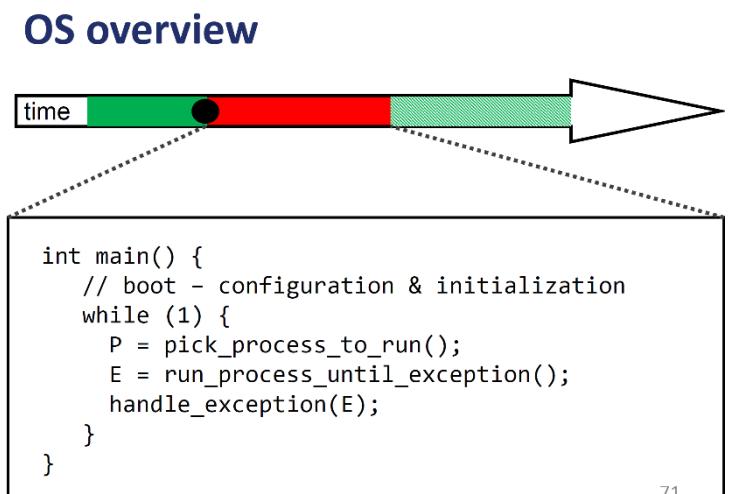
System Call

- רוב השגיאות (exceptions) שקרו – הן בכלל system calls, בשתהילך צריך איזוחה שירות מערכת הפעלה.
 - באשר זה המצב, ה-OS handler יקרא לפונ' המתאימה בהתאם ל-system call.
 - לאחר השירות, הטיפול מסתיים ושאר ה-flow הוא כמו ב-exception רג'ל.
- ה-64 עבור handler syscall**
- 1 ב- %eax למבצע write
 - \$rcx – רגייסטר לשימירת ה-PC של הפקודה הבאה (בחלק מהאריכתקטוורות נשמרת הפקודה הנוכחית ובעזרתה מחשבים את ה-exception).
 - הכטובה של הפקודה הבאה) בתוכנית שיצרה את ה-exception.
 - ניתן לראות שהמעבר ל-OS handler רק לאחר שמיירת ה-PC מוביל לשומר את ערכיו שאור הרגייסטרים. כמו שנאמר לעיל, ערכי הרגייסטרים ה-privileged והאחרים נשמרים רק לאחר שהריצה עבורה ל-OS handler (עד כאן ריגיסטר זהה נדחף לממחסנית).

CPU pre-emption

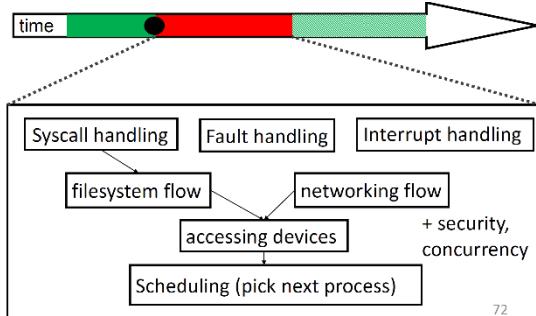
- בשביל לא הגיעו לנצח בו התהילך הנוכחי נכנס לופ אינסופי, מערכת הפעלה מקציבת זמן עם טיימר (פנימי במעבד), כך שבחולוף הזמן יופעל timer interrupt (exception אסינכרוני)
- מצב שבו ה-systemtakes control מהשאב שמיישה משתמש בו עבשו, ואח"כ אולי ייחזר לו אותו ואולי לא. pre-emption

OS – overview



71

בשלב ה-**.timer interrupt**, תגדר את ה-**handler table**, ניתן לתוכה מנגנון הפעלה בתחילת העבודה.



- ו-**handler table** על הקernel/מערכת הפעלה באופן הבא:
 - בקרה על הפעלה יש handler**-ים: הפונקציות אליהם המעבד קופץ במקרה של exception.
 - נתן לואות את מערכת הפעלה בתור שורה בעל סוגים שונים של שירותים לטיפול ב-exception.
 - ה-**handler**-ים הם הפונקציות שטיפולות בכל בקשה.
 - חלוקת מהטייפול של ה-**handler** הוא יכול לקרוא למודולים אחרים שיכולים לעזור לו בטיפול בבקשתה.
 - בד"כ נפוץ להשתמש בהתקנים כלשהם** (למשל: קרייה/כתיבה של קבצים מהאחסון, שליחת מידע רלוונטי ברשות).
 - לבסוף, יש מודול של **scheduling** שמחלית מי יהיה התהיליך הבא שיירוץ על המעבד.
 - מערכת הפעלה נותנת לאוטו תהיליך שנבחר ע"י ה-**scheduler** להמשיך לרוץ מאותה נקודה שבה הוא עצם.

Context & Context Switches

המערכת תמיד נמצאת ב-**context** של איזשהו תהליך.

- באשר המעבד מרים איזשהו תהליך (process) ולא את ה-**kernel** עצמו איזה context הוא אותו תהליך exception.
- הweeney של ה-**context** הוא שטמיד יהיה מוגדר עבור איזה תהליך ה-**kernel** ברגע עובד.
- כל ה-W-flow-ים של הקוד ייקחו את עניין ה-**context** בחשבונו. למשל: בעת בדיקת הרשאות גישה.
- מעבר בין תהליכיים גורר מעבר בין context-ים ונקרא **context switch**.
- kernel context switch הוא מושגalog שבו מוחלף התהיליך הנוכחי מבחינת ה-**kernel**.
- אריך ה-**context switch** ממומש בפועל.

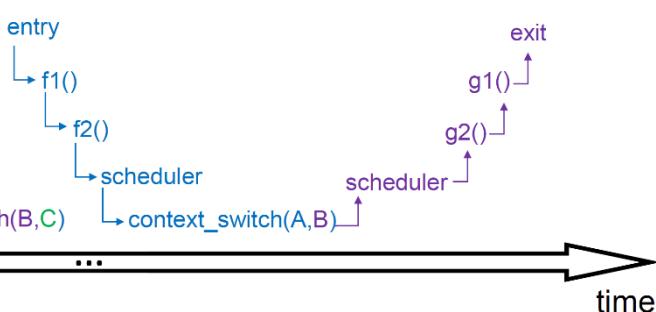
- ה-**kernel** מחזיק מבנה נתונים עבור כל תהיליך – **struct :PCB = process control block** שמוביל את המידע הרלוונטי עבור התהיליך (למשל: מצב המעבד state) שבו התהיליך היה כאשר הוא גרם ל-/חטף exception) ב-**context switch** שלו.
- ל-**kernel** יש משתנה גלובלי של ה-**current process** שמצוין על ה-**PCB** שהמערכת (system) ב-**context switch** בו:
- שינוי ה-PCB עלייו מצבים** = החלפת מצבים (mpsikim להצביע על ה-**PCB** של התהיליך החדש ומתחלים להצביע על ה-**PCB** של התהיליך החדש.

- טעינה ושמירה של מצב המעבד** - טעינה של חלק ממצב המעבד של התהיליך החדש לתוך המעבד ושמירה של חלק ממצב המעבד היישן לזכרון.

```
/* Also unlocks the rq */          ← called by prev but
rq = context_switch(rq, prev, next, &rf);    returns in next
```

הfonktsia שמבצעת את ה-**context switch**:
נקראת ב-**context switch** של התהיליך היישן, אבל בחזרה
מןנה נמצאים ב-**context** של התהיליך החדש.

בפרט זה אומר, שאם בעיתך נרצה
לחזור לתהיליך שמסומן בעט ב-
prev איזה הריצה שלו תמשיך בדיק
מאזדהה נקודה בה הוא נמצא
עבשו. אפשר לחשב על זה בדיק
במו מחסנית קריואת של פונקציות.
המצב שחווזרים אליו בשמתפקידם
במחסנית הוא המצב שבו הינו
באותה נקודה במחסנית:



2 סוגים/סיבות של **context switch** (מתייחסים ל-**context switch** בהקשר לשיבת בغالיה הוא קרה):

1. **involuntary context switch** – כאשר ה-OS מבצע על pre emp מטרת מרצונו על המעבד (למשל: התוכנית שרצה מבצעת sleep, sleep, מצבים שבהם ה-**kernel** לא מסוגל לספק לתהיליך את השירות שהוא בקש ברגע, וכך מחייב לבצע תהליכיים אחרים עד שיהיה ניתן לספק את השירות)
2. **voluntary context switch** – לועמת ה-**kernel** מתייחס ליציאה או כניסה מה-**kernel**.

- קיימים מקורות שמתיחסים למחב המעבד בטור **context**, וכך הם קוראים לפעולה הפיזית של שינוי מצב המעבד – **.switch**

context switch	mode switch	CPU state switch
מושג לוגי שմדבר על "באייה context" מושג שמי ה-mode privilegedpriv של המעבד	שינוי ה-mode ה-privileged של המעבד	שמירת המצב של המעבד

Kernel Threads

- נרצה לאפשר ל-kernel להריץ גם בעצמו תוכניות שיוכזו "בקביל"/ברקע (יקבלו זמן CPU לשירות).
- אם היינו מכנים את אותן תוכניות לאוון רשיימה של שאר התוכניות שאינן ה-system, אז התוכניות הללו לא יקבלו גישה למשתני ה-kernel.
- לכן, מגדירים איזשהו אובייקט היברידי שיפטור זאת – אובייקט שהוא variant של תהליך – kernel thread :

 - יהיה לו PCB ויעשו לו scheduling
 - אבל כשהוא יירוץ הוא יירוץ בתווך תהליכי של מערכת הפעלה וירוץ ב-mode privileged

- **הצורן וירטואלי** נובע מכך שבעת context switch נרצה שככל הזיכרון שRELוגוני לתחילה החדש יהיה זמין עבור המעבד (ב-D-ram).
- לאחר ובכל תחילך לא צריך את כל ה-Ram-D זה יהיה מיותר להקצות כל תחילך שמתבצע את כל הזיכרון.
- במקומות זאת, נחזיק את הזיכרון הנדרש לכמה תהליכי B-Ram-D במקביל ואז לא נדרש להעביר את כל הזיכרון בעט context switch.
- מה virtual memory נתונים לנו?

 - כל תוכנית תקבל מרחב זיכרון מסוים ללא תלות בסך הזיכרון הזמין הקיים (אפשרי בזכות המisor ← מרחב הזיכרון המוצע לתוכנית אינו בהכרח הזיכרון הפיזי הנגיש לה). לעומת, יש יותר זיכרון שניין לעבוד אותו מהר יותר. "נפח של disk בmahiorot של RAM"
 - **security** – תוכנית לא תוכל לגשת למרחב זיכרון של תוכנית אחרת (מאחר והוא-VM מייצר מisor ← הכתובות של תוכנית לא בהכרח יהיו הכתובות הפיזיות אליהן היא ניגשת)
 - **אורותוניות** – כל תוכנית תראה את אותו מרחב בתיבות (בגלל המisor בפועל זה יהיה מרחב בתיבות שונות).
 - **ניהול-VM מתבצע ע"י תוכנה**.
 - DRAM = main memory – physical memory
 - main memory – physical address
 - – כתובות ב-disk – הזיכרון הדמיוני שבפועל נמצא ב-disk.
 - – כתובות ב-disk – virtual memory
 - – כתובות ב-disk – virtual address
 - **שים לב: הזיכרון הוירטואלי הוא בבתים! הכתובות נבדלות בהפרש של בית.**
 - ← אם נתון שרחבת virtual address הוא א' ביטים א' הזיכרון הוירטואלי מביל 2^{32} בתים
 - **page** – יחידת הזיכרון הקטנה ביותר שהיא נבעוד ב-VM. כאשר יש זיכרון שאין נמצאים ב-main memory נביא אותו מה-disk בלבד עם כל ה-page שלו.
 - **אין זה ישבוד?**
 - נחשפ מידע מסוים בזיכרון הפיזי (DRAM) בהתאם לכתובת הוירטואלית.
 - אם הוא לא נמצא שם נביא את הדף שלו מה-disk.
 - **אם he-kernel לא עובד יישירות מול הזיכרון הפיזי, אלא מגדי למרחב בתיבות וירטואלי.**
 - **המיופים של הזיכרון הוירטואלי יכולים להיות דינמיים.** ככל, מערכת הפעלה יכולה לעדכן את המיופים של כל תהליכי זמן הריצה. למשל: אם תחילך צריך עוד זיכרון הוא יעשה system call לבקשת זיכרון נוספת ממערכת הפעלה. היא בתגובה תיצור מיופים חדשים כדי לחתת תחילך זיכרון נוספת. באופן אנלוגי, כאשר תחילך משחרר זיכרון (מע' הפעלה יכולה להקטין את מרחב הכתובות שלו ולהשמיד מיופים).

איך הזיכרון הוירטואלי ממומש בחומרה

- **Memory Management Unit = MMU**
- **הרכיב החומרתי שאחראי על תרגום כתובות וירטואליות לפיזיות.**
- ממוקם בתחום המעבד.
- ב-main memory נעשה ע"י המעבד "מאתורי הקלעים" רחוק מהתחלת, שמהינתו קיבל את המידע שביקש בכתובת הוירטואלית.
- גם מערכת הפעלה עבדת עם זיכרון וירטואלי ולא יישירות מול זיכרון פיזי.
- **הקוד היחידי שעבוד יישירות מול הזיכרון הפיזי הוא הקוד שרצ' בזמן-boot והקוד שאחראי על המיופים.**
- **בגישה לזכרון וירטואלי שלא קיים עבורה מיפוי נקלט hardware exception (MMU fault) fault סוג**
- גם מערכת הפעלה יכולה לקבל MMU fault (מערכת הפעלה היא בסך הכל תוכנית שרצה במצב privileged.).

• מבנה הנתונים עבור ה-MMU

- השימוש במערך (כמו שעשינו במבנה מחשבים) לא יהיה ישם מאחר והמערך יהיה בגודל מרחב הכתובות הווירטואליות או טיפה פחות (אם משתמשים בגרנולריות גדולה יותר בעורף שימוש offset).

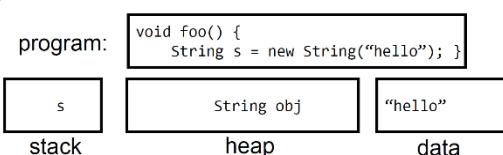
◦ – זיכרנו התוכנית מקובץ מבחינה לוגית לתוכה segments

◦ סגמנטים:

- **Code segment** – הקוד של התוכנית
- **Data segment** – משתנים גלובליים וסטטיים
- **Heap** – זיכרנו המוקצה דינמיית
- **Stack** – משתנים מקומיים

◦ סגמנטי ה-code וה-data הם חלק מהקוד של התוכנית (ה- program binary) בעוד סגמנטי heap וה-stack יתענו לזכרון הראשי רק בזמן ריצת התוכנית.

◦ דוגמא:



- כאשר מערכ הפעלה מרים תוכנית מסוימת היא מעתקה את ה-code וה-data שלהם לזיכרון וצריכה לדאוג להקצות מקום עבור הסגמנטים האחרים dynamic memory = .dynamic memory

◦ segment table

- מערך **בגודל קבוע** המאונדקס לפי סוג הסגמנט: אומר למעבד איפה בזיכרון הפיזי נמצא כל סגמנט.
- ישמור בזיכרון וכתובתו תען לתוכו ורישטר במעבד בעת הריצת התוכנית כדי שה-MMU ידע לחפש בו.

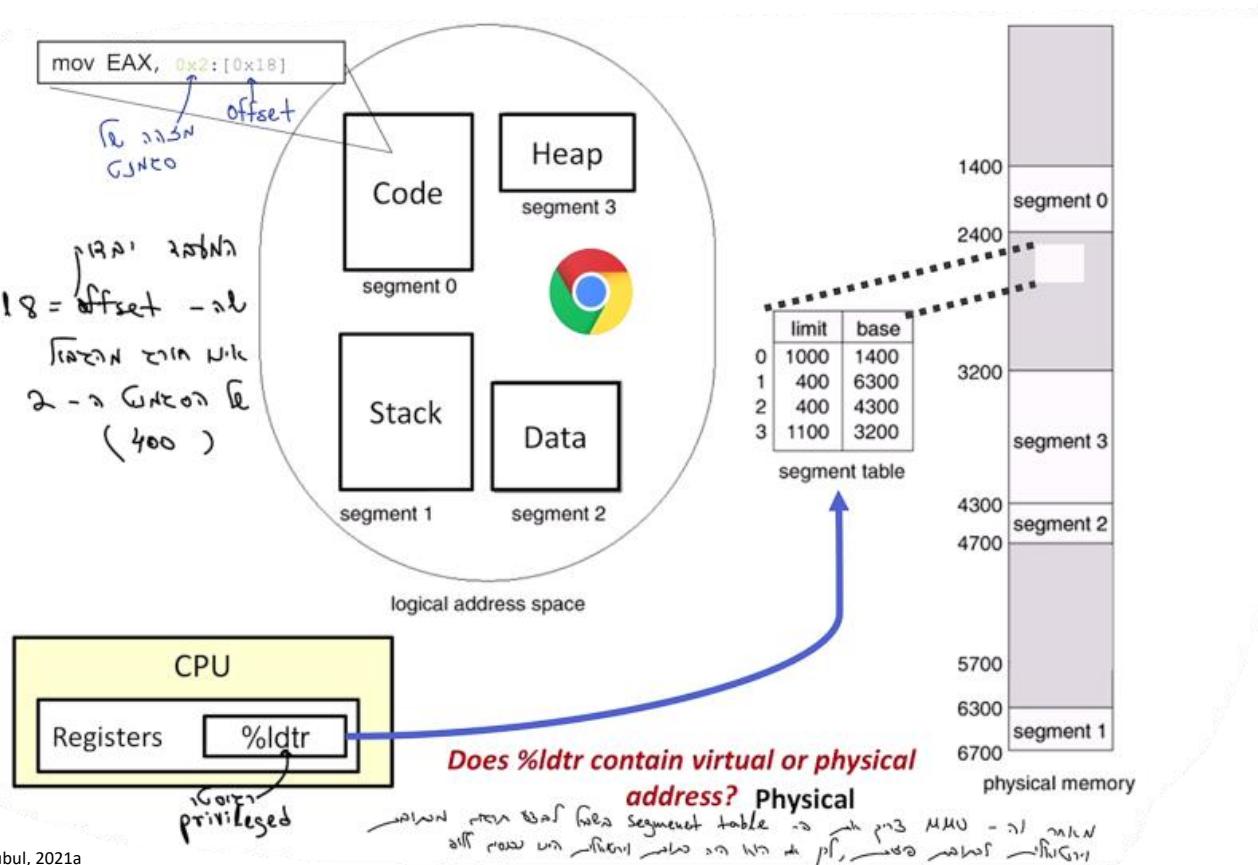
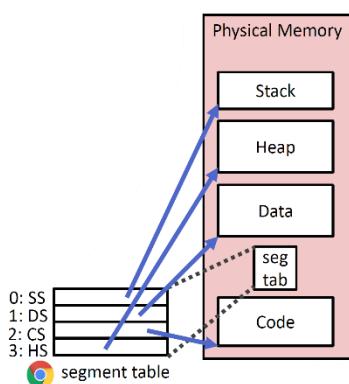
◦ איך המעבד ידע, בהינתן כתובות וירטואלית, לאיזה סגמנט היא שייכת?

◦ כאשר עבדים עם segmentation הכתובות הווירטואלית תורכב מזוג <id, offset>:

- id של הסגמנט אליו הכתובת שייכת
- offset בתוך הסגמנט

◦ מה לגבי הגנה? איך ניתן למנוע מטהילך לגשת לכתובות וירטואלית החורגת מגודל הסגמנט שלה?

◦ הרשומות ב-table segment יכלו גם את limit של ה-segment.



- 2 בעיות עם segmentation
 - (1) מה עושים אם צריך יותר זיכרון דינמי ממה שהקצנוuboר התהילר?
 - (2) – כאשר יש לנו מספיק זיכרון עבור ה-segment אבל לא בצורה רציפה.

הפתרון..

Paging

- הפתרון שנמצא היה בשימושuboר בעית המיפוי בין זיכרון ווירטואלי לזכרון פיזי.
- הבעיה עם segmentation הייתה בעיה של גרכולריות – segment-segment הם גודלים מדי מכדי שטוכבל לניה לפיהם את מרחב הכתובות של תהליכיים.
- היחידה הגרכלרית הקטנה ביותר אליה נעובד היא דף = page.
- הגודל של דף מוגדר על ידי החומרה (Robbins 4K -> הרבה יותר קטן מ- segment).
- דף בודד חייב להיות רציף בזיכרון הפיזי.
- דפים שמופיעים ברכזיות (אחד אחרי השני) בזיכרון הוירטואלי, לא חייבים להיות רציפים בזיכרון הפיזי.
- frame = דף בזיכרון הפיזי
- דפים שונים בזיכרון הוירטואלי יכולים להיות ממוקמים לאוטו ה-frame.
- דפים segment חייבים להיות רציפים וכך לא יוכל היה לאפשר את גודל הזיכרון הוירטואלי ש-paging אפשר לנו.

page table

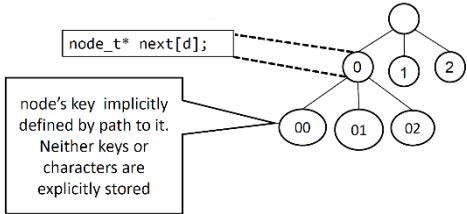
- טבלת המורה מזכיר ווירטואלי לזכרון פיזי. ממפה דפים ווירטואליים (VPNs) לדפים פיזיים (PPNs).
- טבלה לכל process .
- הטבלאות יאוחסנו ב-RAM.
- כתובת הבסיס לטבלה (מבנה הנתונים בעזרתו המיפוי נעשה) נמצאת ב-PTBR = page table base register (מעודכן כאשר עוברים בין תהליכיים – context switch).
- ניתן לכתוב אל הריגיסטר זהה רק במצב privileged .
- כל תא ב-page table :
- בית valid שיצין האם המידע המבוקש נמצא ב-disk או ב-ram:
 - 1 RAM
 - 0 disk
- בית reference/used – משמש כדי לקבוע איזה דף יוסר מהזיכרון בעת page fault .
- הדפים עם בית 0 יוסרו בעת הצורך.
- מודולק באשר ניגשים לדף.
- מערכת הפעלה מכבה אחת לבמה זמן את reference bits .
- בית dirty/modified – האם המידע ב-disk (hard disk) מעודכן למה שיש ב-RAM (main memory) .

הציגו של ה-page table בזיכרון

- מערך (כפי שהוצג במבנה מחשבים) לא מתאים מאחר יהיה בו מס' ענק של כניסה ריקות (יכיל מס' כניסה במס' הכתובות הוירטואליות).
- המטרה: בניית שעריבת הזיכרון שלו היא פונקציה של מס' הדפים הוירטואליים הוולידיים ללא תלות בשמות הדפים שאינם בשימוש (not valid) .
- שימוש ב-table hash אין מתאים מאחר וחיפוש והכנסה אליו יכולים להיות בזמן **لينארי** במקרה הגורען. בנוסף להזמנת הרצה עלול להשנות בין מקרים (לא דטרמיניסטי).
- **עץ חיפוש בינארי** גם אינו מתאים מאחר וחיפוש והכנסה יכולים לקחת זמן **לוגריותמי**. בנוסף להזמנת הרצה עלול להשנות בין מקרים (לא דטרמיניסטי).

Trie Page Table “alphabet” size

- trie – הפתרון האמיתי:
 - זמן החיפוש (מה שה-MMU עושה) וזמן העדכוןים (מה שה-kernel עושה) הם קבועים.
 - לכל צומת יהיו d ילדים כאשר $d = \text{מספר האותיות ב-א"ב}$ מעליו אנחנו עובדים.
 - חיפוש ב-trie:



נכיה שכל המפתחות הם באורך זהה.
כל צומת הוא מערך של d מצביעים

- דוג': נניח שהמפתחות הם מחרוזות המורכבות מ-3 סימנים וכי 8 ביטים כל אחד = מחרוזות של 3 בתים.
 - לכל צומת יהיו $2^8 = 256$ מצביעים לילדים (마חר וכל אות מורכבת מ-8 ביטים -> יש 2^8 אותיות).
 - בرمה الأخيرة, במקום מצביעים לילדים יהיו **מצביעים לכתובות פיזיות** (마חר והנחנו שכול המחרוזות באותו האורך, אז הן יסתימו באויה הרמה, כך שאנו סיכוי שיכללו להמשיך אחר הרמה הזאת לילדים נוספים, למורשת שנייה להכליל את ההנחה ולבצע התאמת כך שהיא ניתנת לאחסן ולהיפש מחרוזות באורכים שונים - ע"י הפיכת המערך למערך של זוגות כך שהאיבר השני מכיל מצביע לכתובות פיזיות אם קיים מיפוי מאותו הצומת).

המפתחות נשמרו ב-trie יהיו **VPNs** (fixed length)

- ה-values יהיו ה-**PTE** = Page Table Entry
- מיפוי (page walk) ייקח **זמן קבוע** לפי איך שנחלק את ה-VPN לשימבולים
- מבנה בזה (המייצג את ה-PT בעזרת trie) נקרא **multi-level page table**.
- דוג' לחיפוש:

- במעבד יש את ה-PTBR = הרגיסטר שמאכיעם לבנייה הנתונים של המיפויים = מצביע לשורש של ה-PT.
- כדי לחפש VPN-MMU שובר אותו לקבוצות החל מהבאים הגבוהים כלפימטה.
- בכל שלב ניגש לczomת ברמה ה-i של ה-PT ומסתכל במערך שלו (PTE) שימושים ב-PPN שלו כדי לשמר את המצביע לשלב הבא) لأن מצביע האיבר שמתאים לחלק ה-i ב-VPN.
- בכל שלב אם בית ה-idx של הכתובת הרלוונטי במערך הוא 0 אז אין VPN במרחב הכתובות עם ה-prefix שchipshnu עד שלב ? (אם זה אכן המצביע – החיפוש יעצר והמעבד יזרוק page fault).
- אם החיפוש של ה-MMU מצליח להגיע לדינה התוחטונה ביותר אזי ב-PTE שהוא מוצאת יהיה את ה-PPN שלו כל ה-VPN מתמפה.
- ברגע שיש את ה-PPN, ה-MMU לוקח את ה-PO Virtual Page Offset = PPO שהוא שווה ל-Physical Page offset = PPO.
- ומחבר אותו ל-PPN ובכך מקבל את הכתובת הפיזית שאליה הכתובת הווירטואלית מתורגם.

Trie Page Table “alphabet” size

- איך קובעים את גודל ה-symbol של Trie = כמה חלקים צריך לשבור את ה-VPN?
- מס' זה קובע את גודל המערך בכל שלב של ה-table page גם את מס' הרמות ב-PT.
- אם נקח בתוך גודל ה-VPN איזי נקבל פשוט את הפתרון של מערך.
- מאחר ומדובר ב-VPN הוא 64 ביטים איזז צומת הראשון יהיה 2^{64} ילדים. בדיק במערך (מבנה מחשבים סטיל)

גודל ה-symbol נקבע לפי האילוץ: צומת יהיה בדיק בגודל של page פיזי = 4KB = frame

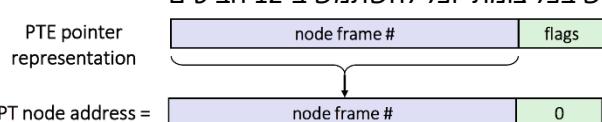
בהתדרה! גודל צומת = גודל frame

בנוסף, אנחנו רצים שהכתובת של הצומת תהיה 4K aligned = כפולה של K.

גורר את זה ש-12 הביטים התוחטונים של צומת תהייה 0.

- לכן, לא צריך לייצג אותם בשארצים לבנות מצביע לצומת -> כל PTE במערך שיש בכל צומת יוכל להשתמש ב-12 הביטים התוחטונים לגדלים (למשל: pval). בכר חוסכים מקום וגם מקצים זמן.
- כשהחומרה עוברת על PTE היא מתעלמת מ-12 הביטים התוחטונים.

גודל ה-frame גורר את גודל הא"ב -> גודל המערך בצוות



physical page (=frame) size = 4KB : מילוי

PTE size = 8B

(השווים) Frame size = Trie node size : מילוי מושג אחד שלן

(node הינו בפונקציית המאץ) גודל הנקודות מושג בפונקציית המאץ *

Trie -> ינטן ON.

G3	48bits	12..11	0
Sign ext	VPN	offset	
16b	36b	12b	

: X86 -> VPN מילוי *

frame -> פוןקציית המאץ

מילוי 16bits כפונקציית המאץ

כפונקציית המאץ כפונקציית המאץ

מילוי 16bits כפונקציית המאץ ← 48bits →

Trie -> ינטן ON

מילוי יונקן 9b = ינטן 9bits

פונקציית (PTE) מילוי יונקן מילוי יונקן מילוי יונקן

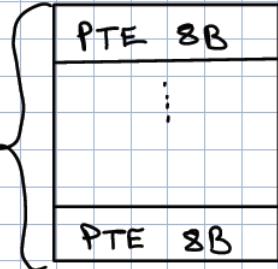
node -> יונקן

$$\frac{4}{4} = \frac{36}{9} = \frac{\text{VPN Size}}{\text{Symbol size}} = \text{מילוי יונקן} = \underline{\underline{\text{ינטן}}} \text{ ON}$$

$$\frac{4\text{KB}}{8\text{B}} = \frac{2^12\text{B}}{2^3\text{B}} = 2^9$$

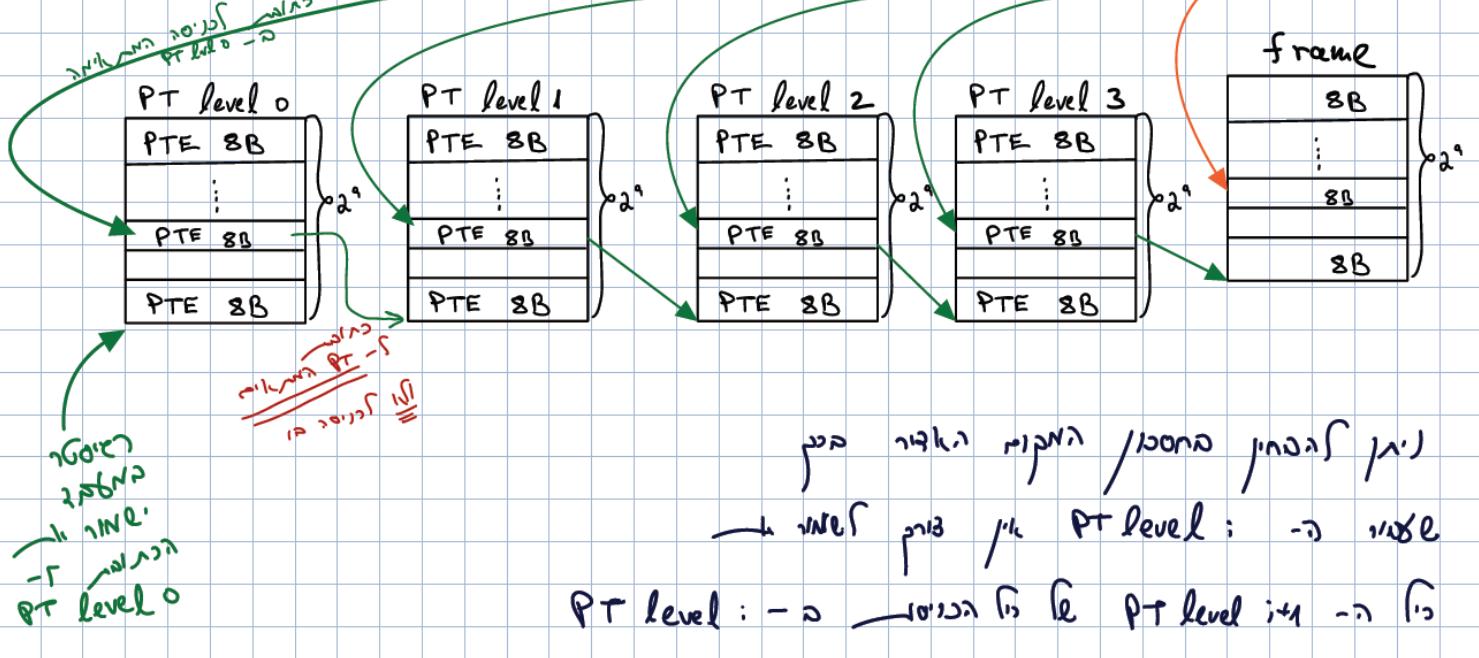
מילוי יונקן

node 4KB

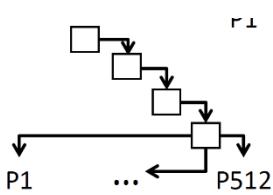


47	39 38	30 29	21 20	12
VPN symbol #1	VPN symbol #2	VPN symbol #3	VPN symbol #4	
9b	9b	9b	9b	

Virtual page offset



- עברו 512 כתובות וירטואליות במבנה של טבלאות דפים ב-4 רמות עם 512 כניסה בכל טבלה – כמה frames נחוץ?



ברמה ה-1: נדרש רק טבלה אחת כי יש בה 512 כניסה.

ברמה ה-2: נחוץ 512 טבלאות כי הדפים יתפרסו על כלם (דף בכל טבלה).

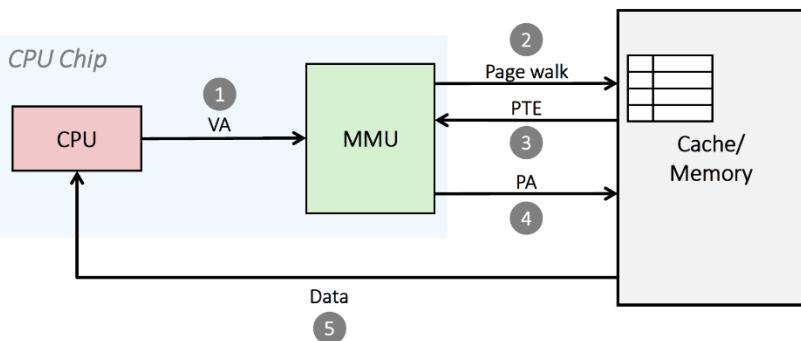
ברמה ה-3 וה-4: כמו ברמה ה-2.

סה"כ: $1 + 512 * 3 = 1537$ טבלאות דפים

במקרה הטוב:

- בכל הרמות למעט הרמה ה-4: כל הכתובות יMOVו לאותה הכתובת - < 4 frames
- לחוב הפיזור של הדפים יהיה קרוב למקרה הטוב מאחר ועובדים עם סגמנטים בר שמרחבי הכתובות יחסית צפופים (spatial locality).

תהליך הגישה לדיברן (כרען)



- .1 המעבד מבקש VA מ-MMU
- .2 ה-MMU עושה page walk ב-pte
- .3 ה-MMU מקבל PTE התואמת ל-VPN
- .4 ה-MMU ניגש לדיברן עם ה-PA שחילץ מה-pte
- .5 המידע הרלוונטי בהתאם ל-PA מגיע ל-CPU

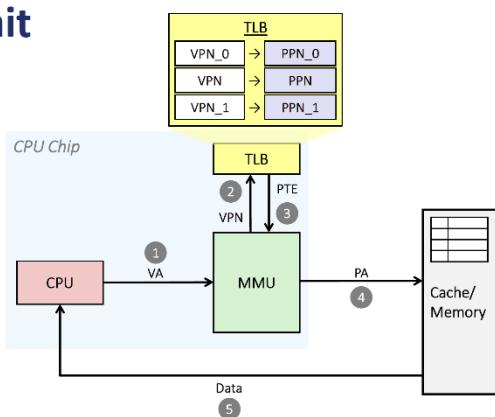
סה"כ 5 גישות לדיברן הפיזי עברו לקבלת המידע

הסבר: בהנחה יש 4 רמות ב-PT נועבור על כלן (4 גישות לדיברן הפיזי) + גישה 5 עם ה-PA

TLB (= Translation Lookaside Buffer)

- נמצא ב-MMU cache
- עושה VPNs ל-PPNs caching בין

TLB hit



תהליך הגישה לדיברן תוך שימוש ב-TLB

- .1 המעבד מבקש VA מ-MMU
- .2 ה-MMU מחפש את ה-VPN ב-TLB

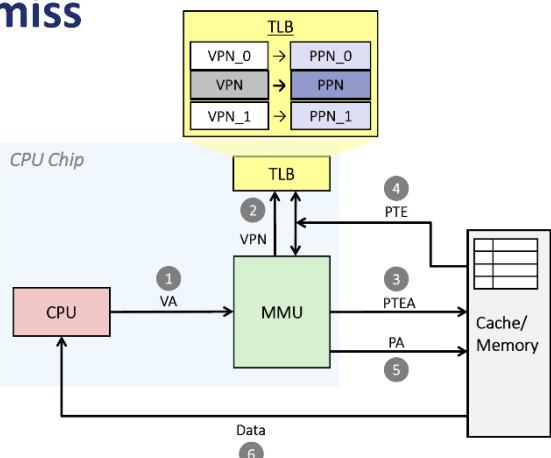
3. TLB hit – ה-VPN קיים ב-TLB

4. עיבוד ה-VPN ל-PA לקבלת PA וגישה לדיברן הפיזי

5. העברת המידע ל-CPU

סה"כ גישה אחת לדיברן הפיזי.

TLB miss



- .3 ה-MMU עושה page walk ב-pte
- .4 ה-MMU מקבל PTE התואמת ל-VPN ומכו尼斯 אותה ל-TLB ול-MMU
- .5 ה-MMU ניגש לדיברן עם ה-PA שחילץ מה-pte
- .6 המידע הרלוונטי בהתאם ל-PA מגיע ל-CPU

סה"כ 5 גישות לדיברן הפיזי.

- פקודה מוכנה שמערכת הפעלה מרים (ניתן להריץ רק במוד privileged) וגורמת לירוקן ה-TLB מכלל המיפויים שהוא שומר. שימושי כאשר יש עדכון של ה-page table וייתכן שה-TLB אינו מעודכן.
- קיימת גם פקודה שעושה invalidate לרשומה ספציפית ב-TLB ולא בכלל.

VM Access Right

- הזכות עבורה דפים היא ברמת המיפוי הווירטואלי ולא ברמת הדף הפיזי.
 - למשל: יתכנו כמה מיפויים וירטואליים לאוטו דף פיזי בך שחלק מהמיפוייםאפשרים מאפשרים X וחולק לאאפשרים X.
- כל ארכיטקטורת חומרה מגדרה את-h-access rights – זיהוי יודעת לאכוף. מערכת הפעלה מחליטה איך ובאיזה השארכטקטורת החומרה החילתה לאכוף.
- (read only – אפשר לתהילכים שונים לחלק את אותו זיכרון פיזי עבורה הקוד שלהם מריצים (למשל: 2 תהליכי chrome) – זיכרון שביביט דלוק אצלם יגרום לזייכון להיות נגיש רק אם מי שביקש את הזיכרון הוא במוד שארכטקטורם. אחרת – page fault – להריץ קוד/לקרא או לבתו בדף).
-

הចורך עבורה bit :privileged accessed right

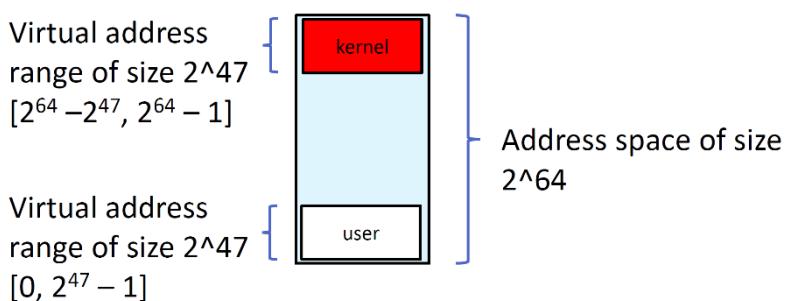
באשר מתאפשר exception, מערכת הפעלה עובה להיות התהילר שמורץ. בחלוקת מהטייפול ב-exception היא צריכה לגשת לזייכון של התהילר שגרם ל-exception (בנוסף לזייכון שROLונטי ל-kernel).

הפתרון: המיפויים מדיכון ווירטואלי לזייכון פיזי עבורה בתובות של מערכת הפעלה (ה-kernel) ישולבו עם הזיכרון הווירטואלי שלו בכל תהילר שරץ וכבה בעת exception נוכן לגשת מיד לזייכון של מערכת הפעלה ומוביל שנצטרך לטעון אותו.

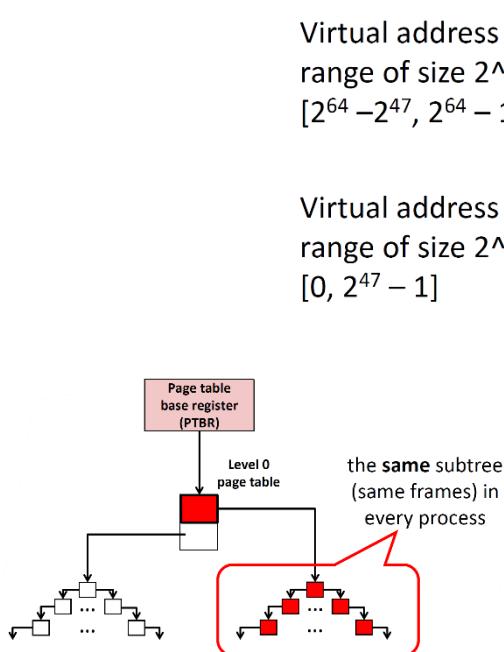
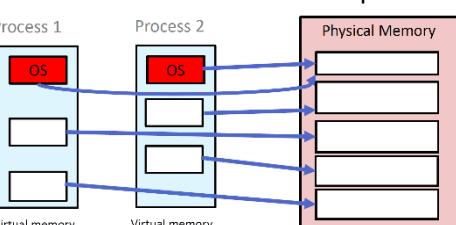
הסכמה: תהליכי יכולו לגשת לזייכון של מערכת הפעלה אחריו והמיפויים שלו מדיכון ווירטואלי לזייכון פיזי ישולבו בכל תהילר. **המשמעותה זאת הוא bit privileged accessed** – רק מי שנמצא במוד OS (ה-kernel) יוכל לגשת לזייכון זה.

איך עובדת שתיילת מרחב הכתובות של kernel במרחב הכתובות של תהליכי אחרים?

- ראשית, איך גורמים שטוחה הכתובות של התהילcis לא יתנגש עם טוחה הכתובות של kernel?
 פתרון: מדירים טוחה הכתובות שלתהליכים אסור להשתמש בו:
 - מדירים את מרחב הכתובות עבור kernel ומרחב הכתובות הנמוכים עבור תהילcis
 למשל: חלוקת הזיכרון בlianoks בארכיטקטורת x86:

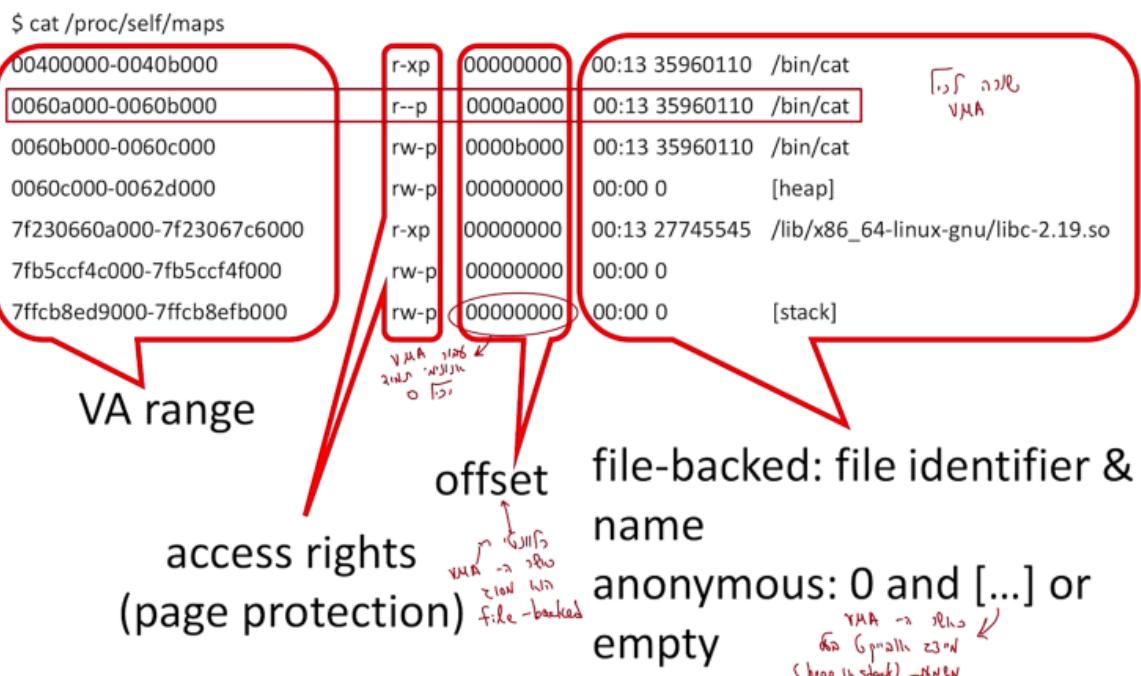


- בעת, כיצד מתחזעת השתיילה?
 החיצי העליון של ה-PT בrama הראשונה מצביע לטע עץ של מרחב הכתובות של kernel.
 בעוד, החיצי התחתיו של ה-PT בrama הראשונה מצביע למיפויים של התהילר.
 המצביעים לטע עץ של מרחב הכתובות העליון (של kernel) בכל תהילר
 מצביעים לאותו טעם.
 ככה ששני במרחב הכתובות של kernel מתעדכנים עבורה כל התהילcis.



VMAs (= Virtual Memory Areas)

- VMA – טווח רציף של כתובות וירטואליות.
- יתכן והכתובות הווירטואליות של VMA אינם ממופות לכתובות פיזיות רציפות.
- מרחב הכתובות של תהליך מורכב מאיחוד של כמה VMAs.
- כמובן, מרחב הכתובות הווירטואלי של תהליך אינו רציף, אך הוא מורכב מאיחוד של מרחבי כתובות וירטואליות שככל אחד מהם רציף.
- דוגמאות ל-VMAs: סגמנטי ה-`code`, `data`, `stack`, ה-`kernel`.
- עבור כל ה-VMA מוחזק (ב-C) הנקרא struct kernel VMA (האו בייקט בקורסל שמייצג את המושג VMA קרי' באותו השם).
- למה לא להשתמש במושג ה-`segment` שהכרנו מוקדם?
- מושג ה-VMA הוא כלל יותר. נראה בהמשך שככל סגמנט הוא VMA, אבל לא כל VMA הוא סגמנט.
- VMA חולקים את אותן הרשות.
- **סוגי VMA:**
- **File-backed .1**
 - VMA המכיל בתים מקובץ הנמצא בהתקן אחסון בלהה (לדוג': סגמנט קוד של תוכנית)
 - הערך ההתחלתי של הבטים ב-VMA יהיה בערך הבטים בקובץ mmap
 - באופן דיפולט זה סוג ה-VMA הנוצר ע"י mmap
- **anonymous .2**
 - ה-VMA יכול בתים הנמצאים באחסון ללא שם (למשל: סגמנט ה-`stack` או ה-`heap`)
 - הערך ההתחלתי של הבטים ב-VMA יהיה 0.
 - סוג ה-VMA קובע מה יהיה הערך ההתחלתי של הביטים שהוקצנו.
 - איך זה נראה בפועל?



Process Address Space Management

- ה-kernel מספק ממשק syscall עבור תהליך כדי שיוכל לנוהל את מרחב הכתובות שלו.
- **system call – mmap()**
- ```
void *mmap(void *addr, size_t length, int prot, int flags,
 int fd, off_t offset);
```
- **addr** – ה-VA שהתהליך מבקש שה-VMA יתחילה ממנו. יתכן שה-OS יתעלם מהבקשה.
- ערך החזרה – ה-VA שה-VMA שהוקצתה יתחילה ממנו.
- **prot** – אילו Access rights יהיו ל-VMA המוקצתה.
- **fd** – מזהה הקובץ שה-VMA יוכל את הבטים שלו (file backed).
- **offset** – ה-offset בתוך הקובץ.

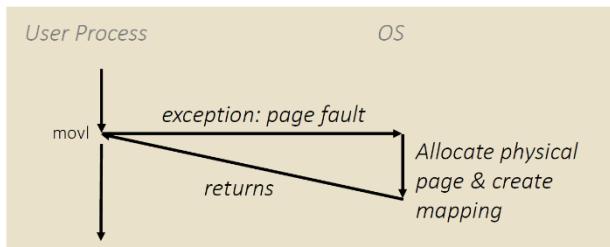
- בכמה בתים מהקובץ.
- כאשר רצום ליצור AVM **masg anonymous** יש להוביל דגל מתאים. כאשר הוא יועבר, ה-map יתעלם מהפרמטרים של ה-fd וה-offset ויתחשב בלבד.
- יצירה של טווח של כתובות וירטואליות ≠ הקצהה של זיכרון פיזי באותו גודל של הטווח.

### הקצת זיכרון פיזי

- הדיכאון הפיזי המוקצה עבור הדיכאון הוירטואלי שМОקצת אינן בשליטת התהילך ש牒בוקש את אותו, אלא לחולוטן באחריות מערכת הפעלה.
- זה יהיה בזבוז (זמן לייצור המיפויים וזכרון) להקצות ל-VMA הנוצר את כל הדיכאון הפיזי שהוא דרוש במעמד הייצור שלו, מאחר והוא לא בהכרח ישתמש בכלל.
- **demand paging**
  - הקצת הדיכאון הפיזי (והמיפויים הרלוונטיים ב-PT) נעשים רק בעת גישה לדיכאון לכתובות הוירטואליות שהוקטו (on demand).
  - כך זה יבוצע:

Process accesses an unmapped virtual page

|          |                      |                       |
|----------|----------------------|-----------------------|
| 80483b7: | c7 05 10 9d 04 08 0d | movl \$0xd, 0x8049d10 |
|----------|----------------------|-----------------------|



- חשוב לשים לב שכאשר ה-handler חוזר, הכתובת של הפקודה הבאה לביצוע תהיה הפקודה שגרמה ל-page fault.
- ה-page fault handler מקבל את הכתובת הוירטואלית של הדף שגרם ל-fault ב כדי שידע למי להקצת זיכרון.
- **ה-page fault handler צריך להבדיל בין 2 מצבים:**
  1. ה-page fault נגרם בתוצאה מגישה לבתוות מחוץ למרחב הכתובות.
  2. ה-page fault נגרם בתוצאה מגישה לבתוות וירטואלית שלא הוקצת עבורה בתובת פיזית.
- **הבדלה בין 2 המצביעים מתבצעת בעזרת ה-VMAs (או הסיבה שיצרו אותם):** כאשר נוצר ה-VMA, ה-page fault handler בודק האם הכתובת הוירטואלית שגרמה ל-page fault שייכת לאחד מה-VMAs.
- אם כן -> מצב 2: יש להקצת זיכרון פיזי וליצור עבورو מיפוי.
- אם לא -> מצב 1 : segmentation fault.
- **איך ה-kernel בודק האם VA נתון שייך לאחד מה-VMAs?**
  - מתחזקים עץ חיפושBINARIY שהמפתחות שלו הן טווחי ה-VMAs.
  - חיפוש על העץ יתבצע בסיבוכיות לוגריתמית ביחס לכמות ה-VMAs.

### demand paging page fault flow

1. ה-PF handler PF מוחפש את ה-VMA בעקבות
2. לא נמצא -> התהילך נהרג (SEGFAULT)
3. נמצא -> מէקה לו זיכרון פיזי ומאתחל את התוכן שלו בהתאם לסוג ה-VMA.

### עקרון הדחיננות/עצלנות (Laziness, deferred work)

- זה העיקנון לפיו פועל ה-demand paging.
- לא לבצע עבודה מראש לפני שבתווחים שורוצים לבצע אותה.

- על מנת להקצות לדף וירטואלי דף פיזי אליו הוא יMOVED יש לפתור 2 בעיות:
  1. מציאת דף פיזי פנוי.
  2. אם לא נמצא דף פיזי פנוי -> מה לעשות?

## מציאת דף פיזי

- המודול ב-kernel שמתמודד עם הבעיה של מציאת frame פנוי הוא: **frame allocator** – מבנה נתונים המממש את ה-interface הבא:

**Alloc** – מציאת frame שלא בשימוש.

**Free** – קבלת frame שכרגע בשימוש והיפיכתו ללא בשימוש.

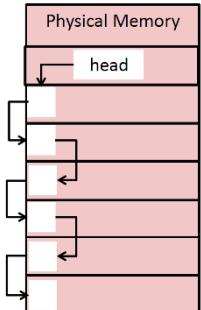
מבנה הנתונים שימושו לזהה שהוא רשיימה מקוורת:

▪ **ראש הרשימה** מצביע ל-frame הפניו הבא.

▪ נשתמש בבייטים העליונים של כל frame לשימור המצביע לאיבר הבא בראשימה (למעט

ה-head שישתמש ב-frame שלם). בכך חוסכים את המקום החדש עבור תחזוקת

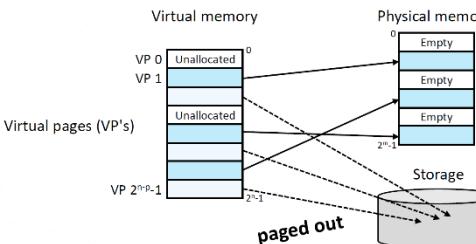
הרשימה המקוורת.



## פתרונות עבור המצב בו לא נמצא דף פיזי פנוי

- מערכת הפעלה תיקח דף מטהlixir שברגע לא רץ ותשמר אותו ב-storage (hard disk) storage (RAM) עבור התהיליך שרצ ברגע (טהlixir דומה ל-preemption).
- התהיליך נקרא **paged virtual memory**.

## Paged Virtual Memory



כל דף וירטואלי יכול להיות באחד מ-3 מצבים:

1. הדף אינן חלק ממוחב הכתובות -> אין לו data והוא לא ממוקם.

2. הדף ממוקם ב-frame (פיזי) שמכיל את ה-data שלו.

3. הדף הווירטואלי לא ממוקה לדף פיזי, אבל הוא נמצא בתיקון אחסון (storage) -> הדף הווירטואלי נמצא מקום בו ה-data שלו נמצא באחסן.

גודל מוחב הכתובות של התהיליך חסום בגודל הזיכרון הפיזי + התקני האחסון (storage)

המיופיע מדף וירטואלי לדף ב-storage מונוה ע"י תוכנה (מערכת הפעלה) בשונה מהמיופיע מכתובות וירטואלית לכתובת פיזית שמבוצעת אוטומטית ע"י המעבד (מנוהל ע"י חומרה).

## page out

זהו התהיליך המבוצע כאשר התהיליך צריך frame, אך אין frame פנוי שניתן להקצתות לו.

זה בעצם מעבר מה-list inactive ל-free list.

במקרה זה יבוצע page out של חלק מהframes storage, בכספי לפנותם ל-storage, ולהיליך שביקש frame לא היה אחד פנוי להקצתות לו.

## לאן הדפים הללו ילו?

אם מדובר בדפים שהם חלק מ-VMA מסווג **file backed** אז הם ילכו לקובץ המתאים להם.

אם מדובר בדפים שהם חלק מ-VMA מסווג **anonymous** אז הם ילכו למקום המוגדר מראש בזיכרון עborך – swap files partitions.

איך מערכת הפעלה מחליטת לאילו דפים לעשות page out? בעזרת אלגוריתם **page replacement**?-> אין לו מיפוי בזיכרון הפיזי ולכן אם מגיע פault גשת אליו ותקבל fault.

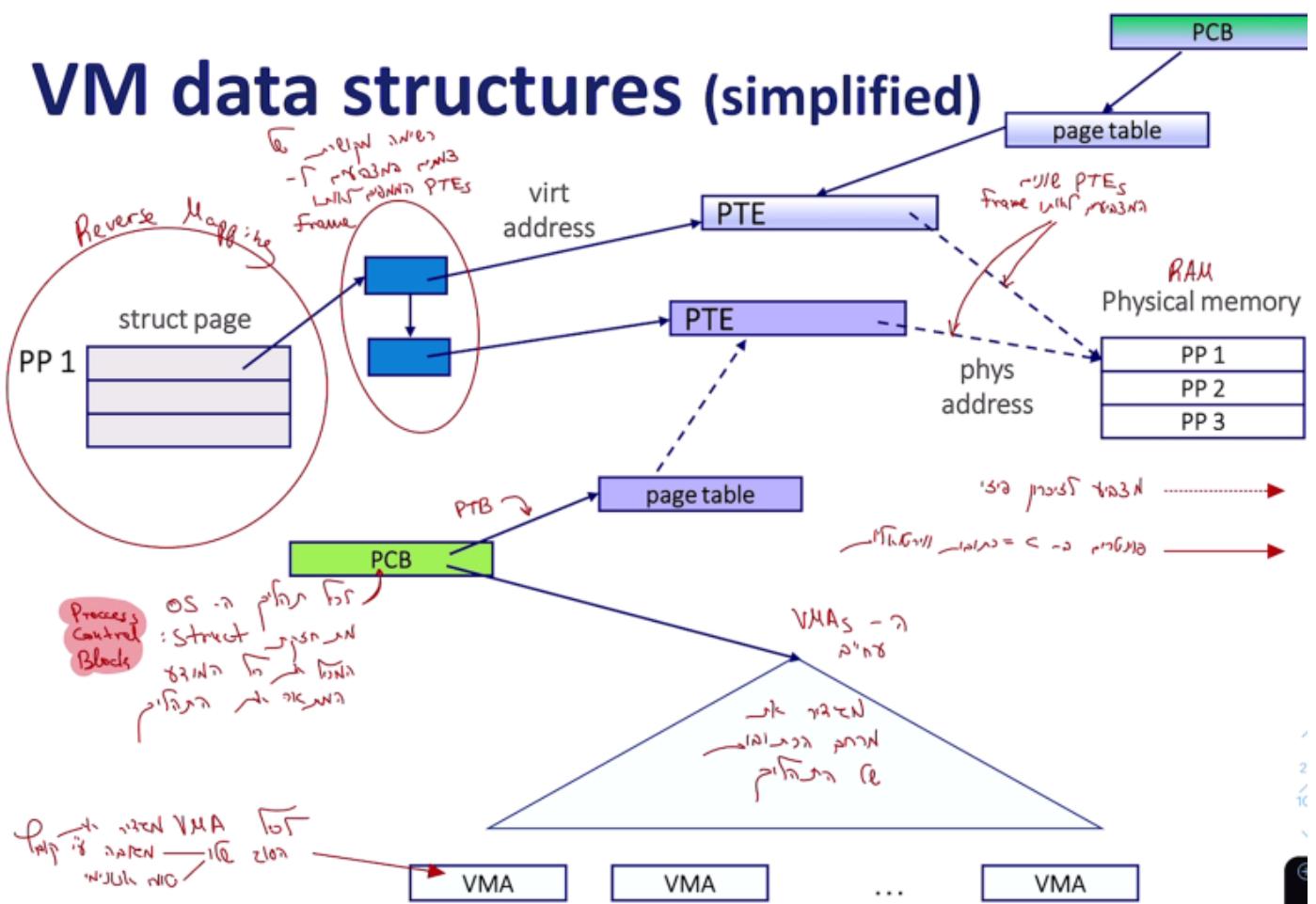
במצב זהה מערכת הפעלה תכנס למצב: **page in**.

- מצב התחלתי: קיים frame ממופה וידוע לאן הוא צריך ללבת באחסון.
- kernel יבצע את השלבים הבאים:

  - 1. מצוא את כל המיפויים** של כל התחלים במרחב המערכת שמופיעים לדף שהולכים לבצע לו **page out**.
  - 2. צריך לסמן את כל המיפויים** שמצאים ב-1 כ-**invalid**. הסימון מורכב מ-2 חלקים:  
 (א) לעדכן את כל המיפויים הרלוונטיים ב-**PTEs** שמנפים לאותו הדף = **אייפוס של ה-bit valid**.  
 (ב) בוצע בעזרת מבנה נתונים הנקרא **reverse mapping** **invalidating the TLB**

## Reverse Mapping

- מבנה נתונים שה-kernel מתחזק בזיכרון = **מערך של struct-ים (של C)**.
- struct page** אחד לכל **frame** בזיכרון הפיזי של המחשב.
- ב-Linux struct זהה נקרא **struct page**.
- לכל struct** במערכת יש מצביע אל רשומה של כתובות של כל ה-**PTEs** במערכת שמצויעים אל ה-**frame** שאותו ה-**struct page** מייצג.
- בכל פעם שמתווסף מיפוי חדש ל-frame אוזי ה-PTE שמנפה לאוטו frame מתווסף רישימה של ה-**struct page**.
- באופן אנלוגי, כאשר מיפוי של page מושמד (מסומן כ-**valid=0**/משנים את ה-**frame** אליו הוא מצביע) -> הכתובת של ה-**PTE** שהושמד נמחקת מהרישימה של ה-**struct page**.



פעולה המחדירה את ה-data של דף שהועבר לאחסן לתוך frame בזיכרון הפיזי.

- דף שהוא out יסמן בתור invalid ב-PTEs.

- לבן, גישה אליו תגרור page fault.

- כדי לטפל ב-page fault, ה-page handler ב-kernel צריך:

1. להקצות frame

2. להעתיק אל ה-frame את ה-data של ה-page מהאחסן (storage).

3. לעדכן את ה-PT של התהיליך כך שיצביע אל ה-frame החדש.

- איך ה-page handler יודע באחסן נמצא נמצא ה-data של הדף שהוא out ?

1. מוחפש את ה-VMA של ה-VA שגורם ל-fault.

2. אם הוא לא קיים (אוון טווח שמכיל את ה-VA) -> הורג את התהיליך (SEGFAULT).

3. אם הוא קיים -> בודק האם הוא out (ולא סתם דף שלא מוקצה עבורי זיכרון פיזי):

אם כן -> מבצע עבורי in :

(a) אם הוא file-backed – מוכניס את ה-data שלו לזכרון הפיזי (ידע איפה ה-data נמצא כי הוא backed).

(b) אם הוא anonymous – מאפס את ה-data שלו (עבור VMAeusame הביטים מאותחלים ל-0).

- איך ה-page handler יודע הדף הוא out ?

- לאחר מכן צורך בביטים התחטונים של PTE שהוא invalid מארח והוא לא ממופה לדף פיזי אדי ניתן לנצל אותם בשבייל

לසמן שהדף הוא paged out .

- זה מזכיר רק ביט אחד, لكن יש עוד ביטים "מיותרם" – אותו ננצל בשבייל לשומר

מצביע ל-struct שיכיל ביט המורה על כך שה-page אכן paged ואת הכתובת שלו באחסן.

- ה-struct הנ"ל ייווצר בעת out page ל-frame ויעודכו המצביעים אליו מככל ה-PTEs שמצוין על ה-frame (ניתן למצוא אותם בעזרת ה-struct page).

- בכך, אם כמה דפים ממופים באותו דף פיזי שהועבר לאחסן, אזי כאשר נחריר בחזרה את הדף לזכרון זה יעדכן עבורי כל המיפויים שלו.

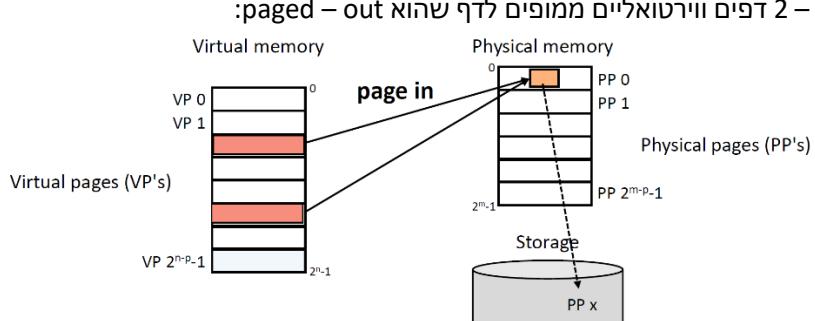
- אם לא היו שומרים בביטים הנоторרים מצביע ל-struct page fault לשיהה משותף לכל המיפויים אלא את הכתובת הפיזית באחסן ->

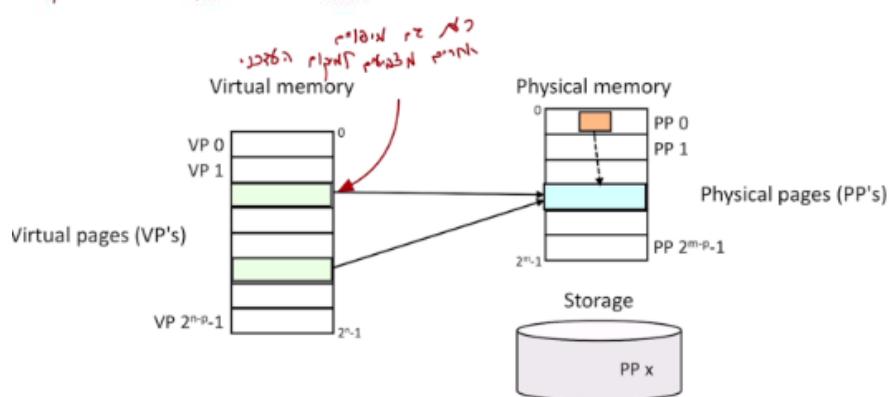
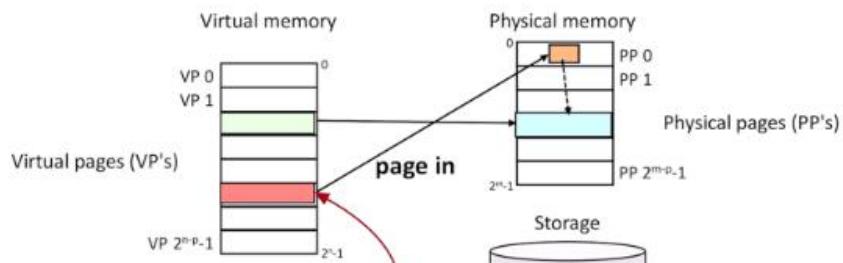
הינו מקבלים aliasing: כאשר ההינו מקבלים page fault עבור אחד המיפויים של הדף איזה הינו עושם לדף in

שהיה מעדכן את המיקום של הדף רק עבורי המיפוי שגרם ל-fault, בעוד שאר המיפויים היו עדין מורים על שהדף paged out .

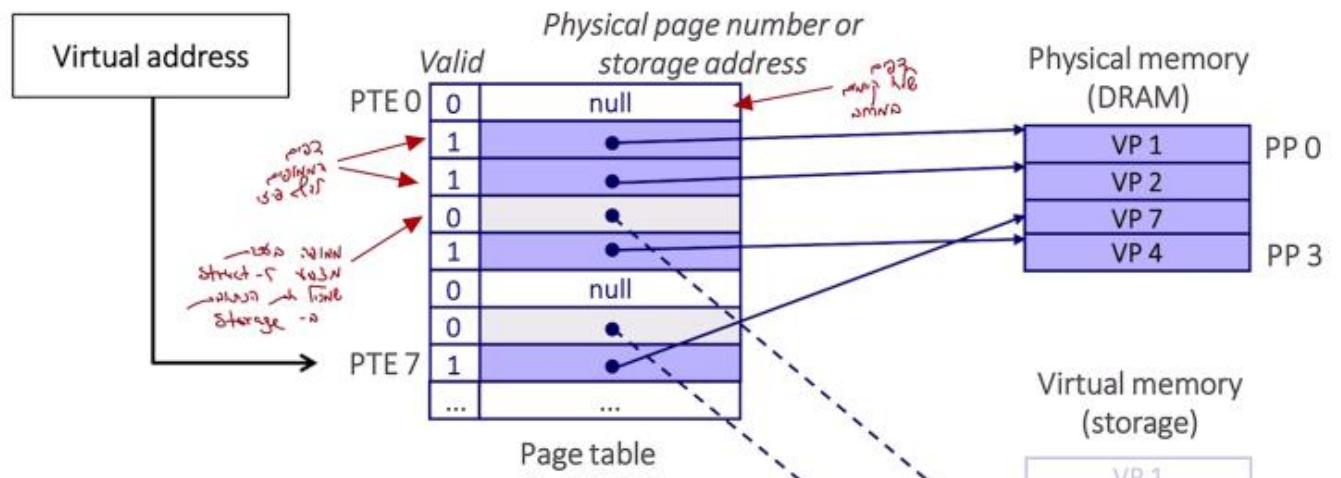
- דוג' לאיך זה יתבצע:

מצב התחלתי – 2 דפים ווירטואליים ממופים לדף שהוא out – :paged





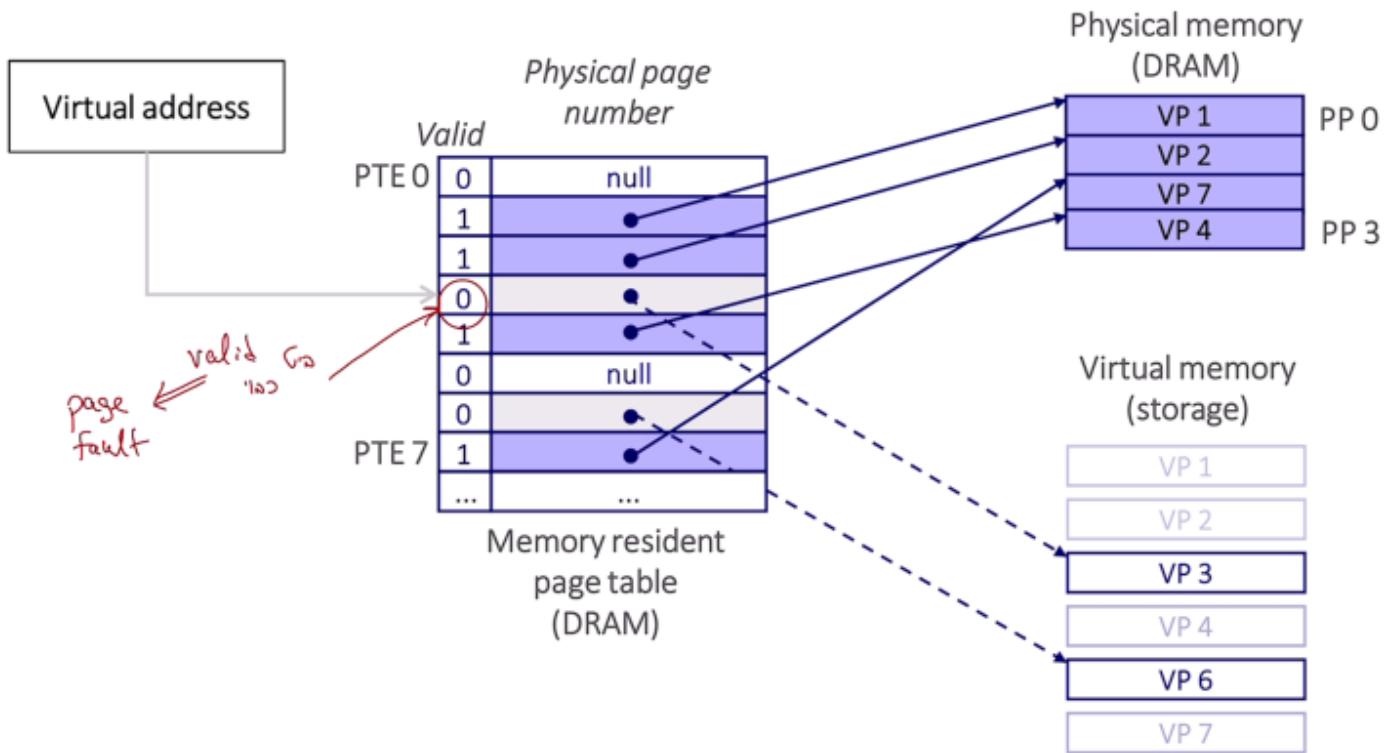
## Page Hit Flow



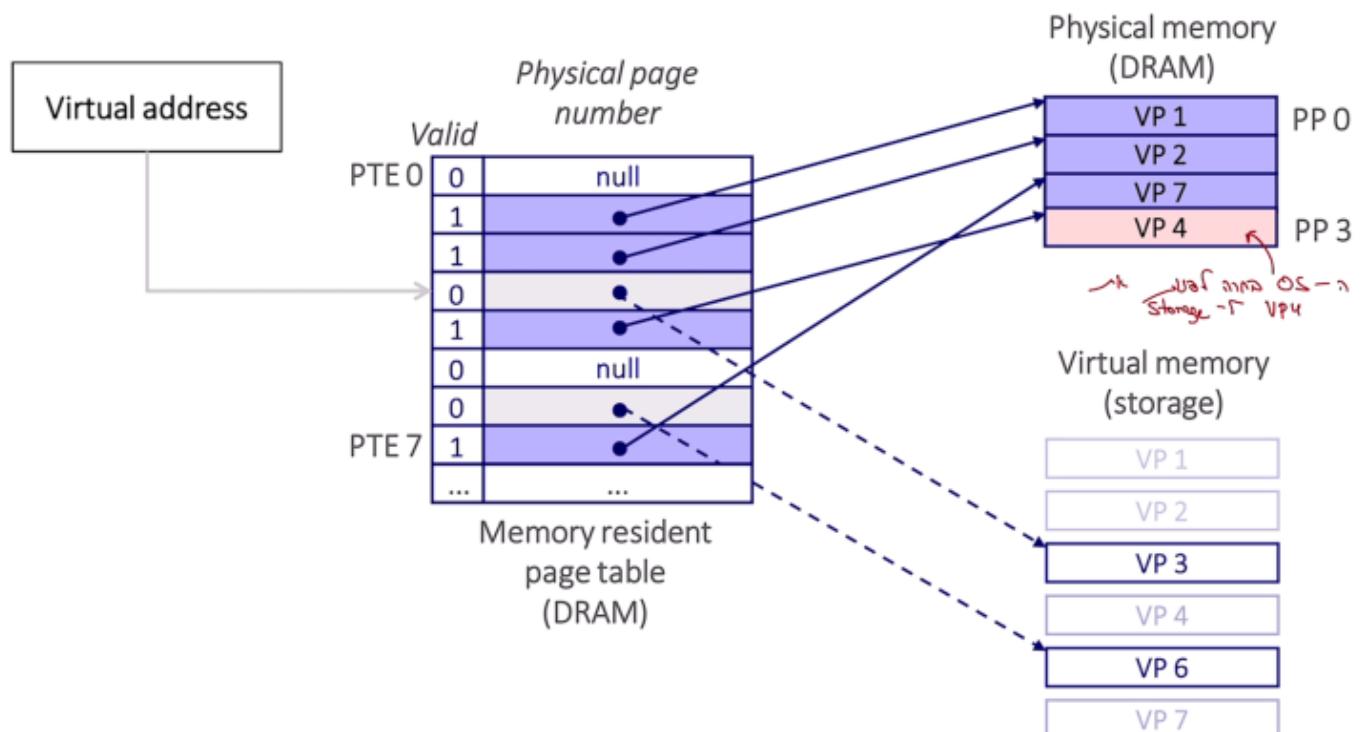
**Example:** Page size: 4 KiB

|                       |          |                        |          |
|-----------------------|----------|------------------------|----------|
| Virtual address:      | 0x00740b | Physical address:      | 0x00240b |
| Virtual page # (VPN): | 0x007    | Physical page # (PPN): | 0x002    |

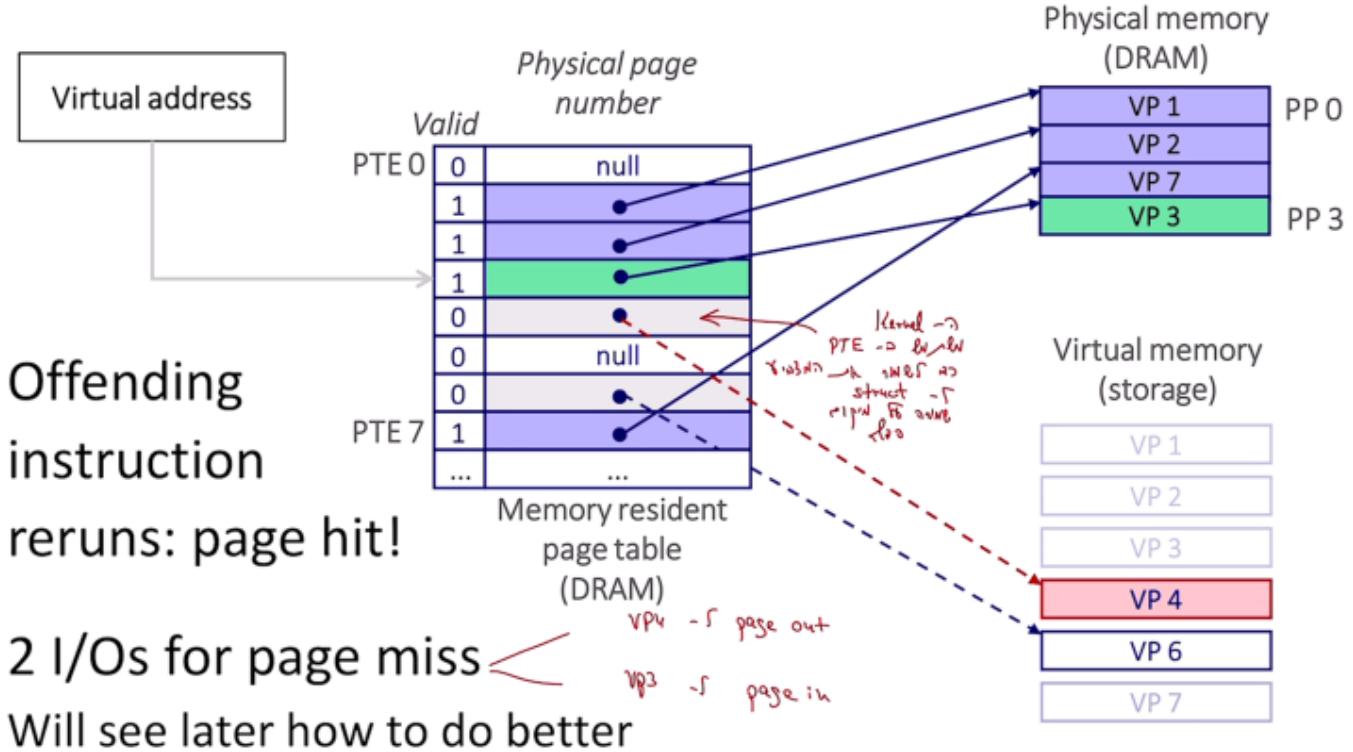
## Page miss causes page fault (exception)



## OS selects victim to evict (VP 4)



## OS evicts victim (VP 4)



### העדרת דף בוחרת איזה דף לפנות ל-storage – Page Replacement

- פתרונות יכול להיות אחד מבינן:

  - פתרון גלובלי – בחירת דף מבין כל הדפים האפשריים (גם דפים של תהליכים אחרים).
  - פתרון מקומי – בחירת דף מבין דפים השוכנים בתהילך הנוכחי בלבד.

יש קשר הדוק בין בעיית page replacement לבין בעיית caching (cache replacement). המשאב המהיר הוא האזכור בעבר והאטי שיעושים לו caching הוא האחסון.

במקרה של page replacement: המשאב המהיר הוא באחסון ובאשר מבאים אותם לדיברונו, זה שקול לעשות להם caching.

המטרה: להביא למינימום את miss rate :

$$\text{miss rate} = \frac{\# \text{misses}}{\# \text{accesses}}$$

שקלול להביא למינימום את hit rate :

$$\text{hit rate} = \frac{\# \text{hits}}{\# \text{accesses}} = 1 - \text{miss rate}$$

הูลות הממוצעת של ביצוע גישה: (AMAT) : Average Memory Access Time

$$\text{AMAT} = \text{hit rate} * \text{Memory Access Time} + \text{miss rate} * \text{Storage Access Time}$$

- מדיניות FIFO – לפנות את הדף שהוכנס ל זיכרון ה כי מזמן מסתמנת במדיניות לא עיליה.
- נרצה למשוך policy המנצל את עקרון הлокליות בזמן: דף שניגשנו אליו בעבר-> בנסיבות ניגש אליו שוב בעתיד הקרוב.

**locality**

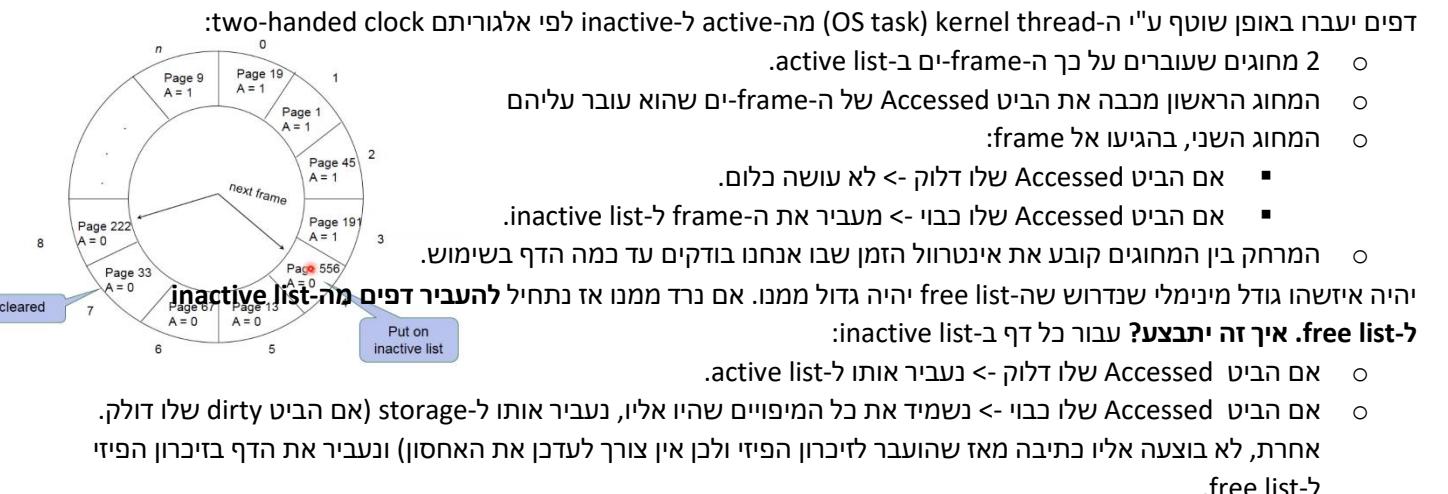
- נגידר את ה-(T) set להיות קבוצת הדפים שניגשנו אליהם ב-T גישות האחרונות.
- עקרון הлокליות מניח ש- T << | working set(T)|
- (קבוצת הדפים שניגשנו אליהם ב-T גישות האחרונות קטנה ממש מ-T).
- כולל 20/80 80% מהגישות ל זיכרון ניגשים ל-20% מהדפים.
- לכן, אם נשמר את אותם 20% נקבל זמני ריצה של זיכרון ולא של אחסון.

**History-based policies**

1. **LRU** – מפנה את ה-page Least Recently Used = הדף שניגשנו אליו הכי רוחק בעבר
2. **LFU** – מפנה את ה-page Least Frequently Used = הדף שניגשנו אליו הכי פחות בעבר
- חשוב לציין שהמדיניות החלפה של LRU אינה חסינה ל-Cyclic scan: מצבים בהם הסריקה של הזיכרון מתבצעת בצורה מעגלית על מידע שגדלו מגודל ה-cache שלו. קיימים אלגוריתמי scan resistance שמתחזדים עם בעיה זו.

**איך נמשך את מנגנון ה-RLRU?**

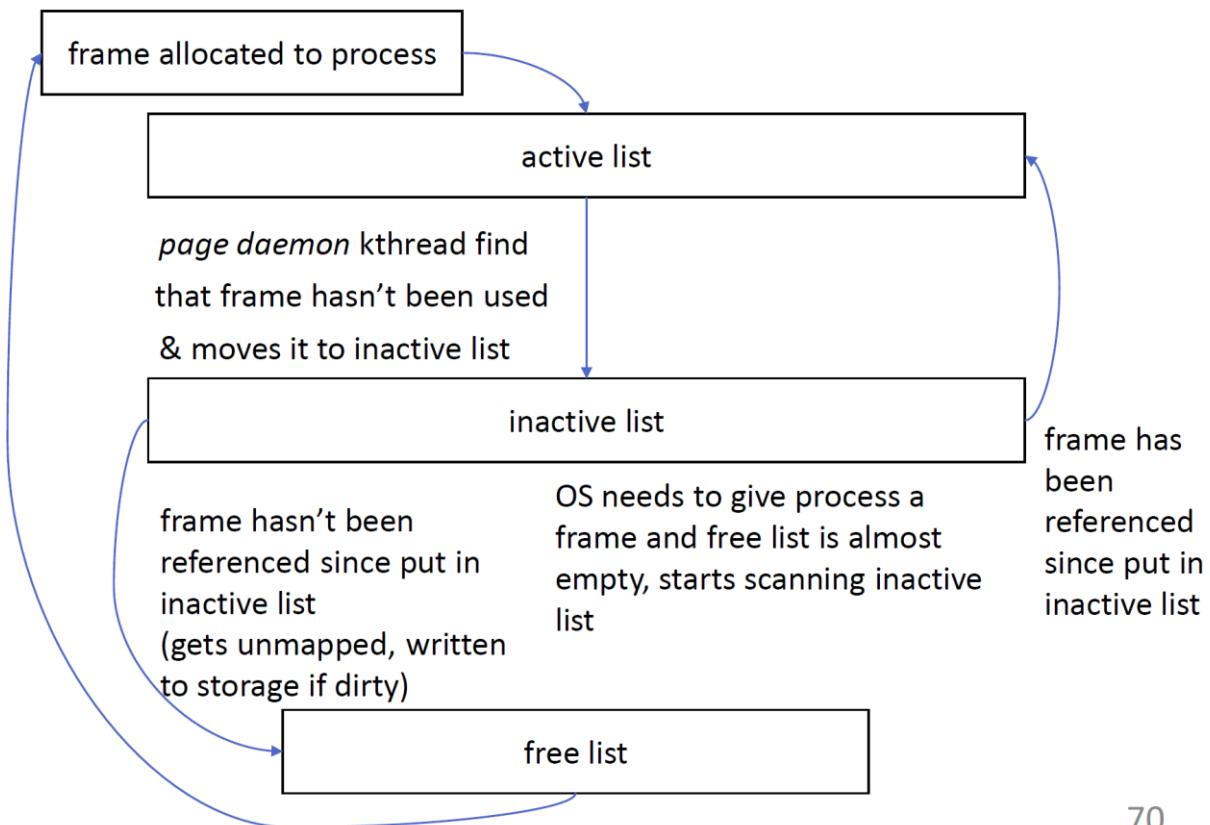
- במקומות לתחזק מבנה נתונים שישמר את המידע הרלוונטי עבור כל הדפים (יקר מאד!) נמשך מנגנון שיחלק את הדפים ל:
  1. **active list** - דפים שנמצאים בשימוש תדיר – "דפים חמים"
  2. **inactive list** - דפים שלא נמצא להם שימוש (מקטין את ממות הגישות ל זיכרון הראשי ל-1 באופן קבוע)
  3. **free list** – דפים פנויים שניתנו למפות אליהם (מקטין את ממות הגישות ל זיכרון הראשי ל-2 ביטים לכל דף (**המעבד יתחזק**)): ב כדי להחליט אילו דפים עוברים ל inactive מ activeinactive מ necessity need to update again. המעבד בכל פעם שמתבצעת קריאה או כתיבה לדף (במידה והביט עדין לא דלוק).
- **Accessed** – יודלק ע"י המעבד בכל פעם שמתבצעת קריאה או כתיבה לדף (במידה והביט עדין לא דלוק). מודיע על כך שניגשנו לדף.
- 2. **Dirty** – יודלק ע"י המעבד בכל פעם שמתבצעת כתיבה לדף (במידה והביט עדין לא דלוק).



**ל-free list. איך זה יבוצע?** עברו כל דף ב inactive list:

- אם הביט Accessed שלו דלוק -> לא עושה כלום.
- אם הביט Accessed שלו בביי -> מעביר את ה inactive listframe ל active listframe.
- המרחק בין המוחגים קבוע את אינטראול הזמן שבו אנחנו בודקים עד כמה הדף בשימוש.
- יהיה איזשהו גודל מינימלי שנדרש מה free list והוא גדול ממנו. אם נרד ממנה אז נתחילה **להעביר דפים מה inactive listframe**.

# Life of a frame



70

- swapping – ביצוע page out למרחב בתובות של התהילך מסויים.
  - מתרחש כאשר יש צורך לפנות הרבה זיכרון ומורה.
- trashing – מצב שקרה כאשר יש 2 פרוטוסים שרצים במקביל ואיחוד ה-*working set* שלהם עולה על גודל הזיכרון (main memory) -> כמעט בכל גישה לזכרון יש in page ו- page out .page daemon kernel thread נקרא גם
  - במעטם בכל גישה לזכרון יש in page ו- page out .
- יש כאן trade off בין המקום שמתבצע עבור ה-list free (יכולנו לשמר שם עוד דפים במקום לשומר buffer של דפים פנויים) לבין הגישות האחסון – 2 אם לא היינו משתמשים ב-list free .
- במנגנון המתואר לא נשמר את כל המידע לגבי תכיפות השימוש בדפים שלא בשימוש תדיר, אלא נשמר את המידע רק על אלה שכן בשימוש תדיר (כ-20% מכלל הדפים) .active list = list of pages used

## סיכום ה-workflow ב-page fault עד עבשוי

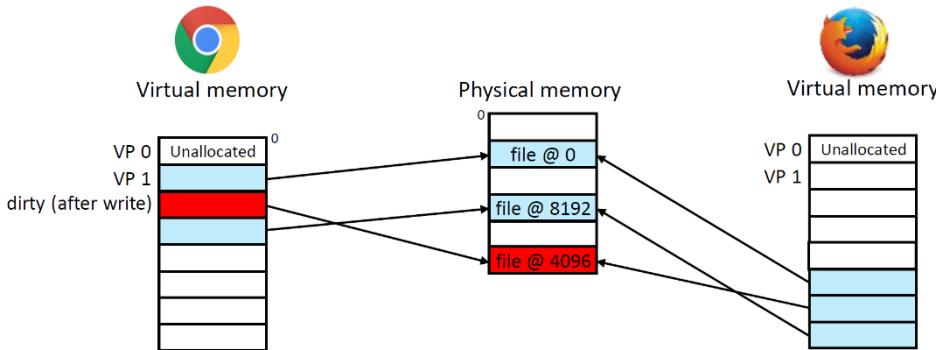
1. התרבעת page fault
  2. ה-*handler* בודק האם הכתובת שגרמה לכך שייכת ל-VMA במרחב הכתובות של התהיליך (SEGFAULT)
- A. אם לא -> גישה לא חוקית -> ה-OS יירוג את התהיליך
  - B. אחרת:
    - i. אם ה-page הוא paged out -> יבוצע in page .hard fault
    - ii. אם ה-page הוא אונוכימי או file backed וגם לא ניגשו אליו בעבר -> יש להקצות לו frame .soft fault
    - iii. אם ה-page הוא page שהטהיליך לא עשה בו שימוש הרבה זמן ולבן ה-page הגיע ל-list free (בחולק מהמעבר ל-list free המיפויים מושמדים) -> יש למפות אותו מחדש .soft fault

- לאחר ויתכן מצב בו 2 תהליכיים שונים ימיפו את אותה בתובת פיזית וינסו לשנות את ערכיה יש ליצור מיפוי שייתן מענה הולם.

### MAP\_SHARED mmap() flag

- מיפוי משותף.
- כל התהליכיים שmapsים את אותו קובץ X יקבלו מיפוי לאותם page-frame-im בזיכרון כך שגם אחד התהליכיים כתוב ל-frame איז כל התהליכיים האחרים יראו את מה שהוא כתוב.
- הכתובות יגעו בסופו של דבר לקובץ באחסון (באשר יוצע *page fault* ל-frame).
- ישamu הפעלה דרך שקובץ שתהילך מסוים רוצה להקצות עבורי frame קיים בבר-*memory*, וכן במקומם להקצות עבורי מקום חדש -> להכיל את הכתובת של ה-*key*.

## MAP\_SHARED



### MAP\_PRIVATE mmap() flag

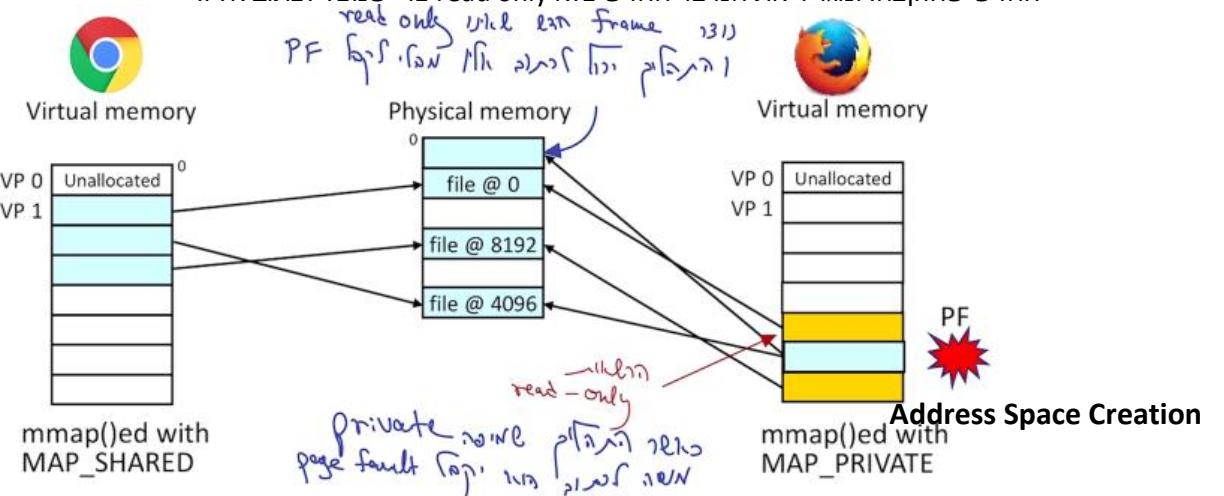
- העדכניםים שתהילך עושים הם פרטיים. תהליכיים אחרים שmapsים את הקובץ לא יראו אותם.
- העדכניםים לא יחללו לקובץ בהתקן האחסון.
- יבוצע בעזרת COW.

### COW – טבונקה עילה לשכפול של משאב משותף

- כל מי שמשתף את המשאב מחזיק איזשהו מצביע אליו יוכל לקרוא מהמשaab.
- כאשר מישחו מנסה לבצע עדכון הוא מקבל עותק של המשaab, שככלו את העדכון בתוכו ומתחילה לעבוד על העותק הפרטיז הזה.
- שאר המשתפים של המשaab לא מודעים לעדכון זהה, כי הוא התבצע על עותק של המשaab.

### איך זה מבוצע בפועל?

1. מיפויים זיכרונו של תהליך ב-*VMA* MAP\_PRIVATE איז כל המיפויים ב-*VMA* מקבלים הרשות גישה של *read only* (בעוד ה-*VMA* יסמן אינם בעלי הרשות *read only* אלא בעלי הרשות *read & write*).
2. כאשר מתבצע ניסיון של כתיבה לאחד המיפויים ע"י התהליך הזה, אז ה-OS מזהה שיש ניסיון כתיבה למוקם בזיכרון שהוא לא *read only* ע"י *VMA* שהמיפוי שלו הוגדר בתו *read only*.
3. ה-OS מייצר עותק של *frame* למקום אחר בזיכרון ומפנה את הכתובת הווירטואלית שניסינו לכתוב אליה אל הזיכרון החדש שהוקצה. ומגדיר את המיפוי החדש כלא *read only* כדי שנוכל לכתוב אליו.



זה בעצם יוצרה של תהליכיים חדשים.

ב-`exec()` אין `syscall` ייחודי לייצור תהליך חדש, אלא:

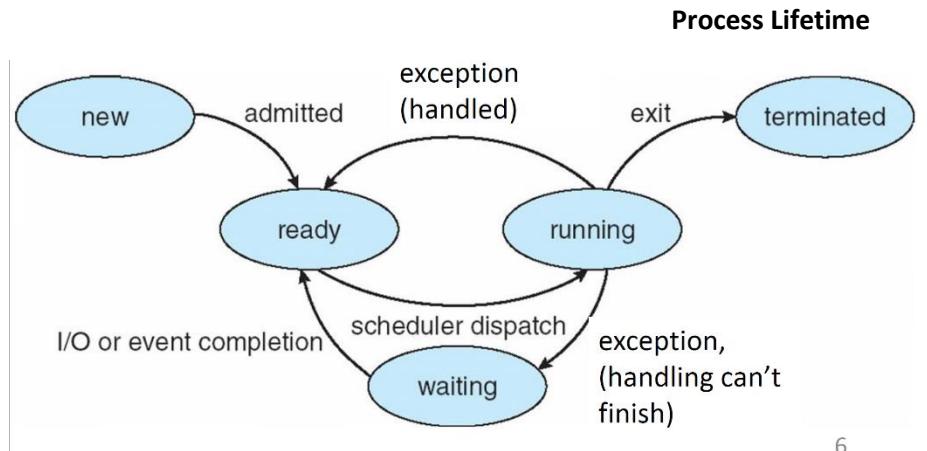
### **fork()** .1

- יוצר שכפול של התהליך הנוכחי (שקרה לו) בדיק באוטו מצב שבו התהליך שקרה ל-`fork` נמצא בו.
- לתהליך שקרה ל-`fork` קוראים `parent`.
- לשכפול החדש קוראים `child`.
- ההבדל היחידי ביןיהם הוא שבילד הקרייה ל-`(fork)` החזירה 0 ואצל ההורה היא החזירה את ה-`fd` של הילד.
- הילד מקבל עותק של כל משאב שיש להורה ובאותו המצב (למשל: קבצים פתוחים בתור קבצים פתוחים).
- ילד יש מרחב כתובות משלו שמשוכפל מהמרחב של ההורה.
- כדי לחסוך את השכפול הנ"ל משתמשים ב-W-COW: כל הכתובות הוירטואליות אצל ההורה והילד יסומנו בתור `read-only`.
- הנ"ל יבוצע רק על מיפויים שלא נוצרו בעדרת `MAP_SHARED` מאחר ואלו אמורים להיות משותפים.

### **exec()** .2

טוען תוכנית חדשה לתוכר מרחב הכתובות של התהליך שקרה לו ועובד להריץ אותה (משמיד את מרחב הכתובות של התהליך הנוכחי) וטוען לתוכו איזשהו `exec` שימושי לו בפרטיטר.

- ה-scheduler בוחר איזה תהליך יהיה התהילך שירוץ על המעבד מבין ה- runnable processes •



6

- new** – תהליך שנוצר ע"י ביצוע (`fork()`) •
- ready** (= **runnable**) – יכול לירוץ על המעבד אם תינטן לו ההזדמנות •
- running** – לאחר שהתהליך יבחר ע"י מערכת הפעלה הוא יירוץ על המעבד •

ברגע שתרחש **exception** העניינים יכולים להתקדם באחת מכמה צורות:

1. **exception** יטופל בצורה מלאה והתהליך יוחזר למצב `ready`.
2. **kernel לא יכול לסייע את הטיפול ב-exception** -> התהליך עובר למצב `waiting` ומচכה לקיום של איזשהו תנאי שיביא לסיום הטיפול ב-exception (למשל: `(sleep)`, ביצוע פעולה O/I, למשל: המתנה להחיצה של מקש) לאחר שתיפול ישתיים ושבור למצב `ready`.
3. **terminated** – התהליך מסיים את חייו. יכול לקרות מכמה סיבות. למשל: call system call של `exit` או גישה לבתובת לא חוקית.

## #1 - process scheduling

1. המודול צריך לבצע החלטה ייחודית על סדר הביצוע של התהליכים. כלומר, **כל התהליכים מגיעים בפעם אחת** (כולם ביחד) והמודול מחליט על סדר הביצוע ביניהם.
2. כל תהליך הוא **sop** שרח **עד שהוא מסתיים** (אין תהליכים שרוצים בולולאה אינסופית).
3. כל **sop** מבצע רק **чисובים** (רח על ה-CPU בלבד לבצע **calls** system calls)
4. **זמן הריצה** של תהליך ידוע מראש.

## #1 - scheduling metric

- המדד העיקרי הוא **quality of service** שמיידד בעזרת **turnaround time** (זמן שעובר מהגעת job עד סיום הריצה שלו) •
- $T_{turnaround} = T_{complete} - T_{arrival}$  •
- אנו מניחים שככל התהליכים מגיעים ביחד:  $0 = T_{arrival}$ . מכאן נסיק:  $T_{turnaround} = T_{complete}$

## algoritmitim shonim l-process scheduling

### FCFS (= First Come First Served)

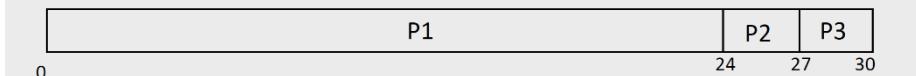
מי שmagiu מוקדם יותר זוכה לרווח על המעבד מוקדם יותר (למרות שהחנכו שכולם מגיעים ביחס אחד ייש בינהם סדר)

| Job | CPU time | Job | CPU time | Turnaround time | Waiting time |
|-----|----------|-----|----------|-----------------|--------------|
| P1  | 24       | P1  | 24       | 24              | 0            |
| P2  | 3        | P2  | 3        | 27              | 24           |
| P3  | 3        | P3  | 3        | 30              | 27           |

Arrival Order:

P1, P2, P3

Average turnaround time:  $(24+27+30)/3=27$



הבעיה של אלגוריתם זה הוא אפקט השיריה

אפקט השיריה = convoy affect – ג'ובים קצריים נתקעים אחרי ג'וב ארוך למרות שהם לא צריכים את המעבד להרבה זמן.

### SJF (= Shortest Job First)

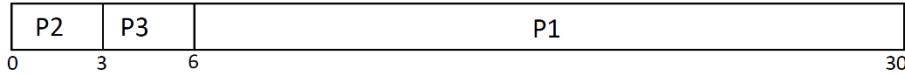
מי ש.short יותר זוכה לרווח על המעבד מוקדם יותר.

| Job | CPU time | Job | CPU time | Turnaround time | Waiting time |
|-----|----------|-----|----------|-----------------|--------------|
| P1  | 24       | P1  | 24       | 30              | 6            |
| P2  | 3        | P2  | 3        | 3               | 0            |
| P3  | 3        | P3  | 3        | 6               | 3            |

Arrival Order:

P1, P2, P3

Average turnaround time:  $(30+3+6)/3=13$



עבור ההנחה שהחנכו האלגוריתם זהה נותן את התוצאה האופטימלית.

### מודול חדש - #2

בעת כניסה את ההנחה שכל הג'ובים מגיעים ביחד.

בעת-SJF אינם בשיא יעילותו, מאחר וѓובים קצריים שmagiuו תז"כ ביצוע של ג'ובים ארוכים צריכים להמתין עד סיום הג'ובים הארוכים.

כדי להתמודד עם בעיה זאת משתמשים ב-preemption – עצירת ג'וב במהלך ביצועו לטובת ביצוע ג'וב אחר, ולאחר סיום העבודה של הג'וב الآخر חוזרת לאווצה נקודה בה הפסיקו את הל'וב הראשון.

### STCF (= Shortest Time to Complete First)

התאמאה של SJF למערכת עם preemption.

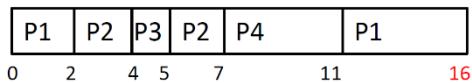
הרעיוון להשתמש ב-preemption כדי לאפשר לאלגוריתם לתקן החלטות קודמות שלו.

בכל נקודה בזמן נרץ את הג'וב הנוכחי לו הכי מעט זמן לרווח.

למשל:

| Job | Arrival time | CPU time |
|-----|--------------|----------|
| P1  | 0            | 7        |
| P2  | 2            | 4        |
| P3  | 4            | 1        |
| P4  | 5            | 4        |

**STCF: Any time job arrives, determine which known job has least time left, and schedule it**



Average turnaround time:

$$(16+5+1+6)/4=7$$

| Time | Scheduler Decision                               |
|------|--------------------------------------------------|
| 0    | Run P1                                           |
| 2    | P1 has 5, P2 has 4. Preempt. Run P2.             |
| 4    | P1 has 5, P2 has 2, P3 has 1. Preempt. Run P3.   |
| 5    | P3 finishes. Shortest remaining time P2. Run P2  |
| 7    | P2 finishes. Shortest remaining time P4. Run P4. |
| 11   | P4 finishes. Run P1.                             |
| 16   | P1 finishes.                                     |

## #2 - scheduling metric

– כמה זה עובר עד שהג'וב מתחילה לרווח.

$$T_{\text{Responsive time}} = T_{\text{first run}} - T_{\text{arrival}}$$

במטריקה הזאת STCF מביא תוצאות רעות.

## Round robin (“time slicing”)

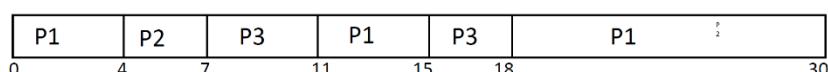
- נתן לבל job לרוח במשך איזשהו time slice (נקרא גם **quantum**).
- לאחר שפרק הזמן נגמר, נעצור את ה-job וביתן לscheduler להחליט איזה job להריץ עבשו.
- האלגוריתם מחלק את זמן הריצה ב策ורה שווה, כך שבאשר job preempted הוא האלגוריתם מעביר להריץ את ה-job הבא אחרי.
- ומעביר את ה-job שנעוצר לסוף התור (**FIFO**).

## Round robin example

| Job | CPU time |
|-----|----------|
| P1  | 20       |
| P2  | 3        |
| P3  | 7        |

All jobs arrive at time 0  
Quantum = 4

| Job | CPU time | Turnaround time | Total waiting time |
|-----|----------|-----------------|--------------------|
| P1  | 20       | 30              | 10                 |
| P2  | 3        | 7               | 4                  |
| P3  | 7        | 18              | 11                 |



Average turnaround time:  $(30 + 7 + 18) / 3 = 18.333$

Average response time:  $(0 + 4 + 7) / 3 = 3.666$

- נוריד את ההנחה שג'וביים משתמשים רק ב-CPU ולא מבצעים קריאות למערכת.
- כאשר ג'וב מבצע system call הוא מסמן שהוא מוגדר על המעבד עד לסיום ביצוע ה-system call (yielding the CPU).
- במקרה להפסיק את זמן ההמתנה עד לסיום ביצוע ה-system call נקבעים עבודה של המעבד על ג'וב אחר (זמן בו המעבד פנוי ואך תהליך לא מנצל אותו)



#### מודל חדש - #4 = העולם האמיתי

- נוריד את ההנחה שמערכת הפעלה יודעת מתי כל ג'וב צריך להסתיים.
- ביחד עם הופעת ההנחה הנ"ל, נפסיק לדרוש לתהליכיים האלה ג'וביים, מאחר ולא ידוע מתי יסתוימו אם בכלל.

אלגוריתמים להתמודדות עם תנאי העולם האמיתי

#### priority based scheduling

##### #1 - Multi-Level Feedback Queue (MLFQ)

- תהליכיים שצרכיהם את המעבד לזמן קצר אחרי הם קוראים ל-system call (bursting) וקבלו עדיפות גבוהה.
- תהליכיים שצרכיהם את המעבד יותר זמן מאשר הקוונטים יקבלו עדיפות נמוכה.

איך זה עובד בפועל?

תור בשלבים

כל התהליכיים יתחלו בשלב הראשון (עדיפות הכי גבוהה)

כל תהליך שצריך יותר זמן מהקוונטים יורד בשלב

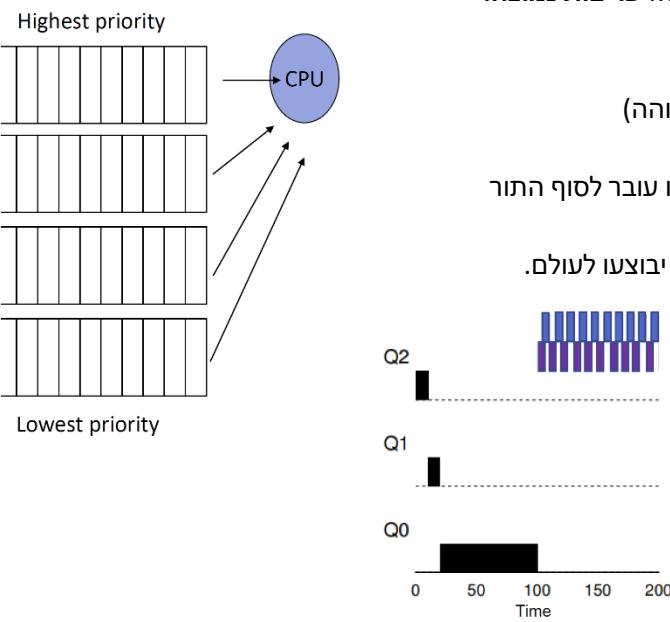
בכל שלב עובדים בתכורת FIFO – מי שסיים את תורו עבר לסוף התור

הבעיות:

1. **starvation:** יכול מצב בו תהליכיים בעדיפות נמוכה לא יבוצעו לעולם.

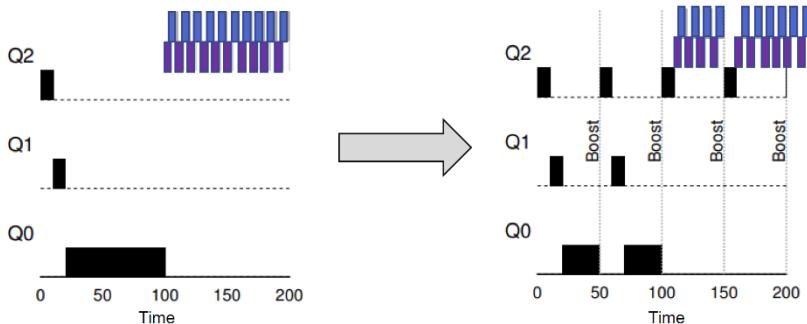
אפשרי כאשר יש הרבה תהליכיים ברמות

הגובהות:



2. תהליכיים ברמת עדיפות נמוכה שנעשים bursting= interactive (bursting) לא יכולים לעלות בחזרה ברמת עדיפות שלהם.

## #2 - Multi-Level Feedback Queue (MLFQ)

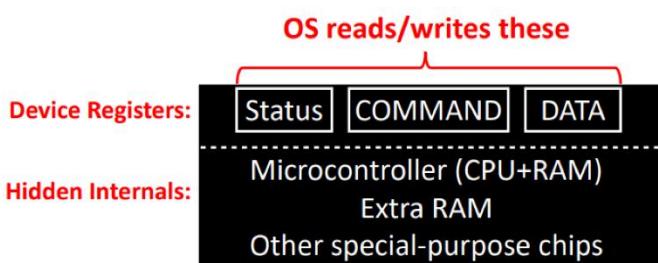
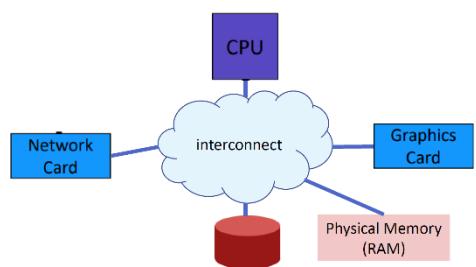


- אותו עקרון כמו קודם רק שינויים priority boost:priority boost לאחר תקופה זמן מסוימת, גוברת את כל התהיליכים לרמת העדיפות העליונה.
- בעיה: תחילן זהני יכול לرمות את ה-scheduler ע"י ביצוע call סתמי רגע לפני תום הקווטנו.

## #3 - Multi-Level Feedback Queue (MLFQ)

- מקום להגדר שתהילך נשאר ברמת העדיפות שלו אם לא מימש את הקוונטים שלו עד הסוף, נגדיר זמן מעבד שמותר לתהילך לנצל בכל רמה. בתום הזמן שהוגדר נוריד אותו רמה.
- בכיה נוכל לגוזם למצב שבו תהיליכים שהם באמת burst (למשל: מנצלים וקעשרה מהקוונטים) ישחוו לשאר יותר זמן ברמת העדיפות העליונה לעומת תהיליכים שהם CPU intense (ומנצלים את רוב הקוונטים).
- responsive time טוב לתהיליכים אינטראקטיביים (bursting).CPU intensive. הוגנות מבחינת ניצול זמן המעובד לתהיליכים שהם זה האלגוריתם שמשתמש בו ביום.
- סיכון של אופן פעולת האלגוריתם:
  - כל תהיליכים יתחלו בrama הראשונה (עדיפות הכיוון).
  - מוגדר זמן מעבד שמותר לתהילך לנצל בכל רמה. בתום הזמן שהוגדר נוריד אותו רמה.
  - בכל שלב עובדים בתכורת FIFO – מי שסיים את תורו עבר לסוף התור.

- דוגמאות להתקנים: disk, network, keyboard, mouse, video.
- התקנים הם רכיבים המופרדים מביינית המעבד -> הhardware עובד בצורה עצמאית.
- המעבד יכול לתקשר עם התקן (בתוכה מתוכנה שרצה על המעבד ובקשת זאת).



### איך עובד החיבור בין המעבד להתקנים?

- **interconnect** – רשת פנימית המחברת בין ה-CPU והתקנים. מאפשר למעבד לתקשר עם התקנים והזיכרון הראשי (RAM).

### Device hardware interface

- לכל התקן יש רגיסטרים.
- הרגיסטרים הם ממשק תקשורת של התקן עם המעבד.
- הם שונים מהרגיסטרים של המעבד בכך שהם לא מהווים את המצביע של התקן, והתקן יכול לשמר את המצביע הפנימי במקומן נסתר.
- מאחורי הרגיסטרים יושבת החומרה של התקן ומטפלת בגישהות שימושית המעבד דרך הרגיסטרים.
- בד"כ הגישות של המעבד לרגיסטרים של התקן הן הטורוג לשינוי המצביע הפנימי של התקן.

### תקשורת בין המעבד לתקן לצורך קריאה וכתייה

- קיימים 2 סטימס של פקודות מכונה עבור צורך זה:
  1. port mapped IO (PMIO)
  2. Memory mapped IO (MMIO)

### port mapped IO (PMIO)

- פקודות מכונה בדומה ל-load/store שנקראות in/out
- הפרדה בשם נועדה בשביל להבדיל את מרחבי הכתובות (רגיסטרים של המעבד לעומת רגיסטרים של התקן)
- בפועל, השיטה הזאת פחות נמצאת בשימוש כיום.

### Memory mapped IO (MMIO)

- במקום להקצות מרחב בתובות מיוחד עבור רגיסטרים של התקנים, החומרה תשתול את הרגיסטרים של התקנים למרחב הכתובות הפיזיות.

בשידורנו בשיעורים הקודמים על הגישה של המעבד ל זיכרון הראשי לא דיקנו בפרטם:

- במקום שהמעבד יגיש ישירות ל זיכרון הראשי

interconnect הוא עבר דרך ה-PCI

- הזיכרון נשלף מהמקום המתאים: זיכרון של התקן/ זיכרון הראשי

כלומר, לא כל כתובות פיזיות שייכת ל זיכרון הראשי

המחשב בונה בתהליך boot את מיפוי הכתובות הפיזיות ל זיכרות (DRAM/ התקנים)

דוגמא למיפוי מלינוקס:

```
$ cat /proc/iomem
00000000-0000ffff
00001000-0009bfff
0009c000-0009ffff
000a0000-000bffff
90000000-c7ffbfcc
91a00000-91a0ffff : tg3
91c00000-91dfffff : PCI Bus 0000:03
91d00000-91d0ffff : megasas: LSI
100000000-407fffffff : System RAM
```

Access to 0x91a00100 will go to  
register 0x100 on the tg3 device

- המטרה: קוד קרNEL יהיה גנרי ולא יצטרך להזכיר את המשק החומרתי של כל התקן.
- מודול של מערכת הפעלה שתפקידו לעבוד מול התקן.
  - הדרייבר בעצם יהיה ערוץ התקשות (מתווך) שלנו מול התקן.
  - הוא יהיה זה שיבתוב לריגיסטרים של התקן.
  - 70% מהקוד בLINукס הוא קוד של device drivers

### 3 סוגי של device drivers .1

#### character device driver

- אבסטרקציה של זרם של נתונים.
- אפשר לקרוא/לכתוב מההתקן אבל אי אפשר לראות מה הערך הנוכחי. למשל: מקלחת, עכבר.

```
struct file_operations {
 struct module *owner;
 loff_t (*llseek) (struct file *, loff_t, int);
 ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
 ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
 ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
 ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
};
```

- אם יש פונקציות שההתקן לא יכול למשוך (llseek) ב"ב תיה אחות בזאת) אז התקן יחזיר קוד שגיאה.

#### block device driver .2

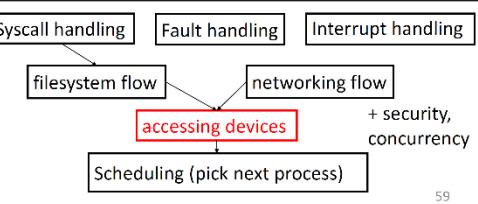
- אבסטרקציה להתקן עם מרחב בתובות שאפשר לכתבו/ולקרוא אליו. למשל: התקני אחסון (Disk, SSD)
- אפשר לקרוא/לכתוב ל-offset offset כלשהו בהתקן.

#### גרנולריות = בלוקים של נתונים.

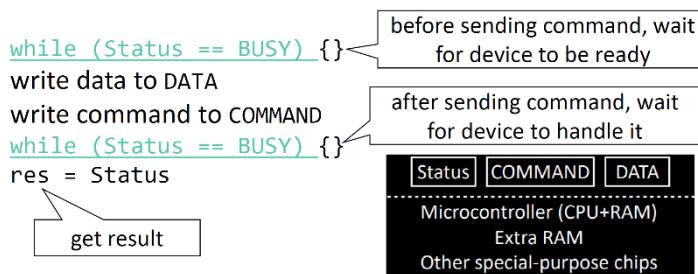
- הדרייבר מספק פונקציית request שמקבלת רשימה של בלוקים שרוצים לקרוא/לכתב אליהם ומבצעת זאת.
- כל בקשה request כוללת:

- ביזון: קריאה/כתיבה
- בתובות: איפא אמרור להיות ה-data-data בתוך התקן
- אורך: כמה data רוצים לקרוא/לכתב
- OS: מכיל את ה-data במקורה שרוצים לכתב להתקן או שהוא היעד של ה-data במקרה שרוצים לקרוא מידע מההתקן.

#### network cards .3



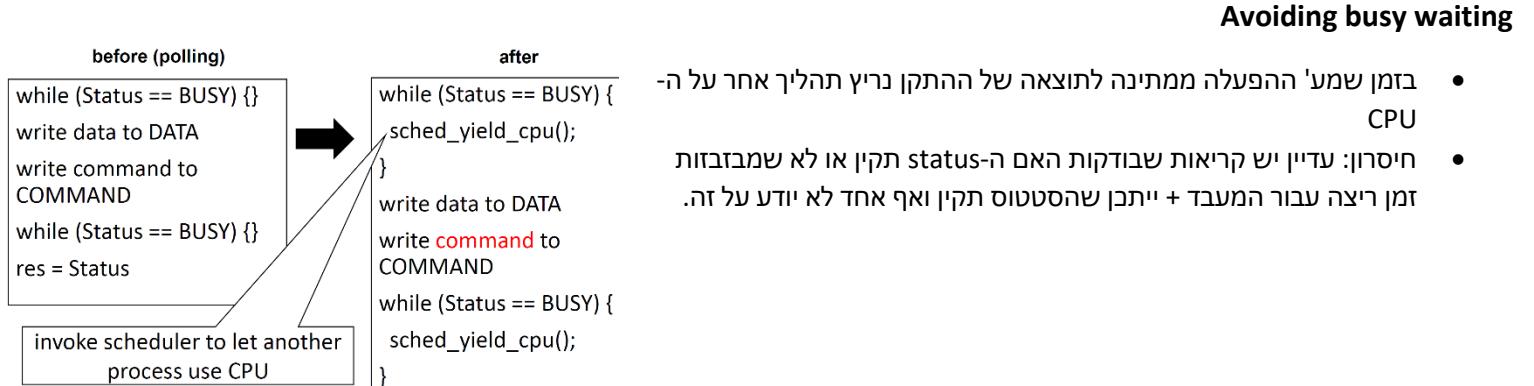
- אין מערכת הפעלה, או ליתר דיוק ה-device driver, יודעת שההתשובה שחייבינו לה בהתקן בבר מוכנה (אחרי שביצעו פעלת שמחייבת לתשובה).
- מס' גישות אפשריות:



### polling the device

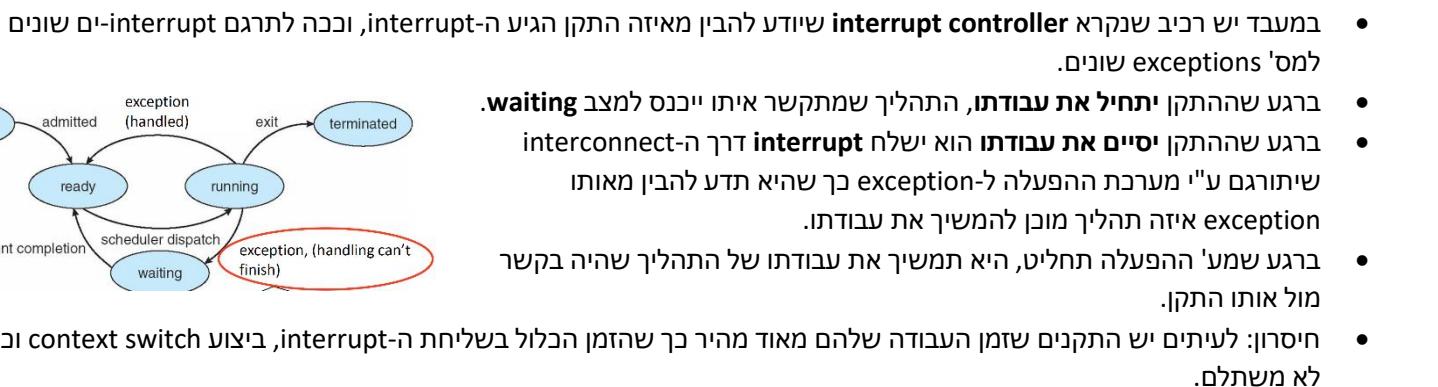
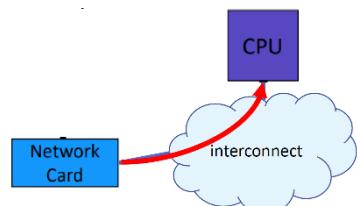
- רגיסטר סטטוס יהווה את החיווי האם התוצאה מוכנה או לא.

חיסרון: בזמן שמערכת הפעלה מחייבת לתוצאה חיובית מהרגיסטר אין תהליכי אחרים שמנצלים את ה-CPU.



### Avoiding busy waiting

- בזמן שמע' הפעלה מחייבת לתוצאה של התקן נרץ תהליך אחר על ה-CPU
- חיסרון: עדין יש קריאות שבזקנות האם ה-status תקין או לא שמבזבזות זמן ריצה עבור המעבד + ייתכן שהסטטוס תקין ואף אחד לא יודע על זה.



### Interrupts

- ברגע שהתקן סיים את עבודתו הוא ישלח interrupt.
- interrupt - מאורע חיצוני שגורם לexception במעבד.
- interrupt נשלח דרך ה-interconnect.

במעבד יש רכיב שנקרא interrupt controller שיודיע להבין מאייה התקן הגיעו interruptים שונים. ברגע שהתקן יתחיל את עבודתו, התהילך שמתקשר אליו יבנש במצב .waiting.

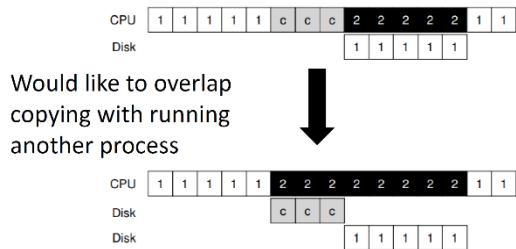
ברגע שהתקן יסימם את עבודתו הוא ישלח interrupt דרך ה-interconnect שיתורגמו ע"י מערכת הפעלה לexception. כך שהbia תדע להבין מאייה exception איזה תהליך מוכן להמשיך את עבודתו.

ברגע שמע' הפעלה תחליט, היא תמשיך את עבודתו של התהילך שהיא בקשר מול אותו התקן.

חיסרון: לעיתים יש התקנים שזמן העבודה שלהם מאד מהיר כך שהזמן הכלול בשילוח interrupt, ביצוע context switch וכו' לא משתלם.

### מצב היברידי

- התחלה עבודה מול התקן במצב polling.
- כאשר רואים שימושים רבים להתקן לעבודים לשימוש ב-interrupts.

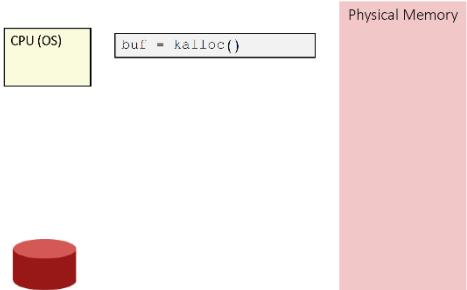


- במקום שבזבז זמן עבודה של CPU על העתקה מידע - או מ- רגיסטרים של התקן, ניתן להתקן לבצע את ההעתקה מהרגיסטרים שלו לדיברן או מהזיכרון לרגיסטרים שלו.

- כך יוכל לנצל את זמן המעבד זהה לritchא של תהליך אחר.
- בשביול לבצע זאת, הוסיפו לתקנים רכיב בשם DMA.

## DMA (= Direct Memory Access)

- אפשר להתקן את היכולת לשלוּח ל-interconnect הודעות קרייה וכתיבה אל ה-DRAM (בדוק באותו אופן שהמעבד עושה זאת).

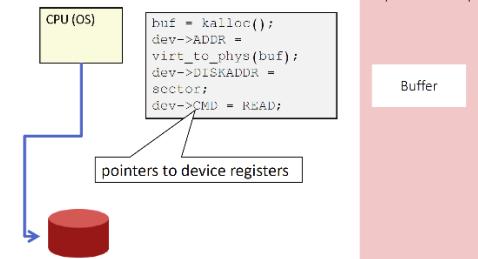


- איך זה מתבצע:

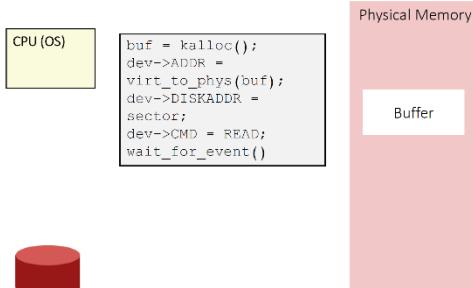
### 1. ה-OS צריך להכין buffer ב-DRAM

- כתיבה ל-device – buffer יוביל את המידע שצריך להיכתב.
- קרייה מה-device – התוכן של ה-buffer לא רלוונטי כי הולכים לכתוב עליו.

2. הernal מבקש מהתקן לבצע את ההעתקה ע"י זה שהוא מתכנת את התקן עם כל הפרמטרים שמנדרים את ההעתקה: הכתובת הפיזית של הבארה, הגודל שלו ועוד..

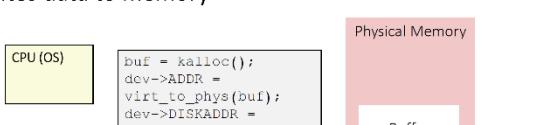


/wait for completion

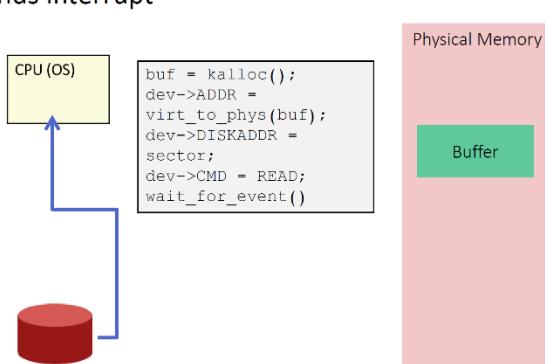


3. התקן מתחילה בקריה/כתביה והתהליך שמתקשר אליו נכנס למצב waiting, כך שמערכת הפעלה יכולה להריץ תהליכים אחרים על המעבד.

Disk writes data to memory



Disk sends interrupt



4. בשעה התקן יסיים הוא ישלח interrupt למערכת הפעלה.

- יתכן מצב בו ה-e-frame-ים שהוקצו עבור ה-buffer out page עד שההתקן יקרה/יכתוב אליהם. במצב זה, ההתקן לא יהיה מודע לכך ויכתוב/יקרא מה-e-frame-ים הללו בכל מקרה. כדי להימנע ממצב זה, עבור כל frame שמקצתה לה-buffer – עושים לו **pinning** – סימון של ה-e-frame-ים כך שלא יהיה ניתן לעשות להם out.page.
- לאחר סיום העבודה ההתקן, ה-e-frame-unpinning כך שבעת ניתן למשתמשם them.out.page.



#### עקרונות שבינן לקחת מהונשאים האחורניים עבור שיפור ביצועים של תוכנה

1. **overlapping** – במקום לחכות ל-event כלשהו, למצוא משאנו לעשות עד שה-event יקרה (busy waiting).
2. **batching** – ייעול ביצוע של משימות ע"י אגירתן וביצוע batch של משימות ולא משימה בודדת. משפר ע"י:
  - (א) **מקביליםות** – התקנים מסוגלים לעבוד במקביל על כמה משימות.
  - (ב) **amortization** – מזעור של פעולות ה"תחזקה". במקרה לרגיסטר בהתקן עבור כל פקודה IO, ניתן לבצע לרגיסטר כמה פעולות IO ביחד. דומה ל-unrolling loop מבנה מחשבים.

```

int *Status = (int *) 0xde00;
while (*Status == BUSY) {
}
 compiler optimization
if (*Status == BUSY)
 while (1) { }

volatile int *Status = ...;
while (*Status == BUSY) {
}

```

#### בעיה בשימוש בדיברין (רגיטר) שאינו מעודכן ע"י ה-OS

- במצב בו ה-OS ממתינה שריגיסטר כלשהו המעודכן ע"י התקן חיצוני קיבל ערך מסוים, וגם לא מופיע בקוד נקודה שהרגיסטר מקבל ערך אחר, יכולה להתבצע אופטימיזציה ע"י הקומפיאילר שתביא לולאה אינסופית.
- כדי להימנע מכך, משתמשים ב-keyword: **volatile**.
- **משמעותה: המשתנה יכול לקבל ערך ע"י מישחו שאינו ה-OS (התקן חיצוני).**
- בתוצאה מכך, הקומפיאילר לא יבצע אופטימיזציה שקשורה לאותו משתנה.

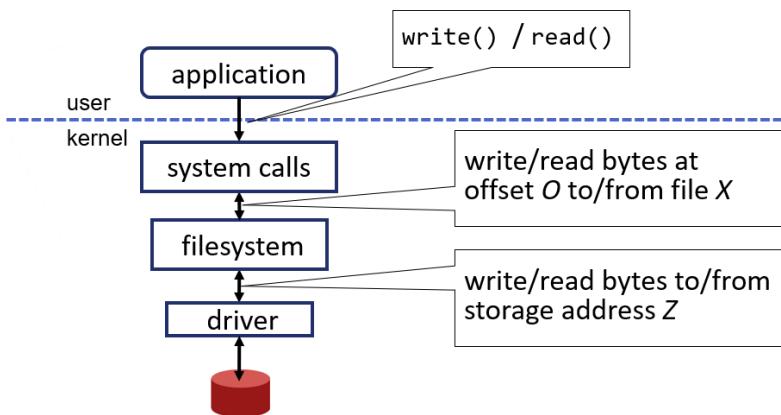
## File

- file הוא אבסטראקציית אחסון.
- הוא מכיל:
  1. Data – רצף של בתים (חסר משמעות עבור מערכת הפעלה).
  2. Meta Data
    - מידע על-h Data (מי הבעלים, הרשות גישה, متى קראו מהקובץ וכו'...)
    - בעל ערך עבור מערכת הפעלה והוא מוגדר על ידה (struct עם שדות)
    - בגלל ההבדלים בין-h data ו-h meta data הם יישמו בצורה שונה בזיכרון).

## File Semantics

- 2 תכונות שמערכת הפעלה מספקת עבור קבצים:
  1. Coherence – מצב בו עדכון של קובץ כלשהו אצל ההילך כלשהו יעדכן את אותו הקובץ **שבר כל תחילה**.
    - בהרבה תשומות תכונות, הפונקציות לבתייה וקריאה מקבצים לא משתמשות בקריאות המערך כתביה וכתייה עבור כל קריאה לפונקציה (יקר). במקרה זה הן משתמשות ב-buffer ומשתמשות בקריאות המערכת רק בsha-buffer מתמלא.
  2. Durability – מצב בו עדכונים שנעשים לקובץ לא נעלמים במקורה בו מבאים את המחשב או שהוא נופל.
- קיימן trade off בין קיומם התכונות הללו לבין **ביטחוניות המעבד**. בפועל, מערכת הפעלה מבטיחה קורנתוניות מיידית, אך לא מבטיחה דוראיות מיידית. במקרה, עבור תוכניות שכן דורשות דוראיות מיידית יש קריאת מערכת שմבטיחה זאת.

## Filesystem



- **Filesystem** הוא תת-מערכת בקרנל = מודול.
- **תפקידו**: למש את אבסטראקציית הקבצים על התקן אחסון כלשהו (= לספק **קורנתוניות ודוראיות** עבור הקבצים שנשמרים על התקן).
- **כלומר**, היא סוג של מתווך.
- **ברגע נניח שיש FS יחיד שמנוהל ע"י OS**
- **ושכל קריאת מערכת גורמת ל-IO ולא חזרה עד SHA-IO מסתיים.**

## Inode (= Index Node)

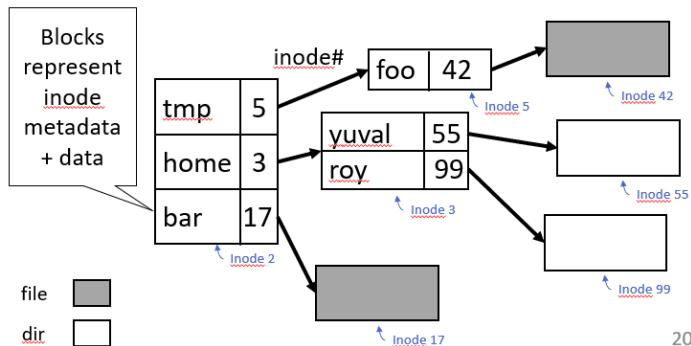
- זה האובייקט (struct) שמייצג קובץ או תיקיה (= ספירה) במערכת הפעלה.
- כל אובייקט inode הוא מבנה נתוני (struct) הכלל:
  1. metadata של הקובץ.
  2. מבצעים אל-h data של הקובץ.
- לכל inode יש מספר ייחודי המהווה את המזהה שלו.
- **inode והמיופיעים** מהמס' שלהם ל-inodes עצם נשמרים בהתקן האחסון -> פריסיטנטים ונשמרים גם בשמהחשב נופל.

## Name Spaces &amp; File Naming (path names)

- **מרחב השמות** שמערכת הפעלה משתמשת בו הוא שטוח = לכל inode מס' ייחודי בלבד.
- מרחב השמות שאנו מכירנו מקרים משתמש ב-h path names = file naming. ככל קובץ יש נתיב היררכי שמערכת הפעלה חושפת עבור התהליכים (כמו c/b/a).
- חשוב להבין שהשמות בשני המרחבים מתייחסים לאותם אובייקטים = ה-h inodes.

- ספירה היא inode מיוחד שה-data שלו מכיל הצבעות ל-inodes אחרים.
- ספירה היא inode רגיל לכל דבר, מלבד התוכן של ה-data שלו.
- המודול של FS יודע לפרוס את ה-data ולהתייחס אליו בהתאם (בצורה היררכית בהתאם).
- ב כדי להבדיל inode של קובץ מ-node של ספירה יש ביט כלשהו במתה דатаה של inode שמסמן זאת.
- כמובן, בפועל, הקבצים אינם ממוקמים בצורה היררכית כמו שנחננו וריגלים לחושב עליהם, אלא נמצאים באחסון בצורה שטוחה וה-inodes של התיקיות הם אלו שגורמים לכך שנוכל לראות ממשק היררכי.

תיקיות ה-node # 2 = root



- כל-node מסוג תייניה (= ספירה) מכיל 2 כניסה ב-data שלו שנוצרות באופן אוטומטי עם ייצור הספירה:
  1. – מצביע ל-node עצמו.
  2. .. – מצביע להורה של ה-node במרחב היררכי.

#### יצירת ספירה, (mkdir(<path>))

system call

1. ייצור inode חדש.
2. ייצור מיפוי מתאים ב-data של inode ה-הילוגני (אל ה-node החדש).
3. הדלקת הדגל המתאים ב-node החדש (במתה דатаה שלו) שמסמן שמדובר בתיקייה.
4. ייצור מיפויים עבורי .. ב-data של inode החדש.

#### יצירת קובץ, (open(<path>, O\_CREAT))

system call

1. ייצור inode חדש.
2. ייצור מיפוי מתאים ב-data של inode ה-הילוגני (אל ה-node החדש).

#### שינוי נתיב/שם, (rename(<src path>, <dest path>))

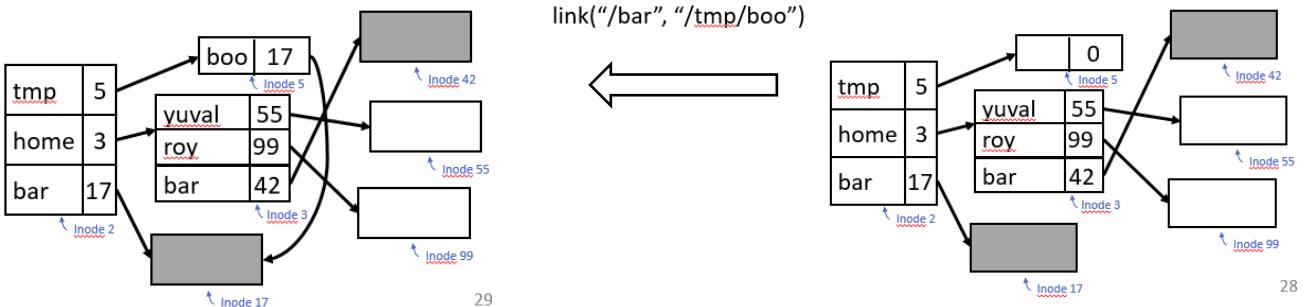
system call

1. מחיקת הרשומה של <src path>
2. הוספה הרשומה <dest path>

מבצעת בצורה אוטומטית – לא יתכן מצב בו client של מערכת הפעלה יראה מצב ביןיהם, אלא הוא יוכל לראות את המצב לפני השינוי או לאחריו.

**link(<old\_path>, <new\_path>), Linking**

- system call שגורמת ליצירת רשותה new\_path המצביעת על-old\_path.
- אם-new\_path כבר קיים אז היא לא מודגשת אותו.
- 2 ה-reference-ים מתיחסים לאותו הקובץ, ולכן חולקים אותו הרשות ו-ownership.
- דוגמא:



הweeney: לאפשר שם שונים לאותו data מבלי הצורך להעתיק אותו.  
שימוש עיקרי הוא לצורך יצירה גיבויים.  
היצירה של היכולת הזאת מוסיפה את הצורך בבדיקה שאין אליו עוד הצבעות. המידע הזה נשמר ב-data של ה-inode.

- 
- 
- 

```
struct stat {
 dev_t st_dev; /* ID of device containing file */
 ino_t st_ino; /* inode number */
 mode_t st_mode; /* protection */
 nlink_t st_nlink; /* number of hard links */
 uid_t st_uid; /* user ID of owner */
 gid_t st_gid; /* group ID of owner */
```

**קריאה meta data של inode**

- יבוצע בעזרת קריאת המערכת: (`stat()`)
- ה-`syscall` זהה ויחזיר את ה-`struct` הבא:
- הביט שאמור האם ה-`inode` הוא ספריה
- או לא והוא אחד הביטים שדה ה-`st_mode`.

**unlink(<path>), Unlinking**

- system call שגורם ל-  
1. הסרת המיפוי <path>
- 2. הודה ב-1 של מס' הלינקים של ה-inode אליו הוסר הקישור (st\_nlink--)
- inode שמתקיים עבורי 0 == st\_nlink נחומר שמדובר שהמקום באחסון בו היה ה-data של ה-inode נחומר
- עכשו פניו מבחינה ה-OS ואפשר להשתמש עבוריקבצים אחרים.

**Deletion vs. Unlinking**

- אין פעולה מפורשת של מחיקת data מהאחסון, אלא מחיקת מיפויים בלבד.
- ברגע של קובץ אין יותר מיפויים שמנפנים אליו, הוא נחומר "מחוק" מבחינת ה-OS.
- מיפויים יכולים להיות גם מיפויים של fd ולא רק מיפויים של links.
- למשל: ("open(""/foo/bar"") <- open(""/foo/bar"")"). במקרה זה יצרכנו unlink לקובץ ומיד אחר בר ניתקנו את ה-link. עם זאת, יש עדין תהליך שב-`file table` שלו יש מצביע ל-`struct file` שמצוין ל-#`inode` של הקובץ. בשונה מ-linking רגיל הנשмар על התקן האחסון, זהו reference שנמצא בזיכרון.
- לכן, ניתן לעשות unlink לקובץ גם אם יש תהליך שמחזק את הקובץ פתוח.

**Path Name Resolution**

- תרגום שם במרחב השמות ההיררכי למס' inode.
- מדובר במעבר פשוט על המסלול בעץ הספריות באחת מ-2 הדרכים הבאות:  
1. אם המסלול מתחילה-/ -> מסלול אבסולוטי -> הטויל מתחילה inode של השורש.  
2. אחרת -> מסלולRELATIVE -> מתחילה בספרייה העבודה הנוכחיית (CWD).
- **CWD (= Current Working Directory)**
- **זהה של inode** שמותוחזק ב-PCB של כל תהליכי.

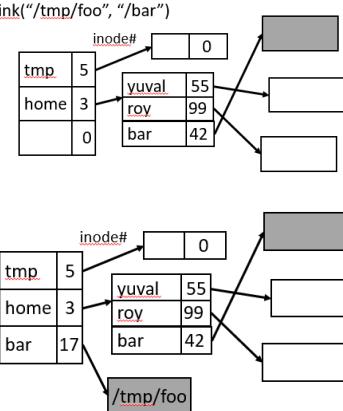
- system call – **chdir(<path>)** שמאפשר לשנות את ה-CWD של PCB.
- אם **<path>** אינו נתיב של inode שקיים במרחב השמות השטוו אז מתקבל שגיאה.
- הבדיקה הנ"ל מתבצעת בעזרת **resolve**.

```
resolve(inode, path):
 # initially, inode is either root or cwd,
 # depending if path is absolute or relative
 if "/" not in path:
 return lookup(inode, path) # search for "path" in inode, which
 # is assumed to be a directory
 first, next = path.split("/")
 inode = lookup(inode, first)
 if inode == NULL: error # not found
 if inode.isSymlink():
 next = contents of inode's file + "/" + next
 return resolve(inode, next)
```

### symbolic(<path1>, <path2>) ,Symbolic Links

system call שגורם ל:

1. ייצור inode חדש path-sha-data שלו יוכל את path והmeta DATA שלו יכול דגל שאומר שהוא symbolic link.
2. ייצור מיפוי-path-sha-data של inode שנדמצה ב-2 אל inode החדש שיצרנו. יש להתייחס לכך שיכולים להיות symbolic links בעקבות ביצוע resolve.リンク סימבולי יכול להצביע על-path שאינו קיים (can be dangling), בשונה מלינק רגיל. לכן, link סימבולי נקרא **soft link**, בעוד link רגיל נקרא **hard link**. המוטיבציה לשימוש ב-soft link היא יצירת links בין FSים שונים. אם משתמש ב-hard links לא יוכל לעשות זאת.



### Symbolic Link Cycles

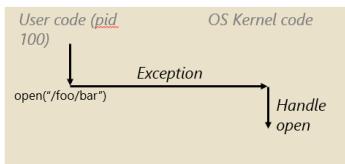
למשל: **(<path1>, <path2>)** symlink(**<path1>**, **<path2>**).path resolution.

- פתרון: הגדרת סף (threshhold) של מס' הLINKS הסימבוליים המקוריים שאלגוריתם path resolution מוכן לעבור דרכם. אם הוא עבר יותר מהסף יחזיר שגיאה (ערך -1 ב-chdir())

### Hard Link Cycles

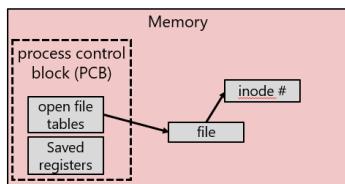
- אין אופציה שנגע למצב של לולה אינסופית לאחר מכל מעבר על link וgil מקצר את path ב-1 וכן מובטח לנו שבסוף של דבר path resolution יסתום.
- הבעיה היא עם תוכניות שורחות לטיל על מרחב השמות בצורה רקורסיבית. תוכנית זאת (למשל: תוכנית של גיבוי קבצים) עלולה להיכנס לולה של hard links מבלי שתוכל להזמין אותם (אחר ויתכן ואו מעגל, אלא מסלול עם תיקיות בשמות שחוזרים על עצמן).
- בשביל לפטור את הבעיה אוסרים על יצירת hard link שיוצר מעגל.

הומצאו בכדי שפניות שראחות לעבוד מול קבצים/input/output/sockets של אותו אובייקט.



- מאחר ולא מדובר רק בקבצים אז אי אפשר להשתמש ב-node בטור מזהה.

- **לכל תחילך יש open files table** המבילה את ה-fds של כל הקבצים שהתחליך פתח.



- כל fd יצביע לאובייקט **struct file** (indirection) שיכיל מצביע לקובץ אותו

אנחנו עובדים בנוסף למידע נוסף הקשור לאותו קובץ:

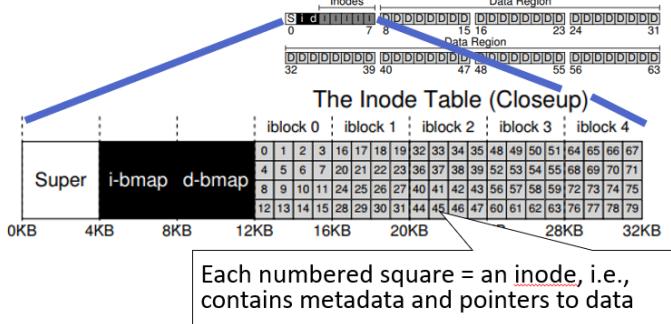
- אילו הרשאות יש לנו אל מול הקובץ דרך ה-fd זהה.
- כמה בתים בבר קראנו ממנו (ה-offset בתוכו).
- reference count – כמה הצביאות מ-fds שונות יש לאותו הקובץ.

- **באשר מבצעים fork לתחלת מסויים** איזה open file table של התחילך

המושתק מועתקת אל התחלת החדש בכשה שכל הקבצים שהתחליך המקוריפתחותם גם עברו התחלת החדש (לכן, בעת ביצוע **fork** ה-fd-struct file reference count עולה ב-1).

- הבעה: מציאת מבני נתונים לייצוג קבצים על התקן האחסון.
- ניתן לפרק ל-2 תתי בעיות:
  1. **בעיית מיפוי – בהינתן offset inode ל-data** – תחזרו את מיקום inode על התקן. ובאופן דומה, בהינתן inode, תחזרו את מיקום ה-data על התקן.
  2. **בעית ניהול משאבים – ניהול המיקום הפנוי בהחסון.**
- האילוץ העיקרי – תחזוקת מבני הנתונים על התקן (לצורך תכונת הפרסיסטנטיות = דוראיות):
  - גישה לסקטור שלם בלבד (היחידה הגרנולרית על התקן) לצורך כתיבתה/קריאה.
  - קיימים FSs בהן היחידה הגרנולרית היא extents (איחוד של סקטורים) – לא יורחב בקורס שלו.
  - הפעולות עוברות דרך דרייבר.

Given inode #, can calculate where inode is on device



⇒ Inode X is in iblock [X/16], at offset X mod 16

- נניח שיש לנו התקן אחסון עם 64 בלוקים של 4KB.
- **= Data Region** – 90% מהבלוקים משמשים לאחסון data
- 56 הבלוקים האחרונים בהתקן.
- ב-10% הנוראים:

- 5 בלוקים לאחסון מערך של inodes – כל בלוק מכיל הרבה inodes.
- 2 בלוקי bitmap עברו הקצתה data והשוו עברו הקצתה (inodes).
- abit ה-*j*-ב-block bitmap של ה-*i*-inode ה-*j*-ב-block נמצא ב-*i*-inode ה-*j*.
- הblk הראשון יהיה **superblock** – מכיל את המיקום של כל מבני הנתונים.
- מס' הקבצים האפשריים הוא חסום כי מס' ה-inodes הוא חסום.

### VSFS (= Very Simple FS)

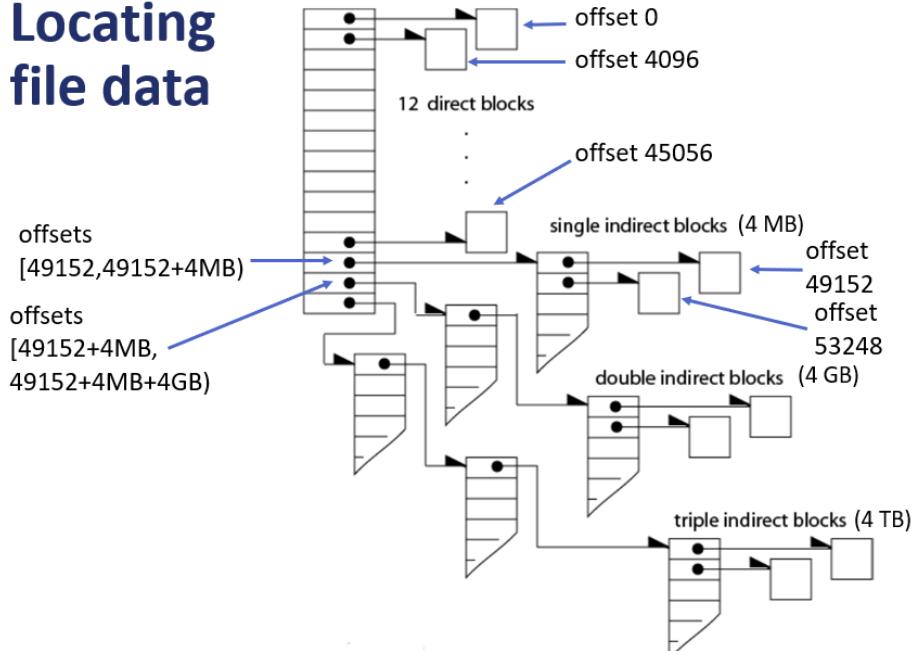
- מוגדר ע"י ה-FS ויכול להשתנות בין FS שונים.
- מכיל:
  1. metadata
  2. pointers to data blocks
- אופן ביצוע המיפוי **inode ל-data** (փינטרים של ה-inodes)
- נניח גודל מצביע הוא 4B.
- **סוגים של פינטרים:**

#### VSFS Inode

- פינטרים שמצביעים לבlok של **data**
- מכיל:
  1. metadata
  2. pointers to data blocks
- אופן ביצוע המיפוי **inode ל-data** (փינטרים של ה-inodes)
- נניח גודל מצביע הוא 4B.
- **סוגים של פינטרים:**

| סוג המצביע      | הסבר                                                | גודל ה-data המצביע                                                           | air מנוטים                                                                                                                                                                    |
|-----------------|-----------------------------------------------------|------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| פינטרים         | פינטרים שמצביעים לבlok של <b>data</b>               | 4KB                                                                          | מחולקים את ה-offset ב-4096 כדי למצוא את המצביע                                                                                                                                |
| indirect block  | פינטרים שמצביעים על בלוקים של <b>מצביעים ל-data</b> | $1024 * 4KB = 4MB$<br>(הנחה גודל מצביע 4B ולכן יש 1024 מצביעים בבלוק של 4KB) | מחולקים את ה-offset ב-4MB כדי למצוא את המצביע<br>ולמצביע.<br>ואז בבלוק שהוא מצביע עליו מחולקים שוב ב-4096.                                                                    |
| double indirect | פינטרים שמצביעים על בלוקים של <b>indirect block</b> | $1024 * 4MB = 4GB$                                                           | מחולקים את ה-offset ב-4GB כדי למצוא את המצביע הראשוני.<br>ואז בבלוק שהוא מצביע עליו מחולקים ב-4MB כדי למצוא את המצביע עליון.<br>ואז מחולקים ב-4KB כדי למצוא את המצביע ל-data. |
| triple indirect | פינטרים שמצביעים על בלוקים של <b>double block</b>   | $1024 * 4GB = 4TB$                                                           | וכך הלאה...                                                                                                                                                                   |

# Locating file data



כיצד מוצגת ספירה (תיקיה) בצורה פיזית על התקן האחסון?

- הבעיה היא בעיית serialization – יציג של אובייקט ברצף בתים.

| Header |         |        | הרשומה עצמה<br>recrlen B |  |
|--------|---------|--------|--------------------------|--|
| 7B     |         |        |                          |  |
| inode# | recrlen | strlen |                          |  |
| 4B     | 2B      | 1B     |                          |  |

- בשביל לעבור בין רשומות נתקדם ב-B recrlen כדי להגיע לרשומה הבאה.

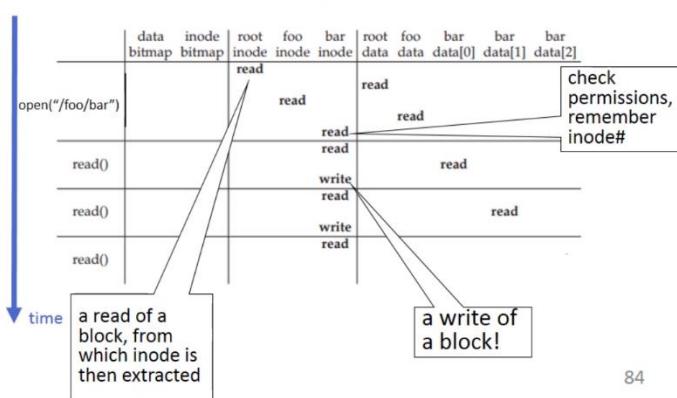
מעקב אחר open & read לקובץ

- זיכרון שהנחנו: כל גישה ל-data או ל-inode מתחבאת מול התקן האחסון.

`open("/foo/bar")`

- קריאת הבלוק של inode של root בשבייל לראות אם יש שם רשומה בשם foo.
- אחרי שמצא רשומה עם השם foo הוא מזמין inode מס' inode# שאליו היא מציבעה.
- הוא עובר אל הבלוק של inode# זהה ומוחפש שם רשומה עם השם bar.
- אחרי שמצא רשומה עם השם bar הוא מחזיר את מס' inode# שאליו היא מציבעה.
- הקרנל בודק ב-metadata inode של inode# אם לתפקיד מותר לגשת ל inode# זהה.
- אם כן, הקרנל מקצה fd שיצביע ל struct file שיכיל את מס' inode# של inode#.

Example: open & read "/foo/bar"



`read("/foo/bar")`

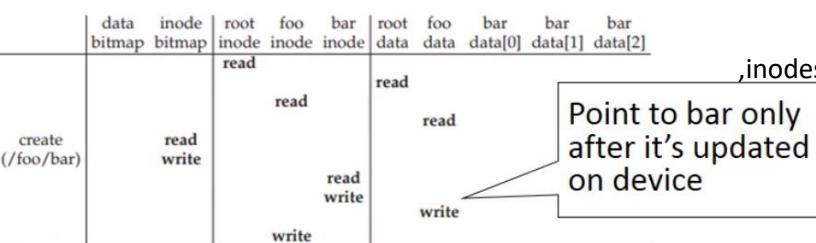
- דרך ה-fd ניתן להגיע למיקום של inode# בהתקן.
- דרך ה-fd מגעים ל-data inode# מה-inode# מציביע עליו.
- קוראים את ה-data.
- מעדכנים ב-meta-data את שדה meta-data offset של struct file שמצויבע ע"י ה-fd.
- הקרנל מעדכן את שדה offset של struct file שמצויבע ע"י ה-fd.

- מטה DATA של inode שאומר מתי בוצעה הגישה המפורשת האחרונה ל-node.inode.
- גישה מפורשת =
- עבר קבצים – הפעם האחרונה שבוצע read/write.
- עבר תיקיות –
- קראה של data ע"י תהיליך (למשל: Ea).
- עדכון של inode (linking/unlinking/rename updates) data.(linking/unlinking/rename updates)
- לא מתעדכן כאשר מבצע **path resolution** למציאת # inode של path בולשויו.

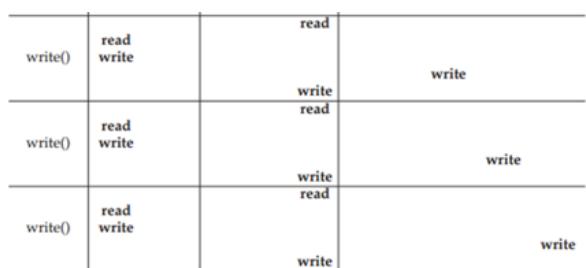
### מעקב אחר create & write לקובץ

חשוב: כאשר רוצים לכתב מידע לתוך התקן, אי אפשר לכתבו אותו בBITSים בוודאים.  
במקרה, יש לקרוא את כל הבלוק לתוך אפר, לעדכן את מה שרוצים ולכתבו את הבלוק מחדש.

**create(""/foo/bar") = open(""/foo/bar")**



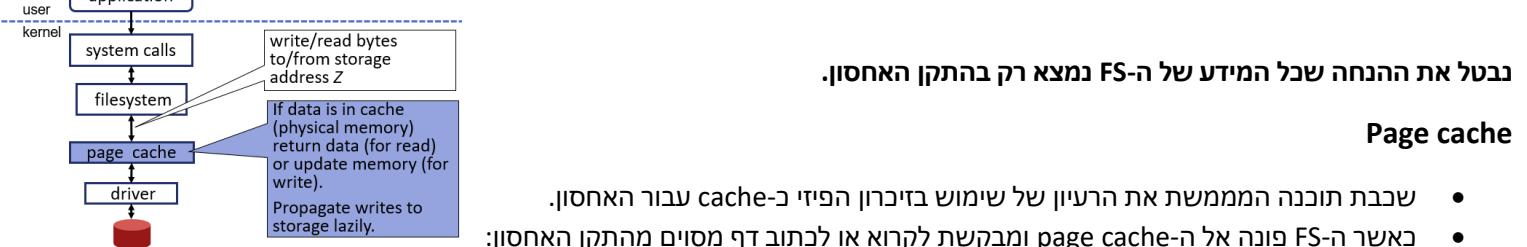
- .1resolution למציאת-h-data של foo.
- .2יצירת inode# inode – מציאת פניו ב-k-bitmap.
- .3אייפוס הביט שלו וכטיבת הבלוק בחזרה להתקן.
- .4קריאת-h inode בהתאם ל-node.inode שנקצנו ב-k-bitmap.
- .5אתחול BITSים של inode-h-data של foo.
- .6כתיבת הצבעה אל bar ב-h-data של foo.
- .7עדכון-he-access time של foo.



**write()**

88

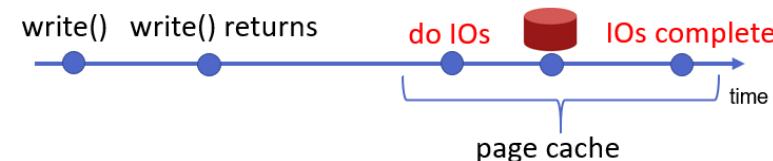
- .1 קריאת-h inode של bar כדי למצאו את המצביע על-blk-h-data.
- .2 מאחר ומדובר בקובץ חדש אז כל המצביעים שלו -data מאופסים ולכן צריך להקצתות לו דרך-h-data bitmap.
- .3 אין צורך לקרוא את המידע של הבלוק שהוקצה, מאחר והוקצה בלוקשלם ולא חלק ממנו, לכן אפשר פשוט לכתבו לתוכו.
- .4 כתובים את inode של bar:
  - שדה הגודל של הקובץ
  - access time



Without page cache (simplifying assumption):



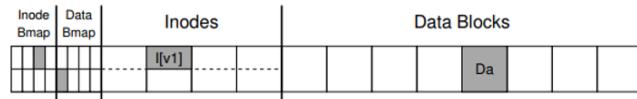
With page cache (in practice):



- **הרעין הקונספטואלי של page cache :** **input**: ID של התקן אחסון + כתובת של דף על התקן. **output**: physical memory frame caching אליו עשו caching של הדף המבוקש מהתקן.
- אם הדף מהתקן לא עבר caching (הוא לא נמצא בזיכרון הפיזי), אז קודם יבוצע לו caching ואז יוחזר הדף מההתקן.
- **frame number** בשימוש ב-**page cache** תכונת הדוראיות נפגעת (יתכן שמיידע שתהיליך כתוב לא יגיע לתקן האחסון).
- **fsync(fd)** לעתים זה לא מספיק לעשות את זה רק על הקובץ ויש צורך לעשות את זה גם על הספרייה.

- בעיה הנובעת מכך שפעולות כתיבה להתקן אחסן לוקחות יותר זמן פעולה בודד, וכיים מצב שבמהלך ביצוע הפעולות הללו החשמל ייפול ולא יוכל להבטיח שעם חזרת המערכת מבני הנתונים של ה-*FS* יהיה מסונכרים.
- **Filesystem inconsistency** – מצב בו המידע במبني הנתונים של ה-*FS* סותר אחד את השני (למשל: ה-*data bitmap* אומר שבלוק *data* כלשהו הוקצה, אבל מגdag אין אף *inode* שמצויב אליו).
- לדוגמה: עבור המצב הבא

Initial state: Filesystem with one allocated *inode*



נניח שאנו רוצים להוסיף עוד בלוק *data* ל-*FS*. זה כולל 3 פעולות:

1. עדכן ה-*data blocks*.
  2. עדכן ה-*data bitmap*.
  3. עדכן הבלוק שמכיל את ה-*inode* (עדכן *access time*, האפשרות הכחית גורעה היא: لكن, יש ! 3 אפשרויות לסדר בתיבת המידע. האפשרות הכחית גורעה היא:)
1. כתיבת ה-*data*.
  2. עדכן ה-*data bitmap*.
  3. עדכן ה-*inode*.

אם נפלנו אחרי שלב 1, אך אין *FS inconsistency*. אם נפלנו אחרי שלב 2, אך יש *FS inconsistency*, אבל הוא עולה לנו רק בעדילift זיכרון של התקן האחסן.

## פתרונות ל-*crash consistency*

### FSCK (= FS checker)

- **גישה התיקון בדיעבד.**
- תוכנית שתורץ עם עליית המחשב לפני שתוכניות אחרות רצות ותבדוק את הקונסיסטנטיות של מבנה מערכת הקבצים.
- דברים שהוא בודק:
  1. יבנה רשימה של כל הבלוקים בשימוש (모צבעים ע"י *inode*) וישווה אותו מול ה-**data bitmap**.
  2. יזודא שה-**count link** של כל *inode* נכון. אם הוא קטן מדי או גדול מדי אז יבצע אליו מיפויים מ-*lost & founds* (*inode#1*).
  3. אם יש *inode* שמספרה לבlok *data* שהabit שלו ב-*data bitmap* בובי אז הוא ידליך אותו.
  4. בדיקות נוספות.
- הבעה העיקרית איתנו היא שהוא איטי.

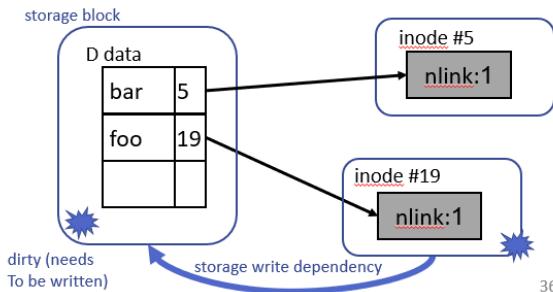
- הרעין: הכתבת סדר שבו ביצוע פעולות ה-IO להתקן יבוצעו כך שהבעה היחידה שתוכל לkerot היא memory leak בהתקן.
- בקרה דזאת, יהיה אפשר להשתמש ב-FS מיד כשהמערכת עולה ולהריץ תוכנית ברקע שתחפש בЛОקים שדלו מוביל להשתמש ב-**FSCK**.

### Soft updates - האלגוריתם ordering העיקרי

- 3 עקרונות:

1. אסור לבוטב להתקן מצביע לאיזשהו מבנה אחר בהתקן לפני שהמבנה האחר אותחל בהתקן.

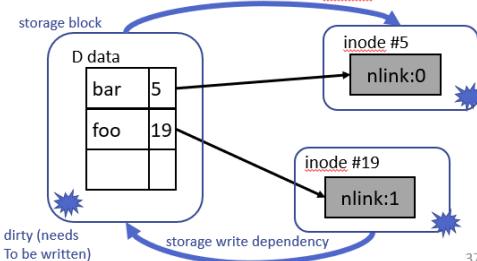
- Link "foo" in directory D
  - Should write "foo" inode before D data



36

2. אסור לעשות שימוש מחדש באיזשהו מבנה או בлок בהתקן לפני שמאפסים את הצבעות אלו בהתקן.

- unlink "bar" from directory D
  - Should write D data before bar inode



37

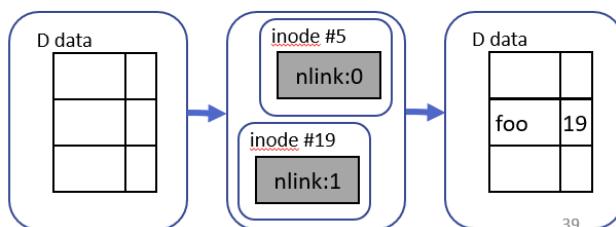
3. אסור לאפס מצביע ישן לאיזשהו מבנה או בлок לפני שהוא שוכתנים את המצביע החדש.

כמו בדוג' לעיל עלולות להיות תלויות מעגליות, במידה ושני inode-ים שייכים לאותו הבלוק בהתקן.

ניתן לפתרור זאת על ידי ביצוע שינויים יותר קטנים, ובמידת הצורך ביצוע roll back לעדכונים מסוימים:

**Problem:** cyclic dependencies

**Solution:** roll-back some changes before writing blocks to break dependencies



38

האלגוריתמים האלה מאד מסובכים וудין גם איתם קשה להבטיח את הסדר.

**פתרון:** ביצוע העדכונים בצורה סינכרונית.

הבעיה היא שזהו פתרון מאד אוטי.

**פתרון:** פתרון חומרתי בהתקן שיבטיח את הסדר.

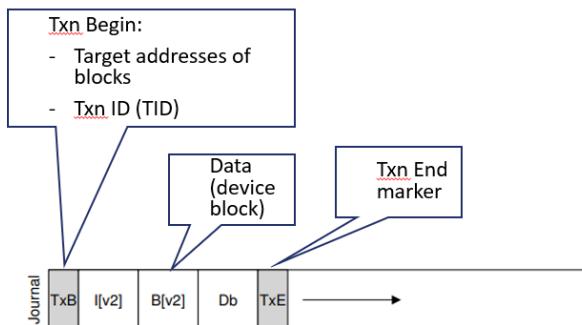
- מעבר להתקן פקודות שנקראות פקודות **barrier** (מחסום).

משמעות פקודה זאת היא שיש לאכוף את הסדר בין הפקודות שבאו לפני פקודות שבאו אחרת.

אבל למעשה כל אחת מקבוצות הפקודות הללו אין הגבלה.

בלומר, יש לדאוג רק שפקודת barrier תקרה לפני כל הפקודות שבאו אחרת.

- הרעיון: ה-FS יתחזק מבנה נתונים שנקרא לוג (או journal) שיביל את כל הפעולות IO של התקן לביצוע.
- בכך, אם המחשב ייפול התקן יוכל לבצע מחדש פעולות שלא בוצעו.
- באשר פעולה שרשומה בלוג בוצעה על ידי התקן. – **checkpointing**



- מבנה של כניסה לוג transaction begin – header**
- התוכן שורצים להעברה להתקן
- transaction end – commit block**

### מבנה ה-journal

- ה-journal יהיה מעגלי ככה שאחריו התא האחרון מגיע התא הראשון.
- נתחזק **journal super** – 2 מצבים:
  1. תחילת ה-journal.
  2. סוף ה-journal.

### air זה יומש?

- כTİיבת לוג – כתיבת התוכן של transaction .transaction
- יש לחכות עד שכל כתיבת התוכן של הפעולות שצרכות להבצע תסתיים או להשתמש ב-barriers .transaction end – כתיבת ה-transaction commit
- כTİיבת הפעולות של הלוג להתקן בצורה אסינכרונית.
- עדכן ה-journal super – עדכן 2 המצביעים (יבוצע בזמן בלשוח מאוחר יותר).

### batching

- במקום לכתוב פעולה בודדת עברו כל טרנסקציה – נרשום כמה פעולות שאין ביןיהן סדר ביחס.
- בכך אפשר גם לחסוך פעולות IO, אם למשל יש 2 פעולות שמבטלות זו את זו, אז אין צורך לבצע אותן.

### recovery

- אחרי קירישה, הקוד של מערכת הקבצים יעבור על הלוג וינגן את הטרנסקציות שבתוכו.
- תוכן שיופיע פעולות שיבוצעו לחינם (redundant), אך ביצועו יהיה נכון מבחן המצביע ש策יר להיות בפועל.

**הבעיה:** **data journaling** מכפילה ב-2 את כמות הכתיבה להתקן האחסן (בכל כתיבת יש כתיבת לוג + מקום האמיית בהתקן).  
פתרונות:

### Meta Data Journaling

הרעיון: במקום לכתוב את המידע גם לוג וגם מקום האמיית, נכתב את המידע ישירות למקום האמיית ובלוג נכתב רק את המטה DATA.

### air זה יומש?

- כTİיבת ה-data לתקן האחסן.
- כTİיבת הלוג – כתיבת התוכן של transaction meta data – transaction commit בלבד.
- יש לחכות עד שכל כתיבת התוכן של הפעולות שצרכות להבצע תסתיים או להשתמש ב-barriers .transaction end – כתיבת ה-transaction commit
- כTİיבת הפעולות של הלוג להתקן בצורה אסינכרונית.
- עדכן ה-journal super – עדכן 2 המצביעים (יבוצע בזמן בלשוח מאוחר יותר).

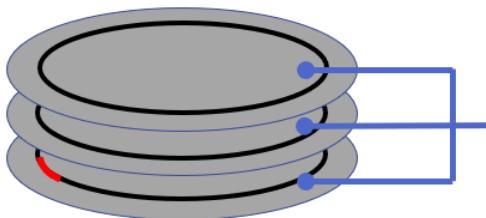
- כעת יש לנו 2 נקודות בהן אנחנו מחכים:

1. בЛОקי ה- data צריכים להיכתב לפני הטרנסקציה.

- 2. בлокי הטרנסקציה צריכים להיבט לפניו ה-commit block.
- במקומות זאת נכתב את בלוק ה-data והטרנסקציה ביחד.
- בעיה: בעת ה-recovery יתכן ונדרס מידע בלחשו עם טרנסקציה ישנה (למשל: עם לינק – פעולה שלא צריכה דאטא בלוג) ולא יוכל לשחזר אותו עם טרנסקציות מאוחרות יותר מהארה והדאטא שלו לא בלוג (דוג' בשקפים 65 – 67 בהרצאה 6).
- פתרונות אפשריים (שלא הורחב עליהם):
  - שיפור ה-FS כך שייעבור על הלוג ויבדק האם יכולות להגרם שגיאות לא רצויות.
  - שיפור מבנה הלוג כך שיהיה בו רשומות שמציניות שאסור לעשות replay לטרנסקציה מסוימת.

## Application Crash Consistency

- במקרה של קriseה, יתכן שה-syscalls שבוצעו ע"י אפליקציה בוצעו בצורה לא נכונה.
- לפי הסטנדרט של POSIX אין הבטחה לדוראיויות של מידע במקורה של קriseה, כמעט ב-*fsync*.
- כמובן, יתכן שישנן FSs שאפליקציות משתמשות בהן ובעת קriseה המידע בתיקון מתחבבש.
- עם זאת רוב ה-*FSs* כן מגדילות ראש ומספקות תוכנות שלא הובטו לפי הסטנדרט של POSIX.
- דוגמא לעיה שהייתה קיימת עד 2009: בהרצאת הקוד הנ"ל, במקרה של קriseה, יתכן שהמערכת הייתה עולה עם file בקובץ ריק. הסיבה לכך: לא היה סדר לביצוע הפעולות (בגל שיקולים של ביצועים) וכותצא מאך, פקודת ה-*rename* הספיקה ל��ות לפני הקriseה, בעוד פעולה ה-*write* לא הספיקה ל��ות לפני הקriseה.
- העזה: **במימוש של journaling שלמדנו זה לא יכול ל��ות.**



- בני ממס' פלטוט שישובות אחת על השניה.
- לכל צד של הפלטה קוראים **משטח (surface)**.
- **RPM** – קצב הסיבובים של הפלטוט.
- **tracks** – המעלגים שמרכבים את המשטח.
- **sector** – בלוקים של מידע המזינים על ה-**tracks**.
- המידע מקודד על הסקטור בмагנטיות של החומר (0 ו-1).
- לכל משטח יש זרע שטוחת ומעדכנת את המagnetיות של הסקטורים.

### המרכיבים המשפיעים על קצב העברת המידע (transfer time)

- – הזמן שלוקח עד שהזרע מגיעה ל-track המתאים.
- – הזמן שלוקח עד שהדיסק מסתובב כך שהזרע יהיה בסקטור המתאים (לאחר שהגעה ל-track המתאים).
- גישות אקריאיות בדיסק מאוד איטיות לעומת גישות סדרתיות (גישה זהה בהן קוראים סקטורים שונים זה אחרי זה באותו track).
- לדוגמה:

## Disk I/O example

- 15000 RPM disk
- 4 ms average seek time
- 125 MB/sec transfer rate

Rate for 100 MB transfer performed with...

Many 4KB random I/Os:

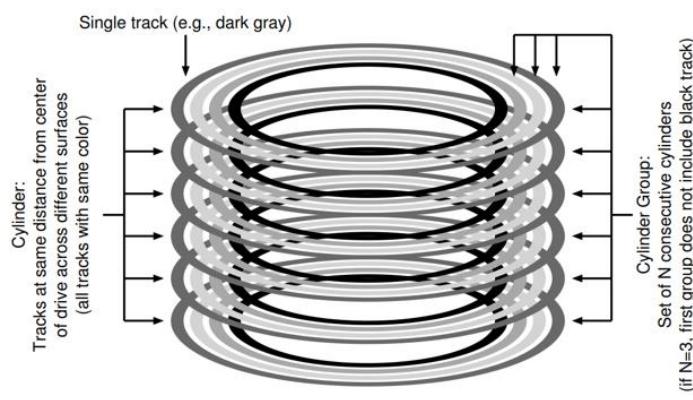
- Seek time: 4 ms
- Rotational delay: 2 ms (250 RPS => 4ms, on avg: 1/2)
- Transfer: 4 KB/125MB = ~30 microsec (neglect it)
- I/O rate: 4 KB/(4+2)ms = **0.66 MB/sec**

Sequential: **~125MB/sec** (modulo initial 6 ms)

⇒ 189x faster than random

12

## FFS (= Fast File System)



- המטרה: להימנע זמן seeking.
- כדי לעשות זאת: מחלקים את הדיסק לפי **קבוצות צילינדרים**.
  - צילינדר – קבוצה כל tracks על פני כל הפלטוט שנמצאים באותו מרחק מהמרכז.
  - **קבוצת צילינדרים** – אוסף של צילינדרים שקרובים אחד לשני.
- **הweeney** – גישות לבlokים שנמצאים באותה קבוצת צילינדרים יהיויחסות מהירות כי הראש לא יצטרך לוזר הרבה.
  - לכל קבוצת צילינדרים:
    1. בלוקי דאטה משלها.

2. בלוקי inode משלה.
3. בלוקי מייפוי נתונים inode ו inode משלה (bitmap).
4. ה-blocksuperblock יהיה זהה עבור כל הקבוצות (העתק אצל כל אחת מהן) – כדי שייהי לו גיבוי אם הסקטור ניזוק.

## FFS Allocation Policy

- המטרה: יצירת מדיניות שתפחית את זמן ה-seek.
- את זה ננסה לעשות ע"י הקצאת בלוקים שקשורים אחד לשני באותו קבוצת צילינדרים, כך שאם ניגשנו לאחד מהם, בכראה שניגש גם לאחר בקרוב (locality) ובין הראש לא יctrck לוז, או אם יctrck לוז איזוד מעת.
- דוגמאות
  - יצירת קבצים עם הספרייה שמכילה אותן
  - יצירת קבצים עם inode שמצויב עליהם
- עבור קבצים גדולים, תוקצה כמות גדולה של בלוקים בכל קבוצת צילינדרים עד ש"נכשה" את כל גודל הקבץ. בלומר, נתחיל מלתקצוץ כמות בלוקים בקבוצה אחת. אם הם לא הספיקו נקצת כמות גדולה של בלוקים בקבוצה נוספת וכך הלאה.
- הארד דיסקים לא חושפים את החלוקה שלהם לפלטות, tracks וכו'... במקומות זה, מוסכם שגישות לבלוקים יהיו יותר מהירות ככל שהם יהיו יותר קרובים זה לזו במרחב הכתובות של הדיסק.

- בשונה מה-FFS שהתבסס על ה-VSFS, ה-LFS עשה משהו שונה לחולוטין.
- פיתוח משנות ה-90 שבא בעקבות:

  1. **העליה האקספוננציאלית בגודל ה-DRAM** -> יותר cache + יכולת להחזיק יותר דפים שהם dirty ולהוציאם בכתיבות לדיסק.
  2. קצת ההערה לדיסק השתרף.

הרעיון: **لتכון FS שבו כל הכתובות יתבצעו בצורה סדרתית לבוקים רציפים.**

  - **אריך?**
  - ה-LSF יוכל רק לוג.
  - נתחזק באפר גודל של סגמנט וכנתוב אותו לדיסק ברגע שהוא שיתמלא.
  - סגמנטים – אזורים רציפים בדיסק המרכיבים את הלוג.
  - כתוב את הסגמנטים **בסוף** הלוג.

בשונה מהelog שבכתבנו לצורך לייצר crash consistency, הלוג של ה-LFS יהיה המקום היחיד שישמר את המידע שצריך להיבט (אין לו גיבוי).

בשונה מהכתבות שיהיו רציפות וגדלות, הקראות לא יהיו כ אלה. ויצטרכו "לחפש" את המידע.

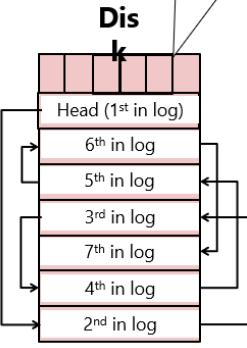
**פירוט לגבי אופן ביצוע הכתובות**

  - הכתובות יהיו בתצורת **write on copy**.
  - ככל פעם נכתב את המידע העדכני עבור inode בלבדarlo נעדכן רק את הצבעות אליו, מבלי למחוק את המידע הקודם והצבעות שהיו אליו.
  - אין נדע אייפה נמצאות הצבעות העדכניות של inode בלבדarlo בלהו.
  - נתחזק מערך שיכיל את המיקום של הצבעה העדכנית (imap) של כל inode ב-**memory**.
  - בנוסף לאחיזתו ב-**memory** נשומר אותו גם ב-**checkpoint region (CR)**, נמצא בתחלת הדיסק) אותו נעדכן כל חצי דקה (או בזמן השקל למודרנו מוגבה מאוד!) או באשר המחשב יוד בקרה מסודרת.
  - על ידי שימוש ב-**imap** אנחנו מצליחים לשמור על אותו מס' של גישות לדיסק בעת קריאה:
    1. גישה ל header |: מצאת המיקום של inode בלבדarlo בדיסק.
    2. גישה לדיסק: מצאת המיקום של data-**page** בעזרת המידע שב-**inode**.
    3. גישה לדיסק: קראת ה-**data**.
  - **بعد לא שינוי** את **כמויות הגישות לדיסק** בעת קריאה, מס' הגישות בעת כתיבה השתנה לחולוטין.

## LFS Crash Recovery

- הבעייה היא שהעדכנים של ה-**CR** יתבצע אחורה זמן, ולכן יתכן שבעת הקריסה יש מידע שעדיין לא הספיק להישמר בו.
- נתחזק 2-CR-ים, כדי שפובל להתמודד גם עם המקרה שבו הקריסה הייתה בזמן עדכון ה-**CR**.
- נעדכן אותם לシリוגן (פעם את הראשון, פעם את השני, ועוד את השלישי...)
- בעליה לאחר קriseה – נבדוק האם מסונכרנים ביניהם.
- אם כן – נעדכן את המערך בזיכרון בעזרת אחד מהם.
- אם לא – נעדכן לפי המעודכן יותר מבין שנייהם.
- העדכון יתבצע כך:
  - ב-**CR** יהיה מצביע לSEGMENT האחרון שה-**CR** עודכן לפיו.
  - **roll back** – נעדכן על כל הSEGMENTS מהSEGMENT הבא אחריו ונעדכן לפי הצבעות בהם את המערך בזיכרון.
- **בכל SEGMENT** נתחזק **header** שנקרא **.summary**.
- מוביל תיאור של כלinode-**page** שבסegment.
- בכמה העדכון ב-**roll back** יעבור על ה-**.summary** מוביל לעבור על כל הSEGMENT.

## איך ה-LFS מנהל את המיקום בדיסק ?

- 
- The diagram illustrates the LFS disk structure. It shows a vertical stack of horizontal bars representing data pages. Above them, a series of colored boxes represent log entries. The first entry is labeled 'Head (1st in log)'. Subsequent entries are labeled '6th in log', '5th in log', '3rd in log', '7th in log', '4th in log', and '2nd in log'. Arrows point from each log entry box to its corresponding position in the data pages below. The word 'Dis' is written vertically next to the top of the log entries, and 'k' is written vertically next to the top of the data pages.
- הדיסק מחולק סטטית לSEGMENTS והלוג במבנה ע"י שרשרת של הSEGMENTS בראשימה מקוושרת.
  - ה-LFS מתחזק רשימה של SEGMENTS פנויים, אליהם יוצאים כתיבות הSEGMENTS הבאים.
  - היתרון באופן העבודה זהה: אם התוכן של SEGMENT לא מתעדכן (המידע שלו עדין תקף) אז הוא לא משנה את מקומו.

## LFS GC (= Garbage Collection)

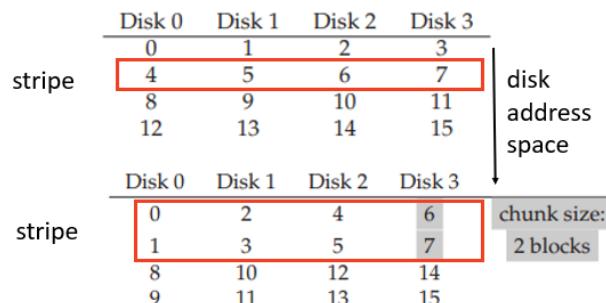
- תפקידו לדאוג שתמיד יהיה סגמנטים פנויים ב-list free.
- יעבדו בתור תהליך רקע (kernel thread) -> מתחילה על זמן של ה-CPU.
- אופן העבודה שלו:
  - עברור על כל הסגמנטים.
  - באשר יתקל בסegment עם מידע לא רלוונטי – יעתיק את המידע הרלוונטי בו לsegment חדש (לثور הסegment הפנוי הבא בו).
- במידת הצורך (אם inode שמצויב לאוטו מידע איינו חלק מהאותו segment שמוועתק) יוסיף inode חדש עם מצבייע אל המידע החדש בסegment שנכתב (מאחר וההצבעה היישנה אל המידע לא רלוונטית, כי מקום המידע השתנה).
- יסמן את segment החדש שהועתק ממנו המידע הרלוונטי בתור segment פנוי -> יוסיף אותוリスト free.
- כולל הקיריאות ובתיות של ה-GC יבוצעו בצורה סדרתית ורציפה, כי הוא קורא וכותב סגמנטים.

ב-all-in-the-LFS עובד טוב יותר מ-SSFs שمبرכעים הרבה בתיבות וקריאות קטנות.  
מצד שני, במצבים בהם הדיסק כמעט מלא, אז ה-GC צריך לסרוק הרבה סגמנטים.

- הרעין – לחת הרבה דיסקים ולבנות מהם התקן אחסון שיהיה יותר מהיר מכל אחד מהדיםקים שמרכיבים אותו וגם שייהיה יותר אמין.
- איך יהיה יותר מהיר? נוכל לקבל את העבודה של הדיסקים, כאשר נקבל רצף בקשות IO שմבוקשות בלוקים מדיסקים שונים.
- איך נשפר אמינות? נשכפל מידע – כתובות אותו על דיסקים שונים (יותר מפעם אחת בסה"כ). זו הכוונה ב-**redundancy**.
- ככל חוץ, ה-RAID נראה למחשב שלו הוא מחובר בתור התקן אחסון רגיל ולא zusätzlich של התקן אחסון.
- בפועל יש אלגוריתם חומרתי על ה-RAID או תוכני במודול של מערכת הפעלה שודאג לתרגום פקודות ה-IO לפקודות על התקן האחסון.

## RAID 0 - striping

- הבלוקים הלוגיים מפוזרים כמו בפס על פני הבלוקים הפיזיים בדיסקים.
- הפס (stripe) יכול להכיל בלוק אחד מכל דיסק/2 בלוקים מכל דיסק/ כמה שנבחרו.
- היתרון בשימוש בו הוא השימוש ביכולת העובודה המקבילה של הדיסקים.



- אין לו אמינות לבך שישמר מידע שנעלם יחד עם דיסק שקרס מאחר ואין לו גיבויים.

## RAID-0 performance

RAID with  $N$  disks; assume:

- Disk transfer rate  $S$  for sequential transfers
- Disk transfer rate  $R$  for random transfers
- Recall  $R \ll S$  for magnetic disks

Single request latency:

- Like a single disk

מדובר בקצב ולא בזמן

Sequential transfers:

- $N*S$

<-

Random transfers:

- $N*R$

## RAID 1 - mirroring

- שומר זוגות של דיסקים ב-mirror, המכילים בדיקות אוטומטיות מידע ומגבאים אחד את השני.

## RAID-1 performance

Single request latency:

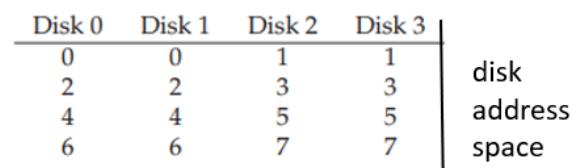
- Read: Like a single disk
- Write: Like a single disk, but a bit worse positioning time (worse of 2 disks)

Sequential writes:  $N/2 * S$

Sequential reads: also  $N/2 * S$

Random reads:  $N * R$

מדובר בקצב ולא בזמן ->



## RAID 4 – Parity

| Disk 0 | Disk 1 | Disk 2 | Disk 3 | Disk 4 |
|--------|--------|--------|--------|--------|
| 0      | 1      | 2      | 3      | P0     |
| 4      | 5      | 6      | 7      | P1     |
| 8      | 9      | 10     | 11     | P2     |
| 12     | 13     | 14     | 15     | P3     |

disk  
address  
space

- הרעין – להפחית את redundancy שהייתה ב-1 RAID.
- איך? אחד הדיסקים יתפקיד בתפקיד parity.
- הביט ה- $j$  בבלוק parity הוא 1 אם סכום הביטים ה- $j$  של כל הבלוקים התואמים לדיסקים האחרים הוא אי זוגי.

- בולם, הערך של כל ביט בבלוק parity הוא XOR של כל שאר הביטים המקבילים בסטריפ.
- ככה אם מ Abedים את אחד מביטי הדאטה אז ניתן לחשבו בעזרת ביט parity.
- הามינות שלוי היא עבר נפילה של בלוק אחד (במקרה הגורע).

ניתוח ביצועים:

- קריית בלוק בודד – כמו עם דיסק יחיד, כי הקריאה מתרגםת לקריאה של איזשהו בלוק ב-RAID.
- כתיבת בלוק בודד

- יש צורך לעדכן את בלוק parity
- בשבייל לעדכן אותו אין צורך לגשת לכל הבלוקים בסטריפ
- נחשבו לפיה:  $P_{\text{new}} = (P_{\text{old}} \text{ XOR } B_{\text{new}} \text{ XOR } B_{\text{old}})$
- לכן, בסה"כ 4 פעולות IO
- מתווכן, ניתן לבצע כל 2 במקביל
- לכן יש 2 פעולות IO

### כתיבת סדרתית

- ניתן לחשב את parity תוו"כ לאחר וככל הסטריפ נכתב
- ניתן לכתוב אל parity במקביל
- לכן הקצב הוא **S\*(1-N)**
- הסבר: S הוא הקצב של כתיבה סדרתית עברו כל אחד מהדיסקים לאחר ובולוק parity הוא רק גיבוי של המידע, אזי הוא "לא מקדם" אותו בכתיבה המידע, אך יש רק 1-N דיסקים שותריםם בכתיבה המידע.

### קרייה סדרתית

- בדיק במו 0 RAID רק שיש 1-N דיסקים עם מידע.
- S – הקצב של קרייה סדרתית.
- לכן, מתקובל קצב: **S\*(1-N)**

### קרייה אקראית

- בדיק במו 0 RAID רק שיש 1-N דיסקים עם מידע.
- R – הקצב של קרייה אקראית.
- לכן, מתקובל קצב: **R\*(1-N)**

### כתיבת אקראית

- כל קרייה צריכה לכתוב גם לדיסק שלה וגם לדיסק parity
- הופך את דיסק parity לצוואר בקבוק שמנוע מקבילות.
- בולם, בעוד כל דיסק צריך לטפל בכתיבה אחת לבולוק אקראי, דיסק parity צריך לטפל ב-1-N בתיות אקראיות.
- סך כל הגישות לא יסימנו עד שדיסק parity יסימן את שבודתו.
- בולם, הקצה מוכתב ע"י קצב עבودת בולוק parity.
- מאחר והקצב של קרייה אקראית הוא R, ובכל גישה לצורך בולוק parity גוררת 2 פעולות IO אזי נקבע קצב **2/R**.

- זהו גם הקצב הכללי, מאחר ובכפי שאמרנו, הצואר בקבוק הוא בדיסק parity.
- זהו מצב לא סקליבלי כי הקצב של כל המערכת תלוי בדיסק אחד ולא ניתן לשפר את הקצב אם נוסיף עוד דיסקים.

|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | P0 |
| 5  | 6  | 7  | P1 | 4  |
| 10 | 11 | P2 | 8  | 9  |
| 15 | P3 | 12 | 13 | 14 |
| P4 | 16 | 17 | 18 | 19 |

הרעין – לפזר את בלוקי parity כדי למנוע את צוואר הבקבוק על דיסק אחד.

כל התכונות שלו זהות למעט גישות אקרואיות:

- קריאה אקרואית
  - כל הדיסקים (N) מכילים מידע
  - R – הקצב של גישה אקרואית
  - $\text{סה"כ } R * N$
- כתיבה אקרואית
  - כל הדיסקים (N) מכילים מידע
  - R – הקצב של גישה אקרואית
  - כתיבה דורשת 2 פעולות O
  - לכן, הקצב בפועל הוא  $R/2$
  - $\text{סה"כ } R/2 * N$

## (בו זמניות) Concurrency

- ישנים סוגים שונים של בו זמניות. כולם מתרחשים כאשר יש משאב משותף בו כולם משתמשים.
- לדוגמא: concurrency למשאב של המעבד.

המקורה בו אנחנו נתעניין הוא **concurrency במרחב הכתובות של תוכנית** = במשתנים שלה, במבני הנתונים שלה.  
קוד לדוגמא שועל לגירום בעיות (נשתמש בו בתור דוגמא בהמשך):  
נניח שיש שני קטעי קוד שונים שרצים ומשתמשים באותו משתנה X.  
נניח שלאחר פקודה 1 בקוד האודם, הוא עצם, והקוד הבהיר תחילה להתבצע עד שסימן.

במקרה זהה הערך של X שונה ממה שהוא אמרו להיות אילו היו מבצעים את כל הקוד האודם ורק אחריו את הקוד הבהיר.

X = t1

X = t1 + 1

t2 = X

X = t2 + 1

- מצב זה יכול לקרות בעת טיפול ב-handlers-ים: הקוד של תחיליך מופסק, ובמקרה הקוד של ה-handler מ被执行 ווחזר לקוד המקורי שרך לאחר שה-handler סיים את עבדתו.

## Concurrency scenarios

- מחייבים תמיד בקורס שיש מעבד יחיד.

### signals

- באשר תחיליך רץ ומתקבל signal, מערכת הפעלה שומרת את מצב התהיליך שהופסק ומתיילה להריץ את ההנדל.
- גם התהיליך וגם ההנדל עושים שימוש באותו מרחב בתובות (כि ההנדל עלול לגשת למרחב הכתובות של התהיליך).
- בו זמניות במרחב הכתובות של תחיליך שאין חלק מהקרナル.**

### interrupts בזמן ריצת תחיליך של הקרナル

- באופן דומה כמו שקרה עם Signals, התהיליך הנוכחי יעצור את עבדתו וההנדל המתאים שמתפלט באינטראפט עלול לגשת לאותו מרחב בתובות.
- בו זמניות במרחב הכתובות של הקרナル.**
- חשוב להבחין בין מצב זה לבין **אין אינטראפט בזמן ריצת תחיליך שאין חלק מהקרナル**. כי במצב זהה מרחב הכתובות של התהיליך שהופסק שונה מרחב הכתובות שהקרナル משתמש בו בזמן הטיפול באינטראפט. אמנם הקוד של הקרナル ממופה במרחב הכתובות של התהיליך, אבל הוא נגש רק למי שmagiu בתור privileged.

### preemption של תחיליך על המעבד

- כלומר, מדובר במצב שבו המעבד החליט להפסיק תחיליך מסוים ולבצע context switch על מנת לבצע תחיליך אחר.
- לא מדובר בו זמניות כי לתהילכים שונים יש מרחב בתובות שונים.**
- אליא אם מדובר בשני תהילכים שэмפפים אותו פריים בזיכרון הפיזי ושניהם כותבים ל'יכרן.

### Kernel pre-emption

- מצב שבו קוד של הקרナル הופסק ע"י עצמו (למשל: בגלל timer interrupt). בשונה מה המצב הקודם של pre-emption שדיברנו לעיל – שבו קוד של **תחליך שאין חלק מהקרナル מופסק ע"י** הקרナル.
- לעומת ה-**non-pre-emption**, כאן **מדובר בו זמניות**, כי הקוד שחוור לביצוע אחרי ה-**pre-emption** הוא גם קוד של הקרナル ועשוי לגעת באותו משתנים של הקוד שהופסק.
- יש מערכות הפעלה שונות מצב זהה בקר שהן לא מספקות אותו מלכתחילה.
  - בLINNOKS אפשר לבחור את 2 האפשרויות.
- יש חסרונות לבכל אחת מהבחירה:

| non preemptive kernel                                                                                                    | preemptive kernel                                                                                                         |
|--------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| חסרון: יכול להיבננס לlolאות אינסופיות<br>יתרון: אין בעיות של בו זמניות של solution pre-emption kernel שצורך לשימוש אליון | יתרון: יותר RESPONSIFI פועלות O!, לא יכול להיבננס לlolאות אינסופיות<br>חסרון: יש בעיות של בו זמניות שצורך לשימוש אליון לב |

- חוטים = חישובים סדרתיים (קטעי קוד) שרצים באותו תהיליך. כל חוט יבצע פונקציה.
- תהיליך שרש יכול ליצור חוטים חדשים שיירצו למרחב הכתובות שלו.
- (... f ...) – קריית מערכת שמייצרת חוט חדש.
  - f – הפונקציה שהחוט יבצע.
- 2 הבדלים בין fork ויצירה של חוט:
  1. החוט מקבל את הקטעי קוד שהוא הולך להריץ בשונה מ-fork ששנו התהיליכים ויריצו אותו קטע קוד.
  2. **חוטים רצים באותו מרחב כתובות בעודו שנוצר ב-fork** מרחיב כתובות חדש.
- מערכת הפעלה מנהלת את הריצה של כל החוטים במערכת ע"י הגדרה של חוט במעין תהיליך מיוחד. מיוחד בכך בכך שחותמים של אותו תהיליך משתפים את כל הדברים של התהיליך **שייצר אותם, למעט מצב המעבד שלהם** (התוכן של הריגיטרים של המעבד).
- כאמור, **חותמים משתפים ביניהם את מצב התהיליך**:
  - מרחב הכתובות (זיכרון) של התהיליך שאליו הם שייכים. לפחות יש אותו PCB, pt, fds.
  - כל שאר המצב של התהיליך. למשל: .signal handlers
  - ועוד ...
- כל שינוי שhot עשוה במצב התהיליך מיד משפיע על שאר התהיליך = שאר החוטים.
- **בשתייה מקבל זמן מעבד אז יש חוט ספציפי בתהיליך שמקבל את הזמן מעבד**, ואם לתהיליך אין חוטים אז ניתן להסתכל עליו בתור תהיליך עם חוט בודד.
- כאמור, כל הטרמינולוגיה שדיברנו עליה בהקשר של התהיליכים שמקבלים זמן ריצה על המעבד, בעצם מדובר על חוטים ולא על **טהיליכים**.

## Multiple CPU

- ביום יש מס' מעבדים למחשב אחד.
- בו זמניות בעבודה עם ריבוי מעבדים (= ליבות, קונספוטואלית. בפועל מבחנים בין ליבות למעבדים בכך שמעבד נחשב לצ'יף שייתכן ומוביל מס' ליבות. וליבה זה רכיב על צ'יף שבפועל מתפרק כמו המעבד, לפי איך שהגדרכנו שמעבד מתפרק).
- אין בו זמניות בין התהיליכים השונים ב-user-mode כי לתהיליכים שונים ב-user-mode יש מרחב כתובות שונים.
- אלא אם מדובר בשני התהיליכים שמניפים את אותו פרוייקט בזיכרון הפיזי ושניהם כותבים לזכרון.
- תמיד יש בו זמניות בין התהיליכים השונים של ה kernell, כי הקוד של ה kernel יכול לרוץ במקביל על כמה מעבדים.
- יכולה להיות בו זמניות בין מרחב כתובות של חוטים שונים (אחר והחותמים חולקים את מרחב הכתובות של התהיליך).

## Concurrency Programming

- מעבשים נתיחס למונח concurrency בטור concurrency במרחב הכתובות.
- נרצה לייצר מצב של concurrency שמתיחס בצורה זהה לכל סוגי concurrency, מבלתי להבחין בין המצבים השונים בגינם ה- concurrency קורה.
- המודל שלנו יתיחס לכל הסוגים השונים של concurrency concurrency בטור concurrency בין קטעי קוד שונים.
- המודל שלנו יוכל גם איישחו scheduler שיחיליט בכל רגע מי יהיה הקוד שירוץ על המעבד ויבצע פוקודה על הזיכרון.
- נקרא לו בשם **memory scheduler**. חשוב להבין שזו לא ישות אמיתית במערכת הפעלה ולא מדובר ב-user-mode של מערכת הפעלה שדיברנו עליו.
- לא ניתן להניח הנחות על memory scheduler. הוא יכול לעשות מה שבא לא. אפשר להסתכל עליו בתור יריד דוני שלא ניתן לצפות כיצד יחלק את התורות בין החוטים.
- אנחנו עובדים במודל שמתנהג אליו יש רק ליבה אחת, למרות שבפועל יתכנו מס' ליבות, לאחר מכן קיימת שיקולות בנוכנות של מקביליות על ליבה בודדת לעומת מס' ליבות (חשוב לדגיש שמדובר בנוכנות, ולא בזמן שלוקח לבצע את אותן קוד על מעבד עם ליבה אחרת לעומת מספר ליבות).
- יש להגדיר לקומפיילר עבור כל משתנה מתי אפשר לעשות עליון concurrency ומתי לא, כדי שעבור משתנים שונים לעשות concurrency ולהבין לנו את העבודה עליהם מקטע קוד שונה – **לא יתבצעו אופטימיזציות** שלא מודעות לקטעי קוד אחרים שיכולים לשנות את ערכו של המשנה.

פתרונות אפשרי – volatile –

- הבעיה – זה לא מספיק טוב, כי יתכונו אופטימיזציות קומפיאליום שוגם משנות את הסדר בין הפקודות מבלתי להתחשב בכך.
- שימושים מסוימים יכולים להשתנות גם ע"י חוטים אחרים.
- משתנה שחווה concurrency = משתנה שיש אליו גישה משני חוטים שונים שלפחות אחת מהן היא בתיבה.
- הגדרה שcolaה: משתנה שחווה concurrency = משתנה שיכל להיות מעורב ב-data race.
- ב-C מגדירים משתנה זהה בתור atomic.
- ב-11 C תוכנית נחשבת ללא נוכנה אם יש משתנה שיכל לחוות data race ולא מוגדר בתור atomic.
- גישות למשתנים שמוגדרים בתור atomic יהיו יותר איטיות ולכנן לא מומלץ להגדיר את כל המשתנים ככלה.

## Locks

### Synchronization Primitives

- קיימים API-ים שמאפשרים למקרה מסוים לרצוץ מקבילית ובכך לאפשר אוטומיות – קטעי קוד שיורכו בשלהם ללא הפסקה להרצאת קטעי קוד אחרים.
- קיימים פרימיטיבים נוספים שעושים דברים נוספים: העברת מידע בין חוט לחוט, לחכות לתנאי מסוים, בפיית סדר בעובדה בין החוטים ועוד ...
- בשביל לאפשר פעולות אוטומיות ניתן ליצור מצב של התעלמות מ-interrupts ע"י יצירת מנגנון שמנע הרצת הנדרלים של interrupts מיד עם קבלתם. במקרה, ההנדרים יוכנסו לתור. לאחר סיום הפעולות האוטומיות ההנדרים יקרו.
- הבעיה היא שמנעה של interrupts רלוונטי רק עבור המעבד הנוכחי ולא עבור מעבד אחר.
- הפתרון: שימוש במנועול.

## Lock

- השתמש בנעילה כדי להבטיח אוטומיות.
- מצד שני, לא נגען את כל הקוד שלנו באותו מנגול כי בכיה נמנע מקביליםות.
- במקום זאת השתמש במנועולים על מנת לאפשר:
  - overlapping – עבודה בזמן המתנה.
  - מקביליםות של עבודה עם מס' ליבות.
- מנגול הוא אובייקט תוכני שתומך ב-2 פונקציות:
  1. () lock = (acquire)
  2. () unlock = (release)

```
Lock L;
Lock (&L);
...
Unlock(&L);
```

- הרעיון של מנגול הוא להגן על קטע שנקרא "הקטע הקרייטי" = הקטע בין הנעילה לשחרור.
- מנגעה הדדית (mutual exclusion) = לכל היוצר חוט אחד או קוד אחד יכול להריץ את הקטע הקרייטי בו זמנית.

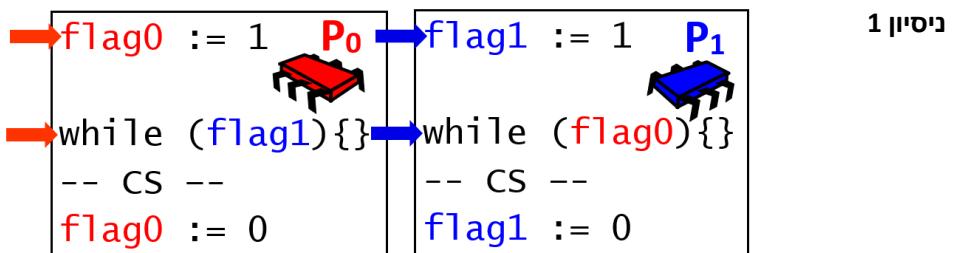
| C1           | C2          |
|--------------|-------------|
| Lock(&L)     | Lock(&L')   |
| t1 = X       | t2 = X      |
| ✓ X = t1 + 1 | X = t2 + 1  |
| Unlock(&L)   | Unlock(&L') |

### חוקי נעילות

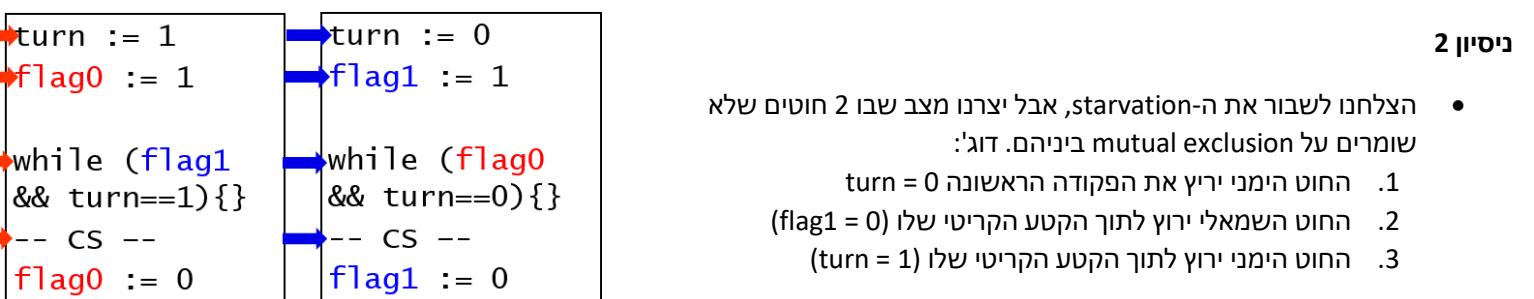
- לא ניתן לעשות lock-unlock למנועול שלא אתה נעלת.
- לא ניתן לעשות lock-unlock למנועול שאף אחד לא מחזיק בו.
- לא בכל המנגוליםאפשרות נעילה ורקסיבית (קריאה ל-lock ואז בתוך הקטע הקרייטי, קריאה ל-lock גוסף על מנגול אחר).
- משתמשים שניגשים אליהם רק תחת נעילותם הם משתמשים שלא יכול להתקיים לגביהם data race וכן לא צריך להגדיר אותם בתור atomic.

## אין מנעלים עובדים?

- נעבור על דוגמא של peterson's lock.
- מכחיהם שהנעה תהיה עברו 2 חוטים/תהליכיים בלבד.



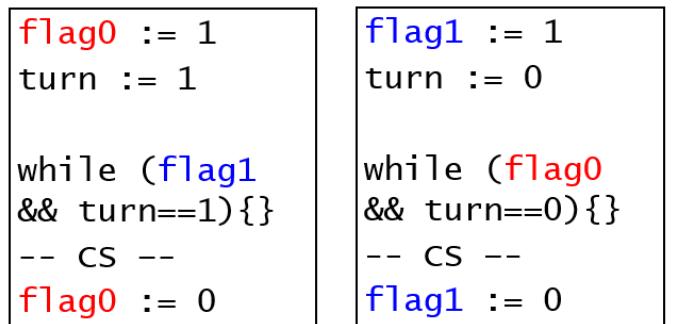
- הבעיה: יתכן שנגע במצב של deadlock אם, למשל, תבוצע השורה הראשונה בבלוק השמאלי, אך יעצור הביצוע של הבלוק השמאלי, תבוצע השורה הראשונה של הבלוק הימני, במצב שבו אף אחד מהחוטים לא יוכל להמשיך.



- הצלחנו לשבור את ה-starvation, אבל יצרנו מצב שבו 2 חוטים שלא שומרים על mutual exclusion ביןיהם. דוג' :
  1. החוט הימני יירץ את הפקודה הראשונה ( $turn = 0$ )
  2. החוט השמאלי ירוץ לתוך הקטע הקרייטי שלו ( $(flag1 = 0)$ )
  3. החוט הימני ירוץ לתוך הקטע הקרייטי שלו ( $(turn = 1)$ )

פתרון: נשנה את הסדר בין הפקודה הראשונה והשנייה

- זהו הפתרון הסופי של peterson's algorithm



- הוכח בכיתה שהוא שומר על mutual exclusion.

## Locks vs. disabling interrupts

- רלוונטי רק למנעלים בטור ה kernell כי תהליכיים לא יכולים לנטרל אינטראפטים.
- אם לא נתמך ב-disabling interrupts אז אנחנו עלולים לעבור לكونטיקט של המדור בזמן שאנחנו בקטע הקרייטי של מנעל, בשונה מההתנהגות שהיאנו מוצפים לה.
- לכן, מנעלים בkernel צריכים לתמוך ב-disabling interrupts.
- בטור ה kernell של Linux יש 2 מימושי מנעלים:
  - spin\_lock - סוג אינטראפטים.
  - mutex - לא סוג אינטראפטים ו אסור (לפי קונבנצייה) להשתמש בו בתוך הנדרלים.

- מצב של המערכת שבו אין אף חוט שיבול להשלים את הפעולה שלו, בגין חוט מחייב שיתקיים איזשהו תנאי שלא יתקיים לעולם.
- מנעול שימושי חייב להבטיח 2 תכונות progress כד' שיהיה מנעול שימושי:
  1. **deadlock freedom** – אם יש איזשהו חוט שמנסה בל' הפסקה להיבנס לקטע קרייטי במשהו ולא מצילח, אז חייב להיות איזשהו חוט שמצילח להיבנס לאזרור הקרייטי או כבר נמצא בו.
  2. **starvation freedom** – מבטיח שלא יתכן שיש איזשהו חוט שינסה להיבנס לקטע הקרייטי מספר אינסופי של צעדים ולעולם לא יצליח כי כל הזמן יהיה שם חוט אחר, אלא מובטח שאחריו מס' סופי של ניסיונות הוא יצליח.

## Resource Deadlock

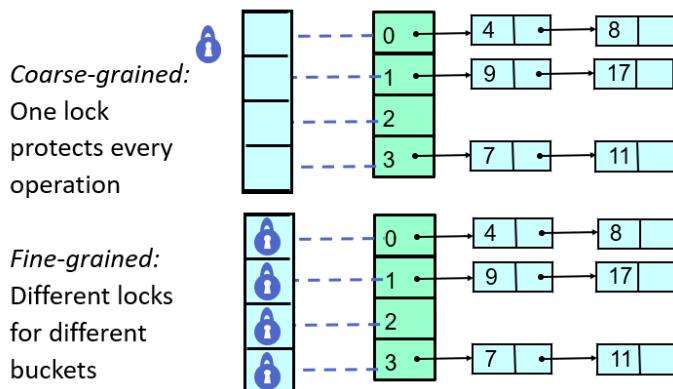
- מצב של המערכת שבו כל חוט מחייב לשחרר של משאב אחר מחזיק בר' שבסומו של דבר נוצרת תלות מעגלית שגורמת להמתנה אינסופית.
- למשל: עבור המשאב של אזרור קרייטי והקוד הבא:
  1. החוט הבהיר תופס את המנגול L'.
  2. החוט האדום תופס את המנגול L.
  3. החוט הבהיר מנסה ולא מצילח לתופס את המנגול L.
  4. החוט האדום מנסה ולא מצילח לתופס את המנגול L'.
- resource deadlock מרכיב מ-4 תנאים:
  1. יש משאבים שהשימוש בהם אסקלוסיבי.
  2. hold and wait – חוטים מחזיקים משאב ומנסים לתפוס בעלות (להשתמש) במשאב אחר.
  3. אין מנגנון pre-emption שיכל להסיר את הבעלות על משאב כלשהו מהבעלים שלו (רק הבעלים של המשאב יכול לשחרר את הבעלות).
  4. Circular wait – תלות מעגלית.
- ישנן מס' דרכים למניעת deadlock: **resource deadlock avoidance**
  1. **Lock ordering**
    - שיטה שמנסה למנוע תלות מעגלית.
    - מגדירים סדר שבו חיבבת להתבצע הנעילה של מנעולים וכל סדר אחר לא מתאפשר.
    - נעשית מסובכת ככל שהקוד גדול.
  2. **Avoiding no-pre-emption**
    - חוט שלא מצילח לתפוס משאב כלשהו, מועותר על כל המשאים שהוא מחזיק ברגע ומנסה לתפוס את הכל מהתחלה.
    - בשביל זה יש להשתמש ב-**Trylock**: החוט ינסה לנעלם ואם יצליח אז ישיג את הנעילה (ויחזר אמת), ואם יכשל אז הוא יוכל לשחרר את הנעילות האחרות שלו ולנסות מהתחלה.
    - הבעיה היא שהטבינה הזאת לא מבטיחה התקדמות, כי יתכן שלא יצליח לא נסיג את הנעילה – המצב הזה נקרא **livelock**
    - – כל החוטים עובדים אך בסה"כ לא מצילחים להתקדם מעבר לנקודה מסוימת.
    - מצב של deadlock עדיף על livelock כי יותר קשה לזהות את ה-livelock ולטפל בו.

## חוק אמדל

- אפשר לחשב את speedup
- נתון
  - זמן ביצוע ישן באורך 1
  - שיפור ק מהזמן פ' י'

$$speedup = \frac{1}{1 - p + \frac{p}{s}}$$

- בשנורצתה לנעול מושהו גדוֹל נשאָפּ לחלק אותו לחלקים קטנים ולנעול בל אחד מהם.
- בכיה נאָפּשֶׂר עבודה מקבילית.
- – כאשר משתמשים במנעולים ייחיד לנעול את כל האלמנט. בולם, **כל** גישה אליו תצריך געילה של אותו מנעול.
- – כאשר משתמשים במנעולים שונים עבור אותו אלמנט. בולם, גישות שונות ישמשו במנעולים שונים.



- לכן, כדי לנצל מקבילות טובות, משתמש ב-**fine grained locking** – חישובים שמאוד קל למקבל.
- embarrassing parallel

### IMPLEMENTATIONS PARALLELISMS OF BENCHMARKS

- האלגוריתם של פיטרסון לוקח זמן ומקום לינאריים ביחס למס' החוטים (בי' יש משתנה לכל חוט – דגל, ובשרוצים לנעול, אז צריך לבדוק אותו אצל כלום)
- כמו כן, הוא מבצע **busy-waiting**, בזמן שהוא מנסה להיבנס לחלק הקרייטי.

נפתר את בעיית הייעולות..

### Atomic RMW (read modify write) Instructions

- בנגד לפקודות שהכרנו, הפקודות האלו עושות גם כתיבה וגם קדאה **בצורה אטומית**.
- הרעיון הוא ביצוע יותר מפעולה אחת בצורה אטומית מבל' לתפוס מנעולים.

```
atomic rmw-op(int* addr) {
 v = *addr;
 *addr = f(v);
 return v;
}
```

מבנה כללי:

```
TAS(int *x) // Test&Set
{ t = *x; *x = 1; return t; }
```

**Test & Set – TAS**

```
SWAP(int *x, w)
{ t = *x; *x = w; return t; }
```

**SWAP**

```
CAS(int *x, old, new) // Compare & Set
{ t = *x; if (t == old) { *x = new; }
 return t; }
```

**Compare & Set – CAS**

## TAS lock

```

atomic_bool L;

lock() {
 while (atomic_flag_test_and_set(&L)) {
 // spin
 }
}

unlock() {
 L = 0;
}

```

This is known as a **spin lock**.  
The code spins (continues executing) until the lock is successfully taken.

14

### מימוש של מנעול בעזרת פעולות RMW

- משתמש במשתנה אטומי בוליאני שיקבל ערך 1 אם הוא תפוס ע"י איזשהו חוט, אחרת – הוא יהיה 0.

(`atomic_flag_test_and_set()` – פונקציית RMW אטומית שמחזירה 0 באשר הצלחה לבצע השמה לערך המשנה שספק לה כפרמטר).

זהו מנעול **spin-lock** כי הוא מנסה לנעול עד אשר הוא מצליח. הוא **mutual exclusion** כי לא יתאפשר 2 חוטים שמריצים את הקטע הקרייטי.

הוא מבטיח **deadlock-freedom**, כי מישחו חייב להצליח להשיג את הנעילה. הוא ייעיל כאשר אין תחרות על הנעילה – אפשר לתפוס אותו בפעם אחת בפקודת `test & set`.

- הוא לא **starvation-free** כי יתכן שיש חוט שייהי ביש מזמן ולעולם לא יצליח להשיג את הנעילה.
- הוא ייעיל כאשר אין חוט שחייב זמן רב לחשוף בשבייל להשיג נעליה
- ואשר יש תחרות אין חוט על הזמן שחייב יכול לחכות בשבייל להשיג נעליה
- 
- 
- 
- 
- 

### נפתר את בעיית ה-`wait busy` ..

- בעיתת ה-`wait busy` לא טובה מ-2 בחרינות:

1. בזבוז זמן של המעבד על המנתנה.

2. אי שוויון בין החוטים – יתכן שיש חוט שלא יצליח לנעול אף פעם.

Instead of spinning, context switch

```

lock() {
 while (TAS(&L))
 yield_cpu();
}

```

Code relevant to both user-mode and kernel.  
In user-mode, requires a system call. Kernel code just calls scheduler.

בכדי למנוע את בזבוז זמן המעבד, חוט שמנסה לנעול ולא מצליח

יעשה **yield** ויזור על זמן המעבד שלו.

עדין בעיתתי, כי יתכן שגם אחרי שהחוט יתעורר הוא לא יצליח לנעול את המנעול.

במקום זאת, משתמש בהמתנה לתנאי מסוים.

נתחזק תור של כל המתנים וברגע שהמנעול יהיה מוכן לנעילה אז נעיר את ההבא בתור.

struct mutex, משתמש ב-`spin_lock`.

```

lock(mutex *m) {
 spin_lock(&m->s); // for example, TAS
 if (!m->mtx) { // fast path
 m->mtx = 1;
 spin_unlock(&m->s);
 return;
 }
 queue_add(m->q, me);
 spin_unlock(&m->s);
 sleep(); // wait for wakeup
}

```

Pointer to thread struct  
Mark process as sleeping and context switch.  
Scheduler will not schedule process again until someone wakes it

Why not inside spinlock?  
Would kill progress...

```

unlock(mutex *m) {
 spin_lock(&m->s);
 if (queue_empty(m->q)) {
 m->mtx = 0;
 spin_unlock(&m->s);
 return;
 }
 first = queue_deq(m->q);
 wakeup(first);
 spin_unlock(&m->s);
}

```

21

- מה עושים כאשר מי שאנו רוצים להעיר לא הלך לישון?
- זהו מצב אפשרי אם, למשל, החוט שרצה עוצר רגע לפני-spin (אחרי spin\_unlock) ואז הזמן מעבר לחוט אחר.
- פתרון: נוסיף "פיהוק" (yawn) בטור-spin\_lock ברגע שאמ יסינו להעיר מישון שפיהוק אך לא הלך לישון אז לבטל את פעולה השינה שלו.

|                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>lock(mutex *m) {<br/>    spin_lock(&amp;m-&gt;s);<br/>    if (!m-&gt;mtx) {<br/>        m-&gt;mtx = 1;<br/>        spin_unlock(&amp;m-&gt;s);<br/>        return;<br/>    }<br/>    queue_add(m-&gt;q, me);<br/>    yawn();<br/>    spin_unlock(&amp;m-&gt;s);<br/>    sleep();<br/>}</code> | <code>unlock(mutex *m) {<br/>    spin_lock(&amp;m-&gt;s);<br/>    if (queue_empty(m-&gt;q)) {<br/>        m-&gt;mtx = 0;<br/>        spin_unlock(&amp;m-&gt;s);<br/>        return;<br/>    }<br/>    first = queue_deq(m-&gt;q);<br/>    wakeup(first);<br/>    spin_unlock(&amp;m-&gt;s);<br/>}</code> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Condition Variable

| P1                                                                                             | P2                                                                                     |
|------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <code>lock(&amp;L)<br/>if (!flag)<br/>    cond_wait(&amp;cv, &amp;L)<br/>unlock(&amp;L)</code> | <code>lock(&amp;L)<br/>flag = true;<br/>cond_signal(&amp;cv)<br/>unlock(&amp;L)</code> |

אובייקט שמאפשר סינכרון בין חוטים.

2 פונקציות עיקריות:

**cond\_wait(&cv, &L)**

שם את החוט שקרה לפונקציה בתור

המתינים.

משחרר את המניעול.

מכניס את החוט למצב שינה.

ברגע שימושים את החוט – נעל את המניעול מחדש.

**cond\_signal** – תוציא את החוט הראשון בתור המתינים ותעורר אותו.

### Producer/Consumer – Bounded Buffer

producers ממלאים את הבאר עם המתודה .Put.

consumers מוחזקים את הבאר עם המתודה .Get.

מידע שמדובר ע"י **תק** צריך להיות מטופל ע"י בדיק get אחד.

אחרי תפיסה מחדש של המניעול חייבים לעשות ולדציה לוודא שהמצב הוא אכן כמו שאנו חושבים שהוא (למשל: הבאר עדי מלא)

ישלח סיגナル לכל consumers שיש בעודה שמחכה שייעשו אותה.

מצב שבו שלחנו cond\_broadcast ועבשו יש הרבה consumers שהתעוררו. – thundering herd problem

פתרונות הסופי שהגענו אליו עברו

consumers,producers (מנחים

שקריה מוחזקת עד הבאר

וכתיבה מלאת אותו עד הסוף).

אם רוצים להימנע ממצב של

dead\_lockcond广播cond信号

(יש דוגמא בסוף המציגת לאי זה

יתכון).

#### Producer (put)

```
lock(&L)
while (state == FULL)
 cond_wait(&cvEmpty, &L)
memcp(buf,in) // fill
state = FULL
cond_signal(&cvFull)
unlock(&L)
```

#### Consumer (get)

```
lock(&L)
while (state != FULL)
 cond_wait(&cvFull, &L)
memcp(out,buf) //empty
state = EMPTY
cond_signal(&cvEmpty)
unlock(&L)
```

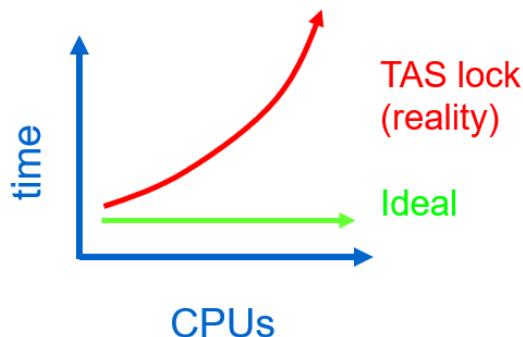
Two condition variables, one for each condition

## Read/Write Locking

- מנעול המבוסס על כך שקוד שרק קורא משתנים שמנעול מג עליהם יכולים לrhoץ במקביל.
- RWLock – אובייקט מנעול שתומך בשני סוגי של נעלות:
  1. () / ReadLock() / ReadUnlock() – עברו קוד שרק קורא את המשתנים.
  2. () – עברו קוד שמעדכן את המשתנים.

## Lock Contention

- תופעה שנוצרת כאשר יש הרבה דרישים למנעול ונוצר עליו contention (ויבוכח) אז זמן התपיסה של המנעול עולה.
- כמובן, זה יותר יקר לתפוס מנגנון שיש לו יותר דרישת, מאשר מנגנון שאף, ברגע, לא מנסה לתפוס אותו במקביל.
- במקרה זה, עברו קוד קרייטי קצר יחסית שיש הרבה ליבות שמנסות לגשת אליו, אנחנו נראה ירידה בביבוצים בכל שנגיד את במות הליבות.



## מימושThreads במערכת הפעלה

### clone (func, stack)

- קראת מערכת לצירת חוט חדש.
- או אפשר ליצור העתק של החוט המקורי כי אז הם יחלקו את אותה מחסנית.
- פרמטרים
  - func – בתובת לפונקציה שאוותה החוט ירץ.
  - stack – בתובת של המחסנית שהוקצתה עבור החוט.

## תחזקה של מרחב הכתובות של התהילין

- כאשר קוד מבצע mmap מתבצעת כניסה ל kernell -> נוצר VMA -> הכתובת שלו מוחזרת.
- בפעם הראשונה שהטהילין ניגש לבתוות שהוואזרה -> יתרחש page fault <- demand paging: יוקצה פריים פיזי עבור הדף הזה וווקצ עבورو מייפוי.
- בשעודיים עם חוטים, ניתן שחווט אחד יעשה את mmap, בעוד הפעולה של ה-demand paging תקרה ע"י גישה של חוט אחר.
- כמו כן, כל הקוד שה-OS משתמש בו בעדכון-hkן חייב להיות מוגן ע"י מנעלים.
- TLB Shootdown – פעולה שתפקידה לוודא שימושים את כל המיפויים על גבי כל התהיליכים (אחרי שמבצעת קריאת munmap) כדי שלא יקרה מצב של אחר מחיקת זיכרון אצל חוט מסוים המכילה לא מספיק להגיעה לשאר החוטים והם ינסו לגשת אותו זיכרון.

## Thread Groups

- קבוצה של החוטים ששוויכים לאותו תהילין.
- כך, שתבוצע פעולה על תהיליך (למשל: exit, kill) אז היא תבוצע על כל החוטים בקבוצה.

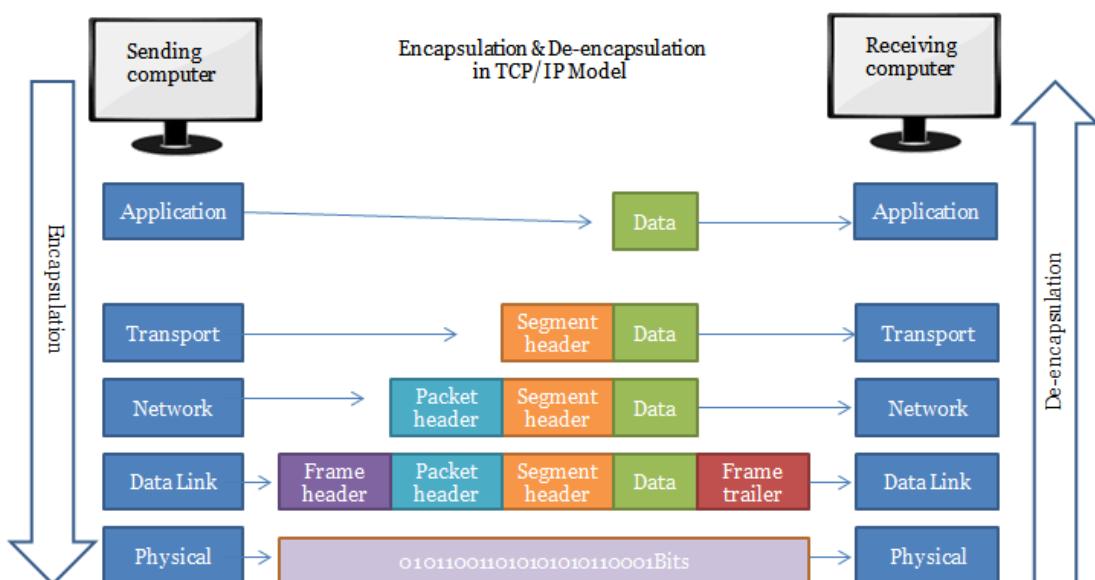
- Abstraction •
- Network Stack •

## Network Protocol

- פרוטוקול תקשורת הוא אוסף של כללים שונים ששני הצדדים שמתקשרים ביניהם מסכימים על הכללים ומתקשרים לפיו.
- פורמט מסוים של פקודות
- פרוטוקול פתוח** – הקוד שמממש את ה프וטוקול זמין לכלום כדי שיוכלו למשר שירותים כאלה בעצם (סוד ההצלחה של האינטרנט)
- פרוטוקול סגור** – פרוטוקולים שלא משתפים את אופן מימושם.
- לכל פרוטוקול מתחת האפליקציה יש מרחב שמות (address space) המאפשר לזהות את כל ה"שחקנים" ברשת.
- אין שחקן ברשת יקבל בתובת?

  1. עם קניית הרכיב כתובות תהיה צרובה עליון.
  2. ישות בלשוני שתקצה בתובת לכל שחקן.
  3. **dynamic assignment**

- לכל פרוטוקול יהיה header ובתוכו מידע המציג את כתובת המקור וכתובת היעד (במרחב כתובות של ה프וטוקול).
- Decapsulation & Encapsulation
- בכל شبكة יורד (Decapsulation) או מתווסף (Encapsulation) מידע מהשכבה הקודמת.



- התועלת העיקרית של זה היא אבטחה מאוחר ומידע שמודבר עטוף בשכבות כך שאינו חשוף לכל השכבות בדרך.

## Robustness Principle

- אי אפשר להניח שהצד השני פועל לפי ה프וטוקול
- Postel's law:** להיות שמרניים בינה ששלוחים וליברליים בינה שמצפים לקבל (לדוחות מידע שלא מגיע בפורמט שציפנו).

## מודל השכבות

- כל שכבה משתמש בפונקציונליות של השכבות מתחתיה כדי לספק פונקציונליות נוספת מתחכמתה.
- פרוטוקולים של תקשורת מוגדרים לפי שכבות.

| מספר | שם          | אופן העברת המידע | תפקיד                                                                                       | address space    | מכשירים/פרוטוקולים                  | קשרים נוספים                                           |
|------|-------------|------------------|---------------------------------------------------------------------------------------------|------------------|-------------------------------------|--------------------------------------------------------|
| 5    | Application | applications     | מספקים את הפונקציונליות שהתוכניות מספקות                                                    |                  | HTTP, HTTPS, FTP, DHCP              |                                                        |
| 4    | Transport   | segments         | מאפשרים תקשורת בין 2 תהליכיים שרצים על מחשבים                                               | port 16b         | TCP, UDP                            |                                                        |
| 3    | Network     | packets          | העברת מידע בין 2 מחשבים שאינם מחוברים באותו תווך פיזי (תווך שימוש בקשרים של המחשבים ביניהם) | IP 32b           | router                              | IP, ICMP, ping, netstat, MTU, fragmentation, subneting |
| 2    | Data Link   | frames           | מגדירים איך מחשבים שמחוברים באותו תווך פיזי יכולים להעביר מידע ביניהם                       | MAC 48b = 6B hex | bridge, switch, Ethernet, wifi, DSL | NIC                                                    |
| 1    | physical    | bits             | פותרים בעיות של העברת נתונים בין מכשירים פיזיים                                             |                  | hub, repeater, cables               |                                                        |

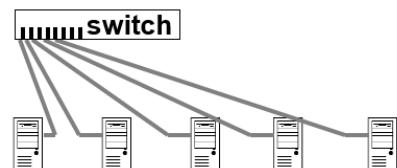
## שכבות 1 ו-2

### WIFI

- פרוטוקול השיר לשכבות 1 ו-2.
- גודל מקס' של פקודות - כ-2300 בתים.
- מהירות מקס' – 866Mb/sec
- תקשורת broadcas – הפקטה נשלחת לכלום ומיעוט destination תואם אליו – קורא אותה.
- בעיה: collision – כאשר 2 מחשבים מחליטים לשדר בו"ז.
- ניתן להתמודד עם אלגוריתם back-off (לאחר התנגשות כל מחשב יכחשה זמן רנדומלי עד שליחתה נוספת).
- המידע בתקשורת WIFI מוצפן.
- שונה מפרוטוקול הסלולר.

### Ethernet

- פרוטוקול השיר לשכבות 1 ו-2.
- גודל מקס' של פקודות - כ-1500 בתים.
- מהירות מקס' – 50MB/sec
- תקשורת broadcas – הפקטה נשלחת לכלום ומיעוט destination תואם אליו – קורא אותה.
- כדי לנצל את הפקודות, כל השחקנים מחוברים ל-switch המנתק את הפקודות על פי כתובת-h-MAC (שכבה 2).



### Modems (= modulator/de modulator), DSL (= digital subscriber line)

- מודם מבצע המרת נתונים דיגיטלי (אות דיגיטלי) לסיגナル אנלוגי (גודל המשנה בזמן שאינו מוגבל לטט סופי של מערכות כמו אות דיגיטלי).
- בעבר לא היה ניתן לשדר במקביל לשיחת טלפון, אך כיום זה אפשרי באמצעות DSL.
- DSL הוא פרוטוקול בשכבה 1.
- הביטים שמועברים ב-DSL מגעים למרכז טלפון גדול, ומשם הם מועברים בעזרת פרוטוקול PPP (Point to Point).
- PPP הוא פרוטוקול בשכבה 2 בעזרתו ניתן לבצע תקשורת בין התחkon שמשדר לבין מרכז הטלפוני.

## שכבה 3

## IP (= Internet Protocol)

- פרוטוקול השieur לשכבה 3.
- כל מכשיר באינטרנט מזוהה ע"י כתובת IP.
- ליתר דיוק, לכל NIC אחד יכול להיות בעל יותר מכטבת IP אחת (במוניהם router).
- מאפשר תקשורת בין hosts הנמצאים ברשתות (subnets) שונות.
- IP Address מחולקת ל-2 חלקים:

### .1. network portion(network ID)

- מזוהה של רשת ספציפית

**host portion** על בסיס ה-**network portion** **ללא** התייחסות ל-**host portion**

לכן, ראיווטרים שולחים פקודות למCAST על פי network portion בלבד – התאמאה בין כתובת היעד ל-

.address

### .2. host portion

- מזוהה של נק' קצה (end point) ברשת.

ראיווטרים ינווטו מידע לרשת יעד (**destination network**) ואחריו שהגיעו אל רשת היעד ינווטו אל ה-**host** הספציפי

בעזרת כטבota ה-IP (יתבצע בעזרת פרוטוקול ARP).

## Router

- הראינוור מעביר packets בין רשת אחת לאחורה בהתאם להtabפס על ה-routing table.
- אפשר לנו ל-**route**/לנווט בין רשתות.
- הראינוור מעביר אותנו בין כטבות IP, למשל "האינטלייגנציה" שלו מבוססת על כטבות IP בשונה מהסוייטץ' שה"אינטלייגנציה" שלו מבוססת על כטבות MAC.

## Routing Table

- טבלה המכילה את הניתובים מהנקודה הנוכחית.
- העמודות בטבלה:

| destination               | Gateway                              | mask           | metric                              | Iface                                                     |
|---------------------------|--------------------------------------|----------------|-------------------------------------|-----------------------------------------------------------|
| ה-subnet שרצים להגעה אליו | לאן צריך לקפוץ מבאן בשביל להגעה ליעד | מה אורך ה-host | priority<br>lower = better priority | ה-label של ה-subnet שצורך לצאת Drvco = ה-interface המתאים |

- אחת השורות בטבלה תהיה ה-**default route** (= default GW) – כאשר אין שורה בטבלה עבור ניתוב כלשהו -> נבחר ב-**route**.
- הבחירה בניתוב תבוצע באופן הבא (קונספטואלית):
  - נמיין את ה-subnet-ים לפי אורך ה-host בסדר יורד.
  - עבור על ה-subnet-ים וננסה לבצע התאמה.
  - עבור 2 subnet-ים עם אותו אורך host המתאימים ל-subnet היעד נבחר את זה עם ה-metric הנמוך יותר.

- המחלקות הן:  
classes A, B, C – unicast traffic . 1.

| class | 1 <sup>st</sup> octet binary | to       | from      | to              | # network bits | # host bits |
|-------|------------------------------|----------|-----------|-----------------|----------------|-------------|
| A     | 00000000                     | 01111111 | 0.0.0.0   | 127.255.255.255 | 8              | 24          |
| B     | 10000000                     | 10111111 | 128.0.0.0 | 191.255.255.255 | 16             | 16          |
| C     | 11000000                     | 11011111 | 192.0.0.0 | 223.255.255.255 | 24             | 8           |

- חלק ה-network תמיד משמאל, בעוד חלק ה-host תמיד מימין.

### Subnet Mask

- כתובות בגודל כתובות IP שביצורה נכל לקבוע איזה חלק בכתובת IP הוא ה-network ואיזה חלק הוא ה-host.
- 1 (בבינארי) ב-subnet mask <- .network
- 0 (בבינארי) ב-.host <- subnet mask
- למשל: 0.0.0.0 (11111111.11111111.11111111.00000000 =) 255.255.255.0 host הוא 8 ביטים.
- איך host יודע אם host אחר נמצא ברשת שלו?
- 1. משווה את אורך חלק ה-network שלו בכתובת ה-IP.
- 1.1. אם האורכים זהים
- 1.1.1. משווה את כתובות הרשות שלהם: אם זהות -> אותה הרשות. אחרת -> רשותות נפרדות.
- 1.2. אם האורכים שונים –> רשותות נפרדות

חשיבות: 2 host-ים שנמצאים באותו הרשות יכולים לתקשר ביניהם באופן ישיר. אחרת, הם צריכים לתקשר דרך gateway.

### DHCP = Dynamic Host Configuration Protocol

- פרוטוקול המשמש להקצאה של כתובות IP ייחודיות ל-host.
- בנוסף לכתובת IP, שירות DHCP בד"כ יספק למחשב גם subnet mask, כתובות שרת DNS, כתובות gateway כך שהמחשב יוכל לתקן ברשות ללא צורך בנזקים נוספים.

התקשורת בין hosts שיינטם באותו LAN תהיה בעזרת hops מנתב לננתב לאישור המספר בין host לננתב, ומנתב לננתב יהיה בשכבה הילינק על בסיס כתובות MAC. כלומר, ה-host ההתחלתי לא יודע את כתובות ה-MAC של host היעד, אבל הוא יודע את כתובות ה-MAC של הננתב הקרוב אליו וכן הלאה.

כאשר ננתב שנמצא באמצע השרשרת יקבל את הפקטה הוא יבין לפי הדר של שכבת ה-link שהיא מיועדת אליו, אך ברמת ה-IP היא אינה מיועדת אליו וכן יש להעביר אותה הלאה.

### DNS = Domain Name System/Service

- פרוטוקול ברמת האפליקציה שמתרגם שמות דומיין לכתובות IP.
- בכיה ניתן לעבד עם שמות נוחים שבפועל מתורגם לכתובות IP.
- ישנים כמה שרתי DNS. כאשר פונים לאחד מהם, אם הוא לא יודע את התשובה אז הוא מעביר את הבקשה לשרת אחר.
- רוב הארגונים מרימים DNS server ברכמתם. אותו שרת עושה caching לכתובות שכבר ניגשו אליו בעבר.
- גם hosts עושים caching לכתובות שכבר הגיעו אליהם בעבר

## NAT (=Network Address Translation)

- לאחר וכנתובות ה-IP החולו להיגמר איזו החילוטו לשימוש בנתובות פרטיות שאין תקיפות מחוץ לרשות המקומית.
- בכיה ניתן לתת לשני מכשרים שאינם באוטה הרשות את אותה נתובת IP.
- בכך שמכשיר עם נתובת פרטית יעביר פקעה מחוץ לרשות המקומית יש צורך בתרגום של נתובת IP מקומית לנתובת ציבורית לפני התקשרות החיצונית.

לחו בתרגום יתבצע ע"י ה-GW.

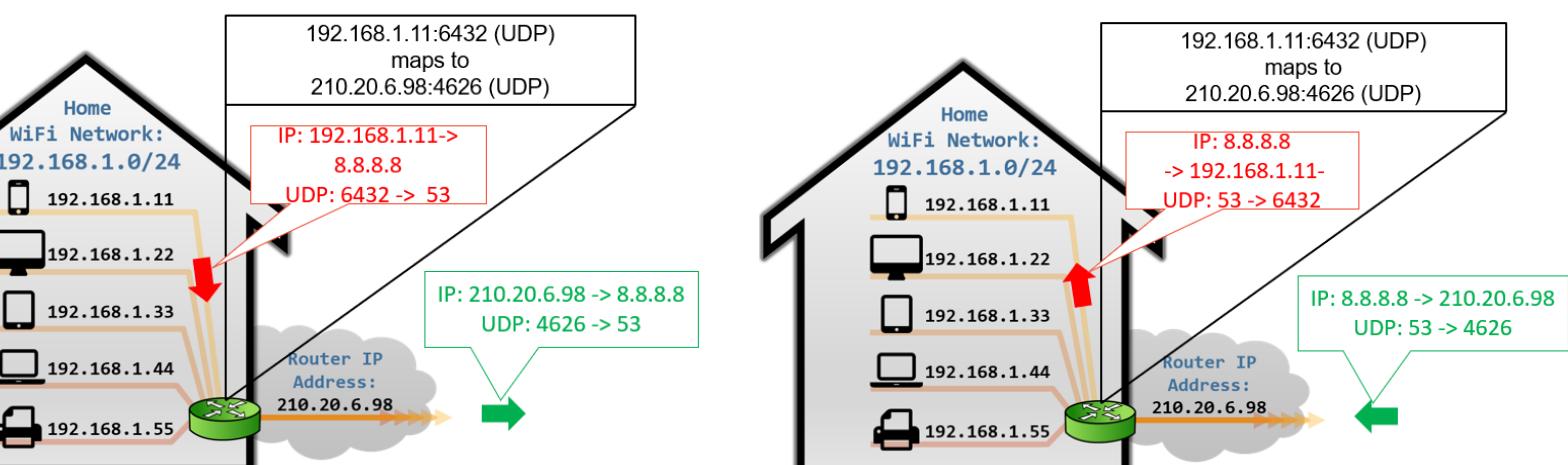
**מוחב הכתובות הפרטיות:**

- (10.0.0.0/8) 10.0.0.0 – 10.255.255.255
- (172.16.0.0/12) 172.16.0.0 – 172.31.255.255
- (192.168.0.0/16) 192.168.0.0 – 192.168.255.255

**איך זה מתבצע?**

1. מכשיר ברשות מבצע בקשה למכשיר באינטרנטן
2. ה-GW (בנהנחה שהוגדר שיבצע NATing) מחליף את נתובת IP המקורי של הבקשה בנתובת IP החיצונית שלו, ומחליף את port המקור ב-port פניו בלבדו.
3. בקבלת התשובה עבור הבקשה ה-GW מבצע את התהליך הפוך: מגדר את יעד התשובה להיות יעד הכתובת שביצע עבורה NATing.

The method of NAT is called Hide NAT



## Port

- communication endpoint = port
- זהו ה-*host*-*address space* הרלוונטי לשכבת ה-*transport*
- אורך 16 ביט
- 1024 הפורטאים הראשונים הם פורטים מוגדרים וקבועים מראש (עבור שירותים ספציפיים שהוגדרו מראש). ניתן למצוא אותם ב-*/etc/services*
- כדי להתחבר (*bind*) ל-port כלשהו מה-port-ים הידועים צריך הרשות root.
- מס' פורטים נפוצים:

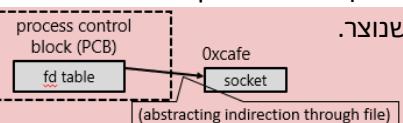
|          |        |
|----------|--------|
| Data FTP | 20     |
| FTP      | 21     |
| SSH      | 22     |
| Telnet   | 23     |
| DNS      | 53     |
| DHCP     | 68 ,67 |
| HTTP     | 80     |
| HTTPS    | 443    |

## Socket API

- ה-socket הוא **אבסטרקציה** שמערכת הפעלה מספקת לתהליכים כדי לעבוד מול פרוטוקולי ה-*transport*.
- תהליכים מייצרים socket-ים דרכם הם יכולים לתקשר עם תהליכים הנמצאים ב-host-ים אחרים (או באותו host), תחת פרוטוקולי ה-*transport*. למשל, מדובר בעוד צורה של IPC.
- **ה-socket מייצר את הקשר בין תהליך ל-port (binding).**
- 2 דרכים ליצור **binding**:

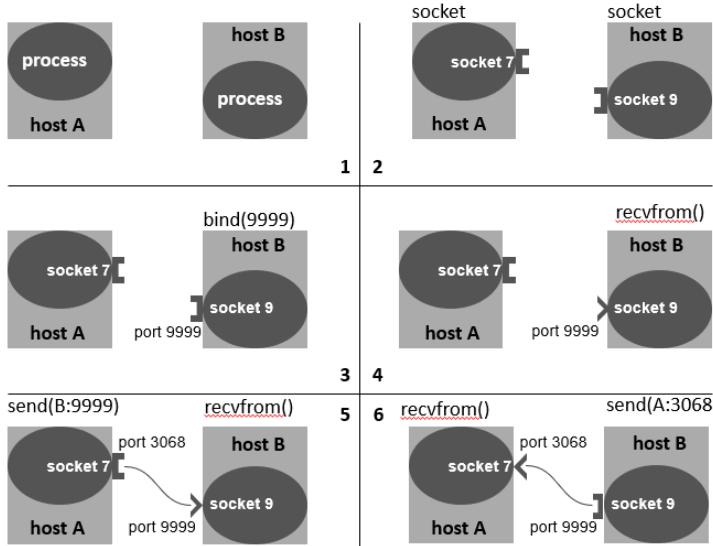
1. **מפורשות** – מס' ה-port יוגדר בהתאם למס' ה-port המוגדר עפ"י IANA. יבוצע ע"י קריאה ל-**(bind)**.
2. **לא מפורשת** – כאשר תהליך יוצר socket ומיד מתחיל לשЛОח עליו מידע, לא bind, אז ה-OS תקזה עבורו port פנוי.

באשר נוצר socket, נוצר struct file המוצבע ע"י ה-PCB של תהליך ומצוין על ה-socket החדש שנוצר.



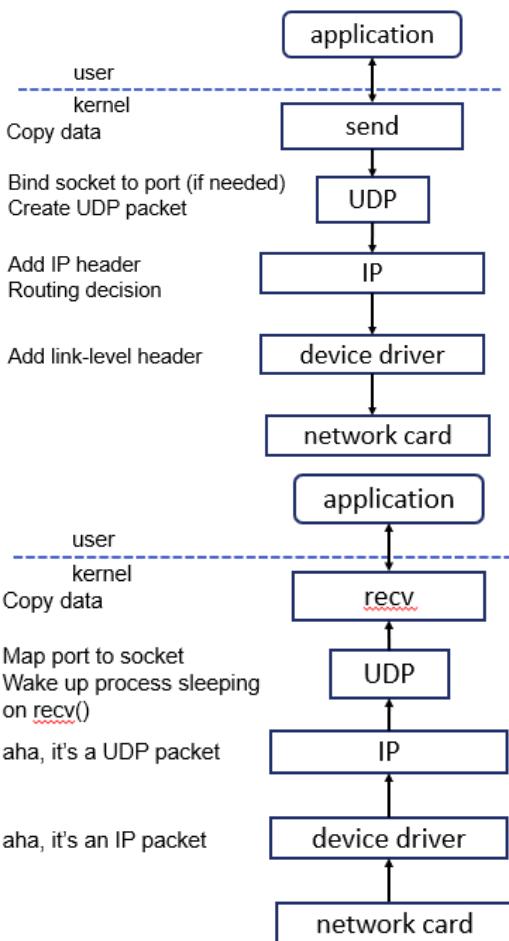
## UDP (= User Datagram Protocol)

- גודל ה-packets קטן יותר מוגדל החבילות המועברות ב-TCP (60% מגדלו)
- **lightweight connectionless**
- יותר שיליטה לגבי **מתי** המידע נשלח.
- יש לו מנגנון לדוחוי מתי packets לא הגיעו להגעה ליעדן, אך בשיזהה זאת הוא לא ינסה לתקן את זה ולא ישלח אותו מחדש.
- **出 of order delivery** – חבילות יכולות להגיע לא סדר המקורי שבו נשלחו.
- UDP לא מתחשב בעומס שיש על הרשות וינסה להעביר את ה-segments בכל מקורה.
- פחות אמין (reliable) לאור הנזודות לעיל -> עלול להיווצר מצב של packet drop שלא יטופל.
- **packet-oriented** : שימושי עבור אפליקציות לשילוח הודעות. למשל: email.

**ה-flow של שילוח פקחת UDP:**

•

1. התהילה מבצע syscall `send` על socket שלו שלו שגורם לבנисה ל-kernel.
2. הקוד שמטפל ב-syscall מעתיק את הבادر (ה-**data** שהועבר משבבת האפליקציה) שהועבר בפונט מהטהילה אל ה-socket.

**ה-flow של קבלת פקחת UDP:**

•

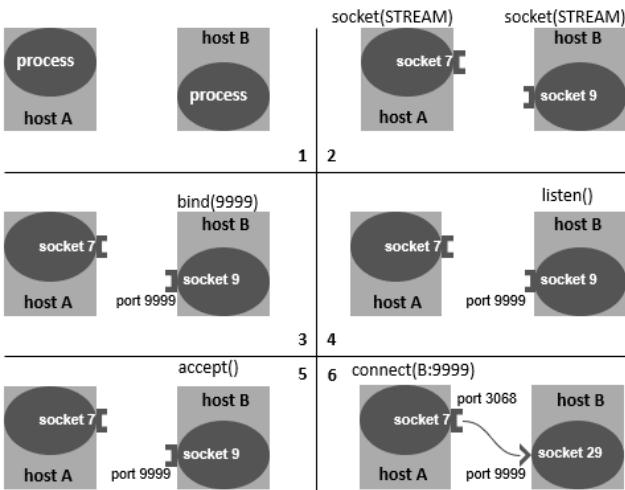
1. הפקחה מגיעה אל ברטיס הרשות. ברטיס הרשות מזהה שהפקחה מיועדת אליו לפי ה-**MAC address** ומעביר אותה אל ה-**driver**.
2. הקוד של פרוטוקול ה-link יקלף את ה-**frame header**, **frame trailer**, **packet header** ויברר אותו הקוד המתאים.
3. הקוד של פרוטוקול ה-**IP** יודא שהפקחה מיועדת אל המחשב (ולא דורשת ניתוב למחשב אחר), תקלף את ה-**packet header**, תזהה שפרוטוקול ה-**transport** הוא UDP ותעביר את הפקחה הקוד ה-**UDP**.
4. הקוד ה-**UDP** יבצע מייפוי מה-**port** מה-**socket** המתאים (שMOVED-IN ב-**destination header**) אל ה-**socket** המתאים ויברר אליו את המידע דרך בادر.
5. כאשר התהילה המתאימה ל-socket יבצע `recv` ביצעrecv את המידע וועבר אליו. אם התהילה ישן, אז הקוד ה-**UDP** יעיר אותו.

בפועל יתכן וה-flows יוצרים מבנים לא יבוצעו בצורה סינכרונית, אלא יופרדו ע"י buffers של producer/consumer。

## TCP (= Transmission Control Protocol)

- **Connection Oriented** – שליחת המידע מהשלוח (transmitter) יכולת להתבצע רק אם נוצר חיבור – session.
- **packet drop reliable** – מבטיח שפקטה שנשלחה תגיע ליעדה ולא יקרה packet drop.
- הסיבות האפשרות ל-**packet drop**:
  1. שגיאה בתווך הפייזי של ה-hop.
  2. עומס ברשת (יתר נפוץ).

### TCP Connection



חיבור TCP מזוהה ע"י  $\langle \text{IP}_A, \text{Port}_A, \text{IP}_B, \text{Port}_B \rangle : 4$  – tuple (IP, Port, IP, Port) המਸמנת שה-socket הוא פסיבי = מחכה לשיתבותו.

- **listen()** – פונקציה המسمנת שה-socket הוא פסיבי = מחכה לשיתבותו.
- **accept()** – מקבלת socket שהופעל עליו `listen()`.
- **connect()** – מושה block לתהליך עד שmagua בבקשת connect להתחברות ל-socket (מהצד השני).
- **connect()** – באשר מגיעה הבקשת, יוצרת socket חדש עבור הבקשת ומחזירה אותו כר שה-socket שהתקבל בפרמטר יישמר עבור יצירת חיבורים חדשים.
- **connect()** – מקבלת את בתובת ה-IP וה포רט של הצד השני ושולחת לשם בקשה ליצירת חיבור.

### Sockets vs. Ports

| Socket | State     | ID            |
|--------|-----------|---------------|
| 0xcafe | LISTEN    | B:9999 *      |
| 0xf00d | CONNECTED | B:9999 A:3068 |

הקוד של TCP ב-kernel מתחזק טבלה שמחזיקה את כל ה-**sockets**.  
מי שbowtz עבורי יוציא `accept` או `connect` מזוהה ע"י ה-**4-tuple**.

מי שbowtz עבורי `listen` בלבד מזוהה ע"י הזוג **port**, **IP** של המקבל.

עבור **השלוח** יש \* שימושוותה שביתן לקבל תüberה מכל IP ו-port מוביל לדריש כתובת ספציפית.  
ברגע שbowtz accept יוציא socket חדש עם הפרטים של השולח (IP ו-port).

חיבור TCP הוא **full duplex** – אפשר גם לבתוב וגם לקרוא ממנו.

### air להפוך את ה-stream לאמין?

- **ack/acknowledgment packets** – פקודות שהצד מקבל ישלח בחזרה אל השולח ע"מ להודיע לו שהפקות הגיעו לעדן.
- **RTT (= Round Trip Time)** – הזמן שעובר מרגע בין שליחת הפקטה עד לקבלת ה-ack.
- אם השולח לא קיבל ack אחרי פרק זמן בלשחו, למשל **time out** (זמן שמודדר דינאמית), אז הוא ישלח אותה שוב (retransmission).
- שדה ב-**header** של כל פקטה שנועד למעקב אחר הבטים שעשו להן Ack וכאליה שעדין לא עשו להן.

### buffering

- **.send buffer, receive buffer**: בצד המקבל ובצד השולח: buffers בצד המקבל write לא יהוו blocking, מאחר והקראה מיד תחזיר לאחר שהמידע יועתק ל-buffer.
- הפעם היחידה זהה לא יקרה כך תהיה באשר לא יהיה מספיק מקום ב-buffer כל המידע שרצינו לרשום, ואז write יצטרך לחכות עד לאחר סיום העברת המידע שבד-buffer בעזרת ה-TCP.
- גם המידע שיועבר ל-socket יכתוב לסופו ה-receive buffer. הkernel יפנה מידע מתחילת הבאför לסופו הapafrim שמועברים בפרמטר `read` על ה-socket.

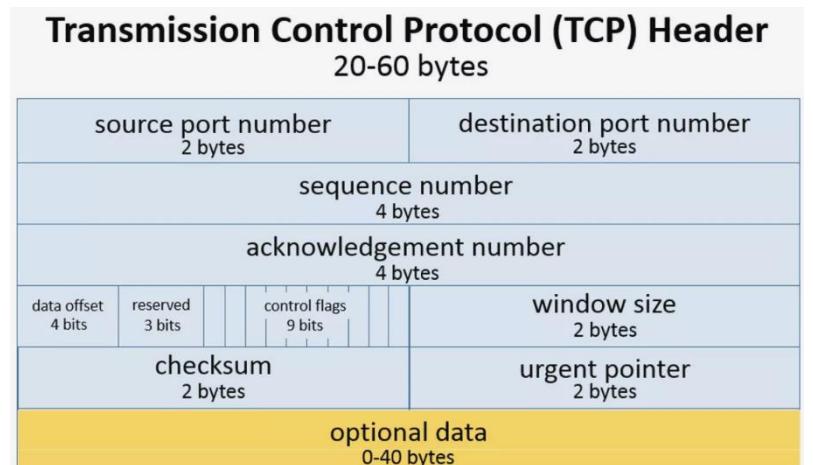
## windowing

- במקום לשלוח פקטה אחת בכל פעם, נשלח סדרה של פקודות (חלון) בהתאם למיקום הפניו ב-buffer.
- המידע על למיקום הפניו ב-buffer הגיע אלינו דרך ה-Ack שהוא ייחודי לנו.

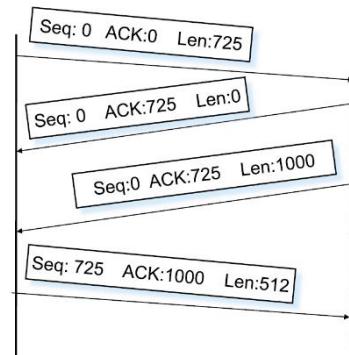
## sliding window

- השולח ישלח פקטה בכל פעם שיקבל Ack מה מקבל.
- **sliding window** = הבטים שהשולח "מחזיק באוויר" = נשלחו ועדין לא קיבל Ack.
- באשר הצד מקבל מגדיר את גודל החלון הוא בעצם מגדר את המיקום (בבטים) בו-stream שעד אליו הצד השולח יכול לשלוח נתונים. במקרה, **הצד השמאלי של החלון מתקדם בכל פעם שהשולח מקבל Ack והצד ימני מתקדם בכל פעם שה מקבל מגדר מחדש את גודל החלון** (או יותר דיוק את הנקודה הימני שלו).

## TCP Headers



- מצין את ה-offset ב-stream מה-source אל ה-destination שמתאים לבית הריאISON בפקטה הזאת.
- **sequence number**
- **acknowledgment number**
- מצין את ה-**sequence number** הבא מה-source מצפה לקבל מה-destination (בזכור TCP הוא אקסclusיבי).
- מתייחסים לשדה זה רק אם הדגל המתאים לו-**ack** דלוק ב-control flags.
- חשוב להבין שכל פקטה יכולה להכיל מידע על שני הכוונים בו זמנית. למשל: גם מידע הנשלח מה-source ל-destination וגם על ה-stream בכוון השני.
- **window size** – גודל החלון שבו ניתן יחסית ל-**ack**.
- לעומת זאת, בית האחרון מה-source יכול לקבל = window size + acknowledgment number.
- דוגמא:



- **delayed ack** – אופטימיזציה שבה מקבל הפקטה לא שולח ack מיד עם קבלתה ומচכה (עד 200 מילישניות) בתקווה שהפקטה הבאה שהוא אמר לשלוח (עם מידע לצד השני/window size חדש) תשלוח ו-ack ושלח אותה (piggybacking).

### יצירת חיבור - 3 way hands-shake

1. תחיליך באחד הצדדים קורא ל-connect.

2. OS - ה-OS שולחת פקטה אל הצד השני בשות FIN אצלת דלוק, והוא מכילה את ה-#seq הראשון של החיבור (מס' רנדומלי בכדי שתוקפים לא יוכל לנחש אותו).

3. אם בצד השני אין socket שמקשיב (בוצעה עליון קריאה ל-(listen) ל-port), אז הצד השני שולח בתגובה פקטה עם ack על ה-#seq שנשלח ודגלו RST דלוק (דגל שנשלח בעבר חיבור לא קיים).

4. SYN ACK - אם בצד השני יש socket שמקשיב (בוצעה עליון קריאה ל-(listen) ל-port), אז הצד השני שולח בתגובה פקטה עם ack על ה-#seq שנשלח ו-#seq ראשון (של החיבור בצד השני).

5. ACK – הקריאה ל-connect חוזרת. נשלח ack עם ה-#seq שנשלח מהצד השני.

6. נוצר חיבור גם בעבר הצד השני. ה-OS מייצרת socket חדש בעבר החיבור ומחייב אותו עם הקריאה ל-accept.

7. בפעם הראשונה, לא בוצעה הקריאה ל-accept על ה-socket אז הוא יוכנס לתור של accept של host-B מבלי host עשה על accept שם ל-accept. לאחר מכן מתחילה אחד ב-A host מנסה להתחבר לאוותה ה-port ב-B host מבליתו host עשה על accept של host-B מבין התהיליכים ב-A host איז התור של ה-listen יהיה מלא והסטודנט של TCP מתר לסרב לייצר את החיבור בעבר התהיליך השני ב-

(להחזר RST) Host A

### סיגרת חיבור

1. תחיליך הצד A קורא ל-close על ה-socket.

2. נשלחת פקטה מ-A ל-B עם FIN flag דלוק בה ו-#seq בהתאם למיקום הנוכחי ב-stream.

3. B שולח ל-A פקטה עם ack על ה-#seq שנשלח.

4. מתישוה, התהיליך הצד B קורא ל-close על ה-socket המתאים אליו (יקבל לפניו EOF על ניסיון קריאה מהאפר של המידע שמגיע אליו מ-A).

5. נשלחת פקטה מ-B ל-A עם FIN flag דלוק בה ו-#seq בהתאם למיקום הנוכחי ב-stream.

6. A שולח ל-B פקטה עם ack על ה-#seq שנשלח.

7. אם B היה מתעכבות עם הקריאה ל-close מעבר ל-timeout כלשהו אז מבני הנתונים הרלוונטיים אצל A היו מושמדים ובתגובה לפקטה FIN שנשלחה מ-B הייתה נשלחת פקטה RST מ-A.

אם לאחר סיגרת חיבור ופתיחת חיבור חדש מגיעה פקטה מהחיבור היישן אז אנחנו יכולים לקבל פקטה שלא הייתה אמורה להתקבל בחיבור החדש. בכדי למנוע זאת, על יוזם החיבור החדש לחכום 2 \* MSL (Maximum Segment Lifetime =) לפני יצירת החיבור החדש.

### congestion control

congestion window – ב모ות הבטים המקסימלית יכולה להיות באוויר בין שני תהיליכים שמתקשרים ב-TCP.

מס' הבטים המקסימלי שניין לקבל מבלי לעשות ack (הבטים באוויר) = המין בין ה-window ו-window.

### in ordered delivery

המידע יגיע ל-application לפי הסדר בו הוא היה אמור להישלח.

עם זאת, זה לא אומר שהמידע צריך להישלח לפי הסדר המקורי, מאחר וכיtan לסדר את המידע גם מצד מקבל לפניו שהוא מגיע ל-application

| TCP                                            | UDP                             |
|------------------------------------------------|---------------------------------|
| at least 20B header                            | lightweight                     |
| Connection Oriented<br>Delivery Acknowledgment | connectionless                  |
| in ordered delivery                            | out of order delivery           |
| congestion control                             | doesn't have congestion control |
| reliable                                       | not reliable                    |
| stream-oriented                                | packet-oriented                 |

**מטרות באבטחת מידע**

1. **Confidentiality** – המידע ישאר פרטי ורק מורשים יוכלו לקרוא אותו.

דוגמאות לימוש ב-OS:

- file permissions
- Virtual Memory
- anonymous page allocation

2. **Integrity** – אמינות המידע. רק גורמים מורשים יכולים לעדכן מידע.

דוגמאות לשימוש ב-OS:

- file permissions
- Virtual Memory

3. **Availability** – גורמים מורשים תמיד יכולים לקרוא/עדכן את המידע.

דוגמאות לשימוש ב-OS:

- CPU time sharing
- Scheduling

**TCB (= Trusted Computing Base)**

• החלק הקטן ביותר של המערכת שחייב להיות נכון כדי לשמור האבטחה תושג.

• כדי להשיג את מטרות האבטחה, ה-TCB צריך להיות בעל 2 תכונות:

1. **unbypassable** – כל הגישות צריכות לעבור דרכו.

2. **tamper resistant** – אסור שלתוכפים תהיה אפשרות לשולוט בו או לשנות את הדרך שבה הוא עובד.

השאיפה לכך שהוא יהיה **可信** ביותר נובעת מכך שאנו רוצים שהוא **verifiable**. כלומר, שיווה ניתן לוודא שאין בו באגים.

במערכות הפעלה מונוליטיות בל קוד שרש על הkernel רץ במצב **privileged**, וזה גורם לקשיי בורפיפיקציה של הקוד. לעומת זאת,

במערכות הפעלה לא מונוליטיות אפשר להוציא חלקים גדולים מהkernel שירצוי מחוץ לו עם מרחב בתובות פרטי במצב לא

**privileged** וכן להקטין את ה-TCB שלו בלבד. הביעה היא שהביטויים יורדים ככה.

**Logic Vulnerabilities**

• מגנן אבטחה שלא ממומש נכון.

• **Time of Check Time of Use = TOCTOU** – כאשר יכול להתקיים מצב של race condition בין הבדיקה של תנאי מסוים לבין השימוש בו.

- תוכנית נחשפת memory safe אם כל גישה שהיא מבצעת לזכרון היא גישה לדיכון ולידי.
- שפה היא memory safe אם היא לא מאפשרת memory safety violations (למשל: java).

### buffer overflow

- כאשר בתיבה למשתנה חורגת מחוץ לגבולותיו ודורשת ערבים של משתנים אחרים.
- עלול לגרום לו:

- command injection
- DoS
- Overwrite Variable Values
- Overwrite function pointers

• דוגמא לתקיפה:

1. התהיליך עושה לאוטו הקובץ (copy ms' עצום של פעמים עד שה-reference counter עוזה overflow ונהיה 1).
2. התהיליך עושה close לאחד ה-fd המשווים לאוטו הקובץ -> ה-reference counter של הקובץ יורד ל-0.
3. התהיליך עושה kernel-chown שאפשר לשחרר את הקובץ ועושה לו free.
4. ה-kernel ישמש בזיכרון שוחרר למטרות אחרות.
5. ה-kernel ישתמש בזיכרון אחריו מאשר ה-fds שמצבעים עליו.
6. הזיכרון יהיה נגיש לתהיליך הדוני מאחר ויש עדין fds שמצבעים עליו.

• הגנות:

1. Canary – הגנה על ה-stack מ-buffer overflow שגולש מחוץ לגבולות ה-stack.
2. Address Space Layout Randomization = ASLR – שינוי שוטף של כתובות השונים ב-space overhead.
3. address sanitizer – קוד שמתזקק מהחזר הקלעים תמורה של אדרוי הזיכרון (לאן מותר לגשת ולאן לא). דורשת overhead גדול.

• דוגמא מעשית לתקיפה:

- המחסנית גדלה בלא מטה ולכן הכתובות של buff יהיו נמוכות יותר מהכתובות של pass.
- לכן, אם נספק ל-buff יותר מהו שהוא יכול להכיל הוא תגלוש הכתובת של pass.
- עבור הקוד הנ"ל, אם ניתן בטור קלט 12345 אז יתקבל 5 = pass וכן נצליח לקבל הרשות גישה (pass = 1).

```
int main(void)
{
 int pass = 0;
 char buff[4];

 puts("Password: ");
 scanf("%s", buff);
 //gets(buff);

 if(strcmp(buff, "123"))
 puts("Wrong Password");
 else
 {
 puts("Correct Password");
 pass = 1;
 }

 if(pass)
 puts("Root permissions granted");

 return 0;
}
```

## Access Control

- קיימים 3 שחקנים:
  - .1. **subjects** – מי שמנסה לבצע פעולות. בפועל:
    - **תהליכיים**
    - **משתמשים וকבוצות משתמשים** – מזוהים בעזרת מרחב שמות שטווח של מספרים (uid, gid).
    - **objects** – המposables.
    - **policies** – מדיניות הגישה של subjects ל-objects.
- **authorization**
  - הגדרת policy שמאפשר גישה של subject ל-object.
  - מה שקבע האם לתהילר מסוים תאפשר subject לבצע גישה ל-object כלשהו הוא ה-uid/gid שלו.
  - subject – כל גישה ל-object ע"י subject עבורה דרך הקוד שמבצע את ה-authorization.
- 2. גישות לפתרון בעיית ה-**complete mediation**:
  - 1. **Access Control List = ACL** – רשימה לכל object שאומרת מי subjects שיכולים לגשת אליו:
    - בפועל שומרים עבור כל object את הרשותות הגישה אליו ברמה קבוצה:
      - uid שהוא הבעלים של הקובץ.
      - gid שהוא הבעלים של הקובץ.
      - אחר.
    - אפשר לשמר bitmask של 9 ביטים של RWX עבור כל subject.
    - עבור subject שנמצא לגשת ל-object – בודקים לאיזה קבוצה הוא משתייך ומהן הרשותות עבור אותה קבוצה.
- 2. **Capabilities** – אובייקטים מיוחדים שיוחזקו עבור תהליכיים (subjects) ויסמן שההיליך רשאי לגשת בדרכים מסוימות אל objects.
  - דוגמא ל-object struct file :capability שמצויב ע"י fd. הוא מייצג את יכולת של התהילך שמחזיק ב-fd לגשת לקובץ.
  - 2. הגישות הנ"ל משלימות זו את זו. בעזרת capabilities ניתן לגלות ב מהירות מהן הרשותות של התהילך (subject) ספציפי. לעומת זאת, בעזרת ACL ניתן לדעת ב מהירות מהן הרשותות של קובץ (object) ספציפי. לכן, לחוב נستخدم ב-ACL על מנת לבדוק אם יש subject כלשהו גישה ל-object מסוים, ואם כן אז נוסיף עבור אותו subject את ה-capability המתאים.

## Special Permissions

|            |   |     |
|------------|---|-----|
| sticky bit | 1 | +t  |
| setgid     | 2 | u+s |
| setuid     | 4 | g+s |

- ישן 3 הרשותות נוספות שניתן לתת לכל קובץ תיוקיה:
- **chmod xxxxxxxx**: הרשותות הללו הן הספרה הראונה ב-

## Sticky Bit

- נدلיק אותו עבור תיוקיה בעזרת אחת הפקודות הבאות:
  - chmod 1xyz dir כאשר xyz הם הראשות כתיבתיה, קרייה, הרצה.
  - chmod +t dir נקבל חיווי על כך שהוא מודלך בכך שהאות האחרונות בהרשאות (הכ"י ימינית) תהיה t.
- אם מಡליקים את הביט הזה עבור **טיוקיה** מסוימת אז רק המשתמשים הקיימים יכולים למחוק קבצים מהתיקייה:
  - file owner ○
  - directory owner ○
  - privileged user ○
- כאשר נדליק את הביט עבור קובץ זה יהיה חסר ערך ולא יעשה כלום (ב-אונליין).

- מנגנון בגרנולריות גסה יותר בסגנון של capabilities, שהיא קיימת במערכות UNIX.
- ה-pid שימש בתור מגדר capability. כלומר, ל-0 = pid היה מותר לבצע כל פעולה ולשאר לא.
- במודל הישן קיימת בעיה – **איך לתת לקבוצת subjects בלשאלה הרשות גישה לפעולות מסוימות על object בלבד** כלשהו?
- הפתרון היה להגדיר אותו בתור קבוע setuid.
- קובץ setuid - קובץ שמסוגל מರיצים אותו, התהילך שנוצר יהיה עם ה-pid של הבעלים של הקובץ.**
- הוא מסמן בכזה בעזרת ביט 5 במקומות א' בהרשות owner שלו.
- בעזרת הפתרון זה נוכל לתת למשתמשים שאינם root לוח בטור root בעת הרצת קבצים מסוימים ולאחר מכן הריצה הם יחזו ב-pid שהוא להם לפני כן.
- המודל החדש יותר מאשר המודל הישן, כי **במודל הישן אם תוקף מצליח להשתלט על ריצת התוכנית במהלך הרצת קובץ שהוגדר כ-pid אז הוא יוכל להריץ מה שהוא רוצה בתור root**. לעומת זאת, במודל החדש, מוגדר עבור אותו משתמש capability שומרה לו להריץ את הקובץ שבמודל הישן הגדרנו כ-setuid אך הוא מרים אותו בתור ה-pid המקורי שלו ולא בתור root כך שאם הוא מצליח להשתלט על ריצת התוכנית אין לו הרשות root.
- למרות זאת, גם במודל החדש עדין משתמשים ב-setuid. למשל בעת הרצת sudo.

## setgid

- באשר הביט מודלק תופיע האות 5 במקומות א' ב-sessions group permissions של הקובץ.
- באשר מרים קובץ קובץ bit-shield setgid שלו מודלק איז שומרה את הקובץ יוציא בהרשות של הקבוצה של הבעלים של הקובץ.
- באשר מדליקים את הביט עבור תיקייה אז הקבצים שבתוך התיקייה יהיו שייכים לאוותה קבוצה כמו הקבוצה של הבעלים של התיקייה.

## Sandboxing

- נקודות המוצאת שם היא שתוקפים יצילו לשבש את ריצת התוכנית.
- לכן, המטרה שלנו לאפשר להם פריבילגיות כמה שיותר מצומצמות לאחר שהצליחו לפרוץ, כך שלא יוכל להסב נזק רב.
- דוגמא ל-sandboxing : Seccomp

## Seccomp

- מאפשר לתהילך לבצע system call שמעביר אותו למצב של restricted execution, במהלךיו מותר לו לבצע רק system calls מסוימים, ואם ינסה לבצע משהו אחר אז יهرג.
- מעבר חד כיונו שלאאפשר לתהילך לחזור למצב של ריצה רגילה.
- למשל: תהילך strict של seccomp יכול לבצע רק read, write, exit, .read, .write, .exit.
- אחר ומדובר במצב מאוד מוגבל אליו יש עבורי הרחבה: bpf – seccomp. במצב זה, בכל פעם שתהילך מבצע system call או kernel מרים פילטר כלשהו שהוגדר עבורי התהילך ומודיא את ה-permissions של system call שמסוגל לבצע את system call או לא.
- למשל: ניתן להגדיר פילטר שמתיר לתהילך להריץ open אבל מודיא את ה-name path שספק כפרמטר ושakan התהילך רשאי לפתוח את אותו קובץ.

## Chroot

- הגדרה חדשה של תיקייה ה-root עבר תהילך ובכך הגבלת הגישה שלו לקבצים שמעלו.

- מעין הכללה של מנגןן -*chroot*.
- *process is name ,FS) kernel* = קבוצת תהליכי שרצה עם מרחבי שמות **פרטיאים** עבור כל האובייקטים של ה-OS.
- *space ,space ,uids name space* – בכר מייצרים אשליה עבור התהליכים שרצים ב-*container* שהם הייחדים שרצים על ה-OS.
- קיימים ברעיון זהה בעית אבטחה קונספטוואלית:
  - ה-TCB של *container* הוא בkernel.
  - נובע מכך ש-containers עובדים בגרנולריות של תהליכים בכר שהם יוצרים סביבת וירטואלית עבור תהליכי.
  - תהליכי נהנים מאבסטרקציות מאוד עשירות מצד מערכת הפעלה.

# Virtual Machines

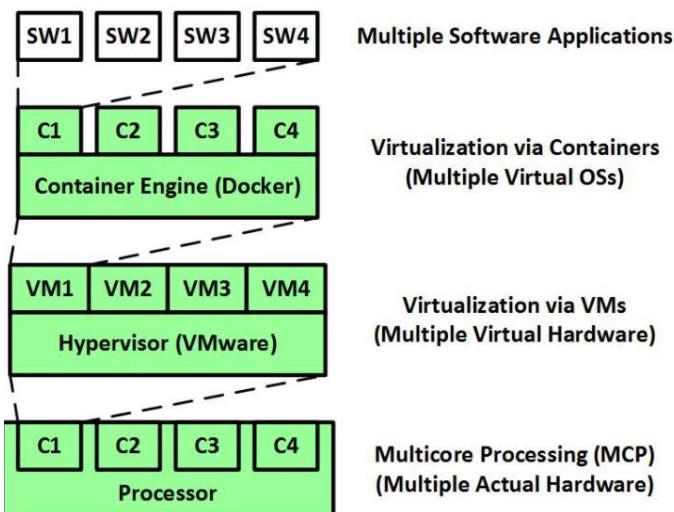
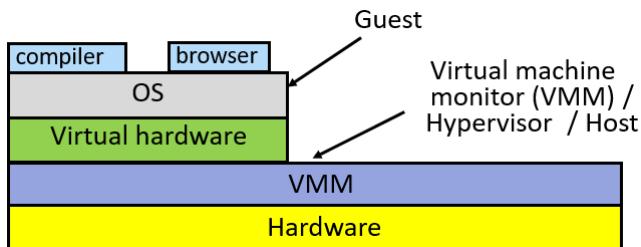
## Virtualization

- **virtualization** – כאשר עובדים מול גרסה וירטואלית של משהו במקום לעבוד ישירות מול הגרסה הפיזית/האמיתית.
- אם לא מצין הקשר של הווירטואליזציה אז המכונה היא **-host**.

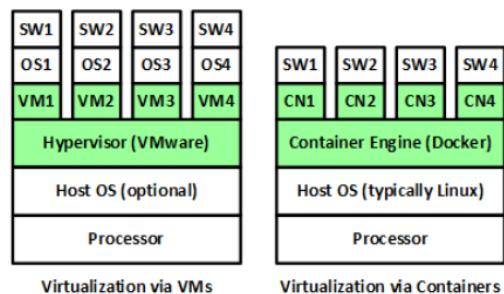
| Virtual Machines                | containers – intra OS isolation                                                                          | OS process isolation                                                      |                                |
|---------------------------------|----------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|--------------------------------|
| שיתוף בرمת משאבי התוכנה והחומרה | שיתוף בرمת משאבי התוכנה:<br>מרחבי שמות פרטיים למשאים שה-OS מספקת עבור קבוצת התהליכים ששוייכת ל-container | שיתוף משאבי החומרה:<br>שימוש מקובל במעבד/זיכרון/התקנים ע"י תהליכיים שונים | <b>isolation</b><br>בקשר של .. |
| החומרה                          | ה-OS                                                                                                     | ה-OS                                                                      | <b>וירטואליזציה</b><br>של ..   |
| +1                              | 1                                                                                                        | 1                                                                         | <b>כמה OSs?</b>                |

## Hypervisor/virtual machine monitor (VMM)/host

- התוכנה שסמכשת את המכונה הווירטואלית (למשל: VMware).
  - תנהל את החומרה האמיתית של המחשב.
- **guest (guest OS)**
  - מערכת ההפעלה שרצה בתוך VM. "מתארכת" ב-VM שה-VMM מספק.
  - תנהל את המכונה הווירטואלית ולא את החומרה האמיתית.
- הממשקים שה-VMM צריכים לספק ל-guest הם הרבה יותר פשוטים מהמשקים שה-OS צריכים לספק לתהליכיים, וגם יש הרבה פחות משקים לספק. נובע מכך שה-VMM הוא רק המתווך בין ה-OS, ולכן כל המשקים שה-OS מספקות ל процesseurs יספקו על ידי ה-guest, וה-VMM צריך רק לפרש את המידע שקשור לחומרה (לדוגמא בקשוט חומרה מה-guest ל процesseurs שבוצעו על החומרה).

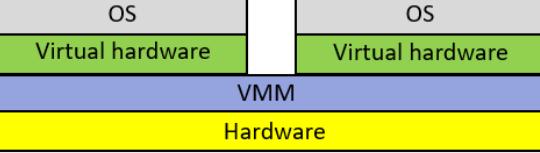


- תמונה נוספת שמחישה את ההבדל בין container לבין VM:
  - בתמונה באן: processor = hardware



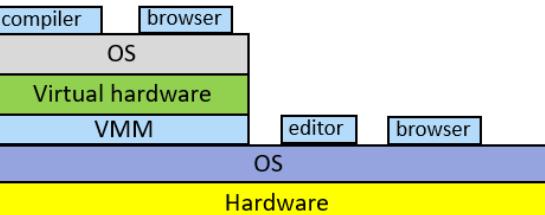
## Bare Metal VMMS (Type 1)

- ה-VMM הוא OS של המחשב עליו הוא מותקן.
- כמובן, אין OS אחרה שה-VMM מותקן אליו, והוא עוזב לשירות מול החומרה.



## Hosted VMMS (Type 2)

- ה-VMM הוא תהילך בתוך ה-OS במחשב עליו הוא מותקן.



איך למעשה VM?

## Emulation

- נכתב תוכנית emulator שתסמלץ את פעולות החומרה (ה-VMM יסמלץ מעבד בשביבו (guests)).
- כמובן, התוכנית תפרסר פקודות מכונה שנמצאות בקוד ותרץ אותן על המעבד אליו היי פקודות אלה מלכתחילה.
- פתרון איטי.

## Bare Metal Virtualization

- במקום לסמלץ את פעולות המעבד, נעשה לו וירטואלייזציה.
- ה-guest יróżץ ישירות על המעבד במצב unprivileged.
- לאחר מכן גם ה-guest kernel ירוזץ במצב unprivileged אשר exception לאלה, למרות שבפועל היי אמורות להציח לרוזץ. לכן, ה-VMM יצטרך לטפל ב-exceptions במקרה שהוא לא יצליח.
- בפועל לא יהיו הרבה exceptions באלה, מאחר ורוב פקודות המכונה שה-guest kernel מרים הן פקודות מכונה רגילים.
- פתרון מהיר.

| Bare Metal Virtualization                                                                                                                               | Emulation                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| ה-VMM מבצע אמולציה של כל פעולות שבוצעות על המעבד<br>דורש ממנו להציג ייצוג תוכנתי רק של מצב המעבד ה-priv, והיתר<br>(מצב המעבד ה-unpriv) נשמר אצל ה-guest | ה-VMM מבצע אמולציה של כל פעולות שבוצעות על המעבד<br>(priv + unpriv)<br>דורש ממנו להציג ייצוג תוכנתי של מצב המעבד ה-priv וה-unpriv |

עבשו נראה איך VM שמספק bare metal virtualization שהוא אמור לספק

- איך לדמות מצב שבו ה-**guest kernel** רצה במצב **priv** למשך זמן רב?
- ה-VMM מחזיק מבנה נתונים בשם **(VMCB)** control block של VM עבור כל **guest** שהוא מנהל.
- ה-VMCB שומר (בין היתר) את מצב המעבד של ה-**guest** (התובן של הריגיסטרים).
- כאשר ה-VMM צריך להפסיק את הריצה של **guest** כלשהו שרך עליו הוא שומר את מצב המעבד של ה-**guest** ב-VMCB שמשoir **priv**.
- כאשר ה-VMM רוצה להריץ "מחדש" **guest** כלשהו שהוא פסק הוא תוען את מצב המעבד של ה-**guest** בהתאם ל-VMCB שהוא מתחזק עבورو.
- בגין מצב המעבד הנשמר עבור התהליכים שכולל רק ריגיסטרים שאינם **priv**, ה-**VMM** שומר גם את ערכי הריגיסטרים שהם **priv** עבור כל **guest** שרך עליו. הצורך בכך הוא שכל **guest** יש ערכי ריגיסטר **priv** שונים, בשונה מה המצב המקורי שבו כל התהליכים שרכיטם על ה-OS יש אותו ערך ריגיסטר **priv**.
- ב-**bare metal virtualization** יש **הרכבה של שני מנגנוני**: **CPU time sharing**:

  1. ה-**VMM** מנהל עבור ה-**guest-guest**-ים שרכיטם עליו.
  2. ה-**guest** מנהל עבור התהליכים שרכיטם עליו.

### ביצוע פעולות **priv**

- ע"י ה-**guest**:
  1. ה-**guest** מנסה לבצע פעולה המוגדרת בתור **priv**.
  2. מתרחש **exception**.
  3. המעבד ירים **handler** של ה-VMM.
  4. אותו **handler** יעשה **emulation** (סמלוץ) של המעבד:
    - a. ה-**handler** יבין מי ה-**handler** לאו דווקא ב-**guest**.
    - b. ה-**handler** של ה-VMM יעביר את המעבד להריץ את ה-**handler** המתאים ב-**guest kernel**.
- ע"י ה-**guest kernel**:
  1. ה-**guest kernel** מנסה לבצע פעולה המוגדרת בתור **priv**.
  2. מתרחש **exception**.
  3. המעבד ירים **handler** של ה-VMM.
  4. אותו **handler** יעשה **emulation** (סמלוץ) של המעבד:
    - a. ה-**handler** של ה-VMM יבצע את הפעולה בתור נווק במקור הייתה במקורה שהוא רצה לעשות בתור **priv**.
- **פעולות תשתיית/הכנה לטובות התפעול של ה-**exception** עבור ה-**guest**:**
  1. בשלב עליית ה-**kernel** הוא יבנה את ה-**exception table** שלו.
  2. הכתובת של ה-**exception table** תשמור בתור ריגיסטר **priv** במעבד.
  3. בתגובה (לניסוי לבצע פעולה **priv** ע"י תהליך שאינו מוגדר **priv**) יתרחש **exception**.
  4. במעבד ירים **handle** מתאים ב-**VMM** (שער במצב **priv**).
  5. ה-**handler** יבין (איבשחו). לא רלוונטי אליו) שה-**guest** ניסתה להגיד למעבד את ה-**exception table** שלו.
  6. ה-**handler** שומר לעצמו את הכתובת של ה-**exception table** של ה-**guest** ב-**VMCB** של אותו ה-**guest**. **shadowing** = (guest = state).
  7. ה-**handler** מסיים את עבודתו וה-**VMM** חוזר מהטיפול ב-**exception**.

• תפקוד ה-VMM:

1. לספק זיכרון פיזי – וירטואלי, אותו ה-*guest* תנהל.
2. לאפשר ל-*guest* להציג מיפויים מכתבות וירטואליות אל כתובות "физיות" של ה-*guest*.

כלומר, הזיכרונות שיופיעו לנו:

| GPA (= Guest Physical Addresses) | GVA (= Guest Virtual Addresses)      | HPA (= Host Physical Addresses) |
|----------------------------------|--------------------------------------|---------------------------------|
| בתובת וירטואלית                  | ימופה אל כתובת פיזית אמיתית<br>במחשב | כתבות הפיזיות האמיתיות במחשב    |

• ה-MMM **VMM** יתחזק מיפויי **HPA**  $\rightarrow$  **GVA**

• ה-*guest* יתחזק מיפויי **GVA**  $\rightarrow$  **HPA**

• מאחר ויש באן **2 מיפויים** בשונה מהמצב הרגיל בו מיפויו M-VA ל-PA, אז יש לבצע התאמה כדי שה-UML יוכל לעשות את המיפוי.

• לצורך כך נשתמש ב-**shadow paging**:

◦ עברו כל *frame* שה-*guest* kernel מייצר, נתחזק *shadow frame* שיבכיל את המיפוי הנוכחי.

- אם מדובר במיפוי של ה-*guest* אל עלה ב-*pt* *guest* אז המיפוי יוביל ל-*HPA* (ישב ב-*HPA*).
- אם מדובר במיפוי של ה-*guest* אל צומת פנימי ב-*pt* *guest* אז המיפוי יוביל לצומת פנימי ב-*pt*.

• **אופן יצירה ה-pt shadow:**

1. ה-*guest* ינסה לבצע השמה אל ה-PTBR על מנת להציג עבור המעבד את ה-*pt* שלו.
2. מאחר ומדבר ברגיטר ז'וק יתקבל exception.

3. ה-*VMM* יבין שה-*guest* מנסה לעדכן את המצביע ל-*pt* עבור המעבד.

4. ה-*VMM* יקצת פריים שיישמש כ-*shadow* עבור השורש של ה-*pt* *guest* והמלא את כל ה-*PTEs* שלו ב-*invalid*.
- ○ יונסה lazy approach – במקום לסרוק את כל העז ישמור רק את השורש).

5. ה-*VMM* יכתוב את הכתובת של ה-*shadow* שיצר בתור ה-PTBR עבור המעבד עבור אותו *guest* (ישמר ביחיד עם תוכן הריגיסטר ז'וק של ה-*guest*-*VMCB*).

6. עברו הגישות הבאות של ה-*guest* שינסו לבתוב/לקראן מידע אל הכתובות ב-*pt* *guest* יתקבל page fault עבור ה-*VMM* שיגרמו לייצור מיפוי shadow ב-*pt* *shadow* עבור המסלול שה-*guest* הלך עליו ב-*pt* *guest* (on demand approach).

7. בתגובה ל-*page fault* ה-*VMM* ישתמש במצביע שיש לו ל-*pt* *guest* walk כדי לעשות *page* בתוכנה ולהבין לאיזה GPA ה-*VA* ממופה.

8. ה-*VMM* יבודק לאיזה HPA ה-*GPA* ממופה.

9. אם לא קיים פריים זהה – ה-*VMM* מקצת פריים חדש עבור המיפוי.

10. ה-*VMM* מוסיף מיפוי ב-*pt* *shadow* מה-*GVA* ל-*HPA*.

11. המעבד מבצע את הגישה שה-*guest* ניסה לעשות.

### בעיות בשימוש ב-pt shadow

בעיה: אין עדכניות של ה-pt shadow GVA שמעופו בעבר ב-pt shadow. נמאר את הבעיה בעזרת המצב הבא:

1. ה-guest מבצע כתיבה ל-GVA שעדיין אין לה מיפוי ב-pt shadow.
2. בתגובה מהכתיבה מתקבל page fault לשא-MMM מטפל בו בכר שודאג לעדכן את המיפוי המתאים ב-pt shadow.
3. בעת ה-guest מצליח לבצע את הכתיבה שרצה לאוותה GVA.
4. בעת נניח שה-guest משנה את המיפוי של אותה GVA ב-pt guest.
5. בעת נניח שה-guest מבצע כתיבה/קריאה לאוותה GVA.
6. לא יתקבל page fault, מאחר וכפי שכבר אמרנו, ה-GVA מופתה ב-pt shadow.

עם זאת, המיפוי של ה-GVA ב-pt shadow אינם עדכני, מאחר ובוצע שינוי של המיפוי ב-pt guest.

פתרונות: נפתרו את הבעיה בעזרת tracing – ה-MMM יציג ש-GVAs ב-pt guest לא יהיו מופיעות ב-pt shadow, כך שבכל גישה של ה-pt kernel ל-pte guest pt fault יתבצע אמולציה של הגישה ולסבךן את המסלול המתאים ב-pt shadow pt.

בעיה: ה-pt kernel צריך לגשת למיפויים במרחב הכתובות של ה-pt kernel שמודדרים בטור priv, ולא ניתן לגשת למיפוי priv ב-MMM.

פתרונות 1:

1. ה-pt shadow לא יכול את מיפוי ה-priv של ה-pt guest.
2. כאשר יבוצע mode switch על ה-pt guest, ה-MMM יטען את כל מיפוי ה-priv ל-pt guest.

חסרון: ירידת משמעותית בביצועים.

פתרונות 2:

- בהרבה ארכיטקטורות חומרה (למשל: 86x) יש מצב נוסף:ovic-semi שכאשר המעבד נמצא בו אז הוא יכול לגשת לדפים וירטואליים שמודדרים בטור priv, אבל עדיין לא ניתן לבצע פקודות מכונה שמודדרות בטור priv.
- ה-MMM יגדיר את ה-pt guest kernel בזיכרון זהה, כך שיוכל לגשת למיפויovic-semi אבל לא למיפוי host kernel.

## Virtualizing I/O Devices

וירטואלייזה של התקנים עובדת בעזרת אמולציה:

1. ה-MMM מנטר תקשורת של ה-pt guest עם התקנים (page fault tracing) והוא גורם לכך שכל גישה תגרום ל-fault.
2. כאשר ה-MMM תופס גישה להתקן הוא עשויה לה אמולציה.

דוגמא לאמולציה על ברטיס רשות:

- ה-MMM מימוש את הפונקציונליות של הברטיס רשות בקוד.
- כאשר ה-MMM ניטר גישה של ה-pt guest (guest) לברטיס רשות אז הוא ידמה עבורי את ההתחנחות של ברטיס הרשות בהתאם ל-spec של ברטיס הרשות.

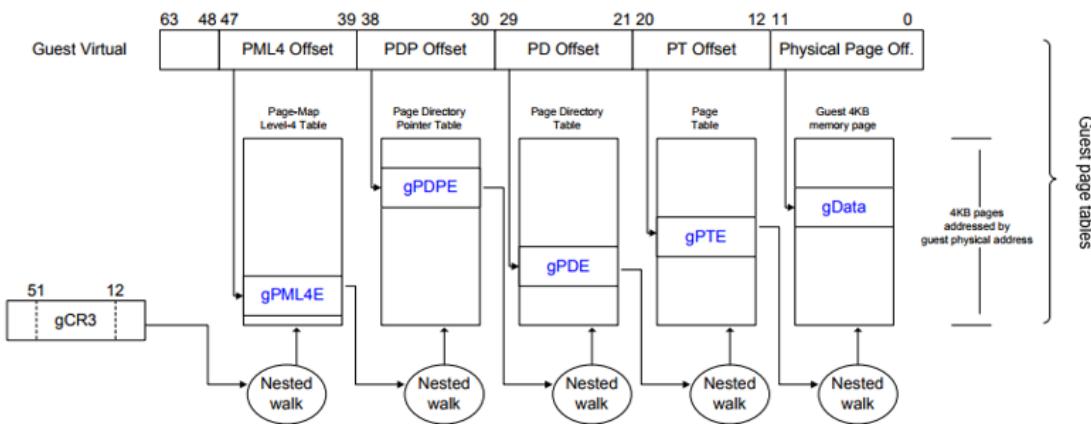
דוגמא לאמולציה על התקני אחסון:

- ה-MMM סיפק ל-pt guest דיסק בגודל 500MB.
- בפועל ה-MMM יוצר קובץ בגודל של 500MB ודואג לתרגם את הגישה של ה-pt guest לבלוק X בדיסק לגישה לאופסט X בקובץ.

- עם ההצלחה של טכנולוגיות המכונות הוירטואליות חלו שיפורים במעדים על מנת לתמוך בוירטואלייזציה:

### שינויים במעדים על מנת לתמוך במכונות של הוירטואלייזציה

- **וירטואלייזציה נכונה:**
  - ה-VMM צריך להיות מוגן מה-guest kernel שרים עליו (לא אפשר פעולות privileged guests עליו) (לא אפשר מהתהילכים שרים עליו (לא אפשר פעולות privileged)).
  - ה-guest kernel צריך להיות מוגן מהתאיליכים שרים עליו (לא אפשר מהתהיליכים שרים עליו).
  - ה-guest kernel לא צריך להזות שהוא רץ ב-VM.
  - בפרט, הוא לא צריך להזות שהוא רץ בשמהעך לא במצב priv.
  - בפרט, הוא לא צריך להזות שפוקודות מסוימות שהוא מריצ' גורמות ל-exception במקומם להתבצע ישירות.
- **binary translation**
  - ה-VMM היה קורא את פקודות המכונה של ה-guest kernel וכותב במקומן פקודות מכונה אחרת שהיה מבצעות פונקציונליות שקולה, אבל מוביל ליצור בעיות לוירטואלייזציה.
  - ה-VMM ייתן ל-guest kernel לרווח במצב priv ויתרגם פקודות שאמורות לגרום ל-exception לפעולות שייערכו ישירות את מבני ה-shadow של ה-VMM.
- **guest mode**
  - במצב זה המעבד מתנהג בצורה שמאפשרת ביצוע וירטואלייזציה נכונה. נרchia בرك על אחד מהם: nested paging.
- **nested paging**
  - כאשר המעבד במצב guest הוא ידע שהוא MCP-E מ-GVA ל-GPA וכן יתמוך בתרגום ההפוך (GVA -> GPA, GPA -> HPA).
  - בכיה יוכל לחסוך את השימוש בshadow paging והתקורה הנלויה ב-page faults tracing –nested pt – יתבצע בעזרת מבנה נתונים חדש –nested pt – nested pt
  - יוצבע ע"י רегистר חדש במעבד.
  - יוכל את המיפויים מ-GVA ל-HPA.
  - יוזכר ע"י ה-VMM.
- **אין זה יעבוד**
  1. בכל רמה של ה-pt המעבד קורא PTE ומחלץ משם את הפריטים שמכיל את הכתובת ברמה.
  2. מאחר והמעבד במצב guest mode הוא יודע שמדובר על פריטים בזיכרון הפיזי של ה-VM.
  3. לכן, המעבד ייגש ל-pt nested.



- בעת כל page walk יעלה 24 גישות לזכרון עבור tk בעומק 4:
- עבר בرمאה, בנוסף לגישה לפירטים של ה-pt geust pt נצטרך לבצע page walk על ה-pt nested על מנת לבצע תרגום מ-GVA ל-HPA. (5 גישות לבטחן עבור 4 רמות).
- כאשר מוצאים את ה-GPA (ה-PTE בברמה האחורונה ב-pt guest) יש לבצע תרגום ל-HPA. 4 גישות.

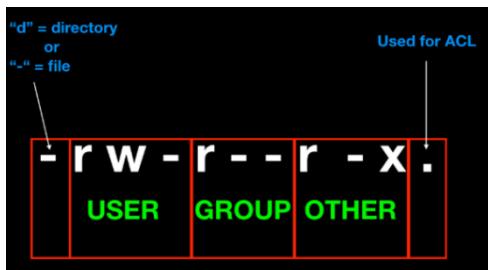
## Permissions

- לכל משתמש מוגדרות הרשאות עבור כל קובץ. הרשאות האפשריות הן:
  - read = r or 4
  - write = w or 2
  - execute = x or 1

ניתן לתת יותר מהרשאה אחת ע"י שילוב של הנ"ל

```
total 0 1. USER 2. GROUP
-rw-r--r--. 1 root root 0 Jun 8 17:27 cat.txt
-rw-r--r--. 1 root root 0 Jun 8 17:27 dog.txt
drwxr-xr-x. 2 root root 6 Jun 8 17:27 Pets
```

- ישנן שלוש קבועות שיכולות לקבל הרשאה:
  - file/directory's owner – user .1
  - file/directory's group - group .2
  - anyone else – other .3



שינוי הרשאות: chmod <group><+/-><permissions combination> <file/dir name>

## gdb

- דיבאגר מובנה לתוכניות C
  - gdb ./<executable name> – init
  - break <function name> – adding breakpoints
  - הdfsata הערך של ה-PC בבל רגע נתון – display/i \$pc
  - הרכצת הורקנות הורקנות הבא – stepi instruction-here
    - ”step instruction”
  - המשך הרכצת הורקנות – cont
  - הרכצת התוכנית – run
  - הdfsata התוכן של ההורקנזה הנוכחי – disas

## printf

הפונקציה שקוראת ל-syscall write היא

## write

- \$rdi – הרגיסטר שמחזק את התוכן לפוי הפונקציה יודעת לאן עליה לכתוב
  - ubo 1 – standard output
- \$rsi – הרגיסטר שמחזק את התוכן אותו אנחנו רצים לכתוב

## רגיסטרים

- \$pc – הרגיסטר עבור ה-PC
- בארכיטקטורת x86 חילקו חלק מהרגיסטרים ל-2 ונתנו שם נפרד לכל חצי ורגיסטר בר ששניון לשנות חלק מהרגיסטר.
- למשל 32 הביטים התוחתונים של %rax הוא %eax.

## File Descriptor

- המזהה int הוא File Descriptor ייחודי עבור קובץ כלשהו הנמצא בשימוש ע"י תהליך.
- Table – טבלה עם כל ה-File Descriptors של תהליך כלשהו.
- חשוב! המספרים הללו הינם ייחודיים לכל תהליך, אך מצביעים למקום בו בירור המשותף לכל התהליכים, כך שיתכן ש-File Descriptors של תהליכים שונים יצביעו באותו הקובץ.

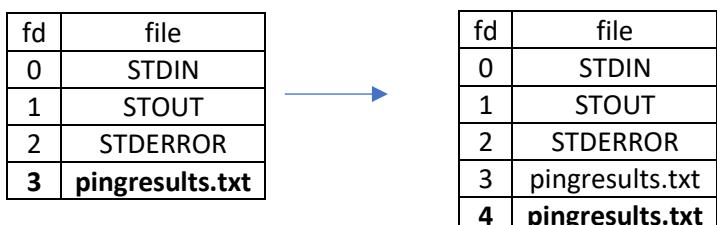
|        |        |
|--------|--------|
| fd = 0 | stdin  |
| fd = 1 | stdout |
| fd = 2 | stderr |

## I/O System Calls

| פונקציה | חתימה                                                     | פרמטרים                                                                                                                                                                       | ערך החזרה                                                             |
|---------|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| create  | int <b>create</b> (char *filename, mode_t mode)           | filename – שם הקובץ שיופיע<br>mode – הרשות לקובץ                                                                                                                              | בהצלחה – ה-fd של הקובץ<br>בכשלון -1                                   |
| open    | int <b>open</b> (const char* Path, int flags [,int mode]) | Path – נתיב מוחלט לקובץ שמתחל עט \ או נתיב יחסית (אם הקובץ נמצא בתיקיה הנוכחיית)<br>flags – O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL                                       | בהצלחה – ה-fd של הקובץ<br>בכשלון -1                                   |
| close   | int <b>close</b> (int fd)                                 | fd – ה-file descriptor שරוצים לסגור                                                                                                                                           | בהצלחה – 0<br>בכשלון -1                                               |
| read    | int <b>read</b> (int fd, void* buf, size_t cnt)           | buffer – תקרא cnt בתים מ-fd לתוכו<br>fd – ה-file descriptor שרוצים לקרוא ממנו<br>cnt – אורך הבuffer<br>תנאי להיווט זהה שהוחזר מ-open()<br>מקום לתוכו המידע שקורא יכתב         | בהצלחה – מס' הבטים שנקראו<br>-EOF – 0<br>בשגיאה -1<br>בקבלת סיגנל -1  |
| write   | int <b>write</b> (int fd, void* buf, size_t cnt)          | buffer – כתוב cnt בתים מה-buffer לתוכו fd<br>fd – ה-file descriptor שרוצים > 0 לכתוב אליו<br>cnt – אורך הבuffer<br>תנאי להיווט זהה שהוחזר מ-open()<br>מקום לתוכו המידע שייכתב | בהצלחה – מס' הבטים שנכתבו<br>-EOF – 0<br>בשגיאה -1<br>בקבלת סיגnal -1 |

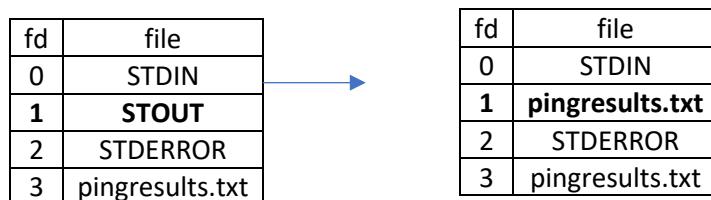
## dup() / dup2()

- system call ופונקציה ב-C.
  - מהמילה duplicate.
  - dup מיצרת עותק של file descriptor כך שהיא אפשר לגשת אליו מ-2 נקודות שונות.
- למשל: dup(file), בהינתן ש-3 = file , יגרור את השינוי הבא:



dup מכתבת fd קיים ל-fd נתון.

למשל: dup2(file, 1), בהינתן ש-3 = file , יגרור את השינוי הבא:



## • חתימה

```
int dup (int oldfd)
int dup2(int oldfd, int newfd)
```

## • פרמטרים

o file descriptor – **oldfd**

o file descriptor – **newfd**

## • ערך החזרה:

o dup – ה-fd החדש שנוצר

o dup2 – ה-fd הישן (= fd old) שהשתמשנו בו בשבייל להציג על ה-fd שסופק בתור פרמטר ראשון.

o dup משתמש ב-descriptor עם המספר ההפוך נמור לצירוף העותק הב"ל.

o חשוב לזכור את ה-fd המקורי ששכפלנו אם אנחנו לא הולכים להשתמש בו.

## fork()

call system ופונקציה ב-C.

משמשת ליצור תהליך חדש שיקרא תהליך בן.

אחריו שתהליך הבן ייווצר, ל-2 התהליכים יהיו נתונים זהים לחלוון למעט pid ו-pid.

הפוקודה הבאה שתבוצע ב-2 התהליכים היא הפוקודה הבאה אחרי ה-fork.

## • ערך החזרה

o ערך שלילי -> יצירת התהליך יلد לא צליחה.

o 0 -> יצירת התהליך יلد הצלחה ואנחנו יכולים להשתמש בתהליך הילד.

o ערך חיובי = ה-pid של תהליך הבן -> יצירת התהליך יلد הצלחה ואנחנו יכולים להשתמש בתהליך האבא.

o gedpid() – ממחזירה את ה-ID של התהליך שקרה לפונקציה.

o gedppid() – ממחזירה את ה-ID של האבא של התהליך שקרה לפונקציה.

o SIGCHLD – סיגナル שתהיליך (fg או bg) שלו לאב שלו להודיע שהוא terminated.

## exec()

משפחת פקודות המאפשרות לחתוך את תמונה הריצה של התוכנית שרצה ולהחליף אותה בתמונה אחרת.

במילים פשוטות: להחליף את התהליך שרצה בתהליך אחר.

## execvp()

חלק משפחת הפקודות של exec()

| חתימה                                                | פרמטרים                                                                                                                                                                                                                  |
|------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| int execvp(const char *filename, char *const argv[]) | <b>filename</b> – נתיב (לא מלא) לתוכנית אותה אנחנו רצים להריץ במקום התהליך הנוכחי.<br>(argv[0] – האיבר הראשון בתוכנה וסביבתו path).                                                                                      |
| ערך החזרה                                            | <b>argv</b><br>argv[0] – האיבר הראשון יהיה filename<br>argv[1] – האיבר האחרון יהיה NULL<br>argv[2] – שאר האיברים יהיו ארגומנטים לתוכנית<br><br><b>בἷילון -&gt; -1</b><br>הצלחה -> לא כודרים. התמונה של הקובץ הרץ מתחלפת. |

## • עם ההחלפה לתהליך החדש

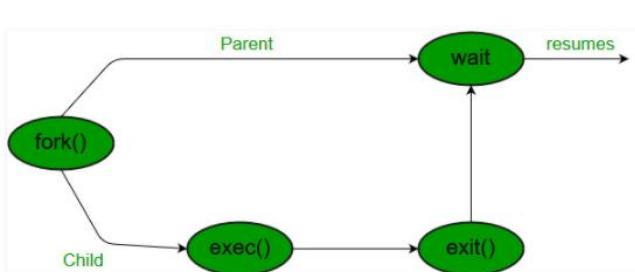
o ה-code וה-data יהיו של התוכנית החדשה.

o המחסנית, ההייפ והרגיסטרים מתאפסים ( מבחינת התהליך הנוכחי). יתכן שיש בהם את הערכים של התהליך הקודם, אך

ombination התהליך החדש הם ערכים זבלאים).

o אין דרך לחזור לתהליך הקודם.

- ה-**id** process של התהיליך החדש זהה לישן.
- האבא של התהיליך החדש זהה לישן.
- אם הוא **so-called file descriptor** פותחים בתהיליך הקודם הם נשארים פתוחים בתהיליך החדש.



### **wait() / waitpid()**

- פונקציות המאפשרות לתהיליך אבא לחכות לסיום העבודה של הבן.
- תהליך הבן יסיים את עובdotו (terminate) בגלל אחת מביב:
  - 1. קריאה-ל-(exit())
  - 2. החזרת **int** מ-**main**.
  - 3. קיבל סיגナル מה-OS או תהליך אחר שגורם ל-termination.
- 4. **reaped** - מצב בו יש קריאה-ל-**wait** לאחר שתהיליך הבן יסיים את עובdotו (מת). "the process son has been reaped"
- במצב זה ה-OS תסיר את התהיליך בן מרשימת התהיליכים
- waitpid** ממחכה לסיום העבודה של **בן ספציפי**, בעוד **wait** ממחכה לסיום העבודה של **אחד מבנים**.

|                       |                                                                   |
|-----------------------|-------------------------------------------------------------------|
| חתיימה                | <b>pid_t wait(int *status)</b>                                    |
| פרמטרים               | <b>status</b> – מידע לגבי באיזה צורה הבן הסתיים (הצלחה/כשלון/...) |
| הצלה- <- ה- <b>id</b> | של התהיליך שישים                                                  |
| ערך החזרה             | אין בנים שאפשר לחכות להם -<-1-                                    |

|                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| חתיימה                | <b>pid_t waitpid(pid_t pid, int *status, int options)</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| פרמטרים               | <b>filename</b> – נתיב (לא מלא) לתוכנית אותה אנחנו רצим להריץ במקום התהיליך הנוכחי.<br><b>path</b> – מחפש את הנתיב במשתנה הסביבה <b>path</b> .<br><b>pid</b> – ה- <b>id</b> של התהיליך שמחייבים<br><b>status</b> – מידע לגבי באיזה צורה הבן הסתיים (הצלחה/כשלון/...)<br>זה המידע שאיתו התהיליך עשה <b>return</b> או <b>exit</b> <ul style="list-style-type: none"> <li>1. <b>WIFEXITED(status)</b>: child exited normally           <ul style="list-style-type: none"> <li>• <b>WEXITSTATUS(status)</b>: return code when child exits</li> </ul> </li> <li>2. <b>WIFSIGNALED(status)</b>: child exited because a signal was not caught           <ul style="list-style-type: none"> <li>• <b>WTERMSIG(status)</b>: gives the number of the terminating signal</li> </ul> </li> <li>3. <b>WIFSTOPPED(status)</b>: child is stopped           <ul style="list-style-type: none"> <li>• <b>WSTOPSIG(status)</b>: gives the number of the stop signal</li> </ul> </li> </ul> |
| הצלה- <- ה- <b>id</b> | <b>options</b> – שילוב OR של 0 או יותר מהדגים הבאים:<br><ul style="list-style-type: none"> <li>• <b>WNOHANG</b> – תחזיר מיד אם אף תהיליך בן לא יצא</li> <li>• <b>WUNTRACED</b> – תחזיר אם תהיליך בן עצר</li> <li>• <b>WCONTINUED</b> – תחזיר אם תהיליך בן חזר ל떙ק כתוצאה מ-<b>SIGCONT</b></li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| ערך החזרה             | הצלה- <- ה- <b>id</b> של התהיליך שישים<br>אין בנים שאפשר לחכות להם -<-1-                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

- בעזרת פונקציות אלו ניתן להכטיב (חלקוות) סדר ביצוע בין תהיליכים לבנים שלהם.

### **Zombies**

- מצב שבו התהיליך אבא רץ בעודו **wait** על הבן סיים ואין קריאה-ל-**wait** שמחכה למוות של התהיליך בן.
- במצב זהה, התהיליך בן יישאר בטבלת תהיליכים, למורת שהוא מת, ויסומן שם בתהיליך לא פעיל (**<defunct>** או **Z**).
- לכן הוא נקרא זומבי. התהיליך מת ואי אפשר להרוג אותו.
- היחסון של זה הוא שהטהיליך בן תופס **pid** למורת שהואobar מת.
- במצב שבו גם האבא יموت (ולא יהיה בו **wait** לבן) **pid 1** (**init**) ייקח אחריות על הבן וישחרר אותו.
- איך להיפטר מזומבים? wait או להרוג את האבא**

- מצב שבו האבא מת והבן עדין חי.
- במקרה זה תהליך ה-init (1 = pid) יאמץ אותו (וירחוג (reap)) אותו כאשר הוא יموت.

## Linux Signals

- ticks interrupts שנשלחים לתוכנית לסמן לה שarious חשוב קרה.
- תהליך שרעץ ומתקבל סיגナル עוצר את מה שביצע ומטפל בסיגナル שקיבל (בעזרת signal handler (signal handler))
- 32 סיגנלים**
- חלק מהסיגנלים שכדאי להזכיר:
  - SIGCHLD** – תהליך בן כלשהו עצר (stopped) או מת (terminated).
  - SIGUSR1, SIGUSR2** – סיגנלים שלא מוגדרים ע"י ה-OS והמשתמש יכול להגדירם לשימושו.
  - SIGTSTP** – תעוצר את התהליך. נשלח ב: **ctrl + z**.
  - SIGINT** – interrupt מהמקלדת. נשלח ב: **ctrl + c**.
  - SIGHUP** – מודיע לתהליך שרעץ שהטרמינל שמרץ אותו נסגר.
  - SIGPIPE** – סיגナル שנשלח באשר הכותב מנסה לכתוב, אבל הקורא כבר סגר את ה-file descriptor.
- חלקים מטופלים ע"י ה-OS וההתהליך לא יכול לטפל בהם אחרת:
  - SIGKILL** – הרוג את התהליך ולא נותן לו אפשרות לעשות משהו אחר.
  - SIGSTOP** – מקפיא את התהליך ועוצר את הפעולה שלו עד שהוא יקבל את הסיג널 המשיך את הריצה שלו.
- חלקים יכולים להיות מטופלים על ידי המשתמש, אך הטיפול הדיפולטי של ה-OS יקרה בכל מקרה:
  - SIGCONT** – תמשיך את התהליך (אם הוא הפסיק). נשלח באשר משתמשים בפקודת **fg**.

## kill()

- system call שמאפשר לשלוח כל סיגナル (לאו דווקא לרצוץ תהליכיים)

| חתימה                                                    | פרמטרים                                   | ערך החזרה               |
|----------------------------------------------------------|-------------------------------------------|-------------------------|
| <code>int kill(pid_t pid, int sig)</code>                |                                           |                         |
| <code>pid</code> – התהליך שאליו אני רוצה לשלוח את הסיגナル | <code>sig</code> – הסיגナル שאני רוצה לשלוח |                         |
|                                                          |                                           | הסיגナル נשלח בהצלחה -> 0 |

- קיים פקודה ב-bash kill לשילוח הסיגנלים הללו: kill
- user יכול לשלוח סיגנלים לתהליכיים שלו בלבד
- super user יכול לשלוח סיגנלים לכל התהליכים

## raise()

- שליחת סיגナル לעצמי
- חתימה: `int raise(int sig)`

## יצירת סיגנלים משלנו

- הגדרת פונקציה עבור הסיגナル

| חתימה                                                                        | פרמטרים                                                                                                                                                                  |
|------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void my_siignal_handler(int signum, siginfo_t *info, void *ptr)</code> | <code>signum</code> – מס' הסיגナル<br><code>info</code> ו- <code>ptr</code> מספקים מידע על הסיגナル<br>למשל: <code>pid = info-&gt;si-&gt;pid</code> של התהליך ששלח את הסיגナル |

| חתימה                                            | פרמטרים                          |
|--------------------------------------------------|----------------------------------|
| <code>void my_siignal_handler(int signum)</code> | <code>signum</code> – מס' הסיגナル |

- **יצירת struct של sigaction**
- איפוסו (לא חברה)
- ביצוע השמה לפונקציה של הסיגנל באמצעות **sa\_sigaction** או **sa\_handler** (מציריך שימוש בדגל **SA\_SIGINFO**)
- **sa\_handler** יכול לקבל גם:
  - **SIG\_IGN** – ותעלם מהסיגנל
  - **SIG\_DFL** – יקרה לסיגנל הנדר הדיפולטי
- **sa\_flags**
- **SA\_SIGINFO** – בשוחטים להשתמש בפונקציה שמקבלת 3 ארגומנטים אחרים, פונקציה שמקבלת **ARGUMENT**
- **ichiid (int signum)**
- **SA\_RESTART** – צריך להשתמש כאשר מייצרים סיגנל הנדר משלנו.
- **קריאה ל-sigaction**
- **פרמטרים**
  - 1: **איזה סיגנל** יטופל
  - 2: **בתובת ה-struct** של **sigaction** שייצרנו
  - 3: **פונטער** ש-**sigaction** יוכל לשומר לתוכו את הסיגנל הקודם (אם לא מעוניינים בכך להעביר **NULL**)
- **ערך החזרה**
  - **הצלחנו** לרשום את הסיגנל -> **0**
  - **נכשלו** ברשום את הסיגנל -> **שונה מ-0**
- יש אפשרות להשתמש ב-**signal**, אבל עובד בצורה לא טובה -> **לא להשתמש ב-signal.**
- **דוגמא מהתרגול:**

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

//-----
void my_signal_handler(int signum,
 siginfo_t* info,
 void* ptr)
{
 printf("Signal sent from process %lu\n",
 (unsigned long) info->si_pid);
}

//-----
int main()
{
 // Structure to pass to the registration syscall
 struct sigaction new_action;
 memset(&new_action, 0, sizeof(new_action));
 // Assign pointer to our handler function
 new_action.sa_sigaction = my_signal_handler;
 // Setup the flags
 new_action.sa_flags = SA_SIGINFO;
 // Register the handler
 if(0 != sigaction(SIGUSR1, &new_action, NULL))
 {
 printf("Signal handle registration " "failed. %s\n",
 strerror(errno));
 return -1;
 }
 while(1)
 {
 sleep(1);
 printf("Meditating\n");
 }
 return 0;
}
```

- עורך מידע חד כיוני שמאפשר להעביר מידע מכתב/ים אל קורא יחיד.
- המידע עובר בתצורה של תור (FIFO).
- המידע מועבר בתצורת once written once read (ניתן לקרוא רק פעם אחת).
- אם רוצים לצרוך את המידע הזה שוב איז צריך לדאוג לשמר אותו בעצמו.

**pipe()**

צורה עבודה כללית

```
main()
{
int pfds[2];
pipe(pfds);
...
fork()
// Parent
close(pfds[0])
write(pfds[1], buf,...)
close(pfds[1])
...
}
```

Parent transfer  
data to child

```
// Child
close(pfds[1])
read(pfds[0], buf,...)
close(pfds[0])
...
```

**int pipe(int pipefd[2])** – מערך של 2 file descriptors

ערך החזרה

- 0 -> הצלחה
- -1 -> כישלון

בסיום עבודת הפונקציה

- [0] – צד הקיראה לתוך הצינור
- [1] – צד הביקפה לתוך הצינור

הסיבה שעושים את ה-fork היא כדי לשותף את ה-socket בין שני התהליכים.

לאחר ה-fork **לבכל תהליך יש מערך של 2 fd פתוחים** (בהנחה

ולא סגרנו אותם לפני ה-fork).

**כותב שמנסה לכתוב באשר כל הקוראים סגורים יקבל SIGPIPE.**

**קורא שמסה לקרוא באשר כל הכותבים סגורים יקבל 0.**

דוגמאות:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFF_SIZE 256

int main(int argc, char *argv[]) {
 int pipefds[2];
 if (pipe(pipefds) == -1) {
 perror("Failed creating pipe");
 exit(EXIT_FAILURE);
 }

 int readerfd = pipefds[0];
 int writerfd = pipefds[1];

 int pid = fork();
 if (pid == -1) {
 perror("Failed forking");
 close(readerfd);
 close(writerfd);
 exit(EXIT_FAILURE);
 } else if (pid == 0) {
 // Child
 close(readerfd); // close read side
```

```

char *msg = "Hello from child!";
int msg_len = strlen(msg);
int bytes_written;
while ((bytes_written = write(writerfd, msg, msg_len)) > 0) {
 msg += bytes_written;
 msg_len -= bytes_written;
}
if (bytes_written == -1) {
 perror("Failed writing to pipe");
 close(writerfd);
 exit(EXIT_FAILURE);
}

close(writerfd);
} else {
 // Parent
 close(writerfd); // close write side

 char buff[BUFF_SIZE+1];
 int bytes_read;
 while ((bytes_read = read(readerfd, buff, BUFF_SIZE)) > 0) {
 buff[bytes_read] = '\0';
 printf("%s", buff);
 }
 if (bytes_read == -1) {
 perror("Failed reading from pipe");
 close(readerfd);
 exit(EXIT_FAILURE);
 }

 printf("\n");
 close(readerfd);
}
}

```

- **למה הצד הקורא סגור את ערזע הכתיבה?**

- יכולם להיות מס' כתובים ל-pipe
- בשביל שנקלט ערך החזרה 0 מ-read צריך שככל הכותבים יהיו סגורים.
- [1]pipefd[0] פתחים בפתיחת דיפולטי עם יצירתם, וכך שבעור התהילה שמתבצע לקרוא מהקובץ יהיה ערזע כתיבה פתוח עם עצמו ועליו לסגור אותו.

- **למה הצד הכותב סגור את ערזע הקריאה?**

- באופן דומה, כאשר הכותב רוצה לכתוב אבל בפועל אין עוד ערזע קרייה פתחים נרצתה שהכותב לא יכתוב.
- מאחר גם לו יש ערזע קרייה שפתחו באופן דיפולטי, הוא לא ידע שאין בפועל ערזע קרייה פתחים (מלבד הוא עצמו).
- לכן, צריך לסגור את ערזע הקרייה שלו, וכך אם יש ערזע קרייה שהם לא הוא עצמו -> אז הוא יכתוב. אחרת-> הוא לא יכתוב (ובצדק כי אין צד קרייה רלוונטי שיקרא את הכתיבה שלו).

## Named Pipes

- דרך נוספת לביצוע סוקט מוביל לשימוש ב-(fork).
- pipe named הוא סוג נוסף של אובייקט (בדומה לתיקיה וקובץ) המסומן ב-**ק** במערכת הקבצים.
- השיטה שבה זה יעבד היא שלאחר יצירת האובייקט, 2 תהליכי שונים יכולים לכתבו ולקרוא מיד לאותו אובייקט pipe named (קורא יחיד והתקשרות חד כיוונית).
- הגישה לאובייקטים אלו מתבצע באמצעות צורה בה ניתן לנקזים.
- הם נשאים במערכת הקבצים גם לאחר תום השימוש בהם, אלא אם מפעילים עליהם **unlink()**.

## mkfifo()

פונקציה ב-C המיצרת את אובייקט ה-named pipe.

חתימה: **int mkfifo(const char \*pathname, mode\_t mode)**

פרמטרים

○ pathname – הנתיב בו האובייקט יוצר

○ mode – הרשות שיננתנו לאובייקט

• ערך החזרה

○ 0 -> הצלחה

○ 1 -> כישלון

## Pipe Errors

SIGPIPE – סיגナル שנשלח באשר הכותב מנסה לכתוב, אבל הקורא בבר סגר את ה-file descriptor.

באשר הקורא מנסה לקרוא, אבל הכותב בבר סגר את ה-file descriptor, זהה התחנות חוקית, לאחר והקורא פשוט קיבל EOF,

ונקרא 0 בתים.

## Memory Mapping

אפשר ל맵ות קובץ לתוך הדיסקון (אפשר לעבד עם קבצים יותר גדולים מהדיסקון הראשי)

• מtbody בעזרת mmap.

אפשר לנו לעבד עם דיסקון משותף לתהליכיים (מעין sque)

## mmap()

memory map •

system call •

חתימה: **void\* mmap (void start, size\_t length, int prot, int flags, int fd, off\_t offset)**

פרמטרים

○ start – מיקום התחלתי מועדף למיקום הקובץ באחסן. אם מועבר NULL אז ה-OS תבחר עצמה.

○ length – במו הביטים שאנו חנכו ממפיים.

○ prot – הרשות גישה לדיסקון.

○ flags (לא הורחב בתרגול)

○ fd – הקובץ שאנו חנכו רצימם למפות.

○ offset – נקודת בקובץ ממנה נרצה להתחיל למפות.

• ערך מוחזר – מצביע לאזור הממופה.

## munmap()

.memory unmap •

system call •

מוחק את המיפוי. •

חתימה: **int munmap(void \*start, size\_t length)**

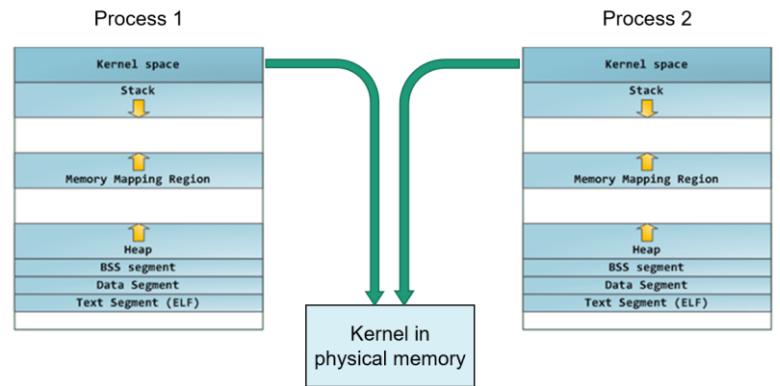
מעדכן (flushing) לאחסן (disk) את כל השינויים שהיו בקובץ.

## **msync()**

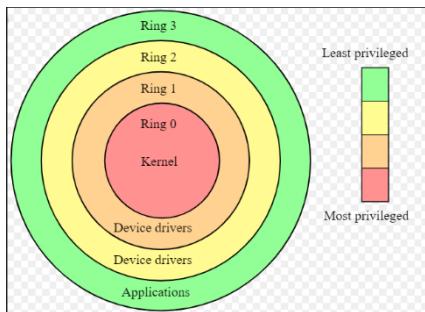
- .memory sync
- system call ופונקציה ב-C.
- מעדכן (flushing) לאחסון (disk) את כל השינויים שהו בקובץ.
- חתימה: **int msync(void \*start, size\_t length, int flags)**
- **דגלים אפשריים**
  - MS – תבצע את הסנכרון מיד (אנו מחייב עד שתסתיים)
  - MS\_ASYNC – תسانכרן בשתוול (אנו אמשר הלאה)
  - בלינוקס האפשרות השנייה היא OP\_NO (לא פעולה חוקית מאחר ובLINUKS תמיד נחבה עד שהסנכרון יקרה)

## Address Space

### Kernel mapping



### Pages Protection



| ring 3                                                                | ring 0                                                    |
|-----------------------------------------------------------------------|-----------------------------------------------------------|
| pages with user program (root too)                                    | pages with kernel code, device drivers, etc ...           |
| מי שכךן מוגבל בפקודות שהוא יכול להריץ ורגיסטרים שהוא יכול לבתוב אליהם | מי שכךן יכול לעשות הכל. יש הרשאות לגשת לכל בתובת בזיכרון. |

- בעת ביצוע קוד שאינו שייך ל-ring 0 נחוץ בתור ring 3. כאשר יהיה קוד שייך ל-ring 0 שנוצר לבצע, נעבר ל-ring 0, בצעו אותו ונחזור ל-ring 3.

| "Userland" / User Space     | Kernel "wonderland" / Kernel Space                                                                    |
|-----------------------------|-------------------------------------------------------------------------------------------------------|
| יש הקשר לתהיליך             | אפשר להיכנס לולאות אינסופית כי אין מישחו שמהפך עליינו כמו שקרה ב-user space user שה-kernel מפהיך עליו |
| יש את הספירה הסטנדרטית של C | אי אפשר להשתמש בספריות הסטנדרטיות                                                                     |
| לא נוגעים בזיכרון           | באגים הם מאוד קריטיים -> עלולים לגרום לקיריסט מערכת (kernel panic)                                    |

- מטרתם: להסתייר את פרטי החומרה של התקן.
- לא אפשרי שיהיה לנו בקורס את כל הדרייברים האפשריים.
- הפתרון: **מודולים טעינים (modular)** - טענים (inject) לKERNEL את מה שצריכים בצורה דינמית בזמן ריצה.
- רק מי שהוא privileged יכול לטען מודולים לKERNEL.

| Network Device                      | Block Device                                                                 | Character Device                                                                    |
|-------------------------------------|------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| Abstraction for <b>data packets</b> | Abstraction for read and written in multiples of <b>block, random access</b> | Abstraction for stream of bytes, read and written directly <b>without buffering</b> |
| e.g. ethernet card                  | e.g. disk, cdrom                                                             | e.g. monitor, keyboard                                                              |

- העבודה מול הדרייברים נעשית בתצורה של עבודה מול קובץ.
- קובץ לבל device.
- מודול = דרייבר לחומרה.

### Overloading of “driver” term

- קיימים 2 דברים שונים שמתיחסים אליהם בתור driver:
1. **דריבר של התקן** – משאנו שיש להתקן המשמש בתור מתווך. לאחר והתקן איטי ומערכת הפעלה מהירה, נוצר הדרייבר שתפקידו לקבל תקשורת מהירה ממכלול הפעלה ולהעביר אותה בצורה איטית עבור התקן ובכך ההפך בכוון השני.
  2. **דריבר של מערכת הפעלה** – מודול שמתפקידים על הKernel כדי שיוכל לתקשר עם התקן בלחשו (כמו שפעם היוינו צריכים להתקין דרייבר עם דיסק כאשר היוינו מחברים אליו למחשב, למשל: דיסק אוון קי).

### Character Device

- זה לא קובץ (אין לו בתים על הדיסק).
- זה נראה כמו קובץ מצד המשתמש ( עושים לו פעולה של קבצים: open, read, close. אבל אין אפשרות לעשות לו ioctl).

### major number

- יש כזה לבב device.
- קובע את סוג ה-device -> קובע איזה driver יורץ.
- עבור כל סוג device יש מס' major שהוא צריך לקבל.
- יש טווח עבור devices שלא הוגדר להם מס' major.

### minor number

- יש כזה לבב device.
- קובע את המס' הסידורי של התקן מאותו סוג.
- ממוספרים החל מ-0.

## struct file

- מוגדר ב-<linux/fs.h>
- אין לו שום קשר עם FILE שמו גדר בספריות של C.
- מיצג קובץ פתוח ונוצר לאחר שקוראים ל-open.
- מעבר ב프로그램 לבב פונקציה שמבצעת משוח על הקובץ.
- השדות החשובים שלו:

|                                                                                                                                                                                   |                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------|
| מצין האם הקובץ ניתן לקריאה/לכתיבה (יתכן שנייהם)<br><b>DMODE_READ, F_MODE_WRITE</b>                                                                                                | <b>mode_t f_mode</b>                |
| מצין את המיקום בקובץ<br><b>loff_t = long long</b>                                                                                                                                 | <b>loff_t f_pos</b>                 |
| דائلם                                                                                                                                                                             | <b>unsigned int f_flags</b>         |
| הפעולות שניתן לבצע על הקובץ                                                                                                                                                       | <b>struct file_operations *f_op</b> |
| פונטרא פמי שאנו יכולים להשתמש בו לאיזה צורך שרק נרצה (לצורך העברת מידע עם ה-struct)<br>למשל: יצירת מבנה נתונים כלשהו שאנו חnage צרכי ואחסן הפונטרא אליו ב-<br><b>private_data</b> | <b>void* private_data</b>           |
|                                                                                                                                                                                   | <b>struct dentry *f_dentry</b>      |

## struct file\_operations

- מוגדר ב-<linux/fs.h>
- מכל מצביעים לפונקציות שניתן להפעיל על **fd**.
- מספק אותו בתוכו פרטן שיתן מידע על אילו פונקציות אנחנו מרשימים להפעיל על ה-device driver שלו.
- אפשר לשים NULL על פונקציות שאנו לא רוצים למש, וזה אם תקרא הפונקציה הזאת אז הקיראה תכשל.
- שדות:
  - נגיד בכי character device לא יסגר באמצעות שאנו עובדים עליו.
  - owner ○
  - read** ○
  - write** ○
  - open** ○
  - unlocked\_ioctl** ○
  - תקרא באשר המשתמש יבצע **close** ○

- // This structure will hold the functions to be called  
// when a process does something to the device we created  

```
struct file_operations Fops =
{
 .owner = THIS_MODULE,
 .read = device_read,
 .write = device_write,
 .open = device_open,
 .unlocked_ioctl = device_ioctl,
 .release = device_release,
};
```

| ערך החזרה            | פרמטרים                                                                                                                                                                                           | חתימה                                                         |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|
|                      | <b>inode</b> – מתחכו אפשר לשלוף את ה-#minor<br><b>file*</b> – פוינטר ל-struct file* שאנו חוננו רוצים לפתוח                                                                                        | int (*open) (struct inode*, struct file*)                     |
| במה בתים נקראו בפועל | <b>file*</b> – פוינטר ל-struct file* שאנו חוננו רוצים לקרוא<br><b>char*</b> – באפר לתוך המידע שנקרה יכתב<br><b>size_t</b> – במה אנחנו רוצים לקרוא<br><b>loff_t*</b> – אופסט פוינטר                | ssize_t (*read) (struct file*, char*, size_t, loff_t*)        |
| במה בתים נכתבו בפועל | <b>file*</b> – פוינטר ל-struct file* שאנו חוננו רוצים לכתוב אליו<br><b>char*</b> – באפר עם המידע שאנו חוננו רוצים לכתוב<br><b>size_t</b> – במה אנחנו רוצים לכתוב<br><b>loff_t*</b> – אופסט פוינטר | ssize_t (*write) (struct file*, const char*, size_t, loff_t*) |
|                      | <b>file*</b> – פוינטר ל-struct file* שאנו חוננו רוצים לכתוב אליו<br><b>loff_t</b><br><b>int</b> – בכמה אנחנו רוצים להזיז את האופסט קובץ<br>character device <b>llseek</b>                         | loff_t (*llseek) (struct file*, loff_t, int)                  |
|                      |                                                                                                                                                                                                   | int (*ioctl) (struct file*, unsigned int, unsigned long)      |
|                      | <b>file*</b> – פוינטר ל-struct file* שאנו חוננו רוצים לכתוב אליו<br><b>unsigned int</b> – המס' פקודה של ioctl (מה אנחנו בוחרים)<br><b>unsigned long</b> – פרמטר נוסף שאנו יכולים להשתמש בו        | int (*flush) (struct file*)                                   |

**ioctl() - Kernel Space**

חתימה: **int (\*ioctl) (struct file\*, unsigned int, unsigned long)** •  
**פרמטרים** •

- **file\*** – פוינטר ל-struct file\* שאנו חוננו רוצים לכתוב אליו
- **unsigned int** – המס' פקודה של ioctl (מה אנחנו בוחרים)
- **unsigned long** – פרמטר נוסף שאנו יכולים להשתמש בו
- **ערך החזרה:** תלויה במימוש (מה אנחנו בוחרים), אבל בד"כ:
  - 0 -> הצלחה
  - <0 -> כישלון.

**ioctl() - User Space**

נשתמש בה בתור syscall לשנרצה לקרוא ל-ioctl על קובץ בלשחו.  
**.sys/ioctl.h** •  
**נכילת-ב-ה-** •

**register\_chrdev()**

פונקציה ב-kernel.  
**חתימה:** **int register\_chrdev (unsigned int major, const char\* name, struct file\_operations\* fops)** •  
**פרמטרים** •

- **major** – המס' major המבוקש. 0 עבור הקצתה דינמית (נקבל מס' באופן אוטומטי ע"י הernal)
- **name** – השם של ה-character device (במו שהוא יופיע ב-/proc/devices)
- **fops** – מבצע ל-struct המכיל מבצעים לפונקציות של fd
- **ערך החזרה**

- הצלחה -> 0
- כישלון -> ערך שלילי + תיאור השגיאה ב- errno (יכול להימשך כי יש בברדריבר שמתפלב ב-#majors שננתנו בפרמטר)
- מס' חיובי -> ה-#major שהוקצה לו אם ספקתי 0 בתור פרמטר ל-major.

## unregister\_chrdev()

.character device kernel שמנטלת את הרישום של character device ב-kernel.  
חtinyה: ●  
**void unregister\_chrdev (unsigned int major, unsigned int minor, unsigned int count, const char\* name)** ●  
פרמטרים ●  
ונקציה major – המס' major המבוקש. 0 עברו הקצאה דינמית (נקבל מס' באופן אוטומטי ע"י הkernel).  
minor – המס' minor הראשון בטוויה ה-major-im של ה-major.  
count – מס' ה-minor-im שה-character device זהה מאבלס.  
name – שם של ה-character device (כמו שהוא יופיע ב-/proc/devices) ○

## יצירה של device חדש

**mknod <name> <type> <major> (<minor>)** - פיקוד באש עברו זה. ●  
פרמטרים ●

name – השם של ה-device החדש.  
.block <- b ,character <- c – type ○  
.device major – major ○  
.device minor – minor ○  
לדוגמה: sudo mknod /dev/simple\_char\_dev c 250 0 ●  
אפשר להסיר עם rm ●  
בשנעשה זו בתקייה שבה נוצר יופיע בתחילת הרשאות c או b בהתאם לסוגו. ●

- יש כמה מימושים עבור ה-kernel של ספריות מה-user space. למשל: .string.h

- למרות שאפשר, מומלץ לא לגשת שירות לבתובות ב-user space (יתכן והבתובות לא חוקיות). במקום עדיף להשתמש ב:

### **put\_user(x, ptr)**

.user space-ל kernel space (int או char) מה-ptr משטנה פשוט (int או char).

#### פרמטרים

- x – המשתנה שערכו יועתק ל-user space.
- ptr – בתובת היעד ב-user space.

#### ערך החזרה

- 0 -> הצלחה
- EFAULT -> שגיאה

### **get\_user(x, ptr)**

kernel space-ל user space (int או char) מה-ptr משטנה פשוט (int או char).

#### פרמטרים

- x – המשתנה שייחסן את התוצאה.
- ptr – בתובת המקור ב-user space.

#### ערך החזרה

- 0 -> הצלחה
- EFAULT -> שגיאה

## **vmalloc()**

פונקציה להקצת זיכרון על ה-kernel לאובייקטים שקטנים יותר מוגדל של דף.  
הזיכרון יוקצה בצורה רציפה בזיכרון הווירטואלי בלבד.

חתימה: **void\* vmalloc(unsigned long size)**

#### פרמטרים

- size – כמה בתים מבקשים להקצתה

#### ערך החזרה

- הצלחה -> מצביע לזכרון שהוקצתה
- כשלון -> 0

## **kmalloc()**

פונקציה להקצת זיכרון על ה-kernel לאובייקטים שקטנים יותר מוגדל של דף.  
הזיכרון יוקצה בצורה רציפה בזיכרון הפיזי ובזיכרון הווירטואלי.

חתימה: **void\* kmalloc(size\_t size, gfp\_t flags)**

#### פרמטרים

- size – כמה בתים מבקשים להקצתה
- flags – איזה סוג זיכרון להקצתה

#### דגלים

GFP\_KERNEL – הקצת זיכרון ram normal kernel. הקראיה לפונקציה עשויה לישון (אם ה-OS החלטה להשהות את התהילר). זהה הדרך הביא אמינה להקצת זיכרון.

#### ערך החזרה

- הצלחה -> מצביע לזכרון שהוקצתה
- כשלון -> 0

## kfree()

- פונקציה לשחרור זיכרון על ה-kernel
- **חתימה:** void **kfree** (const void\* **obj**)
- **פרמטרים**
  - kmalloc – פונטרא ל זיכרון שהוקצה ע"י **obj** – פונטרא ל זיכרון שהוקצה ע"י kmalloc

## printf()

- **אפשרות להדפיס הודעות שונות שייצגו ב-dmesg**
- **לפי סדר החומרה:**
  - printf(" ") – יודפס לוג
  - printf(KERN\_DEUBG " ") – יודפס לוג
  - printf(KERN\_INFO " ") – יודפס לוג
  - printf(KERN\_NOTICE " ") – יודפס לוג
  - printf(KERN\_WARNING " ") – יודפס לוג
  - printf(KERN\_ERR " ") – יודפס לוג
  - printf(KERN\_CRIT " ") – יודפס לוג
  - printf(KERN\_EMERG " ") – יודפס לוג

## מודול שרך על ה-Kernel

- משתמשים במודולים בתור הרחבת kernel.
- אם מודולים של kernel יחוירו שגיאה אז הערך של errno יהיה שלילי.

### #define \_\_KERNEL\_\_, #define MODULE

בשביל שנוכל לקבל את הפונקציות הרלוונטיות ל-.include.

כלומר יש פונקציות ומבני נתונים שרלוונטיים לקוד שרך ב-kernel ועתופים ב-

```
#ifdef __KERNEL__
#include...
#endif
```

```
#ifdef MODULE
```

```
#include...
```

```
#endif
```

לעתים נראה בקוד מקומות שלפני define יכתבו גם undef#, כלומר:

```
#undef MODULE
```

```
#define MODULE
```

אבל זה מיותר ולא נחוץ

### #include <linux/init.h>

- מכילה את ה-macros
  - module\_init()
  - module\_exit()

## פונקציית initialization

בכל פעם שתעוניים (insmod) מודול ל-kernel יש לנו אפשרות להריץ פונקציה שנגדיר בתור פונקציית אתחול.

בפונקציה זו נשים את כל הדברים שנרצה שיתקיים עם אתחול המודול:

- הרשמה לשירותים.

**חתימה:** static int funcName(void)

- ערך החזרה

- הצלחה -> 0

- בישלון -> מס' השגיאה

ניתן לרשום את המאקרו \_\_init\_ לפני שם הפונקציה

## פונקציית unloader

פונקציה שתורץ עם ירידת המודול מה-kernel.(rmmod)

בפונקציה זו ננקה אחרים:

- נשחרר זיכרון שאנו לא צריכים.

**חתימה:** static void funcName(void)

- ערך החזרה: אין

## module\_init(<initFuncName>)

- מארקו המבצע רישום של הפונקציות שלו בטור הפונקציות שירצעו עם הulat המודול מה-kernel.
- קורא ל-.register\_capability()

### module\_exit(<unloaderFuncName>)

- מארקוים המבצעים רישום של הפונקציות שלו בטור הפונקציות שירצעו עם הulat/הורדת המודול מה-kernel.

```
#define __KERNEL__
#define MODULE

#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");

----- loader -----
static int hello_init(void){
 printk("Hello, Kernel World!\n");
 return 0;
}

----- unloader -----
static void hello_cleanup(void)
{
 printk("Cleaning up hello module.\n");
}

module_init(hello_init);
module_exit(hello_cleanup);
```

### המארקוים \_\_exit, \_\_init

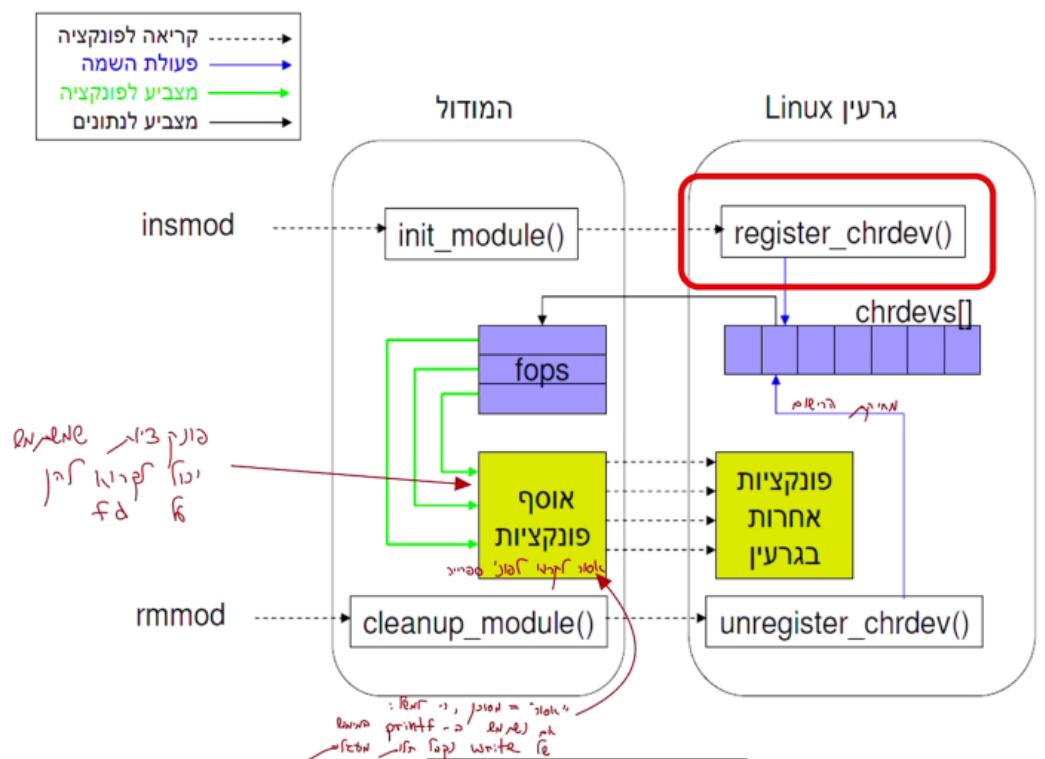
- מארקוים שניצן להוסיף בחתימה של פונקציות הטעינה וההסבירה מהקרבל לפני השמות שלהם.
- נשתמש בהם עבור מודולים שנרצה שלא יוסר מהקרבל (למשל: מודול של מקלחת).
- התוצאה תהיה שהקוד של פונקציית הטעינה ימחק מהקרבל לאחר reboot ושהקוד של פונקציית ההסרה לא יוסר מהקרבל.

### מארקוים נוספים שבדאי להכיר

- MODULE\_LICENSE("GPL") – לא הורחוב עליו. משחו שפיטוט צריך להיות אצלנו מבחינה משפטית.
- MODULE\_AUTHOR("GT") •
- MODULE\_DESCRIPTION("...") •
- נראה את הפירוט זהה אחריו שבירץ: modinfo moduleName •

- makefile •
  - לא נדרש להבין איך הוא עובד.
  - מקבלים אחד מוקן עבור התרגילים.
  - קמפול עם פקודה **.make**.
  - התוצר הוא קובץ עם סימת **ko** (Kernel Object).
  
- load •
  - sudo insmod filename.ko** ○  
"insert module"
  
- verify •
  - sudo lsmod** ○  
"list modules"
  
- unload •
  - sudo rmmod filename** ○  
"remove module"
  
- בshell לראות את ההדפסות ב-log

# Character device driver



ברגע שעושים `insmod` גורמים לקריאה לפונקציית אתחול של המודול.

עבור מודולים שאמורים לעבוד מול char device נבצע קריאה ל-`register_chrdev()` בתוך פונקציית האתחול – וזאת כדי שהKERNEL יידע מי המודול שעבוד מול אותו char device. כל char device מיוצג על ידי מס' מג'ורי ומינורי.

המס' המינורי מאפשר לנו לטפל בכמה char devices במקביל, כי ניתן להגדיר מודול שאחראי על כמה מס' מינוריים. ככלומר, המס' המינורי הנוכחי ועוד עבור ה-device driver, בעוד המס' המג'ורי הנוכחי בכדי "להגיד" לKERNEL מי מטפל ב-character device.

## panic()

- פונקציה שרצה בזמן kernel panic .kernel panic
- מה היא עשויה?
- לא אפשררת interrupts
- מקפיאה את כל התהליכים מלבד התחילר הנוכחי ( panic() - שקרא ל-panic)
- ריצה ב-kernel למשך כמה זמן
- קוראת ל- emergency\_restart()

## File System

- איך זה נראה?

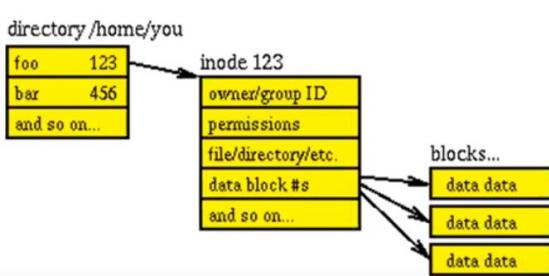
| ὑπόρμηט הפעלה                                                                                                                                           | ὑπόρμηט                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| יש הפרדה בין הייצוג של הזיכרון במערכת הפעלה לבין מה<br>שקורה בפועל<br>התיווך נעשה ע"י FS-<br>ה-FS העיקרי ב-Linux : ext4<br>ה-FS העיקרי ב-Windows : NTFS | מערכת היררכית של קבצים ותיקיות<br>ב-Windows: כוננים<br>ב-Linux: תיקית שורש (root) : "/" |

## VFS (= Virtual FS)

- מספק אבסטרקציה על FSs.
- בזכותו ניתן לעבוד עם כמה FSs שונים ולהתנהל מולם באמצעות צורה.
- ניתן לעבוד כבה מאחר וככל FS זהה VFS יודע לעבוד מולו.

## VFS Objects

- הייצוג של FSs ב-kernel הוא באמצעות האובייקטים (structs) הבאים:



### 1. inode

- מכיל את המידע על inode למעט השם שלו.
- לא מכל את השם מאחר וייתכן ול-inode יש יותר שם אחד.
- מכיל את המס' של ה-inode (מס' ייחודי).
- הרשות.
- Owner.
- Group owner.

### 2. file

- נוצר עבור קובץ שפתחו אותו.
- יש פירוט על ה-struct כמה עמודים מעלה.

### 3. Superblock

- מטה נתונים כללי של FS.

### 4. Directory entry (dentry)

- מכיל שם של אובייקט והמס' inode שלו.
- האובייקטים הנ"ל נשמרים בזיכרון.
- שינויים מהאובייקטים שקיים על הדיסק

|                                                                               |                                                                                                                                                                                                    |       |
|-------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
|                                                                               | root FS                                                                                                                                                                                            | /     |
|                                                                               |                                                                                                                                                                                                    | /sys  |
| • FS מיוחד                                                                    | שמאפשר לנו לדעת הרבה דברים על מה שרצ ברגע על ה-OS<br>חשוב להבין שלא מדובר כאן בספריה, אלא FS. הצד שני, ה-FS זהה לא קיים בשום דיסק, אלא שמור בזיכרון<br>ונוטן לנו לגשת למידע באופן שבו ניתן לקבצים. |       |
| • ספירה לכל תהליך שרע                                                         | מוביל "קובץ" cpufreq - מידע על המעבד<br>diskstats                                                                                                                                                  | /proc |
| • פעם זו הייתה תיקיה ב-<br>device files                                       | root FS<br>היום ממומש בתווך devFS<br>כלומר הוא FS בעצמו שמכיל את כל ה-                                                                                                                             | /dev  |
| • בד"כ ספירה (תיקיה) רגילה ב-<br>root FS                                      | מוביל קבצי קונפיגורציה של המערכת<br>מידע על המשתמשים<br>לא מכיל את הסיסמאות                                                                                                                        | /etc  |
| • בד"כ ספירה (תיקיה) רגילה ב-<br>root FS                                      | קובצי לוג<br>מייל                                                                                                                                                                                  | /var  |
| • בד"כ ספירה (תיקיה) רגילה ב-<br>root FS<br>קבצים זמינים שנמחקים ב-<br>reboot |                                                                                                                                                                                                    | /tmp  |

**Hard Link**

- שם נוסף לאותו קובץ.
- חייב להיות בתוך אותו FS, אך יכול להיות בספריות שונות.
- אסור ליצור Hard Link בספריות.**
- שימושי לעתים רוחקות.
- יצירה של link :hard link
- In file1.txt hard\_link\_to\_file1.txt
- ביצירת hard link
- מס' ה-reference count עולה (מופיע ב-*ls* אחרי ההרשאות)
- לא מייצר inode חדש אלא לוקח inode קיים שמצוין על שם בולשחו ומיציר ממנו הצבעה אל הקובץ
- אם נשנה את ההרשאות של קובץ, אך נשנה בכר את ההרשאות של כל links hard שמצוינים אליו, מאחר וההרשאות נשמרות ב-*inode*.

**Symbolic (soft) Link**

- כמו קיצור דרך ב-windows.
- קובץ שמכיל את השם של הקובץ שעליו הוא מצביע.
- מזה דאטה שאומר שמדובר ב-link symbolic
- משתמשים בו הרבה.
- יצירה של link :soft link
- In -s file1.txt soft\_link\_to\_file1.txt
- ביצירת soft link
- מס' ה-reference count לא משתנה
- מייצר inode חדש שמצוין על שם
- ב-*ls* :
- נראה על מה הוא מצביע (יופיע "*->*" בסוף השורה עם הקובץ שהוא מצביע עליו)
- יופיע בתחילת ההרשאות

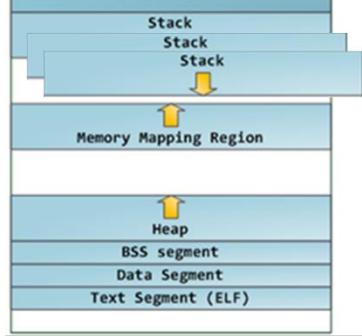
- ההרשאות ל-link symbolic הן תמיד כל ההרשאות האפשרות. ההרשאות אלו בפועל הן ההרשאות של הקובץ אליו הקישור מוביל.

### בגדר: link - קישור ל-node inode, soft link – קישור לנטייב (שמי)

### **fsync(fd)**

- system call שמודיא שככל ה-data meta וה-data flush העברים של ה-fd להתקן האחסון.
- לעיתים זה לא מספיק לעשות את זה רק על הקובץ ויש צורך לעשות את זה גם על הספרייה.

## Threads



- "lightweight processes"
- **כשיש תהליך עם כמה חוטים, אז כמעט הכל משותף למעט המחסנית.**
- תהליכים מתחילהם עם חוט עיקרי אחד – ה-main.
- תהליכים יכולים להוסיף/להוריד חוטים במהלך ריצתם.
- כאשר החוט של main מסתים (עשה exit/main/return/מגיע ל-{}) אז כל החוטים האחרים של אותו תהליך נעצרים (בשינוי בין תהליך אב ובן).
- **לא לעורב יצירה של חוטים עם fork + exec + ↶ יוצר בעיות!**

## clone

- קראת מערכת מאוד "פרימיטיבית" וכן נועד לשימוש בספריות אחרות כדוגמת הספרייה של POSIX.

## POSIX Threads

- ספריה עוטפת של קראת המערכת clone.
- יש להוסיף את הדגל pthread- ב-gcc.
- יש לעשות include pthread.h - ל-

### pthread\_create()

- פונקציה לייצור חוט חדש שיתחיל להריץ את הפונקציה שסופקה לו בתור פרטנר.
- **חתימה**
- **int pthread\_create(pthread\_t\* thread, pthread\_attr\_t\* attr, void\* (\*start\_routine)(void\*), void\* arg)**
- **פרמטרים**
  - – מצביע שייחסן את ה-id thread החוט שנוצר
  - – אפשרות קונפיגורציה שונות (בד"כ נעלם NULL)
  - – מצביע לפונקציה שהחוט יבצע
  - – הארגמנטים עבור ריצת הפונקציה start\_routine
  - – ערך החזרה
- **הוצאה - <0**
- **כשלון - > ערך שונה מ-0**

### pthread\_exit()

- עברו חוט שאלמו החוט הראשי (main) – פונקציה שחוט שරוצה לסיים את עבודתו יקרה לה (שקל עבורי ל-0).
- עברו החוט הראשי (main) – החוט הראשי ימתין לכל החוטים האחרים עד שישו את עבודתם.
- **()() exit על כל אחד מהחוטים יגרום לצאת כל החוטים של התהילן.**
- **חתימה: void pthread\_exit(void\* retval)**
- **פרמטרים**
  - – מצביע לערך ההחזרה של הפונקציה (ניתן למשל לשימוש ב: EXIT\_SUCCESS)

### pthread\_self()

- **חתימה: pthread\_t pthread\_self()**
- **ערך החזרה: ה-id thread של החוט שקרה לפונקציה.**

## **pthread\_cancel()**

- פונקציה לביטול הריצה של החוט שמסופק בפונקטר.
- כל חוט יכול להרוג את הריצה של כל חוט אחר (כולל עצמו והחוט הראשי).
- **חתימה:** `int pthread_cancel(pthread_t th)`
- **ערך החזרה:** ה-`id` של החוט שקרה לפונקציה.

## **pthread\_join()**

- המקבילה של `wait` בתהליכים.
- נשתמש בה כאשר אנחנו רוצים **להמתין לסיום ריצתו של thread מסוים**.
- כל חוט שרצחנו להמתין לחוט אחר וידעע את ה-`id` שלו יוכל לעשות זאת (לא רק החוט הראשי יכול).
- אם החוט שרצינו להמתין לו כבר סיים את עבודתו אז הקראיה תחזיר מייד.
- לא ניתן לעשות `join` לחוט שעשו לו כבר `join` undefined behavior <-join undefined behavior>.
- **חתימה:** `int pthread_join(pthread_t th, void ** thread_return)`
- **פרמטרים**
  - ה-`th` – ה-`id` thread שרצו ללחוץ/להצטרכו אליו.
  - פוינטר שייחסן את ערך ההחזרה של החוט שהצטרכנו אליו.
- **ערך החזרה**
  - הצלחה -> 0
  - כישלון -> מס' שגיאה

## **Synchronization**

- כאשר "רבים" הרבה על הנעילה אז שימוש ב-`mutex` יהיה הרבה יותר איטי מאשר שימוש ב-`atomic` (באשר שימוש ב-`atomic` אפשר).
- במקרים אחרים, בהם לא "רבים" הרבה על הנעילה, אז שימוש ב-`mutex` יהיה יותר מהיר.

## **Atomic Variables**

- מאפשר להגדיר משתנים שכל הפעולות עליהם יהיו אטומיות – רלוונטי עבור משתנים פרימיטיביים בלבד.
- יש לעשות `#include <stdatomic.h>`

## **Mutex (Mutual Exclusion)**

- מאפשר להגדיר קוד שרק חוט אחד יוכל לבצע בו זמנית.

## **pthread\_mutex\_init()**

- **מאתחלת את ה-`t` struct `pthread_mutex_t` – mutex**
- **חתימה:** `int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutex_attr_t * mutexattr)`
- **פרמטרים**
  - `mutex` – struct `pthread_mutex_t` שאנחנו רוצים לאותחל.
  - `mutexattr` – אפשרויות אתחולן שונות. אם רוצים את הדיפולטיבי –> להעביר `NULL`.
- **ערך החזרה**
  - הצלחה -> 0
  - כישלון -> מס' שגיאה

## **pthread\_mutex\_destroy()**

- משמידה את ה-**t**-**pthread\_mutex** שלנו.
- אם הורסים את המנעול תוך כדי שימושו מחייב אותו -> undefined behavior.
- **חתימה:** `int pthread_mutex_destroy(pthread_mutex_t* mutex)`
- **פרמטרים**
  - **pthread\_mutex\_t – mutex** – struct **pthread\_mutex** שאנו רוצים להשמיד.
- **ערך החזרה**
  - הצלחה -> 0
  - בישלון -> מס' שגיאה

## **pthread\_mutex\_lock()**

- מנשה לנעול את המנעול עד שהוא מצליח.
- כמובן, הקראיה לפונקציה **pthread\_mutex\_lock** ממשיכה להתבצע עד שהוא מצליח לנעול את המנעול.
- **חתימה:** `int pthread_mutex_lock(pthread_mutex_t* mutex)`
- **פרמטרים**
  - **pthread\_mutex\_t – mutex** – struct **pthread\_mutex** שאנו רוצים להשמיד.
- **ערך החזרה**
  - הצלחה -> 0
  - בישלון -> מס' שגיאה

## **pthread\_mutex\_trylock()**

- מנשה לנעול את המנועל **פעם אחת** ואם לא הצליח מוביל לנעול אותו.
- **חתימה:** `int pthread_mutex_trylock(pthread_mutex_t* mutex)`
- **פרמטרים**
  - **pthread\_mutex\_t – mutex** – struct **pthread\_mutex** שאנו רוצים להשמיד.
- **ערך החזרה**
  - הצלחה -> 0
  - בישלון -> מס' שגיאה

## **pthread\_mutex\_unlock()**

- משחררת את המנעל.
- **חתימה:** `int pthread_mutex_unlock(pthread_mutex_t* mutex)`
- **פרמטרים**
  - **pthread\_mutex\_t – mutex** – struct **pthread\_mutex** שאנו רוצים להשמיד.
- **ערך החזרה**
  - הצלחה -> 0
  - בישלון -> מס' שגיאה

חוט consumer וחותט producer •

## Events & Condition Variables

- נuded על מנת שנוכל להודיע מוחוט אחד לשני שאירוע מסוים שהחוט השbie חיכה לו קרה, ועד שהאירוע הזה יקרה החוט השbie יוכל לישון ובכך לא לבלבץ זמן מעבד על **busy waiting**.

### **pthread\_cond\_init()**

- מאתחלת את ה-t struct **pthread\_cond** שלנו.
- חתימה:** `int pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* condattr)`
- פרמטרים**
  - struct **pthread\_cond\_t** – **cond**
  - אפשרויות אתחול שונות. אם רוצים את הדיפולטיבי -> להעביר NULL.
- ערך החזרה**
  - הצלחה -> 0
  - בישלון -> מס' שגיאה

### **pthread\_cond\_destroy()**

- משמידה את ה-t struct **pthread\_cond** שלנו.
- אם הורסים את המנגול תוק כדי שימושו מחזיק אותו -> undefined behavior.
- חתימה:** `int pthread_cond_destroy(pthread_cond_t* cond)`
- פרמטרים**
  - struct **pthread\_cond\_t** – **cond**
- ערך החזרה**
  - הצלחה -> 0
  - בישלון -> מס' שגיאה

### **pthread\_cond\_wait()**

- מכניסה את החוט שקורא לפונקציה במצב שינה עד שה-**cv** שהוא מחליף לו יקרה והמנגול יהיה פניו לנעליה.
- בנוסף לכניסה במצב שינה, הקריאה לפונקציה גם משחררת את הנעליה, ובאשר חוזרים לאחר נקודה בקוד לאחר שהחוט קיבל signal איז המנגול ינעל מחדש.
- חתימה:** `int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex)`
- פרמטרים**
  - struct **pthread\_cond\_t** – **cond**
  - mutex – מנגול שעליו הוא ישן. **המנגול חייב להיות נעל וברשותנו!**
- ערך החזרה**
  - הצלחה -> 0
  - בישלון -> מס' שגיאה

### **pthread\_cond\_signal()**

- מעירה את אחד החוטים (רנדומלית) מבין החוטים שישנים על ה-**cv**, כדי שיבדק את המנגול שלו.
- יתעורר לפחות חוט אחד (יתכן שתיתעורר "בטיעות" יותר מחד).
- לא צריך שה mutex יהיה בבעלותנו** כדי לשלוח את הסיגナル.
- אם אף אחד לא מאזין ל-wait אז הסיגナル הולך לאיבוד.
- חתימה:** `int pthread_cond_signal(pthread_cond_t* cond)`

## פרמטרים •

- ○ struct **pthread\_cond\_t** – cond
- ○ ערך החזרה •
  - ○ הצלחה - < 0
  - ○ כישלון - > מס' שגיאה

## **pthread\_cond\_broadcast()**

- מועירה את כל החוטים שישנים על ה-av (כדי שיבדקו את המנגול שלהם).
- לא צריך שה-mutex יהיה בבעלותנו כדי לשלוח את הסיגナル.
- אם אף אחד לא מאמין ל-wait אז ה-broadcast הולך לאיבוד.
- **חתימה:** int pthread\_cond\_signal(**pthread\_cond\_t\*** cond)
- **פרמטרים**
- ○ struct **pthread\_cond\_t** – cond
- ○ ערך החזרה •
  - ○ הצלחה - < 0
  - ○ כישלון - > מס' שגיאה

## פעולות הקשורות ל-SF

### **opendir()**

- פותחת directory stream
- **חתימה:** **DIR\*** opendir(**const char\*** dirname)
- **פרמטרים**
- ○ dirname – הנתיב לתיקייה.
- ○ ערך החזרה: •
  - ○ הצלחה - > מצביע ל-stream שנפתחה.
  - ○ כישלון - > NULL או errno יוביל את הטעות.

### **closedir()**

- סוגרת directory stream
- **חתימה:** int closedir(**DIR\*** dirp)
- **פרמטרים**
- ○ dirp – ה-directory stream שברצוננו לסגור
- ○ ערך החזרה: •
  - ○ הצלחה - < 0
  - ○ כישלון - > errno יוביל את הטעות.

### **readdir()**

- קוראת את התוכן של תיקייה
- **חתימה:** **struct dirent\*** readdir(**DIR\*** dirp)
- **פרמטרים**
- ○ dirp – ה-directory stream שברצוננו לקרוא
- ○ ערך החזרה: •
  - ○ הצלחה - > פוינטר ל-struct dirent

- הגנו לסוף ה-stream `<- NULL` והערך של `errno` לא ישתנה
  - (לשנות את ערכו ל-0 לפני הקריאה כדי להזות שגיאה).
- בישלון `-> NULL` ו-`errno` יכיל את הטעות.

## struct dirent

```
struct dirent {
 ino_t d_ino; /* Inode number */
 off_t d_off; /* Not an offset; see below */
 unsigned short d_reclen; /* Length of this record */
 unsigned char d_type; /* Type of file; not supported
 by all filesystem types */
 char d_name[256]; /* Null-terminated filename */
};
```

## stat()

- **מחזירה מידע.**
- אם הנתיב הוא ל-symbolic link אז היא מחזירה מידע על הקובץ שהוא מקשר אליו.
- **חתיימה:** `int stat(const char* pathname, struct stat* statbuf)`
- **פרמטרים**

  - pathname – הנתיב לקובץ שרוצים עליו מידע
  - statbuf – הבادر לתוכו המידע ישמר

- **ערך החזרה**

  - הצלחה -> 0
  - כישלון -> -1 ו-`errno` יכיל את הטעות.

## lstat()

- **מחזירה מידע על הקובץ שסופק בתור הבודר שסופק כפרמטר.**
- אם הנתיב הוא ל-link אז היא מחזירה מידע על ה-link עצמו ולא על הקובץ שהוא מקשר אליו.
- **חתיימה:** `int stat(const char* pathname, struct stat* statbuf)`
- **פרמטרים**

  - pathname – הנתיב לקובץ שרוצים עליו מידע
  - statbuf – הבודר לתוכו המידע ישמר

- **ערך החזרה**

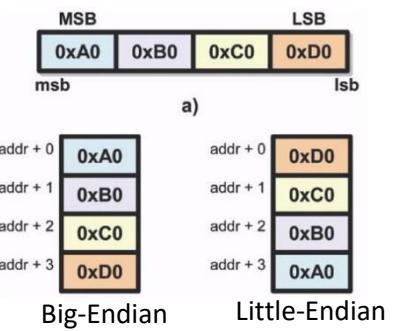
  - הצלחה -> 0
  - כישלון -> -1 ו-`errno` יכיל את הטעות.

```
struct stat {
 dev_t st_dev; /* ID of device containing file */
 ino_t st_ino; /* Inode number */
 mode_t st_mode; /* File type and mode */
 nlink_t st_nlink; /* Number of hard links */
 uid_t st_uid; /* User ID of owner */
 gid_t st_gid; /* Group ID of owner */
 dev_t st_rdev; /* Device ID (if special file) */
 off_t st_size; /* Total size, in bytes */
 blksize_t st_blksize; /* Block size for filesystem I/O */
 blkcnt_t st_blocks; /* Number of 512B blocks allocated */

 /* Since Linux 2.6, the kernel supports nanosecond
 precision for the following timestamp fields.
 For the details before Linux 2.6, see NOTES. */

 struct timespec st_atim; /* Time of last access */
 struct timespec st_mtim; /* Time of last modification */
 struct timespec st_ctim; /* Time of last status change */
```

- ה-MSB יהיה בכתביות הנמוכות יותר. (למשל: פקודות בראשת)
- ה-LSB יהיה בכתביות הנמוכות יותר. (למשל: x86)



- פונקציות להמרה ל-network big-endian : (host to network)

```
u_long htonl(u_long); // host to network long (32 bits)
u_short htons(u_short); // host to network short (16 bits)
```

- פונקציות להמרה ל-host big-endian : (network to host)

```
u_long ntohl(u_long); // network to host long (32 bits)
u_short ntohs(u_short); // network to host short (16 bits)
```

- סדר הקריאה שיש לבצע את ה-client וה-server.

| client                        | server                          |
|-------------------------------|---------------------------------|
| <code>sd=socket()</code>      | <code>sd=socket()</code>        |
|                               | <code>bind(sd, port)</code>     |
|                               | <code>listen(sd, ...)</code>    |
| <code>connect(sd, dst)</code> | <code>new_sd=accept(sd)</code>  |
| <code>write(sd, ...)</code>   | <code>write(new_sd, ...)</code> |
| <code>read(sd, ...)</code>    | <code>read(new_sd, ...)</code>  |
| <code>close(sd)</code>        | <code>close(new_sd)</code>      |

- יש לעשות include ל-:
  - `<sys/types.h>`
  - `<sys/socket.h>`

### socket()

- מיצרת `.socket`
- חתימה: `int socket(int domain, int type, int protocol)`
- פרמטרים
  - `domain` – ה-socket protocol family
    - .AF\_INET – בשעופדים עם IPv4
    - .AF\_INET6 – בשעופדים עם IPv6
  - `type`
    - TCP – עברו SOCK\_STREAM
    - UDP – עברו SOCK\_DGRAM
  - `protocol`
    - 0 – יגרום לכך שיבחר אוטומטית בך שיטאים ל-type.
- ערך החזרה:
  - הצלחה -> ה-socket descriptor שהוקצה עבור ה-.socket
  - כשלון -> errno יוכל את הטעות.

### struct sockaddr

- מכיל את הכתובות של ה-socket (IP + port)
- מכיל את השדות הבאים:
  - `address family` – unsigned short `sa_family`
  - .AF\_INET – בשעופדים עם IPv4
  - .AF\_INET6 – בשעופדים עם IPv6
  - `.port` – מס' ה-port – char `sa_data[14]`

## **struct sockaddr\_in ("in" for internet)**

- ניתן לעשות לו cast ל-`sockaddr`. וב"כ נעבד אותו ופישוט נעשה לו `.cast`.
- לא מתאים ל-IPv6
- מכיל את הכתובת של ה-socket (**socket + port**) (IP + IP).
- מכיל את השדות הבאים:
  - `AF_INET` – short int `sin_family`
  - `.port` – unsigned short int `sin_port`
  - struct `in_addr sin_addr`
  - `unsigned char sin_zero[8]`

## **struct in\_addr**

- לא מתאים ל-IPv6
- מכיל את הכתובת אינטרנט של ה-socket (IP).
- מכיל רק את השדה:
  - `uint32_t s_addr` – כתובת ה-IP

## **client side – connect()**

- מחברת את ה-port ל-socket ו-IP שאנו רוצים ליצור איתם חיבור (בד"כ מדובר ב-IP של host אחר).
- **חתימה:** `int socket(int sockfd, const struct sockaddr* addr, socklen_t addrlen)`
- **פרמטרים**
  - `sockfd` – ה-fd שהריצים להתחבר אליו.
  - struct – `addr` שיכיל את ה-IP וה-port שהריצים להתחבר אליהם.
  - `addrlen` – האורך בbytes של `addr`.
- **ערך החזרה:**
  - הצלחה -> 0.
  - כישלון -> -1 ו-errno יוכל את הטעות.

## **server side – bind()**

- מחברת את ה-port ל-socket ו-IP שאנו רוצים ליצור איתם חיבור (בד"כ מדובר ב-IP של host אחר).
- **חתימה:** `int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen)`
- **פרמטרים**
  - `sockfd` – ה-fd שהריצים להתחבר אליו.
  - struct – `addr` שיכיל את ה-IP וה-port שהריצים להתחבר אליהם.
  - `addrlen` – האורך בbytes של `addr`.
- **ערך החזרה:**
  - הצלחה -> 0.
  - כישלון -> -1 ו-errno יוכל את הטעות.

## server side – listen()

- מסמנת את ה-socket שהתקבל בפרטמר בתור .passive socket
- ניתן להפעיל רק על -socketים מסווג: SOCK\_SEQPACKET או SOCK\_STREAM
- **חתימה:** int listen(int sockfd, int backlog)
- **פרמטרים**
  - sockfd – ה-fd של ה-socket
  - backlog – הגודל המקסימלי של התור המוביל את החיבורים (connect) שבוצעו עבור ה-socket זהה (ambilי שנעשה accept).
- **ערך החזרה:**
  - הצלחה -> 0.
  - כישלון -< 1 - ו-errno יכול את הטעות.

## server side – accept()

- מוציאה את החיבור הראשון שממתין בתור של ה-socket שהתקבל בפרטמר ומיצרת עבורו חדש.
- ניתן להפעיל רק על -socketים מסווג: SOCK\_SEQPACKET או SOCK\_STREAM שבוצע עבורה bind() ו-listen().
- **חתימה:** int accept(int sockfd, const struct sockaddr\* addr, socklen\_t addrlen)
- **פרמטרים**
  - sockfd – ה-fd של ה-socket
  - struct שיביל את ה-IP וה-port שורצים להתחבר אליו. addr
  - addrlen – האורך בbytes של addr
- **ערך החזרה:**
  - הצלחה -< socket descriptor שהוקצתה עבור ה-socket
  - כישלון -< 1 - ו-errno יכול את הטעות.

## setsockopt()

- משמשת לצורך עריכת ה-options של ה-socket שהתקבל בפרטמר.
- **חתימה:** int setsockopt (int sockfd, int level, int option\_name, const void\* option\_value, socklen\_t option\_len)
- **פרמטרים**
  - sockfd – ה-fd של ה-socket
  - level – יש לשים את ה-level protocol number של TCP (אפשר להשיג עם getprotoent())
  - option\_name
  - option\_value
  - option\_len
- **ערך החזרה:**
  - הצלחה -> 0.
  - כישלון -< 1 - ו-errno יכול את הטעות.

**ptrace**

- מספקת כלים בהם תהילך אחד יכול (ה-"/tracer" / "המנפה") יכול להתבונן ושלוט ביצוע של תהליך אחר (ה-"/tracee" / "מנפה").
- חתימה: long **ptrace** (enum **\_\_ptrace\_request**, pid\_t **pid**, void\* **addr**, void\* **data**)
- פרמטרים
  - **\_\_ptrace\_request** ○
  - PTRACE\_TRACEME ▪
  - ▪
  - **pid** ○
  - **addr** ○
  - **data** ○ – באפשרות כתיבה/קריאה של נתונים.

## strace

- כל דיבוג בלינוקס.
- מקליט את כל ה-`system calls` וה-`signals` שהתקבלו ע"י התהילין.
- התקנה ב-Ubuntu:  
`sudo apt install strace`
- `ls -strace` – תראה את כל הקריאות מערכת וסיגנלים שבוצעו ע"י `ls`
- `-strace -e open ls` – תראה את כל הקריאות מערכת וסיגנלים שבוצעו ע"י `ls`
- `ls -strace -o out.txt ls` – תשמור את כל הקריאות מערכת וסיגנלים שבוצעו ע"י `ls` לתוכן `out.txt`
- `sudo strace -p 1725 -o out.txt` – תשמור את כל הקריאות מערכת וסיגנלים שבוצעו ע"י תהליך 1725 לתוכן `out.txt`
- `-strace -t ls` – תראה את כל הקריאות מערכת וסיגנלים שבוצעו ע"י `ls` עם חותמות זמן

## errno

- מבטאים: אරור ננו
- נמצא ב-<errno.h>
- משתנה גלובלי מסוג `int` שמקבל ערך שגיאה כלשהו ע"י `system calls` ובמה פונקציות ספריה.
- הערך שלו רלוונטי (מצביע על שגיאה) רק באשר הערך החזרה מראתה על שגיאה (1- ברוב ה-`syscalls` ו-1 או NULL ברוב הפונקציות ספריה).
- ערכו לעולם לא יהיה 0.
- `strerror` ו-`perror` יכולות לקבל אותו כפרמטר ולהדפיס את השגיאה בצורה טקסטואלית.
- חשוב: אם רציתם להשתמש ב-`errno` יותר מאוחר מאשר החזרה מהפונקציה הביעית, יש לשומר אותו בצד, לאחר ופונקציות אחרות עלולות לבצע אליו השמה.
- כמה קוד שגיאה נפוצים:
  - **EEXIST** – קובץ קיים.
  - **ECHILD** – מתקבלת באשר עושים `wait` ולתהליך אין ילדים שאפשר לחכות להם.
  - **EINTR** – call interrupted function call – הטעינה שתתקבל במקורה הבא:
- במקרה בו קוראים ל-`signal handler` signal handler כלשהו בזמן שמבצעת קריית מערכת (system call) יקרה אחד מביניהם:
  1. הקרייה מבוצעת מחדש מיד לאחר שה-`signal handler` מסיים את עבודתו וחוזר.
  2. מתקבלת השגיאה `EINTR` (וקריית המערכת נכשלה).  
זה יקרה ורק אם בעת יצירת ה-`signal handler` הודלק עבורי הדגל `.SA_RESTART`.
- כאשר מחזירים ערך שלילי מפונקציה שנקרה ע"י קריית מערכת:
  1. `errno` מקבל ערך בהתאם לערך המספר.
  2. מוחזר 1- לפונקציה הקוראת.

## strerror

- ממחזירה את השגיאה הטקסטואלית ש-`errno` מכיל.
- חתימה: **char \*strerror(int errnum)**
- `errnum` יהיה פשוט `errno`
- ערך החזרה: מחזורת המתארת את השגיאה בהתאם לקוד השגיאה שהתקבל כפרמטר.
- כדי להדפיס את הערך של `strerror` נשתמש ב-`fprintf(stderr,"%s\n", strerror(errno));`:  
`fprintf(stderr,"%s\n", strerror(errno));`

## perror

- מדפסה את השגיאה מ-`errno` ל-`stderr`.
- מעין פונקציה עוטפת עבור `strerror` שהופכת הדפסת השגיאה להיות נקייה יותר.
- חתימה: **void perror(const char \*str)**
- פרמטר: `str` – הטקסט שיודפס לפני השגיאה.
- דוג: `(perror("Error occurred"))`

- בילינוקס כל מה שהוא לא 0 או NULL נחשב שגיאה (בקשר של ערכיו החזורה מתħali'r/פונקציה)
- דגלים שווי ערך ל-fd:
  - STDIN\_FILENO = 0
  - STDOUT\_FILENO = 1
  - STDERR\_FILENO = 2

## סיסמאות

- **\etc\shadow**
  - מאחסן את כל הסיסמאות של המשתמשים במחשב בצורה מוצפנת
  - רק ה-root יכול לקרוא אותו
  - شامل ל-\tmp\shadow
  - שינוי הסיסמה של המשתמש הנוכחי: passwd
  - שינוי הסיסמה של משתמש אחר: sudo \$passwd <username>
- **\etc\passwd**
  - מכיל מידע על כל המשתמשים במערכת (בשורות):
    - username – name
    - password – בפועל לא יכול הסיסמאות
    - user ID
    - group ID
    - gecos – מידע על המשתמש, כמו השם האמתי שלו. בד"כ יהיה ריק ולא נראה את העמודה הזאת.
    - home directory
    - shell
  - בשורה הראשונה נראה מידע על ה-user root
  - ככל יכולם לקרוא אותו ורק ה-root יוכל לכתבו אליו.

## Nice To have

- malloc ו-free לא בהכרח מעורבים את מערכת הפעלה. ובפרט, לא בהכרח גורמים ל-all syscall mmap. הפונקציות האלה הן פונקציות ספריה (stdlib), שמנהלות את סגמנט ההיפ של התהילך.
- אם נגמר המקום בסגמנט ההיפ אז הספרייה מבקשת mmap (תבקש הרבה זיכרון כדי שתוכל לנוהל אותו בהמשך מבלי לבצע syscall).
- תחום המשחק של הספריה לניהול הזיכרון הוא גדול דף. בתוך הדף הזה הספריה יכולה להקצת זיכרון לפי שיקול דעתה.
- בר נוצר הבדל בין שפות תכנות: java ו-C מנהלות את הזיכרון שבתחום הדף שהוקצה ע"י mmap בצורה שונה.
- אין דרך ישרה לניהול הזיכרון של תהיליך שתתאפשר עבור כל התהילכים, שכן לא מקבעים את השיטה לניהול הזיכרון של דף בתחום מערכת הפעלה.
- עד סיבתה להפרדה הזאת היא ש-exception הוא יקר, ולכן נשתדל לבצע אותו כמה שפחות (רק אחרי שאזל גודל הזיכרון הפיזי שהוקצה לתהיליך והוא צריך עוד).
- למציאות ניתן לעבוד עם כמה התקני אחסון ואפשר לעבוד כמה FSs דרך VFS (virtual FS).
- שימוש בחוטים יהיה יעיל יותר רק כאשר יש לנו יותר מליבה אחת. אחרת, מאבדים הרבה הרבה זמן על overhead ("לחinem" כי הרי מחליפים בין כוחות חישוב של אותו מעבד (=ליבת בקשר שלם)).
- **ה-kernel הוא לא תהיליך.** הוא יוצר kernel threads שליהם יש סוג של PCB (מבנה נתונים כלשהו שמייצג חוט).